NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF NAVAL ARCHITECTURE & MARINE ENGINEERING
DIVISION OF MARINE ENGINEERING

Diploma Thesis

Development of CAD Algorithms for Optimized Design of Ship Piping Systems

Markos Papatheodorou

Thesis Examination Committee

Supervisor:	C.I. Papadopoulos,	Associate Professor NTUA
Members:	A.A. Ginnis,	Associate Professor NTUA
	L. Kaiktsis,	Professor NTUA

Athens, July 2019

....
Mark Papatheodorou
College Graduate Naval Architect & Marine Engineer NTUA

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΝΑΥΠΗΓΩΝ ΜΗΧΑΝΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΟΜΕΑΣ ΝΑΥΤΙΚΗΣ ΜΗΧΑΝΟΛΟΓΙΑΣ

Διπλωματική Εργασία

Ανάπτυξη Αλγορίθμων CAD για Βέλτιστη Σχεδίαση Δικτύων Πλοίου

Μάρκος Παπαθεοδώρου

Athens, July 2019

....
Μάρκος Παπαθεοδώρου
Διπλωματούχος Ναυπηγός Μηχανολόγος Μηχανικός Ε.Μ.Π

Acknowledgments

Looking back to the day when I first walked into the office of Professor C.I. Papadopoulos in order to accept the proposition of doing my Diploma Thesis under his guidance, mixed emotions come to my mind. For me, it has been a period of intense learning. Through the work that has been done, I have come to realize that my horizons have been significantly widened, both on a scientific and a personal level. As far as scientific knowledge is concerned, I had the chance to become familiar with a field that was completely unknown to me, and would have remained so under other circumstances. In addition, the multiple challenges with which I was presented during this journey of mine, in combination with a multitude of practical issues that had to be overcome during the development phase of my Thesis, made me rethink my attitude towards dealing with complex problem-solving tasks, and for this I am very grateful. When all is said and done, it is safe to say that this has been one of the most productive periods of my life so far. At this point, I would like to reflect on the people who have assisted and supported me throughout this period.

First, I would like to thank my supervisor, Assoc. Prof. Christos Papadopoulos, for his support, patience and his constant availability throughout the span of my Thesis. His steadfast refusal to be satisfied with mediocrity, served as a motivating force for me to always aim for more than just the obvious.

Also, I need to thank Assoc. Prof. Alexandros A. Ginnis, for his unconventional thinking and problem-solving prowess, which served as an inspiration to me. The door to his office was always open whenever I ran into a dead-end in the development phase of my Thesis. No question could close him in, and this taught me to always seek to think out of the box.

At this point, I strongly feel that special reference should be made to PhD Candidate Anastassios Charitopoulos, with whom I closely cooperated throughout the various stages of my Thesis. If it had not been for his constant support, encouragement and unwavering assistance, this Thesis would not have advanced at the present level.

Additionally, I would like to thank my friends and classmates with whom I have spent hours upon hours throughout our studies in NTUA. Their encouragement and their attitude towards me went along way towards helping me keep up with the Thesis demands. Iasonas-Stefanos Kyknas, Giannis Mprilis, Stefanos Mokas, Maria Iosifidi and Konstantina Fountouli, thank you for being by my side all this time.

I would also like to express my heartfelt gratitude to my parents, Athanasios Papatheodorou and Dimitra Geraki, whose raising efforts and sacrifices made me into the person that I am today. I am also thankful to my brother, Ianos Papatheodorou, for his feedback regarding programming facets of my Thesis. Special mention should be made to my grandparents, Christos Papatheodorou (†), Euniki Toutountzaki-Papatheodorou, Georgios Gerakis, Theano Mimilidou-Geraki for their admiration and encouragement.

Last but not least I would like to thank my partner, Rahel Jasmine Wiegand for her love and patience which saw me all the way through my Thesis.

Dedication

This study is wholeheartedly dedicated to my beloved grandfather Christos Papatheodorou (†), who did not get the chance to see me going to study at the NTUA. I know that he would have been really proud...

Abstract

Pipe-routing is ranked among the most important, and at the same time the most time-consuming activities during the detail-design phase of a ship. Furthermore, it is closely related to a multitude of other concurrent tasks within the ship, meaning that a less than optimal solution can cause serious issues during later phases of the design process. The complexity is very high, given the fact that the configuration of the layout space is highly elaborate, with numerous pipelines, obstacles and spaces that should remain free for various operational and safety reasons. In the present Diploma Thesis, automated pipe-routing algorithms are proposed, that go some way towards providing swift and optimal solution suggestions to the designer, thus effectively cutting down on precious manhours and improving the efficiency of the whole design procedure. A cell-degeneration method, based on a combination of surface and solid voxelization techniques, is developed, in order to make the transition from the continuous space to a three-dimensional cubic grid representation of the target layout space. Having established the mathematical model of our layout space, two graph constructs are considered, the standard and the diagonal one, so that the discrete mathematical model can meet various geometric connectivity constraints. Then a path-finding procedure is initiated based on the Dijkstra and the A* algorithms. Furthermore, in order to be able to facilitate the incorporation of directional specifications during the path-finding process, an augmented vertex-split strategy is devised and implemented. In addition, this extension actively solves the problem of bend in a resulting path. Last but not least, the position-level and plane-distance weight allocation methods are introduced, in order to enable the designer to actively manipulate the whereabouts of the resulting optimal path. A series of test runs on various models of increased layout complexity were then carried out, in order to demonstrate the efficiency of the proposed methodology. All in all, the automatic routing result can serve as an optimization guide, that will enable the designer to achieve better results.

Σύνοψη

Η διαδικασία σχεδίασης σωληνουργικών δικτύων (Pipe-routing) κατατάσσεται αναμεσα στις πιο σημαντικές, και συνάμα πιο χρονοβόρες διεργασίες που λαμβάνουν χώρα κατά τη φάση του αναλυτικού σχεδιασμού ενός πλοίου. Επιπροσθέτως, η διαδικασία αυτή είναι άμεσα συνδεδεμένη με μια πληθώρα άλλων διεργασιών εντός του πλοίου, γεγονός που σημαίνει ότι η παραγωγή μή βέλτιστων λύσεων ενδέχεται να οδηγήσει σε σοβαρά προβλήματα σε επόμενες φάσεις σχεδιασμού. Η πολυπλοκότητα που χαρακτηρίζει τη διαδικασία σχεδίασης των δικτύων αυτών είναι μεγάλη, δεδομένου του ότι η διάταξη των διαθέσιμων χώρων εντός του πλοίου είναί εξαιρετικά περίπλοκη, με πληθώρα σωληνώσεων , εμποδίων και μη προσβάσιμων χώρων, τόσο για λόγους συντήρισης όσο και ασφαλείας. Στην παρούσα Διπλωματική εργασία, προτείνεται η ανάπτυξη αλγορίθμων βέλτιστης σχεδίασης δικτύων πλοίου, προκειμένου να δωθεί μια βέλτιστη προτεινόμενη λύση σε σύντομο χρονικό διάστημα, μειώνοντας έτσι τις απαιτούμενες ανθρωποώρες εργασίας και αυξάνοντας την αποτελεσματικότητα της διαδικασίας σχεδίασης. Αρχικά, αναπτύσσεται μια μέθοδος διακριτοποίησης χώρου (Cell-decomposition), που βασίζεται σε τεχνικές επιφανειακής και στερεάς χωρικής αποδόμησης (Voxelization). Η μέθοδος αυτή, καθιστά δυνατή τη μετάβαση από τον συνεχή στο διακριτό χώρο, με τη μορφή ενός τρισδιάστατου κυβικού πλέγματος το οποίο χτίζεται επί του χώρου ενδιαφέροντός μας. Έχοντας πλέον τη μαθηματική μοντελοποίηση του χώρου μας, προχωρούμε στην κατασκευή δύο γράφων (Graph), του απλού και του διαγωνίου, οι οποίοι προσδίδουν την έννοια της γεωμετρικής δομής στο μοντέλο μας. Στη συνέχεια, εκκινείται μια διαδικασία εύρεσης βέλτιστης διαδρομής, η αρχή λειτουργία της οποίας στηρίζεται στους μηχανισμούς των αλγοριθμων Dijkstra και A*. Επιπλέον, προκείμενου να καταστεί δυνατή η λήψη προδιαγραφών κατεύθυνσης στην διαδικασία επίλυσης του προβλήματος, υλοποιείται μια σταρτηγική κομβο-διχοτόμησης (Vertex-split). Πέρα από τις προδιαγραφές κατεύθυνσης, η προτεινόμενη αυτή στρατηγική, αντιμετωπίζει αποτελεσματικά το πρόβλημα των γωνιών (The problem of bend) στα αποτελέσματα. Τέλος, παρουσιάζονται δύο μέθοδοι απόδοσης βάρους στις ακμές (edges) των γράφων, η μέθοδος επιπέδου-θέσης (Position-level) και η μέθοδος απόστασης -επιπέδου (Plane-distance), η εφαρμογή των οποίων καθιστά εφικτή την δυνατότητα ενεργούς επιρροής των περιοχών προτίμησης διέλευσης των παραγόμενων αποτελεσμάτων. Στη συνέχεια, υλοποιούνται μια σειρά από προσομοιώσεις (Test-runs) σε μοντέλα κλιμακούμενης πολυπλοκότητας, προκειμένου να καταδειχθεί η αποτελεσματικότητα της προτεινόμενης μεθοδολογίας σχεδίασης δικτύων. Συμπερασματικά, οι λύσεις, όπως αυτές προκύπτουν απο την αυτοματομποιημένη μέθοδο σχεδίασης που προτείνεται, μπορούν να λειτουργήσουν ως γνώμονες βελτιστοποίησης, οι οποίοι θα επιτρέψουν στον εκάστοτε σχεδιαστή να επιτύχει καλύτερα αποτελέσματα σχεδίασης.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Pipe Routing and applications

A ship could be likened to a living organism, when one takes into consideration the amount of systems that comprise it, such as Auxiliary systems, Sea water systems, Fresh water systems, Fuel and Lubricant handling and storage systems etc. The whereabouts, the connectivity and the arrangement of all components of these systems, and their respective piping apparatus, are decided and designed during the detail-design phase of the ship design process. The ship-wide routing of piping in a ship environment is a very delicate, complicated and strenuous work that approximately accounts for more than half of the total detail-design man-hours[1]. To this very day, most of the pipe-routing work that needs to be done is executed by individuals, know as Piping Engineers, and thus the results of pipe design are very highly dependent on the experience and knowledge of these individuals. As a result, design and time efficiency are seriously undermined, or at best, constrained. On top of that, many of the other activities of detail-design depend on the resulting pipe routing.

Based on the aforementioned facts, it becomes obvious that a different approach needs to be adopted when faced with the complex problem of pipe-routing. A switch has to happen, from the traditional methods, to the development of automatic pipe-routing methods. The widespread use of 3D-CAD systems in the last couple of years, in conjunction with a high demand for extensive piping retrofits, as a result of a potential Water Ballast Treatment System (WBTS) or Scrubber System installation, has made it clear that the future of pipe-routing lies with automation. However, an automatic approach to such a complicated and multifaceted problem is by no means simple, nor straightforward, as there are lots of restrictions and requirements that have to be met in every pipe-routing scenario.

1.2. Literature Review

Extensive research has been carried out in the last couple of decades, with many promising results, which are usually accompanied by new challenges and limitations. As a result, innovative ways to route pipes automatically have been created. By definition, pipe-routing belongs to the class of optimization problems. All the algorithms that have been developed over the years, in order to deal with such problems, utilize techniques that can be categorized in the following three directions:

- Deterministic.
- Non-Deterministic (Meta-Heuristic).
- Combination of both.

As far as the first direction is concerned, it includes several methods, the most prominent of which being Lee's algorithm (or Maze algorithm, MA)[2], Dijkstra's algorithm[3], and the A* (A Star) algorithm[4]. The main advantage of this approach is that it always produces the best solution, in case of course one exists. However, there lies a calculation time and extensive RAM memory occupation trade off. On top of that, applications which include a search space heavily ridden with obstacles, only make the aforementioned disadvantages all the more obvious. Last but not least, not many ready made implementations of these algorithms exist in supported software solutions[5].

In recent years, Non-Deterministic methods slowly, but steadily, have made their way towards the field of pipe-routing. Genetic algorithms (GA)[6],[7],[8],[9], Ant colony optimization algorithms (ACO) [10],[11] and Particle swarm optimization algorithms (PSO)[12],[13],[14],[15] are but a few of the methods that fall into this category. All of them use stochastic procedures in order to tackle the complex problem of pipe-routing, thus resulting into lower memory RAM occupation and consequently faster result production. While this may be true, these methods do not always produce the best solution, even if one exists. Furthermore, use of stochastic methods in complex environments makes it even more difficult for them to achieve high standard results.

In recent years, a combination of both methods is being favored by many, as it incorporates the optimal solution finding capability of Deterministic methods, and the faster computational times and lower RAM occupation combination of Meta-Heuristic methods into one package, thus amplifying the advantages of both methods while trying to eradicate the disadvantages.

H Kimura[16] , proposed a new method based on Dijkstra's algorithm used in conjunction with a GA optimizer. The pipe branches are considered as various equipment, meaning that a pipeline simply connects two pieces of equipment. In order to achieve a fairly good distribution of all the equipment in the available layout space, a Random Equipment Arrangement and a Self-Organization Equipment Arrangement technique is introduced. The suggested connecting paths between each equipment is produced by the Dijkstra algorithm.

H Sui and W Niu[1], developed an improved genetic algorithm for tackling with the problem of branch-pipe-routing. They regarded the branch pipe as a collection of several two point pipes, thus breaking down a complex branch-pipe-routing methodology into a series of simple point to point line connection problems. The initial population, on which the GA's unique crossover and mutation operations would be exercised, was produced by implementing an improved version of Lee's MA, which aimed to expand the search space constructed by two connection points by introducing the concept of the auxiliary point. The auxiliary point serves as a layout space expansion agent, meaning that the path which will be connecting the specific starting and end point of each algorithmic iteration will have to pass through this randomly chosen point, adding a much needed diversity to the individual candidate paths that will afterwards be fed to the GA mechanism for optimization. A similar approach was adopted by W Niu et al.[17], who introduced the pipe grading method, and then presented an optimization module by combining MA, non dominated sorting genetic algorithm II (NSGA-II) and cooperative co-evolution non dominated genetic algorithm II (CCNSGA-II).

Z Dong and Y Lin(†)[18], proposed a path formulation method which utilizes the fixed-length encoding GA by connecting adjacent intermediate points using Dijkstra's algorithm. Whats more, in the case of multi-pipe-routing, they introduce the use of cooperative co-evolution GA (CCGA), in which the optimality of each suggested solution is closely correlated with its ability to collaborate with individuals from other concurrent solutions which refer to different pipe lines being routed. Y Ando and H Kimura[19], proposed a new method based on an improved version of Dijkstra's algorithm, thus managing to include elbows and bends to the pipe-routing process, and at the same time disconnecting the pipe size from the size of the cell grid that decomposes the layout search space. A similar approach was followed also by H Nguyen et al.[20], who in addition also incorporated the solution of the branch-pipe-routing problem in their method implementation. S-H Kim et al.[21], proposed a graph based vertex-split technique in order to regulate the number of bends in the resulting path. Dijkstra's algorithm was utilized in the pipe-routing process, while the search space was divided into non-uniform cells in order to reduce the RAM requirements.

All of the aforementioned endeavors, as far as the processing of the layout workspace is concerned, adopted the cell decomposition approach, in which the space is decomposed into a grid of cells, otherwise known as voxels. The connecting points of each individual machinery piece, or other component, that define each individual pipe-routing problem, can this way be represented by these connected voxels. This approach, being the most common, has both advantages and disadvantages. On the plus side, decomposing the workspace into cells allows for the use of graph-search based algorithms such as Dijkstra, A* etc, who can guarantee the finding of the optimal solution, if such exist. Furthermore, it enables the user to perform sophisticated weight allocation on the different edges of the resulting graph, thus enabling the active manipulation of the resulting paths. The big drawback of this approach however is the fact that good solutions require large number of voxels, leading to longer computational time and higher RAM usage. In order to tackle this issue, some other approaches have been developed.

In 1968, K Mikami and K Tabuchi[22], introduced the first line-search algorithm, and later on D W Hightower[23], in 1969, proposed the so called escape algorithm which is based on the work done by the previous two. This method starts with two perpendicular lines through the starting point S. It tries to find a point such that an escape line will extend beyond one of the previous boundaries of point S. If such an escape point is found, it becomes the new point S. This method repeats the process until the line segment crosses the target point G. The escape algorithm is both fast and uses less memory, but, the same way as Non-Deterministic methods, cannot guarantee a solution. Furthermore, this method cannot be implemented in a weighted graph environment. A Asmara[24], favored the use of the Mikamu-Tabuchi algorithm for conducting the so called blockage checker, whose main purpose is to test whether a pipe can be routed or not. For the actual pipe-routing process he utilized the A* algorithm.

J-H Park and R L Storch[25], having identified the weakness of cell decomposition, proposed a cell generation method. This method introduces the use of a number of predetermined basic and modified pipe patterns, which along with a bridge cell generation process between terminal cells, can achieve acceptable pipe-routing solutions, that also take directional specifications into consideration. Another work was done by L Huibiao et al.[26], who proposed a hanging bridge algorithmic solution for pipe-routing arrangements that make use of free space modeling, thus resulting into more simplified arrangement constraints and easier incorporation of traditional path-finding algorithms into to the problem-solving process.

When all is said and done, comparison between these three directions (Deterministic, Non-Deterministic and Combination of both) is not an easy task, as all of them approach the problem of pipe-routing from quite different angles. As a result, it is all but impossible to know which is better. However, all three of these approaches share some disadvantages:

(1) Traditional optimum methods do not consider the problem of elbows and bends, emphasizing more on the problem of shortest path.

(2) The ability to actively manipulate the resulting path whereabouts is not supported.

(3) Diagonal movement freedom is not included in most of the developed methods.

(4) Directional specifications are not taken into consideration when performing the pipe-routing process.

(5) The decomposition of the search space is closely correlated with the size of the smallest pipe to be routed .

(6) Highly complex search spaces, can result in low quality paths being produced.

1.3. Goals – Outline of present study

1.3.1. Pipe - Routing method

As discussed in the introduction and literature review sections, automatic pipe-routing solutions need to be developed, solutions that will actively improve both the design and time efficiency of the ship-wide routing of piping in ships. Solutions do exist, but most of them are striving to incorporate deterministic approach algorithmic solutions into ready-made, user-friendly software package solutions.

Following previous literature studies, the present work attempts to expand the potential of two traditional Deterministic algorithms, which have seen extensive use in the field of the path-finding and consequently in the pipe-routing industry: Dijkstra and the A*. Both Dijkstra and A* algorithms are guaranteed to find the optimal solution in any pipe-routing problem, regardless of complexity, if of course such a solution exists. However, careful consideration should be given when choosing the heuristic function for the A*, because it can greatly impact the algorithm's optimal path-finding capability. The cell decomposition method is used in order to transform the layout search space into a discrete cubic grid. For this reason, a solid voxelization engine is developed. Afterwards, the centroids of all cubic cells in the target space are used as the vertices of the graph, which will serve as an input to our path-finding algorithms. At this point, one should keep in mind that the vertex and edge connections (line segments connecting every vertex to its neighboring vertices) of the created graph needs to have a topological meaning, as the graph should represent the physical pipe routing. For this reason, a vertex-split strategy is introduced, based on the work of S-H Kim et al.[21], which enables us to differentiate edges that form a straight or a bend pipeline segment. Furthermore, the proposed vertex-split strategy is expanded, for use in graphs that enable diagonal movement as well, thus offering a richer pipe-routing solution pool, from which our method can draw potential optimal results from. Having chosen to delve into the possibilities of graph-search based algorithms, proper edge weight allocation should be conducted in order for the algorithms to function correctly. In this work, we introduce the position-level and plane-distance based weight allocation methods. These methods allow for a complicated edge weighting of the graph structure that can result into realistic pipe-routing solutions being produced, without the use of a Non-Deterministic approach. Last but not least, a set evaluation criteria of the resulting paths is formed, in order to access the optimality of the solutions that are produced by our method.

All in all, the proposed pipe-routing method in question can be broken down to its primary components which comprise:

- Layout workspace CAD model creation.
- Decomposition of the 3D space model into cubic cells (Voxelization).
- Graph structure creation.
- Vertex-split strategy implementation.
- Edge weight allocation.
- Resulting path metrics evaluation.

Given the fact that the previously mentioned steps, that describe the pipe-routing solution, are anything but straightforward, and bearing in mind the inter-connectivity of them all, a standalone pipe-routing application is developed and presented in this study.

1.3.2. Software development

As we have already mentioned in the previous section, the proposed pipe-routing method of this study is presented within the frameworks of a dedicated piece of software that was drawn up exclusively for the needs of the current Thesis. The program was developed using C++ language.

An in-depth overview of the new program is given in chapter 3 Pipe-routing software. However, for the shake of completeness, a summary of the main functionalities of the program is being presented below. The user needs to input a file containing the information needed to recreate a 3D CAD model of a layout workspace as discrete grid of cubic cells, otherwise known as voxels. Such information refer to the size of the cubic cells, dx, dy and dz, the number of cells along each of the three x, y and z axis, the voxel tag that denotes whether a voxel belongs to the free space available for pipe-routing or not etc. The program then offers the user the opportunity to run pipe-routing simulations based on a series of parameters, such as start and end point coordinates and direction, graph type, weight allocation method and path-finding algorithm, which are presented in the path options window through a graphical user interface (GUI). After conducting various test runs, the user can see the evaluation characteristics, or path metrics, of each suggested path and thus draw conclusions accordingly. Last but not least, the program gives the user the ability to save his or her test runs in an output file, which can be loaded in the program at a later date for further use.

1.3.3. Case studies

In order to investigate the potential of the method proposed in this section, as well as the limitations that are imposed to it by several factors, most important of all being the RAM consumption, a series of case studies were carried out.

In the present work, a series of CAD test models are considered. All of these models were created on the grounds of bringing out the method's contribution towards solving particular pipe-routing issues, such as:

- Manage navigation through highly complex and heavily obstacle ridden layout spaces.
- Penalizing bends during the pipe-routing procedure, in order to eradicate zig-zag effects in the resulting paths.
- Adding directional specifications as input parameters for the test runs.
- Force the path-finding algorithms to follow paths that are deemed more preferable according to various edge weighing criteria.
- Avoid obstacle collision issues.

Finally, a 3D CAD engine room model of a typical cargo ship was created based on various General Arrangement and Capacity Plan designs that were provided by Christos I. Papadopoulos, Associate Professor, School of Naval Architecture & Marine Engineering, National Technical University of Athens. The final test runs were performed in this model, as to test the functionalities of our method in almost real-life conditions.

2. Methodology

2.1. Introduction

In this chapter, we will delve into the two main aspects which comprise the methodology that was developed in the current study: Modeling and Algorithmic. At first, the modeling approach to the problem of pipe-routing will be presented and analyzed. This includes information about the layout space modeling and the creation of the graph structure. Secondly, the details of the algorithmic approach to the problem will be discussed. At this point, the functionality of Dijkstra and A* will be explained, alongside with the weight allocation method and the path metrics evaluation process that was followed throughout this work.

2.2. Problem formulation and constraints

2.2.1. Understanding the problem

Pipe-routing in an obstacle populated environment is by no means an easy task. On the contrary, it is a complicated, multi-objective optimization problem which at most cases require a fare amount of computational resources in order to be solvable, alongside with some serious thinking, regarding the issues that have to be dealt with. The most important of these problems are as follow. First and foremost, the space that is occupied by any kind of obstacles, be it walls, machinery, stairs, piping apparatus etc, has to be redacted from the available free space.

Secondly, the pipes that are about to be routed from various start and end points within the feasible search space, should not interfere with any obstacle occupied cell. Last but not least, a graph structure has to be constructed using the centroids of every obstacle-free cell as a vertex, and the edges connecting these points as the graph edges. As we have previously discussed, weights have to be allocated to the various graph edges. These weights represent the penalty which the path-finding algorithm will be called to pay, as it navigates through the various edges in order to reach the destination point that has been inputted by the user. The higher the weight an edge has, the less the probability that the path-finding algorithm will choose to use it. The weight values, that every edge will have , are related to the weight allocation methods that the developer has chosen to include in the path-finding process. For example, its common practice for the pipes to be routed close to the walls or the ceiling. This means that the edges which are located closer to obstacles occupied cells will have a smaller weight value than other edges which will be located in the middle of a room, resulting in interference of movement. The same thing applies with floors and ceilings, with proximity to the later being more favorable than the first. Consequently, weight allocation methods, are closely related with the satisfaction of various geometrical and topological constraints. When more than one constraint is needed to be satisfied, the penalty weight factors are combined, and later added to the original weight value of every edge in the graph construct, which is equal to the actual distance of the vertices that it connects.

All in all, careful consideration should be taken, followed by meticulous planning, when trying to understand how the pipe-routing problem is formulated and which steps should be made in order to deal with the various constraints that are imposed to this problem.

2.2.2. Understanding the constraints

The pipe-routing optimization problem should go some way towards satisfying various requirements and constraints. In his survey of pipe-routing design, X-L Qian et al.[5], quoting from the work of J-H Park and R-L Storch[25], categorizes these various requirements and constraints into the following types:

(1) ***Physical constraints**:* The pipe routing should avoid physical obstacles and always connect to the desired equipment.

(2) ***Economic constraints**:* Minimize the total length and number of bends of the routed pipes.

(3) ***Operation & Maintenance constraints**:* Pipes that need frequent maintenance, or pipes that have apparatus that should be accessible by man, should be routed within arm's reach and clear from inaccessible areas or tight spaces.

(4) ***Production constraints**:* Maximization of support sharing with other pipes. In other words, pipes of similar class and size should be routed in parallel as much as possible.

(5) ***Flexibility constraints**:* Pipes should be routed along walls as much as possible.

(6) ***Safety constraints**:* Keep minimum clearance off from specific equipment.

These requirement/ constraint types can be divided into two main groups: restrictive ((1) - (3)) and quantifiable ((4) - (6)). Bearing the previously mentioned constraints in mind, helps one perceive the problem of pipe-routing in a more structured and tangible way. In other words, pipe-routing seeks to find the best candidate path which meets the requirements of restrictive constraints, while trying to satisfy, to some extent, the quantifiable constraints. However, it should be noted that some times a trade-off has to be considered when pursuing to satisfy all these constraints, as in more than one occasions the fulfillment requirements of one constraint might be contradictory to another. For example, considering the problem of minimizing the bends of a pipeline, might lead to candidate paths being routed away from obstacles such as walls, floor or ceiling, thus interfering with production and flexibility constraints.

All in all, it becomes apparent that pipe-routing, is a multi-objective optimization problem, meaning that more than one optimal solution can exist, depending on the evaluation parameters and the constraints that they strive to meet. While this fact might raise questions as to whether the resulting solutions are truly optimal, and if so under what conditions, it provides a much needed diversity to the solution pool, giving the user the opportunity to assess the results and finally choose the ones which best meet the criteria of his approach to the problem.

2.3 Modeling approach

2.3.1. Introduction

In order to tackle the problem of pipe-routing, a solid and straightforward modeling approach has to be devised first, before the actual problem-solving procedure is initiated. A path-finding algorithm requires the existence of a geometrical and topological model. In the present work, a solid voxelization engine was developed, followed by the construction of a 3D graph structure using the spatial and geometric data acquired from the voxelizer engine.

2.3.2. Voxelization

2.3.2.1. Definition

In order to generate a volumetric representation from a 3D geometric object, a reformulation process is required[27]. This process is usually called voxelization. Voxelization is a stage of paramount importance in the field of computational science, during which a geometric object is modeled into its equivalent discrete voxel representation[28]. A voxel is a single element in a voxelized model, see Figure 1.

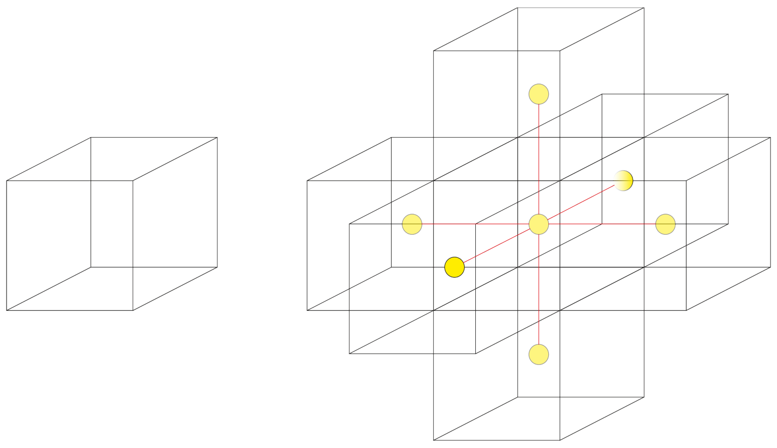


Figure 1: Left: Single voxel unit; Right: Voxel connectivity through their centroids

There are two main approaches regarding voxelization: surface and solid voxelization. When referring to a surface voxelization, all voxels are set that fulfill some overlap or distance criterion with respect to a surface, whereas a solid voxelization sets all voxels which are considered interior to a particular object. Much research has been conducted the last years regarding the topic of voxelization, which still remains a difficult problem because of its computational complexity. However interesting and important these studies are, it goes beyond the scope of this study to present an in depth literature review concerning the matter of voxelization. Nevertheless, in case the reader displays particular interest on this subject, he is advised to look up in the references displayed here[29], [30], [31], [32], [33], [34], [35].

Although the topic of voxelization, and especially that of the solid approach, might be difficult and complex, there is a certain advantage that renders this approach essential to the pipe-routing problem solution. By representing a solid as a set of voxels, it enables us to access each voxel in the grid directly by knowing its position in space or its relative position to another neighboring voxel. This distinct advantage makes the solid voxelization approach the perfect candidate for modeling the environmental layout workspace for a collision detection problem.

2.3.2.2. Implementation

As mentioned above, for the needs of the current work, a solid voxelization engine was developed. At this point, before proceeding with a detailed analysis of the functionalities of our voxelizer engine, it should be underlined that the aim of this whole process, is to formulate a cubic cell decomposition of our layout workspace, which will serve as construction template for the graph creation that will follow.

Basically, the approach to the problem of voxelization, that our engine adopts is based on the following steps.

(1) Create an axes aligned bounding box of the imported CAD geometry.

(2) Set the desired voxel size in mm.

(3) Perform a non-uniform scale of the bounding box, based on the voxel size value, in order to have an integer number cubic cells along the x, y, z axes.

(4) Create a 2D square cell grid on the xy-plane (base plane).

(5) Extrude each cell of the 2D grid, forming a 3D cubic cell grid.

(6) Create a series of 3D cubic cell layers and translate them along the z axis, decomposing the entire space, included in the bounding box, into voxels.

(7) Perform an interference check between the 3D CAD geometry and the cubic grid.

(8) Perform an inclusion test for all the voxels of the grid. In order for such a check to be performed, each voxels is represented by its centroid. In case the point is found to be within a particular object, the voxel is marked as being inside the object.

In Figure 2, the configuration of the voxelizer engine is illustrated.

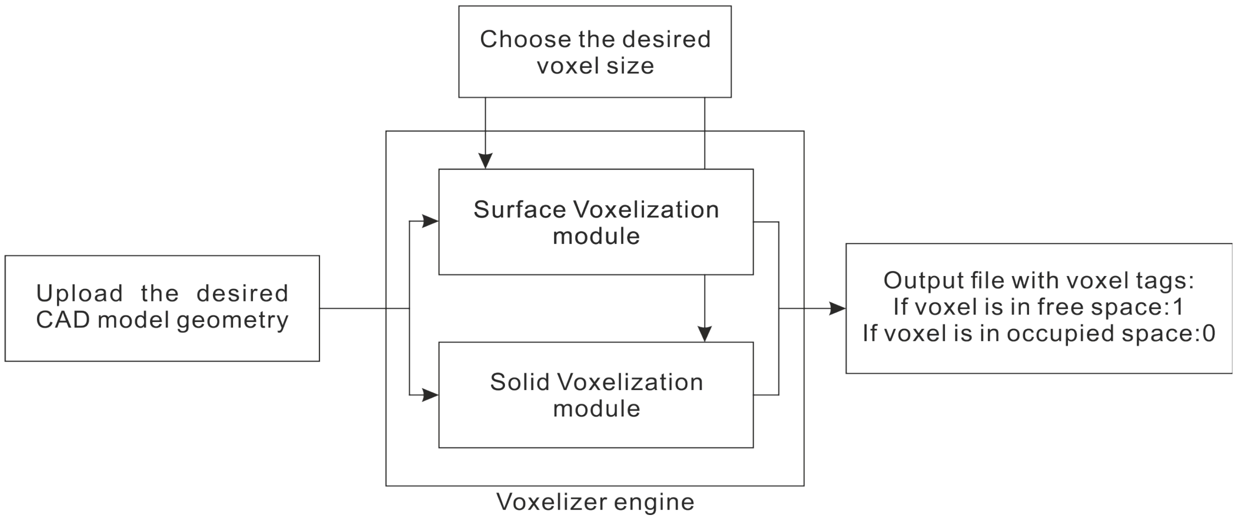

Choose the desired voxel size
Surface Voxelization module
Upload the desired CAD model geometry
Output file with voxel tags: If voxel is in free space:1 If voxel is in occupied space:0
Solid Voxelization module
Voxelizer engine

Figure 2: Voxelizer engine configuration

The developed voxelizer engine, was designed to be as automated as possible, meaning that great effort was made to keep manual intervention in the process to a minimum of inputting the desired geometry parameters, as we all as designating the desired voxel size. Although more efficient voxelization methods exist, the fact that the present study even addresses this particular issue directly, an issue that plays an active role in the pipe-routing problem , without assuming it to be trivial or already solved, renders it quite unique.

Figures 3 – 6, illustrate the basic steps that comprise the proposed voxelization method.

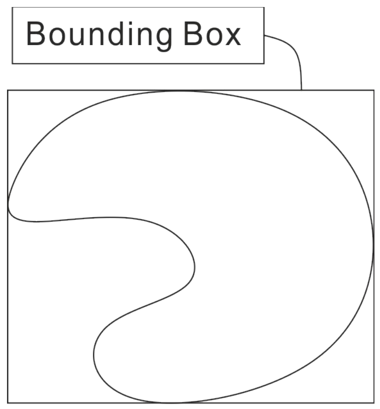


Figure 3: 2D perspective of created bounding box

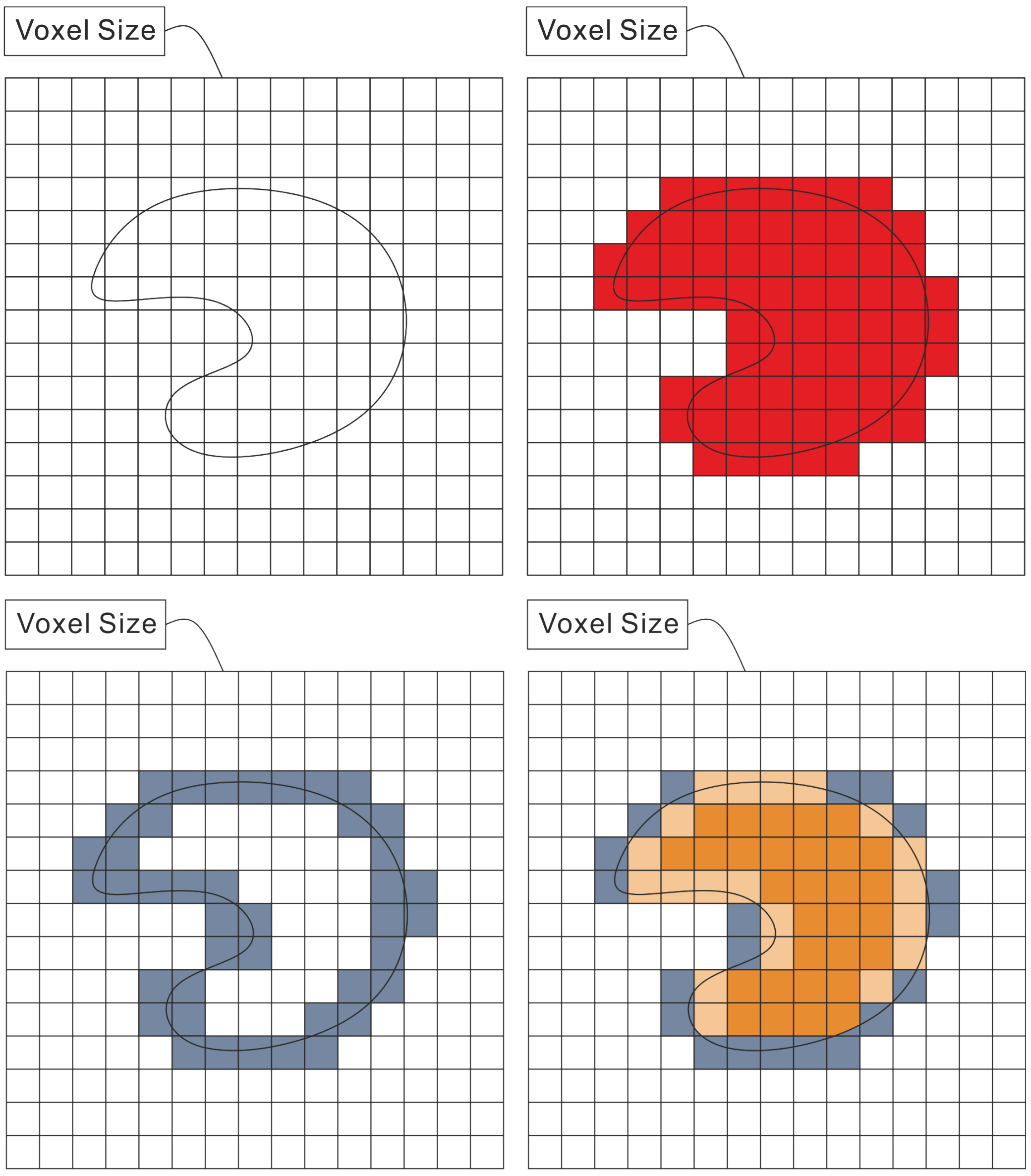


Figure 4: 2D representation of voxelization process. Top Left: Cell Decomposition based on desired voxel size; Down Left: Surface voxelization; Down right: Solid voxelization, with depiction of overlap between the two methods; Top right: Voxelized model

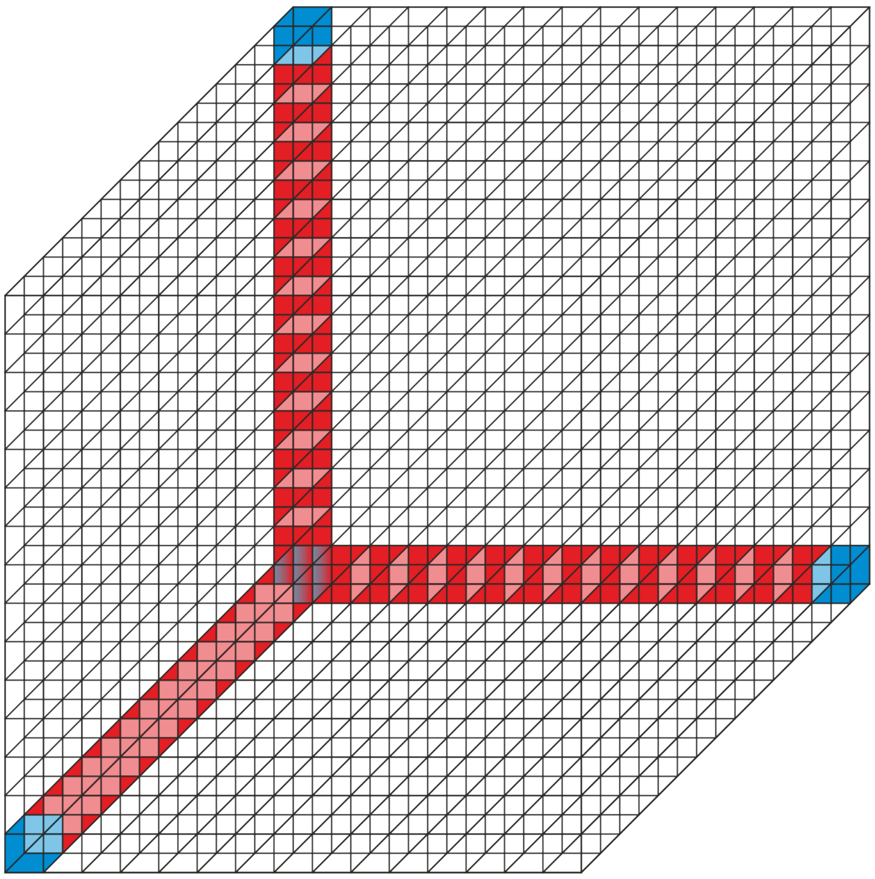


Figure 5: 3D perspective of scaled and decomposed bounding box

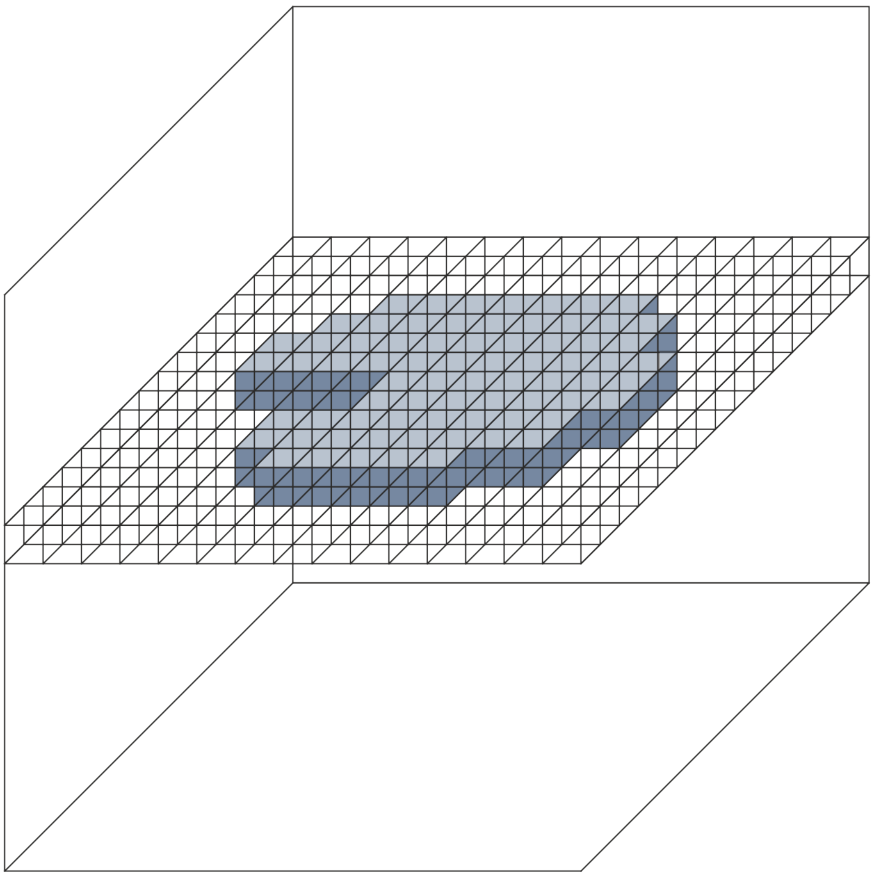


Figure 6: 3D perspective of a slice from the final voxelized model

2.3.3 Graph construction

2.3.3.1. Definition

Graphs are a powerful and versatile data structures that easily allow for representation of real life relationships between different types of data (nodes). There are two main parts of a graph: Nodes, and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a graph can be defined as consisting of a finite set of vertices and set of edges which connect a pair of vertices. In other words, in this context a graph is made up of vertices which are connected by edges. Figure 7, depicts a simple graph.

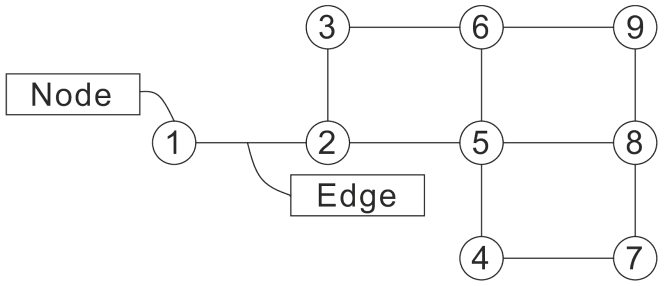


Figure 7: A Simple graph with nine nodes (vertices) and eleven edges

A distinction can be made between undirected graphs and directed graphs. The difference between these two types, lies in the fact that in undirected graphs, edges link two vertices symmetrically, meaning that if vertex 1 is connected with vertex 2, then vertex 2 is also connected with vertex 1, whereas in directed graphs, this link is asymmetrical, and thus connection of vertex 1 with vertex 2 does not imply a two-way connection. To better understand the difference, an illustration is presented in Figure 8:

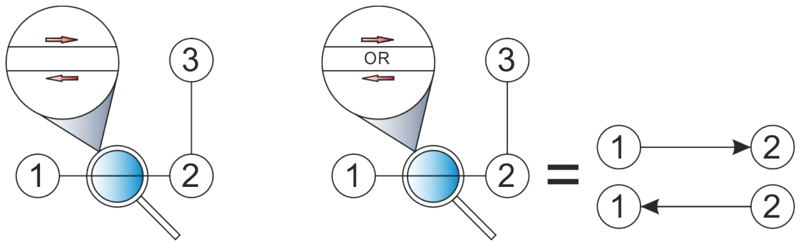


Figure 8: Left: Undirected graph; Right: Directed graph

In addition to this distinction, graphs can also be weighted or unweighted. What this means is that there can be some cost value associated with the edges connecting the several vertices that populate the graph structure, or no cost value at all.

In the current study, an undirected weighted graph is constructed upon the cell decomposed space that was produced by the voxelizer engine, using the centroids of the unoccupied voxels as vertices. The main reasons for choosing to create such a graph are explained. First and foremost, the undirected graph is clearly the best choice, since it allows for a two-way connectivity between any two edge-connected vertices within the graph structure, a feature that serves any pipe-routing problem solution procedure well. Secondly, this goes some way towards reducing the RAM requirements of the developed software since it reduces the number of the edges that would be needed in half, in case a directed graph was opted for. And last but not least, weighted graphs allow the use of traditional path-finding algorithms such as Dijkstra and the A*.

In addition, two types of graphs are introduced, the standard and the diagonal graph. Furthermore, in order to equip the vertex and edge connections of these graphs with topological information, that will enable the pipe-routing to produce realistic results, as far as the issue of bends is concerned, a vertex-split strategy is introduced.

2.3.3.2. Standard graph

Dealing with the task of routing pipes in a ship is by no means easy, as there are multitudes of obstacles in an already free-space deprived environment. However, the use of a cell decomposition method can simplify the problem by providing a free-space frame upon which a graph can be built.

Although a ship's hull is comprised of steel plates that have been given elaborate geometric forms in order to produce a smooth finish, thus resulting into lower frictional resistance, it is regarded as good practice to route pipes along the main axes of ship, avoiding unnecessary diagonal routes. Bearing this in mind, a standard graph is constructed in the decomposed layout space of our 3D CAD model.

With the term standard we refer to the edge connections of the graph. In particular, every vertex, which is represented by the centroids of those cells that belong solely to the free-space of our 3D CAD model, is connected only with its immediate neighboring vertices along the x, y and z axes. In order for all these connections to be established, a ID-based vertex classification technique was devised.

On the assumption that the centroids of the voxels, i.e. vertices, are represented by points P_{ijk}, where $i=0,1,\ldots,numX$, $j=0,1,\ldots,numY$ and $k=0,1,\ldots,numZ$, then the unique ID of every on of these points is given by the formula below:

$$ID = i \cdot numY + j + k \cdot (numX \cdot numY) \tag{1}$$

where, numX, numY and numZ are the number of voxels in each of the respective x, y and z axes. The values resulting from this ID allocation method range from $[0,\ldots,(numX \cdot numY \cdot numZ)-1]$. It should be noted that all the centroids of the output file from the voxelizer engine are taken into account when considering the ID allocation method. In case a voxel is tagged as occupied, then the corresponding ID value will not be present in the ID pool of the graph's vertices. The above presented formula, is formulated in such way, as to match the output results of the voxelizer engine. An explanatory illustration of the ID allocation method can be seen in Figure 9.

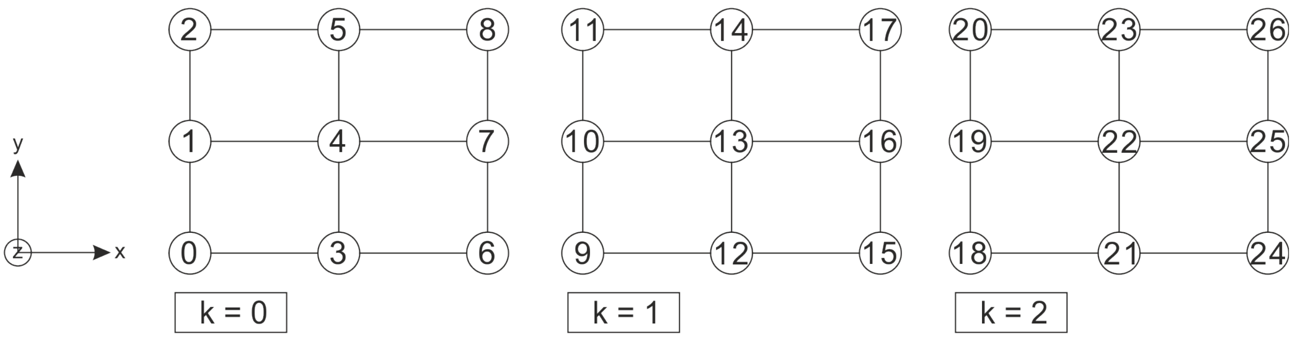


Figure 9: ID allocation of vertices in a simple 3x3x3 voxel grid

Having established this (i, j, k) index based ID allocation, makes the neighbor finding process a pretty straightforward procedure. Each ID refers to one, and only one, set of (i, j, k) indexes according to the following equations:

$$j = ID \ \% \ numY \tag{2}$$

$$i = ((ID - j) \ / \ numY) \ \% \ numX \tag{3}$$

$$k = ((ID - j - i \cdot numY) \ / \ numY) \ / \ numX \tag{4}$$

where, / and % denotes the integer division and the remainder after the integer division respectively.

As a result, a random $P_{(i, j, k)}$ point is connected with the following points: $P_{(i-1, j, k)}$, $P_{(i+1, j, k)}$, $P_{(i, j-1, k)}$, $P_{(i, j+1, k)}$, $P_{(i, j, k-1)}$, $P_{(i, j, k+1)}$, if of course these points exist and are not in an inaccessible area.

2.3.3.3. Diagonal graph

When considering the pipe-routing problem, most of the current studies that have been performed, utilizing a graph-search based algorithm either for mere path-finding or pipe-routing, only considered the freedom degrees of the previously presented standard graph. However practical and reasonable this approach might be, by imposing such a limitation to the degrees of freedom of movement within the graph, potential optimal solutions might be omitted. As a result, the current study also considers the diagonal freedom of movement, thus introducing the diagonal graph.

As we have already pointed out in the previous section, the term diagonal, which precedes the graph, refers to the edge connections. The currently discussed graph type, has all the edge connections that the standard graph has. In addition, each vertex is also connected with all of its immediate diagonal neighbors.

Considering the index based ID allocation formula and method that was presented earlier, a random $P_{(i, j, k)}$ point is connected with the following points:

$P_{(i-1, j, k)}$, $P_{(i+1, j, k)}$, $P_{(i, j-1, k)}$, $P_{(i, j+1, k)}$, $P_{(i, j, k-1)}$, $P_{(i, j, k+1)}$, $P_{(i+1, j+1, k-1)}$, $P_{(i+1, j, k-1)}$, $P_{(i+1, j-1, k-1)}$, $P_{(i, j+1, k-1)}$, $P_{(i, j-1, k-1)}$, $P_{(i-1, j+1, k-1)}$, $P_{(i-1, j, k-1)}$, $P_{(i-1, j-1, k-1)}$, $P_{(i+1, j+1, k)}$, $P_{(i+1, j-1, k)}$, $P_{(i-1, j+1, k)}$, $P_{(i-1, j-1, k)}$, $P_{(i+1, j+1, k+1)}$, $P_{(i+1, j, k+1)}$, $P_{(i+1, j-1, k+1)}$, $P_{(i, j+1, k+1)}$, $P_{(i, j-1, k+1)}$, $P_{(i-1, j+1, k+1)}$, $P_{(i-1, j, k+1)}$, $P_{(i-1, j-1, k+1)}$

Limitations regarding the existence of these potential neighboring points, as well as their accessibility, apply as always. In order to better understand the diagonal connectivity of the vertices in the current graph, the following illustrations are presented.

Figure 10, illustrates all the edge connections among the vertices of 2x2x2 grid. On the left we can see the extended diagonal connections marked with red. Right next to this we have the diagonal edge connections that lay on, or in parallel to planes xz, xy and yz. And finally on the right we can see the orthogonal connections between the vertices.

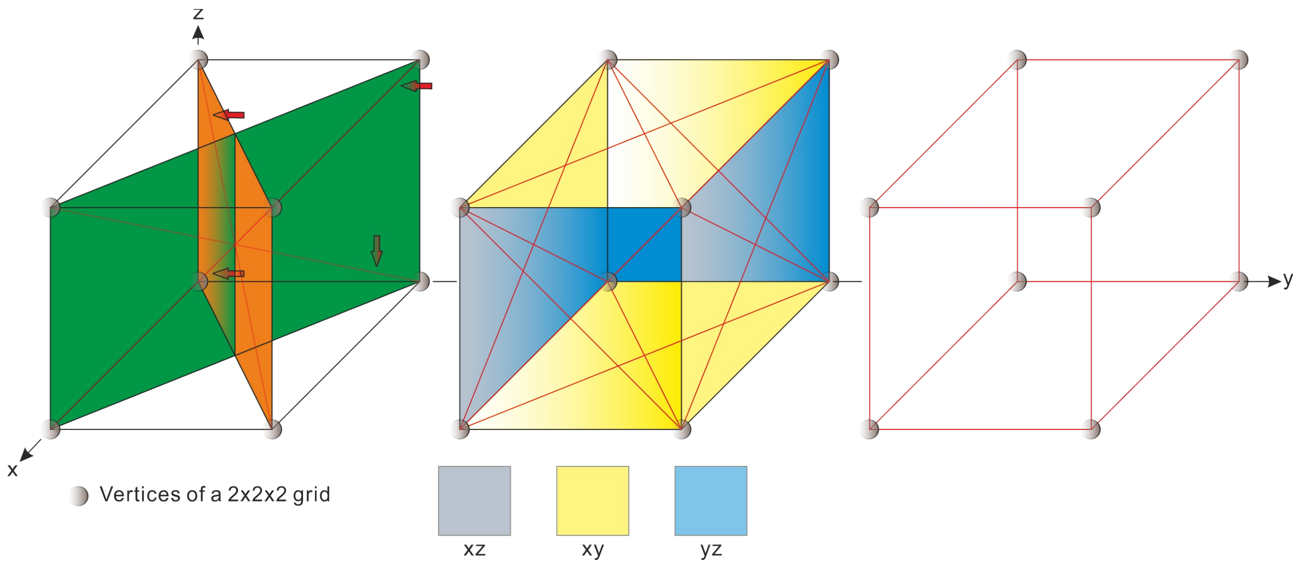


Figure 10: Connection within a diagonal graph voxel. Left: Extended diagonal edges; Middle: diagonal edges; Right: Standard orthogonal edges

Based on the illustration of Figure 10, in Figure 11 we can see all the edge connections of a particular vertex, in particular the one with an ID value of thirteen, within a diagonal 3x3x3 graph structure.

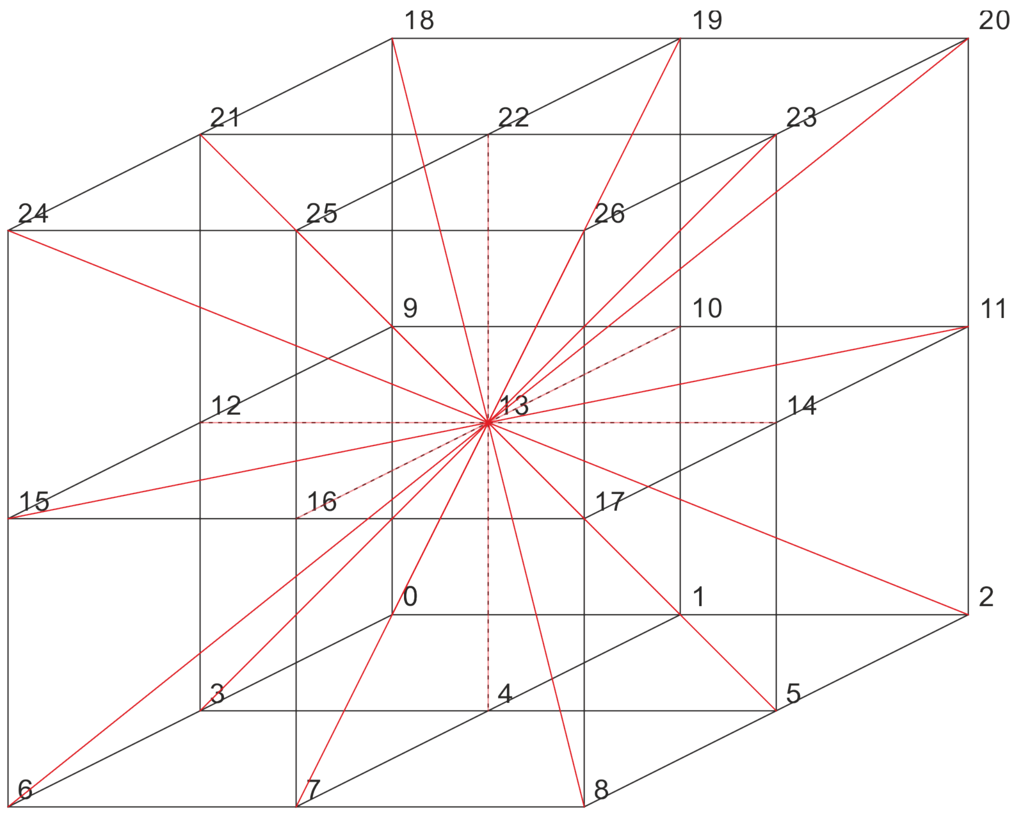


Figure 11: Connecting edges of vertex with ID = 13, in a diagonal graph

2.3.3.4. Vertex-split strategy

When considering a pipe-routing problem in a graph environment, it is all about finding the shortest path between a start vertex and an end vertex in the graph. Regardless of the graph representation being used, traditional path-finding algorithms cannot really tell the difference between a straight path to the end vertex and a path that includes a number of bends and turns. This fundamental issue arises from the fact that no directional information, regarding the relative position of the vertices in the proposed optimal path, are taken into consideration by the algorithms that perform the path-finding procedure. Thus, edges of the graph, cannot have their weights dynamically changed so as to penalize a potential bend of the path. The following illustration depicts this particular problem.

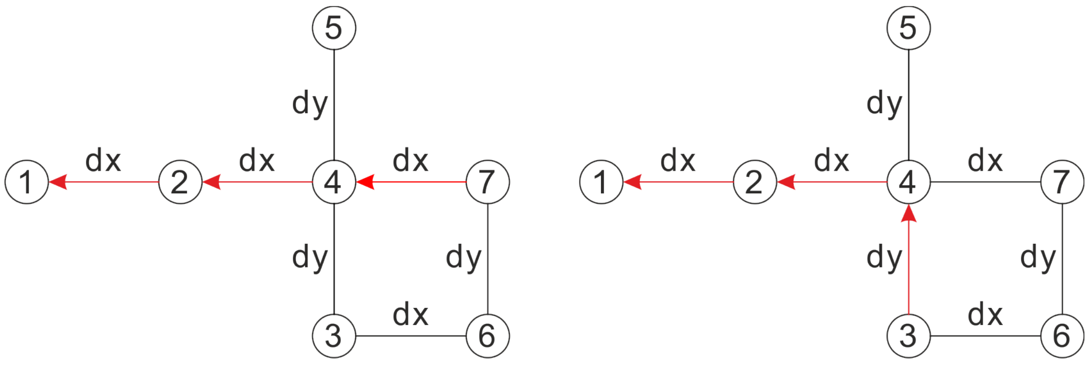


Figure 12: The problem of bend

As mentioned in the literature review section of this work, when dealing with the problem of bends in pipe-routing, solutions ranging from meta-heuristic optimization methods, to post processing the output of the path-finding algorithm have been devised and proposed. One rather interesting and innovative approach was presented by S-H Kim et al.[21], called vertex-split strategy. He clearly stated that:

"*A vertex with two or more incoming edges and outgoing edges should be split when outgoing edges can be used in a different route path, straight or bent simultaneously.... Practically, in a 3D cubic cell space where the number of neighbor vertices cannot exceed six, the number of split vertices is at most three in a case with three incoming edges and three outgoing edges.* "

In Figure 12, there is no way to really tell the difference between the two paths. The 7-4-2-1 path is all straight, whereas path 3-4-2-1 includes a bend. At the moment, the so far devised graph structure cannot somehow penalize this bending movement. One could argue that by putting a larger dy weight value in the edge connecting vertices 3 and 4, this could be averted. However, if that was the case, then the candidate path 3-4-5, would be penalized without good reason.

The current study proposes an extension of the vertex-split strategy. First of all, since the standard graph instance is regarded as undirected, the number of split-vertices in the current graph is raised to six at most, since every edge connecting this particular vertex with the neighboring ones serve both as incoming and outgoing at the same time. Furthermore, the vertex-split strategy is expanded, as to accommodate the needs of the diagonal graph as well. Each vertex is split to a number of split-vertices equal to the number of neighboring vertices, with this having a maximum value of twenty six. In both cases, every split-vertex is connected with each other, with new edges. For more information the reader is advised to refer to Section 2.4.3.1..

In Figure 13, the solution to the problem of bend, in a standard graph is presented. In this graph, route 1-2-4-6 is a straight one whereas route 1-2-4-3 is a bend one. With the vertex split strategy implementation, the edge corresponding to the straight connection and the bend connection is no longer the same, thus enabling the algorithm to choose between a straight path and a bend one. Consequently, the two presented paths in Figure 13, no longer have the same overall weight cost as they would have in an non-split graph.

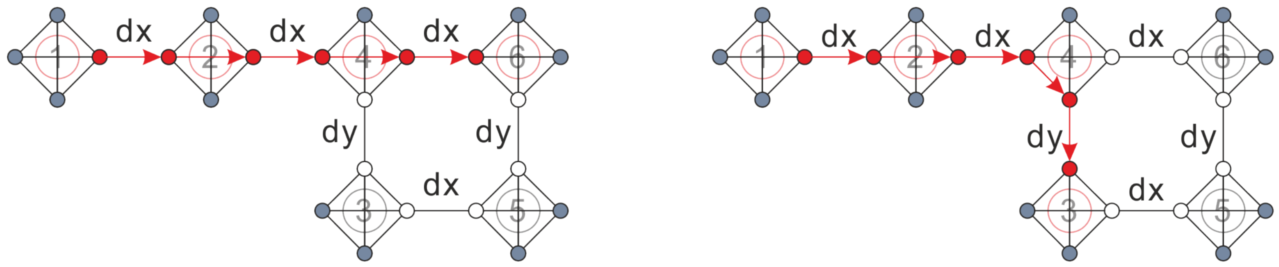


Figure 13: Solution of the problem of bend in a standard graph

In Figure 14, the solution to the problem of bend, in a diagonal graph is presented. Paths 1-2-4-6, 1-2-4-2 and 1-2-4-5 use different edges to reach their respective goal nodes. However, enabling the diagonal movement within this graph structure, expands the concept of penalizing a bend path, as it is introducing different types of bends (For more information, refer to Section 2.4.3. of the present study). Furthermore, Figure 14 depicts the internal connectivity between the split vertices of a random vertex.

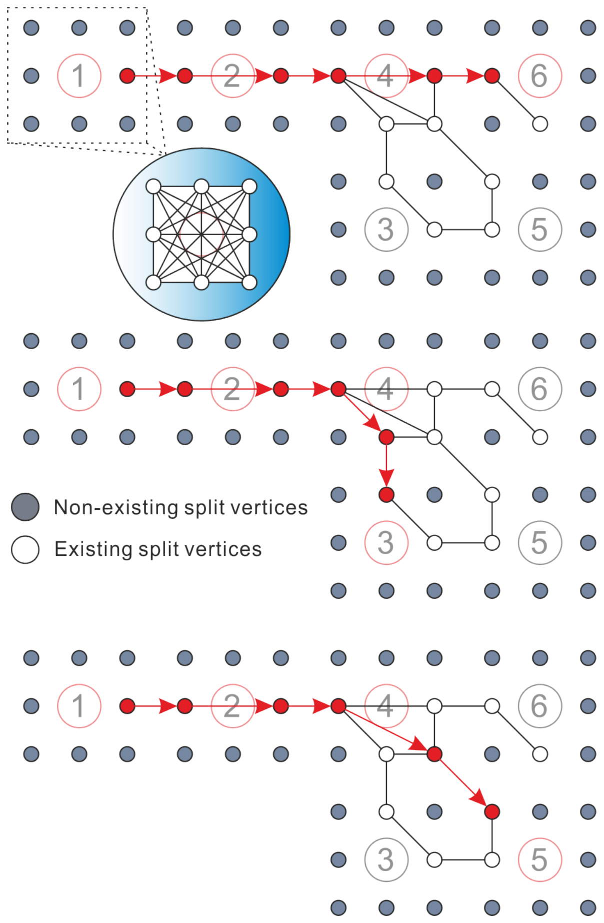


Figure 14: Solution of the problem of bend in a diagonal graph

The introduction of the split vertices present us with two distinct advantages:

- First and foremost, it enables us to differentiate between a straight and a bend pipe. Since every point is split to a number of vertices equal to the number of neighbors that it has, another set of edges is added to our graph, a set that we refer to as directional edges. After performing the vertex-split, every single one of the split vertices that originate from one particular vertex, is connected with every other using these directional edges. Whenever a path-finding algorithm reaches one of these split-vertices,while navigating through the edges of the graph, in order to find the shortest path from one vertex to another, it is faced with a choice. Each of the directional edges that refer to one of the split vertices, has a different weight value, based on whether the path will keep its previous course or change direction. These values are set as parameters to the problem of pipe-routing. Nevertheless, the edge weight corresponding to a straight movement is always significantly smaller than that of a bending movement. For more information on the weight allocation process, concerning these edges, refer to Section 2.4.3. of the current study.
- Secondly, it provides a unique opportunity, which can go some way towards giving a much needed expansion, as far as a directional specification aspect of the problem is concerned. Apart from enabling us to remove the ambiguousness of vertex connections withing the graph, it also gives us the opportunity to choose the direction of both the start an end point of each pipe-routing scenario. This addition, is highly important since the laying down of pipes happens after the positions of all the included equipment have been determined. This means, that both the start/ end coordinates and directional specifications of the problem are known beforehand. However, without introducing this expanded vertex-split strategy, considering directional constraints would be all but impossible, and an acceptable solution highly unlikely.

The introduction of the vertex-split strategy, despite of enabling the incorporation of directional specifications to the problem of pipe-routing, raises the complexity of the graph structure significantly, both in terms of vertex and edge numbers. On top of that, when considering the edges connecting the split-vertices with each other, in case of both the start and end vertex for every pipe-routing scenario, it should nor be possible to obtain a path that crosses more than one split-vertex of each of them, as this would not have any physical meaning. Figure 15 illustrates this very problem.

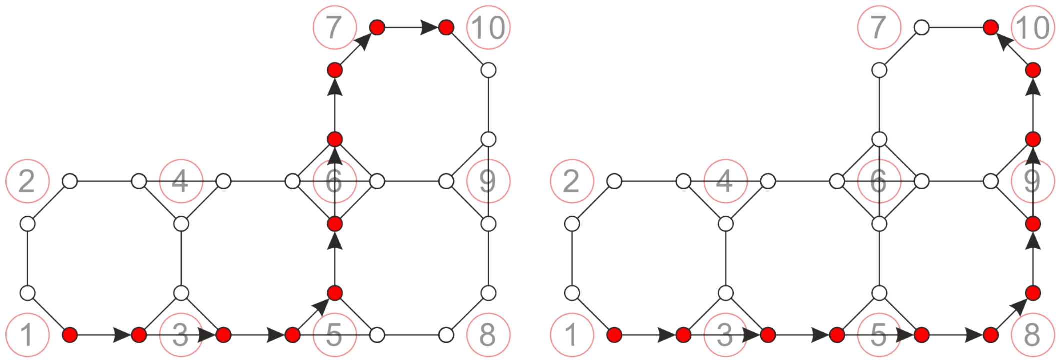


Figure 15: Left: Physically feasible solution; Right: Physically non-feasible solution

In order for this issue to be tackled, every time a pipe-routing scenario is considered, the edges connecting the split-vertices of both start and end vertices, are filtered out of the graph structure, as to avoid such confusing results.

2.4. Algorithmic approach

2.4.1. Introduction

2.4.2. Path – Finding Algorithms

Both Dijkstra and the A* belong to the class of network optimization algorithms. Optimization network techniques aim at seeking the optimal path forms of constructed networks by using graph search algorithms the likes of Dijkstra and A*.

In network based optimization each vertex (node) v_i denotes the junction of a pipe where either a straight or a bend pipe part can be placed; the edge e_{ij} between vertices v_i and v_j denotes the cost c_{ij} from moving from one to the other. The optimization problem is defined by the following equation:

$$G=(V,E,C) \tag{5}$$

where, V denotes the set of vertices, E the set of edges and C the set of edge costs. The pipe-routing optimization problem is to find the shortest path between the initial vertex S and the goal vertex G.

2.4.2.1. Dijkstra

2.4.2.1.1. Algorithm description

The Dijkstra's method is classed as a breadth-first search (BFS). It starts at the starting node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It was first conceived by computer scientist E W Dijkstra in 1956 and was published three years later[3].

As is the case with most shortest path algorithms, Dijkstra is run on weighted graph, starting from an initial node to a goal node and finds the least cost path to the goal node. Dijkstra assigns a tentative distance value to every vertex or node. It sets it to zero for our initial node, and infinity for the rest of the nodes. It also creates a visited set that starts with the initial node, and an unvisited set which starts with the rest of the nodes. It starts running at the initial vertex. For our current vertex we add the vertex to the visited set and remove it from the unvisited set. Then you calculate the distance to the current vertex plus the weight of the edge between the current vertex and its unvisited neighbors. If the value that you calculated for each of the neighbors is less than the current stored tentative distance to that vertex you replace the stored distance with the newly calculated value. When all the unvisited neighbors of the current node have been considered, the current node is marked as visited and it is then removed from the unvisited set and transferred to the visited set If the goal node has been marked visited, then the algorithm is finished, and the goal node has bee reached. Otherwise, the algorithm moves on to set the next current node, which will be the node with the lowest distance from the previously current node, repeating the aforementioned process until the goal node is finally reached.

Figure 16 shows a flow diagram of Dijkstra's algorithm.

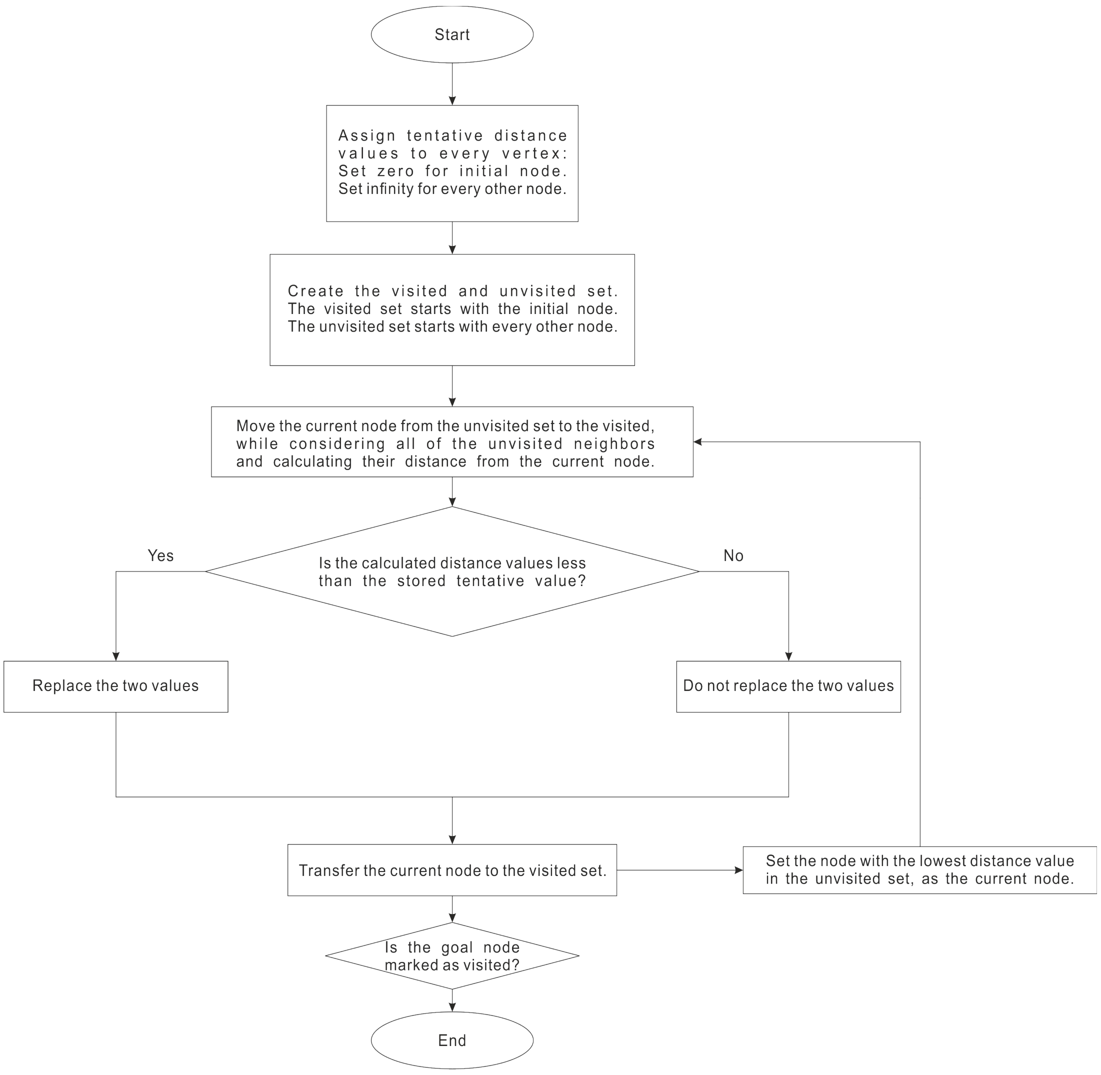
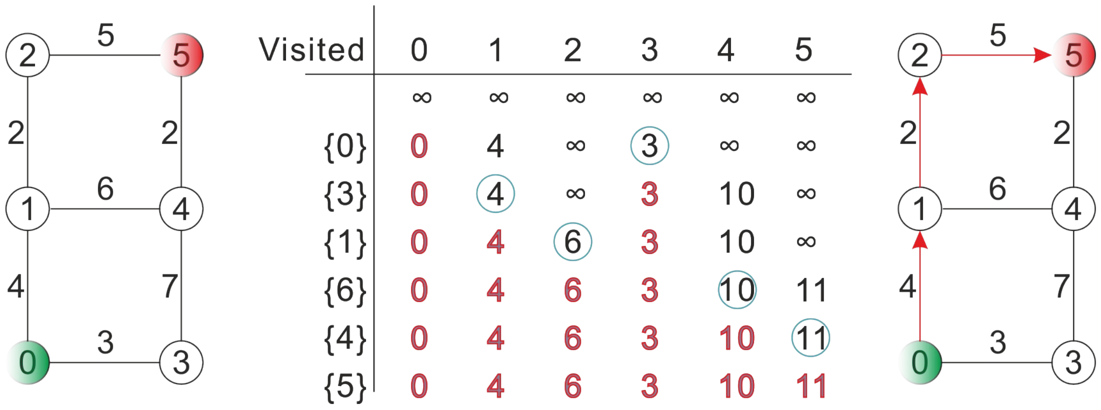

Start
Assign tentative distance values to every vertex: Set zero for initial node. Set infinity for every other node.
Create the visited and unvisited set. The visited set starts with the initial node. The unvisited set starts with every other node.
Move the current node from the unvisited set to the visited, while considering all of the unvisited neighbors and calculating their distance from the current node.
Is the calculated distance values less than the stored tentative value?
Yes
No
Replace the two values
Do not replace the two values
Transfer the current node to the visited set.
Set the node with the lowest distance value in the unvisited set, as the current node.
Is the goal node marked as visited?
End

Figure 16: Dijkstra's algorithm flow diagram

Figure 17 illustrates a simple example using Dijkstra's algorithm.



Visited	0	1	2	3	4	5
	∞	∞	∞	∞	∞	∞
{0}	0	4	∞	3	∞	∞
{3}	0	4	∞	3	10	∞
{1}	0	4	6	3	10	∞
{6}	0	4	6	3	10	11
{4}	0	4	6	3	10	11
{5}	0	4	6	3	10	11

Figure 17: Implemetation of Dijkstra from start node zero (0), to goal node five (5)

*2.4.2.2. A **

2.4.2.2.1. Algorithm description

Contrary to Dijkstra's algorithm, A* is classed as a depth-first search (DFS), namely it explores the highest-depth nodes first before being forced to backtrack and expand shallower nodes.

A* search algorithm is one of the best and most popular technique used in path-finding and graph-traversals. Unlike other traversal techniques, A* is a really “smart” algorithm, because it uses heuristics to guide its search. It can be seen as an extension of Dijkstra's algorithm. This path-finding technique has seen extensive use in many applications, such as game development, web-based map navigation etc, because it achieves better performance and accuracy than most traditional search algorithms through the use of heuristics.

Consider a graph with multiple nodes. The aim is to reach the goal node, or target node, from the initial, or starting node, as quickly as possible. What A* search algorithm does is that at each step, it picks the node according to a value “f” which is equal to the sum of “g” and “h”. At each step, it picks the node having the lowest “f” value, and process it.

$$f(n)=g(n)+h(n) \tag{6}$$

where, n is the previous node on the path, g(n) is the cost of the path from the start to node n and h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal node.

Typical implementations of A* use a priority queue to perform the repeated selection of the minimum estimated nodes to expand. Vertices on the priority queue, also known as open set, have been discovered by the algorithm, but they have not yet been expanded, meaning that their surrounding vertices have not been discovered yet. On the other hand, vertices marked as closed, meaning that they belong to the closed set, have been completely examined by the search algorithm, meaning that they have been expanded and their surrounding vertices have been added to the priority queue.

The algorithm begins by going to the starting node and expanding it, namely looking at all of its surrounding nodes and calculating some values for each of them. These values are the g and h cost . Basically, the g cost values represents the distance of the current node from the starting node, whereas the h cost is practically the opposite from g cost, meaning that its value quantifies the how far the current node is from the goal node. Having calculated these value pairs for each one of these neighboring nodes, it sums these two numbers, creating the so called f cost. The algorithm then proceeds by going to look at all of these nodes, and it is going to choose the one with the lowest f cost to look at first. Once it identifies the lowest f cost node, it marks it as closed, removes it from the queue and then calculates and updates, if needed, the g, h, and f cost values for all of its surrounding nodes, which in turn are added to the priority queue. The algorithm continues until the goal node has a lower f value than any other node within the priority queue, or if the priority queue is empty. The f value of the goal node is then equal to the cost of the resulting shortest path, as the h cost value will be equal to zero because by definition, the distance estimation for moving from the goal node to the same node is zero.

Figure 18 shows the flow diagram of the A* algorithm.

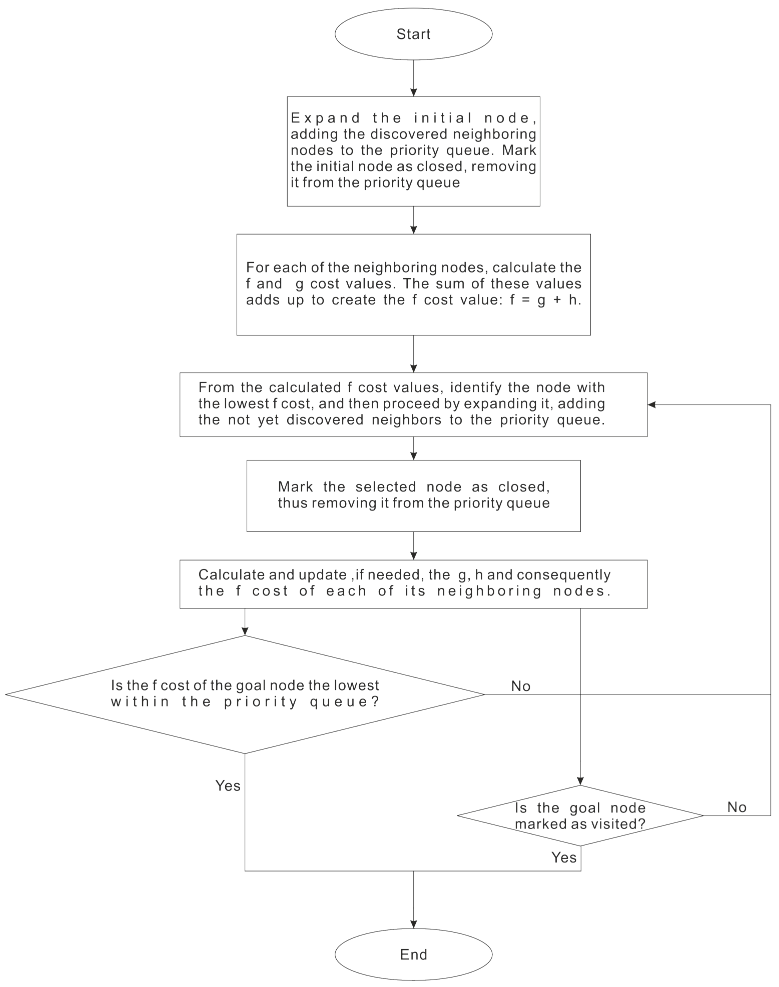
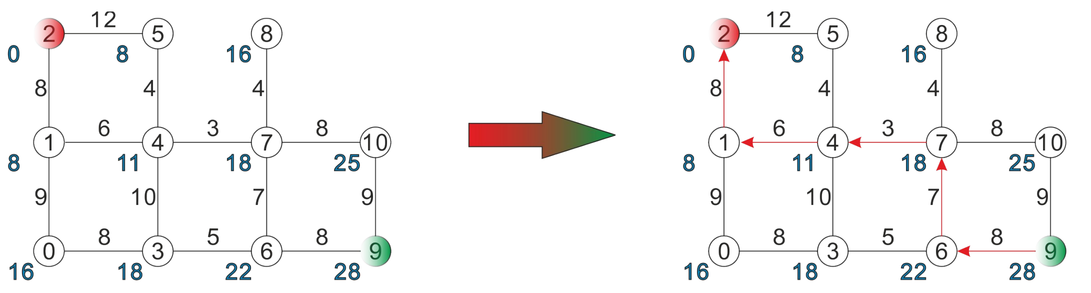

Start
Expand the initial node, adding the discovered neighboring nodes to the priority queue. Mark the initial node as closed, removing it from the priority queue
For each of the neighboring nodes, calculate the f and g cost values. The sum of these values adds up to create the f cost value: f = g + h.
From the calculated f cost values, identify the node with the lowest f cost, and then proceed by expanding it, adding the not yet discovered neighbors to the priority queue.
Mark the selected node as closed, thus removing it from the priority queue
Calculate and update ,if needed, the g, h and consequently the f cost of each of its neighboring nodes.
Is the f cost of the goal node the lowest within the priority queue?
No
Yes
Is the goal node marked as visited?
No
Yes
End

Figure 18: A algorithm flow diagram*

Figure 19 illustrates a simple example using the A*



	0	1	2	3	4	5	6	7	8	9	10	
Start node: {9}	0	1	2	3	4	5	6	7	8	9	10	Add 9 to priority queue
Start at node {9}:	∞	∞	∞	∞	∞	∞	30	∞	∞	-	34	Add 6, 10, remove 9
{6}	∞	∞	∞	31	∞	∞	30	33	∞	-	34	Add 3, 7, remove 6
{3}	37	∞	∞	31	34	∞	30	33	∞	-	34	Add 0, 4, remove 3
{7}	37	∞	∞	31	29	∞	30	33	35	-	34	Add 8, remove 7
{4}	37	32	∞	31	29	30	30	33	35	-	34	Add 1, 5, remove 4
{5}	37	32	34	31	29	30	30	33	35	-	34	Add 2, remove 5
{1}	37	32	32	31	29	30	30	33	35	-	34	Remove 1

The goal node has the lowest f cost within the priority queue.

Figure 19: Implemetation of Dijkstra from start node nine (9), to goal node two (2)

2.4.2.2.2. Heuristic functions

As we have previously illustrated, what sets the A* apart from its other counterparts, is that it utilizes a heuristic in order to guide its search while traversing any given graph. The heuristic function h(n), provides A* with an estimation of the minimum cost from any vertex n to a certain goal point. Thus, it is of vital importance to choose a proper and efficient heuristic function. The efficiency of a heuristic function, which is closely tied with the overall optimality of the A* algorithm, depends on the following two conditions:

(1) *Admissibility*: The heuristic function, must be admissible, meaning that it should never overestimate the actual minimum cost of reaching the goal node.

(2) *Consistency*: Because a closed set implementation of A* is considered, admissibility alone does not guarantee an optimal solution. The heuristic function also has to be consistent, meaning that given any pair of adjacent nodes n and n+1, where c(n, n+1) denotes the weight of the edge connecting them, the following condition must be met:

$$h(n) \leq c(n, n+1) + h(n+1) \tag{7}$$

In the current study, bearing in mind the aforementioned optimality requirements, the following three heuristic functions are considered:

- *Manhattan distance heuristic*: The Manhattan distance between two nodes, the current and the goal node, is defined as the sum of absolute values of the differences in the x, y, and z coordinates of the two nodes. The calculation is performed according to the following formula:

$$h = \sum_{i=x,y,z} |\ goal.i - current.i\ | \tag{8}$$

- *Euclidean distance heuristic*: The Euclidean distance between two nodes, the current and the goal node is defined as the square root of the sum of the squares of the differences between the corresponding coordinates of the two nodes. The calculation is performed according to the following formula:

$$h = \sqrt{((goal.x - current.x)^2 + (goal.y - current.y)^2 + (goal.z - current.z)^2)} \tag{9}$$

2.4.2.3. Comparative performance evaluation analysis

It is of vital importance to understand the different aspects that characterize these two algorithms in order to be able to proceed with a thorough performance evaluation analysis. This analysis will be based on the following three factors: RAM requirements, time complexity, optimal solution finding capability and multi-goal solving flexibility:

- *RAM requirements*: The A* takes lead over Dijkstra when considering their respective RAM requirements. Dijkstra, as a BFS algorithm, explores nodes on all different directions uniformly, thus being reminiscent of a circular wavefront expansion movement. This usually leads, to more nodes being discovered before the goal node is reached, compared to the A* which uses a heuristic in order to expand to directions which seem more promising regarding minimum distance from current to goal node. As a result, Dijkstra tends to occupy more memory than A*
- *Time complexity*: Bounds of the running time of both Dijkstra's algorithm and the A*, which are implemented in the current study, can be expressed as a function of the number of edges and of the number of vertices, using big-O notation. For Dijkstra we have a complexity of $\mathrm{O}(V \cdot \log V)$, whereas for A* we have a complexity of $\mathrm{O}((E+V) \cdot \log V)$, where V denotes the number of vertices within the graph, and E the number of edges.
- *Optimal solution finding capability*: Dijkstra is always guaranteed to find the optimal solution in any non-negative weighted graph environment regardless of complexity, if such a solution exists of course. On the other hand, the performance of the A* depends highly on heuristic function being used to guide the search. The use of an non-admissible, or non-consistent heuristic can result in sub-optimal solutions being produces.
- *Multi-goal solving flexibility*: Although the original variant of Dijkstra's algorithm solves the single-source shortest-path problem, it is possible to fix a single node as the source node and then find shortest paths from the source to all nodes in the graph, thus producing a shortest-path-tree. In this case, A* is not very optimal as it has to be run several times in order to get all the target nodes.

To this point, it is safe to say that both algorithms have their advantages and disadvantages. A comparison regarding which is the best choice, when it comes to pipe-routing problems, could be made. However, such a comparison goes beyond the aim of this thesis, which is to present both the capabilities and the limitations of each method, letting the reader decide whichever approach suits his or her interest in the best possible way.

2.4.3. Weight allocation

2.4.3.1. Definition

In many applications that require the use of a graph structure, pipe-routing included, each edge of the graph, be it normal or directional, has an associated numerical value, called weight. By normal we refer to the edges connecting vertices in the standard graph, whereas by directional we refer to the edges inter-connecting the split vertices of each individual original vertex. This distinction is illustrated in Figure 20. The utilization o Dijkstra and A* calls for the use of non-negative values as weights. The weight of an edge is often referred to as the cost of the edge, meaning the penalty that is imposed when moving from the one vertex end to the other.

In the present study, weights represent the actual distance between each vertex and its neighboring one. Furthermore, a distinction is made between normal and directional edges. In this section, the edge weight allocation methods for both normal and directional edges will be presented. Also, a directional tag (Dir-tag) allocation method is introduced, that will facilitate the weight allocation process.

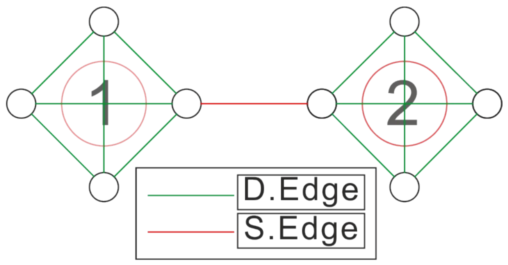


Figure 20: Standard and Directional edges depiction

2.4.3.2. Directional edges weight allocation

2.4.3.2.1. Standard graph

When considering a standard graph, two types of weights are allocated to the directional edges created by the application of the vertex-split strategy: the straight weight and the bend weight. Although the vertex-split enables us to have different edges that correspond with the different direction movement possibilities, the question still remains: How can we tell if one directional edge should be assigned a straight weigh value or a bend one? In order to be able to discern between the directional edges, a directional tag allocation methodology is devised. Every split vertex is assigned a Dir-tag, ranging from one to six, according to the illustration, presented in Figure 21. At this point it should be noted that all the split-vertices of a particular vertex are inter-connected, as we have already mention in Section 2.3.3.4., meaning that every single one of these split vertices are connected through a directional edge with all the others of the same vertex.

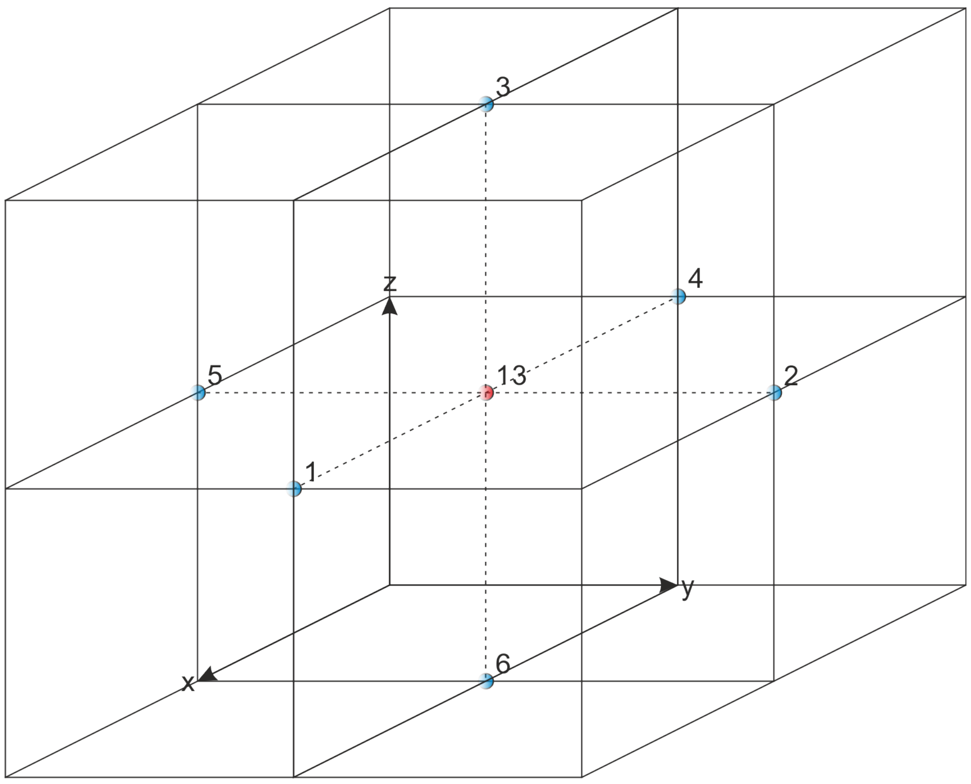


Figure 21: Dir-tag allocation of vertex with ID=13 in a standard graph

In the case of the standard graph, if the difference between the Dir-tags of the split-vertices that define a particular directional edge equals three, then a straight weight penalty is allocated to this edge. Otherwise, a bend weight penalty is allocated. Figure 22 illustrates the proposed weight allocation method concerning the directional edges of a standard graph.

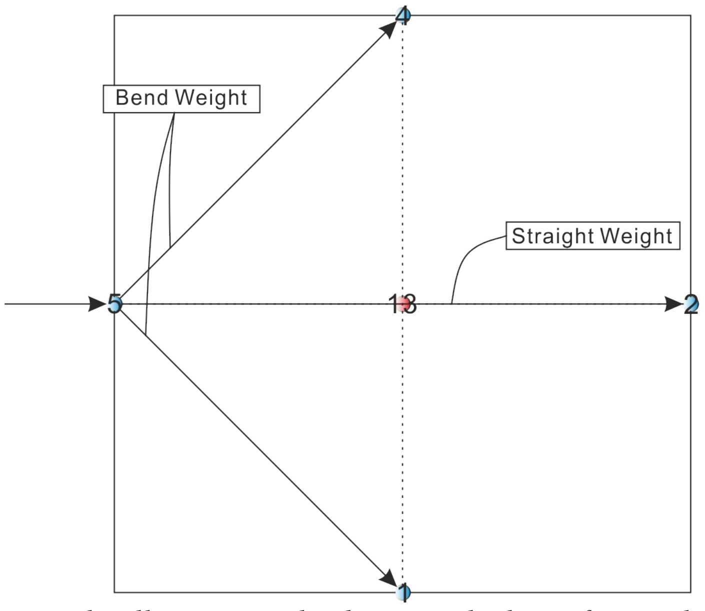


Figure 22: Weight allocation in the directional edges of a standard graph

2.4.3.2.2. Diagonal graph

The proposed directional tag allocation methodology can be expanded, so as to apply to the diagonal graph structure as well. In this case, every split vertex is assigned a directional tag (Dir-tag), ranging from one to twenty six, according to Figure 23. Again, as we have already mention in Section 2.3.3.4., all of the split-vertices of a particular vertex are inter-connected, meaning that every split-vertex is connected with all the other, which originate from the original vertex being split.

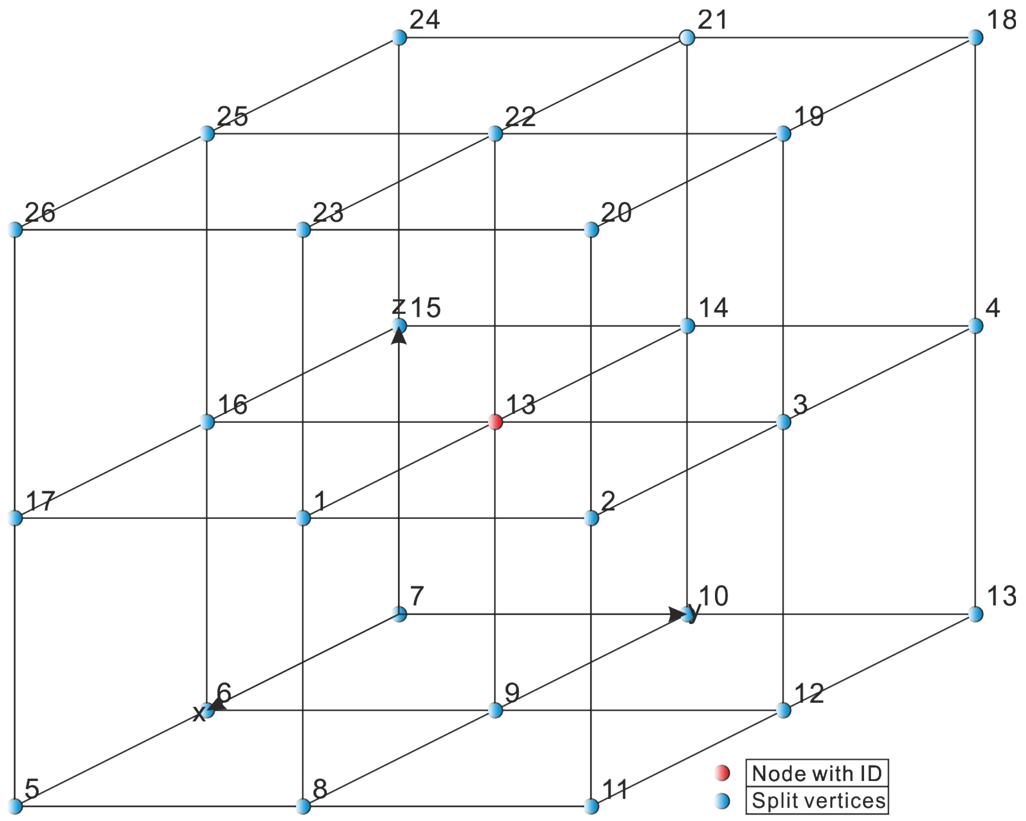


Figure 23: Dir-tag allocation of vertex with ID=13 in a diagonal graph

Since we are currently referring to the diagonal graph, based on the difference between the Dir-tags of the split-vertices that define a particular directional edge, we can discern the following cases:

(1) *Straight weight*: The straight weight value is allocated to an edge, if the following condition is met:

$$|directional\_edge.source.Dir\_tag - directional\_edge.target.Dir\_tag| = 13 \tag{10}$$

(2) *Bend weight*: The 90° bend weight value is allocated to an edge, if the following condition is met:

$$|directional\_edge.source.Dir\_tag - directional\_edge.target.Dir\_tag| \neq 13 \tag{11}$$

Figure 24 illustrates the proposed weight allocation method concerning the directional edges of a d graph.

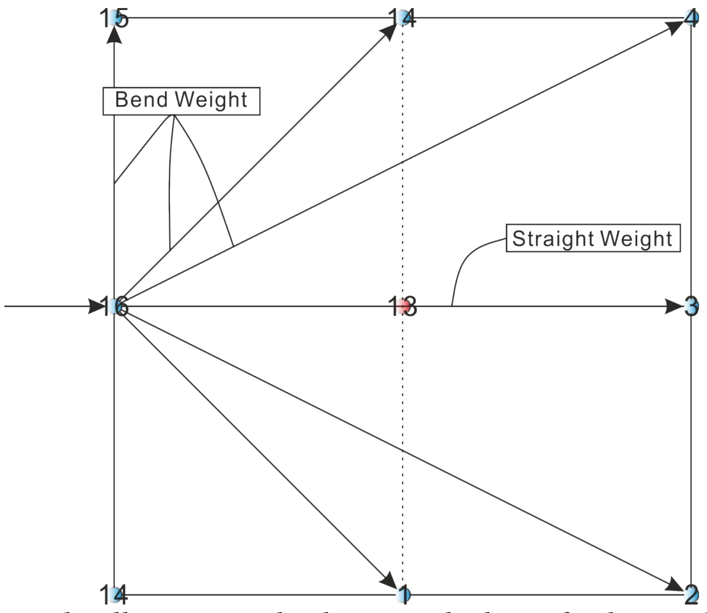


Figure 24: Weight allocation in the directional edges of a diagonal graph

This concept of differentiating only between straight and bend movement , although complete, lacks depth. This means that it penalizes all directional changes with the same value. Because of this, we expand the concept of directional change of movement, by adding some extra cases:

(1) *45^o bend weight*

(2) *Extended 45^o bend weight*

(3) *135^o bend weight*

(4) *Extended 135^o bend weight*

In order for these directional change cases to be better understood, refer to Figures 25 and 26, which depict these directional change weight penalties. At this point, it should be noted that the extended bends, which are presented in this section, are not actually equal to 45^o or 135^o bends. In reality these are custom and not standardized bends. However, the fact remains that changing direction towards them, should be penalized differently.

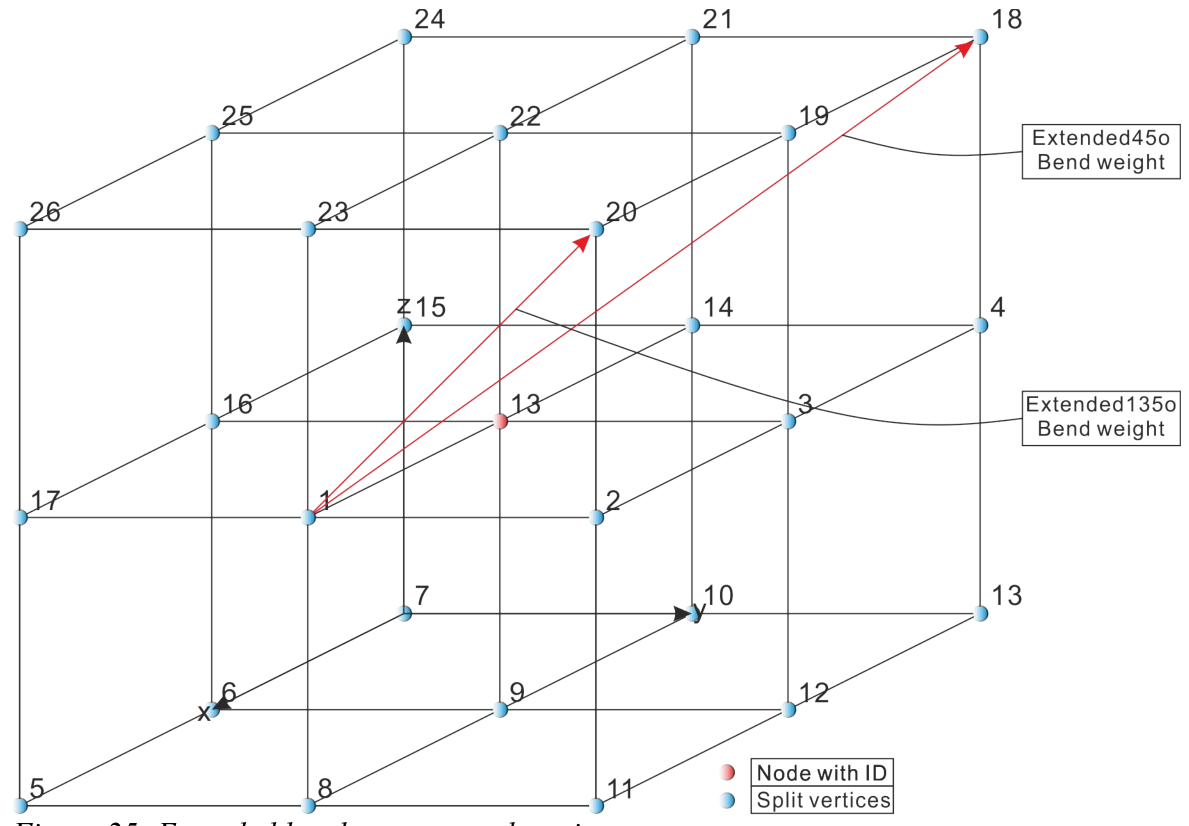


Figure 25: Extneded bend concept explanation

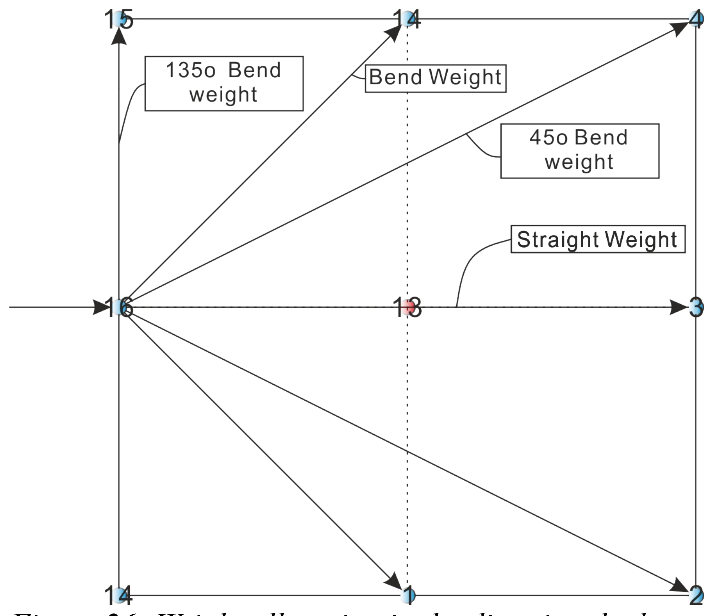


Figure 26: Weight allocation in the directional edges of a diagonal graph, considering different penalties

2.4.3.3. Normal edges weight allocation

2.4.3.3.1. Position-level based weight allocation

Generally speaking, space availability is crucial when considering the problem of pipe-routing. As a result, pipes being routed near walls or other equipment, which are regarded as obstacles, are more preferable than being routed through the middle of an available free space, because it causes movement obstruction and unnecessary waste of free space continuity. In order to achieve resulting optimal paths which are routed near obstacles within our layout workspace, a weight allocation method based on a position-level tag methodology is proposed.

When considering the non-split version of either the standard or the diagonal graph, every vertex in the graph is assigned a position-level tag (Pos-tag), according to the following procedure. Back when were talking about the implementation of the proposed voxelization technique, we reached to a point were every vertex within the decomposed layout workspace was assigned a value of one, if the vertex was in an accessible region, or zero if the vertex was interfering with an obstacle.

One way to figure whether a vertex belongs to the outermost layer of voxels in the graph, meaning next to an obstacle, is to check the number of neighboring vertices that it has. Depending on the type of graph being used at the time, the following apply:

- *Standard graph*: A vertex that has a number of neighbors other than six, is assigned a Pos-tag equal to zero, suggesting that it belongs to the outermost layer of voxels in the graph.

$$\textit{If neighbor count of current\_vertex} \neq 6 \rightarrow \textit{Pos\_tag} = 0 \tag{12}$$

- *Diagonal graph*: A vertex that has a number of neighbors other than twenty six, is assigned a Pos-tag equal to zero, suggesting that it belongs to the outermost layer of voxels in the graph.

$$\textit{If neighbor count of current\_vertex} \neq 26 \rightarrow \textit{Pos\_tag} = 0 \tag{13}$$

From this point on, vertices that are neighboring a zero value Pos-tag vertex, are tagged with a Pos-tag value of 1 etc. This operation continues until every single vertex has a Pos-tag value assigned to it. Afterwards, the average of the Pos-tag values of the start-end vertices of every edge is assigned to the edge.

The position-level based weight allocation uses this Pos-tag values of every edge within the graph, in order to assign an extra weight penalty factor to it. More specifically, the Pos-tag value is passed as an argument to a function, and the result, multiplied by a user defined non-negative multiplication factor, ranging from zero to a hundred, is the weight penalty factor that will be added to the corresponding edge. The developed software, provides the user with four function choices:

- *Exponential*: $f(\text{Pos-tag}) = e^{\text{Pos-tag}}$
- *Linear*: $f(\text{Pos-tag}) = \text{Pos-tag}$
- *Quadratic*: $f(\text{Pos-tag}) = \text{Pos-tag}^2$
- *Cubic*: $f(\text{Pos-tag}) = \text{Pos-tag}^3$

Figure 27, depicts the edge Pos-tag calculation process for a random graph edge.

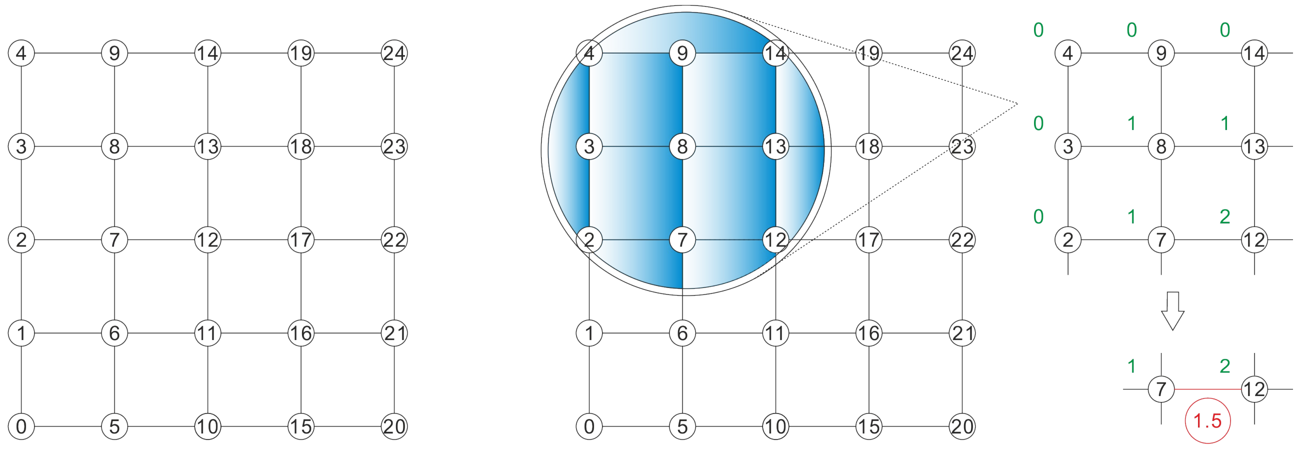


Figure 27: Edge Pos-Tag calculation process

2.4.3.3.2. Plane-distance based weight allocation

Apart from being able to route pipes along obstacles it would also be useful to be able to designate the desired distance from either floor or ceiling. Sometimes, routing pipes along the ceiling of an enclosed space is more preferable than routing them along the floor, while other times the opposite is true. Regardless of the scenario, the need for such a capability becomes evident. Consequently, a plane-distance weight allocation method is proposed.

In order for this method to be able to work the height position of the floor or deck planes within the 3D layout space CAD model have to be explicitly defined by the user, as a $\text{plane\_vector}[i], i=0,\dots,n$. This vector contains the characteristic coordinate value of each plane, be it x, y or z. Although this method can be used with pretty much any series of desired planes, be it x, y, or z, originally it was created bearing in mind the deck plane configuration within a ship.

Once this is done, the plane-distance method uses the average z coordinate value of each edge, in order to assign an extra weight penalty factor to it. At first, it identifies between which planes, from the ones provided beforehand, the average z coordinate value of each edge of the graph is included. Afterwards, this value is passed as an argument to a function and the result, multiplied by a user defined non-negative multiplication factor, ranging from zero to a hundred, is the weight penalty factor that will be added to the corresponding edge. As is the case for the previous method as well, the developed software, provides the user with four function choices:

- *Increasing linear*: Supposedly that the z coordinate value lies between the plane_vector[i] & plane_vector[i+1] values, meaning that the edge in question is located between the i and i+1 vertical planes. This function assigns penalty factor values according to the formula below:

$$(z-plane\_vector\,[i])\ /\ (plane\_vector\,[i+1]-plane\_vector\,[i]) \tag{14}$$

- *Increasing exponential*: This function assigns penalty factor values according to the formula below:

$$e^{(z-plane\_vector\,[i])\ /\ (plane\_vector\,[i+1]-plane\_vector\,[i])} \tag{15}$$

- *Decreasing linear*: This function assigns penalty factor values according to the formula below:

$$1-(z-plane\_vector\,[i])\ /\ (plane\_vector\,[i+1]-plane\_vector\,[i]) \tag{16}$$

- *Decreasing exponential*: This function assigns penalty factor values according to the formula below:

$$1-e^{((z-plane\_vector\,[i])\ /\ (plane\_vector\,[i+1]-plane\_vector\,[i]))-e} \tag{17}$$

For better understanding the aforementioned procedure, Figure 28 visualizes the first function choice, illustrating its use and functionality.

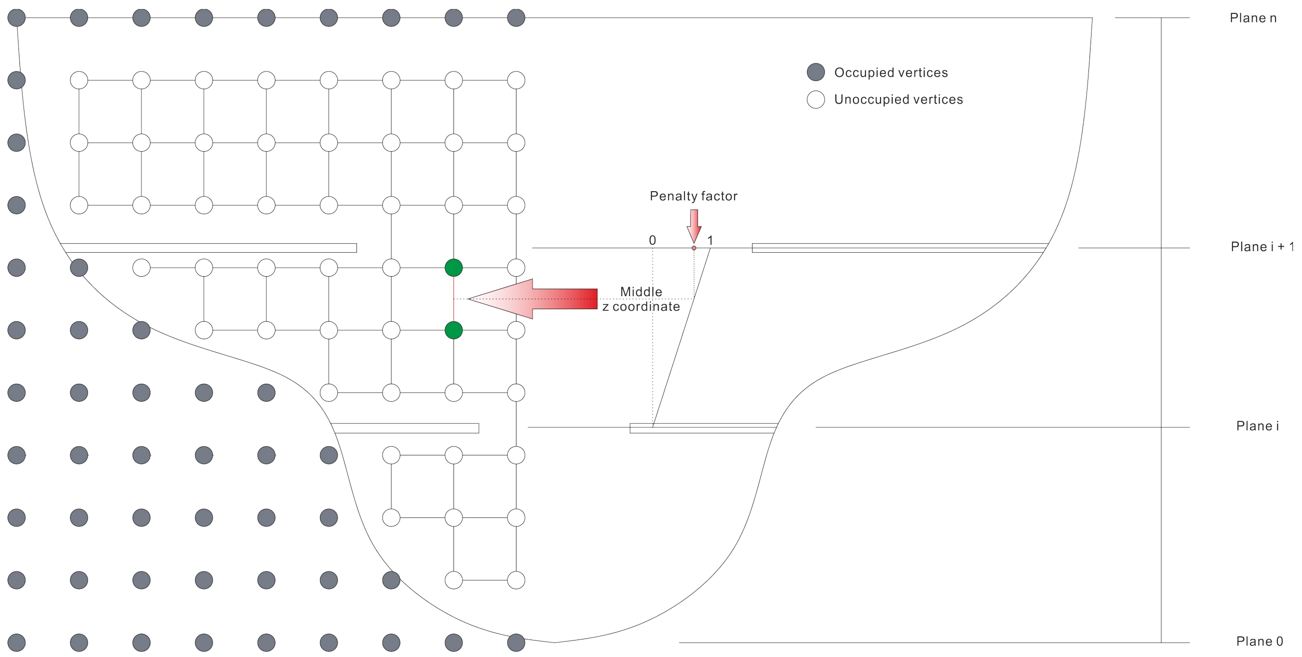


Figure 28: Implementation of the plane distance method

2.4.4. Path metrics

2.4.4.1. Evaluation parameters

At this point we have discussed about all the tools that will enable us to initiate the pipe-routing problem solution procedure. We have talked about different types of graph structures, different path-finding algorithms and also about elaborate weight allocation methods. What we lack is evaluation parameters, that will enable us to compare the path results, from different combinations of these methods.

For the purposes of the current thesis, the following parameters are introduced and considered in the comparative evaluation process:

(1) Number of bends

(2) Algorithm run time (s)

(3) Total x-axis component length (mm)

(4) Total y-axis component length (mm)

(5) Total z-axis component length (mm)

(6) Total path length (mm)

(7) Total distance from boundaries (mm)

(8) Number of boundary attached points (%)

(9) Maximum-minimum distance from boundaries (mm)

3. Pipe-routing software

3.1. Introduction

We have seen that pipe-routing is a complex problem, that requires a fare amount of work regarding both its modeling and algorithmic aspects. Decomposing the target layout workspace into voxels, creating a graph structure, allocating realistic weights its edges and finally running path-finding algorithms are all interconnected processes, that interact heavily with each other during any pipe-routing routine.

In the current study, in order to tackle these issues, a standalone application for automatic pipe-routing is developed. In the current chapter, both the configuration, as well as the user interface of the software in question are presented.

3.2. Configuration

The automatic pipe-routing application relies heavily on the voxelizer engine, which we presented earlier in this study. The output of the voxelizer, serves as the input to the pipe-routing software. Bearing this interaction in mind, the procedure which the application itself implements, consists of the following steps:

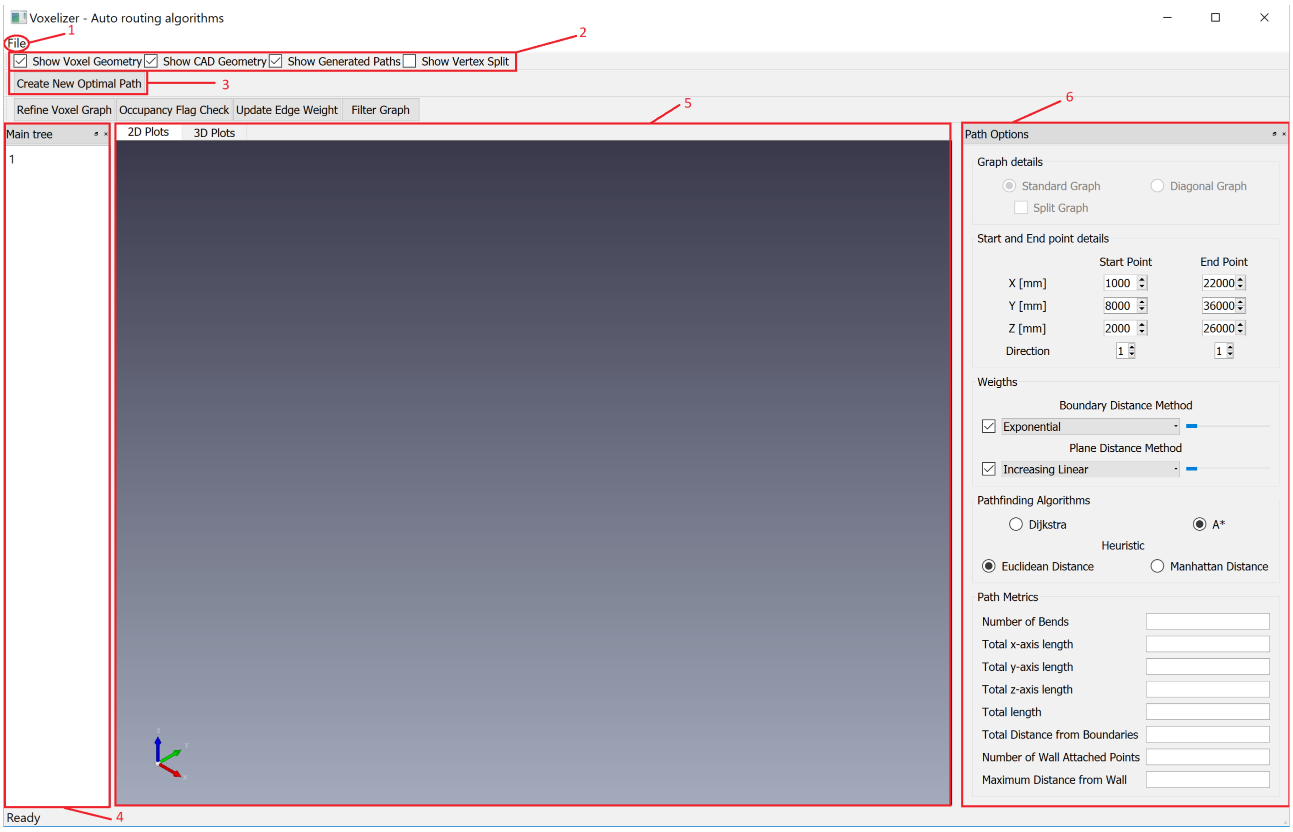
(1) *Input data upload*: During he input data upload step, the application obtains the output of the voxelizer engine in the form of a .vraw file, which in turn uploads as an input into the pipe-routing application. Furthermore, it generates the desired CAD model geometry, as defined in the .vraw file, while at the same time visualizing the cloud of points which will serve as the vertices of the graph structure that will be constructed later on. An example of such a file is presented in the Appendix A section of this thesis.

(2) *Route options setting*: The route options setting component step, is all about providing the user with a variety of choices regarding the pipe-routing procedure parameters, such as graph type, weight allocation methods, path-finding algorithm etc. The user can select and change the parameter set according to their choosing.

(3) *Result acquisition*: In the course of this final step, a visualization of the actual pipe model in the CAD model is created, while at the same time the user is provided with the necessary information regarding the path evaluation parameters. Last but not least, the resulting paths can be saved, in the form of a .path file. An example of such a file is presented in the Appendix B section of this thesis.

3.3 User interface

The developed pipe-routing software, provides an GUI, which is written using C++. In Figure 29 the user interface is presented.



[✓] Show Voxel Geometry [✓] Show CAD Geometry [✓] Show Generated Paths [ ] Show Vertex Split

Main tree: 1

Path Options

Graph details

[● Standard Graph] ○ Diagonal Graph

[ ] Split Graph

Start and End point details

	Start Point	End Point
X [mm]	1000	22000
Y [mm]	8000	36000
Z [mm]	2000	26000
Direction	1	1

Weigths

Boundary Distance Method

[✓] Exponential

Plane Distance Method

[✓] Increasing Linear

Pathfinding Algorithms

○ Dijkstra [● A*]

Heuristic

[● Euclidean Distance] ○ Manhattan Distance

Path Metrics

Number of Bends	
Total x-axis length	
Total y-axis length	
Total z-axis length	
Total length	
Total Distance from Boundaries	
Number of Wall Attached Points	
Maximum Distance from Wall	

Figure 29: GUI representation

As we can see, the GUI is comprised of several components, all of which serve a specific purpose. In particular:

(1) *“ **File** ” Menu bar*: Provides the user with a set of action choices, presented in Figure 30:

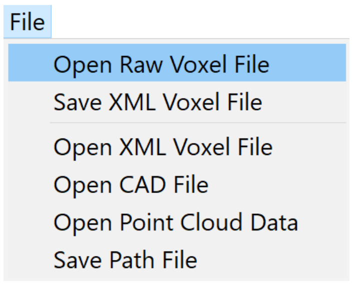


Figure 30: " File " Menu bar actions

(1) *Toolbar with check-boxes*: This toolbar contains two check-boxes, which enable the user to either show or hide both voxel geometry of a graph and the CAD model of the layout workspace, see Figure 29.

(2) *“* ***Create New Optimal Path*** *” Button*: This button initiates the each of the pipe-routing test runs, based on the parameters of the *“* ***Path Options*** *” docked window.*

(3) *Main tree docked window*: After each test run, a path item is added to this window. By right clicking on each one of these items, a custom context menu bar is revealed, that provides the user with the following actions, as depicted in Figure 31.

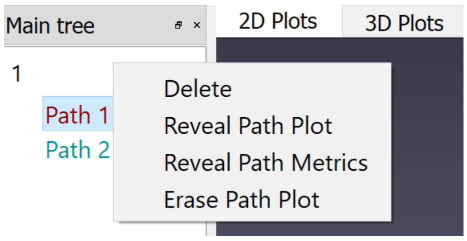


Figure 31: Custom context menu bar

(4) *Graphics main window*: This is the main window of the developed software application, where all visualization happens. This includes the representation of the graph vertices as a point cloud, the CAD model of the layout workspace and the resulting paths of each test run. Figure 32 illustrates all the aforementioned graphic representations.

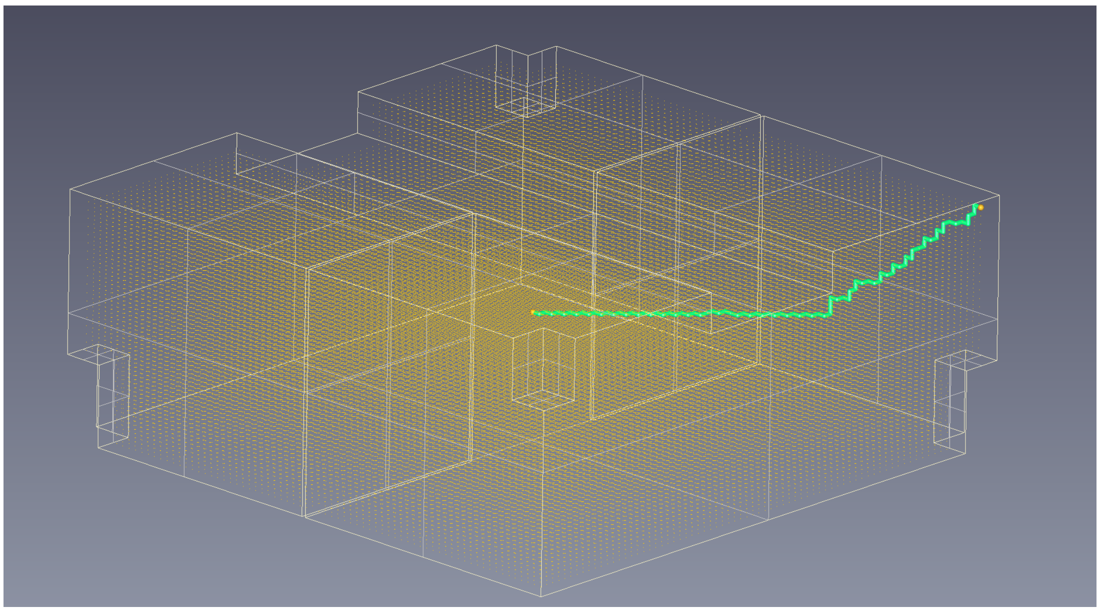


Figure 32: Graphics window visualization

(5) “ ***Path Options*** *” docked window*: As shown in Figure 29, this window contain every single parameter that can be adjusted by the user before each pipe-routing test run. It also contains a “ ***Path Metrics*** ” subsection, where the evaluation parameters are presented after each test run.

4. Case studies

4.1. Introduction

In order to verify the effectiveness of the developed pipe-routing software and its various modeling and algorithmic capabilities, a series of test cases are considered. These test cases refer to various CAD layout workspace representations, of increasing complexity. The idea behind the designing of these spaces is to best portray some performance and feasibility issues that come with traditional automatic pipe-routing solutions. On the other hand, these spaces will serve as a challenge to our proposed pipe-routing methodology, through which the improved results that are brought about by our approach to the problem can be most evident.

4.2 Test run parameters

Before proceeding with analyzing all the different test runs that have been conducted within the framework of the current Thesis, it is deemed necessary to refer in detail to the various parameters which determine the outcome of the pipe-routing process.

Path-finding parameters	**Comments**
Voxel Size (mm)	*Is directly connected with the original edge weights, dx, dy and dz of every edge*
Graph Type	*It has two values: Standard and Diagonal*
Split graph	*It has two values: True and False*
Start point (mm)	*It contains the x,y and z coordinates of the start vertex*
End point (mm)	*It contains the x,y and z coordinates of the start vertex*
Direction (S-E)	*It's value ranges from 1-6 for the standard graph, to 1-26 for the diagonal graph*
Position-level method	*It points to whether this weight allocation method is used or not*
Function	*It provides the following options: Exponential, Linear, Quadratic and Cubic*
Multiplier	*Its value ranges from 0 to 100*
Plane-distance method	*It points to whether this weight allocation method is used or not*
Function	*It provides the following options: Increasing Linear & Exponential, Decreasing Linear & Exponential*
Multiplier	*Its value ranges from 0 to 100*
Algorithm	*It provides the following options: Dijkstra, A**
Heuristic	*In case A* is chosen, it provides the following options: Euclidean and Manhattan*

Table 1: Explanation of test run parameters (1)

Path-finding parameters	**Comments**
Dir.“straight” weight	*It represents the penalty to pay in order for the path to keep its current course*
Dir.“45 bend” weight	*It represents the penalty to pay in order for the path to change its current course by 45º*
Dir. “bend” weight	*It represents the penalty to pay in order for the path to change its current course by 90º*
Dir.“135 bend” weight	*It represents the penalty to pay in order for the path to change its current course by 135º*
“E.45 bend” weight	*It represents the penalty to pay in order for the path to change its current course according to Figure 25*
“E.135 bend” weight	*It represents the penalty to pay in order for the path to change its current course according to Figure 25*
Diagonal weight mult/er	*It represents the value with which the standard diagonal edge weight is multiplied*
E. diagonal weight mult/er	*It represents the value with which the extended diagonal edge weight is multiplied*

Table 2: Explanation of test run parameters (2)

It should be noted, that the parameters previewed in Table 2, have a great potential for optimization, as tweaking them can have a great impact on the resulting path solution.

4.3. Test case setting

In order to prove the effectiveness of the developed pipe-routing software, and consequently of the methods that it utilizes in order to tackle the problem of automatic pipe-routing, five test case models are used:

- *Empty cubic space*
- *Simple room space*
- *Compound space*
- *Complex compound space*
- *Simplified engine room space*

The presented approach is implemented in Visual C++ 2017, under Windows 10, x64 bit OS (Microsoft Corporation). All of the test runs were conducted using a laptop with Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2208 MHz, 6 Core(s), 12 Logical Processor(s). At this point it should be noted that a multitude of test runs for several test cases were conducted over an extended period of time. This led us to gain some much needed experience, as far as picking the right parameters for each individual pipe-routing problem is concerned. Consequently, all of the parameters and path option settings that were used in each test run for the following cases, were inferred from a strenuous trial and error process. These parameters and options are presented and discussed in detail in the following sections.

4.3.1. Model 1: Empty cubic space

4.3.1.1 Test case presentation

Three sets of test runs were conducted for the current model. The first set, consists of three test runs, which aim to illustrate the different results that the Dijkstra and the A* algorithm produce, in a standard weighted graph. As far as the weight allocation methods are concerned, no further penalty was considered, meaning that the edge weights are equal to the respective dx, dy and dz parameters of the voxelized space. Both the Euclidean and the Manhattan distance heuristics were considered. Table 3 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #1*	*Test run #2*	*Test run #3*
Voxel Size (mm)	100	100	100
Graph Type	Standard	Standard	Standard
Split graph	False	False	False
Start point (mm)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
End point (mm)	(1900, 1900, 1900)	(1900, 1900, 1900)	(1900, 1900, 1900)
Direction (S-E)	Not applicable	Not applicable	Not applicable
Position-level method	Not utilized	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized	Not utilized
Algorithm	Dijkstra	A*	A*
Heuristic	-	Euclidean	Manhattan
Dir. “straight” weight	1	1	1
Dir. “bend” weight	20	20	20

Table 3: Model 1: First set test run parameters

The second set now consists of two test runs. Now, both Dijkstra and A* are used in a split, standard weighted graph. Again, no further penalty is applied to the edge weights, while for the A* only the Euclidean distance heuristic is considered. Furthermore, directional specifications are considered. Table 4 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #4*	*Test run #5*
Voxel Size (mm)	100	100
Graph Type	Standard	Standard
Split graph	True	True
Start point (mm)	(0,0,0)	(0,0,0)
End point (mm)	(1900,1900,1900)	(1900,1900,1900)
Direction (S-E)	1-6	1-6
Position-level method	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized
Algorithm	Dijkstra	A*
Heuristic	-	Euclidean
Dir. “straight” weight	1	1
Dir. “bend” weight	20	20

Table 4: Model 1: Second set test run parameters

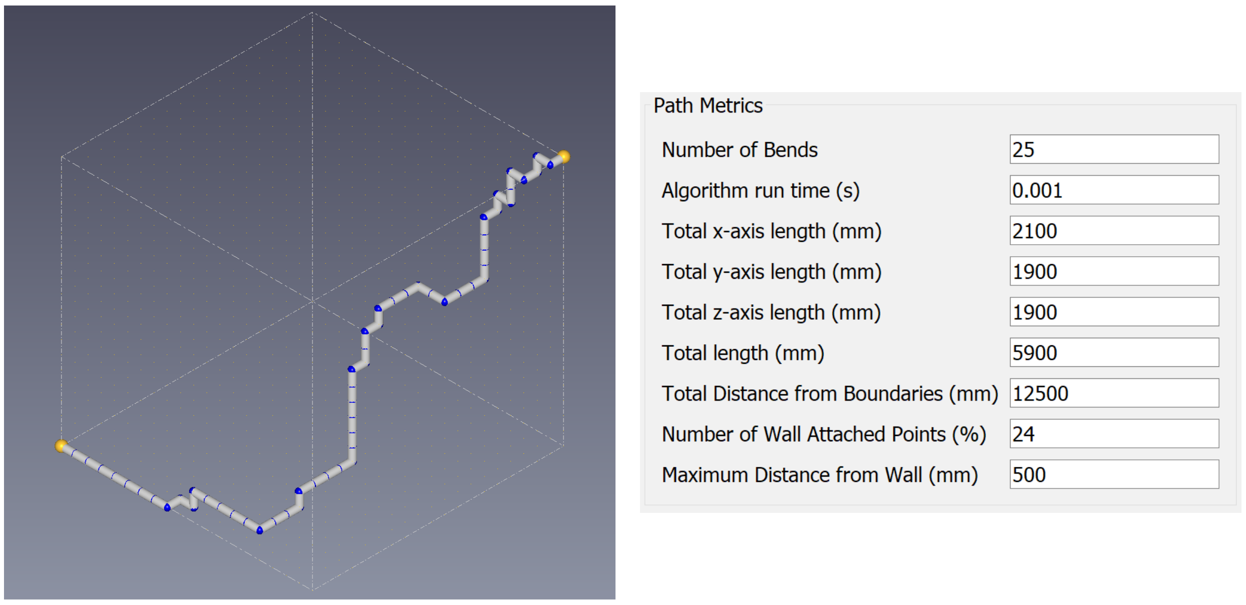
For the third, and final set for this model, two groups of six test runs are considered. Dijkstra is used in a split, standard weighted graph. As with the other two sets, no added penalty is considered. Last but not least, directional specifications play a pivotal role in this set. Table 5 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Group 1 (Test run #6 - #11)*	*Group 2 (Test run #12 - #17)*
Voxel Size (mm)	100	100
Graph Type	Standard	Standard
Split graph	True	True
Start point (mm)	(500,500,500)	(500,500,500)
End point (mm)	(1700,1700,1700)	(1700,1700,1700)
Direction (S-E)	(1:6)-6	1-(1:6)
Position-level method	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized
Algorithm	Dijkstra	Dijkstra
Heuristic	-	-
Dir. “straight” weight	1	1
Dir. “bend” weight	20	20

Table 5: Model 1: Third set test run parameters

4.3.1.2. Results illustration

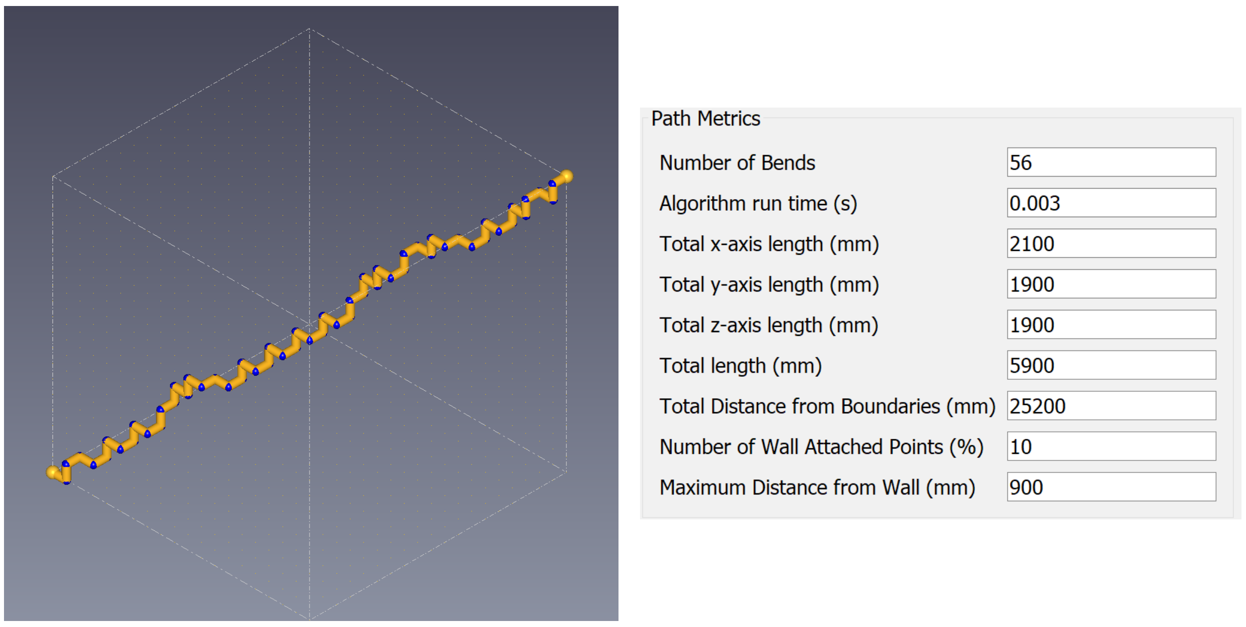
Based on on the data presented in Tables 3 to 5, the results of the various test runs are presented in the figure illustrations below.



Path Metrics

Number of Bends	25
Algorithm run time (s)	0.001
Total x-axis length (mm)	2100
Total y-axis length (mm)	1900
Total z-axis length (mm)	1900
Total length (mm)	5900
Total Distance from Boundaries (mm)	12500
Number of Wall Attached Points (%)	24
Maximum Distance from Wall (mm)	500

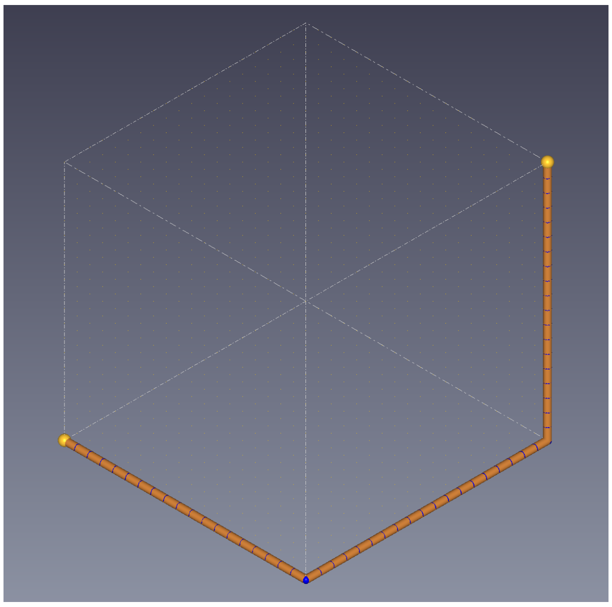
Figure 33: Test run #1 illustration and metrics

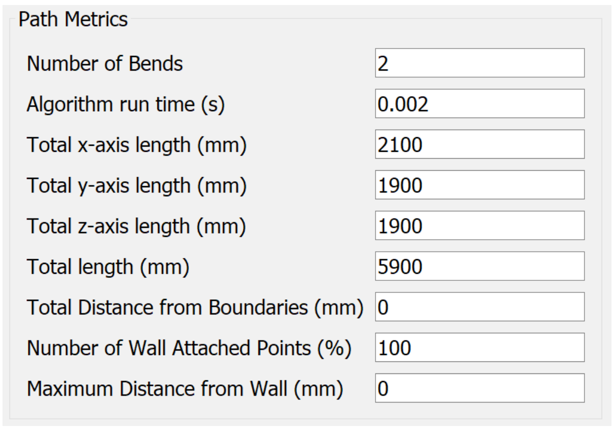


Path Metrics

Number of Bends	56
Algorithm run time (s)	0.003
Total x-axis length (mm)	2100
Total y-axis length (mm)	1900
Total z-axis length (mm)	1900
Total length (mm)	5900
Total Distance from Boundaries (mm)	25200
Number of Wall Attached Points (%)	10
Maximum Distance from Wall (mm)	900

Figure 34: Test run #2 illustration and metrics

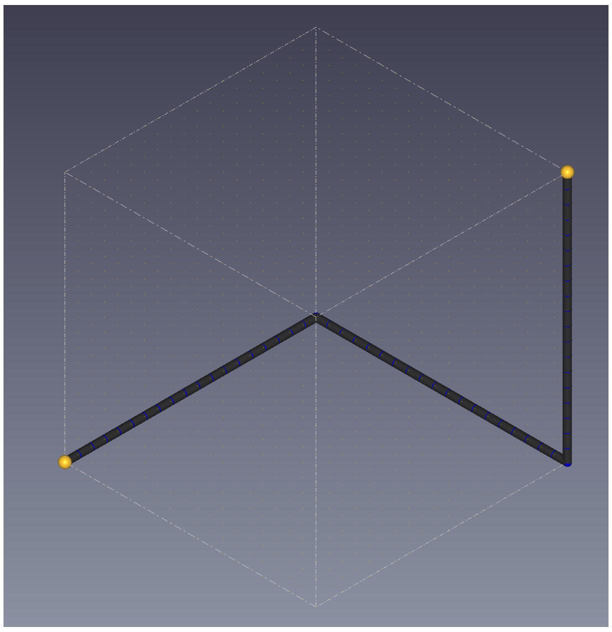


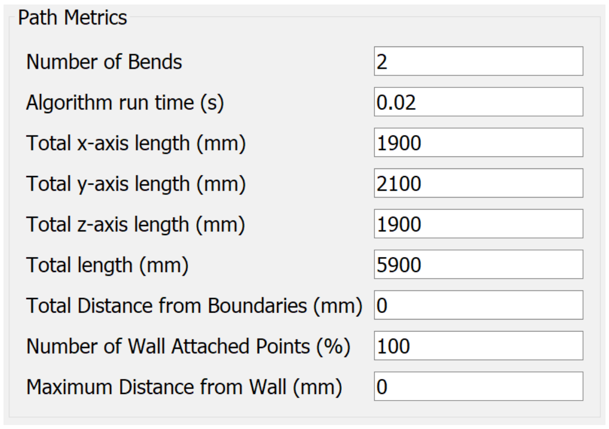


Path Metrics

Number of Bends	2
Algorithm run time (s)	0.002
Total x-axis length (mm)	2100
Total y-axis length (mm)	1900
Total z-axis length (mm)	1900
Total length (mm)	5900
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 35: Test run #3 illustration and metrics

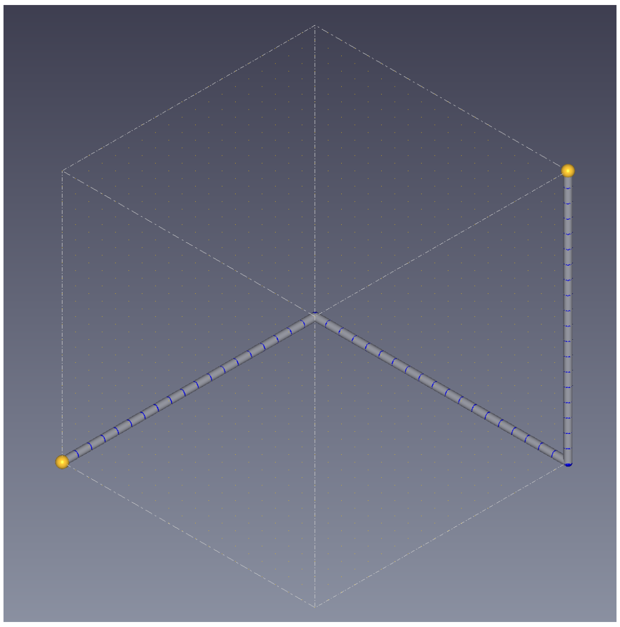


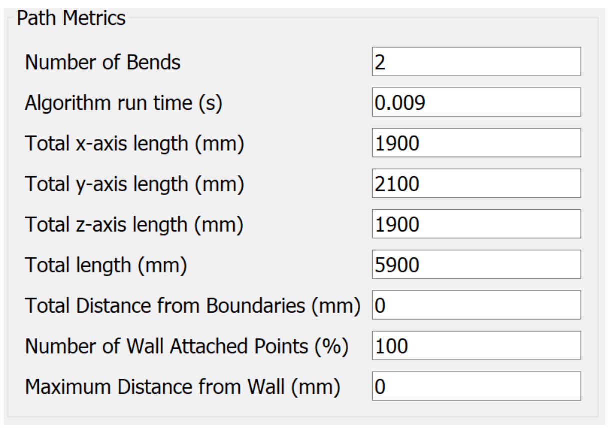


Path Metrics

Number of Bends	2
Algorithm run time (s)	0.02
Total x-axis length (mm)	1900
Total y-axis length (mm)	2100
Total z-axis length (mm)	1900
Total length (mm)	5900
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 36: Test run #4 illustration and metrics





Path Metrics

Number of Bends	2
Algorithm run time (s)	0.009
Total x-axis length (mm)	1900
Total y-axis length (mm)	2100
Total z-axis length (mm)	1900
Total length (mm)	5900
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 37: Test run #5 illustration and metrics

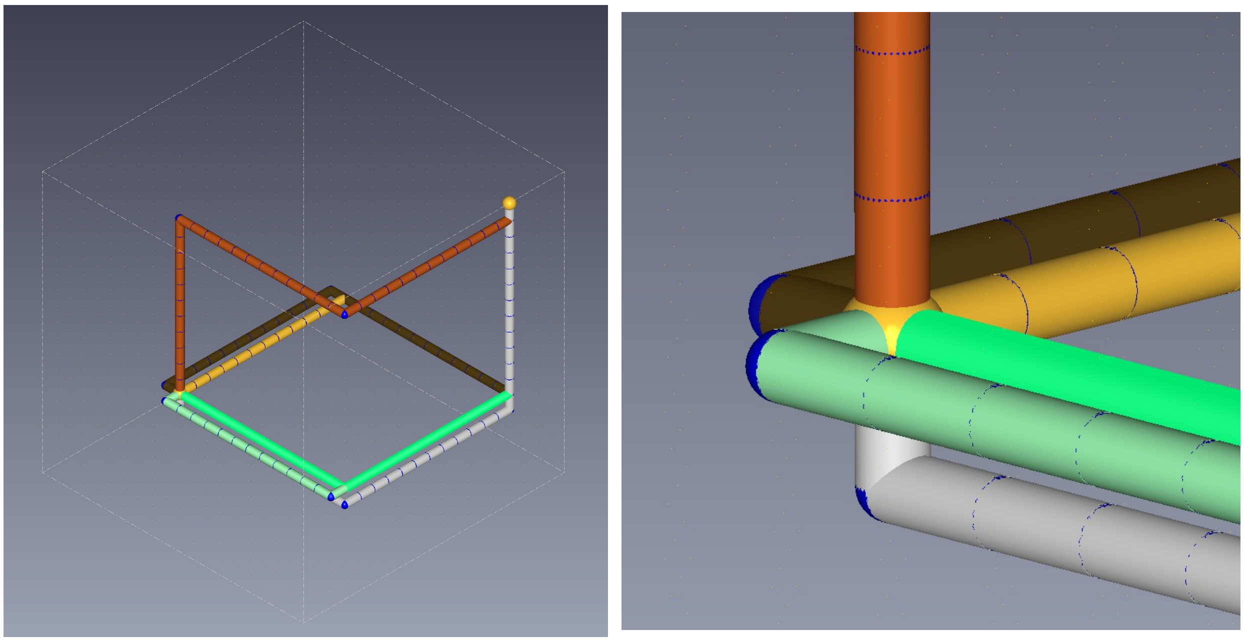


Figure 38: Test run #6 - #11 illustration and close-up on directional specifications

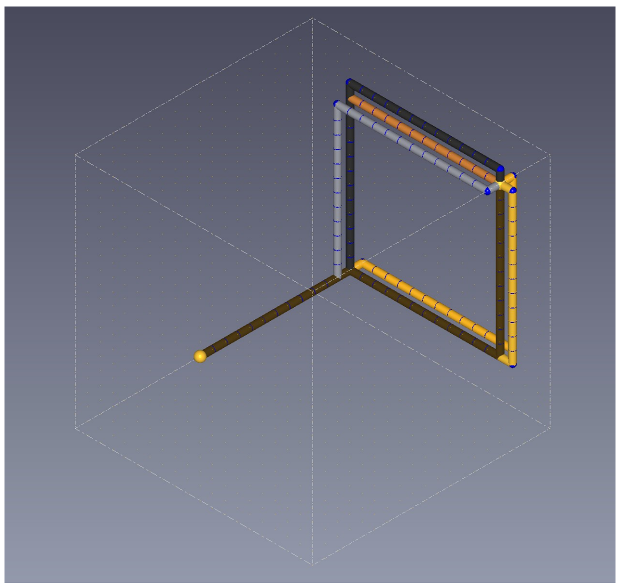
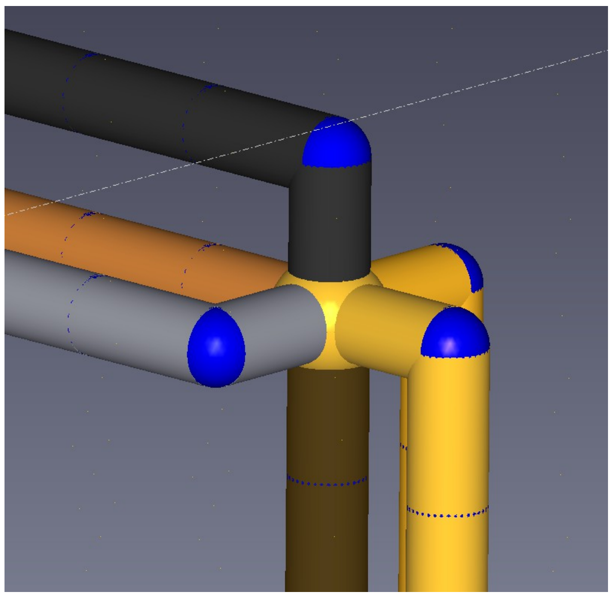



Figure 39: Test run #12 - #17 illustration and close up on directional specifications

4.3.1.3. Results assessment

Considering the previous test runs, the following conclusions can be drawn:

(1) By comparing the results from Figures 33 – 37, it becomes apparent that the implementation of the proposed vertex-split strategy goes along way towards drastically reducing the number of bends in each resulting path. It should be noted that the number of bends dropped from 56 and 25 to a mere 2.

(2) While the previous conclusions is without doubt true, if one takes a closer look at Figures 33 35, we can see that the performance of the A* is really affected by its heuristic. In particular, from these results it becomes apparent that by using the right heuristic function for each situation, in this case the Manhattan distance heuristic, solid path solutions can occur, regarding the number of bends, even without the use of the split graph.

(3) Last but not least, by considering the results from Figures 38 and 39, it is clear that the directional specifications, which are provided to the algorithm in the form of the Dir-Tag values, are met with success. However, care should be taken as to assess whether the proposed solutions can truly be feasible.

4.3.2. Model 2: Simple Room space

4.3.2.1. Test case presentation

At the previous section we have illustrated some of the main functionalities of the pipe-routing software. However, the layout space of our model could be described as anything but complicated, given the fact that it was both a simple box and completely obstacle free. The rest of the models presented in the current work, increase the complexity of the layout space gradually, helping us to see how our proposed pipe-routing method can cope.

Two sets of test runs were conducted for the current model. The first set, consists of two test runs, both of which are using the A*, with its two different heuristic function options. Furthermore, the split standard graph is used. None of the two weight allocation methods are used for these test runs. Table 6 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #18*	*Test run #19*
Voxel Size (mm)	100	100
Graph Type	Standard	Standard
Split graph	True	True
Start point (mm)	(600, 100, 0)	(600, 100, 0)
End point (mm)	(7300, 7300, 2800)	(7300, 7300, 2800)
Direction (S-E)	1-6	1-6
Position-level method	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized
Algorithm	A*	A*
Heuristic	Manhattan	Euclidean
Dir. “straight” weight	1	1
Dir. “bend” weight	20	20

Table 6: Model 2: First set test run parameters

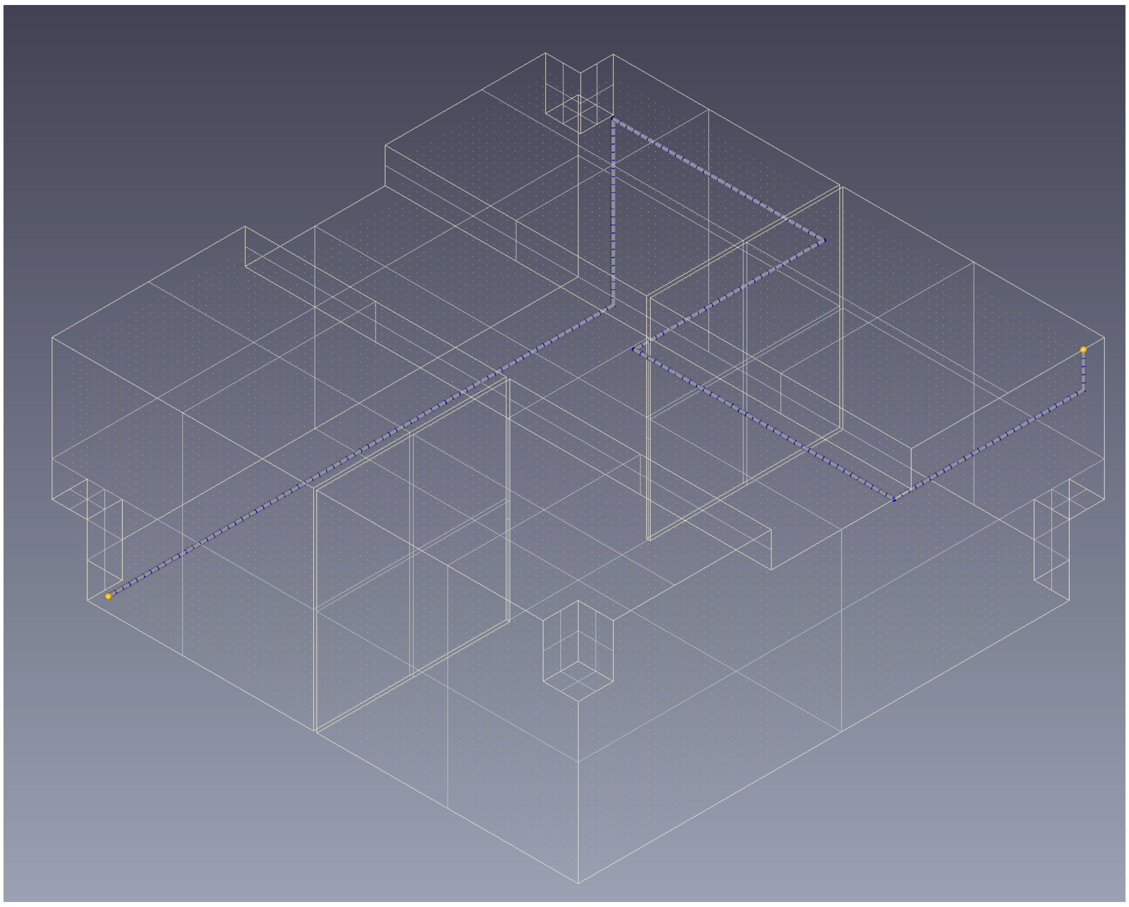
The second set, consists of three test runs. This time, Dijkstra is used. Furthermore, the split standard graph is used. In the first test run, no weight allocation method is used, whereas in the second and third test run, the position-level and the plane-distance weight allocation methods are used respectively. Table 7 contains all the information needed to recreate each of these individual test runs.

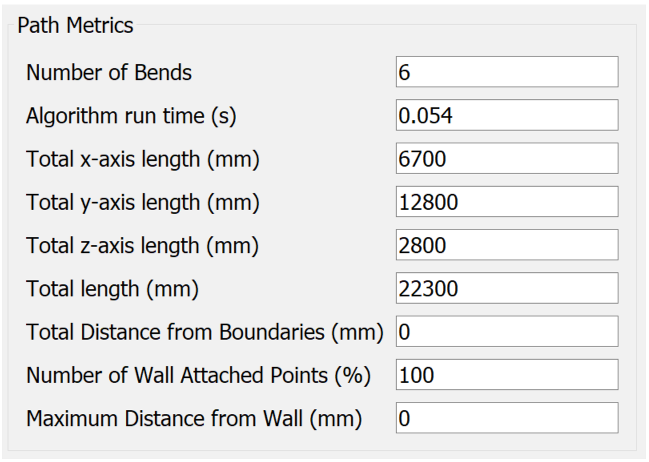
Path-finding settings	*Test run #20*	*Test run #21*	*Test run #22*
Voxel Size (mm)	100	100	100
Graph Type	Standard	Standard	Standard
Split graph	True	True	True
Start point (mm)	(600, 100, 0)	(600, 100, 0)	(600, 100, 0)
End point (mm)	(7300, 7300, 2800)	(7300, 7300, 2800)	(7300, 7300, 2800)
Direction (S-E)	1-6	1-6	1-6
Position-level method	Not utilized	✓	✓
Function	-	Exponential	-
Multiplier	-	100	-
Plane-distance method	Not utilized	✓	✓
Function	-	-	Decreasing exponential
Multiplier	-	-	100
Algorithm	Dijkstra	Dijkstra	Dijkstra
Heuristic	-	-	-
Dir. “straight” weight	1	1	1
Dir. “bend” weight	20	20	20

Table 7: Model 2: Second set test run parameters

4.3.2.2. Results illustration

Based on on the data presented in Tables 6 and 7, the results of the various test runs are presented in the figure illustrations below.

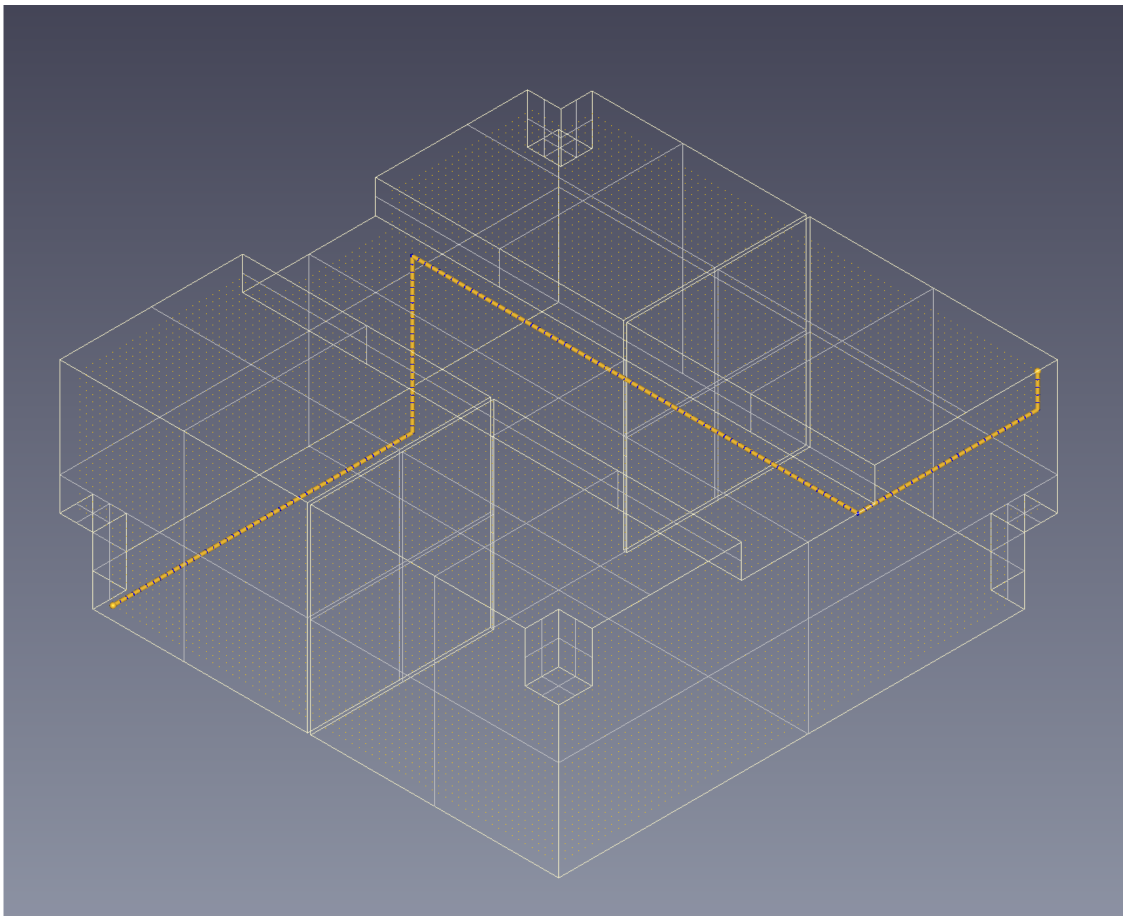


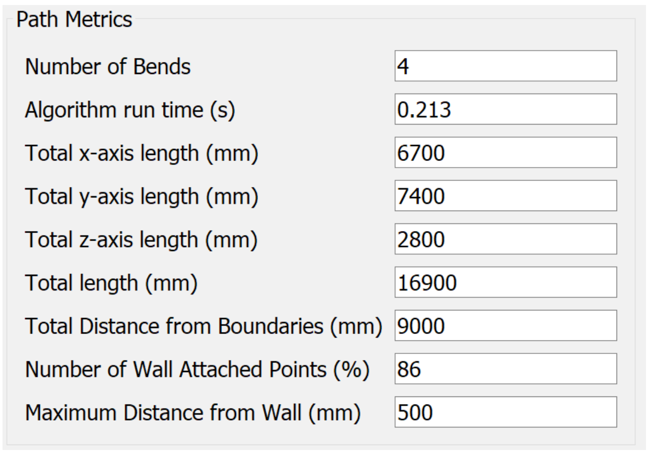


Path Metrics

Number of Bends	6
Algorithm run time (s)	0.054
Total x-axis length (mm)	6700
Total y-axis length (mm)	12800
Total z-axis length (mm)	2800
Total length (mm)	22300
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 40: Test run #18 illustration and metrics

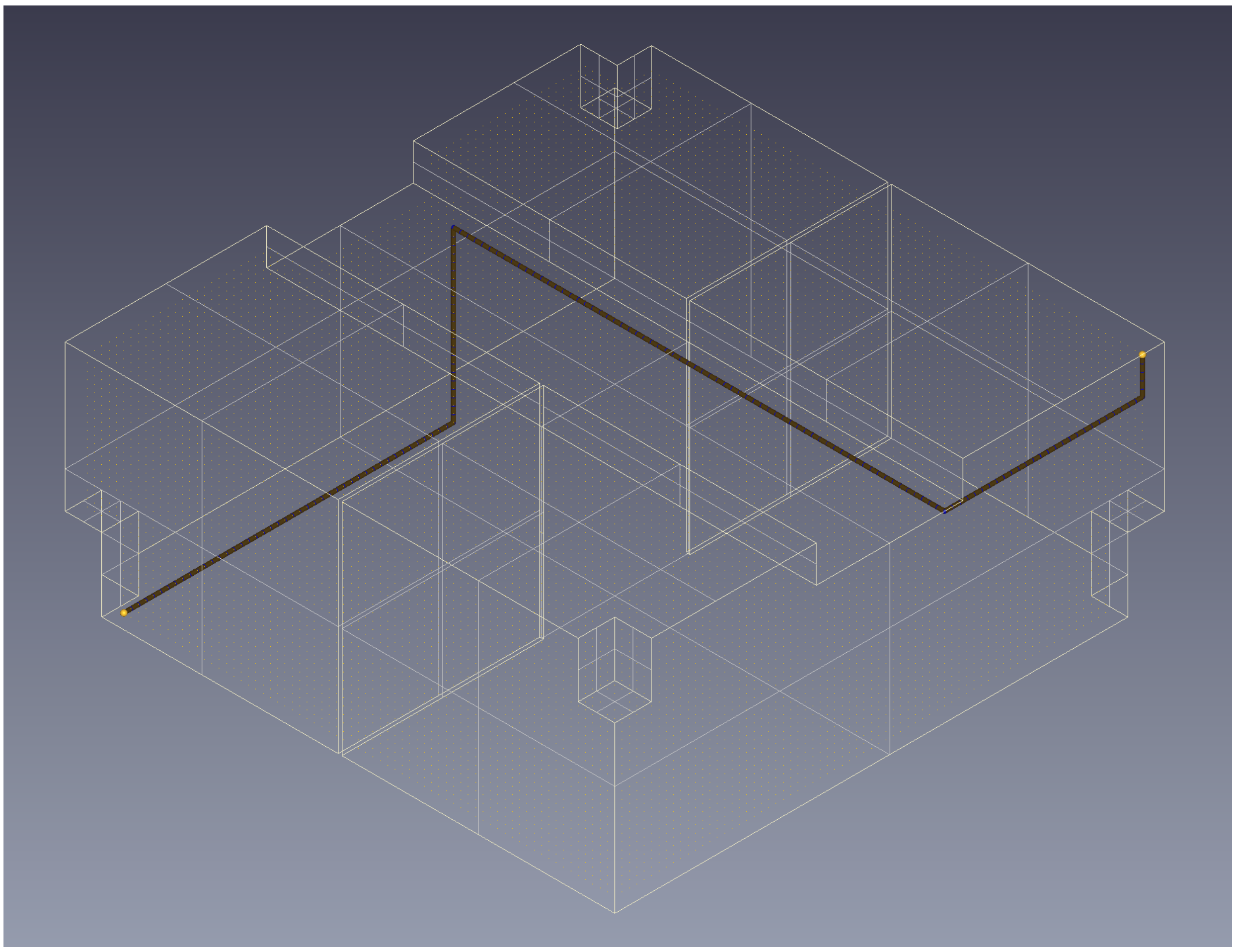


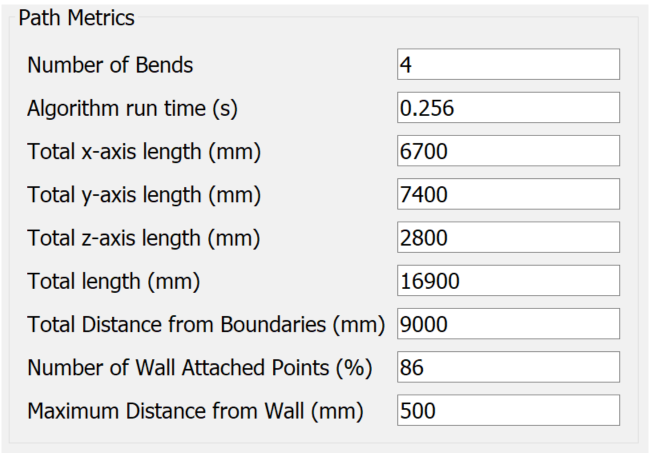


Path Metrics

Number of Bends	4
Algorithm run time (s)	0.213
Total x-axis length (mm)	6700
Total y-axis length (mm)	7400
Total z-axis length (mm)	2800
Total length (mm)	16900
Total Distance from Boundaries (mm)	9000
Number of Wall Attached Points (%)	86
Maximum Distance from Wall (mm)	500

Figure 41: Test run #19 illustration and metrics

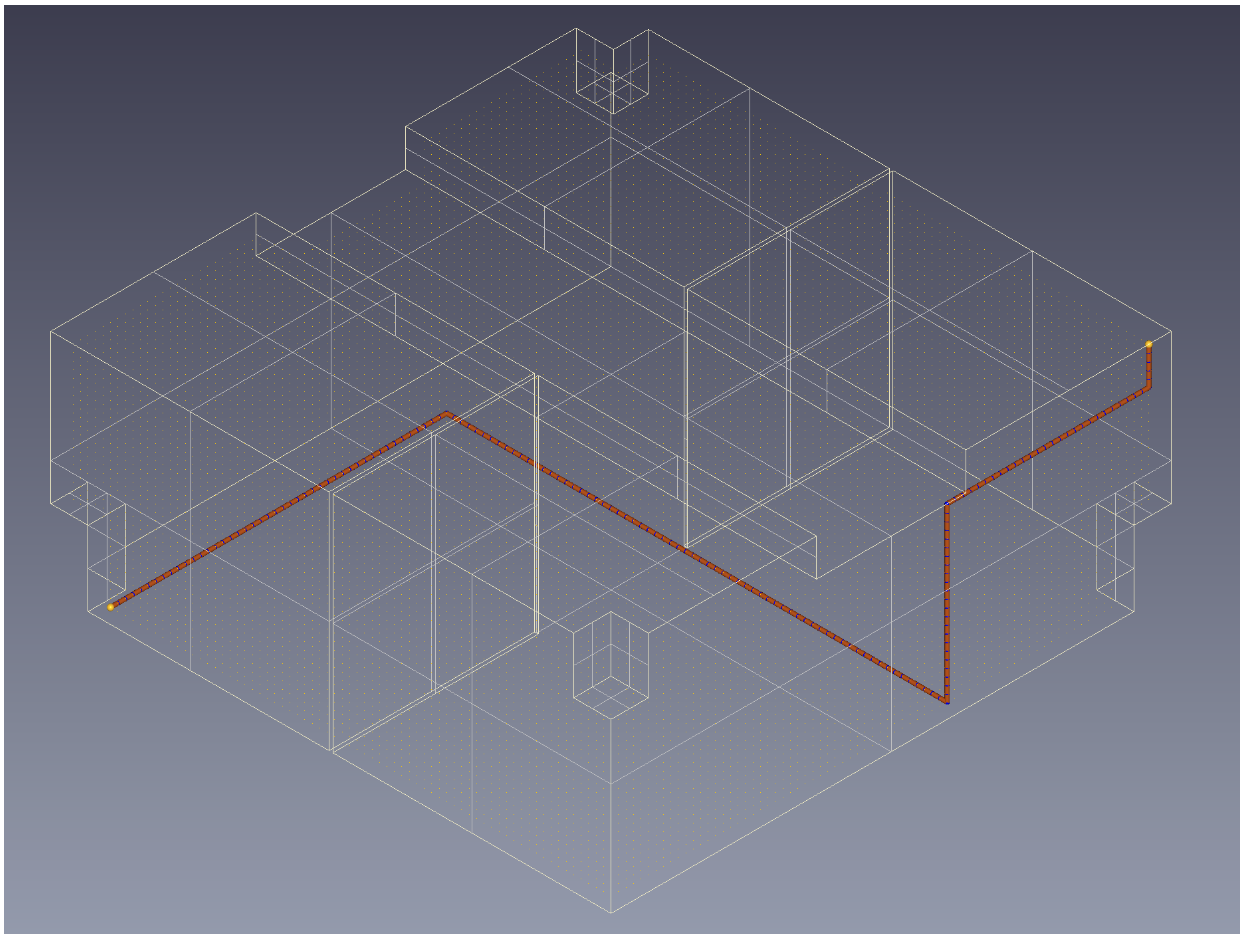


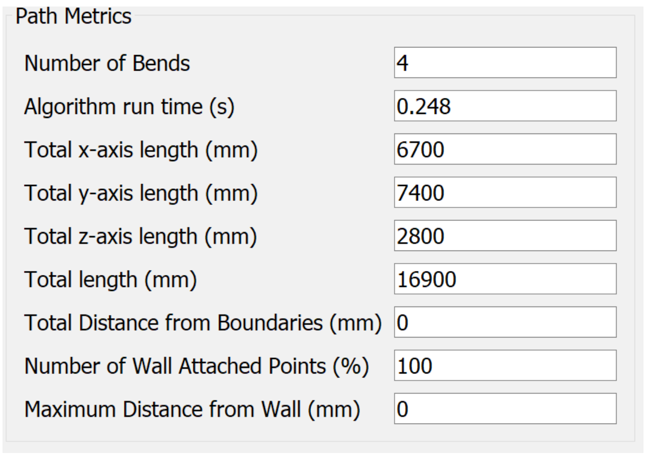


Path Metrics

Number of Bends	4
Algorithm run time (s)	0.256
Total x-axis length (mm)	6700
Total y-axis length (mm)	7400
Total z-axis length (mm)	2800
Total length (mm)	16900
Total Distance from Boundaries (mm)	9000
Number of Wall Attached Points (%)	86
Maximum Distance from Wall (mm)	500

Figure 42: Test run #20 illustration and metrics

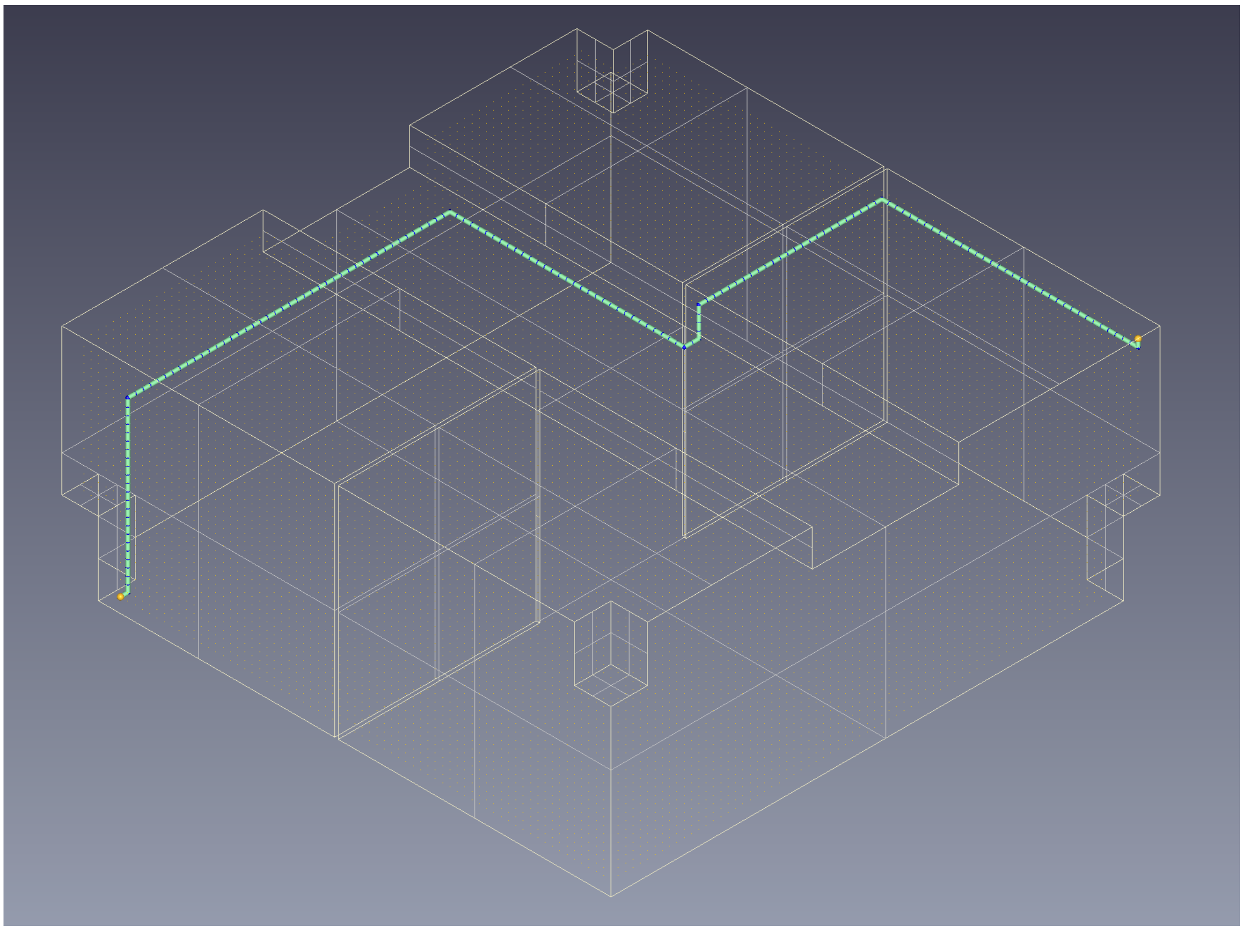


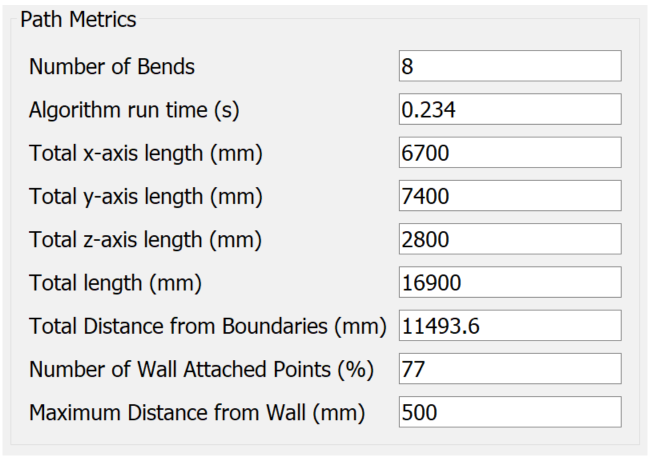


Path Metrics

Number of Bends	4
Algorithm run time (s)	0.248
Total x-axis length (mm)	6700
Total y-axis length (mm)	7400
Total z-axis length (mm)	2800
Total length (mm)	16900
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 43: Test run #21 illustration and metrics





Path Metrics

Number of Bends	8
Algorithm run time (s)	0.234
Total x-axis length (mm)	6700
Total y-axis length (mm)	7400
Total z-axis length (mm)	2800
Total length (mm)	16900
Total Distance from Boundaries (mm)	11493.6
Number of Wall Attached Points (%)	77
Maximum Distance from Wall (mm)	500

Figure 44: Test run #22 illustration and metrics

4.3.2.3. Results assessment

Considering the previous test runs, the following conclusions can be drawn:

(1) We have already realized the importance of a good heuristic as far as the optimality of the A* path solutions is concerned. Keeping this in mind, a closer look at the results depicted in Figure 40 and 41, makes the aforementioned statement all the more vital. In Figure 40, we can see that the A* becomes trapped by its heuristic resulting in an obvious less than optimal pipe-routing solution. This happens because the algorithm does not have foreknowledge of the existence of obstacles in the layout search space, and as a result its heuristic guides the solution towards the most heuristically promising path. Great care should be taken when deciding which heuristic function will be utilized in each scenario.

(2) By comparing the results of Figures 41 and 42, we can see that the resulting paths from the implementation of Dijkstra in a split graph, can be an exact match with the paths resulting from the A*. The only thing that differentiates the two solutions is the run time.

(3) Last but not least, by considering the results from Figures 41 – 43, we can see that the proposed weight allocation methods play an active role in influencing the form and the whereabouts of the resulting paths.

4.3.3. Model 3: Compound Space

4.3.3.1. Test case presentation

Three sets of test runs were conducted for the current model. The first set, consists of three test runs, both of which are using Dijkstra in a standard, non split, weighted graph. None of the two weight allocation methods are used for these test runs. The aim of these test runs is to show that Dijkstra always finds the shortest length path to the goal, as well as efficiently avoiding interference with any obstacle, regardless of the search space complexity. Table 8 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #23*	*Test run #24*	*Test run#25*
Voxel Size (mm)	200	200	200
Graph Type	Standard	Standard	Standard
Split graph	False	False	False
Start point (mm)	(200, 200, 2600)	(1800, 16600, 4400)	(2600, 8400, 1200)
End point (mm)	(10200, 5800, 5200)	(10200, 4600, 3200)	(11800, 3400, 1200)
Direction (S-E)	Not applicable	Not applicable	Not applicable
Position-level method	Not utilized	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized	Not utilized
Algorithm	Dijkstra	Dijkstra	Dijkstra
Heuristic	-	-	-
Dir. “straight” weight	1	1	1
Dir. “bend” weight	20	20	20

Table 8: Model 3: First set test run parameters

The second set, consists of five test runs. This time Dijkstra is being tested in a split standard graph without the use of a weight allocation method. In addition, the effect of the directional weights is investigated. Table 9 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #26 & 27*	*Test run #28*	*Test run #29*	*Test run #30*
Voxel Size (mm)	200	200	200	200
Graph Type	Standard	Standard	Standard	Standard
Split graph	True	True	True	True
Start point (mm)	(200, 200, 200)	(200, 200, 200)	(200, 200, 200)	(200, 200, 200)
End point (mm)	(13800, 9800, 6600)	(13800, 9800, 6600)	(13800, 9800, 6600)	(13800, 9800, 6600)
Direction (S-E)	1-2	1-2	1-2	1-2
Position-level method	Not utilized	Not utilized	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized	Not utilized	Not utilized
Algorithm	Dijkstra	Dijkstra	Dijkstra	Dijkstra
Heuristic	-	-	-	-
Dir. “straight” weight	1 , 10	20	30	200
Dir. “bend” weight	20 , 200	1	1	10

Table 9: Model 3: Second set test run parameters

In the third and final set, four test runs are performed. Dijkstra is again utilized as the path-finding algorithm for solving the pipe-routing problem. However, the search takes place within a diagonal graph structure, both non-split and split. As far as the weight allocation methods are concerned, the position-level method is used. At this point, the concept of the diagonal directional edge weights is expanded and the effects of these values are looked into. Furthermore, the standard weight of diagonal edge is multiplied by a factor, the value of which affects the results directly. Table 10 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #31*	*Test run #32*	*Test run #33*	*Test run #34*
Voxel Size (mm)	300	300	300	300
Graph Type	Diagonal	Diagonal	Diagonal	Diagonal
Split graph	False	False	True	True
Start point (mm)	(300, 300, 0)	(300, 300, 0)	(300, 300, 0)	(300, 300, 0)
End point (mm)	(13500, 12000, 6600)	(13500, 12000, 6600)	(13500, 12000, 6600)	(13500, 12000, 6600)
Direction (S-E)	Not applicable	Not applicable	1-7	1-9
Position-level method	Not utilized	✓	Not utilized	✓
Function	-	Exponential	-	Quadratic
Multiplier	-	100	-	100
Plane-distance method	Not utilized	✓	Not utilized	✓
Function	-	Decreasing Exponential	-	Decreasing Linear
Multiplier	-	100	-	100
Algorithm	Dijkstra	Dijkstra	Dijkstra	Dijkstra
Heuristic	-	-	-	-
Dir.“straight” weight	-	-	1	1
Dir.“45 bend” weight	-	-	30	20
Dir. “bend” weight	-	-	20	20
Dir.“135 bend” weight	-	-	100	20
“E.45 bend” weight	-	-	50	20
“E.135 bend” weight	-	-	200	20
Diagonal weight mult/er	1	2	1.5	1
E. diagonal weight mult/er	1	1.5	1.5	1.5

Table 10: Model 3: Third set test run parameters

4.3.3.2. Results illustartion

Based on on the data presented in Tables 8 to 10, the results of the various test runs are presented in the figure illustrations below. However, for the shake of clarity, the current model on which the test runs were performed, is also presented from different angles.

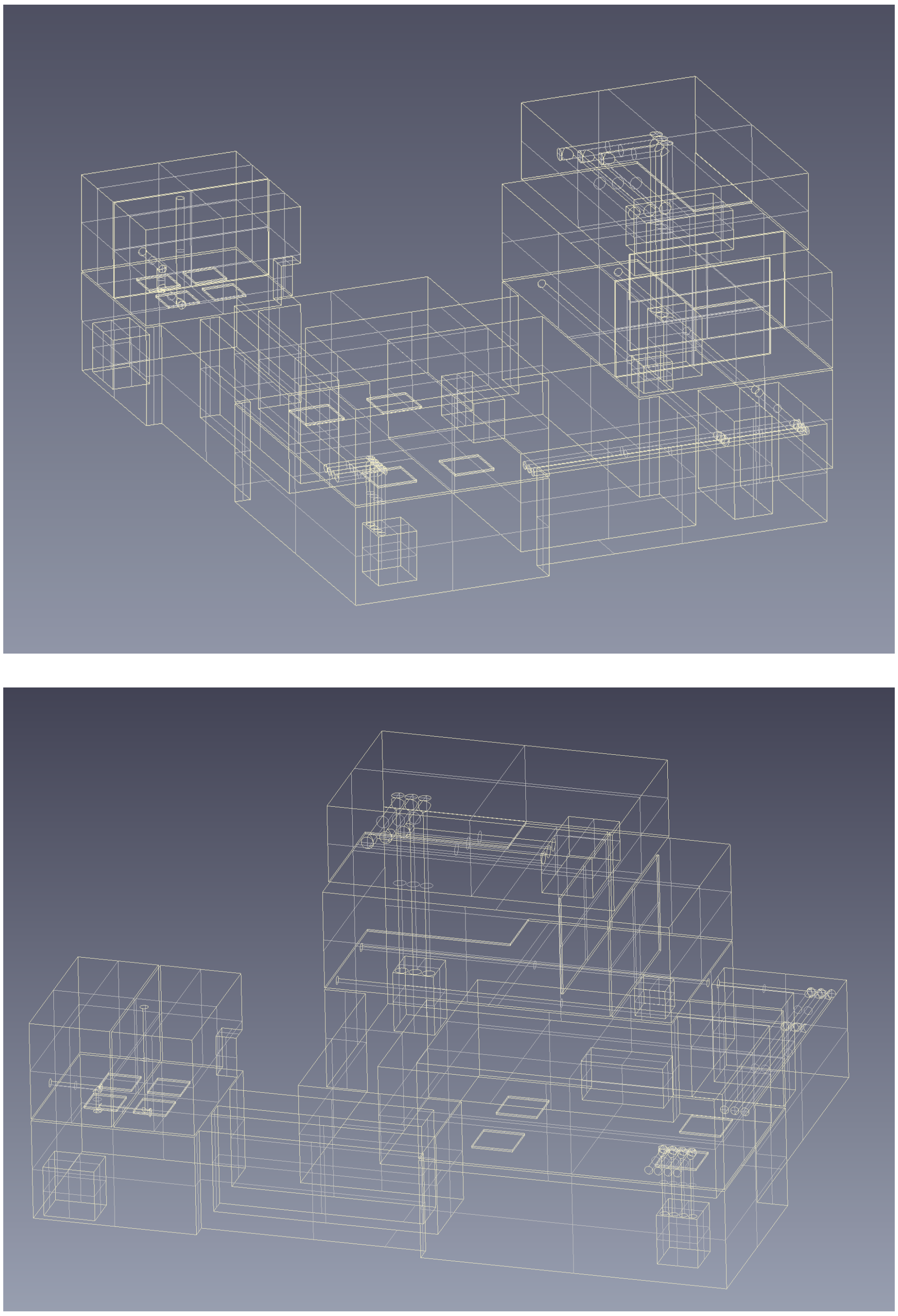
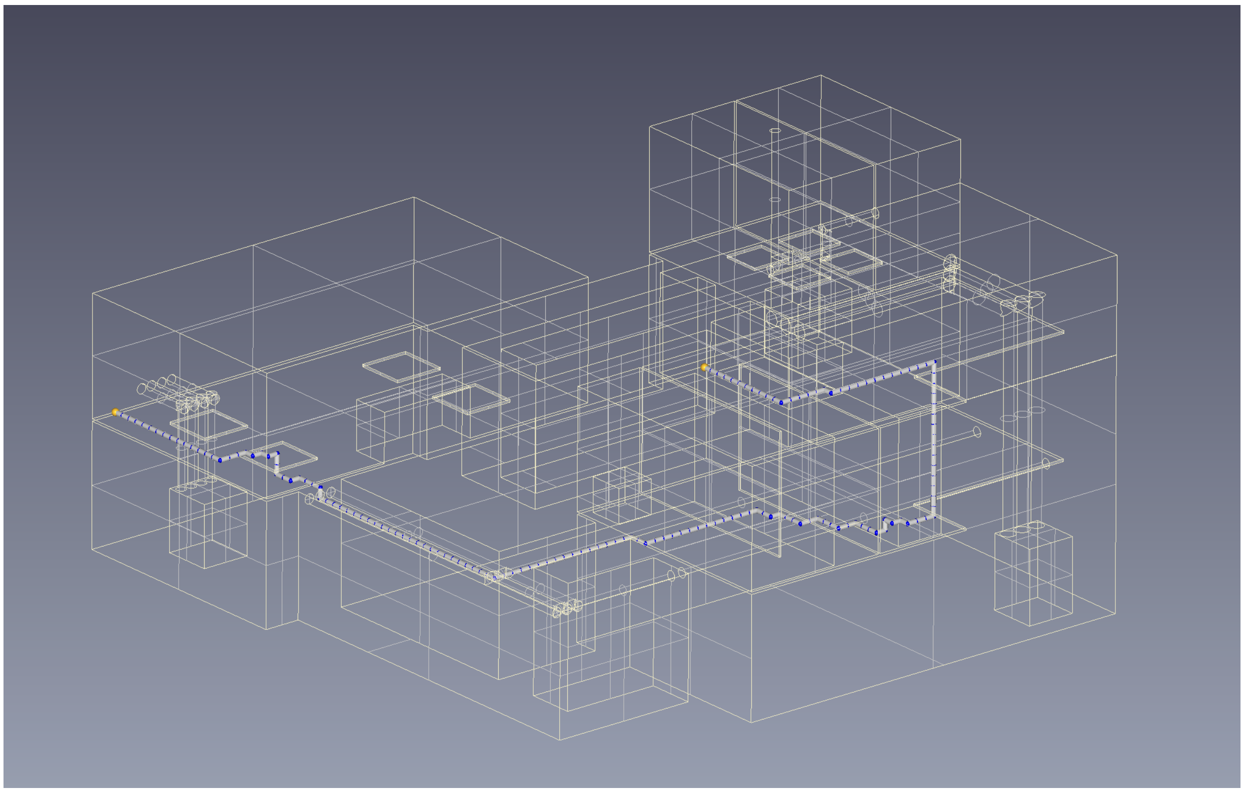
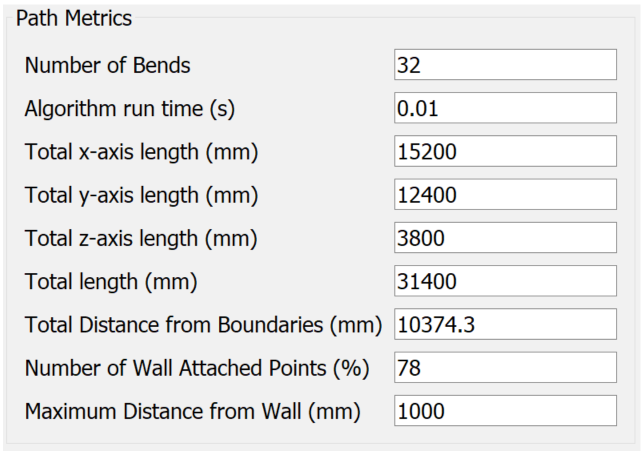


Figure 45: Model 3: Explanatory layout space illustration

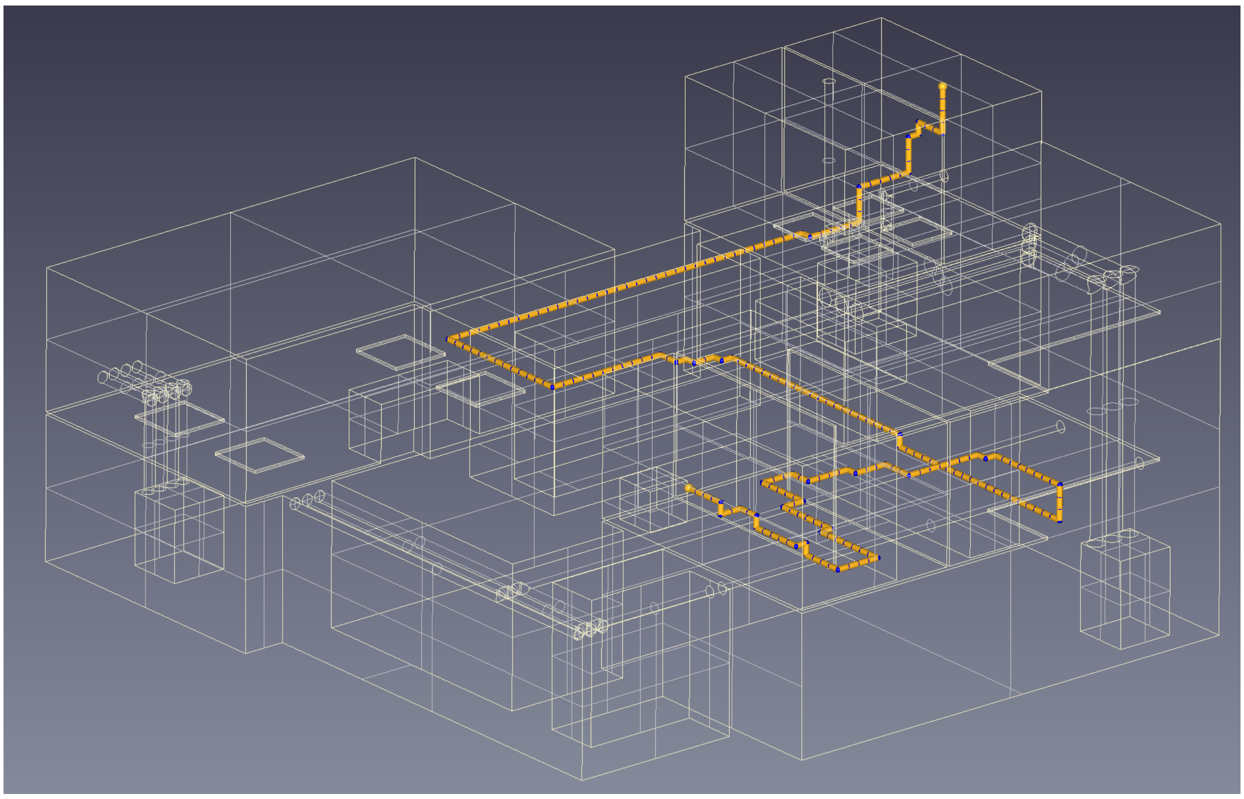


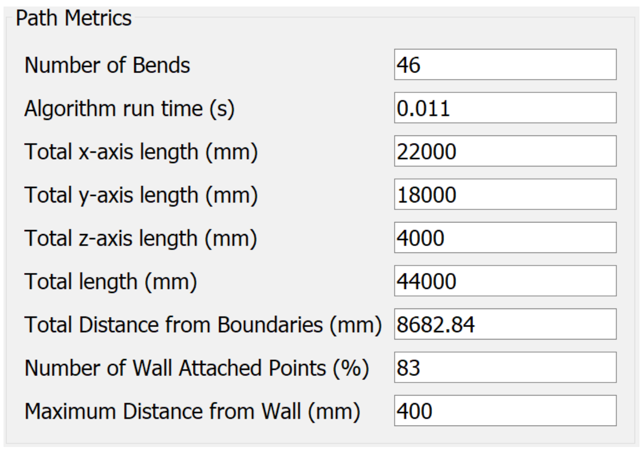


Path Metrics

Number of Bends	32
Algorithm run time (s)	0.01
Total x-axis length (mm)	15200
Total y-axis length (mm)	12400
Total z-axis length (mm)	3800
Total length (mm)	31400
Total Distance from Boundaries (mm)	10374.3
Number of Wall Attached Points (%)	78
Maximum Distance from Wall (mm)	1000

Figure 46: Test run #23 illustration and metrics

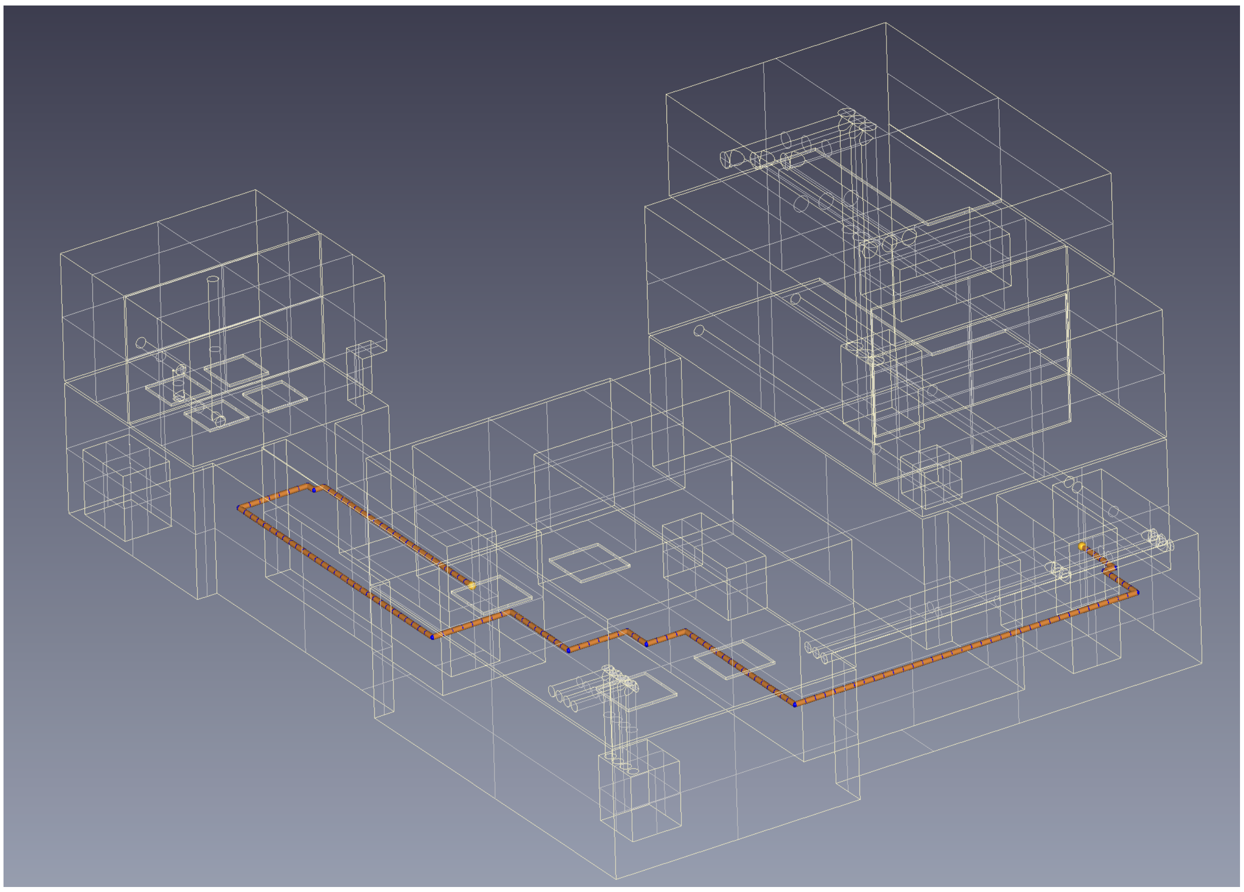


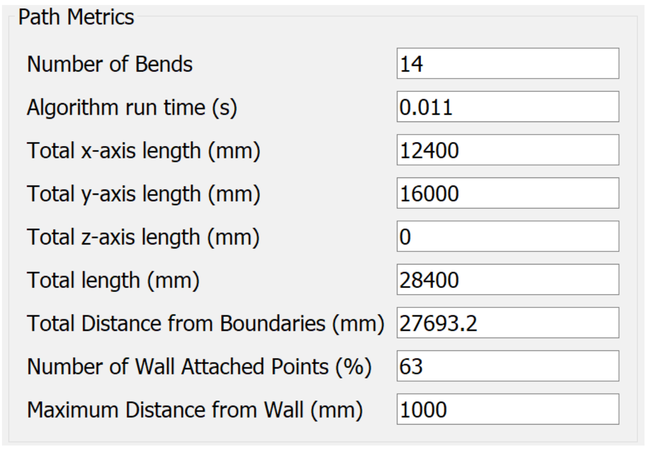


Path Metrics

Number of Bends	46
Algorithm run time (s)	0.011
Total x-axis length (mm)	22000
Total y-axis length (mm)	18000
Total z-axis length (mm)	4000
Total length (mm)	44000
Total Distance from Boundaries (mm)	8682.84
Number of Wall Attached Points (%)	83
Maximum Distance from Wall (mm)	400

Figure 47: Test run #24 illustration and metrics

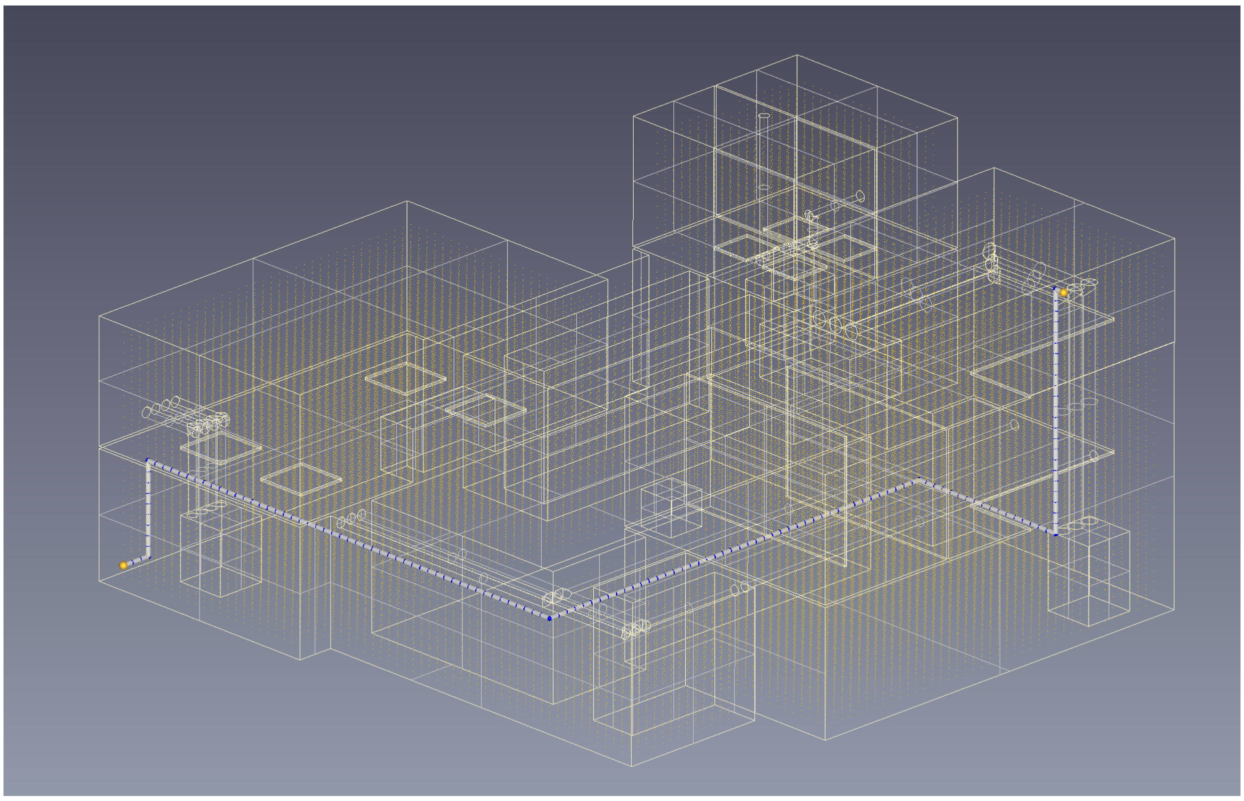


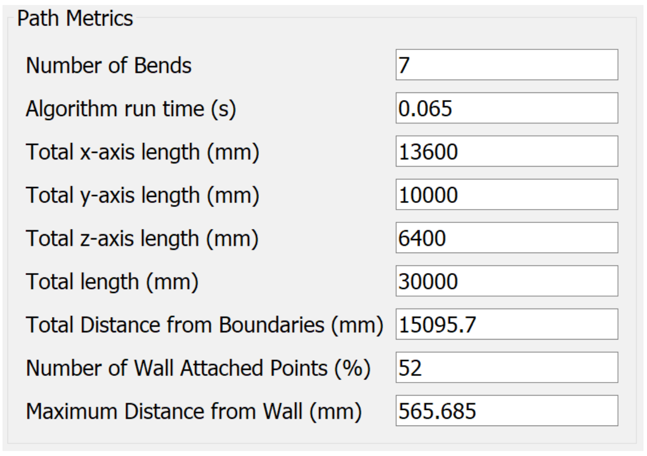


Path Metrics

Number of Bends	14
Algorithm run time (s)	0.011
Total x-axis length (mm)	12400
Total y-axis length (mm)	16000
Total z-axis length (mm)	0
Total length (mm)	28400
Total Distance from Boundaries (mm)	27693.2
Number of Wall Attached Points (%)	63
Maximum Distance from Wall (mm)	1000

Figure 48: Test run #25 illustration and metrics

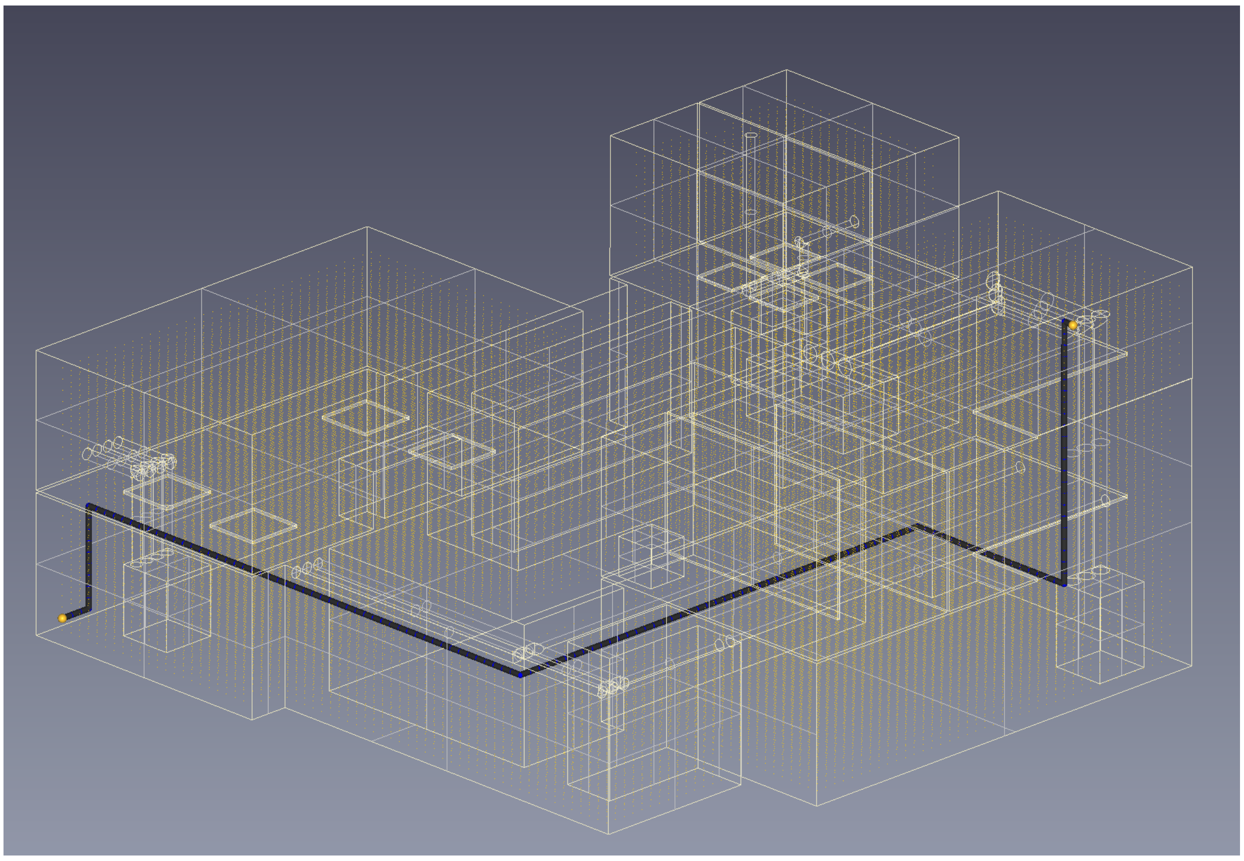


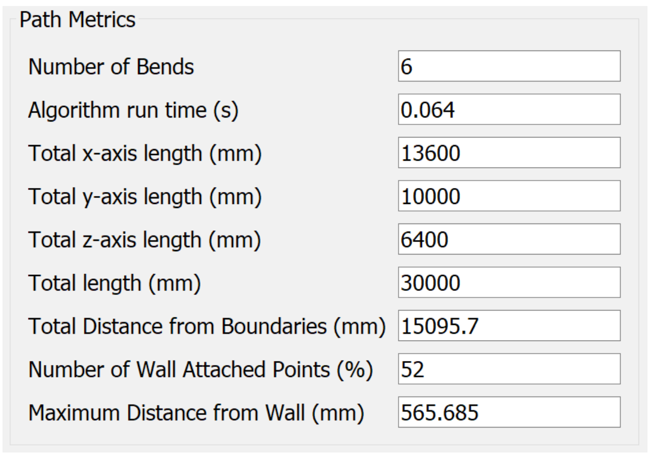


Path Metrics

Number of Bends	7
Algorithm run time (s)	0.065
Total x-axis length (mm)	13600
Total y-axis length (mm)	10000
Total z-axis length (mm)	6400
Total length (mm)	30000
Total Distance from Boundaries (mm)	15095.7
Number of Wall Attached Points (%)	52
Maximum Distance from Wall (mm)	565.685

Figure 49: Test run #26 & #27 illustration and metrics

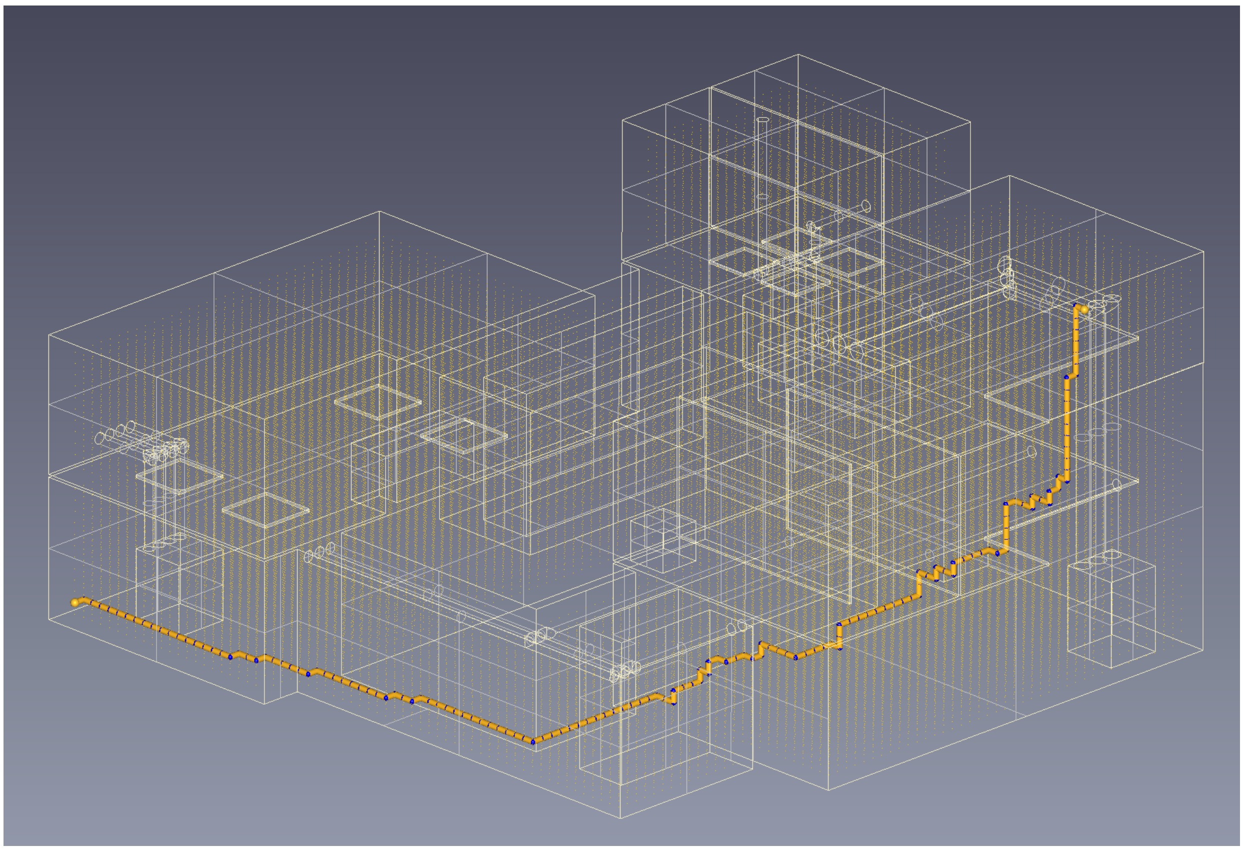


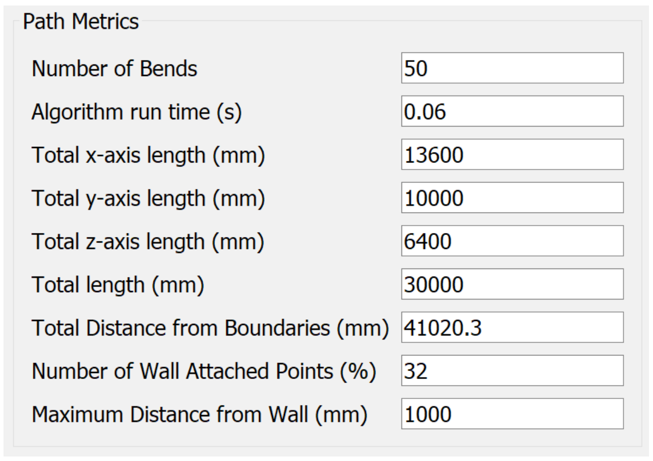


Path Metrics

Number of Bends	6
Algorithm run time (s)	0.064
Total x-axis length (mm)	13600
Total y-axis length (mm)	10000
Total z-axis length (mm)	6400
Total length (mm)	30000
Total Distance from Boundaries (mm)	15095.7
Number of Wall Attached Points (%)	52
Maximum Distance from Wall (mm)	565.685

Figure 50: Test run #28 illustration and metrics

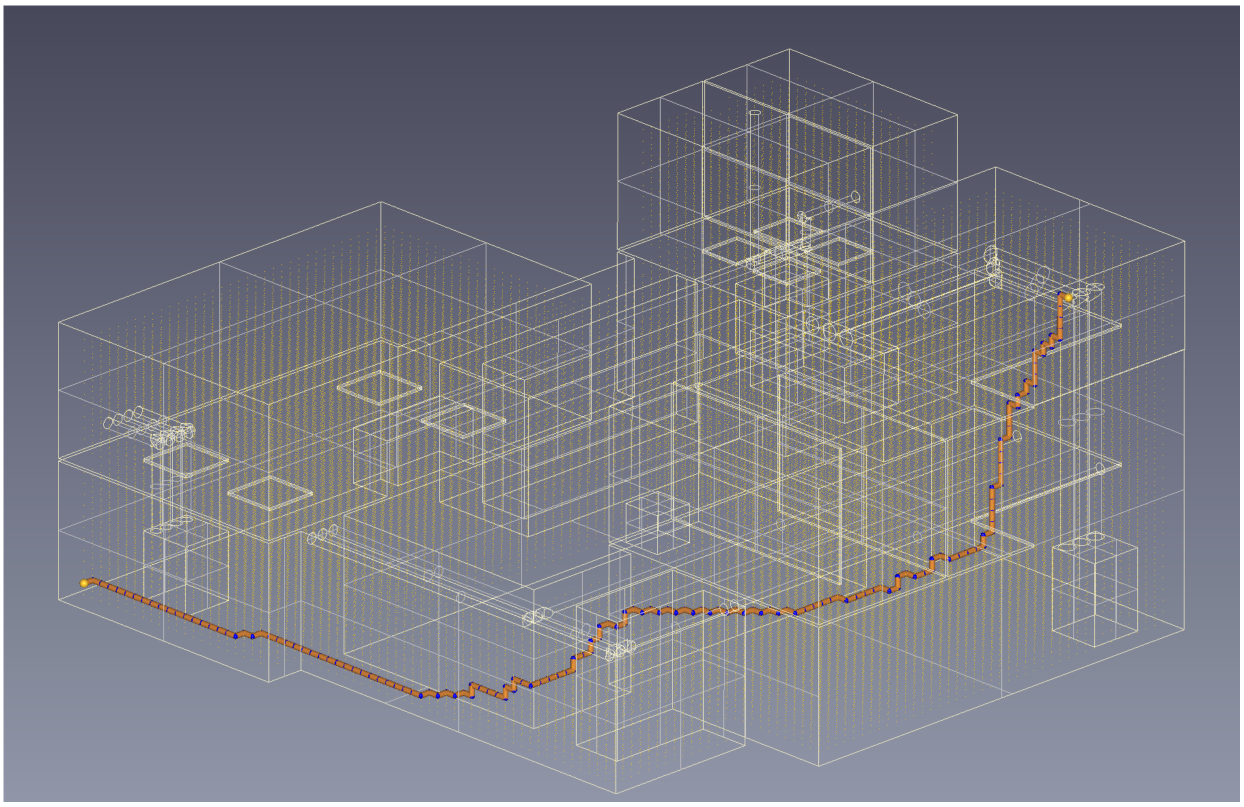


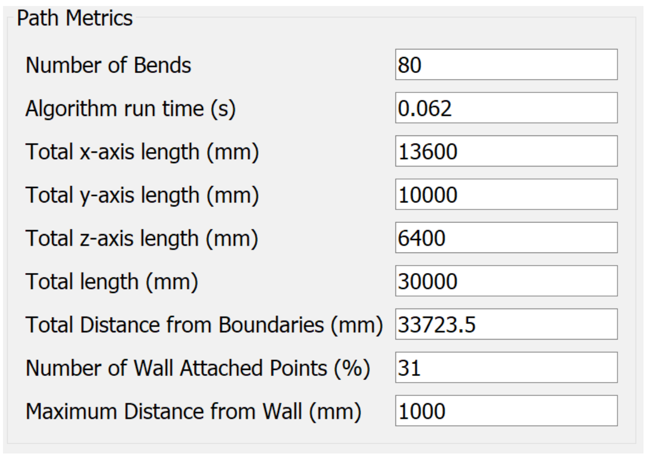


Path Metrics

Number of Bends	50
Algorithm run time (s)	0.06
Total x-axis length (mm)	13600
Total y-axis length (mm)	10000
Total z-axis length (mm)	6400
Total length (mm)	30000
Total Distance from Boundaries (mm)	41020.3
Number of Wall Attached Points (%)	32
Maximum Distance from Wall (mm)	1000

Figure 51: Test run #29 illustration and metrics

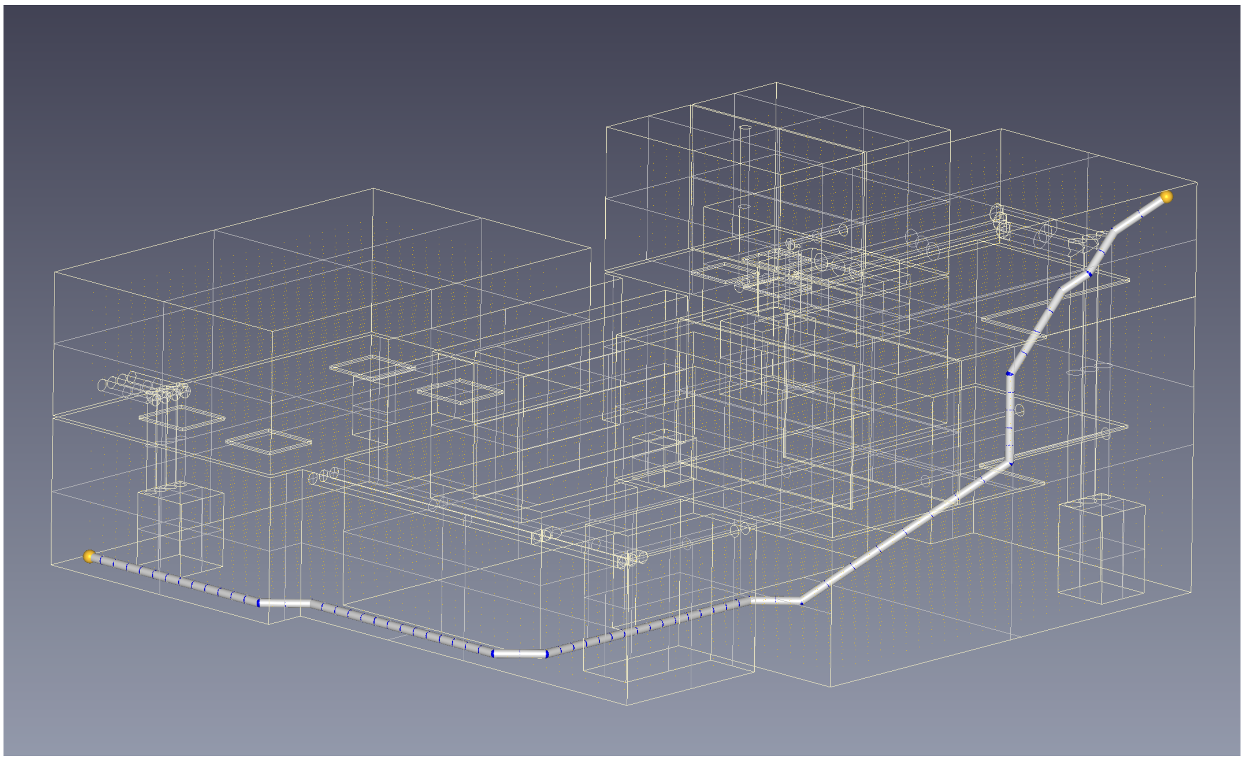


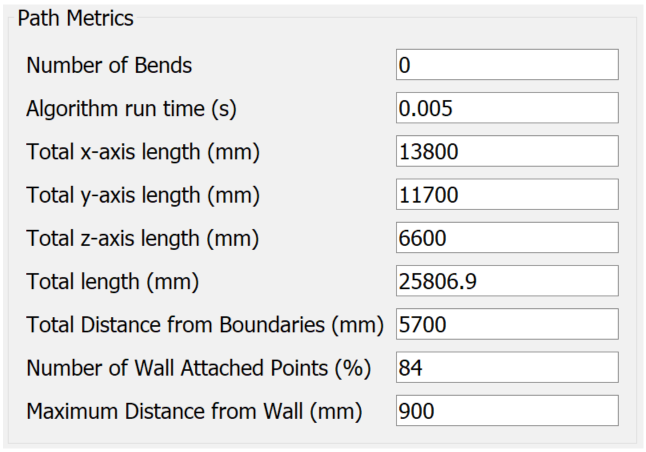


Path Metrics

Number of Bends	80
Algorithm run time (s)	0.062
Total x-axis length (mm)	13600
Total y-axis length (mm)	10000
Total z-axis length (mm)	6400
Total length (mm)	30000
Total Distance from Boundaries (mm)	33723.5
Number of Wall Attached Points (%)	31
Maximum Distance from Wall (mm)	1000

Figure 52: Test run #30 illustration and metrics

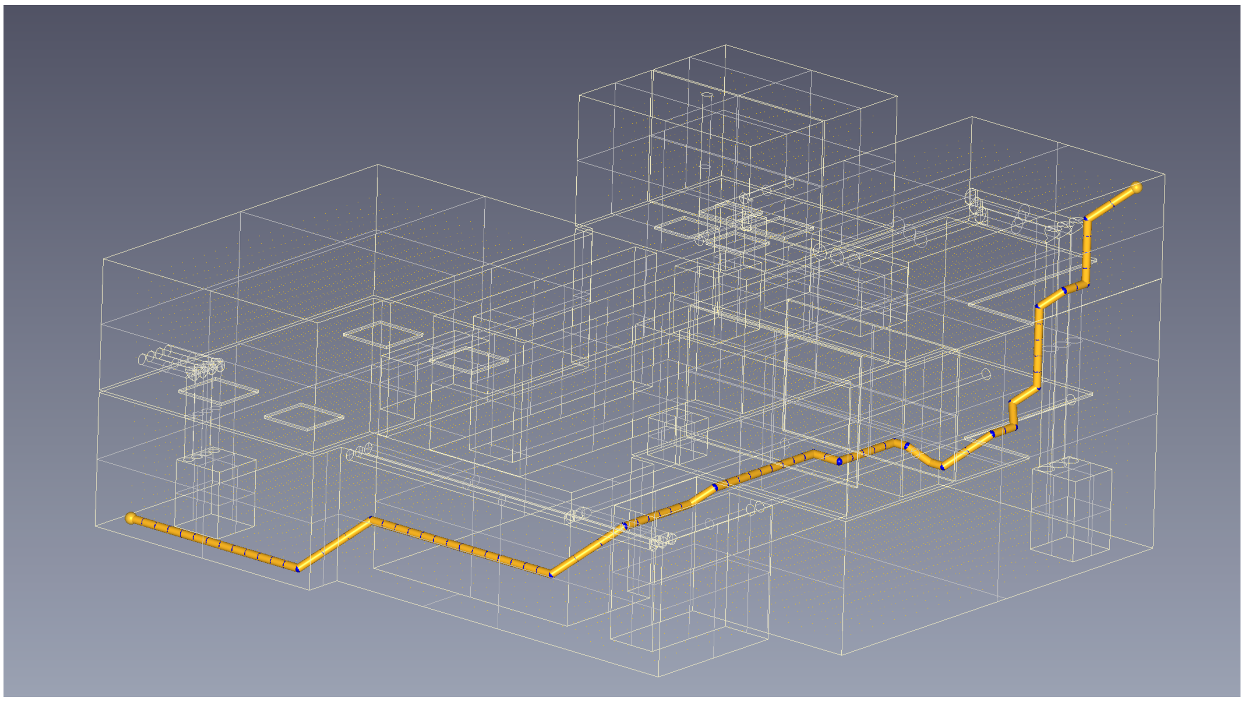


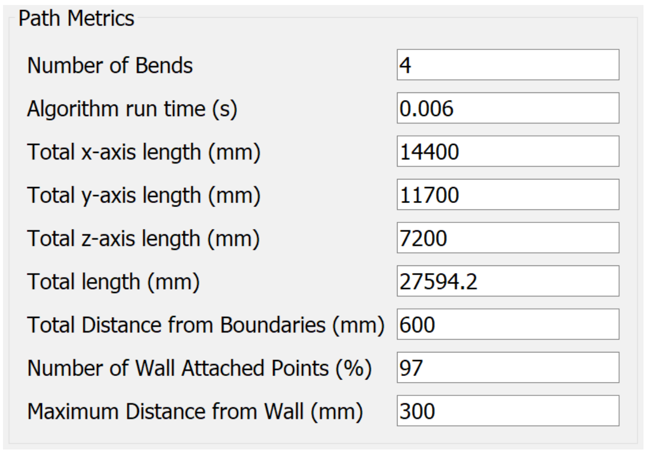


Path Metrics

Number of Bends	0
Algorithm run time (s)	0.005
Total x-axis length (mm)	13800
Total y-axis length (mm)	11700
Total z-axis length (mm)	6600
Total length (mm)	25806.9
Total Distance from Boundaries (mm)	5700
Number of Wall Attached Points (%)	84
Maximum Distance from Wall (mm)	900

Figure 53: Test run #31 illustration and metrics

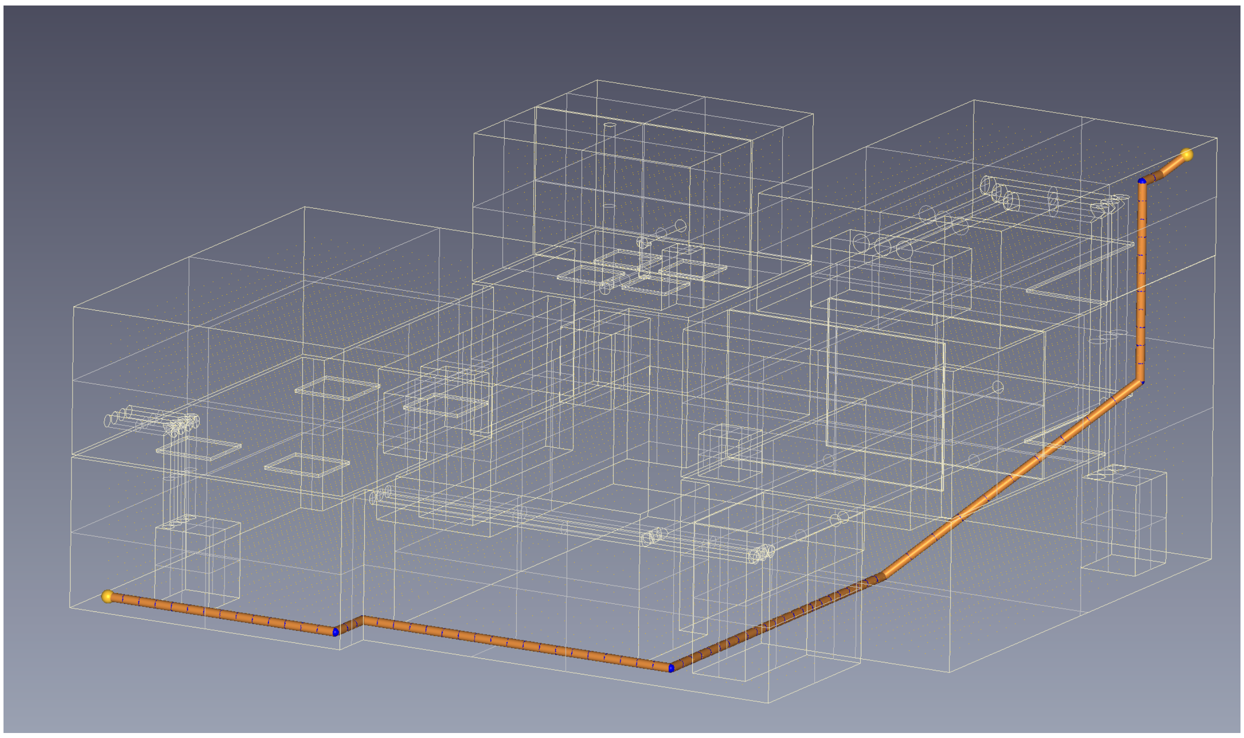


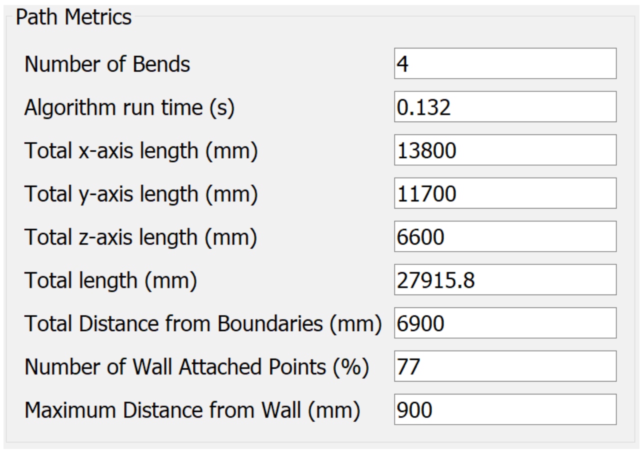


Path Metrics

Number of Bends	4
Algorithm run time (s)	0.006
Total x-axis length (mm)	14400
Total y-axis length (mm)	11700
Total z-axis length (mm)	7200
Total length (mm)	27594.2
Total Distance from Boundaries (mm)	600
Number of Wall Attached Points (%)	97
Maximum Distance from Wall (mm)	300

Figure 54: Test run #32 illustration and metrics

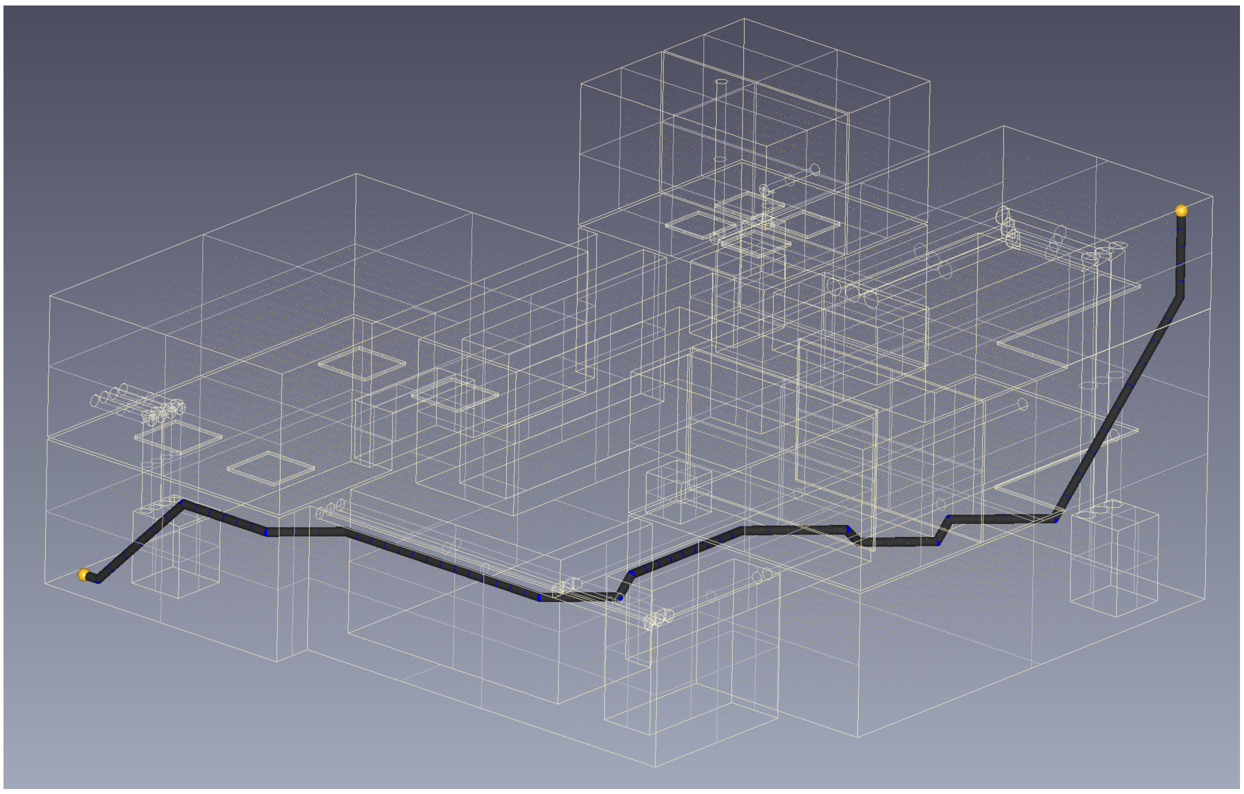


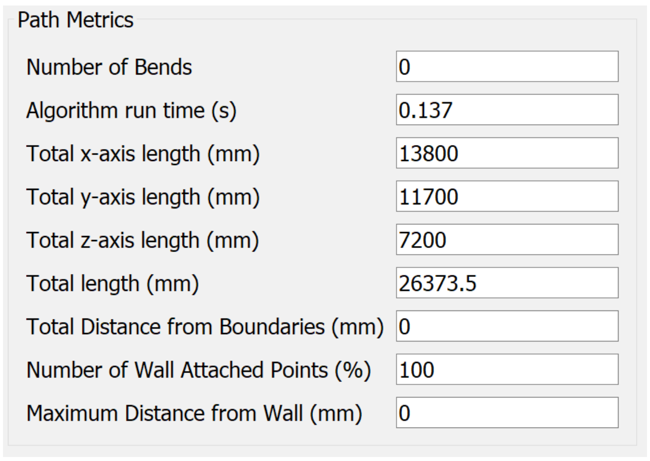


Path Metrics

Number of Bends	4
Algorithm run time (s)	0.132
Total x-axis length (mm)	13800
Total y-axis length (mm)	11700
Total z-axis length (mm)	6600
Total length (mm)	27915.8
Total Distance from Boundaries (mm)	6900
Number of Wall Attached Points (%)	77
Maximum Distance from Wall (mm)	900

Figure 55: Test run #33 illustration and metrics





Path Metrics

Number of Bends	0
Algorithm run time (s)	0.137
Total x-axis length (mm)	13800
Total y-axis length (mm)	11700
Total z-axis length (mm)	7200
Total length (mm)	26373.5
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 56: Test run #34 illustration and metrics

4.3.3.3. Results assessment

Considering the previous test runs, the following conclusions can be drawn:

(1) From Figures 46 – 48, we can see that Dijkstra is always for the shortest path, if one exists of course. Furthermore, we can see that both the proposed modeling and algorithmic approach to the problem of pipe-routing counters the problem of collision with complete success, as it manages to navigate through complex spaces. Finding openings to pass through (see Figure 46), navigating through alternating walls (see Figure 47) and evading dead ends (see Figure 48) are dealt with successfully in each run.

(2) While presenting the weight allocation methods in Section 2.4.3., we saw that the values of the directional edge weights play an active role in the resulting path. While considering the results presented in Figures 49 – 52, we can draw the following conclusions. Firstly, based on Figures 49 and 50, we see that the value sets of 1 – 20 and 10 – 200 for the straight and 90^o directional weights give the same results, leading us to believe that the actual value of these two weights does not really matter, whereas their ratio value does. However, in Figure 49 we shift the value set to 20 – 1, which leads again to the same result. These two observations, leads us to the conclusion that the ability of the directional edges to affect the resulting paths in a way that leads to good solutions, is closely connected with the ratio of the two values in combination with the ratio of these values and the standard edge weights. This conclusion is verified by the result in Figure 51, where a value set of 30 – 1 (straight – 90^o bend) leads to a more bend ridden path. In Figure 52, setting the value set to 200 – 10, only raises the number of bends from 50 in Figure 51, to 80. This, also leads us to the conclusion that misuse of these directional weight values can ruin the optimal path-finding capability of the split graph structure.

(3) Last but not least, by considering the results from Figures 53 – 56, we can see that the diagonal graph structure produces some really interesting results. In general, when routing pipes in a diagonal graph, the resulting optimal path tends to follow the diagonal edges more, as they lead to shortest length path solution candidates. However, extensive use of these diagonal edges is not feasible. For this reason, the diagonal weight multipliers which we introduced, can go some way towards leading to more orthogonal paths, see Figures 53 and 54. Furthermore, by taking a look at Figures 55 and 56, we can see that the use of the extended diagonal weight penalty allocation method, as introduced in Section 2.4.3.2.2. in Figures 23, 24 can lead to paths were much less directional changes occur within a path. However, even the use of a uniform bend penalty, regardless of the angle, as described in Section 2.4.3.2.2. in Figure 24, leads to good alternative pipe-routing solutions.

4.3.4. Model 4: Complex compound space

4.3.4.1. Test case presentation

Two sets of test runs were conducted for the current model. As far as the first set is concerned, two test runs are considered. Both Dijkstra and A* are used respectively, in a standard, non split weighted graph, without the utilization of a weight allocation method. The aim of these test runs is to show the run time superiority of the A* algorithm over Dijkstra, especially in a complex layout space. Table 11 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #35*	*Test run #36*
Voxel Size (mm)	100	100
Graph Type	Standard	Standard
Split graph	False	False
Start point (mm)	(600, 4800, 3400)	(600, 4800, 3400)
End point (mm)	(17300, 7300, 2800)	(17300, 7300, 2800)
Direction (S-E)	Not applicable	Not applicable
Position-level method	Not utilized	Not utilized
Plane-distance method	Not utilized	Not utilized
Algorithm	Dijkstra	A*
Heuristic	-	Euclidean
Dir. “straight” weight	1	1
Dir. “bend” weight	20	20

Table 11: Model 4: First set test run parameters

The second set, consists of t test runs. Dijkstra's algorithm is used, in both standard split and non-split graphs. The aim of this set of test runs is to test the effectiveness of the weight allocation methods proposed, as far as their ability to force the solution through certain areas is concerned. Table 12 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #37*	*Test run #38*	*Test run #39*	*Test run #40*
Voxel Size (mm)	100	100	100	100
Graph Type	Standard	Standard	Standard	Standard
Split graph	False	True	True	True
Start point (mm)	(7300, 3700, 800)	(7300, 3700, 800)	(7300, 3700, 800)	(7300, 3700, 800)
End point (mm)	(17300, 7300, 2800)	(17300, 7300, 2800)	(17300, 7300, 2800)	(17300, 7300, 2800)
Direction (S-E)	Not applicable	3-2	3-2	3-2
Position-level method	Not utilized	✓	Not utilized	Not utilized
Function	-	Exponential	-	-
Multiplier	-	100	-	-
Plane-distance method	Not utilized	Not utilized	✓	✓
Function	-	-	Decreasing Linear	Increasing Linear
Multiplier	-	-	100	100
Algorithm	Dijkstra	Dijkstra	Dijkstra	Dijkstra
Heuristic	-	-	-	-
Dir. “straight” weight	1	1	1	1
Dir. “bend” weight	20	20	20	20

Table 12: Model 4: Second set test run parameters

4.3.4.2. Results illustration

Based on on the data presented in Tables 11 and 12, the results of the various test runs are presented in the figure illustrations below. However, for the shake of clarity, the current model on which the test runs were performed, is also presented from different angles.

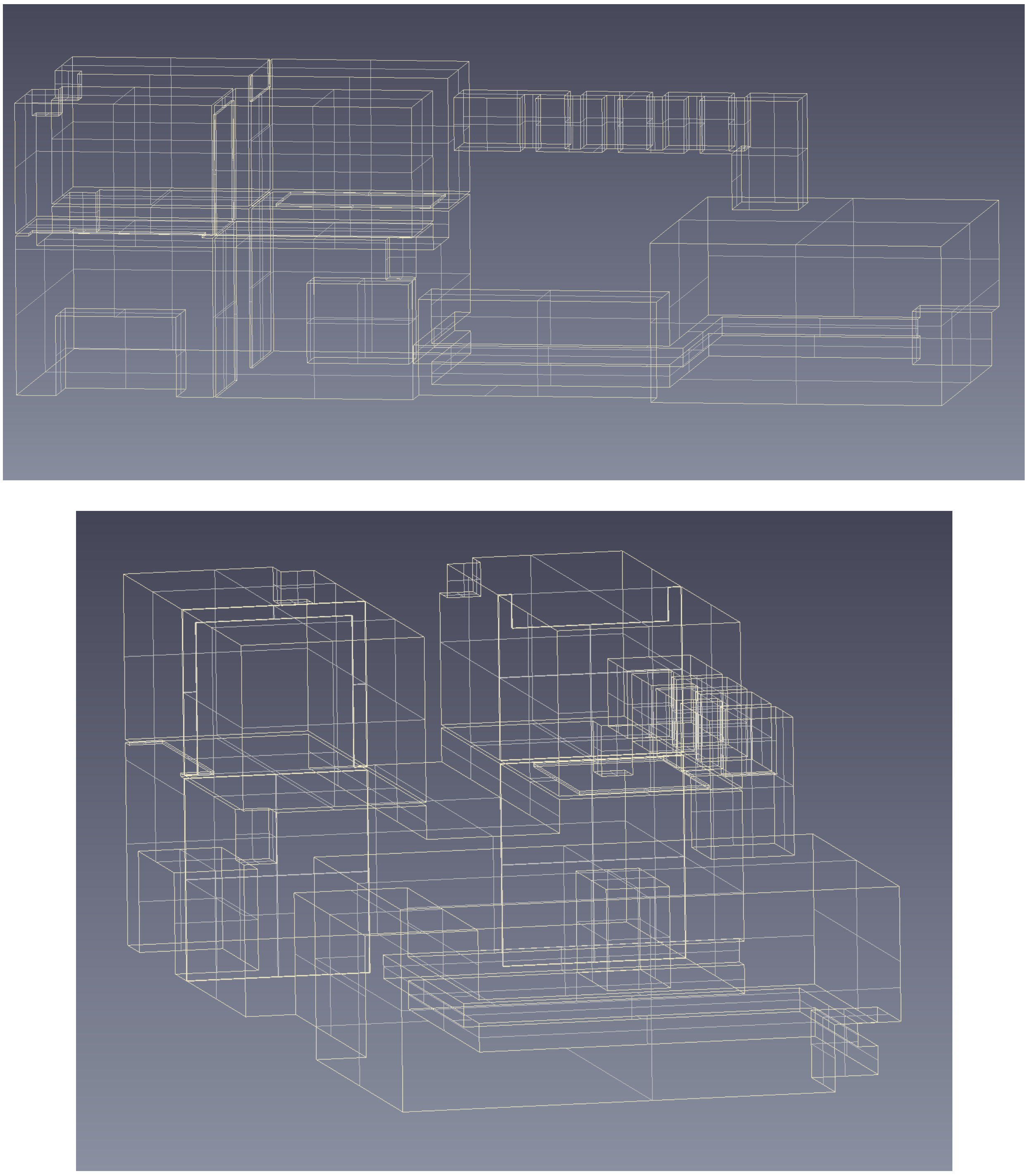
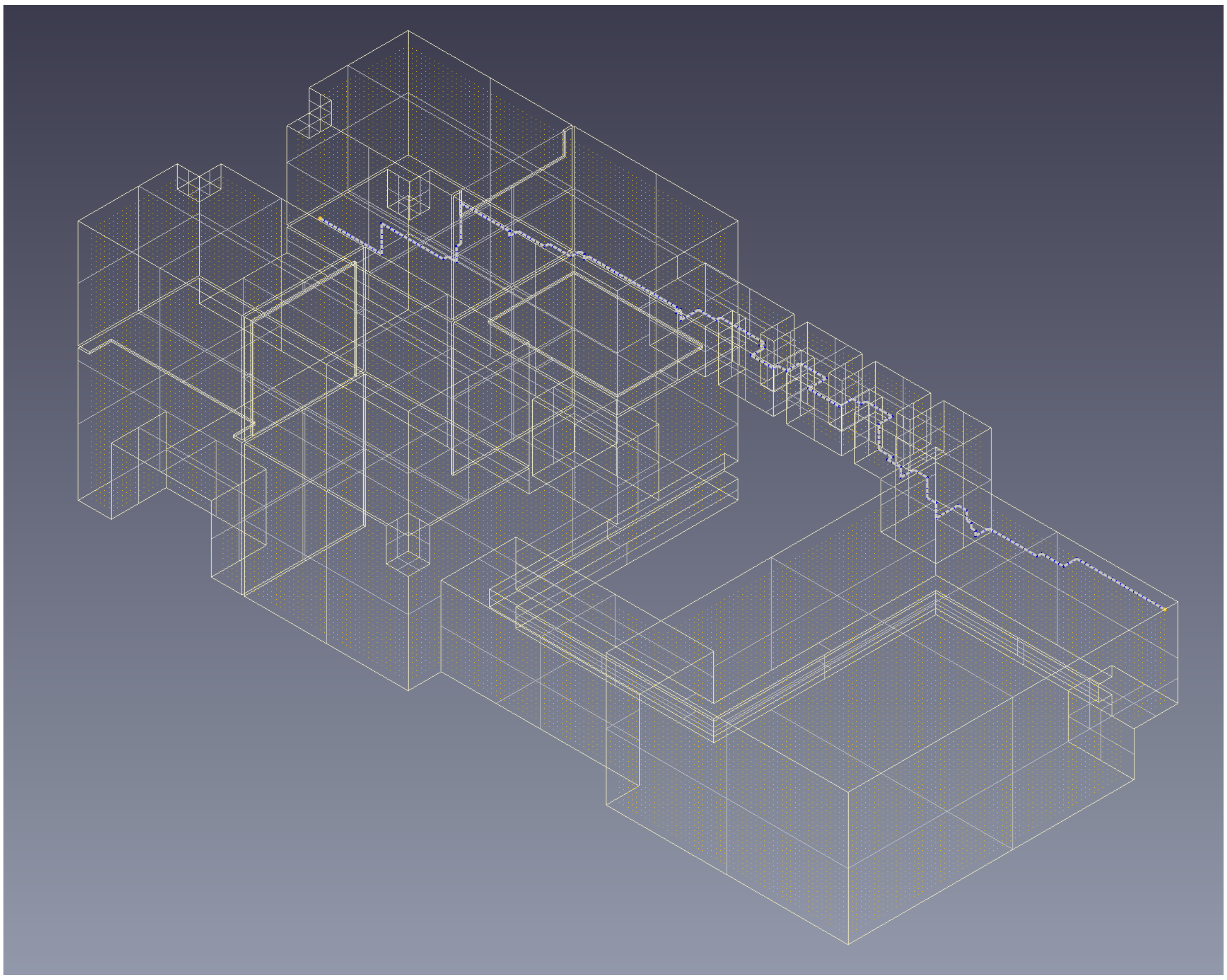
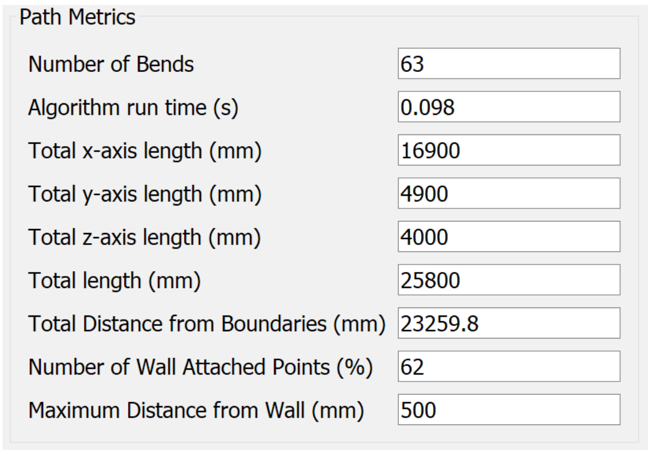


Figure 57: Model 4: Explanatory layout space illustration

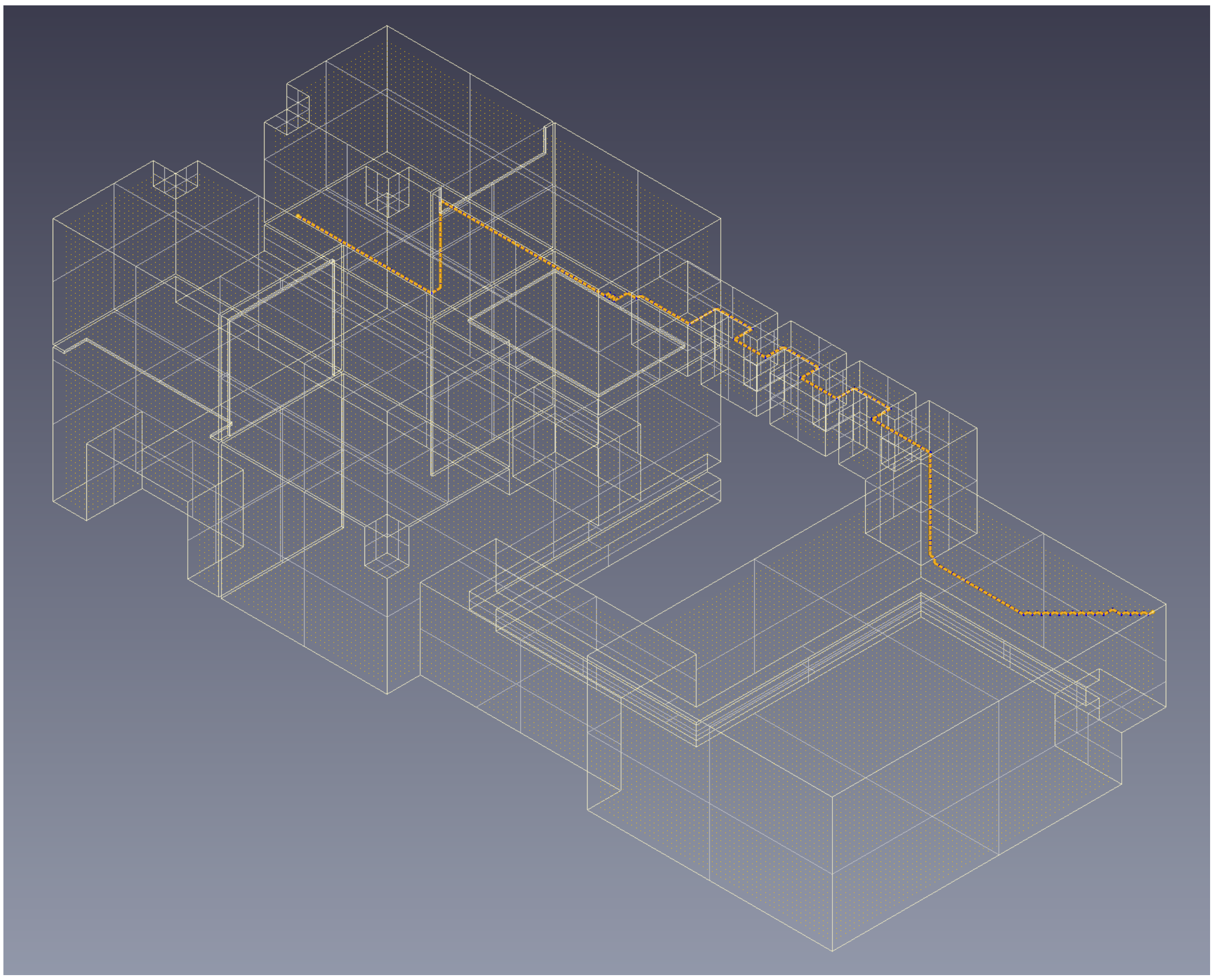


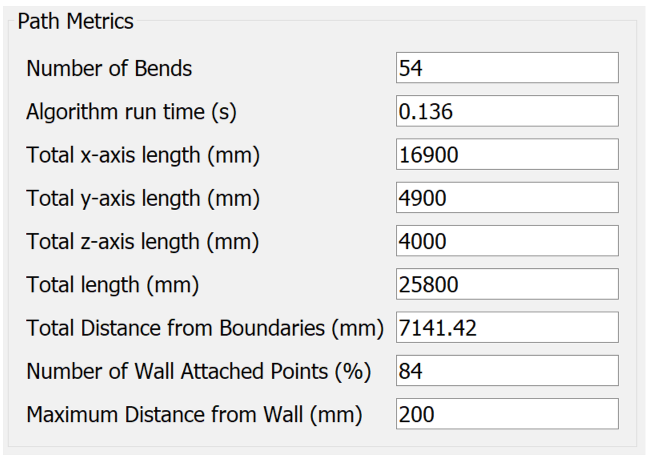


Path Metrics

Number of Bends	63
Algorithm run time (s)	0.098
Total x-axis length (mm)	16900
Total y-axis length (mm)	4900
Total z-axis length (mm)	4000
Total length (mm)	25800
Total Distance from Boundaries (mm)	23259.8
Number of Wall Attached Points (%)	62
Maximum Distance from Wall (mm)	500

Figure 58: Test run #35 illustration and metrics

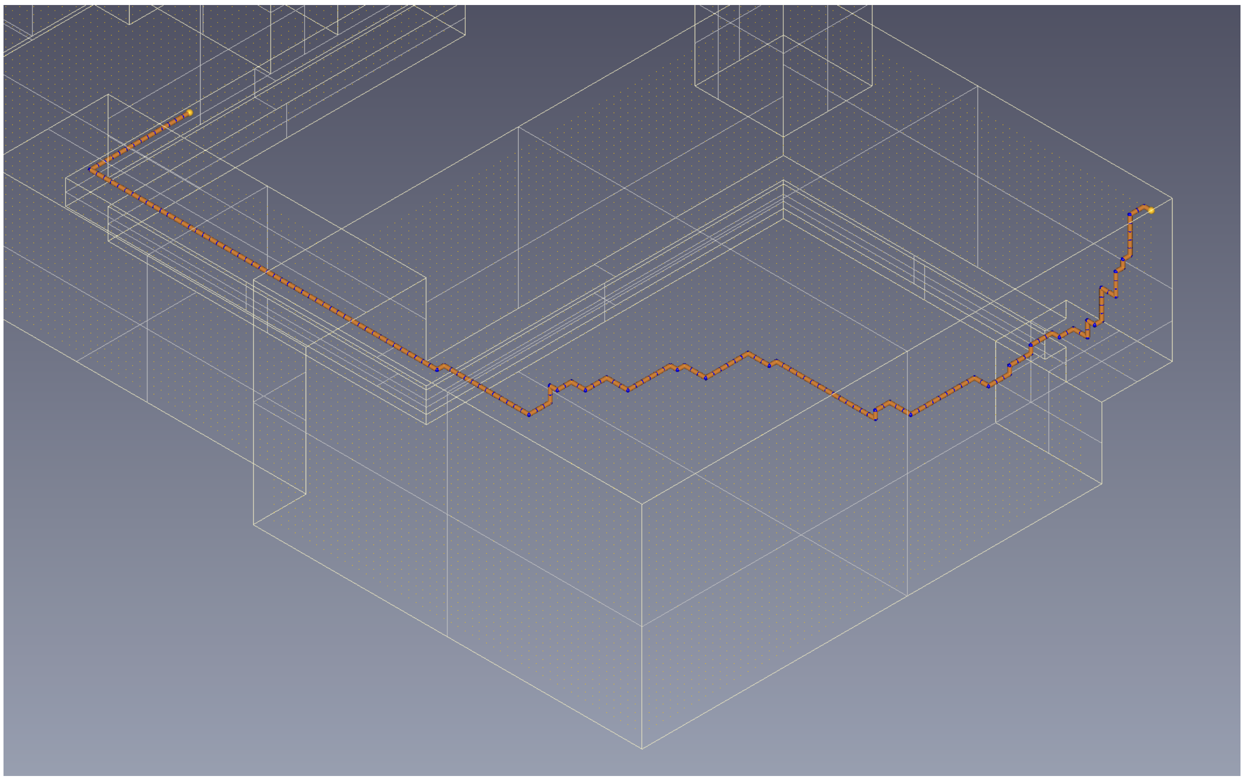


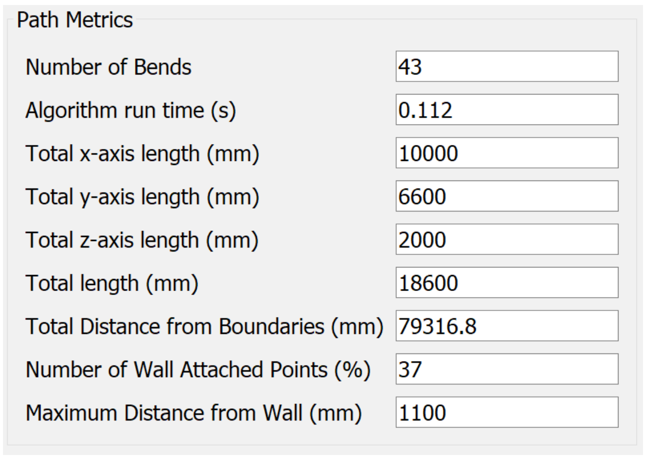


Path Metrics

Number of Bends	54
Algorithm run time (s)	0.136
Total x-axis length (mm)	16900
Total y-axis length (mm)	4900
Total z-axis length (mm)	4000
Total length (mm)	25800
Total Distance from Boundaries (mm)	7141.42
Number of Wall Attached Points (%)	84
Maximum Distance from Wall (mm)	200

Figure 59: Test run #36 illustration and metrics

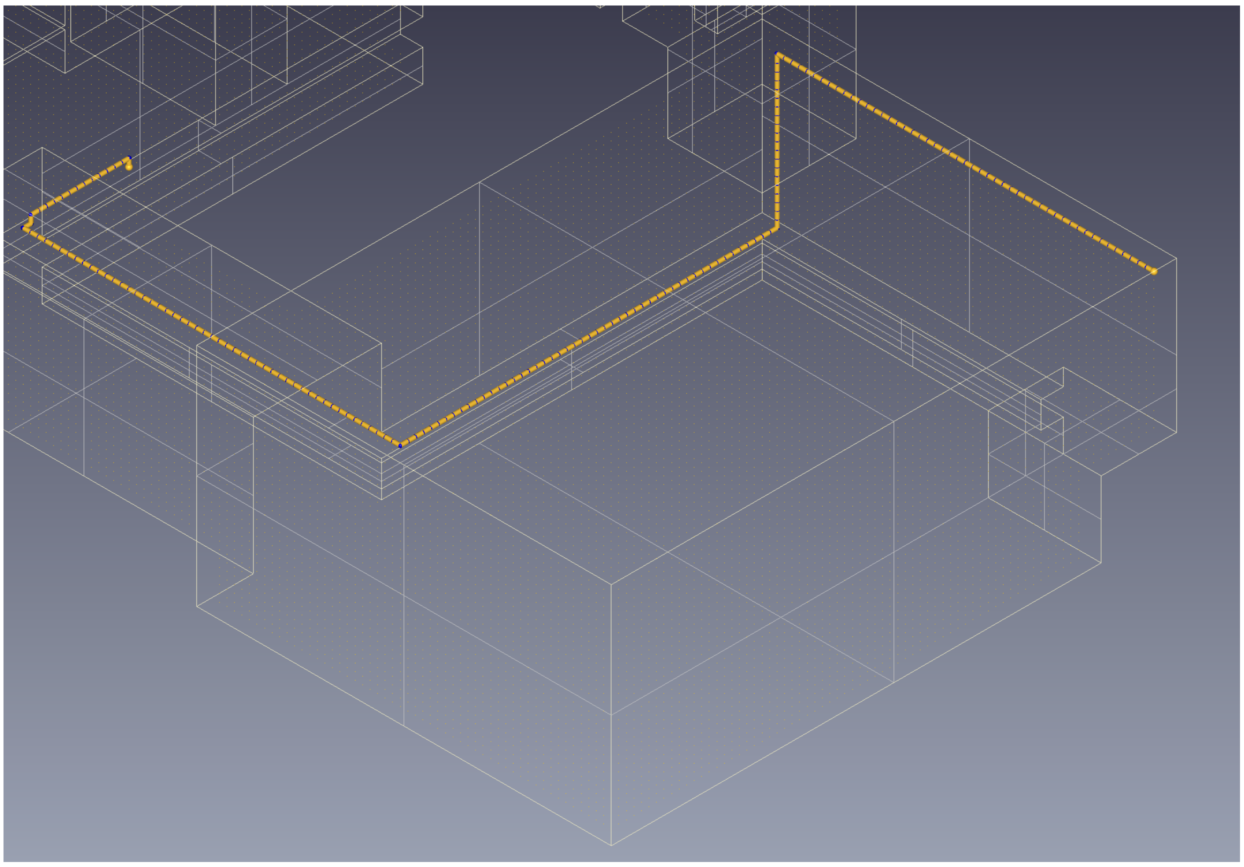


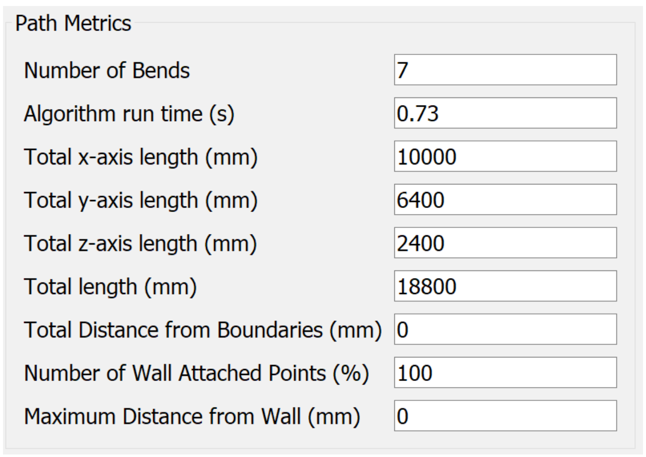


Path Metrics

Number of Bends	43
Algorithm run time (s)	0.112
Total x-axis length (mm)	10000
Total y-axis length (mm)	6600
Total z-axis length (mm)	2000
Total length (mm)	18600
Total Distance from Boundaries (mm)	79316.8
Number of Wall Attached Points (%)	37
Maximum Distance from Wall (mm)	1100

Figure 60: Test run #37 illustration and metrics

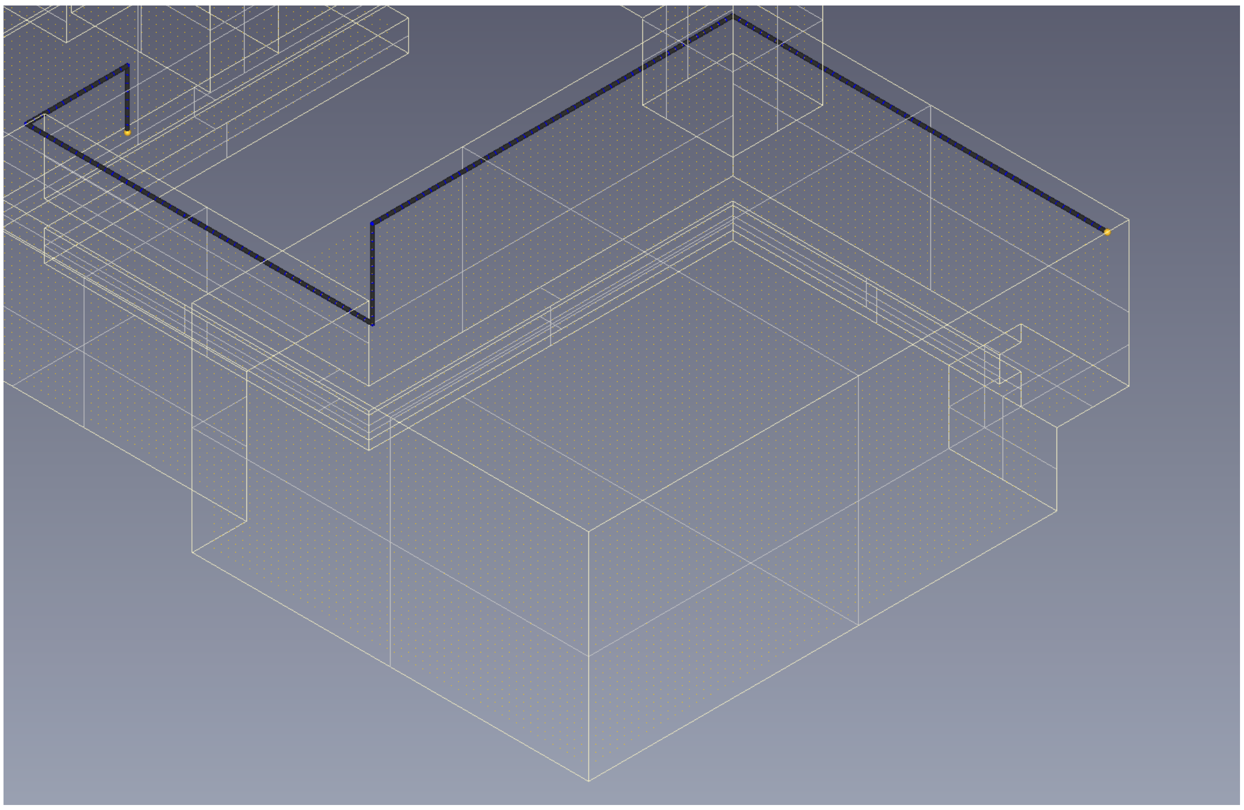


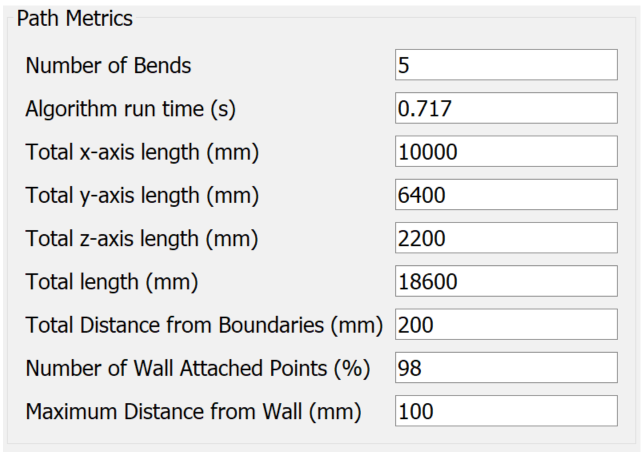


Path Metrics

Number of Bends	7
Algorithm run time (s)	0.73
Total x-axis length (mm)	10000
Total y-axis length (mm)	6400
Total z-axis length (mm)	2400
Total length (mm)	18800
Total Distance from Boundaries (mm)	0
Number of Wall Attached Points (%)	100
Maximum Distance from Wall (mm)	0

Figure 61: Test run #38 illustration and metrics

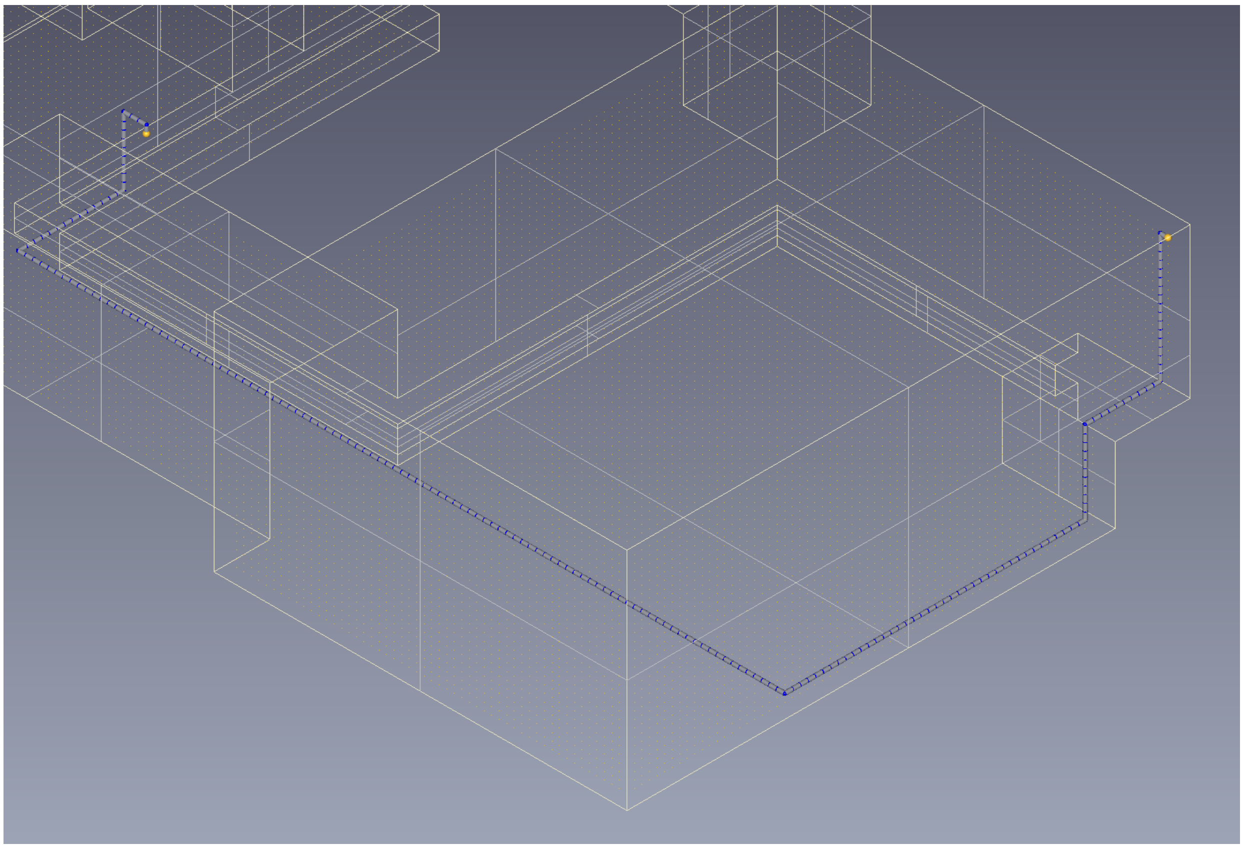


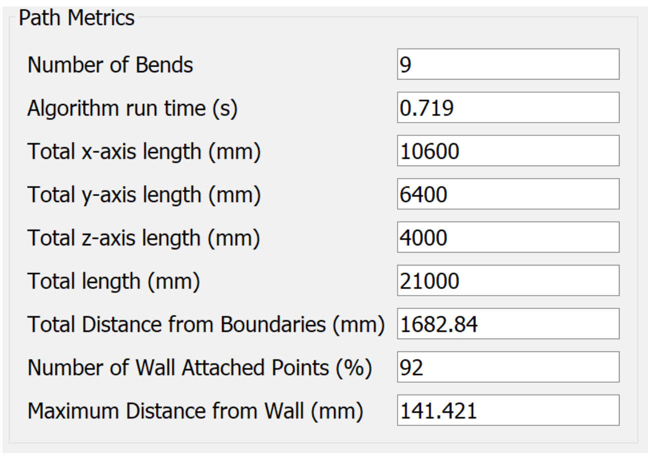


Path Metrics

Number of Bends	5
Algorithm run time (s)	0.717
Total x-axis length (mm)	10000
Total y-axis length (mm)	6400
Total z-axis length (mm)	2200
Total length (mm)	18600
Total Distance from Boundaries (mm)	200
Number of Wall Attached Points (%)	98
Maximum Distance from Wall (mm)	100

Figure 62: Test run #39 illustration and metrics





Path Metrics

Number of Bends	9
Algorithm run time (s)	0.719
Total x-axis length (mm)	10600
Total y-axis length (mm)	6400
Total z-axis length (mm)	4000
Total length (mm)	21000
Total Distance from Boundaries (mm)	1682.84
Number of Wall Attached Points (%)	92
Maximum Distance from Wall (mm)	141.421

Figure 63: Test run #40 illustration and metrics

4.3.4.3. Results assessment

Considering the previous test runs, the following conclusions can be drawn:

(1) From Figures 58 – 59, we can see that the A* outruns Dijkstra as far as run time is concerned. However, these run times are insubstantial. Overall, both algorithms perform really well, with A* gaining the edge bend-wise. This again leads us to the conclusion that in case a non-split graph structure is used for path-finding, A* is the best choice as far as the minimum number of bends is concerned.

(2) At this point, great emphasis should be given on the results presented in Figures 60 – 63. Figure 60, shows the shortest path between that start and goal vertices. However, the solution happens to cross through usable free-space. This would lead to interference of sorts, thus rendering the proposed solution as non-viable. However, when the proposed weight allocation methods come into play, in combination with the use of a split graph, the results get way better. In Figure 62, we see that the resulting path is routed near the ceiling of the layout space, while in Figure 63, it is routed near the floor. As a result, this leads us to the conclusion that the proposed weight allocation methods truly lead to better results, as far as the maximization of the available continuous free-space is concerned.

4.3.5. Model 5: Simplified engine room space

4.3.5.1. Test case presentation

So far, the proposed pipe-routing software has been tested rigorously in four models of escallating complexity. All of this models were created in order to bring up potential issues that traditional pipe-routing methods exhibit, while showing how our proposed method goes some way towards tackling them. Last but not least, a simplified, yet informationally complete ship engine room model is considered.

One set of test runs is conducted for the current near real-life engine room model. These test runs are not evaluated, based on their metric parameters, because they aim only to illustrate the problem-solving capabilities of our proposed pipe-routing methodology. All in all two test runs are conducted. Table 13 contains all the information needed to recreate each of these individual test runs.

Path-finding settings	*Test run #41*	*Test run #42*
Voxel Size (mm)	500	500
Graph Type	Standard	Standard
Split graph	True	True
Start point (mm)	(23600 , 22500, 4000)	(23000, 11100, 14000)
End point (mm)	(23600, 9000, 14000)	(19000, 12600, 6000)
Direction (S-E)	5 - 5	1 - 1
Position-level method	Not utilized	Not utilized
Plane-distance method	✓	✓
Function	Increasing exponential	Increasing exponential
Multiplier	100	100
Algorithm	A*	A*
Heuristic	Euclidean	Euclidean
Dir. “straight” weight	1	1
Dir. “bend” weight	20	20

Table 13: Model 5: Test run parameters

4.3.5.2. Results illustration

Based on on the data presented in Table 13 the results of the two test runs are presented in the figure illustrations below. However, for the shake of clarity, the current model on which the test runs were performed, is also presented from different angles.

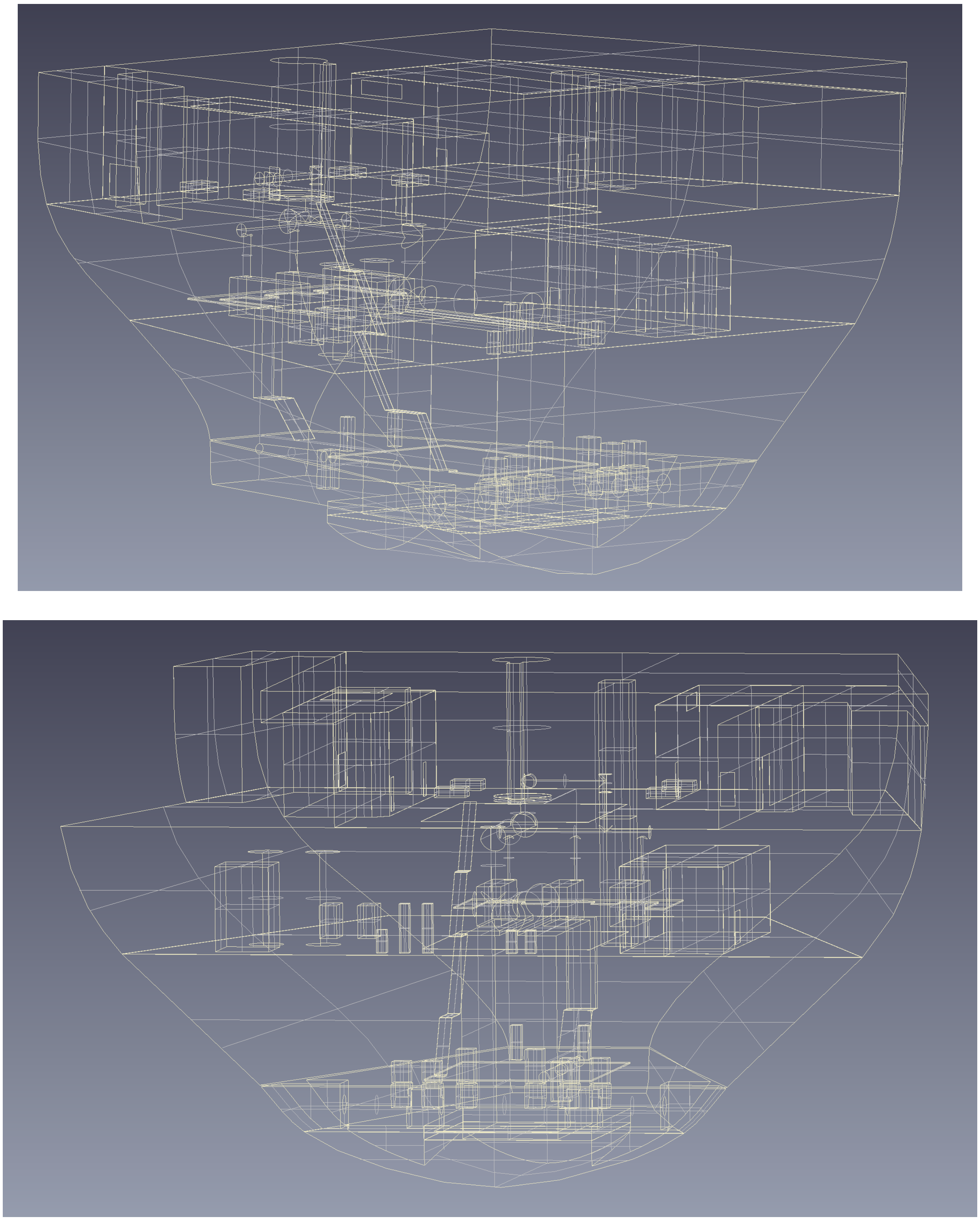
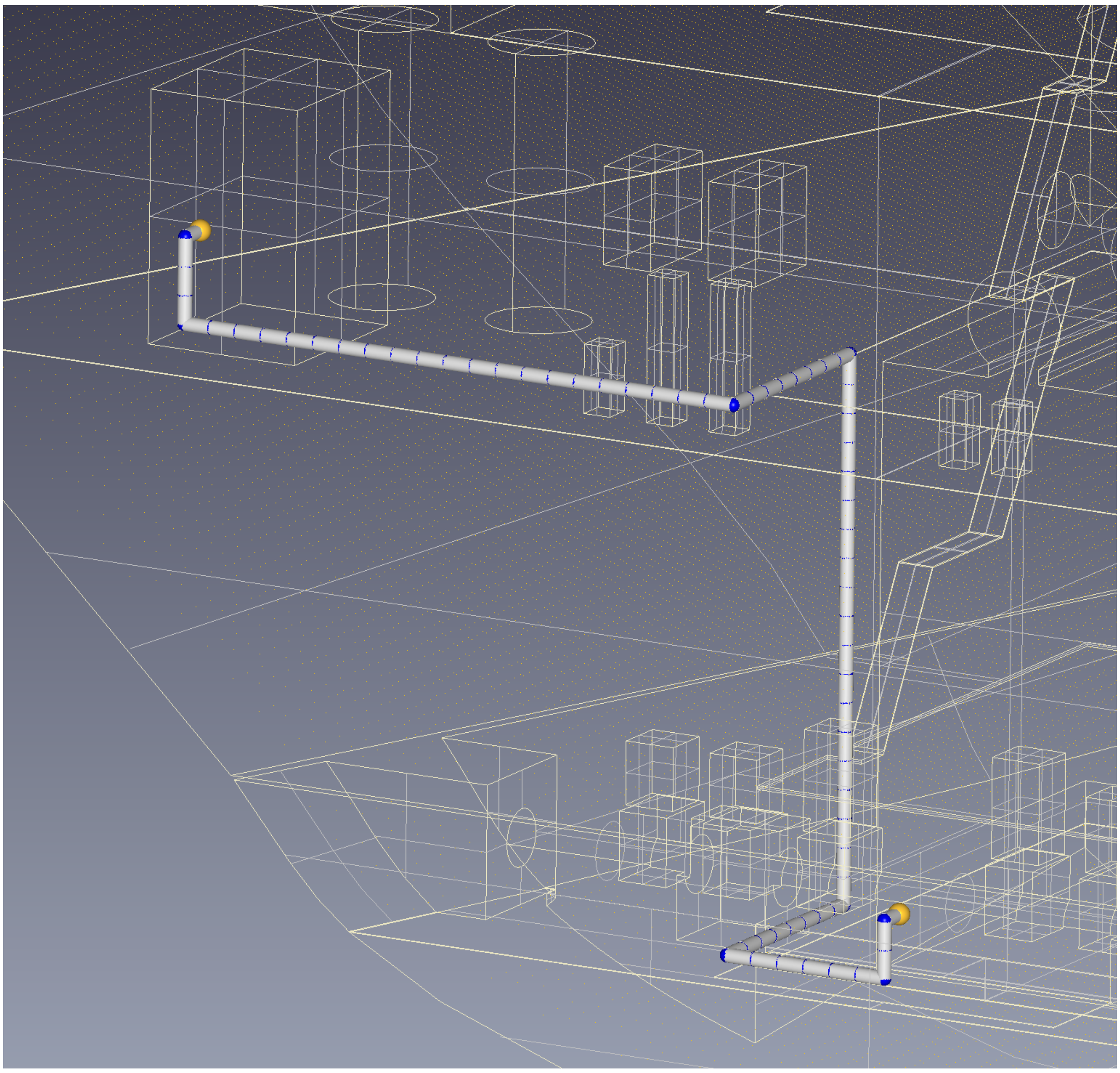
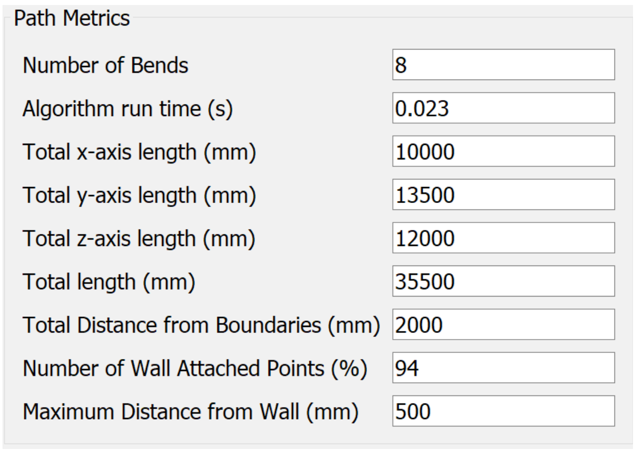


Figure 64: Model 5: Explanatory layout space illustration

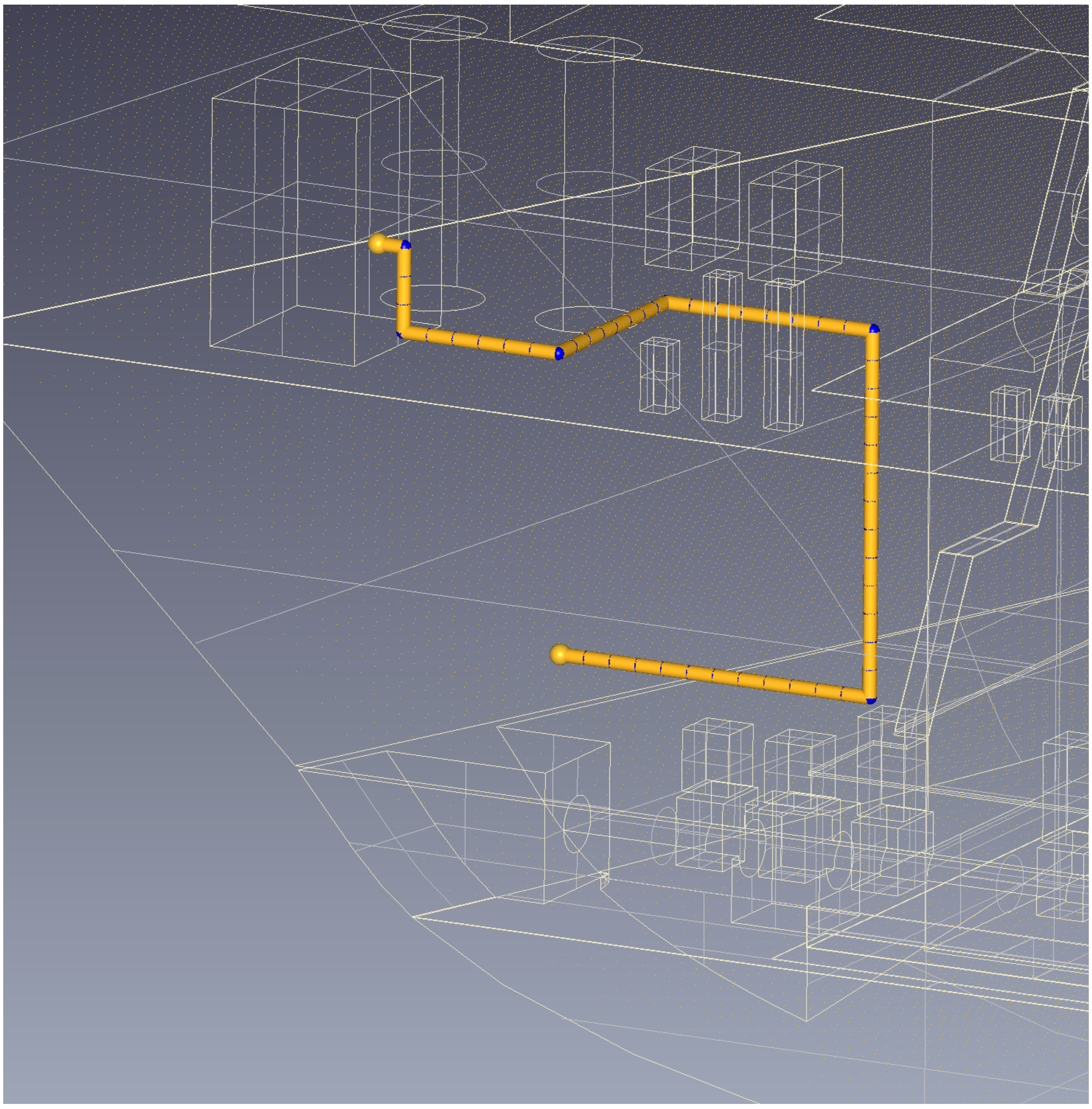


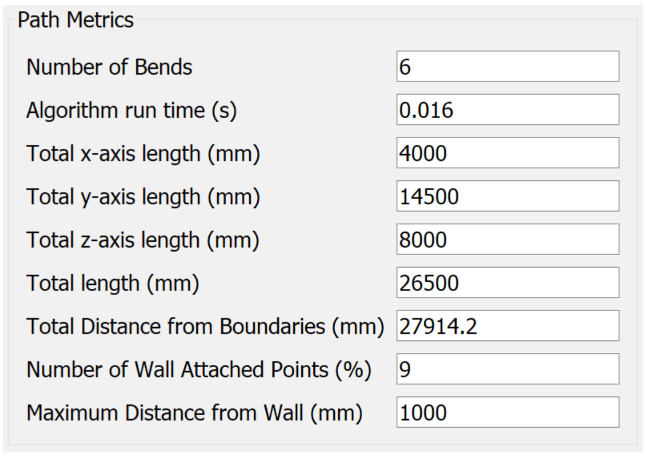


Path Metrics

Number of Bends	8
Algorithm run time (s)	0.023
Total x-axis length (mm)	10000
Total y-axis length (mm)	13500
Total z-axis length (mm)	12000
Total length (mm)	35500
Total Distance from Boundaries (mm)	2000
Number of Wall Attached Points (%)	94
Maximum Distance from Wall (mm)	500

Figure 65: Model 5: Test run #41 illustration and metrics





Path Metrics

Number of Bends	6
Algorithm run time (s)	0.016
Total x-axis length (mm)	4000
Total y-axis length (mm)	14500
Total z-axis length (mm)	8000
Total length (mm)	26500
Total Distance from Boundaries (mm)	27914.2
Number of Wall Attached Points (%)	9
Maximum Distance from Wall (mm)	1000

Figure 66: Model 5: Test run #42 illustration and metrics

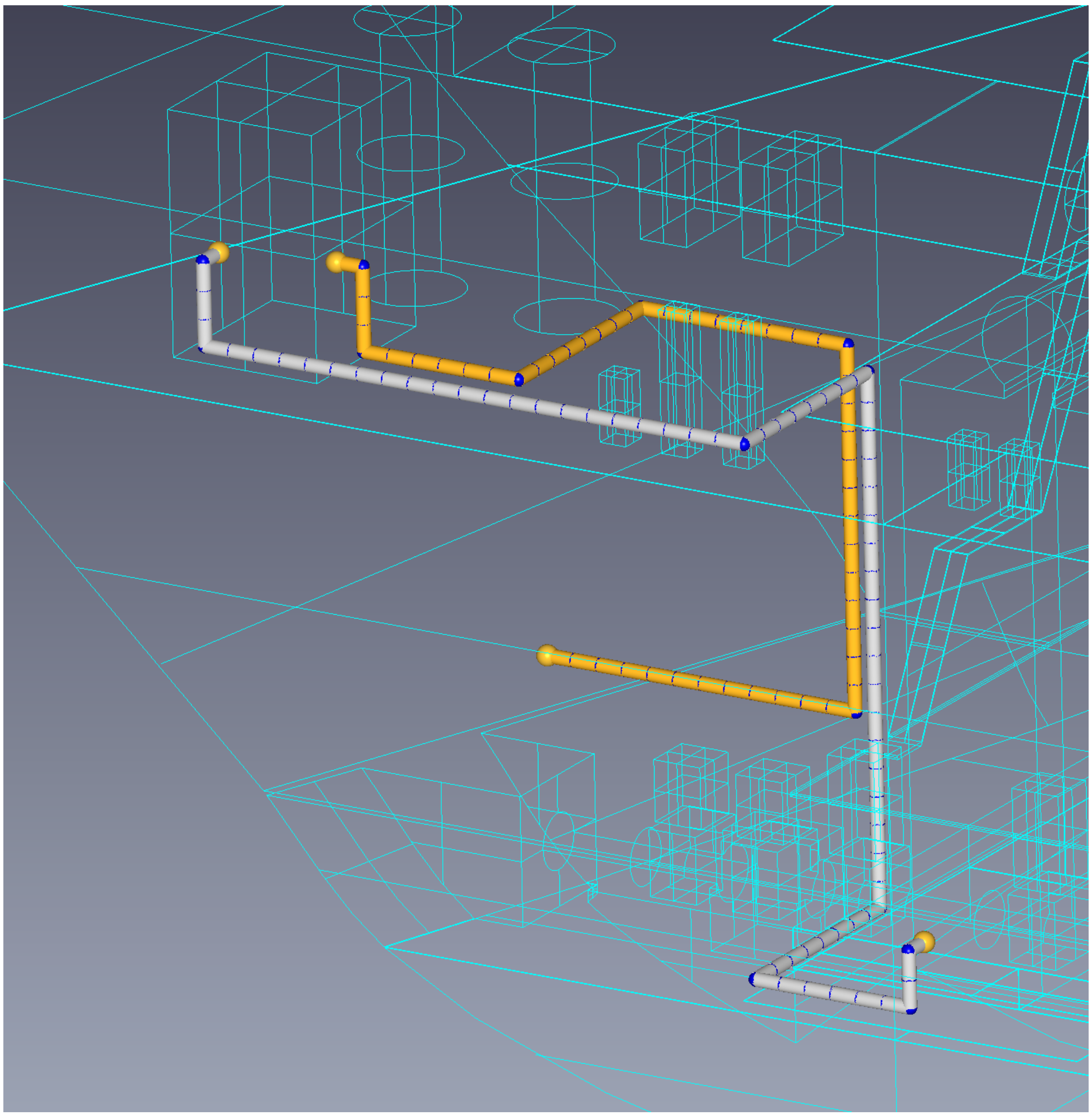


Figure 67: Model 5: Test run #41 & # 42 joined complete illustration

5. Conclusions and future work

5.1. Conclusions

An automatic pipe-routing methodology is devised and implemented within the frameworks of a dedicated piece of software. This methodology is comprised of two main aspects, the modeling and the algorithmic one. Dijkstra and the A* are used within a graph-based structure, built upon the cell decomposed model of a given layout workspace. Both orthogonal, or standard as referred to in the present Thesis, and diagonal graph structures are considered, while at the same time an innovative vertex-split strategy is introduced to the problem, that allows directional specifications being considered in the problem of pipe-routing. Last but not least, two edge weight allocation methods are proposed, the position-level and the distance-from-plane method.

This methodology is tested in five different models, of increasing layout complexity. The results show the following:

(1) The resulting paths are always obstacle-free, meaning that no collision is detected during the pipe-routing process.

(2) The proposed vertex-split strategy drastically reduces the number of bends included in each candidate optimal path-algorithm.

(3) The proposed vertex-split strategy manages to lead to the creation of paths that follow specific directional specifications regarding the start and goal point of each individual pipe-routing scenario.

(4) The proposed edge weight allocation methods, actively affect the whereabouts of the resulting paths, thus enabling the designer to minimize the wasted free-space as a result of a pipe being routed.

(5) The proposed diagonal graph structure, in combination with its extended vertex-split strategy, shows promising results that could be further expanded.

However, the following issues exist:

(1) The developed method so far deals only with single pipe-routing problems, meaning that it does not deal with the problem of branch in pipe-routing.

(2) It is unclear as to what extent it is possible to fit the resulting paths with standardized pipe components, in order to tackle with real life problems.

(3) Many parameters to the problem, such as the directional weights, that have to do with the edges connecting the split-vertices, have a complex and somehow elusive effect on the resulting paths, leading sometimes to less than optimal solutions, in light of the absence of an optimization module.

When all is said and done, the proposed pipe-routing method and its accompanying software, can produce results that can assist any designer involved with the field of pipe-routing. These results, although not applicable in real life situations per se, they can serve as a guide to further required developments, with the goal of reducing the man-hours needed for pipe-routing tasks.

5.2. Future work

The pipe-routing concept proposed in the present work, opens up the path to many additional improvements on the process in question. The software developed in the scope of this study can be further improved to accommodate more functionalities that will make the developed method all the more capable to deal with real life problems. Many things could be said, the most important of them being:

(1) The development of a more efficient voxelization technique should be considered. One that would enable us to decompose any given layout space more efficiently in terms of computational resources and time needed. On top of that, the concept of decomposing part of the layout workspace in question should also be considered.

(2) The algorithmic aspect of the proposed methodology should be extended in order to be able to tackle the problem of routing multiple pipes at the same time, also considering the problem of branch-pipe-routing.

(3) Further edge weight allocation methods should be considered, in order to pluralize the optimal path solution pool.

(4) An optimization module should be created in order to best choose from the multitudes of parameters that define our current pipe routing process.

(5) A thorough study should be conducted, regarding the fitting of the resulting optimal path solutions with actual standardized pipe components.

5. Literature – References

1: SUI, H. T., NIU, W. T., Branch-pipe-routing approach for ships using improved genetic algorithm, Frontiers of Mechanical Engineering, vol.11, no.3, pp.316–323, 2016
2: LEE, C.Y., An algorithm for path connections and its applications. IEEE Trans on Electron Computers, 10(3), pp.346-365. , 1961
3: DIJKSTRA, E. W., A Note on Two Problems in Connection with Graphs, Numerische Mathematlk l, 269 - 27, 1959
4: HART, P., NILLSON, N. and RAPHAEL, B., A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions Systems Science and Cybernetics, 4, 100-107. , 1968
5: QIAN, X. L., REN, T., and WANG, C. E. , A survey of pipe routing design, Proceedings, Institute of Electrical and Electronics Engineers Control and Decision Conference, July 2–4, Yantai, China, 3994–3998. , 2008
6: ITO, T., A genetic algorithm approach to piping route path planning, Journal of Intelligent Manufacturing, 10, 103–114, 1999
7: SANDURKAR, S. and CHEN, W., GAPLUS-Genetic Algorithms Based pipe Routing Using Tessellated Objects, Journal of Computers in Industry, Volume 38, Issue 3, pp.209-223, 1999
8: FAN, X. N. , LIN, Y. and JI, Z. H., A variable length coding genetic algorithm to ship pipe path routing optimization in 3D space, Shipbuilding of China, vol.48, no.1, pp.82–90, 2007
9: REN, T., ZHU, Z. -L. DIMIROVSKI, G. M., GAO, Z. -H., SUN, X. -H., A new pipe routing method for aero-engines based on genetic algorithm, Proceedings of the Institution of Mechanical Engineers, PartG: Journal of Aerospace Engineering, vol.228, no. 3, pp. 424–434, 2014
10: LIU, Q., WANG, C., Pipe-assembly approach for aero-engines by modified particle swarm optimization, Assembly Automation, Vol. 30 Issue: 4, pp.365-377, 2010
11: LIU, W., WANG, C. E., A discrete particle swarm optimization algorithm for rectilinear branch pipe routing, Assembly Automation, vol.31, no.4, pp. 363–368, 2011
12: FAN, X. N., LIN, Y., and JI, Z. S. , Multi ant colony cooperative coevolution for optimization of ship multi pipe parallel routing, Journal of Shanghai Jiaotong University, 43, 2, 193–197. , 2009
13: JIANG, W. -Y., LIN, Y., CHEN, M., YU, Y. -Y., An ant colony optimization - Genetic algorithm approach for ship pipe route design, International Shipbuilding Progress, vol.61, no.3-4, pp. 163–183, 2014
14: JIANG, W. -Y., LIN, Y., CHEN, M., YU, Y. -Y., A co-evolutionary improved multi-ant colony optimization for ship multiple and branch pipe route design, Ocean Engineering, vol.102, pp.63– 70 , 2015
15: FAN, X. N., LIN, Y., and JI, Z. S. , The ant colony optimization for ship pipe route design in 3D space, Proceedings, World Congress on Intelligent Control and Automation, June 21–23, Dalian, China, 3103–3108. , 2006
16: KIMURA, H. , Automatic designing system for piping and instruments arrangement including branches of pipes, Proceedings, International Conference on Computer Applications in Shipbuilding, September 20–22, Trieste, Italy, 93–99. , 2011
17: NIU, W., SUI, H., NIU, Y., CAI, K., GAO, W., Ship Pipe Routing Design Using NSGA-II and Coevolutionary Algorithm, Mathematical Problems in Engineering Volume 2016, Article ID 7912863, 21 pages, 2016
18: DONG, Z., LIN, Y.†, Ship Pipe Routing Method Based on Genetic Algorithm and Cooperative Coevolution, Journal of Ship Production and Design, Vol. 33, No. 2, pp. 122–134 , 2017
19: ANDO, Y.and KIMURA, H. , An automatic piping algorithm including elbows and bends, Proceedings, International Conference on Computer Applications in Shipbuilding, September 20–22, Trieste, Italy, 153–158. , 2011
20: NGUYEN, H., KIM, D. -J., GAO, J., 3D Piping Route Design Including Branch and Elbow Using Improvements for Dijkstra's Algorithm, International Conference on Artificial Intelligence:

Technologies and Applications (ICAITA), 2016
21: KIM, S-., H., RUY, W-., S., JANG, B., S. , The development of a practical pipe auto-routing system in a shipbuilding CAD environment using network optimization, nternational Journal of Naval Architecture and Ocean Engineering. 5: 368 - 477 , 2013
22: MIKAMI, K.and TABUCHI, K. , A computer program for optimal routing of printed circuit conductors, Proceedings, International Federation of Information Processing Societies, H47, August 5–10, Edinburgh, UK, 1475–1478. , 1968
23: HIGHTOWER, D. W., A solution to line routing problems on the continuous plane, Proceedings, the 6th Annual Design Automation Conference, June 8–12, New York, NY, 1–24. , 1969
24: ASMARA, A. , Pipe routing framework for detailed ship design, PhD thesis, Delft University of Technology, Delft, The Netherlands., 2013
25: PARK, J. H. and STORCH, R. L., Pipe-routing Algorithm Development: Case Study of a Ship Engine Room Design, Journal of Expert Systems with Applications, 23(3), 299-309, 2002
26: HUIBIAO, L., ZHEFU, Y., PEITING, S., Hanging Bridge Algorithm for Pipe-Routing Design in Ship Engine Room, International Conference on Computer Science and Software Engineering, Hubei, pp. 153-155, 2008
27: DONG, Z., CHEN, W., BAO, H., ZHANG, H., PENG, Q., A Smart Voxelization Algorithm, 2014
28: GARCIA, F., OTTERSTEN, B., CPU-Based Real-Time Surface and Solid Voxelization for Incomplete Point Cloud, 22nd International Conference on Pattern Recognition, Stockholm, pp. 2757-2762, 2014
29: BARILL, G., DICKSON, N. G., SCHMIDT, R., LEVIN, D. I. W., JACOBSON, A., Fast Winding Numbers for Soups and Clouds, ACM Transactions on Graphics, Vol. 37, No. 4, Article 43, 2018
30: NITA, C., et al., GPU Accelerated, Robust Method for Voxelization of Solid Objects, IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, pp. 1-5, 2016
31: BAERT, J., LAGAE, A., DUTRE, P., Out-of-Core Construction of Sparse Voxel Octrees, Proceedings of the Fifth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics, 2014
32: NOURIAN, P., GONCALVES, R., ZLATANOVA, S., OHORI, K. A., VO, A. V, Voxelization algorithms for geospatial applications Computational methods for voxelating spatial datasets of 3D city models containing 3D surface, curve and point data models, MethodsX 3, pp. 69–86., 2016
33: PANTALEONI, J., VoxelPipe: A Programmable Pipeline for 3D Voxelization, Proceedings - HPG 2011: ACM SIGGRAPH Symposium on High Performance Graphics, 2011
34: SCHWARTZ, M., SEIDEL, H. -P., Fast Parallel Surface and Solid Voxelization on GPUs, ACM Transactions on Graphics - TOG, 2010
35: KOOPMAN, M., 3D Path-finding in a voxelized model of an indoor environment, http://resolver.tudelft.nl/uuid:13788271-e19d-41e1-b827-fe7535a66281, 2016

Appendix A

The typical format of a .vraw input file is presented in this Appendix.

```
<Voxel Size>
500 500 500
<Voxel Count>
50 92 52
<Deck position>
6
0 2852 5242 12000 18600 25640
<Longitudinal Bulkhead position>
0
<Transverse Bulkhead position>
0
<Origin Point>
0 -22779.669274 0
<Cad Filename>
EngineRoomModel.igs
<Voxel tag>
0
0
0
...
```

Appendix B

The typical format of a .path output file is presented in this Appendix.

```
<Number of Paths>
1
<Graph Type>
0
<Split>
0
<Starting Point>
300 300 0
<Start Direction>
0
<Target Point>
13500 12000 6600
<Target Direction>
0
<Position Lvl Weight Penalty Flag>
0
<Position Lvl Weight Penalty Function ID>
0
<Position Lvl Weight Penalty Multiplier>
0
<Plane Distance Weight Penalty Flag>
0
<Plane Distance Weight Penalty Function ID>
0
<Plane Distance Weight Penalty Multiplier>
0
<Path Finding Algorithm>
1
<Heuristic>
1
```

```
<Size of Point ID's vector>
72
<Point ID's>
...
<Number Of Bends>
0
<Total X-axis Length>
13800
<Total Y-axis Length>
11700
<Total Z-axis Length>
6600
<Total Length Sum>
25806.9
<Total Distance from Boundaries>
5700
<Number of Wall Attached Points>
61
<Max-Min Wall Distance>
900
```