Aalto University
School of Science
Degree programme in Engineering Physics and Mathematics

Generating a Navigation Graph for Coastal Waters

Bachelor's Thesis
09.01.2019

Leevi Olander

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of bachelor's thesis

Author Leevi Olander

Title of thesis Generating a Navigation Graph for Coastal Waters

Degree programme Engineering Physics and Mathematics

Major Mathematics and Systems Analysis **Code of major** SCI3029

Supervisor Prof. Ahti Salo

Thesis advisor(s) M.Sc Tech. Juho Roponen

Date 09.01.2019 **Number of pages** 22+8 **Language** English

Abstract

This bachelor's thesis examines how a navigation graph can automatically be generated for coastal waters. Usually graphs are generated manually in game development for example; however, this is not applicable for coastal waters due to the amount of data. In other words, generating navigation graphs manually would entail on immense workload and therefore automatic generation is required.

Towards this end, relevant data is first downloaded and then converted to a more suitable format. The data is then processed via various methods, to produce a navigation graph. The parameters of the generation method are both analyzed and optimized by solving a large number of random path-finding problems. Furthermore, comparative analyses on the advantages and disadvantages of the developed method are presented. These results indicate that the generation process is valid and produces graphs that contain the necessary topological information.

In comparison with a graph which is converted from the input data directly, our method produces a graph whose performance is significantly better on several factors (e.g. pathfinding processing time) while others (e.g. coverage, path length) are only marginally worse. All algorithms and methods in this thesis were implemented in a software with a graphical user interface. This application makes it easy to visually inspect and validate the results.

Keywords navigation, graph, pathfinding, coast, A*, Dijkstra, software

Aalto-universitetet
Högskolan för
teknikvetenskaper

Aalto-universitetet, PB 11000, 00076 AALTO
www.aalto.fi

Sammandrag av kandidatarbetet

Författare	Leevi Olander
Titel	Generering av en navigeringsgraf för kustvatten
Examensprogram	Teknisk fysik och matematik
Huvudämne	Matematik och systemvetenskaper
Huvudämnets kod	SCI3029
Ansvarig lärare	Prof. Ahti Salo
Handledare	DI Juho Roponen
Datum	09.01.2019
Sidantal	22+8
Språk	Engelska

Sammandrag

Detta kandidatarbete undersöker hur en navigeringsgraf kan genereras automatisk för kustvatten. Ofta genereras grafer för hand, till exempel i utveckling av datorspel. Detta är dock inte lämpligt för kustvatten på grund av mängden av data. Det vill säga, generering av navigeringsgrafer för hand skulle leda till enorma arbetsmängder, vilket är varför automatisk generation är nödvändigt.

För att avklara detta nedladdas relevant data, varefter den konverteras till ett mera passande format. Datan blir därefter processerad med hjälp av diverse metoder, vilka slutligen producerar en navigeringsgraf. Genereringsmetodens parametrar analyseras och optimeras genom att lösa ett stort antal av slumpmässiga vägsökningsproblem. Därefter presenteras för- och nackdelar för den utvecklade metoden. Resultaten indikerar att genereringsprocessen är giltig och att navigeringsgraferna innehåller den väsentliga informationen.

I jämförelse med en graf som konverteras direkt från den ursprungliga datan, producerar den utvecklade metod grafer vars prestanda är betydligt bättre i flera aspekter (t.ex. processeringstiden för vägsökningsproblem), medan andra aspekt (t.ex. utsträckning, längden av den funna rutten) försämras marginellt. Alla algoritmer och metoder i detta arbete implementerades i mjukvara med ett grafiskt användargränssnitt. Mjukvaran gör det enkelt att visuellt inspektera och validera resultaten.

Nyckelord navigering, graf, vägsökningsproblem, kust, A*, Dijkstra, mjukvara

Contents

1 Introduction

An important aspect of warfare is to remain undetected by the enemy for as long as possible, and as such an important interim goal is to choose a good route from one location to another. Movement can be modeled by using a weighted graph, in which the weights correspond to the risk of traversing through a specific area. However, the graph can not be too dense, because this would require excessively costly computations. Thus, a critical sub-objective is to generate a graph that models a given area precisely and sparsely.

Graphs have long been a subject of interest and have many practical applications such as navigation systems, agent simulations and traffic direction. However, no single method has been proven superior for automatic graph generation. For example Yang & Worboys (2015) have presented a method for indoor graph generation. Similar approaches work well for graph generation when the modeled area consists of clearly defined and relatively confined subareas, such as rooms and corridors. However, in the case of coastal waters, this would simplify the graph too much and make it difficult to apply weights to the graph, thus rendering it unusable for accurate representation of the waters.

A different method that aims to solve the same task is to convert an area into a navigation mesh, a concept which was introduced by Snook (2000). Several navigation graph generation techniques as well as their pros and cons are discussed by Leonard (2014). In uniformly weighted meshes, the shortest path between two points can quite easily be found, but the weighted region shortest path problem remains unsolved (De Carufel et al., 2014). Due to the unsolvability, navigation meshes are usually converted to graphs, but the automated conversion process often results in unnatural or even untraversable paths between nodes.

A common method to generate a navigation graph is to triangulate the area and then to refine the triangulation (Shewchuk, 2002), this however relinquishes more thorough customization. The refinement produces vertices in a seemingly random fashion and unless there are extra constraints, may ultimately result in too dense a graph. On these grounds navigation graphs are usually created manually in, for example, game development.

This thesis studies how a navigation graph can be generated for coastal waters. Manual generation is not feasible for our purposes due to the sheer amount of data and as such an alternative automatic method is presented. All algorithms and methods presented in this paper have been implemented in an application with a graphical user interface.

2 Methods

2.1 Data

We chose to use the data available at the Finnish Transport Agency (Liikennevirasto, 2018), because it was free and contained all the necessary data. This data is provided in shapefile format, which is a widely used format for geospatial vector data in geographic information systems (Library of Congress, 2017). When downloading a grid cell all intersecting depth areas are included in the resulting data, meaning that extra data is generally downloaded. This in turn leads to a situation where some data is included several times, which is a problem that needs to be managed.

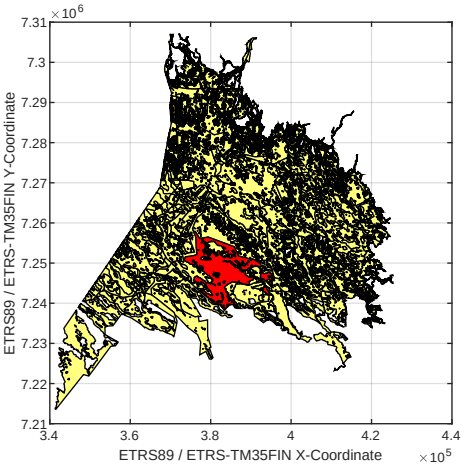


Figure 1: The data in one grid cell.

Field	Value
HISOID	FI428000005150976
HGHTLAKE	-
MAXDEPTH	20
MINDEPTH	10
TYPEDPR	1
CDATE	2018-01-23
NTMENTRY	-
YEARSWEEP	-
IRROTUS_PV	2018-06-11T10:14:23

Table 1: The fields of the red polygon in the output data.

The depth data provided by the Finnish Transport Agency is not explicitly defined, because each feature (a polygon with data values), presented in Figure 1, has a minimum and a maximum depth value. Table 1 shows all available data for one feature. We chose to extract only the minimum depth value, because it guarantees that the area can be safely traversed.

2.1.1 Data Format

Vector data formats are not practical for storing continuous data (Gisgeography, 2018), and because the depth of the sea is continuous, we first rasterize

the data to the ESRI ASCII Raster format (Library of Congress, 2017). Rasterized data has other advantages as well, for example mathematical operations are usually fast. However, there are disadvantages as well. File size can be a problem, and the fact that a large area that consists of exactly the same data must be represented cell by cell, rather than just store the boundary, unlike with vector data formats.

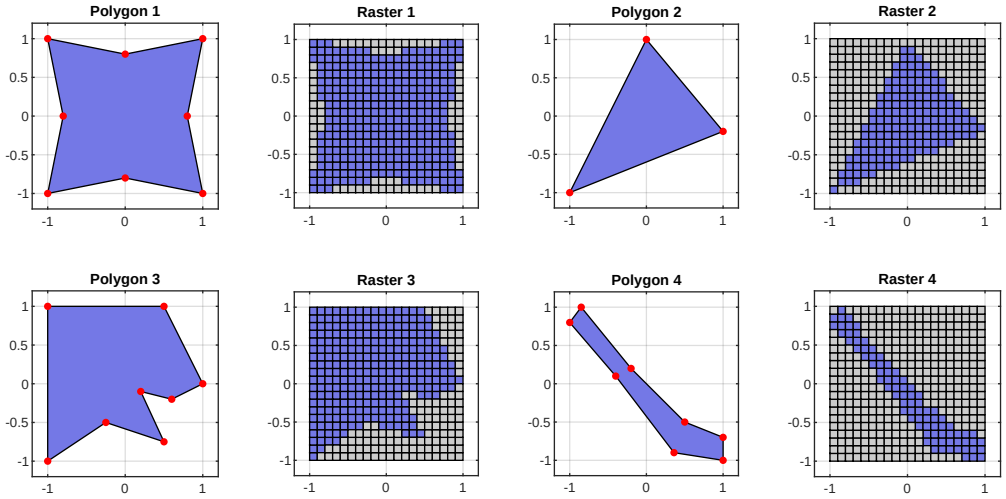


Figure 2: Rasterization results of various polygons. The red dots mark the vertices of each polygon, the blue represents data and the grey cells represent empty cells.

When rasterizing a vector file the output will always be in a rectangular form, which is attained by adding cells with no value to the resulting file. The described phenomena is visualized in Figure 2. Thus, it is not feasible to rasterize all of the vector data at once if the vectors do not define a form with an area that closely resembles the area of its bounding box. A better approach is to rasterize the data in parts that resemble rectangles and thus reduce wasted space. One also has to take into account the fact that the input files do not share a unified relative coordinate system. This can be problematic because the objective is to generate a graph for all of the data, not just for one file.

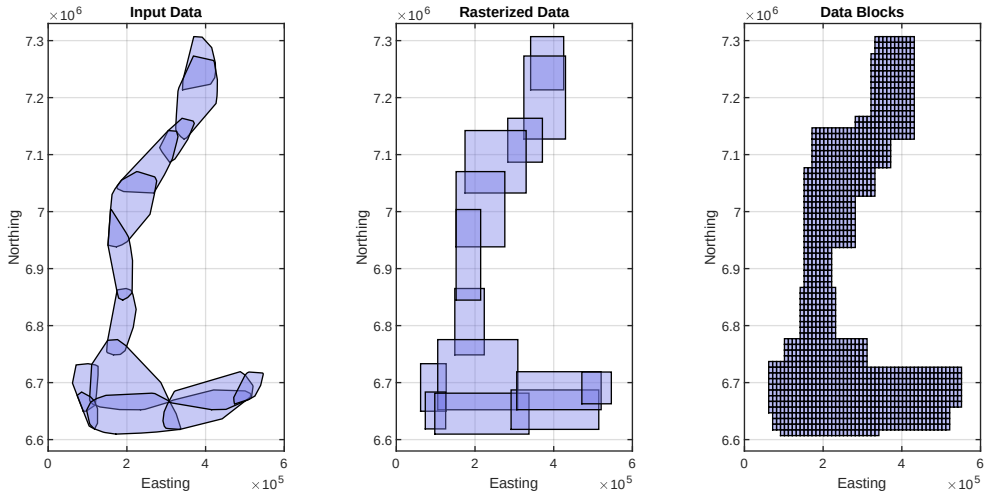


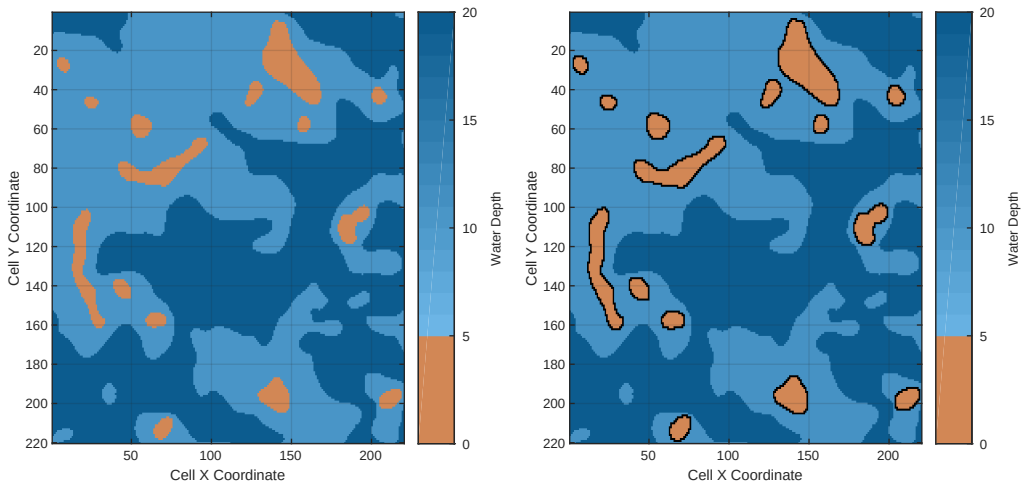
Figure 3: The different phases in data management. Each purple polygon represents one file.

These problems can be solved by splitting the input data in blocks of uniform size and by choosing a common origin for them all. Figure 3 shows how the data is rasterized and then split from its original state to blocks. The splitting comes with at least the following additional benefits: the graph generation can be parallelized to some extent as each block is its own independent entity and graph generation parameters can be optimized for small local problems rather than for a big global problem. Overall memory usage will also decrease, because not everything has to be loaded at the same time. On the downside we have to add empty cells to get a clear relative coordinate system, which generally increases the disk space required. However, the increase in required disk space is often marginal if the block size is chosen with care.

To use computer resources efficiently, the splitting is done so that we minimize the amount of redundant data kept in memory at any given time. A simple but effective approach is to read the number of rows that one block needs and then to split the columns accordingly. After writing the blocks to a file located on the hard drive, the process is repeated until all data has been processed. This method uses significantly less memory than the naïve approach of reading the whole original data at once and then writing the blocks one by one. The duplicate data makes it necessary to handle coordinates that are referenced by more than on file with care. This can be achieved by cross-referencing the minimum depth value at any given coor-

dinate found in any of the original rasterized input files and then validate the data. Of the cross-referenced values it is logical to choose the maximum value to preserve the accuracy of the data, because the rasterization process adds extra cells with zero depth and these added cells do not represent the original data.

2.2 Graph Generation



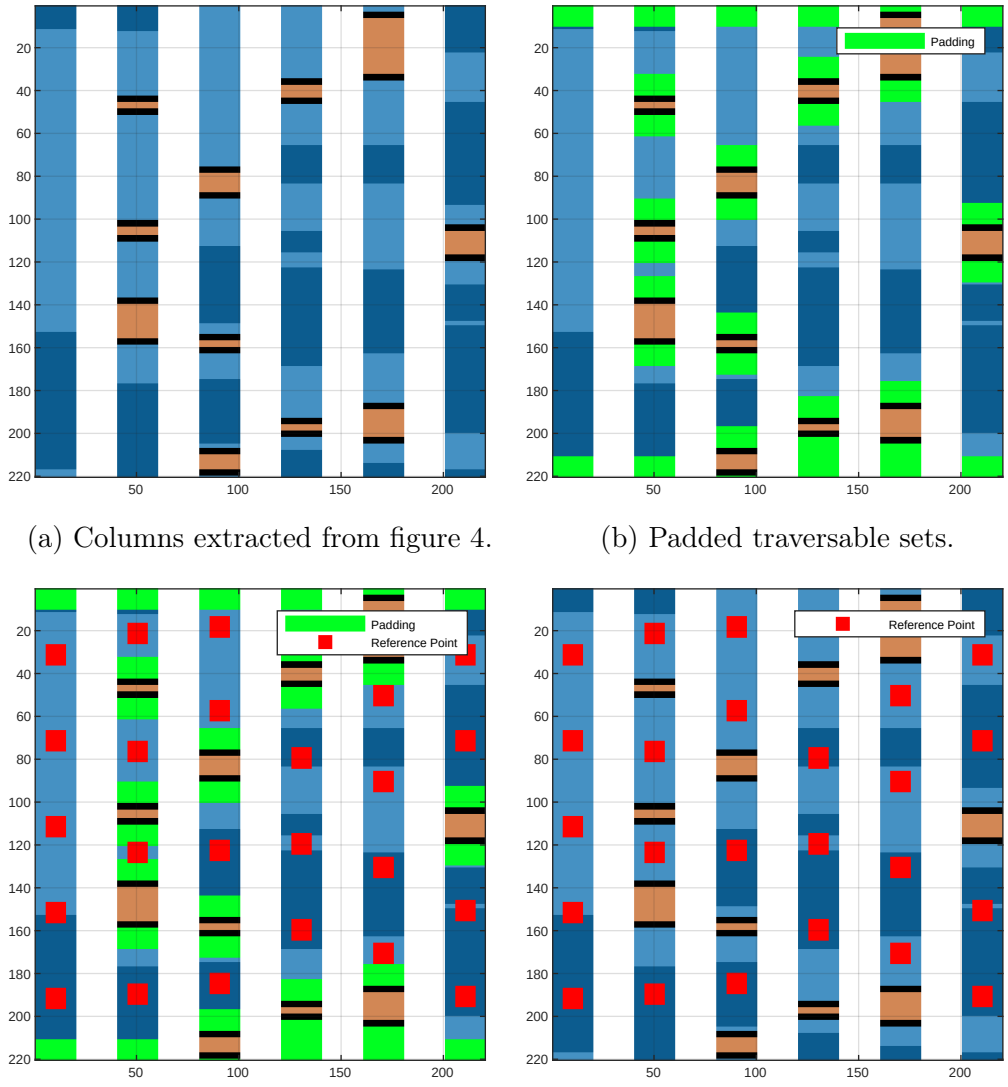
(a) Unbuffered data.

(b) Buffered data.

Figure 4: Visualization of the depth data. Blue symbolizes traversable, brown untraversable and black buffered areas, the black cells are also untraversable.

When generating the graph for a specific block, we split the procedure in three different sections: areas which can be traversed, areas that can not and finally the combination of these two. To decide what is traversable and what is not, we must define a parameter for the minimum depth that is allowed, **MinDepth**. Thereafter cells that have a greater or equal depth to the value specified by **MinDepth** get marked as traversable and the rest as untraversable. We also define a buffer parameter, **BufferRadius**, which marks all cells within a given distance from any originally untraversable cell as untraversable. Figure 4 shows the effect of buffering.

2.2.1 Traversable Areas



(a) Columns extracted from figure 4.

(b) Padded traversable sets.

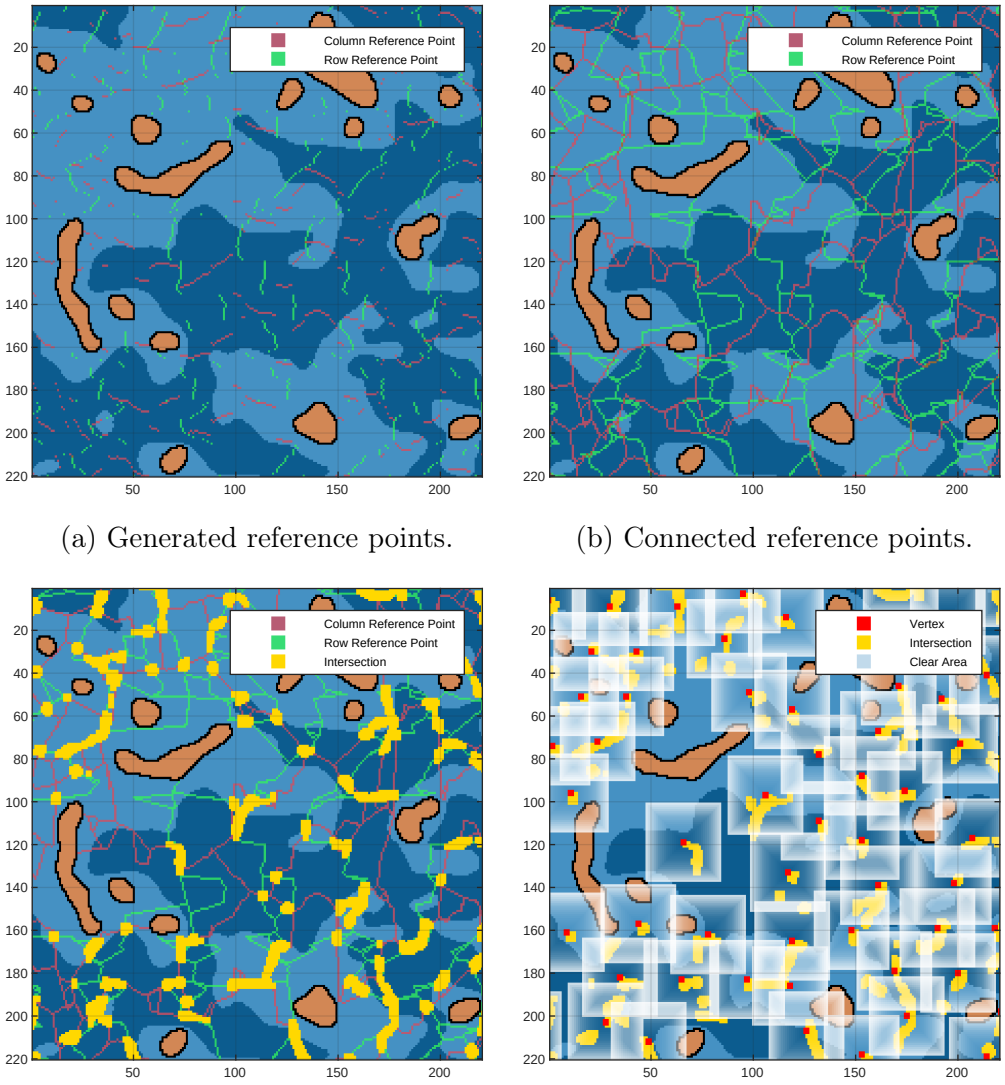
(c) Reference points.

(d) Final result.

Figure 5: Visualization of the steps in the reference point generation process along the y-axis.

The traversable areas have to be converted to a reduced graph without losing too much relevant information. As show in Figure 5, this can be done as follows, by first splitting the block into columns and then extracting continuous traversable sets from within each column. All sets are then padded from both

ends with a value according to the parameter **Padding**. The padding further limits the valid space for reference points, so that we may actually define how close a vertex can be placed to an untraversable area. This is then followed by placing linearly spaced reference points within each set. The number and placing of these reference points are defined by the parameters **Distance**, which describes the distance between two reference points in a traversable set and the value of **Padding** in accordance to the pseudo code presented in attachment A. The same process is then repeated for rows.



(a) Generated reference points.

(b) Connected reference points.

(c) Reference point intersections.

(d) Vertices and their clearing areas.

Figure 6: Visualization of the steps in the reference point generation process.

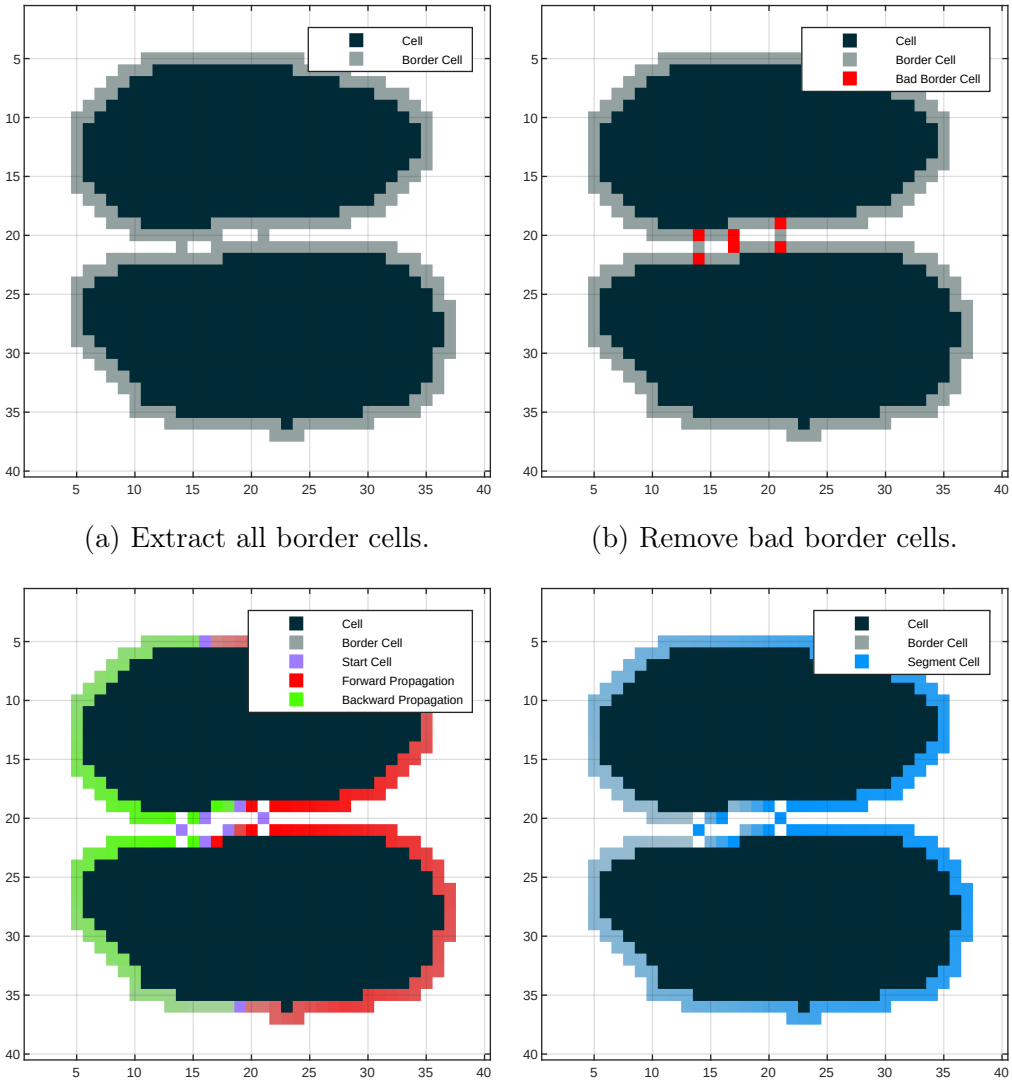
The process of converting traversable areas to vertices is show in Figure 6. Once the reference points have been calculated, they are connected with the Bresenhams Line algorithm (Bresenham, 1965). Then, intersections are calculated, and a vertex is placed at each of them. In the calculation of intersections, the parameter **IntersectionRadius** defines whether there is an intersection or not. In practice, this means that even if a true intersection has

not been found at a specific location we may treat is as such. Furthermore, the parameter **ClearRadius** is introduced to clear vertices within a distance from another vertex. This is done so that uneccessary vertices get omitted.

2.2.2 Untraverasble Areas

To ensure that the graph preserves the main structure of the original data, we want to generate a set of vertices and edges around each untraversable area at some distance from it. Generally, outward polygon offsetting would be applied, but we do not want narrow passes to be blocked off, which is why a slightly different approach is needed. First, we define which cells belong to which area. A common solution is to either apply the Flood Fill algorithm or the Scanline Fill algorithm (Vandevenne, 2018). Both algorithms give the same output on a two-dimensional grid, but the Scanline Fill algorithm is significantly faster of the two, which is why it was implemented.

Next, we need to trace the contour of each area. Several algorithms have been invented to solve this, such as the Square Tracing algorithm, Moore Neighborhood tracing and Theo Pavlidis' algorithm (Ghuneim, 2000). None of these algorithms guarantee that each border cell is visited exactly once. Therefore, a new algorithm, Segmented Border Tracing (**SBT**), was developed. It preserves the order of the contour cells and fulfills the visiting criterion. The pseudocode for the **SBT** is in Appendix B.



(a) Extract all border cells.

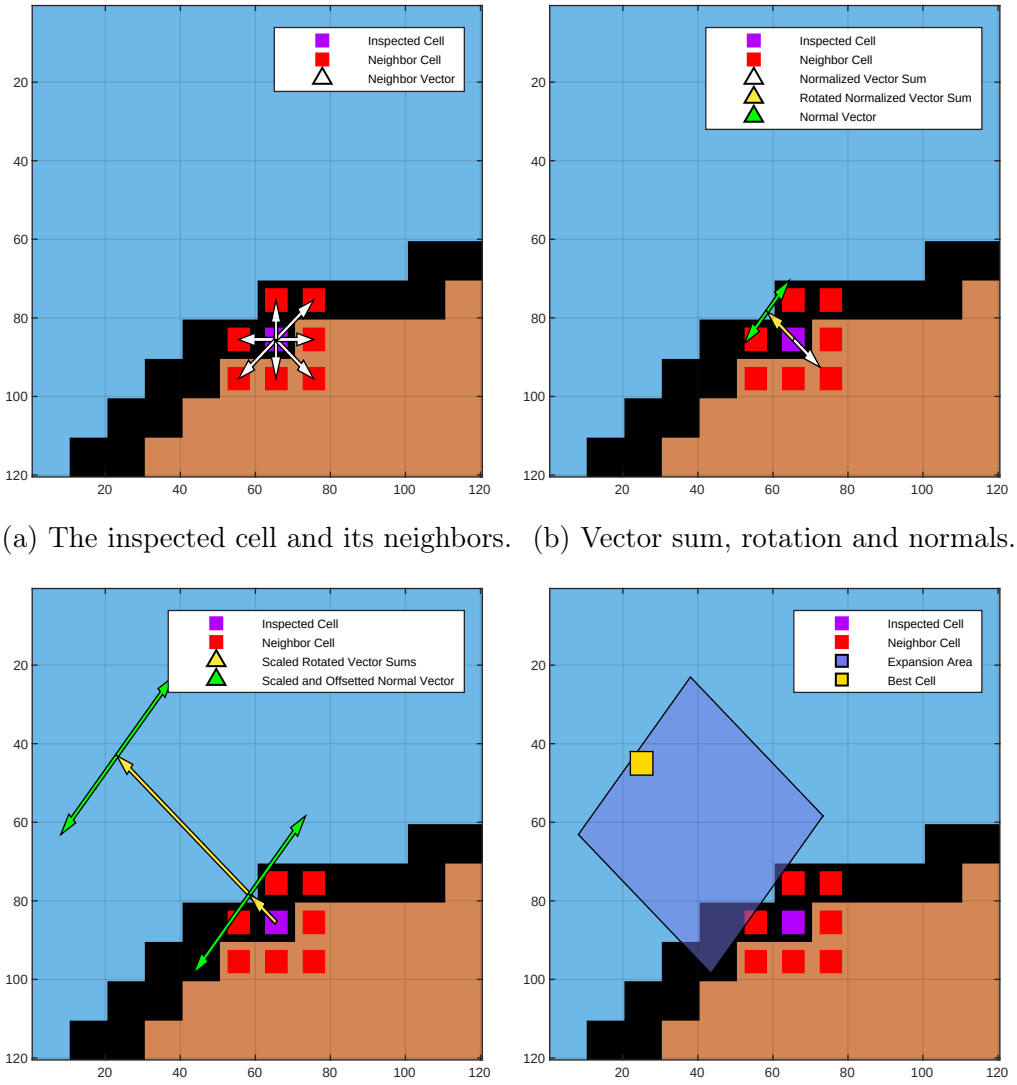
(b) Remove bad border cells.

(c) Propagate backward and forward.

(d) Combination of the propagations.

Figure 7: Visual illustration for the Segmented Border Tracing algorithm. The color intensities denote the order of the cells.

The visualization of the **SBT** is in Figure 7. The algorithm outputs a set of continuous border segments and removes border cells that have more than two adjacent neighboring border cells. These cells make it difficult to find the border as they would sometimes be visited several times. After the border has been successfully traced into continuous segments, it is expanded. The expansion is made so that narrow passages are preserved.



(a) The inspected cell and its neighbors. (b) Vector sum, rotation and normals.

(c) Vector scaling. (d) Search for optimal expansion cell.

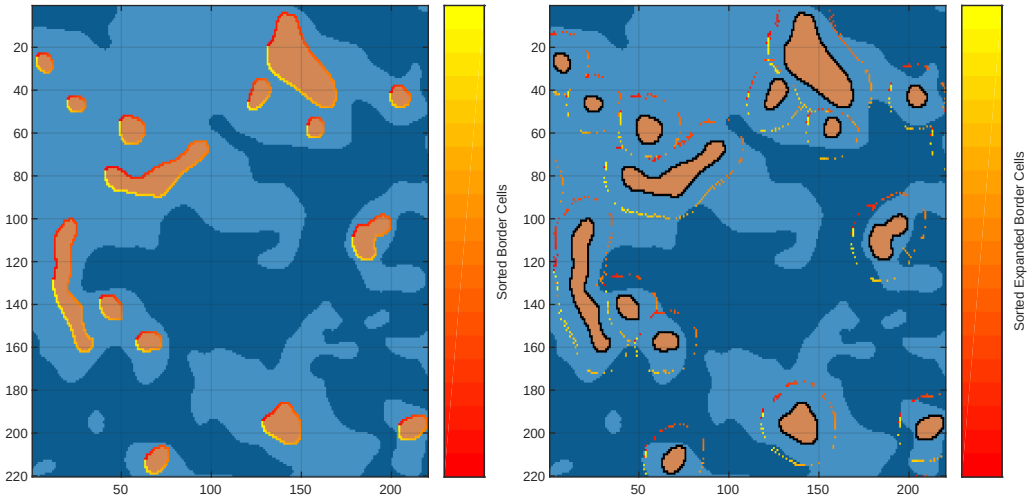
Figure 8: Visual explanation for the border expansion procedure for one cell.

For each cell that is to be expanded, we must decide which cells in its near vicinity are taken into account. This is handled by the parameter **DirectionRadius**, which defines the maximum distance from the origin cell to other cells. We also need to define a parameter for the maximum expansion radius, **ExpansionRadius**, which specifies how far a cell can maximally be expanded. The pseudo code for the expansion algorithm is expressed in Ap-

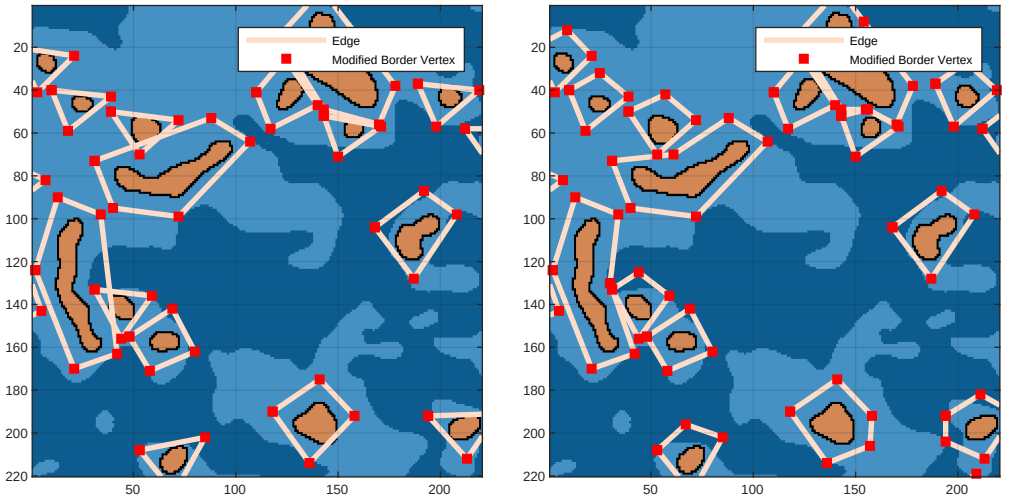
pendix C and its visualization is in Figure 8. The optimal expansion cell is the cell within the expansion area that has the maximum distance to any untraversable cell.

When all cells have been expanded the next step is to simplify them. A suitable implementation is the Ramer Douglas Peucker (**RDP**) algorithm (Douglas & Peucker, 1973), which is used to simplify polylines. The **RDP** algorithm takes as input a set of vertices and a tolerance value. The tolerance value specifies how far any vertex can maximally be from the simplified lines. The parameter **ToleranceValue** is used to represent it.

After the simplification process has been applied a new problem arises. The **RDP** algorithm does not conserve the traversability between simplified vertices. Due to the fact that **RDP** always produces a subset of the vertices that were given as input and because the border is sorted, the problem has a straight-forward fix, which is presented in Appendix D.



(a) Sorted border cells. (b) Expanded border cells.

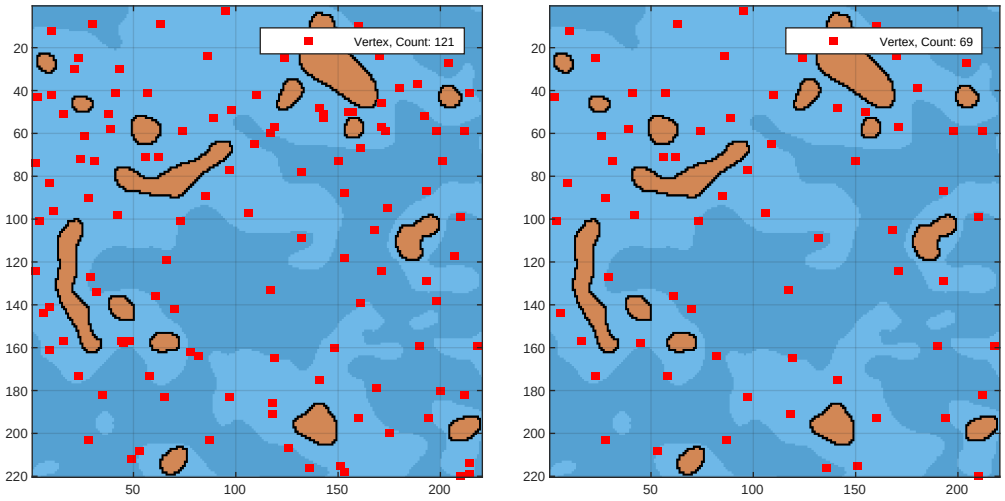


(c) Simplified border vertices. (d) Modified border vertices.

Figure 9: The different phases in untraversable border processing.

After the simplification and the correcting procedures have been completed, we add additional border vertices so that the distance between two subsequent vertices does not exceed the distance defined by the parameter **SimplifiedDistance**. This is achieved by inserting linearly interpolated vertices between two consecutive vertices that are too far away from each other. Figure 9 depicts the whole process starting with the sorted segments, then expansion, simplification and finally modification.

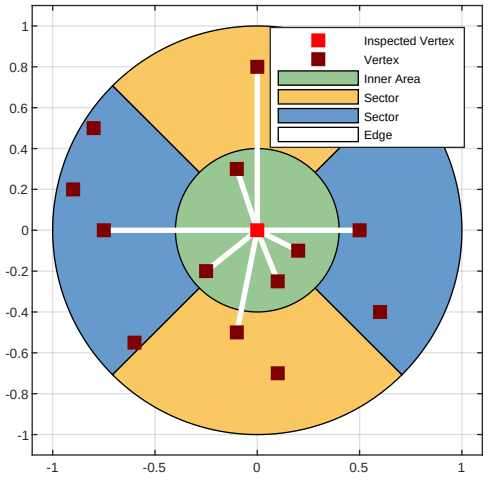
2.2.3 Combination



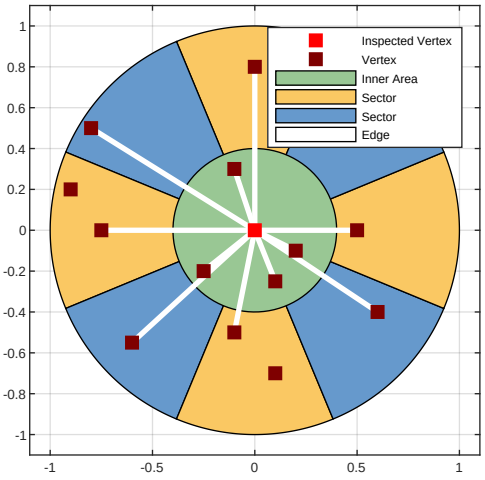
(a) Combined vertices. (b) Combined and merged vertices.

Figure 10: The effect of merging vertices.

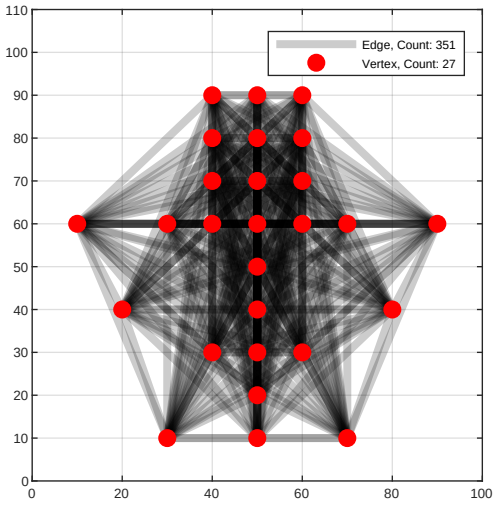
When the vertices for both untraversable and traversable areas have been processed separately, the results are combined. We join all vertices together and then merge those that are within the radius defined by the parameter **MergeRadius**, subject to the condition that any vertex defined by the expanded border may not be modified. This guarantees that narrow passages are not omitted. Then we define another radius, **BorderMergeRadius** that only affects border vertices. This allows some border vertices to be omitted, but only if they are close to another border vertex. The merging process is shown in Figure 10.



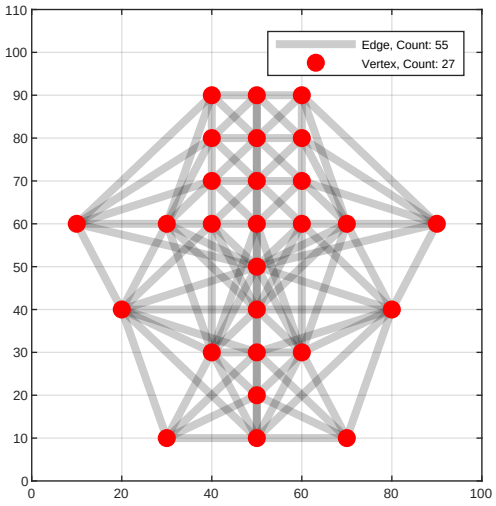
(a) **NDC** with **SectorCount** as 4.



(b) **NDC** with **SectorCount** as 8.



(c) Simple connecting method.



(d) Output of **NDC** with **SectorCount** as 8.

Figure 11: The **NDC** algorithm with two different parameter configurations along with a comparison of the output generated by the simple connecting method and **NDC**.

Once the vertices have been merged, they are connected to create a graph. A natural method would be to connect a vertex to all connectable vertices within a given radius; however, it turns out that this is not a good approach due to the number of unnecessary edges produced. A is to create and ap-

ply the Nearest Division Connect (**NDC**) algorithm. The **NDC** defines two radii: one which functions as the upper limit for distances between connectable vertices, modeled by the parameter **OuterConnectRadius**, and one smaller radius, **InnerConnectRadius**, that will allow a vertex to connect to all connectable vertices within it. Then we divide the disk into equally sized sectors, where the number of sectors is defined by the parameter **SectorCount**. After the division, the inspected vertex is connected to the closest connectable vertex in each sector. Figure 11 illustrates the **NDC** algorithm and compares it to a method in which each vertex is connected to all vertices within a given radius.

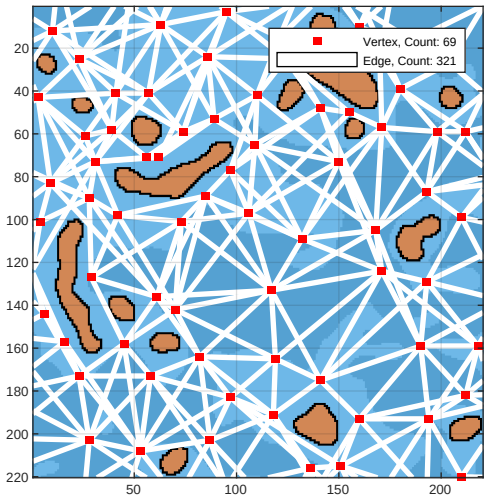


Figure 12: A representative result of the graph generation process.

The model is now complete, and the final output is shown in Figure 12. After generating a graph separately for all blocks, they need to be combined to form one large graph. The connecting is simple as all blocks are given their own relative x and y coordinates, making it possible to connect the vertices of one block to all blocks around it, thus resulting in a graph that contains all input data.

2.3 Parameter Optimization

Parameter	Section	Optimize
MinDepth	General	No
BufferRadius	General	No
Distance	Traversable Areas	Yes
Padding	Traversable Areas	Yes
IntersectionRadius	Traversable Areas	Yes
ExpansionRadius	Untraversable Areas	Yes
DirectionRadius	Untraversable Areas	Yes
ToleranceValue	Untraversable Areas	Yes
SimplifiedDistance	Untraversable Areas	Yes
ClearRadius	Combination	Yes
MergeRadius	Combination	Yes
OuterConnectRadius	Combination	Yes
InnerConnectRadius	Combination	Yes
SectorCount	Combination	Yes
BorderMergeRadius	Combination	Yes

Table 2: The parameters used in the graph generation process.

All parameters used to construct the graph are listed in Table 2. The total number of parameters is 15, of which most will be optimized while others will not. For example, the parameter **MinDepth** is not optimized because its value is dictated by the purpose of the navigation graph. The same applies for **BufferRadius**, which describes how close to untraversable areas we are willing to go.

When the model with the parameters has been constructed, the parameter values should be optimized to create a graph that uses as little disk space as possible and performs significantly better than the graph based on the original unmodified data without losing important information. These values can either be optimized for one block at a time or for the whole graph, but we found it preferable to optimize for each block independently, because this decreases processing time and makes it possible to choose different parameter values for substantially different blocks. Finding the best parameters is an integer optimization problem whose complexity grows extremely fast due to the number of parameters. Thus, by measuring the effect of different parameters on the graph generation process, we can narrow down the search range by omitting parameters with inconsequential impact. Here we chose to use a slightly modified exhaustive key search algorithm for optimization.

The benefit of choosing this method is that we are not only optimizing the parameters, but also testing the stability of the graph generation process.

When the configurations are set, we choose a large number of random coordinate pairs in the graph. All coordinate pairs are selected from the real coordinate space, that is independently of the nodes of the graph that is evaluated. Between each coordinate in a coordinate pair a path will be searched with the A* pathfinding algorithm (Hart, Nilsson, & Raphael, 1968) with the Euclidean distance as the heuristic function. The pathfinding is then used to evaluate the configuration using objective function (1)

$$f(n, p, v, e) = -w_0 n - w_1 p - w_2 v - w_3 e, \tag{1}$$

where n is the average number of nodes opened, p average path length, v number of vertices and e number of edges and definitions for the subjective weights w_0, w_1, w_2 and w_3 are in Table 3.

Weight	Corresponding variable	Value
w_0	Average number of nodes opened	140000
w_1	Average path length	135000
w_2	Number of vertices	20000
w_3	Number of edges	3500

Table 3: Subjectively chosen illustrative weights for the objective function. In real use these weights should be selected with care.

2.3.1 Parameter Sensitivity

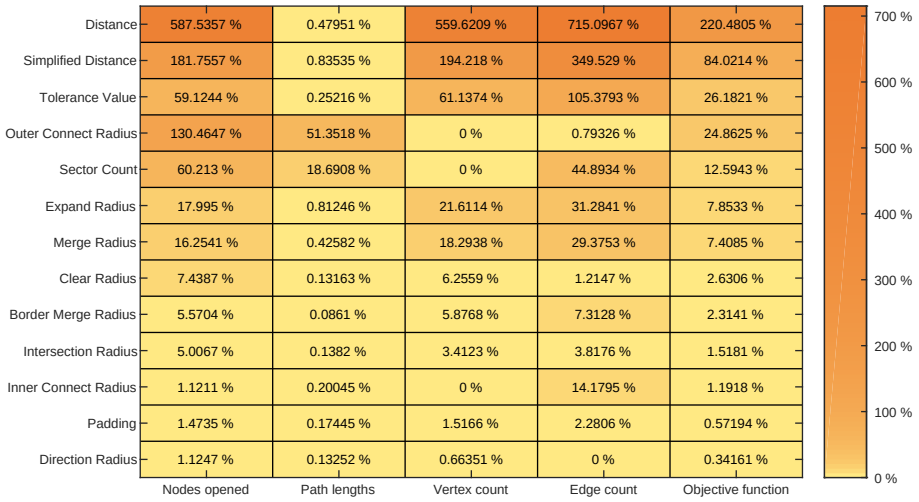


Figure 13: The sensitivity of parameters relative to the starting parameter configuration.

To avoid unnecessarily complex calculations, we first analyze the internal dependencies of the parameter values. For example, it is counterproductive to have a huge value for **Distance** and then have a low value for **OuterConnectRadius**, the dependencies defined by us are presented in Appendix E-1. To further simplify the calculations, we analyze how each parameter affects the graph generation process. The sensitivity analysis, in Figure 13, provides information about which parameters shall be focused on. The sensitivity for the objective function (1), is also evaluated by using the subjective parameters defined in Table 3. The starting parameter configuration and the variation ranges for each parameter are in Appendices E-2 and E-3.

2.3.2 Combinatorial Analysis

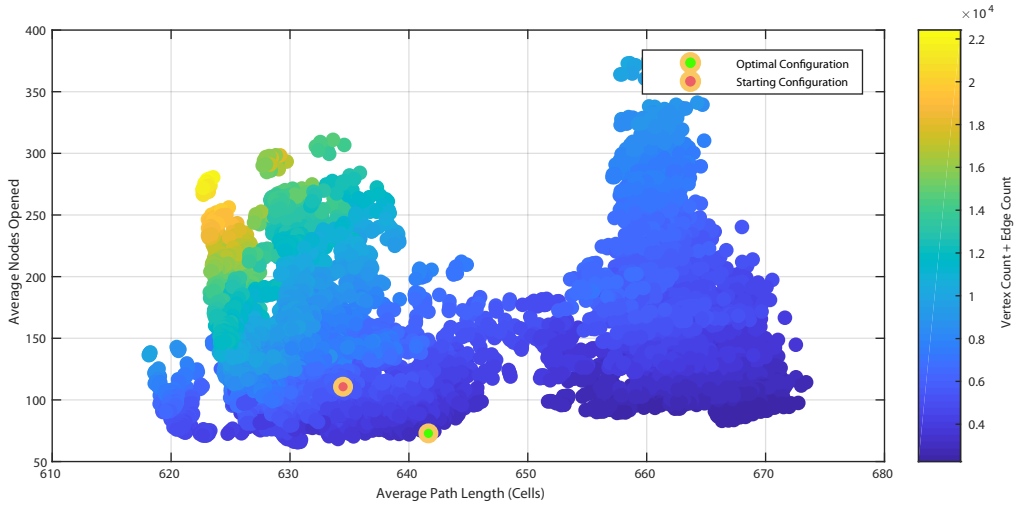
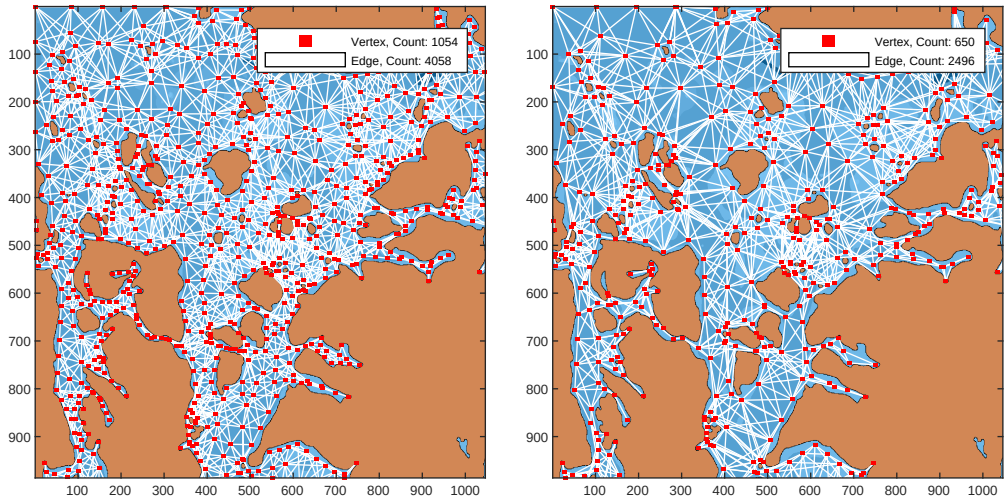


Figure 14: The combinatorially generated data.



(a) The starting parameter configuration. (b) The optimal configuration.

Figure 15: Comparison between graphs generated by the starting configuration and the configuration for which the objective function is maximised.

Figure 14 shows the data generated when running the graph generation procedure over the ranges and values specified in the Appendix E-4. The data has been reduced from four dimensions to three, for visualization purposes, by adding the number of vertices and edges together because these dimensions represent the same data to some extent. Figure 15 shows the difference between the graph generated by the starting parameter values and the computationally optimized graph.

3 Results

Ratio	A*	Dijkstra
Average reduction in number of nodes processed	1400x	1200x
Average increase in path length	1.001x	1.001x
Average reduction in processing time	22000x	3400000x
Reduction in vertex count	1080x	1080x
Reduction in edge count	1100x	1100x

Table 4: Comparison of the optimized graph to a graph created directly from the source data without any preprocessing. The graphs were compared with both A* and Dijkstra's pathfinding algorithm. Only one path was computed with Dijkstra's algorithm due to the excessive computation time.

Table 4 shows how performance regarding different aspects is affected when solving randomly generated pathfinding problems. Note that the ratios depend directly on the optimization criteria as well as the underlying data and cannot as such be regarded as exact values, but should rather be thought of as estimations. The data presented in Table 4 suggests that the generated graph is significantly better in all aspects, except average path length. The average path length is on average 0.1% longer, which is insignificant compared to the gains in other aspects. Due to the heuristic function in the A* algorithm, we included a comparison based on Dijkstra's algorithm (Dijkstra, 1959), which does make use any heuristic function. All edge weights in the graphs were assigned to be equal to the distance between the respective vertices. For a non-uniform weighting strategy, the A* algorithm performs less well while the performance of Dijkstra's algorithm remains unaffected.

4 Conclusions

The purpose of this thesis was to generate a navigation graph for coastal waters. First we acquired the necessary data, where after the data was processed to a more practical format. We then introduced a new graph generation method whose parameters were analyzed with regard to their sensitivity, finally, these parameters were optimized according to a objective function using our own subjective preferences and attained a favorable result. The generated graph was then compared to a graph generated directly from the unmodified data by evaluating them via various pathfinding problems. We demonstrated that the resulting graph performed significantly better on several criteria (e.g. the number of opened nodes, execution time, disk space) while offering almost identical performance when regarding the length of the found paths.

However, there is still room for future research. For one, the data acquired from the Finnish Transport Agency is not ideal in that it inadvertently forces us to discard traversable areas with a high probability. Secondly, the parameter sensitivity analysis is not exhaustive and because it was then used as a basis for the combinatorial analysis we can not be certain that the parameters have been optimized to a true global optimum. Another contributing factor to not finding the global optimum is that the complexity of the problem is so large that we could not possibly test all possible and sensible parameter configurations, which is why genetic algorithms could possibly be implemented.

The evaluation methods for the generated graphs may need to be revisited, because they may not truly reflect the usage purpose of the graph. Moreover, we chose some number of coordinate pairs randomly, which were then used to evaluate the graph. These coordinate pairs affect the optimization procedure. To the extent which they accurately represent the underlying data could be included in further research.

All techniques were developed for raster datasets and as such another interesting research avenue would be to translate them to vector logic. Consequently, some procedures could probably be executed faster and some slower, but the total net effect is unknown. However, the result would most likely not display any serious disparities.

Nonetheless the data in Figure 14 strongly affirms, due to its almost continuous form, that our navigation graph generation process is functional. Visual evaluation suggests the same thing. Thus, we are able to conclude that the process as a whole serves its intended purposes.

5 Appendix

A Reference Point Generation Algorithm

```
Function ReferencePoints(Set, Distance, Padding, RoundMode)
{
  //Set is a one-dimensional continous traversable set
  //Distance is a parameter that defines the distance between reference points
  //Padding is a parameter that defines how each set is padded
  //RoundMode is a function that takes as an input a real value and outputs an integer

  //Calculate the length of Set and take Padding into account
  //The length of Set must be greater than Padding * 2
  length = max(Set) - min(Set) + 1 - Padding * 2


  //Calculate the number of reference points
  n = RoundMode(length / Distance)

  //Create an array in which the reference points will be stored
  referencePoints = empty array of size n

  //Populate referencePoints
  For(i = 0, i < n, i = i + 1)
  {
    //Calculate the position of one reference point
    referencePoints[i] = min(Set) + Padding + length * (i + 1) / (n + 1)
  }

  //return the result
  return referencePoints
}
```

B Segmented Border Tracing Algorithm

```
Function SegmentedBorderTracing(AllUntraversableCells)
{
  //AllUntraversableCells is a set containing all untraversable cells

  //Create the set OpenSet that contains all border cells
  //This set will keep track of which cells must be processed
  //GetBorderCells is a function that returns all cells that have atleast one adjacent traversable cell
  OpenSet = GetBorderCells(AllUntraversableCells)

  //Add all cells from OpenSet with more than two neighbors in OpenSet to the new set BadBorderCells
  //GetBadBorderCells is a function that returns said cells
  BadBorderCells = GetBadBorderCells(OpenSet)

  //Remove the all cells in BadBorderCells from OpenSet
  OpenSet = OpenSet - BadBorderCells

  //Create the set Result that will store the resulting segments
  //A segment is an ordered list of points
  Result = empty set

  //Process all cells in OpenSet
  While OpenSet is not empty
  {
    //All cells in OpenSet have a maximum of two neighbors
    //This mean that one can propagate in a maximum of two directions from each cell

    //Create empty ordered lists Forward and Backward
    //The directions in lists Forward and Backward are arbitrary
    Forward = empty list
    Backward = empty list

    //Choose any cell A from OpenSet and add it to Forward
    A = Any(OpenSet)
    Add A to Forward

    //Propagate "forward" as far as possible
    If A has any connected cells in OpenSet
    {
```

```
      Next = Any connected cell to A that is in OpenSet
      Remove Next from OpenSet
      Add Next to Forward

      While Next has any connected cells in OpenSet
      {
        Next = Any connected cell to Next that is in OpenSet
        Remove Next from OpenSet
        Add Next to Forward
      }
    }

    //Propagate backwards as far as possible
    If A has any connected cells in OpenSet
    {
      Next = Any connected cell to A that is in OpenSet
      Remove Next from OpenSet
      Add Next to Backward

      While Next has any connected cells in OpenSet
      {
        Next = Any connected cell to Next that is in OpenSet
        Remove Next from OpenSet
        Add Next to Backward
      }
    }

    //Unify both Backward and Forward to one list
    //First reverse Backward and then concenate Forward to it
    //By doing this we get a continous segment that contains all cells from both lists in order
    //Then add this segment to Result

    Segment = Reverse(Backward) concenate Forward
    Add Segment to Result
  }

  //If we want all border cells to be accounted for then all cells in BadBorderCells must be added back
  //However, this is not necessary for our purposes

  return Result
}
```

C Border Expansion Algorithm

```
Function ExpandedBorder(BorderCells, R)
{
  //BorderCells is the set containing all border cells
  //R is a constant that determines how much a cell can maximally be expanded

  //Create the empty set Result that will contain all expanded border cells
  Result = empty set

  For each cell C in BorderCells
  {
    //Create a new vector V
    V = (0, 0)

    //Add the vector from C to U to V
    For each untraversable cell U within a distance of N from C
      V = V + (U - C)

    //Rotate V 180 degrees and normalize it
    V = V * -1
    V = Normalize(V)

    //Calculate both normals to V
    N1 = NormalVector(V, 1)
    N2 = NormalVector(V, 2)

    //Define the polygon P with 4 vertices
    P = (C + N1 * R / 2, C + N2 * R / 2, C + V * R + N2 * R / 2, C + V * R + N1 * R / 2)

    //Choose the cell within P that has the largest distance to any untraversable cell
    //D is any distance function
    Best = Max(D(P))

    Add Best to Result
```

```
  }
  //Remove duplicates from Result
  //Unique is a function that removes duplicates
  Result = Unique(Result)

  return Result
}
```

D Ramer Douglas Peucker Modification

```
Function ExpandedBorder(BorderCells, R)
{
  //BorderCells is the set containing all border cells
  //R is a constant that determines how much a cell can maximally be expanded

  //Create the empty set Result that will contain all expanded border cells
  Result = empty set

  For each cell C in BorderCells
  {
    //Create a new vector V
    V = (0, 0)

    //Add the vector from C to U to V
    For each untraversable cell U within a distance of N from C
      V = V + (U - C)

    //Rotate V 180 degrees and normalize it
    V = V * -1
    V = Normalize(V)

    //Calculate both normals to V
    N1 = NormalVector(V, 1)
    N2 = NormalVector(V, 2)

    //Define the polygon P with 4 vertices
    P = (C + N1 * R / 2, C + N2 * R / 2, C + V * R + N2 * R / 2, C + V * R + N1 * R / 2)

    //Choose the cell within P that has the largest distance to any untraversable cell
    //D is any distance function
    Best = Max(D(P))

    Add Best to Result
  }
  //Remove duplicates from Result
  //Unique is a function that removes duplicates
  Result = Unique(Result)

  return Result
}
```

E Parameter Configurations

E-1 Parameter Dependencies

Parameter	Dependency
Distance (D)	Independent
Padding	Independent
Intersection Radius	Independent
Expand Radius	Independent
Direction Radius	Independent

Tolerance Value	Independent
Simplified Distance	Independent
Clear Radius	D / 5
Merge Radius	D / 5
Outer Connect Radius	D ∗ 1,5
Inner Connect Radius	D / 8
Segment Count	Independent
Border Merge Radius	Independent

D = Abbreviation for Distance. Used to denote parameter dependencies.

E-2 Initial Parameter Configuration

Parameter	Value
Distance	100
Padding	10
Intersection Radius	1
Expand Radius	10
Direction Radius	1
Tolerance Value	12
Simplified Distance	50
Clear Radius	20
Merge Radius	20
Outer Connect Radius	150
Inner Connect Radius	12,5
Segment Count	8
Border Merge Radius	4

E-3 Sensitivity Ranges

Parameter	VMin	VMax	S
Distance (D)	20	180	20
Padding	2	18	20
Intersection Radius	0	2	3
Expand Radius	2	18	20
Direction Radius	0	2	3
Tolerance Value	2,4	21,6	20
Simplified Distance	10	90	20

Clear Radius	D / 5 ∗ a	D / 5 ∗ b	20
Merge Radius	D / 5 ∗ a	D / 5 ∗ b	20
Outer Connect Radius	D ∗ 1.5 ∗ a	D ∗ 1.5 ∗ b	20
Inner Connect Radius	D / 8 ∗ a	D / 8 ∗ b	20
Segment Count	2	14	13
Border Merge Radius	0,8	7,2	20

Parameters are tested SEPARATELY.

Complexity = Sum(S)
= 20 ∗ 10 + 13 + 3 ∗ 2
= 219

D = Abbreviation for Distance. Used to denote parameter dependencies.
VMin = Minimum value for parameter variation range.
VMax = Maximum value for parameter variation range.
S = Number of steps in variation range.

a = 0,2
b = 1,8

E-4 Combinatorial Ranges

Parameter	VMin	VMax	S
Distance (D)	20	180	20
Padding	10	10	1
Intersection Radius	1	1	1
Expand Radius	2	18	3
Direction Radius	1	1	1
Tolerance Value	2,4	21,6	5
Simplified Distance	10	90	10
Clear Radius	D / 5	D / 5	1
Merge Radius	D / 5 ∗ a	D / 5 ∗ b	4
Outer Connect Radius	D ∗ 1,5 ∗ a	D ∗ 1,5 ∗ b	5

```
Inner Connect Radius    D / 8          D / 8          1
Segment Count           2              14             4
Border Merge Radius     0,8            7,2            2

Parameters are tested COMBINATORIALLY.

Complexity = Product(S)
           = 20 * 10 * 5^2 * 4^2 * 3 * 2
           = 480000

D    = Abbreviation for Distance. Used to denote
       parameter dependencies.
VMin = Minimum value for parameter variation range.
VMax = Maximum value for parameter variation range.
S    = Number of steps in variation range.

a = 0,2
b = 1,8
```

E-5 Optimal Configuration

```
Parameter              Value
Distance               100
Padding                10
Intersection Radius    1
Expand Radius          10
Direction Radius       1
Tolerance Value        12
Simplified Distance    50
Clear Radius           20
Merge Radius           20
Outer Connect Radius   150
Inner Connect Radius   12,5
Segment Count          8
Border Merge Radius    4
```

References

J. E. Bresenham (1965). "Algorithm for computer control of a digital plotter". *IBM Systems Journal* 4.1, pp. 25–30.

J.-L. De Carufel et al. (2014). "A note on the unsolvability of the weighted region shortest path problem". *Computational Geometry* 47.7, pp. 724–727.

E. W. Dijkstra (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik* 1.1, pp. 269–271.

D. H. Douglas & T. K. Peucker (1973). "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2, pp. 112–122.

A. Ghuneim (2000). "Contour tracing algorithms". URL: http://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/alg.html, accessed 25.08.2018.

Gisgeography (2018). "Vector vs Raster: What's the Difference Between GIS Spatial Data Types?" URL: https://gisgeography.com/spatial-data-types-vector-raster/, accessed 27.08.2018.

P. E. Hart, N. J. Nilsson, & B. Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths". *IEEE transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

T. Leonard (2014). "Procedural Generation of Navigation Meshes In Arbitrary 2D Environments". *Game Behaviour* 1.1.

Library of Congress (2017). "ESRI ArcInfo ASCII Grid". URL: https://www.loc.gov/preservation/digital/formats/fdd/fdd000421.shtml, accessed 26.08.2018.

Library of Congress (2017). "ESRI Shapefile". URL: https://www.loc.gov/preservation/digital/formats/fdd/fdd000280.shtml, accessed 25.08.2018.

Liikennevirasto (2018). "Finnish Transport Agency - Download Service". URL: "https://julkinen.liikennevirasto.fi/oskari/?lang=en, accessed 15.07.2018.

J. R. Shewchuk (2002). "Delaunay refinement algorithms for triangular mesh generation". *Computational Geometry* 22.1-3, pp. 21–74.

G. Snook (2000). "Simplified 3D movement and pathfinding using navigation meshes". *Game Programming Gems* 1.1, pp. 288–304.

L. Vandevenne (2018). "Flood fill". URL: https://lodev.org/cgtutor/floodfill.html, accessed 25.08.2018.

L. Yang & M. Worboys (2015). "Generation of navigation graphs for indoor space". *International Journal of Geographical Information Science* 29.10, pp. 1737–1756.