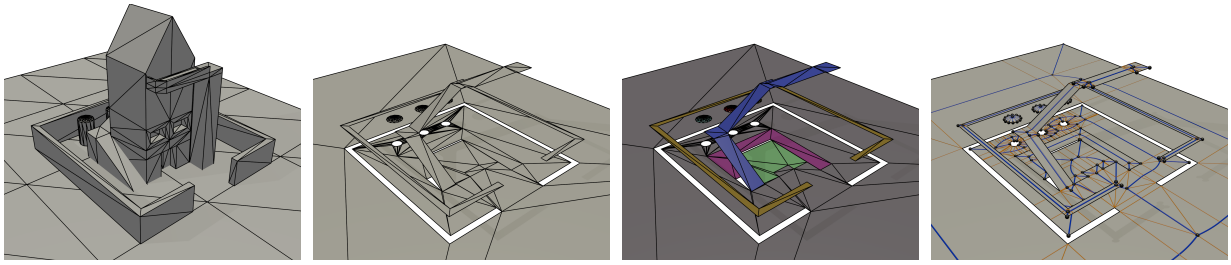
Separating a Walkable Environment into Layers

Arne Hillebrand* Marjan van den Akker Roland Geraerts
Han Hoogeveen
Department of Information and Computing Sciences, Utrecht University, the Netherlands



(a) *Polygonal Environment* **(b)** *Walkable Environment* **(c)** *Multi-Layered Environment* **(d)** *Navigation Mesh*

Figure 1: *(a): A polygonal environment or model. (b): The walkable environment for this model. (c): Its multi-layered environment. Polygons with the same colour are in the same layer. (d): By using the multi-layered environment we can create, for example, a navigation mesh.*

Abstract

A *multi-layered environment* (MLE) [van Toll et al. 2011] is a representation of the *walkable environment* (WE) in a 3D virtual environment that comprises a set of two-dimensional layers together with the locations where the different layers touch, which are called connections. This representation can be used for crowd simulations, e.g. to determine evacuation times in complex buildings, or for finding the shortest routes. The running times of these algorithms depend on the number of connections.

Finding an environment with the smallest number of connections, is an NP-Hard problem [Hillebrand et al. 2016]. Our first algorithm tackles this problem by using an integer linear program which is capable of finding the best possible solution, but naturally takes a long time. Hence, we provide two heuristics that search for MLEs with a low number of connections. One algorithm uses local search to gradually improve the found solution. The other one, called the height heuristic, is very fast and gives good solutions in practical environments.

Keywords: multi-layered environment, optimization

Concepts: •**Mathematics of computing** → *Simulated annealing; Integer programming;* •**Theory of computation** → *Computational geometry;*

*Corresponding author. E-mail: A.Hillebrand@uu.nl

MiG '16,, October 10 - 12, 2016, Burlingame, CA, USA
ISBN: 978-1-4503-4592-7/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2994258.2994271

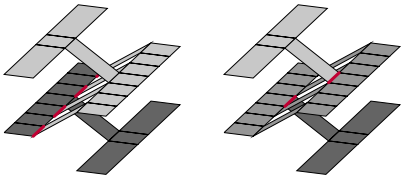
1 Introduction

Evacuation planning and crowd simulations for safety purposes are becoming more and more important in modern society. To perform such simulations in a soccer stadium for example, we need its underlying *polygonal environment* (PE), which is a common format used by architects [Whyte et al. 2000] and 3D modelling tools. A PE is a collection of polygons in $\mathbb{R}^3$ which can be processed through a pipeline to mould it in an appropriate format (see Fig. 1). Fig. 1(a) shows that such a PE usually contains unnecessary details for simulations. We only need a filtered version of the PE that contains the polygons that are traversable by agents. This type of environment is called a WE. Examples of polygons that are not needed in the WE are polygons that are too steep, too close to a ceiling or polygons for which there is not enough clearance for an agent to stand. An example of a WE is shown in Fig. 1(b). Polygons in a WE can overlap or be connected. Two polygons *overlap* when they share at least one point when projected on the ground plane, and the point is not on an edge of both polygons. Two polygons P and Q are *connected* when they do not overlap but share exactly one edge $e_{P,Q}$. When they are connected, it is possible for a virtual point agent to move from P to Q and vice versa.

On a WE we want to run algorithms to e.g. construct visibility graphs [Lozano-Pérez and Wesley 1979] or to create navigation meshes [Snook 2000]. Visibility graphs can be used for finding shortest paths in the environment, and navigation meshes enable fast path planning queries that are used for crowd simulations. However, such algorithms currently only exist for two-dimensional environments. These algorithms can be extended to layered two-dimensional environments by using a MLE, see Fig. 1(c). An MLE is a decomposition of the WE in layers, such that every layer can be embedded in the plane. When two polygons P and Q share an edge $e_{P,Q}$ and do not overlap but are in different layers, $e_{P,Q}$ connects the two layers. This type of edge is called a *connection*.

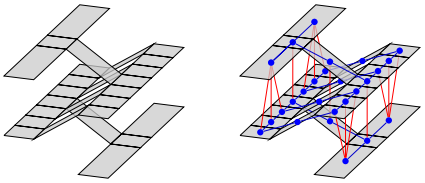
Definition 1 (Multi-Layered Environment, [van Toll et al. 2011]) *An MLE for a given WE consists of a set* $\mathbf{L} = \{L_1, ..., L_l\}$ *of two-dimensional layers and a set* C *of connections, such that:*

- *No layer* L_i $(i = 1, \ldots, l)$ *contains overlapping polygons;*
- *Every polygon in the WE is assigned to exactly one layer* L_i $(i = 1, \ldots, l)$*;*



(a) *MLE* **(b)** *MICLE*

Figure 2: *Two MLEs for the same WE. Polygons with the same shade of grey are in the same layer. The red edges are connections. (a): An MLE with two layers and four connections. (b): An MLE with three layers and only two connections. This MLE is a MICLE.*



(a) *WE* **(b)** *WEG*

Figure 3: *A walkable environment and its corresponding graph representation. (a): The walkable environment. (b): Vertices are added for every polygon in the walkable environment. Edges (blue) are added for connected polygons and overlaps are annotated (red).*

- *When two polygons P and Q are connected, but are in different layers, the connection between P and Q is part of the set C;*
- *Every layer L_i ($i = 1, \ldots, l$) forms a single connected component, i.e. for any two polygons P, Q in L_i we can walk from P to Q without ever leaving layer L_i.*

When we use this definition of an MLE, it is rarely the case that there exists only one possible MLE for a WE. Take for example the MLEs given in Fig. 2. The MLE in Fig. 2(a) consists of two layers and has four connections, whereas the MLE depicted in Fig. 2(b) only has two connections. The MLE depicted in Fig. 2(b) is a *minimally connected multi-layered environment* (MICLE) for this WE.

This is the most useful type of MLE because subsequent operations can benefit from having a low number of connections. For example, Van Toll et al. [2011] show that constructing a navigation mesh for an MLE can be done in $O(k \times n \log n)$ time, where k is the number of connections in an MLE and n is the number of obstacle vertices used to describe the boundaries of the individual layers of the MLE. An example of such a navigation mesh can be seen in Fig. 1(d).

In this paper we will focus on the second step in this pipeline, that is, separating a WE into layers with a low number of connections.

1.1 Related work

There are several applications and algorithms that are already using some form of an MLE. However, the MLEs that they use are often of poor quality. They do not cover all of the walkable space or contain a high number of connections. For example, Rodriguez and Amato [2011] use an MLE to find a strategy to efficiently clear a building. They assume that their MLE is supplied. For an MLE they create a roadmap representation. The different layers correspond to subgraphs of the roadmap, and the connections are the edges connecting the different subgraphs. Van Toll et al. [2011] use an MLE to create a multi-layered navigation mesh, which allows for fast path planning queries. One strength of this type of MLE is that the representation of the corresponding WE is exact. However, they do not supply methods to find an MLE either.

Deusdado et al. [2008] use a discretized height map to automatically extract walkable surfaces from a PE. The locations of the connections are determined by comparing the height information of different walkable surfaces. Oliva and Pelechano [2013] overlay the environment with a three-dimensional grid and mark each grid cell positive when it contains walkable geometry. These grid cells are grouped into layers which are then used to create a multi-layered navigation mesh. Pettré et al. [2005] create multiple elevation maps of a PE by using the graphics card. From these elevation maps, slopes that are too steep are filtered and the elevation maps are joined, resulting in a WE. The downside of these techniques is that they are mostly part of a larger algorithm, and do not offer any easy means to obtain the constructed MLE. Furthermore, the resulting MLE is only an approximation of the WE and does not cover all of the WE.

Instead of extracting an MLE from a PE or WE, Jiang et al. [2009] propose a method which models the environment in simple blocks that can be described in two dimensions. The blocks are linked together in a floor plan-like fashion. When an environment is described this way, which it rarely is, an MLE is easy to extract since the layers and connections are explicitly defined.

1.2 Our contributions

To our knowledge, we are the first who try to extract MLEs of high quality from a WE. In Section 2 we will formally define how to find a MICLE. In Section 3, we will describe three different algorithms for extracting MLEs. The first algorithm is an Integer Linear Program. It is guaranteed to find the MLE with a minimal number of connections, but it can take a very long time. The second algorithm is a local search algorithm using simulated annealing. It is much faster than the Integer Linear Program, but generally finds solutions with a lower quality. The third algorithm is a heuristic that uses the height information of each polygon. It is by far the fastest algorithm, but the quality of the solutions is largely dependent on how level the different floors are in the environment. The algorithms from Section 3 are tested in Section 4. These results are discussed in Section 5. We conclude that in most situations the height heuristic is the algorithm with the best performance.

2 Multi-Layered Environments

We first convert the WE into a graph $G = (V, E, O)$. In this graph, a vertex is added to V for every polygon in the WE. An undirected edge (v, w) is added to E whenever the polygons corresponding to the vertices v and w are connected in the WE. When two polygons with corresponding vertices v' and w' overlap, the unordered pair (v', w') is added to O. This graph is called the *walkable environment graph* (WEG) and an example of it can be seen in Fig. 3. It is also possible to assign weights to the edges of the WEG, resulting in a weighted WEG. For a weighted WEG we have $G = (V, E, O, w)$. Here, w is a function that maps every edge $e \in E$ to a real number. Using the WEG, the problem of finding a MICLE can now be formulated as follows:

Problem 1 (Finding a MICLE) *Given WEG $G = (V, E, O)$ of WE W. Finding a MICLE is now the same as finding the cut set $C \subseteq E$ for which:*

- *$\forall v, w$ such that $(v, w) \in O$: v and w are in different graph components for the graph $G' = (V, E \setminus C)$;*
- *$|C|$ is minimal.*

When we are using a weighted WEG, the size of the cut set C is defined as $|C| = w(C) = \sum_{e \in C} w(e)$. When $|O| = 1$, we have the s-t cut problem, which can be solved using the max-flow min-cut theorem [Ford and Fulkerson 1956]. If $|O| > 1$, we have an instance of the *multi-commodity minimal-cut* (MULTICUT) problem [Schrijver 2003]. Unfortunately, finding a MICLE has been proven to be an NP-Hard problem [Hillebrand et al. 2016].

3 Algorithms for Finding an MLE

Since finding a MICLE is an NP-Hard problem, we search for an MLE with a small number of connections. In this section, we will describe three different types of algorithms. Instead of minimizing the number of connections between layers, these algorithms will solve an equivalent problem. We will search for sets of layers $\mathbf{L}$ in a WEG, such that the sum of internal edges I_i for the layers is maximized. Here, I_i is the cumulative weight of the internal edges of layer L_i, i.e. the cumulative weight of the edges of the sub-graph induced by the vertices in layer L_i. These two problems are equivalent since connections in an MLE are non-internal edges. Therefore, by maximizing the cumulative weight of the internal edges, we minimize the weight of the connections.

All these algorithms use a weighted WEG $G = (V, E, O, w)$. First, we will describe how to find MICLEs using Integer Linear Programming and Brand-and-Price [Barnhart et al. 1998]. While this technique can find an optimal solution, this can take a very long time. Next, we will describe a local search approach, using simulated annealing [Ĉerný 1985]. Although local search does not offer any guarantees with respect to optimality, we expect that this technique is able to find reasonable solutions efficiently. The third method, called the Height Heuristic, does not solely depend on the WEG when searching for a solution. It is designed around the assumption that the different floors are level in most buildings, and that they roughly correspond to layers in an MLE.

3.1 (Integer) Linear Program

In the Integer Linear Programming (ILP) framework, a layer L_i is described using two parameters. M_i is a vector of length $|V|$, with $M_{i,v} = 0$ whenever the polygon represented by vertex v is not part of layer i, and 1 otherwise. The other parameter is I_i, the cumulative weight of the internal edges of layer i. Whenever the set of all possible, feasible layers is given and has size n, the ILP for finding a MICLE can be formulated as follows:

Maximize: $\sum_{i=1}^{n} x_i I_i$
Subject to:

$$\sum_{i=1}^{n} x_i M_{i,v} = 1 \qquad \forall v \in V \qquad (1)$$
$$x_i \in \{0, 1\} \qquad 1 \leq i \leq n \qquad (2)$$

Here, x_i is a so-called decision variable that is set to 1 when layer L_i is part of the final solution. Constraint 1 guarantees that a vertex will be part of exactly one of the selected layers, and Constraint 2 ensures that a layer is selected completely or not at all. When Constraint 2 is relaxed to $0 \leq x_i$, we have the corresponding LP-relaxation, since $x_i \leq 1$ is enforced by Constraint 1.

To solve the given ILP, it would be necessary to determine all $O(2^{|V|} - 1)$ possible combinations of layers for a WE. Since this number is so high, we will employ a technique called column generation [Desaulniers et al. 2006] to solve the LP-relaxation, which we will use in our Branch-and-Price approach. This technique allows us to start searching for a solution while not knowing all layers in advance. Instead, it allows for the addition of new layers after first solving the LP-relaxation with a smaller set of columns. Using the concepts of shadow price π_v for Constraint 1 (i.e. the cost for strengthening Constraint 1) and reduced cost, we can search for layers that increase the solution value. The reduced cost for a new layer L_0 is $I_0 - \sum_{v \in V} \pi_v M_{0,v}$. Only when the reduced costs are positive, which means that the value I_0 is bigger than $\sum_{v \in V} \pi_v M_{0,v}$, it is profitable to introduce this new found layer. Finding such a layer with maximum reduced cost is called the pricing problem.

3.1.1 Solving the pricing problem

We will use another ILP to solve the pricing problem. The objective of the pricing problem is to maximize $I_0 - \sum_{v \in V} \pi_v M_{0,v}$. To encode this in our ILP, we will introduce two new decision variables. First, we have $x_{v,w} \in \{0, 1\}$ for all edges in E. The value of $x_{v,w}$ is 1 if the edge (v, w) is an internal edge of the layer. I_0 can now be defined as $\sum_{(v,w) \in E} x_{v,w} w((v, w))$. The second decision variable, y_v, is used to determine if a vertex v is part of the new layer. The variable y_v has the same role in the pricing problem as $M_{0,v}$ in the master problem. With these variables, we can solve the pricing problem by using the following ILP.

Maximize: $\sum_{(v,w) \in E} x_{v,w} w((v, w)) - \sum_{v \in V} y_v \pi_v$
Subject to:

$$-1 \leq 2x_{v,w} - y_v - y_w \leq 0 \qquad \forall v \in V \qquad (3)$$
$$y_v + y_w \leq 1 \qquad \forall (v, w) \in O \qquad (4)$$
$$y_v \in \{0, 1\} \qquad \forall v \in V$$
$$x_{v,w} \in \{0, 1\} \qquad \forall (v, w) \in E$$

Constraint 3 is needed to ensure that the edge $(v, w) \in E$ can only be counted as an internal edge if both v and w are part of the layer. The feasibility of layer L_0 is guaranteed by Constraint 4, which encodes the overlaps from the WEG. We can search for new columns with positive reduced cost by using this ILP. However, this ILP does not guarantee that the found layer forms a single connected component. Therefore, we search for the connected components of a layer found by this ILP. Each individual connected component in a layer is now introduced as a new column.

3.1.2 Branch-and-Price

We will use Branch-and-Price [Barnhart et al. 1998] and Branch-and-Cut [Padberg and Rinaldi 1991], known as Branch-Price-and-Cut, to solve the ILP given in Section 3.1. We first solve the LP-relaxation using column generation as usual. If the resulting solution is integral, we are finished. However, if the current solution is fractional, we need to continue our search for the optimal solution. We will branch on constraints that we will introduce in our problem, rather than on variables. The constraint that we will introduce will force an edge (v, w) to be cut or not to be cut. To accomplish this we will introduce the constraint $y_v + y_w \leq 1$ (if (v, w) is to be cut) or $y_v = y_w$ (if (v, w) is not to be cut) to the pricing problem. In the master problem, we force all variables x_i with $M_{i,v} = 1$ and $M_{i,w} = 1$ to zero when (v, w) is to be cut. After branching we once again solve the new LP-relaxation to optimality using column-generation. Using these techniques, we can find an optimal solution if enough time is available.

3.2 Local search

In local search, we start with a random valid solution. We try to iteratively improve this solution using local changes, while keeping track of the best solution encountered thus far. Whenever a local change increases the number of internal edges, the change is always accepted. However, when it decreases the solution value, the change is accepted according to the simulated annealing scheme [Ĉerný 1985]. In this scheme, the chance of accepting a solution that is not an improvement is dependent on how bad the change is,

and on a parameter T called temperature. The chance of accepting bad changes decreases over time.

Every solution found during the search process assigns every vertex to exactly one layer. We also guarantee that all layers remain valid during the entire process, i.e. without overlaps. We use three different functions for creating the local changes. The details of these methods are given later.

MERGE Choose a layer L_i randomly. Randomly pick another layer L_j it is connected to. If L_i has no conflict with L_j, a trivial merge is performed. Otherwise, an instance of min-cut max-flow [Ford and Fulkerson 1956] is solved for these two layers only;

MOVE Move x members of layer L_i to neighbouring layer L_j in such a way that the resulting layers remain feasible;

SPLIT Split a layer into two new layers by assigning certain vertices to a set s and other vertices to a set t and solve an instance of min-cut max-flow, separating all vertices in the set s from all the vertices in the set t.

All the neighbourhood operations are defined on one or two layers and always return one or more layers. Whenever the MERGE operation is called, it checks whether the merge is trivial. We say a merge is trivial whenever the two layers L_i and L_j have no overlap. When this is the case, we simply merge the two layers. When the merge is not trivial, we will attempt to redistribute the layers, instead of actually merging the two layers. To accomplish this, we first create the sub-graph G' induced by the members of the layers L_i and L_j. Next, a super source s and a super sink t are added. Edges are added between s and the vertices of layer L_i that overlap vertices of L_j, and vertices of layer L_j that overlap vertices of L_i are connected with edges to t. The capacities of these edges are set to infinity and the weight to one, ensuring that these edges will never be cut. Solving the minimal s-t cut problem for the sub-graph G' will give us a locally optimal redistribution of the vertices in L_i and L_j. If the newly found L'_i and L'_j are the same as L_i and L_j, then the change is rejected.

Note that the MERGE operation will never increase the number of connections. Whenever a trivial merge can be performed, the number of internal edges will increase. When the merge is not trivial, a minimal set of edges needed to separate all overlapping vertices contained within the layers L_i and L_j was found. Since we found the minimal set of edges separating L_i and L_j in the sub-graph, it will always decrease the number of cuts needed, unless that current cut was already minimal. This is a locally optimal solution, but does not guarantee global optimality.

The MOVE operation takes two possibly overlapping layers L_i and L_j that are connected. The operation tries to move x members from L_i to L_j. This allows for both an increase and decrease of the number of cuts in the current solution. It is an important operator since it can move the current solution away from a local optimum.

The SPLIT algorithm takes as input a single layer L_i and splits it into two or more layers. For this we first select sources and sinks at random and connect them to a super source s and super sink t, respectively. Next, we use a minimal s-t-cut algorithm to separate s from t, giving us the cut-set E'. Next, we find the connected components in the sub-graph G' induced by the vertices of L_i, with edges in E' removed from G'. The resulting connected components represent the different layers.

3.3 Height Heuristic

The last method is the Height Heuristic. This heuristic tries to quickly group polygons that are roughly on the same height. The height of a polygon is defined as the distance between the ground plane and the centroid of a polygon. The idea is that such polygons usually correspond to polygons on the same floor of a building. The *cluster height* is the average height of all polygons in that cluster.

The algorithm consists of two distinct phases, called CLUSTER and LOCALMIN. During the CLUSTER phase, polygons are clustered using their height information. We start by creating a list in of $|V|$ clusters, one for each vertex in G, and an empty list out. The clusters in in are sorted on their cluster height. We remove the lowest two clusters K and K' in in, and repeat the following steps until in is empty:

1. If K and K' overlap, we add K to out and make K' the new K. The lowest element in in is removed and becomes K';
2. If K and K' are within a predefined $range$, initially 0, K' is merged into K. The next lowest element in in is removed and becomes the new K';
3. If K and K' do not overlap and are not within the predefined $range$, K is added to out and K and K' are updated as described in step 1. We keep track of the shortest distance between non-overlapping, out-of-range clusters, in a variable $newRange$; $newRange$ is initially ∞. If the difference in cluster height between K and K' is less than $newRange$, we set $newRange$ to this value.

When $newRange = \infty$, we return the clusters in out and create connected components from the height clustered polygons. Otherwise, we set $range = newRange$, and swap the contents of in and out before repeating the procedure described above. We start by merging all clusters that are at exactly the same height, that is, we start with $range$ set to 0.

The clusters found in the CLUSTER phase are first converted to connected layers. These layers are moved towards a local minimum using an adaptation of the MERGE sub-step of local search in the LOCALMIN phase. This operation returns the changed layer(s), as well as all the neighbouring layers of the changed layer(s). During the algorithm, we keep track of layers that can still be changed using the MERGE operation. At the start of the algorithm, this set consists of all the layers, called the $open$-set. In each iteration of the algorithm, we randomly remove one of the members L_i of the $open$-set. Next, we select all layers that are in the $open$-set and that L_i is connected to. We perform the MERGE operation for L_i and each of these layers. If the MERGE increases the internal weight of the connections, the returned layers are added to the $open$-set. This process is repeated until the $open$-set becomes empty. We will always reach this state, since in each iteration of the main loop, only one of three things can happen. First, a trivial merge may be possible for the layers L_i and L_j. Therefore, the number of connections in the current solution must drop, as well as the number of layers. Second, if a trivial merge is not possible, the vertices of the two layers may have been redistributed, as described in Section 3.2. If the number of internal edges of the candidate layers is higher than the number of internal edges currently in L_i and L_j, then we have found a solution that requires fewer connections. Third, it is possible that no changes were performed, and therefore no layers were added to the $open$-set. Since we remove one layer from the $open$-set in each iteration, and only add layers when the number of internal edges increases, the process must be finite.

4 Experiments

We have implemented the algorithms described in Section 3 in C++. For our linear programming solution, we used the SCIP library [Achterberg 2009]. All our experiments were performed on a machine with an Intel Xeon E5-2690 v3 cpu clocked at 2.6 GHz

Table 1: *The tested environments. Column* **T.** *gives the type of environment. V stands for "real" virtual environments, e.g. game levels. R stands for real world environments. Environments of type T are for a specific test. A ✓in column* **Tri.** *means that the environment is triangulated.*

Environ.	T.	Tri.	$\|V\|$	$\|E\|$	$\|O\|$	$\frac{\|O\|}{\|V\|}$
As_oilrig	V	✓	2077	2399	10717	5.16
Halo	V	✓	179	184	346	1.93
Cliffsides	V	✓	748	764	162	0.22
Hexagon	V	✓	2368	2419	20207	8.53
Library	R	✗	298	420	775	2.60
Tower	R	✗	5932	8033	116983	19.72
Station 1	R	✓	206	209	1026	4.98
Station 2	R	✓	82	86	115	1.40
Parking lot	T	✗	59	66	141	2.39
City	T	✗	73001	86148	2415492	33.09
Tower 10	T	✗	5932	8033	116152	19.58
Tower 20	T	✗	5931	8032	105140	17.73
Tower 40	T	✗	5931	8032	76021	12.82

with 32 GB of DDR4 ECC RAM. However, all our experiments only used a single thread. The OS and compiler that were used are Ubuntu 15.10 (64 bit) and g++ version 5.2.1, respectively.

The details of the used environments are given in Table 1. The environments As_oilrig, Library and Parking lot were taken from [Saaltink 2011], Station 1 and Station 2 were provided by Movares, an engineering and consultancy company. The environments Halo, Cliffsides and Hexagon were taken from the Google Sketchup warehouse[1]. The Tower environment was created by the authors, based on a student flat in the Netherlands. The remaining City and different Tower environments were created for specific tests. The City environment was created to test how good the different algorithms scale. The environments with the name *Tower* x are versions of the *Tower* environment that are tilted by x degrees. These environments were included to test the height heuristic to a larger extent, since it is designed with the underlying assumption that the different floors of a building are level.

Besides the size of these environments (which we can see in column $|V|$), there are two other important aspects of the environments. The first one is the ratio $\frac{|O|}{|V|}$, which is an indication of how layered the environment is. The second aspect is what types of geometric primitives were used to model these environments. Besides testing our algorithms on a variation of synthetic environments (the environments of type T), we also tested our algorithms on models of real buildings (R) and on game levels (V).

We also implemented the graph reduction operations described in [Hillebrand et al. 2016]. The reason for this is that we wanted more variation in the underlying WEG. We applied the operations on all the environments except City, reducing the WEGs of this environment simply took too much time. A short description of these operations can be found in Appendix A.

The resulting relative sizes of the WEGs can be seen in Table 2 in Appendix B. We ran the experiments for the algorithms described in Section 3 on both the original WEGs and the reduced WEGs. An individual run was allowed to last no longer than an hour. All experiments were repeated 20 times. When performing local search, one of the three operations was picked uniformly at random. The

[1] https://3dwarehouse.sketchup.com/model.html?id={13c3078fa52d14554b9e177bc9ee06a9, 2ac949d235d65acb46697ff0ff-0b9b2c, 33b2c337108275568c09573a9753f4fd}

Table 2: *The relative change of $|V|$, $|E|$ and $|O|$ of the WEGs after applying the graph reduction rules. The experiments were repeated 20 times and d was set to 1 for* `d-REMOVE`.

Environ.	$\|V\|$	$\|E\|$	$\|O\|$	t (ms)
As_oilrig	0.72	0.76	0.26	4658.15
Halo	0.51	0.53	0.22	13.05
Cliffsides	0.08	0.08	0.14	21.15
Hexagon	0.41	0.40	0.05	2975.75
Library	0.83	0.87	0.39	76.30
Tower	0.96	0.97	0.43	149869.25
Station 1	0.57	0.57	0.15	20.70
Station 2	0.38	0.39	0.19	1.00
Parking lot	0.94	0.95	0.39	1.80
Tower 10	0.98	0.98	0.42	139265.20
Tower 20	0.97	0.98	0.41	144794.40
Tower 40	0.95	0.96	0.40	114833.45

cool down factor was set to 0.9 and the starting temperature was determined using [Ben-Ameur 2004].

5 Results

The results of the experiments are included in Appendix B due to space limitations. For a select number of environments, we have shown their found MLE in Fig. 6. The experiments were designed to test the following:

1. The dependence of running time on the number of vertices, edges and overlaps;
2. The quality of the solutions found by the different algorithms;
3. And finally, the (in)dependence of the algorithm on the actual geometry of the environment.

5.1 Speed dependent on size

We have tested our algorithms on environments with different sizes, as can be seen in Table 3. The height heuristic has proven to be statistically significantly faster, as can be seen in Table 4(a). From the same table, we can also conclude that reducing the WEG also significantly improves the speed of the local search algorithm. However, when we also include the time needed for performing these graph reductions (see Table 2), the local search becomes significantly slower for the As_oilrig, Hexagon and Tower environments.

As was expected, our ILP did not finish within an hour for most environments. It is not always the case that it performed faster on a reduced WEG. For the two samples we have, it was statistically significantly slower ($\alpha = 0.001$) for the parking lot environment, but faster for the Station 2 environment.

5.2 Quality of MLE

The ILP will always give the best result when it is given enough time, and for that reason, we will leave it out of this discussion. For the other algorithms, we refer the reader to Table 4(b). From this table, we can see that the height heuristic is outperformed by local search when we just consider the number of connections. Interestingly, the use of reduced WEGs negatively impacted the results for the height heuristic for the environments Station 1, parking lot, Tower 20 and Tower 40. We think that this is partially due to the higher geometric dependence of this algorithm (see also Section 5.3). As part of the graph reduction algorithms, several

polygons can be placed in fixed groups, resulting in non-planar regions. For local search, it only statistically significantly influenced the results for the Station 1 environment.

5.3 Geometric independence

The Tower 10, 20 and 40 environments were rotated in such a way that they allowed for the same MLE. Unfortunately, both local search and the height heuristic found statistically significant worse solutions for the rotated environments ($\alpha = 0.01$). The influence on local search seems to be less profound however, with only a relative increase of approximately 10 per cent, versus an increase of at least 50 per cent for the height heuristic. This test did not only rotate the environment, since the WEG was generated after rotating the environment. Therefore, a small difference in the found MLEs was to be expected. However, we think that the large decrease of the quality for the height heuristic is partially caused by the underlying assumptions that the different layers are level.

6 Conclusion and future work

In this article, we have presented three different algorithms for finding MLEs with a low number of connections. One of these algorithms, the ILP, is too slow for any practical applications. The other two algorithms, however, can be used to obtain an MLE from a WE. The height heuristic is very suitable for 'traditional' environments, e.g. environments for which the different layers are level. When high quality MLEs are more important than processing time, the local search implementation is a better choice.

When using local search, applying the graph reductions from [Hillebrand et al. 2016] can be beneficial for smaller to medium environments. It is hard to know beforehand if using the reductions will give a performance boost.

These algorithms nicely fit in a pipeline for processing three-dimensional environments for use in games and simulations. The first step of this pipeline is part of our current research. The first results of this research can be seen in Figs. 1(a) and 1(b).

We are also currently exploring the possibility of adding soft-constraints to our methods. One such constraint could be that connections in an MLE should always start and end in obstacles. This constraint is required for generating the navigation mesh described in [van Toll et al. 2011]. For this purpose we are currently exploring triangulating the WE. Another soft-constraint could be preferring short connections over long connections. For instance, wide connections allow for larger visibility between different layers. This can influence the performance of cross-connection nearest neighbour queries that take visibility into account.

In conclusion, we think this research can help speed up currently available algorithms that use MLEs. We also expect that more algorithms that are currently restricted to the plane will be lifted to MLEs as well.

References

ACHTERBERG, T. 2009. Scip: solving constraint integer programs. *Mathematical Programming Computation 1*, 1, 1–41.

BARNHART, C., JOHNSON, E., NEMHAUSER, G., SAVELSBERGH, M., AND VANCE, P. 1998. Branch-and-price: Column generation for solving huge integer programs. *Operations Research 46*, 3, 316–329.

BEN-AMEUR, W. 2004. Computing the initial temperature of simulated annealing. *Computational Optimization and Applications 29*, 3, 369–385.

ĈERNÝ, V. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications 45*, 41–51.

DESAULNIERS, G., DESROSIERS, J., AND SOLOMON, M. 2006. *Column generation*, vol. 5. Springer Science & Business Media.

DEUSDADO, L., FERNANDES, A. R., AND BELO, O. 2008. Path planning for complex 3D multilevel environments. *Proc. 24th Spring Conf. on Computer Graphics*, 187–194.

FORD, L., AND FULKERSON, D. 1956. Solving the transportation problem. *Management Science 3*, 1, 24–32.

HILLEBRAND, A., VAN DEN AKKER, M., GERAERTS, R., AND HOOGEVEEN, H. 2016. Performing multicut on walkable environments. In *10th Annual Int. Conf. on Combinatorial Optimization and Applications, 2016*. To appear.

JIANG, H., XU, W., MAO, T., LI, C., XIA, S., AND WANG, Z. 2009. A semantic environment model for crowd simulation in multilayered complex environment. *ACM Symposium on Virtual Reality Software and Technology*, 2015, 191–198.

LOZANO-PÉREZ, T., AND WESLEY, M. A. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM 22*, 10, 560–570.

OLIVA, R., AND PELECHANO, N. 2013. NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments. *Computers & Graphics 37*, 5, 403–412.

PADBERG, M., AND RINALDI, G. 1991. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM review 33*, 1, 60–100.

PETTRÉ, J., LAUMOND, J.-P., AND THALMANN, D. 2005. A navigation graph for real-time crowd animation on multilayered and uneven terrain. *First Int. Workshop on Crowd Simulation 47*, 2, 81–90.

RODRIGUEZ, S., AND AMATO, N. M. 2011. Roadmap-based level clearing of buildings. In *Lecture Notes in Computer Science*, vol. 7060 LNCS, 340–352.

SAALTINK, W. 2011. *Partitioning polygonal environments into multi-layered environments*. Master's thesis, Utrecht University.

SCHRIJVER, A. 2003. *Combinatorial Optimization - Polyhedra And Efficiency*, vol. 24 of *Algorithms and Combinatorics*. Springer.

SNOOK, G. 2000. Simplified 3D movement and pathfinding using navigation meshes. In *Game Programming Gems*, M. DeLoura, Ed. Charles River Media, 288–304.

VAN TOLL, W., COOK, A., AND GERAERTS, R. 2011. Navigation meshes for realistic multi-layered environments. In *Int. Conf. on Intelligent Robots and Systems, 2011*, 3526–3532.

WHYTE, J., BOUCHLAGHEM, N., THORPE, A., AND MCCAFFER, R. 2000. From cad to virtual reality: modelling approaches, data exchange and interactive 3d building design tools. *Automation in Construction 10*, 1, 43–55.