Mika Murtojärvi

Efficient Algorithms for Coastal Geographic Problems

Turku Centre for Computer Science

TUCS Dissertations
No 210, April 2016

Efficient Algorithms for Coastal Geographic Problems

Mika Murtojärvi

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Tauno Nurmela Hall (Lecture Hall I) on April 29, 2016, at 12 noon.

University of Turku
Department of Information Technology
Vesilinnantie 5, 20500 Turku

2016

Supervisors

Olli Nevalainen
Department of Information Technology
University of Turku
Turku
Finland

Ville Leppänen
Department of Information Technology
University of Turku
Turku
Finland

Reviewers

Martti Juhola
School of Information Sciences
FI-33014 University of Tampere
Finland

Borut Žalik
Faculty of Electrical Engineering and Computer Science
University of Maribor
Slomškov trg 15, 2000 Maribor
Slovenia

Opponent

Jan Westerholm
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5
Finland

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

ISBN 978-952-12-3372-2
ISSN 1239-1883

Tiivistelmä

Tietokoneiden suorituskyvyn kasvaminen on tehnyt mahdolliseksi ratkaista algoritmisesti ongelmia, joita on aiemmin tarkasteltu paljon ihmistyötä vaativilla, mahdollisesti epätarkoilla, menetelmillä. Algoritmien suorituskykyyn on kuitenkin toisinaan edelleen kiinnitettävä huomiota lähtömateriaalin suuren määrän tai ongelman laskennallisen vaikeuden takia.

Väitöskirjaan sisältyvissä artikkeleissa tarkastellaan kahta maantieteellistä ongelmaa. Ensimmäisessä näistä on määritettävä etäisyyksiä merellä olevista pisteistä lähimpään rantaviivaan ennalta määrätyissä suunnissa. Etäisyyksiä ja tuulen voimakkuutta koskevien tietojen avulla on mahdollista arvioida esimerkiksi aallokon voimakkuutta. Toisessa ongelmista annettuna on joukko tarkkailuasemia ja niiltä aiemmin kerättyä tietoa erilaisista vedenlaatua kuvaavista parametreista kuten sameudesta ja ravinteiden määristä. Tehtävänä on valita asemajoukosta sellainen osajoukko, että vedenlaatua voidaan edelleen tarkkailla riittävällä tarkkuudella, kun mittausten tekeminen muilla havaintopaikoilla lopetetaan kustannusten säästämiseksi.

Väitöskirja keskittyy pääosin ensimmäisen ongelman, suunnattujen etäisyyksien, ratkaisemiseen. Haasteena on se, että tarkasteltava kaksiulotteinen kartta kuvaa rantaviivan tyypillisesti miljoonista kärkipisteistä koostuvana joukkona polygoneja ja etäisyyksiä on laskettava miljoonille tarkastelupisteille kymmenissä eri suunnissa. Ongelmalle kehitetään tehokkaita ratkaisutapoja, joista yksi on likimääräinen, muut pyöristysvirheitä lukuun ottamatta tarkkoja. Ratkaisut eroavat toisistaan myös siinä, että kolme menetelmistä on suunniteltu ajettavaksi sarjamuotoisesti tai pienellä määrällä suoritinytimiä, kun taas yksi menetelmistä ja siihen tehdyt parannukset soveltuvat myös voimakkaasti rinnakkaisille laitteille kuten GPU:lle.

Vedenlaatuongelmassa annetulla asemajoukolla on suuri määrä mahdollisia osajoukkoja. Lisäksi tehtävässä käytetään aikaa vaativia operaatioita kuten lineaarista regressiota, mikä entisestään rajoittaa sitä, kuinka monta osajoukkoa voidaan tutkia. Ratkaisussa käytetäänkin heuristiikkoja, jotka eivät välttämättä tuota optimaalista lopputulosta.

Avainsanat: Suunnattu etäisyys, aaltovaikutus, vedenlaatu, GPU-algoritmit

Abstract

The increasing performance of computers has made it possible to solve algorithmically problems for which manual and possibly inaccurate methods have been previously used. Nevertheless, one must still pay attention to the performance of an algorithm if huge datasets are used or if the problem is computationally difficult.

Two geographic problems are studied in the articles included in this thesis. In the first problem the goal is to determine distances from points, called study points, to shorelines in predefined directions. Together with other information, mainly related to wind, these distances can be used to estimate wave exposure at different areas. In the second problem the input consists of a set of sites where water quality observations have been made and of the results of the measurements at the different sites. The goal is to select a subset of the observational sites in such a manner that water quality is still measured in a sufficient accuracy when monitoring at the other sites is stopped to reduce economic cost.

Most of the thesis concentrates on the first problem, known as the fetch length problem. The main challenge is that the two-dimensional map is represented as a set of polygons with millions of vertices in total and the distances may also be computed for millions of study points in several directions. Efficient algorithms are developed for the problem, one of them approximate and the others exact except for rounding errors. The solutions also differ in that three of them are targeted for serial operation or for a small number of CPU cores whereas one, together with its further developments, is suitable also for parallel machines such as GPUs.

Keywords: Fetch length, wave exposure, water quality, GPU algorithms

Acknowledgements

This thesis was written under the supervision of Professors Olli S. Nevalainen and Ville Leppänen. Their help was invaluable at every stage of the work. In particular, I often found it difficult to write articles and the thesis in a way that is easy to read and still covers the subject in a sufficient detail. At times, I also needed some persuasion when continuing with the thesis felt rather pointless. Although I am still unsure whether the university degree will be of any advantage to me, at least writing the thesis was not as bad an experience as I expected. Sometimes it was even a welcome addition to my daily routine.

I would also like to thank Mikko Laakso who helped me in many ways not directly related to the thesis work. Without him I would likely not have chosen to finish this work. Outside the department of Information Technology, researchers of geography, especially Tapio Suominen and Harri Tolvanen, provided the interesting problems studied in this thesis and were also involved in writing some of the articles.

When starting work on this thesis, I worked at the department of Information Technology of University of Turku, with enough work time reserved for the thesis. In addition to the thesis work I had the opportunity to take part in teaching several courses. This was very important for providing variety to my work, and I thank the department of Information Technology for having had this opportunity.

Finally, I wish to thank my closest relatives, especially my parents and sister, for being there to support me when that was necessary.

Turku, March 1, 2016.
Mika Murtojärvi

List of original publications

P1 Murtojärvi M., Suominen T., Tolvanen H., Leppänen V. and Nevalainen O.S., Quantifying Distances from Points to Polygons - Applications in Determining Fetch in Coastal Environments. *Computers & Geosciences*, 33(7), 2007.

P2 Murtojärvi M., Leppänen V. and Nevalainen O.S., Determining directional distances between points and shorelines using sweep line technique. *International Journal of Geographical Information Science*, 23(3), 2009.

P3 Murtojärvi M., Leppänen V. and Nevalainen O.S., A parallel GPU implementation of an algorithm for determining directional distances. *Proceedings of 12th International Conference on Computer Systems and Technologies, CompSysTech'11*, 2011.

P4 Murtojärvi M., Leppänen V. and Nevalainen O.S., Performance tuning and sparse traversal technique for a cell-based fetch length algorithm on a GPU. Concurrency and Computation, Practice and Experience, accepted for publication, 2015.

P5 Murtojärvi M., Suominen T., Uusipaikka E., Nevalainen O.S., Optimising an observational water monitoring network for Archipelago Sea, South West Finland. *Computers & Geosciences*, 37(7), 2011.

Contents

I Synopsis 1

1 Introduction 3
1.1 The fetch length problem . . . 4
1.1.1 Fetch length in coastal research . . . 6
1.1.2 Methods for computing fetch lengths . . . 7
1.1.3 Related problems . . . 8
1.2 The water monitoring network problem . . . 10
1.2.1 Problem setting . . . 11
1.2.2 Related research . . . 11
1.3 Goals and results of this research . . . 12

2 The fetch length problem 15
2.1 Problem setting . . . 15
2.2 A brute force algorithm . . . 17
2.3 Data structures . . . 18
2.3.1 Interval trees . . . 18
2.3.2 Order-statistic trees . . . 19
2.4 Sweep line technique . . . 20
2.4.1 Time complexity . . . 22
2.4.2 A raster-based sweep line algorithm . . . 24
2.5 Cell-based algorithm . . . 28
2.5.1 Time complexity . . . 30
2.6 Tailoring the cell-based algorithm for GPU . . . 32
2.6.1 OpenCL programming model . . . 32
2.6.2 The architecture of a GPU . . . 34
2.6.3 Cell-based fetch length algorithm on a GPU . . . 37

3 The water quality monitoring network problem 39
3.1 Statistical model . . . 40
3.2 Missing and incorrect values . . . 41
3.3 Network optimization . . . 42

4 Summary of publications **45**
4.1 P1: Quantifying Distances from Points to Polygons - Applications in Determining Fetch in Coastal Environments 45
4.2 P2: Determining directional distances between points and shorelines using sweep line technique 46
4.3 P3: A parallel GPU implementation of an algorithm for determining directional distances 49
4.4 P4: Performance tuning and sparse traversal technique for a cell-based fetch length algorithm on a GPU 51
4.5 P5: Optimising an observational water monitoring network for Archipelago Sea, South West Finland 54

5 Conclusions **57**

II Publication reprints

Part I

Synopsis

Chapter 1

Introduction

The performance and memory capacity of computers have increased rapidly in the recent decades. This has enabled tackling larger problems than before and improving accuracy in cases where one previously had to resort to manual methods. Geographic information systems (GIS) can now be found for problems dealing with map data, as both commercial and free of charge software.

On the other hand, making measurements has also become much easier with an increasing degree of automation. As a simple example, it is now possible to record the air temperatures at a particular location at regular intervals with very little manual labor. Techniques such as aerial photography and satellite measurements can provide detailed information about large areas. The result is that the datasets used in geographic research can be very large. The efficiency of the algorithms used for processing that data then becomes important despite the advances in computer technology.

In some cases the result of interest may be time-consuming to compute even if the dataset is quite small. As an example one might consider the problem of selecting a subset of a set S of n items that is optimal according to some criterion. If the size of the desired subset is not restricted, there are 2^n subsets of S. Unless there is a way to avoid checking all possible subsets, the running time of an algorithm for finding the optimal subset would be $\Omega(2^n)$. As the exponential function grows very rapidly, only relatively small problems could be solved using any computer. Unfortunately, large problem sizes do occur in real-world problems. In such a case one possibility is to resort to approximation algorithms [1] to find some, possibly suboptimal, solution to the problem.

Two algorithmic problems are considered in this thesis. In the *fetch length problem* the goal is to find distances from given points to the nearest land areas in a specified direction. The challenge in this problem is the size of the input: There can be hundreds of millions points for which the distances

are to be computed, several directions of interest and the map itself may be large. The other considered problem is that of *optimizing a water monitoring network*. In this problem there are a number of sites at which water quality samples have been collected. To reduce costs their number should be reduced, while losing as little water quality information as possible. This rather imprecise problem setting allows several interpretations. Among the most important is whether one is allowed to move the observational sites or whether the new network should be formed by only removing some sites from the existing network. In this work the latter choice will be taken and the task is then to find an optimal subset of the existing set of observational sites. The number of sites in the network is moderate (about 60) but approximate algorithms are still needed to achieve an acceptable running time. The problem is also complicated by errors and missing values in the observational dataset and by the need to choose a model that allows assessing which sites to remove from the network. While these choices are mentioned, they are outside the scope of this thesis and previously known methods and software are used for dealing with the problems.

1.1 The fetch length problem

In the fetch length problem the input consists of a map and a set of points, called *study points* to distinguish them from the points occurring in the map data. Additionally, one is given the directions in which the fetch lengths are computed. The directions are usually the same for all study points. The goal is to compute the distance from each point to the nearest land area in each direction, see Figure 1.1.

The map is two-dimensional, i.e., no water depth or land height information is available. The map is specified as a set of islands, each island being represented by a list of points that lie on its coastline. An island is then approximated by a polygon whose vertices are the points of the input data, with consecutive points joined by line segments. In the algorithms for solving the problem the information about which line segment belongs to which island is not needed and the problem is slightly simplified.

For notation, the symbol S will be used for the set of study points and a single study point is represented by the symbol p. The set of all line segments of the polygon boundaries is denoted by L and a single segment by l. The set of directions is Θ and a single direction is marked by θ. Using these notations, the goal is then to compute for each study point $p \in S$ and direction $\theta \in \Theta$ the fetch length $d(p, \theta)$, where the symbol d stands for distance; $d(p, \theta)$ is the minimal distance from p to the interior of a polygon in the direction θ. For a point that is in a water area, we clearly have $d(p, \theta) = \min_{l \in L} d(p, l, \theta)$, where $d(p, l, \theta)$ is the distance from the study point to the line segment l in

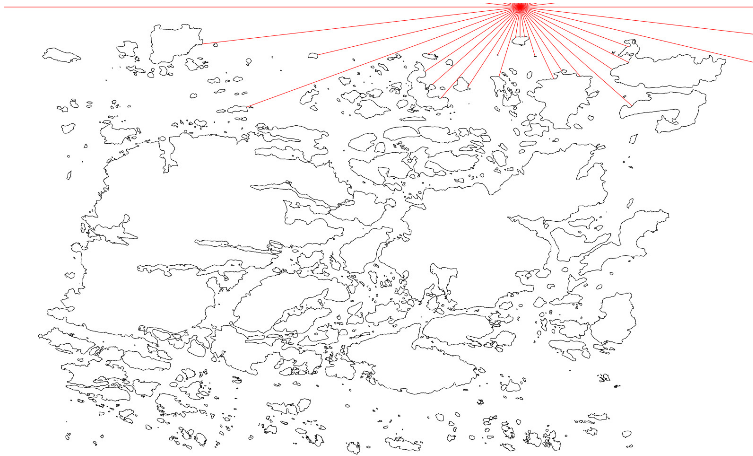

Figure 1.1: Fetch lengths in 48 directions for one study point that is on water.

the direction θ. For this to hold it is defined that the distance $d(p, l, \theta)$ is infinite if the half line $h(p, \theta)$ starting from p and having the direction θ does not intersect l. For a point that is in a land area, i.e., inside a polygon, the fetch length is by definition 0.

A special case occurs when a study point lies on the coastline of an island, i.e., on a polygon border. While the above definition technically covers the case, this is an important practical case. The convention for the border points is that the fetch length $d(p, \theta) = 0$ if the half line $h(p, \theta)$ points into the island polygon. Otherwise, the fetch length is defined similarly to points on water, except that the segment on which p lies is excluded from the set of line segments to get a non-zero fetch length. The points lying in water are called *exterior points* and the points located on land *interior points*. The remaining points are *border points*.

Special cases have still been ignored above. For instance, it was assumed that a border point lies on a segment, but it can also coincide with a vertex - a point shared by two consecutive line segments of a polygon. Also, the half line $h(p, \theta)$ may hit a vertex instead of a segment so that the half line does not enter the polygon. The former case is easy to handle by excluding all segments on which p lies. The latter case is, on the other hand, not very important in practice as any imprecision of the map could lead to a different conclusion about whether the half line intersects the island polygon or not.

While the cases mentioned above pose no problem for defining fetch length sufficiently, it is necessary to identify all relevant special cases in the algorithms for solving the problem. A failure to do so can lead to wrong conclusion about whether a study point lies on land or on water, causing

zero lengths for exterior points and non-zero lengths for interior points. The somewhat tedious special cases vary from one algorithm to another and will be considered only when discussing the publications included in this thesis.

1.1.1 Fetch length in coastal research

Having defined what fetch length is, let us briefly consider the motivation for computing such quantities: Fetch lengths are used in a method for estimating wave exposure at a particular location [2]. It is assumed that wind blows in the same direction for a long period of time, leading to a state where the average wave height and direction stabilize. It is also assumed that waves always move in the same direction as the wind. The latter assumption is not accurate because obstacles, i.e., shorelines and shallow areas, lead to phenomena such as diffraction and refraction. These possibilities are ignored, so a wave is taken to vanish when it hits a shoreline. Similarly, after an obstacle a wave starts growing from zero height no matter how small the obstacle is.

While this essentially one-dimensional model of wave formation is very simple, fetch length has still been found to be a useful quantity for estimating wave exposure [2]. It should now be clear why fetch lengths are computed in several different directions: A fetch line direction corresponds to the direction of the wind and wind may blow in any direction. One should then compute fetch lengths in many directions to have a decent sampling of the possible directions of the wind. On the other hand, the number of directions may need to be kept moderate to reduce the time required for computing fetch lengths.

Computing fetch lengths in a predetermined set of directions has the unfortunate effect that features of the map may be essentially ignored for some study points. For instance, it may happen that none of the given fetch lines originating from a point hit an island that is reachable from the point. Conversely, a fetch line might hit a very small island that would not have much effect on the estimated wave exposure if fetch could be computed in all directions. It would be possible to refine the concept of fetch length to take into account a sector of directions instead of a single direction, but doing so would need to take into account the fact that wave exposure is a nonlinear function of fetch length [2]. Computing the lengths for a sector would be more time-consuming than for a single direction, and any model based on fetch lengths is still only a rough approximation of wave exposure. For these reasons, the concept of fetch length is accepted as is in this work.

1.1.2 Methods for computing fetch lengths

A GIS-based algorithm for the fetch length problem was given by Ekebom *et al.* [2]. The algorithm can compute fetch lengths for one or multiple study points in different directions, using vector-based input data similar to described in Section 1.1. The procedure starts by adding, for each study point, the candidate fetch lines. A candidate fetch line for point p and direction θ is a line segment having p as an end point and forming the angle θ with the x-axis. The candidate fetch lines must be longer than the maximal distance within the map area to ensure that all intersections can be found. The candidate fetch lines are then cut using the map data. The map represents each island polygon as a hole, and the result of cutting a candidate fetch line is then a set of line segments that are subsegments of the candidate fetch line, each segment spanning the water area between consecutive islands. The exceptions are the first and the last segment that may end in the study point or the original other end point of the candidate fetch line, respectively. The segments that do not end in the study point are irrelevant and are removed. Now, only the fetch lines are left and the fetch lengths are determined by computing the lengths of the fetch lines.

The time complexity of the GIS-based method has not been analyzed and doing so would require knowing how the GIS software and the extensions handle the operation of cutting a set of line segments with the map data. Direct performance measurements were also not available, but a bound can be inferred from one of the publications included in this thesis (P1). There it is noted that a new algorithm allowed increasing the number of study points by a factor of 1600 while still requiring a fraction of the time compared to the GIS-based method. The computations using the new algorithm took about five hours for a study area of 40 km $\times$ 40 km containing 2.56 million study points. This implies that the processing time using the GIS-based method would have been years. Indeed, the GIS-based method has been used for moderate numbers of study points. An example is a case where there were about 7800 exterior and 2600 border study points, with interior study points pruned from the input without running the fetch length algorithm for them [3].

A raster-based method for computing fetch lengths is also known [4][1]. Here, the input data is also given in a raster format, i.e., as a grid of cells, each cell containing its elevation from the sea level. To compute fetch lengths for all grid cells in a vertical direction the algorithm starts from the top of the map. A one-dimensional array $fetch$ whose length is equal to the width of the map is initialized with negative values to indicate unbounded fetch lengths. Then, the map is scanned downwards one row at a time. A row is

[1] The source code can be found at `http://www.umesc.usgs.gov/management/dss/wind_fetch_wave_models.html`.

processed by checking for each of its columns whether the cell is in a water area. If that is the case for the cell $currentRow[i]$, $fetch[i]$ is incremented by the size of the cell[2], because the distance from the current cell to the nearest land area in the upwards direction is one cell more than it was from the corresponding cell of the previous row. If $currentRow[i]$ is on land, fetch length $fetch[i]$ is set to zero. Once the row has been processed, the current array of fetch lengths is written to the output file. To handle other wind directions the raster map can be rotated before applying the algorithm described above.

To use the raster-based method with a vector map, the map first needs to be converted into a raster format. This causes a loss of precision whose magnitude depends on the chosen cell size. The raster-based algorithm also has its own inaccuracies that result from, e.g., rotating the map and requiring a post-processing step to fill in any data gaps caused by the rotations. The comments in the source code indicate that computed fetch lengths have an accuracy of about 6 cell lengths while coordinates may be inaccurate by 3 cells. On the other hand, a small cell size increases the processing time and memory requirements of the algorithm.

In 2010, a cell-based algorithm for determining fetch lengths was given by Yang *et al.* [5]. The method subdivides the map by a regular grid into non-overlapping rectangular areas (cells). A major difference to purely raster-based method is that, for each cell, the entire set of original line segments contained in the cell is stored, making it possible to maintain full precision in the fetch length computation. Examining a cell is more time-consuming than in raster-based methods since all of its line segments are examined for intersection with the candidate fetch line. However, the number of line segments in a cell is typically small. Furthermore, the cells can be larger than in a raster-based method since the cell size has no effect on the accuracy of the algorithm. The algorithm can also determine a fetch length in any direction with the same cell-based representation of the map, making it very efficient in the case where the number of study points is moderate compared to the number of polygon vertices in the map data. Indeed, the algorithm was found to be faster in practice than those algorithms presented in this thesis that were developed before the cell-based method [5]. In this work the cell-based algorithm is adapted for operation on a GPU and new optimizations are developed to further improve its performance on a GPU.

1.1.3 Related problems

The wave model based on fetch lengths is rather simplistic. In essence, the model is one-dimensional and does not therefore take into account the

[2] The exception is if the fetch length is still unbounded. Then the fetch length is decremented by the cell size to still indicate unbounded fetch.

effect of water depth and diffraction and reflection caused by waves hitting obstacles. More sophisticated wave models are now readily available [6, 7, 8, 9, 10]. Software implementations of such models were also found. However, it was the decision of domain experts not to use those implementations as they were considered unsuitable for the complex archipelago environment. Also, the computations had to be carried out using only a two-dimensional map as full depth information was not available for the area of interest.

A raster-based method for determining *maximum fetch* has also been proposed [11]. Here, for a given cell and a sector of directions the maximal fetch length within that sector is determined. The algorithm uses a visibility principle for computing the maximum fetch. First, each cell in a land area is given the height 100 and each water cell the height 0. The cell of interest (target cell) is elevated slightly to height 5. Then, each cell is checked for visibility from the target cell. After finding all visible cells, their distances and directions from the target cell are determined. The cell with the maximal distance within the sector of interest is finally reported. When implemented on GIS software, the algorithm could determine fetch lengths for 55000 cells in eight compass directions in 13 days, using a 1 GHz processor [11]. Using the method for millions of study points in a complicated map would be rather time-consuming, and it is unclear whether maximum fetch is a more useful quantity than fetch length.

The fetch length problem is similar to other known problems of computational geometry. Among the most researched are intersection problems, particularly the problem of finding the intersections between line segments. This problem was solved using the sweep-line technique already in 1979 [13], and several other algorithms have been published later. There are two significant differences between the segment intersection problem and the fetch length problem. In the fetch length problem only one intersection point for a given study point p and direction θ is of interest: the closest (to p) of all points where the half line $h(p, \theta)$ intersects a segment $l \in L$. Solving the fetch length problem as an intersection problem would also produce several points where different candidate fetch lines, originating from different study points, intersect each other. They are of no interest in the fetch length problem and can be ignored. Computing those unnecessary intersections would have an unfavorable impact on the time required for computing fetch lengths using an intersection algorithm. Nevertheless, the similarities of the problems make it possible to modify known segment intersection or intersection detection algorithms [12, 13, 14] to the purpose of determining fetch lengths.

1.2 The water monitoring network problem

Eutrophication is a process where an excess of nutrients in water leads to increased growth of algae and plants. Among the most important nutrients are nitrogen and phosphorus whose concentration in water can increase, e.g., as a result of human activity. One area where the problem is significant is the Baltic Sea. Although the anthropogenic input of nutrients has declined recently, their concentrations remain elevated and nearly the entire sea area is affected by eutrophication [15].

To assess whether the state of a water area is changing in a favorable direction it is necessary to monitor the water quality with good coverage both spatially and temporally, i.e., water samples should be collected often and at several locations. On the other hand, a dense monitoring network is expensive to maintain, leading to an economic pressure to reduce the number of sites in the network. In this case the number of sites in the new network can be considered to be a predefined constant as it directly affects the cost of the monitoring program.

This is the situation in the case study of this thesis (P5). Previously, a fixed set of 60 water quality monitoring sites in the Archipelago Sea of Finland have been visited three times a year. The monitoring is currently largely manual, i.e. the sites are visited by boats instead of having automated monitoring systems at the sites. Despite this, the locations of the sites can be considered to be constant. The observations have also been made at almost the same time at all sites: Three trips have been made during summer, each trip taking roughly 4 days. Thus, the time between the trips is far greater than the time required for collecting the samples from all sites, i.e., the duration of a single trip.

One consequence of visiting the sites by boats is that it is not always possible to collect the samples due to, e.g., bad weather. This leads to the problem of missing data which cannot be ignored because the amount of observations for each site turns out to be rather small. Sometimes the recorded values are also incorrect because of typing errors or contaminated water samples. Many of the incorrect values are very different from the other observations and can be detected using simple methods. The problem of missing data is more difficult, but *imputation methods* [16, 17, 18] are now available as ready-made software [19, 20]. Imputation replaces the missing values by estimated values derived from the values that are available. In the case of multiple imputation several different estimates are generated to reflect the uncertainty in the estimates.

The quantities of interest in the water monitoring program are P_{tot} (unit µg/l), N_{tot} (µg/l), $chlo_a$ (µg/l) and *Secchi* (m). The first two are the amounts of phosphorus and nitrogen while the third, chlorophyll-a, is related to photosynthesis and is found in phytoplankton. Secchi depth is a

measure of visibility, measured in meters. These quantities are among the known indicators of the degree of eutrophication [15].

Visiting 60 sites three times each year by boats and analyzing the water samples requires a considerable amount of human labor. To make the monitoring less expensive it is of interest to find a smaller set of sites that can still give adequate information on water quality in the area. Certain requirements still need to be satisfied: The new network should have a good spatial coverage of the area and the dataset should be as continuous as possible. The latter requirement means that the monitoring sites should not be moved as that would mean ending the monitoring at one site and beginning at another one. The first requirement is taken into account simply by partitioning the area of interest and requiring that there remains at least one site in each area. In the case studied in this thesis (P5) the areas actually come as external inputs from other research.

1.2.1 Problem setting

With the above restrictions the problem is one of selecting a subset S' of the existing set of sites S. In addition to the sites a set of non-overlapping regions R is given. It is required that for each region $r \in R$ there is at least one site $s \in S'$ such that s is inside r (the coverage constraint). The sites are taken to be points in the two-dimensional map, meaning that a site can only reside in one region. The number of sites in S' is taken as external input (the size constraint).

With a given subset S' of sites one can associate a function $Error(S')$ that is a measure of how much the water quality information obtained using only sites in S' deviates from those obtained using all sites in S. The goal is then to find a subset S' that satisfies the size and coverage constraints, with $Error(S')$ being as small as possible. The error measure is left undefined at this point to keep the problem setting general. In the solutions to the problem a simple error measure will be used and the algorithms themselves are heuristic in nature. This means that the found subset S' may not be optimal with respect to the given error measure.

1.2.2 Related research

Previous research exists on the problem of optimizing observational networks. However, much of the work focuses on physical or statistical modeling which is outside the scope of this thesis. Hence, only aspects relevant to site selection are reviewed here.

Frolov *et al.* [21] consider the problem in a more general setting where one can both remove and add monitoring sites. They frame the problem as an optimization problem where there is a set of candidate locations for

sites. The goal is to find a set of sites that minimizes a cost function, the role of the cost function being similar to the error function described above. A high cost indicates that the network does not accurately reproduce the field of values of the observed quantity, the ground truth coming from a simulation. The optimization is done using a greedy method that is called an exchange-type algorithm. A basic step is the removal of one site and the addition of another one. The site to be removed is the one that decreases the cost function less than any other site in the existing network. Similarly, a new site is added to the candidate location whose observations are estimated to be most useful for reducing the value of the cost function. The process is repeated until no improvement is obtained. Different randomly generated initial networks can be used as the starting points of the optimization. Also add-only and delete-only versions of the algorithm were tested. The results indicate that the delete-only algorithm produces better results than the add-only method, while the goodness of the network produced by the exchange-algorithm depends on the initial configuration ranging from worse than add-only to better than delete-only [21]. The add-only and remove-only algorithms are also known as sequential forward and backward feature selection, respectively, in the literature on pattern recognition [22].

Lin *et al.* [23] propose an algorithm that is based on quadtrees. Here the goal was not to remove sites from an existing network but to freely determine good sampling locations. The study area is first divided into four quadrants of equal size. For each region a variance measure is computed from simulated data, and the area with the highest variance is further divided into four subregions, i.e., that quadrant is replaced by four new quadrants. A new site is placed in the center of each new quadrant, although a random placement of the sites within the quadrants was also tested. The process is then repeated, using the current set of quadrants as the starting point at each step, until a stopping criterion such as the maximum number of sites or a small enough variance is achieved. The method was seen to yield much better results than random sampling [23]. However, the procedure is unsuitable for the case where the purpose is to remove some sites from an existing network.

1.3 Goals and results of this research

The goal of this thesis is to develop solution methods for the fetch length problem and the water quality monitoring network problem. In particular, the research goals are:

- RG1: Develop efficient algorithms for determining fetch lengths.

- RG2: Develop a practical method for selecting a reduced number of monitoring sites.

For the fetch length problem three new sequential algorithms are designed and analyzed. Two of them are exact while one is approximate. The approximate algorithm is similar to the known raster-based algorithm [4], with the important difference that memory is conserved by rasterizing the map data only when needed. The rotation step is done for the original vector map instead of the raster representation, which improves accuracy. The exact methods have their roots in line segment intersection problems [14], modified to compute only the intersections of interest. These algorithms are a satisfactory solution to RG1 for serial implementations.

For parallel devices RG1 is examined further by adapting a previously known algorithm for the fetch length problem for GPU devices. The first version of the algorithm is a simple parallelization without taking the properties of GPUs into account except in some low-level details. This implementation is then improved by optimizing the memory access patterns and by introducing *sparse rasterization algorithm* for line segments. This technique allows a cell-based method to skip over many empty cells when looking for intersections, improving execution speed. Several unsuccessful attempts to improve the performance of the algorithm on a GPU are also documented and analyzed to understand why the methods failed.

For RG2 suboptimal algorithms for removing sites from the network are presented. In addition to the site selection algorithms the solution method also needs to incorporate other steps to deal with the limitations of the input data. For the problem of incorrect data (outliers) simple methods are used while pre-existing software is used for the imputation of missing data. In the case considered in this thesis, it turned out that the work needs to be carried out using only a small number of observations, and the optimization results are then somewhat sensitive to the missing values. The running times of even the simplest algorithms are rather high when implemented using statistical software. Thus, no attempt is made to improve the simple algorithms. Nevertheless, the results are found to be better than a random selection of sites, even when the imputed values in the testing dataset differ from those used for selecting the new network.

Chapter 2

The fetch length problem

In the fetch length problem there may be millions of line segments in the map data, millions of study points and typically a few dozen directions in which the lengths are to be computed. A brute-force algorithm for the problem would, given a point p and direction θ, iterate over all line segments $l \in L$. For each segment the distance from the point p to the segment l in the direction θ, i.e. $d(p, \theta, l)$ would be computed, with the smallest found distance being the required fetch length. However, if the study point lies in the interior of a polygon, fetch length is taken to be zero.

To achieve acceptable computation time the number of intersection computations must be reduced from that required by the brute-force algorithm. In this work four different techniques for solving the fetch length problem are explored, three of them being in principle exact while one computes an approximate result whose accuracy depends on the chosen rasterization resolution. The exact methods use three different algorithmic techniques: an interval tree, the sweep line technique and a cell-based technique. Of these only the algorithm based on the sweep line technique has a worst-case time complexity that is better than that of the brute-force algorithm. However, in practice the cell based method is found to be the fastest. Before introducing the algorithmic techniques let us first look more carefully at what is to be computed.

2.1 Problem setting

In the fetch length problem for a single point the goal is to compute the shortest distance from the study point p to land area in the given direction θ. If the point p is in the interior of a polygon, i.e. on land, fetch length $d(p, \theta)$ is zero. For a point that lies on the border of a polygon the fetch length is zero if the half line originating at p and having the direction θ points into a polygon. Otherwise it is the same as for an exterior point, with

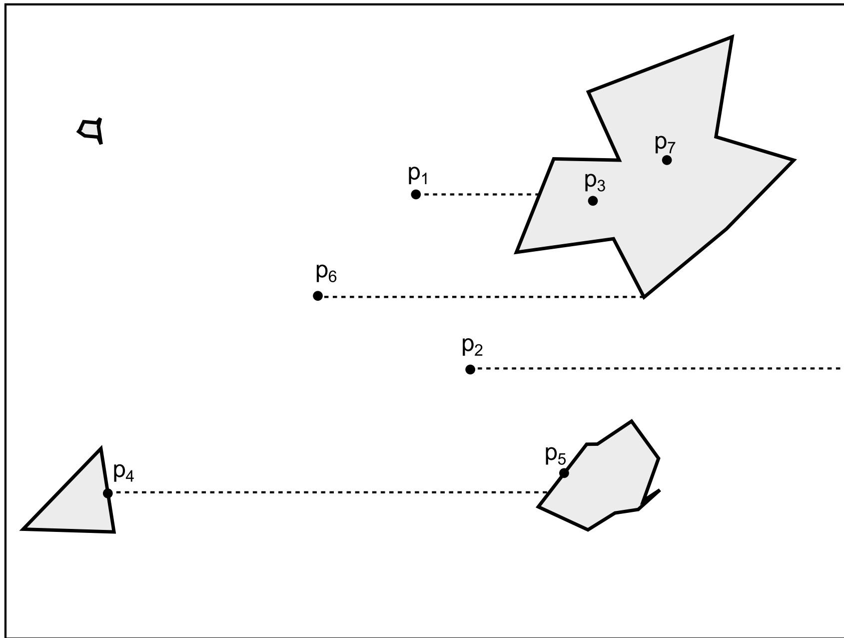

Figure 2.1: Fetch lengths in the horizontal direction $\theta = 0$ for seven study points. Fetch length is zero for the points p_3, p_5 and p_7, positive and finite for p_1, p_4 and p_6 and infinite for p_2.

the segment on which p lies excluded from the distance computation to avoid a zero result. The different cases are illustrated in Figure 2.1.

For determining whether a point is an interior or an exterior point two basic approaches can be used. One is counting how many times the half line originating at p and having direction θ intersects the border of any island polygon: p is then an exterior point if and only if this number is even. The other approach assumes that the input line segments are not given in an arbitrary order but in the order in which they are encountered when walking around the polygon borders. For instance, if the vertices of a polygon are traversed in a counterclockwise order, the interior of the polygon lies on the left side of the line segments of the polygon. This can be used to classify the study points: given the line segment l that is closest to p in the direction θ, it is enough to test whether p lies to the left or to the right of the line that passes through l, left and right being taken with respect to the direction of l.

One must be careful to take into account the special cases that occur in the classification methods. The exceptional case is when the half line hits a vertex of a polygon. This happens for points p_6 and p_7 in Figure 2.1 when the chosen direction is $\theta = 0$. In the segment counting method it is enough to exclude from the count all line segments that lie totally on a preselected

side of the half line while counting all segments that either cross the half line or lie totally on the other side of the half line. For instance, in Figure 2.1 we might choose to exclude all line segments that are, except for a vertex, above the horizontal half line through a chosen study point. For p_6 the count would then be zero so the point would be correctly classified as an exterior point. For point p_7 the count would be one and the point would again be correctly classified as an interior point. In the directed line segment approach point p_7 would cause no difficulty: both segments meeting at the vertex hit by the half line have a similar orientation, leading to the same classification for the study point. The line segments intersecting at the vertex hit by the horizontal half line through p_6, however, have a different orientation: one segment is traversed from top to bottom while the other one is traversed in the opposite direction in the polygon order. It is then important to choose as the closest line segment the one that is closer to p_6 in any direction except $\theta = 0$, in which the distances are the same.

The case of interest in this work is computing fetch lengths for a large batch of study points. The directions in which the lengths are computed are the same for all study points.

2.2 A brute force algorithm

For a moderate number of study points and directions, fetch lengths can be computed by the means of a brute force algorithm. To compute fetch length for a study point p in the direction θ, first form the half line $h(p, \theta)$ whose starting point is p and angle with respect to the positive direction of the x-axis is θ. Set the minimum found distance to $d := \infty$ to indicate that no intersecting segment has been found. Next, iterate over all line segments $l \in L$ of the map. For each segment l it is checked whether $h(p, \theta)$ intersects l. If there is an intersection, the distance from p to the point of intersection, i.e. $d(p, l, \theta)$, is computed. If $d(p, l, \theta) < d$, a new minimal distance has been found, so update $d := d(p, l, \theta)$. Otherwise, nothing needs to be done as d already is the smallest distance found so far.

Once all segments have been processed, d is the smallest distance from p to a line segment $l \in L$ in the direction θ. The actual fetch length $d(p, \theta)$ is d if p is an exterior point or a border point with the half line $h(p, \theta)$ pointing away from a polygon. Otherwise, $d(p, \theta) = 0$. It makes sense to include the classification of the point to the algorithm described above. While the classification could be done only once for the interior and exterior points, it is direction-dependent for border points. Regardless of which classification method is used (intersection counting or orientation test), it is easy to add the classification to the brute-force method.

Although the brute-force algorithm will not be used for computing fetch lengths, it forms a baseline whose performance should be exceeded for an algorithm to be considered successful. The time complexity of the brute-force algorithm is easy to evaluate. While the method was described for one study point, for a set of study points and directions the brute-force method is to perform the above computations for all study points and directions. If there are m study points and D directions, there are $m \cdot D$ pairs of a study point and a direction. For each of them all n line segments are tested for intersection, the intersection test and any other required processing being constant-time operations. Hence, the time complexity of the algorithm is $\Theta(m \cdot n \cdot D)$. Usually D is considered to be a constant, leading to the time complexity $\Theta(m \cdot n)$ for the brute-force algorithm.

2.3 Data structures

It was told before that the number of intersection tests must be kept small to compute fetch lengths quickly. In fact this is somewhat inaccurate: intersection tests are rather simple but their number is so large in the brute-force algorithm that they would still take much time. To achieve a significant speedup it is then not sufficient, e.g., to just eliminate most intersection tests using a simpler preliminary test that can reject most segments. Instead, the goal must be not to iterate over all line segments at all when computing the fetch length for a study point. In the cell-based algorithm only array-based lists are required but the other exact methods use more complicated data structures for quickly finding the line segments of interest.

These data structures are variants of the red-black tree [1]. A red-black tree is a binary search tree that keeps itself balanced while items are added to and removed from the tree. The usefulness of such trees follows from the fact that many tree operations require time that is proportional to the height of the tree. For an n-node red-black tree the height and the time complexity of operations such as finding, inserting or deleting a node is $O(\log n)$.

For the fetch length problem *augmented* versions of the red-black tree are used. In an augmented tree each node contains, in addition to its ordinary data, additional information that makes it possible to perform quickly queries that would be inefficient on an ordinary balanced tree. The fetch length algorithms use two kinds of data structures based on the red-black tree: an interval tree and an order-statistic tree.

2.3.1 Interval trees

An interval tree [1] can be used for efficiently finding in a set I all intervals $i \in I$ whose intersection with the query interval q is nonempty. Each node of the tree contains one interval, the tree order being determined by the lower

end points of the intervals. In addition to this a node stores the highest end point (*max*) of any interval that is in the subtree rooted at the node. This extra information is used for limiting the number of nodes that need to be checked to find the intervals of interest. For instance, if the left child of the current node contains a *max* value that is smaller than the lower end point of the query interval, there is no point in checking the left subtree, since all intervals in the subtree have strictly smaller values than the query interval. Similarly, for any interval in the right subtree it is known that the lower end point is no lower than the lower end point of the current node (because of the search tree property) and that the higher end point is no greater than the *max* value stored in the right child of the current node.

The interval tree structure can be used for finding all k intervals that intersect the query interval in $O(\min(k \log n, n))$ time when the interval tree contains n intervals. Data structures that can achieve a lower running time of $O(k + logn)$ are also known (e.g. [24]) but were not used in the present work.

In the fetch length problem the interval tree can be used to limit the number of line segments that need to be tested for intersection with a given half line. It is assumed that the coordinate system has been rotated so that the half line is horizontal. A line segment $s = (x_1, y_1) - (x_2, y_2)$ intersects the horizontal line $y = y_0$ if and only if $y_1 \leq y_0 \leq y_2$ or $y_2 \leq y_0 \leq y_1$. Line segments with this property can be found by performing a query with the interval $[y_0, y_0]$, after constructing the interval tree to contain the projections of the polygon line segments onto the y-axis. With each interval the original line segment, whose projection the interval is, is also stored to enable further intersection testing: we still need to check whether the found segment also intersects the half line through p with direction $\theta = 0$ and of all such intersections the closest one needs to be found.

In the interval tree method potentially a large number of line segments still needs to be tested to find the fetch length for a given point p in one direction. However, the fetch length itself is the distance from p to one line segment or zero if p is inside a polygon. This observation suggests a way to limit the number of intersection tests further.

2.3.2 Order-statistic trees

Earlier it was assumed that the interval query returns the line segments intersected by the horizontal line $y = y_0$ in an arbitrary order. It was then necessary to iterate over all of them to compute a fetch length. If the line segments were given in a left-to-right order, it would be easier to find the segment that is closest to p in the direction $\theta = 0$. If the segments are stored in a binary search tree this segment could be found very similarly to how one searches for an element that is in the tree: the tree is traversed until a

leaf node is found, going to the left subtree if and only if the intersection of the segment of the current node lies to the right of p. The search does not typically end in the correct segment as one needs to check the left subtree of the closest segment to ensure that there are not segments that are even closer to p. However, the segment of interest is always one of the segments in the search path, so it is enough to maintain a "best candidate" while traversing the tree.

Two problems were ignored in the above discussion. First, it was assumed that the tree containing the line segments intersecting the line $y = y_0$ is available. Second, while the closest segment was found, it is also necessary to determine whether p is an interior or an exterior point. Let us ignore the first difficulty for a while and consider the second one. Recall that we introduced two methods for classifying the study point p. If the segments are stored so that they retain their orientation in the polygon, it is possible to use the simpler second classification method. Otherwise we resort to counting the number of line segments to the right of p. Now, iterating over all segments is precisely what we wanted to avoid, so the number of segments must be computed in another manner.

Fortunately, a well-known data structure called order-statistic tree [1] is well suited for this purpose. A node of an order-statistic tree stores, in addition to normal node data, the number of nodes in the subtree rooted at the node. With this information it is easy to compute the *rank* of any node. Here, rank means the position of an element in a sorted order, i.e. an element with rank i is the ith smallest element in the collection. When computing a fetch length we require the number of segments that lie to the right of p on the line through p. If there are n_p segments in the tree and we know the rank r of the segment that was found in the tree search, there are then $n_p - r + 1$ segments to the right of p. This number is then used to classify point p in the current fetch line direction.

2.4 Sweep line technique

In the above the problem of constructing the order-statistic tree was ignored. One cannot simply build a new tree for each study point p after finding the segments of interest because this construction would require more time than a full fetch length computation for p. What is needed is a method to maintain the order-statistic tree without rebuilding it completely for each study point.

One approach to achieve this is the well-known *plane sweep*, also called *sweep line*, technique [24]. While there are many variants of this technique, not all of them using a straight sweep line [25] or even dealing with two-dimensional data [26, 27], for the fetch length problem an old algorithm for

finding intersections between line segments [13] provides a good basis. Let us review how this algorithm works.

The operation of the algorithm can be visualized by moving a horizontal sweep line upwards, starting from below all objects of the input data and stopping when all objects have been processed. At all times the line segments intersected by the current sweep line are kept sorted by the horizontal coordinates of their intersections with the sweep line. Although the intersection points of the input line segments with the moving sweep line change continuously, only at discrete points are there changes in the order of the segments. These points are the lower end points of line segments, where a new segment is inserted to the order, the upper end points, where a segment is deleted from the ordered structure, and intersection points, where the order of the intersecting segments is reversed. The points where processing is done to compute output data or to update the data structures of the algorithm are called *event points*.

In the fetch length problem there are no intersections between the line segments of the input data. Instead, the intersections of interest are between the horizontal half lines originating from the study points and the line segments of the input data. The end points of the line segments are then the only points where the order of the line segments needs to be updated. At a study point a fetch length is computed, with no changes to the line segment data structure.

The sweep-line algorithm for the fetch length problem proceeds as follows. First, the coordinate system is rotated so that the direction in which fetch lengths are to be computed is horizontal. All event points (end points of line segments and the study points) are sorted into an increasing order of y-coordinates. If two points have the same y-coordinate, their order is determined by their x-coordinates. An empty order-statistic tree is created, corresponding to the fact that the sweep line intersects no segments below all end points. Then, all event points are processed in order. When the next point is a lower end point, a line segment is added to the order-statistic tree. When the point is an upper end point, a segment is removed from the tree. When the point is a study point p, a fetch length is computed as described above. The procedure is given as pseudocode in Algorithm 1.

It is assumed that the map has already been converted into the format required by the algorithm. This means that the map is given as a set of vertices (*endpoints*), each vertex containing references to the (at most 2) line segments that have the vertex as an upper or a lower endpoint. There is also assumed to be an initialized data structure for the fetch lengths $d(s, \theta)$, where the results are stored. It is also assumed that reading the next element of a list advances the list pointer to the next element. Thus, after reading an end point or a study point it is necessary to step back in the list (S or *endpoints*) whose element was not processed in the current iteration of

Algorithm 1 Sweep line algorithm for determining fetch lengths. The code is simplified from P2 by ignoring special cases.

Rotate all coordinates by $\theta - \pi$ radians;
endpoints := a list of all end points of L, with references to the segments;
Sort S and *endpoints* into increasing order of y-coordinates;
T := empty order-statistic tree of line segments;
while there are unprocessed study points in S **do**
    s := next point of S; p := next point of *endpoints*;
    if $p.y < s.y$ **then**
        // The next point is an end point.
        Remove segments ending at p from T;
        Add segments starting at p to T;
        Step back by one element in S;
    else
        // The next point is a study point.
        $e_{left} := T$.predecessor(s); $n_{left} := T$.rank(e_{left});
        if n_{left} is even **then**
            $L(s, \theta) := d(s, e_{left}, 0)$;
        else
            $L(s, \theta) := 0$;
        Step back by one element in *endpoints*;

the while-loop. One also needs to take into account that the closest line segment to the left of a study point may not exist, i.e. $e_{left} = nil$. In such a case the rank of the segment is defined to be 0 and the horizontal distance $(d(s, e_{left}, 0))$ from s to $e_{left} = nil$ is infinite. All operations are performed in the rotated coordinate system. The operation of the sweep line algorithm is illustrated in Figure 2.2.

2.4.1 Time complexity

If there are m study points and n polygon vertices (and hence n line segments) in the map data, sorting all $m+n$ points into an increasing order of y-coordinates requires $O((m+n)\log(m+n))$ time using a comparison-based sort [1]. For an end point p_i of a line segment, inserting or removing the line segment starting or ending at p_i requires $O(\log n_i)$ time where n_i is the number of segments that are in the status structure when handling point p_i, i.e., the number of line segments intersecting the horizontal line through p_i. For a study point p_j the nearest line segment to the right of p_j is found in time proportional to the height of the tree, which is $O(\log n_j)$ for a balanced binary tree with n_j elements [1]. Finding the rank of this segment in the

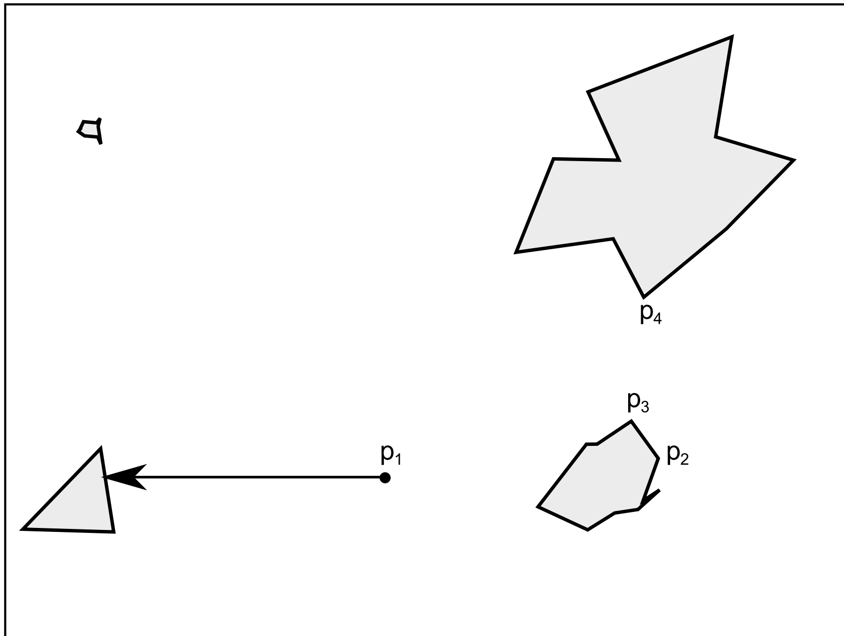

Figure 2.2: Handling event points in the sweep line fetch length algorithm. For the study point p_1 the closest line segment l to the left of p_1 is found using T. The rank of l is 2, an even number, so p_1 is an exterior point and the fetch length is the horizontal distance from p_1 to l. At the end point p_2 one line segment is removed from the status structure T and another one is added. At p_3 two segments are removed and at p_4 two segments are added to T.

tree and hence determining whether p_j is an interior or an exterior point can also be done in $O(\log n_j)$ time.

The objects stored in the order-statistic tree are a subset of the line segments of the map, so $n_i \leq n$ for each study point or polygon vertex. The time required for processing a vertex is then $O(\log n)$ and the total time complexity of the algorithm is $O((m+n)\log(m+n)+(m+n)\log n) = O((m+n)\log(m+n))$. This is assuming that the number d of directions in which fetch lengths are determined is a constant, otherwise the time complexity is multiplied by d.

Although not implemented in this work, it would be possible to improve the time complexity to $O((m+n)\log n)$. To see this, one may first note that the time complexity is dominated by sorting. But the order-statistic tree remains unchanged when processing a study point, making it possible to process the study points occurring between two end points in any order. It is then only necessary to sort the end points. For each study point it is enough to know the two consecutive end points between which the study point lies

vertically. All study points belonging to such an interval can be recorded, for instance, in a list structure, with one list for each vertical interval between consecutive end points and two additional lists for the intervals $(-\infty, y_{min}]$ and (y_{max}, ∞). Finding the correct list takes $O(\log n)$ time and storing the study point to the end of the list requires $O(1)$ time [1]. The end points can be sorted in $O(n \log n)$ time, storing the m study points to the lists takes $O(m \log n)$ total time and processing all study points and end points is done in $O((m+n) \log n)$ time as before, leading to the claimed time complexity. This technique has been previously used in a general line segment intersection algorithm [28], although there the purpose was to improve the accuracy of an algorithm for the general line segment intersection problem, instead of improving time complexity.

2.4.2 A raster-based sweep line algorithm

In the vector-based sweep line algorithm processing a study point requires $O(\log n)$ time due to traversing a path in a balanced binary search tree from the root to a leaf. By accepting a loss of precision it is possible to reduce this time to $O(1)$. This is achieved by a raster-based version of the fetch length algorithm. The basic idea is similar to a known algorithm by Finlayson [4], but there are some differences as well. In contrast to [4], the map is given as a set of line segments instead of a raster representation and is only rasterized when needed. This makes rotating the map data more accurate, since the rotation is performed in full precision for the vector (line segment) data, and rasterization only takes place after the rotation. The memory requirements are also reduced by not having to store the entire rasterized map anywhere. Another difference is that fetch lengths are not computed for every raster cell but only for those that correspond to the user-defined study points $s \in S$. The sweep line status is modified so that it only needs to be updated when rasterizing the line segments of the map. This speeds up the algorithm when many of the cells do not contain study points.

Unlike the vector-based algorithm described in Section 2.4, the raster-based version computes fetch lengths in a direction opposite to the direction in which the sweep line moves. That is, if the sweep line is horizontal and moves upwards, the raster-based algorithm determines the lengths in a downwards direction whereas the vector-based version does this in a left-to-right or right-to-left direction. Otherwise the algorithms are somewhat similar, both using rotation to deal with the fetch line directions that do not align with the coordinate axes. The pseudocode is given in Algorithm 2.

The preprocessing begins by rotating the map and normalizing the rotated coordinates of the study and end points so that all x-coordinates lie in the range $[0, X]$ and y-coordinates in $[0, Y]$, where X and Y depend on the chosen resolution (cell size, called *scale*) of the rasterization. The study

Algorithm 2 A raster-based sweep line algorithm for determining fetch lengths (P2).

```
Rotate all coordinates by θ + π/2 radians;
Find minimal and maximal map coordinates x_min, x_max, y_min, y_max;
(X, Y) := (⌈(x_max − x_min)/scale⌉, ⌈(y_max − y_min)/scale⌉);
(s_points, v_points) := new arrays of Y + 1 empty lists;
Store study points in s_points and end points in v_points;
(last_y, label) := arrays of size X + 1, with initial values (−∞, exterior);
active_lines := a new empty list of line segments;
for index := 0 to Y do
    for each study point s in s_points[index] do
        if label[s.x] = interior then
            L(s, θ) := 0;
        else
            L(s, θ) := scale · (s.y − last_y[s.x]);
    for each end point p_lower in v_points[index] do
        add the line segments stored in p_lower to active_lines;
    for each line segment line in active_lines do
        rasterize(line, index, active_lines, last_y, label);
```

points and lower end points of line segments are stored in two separate arrays of size $Y+1$, s_points and v_points. Each element of these arrays is a reference to a list, with the list $v_points[j]$ containing the lower end points whose normalized y-coordinate lies in the range $[j, j+1)$. Together with a lower end point a reference to the segment starting at the point is stored in full precision to allow accurate rasterization of the segment, i.e., the end points of the segment are not rounded. In the pseudocode it is assumed that the coordinates of the study points are rounded to integers while storing them to s_points, allowing indexing arrays based on these coordinates. This is only for notational convenience as the rounding could also be done when converting a coordinate to an array index.

The sweep line status of the algorithm consists of two arrays of size $X+1$, $last_y$ and $label$ and one list of variable size, $active_lines$. The array position $last_y[i]$ contains the greatest y-coordinate of a nonempty cell with x-coordinate i. Only cells that are below the current position of the sweep line are taken into account. The array is initialized with values $-\infty$ to indicate that there are no nonempty cells below the sweep line. As usual, the sweep line lies below all objects in the beginning of the algorithm. Similarly, $label[i]$ contains the label of the cell i at the position of the sweep line and is either *interior* or *exterior*. Initially, the sweep line is outside all island polygons and $label$ is therefore filled with the value *exterior*. The list $active_lines$

contains those line segments whose lower end point has been met by the sweep line but whose upper end point is above the sweep line. Initially, all line segments are above the sweep line and *active_lines* is empty.

The initialization is followed by the sweep line phase. The horizontal sweep line takes the vertical positions $index = 0, 1, 2, ..., Y$. For a given position of the sweep line, fetch lengths are computed for all study points whose rounded y-coordinate is equal to *index*, consulting the arrays *label* and $last_y$. If $label[s.x] = interior$ for the study point s, the point in inside an island and the fetch length is zero. Otherwise, $s.y - last_y[s.x]$ is the vertical distance between the study point s and the closest nonempty cell below the study point, measured in cells. This distance is multiplied by the side length *scale* of a map cell to yield the approximate fetch length.

Next, the line segments whose lower end point lies on the sweep line are added to *active_lines*. This is followed by rasterizing all currently active line segments. Here, rasterization does not mean converting the entire segment into a cell representation. Instead, the procedure *rasterize* determines the minimum and maximum x-coordinates (x_{min} and x_{max}) of a given line segment that occur in the current sweep line position, i.e., between $y = index$ and $y = index + 1$. The array positions $last_y[x_{min}, ..., x_{max}]$ are set to *index* to indicate that there were nonempty cells at those locations. The array *label* is updated by flipping $label[x]$ between *interior* and *exterior* if the current line segment extends horizontally over the middle position of the current cell. The procedure *rasterize* is also responsible for removing a line segment from *active_lines* if its upper end point lies at the current position of the sweep line. Although the coordinates of the end points are not rounded, it is assumed that they have been rotated and normalized as described above. The algorithm is illustrated in Figure 2.3.

Unlike in the vector-based algorithms, the classification of a study point (or cell) into an interior or an exterior point is uncertain for points that are close to a shoreline. The algorithm should only be used for points that are in a water area. It is possible to modify the algorithm to detect the cases where the classification is uncertain: It happens when the study point lies in a nonempty cell. Note that the emptiness of the cell containing a study point can depend on the chosen fetch line direction, because the rasterization is done after rotating the vector-based map data.

The time complexity of the algorithm can now be determined. Rotating and normalizing the coordinates requires $O(m+n)$ time for the n end points and m study points. Initializing the lists $last_y$ and *label* takes $O(X)$ time, while *active_lines* is initially empty and is initialized in a constant time. The lists *s_points* and *v_points* are initialized in $O(Y)$ time. Since the horizontal order of the line segments and study points is irrelevant for the algorithm and each point is stored exactly once in a list of *v_points* or

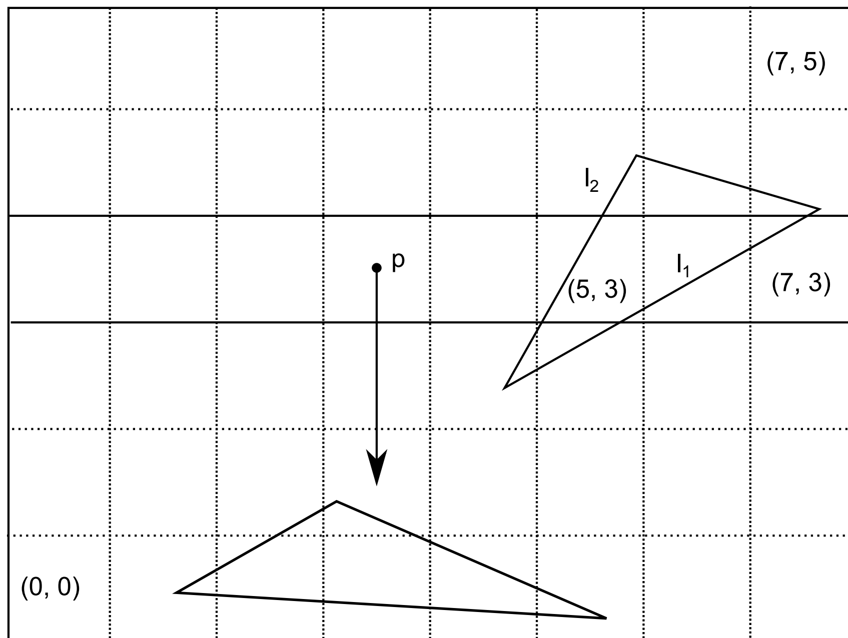

Figure 2.3: Raster-based method in the sweep line position $index = 3$. For point p fetch length is found to be $2 \times scale$. Rasterizing l1 for the current sweep line position means setting $last_y[i]$ to 3 for $i = 5, 6, 7$. The value of $label$ is flipped to $interior$ for indices 6 and 7 but not for 5 since l1 does not extend (in the sweep line position) horizontally over the middle of the cell, i.e., the line $x = 5.5$. Rasterizing l2 sets $last_y[5] = 3$ and $label[5] = exterior$. The sweep line position $index = 3$ is taken to mean the area between $y = 3$ and $y = 4$, shown as non-dashed lines in the figure.

s_points, the points can be added to the lists in $O(m + n)$ time by always appending to the end of a list.

The time required for rasterizing a line segment is proportional to the number of cells intersected by the segment, i.e., for a segment of length len it is $O(\lceil len/scale \rceil)$ The total time complexity of the algorithm is then $O(m + n + X + Y + len_{tot}/scale)$, where len_{tot} is the total length of all line segment in the map, i.e. $len_{tot} = \sum_{l \in L} length(l)$.

2.5 Cell-based algorithm

To further reduce the time required for determining fetch lengths, one possibility is to utilize parallel execution units in a CPU or a GPU (Graphics Processing Unit). A problem here is that the plane sweep technique is serial in nature. Also in the interval tree method it is easiest to implement the tree generation as a serial algorithm, even if the actual distance queries could easily be done in parallel. The preceding techniques then offer limited possibilities for parallel operation. One could certainly process the different fetch line directions at the same time, but this is only enough for a system with a moderate number of processing units. For a GPU implementation, in particular, much more work needs to be performed in parallel to achieve maximum performance.

A fetch length algorithm that is well suited for parallelization was published in 2010 by Yang *et al.* [5]. The method is based on spatial partitioning using a uniform grid [14, 29]. This means that it subdivides the map into a grid of rectangular *cells*. A preprocessing step stores each line segment of the map into all cells through which the segment passes. Then, finding a fetch length starts by finding the cell containing the study point of interest. The cells intersecting the half line starting at the study point are examined in the order of increasing distance from the study point. The examination of a cell consists of a brute-force search for intersections between the half line and the line segments of the cell. If there are such intersections inside the cell, the segment whose intersection with the half line is the closest to the study point is sufficient for determining the fetch length. Otherwise the processing is repeated for the next cell until an intersection is found or the map border is reached.

The cell-based approach has some advantages over the interval tree and sweep line methods. A fetch length can be determined in any direction using the same cell representation of the map. The preprocessing, i.e., constructing the cell-based map representation, has been reported to take only 0.7 s for a map containing over one million vertices on a computer with 2 GHz Intel Core processor [5]. It is also relatively easy to deal with inaccuracies in the input data such as rounded coordinates of study points, which can be

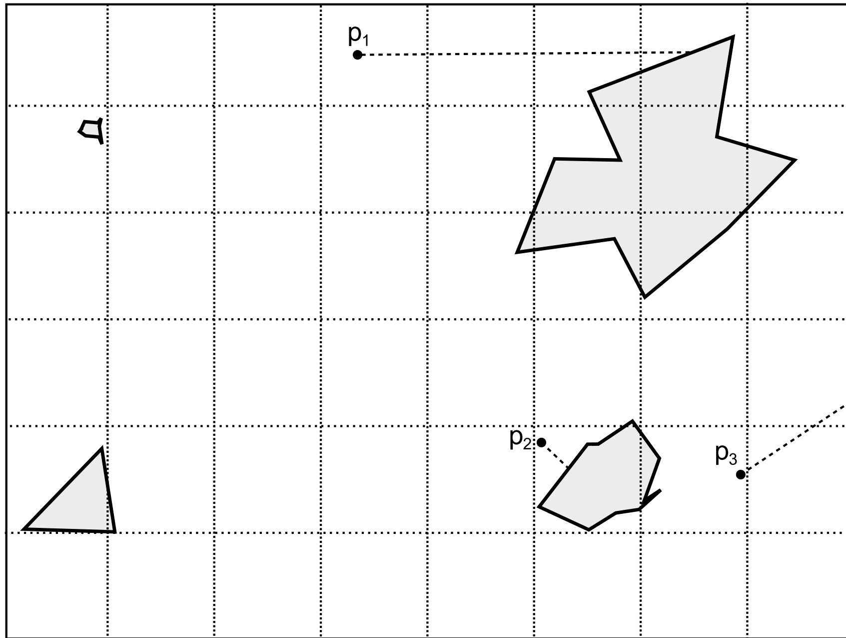

Figure 2.4: Determining fetch lengths in different directions using a cell-based algorithm. For p_1 two empty cells and two nonempty cells are examined. For p_2 only one cell is examined. For p_3 the search ends in the map border after visiting one nonempty and two empty cells.

difficult to do when using the sweep line technique. The main advantage in practice is performance: The cell-based algorithm has been reported to be several times faster than the sweep line method when processing a large number of study points, and the difference is greater when the number of study points is small [5]. The good performance can be attributed to three main factors. The short preprocessing time makes the algorithm suitable for both small and large numbers of study points. For a suitably chosen cell size there are typically only a small number of line segments in a cell, making the brute-force intersection computation in a cell efficient. On the other hand, the cell size must not be too small, so that only a moderate number of cells need to be examined. In [5] the cell size is set by choosing the numbers of cells in the horizontal and vertical direction to be $cells_x = \lfloor \sqrt{n \cdot w/h} \rfloor$ and $cells_y = \lfloor \sqrt{n \cdot h/w} \rfloor$, where w and h are the width and the height of the map, respectively. Hence, there are approximately as many cells as there are line segments in the map, typically leading to a small average number of line segments in a cell[1]. The algorithm can also use simple data structures such as

[1]This average number need not be approximately equal to 1 because a segment may be stored in several cells.

array-based lists, storing the line segments of a particular cell in consecutive memory addresses. This can lead to more efficient memory accesses than in the sweep-line method where a typical operation is the examination of a root-to-leaf path in a binary search tree.

The cell-based procedure for determining a fetch length for one study point in one direction is shown as pseudocode in Algorithm 3. The inner while-loop finds the next nonempty cell along the half line $h(p, \theta)$, moving at each iteration by one cell either horizontally or vertically. Once this cell is found, the for-loop is the brute-force search for intersections between $h(p, \theta)$ and the line segments within the current cell. If intersections are found, the computation is ready. Otherwise the outer while-loop is repeated to find the next nonempty cell.

Algorithm 3 Determining a directional distance using the cell-based method (P3).

```
procedure DIRECTIONALDISTANCE(Map m, Point p, Direction θ)
    Initialize ∂x, ∂y, dx, dy, i, j, px and py using m, p and θ
    minDist := ∞; nearestSegment := null; ready := false
    numLines := linesInCell(m, (i, j))
    while not ready do
        while numLines = 0 and inBounds(m, (i, j)) do
            if dx ≤ dy then
                i := i + px; dx := dx + ∂x
            else
                j := j + py; dy := dy + ∂y
            numLines := linesInCell(m, (i, j))
        for all line segments in the cell (i, j) of m do
            update minDist, nearestSegment and ready if necessary
        numLines := 0
    return getDistance(p, θ, nearestSegment)
```

A theoretical disadvantage is that the worst-case time complexity of the algorithm is poor, even when compared to a brute-force algorithm. The memory requirements also depend not only on the amount of line segments and study points but also on the lengths of the segments of the island polygons.

2.5.1 Time complexity

While the cell-based algorithm performs well in practice, little is known about its theoretical performance. The worst case time complexity can be easily shown to be $O(mn^2)$, while that of the brute-force algorithm is $O(mn)$. As before, n is the number of line segments in the map data and m is the

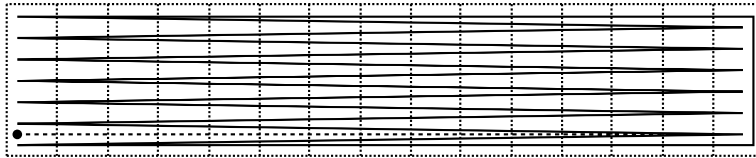

Figure 2.5: A worst case for the cell-based algorithm. Every cell contains all line segments except the vertical segment on the right. The width and the height of the map have been chosen to ensure that there is only one cell in the vertical direction. The cells are not square-shaped due to different zoom factors in the horizontal and the vertical direction.

number of study points and it is assumed that the number of fetch line directions is a constant.

The worst case of the cell-based algorithm appears when the map is so wide that it only contains one cell vertically and n cells in the horizontal direction. Furthermore, the number of line segments in the cells must be maximized. In Figure 2.5 this is achieved by using one polygon whose vertical right side only appears in the rightmost cell but the other $n-1$ long segments occur in all n cells. The fetch lengths are to be determined in the horizontal direction, with the study points located so that all cells need to be examined before the half line intersects the polygon boundary. Now, for all m study points, the algorithm examines $n-1$ segments in $n-1$ cells and n cells in the rightmost cell for an intersection, achieving the claimed worst-case time complexity $O(mn^2)$.

Of more interest is the average case time complexity of the algorithm, given that the worst case occurs in a highly unrealistic situation. This has been considered by Yang *et al.* [5] and they show that processing one study point runs in expected $O(1)$ time. However, their assumptions may not be fully realistic. The first assumption is that the ratio between the height and the width of the map is bounded, i.e., does not depend on n. This assumption is certainly acceptable, but their second assumption states that the number of segments in a nonempty cell is a constant independent of n. It is questionable whether this holds when one increases n by sampling the map in a higher resolution. Furthermore, the number of nonempty cells that need to be examined before finding an intersection almost certainly depends on chosen map resolution: In a high resolution map more cells are required to represent the same distance compared to a map with a lower resolution.

Hence, it is questionable whether the average-case time complexity of the algorithm is $O(1)$ for a study point. Taking into account all study points and the preprocessing, the total time complexity would then be $O(m+n)$. In any case, the practical performance of the algorithm has been tested to be better than that of the interval tree or the sweep line based algorithm [5].

2.6 Tailoring the cell-based algorithm for GPU

When using a traditional multi-core CPU, it is trivial to implement a parallel version of the cell-based fetch length algorithm. Since the preprocessing time is small, it is not necessary to parallelize that step. Once the preprocessing has been done the cell representation is unchanged during the execution of the algorithm. Since there are millions of study points in the cases of interest and the fetch length computations for the different points are independent of each other, a large number of cores can easily be utilized.

On a GPU, however, such parallelization may not yield the best possible performance. Compared to a CPU the difference is that the individual "threads"[2] (or *work-items* in OpenCL) do not execute independently of each other. Instead, a block of work-items, i.e., a *wavefront* will typically execute in lock step [30]. To understand this it is necessary to study the OpenCL programming environment and the internal organization of GPU devices.

2.6.1 OpenCL programming model

The GPU implementations of this work were programmed in the OpenCL language [31, 32], which can be seen as a subset of the C programming language. From the programmers point of view one important limitation is that in OpenCL there are no built-in functions or standard libraries for allocating memory while a program is running. Instead, all required memory must be reserved before running the OpenCL program. Thus, OpenCL can only be used to implement some parts of a piece of software while the rest is implemented in some other language such as C.

Programs implemented in OpenCL are called *kernels* or *compute kernels* and are similar to functions in C language. However, because the OpenCL is a separate execution environment, using a compute kernel is more complicated than calling a function in the host environment. As preliminary steps one must initialize the OpenCL environment for the device that is used for executing it, compile the kernel, and set all parameters of the kernel. Once the kernel has finished its execution, the computed results are transferred to the host environment for further use.

An important difference between a function and a compute kernel is that running a compute kernel is actually similar to performing several function calls instead of only one. Before running the kernel the host environment specifies an index space, with the meaning that the kernel is to be executed for all specified indices. The indices can be 1-, 2- or 3-dimensional and for each dimension the set of indices is a consecutive range of integers from 0 to the maximum index in that dimension. The index is essentially an implicit

[2] In general using the word "thread" is avoided in this work as its meaning is somewhat varied in GPU programming.

parameter for the kernel: It does not occur in the parameter list but can be accessed using OpenCL functions.

An OpenCL term that will be used extensively is *work-item*. It means simply one kernel instance that is executed for a particular index of the index space. How the work-items will be executed depends on the implementation of OpenCL for the device executing the kernel. Often there are much more work-items than there are execution resources on the device. The OpenCL environment can then use multi-threading and loops in order to process the set of work-items.

OpenCL has a relaxed memory consistency model. This means that if one work-item writes some data into the device memory, it is not guaranteed that another work-item can later access that written value. This allows using cache memories that are not coherent across the device, i.e., there may be local caches that have different values for the same memory location[3]. Only after finishing kernel execution memory consistency is guaranteed. There are other possibilities for synchronization in OpenCL but they are limited to typically small consecutive subsets of work-items (called work-groups).

A simple OpenCL implementation of the cell-based algorithm. For simplicity suppose that fetch lengths are computed in only one direction. The index space for the compute kernel is a one-dimensional range $0, 1, ..., m-1$, where m is the number of study points. As its explicit parameters the kernel takes the constructed cell representation of the map, the locations of the study points (in a linear array), the fetch line direction, and an array for the output of the algorithm. The implicit parameter is the index i of the study point. The kernel first finds i (the global index of the work-item) and, with this information, gets the geometric location of the study point p_i from the array of study points. Then, the kernel runs as described above, finding the fetch length for p_i. Finally, the found length $d(p_i, \theta)$ is written to location i in the output array. The compute kernel is thus almost identical to a C function that determines a fetch length for one study point, except that the kernel stores the result into a location of the output array instead of returning it to the caller. As another difference, the coordinates of the study points are not direct parameters for the kernel but they are found from the array of study points using the work-item indices.

The OpenCL language in itself does not impose any limitations that are relevant for the kernel sketched above. The fetch length computations for different study points are independent of each other, so there is no need for synchronization between the work-items. However, the *executions* of different work-items need not be independent of each other, provided that

[3]The advantage of non-coherent caches compared to coherent ones is reduced amount of data transfers within the GPU, saving energy and making the chip smaller.

this can only affect the performance, not the correctness, of the program. Indeed, on current GPU devices such dependencies are common.

2.6.2 The architecture of a GPU

As a concrete example of a GPU architecture let us consider the AMD Radeon 7970 card, which is, except for clock speed, identical to the newer Radeon R9 280X used in this work, both equipped with a GPU called Tahiti. The information of this section is collected mainly from programming and architecture manuals of AMD [30, 33].

Architecture. At the highest level the GPU consists of 32 compute units (CU, Figure 2.6). The different compute units are independent of each other, making a CU to have a similar role as a core in a CPU[4]. Each CU contains one scalar unit (SALU, Scalar Arithmetic Logic Unit) and four vector units (VALU). The vector units are physically 512-bit wide SIMD (Single Instruction, Multiple Data) units. When using 32-bit values this means that a VALU can perform the *same* arithmetic or logical operation for 16 values in parallel. However, the SIMD units are used in such a way that they process the same instruction for four consecutive vectors in consecutive clock cycles. Hence, the width of the SIMD units is logically 64 instead of 16 lanes[5]. The benefit of this organization is reduced amount of control logic in the GPU: It is only necessary to issue a 64-wide vector operation every four clock cycles, instead of a 16-wide vector operation every clock cycle.

Wavefronts. Writing a program in a way that uses 64-wide vector instructions would be tedious for a programmer. Furthermore, there are different GPU architectures, making portability an issue. In the case of AMD APP SDK [30] the solution to this problem is that the compiler takes care of using the vector instructions. The work-items are mapped into individual lanes of the SIMD units. Thus, one instruction of a VALU processes 64 work-items in parallel. A group of work-items being executed using the same SIMD instructions is called a *wavefront*. The benefit of this organization is that it is not necessary to find instruction level parallelism (for the SIMD units) inside a compute kernel to effectively use the vector units. It will be seen later that handling divergent control flow, such as loops or conditional blocks, becomes somewhat complicated for the compiler, since the SIMD execution forces all work-items of a wavefront to perform the same sequence of instructions[6].

The scalar unit is used for operations that are done in a per-wavefront basis. This can include arithmetic that is the same for all work-items, but

[4]An important difference is that in a GPU the caches of the different CUs need not be coherent because of the relaxed memory consistency model.

[5]So, an add command for VALU means that given 64-element vector registers a and b, the VALU computes a 64-element vector c such that $c[i] = a[i]+b[i]$ for each $i = 1, 2, ..., 64$. For the SALU a, b and c are single values instead of vectors.

[6]This is what was meant earlier by stating that the work-items execute in lock-step.

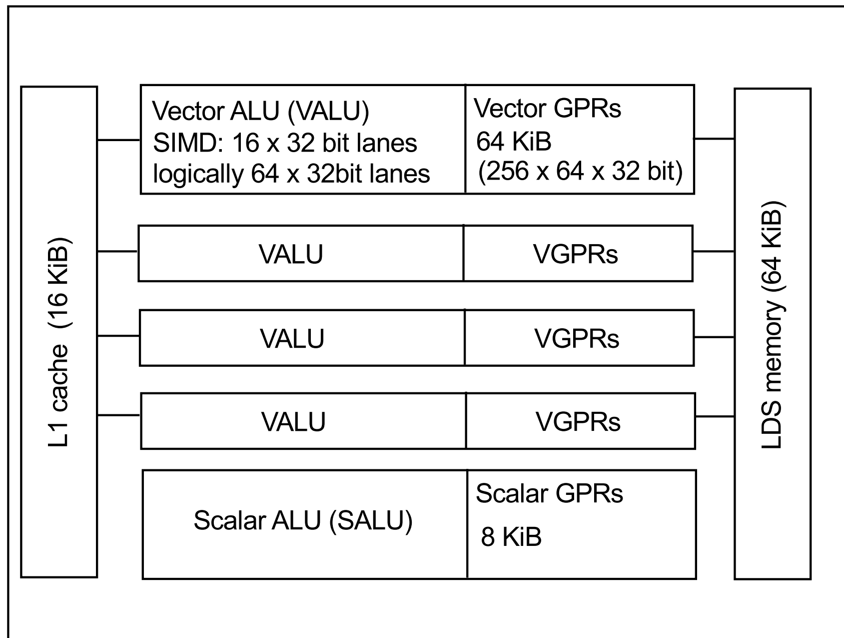

Figure 2.6: A single compute unit (CU) of the Radeon R9 280X GPU. The CU consists of one scalar unit, four vector units and LDS and cache memory. The scalar and vector ALUs contain several general purpose registers.

among the most important functions of the scalar unit is branching. Unconditional branching can be used, e.g., for function calls, whereas conditional branching is required for if-else blocks and loops. Now, the sequence of vector instructions may depend on the scalar instructions as a result of branching, so one scalar unit might not seem sufficient for four vector units in the compute unit. This is where the usage model of the vector units helps. The vector unit is pipelined and can start a new operation every clock cycle, but this is always the same operation over four consecutive clock cycles. For instance, a VALU could start an addition operation for the work items (WIs) $1, 2, ..., 16$ in the first clock cycle, the same operation for WIs $17, 18, ..., 32$ in the next clock cycle, then for WIs $33, ..., 48$ and finally for WIs $49, ..., 64$. Since a new vector operation is started only every four clock cycles, one scalar unit processing one instruction in each clock cycle is enough for the four vector units.

Execution masking. The remaining difficulty for executing complicated programs is that OpenCL compute kernels have full support for loops with varying iteration counts and other constructs requiring conditional branching. So, an OpenCL program can have different branching choices for every work-item while the hardware only supports branching at a wavefront granularity. The solution to this problem is in principle relatively straightforward

The hardware maintains for each wavefront an *execution mask exec*, which is a 64-bit vector with one bit for each work-item. The meaning of the mask is that the next instruction is ignored for SIMD lane i if the mask has $exec[i] = false$. For example, to handle an if-block the condition is first evaluated using vector instructions. This creates an execution mask. Then the block is executed as usual. As a result of the mask, executing the block has no effect for a SIMD lane i with $exec[i] = false$. If the block is a long sequence of instructions, the scalar unit can be used to skip over it if $exec[i] = false$ for all lanes $i = 1, 2, ..., 64$.

The above discussion omits several details. For instance, the old value of the execution mask needs to be saved and later restored so that after the conditional block the correct lanes continue executing. One also needs to support loops and function calls in compute kernels. For information on handling complicated control flow, see [33]. All these details are handled by the OpenCL compiler, but knowing them in some detail is useful in order to understand the performance implications of running branching code on a GPU.

Memory accesses. In addition to SIMD execution, the performance of a GPU implementation may be limited by memory accesses. The Radeon R9 280X GPU has four main types of memory: General purpose registers (GPRs), local (scratchpad) memory, cache memory (two levels) and global memory. The bandwidths of the different memory spaces vary considerably, with GPRs offering about 100 times as much bandwidth as the global memory [30].

The latency of a memory access may be covered to some extent by multithreading. A CU of the Radeon R9 280X GPU can have at most 40 wavefronts in progress, i.e., 10 for each vector unit. The wavefronts of a vector unit are not processed simultaneously; instead, the next active wavefront is chosen for execution after completing an instruction of the current one. This arrangement allows the GPU to process other wavefronts while some of them are waiting for a memory access. However, the actual number of running wavefronts depends on the resource requirements of the compute kernel. The resources of interest are the scalar registers (SGPRs), vector registers (VGPRs) and the local memory. These resources are shared by all wavefronts running on the same CU, so a compute kernel with high resource requirements limits the number of active wavefronts, which has a negative impact on the latency hiding of memory accesses. On the other hand, global memory is much slower than the registers or the local memory. Thus, using global memory instead of registers may lead to decreased performance even if the number of active wavefronts is increased by such a change. Nevertheless, *kernel occupancy* is one possible target for optimization in a GPU algorithm. Kernel occupancy is, by definition, the number of active wavefronts divided

by the maximum possible number of active wavefronts supported by the device [31].

2.6.3 Cell-based fetch length algorithm on a GPU

While a GPU can run compute kernels containing conditional statements, their effect on performance can be significant. A practical worst case for an if-statement occurs when all but one work-item of the same wavefront need to be masked out of execution because the condition is $false$ for these items[7]. This can lead to a kernel running at only 1/64 of the theoretical performance of the device, 64 being the number of work-items in a wavefront on the Radeon. In cases where work-items need to be masked out of execution the control flow of the program is said to be divergent.

Control flow divergence. There are several sources of divergent control flow in the cell-based fetch length algorithm. After examining a cell, the rasterization algorithm moves either horizontally or vertically to the next cell, depending on the current values of some variables. If the wavefront contains at least one work-item making a horizontal transition and at least one making a vertical transition, the code for both transitions must be executed for the wavefront. A far greater problem is that the numbers of line segments in the nonempty cells can be highly variable, requiring the brute-force searches done in the cells to use loops with different numbers of iterations for different work-items. The number of iterations for the wavefront is then determined by the work-item requiring the highest number of iterations. As such a loop proceeds, an increasing number of work-items are masked out of execution, until the processing for the last work-item becomes ready and the loop can be exited. This means a decreasing utilization rate for the SIMD units. The same is true for the cell traversal. In the best case the starting cell contains a line segment intersecting the half line starting from the study point, while in the worst case the rasterization proceeds from one edge of the map to the other edge without meeting any nonempty cells. With a map containing millions of line segments, the maximal number of iterations to find a nonempty cell can then be thousands of times larger than the minimal number. Again, in such a case a work-item requiring only one iteration would in effect need to wait until the work-item requiring the maximal number of iterations (in the same wavefront) has finished processing.

Solutions. The GPU implementations will be discussed in more detail later, but here is a short summary of the results. The simplest way to implement the cell-based algorithm is to ignore the SIMD execution and implement it as if the work-items were truly independent of each other. This

[7]It is possible that *none* of the work-items satisfy the condition but the block is still executed. Such code may be desirable when the block is so short that executing it takes only a similar amount of time as a conditional branch.

was done in P3 and the result was surprisingly good. The algorithm was up to 8 times faster on a Radeon 5850 GPU than a parallel implementation on an AMD Phenom 9850 quad-core CPU. One must note, though, that the GPU was a high-end device at the time, while faster CPUs were already available at reasonable prices.

The cell-based algorithm was later revisited with the goal of better optimizing it for a GPU (P4). The hardware used in the study was faster than before, an Intel Core i7 3770K CPU and AMD Radeon R9 280X GPU, both high-end consumer devices at the time the work was started. Many of the attempts to improve performance were unsuccessful, but two approaches had a positive effect: *presorting* of the study points and *sparse rasterization*. The goal of the presorting was to group together study points that are geometrically close to each other. They could then perform similar memory accesses, leading to better cache utilization than with a random placement of study points into wavefronts. The benefit of the sparse rasterization is that it does not examine all empty cells along a half line, reducing the number of memory accesses compared to using the original rasterization algorithm by Cleary and Wyvill [34]. Both techniques can also have a positive effect on the wavefront divergence problem: sparse rasterization can quickly step over large empty areas, while the benefit of the presorting is that fetch lines originating from geometrically close study points often end in cells that are close to each other. The varying number of line segments in the nonempty cells was not a significant problem in practice.

Chapter 3

The water quality monitoring network problem

In the monitoring network problem the input consists of a set S of observational sites and a set of observations for each site. The goal is to select a subset $S' \subset S$ with a minimal (negative) effect on the quality of observations compared to using the whole network. The particular water monitoring network studied in this work is shown in Figure 3.1.

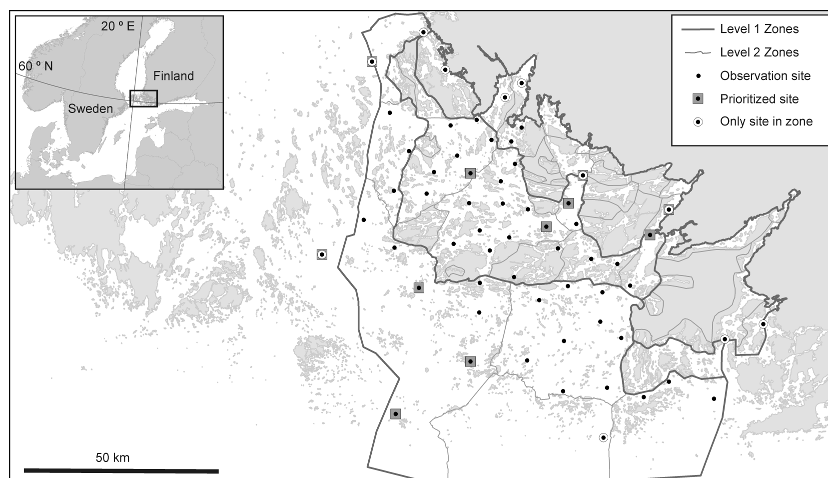

Figure 3.1: The water monitoring network in the Archipelago Sea. Prioritized sites should not be removed from the network. Additionally, at least one site should remain in each level 2 zone (P5).

There were additional prerequisites for the site removal, as shown in the figure. First, some sites were considered to be especially important, perhaps as a result of having a particularly long history of taking observations. These prioritized sites should not be removed from the network. There should

also not be large areas without any monitoring sites. This requirement of good spatial coverage was taken into account simply by requiring that there remains at least one site in each predefined area called a level 2 zone. There were some zones with only one observational site in the network, making these sites essentially similar to the prioritized sites. When describing the site pruning algorithms, these details are omitted because they are easy to take into account in implementations.

When starting to work on the problem it was clear that neither a physical simulation model nor sufficient data for using such a model were available. Only the locations of the sites and the observations collected so far could be used, possibly in addition to a two-dimensional map. As a result of this limitation, a simple error measure was chosen for the network optimization problem: The observations of a removed site are estimated using the corresponding observations of sites that remain in the network. The error for a site is related to the differences between the actual observations and the estimated values. The error function for the entire network is a sum of all site-specific estimation errors.

3.1 Statistical model

For modeling the observations of a site, multivariate linear models [35] were chosen. In such a model the observations of a site are expressed as a linear combination of the observations of other sites, in addition to a constant term. However, there were only 27 observations of each quantity at each site. For a network with 60 sites it would not be possible to determine the model coefficients in a unique way. Therefore, the initial idea was modified so that only a small number of sites closest to the current one are taken into account in the model. Hence, the observations of a site are modeled as

$$q_{s,i} = \left(\alpha_{s,q} + \sum_{s' \in nn(s,k)} \beta_{s',q} q_{s',i} \right) + u_{s,q,i}. \quad (3.1)$$

Here, $q_{s,i}$ stands for the ith observation of the quantity q at site s and the set $nn(s,k)$ contains the k nearest neighbor sites of s. The term $u_{s,q,i}$ is the *residual*, i.e., the error done when modeling the observation using the linear model. The constant term for the site s and quantity q is $\alpha_{s,q}$, while $\beta_{s',q}$ is the coefficient for the observations of the site s' and quantity q. The coefficients (α, β) can be determined using statistical software.

Next, an error measure needs to be defined. For an individual site s and time of observation i the error $e_{s,i}$ is defined as the Mahalanobis distance [35] and then, for the site s as $e_s = (\sum_i e_{s,i})/n_s$, where n_s is the number of observations at site s. For the entire network the estimation error is then

$e_{del} = \sum_{s \in del} e_s$, where *del* is the set of sites that has been removed from the network.

3.2 Missing and incorrect values

As noted before, some (ca. 4.5%) of the observations were missing in the dataset used in the study. The problem is made worse by the fact that Equation 3.1 uses the observations from the k nearest neighbors of site s. Using this model it is not possible to compute an estimate for a quantity if any of the neighbor sites used in the model contain a missing value for the same quantity and time. With an already sparse dataset this is a significant problem.

One way to deal with the problem is *imputation* of missing values [16]. It is a process where the missing values are replaced with estimates that are based on the values of other quantities. In the water quality data missing values often occur in groups, i.e., all observations of a given site are missing because the site could not be visited at all. The relationships between the different quantities are then not useful for imputing many of the missing values. Nevertheless, by including, e.g., the times and coordinates of observations it was possible to obtain better estimates than the mean values of the observed values.

When using imputation there is a choice between using a single or a multiple imputation method. The difference between these two approaches is that in single imputation a missing value is replaced by the best estimate for that value. In multiple imputation the uncertainty of the value is estimated in addition to the value. For each observation to be imputed, several estimates are then drawn at random from the probability distribution determined by the imputation method. In this work multiple imputation was favored because when using it, it is possible to see whether the optimization results are robust with respect to the imputation errors.

The other data quality problem, incorrect values, was handled using a method based on the site-wise means and standard deviations of the quantities. An observation was considered to be an *outlier* if it differed from the mean value of the quantity by at least a certain number of standard deviations for the quantity. After an examination by a human expert, all found outliers were removed.

Removing an outlier adds a missing value to the dataset. To avoid this, outlier elimination was done before the imputation. Typically, at most one of the observed values at a given site and time was eliminated in the outlier elimination process. Hence, imputing the values that have been eliminated by the outlier removal process can use the observations of the other quantities at the same site and time.

3.3 Network optimization

Having chosen a way to estimate the observations of the removed sites and an error measure, it remains to remove sites from the network. All site pruning methods used in this work follow the greedy approach where sites are removed one at a time. At each step the site to remove is the one that is, according to a chosen cost function, the best candidate for removal. The approach is heuristic in nature, i.e., there is no guarantee that the final pruned network is the best possible network in terms of the chosen metric. Three different heuristics were tested: Two variations of greedy pruning with local costs (*pruning_local_taboo*, *pruning_local_notaboo*) and one variation of greedy pruning with global costs (*pruning_global*).

In the first heuristic (*pruning_local_taboo*), the idea is to always remove the site s whose observations can be best modeled using the observations of the nearest neighbors, according to the given cost function. Now, there is the problem that if one of the k nearest neighbors of the removed sites are later removed, the model for the removed site is no longer the same as it was when the site was removed. Thus, in the first version of the algorithm the nearest neighbors of a removed site are no longer candidates for removal. This is achieved by adding the neighboring sites to a *taboo list*, with the site to remove always being chosen from the non-taboo candidates.

One consequence of using such a taboo list is that it is only possible to remove $\lceil n_s/(k+1) \rceil$ sites from the network. If one needs to remove a greater number of sites, one possibility is to empty the taboo list once there are no sites to remove. However, the idea behind the taboo list is lost when doing so, because the neighbors of a removed site in the final network will be different from those used when constructing the linear model for the observations. Therefore, another variation of the algorithm (*pruning_local_notaboo*) does not use the taboo list at all and always removes the site with the smallest estimation error. Except for the removal of the taboo list, the method is identical to the first one.

The second heuristic (*pruning_local_taboo*) allows choosing the site to remove from a larger set of candidates than the first one. However, the value of the cost function for a site may increase when one of its neighbors is later removed from the network, because a different set of neighbors will then be used for modeling its observations. One way to take this into account is to modify the cost function as is done in the third heuristic (*pruning_global*). The new cost function is defined for the entire set of removed sites and is simply the sum of the cost functions of the individual removed sites. Thus, a site with a low estimation error but whose removal would increase the estimation error for other, already removed, sites may no longer be the best candidate for removal. Other approaches for obtaining a better network are also known, such as allowing to insert a removed site back to the network,

known as an exchange-type algorithm [21]. However, this possibility was not considered in this work because the running time of the remove-only algorithm was already found to be excessive.

Chapter 4

Summary of publications

4.1 P1: Quantifying Distances from Points to Polygons - Applications in Determining Fetch in Coastal Environments

The interval-tree based fetch length algorithm described in Section 2.3.1 was designed and implemented in Java. Performance testing was carried out using a computer with 1 GiB main memory and a 1.92 GHz single-core Athlon XP processor. The testing data, a coastal map, contained 1277 islands represented as polygons with a total of 53301 vertices. Since the algorithm was to be used with a far larger map, artificial maps were generated by tiling various numbers of copies of this map side by side. There were always the same number of tiled copies in both the vertical and the horizontal direction, and no extra gaps were added between the tiles. The same map and the approach of tiling several copies of the map was later used in the three other publications dealing with the research question RG1.

The running time for generating the interval tree ranged from 4.7 s for a map with 53301 vertices to 730 s when there were 6.4 million vertices. In other words, increasing the size of the map by a factor of 121 increased the preprocessing time by a factor of 160. The size of the map had a significant effect also on the time required for computing fetch lengths. For instance, computing the lengths for 50000 study points in 48 directions took 25.8 s (excluding preprocessing) with a map containing the smallest tested amount of vertices but 370 s with the largest map. This increase is not surprising since the algorithm iterates over all map line segments intersecting the horizontal line (in the rotated coordinate system) through a given study point when determining the fetch length for the point. Since the same number of maps was tiled in both directions, the number of line segments to examine is proportional to $\sqrt{n}$, where n is the number of vertices in the map, at least when considering vertical or horizontal fetch line directions. The process-

ing time for a study point indeed roughly followed the square root[1] of the number of vertices.

The method was also applied for determining fetch lengths using a real map containing 3 million vertices. There were 270000 study points on a shoreline and 2.56 million study points on a regular grid with a 25 m distance, covering an area of 40 km $\times$ 40 km. A different computer was used in this case, and the shoreline points were processed in 39 minutes and the grid data in about 5 hours. In order for the data to fit in the 1 GiB memory of the computer, the study points were processed in batches of at most 100000 points.

During the work there was a small change in requirements. Instead of computing a zero fetch for a point that is on land, the distance to the nearest shoreline was computed similarly to the points on water, but it was multiplied by -1 to indicate a point in a land area. Furthermore, for the grid data average fetch lengths were to be computed, but some of the generated points happened to be on a shoreline. In this case a zero fetch was to be output for any points on a shoreline to avoid computing averages of positive and negative lengths. Computing negative lengths was an easy change but identifying a point as lying on a shoreline was a bit more difficult. It required using a pre-defined tolerance ϵ so that a point was considered to be on land if its distance to the nearest shoreline was at most ϵ. However, the interval tree method originally only took into account those segments that intersect the horizontal line through the study point. This was changed (Figure 4.1) by modifying the tree query interval from $[-p_y, p_y]$ to $[p_y - \epsilon, p_y + \epsilon]$.

The interval tree method was found to be more than 1000 times faster than the GIS-based method that was previously in use (P1). It was later used for computing fetch lengths for a far greater number of study points at the department of Geography of the University of Turku. The work was carried out during a longer than one month time period, indicating a need for faster algorithms for the fetch length problem.

4.2 P2: Determining directional distances between points and shorelines using sweep line technique

This article continued with the research question RG1 by developing new algorithms that are highly efficient when the number of study points is large. The vector- and raster-based sweep line algorithms described in Section 2.4 were implemented and compared to the interval-tree based method (P1). The computer and implementation language were the same as in the first article. As told earlier, the vector-based method needs to consider a far

[1]To be more precise, there is also a logarithmic factor in the time complexity due to searching a balanced tree.

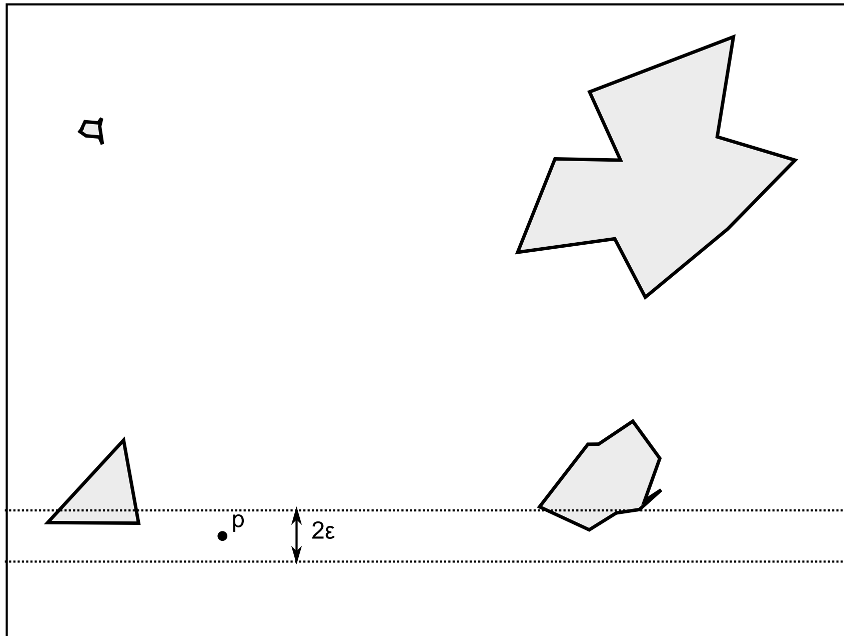

Figure 4.1: To find all borderlines that are within the distance ϵ from the study point, a query interval $[p_y - \epsilon, p_y + \epsilon]$ is used for filtering out segments that are certainly further away from p. All line segments passing this initial test are then checked for proximity to p.

smaller number of line segments than the interval tree method when computing a fetch length for a study point. On the other hand, processing the end points requires more time than the interval tree generation. The net result was that the interval tree method was in some test cases more than twice as fast as the sweep line method when the number of study points was small (10000). When the number of study points was increased, the sweep line method outperformed the interval tree method. With 3 million study points and 3.4 million vertices, the sweep line method achieved over 10-fold performance compared to the interval tree method.

The performance of the raster-based method depends on the chosen rasterization accuracy (cell size). With a 10 m cell size the raster-based method was generally faster than the vector-based one, although the difference was less than 2-fold in all cases. Reducing the cell size to 2 m caused the raster-based algorithm to always be slower than the vector sweep method, although the time difference was minor in most cases. However, the most significant shortcoming of the method was that the results are not reliable for study points that are close to a shoreline. The reason is that the classification of a point as an interior or an exterior point is inaccurate and depends also

on the chosen fetch line direction. The vector-based algorithm is then more appealing than the raster-based one despite being slightly less efficient.

The requirement of detecting whether a point lies on a shoreline was dropped before designing the sweep-line algorithms. The reason was that in most cases users did not want zero fetch lengths for study points lying on a shoreline. If such functionality was desired, it might be difficult to achieve using the sweep-line method. The reason for this is that the sweep line status only contains the map line segments intersected by the current sweep line, making it impossible to find segments that are completely below or above the sweep line. If such segments were included, it could be difficult to maintain a sensible ordering of the line segments.

The handling of special cases was omitted in Section 2.4, so let us briefly consider them. First, if a line segment of the map happens to be horizontal, it can be omitted from the status structure. The only distances that it can affect are horizontal fetch lengths for points that are on the segment. Such a distance is not well-defined without agreeing on whether the shoreline itself is inside or outside the island. This problem is not relevant in practice, because any (small) inaccuracies in measuring the shoreline could change the result. The fetch length can also be multiplied by the cosine of the angle between the normal of the shoreline and the fetch line direction to obtain a quantity known as effective fetch (P1), and this cosine is zero.

Another special case occurs when there are vertices on the horizontal line through the study point. In this case the method of classifying the study point as an interior or an exterior point can fail if the method of counting intersections is applied without thought. It turns out that it is enough to count a segment with a vertex on the sweep line if and only if the segment extends below the sweep line. In addition, all segments crossing the sweep line are counted as usual. For computing the distance, however, a segment is taken into account regardless of whether its upper or lower vertex is on the sweep line[2]. This is done by processing a study point twice if its y-coordinate is the same as that of some vertex. The first pass finds the intersections with the segments whose upper vertex is on the sweep line, while the second pass takes care of those segments with a lower end point on the sweep line. Between these passes the segments ending at the sweep line are removed from and the starting segments are added to the sweep line status, an order-statistic tree.

The special cases shown in Figure 4.2 also apply for the interval tree method. The only exception is that only in the sweep line algorithm it is necessary to explicitly take care of including all line segments having an upper *or lower* end point on the sweep line method when determining the

[2]Like for the horizontal segments, any sampling inaccuracies would have an effect on whether the fetch line should end at the vertex or not. Hence, the special case is unlikely to be important in practice.

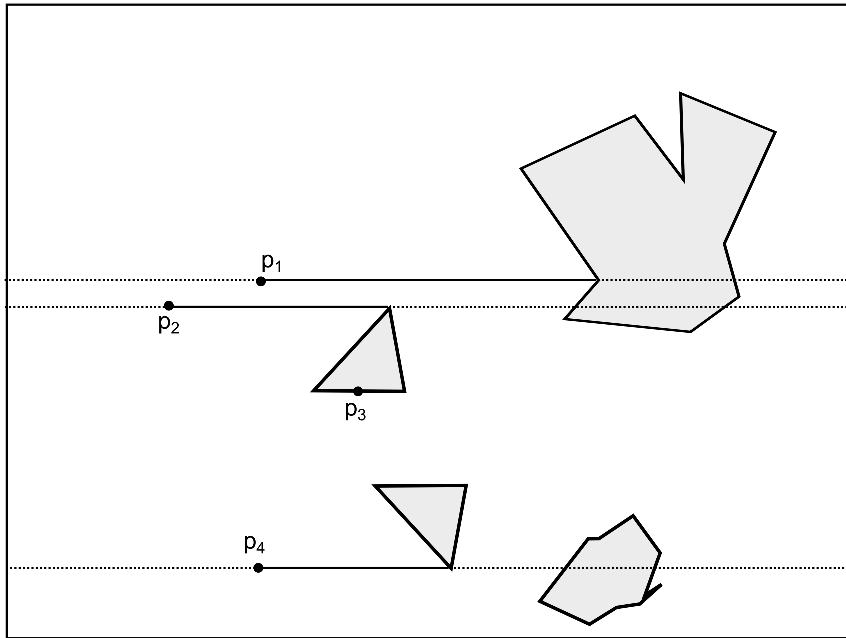

Figure 4.2: Handling of special cases in the vector-based sweep line algorithm. The horizontal line through p_1 meets a vertex that is counted as one intersection. For p_2 and p_4 the vertices are counted as 2 and 0 intersections, respectively. For p_3 the fetch length is determined as if the horizontal coastal line did not exist.

distance. For the interior/exterior classification the segments whose lower end point lies on the sweep line are excluded, which is similar to the interval tree based method.

4.3 P3: A parallel GPU implementation of an algorithm for determining directional distances

The third article started the work on RG1 for parallel devices, particularly GPUs. While the algorithms introduced in the previous two articles were not very promising for parallelization, a cell-based algorithm [5] had been published by other researchers. The method has a short preprocessing (map building) time, making it unnecessary to parallelize that phase. Once the preprocessing is ready, the processing of the different study points can be done in parallel. Parallel versions of the cell-based algorithm were implemented for both a CPU and a GPU.

The computer used in the tests was equipped with a 2.5 GHz AMD Phenom 9850 quad-core processor, 4 GiB of main memory and an ATI Radeon

5850 GPU with 1 GiB memory. The architecture of this older GPU is somewhat different from that described in Section 2.6.2. In particular, the machine-language instructions are in VLIW5 (very long instruction word) form, which means that one instruction can specify up to five different operations applied on different data. The instructions are still applied for wavefronts of 64 work-items, meaning that an instruction can effectively perform up to 320 operations. An important consideration is that the GPU can only achieve its peak performance if, for every work-item, it is possible to perform the maximal number of operations (5) in parallel. It is often necessary to use explicit vector operations to assist the compiler in producing well-parallelized code, a stark contrast to the newer GPU (see Section 2.6.2), where parallelism within a work-item is neither required nor useful.

The parallel fetch length computation was implemented using the OpenCL programming language for both the CPU and the GPU. The compute kernel was tuned separately for the different devices, consisting mainly of low-level changes. The performance benefit of the changes was generally small.

As is expected for a parallel device, using the GPU was only beneficial when the number of study points was large enough. With 1000 study points the CPU was generally faster but already with 100000 points the GPU could achieve a speedup by a factor ranging from 3.6 to 7.7 depending on the size of the map (greater speedup with a larger map). Increasing the number of study points to 5 million further increased the performance difference between the GPU and the CPU slightly. At best the GPU was almost 8.3 times as fast as the CPU with four cores.

The theoretical peak performance of the used GPU was more than 20 times higher than that of the CPU (2088 GFLOPS [30] vs. 80 GFLOPS) - a much greater difference than the observed performance difference. Note that also the CPU requires parallel operations within a work-item in the form of (4-wide, i.e. 128 bit) SIMD instructions. They are more restricted than the VLIW instructions since a SIMD instruction performs the same operation for different data while in a VLIW instruction the operations can be different. This means that while insufficient VLIW packing can prevent the GPU from achieving its maximum performance, this does not explain the smaller than ideal speedup; the CPU has similar requirements for achieving its peak performance.

A significant bottleneck for the GPU implementation was due to the memory accesses as the algorithm performs only a small number of arithmetic operations between the accesses. Also the synchronous execution of the work-items in a wavefront[3] can be a significant problem for a GPU implementation, although that was not studied in the article. One might also note that the

[3] This is the wavefront divergence problem described in Section 2.6.2.

GPU used in the tests was more high-end than the CPU, making the speedup seem somewhat better than would be achieved with more balanced hardware.

4.4 P4: Performance tuning and sparse traversal technique for a cell-based fetch length algorithm on a GPU

The fourth article continued investigating a GPU implementation of the cell-based algorithm. A new sparse rasterization algorithm for line segments was developed, and a great speedup was achieved by using both sparse rasterization and sorting of study points. Now, a quad-core 3.5 GHz Intel Core i7 3770K processor and an AMD Radeon R9 280X GPU were used. For the implementation the main change is that the GPU does not require parallel operations within a single work-item, unlike the older Radeon 5850.

The first optimizations were low-level modifications that decreased the resource usage of the compute kernel in the GPU, achieving higher occupancy than before and, as a result, better performance. Next, the data structures were reorganized to decrease the amount of memory accesses and to improve their speed by using the OpenCL image data structure.

Unsuccessful divergence optimizations were then done, as described in the appendix of the article. One of them replaced a study point with a new one when its processing was finished. The problem in the approach was that for each study point a varying number of cells are traversed in order to find the next nonempty cell intersected by the half line starting at the study point. Similarly, the number of intersection computations is different for different nonempty cells. The number of cells to traverse to find the next nonempty cell was observed to be far greater when finding the first nonempty cell than when looking for the next cells to examine. Because of the synchronous operation of the work-items in a wavefront on a GPU, all points were then forced to proceed at the same slower speed as the new points.

Another attempt was targeted at the inner loops of the algorithm. In particular, instead of finding the next nonempty cell for one study point, it was done for several points. A nested loop is not suitable here because it is subject to the same synchronizity requirements as the original one. Instead, a variable was introduced for keeping track of the ordinal number of the current study point, incrementing it when the rasterization was ready for the current point. The basic reason for expecting this change to be useful is that the sum of iteration counts tends to be less variable than the iteration counts themself. Unfortunately, keeping track of a greater number of study points requires a greater number of hardware registers, decreasing kernel occupancy. Furthermore, the GPU does not allow an indexed access to the registers, so

the slower LDS memory had to be used instead. Although this change also decreased performance, it was seen to be potentially useful: The original implementation was slower than the new one if the kernel occupancies of the implementations were limited to the same value.

A more successful change was sparse traversal. The idea is to make the search for the next nonempty cell more efficient by skipping over several nonempty cells at a time. The map is augmented by including in each cell a number indicating the minimum number of transitions required to hit a nonempty cell when starting from the current cell (Figure 4.3). A transition is here a horizontal, vertical or diagonal move from a cell to one of its 8 neighbor cells. With this information the rasterization can immediately proceed to a cell that is outside the area that is known to be empty of line segments. Note that the rasterization will still examine many cells that are empty. For instance, in Figure 4.3 only three of the cells at the distance 3 are nonempty. A more accurate distance map would accelerate the search for nonempty cells, but the preprocessing time would increase because such a map would not be the same for all fetch line directions.

The greatest improvement in performance was obtained by presorting the study points. The objective of the sorting is to have wavefronts contain points that are geometrically close to each other. Then, similar memory accesses are usually required when processing the points. The wavefront divergence problem is also reduced because in many cases the fetch lengths for nearby points will be similar. Like for the sparse traversal, the sorting must be fast in order to have a positive effect on the total performance of the algorithm. Cell-based sorting was used where points belonging to the same sorting cell are consecutive in the sorted order. The sorting cells were allowed to be different from the map cells in order to adapt the sorting for different densities of study points. No particular order is defined for points belonging to the same cell. Still, one must choose an order for the points that are in different cells. For this a zig-zag ordering (Figure 4.4) of the cells was used. The first row of cells is traversed from the left to the right, then the second row of cells from the right to the left, and so on. This order avoids cases where geometrically distant cells are close to each other in the sorted order.

The improvement of performance caused by the different optimizations depended on the number and locations of study points and on the size of the map. With randomly located 100 million study points all low-level optimizations together improved performance by more than 60%, while for study points lying in a regular grid the corresponding improvement was 25%. With a smaller map only small improvements were obtained using the low-level optimizations. Sparse rasterization generally yielded an additional improvement of $40\% - 50\%$, although both smaller and larger speedups were observed in some of the tests. Sorting greatly improved performance (up to

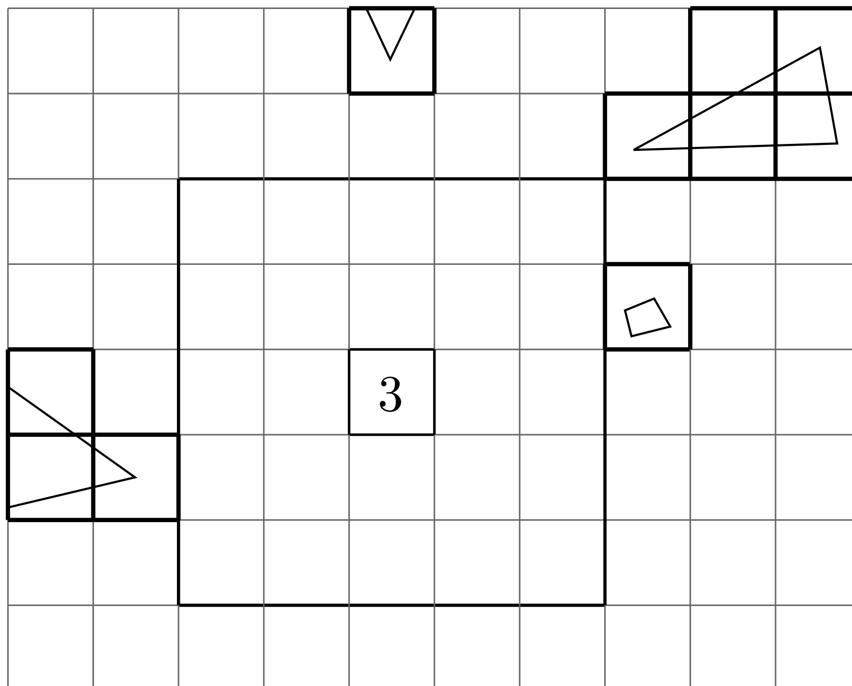

Figure 4.3: Sparse rasterization skips over several empty cells in one step. The number 3 shown in the current cell indicates how far from the cell the sparse rasterization step may proceed. The distances for other cells are not shown. The nonempty cells are shown as small bolded squares while the larger bolded square indicates the area over which a single rasterization step can skip (P4).

more than 7-fold) with randomly located study points while a smaller effect was found for the grid data (ca. 35% − 70% improvement). Using sparse traversal in addition to sorting gave the best performance, in some cases more than twice as good as sorting alone.

The map used in the tests was quite dense in islands. Additional tests were done by increasing the number of cells used to represent the map, to simulate sparser and denser maps. With a very sparse map more than 40-fold speedup was achieved when using all the optimizations. Only randomly located study points were used in these tests, although the grid data is likely closer to what a researcher of geography would use.

The running times of the different algorithms are summarized in Table 4.1. Since the tests were performed on different computers, the results include the effects of both algorithm optimization and the increased performance of computers. The first computer was equipped with a 1.92 GHz AMD Athlon processor and 1 GiB memory, the second with AMD Phenom

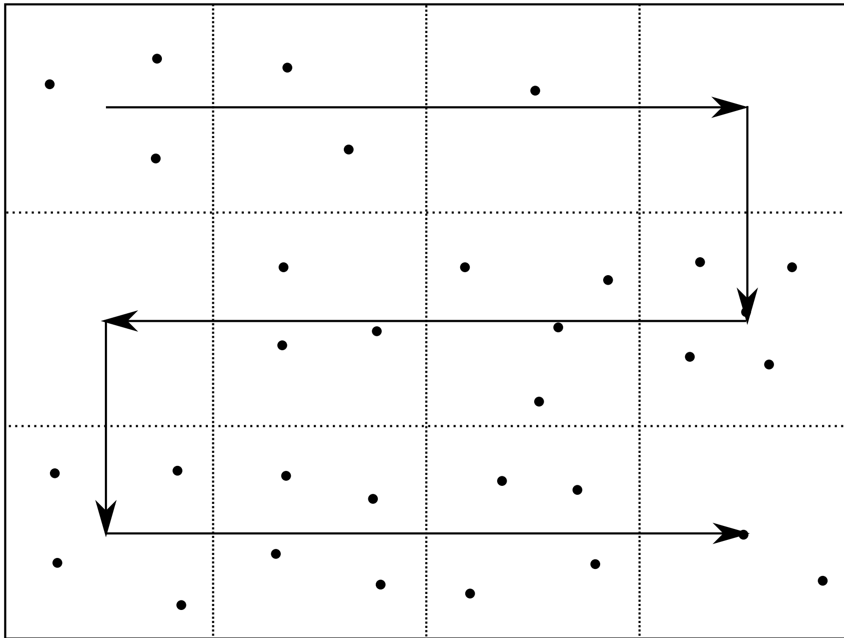

Figure 4.4: Partial sorting of study points. The points belonging to the same cell occupy consecutive positions in the sorted order, but there is no specific order for different points in the same cell. The cells are visited in a zig-zag order (shown by arrows) when merging the point lists of the different cells into a single list.

9850 CPU, 4 GiB main memory and an ATI Radeon 5850 GPU (1 GiB, 725 MHz, 1440 ALUs, 2 TFLOPS). The third machine had an Intel Core i7 3770K CPU, 8 GiB main memory and an AMD Radeon R9 280X GPU (3 GiB, 1100 MHz, 2048 ALUs, 4.5 TFLOPS). Tests were also performed with different numbers of vertices and study points, depending on what was feasible with the particular computers used for the tests.

4.5 P5: Optimising an observational water monitoring network for Archipelago Sea, South West Finland

In the fifth article the number of sites in a water quality monitoring network had to be reduced in order to reduce economic cost. A dataset containing observations collected during several years was available. Around 4.5% of observations were missing and the number of observations was rather small: For each of the 60 sites in the network there were only 21 observations of each

Table 4.1: Running times (in seconds) of the fetch length algorithms. The first two columns give the numbers of vertices and study points. IVTree, VS and RS 10 m refer to the interval tree method , vector based sweep and raster based sweep (cell size 10 m) algorithm. GPU and GPU2 are similar GPU implementations of the cell-based algorithm, running on different machines. GPU3 uses both sparse rasterization and sorting for increased performance. The fetch lengths were computed in 48 directions, except for lines marked by * (2 directions). The study points were placed randomly, except for the case marked by (G) (regular grid). (P1, P2, P3, P4)

		Computer C1			C2	C3	
$\|V\|$	$\|S\|$	IVTree	VS	RS 10m	GPU	GPU2	GPU3
53301	10000	10.2	13.1	11.8	0.19	-	-
213204	200000	212	76.5	54.5	-	-	-
3411264	50000*	31.7	44.3	35.1	-	-	-
3411264	500000*	654.1	62.4	52.5	-	-	-
53301	5000000	-	-	-	8.52	1.16	0.16
3411264	5000000	-	-	-	17.8	4.10	0.43
3411264	$100 \cdot 10^6$	-	-	-	-	81.4	4.75
3411264	$100 \cdot 10^6$ (G)	-	-	-	-	14.7	4.77

of the four water-quality parameters. Some observations were also clearly erroneous.

The erroneous observations were found by comparing each observation to the mean value of the same site for the same quantity. If the difference was large compared to the standard deviation of the quantity in question, the observation was removed. Then, missing values were replaced with estimated values, i.e., imputed. For the imputation two advanced imputation methods and three simple methods were evaluated using a dataset that was obtained by removing observations randomly from the original dataset. The simple methods replaced a missing value by the mean value of the same quantity. As a refinement site-specific mean values were used instead of the global mean. Also year-specific means were tested but no improvement to site means was obtained. The advanced methods (BPCA and Amelia II) were superior to the simple methods. However, in the actual dataset typically all observations of a particular site and time were missing together. Additional tests were done with this pattern of missing values. Then BPCA was inferior to site mean imputation, and Amelia II and site mean imputation had a similar accuracy. The advantage of Amelia II is that it can generate several different estimates for the missing values, allowing one to observe the uncertainty caused by the imputation. Therefore the imputation was done using Amelia II, generating five imputed datasets.

Site pruning was then done using the algorithms described in Section 3.3. 10, 20 and 30 sites were removed from the network. The number of neighbors in the statistical model was 3 or 5, and the pruning was done separately using each of the 5 different imputed datasets. In terms of total estimation error, the local pruning heuristic with taboo list performed worst, followed by the same heuristic without the taboo list, and the pruning with total costs performed best. However, the differences in heuristics, number of neighbours and the different imputed datasets led to differences in the pruned network. The pruned networks were most dissimilar when removing a small number of sites, whereas with 30 sites removed the final networks were somewhat similar to each other. This indicates a lack of robustness in the results, partially caused by the small amount of data available.

While the results were not robust with respect to which sites were removed, the situation was better when considering the total estimation error. That is, if the site pruning was done with a given set of parameters (type of heuristic, number of neighbours and imputed dataset) and the total estimation error was computed using another imputed dataset, the network was still better than a random selection of sites. In total, 15000 randomly selected networks were compared with those produced by the heuristics.

The running time of the heuristics was rather long, perhaps as a result of being implemented using the statistical software R instead of a general purpose programming language. In total about one week of time was required for all the experiments, including the time required for manual work. However, given the sensitivity of the results to the imputation of missing data, performance tuning or further refinements of site pruning heuristics were not considered.

The developed site pruning heuristics can be seen as a special case of the delete-only method (see [21]) with suitably chosen cost functions. In hindsight, investigating the effects of missing values and their imputation was perhaps the most important result of the work. The pruning results were somewhat sensitive to the small differences between datasets caused by different imputations. This sends a clear message that one should not simply ignore the problem of incomplete data even when only a small subset of observations are missing, around 5% in this case.

Chapter 5

Conclusions

From a computer science perspective, the problem of determining fetch lengths (RG1) has been solved for practical purposes. Although the first presented algorithm based on interval trees is no better than the brute-force algorithm in the worst case, with realistic data it was still fast enough to be applied for large datasets. The vector-based sweep line algorithm improved the performance by an order of magnitude when compared to the interval tree method using semi-artificial data. The sweep line algorithm also improves on the worst case time complexity. For determining a fetch length the algorithm only needs to examine a path from the root of a balanced binary search tree to one of its leaves. For a single study point, this gives a reduction of worst-case time complexity from $O(n)$ to $O(\log n)$ when compared to the same operation in the interval tree method. A small improvement to the implemented version is possible, though, as described earlier, by sorting only the vertex points of the islands instead of all study points and end points. The raster-based sweep line method was slightly faster than the vector-based version when using a reasonably small cell size. However, a shortcoming of the method is that study points that are close to a shoreline may not be classified correctly as being interior or exterior points, because rasterization loses precision. The vector-based algorithm is therefore more useful in practice.

The above three algorithms (interval tree method and vector- and raster-based sweep algorithms) were implemented on a computer with a single-core processor. No attempt was therefore made to parallelize them. The interval tree method could easily perform the different fetch length queries in parallel. However, building the interval tree might be more difficult to do in parallel and it requires a significant amount of time. Parallel implementations for the sweep line algorithms may also be difficult to realize, since the algorithm is dealing with data structures that are updated every time a new end point is processed. A small number of cores could easily be used by performing the fetch length computations for the different directions at the same time.

An algorithm that can be more easily adapted for parallel processing was published by other researchers [5]. Although their implementation was serial, it was clear that the fetch lengths for different study points could easily be done in parallel when using the cell-based approach. The preprocessing time of the algorithm is also so low that it is not necessary to parallelize that step, although it could be done if needed. The cell-based algorithm is also faster in practice than the interval tree or sweep line algorithms. However, in the worst possible case the cell-based algorithm is even slower than the brute-force algorithm.

When the parallel implementation of the cell based algorithm is done for a GPU instead of a multi-core CPU, new challenges arise. The algorithm is memory-intensive[1] and the amount of work done for different study points can vary greatly. The first point means that a GPU is unlikely to achieve its peak performance, while the latter one means that the handling of divergent execution paths on a GPU can further reduce performance.

Nevertheless, the algorithm was first implemented for a GPU with only minor low-level modifications. This already yielded good performance, but the introduction of a new rasterization algorithm (sparse rasterization) and simple presorting further improved the execution speed of the algorithm. Using a modern computer, more than 1000 million fetch lengths per second were computed with a map containing 3.4 million vertices.

Further research on the subject should likely no longer focus on fetch lengths but on modeling the waves in a water area in a more realistic manner, taking into account also the depth of the water. This allows incorporating natural phenomena such as the diffraction of waves. It is noteworthy, though, that some further work based on fetch lengths has been published [36]. As a possible improvement to fetch lengths using depth information to modify fetch lengths was proposed. In addition, the sensitivity of model results to parameters such as the number of fetch line directions was studied, with the observation that the models are sensitive to such parameters and further work should be done in order to be able to choose them in a rational manner.

The water quality monitoring network problem was considered in the last article covered in this thesis. While the method was practical, achieving the research goal RG2, there were several subject areas for further research. From a computer science perspective the site selection algorithm is the most obvious candidate for improvement. The error measure used in the work could also be reconsidered. In particular, one might consider what kind of analyses are actually done using the water quality data. For instance, if the water quality is interpolated for points that do not coincide with the measurement points, it would be sensible to perform the same kind of inter-

[1] Memory intensive means here that the amount of arithmetic operations is small compared to the amount of memory accesses.

polation using a reduced set of observational sites, removing the sites that least affect the result of the interpolation. Dealing with missing and incorrect values could possibly also benefit from a more application-specific approach.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest. Introduction to algorithms, The MIT Press, USA, 20th printing, 1998.

[2] Jan Ekebom, Pasi Laihonen, Tapio Suominen. A GIS-based step-wise procedure for assessing physical exposure in fragmented archipelagos. Estuarine, Coastal and Shelf Science, 57(5-6), pp. 887-898, 2003.

[3] Harri Tolvanen, Tapio Suominen. Quantification of openness and wave activity in archipelago environments. Estuarine, Coastal and Shelf Science, 64 (2–3), 2005, pp. 436-446.

[4] Jason Rohweder, James T. Rogala, Barry L. Johnson, Dennis Anderson, Steve Clark, Ferris Chamberlin, Kip Runyon. Application of Wind Fetch and Wave Models for Habitat Rehabilitation and Enhancement Projects. `pubs.usgs.gov/of/2008/1200/pdf/ofr2008-1200_web.pdf`, 2008. Accessed 20 May 2015.

[5] S. Yang, J. H. Yong, J. G. Sun, H. J. Gu, J. C. Paul, A cell-based algorithm for evaluating directional distances in GIS. International Journal of Geographical Information Science, 24(4), pp. 577-590, 2010.

[6] Hendrik L. Tolman. A Third-Generation Model for Wind Waves on Slowly Varying, Unsteady, and Inhomogeneous Depths and Currents. Journal of Physical Oceanography, 21, pp. 782-797, 1991.

[7] Michel Benoit, Frederic Marcos, Francoise Becq. Development of a third generation shallow-water wave model with unstructured spatial meshing. Coastal Engineering Proceedings, 1996, no. 25, pp. 465-478.

[8] N. Booij, R.C. Ris, L.H. Holthuijsen. A third-generation wave model for coastal regions: 1. Model description and validation. Journal of geophysical research, vol. 104, no. C4, pp. 7649-7666, 1999.

[9] Ole R. Sørensen, Henrik Kofoed-Hansen, Morten Rugbjerg, Lars S. Sørensen. A third-generation spectral wave model using an unstructured finite volume technique. Coastal Engineering 2004, Proceedings of the 29th International Conference, chapter 71, pp. 894-906.

[10] Jinhai Zheng, Hajime Mase, Zeki Demirbilek, Lihwa Lin. Implementation and evaluation of alternative wave breaking formulas in a coastal spectral wave model. Ocean Engineering, 35(11-12), pp. 1090-1101, 2008.

[11] D. Lundqvist, D. Jansen, T. Balstroem, C. Christiansen. A GIS-Based Method to Determine Maximum Fetch Applied to the North Sea-Baltic Sea Transition. Journal of Coastal Research, vol. 22, no. 3, 2006, pp. 640-644.

[12] Michael Ian Shamos, Dan Hoey. Proc. 17th Annu. IEEE Symp. Foundations of Computer Science, pp.208-215, 1976.

[13] J.L. Bentley, T.A. Ottmann, Algorithms for Reporting and Counting Geometric Intersections. IEEE Transactions on Computers, vol.28, no. 9, pp. 643-647, September 1979, doi:10.1109/TC.1979.1675432.

[14] D.S. Andrews , J. Snoeyink , J. Boritz , T. Chan , G. Denham , J. Harrison , C. Zhu. Further Comparison of Algorithms for Geometric Intersection Problems. Proc. 6th Int'l. Symp. on Spatial Data Handling, 1994.

[15] HELCOM, 2014. Eutrophication status of the Baltic Sea 2007-2011 - A concise thematic assessment. Baltic Sea Environment Proceedings No. 143. Available online at `http://www.helcom.fi/Lists/Publications/BSEP143.pdf`. Accessed 1.6.2015.

[16] R.J.A. Little, D.B. Rubin. Statistical Analysis with Missing Data, 2nd ed., John Wiley, Hoboken, NJ, 2002.

[17] Gary King, James Honaker, Anne Joseph, Kenneth Scheve. Analyzing Incomplete Political Science Data: An Alternative Algorithm for Multiple Imputation. American Political Science Association, pp. 49-69, 2001.

[18] Peter Schmitt, Jonas Mandel, Mickael Guedj. A Comparison of Six Methods for Missing Data Imputation. J Biom Biostat 6:224, 2015. doi: 10.4172/2155-6180.1000224.

[19] J. Honaker, G. King, M. Blackwell. Amelia II: A Program for Missing Data. Journal of Statistical Software, vol. 45, issue 7, 2011.

[20] Nicholas J. Horton, Stuart R. Lipsitz. Multiple Imputation in Practice: Comparison of Software Packages for Regression Models With Missing Variables. The American Statistician, 55(3), pp. 244-254, 2001.

[21] Sergey Frolov, António Baptista, Michael Wilkin. Optimizing fixed observational assets in a coastal observatory. Continental Shelf Research, 28(19), 2008, pp. 2644-2658.

[22] Sergios Theodoridis, Konstantinos Koutroumbas. Pattern recognition. Academic Press, USA, 1999.

[23] Pengfei Lin, Rubao Ji, Cabell S. Davis, Dennis J. McGillicuddy Jr. Optimizing plankton survey strategies using Observing System Simulation Experiments. Journal of Marine Systems, 82(4), 2010, pp. 187-194.

[24] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf. Computational Geometry Algorithms and Applications, 2nd edition, Springer, Germany, 2000.

[25] H Edelsbrunner, L J Guibas. Topologically sweeping an arrangement. STOC '86 Proceedings of the eighteenth annual ACM symposium on Theory of computing, pp. 389-403, 1986. doi: 10.1145/12130.12171.

[26] Stefan Hertel, Martti Mäntylä, Kurt Mehlhorn, Jurg Nievergelt. Space sweep solves intersection of convex polyhedra. Acta Informatica, 21(5), pp. 501-519, 1984.

[27] Efthymios G. Anagnostou, Leonidas J. Guibas, Vassilios G. Polimenis. Topological sweeping in three dimensions. Volume 450 of the series Lecture Notes in Computer Science, pp. 310-317, 2005.

[28] J. Boissonnat, F.P. Preparata, Robust Plane Sweep for Intersecting Segments. SIAM Journal on Computing, vol. 29, issue 5, pp. 1401-1421, 2000.

[29] V. Akman, W. R. Franklin, M. Kankanhalli, C. Narayanaswmi. Geometric computing and uniform grid technique. Computer-Aided Design, 21(7), pp. 410-420, 1989.

[30] AMD, AMD Accelerated Parallel Processing OpenCL™Programming Guide, 2013. Available at `http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf`

[31] B.R. Gaster, L. Howes, D.R. Kaeli, P. Mistry, D. Schaa, Heterogeneous Computing with OpenCL. Revised OpenCL 1.2 edition. Morgan-Kaufmann, USA, 2013.

[32] Khronos OpenCL Working Group, Editor Aaftab Munshi, The OpenCL Specification, Version: 1.2, Document Revision: 19. 2012. Available at `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`

[33] AMD, Reference Guide: Southern Islands Series Instruction Set Architecture, 2012. Available at `http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf`

[34] J.G. Cleary, G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. The Visual Computer 1988; 4 (2), 65-83.

[35] K.V. Mardia, J.T. Kent, J.M. Bibby, 1979. Multivariate Analysis. Academic Press, London.

[36] Austen Pepper, Marjetta L. Puotinen. GREMO: A GIS-based generic model for estimating relative wave exposure. The 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, pp. 1964-1970, 2009.

Part II

Publication reprints

Publication I

Mika Murtojärvi, Tapio Suominen, Harri Tolvanen, Ville Leppänen, Olli S. Nevalainen. Quantifying distances from points to polygons—applications in determining fetch in coastal environments. *Computers & Geosciences* 33 (7), 2007, 843–852. DOI: 10.1016/j.cageo.2006.10.006. Reprinted with permission from Elsevier.

Computers & Geosciences 33 (2007) 843–852

COMPUTERS & GEOSCIENCES

www.elsevier.com/locate/cageo

Quantifying distances from points to polygons—applications in determining fetch in coastal environments

Mika Murtojärvi[a,*], Tapio Suominen[b], Harri Tolvanen[b], Ville Leppänen[a], Olli S. Nevalainen[a]

[a]*Department of Information Technology and Turku Centre for Computer Science (TUCS), University of Turku, FI-20014, Finland*
[b]*Department of Geography, University of Turku, FI-20014, Finland*

Received 14 October 2005; received in revised form 6 October 2006; accepted 7 October 2006

Abstract

Distance from a point to adjacent borderlines is a variable that has many applications in environmental research. Geographical information systems (GIS) include tools for measuring such distances, but these tools are inefficient if there are multiple, i.e. millions of distances to be calculated. In this paper we propose an efficient algorithm which calculates the distances in multiple predetermined directions from a large number of points to polygon borders.

The problem is significantly simplified by the fact that the distances are calculated in some directions, only. An interval tree is utilized for efficiently retrieving those line segments describing the coastal lines and the borders of the islands that are relevant in determining these distances. The algorithm is also robust so that it gives meaningful results in the presence of rounding errors regardless of the positions of the study points with respect to the polygon borders. In coastal environments the straight-line distance from a point to the nearest shoreline over an open water surface is referred to as fetch length. The fetch lengths in multiple directions indicate general openness around a studied point and it may also be used as a variable in wave power calculations. An implementation of the algorithm was used for calculating fetch data for the archipelago of SW-Finnish coast in the Baltic Sea. The map data contained 3 million vertices and fetch lengths were calculated for 2.5 million points in 48 directions. The algorithm enabled determining fetch lengths in the complex archipelago environment quickly in high spatial accuracy and it may have applications also in other geographical research and image processing.

© 2007 Elsevier Ltd. All rights reserved.

Keywords: Efficient algorithms; GIS; Fetch length; Exposure; Computational geometry

1. Introduction

Borders and transitional zones in the landscape are significant biogeographical environments. Human influence causes sharp boundaries, such as roads and edges of cultivated areas. In natural environments abiotic factors like soil characteristics may change over a short distance and the biotic environment usually reflects the change. The physical or biological environment may apparently change sharply, but the neighboring environments influence each other to a certain degree. Distance and direction to borderlines are hence important parameters in environmental sciences.

*Corresponding author. Tel.: +358 2 333 8658; fax: +358 2 333 8600.
E-mail address: mika.murtojarvi@it.utu.fi (M. Murtojärvi).

0098-3004/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2006.10.006

One of the most fundamental borders in nature is the littoral zone between terrestrial and aquatic environments. On the shores both abiotic and biotic environments are dynamic. Wave activity is a principal force affecting the littoral zone. Many of the methods for estimating wave effects require the measurement of fetch length, i.e. the straight-line distance in the wind direction where the friction between moving air and the water surface interact to create waves. Thus, the fetch is the distance from a studied point to the nearest shoreline in a given direction (Ekebom et al., 2003; Håkansson, 1981; Tolvanen and Suominen, 2005).

A simple method to quantify general openness of a selected point is to calculate the average of the distances to nearest land in directions of equal and predetermined intervals. A commonly applied method is the *effective fetch* (F_{eff}), in which the distances from a point to shore are weighted according to the cosine of the angle between the line direction and central radius, i.e. line which is set at right-angle to the shoreline or to the direction which yields the highest F_{eff} value (Anon, 1977).

Geographical information systems (GIS) offer cartographical tools for such calculations (Ekebom et al., 2003; Tolvanen and Suominen, 2005). The essential procedure in fetch length measurement is the same as determining intersections of lines. However, the tools in GIS software packages do not include optimized vector calculation algorithms that would suit the fetch quantification in large volumes. The operations in GIS would include vast data sets and the vector overlay operations in very large batches do not work efficiently.

We present an efficient and easily applicable algorithm for determining distances from points to polygons in large quantity. The algorithm has been applied in coastal studies but it may have applications in other geographically oriented sciences as well. This research originates from a practical project where it was necessary to produce spatially detailed fetch data in the fragmented archipelago environment of SW Finland. The work required efficient calculation of fetch lengths for multiple study points in 48 fetch line directions. Due to these preconditions both the number of fetch lines and the number of island shoreline segments, i.e. the detailed island polygons of 1:20 000 map data, were considerable. The produced numerical data should be feasible to be used as an estimation of exposure as such or as base data for more sophisticated calculations, such as effective fetch and wave power.

The problem of finding line segments that intersect a given line was transformed into the problem of finding all intervals that contain a given real number. This was done by rotating the map coordinate system so that the studied line is parallel to the x-axis. Checking whether the half line (i.e. that part of the study line which emanates from the study point to the direction of the positive x-axis) also intersects the line segments can be done by comparing the x-coordinates of the intersection point and the starting point of the half line. Based on these observations, an algorithm for determining the fetch lengths for a set of points is presented. The efficiency of a Java based implementation of the algorithm was tested using real world and semi-artificial input data. The performance was found high enough for making fetch length based wave exposure estimations for large areas. In addition to the implementation of the actual algorithm, a simple software application[1] that has a graphical user interface and uses MySQL database for I/O was produced.

2. Definition of fetch length

We assume that the shoreline data are available in such a format that an island i is represented as an ordered sequence of two-dimensional points $(p_{i1}, p_{i2}, \ldots, p_{im_i})$, where the points form a simple polygon approximating the shoreline of the island when connected in the specified order and the point p_{i1} is both the first and the last point of the polygon. The outline of the island thus consists of line segments $v_{i1} = \overline{p_{i1}p_{i2}}, v_{i2} = \overline{p_{i2}p_{i3}}, \ldots, v_{im_i} = \overline{p_{im_i}p_{i1}}$. As we need neither the information that the line segment v_{ij} belongs to island i nor the order of these line segments within the islands, we can simply consider the map data to consist of a set of line segments $V = \{v_1, v_2, \ldots, v_n\}$. As input data we also have a set of points $P = \{p_1, p_2, \ldots, p_r\}$ and a set of angles $\Theta = \{\theta_1, \theta_2, \ldots, \theta_s\}$. The fetch lengths are to be calculated for all points $p \in P$ for all angles $\theta \in \Theta$. There are no restrictions on the locations of the points, so we must be careful about special cases such as points that lie on a line segment.

Let us denote by $d_\theta(p, v)$ the distance from a point p to the intersection of a line segment v and the half

[1]Murtojärvi, 2005. Software available by email: mianmu@utu.fi.

line starting from p and running in direction θ relative to the direction of the positive x-axis. If there is no such intersection, it is defined that $d_\theta(p, v) = \infty$. In the definition of fetch length different cases have to be separated. These deal with points that are not inside any polygon, inside a polygon or on the boundary of a polygon. If the point is on the boundary of a polygon it still has to be observed whether the half line is directed towards the interior or towards the exterior of the polygon. This gives us a formal definition of *fetch length*:

$$L(p,\theta) = \begin{cases} \min\{d_\theta(p,v) | v \in V\} & \text{if } p \text{ is not inside any polygon,} \\ 0 & \text{if } p \text{ is inside a polygon,} \\ 0 & \text{if } p \text{ is on the boundary of a polygon and its immediate vicinity on the half line under study is inside a polygon,} \\ \min\{d_\theta(p,v) | v \in V \text{ and } d_\theta(p,v) > 0\} & \text{otherwise.} \end{cases}$$

In the last alternative (p is on the boundary of a polygon and the half line points outwards from the polygon), the condition $d_\theta(p, v) > 0$ is required because $d_\theta(p, v) = 0$ holds for a line segment that contains the point p. Note that the cases of the definition where the point p is on the boundary of a polygon actually also cover the cases where the point p is either outside or inside a polygon (Fig. 1).

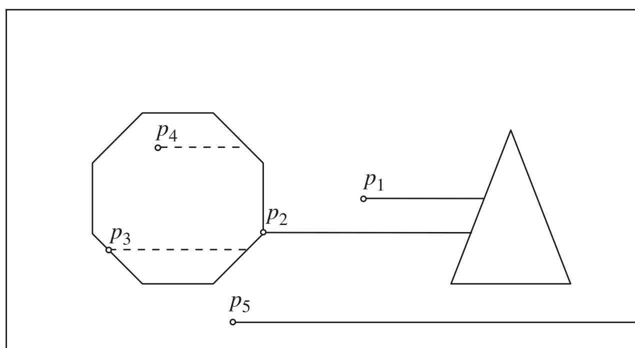

Fig. 1. Fetch lengths for five points in horizontal direction $\theta = 0$ (i.e. from left to right). Horizontal lines represent distances to nearest intersections of half lines and line segments of the map. Dashed lines indicate zero fetch lengths, solid lines have lengths that correspond to fetch lengths. Points p_2 and p_3 are boundary points. Fetch length for p_3 is 0 due to inward direction of its half line. Point p_1 is an exterior point and p_4 an interior point. Half line originating from p_5 does not intersect any line segments, and fetch length for p_5 in direction $\theta = 0$ is ∞.

3. Determining fetch lengths

3.1. Starting point for the calculations

The fetch lengths may be calculated for an individual study point or for example multiple points located equidistantly to create an exposure classification for a stretch of shoreline. Sometimes spatially continuous surfaces in raster format are needed in order to estimate the wave climate or spatial distribution of exposure. The number of required fetch values may then be several millions. This practically rules out using manual methods for determining fetch. The complexity of shoreline data also varies according to the coastal environments and map scale. Efficiency and memory consumption of an algorithm for determining fetch lengths are therefore important.

It was assumed that the fetch lengths need to be calculated in predetermined directions for a very large number of study points. Therefore it is acceptable that an algorithm for determining fetch lengths performs preprocessing for speeding up its operation. An obvious method for solving this problem without preprocessing is to calculate for the study point p_s the distances $d_\theta(p_s, v)$ for all line segments v of the island polygons, and select the shortest distance. Typically, the majority of the shoreline segments do not intersect the given fetch line and most of the time is spent considering distances that are infinite by definition. If most or all of the line segments that do not intersect the fetch line are eliminated from consideration, the saving in time may be considerable.

We first consider the determination of fetch length in the special case where the fetch lines under consideration are horizontal, more specifically $\theta = 0$. The determination of fetch lengths in other directions can easily be transformed into this special case by rotating the coordinate system. The determination consists of two basic steps, checking whether a given point is an interior or an exterior point and finding the minimum among the candidate distances.

3.2. Point in polygon check

The first part of the problem, determining whether a given point is inside or outside a polygon, is a well-known problem and an algorithm that solves this problem is discussed by Preparata and Shamos (1988). The algorithm is based on the observation that, except for some special cases, the number of intersections between a horizontal half line whose endpoint is p_s and the line segments of the polygon is even if and only if p_s is outside the polygon. As special cases one must consider line segments whose end point lies on the half line, because each end point belongs to two line segments, in contrast to other points of the line segments which belong to only one line segment.

In the aforementioned algorithm, line segments whose lower end point lies on the half line are not considered when counting the number of intersections. The algorithm also determines correctly whether the given point is inside some polygon when the input data consist of the line segments of multiple non-intersecting polygons. In the problem at hand the only difference is that a point on the boundary of a polygon is considered to be inside the polygon (i.e. the fetch length is zero) if and only if its immediate vicinity in the direction θ ($\theta = 0$ in the rotated coordinate system) is inside the polygon. The only needed modification to the algorithm is that also line segments that contain the point p_s are ignored when counting the number of intersections. Since the operation of detecting whether a point lies on a line segment is done with finite precision, one must allow that p_s is some small distance away from a line segment and still consider that p_s is on the line segment.

Without preprocessing, the algorithm for determining whether a point is inside a polygon runs in time $O(n)$, where n is the number of line segments (or, equivalently, the total number of vertex points of all island polygons) in the map data. This time can be reduced considerably with some preprocessing that is also used for reducing the number of line segments to which the distances $d_\theta(p_s, v)$ are calculated (see Section 3.3).

3.3. Closest intersection point

Another part of the problem is determining the shortest distance between the study point $p_s = (x_s, y_s)$ and an intersection of the horizontal half line with a line segment in the set V, i.e. $\min\{d_\theta(p_s, v) | v \in V\}$, where $\theta = 0$. A horizontal line with a given y-coordinate y_s intersects the line segments whose lower and upper endpoints $p_l = (x_l, y_l)$ and $p_u = (x_u, y_u)$ fulfill the condition $y_l \leqslant y_s \leqslant y_u$. This is the same test one would perform if one were to check whether the interval $[y_s, y_s]$ overlaps the vertical interval $[y_l, y_u]$. Thus, a known algorithm for finding intervals that overlap a given interval can also be used for finding all line segments that intersect a given horizontal line.

An example of such an efficient algorithm and a suitable data structure, called the interval tree, can be found in Cormen et al. (2001). The data structure is basically a balanced binary search tree whose nodes contain the desired intervals and, in addition, each node contains the largest value (highest end point) contained in any of the intervals stored in the subtree rooted at that node. The intervals are ordered by their lower end points. Such a tree can be built in $O(n \log n)$ time and all intervals that overlap a given interval can be found in $O(\min(k \log n, n))$ time, where n is the number of intervals in the tree and k is the number of intervals in the tree that overlap the given interval.

With real-world map data the number of line segments that intersect a horizontal line (i.e. k) can be expected to be quite small compared to the total number of line segments n in the map data. Such a situation along with a situation where the use of an interval tree is of little value is shown in Fig. 2. The latter situation occurs rarely in practice, and the use of an interval tree is expected to lead to good performance. It should be noted that algorithms that can find the overlapping intervals in $O(k + \log(n))$ time are known (McCreight, 1985; Preparata

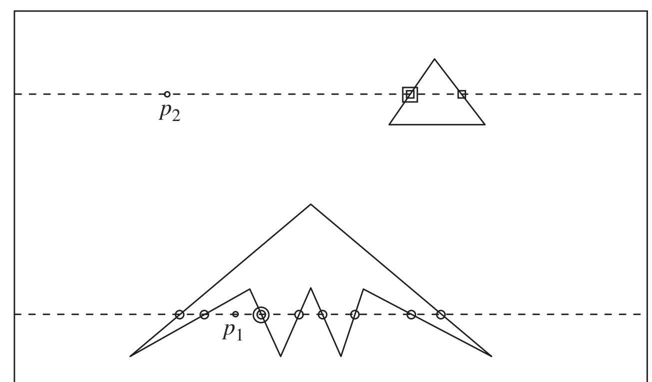

Fig. 2. Finding closest intersections. Line segments marked with a circle (square) are taken into account when determining fetch length for point p_1 (p_2). Intersection points marked with a double circle (double square) are closest intersection points for p_1 (p_2).

and Shamos, 1988). For our purposes the data structure and algorithms given in Cormen et al. (2001) are sufficiently efficient.

For calculating the coordinates of the intersection point $\tilde{p} = (\tilde{x}, \tilde{y})$ of a line segment v and the horizontal line passing through the study point $p_s = (x_s, y_s)$, it is also necessary to store additional information, e.g. the x-coordinates of the two end points of the line segments with each interval. For the line segment $v = \overline{p_l p_u}$ $(p_l = (x_l, y_l), p_u = (x_u, y_u))$ that is not horizontal, the coordinates $(\tilde{x}, \tilde{y})$ of the intersection point $\tilde{p}$ are then simply

$$\begin{cases} \tilde{x} = x_l + \dfrac{(\tilde{y} - y_l)}{y_u - y_l} \cdot (x_u - x_l), \\ \tilde{y} = y_s. \end{cases} \tag{1}$$

The distance is $d_0(p_s, v) = \tilde{x} - x_s$ if this expression is positive. Otherwise the intersection point is not on the half line whose direction is $\theta = 0$ but in the direction $\theta = \pi$.

Because the method for removing irrelevant line segments from consideration returns a result set that contains all the line segments that intersect the horizontal half line with starting point p_s, it can also be used for speeding up the point-in-polygon testing by using the observation mentioned before: if an intersection point satisfies the condition $\tilde{x} - x_s > 0$, the intersection is located in the direction $\theta = 0$. Counting the number of such line segments, except for those whose lower end point lies on the half line, yields the same number of intersections as the method described in Section 3.2. Determining whether p_s is an inner point of some polygon therefore has the same time complexity as determining the intersection points (i.e. $O(n \log n)$ preprocessing time and $O(\min(k \log n, n))$ query time).

For half lines with inclination $\theta \neq 0$ we rotate the coordinates using the well-known equations (Arfken and Weber, 1995):

$$\begin{cases} x' = x\cos(\theta) + y\sin(\theta), \\ y' = -x\sin(\theta) + y\cos(\theta). \end{cases} \tag{2}$$

4. Algorithm for determining fetch lengths

Algorithm fetch_length first rotates the coordinates of all points by θ radians. An interval tree is then constructed for restricting the distance calculations to relevant line segments. The intersection points along with their number are determined for each point using the tree, see Fig. 2. The number of intersections is used for determining whether the studied point is an interior or an exterior point. The fetch length is then set to $\min\{d_\theta(p_s, v) | v \in V\}$ or 0 depending on whether the number of intersections is even or odd.

It is easy to modify algorithm fetch_length so that the angles θ and $\theta + \pi$ are handled simultaneously. In this case, one must record the minimum distances to line segments and the number of intersections in both directions. Among other important modifications are those that increase the robustness of the algorithm, see Sections 5.1 and 6.

Algorithm fetch_length. An algorithm for determining fetch lengths in direction θ for all points in the set P. The function 'rotate' rotates the given coordinates according to formula (2) and the function 'calculate_x' calculates the x-coordinate of the intersection of a line segment and a horizontal line according to formula (1), using the coordinates of the given points in the rotated coordinate system. The nodes of the interval tree contain the end points of the real-valued intervals and also the end points of the line segment whose projection the interval is. These end points are retrieved using the function get_endpoints. Function 'get_overlapping_segments' returns all intervals stored in the interval tree whose intersection with the given interval is non-empty. It is assumed that each study point p_s in P has a data structure for storing the results $L(p_s, \theta)$.

Procedure fetch_length(P: set of points, V: set of line segments, θ: real)
ivtree ← an empty interval tree
for each point p_s in P {
  $(x'_s, y'_s) \leftarrow$ rotate(x_s, y_s, θ)
}
for each line segment $v = \overline{p_1 p_2}$ in V{
  $(x'_1, y'_1) \leftarrow$ rotate(x_1, y_1, θ)
  $(x'_2, y'_2) \leftarrow$ rotate(x_2, y_2, θ)
  $y_{min} \leftarrow \min\{y'_1, y'_2\}$
  $y_{max} \leftarrow \max\{y'_1, y'_2\}$
  ivtree.add($[y_{min}, y_{max}]$, p_1, p_2)
}
for each point p_s in P {
  $S \leftarrow$ ivtree.get_overlapping_segments ($[y'_s, y'_s]$)
  mindist $\leftarrow \infty$
  num_intersections $\leftarrow 0$
  for each interval I in S{

```
    (p1,p2) ← I.get_endpoints
    x̃' ← calculate_x(p1,p2,y's)
  if x̃' − x's > 0 then {
    if x̃' − x's < mindist then mindist ← x̃' − x's
    if y's ≠ I.get y_min then num_intersections ←
      num_intersections +1
    }
  }
  if num_intersections is even then
    L(ps,θ) ← mindist
  else L(ps,θ) ← 0
}
```

5. Implementation of the algorithm

5.1. Specifications

In order to test the efficiency of algorithm fetch_length and demonstrate its usability, two programs were created using Java programming language. The first program was used for testing the performance of the algorithm, and it only includes an implementation of the algorithm and some I/O functionality that is required for reading the map data. Another program was created for a case study and this version includes a basic user interface and uses MySQL database for accessing the data. Both programs share the same implementation of the algorithm.

It should be noted that our implementation differed slightly from algorithm fetch_length. One difference is that the distances were calculated simultaneously for angles θ and $\theta+\pi$ as outlined before. This approximately doubles the efficiency of the algorithm. Another difference was that the condition $\tilde{x}'-x'_s>0$ was not considered sufficient for determining that the point p_s is not on a line segment I because of the possibility of rounding errors. A function was therefore included for determining whether a point is close enough to a line segment so that it should be considered to lie on the line segment. This test consists of two phases. The first phase tests whether the point is inside a bounding box whose sides are parallel to the coordinate axes. The box includes the line segment and some small tolerance ε of extra space in all directions. If the point lies outside this box, the point is farther than ε units away from the segment. Otherwise its distance to the line segment is calculated and compared to ε. If the point passes both of these tests, it is considered to lie on the line segment.

5.2. Performance test

The efficiency of the algorithm was tested using a PC equipped with 1 GB RAM and AMD Athlon 2600 + processor (1.92 GHz). The Java environment used for compiling and executing the program was Sun's J2SE version 5.0. As input data we had a shoreline map of a portion of the Finnish archipelago. The map consisted of 1277 polygons which contained a total of 53 300 vertices. For testing the efficiency of the algorithm with a larger number of polygons and points, semi-artificial map data were created by tiling several copies of the map next to each other.

The running times of our implementation in some cases are given in Table 1. The performance of the algorithm is found to be adequate for the purposes for which it is intended. Constructing the interval

Table 1
Running times for an implementation of algorithm fetch_length

Case	Points in P	Points in V	Angles	Tree generation time (s)	Total time (s)
1	50000	53301	48	4.7	30.5
2	50000	213204	48	18.1	72.2
3	50000	479709	48	44.6	130.1
4	50000	852816	48	84.3	198.8
5	50000	1918 836	48	183.3	373.4
6	50000	3411 264	48	398.4	657
7	50000	6449 421	48	732.9	1103.3
8	10000	5330 100	48	635	697
9	50000	5330 100	48	Not measured	957.4
10	200000	5330 100	48	Not measured	1276.1

"Points in P" represents number of points for which fetch lengths were calculated. "Points in V" represents number of vertices in island polygons. Times required for loading test data and saving results are excluded.

tree requires a significant amount of time, and therefore the algorithm performs best when the fetch lengths are required for a large number of points. The total time in case 8 consists almost entirely of the interval tree generation time, and cases 9 and 10 show how the required time grows if the number of calculation points is increased. From the table one can also observe that while the tree generation time is significant, it is still small enough so that the calculations can be performed efficiently in relatively small batches of calculation points even if the tree is regenerated for each batch.

5.3. Case study

The algorithm was used in a case study to examine the exposure on a 40 km × 40 km area in SW Finland. The study area is characterized by numerous islands and complex shoreline (Frisén et al., 2005). The objective of the study was to test the calculation method and to find a proper way to store, visualize and use the data. Two sets of input data were needed, i.e. the shoreline data and the calculation points. The shoreline data of the Finnish coast are based on polygon map data (1:20 000) produced by the National Land Survey of Finland. The shoreline for other parts of the Baltic Sea was derived from the Europe Countries database (ESRI, 2005). The shoreline data include altogether 3 million vertices. The coordinate pairs were stored in a MySQL database, see Table 2 for sample data expressing a portion of a polygon border.

Two approaches were used when defining the study points. In the first approach, points were positioned on a regular grid on the water surface and in the second case equidistantly along the shoreline. The coordinate pairs of the grid were created by a simple PHP script but they could be created by GIS tools or a spreadsheet program as well. The grid points covered the study area equidistantly 25 m apart and their number was 2.56 million. The calculation points along the shoreline were created using GIS tools. The shoreline points were located equidistantly 10 m apart and their total number was 270 000. Both sets of points were stored in a MySQL database including point ID and cartesian x- and y-coordinates (Table 3). The overall workflow including the operation of algorithm fetch_length is illustrated in Fig. 3.

Table 2
A sample input table for island polygon vertex data

ISL_ID	PID	x	y
600000	0	3373 690.75	7306 959
600000	1	3373 679	7306 962
600000	2	3373 661.5	7306 972.5
600000	3	3373 639.75	7306 979.5
600000	4	3373 604.75	7307 001.5

Each line contains identification numbers and x- and y-coordinates (Finnish coordinate system YKJ) of one vertex point. PID is an identification number of a point within an island and ISL_ID is an identification number of an island. Five consecutive points that belong to one island are shown.

Table 3
Input table for study points

ID	x	y
0	3 223 582.95863	6 719 997.70291
1	3 224 049.02434	6 718 849.19557
2	3 224 039.36884	6 718 851.79776
3	3 224 029.71334	6 718 854.39995
4	3 224 020.05785	6 718 857.00214

The calculations were performed on a desktop computer (Pentium 4 CPU 2.79 GHz, 1.00 GB RAM) in batches of at most 100 000 points. The total calculation time for shoreline points P_2 was approximately 0:39 h and for point grid data P_1 4:55 h. The results can be stored directly into the MySQL database but it was noticed that writing the results to a text file was much faster. The resulting text files were finally imported into a MySQL database. The data include the identification numbers and the fetch lengths in 48 directions for each point. The distances were converted to integers in order to limit the file size. The shoreline fetch database was linked to GIS to be used as a thematic layer with other geographical data sets. We produced a sample map of shoreline exposure with average fetch values over the 48 directions on a small study area in SW Finland, see Fig. 4. Another map was produced to portray the exposure values on the water surface. We grouped the 48 lines by eight compass directions, resulting in six lines for each direction, and calculated the average fetch for each compass sector. The raster map pixel value is the highest such sector average for each cell in the water area, see Fig. 5.

6. Discussion

We presented an efficient and easy way to determine fetch lengths in different predetermined

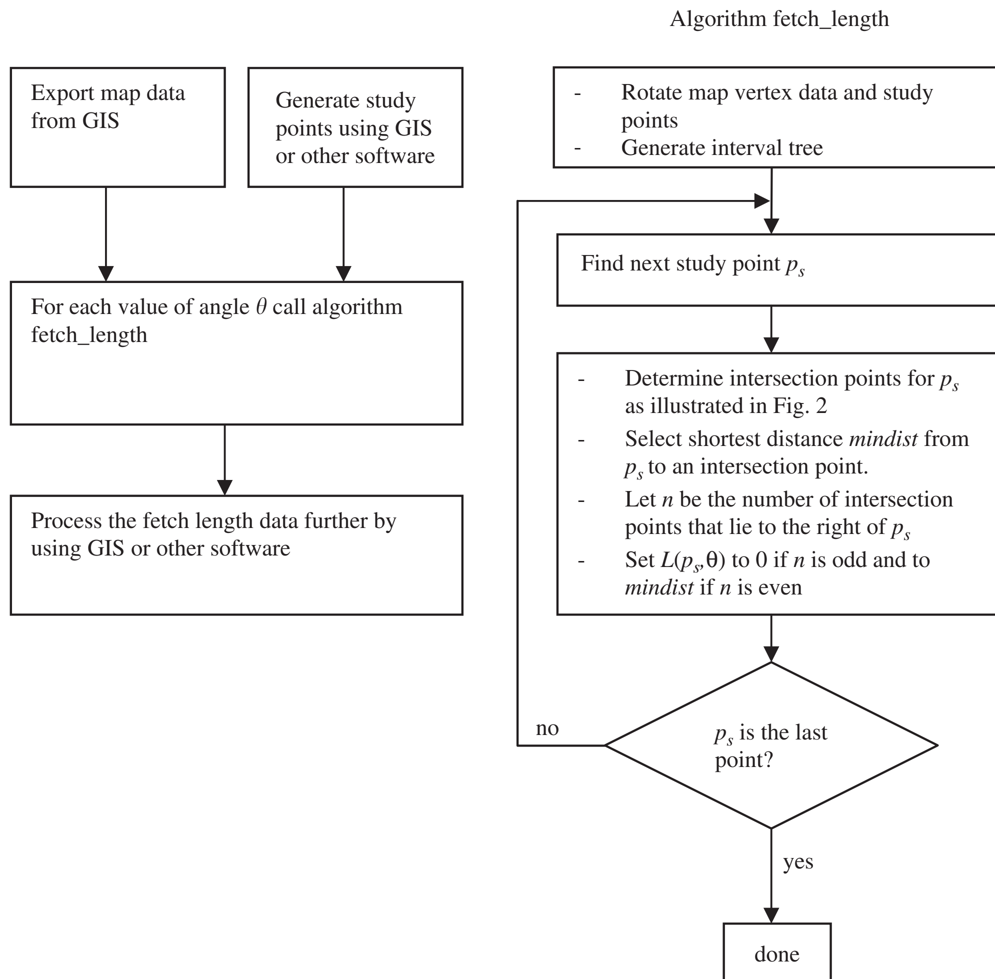

Fig. 3. Workflow in determining and using fetch lengths. Operation of algorithm fetch_length is shown on right.

directions for a large number of study points. The observation leading to our algorithm was the fact that, typically, a fetch line intersects only a small fraction of the shoreline segments in a map. In our test cases, for example, there were approximately 50 intersections for each fetch line in the shoreline data set consisting of 53 000 vertices. However, the tree generation in our algorithm is relatively time-consuming. Calculating fetch lengths for a small number of points could thus be done more efficiently with a more straightforward algorithm that does not require preprocessing.

While the efficiency of our algorithm is good, it is unlikely to be optimal. The algorithm determines the distances to all the line segments of the map that intersect the line passing through the point under consideration, although it would be sufficient to calculate only the shortest such distance. One promising approach for further reducing the number of measured line segments is the sweep line technique (Preparata and Shamos, 1988). In our case study, however, the calculations had to be done in relatively small batches due to the high memory requirements. The performance of an algorithm when the number of calculation points is moderately high was therefore most important.

Also the robustness of an algorithm in the presence of rounding errors was found crucial when determining a quantity that depended on whether the point lies on a line segment. Although algorithm fetch_length and the extension described in Section 5.1 were not totally satisfactory, the situation was easily remedied. We only needed to replace the statement $S \leftarrow$ *ivtree.get_overlapping_segments* $([y'_s, y'_s])$ by a statement that returns all line segments that may be closer than ε units away from p_s, namely $S \leftarrow$ *ivtree.get_ xoverlapping_segments* $([y'_s - \varepsilon, y'_s + \varepsilon])$.

The spatial resolution and level of detail of the resulting data far exceed the capability of standard GIS software. For instance, an average fetch map of our whole study area in SW Finland was previously

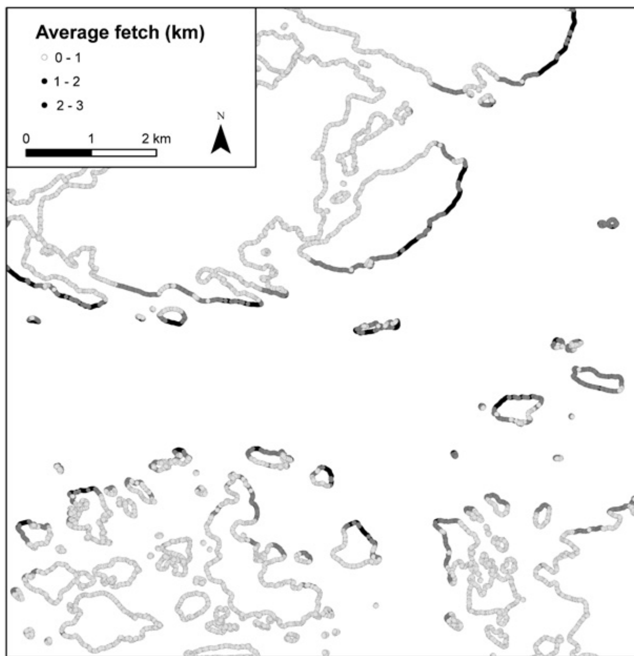

Fig. 4. Average fetch lengths in a sample region measured for 48 directions (out of which 24 have zero length) for points on shoreline.

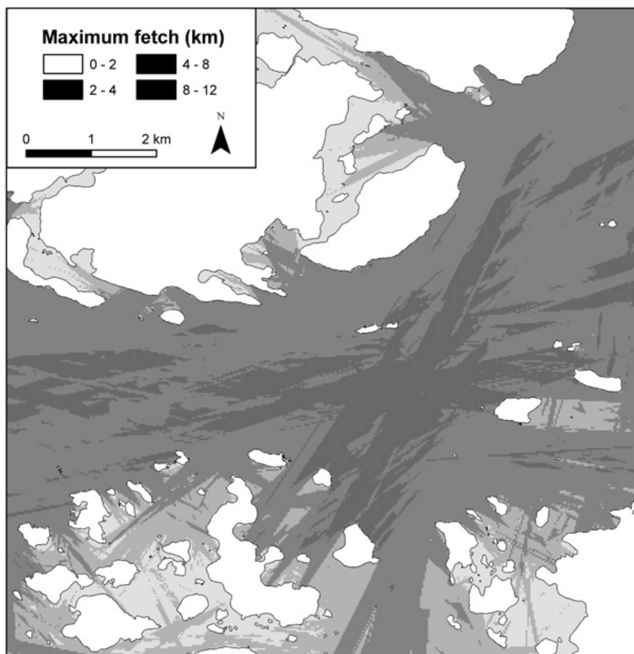

Fig. 5. Highest average fetch lengths (mean of six line lengths) out of eight compass directions for each 25 m × 25 m cell.

produced for a grid with 1000 m cell size using GIS (Tolvanen and Suominen, 2005), while with the present algorithm the spatial resolution was improved to 25 m in a fraction of the time in comparison to the GIS software. The advantages of using a specialized algorithm instead of a standard desktop GIS software are probably related to the less complex input data structure, i.e. no topological data are needed when points of P are expressed by coordinate pairs and shorelines as coordinate pairs of vertices, not as complete polygons. This lowers the memory consumption of the algorithm.

The output data, whether used for a water surface grid or for a shoreline exposure classification, can be utilized in new contexts. The data allow for calculating different exposure parameters, such as averages, maximums or effective fetch. In addition to fetch length variables, the data can be further refined as part of wave power calculations.

The actual wave formation is a function of fetch, wind speed and duration and the characteristics of water (Anon, 1977, 1984; Kahma and Calkoen, 1991, Pethick, 1991). In addition, the propagating waves bend due to friction between water particles and sea bed. This results in phenomena known as refraction and diffraction (Gamito and Musgrave, 2002; Pethick, 1991). Refraction and diffraction are not incorporated in the proposed algorithm. These variables are incorporated in more sophisticated wave models, but these models are not easily applicable in complex archipelago environments due to lack of detailed bathymetry and wind data. Such models would also require a high calculation capacity because the spatial resolution needs to be very detailed due to shoreline complexity. Even though wave activity is not a linear function of the distances over which the wind is affecting the surface water, fetch is an essential concept since it can be used to quantify general exposure. While recognizing the limitations to model actual wave behavior, relatively accurate calculations concerning wave power can be made based on the fetch data. Classifications based on long term average winds or detailed temporal analyses are among interesting applications incorporating wind data (Tolvanen and Suominen, 2005).

The presented algorithm was applied in coastal research with good results. However, also other disciplines could benefit from the ability to quantify distances and directions to polygon borders. Landscape ecology, for instance, considers patch sizes and distances. Ecological units in the landscape interact through boundaries, which are equivalent to polygon outlines like shorelines. Also disciplines such as archeology and infrastructure studies would likely have needs to quantify directional distances in the landscape.

Acknowledgments

The authors wish to thank professor Risto Kalliola for comments on the manuscript. The Department of Geography and Laboratory of Computer Cartography at the University of Turku provided hardware and software facilities for the research.

References

Anon, 1977. Shore Protection Manual, vol. 1, third ed. U.S. Army Corps of Engineers, Coastal Engineering Research Center, Washington DC.

Anon, 1984. Shore Protection Manual, vol. 1, fourth ed. U.S. Army Corps of Engineers, Coastal Engineering Research Center, Washington DC.

Arfken, G.B., Weber, H.J., 1995. Mathematical Methods for Physicists, fourth ed. Academic Press, USA, 1029pp.

Cormen, T., Leiserson, C., Rivest, R., Stein, C., 2001. Introduction to Algorithms, second ed. MIT Press, Cambridge, MA, USA, 1180pp.

Ekebom, J., Laihonen, P., Suominen, T., 2003. A GIS-based stepwise procedure for assessing physical exposure in fragmented archipelagos. Estuarine, Coastal and Shelf Science 57, 887–898.

ESRI, 2005. Data & Maps, Europe Countries SDC feature database. ESRI Inc., Redlands, CA, USA.

Frisén, R., Johansson, C., Suominen, V., 2005. Archipelagos in the Baltic Sea. In: Seppälä, M. (Ed.), The Physical Geography of Fennoscandia. Oxford University Press, Oxford, pp. 267–281.

Gamito, M., Musgrave, F., 2002. An accurate model of wave refraction over shallow water. Computers & Graphics 26, 291–307.

Håkansson, L., 1981. A Manual of Lake Morphometry. Springer, Berlin, 78pp.

Kahma, K., Calkoen, C.J., 1991. Reconciling discrepancies in the observed growth of wind-generated waves. Journal of Physical Oceanography 22, 1389–1405.

McCreight, E.M., 1985. Priority search trees. SIAM Journal on Computing 14, 257–276.

Pethick, J., 1991. An Introduction to Coastal Geomorphology, fifth ed. Edward Arnold, London, 260pp.

Preparata, F., Shamos, M., 1988. Computational Geometry, an Introduction, second ed. Springer, New York.

Tolvanen, H., Suominen, T., 2005. Quantification of openness and wave activity in archipelago environments. Estuarine, Coastal and Shelf Science 64, 436–446.

Publication II

Mika Murtojärvi, Ville Leppänen, Olli S. Nevalainen. Determining directional distances between points and shorelines using sweep line technique. *International Journal of Geographical Information Science* 23(3), 2009, 355–368. DOI: 10.1080/13658810801909607.

This article is included only in the printed version of the thesis.

Publication III

Mika Murtojärvi, Ville Leppänen, Olli S. Nevalainen. A parallel GPU implementation of an algorithm for determining directional distances. International Conference on Computer Systems and Technologies - CompSysTech'11, Vienna, Austria, 2011, 198–203. DOI: 10.1145/2023607.2023642.
© 2011 Association for Computing Machinery, Inc. Reprinted by permission.

A parallel GPU implementation of an algorithm for determining directional distances

Mika Murtojärvi, Ville Leppänen and Olli S. Nevalainen

Abstract: *The paper describes a parallel implementation of a cell-based algorithm for determining directional distances from points to polygons. Such distances have been applied, e.g., in coastal research. While the parallelization of the algorithm is in principle straightforward, the limitations of GPU devices lead to challenges in obtaining good performance. Our simple parallel GPU implementation achieves 8-fold speedup compared to a CPU implementation, yet maximal possible speedup is not achieved.*

Key words: *Computational Geometry, Directional Distance, GPGPU.*

1. INTRODUCTION

Distances from points to shorelines have been used, e.g., in coastal research for estimating wave exposure [3]. Of particular importance are quantities called *fetch lengths* (Fig. 1). For a given study point and direction, the fetch length is the distance from the point to the nearest land area. For a point that is located on a shoreline, fetch length is zero if the direction points into land area. Otherwise, the fetch length for a point on a shoreline is defined similarly as when the point is in open water. The application to wave exposure estimation results because waves in a water area grow in amplitude when wind interacts with them over a distance, although other factors must also be considered [3].

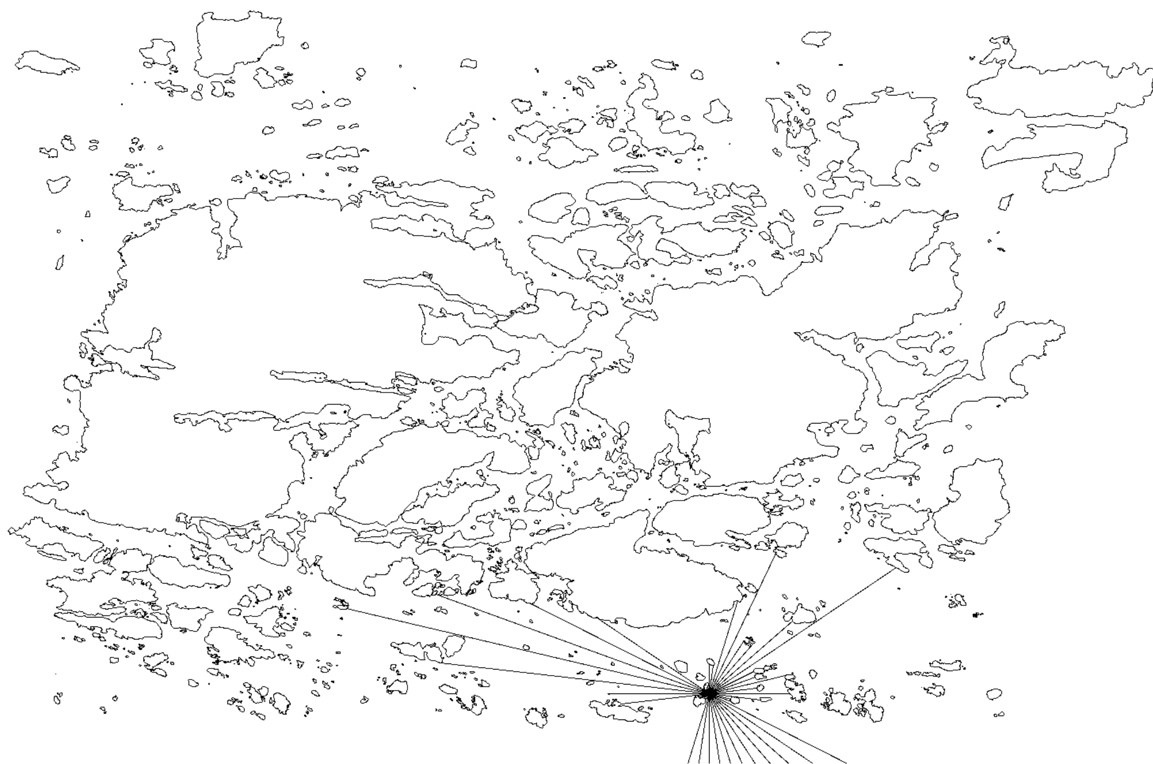

Figure 1. Fetch lengths for one study point in 48 directions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CompSysTech'11, June 16–17, 2011, Vienna, Austria.
Copyright ©2011 ACM 978-1-4503-0917-2/11/06...$10.00.

Geographical information systems (GIS) offer tools for determining directional distances. However, if distances are required for millions of study points, the tools have turned out to be too slow [6]. Therefore algorithms (e.g. [6], [7], [8]) have been developed for this purpose. The algorithms assume that the map is represented as a set of polygons, where one polygon represents the border of an island or the shoreline of the mainland. Some algorithms also require that fetch lengths are computed in the same directions for all study points [6], [7]. This assumption makes finding fetch lengths efficiently easier without being too restrictive for the geographical applications. Nevertheless, the fetch length algorithm [8] that has performed best in practical tests does not require the distances to be determined in the same directions for all study points. The algorithm [8] is based on uniform subdivision of the map. A fetch length is found by traversing the grid cells of the map in the direction of the fetch line until a cell that contains a line segment intersecting the fetch line is found. The worst-case time complexity of the algorithm [8] is not good, but this is often not relevant with real-world digital maps where the line segments describing the shorelines are short compared to the size of the map.

One appealing feature of the cell-based fetch length algorithm [8] is that it has very short preprocessing times. After the preprocessing step, the fetch length computations can be done independently for all study points. Therefore, when traditional CPUs are used, it is straightforward to make an efficient parallel implementation of the algorithm.

A modern computer may also have another powerful processor, called GPU (graphics processing unit). GPUs can be several times faster than CPUs in terms of peak floating point performance. For example, the ATI Radeon 5850 used in this study has peak floating point performance of 2088 Gflops, whereas the quad-core AMD Phenom 9850 CPU (2.5 GHz) is capable of 80 Gflops [1]. To appreciate the challenges in implementing algorithms on GPUs, let us review some characteristics of the GPU used in this work [1]. At the lowest level, the Radeon 5850 GPU has 1440 Processing Elements, capable of performing floating point and integer operations. The Processing Elements are grouped into Stream Cores. One Stream Core has five processing elements and circuitry for flow control, and there are thus 288 Stream Cores. Each Stream Core can execute a thread of a program. However, the Stream Cores are not independent. The Radeon 5850 has 18 independent Compute Units, each containing 16 Stream Cores. Within a Compute Unit, all Stream Cores execute the same instruction on each clock cycle. Furthermore, to hide the latencies of arithmetic operations and memory accesses, each Stream Core executes the same instruction with different data for four consecutive clock cycles. For program correctness it is not necessary to know these details, but performance implications can be significant. With suitably crafted program code the synchronous execution of the threads within a Compute Unit can reduce the performance to $1/64^{th}$ of the theoretical peak performance.

Memory accesses are also a potential bottleneck. For instance, the onboard memory (Global Memory) of the Radeon 5850 can transfer 128 GB/s in the best case [1]. Given that a single-precision floating point number requires 4 bytes, the transfer rate leads to a performance of only 32 Gflops if every operation requires reading one new value from the memory. To avoid such bottlenecks there are small cache memories on the GPU. The Radeon 5850 has 8 kB of level 1 cache in each Compute Unit and there is also 512 kB of slower level 2 cache. Each Compute Unit also contains 32 kB of local memory (LDS, Local Data Store). Unlike the caches, the contents of the LDS need to be modified explicitly in the program. It can therefore be difficult to use the LDS to yield good performance.

This study deals with a parallel implementation of the algorithm [8] on a GPU device and compares its speed to the same algorithm running on a CPU.

2. Solving the fetch length problem

2.1 Fetch length algorithms

Both approximate and exact methods for determining fetch lengths have been developed [3], [6], [7], [8]. Also an approximate algorithm for determining the maximum fetch length is known [5]. However, the implementations of the approximate algorithms have not been much faster than the exact algorithms. We focus on the exact cell-based algorithm [8] because of its good performance and relative ease of parallelization.

In the uniform subdivision approach [8] the plane is divided into rectangular *cells* by vertical and horizontal lines. Consecutive horizontal (and vertical) lines defining the subdivision of the plane into cells are at equal distance from each other. Each cell contains a list of all line segments of the map that are partially or entirely within that cell. The algorithm can be implemented using relatively simple data structures. The cells can be organized as a two-dimensional array and the contents of a cell can be represented by a variable-sized one-dimensional array. The preprocessing step of the algorithm rasterizes the line segments of the map to find the cells to which they are stored. The preprocessed map can be used for determining fetch lengths in any direction. Computing a fetch length is similarly based on rasterization. To find a fetch length, the cells through which the fetch line passes are traversed until a cell that contains a line segment intersecting the fetch line (within the cell) is found. Within each cell the intersections of the line segments of the cell with the fetch line are found using a brute force search that considers all line segments of the cell. A cell usually only contains a small number of line segments and the brute force scan is therefore efficient. Indeed, it has been found [8] that the cell-based approach outperforms the sweep line based method [7] with real-world maps, even if it performs more intersection computations. Furthermore, the preprocessing times of the algorithm were found to be less than one second with a map containing up to one million line segments [8]. Therefore it is not necessary to parallelize the preprocessing step to achieve good performance. The processing of the different study points, on the other hand, can easily be done in parallel. However, for an efficient implementation on a GPU there are several difficulties. One challenge is that the memory accesses can be scattered, consecutive memory positions being accessed in consecutive iterations only when the rasterization of a fetch line proceeds in a horizontal direction. The fetch lengths for the study points may also differ significantly from each other. This can cause inefficiencies in a GPU implementation where all threads within a group proceed synchronously.

2.2 A parallel GPU implementation of the cell-based algorithm

We implemented the uniform subdivision based method [8] using OpenCL 1.0 [4]. Using OpenCL allows easy evaluation of the speedup that is achieved by using a GPU instead of a CPU, since the same program can be compiled for both devices. However, there are significant differences between the devices and ultimately separate OpenCL implementations of the algorithm were developed for the CPU and GPU. The OpenCL programs running on a GPU or a CPU are also referred to as *compute kernels*.

The preprocessing of the algorithm is fast [8] and it was therefore implemented as a sequential C program running on the CPU. The preprocessing stores the line segments of the map into grid cells. This step was implemented using a rasterization algorithm that has been described by Cleary and Wyvill in the context of ray tracing [2]. The result of the preprocessing is a two-dimensional array where each element contains a list of all line segments of the map that are at least partially inside the grid cell. For the OpenCL compute kernel this array was transformed into two one-dimensional arrays. The first of the arrays ($segments$) contains the line segments that intersect each cell. A line segment is represented as a four-element vector of floating point numbers. The line segments of the same cell are stored in consecutive positions of the array $segments$. Different cells contain a different number of line segments, and the second array ($positions$) contains the indices

of the first line segments of the cells in the array $segments$. Thus, the line segments of a cell i are found in positions $segments[positions[i]],...,segments[positions[i+1]-1]$. Here $i = y \cdot num_cells_x + x$, where x and y give the position of the cell in the original two-dimensional array of cells and num_cells_x is the number of cells in the horizontal direction. The number of cells is chosen as in [8]: $num_cells_x = \lfloor \sqrt{n \cdot w / h} \rfloor$ and $num_cells_y = \lfloor \sqrt{n \cdot h / w} \rfloor$, where num_cells_y is the number of cells in the vertical direction, n is the number of line segments in the map and w and h are the width and the height of the map. Thus, there are approximately as many cells as there are line segments in the map, and the cells have nearly the same size in vertical and horizontal direction.

For the OpenCL compute kernel, we implemented two simple approaches. In both cases a single invocation of a kernel computes the fetch length for one study point. OpenCL takes care of invoking the kernel the specified number of times. Thus, the source code of the kernels is similar to a sequential C implementation except that there is no outer loop iterating the computation through the study points. Different directions of the fetch lines are processed separately, because the amount of memory available to an OpenCL compute kernel is only 512 MB for the Radeon 5850 with 1 GB of memory.

The first compute kernel (K1) processes at every iteration of the rasterization all line segments that are stored in the current cell. It then proceeds to the next cell within the same iteration of its outer loop. This approach might be highly inefficient on a GPU device. For example, if only one of the cells processed by a group of Stream Cores contains line segments, all threads that are executed on cores within the same compute unit proceed synchronously with the thread that needs to process line segments.

In the second kernel (K2) the rasterization of a fetch line proceeds until a nonempty cell is found. All line segments in this nonempty cell are then processed. The procedure ends if a line segment intersects the fetch line within the cell. Otherwise the rasterization is continued until the next nonempty cell. Also this approach can lead to some of the threads being idle due to the differences in fetch lengths and numbers of line segments in nonempty cells. However, this approach may lead to somewhat smaller negative consequences than K1 resulting from the synchronous execution of the threads.

3. Experimental results

The two implementations (K1 and K2) of the algorithm were tested using an AMD Phenom X4 9850 quad-core CPU and an ATI Radeon 5850 GPU. The computer had 4 GB of main memory and 1 GB of memory for the GPU and it was running 64-bit Linux operating system. All operations that are used frequently in the algorithm were optimized separately for the CPU and the GPU. The map contained 53301 vertices. Larger maps were generated for performance testing by tiling several copies of the map next to each other. Different numbers of study points were generated randomly within the map area.

The running times of the kernels that are faster on each device (K1 for the CPU and K2 for the GPU) are shown in Table 1. When the number of study points is small, the execution times are smaller on the CPU. With a large number of points the algorithm runs faster on the GPU than on the CPU. The performance difference depends on the size of the map. When the size of the map was increased from 53301 to 3.4 million vertices, the execution time of the algorithm increased considerably on the CPU. On the GPU the time increased less than on the CPU. In general, the algorithm runs about 4-8 times faster on the GPU than on the quad-core CPU with a large number of study points. In comparison, a parallel brute force algorithm for determining fetch lengths was found to perform 20 times faster on the GPU than on the CPU; this is close to the difference between the floating point performances of the devices. The rasterization was also almost 20 times faster on the GPU than on the CPU when the rasterization was performed to the border of the map

without memory accesses. For the rasterization the difference is even greater than expected considering that on the GPU some of the threads are idle due to different distances from the study points to map borders. Indeed, it was possible to construct a case where the rasterization worked over 50 times faster on the GPU than on the CPU.

Table 1. Kernel execution times on an AMD Phenom 9850 X4 quad core CPU and on an ATI Radeon 5850 GPU. In all cases fetch lengths were determined in 48 directions.

Vertices	Study points	CPU K1 (s)	GPU K2 (s)	GPU speedup
53301	1000	0.01	0.02	0.41
53301	100000	0.68	0.19	3.59
53301	5000000	31.63	8.52	3.71
213204	1000	0.01	0.02	0.61
213204	100000	1.02	0.24	4.26
213204	5000000	47.42	10.57	4.48
852816	1000	0.02	0.03	0.68
852816	100000	1.81	0.29	6.19
852816	5000000	86.57	13.24	6.54
3411264	1000	0.03	0.04	0.93
3411264	100000	2.96	0.39	7.67
3411264	5000000	147.49	17.82	8.28

Possible explanations for the smaller than expected difference in performance between the CPU and the GPU are the synchronous operation of the threads and the smaller caches in a GPU. In further tests the fetch lines were rasterized but the intersection tests were omitted. Two cases were tested. In the first case the fetch line was rasterized to the first nonempty cell and in the second case to the border of the map. In both cases similar memory accesses were performed. The relative times between these two test cases indicate how much rasterization work the CPU and GPU perform compared to rasterizing to the border of the map. As can be seen in Table 2, the relative times of rasterizing to border vs. rasterizing to the first nonempty cell are slightly lower on the GPU, suggesting that the synchronous execution of the threads on the GPU explains a part of the worse than optimal performance. Note that for the GPU the comparison is to a time that is already affected by the synchronous execution of the threads.

The most significant bottleneck on the GPU, at least for the rasterization, seems to be the time required for memory accesses. It was observed that memory accesses slow down the rasterization to the border of the map by a factor of 7 on the GPU with a small map but only 1.6-fold on the CPU. With a larger map memory accesses affected performance on both devices: when the map contained 3.4 million vertices, memory accesses slowed down the rasterization by a factor of 7 and 9 on the CPU and GPU, respectively. Then, the larger caches of the CPU could not likely achieve good hit rates.

Table 2. Running times for rasterizing the fetch lines to the first nonempty cell and to the border of the map on the CPU and the GPU. In all cases, there were 5 million study points and 48 directions. Ratio gives the time required for rasterizing to the border divided by the time required for rasterizing to the first nonempty cell.

Vertices	GPU (s)	GPU, to border (s)	CPU (s)	CPU, to border (s)	ratio (GPU)	ratio (CPU)
53301	2.93	17.06	13.66	81.43	5.83	5.96
213204	4.12	40.03	18.16	193.28	9.71	10.64

852816	5.69	88.75	40.89	764.86	15.61	18.71
3411264	8.24	183.33	88.41	2690.76	22.24	30.44

4. CONCLUSIONS AND FUTURE WORK

A cell-based algorithm for evaluating fetch lengths was implemented on a GPU. Its performance was found to be much better than that of the same algorithm running on a quad-core CPU. However, the performance difference was smaller than ideal. For the rasterization step of the algorithm, memory accesses were found to be the most significant explanation for the worse than expected performance. Suitable sorting of the study points might improve the performance. It remains to be determined whether memory accesses are the most important bottleneck also for the entire algorithm.

The implementation was not work-efficient on a GPU, because the fetch lengths and the numbers of line segments in nonempty cells vary. The efficiency could be improved by organizing the work in a different way.

REFERENCES

[1] Advanced Micro Devices, Inc. (2011, Jan). *AMD Accelerated Parallel Processing OpenCL™ Programming Guide* (v1.2). [Online]. Available: http://developer.amd.com/gpu/AMDAPPSDK/assets/ AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.

[2] J.G. Cleary and G. Wyvill, "Analysis of an algorithm for fast ray tracing using uniform space subdivision," *The Visual Computer*, vol. 4, no.2, pp. 65-83, 1988.

[3] J. Ekebom, P. Laihonen and T. Suominen, " A GIS-based step-wise procedure for assessing physical exposure in fragmented archipelagos," *Estuarine, Coastal and Shelf Science*, vol 57, no. 5-6, pp. 887-898, Aug. 2003.

[4] Khronos Group. (2009, Oct. 6). *OpenCL 1.0 Specification* (revision 48). [Online] Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[5] D. Lundqvist, D. Jansen, T. Balstroem and C. Christiansen, "A GIS-Based Method to Determine Maximum Fetch Applied to the North Sea–Baltic Sea Transition," *Journal of Coastal Research*, vol. 22, no. 3, pp. 640-644, 2006.

[6] M. Murtojärvi, T. Suominen, H. Tolvanen, V. Leppänen and O. S. Nevalainen, "Quantifying distances from points to polygons – applications in determining fetch in coastal environments," *Computers & Geosciences*, vol. 33, no. 7, pp. 834-852, July 2007.

[7] M. Murtojärvi, V. Leppänen and O.S. Nevalainen, "Determining directional distances between points and shorelines using sweep line technique," *International Journal of Geographical Information Science*, vol. 23, no. 3, pp. 355-368, Mar. 2009.

[8] S. Yang, J. H. Yong, J. G. Sun, H. J. Gu and J. C. Paul, "A cell-based algorithm for evaluating directional distances in GIS," *International Journal of Geographical Information Science*, vol. 24, no. 4, pp. 577-590, Apr. 2010.

…………………………………………………………………………………………………

ABOUT THE AUTHORS

Doctoral student Mika Murtojärvi MSc, adjunct professor Ville Leppänen PhD and professor Olli S. Nevalainen PhD, Department of Information Technology & Turku Centre of Computer Science (TUCS), FI-20014, University of Turku, Finland. E-mail: mika.murtojarvi@utu.fi, ville.leppanen@utu.fi, olli.nevalainen@utu.fi.

Publication IV

Mika Murtojärvi, Olli S. Nevalainen, Ville Leppänen. Performance tuning and sparse traversal technique for a cell-based fetch length algorithm on a GPU. *Concurrency and Computation: Practice and Experience* 27 (17), 2015, 5114–5133. DOI: 10.1002/cpe.3529. Reprinted with permission from John Wiley and Sons.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3529

Performance tuning and sparse traversal technique for a cell-based fetch length algorithm on a GPU

Mika Murtojärvi[1,*,†], Olli S. Nevalainen[2] and Ville Leppänen[2]

[1] *Vaadin Ltd, FI-20540 Turku, Finland*
[2] *Department of Information Technology and Turku Centre for Computer Science (TUCS), University of Turku, FI-20014 Turku, Finland*

SUMMARY

For determining distances (fetch lengths) from points to polygons in a two-dimensional Euclidean plane, cell-based algorithms provide a simple and effective solution. They divide the input area into a grid of *cells* that cover the area. The objects are stored into the appropriate cells, and the resulting structure is used for solving the problem. When the input objects are distributed unevenly or the cell size is small, most of the cells may be empty. The representation is then called *sparse*. In the method proposed in this work, each cell contains information about its distance to the nonempty cells. It is then possible to skip over several empty cells at a time without memory accesses. A cell-based fetch length algorithm is implemented on a graphics processing unit (GPU). Because control flow divergence reduces its performance, several methods to reduce the divergence are studied. While many of the explicit attempts turn out to be unsuccessful, sorting of the input data and sparse traversal are observed to greatly improve performance: compared with the initial GPU implementation, up to 45-fold speedup is reached. The speed improvement is greatest when the map is very sparse and the points are given in a random order. Copyright © 2015 John Wiley & Sons, Ltd.

Received 01 October 2014; Revised 19 February 2015; Accepted 10 April 2015

KEY WORDS: GPGPU; sparse rasterization; cell-based algorithms

1. INTRODUCTION

In computational geometry, cell-based algorithms have found applications, for example, in determining intersections of line segments [1], inclusion tests [2], and finding minimal distances from a set of study points to polygons [3]. The operating principle of these methods is relatively simple. As a preprocessing step, the considered area is divided by evenly-spaced horizontal and vertical lines into equal-sized *cells*, and the input objects are stored into the appropriate cells. The resulting cell-based representation is then used for solving the problem of interest.

1.1. Fetch length problem

The problem of determining distances from a given set of study points to shorelines originates from coastal research, where such distances are called *fetch lengths* [4]. It is assumed that the map of islands and the mainland is represented as polygons. The problem is then equivalent to finding, for each study point p and direction θ, the smallest distance $d(p, \theta)$ from p to the border or the interior of a polygon in the direction θ.

*Correspondence to: Mika Murtojärvi, Vaadin Ltd, FI-20540 Turku, Finland.
†E-mail: mianmu2@hotmail.com

Copyright © 2015 John Wiley & Sons, Ltd.

1.2. Previous work

When the number of study points is small, the fetch length problem can be solved simply by iterating over all line segments of the map for every study point and direction, recording the smallest found distances for the study points in the given directions. However, when the number of both study points and line segments of the polygons is large, this method is no longer efficient enough. Most published algorithms for the fetch length problem therefore aim at limiting the number of line segments that need to be examined. The present study also deals with this case where there are numerous study points and line segments.

An algorithm that utilizes the tools available in geographical information systems (GIS) has been applied for determining fetch lengths [4, 5]. While the algorithm has been described in some detail, it is difficult to determine whether it uses any method for limiting the number of intersection computations; that depends on the operation of the libraries that were used for clipping line segments with polygons.

Interval trees [6] and sweep line technique [7] have been used for limiting the number of intersection computations. Both approaches are based on the assumption that the fetch lengths are determined in the same directions for all study points. One can then rotate the map and point data and, after the rotation, determine the lengths in the horizontal direction. Then, for determining the fetch length for a point, the method based on interval trees performs an intersection computation for every line segment that intersects the horizontal line passing through the point. The sweep line method restricts the number of intersection computations further by keeping the line segments sorted in the horizontal direction. For a given study point, it is then only necessary to perform the intersection computation for at most one line segment for each level of a balanced binary search tree containing the line segments intersected by the current horizontal sweep line. The interval tree method has been reported to compute fetch lengths faster than the method using GIS software even when the distance between study points was decreased from 1000 to 25 m, a 1600-fold increase in the number of points [6]. When the number of study points is large, the sweep line method has been reported to be several times faster than the interval tree method [7].

One shortcoming of these methods is that they seem difficult to parallelize. In the interval tree method, the queries could be easily performed in parallel, but building the interval tree is more difficult. The sweep line method considers all study points and end points of the line segments in the order of increasing *y*-coordinate. The tree structure is also updated at every end point of a line segment.

Raster-based methods for the fetch length problem are also available [7, 8]. The inherent shortcoming of these methods is that the accuracy of the results depends on the resolution of the rasterization.

In practice, a cell-based algorithm [3] has been found to be the most efficient published algorithm for determining fetch lengths. When the number of study points is large, it has achieved fourfold performance compared with the sweep line-based method. The speedup is even greater when this number is small. The greater speedup in the latter case is to some extent explained by the fact that the algorithm uses the same data structure for all directions: there is no need to rotate the map data. In this approach, the map is divided into equal-sized cells by vertical and horizontal lines. For an illustration of the fetch length problem and its cell-based solution, see Figure 1.

The objects stored in the cells are the line segments of the polygon borders. The preprocessing step stores each line segment in all cells that it intersects. The result of the preprocessing is a two-dimensional array of cells with each cell containing a list of line segments. The number of cells is chosen so that the expected number of segments in a cell is small. More precisely, the numbers of cells in the horizontal and the vertical direction are $cells_x = \lceil \sqrt{n(w/h)} \rceil$ and $cells_y = \lceil \sqrt{n(h/w)} \rceil$, where n is the total number of line segments in the input data and w and h are the width and the height of the input area [3]. If the segments are short, there is approximately one line segment per cell. Otherwise, the formula could be refined to take into account the lengths of the segments. Nevertheless, the line segments are often distributed unevenly, and there may be areas containing empty cells and cells containing several line segments.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

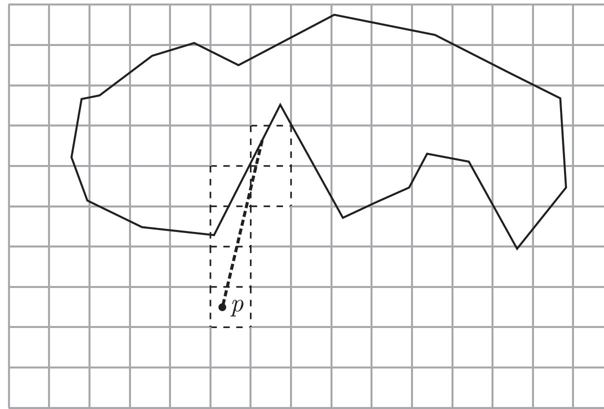

Figure 1. Determining directional distance for point p in one direction using a cell-based approach. The cells with broken lines are examined, starting from the cell containing p.

The processing step determines the directional distances for the study points and is similar to the preprocessing phase. Given a point p and a direction θ, the algorithm examines, starting from the cell containing p, the cells intersected by the half-line originating from p and having direction θ. If a cell is empty, the algorithm proceeds to the next cell intersected by the half-line. In a non-empty cell, all line segments stored in the cell are examined for possible intersection with the half-line. If one or more intersections are found inside the current cell, the distance computation is ready. Otherwise, the algorithm proceeds to the next cell. The process of visiting the cells intersected by a line segment or a half-line is called *traversal* or *rasterization* and is performed in the order of increasing distance from the starting point. The pseudocode of the algorithm is given in Section 2.

The distance computations for different study points or directions are independent of each other and can be performed in parallel. A simple GPU implementation of the algorithm has been reported to significantly outperform a parallel implementation running on a quad-core CPU [9]. Nevertheless, there are several factors that may limit the performance of the algorithm on a GPU.

Traversing empty cells requires very little arithmetic, but a memory access is needed in each cell to find out whether it is empty. Memory accesses are then an obvious target for optimization. The sparseness of the data can be exploited by examining only a part of the empty cells during the processing phase. Both sorting and sparse techniques have been used, for example, in computer graphics [10, 11], but for the cell-based distance determination algorithm, new variations of these general techniques are required.

1.3. The operation of a graphics processing unit

Another factor that may limit the performance of an algorithm on a GPU is divergent control flow. To illustrate the problem, consider as a concrete example the Radeon R9 280X GPU (AMD, Sunnyvale, CA, USA) that is used in most of the tests in this article. For a more detailed description of the GPU or the almost identical Radeon 7970, see [12, 13] and [14]. The GPU consists of 32 compute units (CU). Each compute unit contains one scalar ALU (Arithmetic and Logic Unit) and four vector ALUs (VALU), each VALU containing 16 processing elements. The scalar ALU is used, for example, for the branching decisions required by a program. The vector ALU is, *from the programmer's point of view*, a 64-wide single instruction, multiple data (SIMD) unit, that is, an instruction of the VALU applies the same operation on each element of a vector consisting of 64 (32-bit) values. Because the VALU is physically only 16-wide, it performs each instruction in four parts.

The organization of the GPU is hidden from an OpenCL programmer; the use of vector instructions is handled automatically by the compiler. The typical execution model of GPU programs (*compute kernels*) makes the process reasonably easy for the compiler. Namely, it is typical to apply the *same* compute kernel for a large number of items, called *work-items* in OpenCL. It is then straightforward to use the (logically 64-wide) SIMD unit for processing 64 work-items in parallel. Such a block of work-items is called a *wavefront*. The only difficulty is that the compute kernel

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

may contain conditional statements that should be executed by some but not all work-items of the wavefront. This can be handled by masking operations. Before entering a conditional block, the condition is evaluated for all work-items, obtaining an execution mask. Then, the conditional block is evaluated using vector instructions. As a result of the execution mask, only those work-items whose condition is $true$ will actually update any variables. This method allows the GPU to correctly execute conditional code blocks. From a performance point of view, they can still be problematic. In the worst case, all but one of the work-items of a wavefront have been masked out of execution, leading to a loss of performance. For more details, see [12].

Divergent control flow within a wavefront, also called wavefront divergence, can be a significant problem for the cell-based distance determination algorithm. For some study points, it is only necessary to find the first nonempty cell and process its segments to determine the distance. If no intersection is found in the first nonempty cell, the process is repeated until an intersection is found or the map border is reached. The number of cells that are traversed to find the first non-empty cell also depends on the position of the point and on the direction. Finally, different nonempty cells contain different numbers of line segments.

1.4. Plan of the work

Optimization of the cell-based fetch length algorithm for a GPU is studied in the present work. While the algorithm is simple to implement, improving its performance using low-level and high-level optimizations turns out to be more difficult than was expected. Indeed, many of our experiments are unsuccessful. The main contributions of the article are a sparse rasterization algorithm for line segments and a simple yet effective partial sorting method. The aim of the sorting is to make the work-items of the same wavefront to access a similar set of cells, improving the use of cache memories and reducing the negative effects of wavefront divergence. In this work, it is assumed that the fetch lengths are determined in the same directions for all study points. Otherwise, the sorting algorithm would need to be refined to also consider the directions.

This article is organized as follows. Section 2 describes the studied problem and the initial implementation of the cell-based algorithm. Low-level performance tuning is discussed in Section 3. This includes reducing unnecessary register usage and optimizing the memory access patterns of the algorithm. A summary of the failed attempts to reduce the effects of wavefront divergence is also given. A more comprehensive discussion of the attempts can be found in the Appendix. In Section 4, we propose a sparse traversal algorithm and partial sorting of the study points. Results of performance tests are summarized in Section 5. Section 6 concludes the paper.

2. DETERMINING FETCH LENGTHS

A cell-based algorithm for solving the fetch length problem is given in [3]. The initial GPU implementation of the algorithm is similar to that described in [9]. The rasterization algorithm used in both the preprocessing and the processing phases of the algorithm is based on the algorithm by Cleary and Wyvill [15].

The rasterization algorithm is the basis for sparse rasterization (Section 4.1) and is briefly reviewed here. The algorithm keeps track of the indices (i, j) (the column and the row) of the current cell. Initially, (i, j) is the cell containing the starting point p of the line segment or half-line. The variables dx and dy contain the distances from p to the next points where the segment crosses vertical and horizontal cell borders, respectively. If $dx \leqslant dy$, the rasterization next proceeds in the horizontal direction. Then, i is incremented by px and dx by ∂x, where $px = \pm 1$ and ∂x is the distance between two consecutive points where the segment crosses a vertical cell border. Similar updates are done when the rasterization proceeds vertically. For an illustration of the rasterization algorithm, see [15] or Section 4.1.

Because the preprocessing (building the cell-based acceleration structure) only needs to be performed once regardless of the number of study points and directions, its performance is less important than that of the processing phase. The focus of optimization is therefore on the processing phase. For one study point p and direction θ, the processing is shown in Algorithm 1.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 1 Determining a directional distance [3].

```
procedure DIRECTIONALDISTANCE(Map m, Point p, Direction θ)
    Initialize ∂x, ∂y, dx, dy, i, j, px and py using m, p and θ
    minDist := ∞; nearestSegment := null; ready := false
    numLines := linesInCell(m, (i, j))
    while not ready do
        while numLines = 0 and inBounds(m, (i, j)) do
            if dx ⩽ dy then
                i := i + px; dx := dx + ∂x
            else
                j := j + py; dy := dy + ∂y
            numLines := linesInCell(m, (i, j))
        for all line segments in the cell (i, j) of m do
            update minDist, nearestSegment and ready if necessary
        numLines := 0
    return getDistance(p, θ, nearestSegment)
```

The inner while-loop of the algorithm finds the next non-empty cell on the half-line originating from p and having direction θ. The for-loop examines all line segments of the current cell (i, j) for intersection with the half-line within the cell. If there are such intersections, the outer loop is terminated by setting the variable $ready$ to $true$. Otherwise, the outer loop is repeated in order to find the next non-empty cell along the half-line. The process continues until an intersection is found or the border of the map is reached.

The initial GPU implementation of the algorithm stores all line segments of all cells in one linear array ($segments$), arranged so that the line segments of any particular cell are in consecutive positions of the array. Another array ($firstSegment$) contains, for every cell, the index of the first line segment of the cell in $segments$. The segments of a cell (i, j) are thus stored in $segments[firstSegment(i, j), ..., firstSegment(i+1, j)-1]$. The representation is compact, but two memory accesses are required to test whether a given cell is empty.

Although the algorithm can be used when the directions θ are different for different points, in this article, the directions are supposed to be the same for all points. This corresponds to an actual use case [4] and allows the sharing of some variables for all study points. Different directions are handled using separate kernel launches. In the initial implementation, there are as many work-items as there are study points. The compute kernel is then similar to Algorithm 1.

3. LOW LEVEL PERFORMANCE TUNING

Algorithm 1 was implemented as an OpenCL compute kernel. Low level performance tuning was then done to identify unnecessary register usage and to rearrange the cell representation of the map so that it can be accessed efficiently. The tuning was done on the level of OpenCL program code. The effects of program code modifications are then not entirely predictable, because an attempted optimization may already have been done by the compiler. A profiling and debugging tool (AMD CodeXL) was used to obtain information on the resource usage of the compiled kernel.

3.1. The memory hierarchy of the graphics processing unit

Low level performance tuning depends on the characteristics of the particular GPU used for running the algorithm. A simplified block diagram of the Radeon R9 280X GPU is shown in Figure 2. As stated earlier, the GPU consists of 32 CU, each containing one scalar ALU (SALU) and four VALUs.

As for the memory hierarchy [12, 13], each CU contains general purpose registers, cache, and LDS (local data share) memory. The 16 KiB read/write level 1 cache is shared among the four VALUs of the CU, as is the 64 KiB LDS memory. In contrast to the cache, the contents of the LDS

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

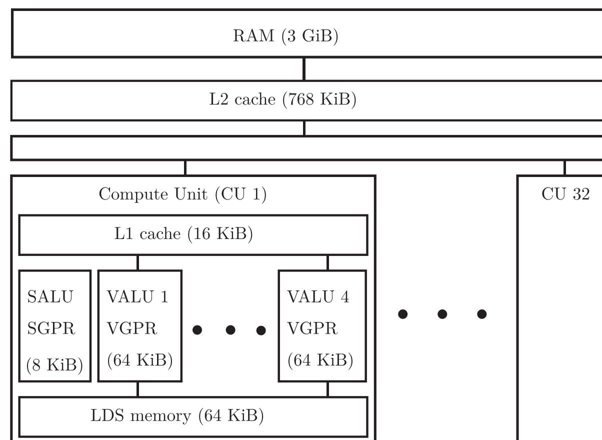

Figure 2. The structure of the Radeon R9 280X GPU. Global data share (GDS) memory is not shown. SALU, scalar Arithmetic and Logic Unit; VALU, vector Arithmetic and Logic Unit; LDS, local data share.

are explicitly maintained by the compute kernel. Each VALU also contains 256 vector registers (VGPR) with a vector size of 64 32-bit values, for a total of 64 KiB of register space (256 KiB in a compute unit containing four VALUs). Similarly, a SALU contains 8 KiB of scalar registers (SGPR). Shared among the CU, there is 768 KiB of level 2 cache. Finally, the board contains 3 GiB of global memory.

The memory spaces differ greatly in performance. The global memory has the lowest bandwidth (288 GB/s), followed by the L2 and L1 caches and the LDS memory. The registers can be accessed much faster than any other memory, with a total bandwidth of 27,000 GiB/s [13][‡]. This range of bandwidth needs to be taken into account in efficient implementations of algorithms.

The resources of a CU or a VALU are shared among all wavefronts running on the same unit. Low level performance tuning then has possibly conflicting goals: Resource requirements should be reduced to allow running a greater number of active wavefronts, but the performance characteristics of the different memory spaces need to be taken into account.

3.2. *Improving kernel occupancy*

Kernel occupancy is defined as the ratio of the number of active wavefronts and the maximum number of active wavefronts supported by the device [14]. On a GPU, it is usual that there are more active wavefronts than units executing them. The active wavefronts then run in a time-interleaved fashion,that is, the hardware repeatedly switches the wavefronts that are currently running. This can hide some of the latency associated with memory accesses [14] because it may be possible to run another wavefront when one or more wavefronts are waiting for a memory access. A greater number of active wavefronts gives better possibility for covering the latency.

Reducing the register usage. On the Radeon R9 280X, three resources may affect the kernel occupancy: vector registers (VGPRs), scalar registers (SGPRs) and LDS memory [12, 14]. The initial implementation did not use LDS memory, leaving the required numbers of VGPRs and SGPRs as optimization targets. Using the AMD CodeXL profiling tool VGPRs were found to be the main limiting factor: The kernel required 40 registers, and at most, six wavefronts could then run on a vector ALU. The maximum supported number of wavefronts per VALU is 10 [13], so the kernel occupancy was 60%.

[‡]The guide does not state the bandwidths for the memory spaces of the R9 280X. However, Radeon 7970 is identical to R9 280X except for clock speeds. The R9 280X used in this article has 1500 MHz memory clock and 1100 MHz clock speed for the compute units. Because of hardware failure, the performance tests of this section were run on a Radeon 7950, whereas the R9 280X is used in Section 5. The GPUs differ in clock speeds and the number of CU (28 in Radeon 7950).

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Because the directions θ are the same for all study points, several variables $(\partial x, \partial y, px, py)$ in Algorithm 1 are also the same for all points. Their values can be stored in the SGPRs instead of the VGPRs. This was done by precomputing these four values using the CPU and giving them as parameters to the kernel. Other possibilities for reducing register usage were also found in the initial implementation. For instance, contrary to the pseudocode, the variable $nearestSegment$ was not a reference to a segment but contained all four coordinates of the end points of the segment. By reducing the VGPR usage, the kernel occupancy was improved to 80%. This had a significant effect on performance: The running time of a test case was reduced from 4.5 to 3.4 s. Further reducing VGPR usage did not improve performance, because the number of active wavefronts was limited to 8 also by the required scalar registers. It is also possible to use the LDS (or global memory) instead of registers for some variables to further increase kernel occupancy. However, these memory spaces are slower than the registers, leading to a loss of performance when frequently used variables are stored in them.

3.3. *Arranging the data*

As told before, our initial implementation stores the map using two arrays, $segments$ and $firstSegment$. Two memory accesses ($firstSegment(i, j)$ and $firstSegment(i + 1, j)$) are required to determine whether a particular cell is empty. Because ca. 84% of the cells were empty in the test map, the memory accesses of the empty cells require a considerable amount of time. By using slightly more memory to store the map data, the number of memory accesses was reduced to one for an empty cell. Now, a third array $numSegments$ contains the number of line segments in each cell.

A two-dimensional array may not be an ideal data structure for cell traversal (the inner while-loop of Algorithm 1) on a GPU. GPUs contain dedicated hardware for the processing of image data, and OpenCL has a data type that can use this hardware [14]. The OpenCL image data type is suitable for storing the contents of the two-dimensional arrays, $firstSegment$ and $numSegments$. Because there were only a small number of line segments in each cell (at most 59 and on average six segments in a non-empty cell), 16-bit integers were used in the structure $numSegments$. In the case study, the use of OpenCL images improved performance, but only when the arrays $numSegments$ and $firstSegment$ were stored as two separate images.

The initial implementation checked after each rasterization step that the current cell is still in the map area. The bounds-checking can be eliminated by surrounding the map with a thin strip of values that do not occur in the actual map area [15]. When OpenCL images are used, even indexing outside this augmented map can be allowed. An appropriate *sampler* [16] then returns the border value.

3.4. *Wavefront divergence*

As stated earlier, the Radeon R9 280X processes the work-items in groups of 64 items, called wavefronts [12]. The work-items of a wavefront share a single program counter and hence execute the same sequence of instructions. Divergent control flow, such as conditional statements or loops with varying numbers of iterations, may lead to poor utilization of the hardware as some of the work-items of a wavefront need to be masked out of execution. Control flow divergence within a wavefront occurs in all three loops of Algorithm 1, because their iteration counts depend on the location of the study point processed by the work-item. Also the if-else block needs to execute both the if-branch and the else-branch if the wavefront contains work-items choosing the if-branch and work-items choosing the else-branch. This cannot happen when the rasterization direction is horizontal or vertical but for other directions the location of a study point within its cell affects the choices made in the rasterization loop. Compared with the loop, the divergence caused by the if-else block is minor: In the worst case, the amount of work required for the if-else block is twice the amount of work that would be required if the work-items were fully independent threads.

Unfortunately, the attempts to improve performance by reducing wavefront divergence were unsuccessful. The techniques that were tested include

(1) replacing a study point with a new one when its processing is carried out,

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

(2) proceeding the rasterization for multiple study points instead of only one in Algorithm 1, and
(3) low-level tuning for if-else blocks.

These techniques are described and analyzed in more detail in the Appendix, but here is a short summary of the results. The first and second methods require that one work-item processes multiple study points. The rationale behind the first method is that after each iteration of the outer loop, the number of active work-items in a wavefront decreases as the processing becomes ready for some study points. This causes the later iterations to underutilize the SIMD units of the GPU. However, the first iteration of the outer loop tends to do much more work than the other iterations. Then, replacing the finished study points with new ones worsens the divergence problem in the inner loops as the wavefronts contain study points in both the first and in the later iterations of the outer loop. The second method is based on the knowledge that the sum of iteration counts of several independent points tends to be less variable than the iteration counts themselves. Implementing the method required using the LDS memory to gain fast indexed access to the variables of the different study points. Because the LDS is small, kernel occupancy was reduced, leading to a loss of performance. The effect of the low-level modifications of the if-else block was too small to measure reliably.

4. SPARSE RASTERIZATION AND PARTIAL SORTING

Traversing empty cells can require a significant amount of time in the cell-based algorithm. When most cells of the map are empty, the performance of the rasterization can be improved by skipping over several empty cells in a rasterization step. This requires a revised cell traversal algorithm and a preprocessing step that computes the skip distances for the cells. The skip distances must be chosen so that all non-empty cells along a half-line are traversed. On the other hand, the distances can be conservative so that some empty cells are still traversed.

4.1. Sparse rasterization algorithm

The rasterization algorithm by Cleary and Wyvill [15] is used as the basis for the proposed sparse rasterization algorithm (SRA). As in the original algorithm, the variables dx and dy give the distances from the starting point of the half-line to the next vertical and horizontal cell borders, both measured along the half-line. The integer-valued variables i and j contain the x- and y-coordinates of the current cell. It is assumed that the map coordinates have been scaled so that all x-coordinate and y-coordinate lie in the ranges $[0, \text{cells}_x + 1]$ and $[0, \text{cells}_y + 1]$, respectively. A cell with coordinates (i, j) then contains a point (x, y) if $i \leqslant x < i + 1$ and $j \leqslant y < j + 1$.

If it is known that the next d cells are empty, these cells can be traversed without performing additional memory accesses. When run on a GPU, the traversal might still be inefficient because of wavefront divergence: If the work-items of the same wavefront require different numbers of rasterization steps, the running time of the wavefront is determined by the work-item requiring the greatest number of steps. Therefore, the rasterization algorithm is modified to proceed by multiple cells in one step. In the distance map of SRA, each empty cell of the map stores a number D indicating that the rasterization may proceed from the cell until D horizontal or vertical transitions have been performed, whichever comes first (Figure 3).

This means that at the rasterization step, either dx and i are incremented by $D \cdot \partial x$ and $D \cdot px$ or dy and j are incremented by $D \cdot \partial y$ and $D \cdot py$, respectively. It remains to determine which of these cases applies and how to update the other two variables.

The smallest distance from the study point p to the first cell (on the half-line) that is horizontally D cells away from the current cell is $dx + (D-1)\partial x$. A similar formula applies for the first cell that is vertically D cells away from the current cell. The rasterization meets the cells in an increasing order of their distance from the starting point, so the number of horizontal transitions is D exactly if $dx + (D-1)\partial x \leqslant dy + (D-1)\partial y$. Otherwise, the rasterization proceeds vertically by D cells. The number of transitions in the other direction is given by the following lemma.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

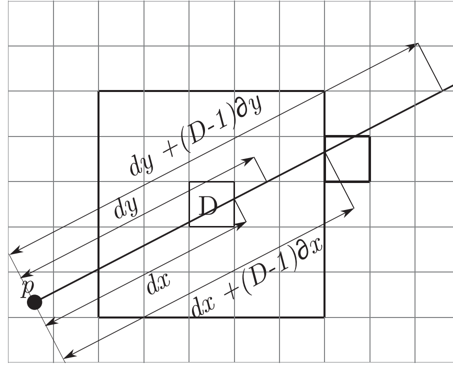

Figure 3. Sparse rasterization in a current cell containing the distance $D = 3$. Rasterization may proceed directly to the nearest cell (bolded cell on the right) that is not contained in the area that is known to be empty (the bolded big square). The distances of other cells than the current cell are not shown.

Lemma 4.1
Suppose that a sparse rasterization step moves horizontally by D cells, where D is the distance stored in the current cell. It then moves vertically by $D' = \lceil (dx + (D-1)\partial x - dy)/\partial y \rceil$ cells.

Proof
Consider the operation of the non-sparse rasterization algorithm, starting from the current cell and proceeding until D horizontal transitions have been done. Let dx' and dy' stand for the values of the variables dx and dy in the target cell and let dx'' and dy'' be their values before the last transition in the non-sparse rasterization. Because the last transition is horizontal, the value of the variable dy is the same before and after the last transition, $dy'' = dy' = dy + D' \cdot \partial y$. The value of the variable dx before the transition is $dx'' = dx' - \partial x = dx + (D-1)\partial x$. The non-sparse rasterization algorithm makes a horizontal transition if and only if the current value of dx is no greater than that of dy. For the transition entering the target cell, this means that $dx'' \leqslant dy''$, that is, $dx + (D-1)\partial x \leqslant dy + D' \cdot \partial y$. Solving for D' gives that $D' \geqslant (D-1)\frac{\partial x}{\partial y} + (dx - dy)/\partial y$. It is also clear that D' is the smallest integer satisfying the inequality, for otherwise the variable dy would have been incremented when it was greater than $dx + (D-1)\partial x$. Thus, $D' = \lceil (D-1)\frac{\partial x}{\partial y} + (dx - dy)/\partial y \rceil$, which is equivalent to the lemma. □

A similar formula can be derived for the case where the number of vertical transitions is D. The rasterization step of SRA is shown in Algorithm 2.

Algorithm 2 Sparse rasterization step in a cell with distance D.

$dx_cand := dx + (D-1)\partial x$
$dy_cand := dy + (D-1)\partial y$
$inc_x_steps := inc_y_steps := D$
if $dx_cand \leqslant dy_cand$ **then**
  $inc_y_steps := \lceil (dx_cand - dy)/\partial y \rceil$
else
  $inc_x_steps := \lfloor (dy_cand - dx)/\partial x + 1 \rfloor$
$i := i + px \cdot inc_x_steps$; $j := j + py \cdot inc_y_steps$
$dx := dx + \partial x \cdot inc_x_steps$; $dy := dy + \partial y \cdot inc_y_steps$

Algorithm 2 is slightly asymmetric with respect to the formulas for computing inc_x_steps and inc_y_steps. This follows from the condition $dx \leqslant dy$ in the non-sparse rasterization step of Algorithm 1. The other steps of Algorithm 1 are the same as before.

Copyright © 2015 John Wiley & Sons, Ltd.
DOI: 10.1002/cpe

4.2. Preprocessing the map

When SRA is used, a preprocessing step is required for computing the distance information for each cell. Here, a distance map that is the same for all directions is used. The number D stored in a cell is the minimal number of vertical, horizontal, and diagonal transitions between neighboring cells that must be made in order to reach a non-empty cell starting from the given cell.

The computation of the distance map starts by an initial pass that finds all distance-0 (non-empty) cells. These cells are stored into a queue[§]. All other cells, including the border cells, are given the value ∞. The distance-1 cells are then those neighbors of the distance-0 cells that do not themselves have distance 0. They are now inserted into another queue, making it easy to find the distance-2 cells. The process is repeated, with increasing distances, until all cells have been given a finite distance. This preprocessing step is similar to the breadth-first search (BFS) algorithm [17]. The difference is that in the BFS algorithm, there is one starting node that is given the distance 0, whereas in the map preprocessing, there are several distance-0 cells. Otherwise, the two algorithms are almost identical when one considers the cell-based map as a graph whose nodes (cells) are connected to all (up to 8) their neighboring cells that share either a borderline (the vertical and horizontal neighbors) or a border point (the diagonal neighbors) with the cell.

To reduce the amount of memory operations, the distances and the numbers of line segments of the non-empty cells can be stored into the same array. This can be done by using positive numbers to represent distances and negative numbers to represent the numbers of line segments in the non-empty cells. The value 0 is stored into the border cells to indicate that rasterization cannot proceed any further. When the OpenCL image data type and a suitable sampler are used, bounds checking can still be avoided.

4.3. Partial sorting

When the distances are determined in the same directions for all study points, one can expect points that are geometrically close to each other to require accessing the same or almost the same cells. The fetch lines originating from such points may also end in the same or nearby objects, which leads to similar iteration counts in the loops of Algorithm 1. This means that inefficiency caused by both memory accesses and by wavefront divergence can be reduced by having the wavefronts consist of points that are close to each other. The work-items (here corresponding to points) are assigned to wavefronts in a sequential order on the Radeon R9 280X [13], that is, the first 64 work-items form the first wavefront, and the rest of the work-items are assigned to wavefronts in a similar manner. The geometrical proximity of the points belonging to the same wavefront can thus be achieved by sorting them before any other processing is carried out. However, preprocessing must be fast to improve the overall performance.

A simple cell-based sorting method is used here. A separate initially empty list of points is stored in each cell. Then, the set of study points is scanned, and each point is assigned to the list of the cell in which the point lies. Finally, the lists are merged to form the sorted list of points.

The cells are traversed in a zigzag order when merging the point lists. The first row of cells is traversed from left to right, then the second row from right to left, and so on. Compared with lexicographic order, this order reduces the occurrence of cases where two points are close to each other in the sorted order but far from each other geometrically. In the lexicographic order, the rightmost cell of a row would be followed by the leftmost cell of the next row, leading to cases where points belonging to the same wavefront are horizontally far from each other. The sorting is called *partial* because there is no specific order for points belonging to the same cell. The sorting can use a different division of the input area into cells than the map, because the optimal sorting cell size may depend on the density of the study points.

[§]The data structure is called a queue just to emphasize the similarity to the breadth-first search algorithm.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

4.4. An optimization for a large number of study points

The cell-based fetch length algorithm has been designed so that it can determine a distance for a point in any direction. However, if the distances were needed only in the horizontal direction, the procedure could be made more efficient. The rasterization algorithm then only needs to move in one direction, eliminating the need for several variables and possibly increasing kernel occupancy. Finding out whether a half-line and a line segment intersect, and if so determining the intersection point, is also simpler in the case where the half-line is horizontal. When sparse rasterization is used, the distance map can easily be constructed to contain the exact skip distances, eliminating the need for traversing almost all empty cells. The distance map computation can be implemented as a right-to-left scan of each row of the cell representation of the map. A non-empty cell has a distance zero, and for an empty cell, the distance is increased by one compared with the right neighbor of the cell.

In practice, the fetch lengths may be required in the same directions for all study points, but most of the directions are neither horizontal nor vertical. It is then necessary to rotate the map so that in the rotated map the distances are determined horizontally, the fetch lines extending right from the study points. As the cell representation of a rotated map differs from the representation of the original map, it needs to be rebuilt for each direction. The same is true for the skip distance map. The execution times of these procedures depend on the characteristics of the map, but not on the number or location of study points. Hence, when the number of study points is large enough, performing the fetch length determination on a rotated map can be faster than on the original map.

5. PERFORMANCE TESTING

Performance testing was done for different variants of the fetch length determination algorithm on a computer with 8 GiB memory, Intel Core i7 3770K CPU (Santa Clara, CA, USA) and AMD Radeon R9 280X GPU. The operating system was 64-bit openSUSE 12.3 Linux. A map of a small portion of the Finnish Archipelago was used as input data. The original map contained 1278 islands represented as polygons with a total of 53,301 vertices. Because of the rather small size of the input map, larger semi-artificial maps were obtained by tiling several copies of the original map next to each other. In all cases, the same number of copies was tiled in the horizontal and the vertical direction. The preprocessing steps (building the cell structure, sorting, and distance map generation for sparse rasterization) were implemented as serial algorithms running on the CPU.

The cell-based algorithm was also implemented as both a serial and a parallel algorithm running on the CPU. The algorithm has been reported to outperform the plane sweep algorithm [3] which, in turn, is faster than the algorithm based on interval trees [7]. The method using GIS software [4, 5] is also known to be much slower than the interval tree method [6]. The sweep line and interval tree methods are also difficult to fully parallelize. For these reasons, only the cell-based method was implemented for the performance testing.

For the set of study points, two generation strategies were tested. The first strategy chooses the locations of the study points randomly in the map area, with uniform distribution. This can be expected to be a difficult case for the performance of the cell-based algorithm, because memory accesses are highly scattered. Another strategy was to place the points in a $n * m$ grid covering the entire map area and with n and m chosen so that the total number of points (nm) was as desired. It was further preferred that n and m are relatively close to each other: in all tested cases $\max(n/m, m/n) \leqslant 2.5$.

5.1. Tests with fixed sparseness

In the first set of tests, all maps were almost equally sparse, because they were obtained by tiling the same map, and the same formula for computing the number of cells was used. The computations were carried out for 48 directions for every study point, using a separate run of the compute kernel for the different directions. The running times of the different kernels are shown in Table I. Note that non-kernel times will be discussed separately and are not included in the table. In tests with

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table I. Kernel running times (in seconds) of determining fetch lengths in 48 directions. The first two columns give the numbers of polygon vertices and study points. G indicates that the study points were placed in a grid, in other cases, the locations of the study points were random. 'CPU serial' is an unoptimized serial implementation of the algorithm. 'Init.' refers to the initial GPU implementation of the algorithm, 'Occ.' to its occupancy-optimized version. '2buf' uses a separate table containing the numbers of line segments in the cells. 'Im.' is like '2buf' but uses OpenCL images. 'SRA' uses sparse traversal, 'Sort' presorts the data, and 'So+SRA' uses both presorting and sparse traversal.

Vertices	Points	Serial CPU	Init.	Occ.	2buf	Im.	SRA	Sort	So + SRA
53,301	1,000,000	12.2	0.23	0.20	0.19	0.19	0.12	0.06	0.05
53,301	5,000,000	58.9	1.16	1.02	0.90	0.92	0.56	0.27	0.16
213,204	1,000,000	16.8	0.32	0.29	0.26	0.26	0.17	0.09	0.07
213,204	5,000,000	83.8	1.57	1.39	1.25	1.26	0.84	0.29	0.23
852,816	1,000,000	26.3	0.53	0.54	0.43	0.37	0.28	0.14	0.09
852,816	5,000,000	130	2.60	2.59	2.09	1.81	1.38	0.74	0.33
3,411,264	1,000,000	37.9	0.84	0.85	0.71	0.54	0.35	0.20	0.12
3,411,264	5,000,000	188.3	4.10	4.13	3.44	2.53	1.76	0.74	0.43
3,411,264	$25 \cdot 10^6$	966	20.4	20.5	17.1	12.5	8.54	2.39	1.63
3,411,264	$100 \cdot 10^6$	—	81.4	81.9	68.3	49.9	39.1	7.00	4.75
53,301	1,000,000 (G)	10.3	0.11	0.097	0.088	0.090	0.072	0.060	0.048
53,301	5,000,000 (G)	50.0	0.36	0.33	0.30	0.30	0.26	0.19	0.15
3,411,264	5,000,000 (G)	82.1	1.12	0.99	0.93	0.91	0.54	0.67	0.44
3,411,264	$25 \cdot 10^6$ (G)	400	4.53	3.90	3.60	3.60	2.34	2.41	1.66
3,411,264	$100 \cdot 10^6$ (G)	—	14.7	12.6	11.8	11.7	7.90	7.00	4.77

GPU, graphics processing unit; SRA, sparse rasterization algorithm.

more than 10 million study points, memory transfers from the GPU to main memory were omitted because there was not enough main memory to store all the results. The CPU version was also not tested in these cases.

The CPU versions were mainly intended as a starting point for developing the GPU compute kernels. They were implemented in C language but were less optimized than the GPU versions. As a result, the performance difference between the serial CPU implementation and the initial GPU implementation was greater than expected. A parallel version of the CPU algorithm was also tested (results not shown in the table). When using random study point data, it was found to be four to five times faster than the serial version on the quad-core CPU with Hyper-Threading, capable of running eight threads simultaneously. With grid data, the parallel CPU version was about three times faster than the serial version.

The performance difference between the different GPU compute kernels depended somewhat on the location generation strategy of the study points. For random data, the occupancy optimizations ('Occ.'(Section 3.2)) had very little and sometimes negative effect on performance, whereas with grid data these optimizations slightly improved performance. Using a *separate buffer* ('2buf') for storing the numbers of line segments in the cells improved performance, since only one memory access was then needed to determine whether a given cell is empty. Using the OpenCL image data type ('Im'.) had a positive effect when using random data, but with grid data the benefit was very small. *Sparse rasterization* ('SRA') reduced the number of memory accesses and rasterization steps, leading to better performance with both datasets. The sparse rasterization step is, however, more complicated than that of the original rasterization, and in a dense map the method can have a negative effect on performance. For a worst case test, the distances of the nonempty cells were artificially set to $D = 1$. Then, in a test case the implementation using SRA required 20% more time than the non-sparse version. The difference can be attributed to the increased complexity of the rasterization step, since the kernel occupancy was the same (80%) in both cases and both methods traversed the same cells.

The greatest improvement in performance was obtained by sorting the study points. For the grid data most of the benefit was already achieved by the sorting caused by the point generation method: the study points were sorted in a lexicographic order, with a horizontal row of points being consecutive in the order. The *partial sorting* ('Sort') did have a positive effect on performance with both

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

data sets but for randomly chosen study points the benefit was far greater than for grid data. Sorting was also the only method whose performance improved considerably, i.e., processing time per study point was decreased, when the number of study points was very large (greater than $5 \cdot 10^6$). The cells used in the sorting corresponded to blocks of cells in the map. It was required that there is on average at least a user-specified number N_p of study points in a sorting cell. This was achieved by merging the sorting cells so that there were $cells_x/n$ and $cells_y/n$ cells in the horizontal and the vertical direction. For the test case involving the largest map and 5 million study points, best performance was achieved by merging the cells so that there are at least 16 points / cell. The same number $N_p = 16$ was used in all tests. The merging was only done for the cell-based sorting, i.e. the cell representation of the map was unaffected.

Using sparse rasterization in addition to sorting further reduced the kernel time. With the largest map the preprocessing time required for building the distance map was 0.18 seconds (note that this is non-kernel time and is not included in Table I). Sparse rasterization then improved performance compared to only sorting the data when at least 5 million study points were used, but with 1 million points, the preprocessing time exceeded the difference (0.10 s) in kernel running times. The partial sorting required 0.57 seconds for 5 million study points. Building the cell-based representation of the largest map required ca. 0.7 s. All preprocessing phases used single-threaded implementations running on the CPU. Transferring data between the main memory and the global memory of the GPU required 0.3 s for 5 million study points and a map with 3.4 million vertices.

The optimization for a very large number of study points, that is, computing the distances in a horizontal direction and rotating the map when necessary (Section 4.4), was only tested with 10 million study points and in one horizontal direction. In that case, the method was ca. 40% faster than the generic implementation when sorting and sparse rasterization were used. However, when the distances are determined in 48 directions, the method using rotation would need to build 48 cell representations of the map[¶] instead of only one for the generic implementation. The distance map computation also needs to be performed separately for each direction to get a map that allows skipping as many empty cells as possible. One may then calculate that with 100 million study points and 48 directions (Table I), a 40% speedup would save about 1.36 s. Because 47 additional cell structures and distance maps need to be built, they would need to be built in less than 0.029 s for one direction, while the actual time using a serial implementation was more than 0.8 s. This might be possible with a parallel GPU implementation, but the total speedup taking into account the additional preprocessing times would be minor, and a full implementation of the method was therefore omitted.

5.2. *The impact of sparseness*

To study how sparseness affects the performance of the different variations of the cell-based algorithm, the formula for computing the number of cells was modified into $cells_x(\alpha) = \lceil\sqrt{\alpha \cdot n(w/h)}\rceil$ and $cells_y(\alpha) = \lceil\sqrt{\alpha \cdot n(h/w)}\rceil$, where w and h are the width and the height of the map. In other words, compared with the original formula, the total number of cells was multiplied (approximately) by the expansion factor α. In all tests, there were 10 million randomly chosen study points and 3.4 million vertices in the map. The results of this test are shown in Figure 4.

As expected, sparse rasterization is most useful when the map has large empty areas, corresponding to the tests with a large value of α. When used together with the partial sorting, up to 45-fold performance compared with the initial implementation is achieved when $\alpha = 7.5$. Sparse rasterization alone yields up to 13-fold performance and partial sorting alone up to 11-fold performance compared with the initial implementation. At lower expansion factors, partial sorting improves performance more than sparse rasterization, and in all tested cases, partial sorting with sparse traversal is at least as efficient as sorting alone. The use of OpenCL images without sorting or sparse rasterization gives at most 2.9-fold performance compared with the initial implementation.

[¶]In our test cases this could be reduced to 12 because the same map could be used for two vertical and two horizontal directions. Distance maps are still required for every direction.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

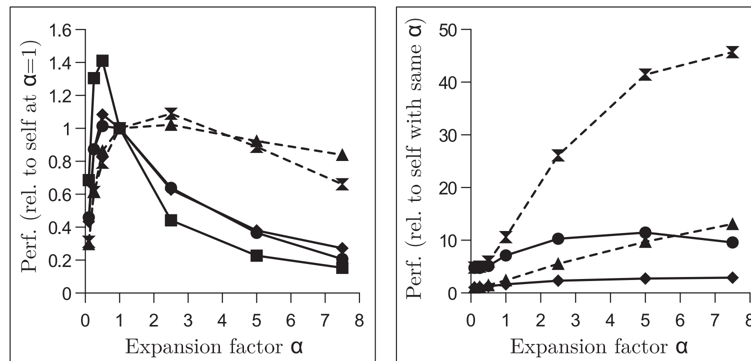

Figure 4. The impact of sparseness on the performance (Perf.) of the different methods. The dashed lines represent methods using sparse rasterization (triangle: SRA, double triangle: SRA + sorting), solid lines non-sparse versions (square: initial, tilted square: OpenCL image, circle: sorting). Left: The performance of each method compared with itself with expansion factor $\alpha = 1$. Right: The performance of each method compared with the initial implementation.

The tests indicate that the initial implementation performs best when the number of cells is slightly reduced (the left panel of Figure 4). It is then 41% faster (when $\alpha = 0.5$) than with the original number of cells. The version using OpenCL images without sorting also performs best with $\alpha = 0.5$, but it only gains 8% performance compared with using $\alpha = 1$. The sparse rasterization or sorting alone performs best when $\alpha = 1$, but when they are used together, best performance is achieved with $\alpha = 2.5$. For all methods, the expansion factor has a significant effect on performance: The sparse methods prefer a high expansion factor, the non-sparse methods, a relatively low expansion factor. However, even with expansion factor 0.1, sparse rasterization was as efficient as the initial implementation or the version using OpenCL images alone.

6. CONCLUSIONS

An algorithm [3] for determining directional distances was studied to improve its performance on a GPU. Successful low-level optimization included reducing register usage to improve kernel occupancy, reducing the number of memory accesses in empty cells, and using the OpenCL image data type. On the other hand, many attempts to reduce the negative effects of wavefront divergence were unsuccessful. One of these decreased kernel occupancy while another one replaced the finished study points with new ones to reduce the occurrence of cases where some work-items of a wavefront are idle for the entire iteration of the outer loop of the algorithm. However, this had a negative effect on the divergence in the inner loops. Using different kernel invocations for the different iterations of the outer loop increased non-kernel time, leading to reduced overall performance.

The greatest performance increase was achieved by partially sorting the study points before determining the distances. The sorting was designed so that geometrically close points tend to be close to each other in the sorted order. Because geometrically close points can be expected to require a similar sequence of cells to be traversed, the sorting can improve cache hit rate and reduce wavefront divergence. The sorting had the greatest positive effect on performance when the study points were initially in a random order, but for study points located in a grid improvement by sorting was still observed.

Sparse traversal improved the performance of the algorithm by skipping over several empty cells in a rasterization step. While the effect of sparse traversal was in general smaller than that of sorting, the method was equally useful for both grid and random data. The best performance was achieved by using both partial sorting and sparse traversal.

A simple distance map that is the same for all directions was used for the sparse traversal. The skip distances could be further increased if a direction-specific distance map was used instead. Computing several maps increases the preprocessing time, and a parallel implementation for this preprocessing would then be necessary.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

An area for further research is applying the sparse rasterization and the sorting methods to other problems. A problem similar to fetch length determination has been studied when simulating wireless communication channels [18]. Intersection problems [1], proximity queries [19], and point-in-polygon tests [20] are examples of problems where sorting or sparse techniques could also be useful. However, the required preprocessing steps limit the usability of these techniques to cases where the same cells are accessed frequently.

APPENDIX A

The attempts to reduce the negative effects of wavefront divergence were briefly discussed in Section 3.4. Here, the reasons for why the methods were not successful are studied in more detail. Because of hardware failure, the tests of this section were done on a Radeon 7950 instead of the Radeon R9 280X. The cards use the same graphics processing unit (GPU), but on the Radeon 7950, only 28 of the 32 compute units are enabled, and the clock speed is lower.

A.1. Low level divergence optimization

Low level divergence optimization methods include replacing if-blocks with functionally equivalent code using no conditional statements but using conditional statements when they can prevent the evaluation of significant code segments for the entire wavefront.

The first optimization can be used for the if-else block of Algorithm 1. Because the test $dx \leqslant dy$ has the value 0 or 1, the if-branch can be replaced by $i := i + px \cdot (dx \leqslant dy); dx := dx + \partial x \cdot (dx \leqslant dy)$. The else-block can be modified similarly. In this modified rasterization step, both the if-branch and else-branch are always evaluated. However, the arithmetic in the branches is very simple, and in most cases, both branches are evaluated also in the original algorithm: The evaluation of a branch is only avoided if no work-items in a wavefront take that branch. In practice, the effect of this change on performance was minor. In some cases, small performance improvements (up to 1.5%) were consistently seen, but in other cases, there was no or slightly negative change in performance. Other attempts to deal with this source of divergence were even less successful, causing a loss of performance because of using a slower memory space (LDS) than the registers to gain indexed access to variables such as dx and dy.

The second optimization was used in the intersection detection test that consisted of two phases. In a majority of cases, the first phase was already sufficient for establishing that the half-line does not intersect the current line segment. It was then faster to return the result (that there is no intersection) than to compute the result of the second phase. Other parts of the algorithm, such as the determination of an intersection point, were similarly tuned by experimentation to get the best performance. The variants of intersection tests that were found to perform best were used in all performance tests of this article, that is, they were not re-tuned for the different performance tests.

A.2. The outer loop

An iteration of the outer while-loop of Algorithm 1 finds the next non-empty cell and tests all segments of the cell for intersection with the half-line. The process continues until an intersection is found or the border of the map is reached. In the case study, the number of iterations of the outer loop ranged from 1 to over 50. In 56% of cases, at least two iterations were required and at least five iterations were required in 8% of cases. Low utilization of the SIMD units can then be expected after the first iteration.

A potential solution to the divergence problem of the outer loop has been used in path tracing [21]. It can be applied to the fetch length determination problem by using less work-items than there are study points. When a study point has been processed, a new point takes its place. Several different implementations of the idea are possible:

1. The variables for all points can be initialized and stored in global memory as a preprocessing step.
2. The initialization can be done when the processing of a new study point starts.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

(3) The set of points processed by a work-item can be determined before execution.
(4) The set of points processed by a work-item can be determined dynamically by using global or local memory atomic operations.
(5) If a work-item processes a small number of study points, their variables can be stored in a fast memory space so that switching to a new point can be done effectively.

Unfortunately, all implementations of the aforementioned variations led to *worse* performance than the initial implementation. A possible reason is that the first iteration of the outer loop takes more time than the other iterations. When a new study point takes the place of a finished study point, it begins its first iteration. Because of the SIMD execution, the other work-items are forced to proceed at the same slower rate as the work-item processing the new study point. To explore the reasons for the disappointing results of the experiments, a closer look at the execution of the parallel implementation is needed.

A.2.1. Modeling the execution time. There are two difficulties in developing a model for the running time of Algorithm 1 on a GPU: the effects of SIMD execution and global memory accesses. For arithmetic operations and register–register transfers, it is reasonable to assume that the execution time of a wavefront does not depend on the number of active work-items, because the operations are performed in parallel for the wavefront. The access to global memory, however, is shared among all execution units of the GPU. On the Radeon 7950, the bandwidth of the global memory is 300 bytes in clock cycle [13]. There are 1792 processing elements on the GPU, so it is possible to saturate the memory bus even if many work-items are idle. Once the memory bus is saturated, the time taken by a memory-bound wavefront is expected to grow almost linearly with the number of active work-items, because eventually all of them need their data from the global memory. A further difficulty originates from the two-level cache memory hierarchy. Instead of attempting to model the effects of the memory system, two highly simplified models are used to study whether the first iteration takes more time than the other iterations. If this is the case, replacing finished points with new ones can be expected to reduce performance because of wavefront divergence within the inner loops. Otherwise, it is possible that the observed reduction of performance resulted from an insufficient exploration of the different implementation choices.

Model 1: The time taken by n active work-items is supposed to be directly proportional to the number n. This model is best suited for a single-core CPU, but if the number of work-items is large, the parallel nature of the GPU can essentially be ignored. The model is then a simplification of the case where global memory accesses limit the performance.

Model 2: The time required for a task is supposed to be directly proportional to the number of wavefronts regardless of the numbers of active work-items in the wavefronts. This model is suitable for a case where only SIMD execution limits performance. The model does not take into account the divergence within the inner loops. Further tests are therefore performed to bound the degree of divergence in the inner loops.

For *Model* 1, suppose that the first iteration of the outer loop requires t_1 and the other iterations t_2 time. A study point requiring i iterations then takes $t = t_1 + (i-1)t_2$ time to process. Let p_i be the probability that exactly i iterations of the outer loop are required for a study point. Then, according to Model 1, the expected time required for a study point is [22]

$$T_{\text{tot}}^{(1)} = E(t) = \sum_{i=1}^{I_{\max}} (p_i \cdot (t_1 + (i-1)t_2)) = t_1 + t_2 \sum_{i=2}^{I_{max}} (p_i(i-1)), \tag{1}$$

where $I_{\max}$ is the maximum number of iterations required for any study point. When there are n points and w points can be processed in parallel, multiplying the right-hand side of Equation (1) by n/w gives the expected total execution time.

The times t_1 and t_2 can be determined by first running only the first iteration of the outer loop. This gives t_1 for the given input data, hardware, and number of study points. In another test, the algorithm is run until completion to obtain the total execution time $T_{\text{tot}}^{(1)}$. The estimates for the

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

probabilities p_i can be determined by recording the numbers of times that a work-item requires i iterations, for all values of i. Because recording the iteration counts increases the execution time, it is done in a run whose execution time is discarded. The time t_2 is finally solved from Equation (1).

Model 2 is based on the probabilities P_i that a *wavefront* requires exactly i iterations. A wavefront requires exactly i iterations if some of its work-items require exactly i iterations, and all of them require at most i iterations. A different formulation, however, leads to a simpler derivation for P_i: A wavefront requires exactly i iterations if all of its work-items require at most i iterations, but it is not true that all work-items require at most $i-1$ iterations. Let B_i stands for the event that all work-items of the wavefront require at most i iterations and let the number of work-items in a wavefront be N. Then $P_i = Pr(B_i \setminus B_{i-1}) = Pr(B_i) - Pr(B_{i-1})$. The probability that an individual work-item requires at most i iterations is $\sum_{j=1}^{i} p_j$. For a set of N independent work-items, the probability that all of them require at most i iterations is then $Pr(B_i) = \left(\sum_{j=1}^{i} p_j\right)^N$ and

$$P_i = Pr(B_i) - Pr(B_{i-1}) = \left(\sum_{j=1}^{i} p_j\right)^N - \left(\sum_{j=1}^{i-1} p_j\right)^N. \quad (2)$$

The total execution time is then

$$T_{\text{tot}}^{(2)} = \sum_{i=1}^{I_{max}} P_i \cdot (t_1 + (i-1)t_2) = t_1 + t_2 \sum_{i=2}^{I_{max}} P_i \cdot (i-1). \quad (3)$$

As before, it is assumed that there are much more wavefronts than there can be *active* wavefronts on the device. Otherwise, the assumption that the execution time is directly proportional to the number of wavefronts is not true, because it is not possible to perform enough work in parallel. The times t_1 and t_2 are determined as in Model 1.

A.2.2. Experimental results. For testing the possible difference between the times for the first and the other iterations of the outer while-loop, a semi-artificial map containing 3.4 million vertices was used. The map was obtained by tiling 64 copies of a real-world map in an 8×8 pattern. Five million points were selected randomly from the map area, and fetch lengths were determined for them in 48 directions. The probabilities p_i and the running times of the first iteration (t_1) and of all iterations ($T_{\text{tot}}^{(1)}$, $T_{\text{tot}}^{(2)}$) were determined as described earlier. Because the OpenCL compute kernel determined distances in one direction at a time, the reported times are the sums of 48 kernel execution times for the batch of 5 million points. The iteration counts of the loops were determined separately for the different directions, because otherwise the derived iteration count probabilities for the wavefronts would not correspond to the actual use case where the directions are handled separately.

The first iteration required $t_1 = 2.48$ s, while all iterations took $T_{\text{tot}}^{(1)} = T_{\text{tot}}^{(2)} = 3.44$ s. However, kernel occupancy was 100% when determining t_1 but only 80% for $T_{\text{tot}}^{(1)}$. Another experiment was therefore performed where the kernel occupancy was limited to 80% by introducing an LDS buffer of appropriate size. Surprisingly, the running time of the first iteration was then slightly lower than with 100% occupancy, $t_1 = 2.29$ s.

Estimates for the iteration count probabilities p_i were determined as described previously. Using these estimates, the value of the expression $\sum_{i=2}^{I_{max}} p_i \cdot (i-1)$ was ca. 1.17. Equation (1) then gives $t_2 \approx 0.99$ s, that is, according to Model 1, the first iteration takes more than twice the time of the other iterations, $t_1 \approx 2.3 t_2$.

For Model 2, the probabilities P_i were determined using formula 2, wavefront size $N = 64$, and the probabilities p_i. For coefficient of t_2 in Equation 3, the average value of the coefficients obtained for the different directions was used. Then, $\sum_{i=2}^{I_{max}} P_i \cdot (i-1) \approx 6.60$. Using this and the times $t_1 \approx 2.29$ s and $T_{\text{tot}}^{(2)} \approx 3.44$ s, Equation (3) gives $t_2 \approx 0.174$ s, so $t_1 \approx 13.1 t_2$. The difference of the results for the two models suggests that SIMD execution can have a large effect on the execution of the algorithm.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The expected iteration counts were then determined for the inner loops of the wavefronts using Equation (2). The results indicated that on the first iteration of the outer loop, a wavefront may require, on average, up to 15 times as many iterations of the inner while-loop than its work-items would require when considered individually. For most directions, this difference was much smaller, with an average ratio of 8.6. For the inner for-loop, the ratio did not depend much on the direction and was on average 3.8. Because both ratios are smaller than the ratio of the execution times[‖] of the first and the other iterations, Model 2 also indicates that the first iteration is more time-consuming than the other iterations even when the SIMD execution of the inner loops is taken into account.

It was also observed that the first iteration of the outer loop requires on average about eight times as much rasterization steps as the other iterations to find the next non-empty cell. The average numbers of line segments in the cell, on the other hand, were quite similar for the first and the other iterations of the outer loop (6.0 vs. 7.4 segments). It can then be expected that replacing a study point whose processing is ready with a new one considerably increases the execution time of the inner while-loop for the wavefront but does not have much effect on the for-loop. A memory bottleneck was also evident: It was up to 15 times faster to perform the rasterization to the map border when the memory accesses were omitted.

Another approach was also tested for dealing with the problem that the number of active work-items in a wavefront decreases as the outer loop proceeds. In that approach, the kernel was terminated after each iteration of the outer loop, and the remaining active work-items were handled using another invocation of the compute kernel, in effect regenerating the wavefronts so that they consist of active work-items. This approach was also not beneficial because of increased non-kernel time.

A.3. The inner loops

In the preceding tests, the first iteration of the outer loop required more time than all the other iterations together. The analysis of the inner loops is therefore performed assuming that the outer loop is in its first iteration. Similar observations can be made for the other iterations, but the number of active work-items in a wavefront is not a constant for the other iterations of the outer loop.

Depending on the position of a study point, a variable number of cells are traversed in the inner while-loop of Algorithm 1 before reaching a non-empty cell. Because of the SIMD execution, the number of iterations required by a wavefront is determined by the work-item requiring the greatest number of iterations. The number of iterations required for a wavefront is $I_{wf} = \max_{i=1}^{N} I_i$, where N is the number of work-items in the wavefront and I_i is the number of iterations required by the work-item i of the wavefront.

Suppose, then, that B wavefronts are run after each other on a compute unit. The total number of iterations is

$$I_{tot} = \sum_{j=1}^{B} I_{wf}^{j} = \sum_{j=1}^{B} \left(\max_{i=1}^{N} I_i^j \right), \tag{4}$$

where I_{wf}^{j} and I_i^j are the iteration counts for the jth wavefront and for the ith work-item of the jth wavefront.

The total number of iterations can be lowered by having each work-item compute a distance for a batch of B points instead of only one. The rasterization can then be implemented as given in Algorithm 3, where j is the index of the current study point, $p[j]$ is the current study point, and $cur_cell(p[j])$ is the current cell for the study point $p[j]$. The outer loop and the intersection detection phase are similar to Algorithm 1 and are omitted. As in Algorithm 1, rasterization and intersection computation phases are repeated until the distance computation is ready for all points. The number of remaining points in the batch decreases as the outer loop proceeds. This can be

[‖]This is not a necessary condition for the conclusion to hold. In general, the divergence in the other iterations of the outer loop should also be considered. Here, even the most unfavorable assumption of no divergence in the other iterations leads to the conclusion.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

handled in Algorithm 3 by decreasing B and rearranging the points or by ensuring that all points whose processing is ready are considered to lie in a non-empty cell.

Algorithm 3 Finding the next nonempty cells for B points.

$j := 1$
while $j \leqslant B$ **do**
  if $cur_cell(p[j])$ is empty **then**
    set $cur_cell(p[j])$ to the next cell
  else
    $j := j + 1$

Taken individually, work-item i would require $\sum_{j=1}^{B} I_i^j$ iterations of the while-loop. The number of iterations for the whole wavefront is then equal to the maximal number of iterations of its work-items, that is,

$$I'_{tot} = \max_{i=1}^{N} \left(\sum_{j=1}^{B} I_i^j \right). \tag{5}$$

The formula (4) of I_{tot} is a sum of maximums while I'_{tot} (5) is a maximum of sums. Hence, the modified rasterization requires less iterations than the original one. The actual degree of reduction depends on the distribution of the iteration counts of the rasterization. Choosing a large batch size B reduces wavefront divergence, but it also increases the resource requirements of the compute kernel, because several variables are required for all active points. These variables need to be stored in a fast memory space to allow quickly switching between points in Algorithm 3.

The attempts to improve rasterization performance using this method were unsuccessful because of lowered kernel occupancy. Rasterization uses four variables (dx, dy, i, and j) for each study point, and the LDS memory was found to be the only suitable memory space for allowing fast indexed access to these variables. There is 16 KiB of LDS memory in each vector ALU on the Radeon 7950. With only $B = 4$ study points/work-item, $N = 64$ work-items/wavefront, and four 32-bit variables for each study point, 4 KiB LDS memory is required for one wavefront. Only four active wavefronts/vector ALU can then be running, corresponding to 40% kernel occupancy. The reduced occupancy has an unfavorable impact on performance, as does the slower operation of the LDS memory compared with the registers (VGPRs).

Although this method was unsuccessful, the tests indicated that it might work on a device with more LDS memory: when the original kernel was limited to the same number of active wavefronts as the batched version, it was slower than the batched version. With $B = 16$, the rasterization performance of the batched version was more than twofold compared with the original version, but both implementations were slow because only one wavefront was running on each vector ALU.

The batched approach can also be used for the intersection detection phase. Similarly to the rasterization phase, no speedup was obtained.

REFERENCES

1. Andrews DS, Snoeyink J, Boritz J, Chan T, Denham G, Harrison J, Zhu C. Further comparison of algorithms for geometric intersection problems. *Proceedings of the Sixth International Symposium on Spatial Data Handling*, Taylor & Francis: London; Bristol, Pa, 1994; 709–724.
2. Žalik B, Kolingerova I. A cell-based point-in-polygon algorithm suitable for large sets of points. *Computers & Geosciences* 2001; **27**(10):1135–1145.
3. Yang S, Yong JH, Sun JG, Gu HJ, Paul JC. A cell-based algorithm for evaluating directional distances in GIS. *International Journal of Geographical Information Science* 2010; **24**(4):577–590.
4. Ekebom J, Laihonen P, Suominen T. A GIS-based step-wise procedure for assessing physical exposure in fragmented archipelagos. *Estuarine, Coastal and Shelf Science* 2003; **57**(5-6):887–898.
5. Tolvanen H, Suominen T. Quantification of openness and wave activity in archipelago environments. *Estuarine, Coastal and Shelf Science* 2005; **64**(2-3):436–446.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

6. Murtojärvi M, Suominen T, Tolvanen H, Leppänen V, Nevalainen OS. Quantification of openness and wave activity in archipelago environments. *Computers & Geosciences* 2007; **33**(7):843–852.
7. Murtojärvi M, Leppänen V, Nevalainen OS. Determining directional distances between points and shorelines using sweep line technique. *International Journal of Geographical Information Science* 2010; **23**(3):355–368.
8. Rohweder J, Rogala JT, Johnson BL, Anderson D, Clark S, Chamberlin F, Runyon K. *Application of wind fetch and wave models for habitat rehabilitation and enhancement projects*. pubs.usgs.gov/of/2008/1200/pdf/ofr2008-1200_web.pdf. [Accessed on 19 November 2014].
9. Murtojärvi M, Leppänen V, Nevalainen OS. A parallel GPU implementation of an algorithm for determining directional distances. *Computer Systems and Technologies, 12th International conference, CompSysTech '11*, Association for Computing Machinery: New York, NY, 2011; 198–203.
10. Garanzha K, Loop C. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum* 2010; **29**(2):289–298.
11. Barringer R, Akenine-Möller T. A4 : asynchronous adaptive anti-aliasing using shared memory. *ACM Transactions on Graphics (TOG) - SIGGRAPH 2013 Conference Proceedings*; **32**(4):100:1–100:9. article no. 100. DOI: 10.1145/2461912.2462015.
12. Advanced Micro Devices. *Reference Guide: Southern Islands Series Instruction Set Architecture*: Advanced Micro Devices, Inc, 2012. (Available from: http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf.) [Accessed on 31 August 2014].
13. *AMD Accelerated Parallel Processing OpenCL^{TM} Programming Guide*: Advanced Micro Devices, Inc, 2013. (Available from: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.) [Accessed on 31 August 2014].
14. Gaster BR, Howes L, Kaeli D R, Mistry P, Schaa D. *Heterogeneous Computing with OpenCL* (2nd ed). Morgan Kaufmann: USA, 2013.
15. Cleary JG, Wyvill G. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer* 1988; **4**(2):65–83.
16. Munshi A. *The OpenCL specification,* Version 1.2, document revision 19. Khronos OpenCL working group, 2012. (Available from: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.) [Accessed on 22 August 2013].
17. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*. The MIT Press: Cambridge, Massachusetts, 2009.
18. Bai S, Nicol D M. Acceleration of wireless channel simulation using GPUs. *2010 European Wireless Conference (EW 2010). Institute of Electrical and Electronics Engineers (IEEE)*, posted on 2010:841–848. DOI: 10.1109/EW.2010.5483525, (to appear in print).
19. Knorr EM, Ng RT. Algorithms for mining distance-based outliers in large datasets. *Proceedings of the 24rd International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc: San Francisco, 1998; 392–403.
20. Zhang J, You S. Speeding up large-scale point-in-polygon test based spatial join on GPUs. *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, Association for Computing Machinery: 2012; 23–32.
21. Novák J, Havran V, Dachsbacher C. Path regeneration for interactive path tracing. *Proc, EUROGRAPHICS Short Papers.* The European Association for Computer Graphics, 2010; 61–64.
22. DeGroot MH, Schervish MJ. *Probability and Statistics* (3rd ed). Addison-Wesley: USA, 2002.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Publication V

Mika Murtojärvi, Tapio Suominen, Esa Uusipaikka, Olli S. Nevalainen. Optimising an observational water monitoring network for Archipelago Sea, South West Finland. *Computers & Geosciences* 37 (7), 2011, 844–854. DOI: 10.1016/j.cageo.2011.01.006. Reprinted with permission from Elsevier.

Computers & Geosciences 37 (2011) 844–854

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

Optimising an observational water monitoring network for Archipelago Sea, South West Finland

Mika Murtojärvi [a,*], Tapio Suominen [b], Esa Uusipaikka [c], Olli S. Nevalainen [a]

[a] Department of Information Technology and Turku Centre for Computer Science (TUCS), University of Turku, FI-20014, Finland
[b] Department of Geography, University of Turku, FI-20014, Finland
[c] Department of Statistics, University of Turku, FI-20014, Finland

ARTICLE INFO

Article history:
Received 3 September 2009
Received in revised form
28 January 2011
Accepted 31 January 2011
Available online 1 March 2011

Keywords:
Water quality
The Baltic Sea
Observational network
Pruning algorithms

ABSTRACT

Water quality monitoring in topographically fragmented archipelago coasts calls for a dense observational network. However, visiting multiple sites and analyzing the samples requires a significant amount of work, leading to considerable economic cost. It is of interest to determine an efficient set of sites, which still offers adequate information on the water quality with a sufficient spatial accuracy. A method for optimizing an existing observational network is proposed. The method is concretized by applying it for an observational network in the Archipelago Sea, South West Finland. The network is pruned with the requirement that the observations of the removed sites can be estimated using those of the remaining sites. Suboptimal heuristics are used in pruning to keep the computational time acceptable. Some observations are not available and need to be estimated (imputed) before the pruning. For the network in the Archipelago Sea, the results of the pruning are somewhat sensitive to differences in imputed datasets and heuristics used for site selection.

© 2011 Elsevier Ltd. All rights reserved.

1. Introduction

The European Union's Water Framework Directive (WFD) (Anonymous, 2000) obliges the member states to provide sufficient data for an effective coastal and sea area management. However, similar sampling designs cannot be applied for all the surface waters, and monitoring must be adapted according to environmental conditions; factors such as the variability of the monitored quantities should be taken into account (e.g., Anonymous, 2003a; Ferreira et al., 2007; Nordic Council of Ministers, 2006). Special cases in this sense are the transitional estuarine and coastal waters, where highly dynamic water quality regimes require spatially and/or temporally dense sampling designs.

The state of the Baltic Sea has become a serious concern during recent decades (e.g., HELCOM, 2009). It is an inland sea with limited water exchange with the oceanic waters, which results in both vertical and horizontal gradients of water properties. The sea is also located in relatively high latitudes where the annual temperature range is wide, inducing strong temporal physical, chemical, and biological cycles in the waters above the permanent halocline (e.g., Myrberg and Leppäranta, 2009; Wulff et al., 2001). The drainage basin of the Baltic Sea is densely populated and the anthropogenic influence on water quality is significant in the region. The excess of nutrients in the sea water is evident, and has led to the process of eutrophication with its multiple implications to chemical and biological environments (HELCOM, 2009).

For evaluating the water quality in the Baltic Sea, expedient water quality monitoring is needed. This is an especially challenging task in the Archipelago Sea, SW Finland, where the topographically fragmented coastal waters set demanding prerequisites for the design of monitoring programs (Erkkilä and Kalliola, 2007). The area is located between two large subbasins of the Baltic Sea, and its waters originate from adjacent sea basins and from the mainland. Thus, the Archipelago Sea has both sill and transitional estuary characteristics, and a spatially dense network of observed stations is necessary.

There are multiple monitoring programs ongoing in the area. The most extensive of them includes 60 sites in the eastern half of the Archipelago Sea (Fig. 1). The sampling of these sites is limited to three annual field campaigns in July–August. However, even these field campaigns incur considerable economic costs due to causes such as the required human labor of visiting the sites and analyzing the samples. It is therefore of interest to find out whether a smaller set of sites would be sufficient for monitoring water quality. More specifically, the primary focus of this article is to find a subset of the 60 sites located in the eastern Archipelago Sea such that the observations done at the rest of the sites can be estimated with a sufficient accuracy using the observations of the sites in this subset.

* Corresponding author. Tel.: +358 2 333 8658; fax: +358 2 333 8600.
E-mail address: mika.murtojarvi@utu.fi (M. Murtojärvi).

0098-3004/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2011.01.006

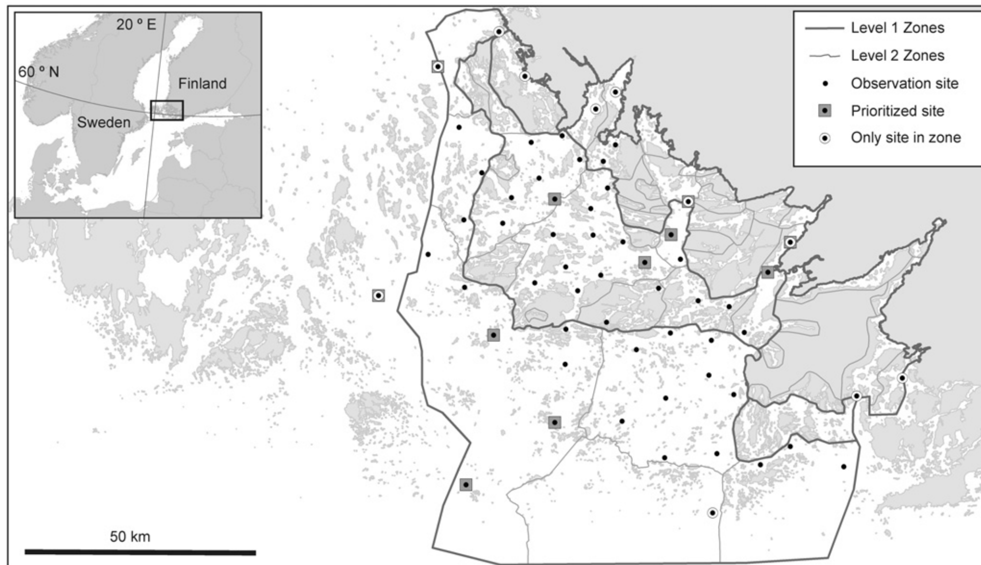

Fig. 1. Observational network. Thick lines enclose the level 1 zones (inner, middle, and outer archipelago) and the thin lines the level 2 zones within them. Prioritized sites and the sites which are the only sites in their level 2 zones are also marked.

The observations of a site *s* are modeled by fitting a multivariate linear model (see, e.g., Mardia et al., 1979) based on the observations of the other sites so that the modeled values at site *s* match the observed ones as well as possible. The goodness of fit is used as an indication of whether the removal of the site is feasible. Only a certain number of the nearest neighbors of a site are used in the models. This choice is based on the assumption that nearby sites tend to behave more similarly to each other than sites that are far apart. These regression models are then used in the network pruning process.

In addition to the fitted statistical models, other factors must also be considered when evaluating an observational network and the significance of a site. First, the network must cover the monitored area with a reasonable spatial density. To attain an evenly distributed network, we use hierarchical typology of the transitional and coastal waters. The typology is based on physical and chemical properties such as salinity and wave exposure (Anonymous, 2003b). The typology used here has two levels; first the Archipelago Sea was divided to inner, middle, and outer zones (i.e., level 1 zones) for the purposes of the Water Framework Directive (Vuori et al., 2006). The typology has been further refined into smaller regions (level 2 zones) by the local environmental authorities (Anonymous, 2009; Rautio et al., 2008). In the present study it is a precondition that if there are sites within a level 2 zone, at least one of them must remain. The second factor to consider is that some of the sites should not be removed. For example, a site might already have long and coherent time series and its sampling should continue.

Although the primary focus of this article is the method for finding a good subset of the observation sites, there are also two dataset-specific preprocessing steps that are described for completeness. Some observations are missing and some are believed to be incorrect. An incorrect observation may result from an erroneous sampling and a contaminated water sample or from a typing error. Because incorrect observations may have an effect on the accuracy of the obtained statistical models, an attempt is made to detect and remove them before the network optimization process. Many of the *outliers* (Barnett and Lewis, 1978), i.e., observations that are significantly different from the others, can be easily recognized either manually or by using an automated procedure. A simple automated method was adopted in this study to recognize possible outliers, although consideration of whether to remove an observation was eventually done for each observation separately. The other data-related problem, missing observations, is solved by restricting the considerations to a time period when the amount of missing observations is relatively small and by using *imputation* for estimating the missing observations (King et al., 2001; Little and Rubin, 2002; Oba et al., 2003).

There are some known approaches that could be used for the network optimization problem. For example, cluster analysis (Shrestha and Kazama, 2007; Singh et al., 2005; Theodoridis and Koutroumbas, 1999) has been used in many research areas, and it could identify sets of sites whose observations behave similarly to each other. However, if the observations of a site can only be explained using the observations of several other sites, cluster analysis might fail to identify good candidate sites for removal. The problem of network optimization has also been specifically addressed in several articles (e.g., Frolov et al., 2008; Lin et al., 2010; Sakov and Oke, 2008). In many cases, network optimization is based on the results of dynamic simulation that takes into account the known physical and chemical processes. The goal is to choose the observation sites in such a way that the simulated field of interest can be reproduced as well as possible given the constraints (e.g., the allowed number of sites) of the optimization. In these approaches, also the observations of the selected sites are obtained using the simulation. A similar network construction has also been done using observed values instead of simulation (Sakov and Oke, 2008). In principle, these methods could be used for the purposes of this article, but the problem is the lack of both a high-resolution observational dataset and a simulation model that has been validated in the target area.

While the proposed method is discussed in the context of optimizing a specific observational network for water quality monitoring, the approach is of a general nature and it is expected to be applicable also in other research areas where one has to plan on making measurements with minimal cost and good accuracy. The pruning algorithms are implemented in R language (R Development Core Team, 2010).

2. Materials and methods

2.1. Material

The water quality monitoring started in the Archipelago Sea by the 1960s with few annual observations. Most frequent sampling

was done in the years 2002–2008 when the data were collected three times annually, usually at weeks 29, 32, and 35 (in July–August). In most cases all 60 observation sites are sampled within 4 days. The years 2002–2008 are considered here and as a result the data consist of 21 more or less simultaneous samplings from 60 sites (Fig. 1). Four most commonly analyzed water properties with coherent time series were selected. *Chlorophyll-a* is one of the photosynthetic pigments of phytoplankton and its content is used, alongside with the total concentrations of two main nutrients, *nitrogen* (N_{tot}) and *phosphorus* (P_{tot}), as a measure of eutrophication level. *Secchi depth*, i.e., a measure of visibility in the water, is routinely measured in the field. The N_{tot} samples are taken from the depth of 1 m and they are analyzed according to the standard SFS-EN ISO 11905-1 (Finnish Standards Association (SFS), 1998). Also P_{tot} samples are collected from 1 m and they are analyzed according to an in-house analysis method, which is based on the Lachat QuickChem method 10-115-01-4-F. The chlorophyll-*a* samples are taken from the surface layer as a combined sample, where the lowest limit of the subsamples is chosen as twice the Secchi depth. Chlorophyll is analyzed spectrophotometrically from an ethanol extract according to the standard SFS 5772 (Finnish Standard Association (SFS), 1993). The laboratory analyses are made by Water Protection Association of South West Finland. The Secchi depths are determined visually by the white cap (diameter 100 mm) of a Limnos water sampler. The number of observations is altogether 5040, out of which about 4.5% (Table 1) are missing.

2.2. Methods

Solving the network optimization problem requires a model for the observations and a site pruning algorithm. Two preprocessing steps, outlier elimination and imputation, are used for dealing with incorrect and missing observations (Fig. 2). These steps are followed by two intermediate steps: (1) choosing the number of neighbors for the statistical models and (2) choosing a heuristic to use as the site pruning method. Finally, the network is pruned using the parameters determined in these two steps. The pruning process is repeated with several different numbers of sites to remove.

The workflow is illustrated diagrammatically in Fig. 3. The preprocessing steps (outlier elimination and imputation) are separate from the other steps of the pruning process, and they can easily be replaced with other methods or even omitted. Omitting these steps should only be considered with datasets with very few missing or incorrect values. The remaining three steps, on the other hand, are more tightly related to the pruning process. As was stated earlier, the observations of the nearest neighbors of a site are used for fitting a multivariate linear model to the observations of the site in question. The number of

Table 1
Properties of the water quality quantities in the time period 2002–2008.

Variable	Unit	Range	Mean	Missing	Outliers
Secchi	m	0.6–6.7	2.8	59 (4.7%)	2 (0.17%)
N_{tot}	$\mu g\,L^{-1}$	120–650	350	55 (4.4%)	5 (0.41%)
P_{tot}	$\mu g\,L^{-1}$	10–50	23	57 (4.5%)	5 (0.42%)
Chlo-*a*	$\mu g\,L^{-1}$	1.1–15	4.3	58 (4.6%)	1 (0.08%)

The total number of observations of each quantity is 1260.

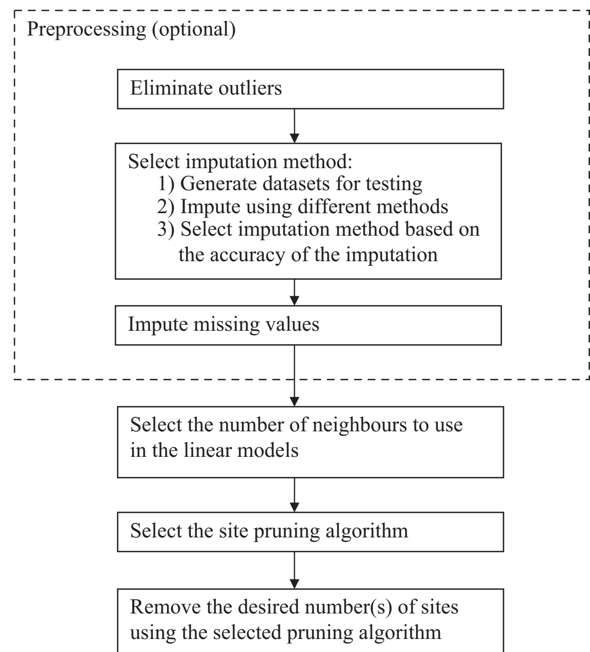

Fig. 3. The workflow of the network optimization process, shown on a coarse level.

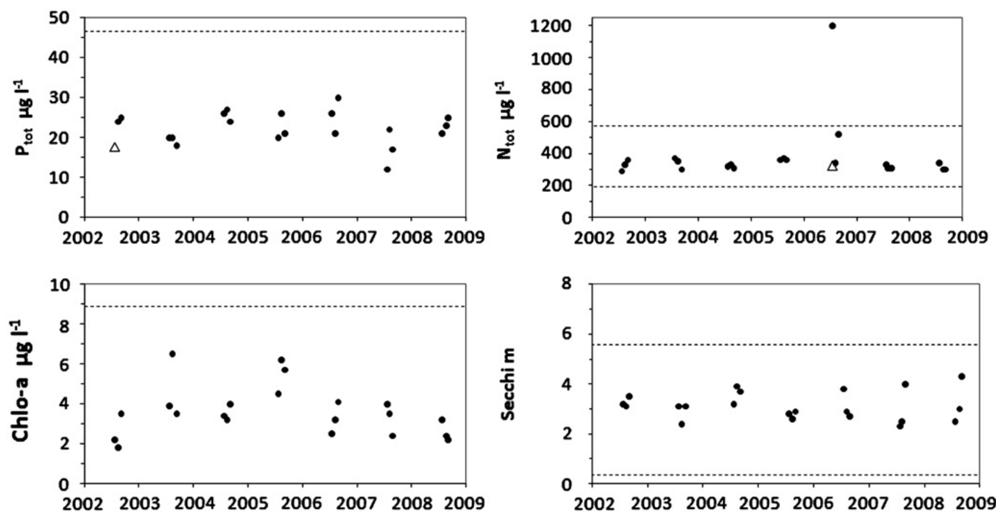

Fig. 2. Observations of four quantities at a sample site in the time period 2002–2008. One observation (P_{tot} in 2002) is missing, and one outlier (N_{tot} in 2006) is visible. The horizontal lines show the limits of the outlier detection procedure for this particular site; values that are outside this range are subjected to further examination. Imputed values are shown by triangles.

neighbors to use is therefore a parameter of the pruning process that must be chosen. Furthermore, the heuristics used in the pruning process are not guaranteed to give optimal results with respect to any error metric. Therefore, tests with three different heuristics are performed in order to select the site pruning algorithm. The actual pruning is then carried out using the values for the parameters that were chosen in the previous steps.

2.2.1. Outlier elimination

The outlier elimination is based on a method that is sometimes called Wright's procedure (Barnett and Lewis, 1978). The method rejects samples that deviate from the sample mean by more than a specified number of sample standard deviations. Because the mean values of the observations differ among the sites, the procedure is modified slightly. The observations are compared to the site mean of the quantity. Also, site-dependent (zone-wise) standard deviation is used, where the zones correspond to the prespecified level 1 zones. Sample q_i is thus considered a possible outlier if

$$|q_i - \overline{q}_{site}| \geq c \cdot s_{q,zone(site)}, \tag{1}$$

where $\overline{q}_{site}$ is the sample mean of the quantity q at the site and $s_{q,zone(site)}$ is the sample standard deviation of the observations of this quantity within the level 1 zone of the site. Constant c should be chosen so that most incorrect observations but only a small amount of correct observations are discarded. However, it may be impossible to verify which observations are incorrect, and the choice of the constant c is then at least partially guided by subjective means such as visual inspection of the data. Another guiding principle is that the method should only reject a small number of samples as it was expected that only a few observations out of 1000 might be erroneous. The same value $c=3$ is used for all sites and quantities, but due to differences in the zone-wise standard deviations the acceptance region will be wider for some sites and quantities than for others. For the purposes of the present study the simple outlier procedure worked well enough; the number of detected outliers remained low and they were clearly distinguished. When datasets with a greater fraction of incorrect values are used, one should study the literature (e.g., Barnett and Lewis, 1978) for more advanced outlier detection procedures.

2.2.2. Imputation

Although only 4.5% out of all the observations were missing before outlier elimination (Table 1), the problem of missing data cannot be ignored, because multilinear models (see Section 2.2.3) are used during the network pruning. The model used in the pruning process represents the observations of a quantity at a given site as a linear combination of the observations of the same quantity at several nearest neighbors of the site. In such a model, the modeled value of a quantity is missing whenever the observation of the same quantity at any neighboring site is missing. Depending on the size of the neighborhood a significant fraction of the modeled values could have a missing value, and the number of observations of each site (21 for each quantity) is relatively low even without the problem of missing values.

To produce a complete data matrix, the missing values are imputed; i.e., they are replaced with estimated values. For the theory of missing value imputation, see Little and Rubin (2002) and for applications, see Horton and Kleinman (2007), King et al. (2001), Oba et al. (2003), and Troyanskaya et al. (2001). The imputation methods are based on the assumption that there are dependencies between the observed quantities. In the water quality dataset the intervariable correlations were indeed statistically significant, although the absolute values of the correlation coefficients were rather small: the absolute values of Pearson's correlations between the four quantities ranged from 0.16 to 0.53 before eliminating outliers. Missing data were ignored in computing the correlations. Another characteristic of the dataset is that in many cases all observations of a particular site and time are unavailable, so the imputation methods cannot be expected to perform very well. Furthermore, the imputation methods evaluated in this study have been developed for different application areas, and it is necessary to test how they perform compared to simpler approaches with the water quality data.

Three simple (ad hoc) and two advanced imputation methods were evaluated in the present study. The first simple method replaces each missing value with the mean of all observed values of the quantity in question. The second method uses site means instead of global means. Also in this method the observations of all years are used for computing the mean values. A further refinement is to take also time into account: only the observations of the same year and site are used for computing the means.

The more advanced imputation packages are BPCA impute (Oba et al., 2003) and Amelia II (Honaker et al., 2010). BPCA has been successfully used in bioinformatics. Amelia II has been designed for social science applications, and it can handle time-series cross-sectional (TSCS) data. The concept of TSCS data has been defined, e.g., in the following way: "Time-series cross-section data are characterized by having repeated observations on fixed units, such as states or nations" (Beck and Katz, 1995). In the water quality dataset, the units correspond to the different observation sites. Amelia II (unlike BPCA) is a multiple imputation package, i.e., it generates several estimates for the same missing values, reflecting the uncertainty in the imputation.

2.2.3. Statistical models

Multivariate linear models (Mardia et al., 1979) are used for modeling the observations of the potential sites to be removed. The observation number i of a quantity q at site s is modeled as

$$q_{s,i} = \left(\alpha_{s,q} + \sum_{s' \in knn(s)} \beta_{s',q} q_{s',i} \right) + u_{s,q,i}, \tag{2}$$

where $\alpha_{s,q}$ is a constant term for the site and quantity in question, $knn(s)$ is the set of k nearest neighbor sites of s, and $u_{s,q,i}$ is the residual, i.e., the error in estimating the observation using the linear model. The set $knn(s)$ contains the sites that have not been removed from the network and have smaller distance to site s than any other sites that have not been removed from the network. The coefficients $\alpha_{s,q}$ and $\beta_{s',q}$ are determined for each quantity so that the sum of squares of the residuals, where summation is over all observations i, is minimal (Mardia et al., 1979). Fitting models of this kind can be done using standard statistical software.

For specifying a cost function for the site pruning process, the estimation error for a single site (s) and time (i) is first defined as the *Mahalanobis distance* (Mardia et al., 1979) given by the formula

$$e_{s,i} = ((\mathbf{q}'_{s,i} - \mathbf{q}_{s,i})^{\mathbf{T}} \mathbf{\Sigma}^{-1} (\mathbf{q}'_{s,i} - \mathbf{q}_{s,i}))^{1/2}, \tag{3}$$

where $\mathbf{q}_{s,i}$ is the vector containing the ith observations of all quantities at site s and $\mathbf{q}'_{s,i}$ is the vector of linear estimates of the quantities, as given by (2) without the residual term $u_{s,q,i}$. In the case of the water quality dataset these vectors contain four elements, which is the number of different quantities taken into account in the modeling. Matrix $\mathbf{\Sigma}$ is the covariance matrix that contains the variances of the different quantities and their covariances. In practice, $\mathbf{\Sigma}$ is replaced by its estimate $\hat{\mathbf{\Sigma}}$, obtained based on the observed data. As in outlier elimination, different covariance matrices are used for the three different zones. Hence, larger differences between the observed and the modeled values

are allowed in the zones where the observations vary more. The total average estimation error of the observations of site s is computed as

$$e_s = \left(\sum_i e_{s,i}\right) \Big/ n_s, \tag{4}$$

where n_s is the total number of observations per quantity at site s.

The goodness of the result of the optimization process is defined as a cost function related to the entire network. It is defined as the sum of the estimation errors e_s over all sites

$$e = \sum_{s \in del} e_s, \tag{5}$$

where *del* is the set of sites that have been removed from the network.

2.2.4. Network optimization

The goal of network optimization is to find a set of sites that can be removed so that the cost function (5) is minimal, given a specified number d_0 of sites to be removed (i.e., $|del| = d_0$). The observational network considered in this work contains originally n_s=60 sites, and there are 2^{60} subsets of this set. When the number of sites to be removed is fixed, there are $\binom{n_s}{d_0}$ different selections of d_0 sites to remove. For example, if 30 of 60 sites need to be removed, there are approximately 1.2×10^{17} subsets to evaluate if the optimal solution is to be found. This number is too great to test in an acceptable amount of time, and resorting to suboptimal heuristic solutions is justified.

The problem of selecting subsets of a given set has been studied, e.g., in the context of network optimization (Frolov et al., 2008) and in the field of pattern recognition (Theodoridis and Koutroumbas, 1999). One simple and efficient method is called *sequential backward selection* in the field of pattern recognition, and it is also known as a delete-only optimization algorithm (Frolov et al., 2008). The method starts with the whole network and removes the site that is in some sense a good choice for removal, e.g., the site whose observations can be modeled best using the observations of other sites that are still in the network. Then, another site is removed from the remaining network, and the process is repeated until some termination criterion is met. The backward selection technique never considers moving a removed site back to the network, which limits its capability of finding good solutions. Furthermore, even if a limit is set on the allowed estimation error of a site, this limit may be exceeded when the algorithm terminates. This can happen if one or more of the nearest neighbors of a site are removed from the network after the site has been removed. In that case different sites will be used for estimating the observations of the site in the final network and in the network that existed when the site was removed. Despite these shortcomings, the three implemented heuristics are based on this scheme of removing sites iteratively according to a cost function. It should be noted that an exchange-type algorithm that both deletes and adds sites has been shown to perform slightly better than the delete-only algorithm (Frolov et al., 2008). However, it was observed in Frolov et al. (2008) that the difference between the results of the delete-only and exchange-type algorithms is rather small, and the exchange-type algorithm must be run several times using different randomly initialized networks as starting points in order to achieve better results than the delete-only algorithm.

2.2.4.1. Greedy pruning with local costs. The first algorithm (referred to as *pruning_local_taboo*) was originally designed for removing a small number of sites from the observational network. At each iteration it removes from the set of removal candidates the site with the smallest estimation error (4). Because removing one of the neighbors of a removed site s might change the estimation error of s, all neighbors of s are also removed from the set of removal candidates; i.e., they are added to a *taboo list*. The procedure is repeated until no more sites can be removed. This happens when the set of removal candidates is empty, specified number of sites has been removed, or when the observations of none of the remaining sites can be estimated well enough. The latter condition requires comparing the estimation errors of the sites to a specified threshold θ. If the number of sites to remove is given, one may search for the smallest value of θ that allows removing the given number of sites. Another possibility is to use an infinite value for θ and let the algorithm prune the desired number of sites. In the case of the algorithm *pruning_local_taboo* using an infinite θ can lead to the (undesirable) removal of sites with high estimation errors.

If k neighbors are used for estimating the observations of a site and the total number of sites is n_s, at most $\lceil n_s/(k+1) \rceil$ sites can be removed by the above method. For removing a somewhat greater number of sites, *pruning_local_taboo* repeats the procedure by removing all sites from the taboo list and continuing the execution of the algorithm. In this case the estimation errors of removed sites may exceed θ. A postprocessing step moves the sites that can no longer be estimated well enough back to the observational network. Since the idea behind the taboo list is largely lost when several sites must be removed, another variation of this algorithm is also tested. This second algorithm (referred to as *pruning_local_notaboo*) is similar to the first one, but it does not use the taboo list. This increases the number of sites from which the algorithm may choose the most promising one for removal.

2.2.4.2. Greedy pruning with total costs. The two algorithms outlined above may not make even a locally optimal choice with respect to the cost function (5). The greedy choices only consider the estimation error of each site to be removed, but removing a site s affects the estimation errors of those sites that have already been removed and contain s among k nearest neighbors. Therefore, as a third algorithm, a method that removes sites based on the cost function (5) is tested (referred to as *pruning_total*). It should be noted that this criterion does not guarantee that the estimation error of a removed site is small. One site gives only a relatively small contribution to the total estimation error, and the estimation errors of several other sites may be affected when site s is removed. Again, a threshold θ may be used to prevent large estimation errors. However, the use of thresholds has significant drawbacks. One is the difficulty of selecting a suitable threshold value, and the other is that once a threshold is given, it may not be possible to remove the desired number of sites.

3. Results

3.1. Outlier elimination

Based on visual inspection, value c=3 was selected for the outlier elimination threshold in Eq. (1); i.e., observations deviating by more than 3 standard deviations from the site mean were considered as possible outliers. The outlier elimination process was done separately for all four water quality quantities included in the analysis (see Table 1). The values of chlorophyll-*a* were transformed by taking (natural) logarithms before outlier elimination. The number of detected outliers was rather low (Table 1), and the detected outliers were also considered questionable using a subjective evaluation. All of the detected outliers were removed before proceeding to the next preprocessing step, imputation.

3.2. Imputation

Because the real values of the missing observations are unknown, two sets of artificial data matrices were generated for testing the imputation methods. Both sets contained five matrices in order to obtain a reasonably accurate estimate of the imputation errors. The matrices were obtained by randomly removing from the water quality data 5% of the nonmissing observations of all four quantities P_{tot}, N_{tot}, chlorophyll-*a*, and Secchi. To reflect the patterns of the full dataset, it was required that at least two of the three observations of any combination of site, year, and quantity remain in the dataset. In the first set of five artificial matrices (*dataset I*), the removal was done independently for the different quantities. However, in the original dataset it was typical that all observations of a particular site and time are missing; i.e., the site was not visited at all in certain occasions. To test the accuracy of the imputation methods in such cases, another set of five artificial data matrices was constructed (*dataset A*) so that all water quality parameters were removed for the randomly selected sites and times. Also in this case 5% of the observations were removed.

Experimentation was done with the advanced imputation methods to find the parameters that lead to the best imputation results (see Table 2). In all tests the observation times were used as imputation parameters, in addition to the water quality quantities. In some of the tests also the coordinates of the sites were included. With BPCA these additional variables were handled in the same way as any other variables. With Amelia II additional tests were made where the data were considered as TSCS data with the site IDs separating the sites from each other, and the time points of the observations defined the time series nature of the data. The imputation errors in comparison to the original removed values (RMSE, root mean square error) were determined for each of the five imputation methods and both sets of five artificial data matrices. For Amelia II the means of 20 imputations were compared to the observed values.

The root mean square errors (RMSE) of the five different imputation methods are summarized in Table 2 for *dataset I* and *dataset A*. For *dataset I*, i.e., when observations of different quantities were removed independently, the advanced imputation methods (BPCA and Amelia II) gave better or similar results compared to the site mean imputation, which, in turn, performed better than the mean imputation based on both site and year of observation. Simple mean imputation performed worst. With Amelia II it was also possible to handle the dataset as TSCS data, and it improved the results slightly with *dataset I*. When all observations of a particular time and site were removed simultaneously (*dataset A*), advanced imputation methods performed mostly worse than site mean imputation. The exception was Amelia II with the option of treating the dataset as TSCS data, which gave similar results to the site mean imputation. One may also note that with *dataset A* the advanced imputation methods that use only water quality quantities and time perform similarly to the simple mean imputation. This could be expected, because there are no clear trends in the water quality quantities in the observed time period. The intervariable correlations are of no use, either, because whenever the observation of one quantity is missing in this dataset, the observations of the other quantities are missing as well. Including coordinates improved the imputation results, but site mean imputation performed still better than the advanced imputation methods. Treating the dataset as TSCS data allows Amelia II to determine different means for the observations of the different sites. In fact, Amelia II allows even trends in time, and they can be different for different sites. This option, however, did not seem to have much effect on the results.

On the basis of these results, two imputation methods perform better than the others. One is Amelia II with the TSCS option, and the other is a combination of BPCA and site mean imputation. In the combined imputation method missing observations are imputed using BPCA when some observations are available and site mean imputation is used when all observations are missing for a particular site and time. The main advantage of the combined imputation method is ease of use: any analysis only needs to be performed once, because the method generates only one imputed dataset. There is, however, also a significant drawback: when only one imputed dataset is used, it is not possible to study whether reasonable but different imputations would lead to different results. For this reason, the multiple imputation package Amelia II is used in the final network pruning process, using five different imputations. Amelia II could also be used for the intermediate steps (selecting the pruning heuristic and the number of neighbors to use in the models), but to reduce the amount of work the combined single-imputation method is used here. The dataset used for the imputations used in the remaining part of this work is the original dataset with the outliers removed.

3.3. Network optimization

3.3.1. Choosing the number of neighbors

The dataset used in all these tests was obtained using the combined imputation method (BPCA and site mean). A series of experiments was conducted. (1) To determine whether the neighborhood size has an effect on the pruning results, network pruning was performed using three, five, and seven neighbors in the models. (2) To see whether the neighborhood size can be determined on the basis that the errors of the models do not change much beyond some neighborhood size, the estimation errors were determined for several sites using three different neighborhood sizes. (3) To test how the fitted models perform outside their fitting period, the sample was partitioned into a training set and a testing set. The training set was used for fitting

Table 2
Imputation errors (RMSE) of three ad hoc methods (mean, mean (site), and mean (site, year)) and two advanced imputation methods (BPCA, Amelia II).

	Mean	Mean (site)	Mean (site, year)	BPCA (1)	BPCA (2)	Amelia (1)	Amelia (2)	Amelia (3)
Secchi (I)	1.09	0.65	0.73	0.92	0.67	0.93	0.68	0.59
N_{tot} (I)	54.06	43.61	43.50	34.82	34.01	35.49	33.09	33.27
P_{tot} (I)	5.45	4.41	4.58	4.43	4.39	4.54	4.51	4.07
Chlo-*a* (I)	2.21	1.76	2.07	1.36	1.37	1.41	1.40	1.44
Secchi (A)	1.10	0.70	0.74	1.10	0.80	1.11	0.82	0.70
N_{tot} (A)	54.97	43.63	44.30	53.95	48.22	54.76	49.07	44.08
P_{tot} (A)	5.88	4.77	5.11	5.88	5.75	5.90	5.86	4.89
Chlo-*a* (A)	2.10	1.78	2.01	2.06	1.95	2.07	2.03	1.81

In BPCA (1) and Amelia II (1) water quality quantities and time were taken into account, in BPCA (2) and Amelia II (2) also the coordinates of the sites were considered. In Amelia II (3) the time-series cross-sectional (TSCS) nature of the data was taken into account. Rows marked with (I) correspond to *dataset I*, and the rest to *dataset A*. In *dataset I* observations were removed independently of each other, in *dataset A* all observations of the selected sites and times were removed.

the models and the testing set for evaluating the estimation errors. As before, the test was repeated using three, five, and seven neighbors in the models. Several different partitions of the data into the training set and the testing set were also used.

In the first test, the network pruning was performed using the heuristic *pruning_total*. When 20 sites were removed, 15 of the removed sites were the same using five or seven neighbors. The same was true with three or seven neighbors. When using three or five neighbors, the results agreed in 14 cases out of 20. Thus, the number of neighbors has some effect on the results of the pruning process, and further tests are required to determine the number of neighbors to use in the models.

During the second test it was observed that the observations of some sites were modeled surprisingly well using only three neighbors, but some observations of the removed sites were not modeled well even with seven neighbors. More specifically, the value of the cost function (Eq. (4)) was about 0.62 for the site that was estimated best using three neighbors, whereas the cost was 1.17 for the site that was estimated worst using seven neighbors. For the same sites, increasing the number of neighbors decreased the cost (4), as can be expected in multivariate regression (note that a smaller neighborhood of a site is a strict subset of a larger neighborhood of the same site). For the tested neighborhood sizes, one could not observe any particular neighborhood size beyond which the costs (4) no longer change significantly. The results of the test were thus inconclusive.

In the third test it was observed how much the modeled values deviate from the reality after the time period that is used for fitting the models. This is relevant because when the monitoring of a site has been stopped, only modeled values for that site are available. In this test only a part of the data are used for fitting the models (i.e., as a training set), and the rest of the data (the testing set) are used for evaluating the error (4). In the first test case the data were divided approximately in half: the first 11 observations of each site and quantity were used for fitting the models and the remaining 10 were used for evaluating the fit. The smallest and largest values of the function (4) were determined using different neighborhood sizes (3, 5, and 7 sites). The smallest and largest errors increased with increasing neighborhood size, in contrast to the case when the same data were used for fitting and evaluating the models. The most dramatic increase was in the largest estimation error: when the number of neighbors was increased from three to seven, the largest estimation error (4) increased from about 2.8 to 7.5. The smallest error increased from 0.9 to 1.3. Further testing was done by computing the sum of the errors (4) over all sites using different numbers of neighbors in the models. When the number of neighbors was increased, this sum increased: it was 88, 107, and 147 for three, five, and seven neighbors, respectively. Similar tests were also performed using a larger training set. In that case, the influence of the size of the neighborhood on the accuracy of the models outside their fitting periods was smaller, but for all sizes of the training set using seven neighbors led to the worst results outside the fitting periods. Using three neighbors outperformed using five neighbors, except in the case where only one observation (the latest one) of each site and quantity was used for evaluating the function (4).

Based on these results, using a relatively small neighborhood is best justified: the models that use a large number of neighbors only behave well in the time periods for which they were fitted. In the case of the dataset used in this study, the network pruning process is carried out twice, using three and five neighbors in the models.

3.3.2. Comparing the pruning heuristics

None of the heuristics described in Section 2.2.4 are guaranteed to give optimal results with respect to error measures (4) or (5), even if *pruning_total* bases its operation on function (5). Thus, the heuristics are compared to see which one of them performs best. In these tests, five neighbors are used in the linear models, and all observations are used for fitting the linear models and evaluating functions (4) and (5). The dataset is the same as in Section 3.3.1, i.e., obtained using the combined imputation method.

Test 1. When θ was the average of the smallest and largest estimation errors (4) of the sites, *pruning_local_taboo* and *pruning_local_notaboo* removed the same 18 sites, although in a different order. The heuristic *pruning_total* removed 15 sites, 14 of which were the same as with the two other heuristics. Because *pruning_total* removed a different number of sites compared to the other heuristics, the total estimation errors (5) are not comparable to each other. Therefore, as another test, the same number of sites (18) was removed using *pruning_total* without using a threshold value. The total estimation error (5) of the obtained solution was lower than those of the solutions obtained using the other heuristics with threshold values, but the difference was very small (less than 2%). More surprisingly, the largest site-specific estimation error (4) was also almost the same. Although the estimation errors were similar, only 15 of the 18 removed sites were the same as in the solution obtained using the other two heuristics. All heuristics were fast enough to be practical, although *pruning_total* was much slower than the other two heuristics.

Test 2. As another test, 20 sites were removed without using thresholds, and 16 of the removed sites were the same for *pruning_local_notaboo* and *pruning_total*. The results for *pruning_local_taboo* agreed with both *pruning_local_notaboo* and *pruning_total* in 15 of the 20 sites. The total estimation errors (5) for the heuristics *pruning_local_taboo*, *pruning_local_notaboo*, and *pruning_total*, were 17.1, 16.5, and 15.9, respectively. The largest estimation errors (4) were virtually the same for *pruning_local_notaboo* and *pruning_total*, whereas for *pruning_local_taboo* the error was somewhat higher (about 1.15 vs. 0.99). Removing sites randomly led to worse results than any of the heuristics, but when the best random network selection out of 1000 was evaluated, error measures (4) and (5) were very similar to those of the worst performing heuristic, *pruning_local_taboo*. The running time of the repeated random selection was higher than that of any tested heuristics.

As a conclusion, the results obtained using the three different heuristics are quite similar in terms of which sites are removed. Nevertheless, some differences in the values of the objective functions (4) and (5) are observed. With respect to these objective functions, the local heuristic without the taboo list (*pruning_local_notaboo*) is a reasonable pruning method even when the total estimation error is considered. With the taboo list (*pruning_local_taboo*) the local heuristic performs well if a threshold value for the estimation errors can be specified, but without a threshold it performs worse than the other heuristics. With larger datasets the low execution times of the local heuristics may be an important benefit when compared to *pruning_total*. With the present water quality dataset the running times of all heuristics are acceptable. The network optimization process is performed using *pruning_total* because of its best performance according to cost function (5). Even the site-specific estimation errors (4) of *pruning_total* were competitive with the heuristic *pruning_local_notaboo* that performed best with respect to this criterion. Hence, although in principle the heuristic *pruning_total* might have the undesirable property of removing sites with rather high estimation errors, in practice such behavior is rare with the water quality dataset.

3.3.3. Network pruning

The pruning process was carried out using three and five neighbors of each site in the linear models and removing 10, 20,

and 30 sites from the network (leaving 50, 40, and 30). The heuristic *pruning_total* without a threshold was used. To gain information on how sensitive the results are to the imputation of missing data, the process was repeated using five different imputed datasets obtained using Amelia II. All imputed datasets were based on the water quality dataset with outliers removed. This gives five possibly different sets of removed sites for each number of neighbors and number of removed sites.

The modeled and measured values of the four quantities are shown in Fig. 4 for one of the removed sites in the case where 30 sites were removed from the network and five neighbors were used in the models. While considerable differences between the observed and the modeled values can be seen in some cases, especially in Secchi depths, most of the modeled values are close to the observations for this particular site. Fig. 5 shows a scatter plot of the results of the linear models. All observed and modeled values of the 30 removed sites and four quantities are included in the figure. One can observe that there can be considerable differences between the modeled and the observed values and that the models tend to somewhat reduce the range of the values, i.e., the modeled values corresponding to the most extreme observations often lie closer to the mean of the quantity than the observed values do. Nevertheless, most values of the quantities are reproduced reasonably well by the models.

A simple way to test the similarity of the results is to count how many times each site appears in the different removal sets (*del*). Ideally the same sites would occur in all removal sets, reflecting that the uncertainty caused by the imputation is small. However, this is not the case (Table 3). When 30 sites are removed from the network, the agreement on which sites to remove is relatively good, the same 22 or 23 sites being removed regardless of which imputed dataset is used, depending on the neighborhood size (3 or 5) in linear models. When 20 sites are removed, the agreement between different imputations is worse. When 10 sites are removed, the removal sets are rather dissimilar with three neighbors in the models but more similar with five neighbors. The good agreement between the removal sets, when a large number of sites are removed, is partially explained by the constraints of the optimization: at least one site in each pre-specified area must remain in the network, and some of the sites must not be removed. On the other hand, the poor agreement of the result sets in other cases does not tell whether the obtained solutions are poor; several different networks may be good with respect to function (5). It was found (data not shown) that the network obtained using one imputed dataset was suboptimal with respect to the cost function (5) when evaluated using another imputed dataset, but in all 15 tested cases the result was clearly better than the best of 1000 randomly pruned

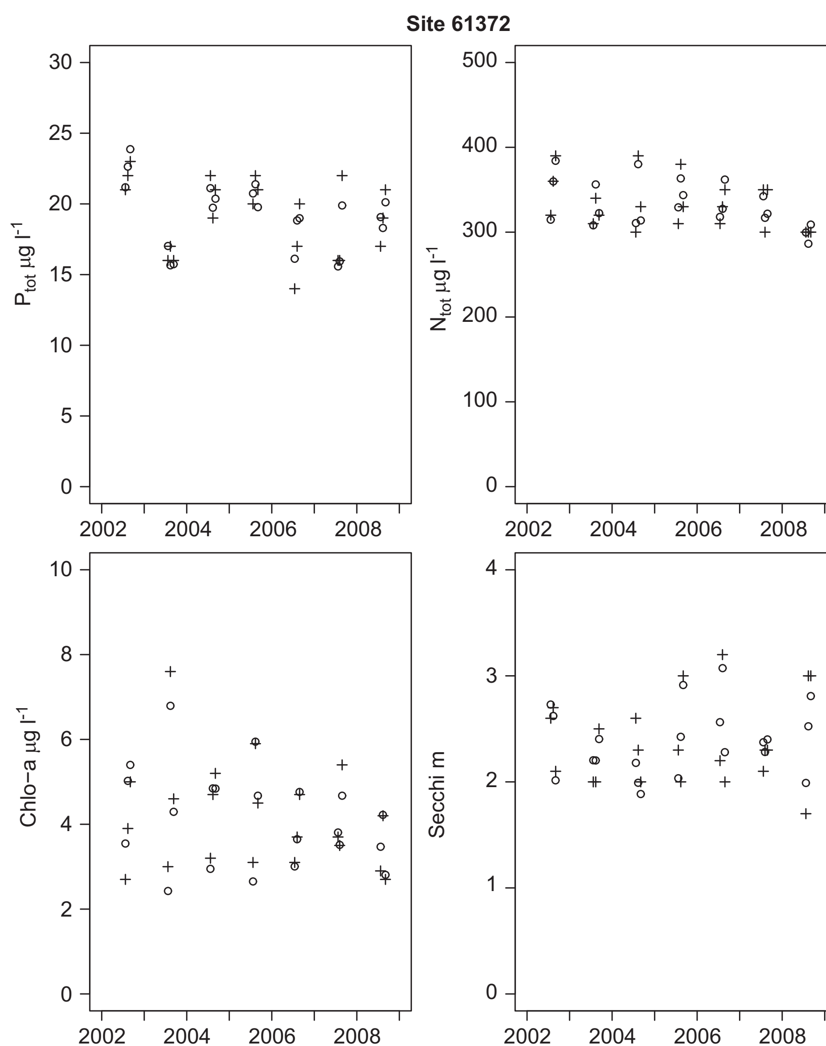

Fig. 4. The modeled and observed values of the four quantities at one particular removed site, when 30 sites were removed and five neighbors were used in the models. Circles indicate the modeled values and plus signs the observed values.

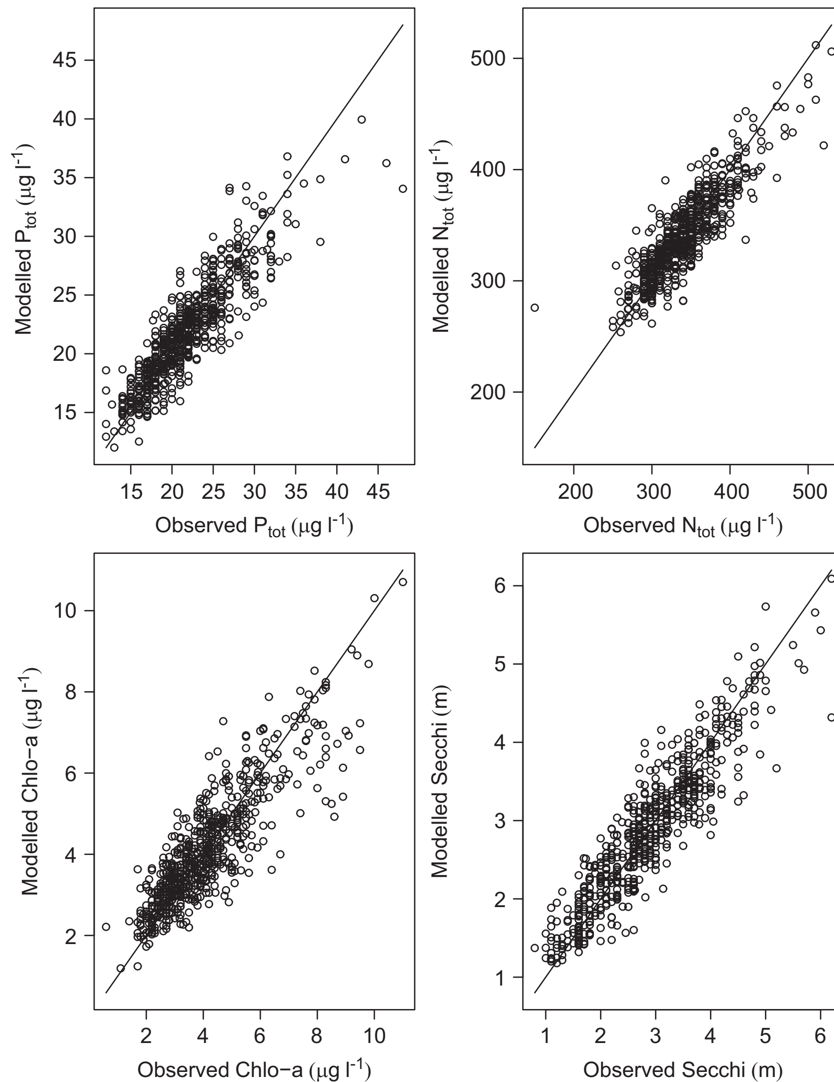

Fig. 5. The modeled and observed values of the four quantities for all 30 removed sites when five neighbors are used in the models. The horizontal coordinate of each circle gives the observed value and the vertical coordinate gives the modeled value corresponding to the same observation. The solid line represents the ideal case where the modeled and observed values are the same.

Table 3
The numbers of sites removed in exactly 0, 1, 2, 3, 4, or 5 (# occ) pruning runs using five different imputed datasets and three or five neighbors (3 or 5) in linear models.

# Occ	3 neighbors			5 neighbors		
	10 del	20 del	30 del	10 del	20 del	30 del
0	44	32	22	47	33	24
1	5	4	5	2	4	2
2	4	4	3	1	2	3
3	3	3	3	2	5	5
4	2	6	5	0	3	3
5	4	11	22	8	13	23

The numbers of deleted sites range from 10 to 30. The entries on row # occ=0 indicate how many sites were never removed by the pruning algorithm, those on row # occ=5 indicate how many sites were removed in all the pruning runs, and in general the entries in the row # occ=i indicate how many sites were removed in exactly i pruning runs out of five. Sites that were removed in 1–4 runs appear due to the differences in imputed datasets; the pruning process itself is deterministic.

networks. In this sense the uncertainty caused by the missing values is small enough so that the results of the pruning algorithms are useful.

Using five neighbors led to better agreement between the removal sets than using three neighbors. For the results with five neighbors and five repetitions of the pruning process (with five different imputed datasets), see the left column of Fig. 6. The figure shows, for every site, how many different imputed datasets led to the removal of the site in the pruning.

Missing data had an effect on the results; i.e., different imputations led to different removal sets, as is seen from the fact that there are symbols on the map that indicate that a site was removed 1–4 times. When considering the similarity of the result sets, sites that have been removed 0 or 1 times are strongest candidates to stay in the network. Similarly sites that were removed 4 or 5 times are good candidates for removal. Human judgment is emphasized for sites that were removed 2 or 3 times.

The pruning process was also executed without any preconditions on site removal; i.e., in this case there were no prioritized sites that could not be removed and no requirement that at least one site in each level 2 zone should remain in the network. Then, using three neighbors gave more coherent results, and the results are shown in the right column of Fig. 6. It is observed that the areas where the observational network was originally unnecessarily dense are similar whether the prioritization for preconditions is used or not. One should note that the removal of a site in

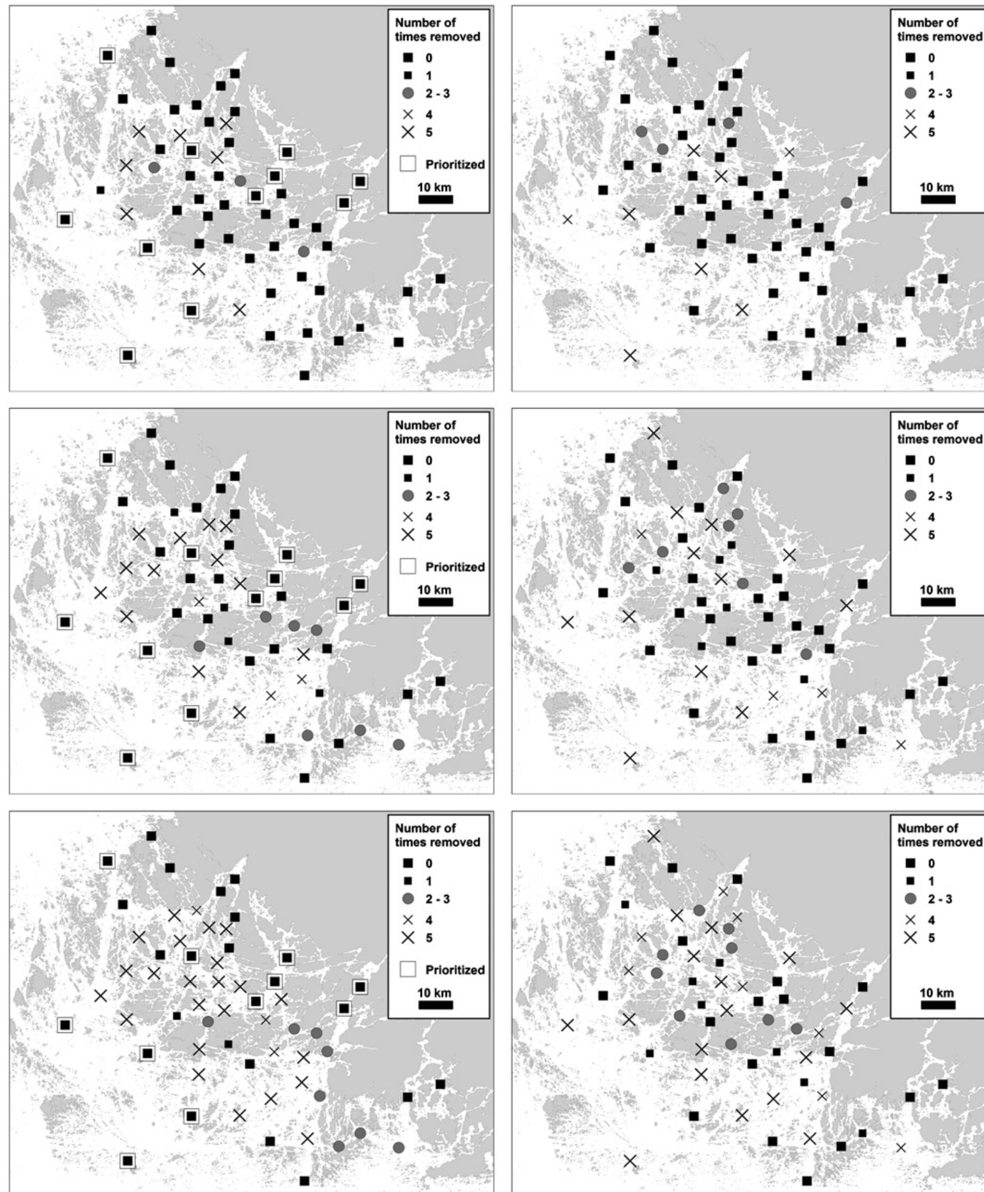

Fig. 6. Sites removed using 0, 1, 2, 3, 4, and all 5 different imputed datasets. Black rectangle (site removed 0–1 times) indicates that the site could not be estimated with good accuracy using the surrounding sites and thus should not be removed from the network. The numbers indicate how many different imputed datasets (out of 5) led to the removal of the site during the pruning process. In the uppermost, middle, and lowermost panels 30, 20, and 10 sites, respectively, were removed for each imputation. In the left column the prioritized sites, which could not be removed, are shown with a white enclosing rectangle. In the right column such prioritizing was not applied.

the pruning process does not necessarily mean that the measured values of the site are similar in magnitude with those of its neighbors. This is because the multiple regression models are allowed to add offsets and to scale the measured values when fitting a model to the observations.

4. Discussion

A method for removing sites from an observational network was presented. The method was applied for pruning an existing monitoring network. To deal with the imperfections in the observed data (missing and incorrect observations) two preprocessing steps, outlier detection and imputation, were used. A multiple imputation package (Amelia II) was used in order to see whether different imputations of the missing data lead to different pruning results. In the case of the sample observational network used in this study, the results were found to be somewhat sensitive to the differences in imputed datasets. Nevertheless, the pruning results are still useful. While different imputations led to different pruned networks, any of these solutions could be considered to be good even when evaluated using a different imputed dataset. In particular, the pruning algorithm always outperformed random removal of sites, although altogether as many as 15,000 different randomly pruned networks were evaluated. The random removal method was also time-consuming compared to the pruning algorithm due to the fact that it has to evaluate a large number of different networks to find an acceptable solution.

The pruning algorithms had the option of using a threshold value θ for preventing the algorithms from removing sites whose observations cannot be estimated well based on their neighbors. When the number of sites to remove is given, one may search for the smallest value of θ that allows the removal of the specified

number of sites. In the case of algorithms *pruning_local_notaboo* and *pruning_total*, one may also select $\theta = \infty$, because these algorithms tended to remove sites (roughly) in the order of increasing estimation error (data not shown).

There were several limitations in our method. (1) Spatial aspects were mostly ignored. The knowledge of the locations of the sites was only used for determining the nearest neighbors of each site. It could be useful to study the spatial variability using statistical methods (e.g., Schabenberger and Gotway, 2005). (2) The observations of the different sites were treated as if they were taken simultaneously, although visiting all 60 sites requires up to 4 days. Because the sites were sampled infrequently, in the present work it was not possible to study whether the violation of the assumption of simultaneous observations is relevant in practice. While a dataset with more frequent sampling would be required to properly study this feature, some practical support for the methodology is given by the fact that the models often gave acceptable results, in many cases even when different time periods were used for fitting the models and for evaluating the accuracy of the model. (3) The pruning algorithms base their operation on a relatively simple cost function, and there may well be other important factors to consider. Human consideration of the results is therefore still important in a successful pruning process. (4) The models used in the pruning process were based only on statistical fits; i.e., they did not incorporate any knowledge of the physical, chemical, and biological processes affecting water quality. Models that take such processes into account have recently been used in network optimization (e.g., Lin et al., 2010). In such studies one may, e.g., treat the results of the simulation as the "truth" that should be reproduced as well as possible given the simulated observations at a limited number of observation sites. (5) The possibility of changing the locations of the observation sites cannot be taken into account using the methodology used in this work. It was considered that the sites that remain in the network should have continuous time series based on measurements. Removing a site results in temporally and spatially fragmented data structure, which decreases the applicability of the water quality data (Erkkilä and Kalliola, 2007). In the case of the Archipelago Sea, the history and the purpose of the monitoring must be carefully considered when pruning the observational network. The goal of the monitoring is to offer information on the development of water quality with sufficient spatial accuracy, in view of the fact that the aim is to follow the general features of the water quality, not the behavior of a single bay or local subbasin.

Acknowledgments

The authors thank the South West Finland regional environment center for providing their field data and Janne Suomela for his comments. The study was partially funded by the Academy of Finland (Project 114083). The authors also thank the reviewers for their valuable suggestions.

References

Anonymous, 2000. Directive 2000/60/EC of the European parliament and of the council of 23 October 2000 establishing a framework for Community action in the field of water policy. Official Journal of the European Communities L 327, 1–73.

Anonymous, 2003a. Common implementation strategy for the Water Framework Directive (2000/60/EC). Monitoring under the Water Framework Directive. Guidance Document No. 7, Office for Official Publications of the European Communities, Luxemburg, 153 pp.

Anonymous, 2003b. Common implementation strategy for the Water Framework Directive (2000/60/EC). Transitional and coastal waters—typology, reference conditions and classification systems. Guidance Document No. 5, Office for Official Publications of the European Communities, Luxemburg, 107 pp.

Anonymous, 2009. Varsinais-Suomen pintavesien toimenpideohjelma vuoteen 2015 (action plan for the surface waters of the Southwest Finland until 2015). Southwest Finland Regional Environment Centre, Turku, 155 pp [in Finnish]. 〈http://www.ymparisto.fi/download.asp?contentid=112398&lan=fi〉 (accessed 11 August 2010).

Barnett, V., Lewis, T., 1978. Outliers in Statistical Data. Wiley, Chichester 365 pp.

Beck, N., Katz, J.N., 1995. What to do (and not to do) with time-series cross-section data. American Political Science Review 89 (3), 634–647.

Erkkilä, A., Kalliola, R., 2007. Spatial and temporal representativeness of water monitoring efforts in the Baltic Sea coast of SW Finland. Fennia 185 (2), 107–132.

Ferreira, J.G., Vale, C., Soares, C.V., Salas, F., Stacey, P.E., Bricker, S.B., Silva, M.C., Marques, J.C., 2007. Monitoring of coastal and transitional waters under the E.U. Water Framework Directive. Environmental Monitoring and Assessment 135, 195–216.

Finnish Standard Association (SFS), 1993. SFS 5772. Veden a-klorofyllipitoisuuden määrittäminen. Etanoliuutto. Spektrofotometrinen menetelmä (determination of chlorophyll a in water. Extraction with ethanol. Spectrophotometric method). Finnish Standards Association, Helsinki, 3 pp [in Finnish].

Finnish Standards Association (SFS), 1998. SFS-EN ISO 11905-1. Water quality. Determination of nitrogen. Part 1: method using oxidative digestion with peroxodisulfate (ISO 11905-1:1997). Finnish Standards Association, Helsinki, 23 pp.

Frolov, S., Baptista, A., Wilkin, M., 2008. Optimizing fixed observational assets in a coastal observatory. Continental Shelf Research 28 (19), 2644–2658.

HELCOM, 2009. Eutrophication in the Baltic Sea—an integrated thematic assessment of the effects of nutrient enrichment and eutrophication in the Baltic Sea region. In: The Baltic Sea Environment Proceedings 115B, Helsinki Commission, Helsinki, 148 pp.

Honaker, J., King, G., Blackwell, M., 2010. Amelia II: A Program for Missing Data. 〈http://gking.harvard.edu/amelia/docs/amelia.pdf〉 (accessed 11 August 2010).

Horton, N.J., Kleinman, K.P., 2007. Much ado about nothing: a comparison of missing data methods and software to fit incomplete data regression models. The American Statistician 61 (1), 79–90.

King, G., Honaker, J., Joseph, A., Scheve, K., 2001. Analyzing incomplete political science data: an alternative algorithm for multiple imputation. American Political Science Review 95 (1), 49–69.

Lin, P., Ji, R., Davis, C.S., McGillicuddy Jr., D.J., 2010. Optimizing plankton survey strategies using observing system simulation experiments. Journal of Marine Systems 82 (4), 187–194.

Little, R.J.A., Rubin, D.B., 2002. Statistical Analysis with Missing Data, second ed. John Wiley, Hoboken, NJ, 381 pp.

Mardia, K.V., Kent, J.T., Bibby, J.M., 1979. Multivariate Analysis. Academic Press, London 518 pp.

Myrberg, K., Leppäranta, M., 2009. Physical Oceanography of the Baltic Sea. Springer-Praxis Books In Geophysical Sciences, Berlin, 378 pp.

Nordic Council of Ministers, 2006. Ecological status classification of marine waters. Indicator Development and Monitoring Requirements, TemaNord 2006:582.

Oba, S., Sato, M., Takemasa, I., Monden, M., Matsubara, K., Ishii, S., 2003. A Bayesian missing value estimation method for gene expression profile data. Bioinformatics 19 (16), 2088–2096.

R Development Core Team, 2010. R: A Language and Environment for Statistical Computing. 〈http://www.r-project.org/〉 (accessed 11 August 2010).

Rautio, L.M., Siiro, P., Haldin, L., Storberg, K.-E., Nuotio, E., Westberg, V. (Eds.), 2008. Ehdotus Kokemäenjoen-Saaristomeren-Selkämeren Vesienhoitoalueen Vesienhoitosuunnitelmaksi Vuoteen 2015 (Proposal for the Water Management Plan in the Area of Kokemäki-Archipelago Sea-Bothnian Bay until 2015). Western Finland Regional Environment Centre, Vaasa 231 pp [in Finnish]. 〈http://www.ymparisto.fi/download.asp?contentid=93401&lan=FI〉 (accessed 11 August 2010).

Sakov, P., Oke, P.R., 2008. Objective array design: application to the tropical Indian Ocean. Journal of Atmospheric and Oceanic Technology 25, 794–807.

Schabenberger, O., Gotway, C.A., 2005. Statistical Methods for Spatial Data Analysis. Chapman and Hall/CRC Press, Boca Raton, Florida, 488 pp.

Shrestha, S., Kazama, F., 2007. Assessment of surface water quality using multivariate statistical techniques: a case study of the Fuji river basin, Japan. Environmental Modelling and Software 22 (4), 464–475.

Singh, K.P., Malik, A., Sinha, S., 2005. Water quality assessment and apportionment of pollution sources of Gomti river (India) using multivariate statistical techniques—a case study. Analytica Chimica Acta 538 (1–2), 355–374.

Theodoridis, S., Koutroumbas, K., 1999. Pattern Recognition. Academic Press, San Diego, 625 pp.

Troyanskaya, O., Cantor, M., Sherlock, G., Brown, P., Hastie, T., Tibshirani, R., Botstein, D., Altman, R.B., 2001. Missing value estimation methods for DNA microarrays. Bioinformatics 17 (6), 520–525.

Vuori, K.-M., Bäck, S., Hellsten, S., Karjalainen, S., Kauppila, P., Lax, H.-G., Lepistö, L., Londesborough, S., Mitikka, S., Niemelä, P., Niemi, J., Perus, J., Pietiläinen, O.-P., Pilke, A., Riihimäki, J., Rissanen, J., Tammi, J., Tolonen, K., Vehanen, T., Vuoristo, H., Westberg, V., 2006. Suomen Pintavesientyypittelyn ja Ekologisen Luokittelujärjestelmän Perusteet [the Basis for Typology and Ecological Classification of Water Bodies in Finland], Suomen Ympäristö 807. Finnish Environment Institute, Helsinki 151 pp [in Finnish]. 〈http://www.ymparisto.fi/download.asp?contentid=48905&lan=fi〉, (accessed 11 August 2010).

Wulff, F., Rahm, L., Larsson, P. (Eds.), 2001. A Systems Analysis of the Baltic Sea. Springer-Verlag, Berlin, 455 pp.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems
194. **Olli Lahdenoja**, Local Binary Patterns in Focal-Plane Processing – Analysis and Applications
195. **Annika H. Holmbom**, Visual Analytics for Behavioral and Niche Market Segmentation
196. **Sergey Ostroumov**, Agent-Based Management System for Many-Core Platforms: Rigorous Design and Efficient Implementation
197. **Espen Suenson**, How Computer Programmers Work – Understanding Software Development in Practise
198. **Tuomas Poikela**, Readout Architectures for Hybrid Pixel Detector Readout Chips
199. **Bogdan Iancu**, Quantitative Refinement of Reaction-Based Biomodels
200. **Ilkka Törmä**, Structural and Computational Existence Results for Multidimensional Subshifts
201. **Sebastian Okser**, Scalable Feature Selection Applications for Genome-Wide Association Studies of Complex Diseases
202. **Fredrik Abbors**, Model-Based Testing of Software Systems: Functionality and Performance
203. **Inna Pereverzeva**, Formal Development of Resilient Distributed Systems
204. **Mikhail Barash**, Defining Contexts in Context-Free Grammars
205. **Sepinoud Azimi**, Computational Models for and from Biology: Simple Gene Assembly and Reaction Systems
206. **Petter Sandvik**, Formal Modelling for Digital Media Distribution

207. **Jongyun Moon**, Hydrogen Sensor Application of Anodic Titanium Oxide Nanostructures
208. **Simon Holmbacka**, Energy Aware Software for Many-Core Systems
209. **Charalampos Zinoviadis**, Hierarchy and Expansiveness in Two-Dimensional Subshifts of Finite Type
210. **Mika Murtojärvi**, Efficient Algorithms for Coastal Geographic Problems

Turku Centre *for* Computer Science

http://www.tucs.fi
tucs@abo.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3372-2
ISSN 1239-1883

Mika Murtojärvi

Efficient Algorithms for Coastal Geographic Problems

Mika Murtojärvi

Efficient Algorithms for Coastal Geographic Problems