Graph-based Terrain Generation

Toby Rufinus

A thesis presented for the degree of
Master of Science

June 30, 2018

Supervisors:
Frank Staals
Marc van Kreveld

Game and Media Technology
ICA-4097319

Utrecht University
The Netherlands

Abstract

This thesis introduces a technique for procedurally generating terrains for strategy video games. In our approach, we use graphs to represent regions and corridors in the terrain. We implemented a tool that a game designer can use to define a terrain graph with geometric properties that the maps should have, such as the connectivity structure and physical dimensions of the regions and corridors. Our terrain generator uses a graph layout algorithm to find a terrain graph layout that meets the specifications, and generates a heightmap based on that layout. We tested the effectiveness of our method with various test cases in different experiments, and found that the generated terrains match the designer's intent quite well.

Contents

1 Introduction **3**
1.1 Problem Statement . . . 3
1.2 Contributions . . . 4
1.3 Outline . . . 5

2 Related Work **7**
2.1 Heightmap Generation . . . 7
2.2 Terrain Generation for Strategy Games . . . 8
2.3 Strategy Game Terrain Analysis . . . 9

3 Method **11**
3.1 Overview . . . 11
3.2 Algorithm Input . . . 12
3.3 Constrained Graph Layout . . . 12
3.4 Modifying the Fruchterman–Reingold Layout Method . . . 14
3.5 The Corridor Layout Phase . . . 15
3.6 Improving the Corridors . . . 17
3.7 Creating the Accessibility Map . . . 19
3.8 Using Repulsion Radii to Resolve Node Overlap . . . 20
3.9 Generating the Terrain . . . 21
3.10 Implementation . . . 22

4 Experiments **23**
4.1 Measurements . . . 23
4.2 Test Cases . . . 25
4.3 Performed Experiments . . . 26
4.4 Results . . . 28
4.5 Discussion . . . 35

5 Conclusion **40**
5.1 Summary . . . 40
5.2 Future Work . . . 41

A Test Cases **46**

B Generated Graph Layouts and Accessibility Maps **49**

C Diversity Images **55**

D Generated Terrains **58**

1 Introduction

Terrains are an important component of most strategy video games. They form the battlegrounds upon which virtual wars are waged. Game units—from swordsmen to hovertanks—move around the map, navigating around impassable obstacles like rivers and cliffs, trying to outflank or outrun enemy forces. Barracks and factories are built in flat, open regions to raise and supply armies, protected by bunkers and turrets placed near narrow choke points. Players send out scouts to explore previously unknown terrain, on the lookout for enemy troops and new resources to be gathered.

Making a good map is essential if you want to build a great strategy game. However, making one map is not enough, no matter how well-designed that map is. Most players would lose interest if they had to play on the same map every time. That is why good strategy games have a variety of maps. With gamers constantly demanding more, larger and more detailed maps, it is hard for game designers to keep up. The production of game content has become a bottleneck in the game development process, making it cost ever more time, money and manpower to produce video games [14].

Procedural content generation techniques aim to help out by automating the creation of video game content. A wide range of procedural techniques exist, capable of generating all kinds of content—from the trees and plants in a forest visited in a first-person shooter to the layout of a roguelike game's labyrinthine underground dungeons. Procedurally generated maps have been used in some strategy games, such as the *Civilization* series. However, for many games, current procedural generation techniques are unable to produce terrains that match human-designed maps when it comes to aspects such as gameplay balancing and interestingness [2]. Consequently, most strategy games still use handmade maps.

1.1 Problem Statement

Terrains for strategy games contain *accessible* and *inaccessible* areas. Units move over accessible heightmap cells, and structures can be placed in accessible areas. The parts of the map that are not part of a region or corridor are inaccessible. Ground-based units and structures cannot be inside these areas. They will have to navigate around inaccessible parts of the terrain to reach their destinations.

The accessible areas on the map are all connected, and form a system of open *regions* (which are well suited for constructing bases) connected by narrower *corridors* (which form defensible choke points).

One of the most important requirements that procedurally generated terrains for strategy games must meet is that they should be *balanced*. Each player should have the same chance of winning the game when they begin a match at their designated starting location.

In this thesis, we will investigate the problem of procedurally generating balanced terrains with a clear structure of regions and choke points. There are multiple geometric properties of the map that can influence players' winning chances, which we will list below. A map generator that produces balanced maps should take such properties into account.

In order for a map to be balanced, different players should start in regions of a similar size—it would be unfair if some players had more room to build base structures than others. All players should be provided with equally promising neighboring locations where secondary bases can be constructed, so the non-starting regions should be similarly sized on each player's side of the map, too.

Each player should have equal access to resources and other special objectives on the map. The distances it takes units to travel between player bases and the various regions that contain those resources and objectives should be the same for each player.

Finally, the regions controlled by one player should be just as defensible as the regions of the other players. The number and width of the corridors leading into the regions on each player's side of the map should be the same.

1.2 Contributions

We present a new procedural content generation technique that attempts to tackle the problem of generating balanced terrains. Our approach combines the speed and versatility of automated generation methods with the expertise of game designers, who can take the gameplay considerations listed in the previous subsection into account.

We have implemented a tool that aids game developers in the creation of strategy game maps. The tool allows a designer to quickly and intuitively define the overall layout that a terrain should have. The user of our tool draws a planar graph that represents the high-level structure of the terrain that will be generated. Graph nodes represent regions, and edges stand for corridors. The procedural generation method we will present generates terrains with regions and corridors structured according to the user-drawn graph.

For each node p in the graph, we ask the user to define a radius r_p. For each edge e in the graph, the user should specify a length l_e, a corridor width w_e, and a slack factor s_e.

Our method applies a graph layout algorithm based on the approach described by Dwyer [5] to the user-drawn graph. We also make use of a modified version of the force-directed method introduced by Fruchterman and Reingold [10]. If the layout method is able to find a graph layout without overlapping nodes or intersecting edges, our method ensures that the generated terrains will have a number of geometric properties that match the user's specifications.

Connectivity structure. If there are n nodes in the initially provided graph drawing, there will be n nodes in the generated terrain. There will be a terrain corridor directly connecting the two regions corresponding to two graph nodes p and q if and only if there is an edge connecting two nodes p and q in the initial graph. The terrain's regions and corridors have a connectivity structure equivalent to the user-provided graph embedding.

Region size. The size of each region in the produced terrains is at least as large as the user wanted. It is difficult to define where the exact boundaries between open regions and corridors lie in the generated terrains, and humans do not always subdivide terrains in the same way (see for example the study by Perkins [25]). Therefore, we define the size of a region to be the distance from its center to the nearest inaccessible obstacle. Formally, for each region P with corresponding graph node p with radius r_p, we define the position of the center of that region, $\mathbf{P}$, to be equal to the position $\mathbf{p}$ of the node, p. For each region in the generated terrain, the following property holds:

$$d_t(\mathbf{P}, v) \geq r_p$$

where v is the inaccessible point on the map nearest to $\mathbf{P}$, and d_t indicates the length of the shortest path that ground-based game units traveling only over accessible areas would take to travel between two points.

Corridor width. For each initial edge e, the width of the corresponding terrain corridor is at least as large as the user-defined value w_e. Corridors are bounded by two walls of inaccessible obstacles. If we define the medial axis A of the corridor as the set of all points in that corridor where the distance to the nearest points on both corridor walls is equal, then the following property holds:

$$\forall a \in A : d_t(a, v) \geq w_e$$

where v is any of the inaccessible points on the map nearest to a, which is a point on the medial axis. There is also at least one point a' on the corridor's medial axis where $d_t(a', v) = w_e$.

Inter-regional distance. If a pair of nodes (p, q) is connected by an edge e, the corresponding regions P and Q (with region center positions $\mathbf{P}$ and $\mathbf{Q}$) are directly connected by a corridor. Our method attempts to generate terrains with the following property:

$$d_t(P, Q) = l_e \cdot s_e$$

where l_e is the user-specified edge length and s_e is the edge's slack factor, also set by the user. Unlike the previously listed properties, this property is only approximated; it is not guaranteed to be present.

As we mentioned earlier, if the obtained graph layout has overlapping nodes or intersecting edges, these properties will no longer be true for all regions and corridors.

These geometric properties affect the gameplay balance. If the designer creates graph with balanced specifications (ensuring, for instance, that the distance between all players' starting regions is equal), and the input is reasonable (meaning it is not impossible to place all nodes on the map without overlap), our method produces reasonably balanced terrains, since it takes the mentioned properties into account.

We experimentally evaluated our approach to find out how well the geometric properties of the produced terrains match the properties that were specified by the designer. Since we focus on the generation of balanced maps, the ten test cases used as input in our experiments, which are based on *StarCraft II* ladder maps, are symmetric and perfectly balanced. However, it should be noted that our method can be used to create a wide range of gameplay scenarios—symmetric or asymmetric, balanced or imbalanced.

We found that our algorithm was able to find graph layouts without overlapping nodes or edge crossings—and therefore realizing the first three of the mentioned geometric properties—for a wide range of tested values for the corridor widths (we tested values between 5 and 25; for comparison, the total size of our generated maps is 513×513 units). Lower values for the slack factor (up to 1.4) resulted in high-quality layouts, but high slack factors were found to be problematic, particularly for denser graphs; in those cases, we cannot guarantee that all user-specified constraints are satisfied.

Finally, we found that the length of the paths between different regions in the produced terrains are on average about 80% of what the designer specified. It should be easy to correct for this factor in the graph design tool, resulting in paths whose length more closely matches the designer's intentions. We believe the resulting terrains would be suitable for casual gameplay.

1.3 Outline

The remainder of this thesis is structured as follows. In Section 2, we give an overview of prior academic work related to this thesis. We review different types of generic procedural terrain generation techniques, strategy game map generators, and strategic map analysis methods for strategy games. Section 3 describes our terrain generation algorithm. The experiments we performed to evaluate our algorithm, as well as the obtained results, are described in Section 4. We conclude with a summary and suggestions for future work in Section 5.

Appendix A contains a listing of all the test cases used in our experiments. Appendix B shows a selection of generated graph layouts, along with the corresponding accessibility maps. A number of diversity images are shown in Appendix C. Finally, Appendix D contains renderings of some of our generated terrains.

2 Related Work

Procedural content generation techniques have been used to automatically generate a wide variety of content for video games. Examples of procedurally generated game content include textures, vegetation, puzzles, stories, levels, and more. See the book by Shaker et al. [27] and the survey by Hendrikx et al. [14] for an overview of procedural content generation methods for games.

The methods that can be used to procedurally generate outdoor game environments are particularly relevant to this thesis. In video games, such environments are often based on a grid data structure. Two-dimensional maps are usually divided into tiles of varying terrain types, while three-dimensional maps are typically based on heightmaps—two-dimensional grids of terrain elevation values.

In Section 2.1, we will look at various methods for generating random heightmaps. These can be used to generate landscapes for many different applications, including but not limited to video games. In Section 2.2, we examine procedural techniques that have been specifically designed to generate video game maps, and take gameplay concerns into account. Finally, we look at some terrain analysis techniques used in game AI agents in Section 2.3.

2.1 Heightmap Generation

Noise functions are often used to generate elevation values or gradient vectors on a course grid, which are then interpolated to compute heightmap values; the method by Perlin [26] is a well-known example. Several levels of noise can be rescaled and added together to create natural-looking mountainous terrains, like in the technique presented by Musgrave et al. [22]. The book by Ebert et al. [6] provides an overview of noise-based procedural techniques.

Another class of heightmap generation algorithms starts off by generating a low-resolution heightmap. This map is then iteratively subdivided, with finer, randomized details being added in each subdivision step. See for example the pioneering papers by Fournier et al. [8] and Miller [21].

The aforementioned procedural generation methods are able to produce terrains that have a similar visual appearance to natural landscapes. This is fine for games such as flight simulators, where the terrain merely serves as a backdrop and has little influence on gameplay. However, for many other kinds of games, such as first-person shooters or strategy games, the layout of the terrain greatly affects the player's movements and actions. Generating terrains that simply look natural is not good enough for such games; we would like the the terrain generators to be able to handle more strict constraints.

For instance, it is usually desirable that strategy game maps are balanced. A terrain generator that created randomized terrains without regard for gameplay concerns would be unlikely to produce balanced maps, which makes it unsuitable for generating maps for competitive strategy games.

Some procedural methods offer the user more control over the placement of mountains, valleys, and other terrain features. These techniques can be used by a game designer to craft terrains that are suited for the game they are developing. The designer could, for example, place terrain features symmetrically such that the maps are balanced.

Stachniak and Stuerzlinger [29] allow the user to define constraints via mask images. A local search algorithm is used to find a set of deformation operations that are applied to a randomly generated base terrain to fit the provided constraints.

Zhou et al. [35] allow the user to sketch a map of large-scale terrain features, such as mountain ridges. Patches of terrain are then extracted from a user-provided heightmap and placed in positions that match the features sketched by the user. This does not provide the user with control over small details in the terrain, however. By contrast, the category of sketch-based generation techniques does allow the user to determine the precise placement of terrain features. This category—which includes methods by Gain et al. [11], Hnaidi et al. [16], and Bernhardt et al. [3]—consists of techniques that generate terrains that closely follow 3D curves sketched by the user.

A downside of methods that heavily depend on the user to define what the terrain should look like is that the use of such methods is rather labor-intensive for the game designer. This is especially true if the designer has to take gameplay balancing into account; ensuring that the distances between all player bases are equal, for example, requires a lot of painstaking tweaking. Additionally, since there are so many user-specified constraints that define the exact placement of terrain features, the diversity of the output of this class of terrain generators is also rather limited.

2.2 Terrain Generation for Strategy Games

There are some procedural methods that are specifically geared towards generating terrains for strategy games. Olsen [24] combined a noise-based terrain with hills with adjustable heights in a terrain generator that creates terrains that can be used in strategy games. The generator aims to generate terrains that meet two playability criteria: the heightmap cells with a slope small enough to allow for unit movements must form a large connected area, and there should also be enough flat area of sufficient size to allow for the placement of buildings.

Another procedural generation technique that ensures generated terrains have enough accessible area to be used in a strategy game was invented by Frade et al. [9]. They used an evolutionary algorithm to evolve mathematical expressions that define the elevation values of heightmap cells. The terrains generated by this technique are filled with smooth, curving features and geometric shapes and have a distinct, otherworldly look. They have a lot of accessible area, while there is simultaneously a lot of inaccessible terrain forming obstacles for units to move around.

The previous methods do not consider the balance of the generated maps. Strategy game matches played on the produced terrains could be very unfair. The method by Togelius et al. [31], on the other hand, does aim to produce fair maps. This method makes use of an evolutionary algorithm. It optimizes the placement of player bases, resource locations, and Gaussian-distribution-shaped mountains on a heightmap such that games played on the map would be fair for all players. They used various fitness functions to produce balanced maps, favoring terrain properties such as equal distances between all player bases, and equal distances between each player's base and the nearest resource location.

Additionally, Togelius et al. generated 2D tile-based maps for the strategy game *StarCraft* using similar fitness functions, evolving the locations of player bases, resources, and patches of inaccessible tiles.

Another method by Mahlmann et al. [20] generates 2D maps for a simpler strategy game, *Dune 2*, using an evolutionary algorithm. The placement of patches of different types of terrain (including the player starting zones) are evolved to obtain random two-player maps that are fair for both players.

Uriarte and Ontañón [33] generated symmetric, balanced *StarCraft* maps. They created terrain regions based on the Voronoi diagram of random 2D point sets, complete with game resources and player starting positions. According to the authors' metrics, the balance is comparable to real tournament maps.

Barros and Togelius [2] used real-world terrain data and resource maps to produce maps for the strategy game *Civilization*. The player starting positions were evolved with an evolutionary algorithm to find maps where the distance between player base and resources was equal for each player.

Liapis et al. [19] created a tool that allows the user to make a sketch of a strategy game map on a small 2D grid. Each tile on the grid can be passable, impassable, a player base, or contain resources. The tool evaluates the sketched map using various metrics, such as resource safety (the number of resources that lie very close to a single base) and an exploration score (determining how hard it is for enemies to find each base). These allow the user to see how balanced their current sketch is. The tool also provides suggested alternative or improved maps, evolved using a genetic algorithm, and can generate a detailed, high-resolution map based on the user's final sketch.

Smith and Mateas [28] describe how answer set programming is used in the real-time strategy game *Warzone 2100* to generate maps. The user defines a number of desired properties up front that the maps should have. For instance, the distance between player bases and resources should be maximized, and different player bases should be equally well protected by surrounding cliffs. An answer set solver then searches for the optimal terrain elevation values and base and resource positions. Additionally, answer set programming was used to optimize the placement of buildings inside player bases.

2.3 Strategy Game Terrain Analysis

Procedural content generation is not the only area of research interest related to strategy games. A number of papers describe techniques for analyzing terrains; such techniques are often used in game AI agents. AI players in a strategy game need to know in which locations to build their bases, where to place defenses and retrieve resources, and where to attack the enemy. There are various techniques, such as influence maps [32] and potential fields [12], that allow AI agents to perform spatial reasoning. The paper by Lara-Cabrera et al. [18] provides an overview of such methods.

Many such techniques depend on game-specific information about dynamically changing and often only partially-known game state, such as the position of enemy forces, or the type and location of resources. These methods do not fall inside the scope of this section. We will only look at techniques that make a strategic analysis of the base terrain itself—a tile-based map or heightmap with passable and impassable areas.

Forbus et al. [7] computed a Voronoi diagram of a terrain's impassable obstacles, with each Voronoi cell containing all points that are closest to one particular terrain obstacle. Regions of free space, which lie around Voronoi vertices, are then identified. The traversable areas that do not lie inside free regions are marked as corridors. This subdivision of the terrain into free regions and corridors can be used by an AI agent to make decisions. For instance, when a new base should be constructed, an agent can choose a large free region with enough room to contain various structures.

Higgins [15] presented a technique for identifying choke points. An influence aura around each obstacle is grown outwards. The points where two different auras meet are marked as choke points. Choke points are good places for an AI agent to place defenses, and are likely locations where enemy units may lie in wait.

A similar technique that uses expanding auras to detect choke points and regions was introduced by Obelleiro et al. [23]. In addition to computing auras, they calculated a large number of paths between random points on the terrain. This pathfinding data was used to discard false choke points in isolated corners of the map (those choke points had few paths crossing through them) and sort the remaining choke points based on strategic importance (the more paths cross through a choke point, the more important it is).

The map decomposition method by Halldórsson and Björnsson [13], which can identify regions and choke points, is similar to the expanding obstacle auras used by Obelleiro et al. and Higgins. The difference is that this algorithm simulates the map being flooded with water while keeping track of the rising water level. Essentially, auras are grown from the centers of accessible areas, rather than around inaccessible obstacles.

Perkins [25] described how most aforementioned choke point detection methods fail to detect choke points whose two walls are part of the same obstacle. Such choke points form the entrance to a region that has no other entrances; that region would be cut off from the test of the map if that choke point were to be walled off. The method by Perkins, on the other hand, can detect that type of choke point, as well as the usual kind where the choke point walls belong to two different obstacles. Perkins computed the Voronoi diagram of the line segments that form the boundaries of impassable polygonal obstacles. Next, this Voronoi diagram was pruned, forming a graph with nodes that are then marked as belonging to regions and choke points. Uriarte and Ontañón [34] further improved this method, making it more robust and efficient.

The algorithm by Bidakaew et al. [4] finds the medial axis of a map's accessible areas. This medial axis is then used to detect choke points. This method can detect three different types of choke point: narrow openings between two different obstacles, narrow openings between two parts of the same object, and area choke points, which are narrow corridors rather than a single point.

3 Method

A large part of gameplay balancing concerns the placement and dimensions of regions and choke points in the terrain. As we have seen in the previous section, strategy game AIs often make use of this information. Most map generator algorithms, however, do not explicitly take this into account; our method, which aims to procedurally generate balanced strategy game maps, is one of the first.

We implemented a tool that allows a game designer to specify how many regions there should be, and how large they would like each region to be. They can also define which regions should be connected by corridors, how wide each corridor should be, and the distance units have to travel between connected regions. With this tool, the designer can craft a specification that describes a balanced terrain. They can ensure that each player's starting region has the same size, that the distance between all player bases is the same, that choke points have the same width on each player's side of the map, and so on.

Our terrain generation method will attempt to generate terrains that match the designer's specifications, while preventing different regions and corridors from overlapping with each other. If they did overlap, a direct connection between different areas of the map is introduced at the point of overlap, while those different areas should have been separated. This would result in game units being able to take shortcuts, reaching distant locations much faster than intended, which would upset the game balance.

In Section 4, we evaluate how well our generator performs based on the results of several experiments. In the rest of this section, we will describe the terrain generation method we implemented.

3.1 Overview

Our method accepts a graph drawing as input, drawn by a game designer. In this graph, nodes represent terrain regions and edges represent corridors. The designer assigns various properties to the nodes and edges, such as node sizes and edge lengths, that affect the map's balance; see Section 3.2.

In the *initial layout phase*, the terrain generator takes the designer's graph drawing and applies a graph layout algorithm that makes use of constraint projections, first presented by Dwyer [5]. The layout method is used to compute a graph layout with edge lengths that match the designer's intentions as closely as possible, without any overlapping nodes. This is described in Section 3.3.

One component of our layout method is a modified version of the Fruchterman–Reingold force-directed graph layout algorithm, described in Section 3.4. Attractive and repulsive forces between all connected pairs of nodes move the nodes around in an attempt to give all edges the desired length specified in the input. Additionally, repulsive forces force apart overlapping nodes. These forces work in conjunction with the aforementioned constraint projections.

The *corridor layout phase* comes after the initial layout phase. In this phase, chains of corridor nodes, connected by corridor edges, are added to the graph. These chains represent space in the terrain that will be occupied by corridors. Another run of the graph layout algorithm finds good positions for the corridor nodes such that they do not overlap each other and each corridor node touches the neighboring nodes in its chain. This process is described in Section 3.5.

A third kind of force, the stiffness force, was added to the force model to increase the angles between incident corridor edges in each chain. This straightens the corridors, making for larger, smoother bends. See Section 3.6 for more details.

After the graph layout method has converged (or it has been terminated after reaching the maximum number of iterations), an accessibility map is created; see Section 3.7. This two-dimensional grid contains Boolean values that determine whether the terrain in that cell should be accessible or not. All accessibility map cells that are covered by a graph node's circular shape, as well as all cells that fall inside the convex hull of two nodes that are connected by a corridor edge, are marked as accessible. All other map cells are marked as inaccessible.

An important component of our algorithm that improves the quality of the created accessibility maps are the node repulsion radii. These radii, which are larger than the nodes' regular radii, are computed for every node after the corridor node chains are added. They are used to move overlapping nodes apart, preventing them from overlapping with the convex hull of other pairs of nodes that are connected by a corridor edge. This prevents corridors from overlapping. A detailed description is given in Section 3.8.

Section 3.9 explains how the accessibility map is used to generate a heightmap that describes a three-dimensional terrain. A 3D terrain mesh, ready to be used as the base terrain in a strategy game, can be generated based on the heightmap.

Finally, Section 3.10 describes various details that are specific to our implementation of the described terrain generation method.

3.2 Algorithm Input

Our terrain generation method takes a user-provided embedding of a weighted planar graph $G = (V, E, w_v, w_e)$ as input. V is the set of nodes, and E is the set of edges; all edges are straight line segments. The nodes and edges provided by the user in the graph drawing are called *initial nodes* and *initial edges*. This graph drawing is an abstract representation of the terrain that will be generated; see Figure 1 for an example.

For each initial node, a circular-shaped region of open, accessible space will be placed in the generated terrain's heightmap, centered at that node's final position. $w_v \colon V \to \mathbb{R}_{>0}$ maps nodes to their radius; a positive value that determines how wide that node's region in the terrain should be. The designer can use this to influence gameplay, since larger regions offer more space in a strategy game for players to construct buildings and place defenses than smaller regions.

A corridor of traversable terrain which connects two regions will be generated for each initial edge. $w_e \colon E \to \mathbb{R}_{>0}$ associates a desired positive length with each edge. The algorithm will reposition connected nodes in an attempt to make the distance between them equal to the desired edge length. The value the user defines is taken to be the distance between the boundary of two nodes. By constraining this property to positive values, we ensure two regions will never overlap when the desired distance between them is achieved. By tweaking the assigned edge lengths, the user can change the distances between terrain regions. The algorithm cannot set edges to their desired length if the specification contains impossible situations (such as a cycle of three edges, whose lengths violate the triangle inequality), so such situations should be avoided.

3.3 Constrained Graph Layout

The terrain generator starts off by performing a two-phase graph layout algorithm. It starts with the *initial layout phase*, which operates on the input graph and is described in the next two sections. After that, the graph is extended with new nodes and edges, and a modified version of the layout method then finds a good layout for that extended graph; see Section 3.5.

The goal of the initial layout phase is to produce a graph drawing without any overlapping nodes or crossing edges, where each initial edge has a length equal to the user's specifications. This is important because the nodes represent regions in the terrain. If the terrain is to be used in a strategy game, the distances between regions need to match up with the designer's intent.

For example, it would be unfair if one player had to travel a much shorter distance from their starting region to the nearest resource location than other players. The map designer could set the desired edge lengths between starting regions and nearby resource regions to be equal for each player, but this is only useful if the right layout method is chosen. A layout algorithm that produces wildly diverging edge lengths could result in an imbalanced map where one player has significant advantages.

We chose to apply a constrained graph layout algorithm based on the method introduced by Dwyer [5]. This algorithm supports graph layout subject to Euclidean distance constraints between nodes.

We generate a distance constraint between each pair of nodes that is connected by an initial edge. Specifically, for the positions $\mathbf{p}$ and $\mathbf{q}$ of two nodes connected by an edge with desired length k, the following constraint is generated:

$$|\mathbf{p} - \mathbf{q}| = k$$

Dwyer's method can easily incorporate non-overlap constraints for circular nodes. For each pair of nodes p and q with radii r_p and r_q, we generate the following constraint:

$$|\mathbf{p} - \mathbf{q}| \geq r_p + r_q$$

The layout method considers each constraint in turn, one at a time. For every constraint, the two involved nodes are moved by the same distance, chosen to be as short as possible to satisfy that constraint. This adjustment of the node positions is called a *constraint projection.* For the edge length constraint, for example, a constraint projection moves a pair of nodes by the smallest vector $\mathbf{r}$ that satisfies the following:

$$|(\mathbf{p} - \mathbf{r}) - (\mathbf{q} + \mathbf{r})| = k$$

Like Dwyer, our layout algorithm is performed in three stages. First, an iterative graph layout method without any constraints is executed to obtain an initial layout. This layout method is repeated until the nodes have converged upon their final positions, or a maximum number of iterations is reached.

Next, a number of layout iterations are performed with added distance constraint projections that are performed at the end of every layout iteration; this is the second stage. Finally, in the third stage, non-overlap constraints are added, which are also projected at the end of every layout iteration.

Since each distance constraint is projected in turn, a projection can violate constraints that were solved by earlier projections; however, Dwyer found that simply cycling over all constraints several times per layout iteration, projecting them one by one, gives good results. We perform 50 constraint projection cycles after each layout iteration in the second stage. In the third stage, with the added non-overlap constraints, 10 constraint projection cycles are performed at the end of each layout iteration. In our implementation, we cycle through all nodes and edges in the order they were added to the graph when performing both constraint projections and force-based graph layout iterations.

3.4 Modifying the Fruchterman–Reingold Layout Method

The method by Dwyer requires a graph layout algorithm that performs layout iterations. We chose to use a force-directed graph drawing algorithm with a force model based on the method by Fruchterman and Reingold [10].

In the force-directed method by Fruchterman and Reingold, each node in the graph applies an attractive force f_a to the other graph nodes to which it is connected via a graph edge. This attractive force pulls connected nodes closer together. Simultaneously, repulsive forces f_r act between all pairs of nodes (whether those nodes are connected or not), pushing nodes apart. The forces are defined as follows:

$$\begin{aligned} f_a &= \frac{d^2}{k} \\ f_r &= \frac{-k^2}{d} \end{aligned}$$

d is the distance between the two nodes for which the forces are being calculated; k is the optimal distance between nodes set by the user. Crucially, the attractive and repulsive forces cancel out at a distance of exactly k, which is why this force model was chosen for our graph layout method.

Unfortunately, the original Fruchterman–Reingold method only guarantees that the distance between nodes is k for a single, connected pair of nodes. Once more nodes are added to the graph, the additional nodes' repulsive forces will result in a total force sum that does not cancel out at a distance of k—the k parameter merely serves as a hint to the algorithm, not as a hard constraint.

As a result, using a plain, unmodified Fruchterman–Reingold method for the layout iterations in Dwyer's constraint projection algorithm results in blown-up graphs whose nodes are pushed away from the center, towards the edges of the surface upon which the graph is drawn. This is the *peripheral effect* described by Hu [17]: nodes close to the center tend to be further apart than nodes near the boundary, even if the desired distance between nodes provided to the layout method is constant.

In one example, Hu describes how edge lengths varied between about 1.5 and 4.1, while k was set to 1. While Dwyer's constraint projections could restore the proper distances between nodes, starting with a blown-up graph could result in a different graph embedding than the user drew and often introduces edge crossings, which is unacceptable. For our purposes we need a layout method that preserves the user's graph embedding and produces edge lengths that closely match k.

In order to remedy this, we opted to modify the Fruchterman–Reingold algorithm. Specifically, we chose to compute the attractive and repulsive forces for each pair of connected nodes without taking other nodes' repulsive forces into account. In other words, rather than repulsive forces acting on *all* node pairs, there are now repulsive forces pushing each pair of *connected* nodes apart—just like the attractive forces pull each pair of connected nodes together.

Fruchterman and Reingold chose a value of k that results in a uniform distribution of nodes. That is not desired here; instead, the input graph edges contain desired lengths, which are used as values for k. Note that k was a global constant in the original algorithm, while it varies per graph edge here.

Additionally, if two non-adjacent nodes p and q overlap, a small repulsive force with $k = r_p + r_q$ is applied to the two nodes to remove the overlap. Since they are not connected by an edge, the repulsive force is removed once p and q cease to overlap.

During the graph layout algorithm, the positions of the nodes are constrained, forcing the entire node to remain inside the rectangular surface that contains the graph drawing.

The result of the modified Fruchterman–Reingold algorithm is a graph whose nodes are not pushed apart as strongly as they would have been if the unmodified algorithm was used. The resulting layout will not have uniformly distributed nodes (which is a property that the original Fruchterman–Reingold algorithm tries to achieve), but the distance between each pair of connected nodes should closely match the desired edge length as specified by the user, with few overlapping nodes.

The force-directed layout method only makes a best effort to produce a layout that approximately satisfies the distance and non-overlap constraints; it cannot make any guarantees. The layout method is therefore combined with the constraint projections by Dwyer to better meet the constraints. If the user gives a realistic input, without constraints that are impossible to satisfy, the layout algorithm should produce a graph layout with no crossing edges, no overlapping nodes, and edges whose lengths closely match the desired edge lengths given by the user.

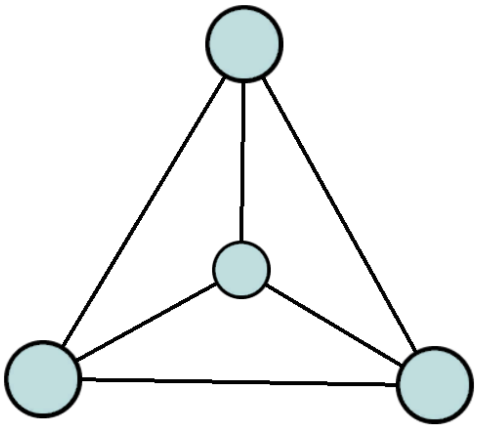
Figure 1: An example of a graph after the initial layout phase.

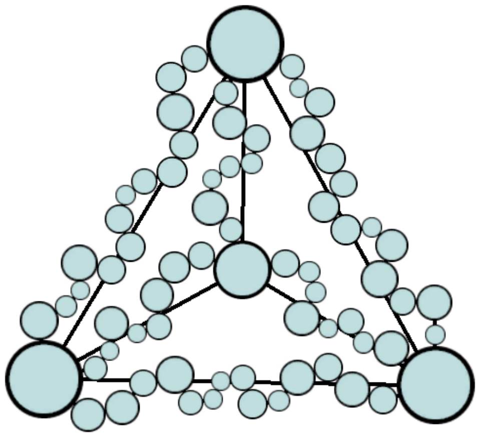
Figure 2: The graph from Figure 1 after the corridor layout phase.

3.5 The Corridor Layout Phase

After the graph layout method has produced a good layout for the initial graph, the approximate location of open regions in the terrain is known. Terrain corridors must be placed for each initially drawn graph edge, connecting the various regions. In addition to the desired length of an initial edge, there are two more user-defined properties per edge that allow one to influence the corridors that will be generated.

First of all, each initial edge has a width property, determining the width of the corridor in the generated terrain. In a strategy game, it is easier for large troop formations and large units (such as tanks) to move through wide corridors than narrow corridors. The width is typically smaller than the diameter of the nodes on either end of the edge, resulting in a relatively narrow corridor between two wider regions.

Secondly, the length of the corridor can be changed too. As mentioned before, each initial edge has a desired length property that determines the Euclidean distance between two regions. The longer the desired edge length, the longer the length of the corridor. However, in a natural terrain, corridors connecting regions are not perfectly straight lines—they tend to meander. The more a corridor twists and turns, the longer it would take for units to travel from one end of the corridor to the other.

A user-specified *slack factor* property is assigned to each initial graph edge to determine how twisting the corresponding terrain corridor will be. Like a rope, little slack results in a relatively tight, straight path from beginning to end, while a lot of slack results in more bends in the path, increasing the travel time between the two ends of the corridor.

Given two regions P and Q that are connected by a corridor, with $\mathbf{P}$ and $\mathbf{Q}$ being the positions of the region centers and r_P and r_Q the radii of the regions, the slack factor property of the edge between P and Q influences the length of the connecting corridor—and therefore the time it would take units to travel between $\mathbf{P}$ and $\mathbf{Q}$ in a strategy game—in the following way:

$$\begin{aligned}&\text{distance}(\mathbf{P}, \mathbf{Q})\\ =&\,\text{slack-factor}(\text{initial-edge}(P, Q)) \cdot \text{length}(\text{initial-edge}(P, Q))\\ =&\,r_P + r_Q + \text{length}(\text{corridor}(P, Q))\end{aligned}$$

After the initial layout phase has concluded, a chain of *corridor nodes*, each one connected to the next in the chain by a *corridor edge*, is inserted in the graph for each initial edge connecting two initial nodes p and q. The first corridor node in the chain is connected by a corridor edge to p, and the chain's last corridor node is connected to q. Each chain of corridor nodes determines the position of one terrain corridor. For an example, see Figure 2.

Every corridor node gets a diameter based on the corresponding initial edge's width w that was set by the user. In order to generate natural-looking corridors, we try to select random corridor node diameters between w and $2w$ when possible. The upper limit of $2w$ was chosen because the resulting corridors' visual appearance seemed acceptable to us; other implementation choices are possible too. In edge cases where the desired corridor length is very wide (more than half as wide as the smaller of the two regions being connected), we additionally constrain the corridors to be no wider than the two regions between which the corridor runs.

Corridor nodes with random diameters are added to the chain until the sum of the corridor node diameters in the chain is at least length(corridor(P, Q)). We also ensure that at least one of the nodes in each chain has the minimum diameter w. This is important for gameplay purposes; if the narrowest point in the corridor were wider than w, it would be too easy to move through the corridor, violating the designer's specifications. Finally, the list of corridor nodes in the chain are randomly shuffled to avoid any directional artifacts.

The corridor nodes are placed in between the regions P and Q. Specifically, they are placed at uniform intervals on the part of the edge between P and Q that lies outside the boundaries of P and Q. In this initial position, the corridor nodes in a chain overlap each other. The layout algorithm will quickly push the nodes apart, removing any overlap. If all corridor nodes have an initial position exactly on the straight line between two region centers, an edge case occurs in which the nodes do not push each other outwards, but only along the edge, which causes node overlap to remain present. We add a very small random offset to the initially computed position of each node; this causes the nodes to be correctly separated during the graph layout algorithm.

All corridor edges between two nodes p and q in the chain have a desired length of $r_p + r_q$; we want the nodes in the corridor chain to touch each other without overlap.

When a chain of corridor nodes has been inserted for each initial graph edge, the three-stage layout method by Dwyer is run again. This results in a final graph layout with a chain of touching corridor nodes of the appropriate length running between each connected pair of initial graph nodes.

If the input constraints are hard to satisfy, the constraint projections can introduce edge crossings in the corridor layout phase. These crossings may or may not be resolved in later layout iterations. If we detect edge crossings in 50 consecutive layout iterations, we remove all corridor nodes and edges and create new corridors, restarting the corridor layout phase. It should be noted that edge crossings involving initial edges do not matter and are ignored. Crossings between two corridor edges, on the other hand, result in corridors that cross each other in the generated terrain, which should be avoided.

3.6 Improving the Corridors

The graph layout method we described thus far tends to form zig-zagging corridor node chains, with a large number of sharp turns along their paths. This does not look very much like corridors in naturally formed terrains. The zig-zagging artifacts can also result in a significantly shortened travel time between regions; see Figure 3. We added a third force, the *stiffness force*, to the force-directed layout model alongside the attractive and repulsive forces to improve the appearance of the corridors. The stiffness forces straighten the corridor node chains, causing corridors to smoothly curve rather than zig-zag. This also causes the travel time through a corridor to more closely match the game designer's specifications; see Figure 4.

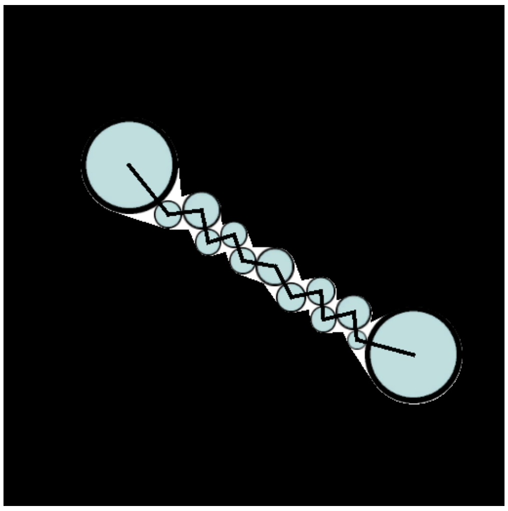
Figure 3: A chain of corridor nodes, with the corridor edges drawn on top. This layout was produced without stiffness forces. The corridor edges form an unnatural zig-zagging chain, and units traveling through the corridor would be able to travel in a straight line, making the travel time far shorter than intended.

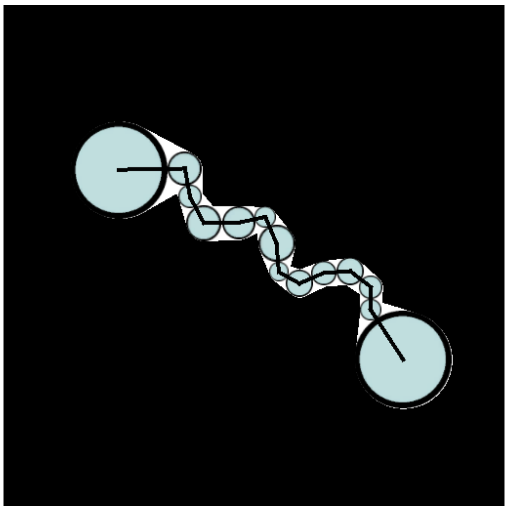
Figure 4: A chain of corridor nodes, with the corridor edges drawn on top. Unlike the situation from Figure 3, this layout was produced with stiffness forces. The chain of corridor edges curves smoothly, and units would have to follow the bends as well, making the travel time much closer to what the designer had in mind.

In the corridor layout phase, the stiffness forces are calculated and accumulated for each node, just like the other forces; a node's displacement in the force-directed layout method is based on the sum of all three types of forces. In the initial layout phase, there are no stiffness forces.

For every corridor node p, the stiffness forces aim to push apart the two neighboring nodes q and r; see Figure 5. In order to determine the stiffness force that p applies to its neighbors, the angle θ between the two corridor edges incident to p is calculated; $\theta = \angle\mathbf{qpr}$.

The stiffness force f_s acting on q caused by node p is perpendicular to **pq**, the corridor edge connecting p and q. The force points away from r. Similarly, p also causes a stiffness force to act on r, which is perpendicular to **pr** and points away from q. See Figure 5. Both stiffness forces have the same magnitude, described by the following formula:

$$|f_s| = c_{stiffness} \cdot \text{stiffness}(p) \cdot \frac{\pi - \theta}{\pi}$$

θ is measured in radians here. The stiffness force is largest when θ is small; it linearly decreases to zero as θ increases to the maximum angle of π.

Each node p has a stiffness constant stiffness(p) that determines how hard its adjacent corridor nodes are pushed apart. We opted to give each node a random value between 0 and 1 when that node is created. The effect of giving different corridor nodes different stiffness values is that the curvature varies along the path of the corridor. The corridor will have sharper turns when it passes through corridor nodes with a small stiffness value, since those nodes are unable to push their neighbors very far apart. Smoother bends are encountered where the corridor nodes have a high stiffness value, since those nodes pushed their neighbors far apart, straightening the path of the corridor.

Finally, a global constant $c_{stiffness}$ is used to tweak the overall appearance of the generated corridors. A large value for this constant results in smoothly curving corridors, while a small value results in more and sharper turns.

Since most corridor nodes p are connected to two adjacent corridor nodes, both neighbors will apply a stiffness force to p. These two stiffness force vectors are added up (together with the attractive and repulsive force vectors) to calculate p's displacement vector during the layout iterations in the corridor layout phase.

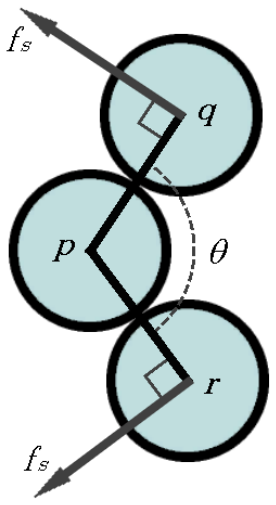

Figure 5: Corridor node p causes stiffness forces f_s to act on its neighboring corridor nodes, q and r. The magnitude of f_s depends on the indicated angle, θ. Note that q and r would also both cause a stiffness force to act on p; those vectors are not displayed here.

3.7 Creating the Accessibility Map

Once the layout of the graph with added corridor nodes has been computed, an *accessibility map* is created. This map indicates which parts of the plane are accessible to game units and structures, and which parts are inaccessible. In our implementation, the accessibility map is a two-dimensional grid that covers the surface which contains the graph drawing. We chose to use a grid-based representation because, as will be seen in Section 3.9, it is straightforward to convert a grid-based accessibility map into a heightmap. In our particular implementation, this heightmap can then be used to generate a 3D terrain with little effort; see Section 3.10. We found that a grid with a few hundred thousand cells offered a high enough resolution to produce moderate-sized heightmaps and terrains for strategy games. However, it would also be possible to use a vector-based accessibility map, which would have an effectively unlimited resolution, should one so desire.

Each grid cell contains a Boolean value that determines whether the terrain in that cell is accessible or not. The terrain generator will ensure that all map cells marked as accessible will indeed be reachable by game units that start from some known accessible start cell, such as a player's starting position. All cells marked as inaccessible, on the other hand, cannot be reached by game units from any accessible position.

All map cells whose center lies within a node's circular region are marked as accessible. For every pair of nodes that is connected by a corridor edge, the convex hull of the two nodes' circles is constructed. All accessibility map cells whose center that lies inside such a convex hull are also marked as accessible. Every cell whose center does not lie inside any of the node circles nor any of the constructed convex hulls is inaccessible. Figure 6 shows an example.

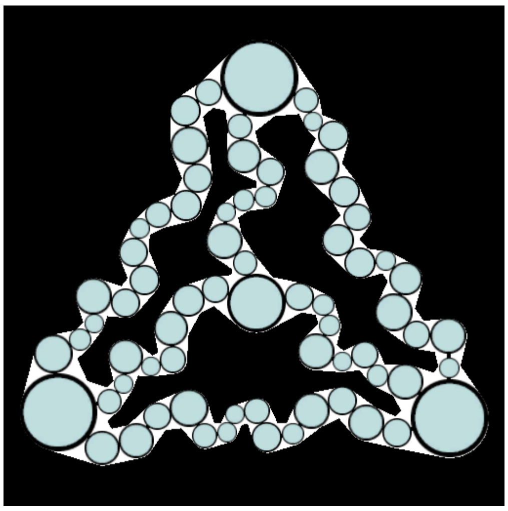
Figure 6: The graph from Figure 2 on top of its accessibility map. Black pixels denote inaccessible cells. Cells that are white or underneath nodes are accessible. The initial graph edges have been hidden for clarity.

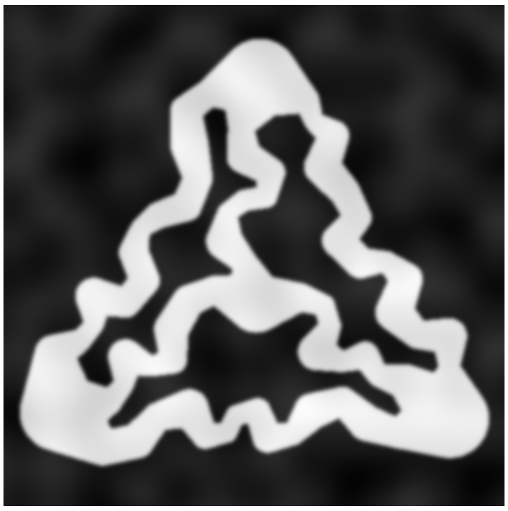
Figure 7: The heightmap that was generated based on some Perlin noise and a blurred version of the accessibility map from Figure 6. The lighter a pixel is, the lower the corresponding heightmap cell's elevation.

3.8 Using Repulsion Radii to Resolve Node Overlap

As we mentioned earlier, different regions and corridors must not overlap to prevent the formation of unwanted shortcuts in the terrain. Due to the non-overlap constraints, the graph layout method already ensures any nodes (both initially placed nodes, which become terrain regions, and corridor nodes that form part of the corridors) do not overlap each other. However, the previously described non-overlap constraints do not fully prevent a node from entering the area inside the convex hull of another pair of nodes. If that were to happen, separate areas of accessible terrain would be erroneously connected, as is demonstrated in Figure 8.

To prevent nodes from overlapping with convex hulls, we modify the generated non-overlap constraints. Figure 9 shows an example of a situation where repulsion radii prevent overlap. Our approach is inspired by the paper by Stolpner et al. [30], who approximated a 3D shape by a union of spheres.

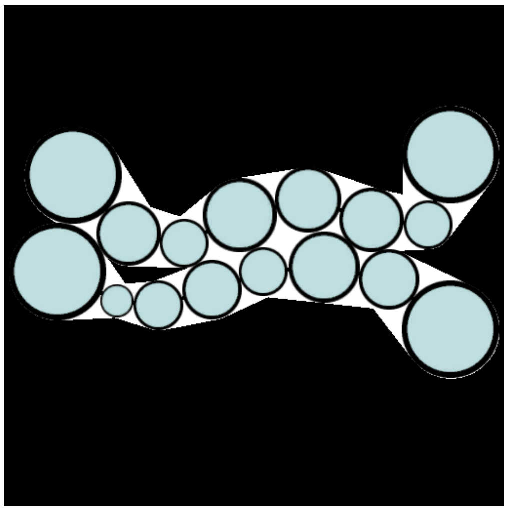

Figure 8: A problematic situation that occurred when repulsion radii were disabled. Nodes from one corridor overlap with the convex hull of connected node pairs from a neighboring corridor. This causes the corridors in the accessibility map (seen in the background of the image) to overlap, while they should have been separated.

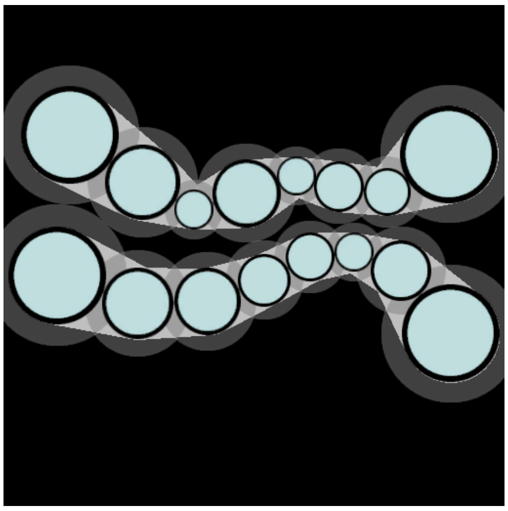

Figure 9: A situation similar to the one displayed in Figure 8. This time, repulsion radii (the large gray radii around the nodes) were enabled. The layout method uses the repulsion radii in the calculation of non-overlap constraints and repulsive forces, ensuring the corridors in the accessibility map do not overlap.

Each convex hull is formed out of the circles of two graph nodes p and q that are connected by a corridor edge. Since, as mentioned before, the length of that corridor edge is equal to $r_p + r_q$ (the sum of the radii of the two incident nodes), the circles should be touching each other at a single point v after the graph layout algorithm has concluded (see Figure 10). In such a situation, there are two external bitangent lines: two lines that are tangent to both circles, and for which the circles fall on the same side of the line.

For each pair of nodes connected by a corridor edge, we find the circle intersection point v and the point v' on one of the external bitangent lines t (it does not matter which one) nearest to v. If the radius of the two circles were increased until both circles touched v', the whole convex hull of the original two circles would be inside the area covered by the enlarged circles. We store the radii of the enlarged circles, r'_p and r'_q; such a radius is called the *repulsion radius.*

A node p usually has multiple incident corridor edges. A convex hull is computed for each one, resulting in multiple computed values for r'_p; each value is the enlarged radius necessary to contain a different convex hull. A node's repulsion radius is the *maximum* of all computed enlarged radii for that particular node.

During the graph layout algorithm, we use the repulsion radii when generating non-overlap constraints, rather than the regular node radii:

$$|\mathbf{p} - \mathbf{q}| \geq r'_p + r'_q$$

This ensures no nodes overlap with any convex hulls of other node pairs. If a node pair is connected by a corridor edge, the nodes are supposed to touch, and the repulsion radii should indeed overlap, as in Figure 10. That is why the non-overlap constraints are only generated for pairs of nodes that are not connected by a corridor edge.

We also use the repulsion radii in the computation of the repulsive forces between two non-adjacent overlapping nodes, again pushing those nodes apart until the distance between the nodes is at least $r'_p + r'_q$. The regular node radius is still used when generating the accessibility map.

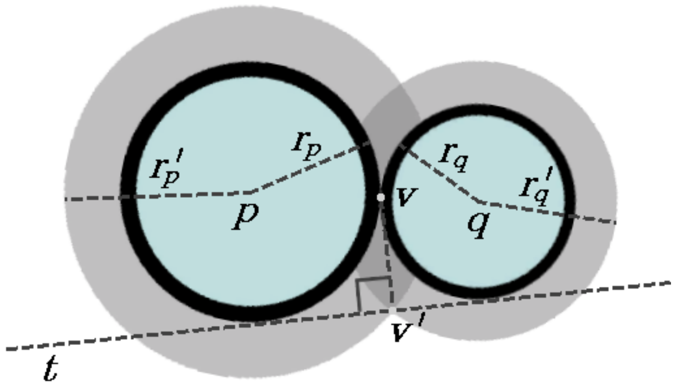

Figure 10: Computation of the repulsion radii r'_p and r'_q of the nodes p and q. We take the point where the two nodes intersect, v, and find its orthogonal projection v' on one of the circles' external bitangent lines, t. The repulsion radius of a node is the distance from its center to v'.

3.9 Generating the Terrain

The next step is to generate the terrain on which the game will be played. For simple tile-based games which only need a Boolean accessibility value per tile, the accessibility map can be used directly. For more complicated games with three-dimensional terrain, a heightmap can be generated based on the generated accessibility map, and after that the terrain itself.

The heightmap is a grid of the same size as the accessibility map. Every cell has an elevation value that indicates the height of the terrain at that location. The exact details of the heightmap generation phase will vary depending on the game. For example, the inaccessible areas could be given much higher elevations than the surrounding accessible areas, resulting in inaccessible mountainous terrain. Alternatively, inaccessible map cells could be given lower elevation values than accessible cells. The inaccessible parts of the map could then optionally be flooded with water or lava. Inaccessible terrain could also be covered by other obstacles, such as dense forests or tall buildings.

After a heightmap has been created, a three-dimensional polygonal terrain mesh can be generated, with the vertex heights being determined by the heightmap elevation values.

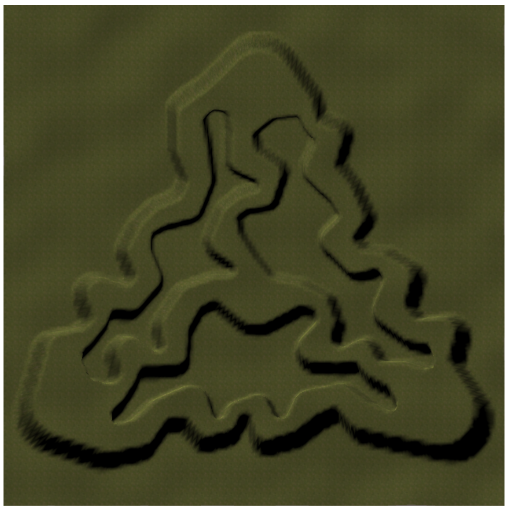

Figure 11: The 3D terrain that was generated based on the heightmap from Figure 7.

3.10 Implementation

Our implementation of the described terrain generation method was written in C# as a set of scripts for the Unity game engine [1]. The produced heightmaps have a size of 513×513 cells (while our method can be used to generate heightmaps of any size, Unity only supports square heightmaps whose sides have a length equal to a power of two plus one). The three-dimensional terrain meshes, complete with navigation meshes for pathfinding, were constructed by Unity based on our generated heightmaps. The generator we implemented takes about one or two minutes to generate a 3D terrain from an initial user-supplied graph drawing.

In our implementation, we gave all accessible heightmap cells a very low elevation value, and all inaccessible areas a very high elevation. A Gaussian blur was applied to the heightmap to smooth out the borders between accessible and inaccessible areas. We found that a kernel size of 15×15, with $\sigma = 2.0$, produced nice, steep hills in our terrains. On top of that, some Perlin noise was added using Unity's built-in noise function. We used the terrain system in the Unity game engine to create a 3D mesh out of the generated heightmap.

A strategy game that used the heightmaps generated by our specific implementation would take place in mountainous terrain, with the corridors forming canyons running between different regions. The accessible areas are bounded by tall cliffs; a small radius was chosen for the Gaussian blur so that the slope of the mountains is too steep for units to climb.

4 Experiments

We performed a number of automated experiments to assess the effectiveness of the terrain generation method. We wanted to know how well our generated terrains satisfied the constraints specified by a game designer. Additionally, we wanted to know what kind of test cases our algorithm would support well, and in which cases it was unable to produce good terrains. Our experiments also served to find good values for the input parameters for the terrain generation algorithm. Moreover, we ran some experiments with certain parts of the algorithm disabled, to find out whether each component of the generation method does indeed improve the quality of the output. Finally, we desired to know how different the various terrains we generated were. In order to determine the diversity of the output of our algorithm, we made various visualizations.

Section 4.1 contains a list of the different measurements we performed on the generated graph layouts and terrains. Section 4.2 describes the test cases that were used as input for the generation algorithm. This is followed by Section 4.3, which contains a discussion of the different experiments that were conducted. Section 4.4 gives an overview of the experiment results. Finally, Section 4.5 discusses the observed results.

4.1 Measurements

In order to determine how well the geometric properties of the generated terrains match up with the properties defined by the designer in the initial graph, we measured various aspects of the generated graph layouts, accessibility maps, and 3D terrains.

We want each region and corridor in the terrain to be clearly separated from other regions and corridors. A graph layout with overlapping nodes or intersecting edges causes shortcuts to be formed in the terrain that allow units to travel between regions faster than intended, and results in terrains with a different high-level structure than the designer intended. In order to determine whether regions and corridors are actually separated, we record the number of corridor edge crossings and the number of overlapping node pairs present in the produced graph layouts. The lower these numbers are, the better. Ideally, there should be zero edge crossings and no overlapping node pairs.

A very small amount of overlap was found to occur frequently for pairs of adjacent nodes (in this section, the term “adjacent nodes” will be used to refer to two nodes connected by a *corridor* edge—initial edges do not count). Since adjacent nodes are part of the same corridor anyway, those cases of overlap do not lead to different parts of the terrain being erroneously connected. That is why we chose to measure the number of *non-adjacent* overlapping node pairs—nodes that overlap, but are not connected by a corridor edge. Ideally, there will be zero such node pairs in each generated graph layout. We do not use the repulsion radii of nodes to determine whether they overlap. Instead, the regular node radius is used, since terrain shortcuts only form when the distance between two non-adjacent nodes is smaller than the sum of their regular radii.

The number of nodes and edges is different in the various test cases we will describe in the next section. In some experiments, different tested values for the input parameter also resulted in different numbers of nodes and edges (for example, a higher slack factor will result in more corridor nodes and edges being placed in each corridor). There were typically between 100 and 550 nodes, and about 130 to 600 edges in a graph after the addition of corridor nodes and edges.

In order to perform a fair comparison between different counts of edge crossings and overlapping node pairs, we normalized the values. Specifically, we divided the number of intersecting corridor edge pairs by the total number of corridor edge pairs; similarly, the number of non-adjacent overlapping node pairs was divided by the total number of node pairs. This allows us to compare the *fraction* of object pairs that overlaps or intersects.

We also wanted to know how well the distances units have to travel between regions in the terrain matched the designer's specifications. If the designer specified a balanced map, and these distances are not what the designer intended, the map will be imbalanced. We measured two distances: the graph distance and the terrain distance.

For every pair (p, q) of initial nodes in the graph, we find the shortest path in the graph between those nodes, moving only across corridor edges. There are n initial nodes $p, r_1, \ldots, r_{n-2}, q$ on this path, with $n \geq 2$; there are chains of corridor nodes between those initial nodes. For every consecutive pair $(p, r_1), (r_1, r_2), \ldots, (r_{n-2}, q)$ of initial nodes encountered on this path, we take the initial edge e_i connecting that pair. We calculate the *graph distance* d_g between p and q as follows:

$$d_g(p, q) = \sum_{i=1}^{n-1} \text{slack-factor}(e_i) \cdot \text{length}(e_i)$$

The graph distance tells us how long the user intended the path from p to q to be.

We use the three-dimensional terrain's navigation mesh, created by Unity, to find the paths that units would take to travel between the centers of every pair of regions in the terrain (with one region center being positioned at the position of each initial graph node). We measure the lengths of those paths; this is the *terrain distance* d_t.

The graph distances and terrain distances are compared by computing d_t/d_g. The closer this is to 1, the closer the terrain distance matches the designer's intentions. Our terrain generation method can only produce balanced maps if the designer's specifications are closely matched. For example, a game designer might specify that the corridors connecting each player's base to the nearest region containing resources all have the same length. If the d_t/d_g score of the corridors significantly deviate from 1, that would upset the game balance—it could be too easy for players to access resources (if $d_t/d_g < 1$) or too hard (if $d_t/d_g > 1$). Additionally, if the values for d_t/d_g vary a lot, some players might have easier access to resources than other players, which would be unfair.

There are also several properties of the algorithm's output that we know are always present; we do not have to measure those properties. Each region will have the size specified by the user, and each corridor will have the specified width. The region connectivity structure will also match the designer's intentions—regions are connected by corridors to other regions according to the configuration of nodes and edges in the initial graph drawing.

Some test cases are harder for the algorithm to find a good layout for than other test cases. As was mentioned in Section 3.5, the corridor layout phase is restarted when edge crossings appear during the graph layout algorithm. The higher the number of restarts, the more the algorithm struggled to find a layout that is free of edge crossings. In order to find out how hard different kinds of test cases are for our algorithm, we record the number of layout restarts in each run of the graph layout algorithm.

Finally, we also wanted to find out how much diversity was present in the different terrains the terrain generator produced. In each experiment, the generator was executed ten times for each test case. This means that ten accessibility maps were produced per graph in every experiment. We overlaid all ten produced accessibility maps for each test case. For each cell in the map, we counted the number of times it was accessible in the ten runs of the algorithm, producing a value between 0 and 10 for every map cell.

For each experiment, we then created a grayscale image based on these values for every graph. Each pixel in an image corresponds to one map cell. Black pixels denote an accessibility map cell that was marked as inaccessible during all ten terrain generation runs; white pixels denote cells that were always accessible. Various shades of gray denote cells that were sometimes accessible, and sometimes not; the lighter the shade of gray, the more often that accessibility map cell was accessible. Appendix C shows a selection of these diversity images.

4.2 Test Cases

We created ten different initial graph drawings to use as input in the experiments. These graph drawings were created based on maps from the real-time strategy game *StarCraft II*. Specifically, five two-player and five four-player multiplayer maps were picked from the pool of competitive ladder maps. These maps represent typical well-balanced strategy game maps; they have been used in *StarCraft* tournaments. The maps are symmetric—no player has an advantage over other players. Appendix A shows all graph drawings that were used in our experiments. In many figures and tables presented later, we will refer to graphs by their abbreviated names. Table 1 shows what the abbreviations mean.

Graph name	Abbreviation
Abiogenesis	A
Acid Plant	AP
Catalyst	C
Eastwatch	E
Neon Violet Square	NVS
Frost	F
Whirlwind	W
Cactus Valley	CV
Deadwing	D
Invader	I

Table 1: The names of the graphs used in the experiments, along with their abbreviations.

In order to create the test cases, we made a Unity editor tool that allows the user to draw a graph and specify properties such as the desired edge lengths, slack factors, and the stiffness force multiplier. We converted the *StarCraft* maps into initial graph drawings by placing an initial node in the location of each open region on a map. If two neighboring regions were connected via a choke point (such as a ramp), an initial edge was drawn between the two corresponding nodes. Figure 12 shows an example.

Initial nodes were given radii ranging from 15 to 40, depending on the size of the region on the map. We assigned lengths ranging from about 30 to 100 to the initial edges. In our test cases, initial nodes have between one and six adjacent nodes, with typical nodes having two or three neighbors.

The terrains generated by our method have the same high-level structure as the *StarCraft* maps, as described by the initial graph: there are as many regions in our terrains as there are in the *StarCraft* maps, and two regions in the generated terrains are directly connected by a corridor if and only if the corresponding *StarCraft* regions were connected.

Note that the purpose of our terrain generator is not to replicate the maps from *StarCraft*; the generated terrains differ from the *StarCraft* maps in some ways. For example, in the *StarCraft* maps, two different regions can have different elevations, touching each other while being separated by a cliff. There are no such cliffs in our terrains—with the heightmap generation step that we implemented, regions will always have a low elevation, and there will always be high, mountainous inaccessible terrain in between different regions.

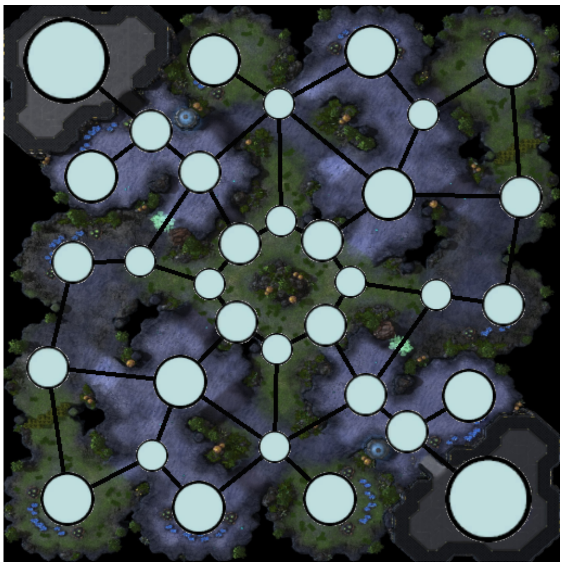

Figure 12: The Abiogenesis map from *StarCraft II*, overlaid with the corresponding graph drawing that was used in our experiments.

4.3 Performed Experiments

We performed a number of different experiments. In each experiment, we set all input parameters to certain constant values, and executed the terrain generation method ten times with each of the ten graphs, producing a total of 100 terrains per experiment. In each run of the terrain generator, both phases of the graph layout method (the initial layout phase and corridor layout phase) are run until convergence or the maximum number of iterations (200) has been reached. For every graph, we cleared the heightmap and then reloaded the same initial graph drawing at the start of all ten runs, making each run independent of any previous runs.

If corridor edge crossings are detected in 50 consecutive layout iterations, the corridor layout phase is restarted with new chains of corridor nodes and edges. In these experiments, we restarted a maximum of ten times; if there are still crossings after ten restarts, the terrain generator will proceed with the last graph. Since there are crossing edges in that case, there will be crossing corridors in the terrain; at the point where two corridors cross, game units would be able to take an unintended shortcut to get to their destinations earlier than they are supposed to.

The experiments are organized into sets. The purpose of each set of experiments was to investigate the influence of one input parameter. For every experiment in one set, a different value was used for one input parameter of the algorithm. The values for all other input parameters were fixed to values that we knew worked well in other tests.

Varying stiffness force multiplier. We conducted some experiments with varying values for the stiffness force multiplier. We tested multiplier values of 0, 0.001, 0.01, 0.1, and 1. This experiment was performed because were initially unsure which value for the multiplier would result in the closest match between the generated terrains' properties and the designer's specifications. The stiffness force affects the smoothness of the created corridors—the larger the stiffness forces, the smoother the bends—but we suspected that very strong stiffness forces might interfere too much with the attractive and repulsive forces, making it hard for the algorithm to produce good graph layouts. For hard test cases, we suspected we would see more layout restarts, more graph errors (intersecting pairs of edges and overlapping pairs of non-adjacent nodes) and terrain distances that deviated more from graph distances. During these experiments the slack factor of each edge was set to 1.25, and all corridors had a width of 20 heightmap cells.

Varying corridor width. Next, the stiffness force multiplier was fixed to 0.01. With the slack factor of every edge again set to 1.25, the desired width of all initial edges was set to different values; we tested graph where all corridor widths were 5, 10, 15, 20, and 25 (recall that the size of the heightmap is 513×513). The purpose of this experiment was to test whether our algorithm was able to support a wide variety of map specifications while producing high-quality maps—game designers might want to create some maps with narrow corridors, and other maps with wider ones. Narrow corridors make it harder for players to send large troop formations through, and could even block large units from passing through, affecting the balance of the game. Our hypothesis was that narrow corridors were easier to fit in the map than wider corridors; large corridor widths would therefore lead to more layout restarts, as well as a layout that would not match with the initially specified distances quite as well as a layout performed with small corridor widths.

Varying slack factor. Different slack factors were tested too. Game designers might want to vary the slack factors used in the initial graph drawing, just like they might vary the corridor widths. Different slack factors result in different map styles. Since longer slack factors result in a longer travel time through corridors, the slack factor is also a tool that can be used by the designer to balance the map. It is therefore desirable that our method can work with a variety of slack factors while satisfying all the constraints specified by the designer. With a corridor width of 20 and a stiffness force multiplier of 0.01, we set the slack factors of all initial edges to 1.2, 1.4, 1.6, 1.8, and 2.0. Large slack factors mean the algorithm has to fit in a lot of corridor nodes in the available space, which we thought could be a difficult set of constraints for the algorithm to handle.

No constraint projections. Additionally, we ran some experiments without constraint projections. After all, the attractive and repulsive forces already cancel out at the desired edge lengths, and non-adjacent overlapping nodes are pushed apart by repulsive forces too. We wondered if the constraint projections were really required to obtain a high-quality layout. For these experiments, the stiffness force multiplier was again 0.01, the corridor widths were 20, and different values for the slack factor (slack factors 1.2, 1.4, 1.6, 1.8, and 2.0) were tested, representing increasingly difficult constraints for the algorithm to meet.

No repulsion radii. We also tested the algorithm with constraint projections enabled, but without computing a repulsion radius for every node. Instead, the regular radii would be used when pushing apart overlapping nodes. On the one hand, the repulsion radii ensure the convex hull of adjacent node pairs does not overlap with other nodes; on the other hand, each node covers a larger area when using repulsion radii than when using the regular radii, so it might be harder for the layout algorithm to fit all the nodes in the map. We used the same settings for this experiment as in the experiment without constraint projections to see whether the layouts were better with or without the use of repulsion radii.

Unmodified Fruchterman-Reingold forces. As an aside, we also ran a couple of tests with the regular, unmodified Fruchterman-Reingold force model, in which repulsive forces are used to push apart *all* pairs of nodes, including unconnected pairs. The layouts we saw in those tests were of a very low quality. Almost all nodes were pushed to the sides of the map, overlapping with each other a lot; see Figure 13. We did not think the layouts were of a sufficient quality to warrant extensive tests.

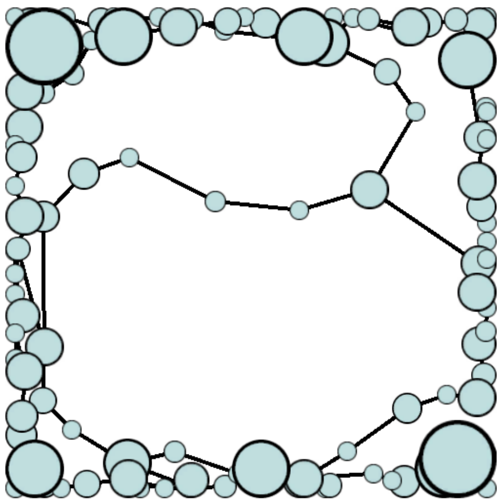

Figure 13: A graph layout that was obtained with the unmodified Fruchterman-Reingold forces.

4.4 Results

This section describes the data we gathered in the various experiments. In the next section, Section 4.5, we analyze the data, give explanations for observed effects, and draw conclusions based on the various measurements. Appendix B contains a number of graph layouts and corresponding accessibility maps that were generated in our experiments, and Appendix D shows some of the three-dimensional terrains that were produced.

Varying stiffness force multiplier. In the experiment with varying stiffness forces, we found that the total number of layout restarts was zero in most cases (see Table 2). Strong stiffness forces slightly increased the number of required restarts, although the effect is not very strong in most cases. There was one graph, Whirlwind, that resulted in noticeably more restarts than the other graphs. This was also the only graph that had corridor edge crossings (a single crossing in one of the experiments with a stiffness force multiplier of 1) and overlapping pairs of non-adjacent nodes (usually about 1 or 2 in each experiment, with slightly more overlap occurring with strong stiffness forces than with weak stiffness forces).

As can be seen in Figure 14, the graph distance is about 80% of the corresponding terrain distance on average. The range between the first and third quartile is relatively small; there are some outliers where the graph distance is either significantly longer or shorter than the graph distance. The relative difference between graph distances and terrain distances does not seem to change when the stiffness forces are varied.

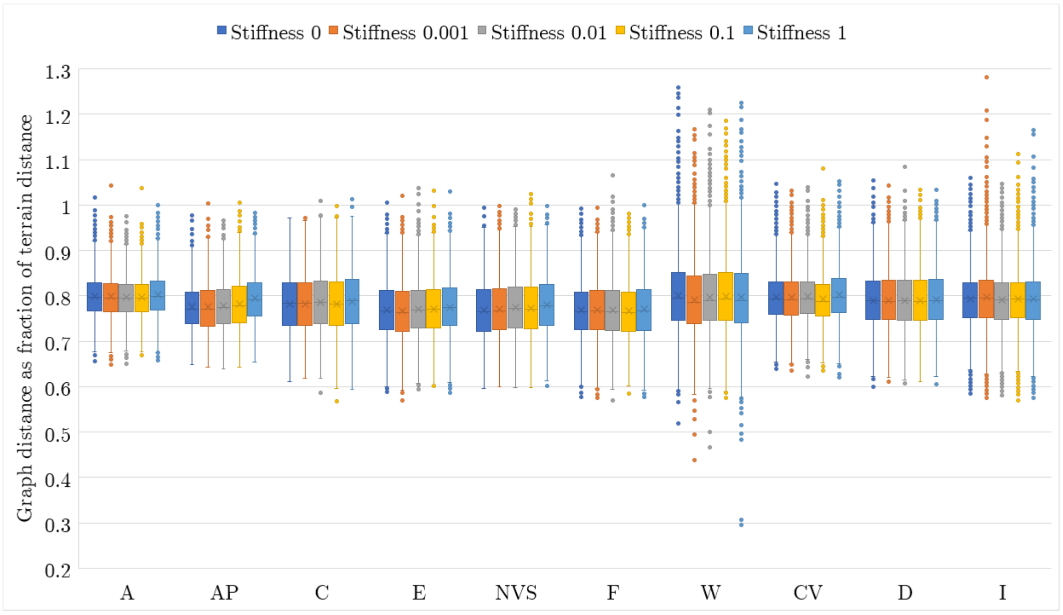

Figure 14: The terrain distances between all pairs of initial nodes, divided by the corresponding graph distance, for varying values of the stiffness force multiplier.

Graph name (abbreviated)	Stiffness force multiplier 0	0.001	0.01	0.1	1
A	0	0	0	0	0
AP	0	0	0	0	0
C	0	0	0	0	3
E	0	0	0	0	0
NVS	0	0	0	0	0
F	0	0	0	0	0
W	3	1	2	2	20
CV	0	0	0	0	0
D	0	0	0	0	1
I	0	3	0	2	1

Table 2: The total number of layout restarts after generating ten terrains per graph for various values of the stiffness force multiplier.

Graph name (abbreviated)	Corridor width 5	10	15	20	25
A	0	0	0	0	0
AP	1	0	0	0	0
C	1	0	0	0	0
E	3	0	0	0	0
NVS	1	0	0	0	0
F	2	0	0	0	0
W	2	0	0	7	4
CV	1	0	0	0	0
D	2	0	0	0	0
I	8	0	0	2	100

Table 3: The total number of layout restarts after generating ten terrains per graph for various values of the corridor width.

Varying corridor width. In the experiment with a varying corridor width, we found that the total number of graph layout restarts was usually small. For most tested corridor widths, we saw zero restarts for most graphs. The only width that most graphs struggled width was 5; that corridor width tended to result in a couple of restarts during the ten layout runs performed per graph. The Invader graph formed the most notable outlier. The layout algorithm was never able to find a layout for Invader without intersections when the corridor widths were at the maximum tested value of 25. In that case, all ten layout runs reached the maximum of ten restarts, for a total of 100 restarts (see Table 3).

This is reflected in the number of intersecting edge pairs, which was zero everywhere except when Invader was tested with a width of 25—in that case, there were between one and four edge crossings every run. Similarly, the number of overlapping node pairs was zero almost everywhere. For the highest tested corridor widths (20 and 25), there were on average a few overlapping pairs in the Whirlwind and Invader graph; this increased to a few dozen overlapping pairs in the most difficult test case for the algorithm—the Invader graph with a width of 25.

The mean value of d_t/d_g is about 0.8, just like in the experiments in which the stiffness force multiplier was varied. There is one exception: in the experiment where the corridor widths were 5, the terrain distances seemed to better match the graph distances, with d_t/d_g being about 0.9 on average (see Figure 15).

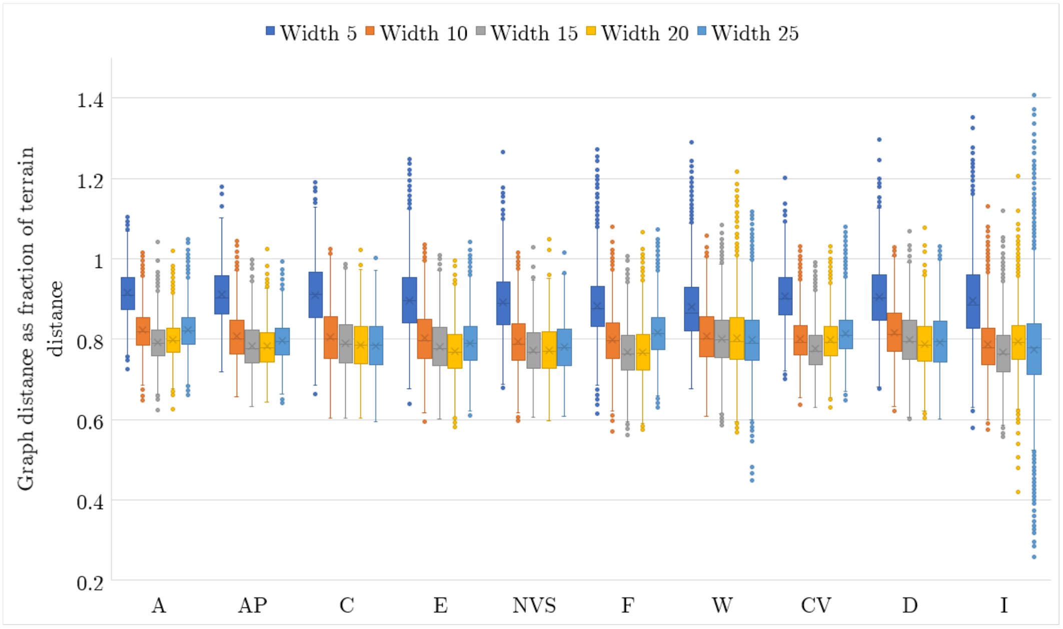

Figure 15: The terrain distances between all pairs of initial nodes, divided by the corresponding graph distance, for varying values of the corridor width. For the Invader graph at width 25, there are three more outliers at the high end that are not shown, maxing out at 1.93.

Varying slack factor. We saw very different results in the experiment where varying slack factors were tested. There were very few layout restarts in the experiments with a low slack factor (1.2 and 1.4). In the experiments with high slack factors, there were far more restarts; often, the maximum number of ten layout restarts was reached, after which the terrain generator proceeded with imperfect graph layouts containing edge crossings (see Table 4).

The number of edge crossings also started off very low for low slack factors, increasing as the slack factor went up. The fraction of corridor edge pairs that intersected ranged from 0 for low slack factors to about 0.0002 for high slack factors. That last case corresponds to about 20 edge crossings out of up to 90,000 pairs of edges (see Figure 18).

The number of non-adjacent overlapping node pairs showed a very similar pattern, increasing as the slack factor increases; see Figure 19 for the fraction of node pairs that overlapped. The fractions of overlapping node pairs are quite small. There were 0 overlapping pairs for slack factor 1.2, rising to typically about 100 or 200 overlapping node pairs (compared to a total node pair count of up to 80,000) for slack factor 2.0.

When the graph distances are compared to the terrain distances, we found significant differences between experiments with different slack factors. With a slack factor of 1.2, the mean value for d_t/d_g is about 0.8 (similar to the experiment with varying a stiffness force multiplier, which was performed with a slack factor of 1.25). As the slack factor increases to 2.0, the mean value of d_t/d_g decreases to about 50% in most cases, meaning the terrain distances do not match as well with the graph distances; see Figure 16. There are also many more outliers at the low end for high slack factors, representing paths in the terrain that were much shorter than the user specified.

Notably, there was one graph, Acid Plant, where the graph layout algorithm did not reach the maximum number of layout restarts. There were no intersecting edges and no overlapping node pairs in the graph layouts for this test case; d_t/d_g decreases far more slowly when the slack factor increases than it does for the other graphs.

Graph name (abbreviated)	Slack factor 1.2	1.4	1.6	1.8	2.0
A	0	1	7	89	100
AP	0	0	2	1	19
C	0	0	17	100	100
E	0	2	11	100	100
NVS	0	0	2	100	100
F	0	0	16	100	100
W	3	21	100	100	100
CV	0	0	14	100	100
D	0	4	67	85	100
I	1	19	100	100	100

Table 4: The total number of layout restarts after generating ten terrains per graph for varying values of the slack factor.

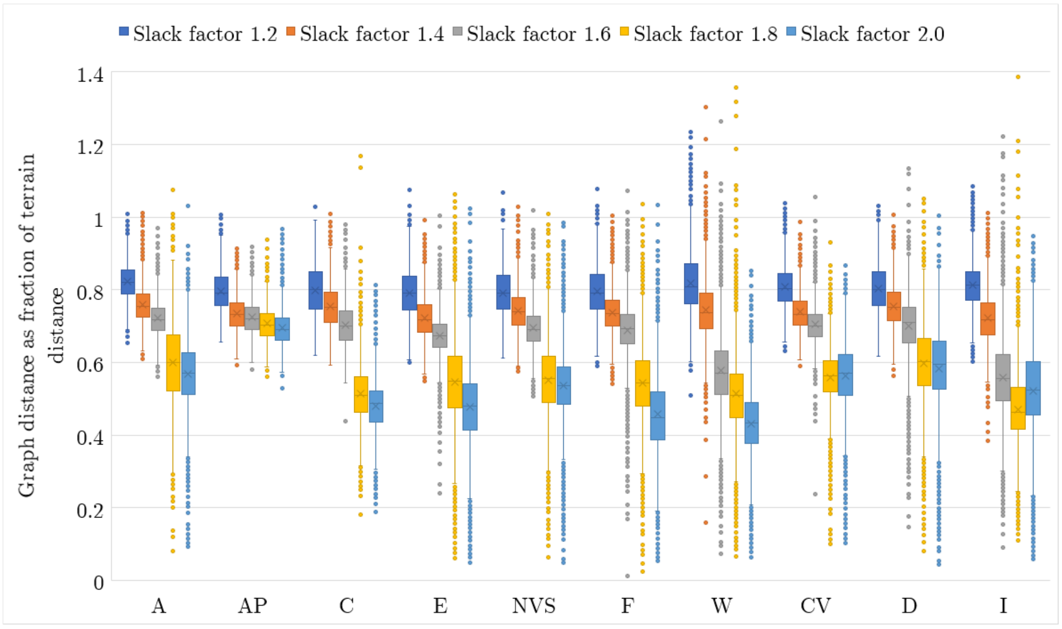

Figure 16: The terrain distances between all pairs of initial nodes, divided by the corresponding graph distance, for varying values of the slack factor.

Graph name (abbreviated)	Slack factor 1.2	1.4	1.6	1.8	2.0
A	0	0	0	2	100
AP	0	0	0	0	4
C	0	0	0	2	6
E	0	0	0	0	0
NVS	0	0	0	11	82
F	0	0	0	2	16
W	0	0	0	0	7
CV	0	0	0	13	100
D	0	0	0	15	68
I	0	0	0	2	15

Table 5: The total number of layout restarts after generating ten terrains per graph without constraint projections for varying values of the slack factor.

Graph name (abbreviated)	Slack factor 1.2	1.4	1.6	1.8	2.0
A	0	0	3	39	100
AP	0	0	3	6	92
C	0	0	10	87	100
E	0	1	6	44	100
NVS	0	1	8	64	100
F	0	0	4	100	100
W	0	1	51	100	100
CV	0	0	6	100	100
D	0	0	4	92	100
I	0	2	48	100	100

Table 6: The total number of layout restarts after generating ten terrains per graph without repulsion radii for varying values of the slack factor.

No constraint projections. We saw far fewer layout restarts in the experiment with constraint projections disabled (see Table 5) than in the tests where they were enabled (see Table 4). There were also fewer edge intersections without constraint projections (see Figure 18)—for half of the test graphs, there were zero edge crossings in every layout run. For the other half, there

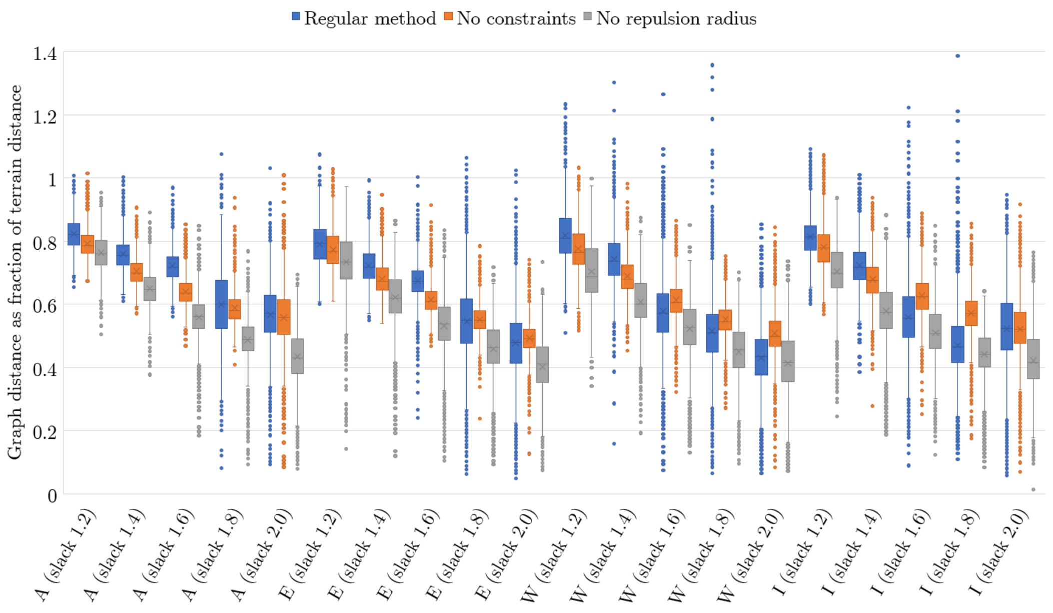

Figure 17: The terrain distances between all pairs of initial nodes, divided by the corresponding graph distance, for varying values of the slack factor. Results are displayed for a selection of graphs; results for the other graphs were similar.

were no edge crossings for low slack factors, and very few (≤ 10) crossings per experiment for the highest slack factors (1.8 and 2.0).

Without constraint projections, there are *more* overlapping node pairs than with constraint projections for low slack factors (although the amount of overlap is still very small). For higher slack factors, there is *less* overlap without constraint projections than when the regular method is used (see Figure 19).

For low slack factors, the mean value for d_t/d_g is somewhat lower in the experiments without constraint projections than in the experiments that used the regular layout method; see Figure 17. Since it is further away from 1, that means a poorer match between graph distance and terrain distance. At very high slack factors, on the other hand, d_t/d_g was frequently higher when constraint projections were not used. The variance was also lower without constraint projections—the values for d_t/d_g for all the paths between different pairs of initial graph nodes were more concentrated around the median.

No repulsion radii. The number of layout restarts is somewhat lower without using repulsion radii (see Table 6) than when the normal method is used (see Table 4). The number of edge crossings in the experiment without repulsion radii is sometimes lower and sometimes higher than in the tests with the regular method (see Figure 18); there seems to be no clear trend in either direction.

There was significantly more overlap between non-adjacent node pairs in every test case when repulsion radii were not used (see Figure 19). Compared to the regular method, the mean d_t/d_g value is always lower when repulsion radii are not used, and there are fewer high outliers; see Figure 17.

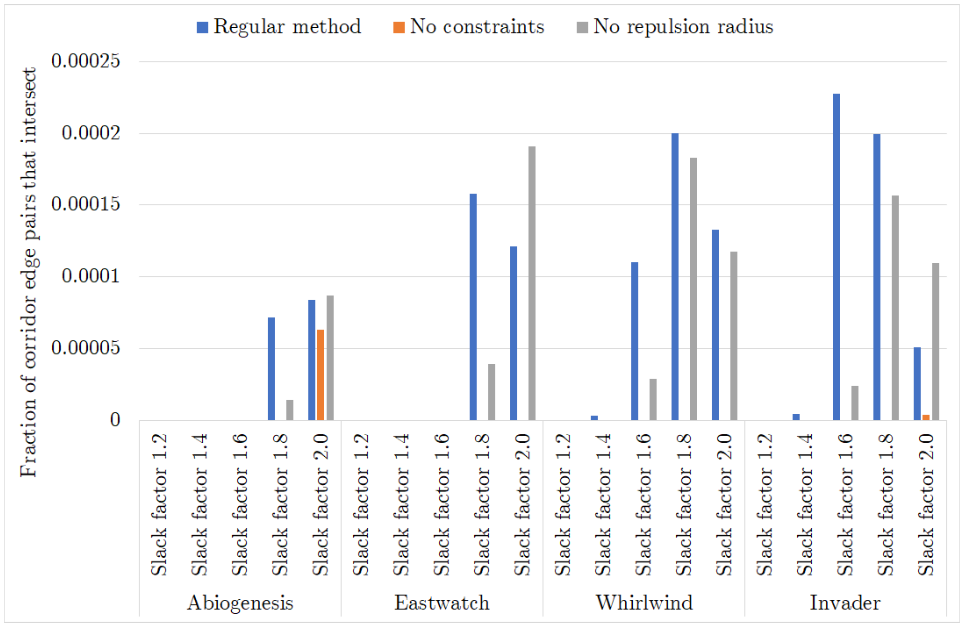

Figure 18: The mean fraction of corridor edge pairs that intersect for varying values of the slack factor. Results are displayed for a selection of graphs; results for the other graphs were similar.

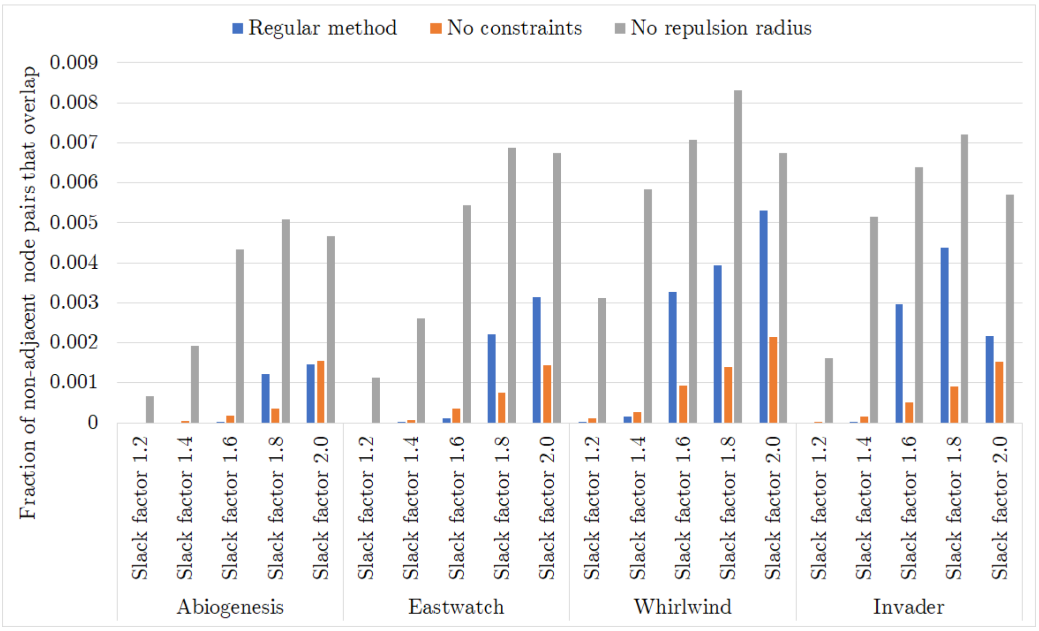

Figure 19: The mean fraction of non-adjacent node pairs that overlap for varying values of the slack factor. Results are displayed for a selection of graphs; results for the other graphs were similar.

4.5 Discussion

Overall, the algorithm seemed to perform quite well in reasonable cases where all nodes were able to be placed on the map. In those cases, we are able to make terrains with region sizes, corridor widths and an overall connectivity structure that matches the user's specifications.

The mean value of d_t/d_g typically seemed to be about 0.8, and the first and third quartiles were not far off. Taking the outliers into account, most d_t/d_g values usually extended between 0.6 and 1. That means that paths between regions were typically somewhat shorter than the designer intended. For some graphs, the layout method performed less consistently, producing more outliers than for other graphs—some test cases were clearly more difficult for the algorithm.

The ideal value of d_t/d_g is 1—with terrain distances that perfectly match the realized graph distances—but with the current graph layout method, we most often get a d_t/d_g below 1. It is straightforward to see why this is the case. The graph distances specified by the user are the distances one would travel when moving through the *middle* of each corridor, staying away from the walls. The terrain distance, on the other hand, is the length of the path that Unity calculates using its navigation mesh. This path is as short as possible, staying close to the corridor walls. This is why the terrain distance is usually smaller than the graph distance.

In some rare cases, the terrain distance can also be longer than the graph distance. This can happen when two corridor nodes in a chain that should have been touching each other are forced apart by surrounding nodes. If the corridor nodes lie further apart than intended, the corridor will be longer than the designer wanted it to be. Also, during the construction of the corridor node chains, corridor nodes with random radii are added to a chain until the sum of the corridor node diameters in that chain is *at least* as large as the desired corridor length. In our implementation, it is possible for the total length of a corridor chain to slightly exceed the desired length by up to the minimum corridor node radius. Since corridors are usually far longer than the minimum radius, and corridor nodes are not usually forced apart very far, the terrain distance usually still ends up shorter than the desired distance.

We found that the cases where d_t/d_g is greater than 1 tend to involve very short paths. When we measure the actual length of the terrain path in heightmap units (with a single heightmap cell having a size of 1×1 unit, and the entire map being 513×513 units large), the paths in the generated terrain that were *longer* than desired would often only be about 10 or 20 units too long. On the other hand, when there were a lot of overlapping nodes and edge crossings creating shortcuts in the terrain that shortened paths between regions, terrain paths that were *shorter* than the desired distance could in the worst cases be hundreds of units shorter than desired.

Since the values for d_t/d_g are usually clustered around 0.8, a simple improvement to the algorithm would be to take all the desired distances d between regions given by the user, and set the corresponding initial edge lengths to $1.25d$. The produced corridors would then be 25% longer as well: $d_g = 1.25d$. The final realized terrain distance would then be equal to the distance the user desires: $d_t = (d_t/d_g) \cdot d_g = 0.8 \cdot 1.25d = d$.

Correcting for a mean d_t/d_g would only ensure the *average* (and median) terrain distance is right; there would still be paths both longer and shorter than the user desired. Two initial edges that might have had the same desired length could result in corridors with slightly different lengths. This indicates that we may not be able to guarantee *perfect* gameplay balancing of the kind required for competitive tournaments. However, the generated terrains do seem to be approximately balanced. While we have not performed any strategy game playtests on the generated terrains, based on a visual inspection the generated terrains seem suitable for more casual gameplay.

In very difficult cases (for example, in the experiments with very high slack factors), the mean value for d_t/d_g may be much lower than 0.8. In such cases, there is so much overlap between different regions and corridors that the produced terrain loses much of the high-level structure described by the input graph drawing. A reasonable gameplay designer would not be interested in those cases anyway, since it is usually not physically possible to place all regions and corridors on the map without intersections or overlap. The fact that our algorithm cannot produce good layouts in such cases, then, is unsurprising but fortunately also of little importance for the intended application.

Our tool works with maps of a fixed size. Nodes cannot cross the boundaries of the map. If they were allowed to do so, it would be easier for the algorithm to find a good layout when a lot of regions and corridors have to be placed on the map. If a game designer does not require their map to have a certain size, a useful option that could be added to our generator would run the graph layout algorithm on a plane without boundaries. After a layout has been computed, the bounding box of the final graph layout would then indicate the size of the map. This could improve the quality of the produced terrains in difficult cases with many graph nodes and edges.

Our approach offers some advantages compared to other strategy map generation methods discussed in Section 2.2. Unlike the methods by Olsen [24] and Frade et al. [9], our method takes the balance of the map into account. The methods by Mahlmann et al. [20] and Barros and Togelius [2] focus on the placement of resources and player bases, and do not generate terrains with impassable obstacles. Adding terrain generation to their methods would appear to be significantly harder than adding the placement of resources and bases to ours, since our method already produces terrains with open, defensible regions that are well-suited for containing player bases and resources.

Well-designed maps created by human designers tend to be divided into regions and choke points; strategic terrain analysis techniques, such as those reviewed in Section 2.3, make use of that structure. Many terrain generation methods, such as the methods mentioned in the previous paragraph as well as those described by Togelius et al. [31] and Smith and Mateas [28], produce terrains that are not as clearly structured as human-built maps. We believe that this negatively affects gameplay balance and player enjoyment. Our method, on the other hand, produces well-structured terrains.

The algorithm by Uriarte and Ontañón [33] does make maps based on subdividing the plane into different regions. However, their approach is heavily based on randomization and does not make any guarantees about geometric properties such as the size of regions, the distance between regions, the width of choke points, and the overall connectivity structure of the regions. On the contrary, our method can make such guarantees. In addition to that, in order to guarantee balanced maps, Uriarte and Ontañón only generated perfectly symmetrical maps; our approach can be used to generate asymmetrical maps.

The tool by Liapis et al. [19] allows a game designer to make a coarse sketch of a map, which is then refined. The user has the freedom to make regions of any size, but they also bear the responsibility of creating a well-structured map with regions and choke points that have balanced properties. Our method, on the other hand, guarantees that the region and choke point structure, complete with certain geometric properties, is present. The maps produced by Liapis et al. appear to contain some grid-like artifacts, since they are based on a sketch that was made in a coarse grid; our method does not suffer from such artifacts. The diversity of the output their method can produce is also intrinsically limited, since the terrain features that were manually placed by the game designer in their sketch will exist in every generated map.

Finally, the initial terrain representation of Liapis et al. (a small, tile-based map) is similar to the full-resolution map, but smaller. Our method, on the other hand, uses a more abstract terrain representation—a graph drawing. We believe that is would be easier to fully automate our method, which would require an algorithm that produced an initial graph drawing. Conversely, fully automating the method by Liapis et al. would involve automatically placing accessible and inaccessible tiles on the low-resolution sketch grid, which is essentially the same problem as placing tiles on the full resolution map; deciding where to place the tiles is not trivial.

In the remainder of this section, we discuss the experiment results that were described in Section 4.4.

Varying stiffness force multiplier. Varying the stiffness force multiplier did not affect the quality of the graph (measured in terms of the number of edge crossings, overlapping node pairs, or value of d_t/d_g), with the exception of a minor reduction in quality at the highest tested stiffness force value (1) for one particularly hard test case. We hypothesized that high stiffness forces would interfere with the attractive and repulsive forces, resulting in worse layouts; this seems to be true, but only to a very small degree.

The appearance of the corridors changes depending on the stiffness force multiplier. A very small value will produce zig-zagging corridors with a lot of sharp twists; high values lead to corridors with large, smooth curves. The best value for the stiffness force multiplier will depend on what appearance the user desires; we found a value of 0.01 to work well.

The algorithm struggled somewhat with the Whirlwind graph; it was the only graph with overlapping nodes, and it generally took more layout restarts than the other graphs in the experiment where the stiffness force multiplier was varied. Whirlwind has several nodes with a high degree (see Figure 26). Arranging all corridors around the corresponding regions without overlap is difficult. This is especially true when the stiffness forces are strong, pushing the corridor nodes around in an attempt to form smoothly curving corridors. See Figure 40 for a produced graph layout and accessibility map of Whirlwind, and Figure 50 for Whirlwind's diversity image for the same experiment.

Varying corridor width. The corridor widths were varied from very narrow corridors (a width of 5) to corridors about as wide as the regions they connected (a width of 25). We thought that wide corridors would be harder for the algorithm to fit in the map, leading to a reduced quality. However, even very wide corridors usually did not result in problems, so the best value for the corridor width will depend on the appearance and gameplay properties the designer desires.

There was one exception in the experiment with Invader for a corridor width of 25; see Figure 41 and Figure 51. We think that is because the Invader graph has a several large parts where initial graph edges lie unusually close to each other (see Figure 29); it is hard for the layout method to place corridor chains there such that they do not overlap.

For the other graphs, varying the corridor width was no problem. Figures 42 and 43 show results of experiments with very narrow and very wide corridors, respectively; Figure 52 and Figure 53 are the corresponding diversity images.

The value of d_t/d_g was usually about the same when corridor widths were varied, except when the narrowest corridors (with a width of 5) were tested—that resulted in terrain distances that were noticeably closer to the corresponding graph distances. If the corridors are very narrow, the shortest path in the terrain will deviate less from the graph distance path, which goes through the middle of the corridors. This results in a d_t/d_g score closer to 1.

So, why was the value of d_t/d_g only significantly different for a corridor width of 5? Why did it not not decrease further when the corridor width was increased from 10 to 25? After looking at the generated corridor chains, we believe the most likely explanation has to do with the number of corridor nodes in each chain. We found that this number was not that much different when the corridor width was set to values between 10 and 25. There were typically about six nodes in each chain when the width was 10, reducing to about three or four per chain for a width of 25. This small difference in the number of corridor nodes seems to explain why d_t/d_g did not vary much for corridor width values of 10 to 25. For a corridor width of 5, on the other hand, there were often a dozen or more very small corridor nodes; the navigation mesh paths were forced to follow the paths through the node centers far more closely. This could explain why d_t/d_g was higher for a corridor width of 5.

Varying slack factor. Higher slack factors clearly result in lower quality graphs; this was indeed what we expected. The long corridors that have to be placed on the map when the slack factors are high simply do not fit without overlapping (and often intersecting) other corridors. With the test cases we constructed, the layout runs with slack factor 1.2 and 1.4 performed well, but the higher slack factors resulted in a significant drop in quality. Therefore, if graphs like our test cases are used, we do not recommend a slack factor much higher than 1.4.

If a sparser initial graph were used, there would be more room for corridors to expand without running into other corridors. In fact, one of our test graphs, Acid Plant, was so sparse that all tested slack factor values resulted in no edge crossings and no overlapping nodes (see Figure 44 and Figure 45 for produced graph layouts and accessibility maps for slack factors 1.2 and 2.0, and Figure 54 and Figure 55 for the corresponding diversity images). There seems to be a tradeoff here: the denser the initial graph, the lower the slack factor should be.

No constraint projections. The constraint projections appear to increase the number of layout restarts and intersections. Without constraint projections, the attractive and repulsive forces push the nodes around, trying to achieve the desired distances between connected node pairs; however, if there are too many other surrounding nodes pushing back, nodes will be unable to reach their desired distances. With constraint projections, on the other hand, the positions of pairs of nodes are adjusted such that the distances are correct, without regard for any surrounding nodes—which can introduce edge crossings in the process.

For low slack factors, there are fewer overlapping node pairs when constraint projections are active than when they are inactive. The non-overlap constraints help to force apart overlapping nodes. Compare Figure 46 (a graph layout produced without constraint projections) with Figure 44 (the same situation with constraint projections enabled) to see the difference that constraint projections make. Figures 56 and 54 show the corresponding diversity images.

For high slack factors, this is sometimes also true. However, for some test graphs, the method *without* constraints actually performed better. In the test cases with high slack factors, achieving the desired slack factors without any intersections and overlap was actually not possible for most graphs. In such impossible cases, the constraint projections introduce edge crossings; we saw fewer edge crossings when constraint projections are disabled. Edge crossings also result in overlapping nodes in the neighborhood of the points where edges intersect. Such cases of overlap cannot be fixed by the non-overlap constraints (which is why we recommend restarting the layout algorithm with new chains of corridor nodes when edge crossings are detected).

So, in cases were meeting all constraints is impossible, the layout method without constraints degrades more gracefully. However, in the more reasonable test cases—where the slack factor was low, and all corridors were possible to fit in the graph with virtually no overlap and no intersections—the constraint projections helped reduce the overlap between nodes, and resulted in better layouts with a higher d_t/d_g. Figure 47 shows how the algorithm performs without constraint projections on a reasonable test case with a high slack factor; compare it with Figure 45.

No repulsion radii. When the regular node radii were used to push apart overlapping nodes (rather than the repulsion radii used in the full algorithm) we did not see a noticeable increase or decrease in edge crossings. The number was sometimes higher without repulsion radii, and sometimes lower; this can be attributed to the randomization present in the construction of the corridor node chains. The number of overlapping node pairs, on the other hand, was far higher without repulsion radii in all cases, and the d_t/d_g score was also worse without repulsion radii. When the regular node radii are used instead of the repulsion radii there are many cases where nodes overlap with other nodes or the convex hull of a consecutive pair of nodes in a corridor chain, forming many unwanted shortcuts in the terrain. See Figure 48 (compare with Figure 44) and Figure 49 (compare with Figure 45). Figure 57 shows the diversity image of an experiment without repulsion radii; Figure 55 shows the results of the experiment with the same settings with repulsion radii enabled.

When the algorithm has to fit in many long corridors on the map, the corridors are spread around when repulsion radii are used, filling in the available space. Interestingly, without repulsion radii, the corridors are not evenly distributed on the map; instead, the layout method forms zig-zagging corridors everywhere with the particular value of the stiffness force multiplier we used in that experiment (0.01). In fact, most of the cases where a corridor node p overlapped with the convex hull of some other pair of nodes (q, r) happened when p, q, and r were all part of the *same* corridor. When such corridors were folded back upon themselves, the terrain distance between the two end regions was far shorter than the graph distance.

This means that the repulsion radii do not only ensure that a corridor remains well separated from other corridors; just like the stiffness forces, the repulsion radii also help ensure that a corridor node does not get too close to other corridor nodes further down its own chain. Without repulsion radii, the stiffness force multiplier has to be set a lot higher to prevent the formation of zig-zagging corridors.

Diversity. While we were primarily concerned with ensuring our generated terrains satisfied the given distance and non-overlap constraints, a secondary goal was to achieve a diverse output. As the diversity images in Appendix C show, the positions of the corridors are often somewhat varied, but regions tend to be placed at the same position every time—even if there is plenty of room to shuffle parts of the graph around more. We think more improvements could be made to increase the diversity of the output. For instance, random forces could be applied to the nodes in the initial graph drawing during the initial graph layout phase, pushing the nodes around in random directions. These forces could result in more diverse initial graph layouts that still satisfied all distance constraints. This, in turn, would lead to more diverse accessibility maps and, by extension, more diverse terrains.

5 Conclusion

The goal of this thesis was to develop a procedural content generation technique that generates balanced maps for strategy games. A terrain generation tool has been created that allows a game designer to specify properties that affect game balance, such as the length of terrain corridors, at a high level of abstraction. This allows the designer to easily create balanced maps. Our generator can randomly generate heightmaps that take the designer's specifications into account. The resulting maps look quite playable and could plausibly form the base terrain for a strategy video game. We conducted various experiments with symmetric—and therefore balanced—test cases as input. Based on these experiments, we believe our method can generate maps that are balanced enough to be used for casual gameplay.

In Section 5.1, we summarize the contents of this thesis, including the procedural method that was presented as well as the performed experiments and the obtained results. In Section 5.2, we make suggestions for possible improvements and provide directions for further research.

5.1 Summary

We have presented a novel terrain generation technique that uses graph nodes and edges to represent the different regions and corridors in a terrain, respectively. This abstract representation captures the high-level structure of a terrain that we believe determines in large part how strategy game matches play out. The terrain generation tool that we have implemented lets a game designer draw such a graph and specify geometric properties that the generated terrain should have. Designers can choose how many regions a terrain should have, which regions should be connected by corridors, and how large each region should be. They can also determine the Euclidean distance between regions, as well as the width and length of the different corridors.

The terrain generator takes the designer's graph drawing and finds a good initial graph layout that satisfies the designer's constraints. In order to do that, a graph layout algorithm that uses both Dwyer's constraint projections and a modified version of the Fruchterman-Reingold method is used. Next, chains of corridor nodes, representing the layout of terrain corridors, are added to the graph. The graph layout method is then run again, this time with the addition of stiffness forces and repulsion radii to improve the quality of the layout. After the graph layout method has converged, an accessibility map is created based on the final graph layout, indicating which areas of the map are accessible to game units and structures. This accessibility map is used to generate a heightmap that describes a three-dimensional terrain. Finally, a 3D terrain mesh, ready to be used as the base terrain in a strategy game, is generated based on the heightmap.

We have created ten graph drawings based on *StarCraft II* ladder maps. These were used as test cases in a number of experiments. Various values for different algorithm parameters were tested. We found that a stiffness force multiplier of 0.01 works well. On the maps we generated, which had a size of 513×513 cells, we found that our algorithm supported all tested corridor width values—widths between 5 and 25. For inputs like the test cases we made, we found slack factors up to 1.4 to work well, meaning the corridor length between two regions should not be much longer than 1.4 times the Euclidean distance between those regions.

When the best parameters were used, the distance units have to travel between regions is on average about 80% of the distance the designer specified, with half of the distances between regions falling between 75% and 85% of the specifications. This can be attributed to the fact that the shortest paths units can take to travel between regions tend to stay close to the corridor walls, while the designer's specifications—which our algorithm tries to match—specify the length of paths through the middle of the corridors. A design tool that takes this into account, and scales up the initially specified lengths by 25%, should make the average travel time between terrain regions more closely match what the designer had in mind.

When we looked at path lengths, we found some outliers; cases where the travel time for units does not closely match the specifications. This means that a perfect balance, which would be desirable for competitive maps such as those used in tournaments, cannot be guaranteed. However, the imbalances appear to be minor enough to support more casual, non-competitive matches.

We also tested our method with parts of the algorithm disabled to verify whether those components helped to improve the quality of the generator's output. Without stiffness forces, a lot of zig-zagging artifacts tend to form in the corridor chains. The constraint projections help place connected nodes at the correct distance, and resolve cases of overlap. The repulsion radii also helped reduce the amount of overlap, preventing the convex hulls of connected node pairs from overlapping with other nodes. Finally, without our modification to the Fruchterman-Reingold forces, nodes are pushed away from the center of the map, piling up at the sides. Based on the results of our experiments, we conclude that all the additions we made to the graph layout algorithm were beneficial.

5.2 Future Work

There are multiple improvements and extensions that could be made to our method. First of all, as we mentioned before, a simple improvement that could be made to the terrain design tool would take into account that the average distance units have to cross in the terrain will end up 20% shorter than the what the designer specified. This can be corrected by scaling up the user-provided desired initial edge lengths by 25%.

Even after the mean distance between regions has been corrected, we cannot guarantee that all geometric properties specified by the game designer are satisfied. By design, some properties will always be present in the generated terrains for all possible inputs, as long as that input is reasonable—meaning it is possible to fit all nodes on the map without overlap or edge crossings. The size of each region, the width of each corridor, and the overall connectivity structure of the regions and corridors will be what the designer intended them to be. This is not always the case for the distances between different regions, however. In the ten test cases we used in our experiments, we found that the produced maps still contained numerous cases where the paths through the terrain were either too short or too long. One could create a more sophisticated way of generating terrains that takes into account that units take the shortest path through each corridor, traveling close to the sides of corridors. This would allow for the creation of more balanced maps.

The corridors consist of all areas that are covered by nodes, or the convex hull of connected node pairs. The boundaries of those convex hulls have long straight segments, which results in corridor walls with many straight segments. This looks somewhat unnatural, and is particularly noticeable in the transitions from wide regions to narrow corridors. If the corridor walls were slightly randomly perturbed, the terrain might look a bit more realistic. Additionally, the boundaries of regions could be improved as well. Our method currently only supports circular regions. An improvement to our algorithm that resulted in the ability to generate regions with a wider variety of shapes would certainly be desirable.

The repulsion radii were introduced to keep the area covered by the convex hull of connected node pairs clear of other nodes. Since the nodes with repulsion radii have circular boundaries, we were able to resolve overlap with simple circle-circle intersection tests. These circular shapes cover more than just the nodes and their convex hulls, however. There is some extraneous area around each node; this extra area causes nodes to push each other away more than strictly necessary, making it harder for the algorithm to find a good layout. If more complex geometric primitives than circles were used, the convex hull of connected pairs could be covered without any additional area sticking out. This would give the algorithm more room to place nodes, making it possible to support more difficult constraints (such as higher slack factors) and possibly improving the quality of the layout, matching the designer's desired distances more closely.

Another option that could be added to our method to support situations with a large number of nodes is the ability to run the graph layout algorithm on an unbounded plane. The map size would then be determined by the bounding box of the final graph layout.

There is often a lot of room on the map in which nodes can be placed such that the user-provided constraints are met. However, in our experiments, we frequently found little difference in node positions between different runs of the layout method. The nodes could be moved around more to improve the diversity of the generator's output. The initial graph layout phase—the layout phase before the corridor node chains get added—gives virtually the same layout every time it runs. Introducing more randomization in that phase, for example by using forces to pull the nodes in random directions, could lead to more diverse generated terrains.

We focused on generating a heightmap. The corresponding terrain can be used as the base terrain for a strategy game map. For a full strategy game terrain generator, additional game-specific elements would have to be added to the map, such as resources or different types of terrain, like one-way traversable terrain. The placement of such features also affects game balance. Incorporating those elements in our terrain generator would be a possible direction for future research.

Our current implementation is fast enough to generate moderate-sized maps (with a few dozen initial graph nodes, and just over 260,000 heightmap cells) in about one or two minutes. If one desires to generate very large strategy game maps, our terrain generator may not be fast enough, since the running time scales up rapidly as the map size increases. We used a simple but naive implementation with a graph layout time complexity of $O(i \cdot n^2)$, where i is the maximum number of layout iterations and n is the number of nodes in the graph, including corridor nodes. The accessibility map generation phase takes $O(m \cdot n)$ time, where m is the number of heightmap cells. With the right implementation—for example, when spatial data structures are used—we believe it should be possible to reduce this complexity, allowing for larger maps to be be supported.

Finally, it is possible to use our method as part of a fully automated terrain generator. The method we presented is semi-automated—it requires a designer to provide a graph drawing as input, with specifications for properties such as the desired edge lengths. A method that automatically generates a graph that describes the high-level structure of a terrain could be used together with our generation method to fully automate the terrain generation process.

Our procedural generation method could also be adapted for other applications. First-person shooters, for instance, also often make use of outdoor environments. For those games, line-of-sight mechanics are important, so those would have to be taken into account in the generation process. Our method could also be used to generate outdoor tracks for racing games; the curvature of corridors plays an important role in such games, and should preferably be configurable by a game designer. It would also be possible to use our approach to generate maps for other applications, such as terrains for role-playing games or cavelike dungeons for roguelike games.

References

[1] Unity game engine. `https://unity3d.com/unity` (archived at `http://www.webcitation.org/707yP8b8f` on 2018-06-12).

[2] Gabriella AB Barros and Julian Togelius. Balanced civilization map generation based on open data. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pages 1482–1489. IEEE, 2015.

[3] Adrien Bernhardt, André Maximo, Luiz Velho, Houssam Hnaidi, and Marie-Paule Cani. Real-time terrain modeling using CPU-GPU coupled computation. In *Graphics, Patterns and Images (Sibgrapi), 2011 24th SIBGRAPI Conference on*, pages 64–71. IEEE, 2011.

[4] Wittaya Bidakaew, Pavadee Sompagdee, Sukanya Ratanotayanon, and Pongsagon Vichitvejpaisal. Rts terrain analysis: An axial-based approach for improving chokepoint detection method. In *Knowledge and Smart Technology (KST), 2016 8th International Conference on*, pages 228–233. IEEE, 2016.

[5] Tim Dwyer. Scalable, versatile and simple constrained graph layout. In *Computer Graphics Forum*, volume 28, pages 991–998. Wiley Online Library, 2009.

[6] David S Ebert. *Texturing & modeling: a procedural approach.* Morgan Kaufmann, 2003.

[7] Kenneth D Forbus, James V Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17(4):25–30, 2002.

[8] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[9] Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta. Automatic evolution of programs for procedural generation of terrains for video games. *Soft Computing*, 16(11):1893–1914, 2012.

[10] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[11] James Gain, Patrick Marais, and Wolfgang Straßer. Terrain sketching. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 31–38. ACM, 2009.

[12] Johan Hagelbäck and Stefan J Johansson. Using multi-agent potential fields in real-time strategy games. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 631–638. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[13] Kári Halldórsson and Yngvi Björnsson. Automated decomposition of game maps. *AIIDE*, 15:122–127, 2015.

[14] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1, 2013.

[15] D Higgins. Terrain analysis in an RTS – The hidden giant. *Game Programming Gems*, 3:268–284, 2002.

[16] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. In *Computer Graphics Forum*, volume 29, pages 2179–2186. Wiley Online Library, 2010.

[17] Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[18] Raúl Lara-Cabrera, Carlos Cotta, and Antonio J Fernández-Leiva. A review of computational intelligence in RTS games. In *Foundations of Computational Intelligence (FOCI), 2013 IEEE Symposium on*, pages 114–121. IEEE, 2013.

[19] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *FDG*, pages 213–220, 2013.

[20] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Spicing up map generation. In *European Conference on the Applications of Evolutionary Computation*, pages 224–233. Springer, 2012.

[21] Gavin SP Miller. The definition and rendering of terrain maps. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 39–48. ACM, 1986.

[22] F Kenton Musgrave, Craig E Kolb, and Robert S Mace. The synthesis and rendering of eroded fractal terrains. In *ACM Siggraph Computer Graphics*, volume 23, pages 41–50. ACM, 1989.

[23] Julio Obelleiro, Raúl Sampedro, and D Cerpa. RTS terrain analysis: An image-processing approach. *AI Game Programming Wisdom*, 4:361–372, 2008.

[24] Jacob Olsen. Realtime procedural terrain generation. 2004.

[25] Luke Perkins. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. *AIIDE*, 10:168–173, 2010.

[26] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[27] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

[28] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[29] Szymon Stachniak and Wolfgang Stuerzlinger. An algorithm for automated fractal terrain deformation. *Computer Graphics and Artificial Intelligence*, 1:64–76, 2005.

[30] Svetlana Stolpner, Paul Kry, and Kaleem Siddiqi. Medial spheres for shape approximation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(6):1234–1240, 2012.

[31] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, Georgios N Yannakakis, and Corrado Grappiolo. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14(2):245–277, 2013.

[32] Paul Tozour. Influence mapping. *Game programming gems*, 2:287–297, 2001.

[33] Alberto Uriarte and Santiago Ontañón. Psmage: Balanced map generation for starcraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[34] Alberto Uriarte and Santiago Ontañón. Improving terrain analysis and applications to rts game ai. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[35] Howard Zhou, Jie Sun, Greg Turk, and James M Rehg. Terrain synthesis from digital elevation models. *IEEE transactions on visualization and computer graphics*, 13(4):834–848, 2007.

A Test Cases

We used the same set of ten initial graph drawings as test cases throughout our experiments. These graph drawings are shown in this section. They were based on *StarCraft II* ladder maps; see Section 4.2 for more information. The first five maps—Abiogenesis, Acid Plant, Catalyst, Eastwatch, and Neon Violet Square—are two-player maps. The remaining maps— Frost, Whirlwind, Cactus Valley, Deadwing, and Invader—are four-player maps.

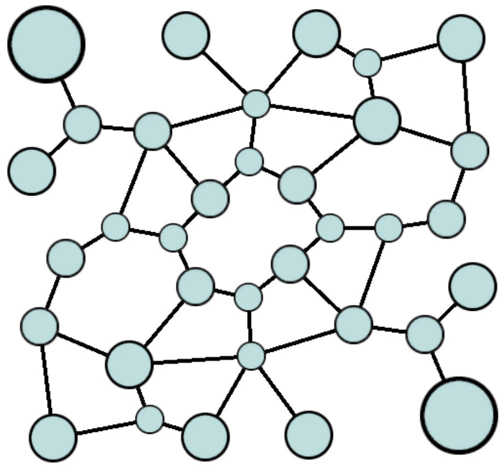

Figure 20: A graph drawing based on the Abiogenesis map.

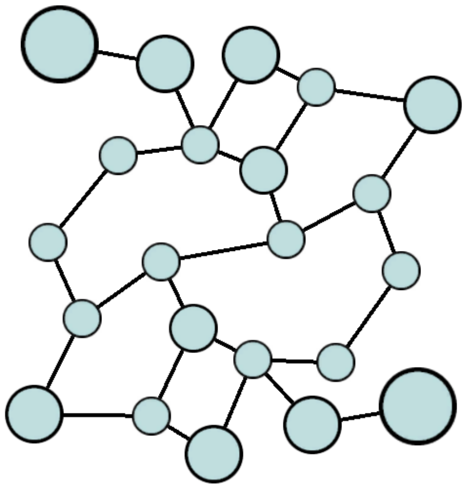

Figure 21: A graph drawing based on the Acid Plant map.

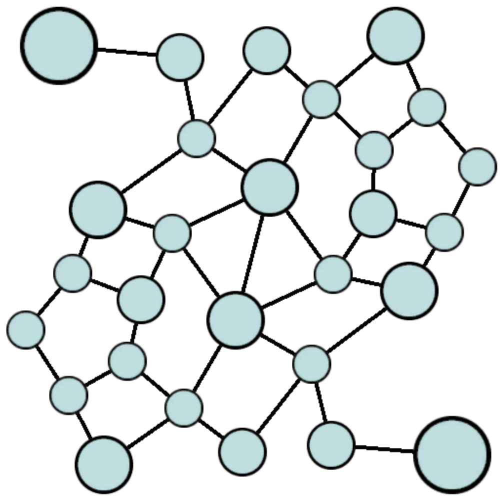

Figure 22: A graph drawing based on the Catalyst map.

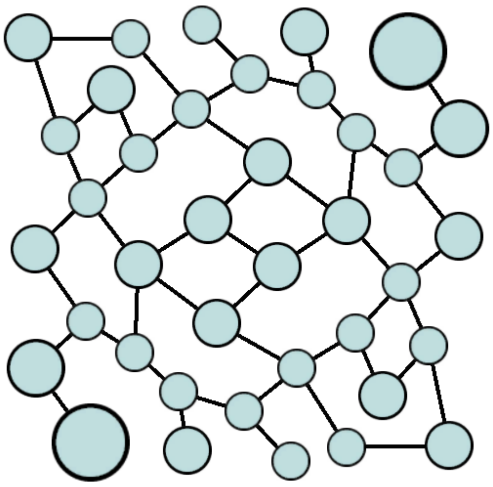

Figure 23: A graph drawing based on the Eastwatch map.

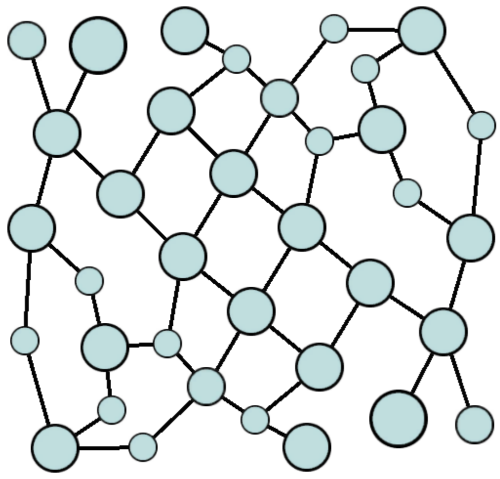

Figure 24: A graph drawing based on the Neon Violet Square map.

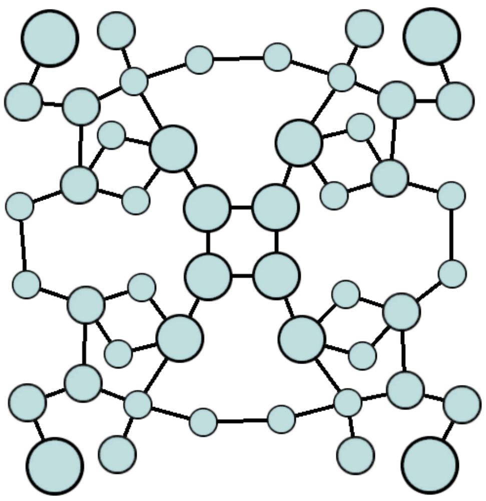

Figure 25: A graph drawing based on the Frost map.

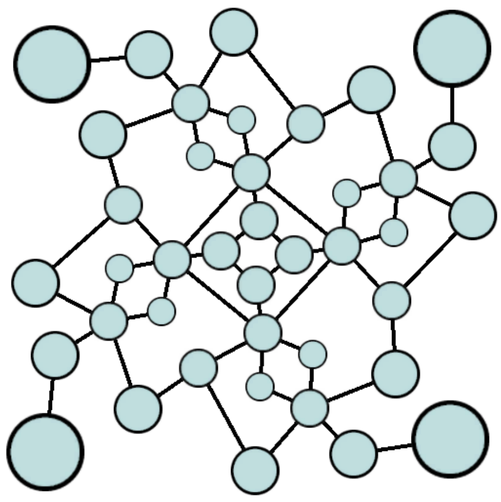

Figure 26: A graph drawing based on the Whirlwind map.

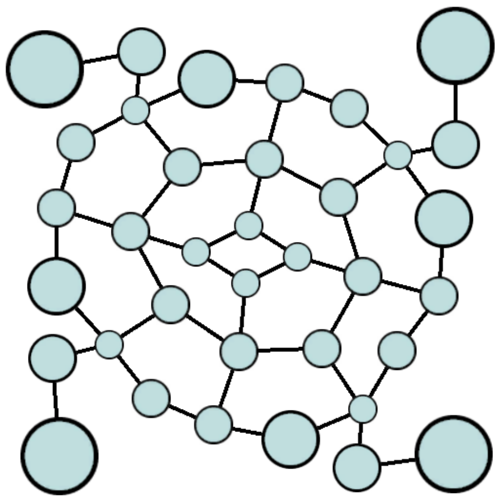

Figure 27: A graph drawing based on the Cactus Valley map.

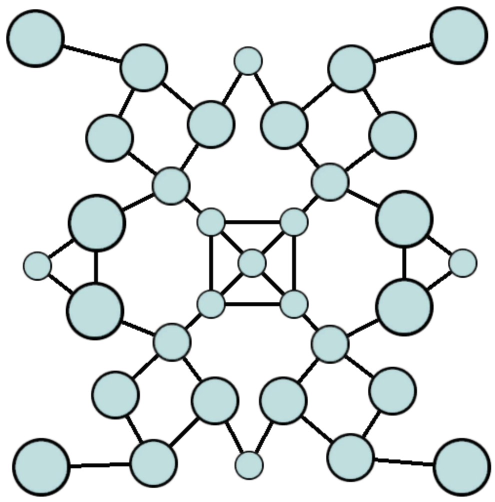

Figure 28: A graph drawing based on the Deadwing map.

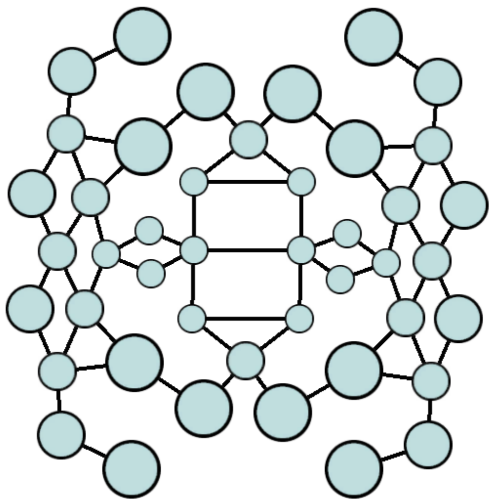

Figure 29: A graph drawing based on the Invader map.

B Generated Graph Layouts and Accessibility Maps

In this section, we present a selection of graph layouts that were produced in our experiments. The corresponding accessibility maps are also displayed. Unless stated otherwise, the displayed results were obtained used a stiffness force multiplier of 0.01, a slack factor of 1.25, and a corridor width of 20.

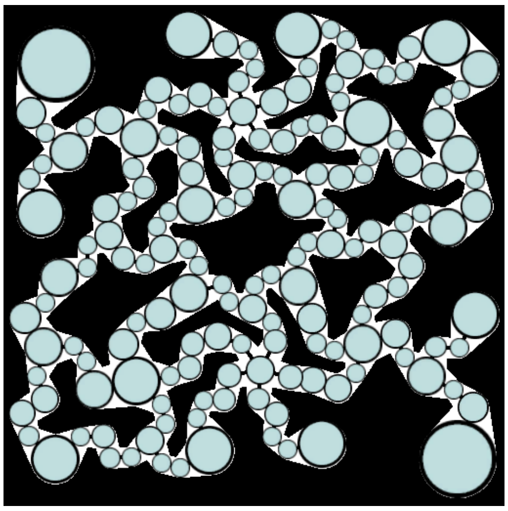
Figure 30: A graph layout and accessibility map generated based on the Abiogenesis test case from Figure 20.

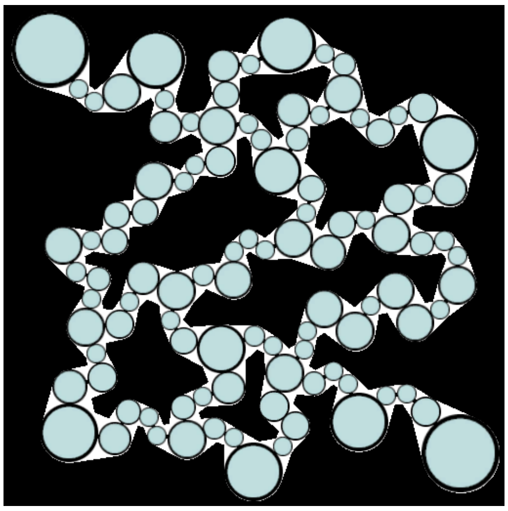
Figure 31: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21.

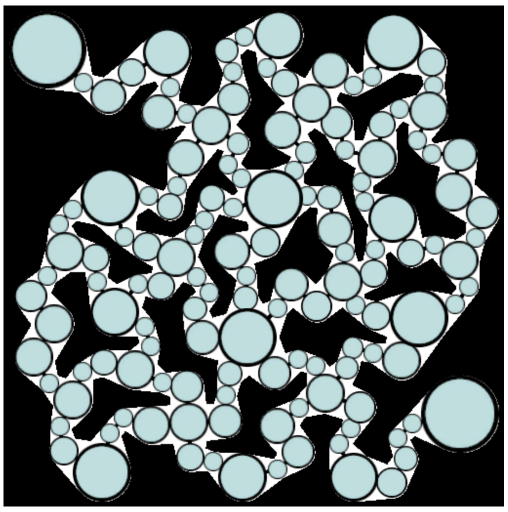
Figure 32: A graph layout and accessibility map generated based on the Catalyst test case from Figure 22.

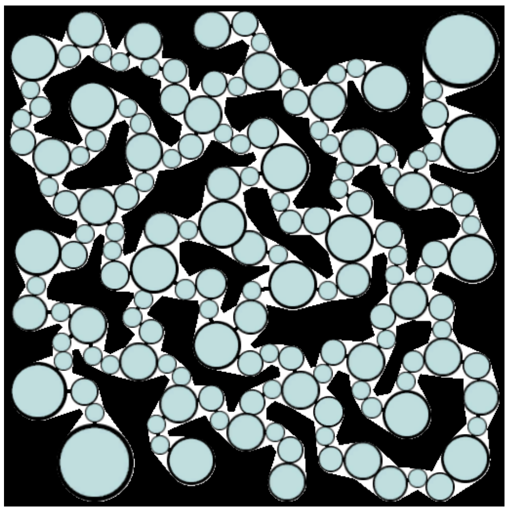
Figure 33: A graph layout and accessibility map generated based on the Eastwatch test case from Figure 23.

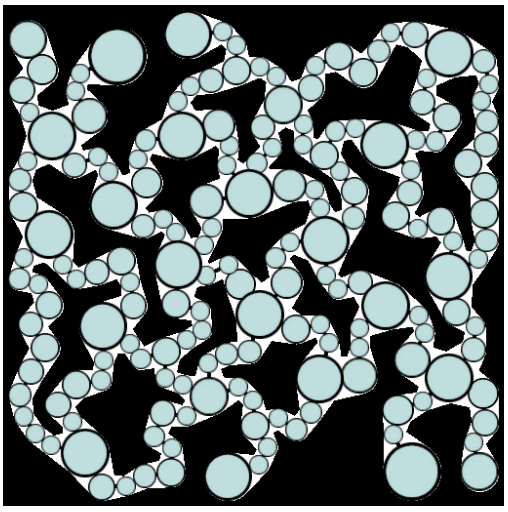
Figure 34: A graph layout and accessibility map generated based on the Neon Violet Square test case from Figure 24.

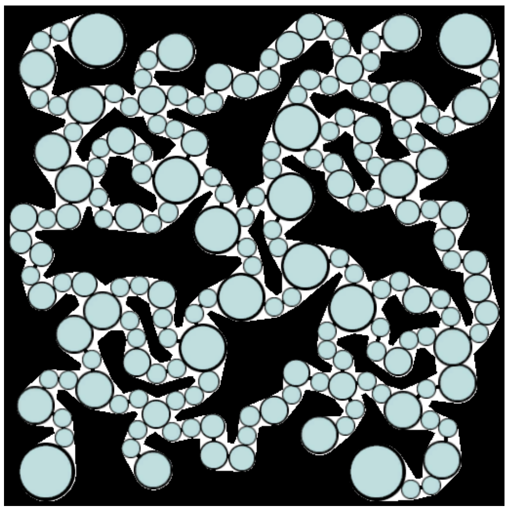
Figure 35: A graph layout and accessibility map generated based on the Frost test case from Figure 25.

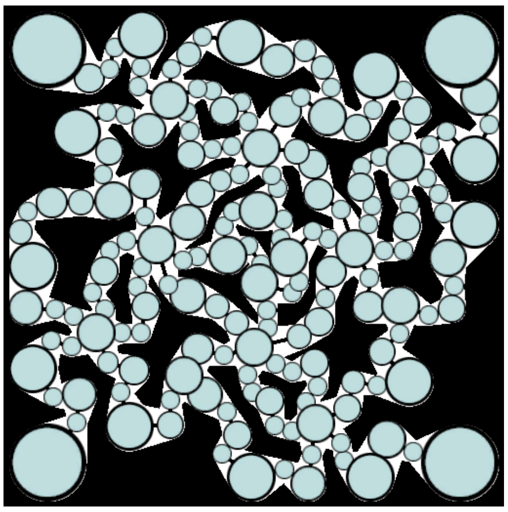
Figure 36: A graph layout and accessibility map generated based on the Whirlwind test case from Figure 26.

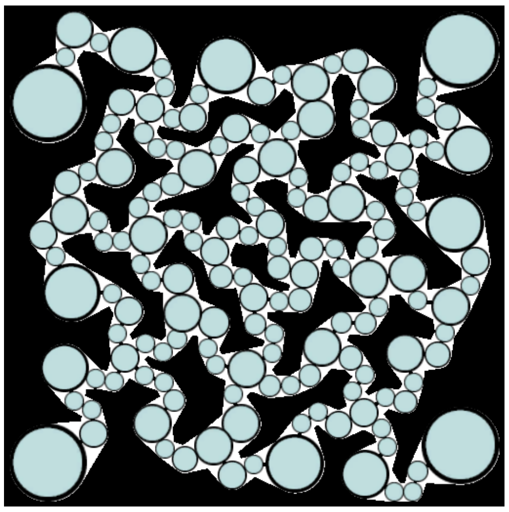
Figure 37: A graph layout and accessibility map generated based on the Cactus Valley test case from Figure 27.

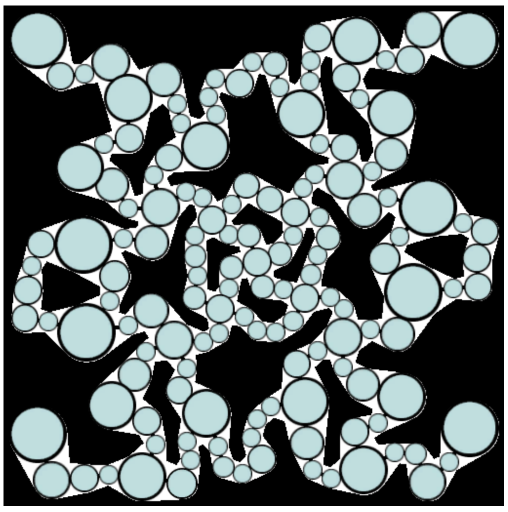
Figure 38: A graph layout and accessibility map generated based on the Deadwing test case from Figure 28.

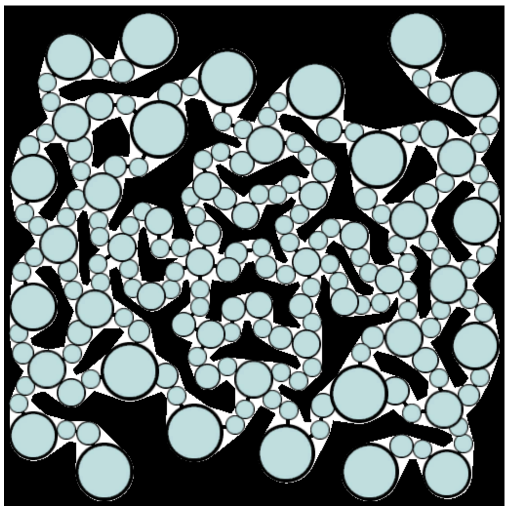
Figure 39: A graph layout and accessibility map generated based on the Invader test case from Figure 29.

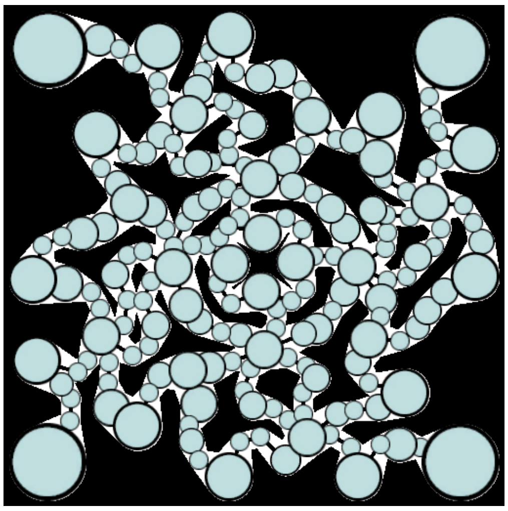
Figure 40: A graph layout and accessibility map generated based on the Whirlwind test case from Figure 26. The stiffness force multiplier was 1.0.

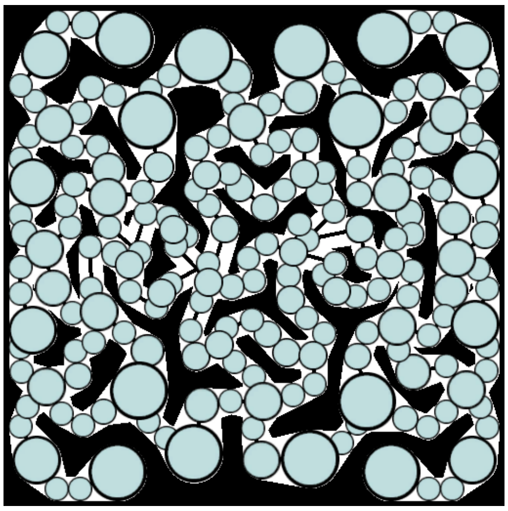
Figure 41: A graph layout and accessibility map generated based on the Invader test case from Figure 29. The corridor widths were 25.

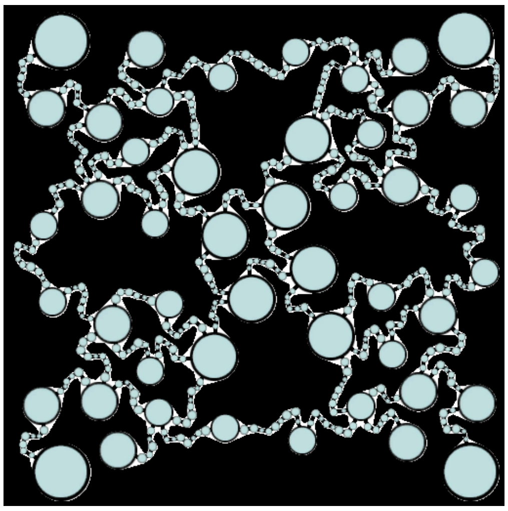
Figure 42: A graph layout and accessibility map generated based on the Frost test case from Figure 25. The corridor widths were 5.

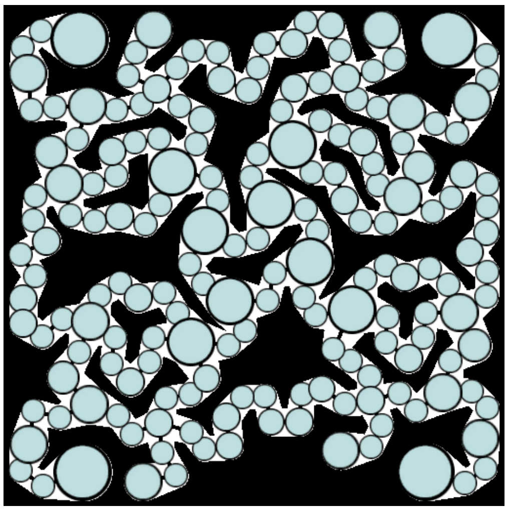
Figure 43: A graph layout and accessibility map generated based on the Frost test case from Figure 25. The corridor widths were 25.

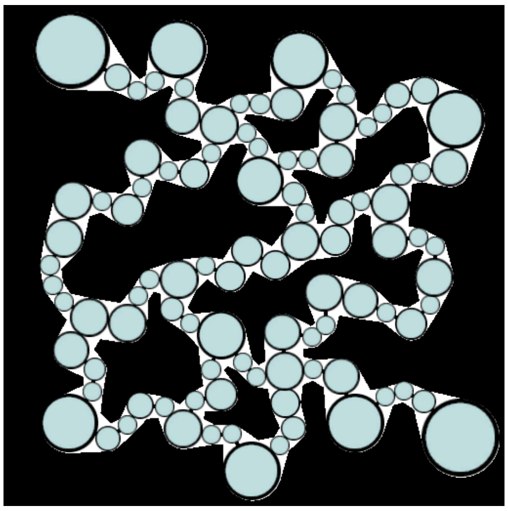

Figure 44: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21. The slack factor was 1.2.

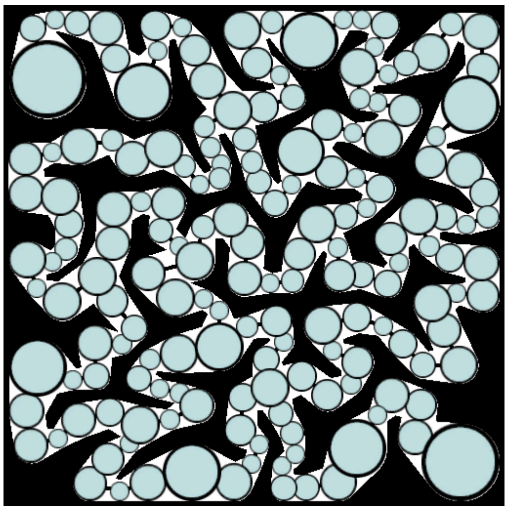

Figure 45: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21. The slack factor was 2.0.

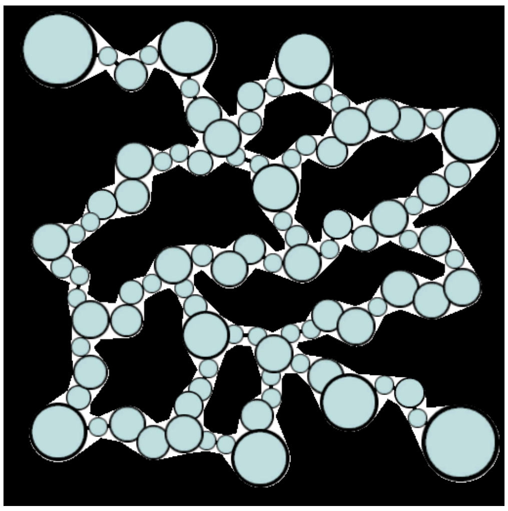

Figure 46: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21. The slack factor was 1.2. Constraint projections were disabled in this experiment.

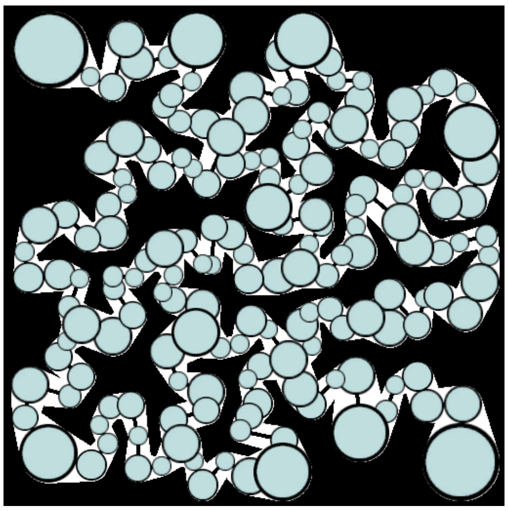

Figure 47: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21. The slack factor was 2.0. Constraint projections were disabled in this experiment.

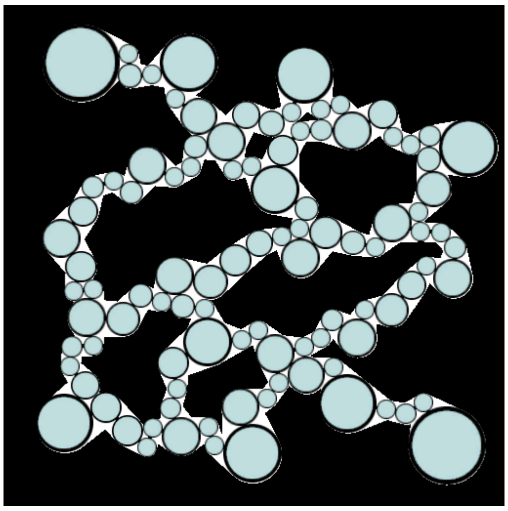
Figure 48: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21. The slack factor was 1.2. Repulsion radii were disabled in this experiment.

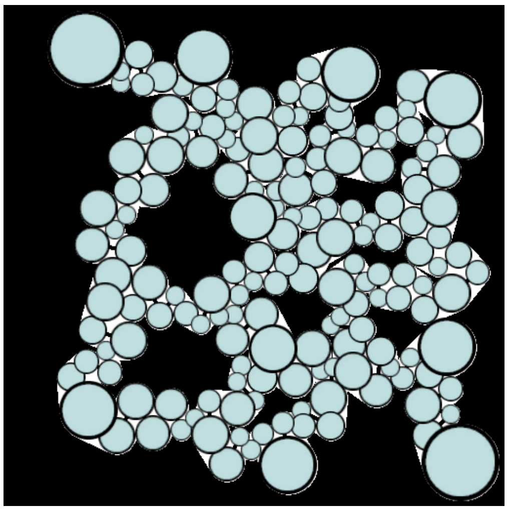
Figure 49: A graph layout and accessibility map generated based on the Acid Plant test case from Figure 21. The slack factor was 2.0. Repulsion radii were disabled in this experiment.

C Diversity Images

This section showcases the diversity of the maps produced in the various experiments we performed. As described in Section 4.1, each image is based on ten generated accessibility maps. The lighter a pixel's color, the more often the corresponding accessibility map cell was marked as accessible. Black cells were never accessible; white cells were accessible in all ten maps. Unless stated otherwise, the displayed results were obtained used a stiffness force multiplier of 0.01, a slack factor of 1.25, and a corridor width of 20.

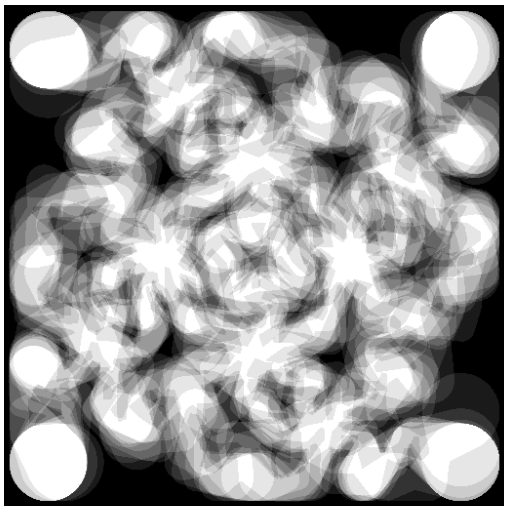

Figure 50: All ten accessibility maps for the Whirlwind graph in the experiment with varying stiffness overlaid on top of each other. The stiffness multiplier was 1 here.

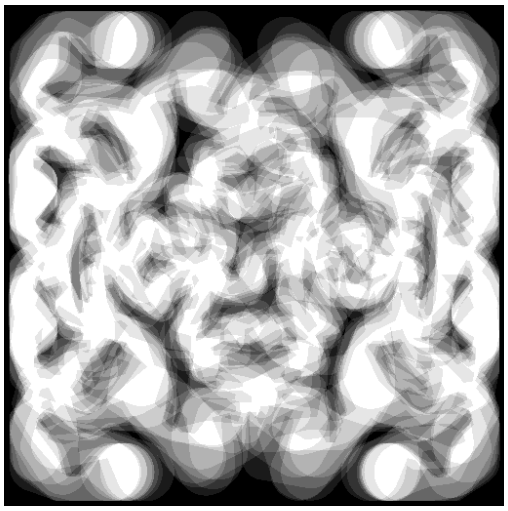

Figure 51: All ten accessibility maps for the Invader graph in the experiment with varying corridor width overlaid on top of each other. The corridor width was 25 here.

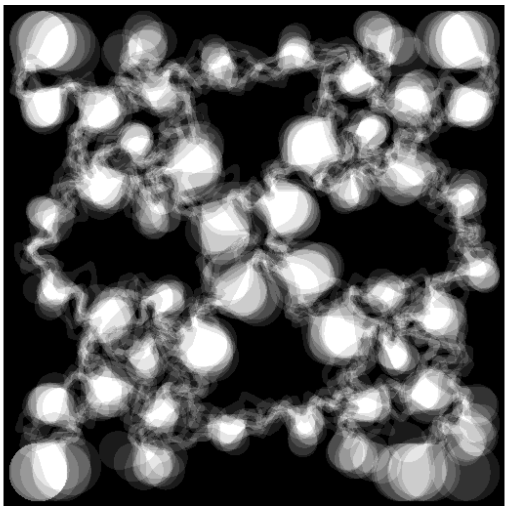

Figure 52: All ten accessibility maps for the Frost graph in the experiment with varying corridor width overlaid on top of each other. The corridor width was 5 here.

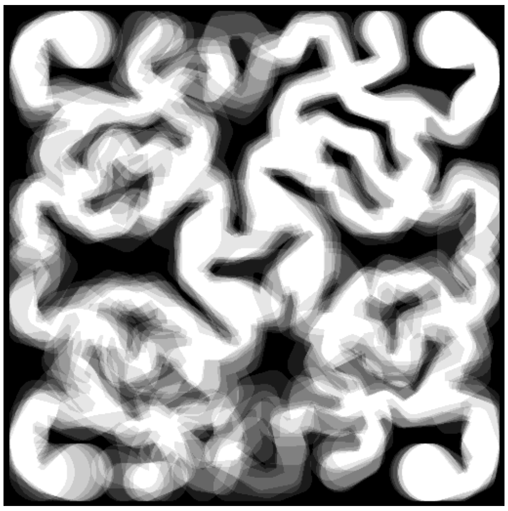

Figure 53: All ten accessibility maps for the Frost graph in the experiment with varying corridor width overlaid on top of each other. The corridor width was 25 here.

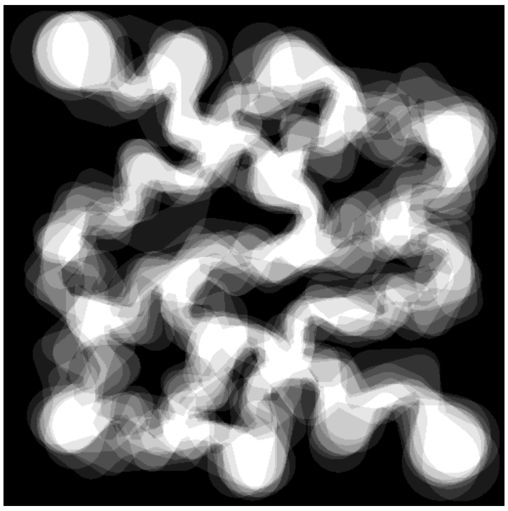

Figure 54: All ten accessibility maps for the Acid Plant graph in the experiment with varying slack factor overlaid on top of each other. The slack factor was 1.2 here.

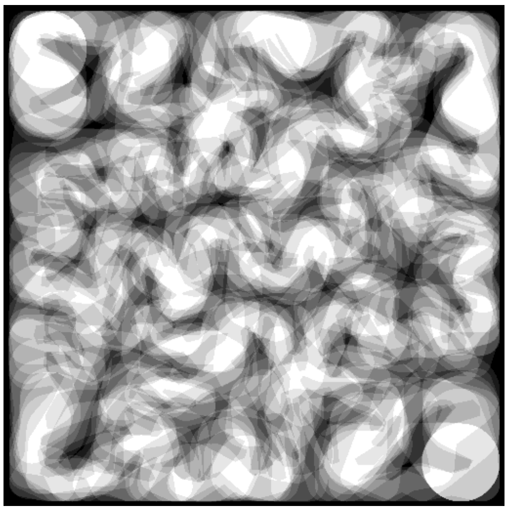

Figure 55: All ten accessibility maps for the Acid Plant graph in the experiment with varying slack factor overlaid on top of each other. The slack factor was 2.0 here.

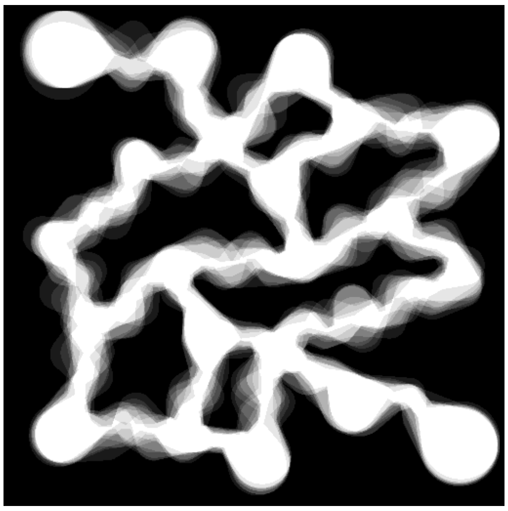

Figure 56: All ten accessibility maps for the Acid Plant graph in the experiment with varying slack factor and disabled constraint projections overlaid on top of each other. The slack factor was 1.2 here.

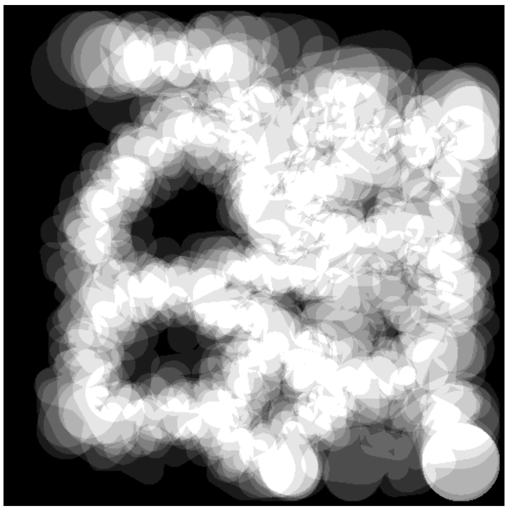

Figure 57: All ten accessibility maps for the Acid Plant graph in the experiment with varying slack factor no repulsion radii overlaid on top of each other. The slack factor was 2.0 here.

D Generated Terrains

In this section, we show a few renderings of some terrains that were generated by our method. The displayed results were obtained used a stiffness force multiplier of 0.01, a slack factor of 1.25, and a corridor width of 20. See Section 3.10 for more details on how these terrains were created.

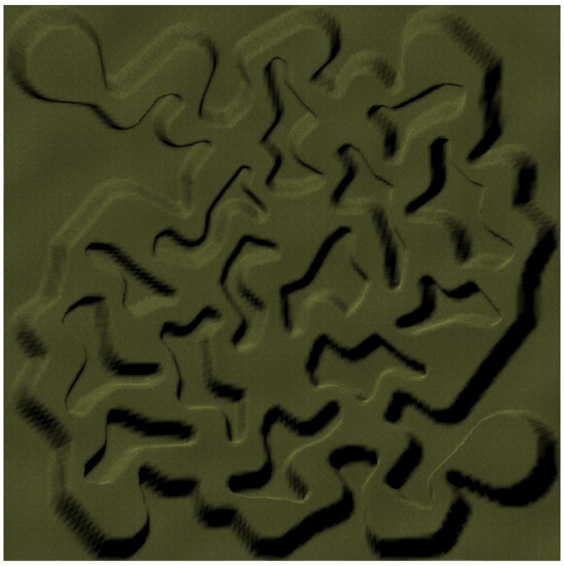

Figure 58: A view of the terrain generated based on the accessibility map of the Catalyst graph from Figure 32.

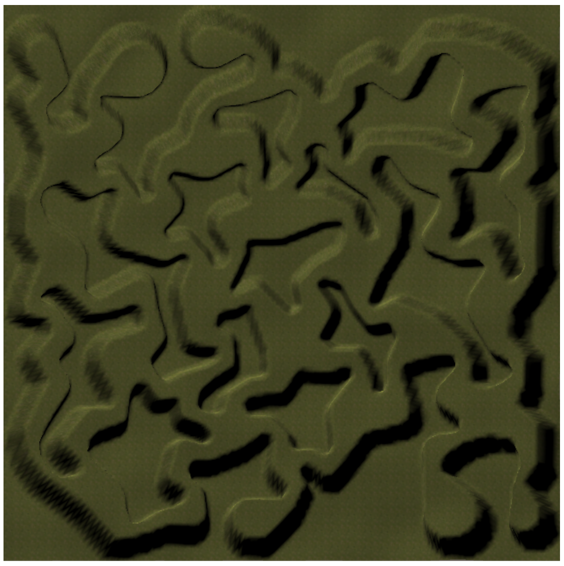

Figure 59: A view of the terrain generated based on the accessibility map of the Neon Violet Square graph from Figure 34.

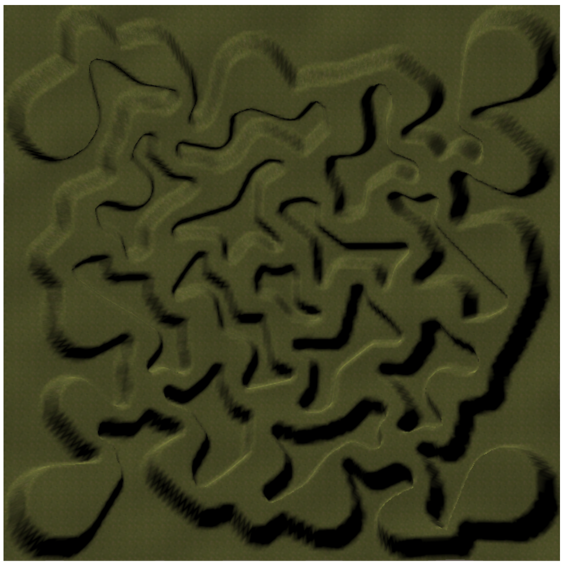

Figure 60: A view of the terrain generated based on the accessibility map of the Cactus Valley graph from Figure 37.

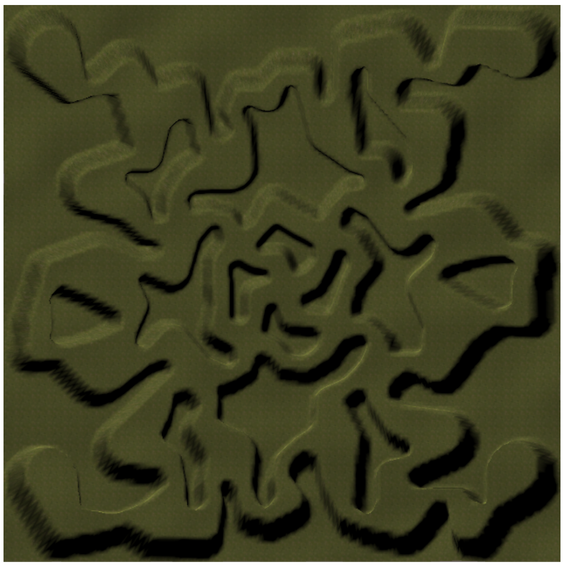

Figure 61: A view of the terrain generated based on the accessibility map of the Deadwing graph from Figure 38.

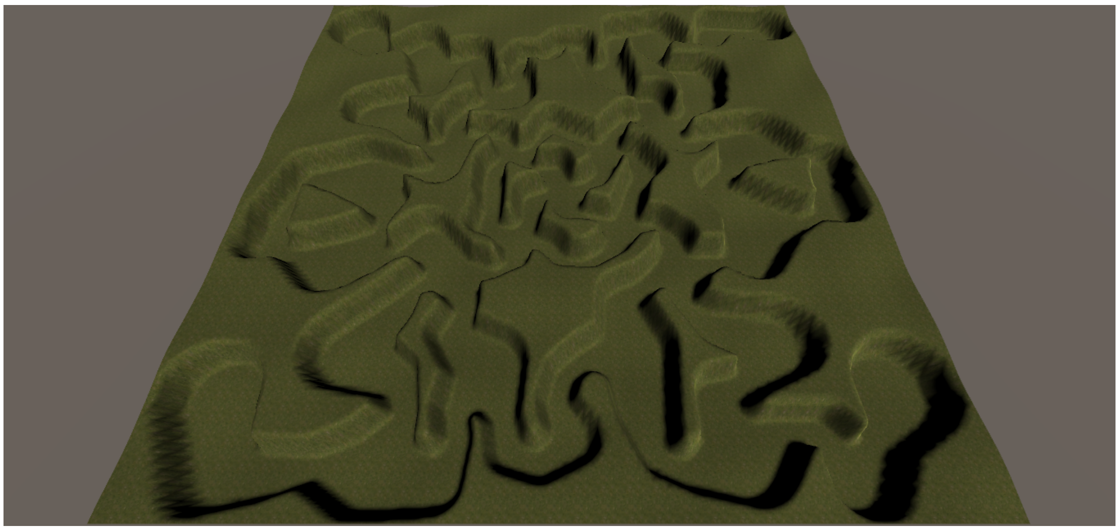

Figure 62: Another view of the terrain from Figure 61.

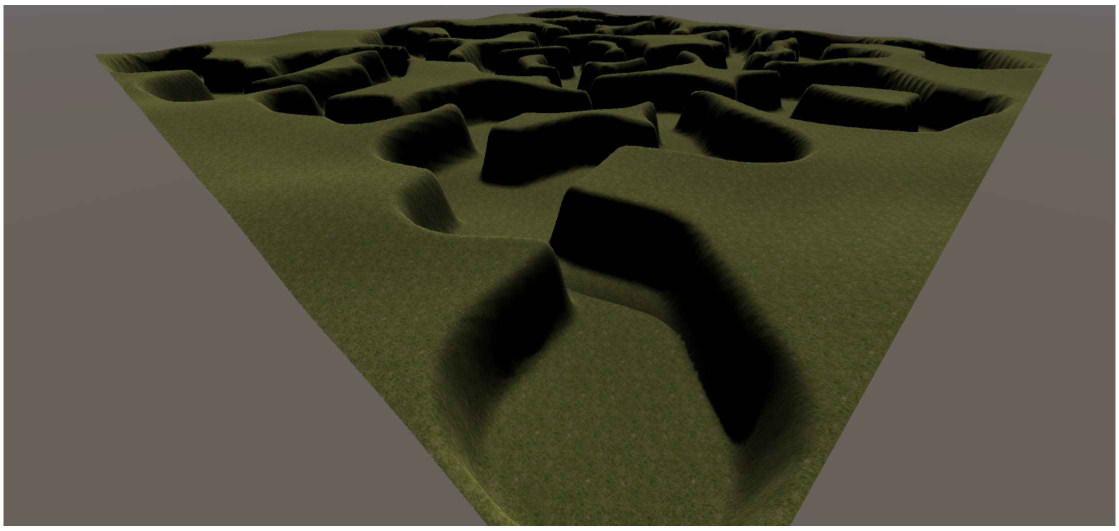

Figure 63: Another view of the terrain from Figure 61.

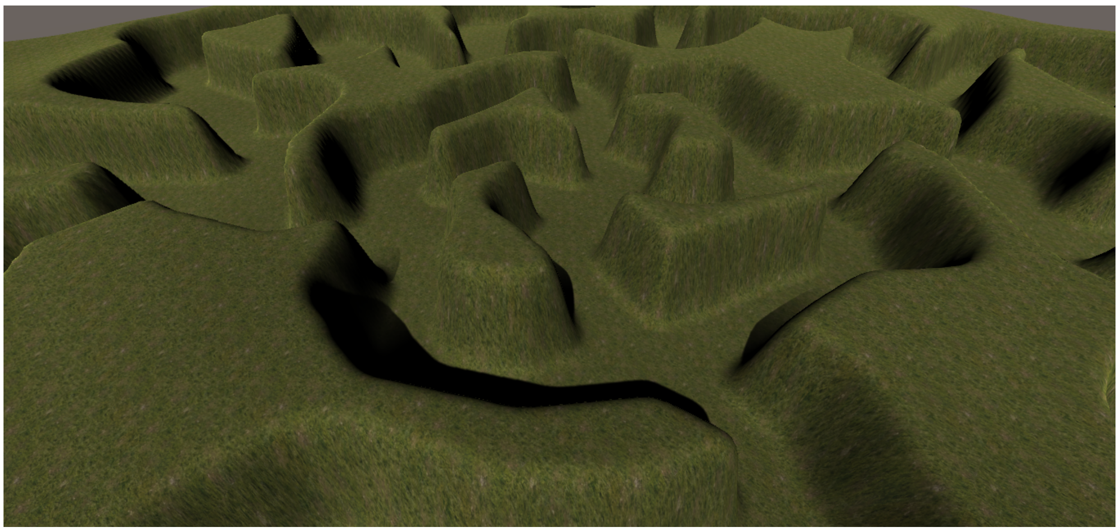

Figure 64: Another view of the terrain from Figure 61.

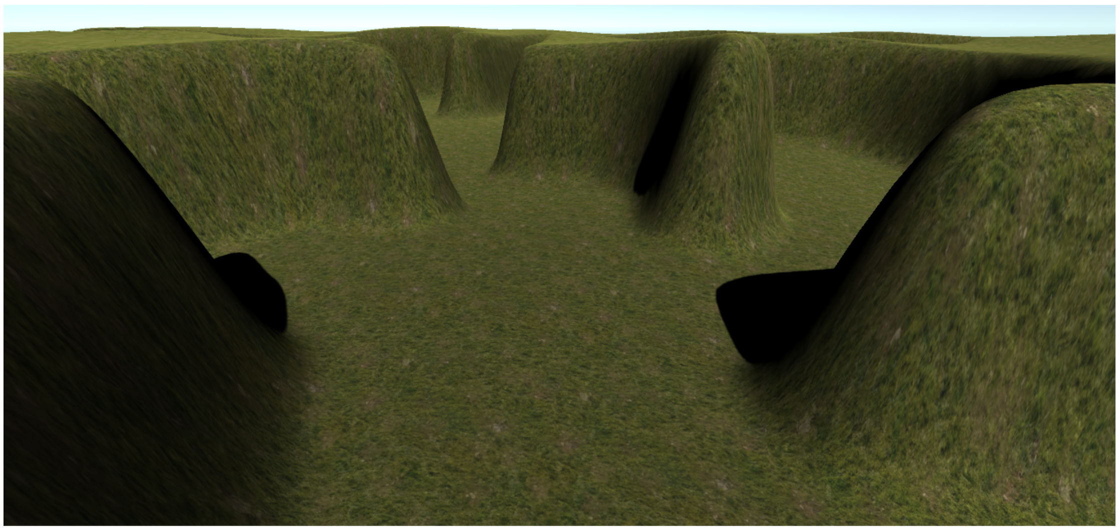

Figure 65: Another view of the terrain from Figure 61.