

Scalable parallel rank order filters

Citation for published version (APA):

Horst, van der, M. G., & Mak, R. H. (2007). *Scalable parallel rank order filters*. (Computer science reports; Vol. 0705). Technische Universiteit Eindhoven.

Document status and date:

Published: 01/01/2007

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Scalable Parallel Rank Order Filters

M.G. van der Horst R.H. Mak

February 26, 2007

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 2 | Problem Description | 6 |
| 2.1 | Mathematical Definition | 6 |
| 2.2 | Complexity measure | 9 |
| 3 | Problem Approach | 10 |
| 3.1 | Divide and Conquer | 10 |
| 3.2 | Divide, Re-use and Conquer | 12 |
| 3.3 | Calculation Graph | 13 |
| 3.4 | Cost of Merging Networks | 14 |
| 3.5 | Mirroring | 17 |
| 4 | Generator Graphs | 19 |
| 4.1 | Preliminaries | 19 |
| 4.2 | Canonical generator graphs | 21 |
| 4.3 | Cost calculation | 23 |
| 4.4 | Constructing generator graphs | 24 |
| 5 | Heuristic Algorithms | 27 |
| 5.1 | Preliminaries | 27 |
| 5.2 | Divide and Conquer | 28 |
| 5.3 | Recursive Tiling | 29 |
| 5.4 | Bit-pattern Tiling | 30 |
| 5.5 | Split Tiling | 32 |
| 5.6 | Overlap Recursive Mirroring | 35 |
| 5.7 | Multiple Dimensions | 41 |
| 5.8 | Maximum reuse | 42 |
| 6 | Results | 43 |
| 6.1 | Filter classes | 43 |
| 6.2 | Heuristic algorithm evaluation | 44 |
| 6.3 | Comparison with known filters | 51 |
| 6.3.1 | Window sorters | 51 |
| 6.3.2 | Median filter | 52 |
| 6.3.3 | Minimum filter | 53 |
| 6.4 | Summary | 53 |

| | | |
|----------|--------------------------------------|-----------|
| 7 | Parallel ROF Implementations | 55 |
| 7.1 | Sequential Implementation | 55 |
| 7.2 | Parallel Implementation | 58 |
| 7.2.1 | VLSI Circuit | 58 |
| 7.2.2 | SIMD Processor | 60 |
| 7.3 | Finite input streams | 62 |
| 7.4 | Philips EVP ₁₆ | 63 |
| 8 | Conclusion | 65 |
| | Bibliography | 66 |
| A | Algorithm Results | 68 |
| A.1 | One Dimensional Problems | 68 |
| A.1.1 | Window Sorting | 69 |
| A.1.2 | Median Filtering | 71 |
| A.1.3 | Minimum Filtering | 74 |
| A.2 | Two Dimensional Problems | 77 |
| A.2.1 | Window Sorting | 77 |
| A.2.2 | Median Filtering | 77 |
| A.2.3 | Minimum Filtering | 77 |
| B | Sample solutions | 79 |
| B.1 | Representation of the ROFs | 79 |
| B.2 | One Dimensional Problems | 79 |
| B.2.1 | Window Sorting | 79 |
| B.2.2 | Median Filtering | 81 |
| B.2.3 | Minimum Filtering | 82 |
| B.3 | Two Dimensional Problems | 84 |
| B.3.1 | Window Sorting | 84 |
| B.3.2 | Median Filtering | 85 |
| B.3.3 | Minimum Filtering | 87 |

Chapter 1

Introduction

Rank order filters are non-linear filters used in a wide range of applications. The median filter is perhaps the most common of the rank order filters. It can be used to remove noise from a signal while preserving edge information (see Figure 1.1), which is something that linear filters cannot do. This makes the median filter a popular filter in speech and image processing applications.

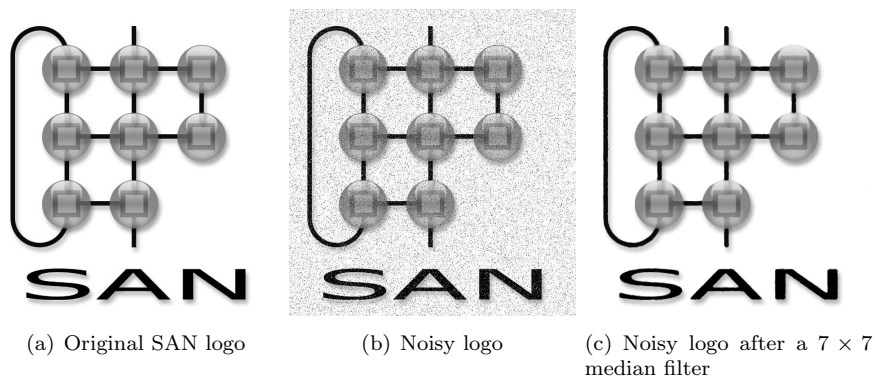


Figure 1.1: Median filtering

Rank order filters work by sorting a window of inputs and selecting the input with a certain rank as the output. In the case of the median filter the element with the middle rank is selected, hence the name (although it is called a speckle filter in some applications, after the type of noise it removes). Other common rank order filters are the minimum and maximum filter, selecting respectively the input with the lowest and highest rank from the window. In image processing applications these minimum and maximum filters are sometimes also called dilation and erosion filters, because they make dark regions in an image respectively larger or smaller, as shown in Figure 1.2.

Rank order filters take a lot of processing power to implement, a lot more than the well-known linear filters. This is why rank order filters are rarely used in real-time signal processing applications. However, just like linear filters, rank order filters can benefit from parallelism. If enough parallelism is available any

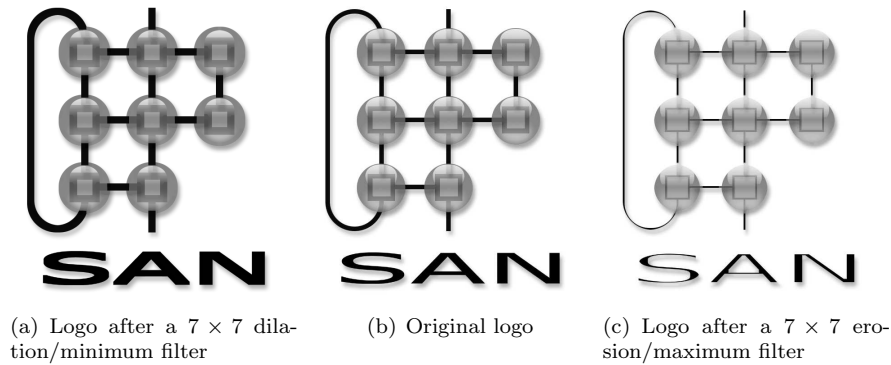


Figure 1.2: Minimum and maximum filtering

desirable throughput can be achieved.

In this report we take a look at rank order filters and implement them in an efficient, parallel fashion such that the throughput of the implementation scales linearly with the amount of parallelism. What constitutes as "the amount of parallelism" depends on the implementation method used. When implementing the filter in hardware, as a VLSI circuit, the amount of parallelism would be the number of components used. When implementing the filter in software on a SIMD processor, the amount of parallelism is determined by the number of processing elements of the SIMD processor.

Implementing the filters in such a manner that the throughput scales linearly with the amount of parallelism is quite trivial. The calculations for the outputs of the filter have constant complexity and are independent of each other, therefore unlimited parallelism can be achieved. However, more efficient implementations are possible; the calculations of successive outputs have enough in common to allow the reuse of sub-calculations, thereby increasing efficiency. By exploiting this reuse we are able to find the most efficient implementations to date.

The implementation of rank order filters clearly requires some form of sorting. To make practical implementation possible we resort to so-called oblivious sorting algorithms. Oblivious sorting algorithms always perform the same number of operations in the same order, regardless of the data that has to be sorted. The well-known quick-sort algorithm, for example, is not an oblivious algorithm, whereas another well-known sorting algorithm, namely merge sort, is oblivious. We choose oblivious sorting algorithms because they are especially suited both for implementation on SIMD processors, since there is no data-dependency in the execution path, and for hardware implementation, since there is no overhead in the form of control logic.

In the literature there are already a number of parallel designs available for both hardware [2, 3, 9] and SIMD [8] implementations, but we improve upon all of them. The most important contributions described in this report, however, are the optimizations that can be applied to existing filters and the mathematical framework in which the design problem, and its solutions, can be described.

We introduce these contributions the following manner. First we formally define the problem of designing the implementation of a rank order filter in Chapter 2. In Chapter 3 we introduce the so-called calculation graph as an abstract representation of the calculations performed by a rank order filter based on oblivious sorting networks. Since a filter calculates (theoretically) infinitely many outputs, the associated calculation graph is also infinite. However, a calculation graph has regularities and in Chapter 4 we introduce generator graphs which are finite, yet still describe an entire calculation graph. The design of the filter implementation can be read from such a generator graph, but a suitable generator graph has to be found first, which is the subject of Chapter 5. In Chapter 6 we compare our designs with those found in the literature. The practical issues that arise when a design is turned into an actual implementation are discussed in Chapter 7. And finally, there is the conclusion in Chapter 8.

Chapter 2

Problem Description

The problem we address in this report is that of constructing efficient, scalable parallel implementations of rank order filters. We will not limit ourselves to filters that determine a single rank, but allow filters that produce a range of ranks as output.

For implementing rank order filters we use oblivious sorting methods, since the absence of data dependencies facilitates the implementation on SIMD machines and in VLSI circuits.

The outline of this chapter is as follows. The next section introduces a rigorous mathematical formulation of the problem and Section 2.2 gives the complexity measure by which we can judge the various algorithms used to solve the problem.

2.1 Mathematical Definition

A common property of filter applications is that the data to be filtered is organized in some underlying structure. For each location of this structure a filtered value has to be determined based on the original data item at that location and on the values of data items in some finite neighborhood. The precise way in which such a filter operates is given by its kernel, which describes both the structure of the neighborhood and the contribution of the individual values in that neighborhood to the resulting value.

In this report we restrict ourselves to very simple underlying structures, viz. to grid topologies. So, for $D > 0$, let \mathbb{Z}^D be a D -dimensional grid with integral coordinates. The notion of a neighborhood is captured by a tile which is nothing but a finite set of grid locations. The size of a tile $\mathbf{T} \in \mathbb{P}(\mathbb{Z}^D)$ is the number of grid locations in \mathbf{T} and is denoted by $|\mathbf{T}|$. We say that two tiles \mathbf{S} and \mathbf{T} have an equivalent shape when there exists a translation vector $\mathbf{t} \in \mathbb{Z}^D$ such that $\mathbf{T} = \mathbf{S} + \mathbf{t}$ with tile $\mathbf{S} + \mathbf{t}$ defined as $\{\mathbf{s} + \mathbf{t} \mid \mathbf{s} \in \mathbf{S}\}$. Note that tiles that differ by a rotation are *not* considered equivalently shaped. In the sequel we will be interested in sets of tiles with equivalent shapes, laid out in repetitive patterns. So we define the function $\text{tile} : (\mathbb{P}(\mathbb{Z}^D) \times \mathbb{Z}^D \times \mathbb{Z}^D) \rightarrow \mathbb{P}(\mathbb{P}(\mathbb{Z}^D))$ given by:

$$\text{tile}(\mathbf{S}, \mathbf{f}, \mathbf{p}) = \{\mathbf{S} + \mathbf{f} + \mathbf{g} \cdot \mathbf{p} \mid \mathbf{g} \in \mathbb{Z}^D\} \quad (2.1)$$

In this definition the dot-operator “ \cdot ” stands for coordinate-wise vector multi-

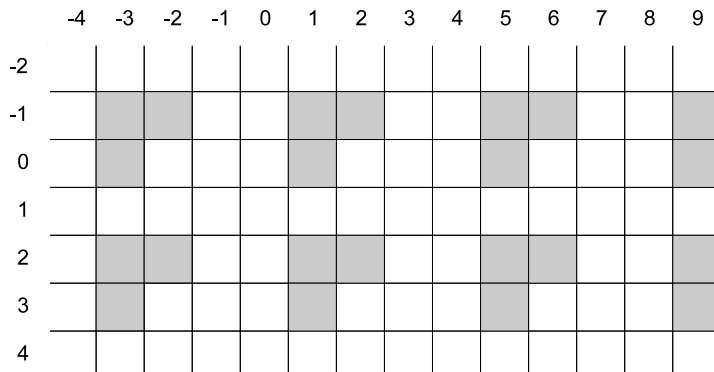


Figure 2.1: The tiles generated by $\text{tile}(\{(0,0), (1,0), (0,1)\}, (1,2), (4,3))$

plication. Function `tile` produces an infinite number of copies of tile \mathbf{S} that are spread over the entire grid in a periodic fashion, as illustrated in Figure 2.1. We call \mathbf{S} the *shape*, \mathbf{f} the *phase*, and \mathbf{p} the *period* of the resulting tile set. When a tile set \mathcal{T} can be described by the function `tile`, we use the triple $(\mathbf{S}_{\mathcal{T}}, \mathbf{f}_{\mathcal{T}}, \mathbf{p}_{\mathcal{T}})$ to describe it (i.e. $\mathcal{T} = \text{tile}(\mathbf{S}_{\mathcal{T}}, \mathbf{f}_{\mathcal{T}}, \mathbf{p}_{\mathcal{T}})$). Note that this triple is not unique, since $\text{tile}(\mathbf{S}_{\mathcal{T}} + k\mathbf{f}_{\mathcal{T}}, (1-k)\mathbf{f}_{\mathcal{T}}, \mathbf{p}_{\mathcal{T}})$ generates the same tile set for every $k \in \mathbb{Z}$.

We also introduce the two basic vectors $\mathbf{0}_D$ and $\mathbf{1}_D$. These two vectors are of length D and contain only zeroes or ones respectively. Using these vectors we can construct such functions as:

$$\text{hypercube}(k, D) = \{\mathbf{g} \mid \mathbf{g} \in \mathbb{Z}^D \wedge -k\mathbf{1}_D \leq \mathbf{g} \leq k\mathbf{1}_D\} \quad (2.2)$$

In this definition the relationship “ \leq ” stands for coordinate-wise comparison, and the relationship “ \leq ” only holds for the entire vector when it holds for all of its elements. The function `hypercube` therefore produces a tile centered on the origin, with a volume of $(2k+1)^D$, i.e. a hypercube in D dimensions.

Let \mathbb{D} be a domain of data values. The precise nature of the elements of \mathbb{D} is irrelevant apart from the fact that they can be ordered according to some linear order \preceq . Furthermore, let $\text{data} : \mathbb{Z}^D \rightarrow \mathbb{D}$ be a function that associates a data value with each grid location, i.e. $\text{data}(\mathbf{g})$ is the value at grid location $\mathbf{g} \in \mathbb{Z}^D$. Denoting the sorted list of data elements of \mathbf{T} by $\overline{\mathbf{T}}$ we can indicate the data element of rank i of \mathbf{T} by $\overline{\mathbf{T}}(i)$ and the sorted sublist of elements with ranks ranging from l up to and including u with $\overline{\mathbf{T}}[l..u]$.

With these preliminaries done, we can define what we consider to be a rank-order filter.

Definition 2.1.1 (Rank order filter (ROF)). *A rank order filter is an algorithm parameterized by*

- a number $D \in \mathbb{N}^+$ that specifies the dimension of the grid,
- a set of similar shaped tiles, called windows $\mathcal{W} = \text{tile}(\mathbf{S}_{\mathcal{W}}, \mathbf{f}_{\mathcal{W}}, \mathbf{p}_{\mathcal{W}})$,
- two numbers l and u such that $0 \leq l \leq u < |\mathbf{S}_{\mathcal{W}}|$.

Given an arbitrary function $\mathbf{data} : \mathbb{Z}^D \rightarrow \mathbb{D}$, the rank order filter determines for each $\mathbf{W} \in \mathcal{W}$ the sorted list $\overline{\mathbf{W}}[l..u]$, that is the elements of \mathbf{W} with ranks ranging from l up to and including u . \square

Note that most rank order filters are designed for a specific value or range of values for the parameters D , \mathcal{W} , l and u . An image processing application, for example, only needs 2-dimensional rank order filters ($D = 2$). Also, in most cases we have $l = u$, i.e. rank order filters that select one specific rank. However, the definition above also includes so-called window sorters ($l = 0$ and $u = |\mathbf{S}_{\mathcal{W}}|$) and other intermediate problems which will be useful in designing rank order filters.

Some special cases of rank order filters are:

The 1-D median filter with window size $2K + 1$:

- The number of dimensions is $D = 1$
- The input signal is represented by the **data** function
- The shape of the windows is $\mathbf{S}_{\mathcal{W}} = \mathbf{hypercube}(K, D)$
- The windows themselves are given by $\mathcal{W} = \mathbf{tile}(\mathbf{S}_{\mathcal{W}}, 1, 0)$
- We are interested in the item with rank K , hence $l = u = K$

The 2-D median filter with kernel size $(2K + 1) \times (2K + 1)$:

- The number of dimensions is $D = 2$
- The input image is represented by the **data** function
- The shape of the windows is $\mathbf{S}_{\mathcal{W}} = \mathbf{hypercube}(K, D)$
- The windows themselves are given by $\mathcal{W} = \mathbf{tile}(\mathbf{S}_{\mathcal{W}}, (1, 1), (0, 0))$
- We are interested in the item with rank $2K^2 + 2K$, hence $l = u = 2K^2 + K$

Note that the function **tile** produces an infinite set. In practice the set of windows for a rank order filter is finite. However, for now we will assume the set of windows to be infinite, and save this aspect of practical implementation for Section 7.3.

Using the definition of a rank-order filter we can define what we consider to be a rank-order filter design problem.

Definition 2.1.2 (Rank order filter design problem (ROFDP)). *Assume that we are given*

- a number $D \in \mathbb{N}^+$ that specifies the dimension of the filter,
- a set of similar shaped tiles, called windows $\mathcal{W} = \mathbf{tile}(\mathbf{S}_{\mathcal{W}}, \mathbf{f}_{\mathcal{W}}, \mathbf{p}_{\mathcal{W}})$,
- two numbers l and u such that $0 \leq l \leq u < |\mathbf{S}_{\mathcal{W}}|$.

Design a rank order filter for these parameters. \square

To solve this problem and to find an algorithm that calculates the rank $[l..u]$ elements of every window in an oblivious manner, we will need some form of sorting network. However, the optimum networks for sorting lists of arbitrary size are unknown. We only know good instances when it comes to minimum-comparison sorting networks [7]. Therefore we are content to resort to heuristic methods to find the implementations of rank order filters.

2.2 Complexity measure

The elementary operation of oblivious sorting is a so-called compare and swap (C&S) operation. This operation effectively sorts a list of two inputs by comparing them and swapping them if necessary. The same result can be obtained by using a minimum and a maximum operator (the so-called extrema operations). Sorting is then accomplished by making the first element of the resulting list the minimum of the two inputs and the last element of the resulting list the maximum of the two inputs.

In this report we will use the number of extrema operations to measure the complexity of a ROF. We do so because the extrema count allows for a more fine-grained determination of the cost. If, for example, we only need one of the outputs of a C&S operation, we still need a single C&S operation. However, whenever two extrema operations are used to implement the C&S operation we can eliminate the one that produces the unneeded output.

Since there is no consensus in literature whether to use extrema or C&S operation counts, one has to be careful when comparing implementations. Even when the difference is taken into account there is no quick translation of costs. An C&S operation consists of two extrema operations, but it is possible that the design discussed in the literature can be optimized by converting all the C&S operations into extrema operations and subsequently removing the unnecessary operations.

However, whether C&S or extrema operations are used to implement a rank order filter, it is beneficial to minimize the operation count. We could have used an other complexity measure like the throughput (i.e. the number of results produced per unit of time). This measure, however, is implementation specific. Moreover, the operation count signifies the amount of work that needs to be done, therefore minimizing the amount of work will speed up any implementation, regardless what particular method of implementation is used.

Therefore the problem we investigate in this report is: What is the minimum number of extrema operations needed for a rank-order filter?

Chapter 3

Problem Approach

In this chapter we analyze the problem, derive some properties and introduce an abstract representation for our solutions, which we call the calculation graph. This graph captures the structure of the calculation performed by a rank order filter. To find this structure we first examine the basic operation of an oblivious sorting algorithm (namely the merge operation) in Section 3.1. In Section 3.2 we show that the problem of calculating the filter result provides the opportunity to reuse intermediate results. This leads to the introduction of the calculation graph in Section 3.3, which specifies which merge operations have to take place, provides a (partial) order for them and specifies where the operation results are (re)used. A cost function is needed to calculate the cost (in extrema operations) of an algorithm represented by a calculation graph. One such cost function, based on pruned Batcher's merging networks, is introduced in Section 3.4. In Section 3.5 we show how a simple optimization leads to better merging networks and thus a different cost function.

3.1 Divide and Conquer

A familiar and efficient way of solving problems is the divide and conquer strategy. When this strategy is applied to a sorting problem the result is the merge sort algorithm. The basic operation of a merge sort algorithm is the merging of two sorted lists into a (larger) sorted list denoted by the function:

$$\overline{\mathbf{T}} = \text{merge}(\overline{\mathbf{T}}_1, \overline{\mathbf{T}}_2) \quad (3.1)$$

where $\{\mathbf{T}_1, \mathbf{T}_2\}$ is a partition of \mathbf{T} , i.e. $\mathbf{T}_1 \cup \mathbf{T}_2 = \mathbf{T}$ and $\mathbf{T}_1 \cap \mathbf{T}_2 = \emptyset$.

As a first step in designing a rank order filter we look at the possibility of dividing the problem of calculating a single window into smaller subproblems in a similar manner. As proper generalization of (3.1) we introduce

$$\overline{\mathbf{T}}[l_0..u_0] = \text{merge}(\overline{\mathbf{T}}_1[l_1..u_1], \overline{\mathbf{T}}_2[l_2..u_2])[l_3..u_3] \quad (3.2)$$

However, this equality only holds if the parameters of the `merge` function adhere to the following properties:

1. (a) $0 \leq l_i \leq ((l_0 - |\mathbf{T}| + |\mathbf{T}_i|) \uparrow 0)$ for $i = 1, 2$
(b) $(u_0 \downarrow (|\mathbf{T}_i| - 1)) \leq u_i < |\mathbf{T}_i|$ for $i = 1, 2$

2. (a) $l_3 = l_0 - (l_1 + l_2)$
- (b) $u_3 = u_0 - (l_1 + l_2)$

A merge operation that adheres to these properties is called a valid merge operation. An example of such an operation is depicted in Figure 3.1.

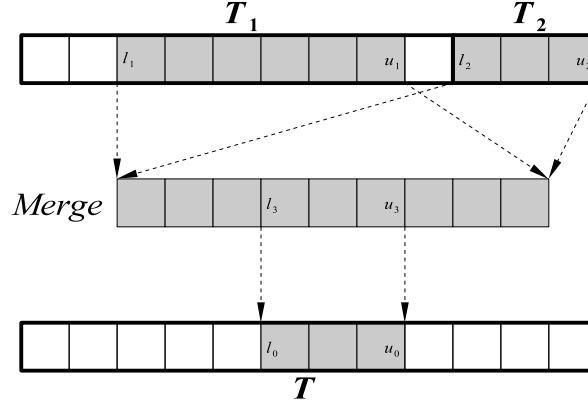


Figure 3.1: A merge operation according to (3.2)

To explain property 1 we introduce the function “ rnk ” to denote the rank of $\text{data}(\mathbf{g})$ in $\overline{\mathbf{T}}$. More formally:

$$\text{rnk}(\mathbf{g}, \mathbf{T}) = (\downarrow i : 0 \leq i < |\mathbf{T}| \wedge \overline{\mathbf{T}}(i) = \text{data}(\mathbf{g}) : i) \quad (3.3)$$

Observe that for any element $\mathbf{g} \in \mathbf{T}_i$ (for $i = 1, 2$) it holds that $\text{rnk}(\mathbf{g}, \mathbf{T}) \geq \text{rnk}(\mathbf{g}, \mathbf{T}_i)$. In other words, adding elements to a set can only increase the rank of an element already in the set, but it cannot decrease it.

Because \mathbf{T} contains $|\mathbf{T}| - |\mathbf{T}_i|$ elements more than \mathbf{T}_i , we therefore know that $\text{rnk}(\mathbf{g}, \mathbf{T}_i) \leq \text{rnk}(\mathbf{g}, \mathbf{T}) \leq \text{rnk}(\mathbf{g}, \mathbf{T}_i) + (|\mathbf{T}| - |\mathbf{T}_i|)$. So the interval $[l_i..u_i]$ must be such that it contains (at least) all elements $\mathbf{g} \in \mathbf{T}_i$ with $\text{rnk}(\mathbf{g}, \mathbf{T}_i) \leq u_0$ and $\text{rnk}(\mathbf{g}, \mathbf{T}_i) + (|\mathbf{T}| - |\mathbf{T}_i|) \geq l_0$ (i.e. at least all the elements \mathbf{g} of \mathbf{T}_i for which $l_0 \leq \text{rnk}(\mathbf{g}, \mathbf{T}) \leq u_0$ might hold). This is exactly what property 1 expresses.

Property 2 can be derived by noting that the merge operation results in a list that is a sublist of $\overline{\mathbf{T}}$. The missing elements are elements that, as a consequence of property 1, have a rank in \mathbf{T} that is not within the interval $[l_0..u_0]$. Of the removed elements there are $l_1 + l_2$ elements with a rank lower than l . The removal of these elements means that an index r in the merged list corresponds to a rank $r + l_1 + l_2$ in the set \mathbf{T} for $l_0 \leq r + l_1 + l_2 \leq u_0$.

From these properties we can draw some interesting conclusions. First of all, let us return to our original problem of calculating for a single window $\mathbf{W} \in \mathcal{W}$ a range of ranks $\overline{\mathbf{W}}[l..u]$. When we write such a calculation as a series of nested merge operations on tiles \mathbf{T}_j for some j , we can use property 1 to calculate the ranges for the lower and upper bounds for each of the tiles \mathbf{T}_j . It turns out that these bounds depend only on $|\mathbf{T}_j|$ and $|\mathbf{S}_{\mathcal{W}}|$. So introduce functions that calculate the highest valid lower bound, and the lowest valid upper bound:

$$\begin{aligned} \text{low}(\mathbf{T}) &= (l - |\mathbf{S}_{\mathcal{W}}| + |\mathbf{T}|) \uparrow 0 \\ \text{up}(\mathbf{T}) &= u \downarrow (|\mathbf{T}| - 1) \end{aligned} \quad (3.4)$$

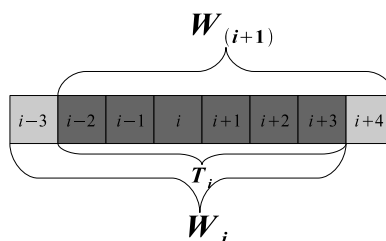


Figure 3.2: An example of reuse

Using these functions guarantees validity of the merge operation, w.r.t. property 1, and at the same time minimizes the size of the interval $[\text{low}(\mathbf{T}).. \text{up}(\mathbf{T})]$. A small interval means little information to calculate and keep track of, which benefits performance. So, once \mathbf{T}_1 and \mathbf{T}_2 have been chosen, the best choice for l_1, u_1, l_2 and u_2 is given by (3.4). Therefore we will be using these bounds throughout this report, and we introduce a special notation for it:

$$[\mathbf{T}] = \overline{\mathbf{T}}[\text{low}(\mathbf{T}).. \text{up}(\mathbf{T})] \quad (3.5)$$

Also, since property 2 uniquely determines l_3 and u_3 , we can omit them in our notation of any valid merge operation. Therefore we introduce the operator \bowtie , so that we can write (3.2) as:

$$[\mathbf{T}] = [\mathbf{T}_1] \bowtie [\mathbf{T}_2] \quad (3.6)$$

Note that the operator we introduce is associative and commutative, as can be expected from a generalization of the standard merge operation and therefore it is functionally correct to write:

$$([\mathbf{T}_1] \bowtie [\mathbf{T}_2]) \bowtie [\mathbf{T}_3] = ([\mathbf{T}_2] \bowtie [\mathbf{T}_3]) \bowtie [\mathbf{T}_1] \quad (3.7)$$

However, the costs of implementing the left and right hand side of this expression may differ. Consider the following example: $l = 0, u = |\mathbf{S}_W| - 1, |\mathbf{S}_W| > 4, |\mathbf{T}_1| = 1, |\mathbf{T}_2| = 1$ and $|\mathbf{T}_3| = 2$. In this example the size of the tiles is the same as the size of the interval of ranks required from them. So, the left hand side of equation (3.7) tells us to merge two tiles of size 1 and the resulting tile (of size 2) with a tile of size 2. The right hand side, on the other hand, tells us to merge a tile of size 1 and size 2 and subsequently merge the resulting tile (of size 3) with a tile of size 1. These are all quite different merging operations and it is not surprising that they differ in cost.

3.2 Divide, Re-use and Conquer

Divide and conquer suffices when we have to "rank-filter" a single window, but our problem statement is concerned with multiple windows that overlap. This overlapping means that there could be common subexpressions that can be reused. This is illustrated in Figure 3.2. The figure shows that we can calculate the two windows \mathbf{W}_i and \mathbf{W}_{i+1} as $[\mathbf{W}_i] = [\{i-3\}] \bowtie [\mathbf{T}_i]$ and

$[\mathbf{W}_{i+1}] = [\mathbf{T}_i] \bowtie [\{i+4\}]$. This allows us to use $[\mathbf{T}_i]$ twice, and thus the cost for calculating the intermediate result $[\mathbf{T}_i]$ is amortized over two window calculations.

Dynamic programming is a standard technique for dealing with such situations, but it can only be applied if a problem exhibits two properties: optimal substructures and overlapping subproblems ([4] Chapter 15). We have already shown that the problem has overlapping subproblems, so we only have to show that it exhibits optimal substructures. Unfortunately this is not possible, since the subproblems are not independent; they share resources. Each subproblem chooses the tiles used in the construction of its windows, these tiles represent the resources and the cost for these resources is shared between the subproblems if reuse takes place. A good, every-day, analogy is car-pooling; perhaps the cheapest way to travel for each individual is by public transport, but travel by car becomes cheaper if you travel in a group and the costs can be amortized.

So, applying dynamic programming to our design problem is not possible. However, the variant of dynamic programming known as “memoization” can still be used in the implementation of the filter ([4] Chapter 15). Using memoization amounts to an implementation that stores all tiles in a table and uses a table-lookup whenever possible. This minimizes the number of extrema operations, at the cost of table lookup operations, and makes sure all previously calculated results are reused whenever possible. In Chapter 7 we show that the regularity of our designs can be used to implement this table and the lookup operations efficiently with a small number of buffers.

3.3 Calculation Graph

In this section we introduce the notion of a calculation graph $G = (V, E)$ to capture the computation performed by a ROF. The nodes of this graph, given by set $V \subseteq \mathbb{P}(\mathbb{Z}^D)$, represent all the (intermediate) results produced during the calculation, i.e. the tiles that will be put in the look-up table. The set of nodes must contain at least:

- all the windows $\mathcal{W} \subseteq V$, and
- all the necessary elements $(\forall \mathbf{W}, \mathbf{i} : \mathbf{W} \in \mathcal{W} \wedge \mathbf{i} \in \mathbf{W} : \{\mathbf{i}\} \in V)$

The set of edges $E \subseteq V \times V$ denotes how the tiles are used to construct other tiles. An edge from \mathbf{T}_1 to \mathbf{T} means that tile \mathbf{T}_1 is used to construct tile \mathbf{T} with a merge operation. The following constraints must hold for this set:

1. $(\forall \mathbf{T} : \mathbf{T} \in V \wedge |\mathbf{T}| > 1 : [\mathbf{T}] = (\bowtie \mathbf{T}_1 : (\mathbf{T}_1, \mathbf{T}) \in E : [\mathbf{T}_1]))$
2. $(\forall \mathbf{T} : \mathbf{T} \in V \wedge |\mathbf{T}| > 1 : |\{(\mathbf{T}_1, \mathbf{T}) | (\mathbf{T}_1, \mathbf{T}) \in E\}| = 2)$

Property 1 expresses that each tile that does not consist of a single data element is formed by merging the tiles specified by the set E . Note that this implies that the constraints for a valid merge operation, as derived in Section 3.1, must also hold: i.e. together all the tiles used in the construction of a tile \mathbf{T} must form a partition of that tile \mathbf{T} . This immediately implies that the graph must be acyclic, since merging tiles can only result in larger tiles. Furthermore, it implies that for each window there must exist paths that start in the tiles representing the window’s components and end in the tile representing the window.

The calculation graph effectively describes the calculation of the windows during the implementation. Each node in the graph represents the result of a calculation and the node's incoming edges represent the recursive calculations needed to for the sub-tiles. However, we need another property besides 1 to define the calculation done by the implementation unambiguously.

Property 2 enforces that each merge operation merges only two tiles. This removes any ambiguity that arises when more than two tiles are merged. As mentioned in Section 3.1 the costs may differ depending on the order in which tiles are merged. By limiting the calculation graph to an in-degree of 2, we enforce the specification of the evaluation order of the merge operations, and thus remove the ambiguity.

In summary, any solution for a ROFDP can be described as a calculation graph G . The nodes of this graph (V) are called tiles and represent the (intermediate) results of the calculation. The edges (E) of the graph represent the way in which tiles are combined, using merging operations, to form larger tiles.

The cost of the solution can be expressed as $\mathcal{C}(G)$:

$$\mathcal{C}(G) = \lim_{k \rightarrow \infty} \left(\sum_{\mathbf{T}, \mathbf{T}_1, \mathbf{T}_2 : \mathbf{T} \in (V \cap \text{hypercube}(k, D)) \wedge (\mathbf{T}_1, \mathbf{T}) \in E \wedge (\mathbf{T}_2, \mathbf{T}) \in E : \frac{\text{MC}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T})}{|\mathcal{W} \cap \text{hypercube}(k, D)|} \right) \quad (3.8)$$

where $\text{MC}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T})$ is the cost function that determines the cost, in extrema operations, for a merge operation $[\mathbf{T}] = [\mathbf{T}_1] \bowtie [\mathbf{T}_2]$. Note that we have to take the limit here because \mathcal{W}, V and E are infinite sets. So $\mathcal{C}(G)$ effectively gives us the cost in extrema operations per window. From this point on we abbreviate this unit of cost as ‘‘epw’’.

The Rank Order Filter Design Problem can now be restated as follows: given a specification find the calculation graph G that adheres to the properties above and minimizes $\mathcal{C}(G)$ for a certain cost function MC . Before we solve this problem we first examine the cost function.

3.4 Cost of Merging Networks

In principle the calculation graph is indifferent to the implementation of the merge function. In order to associate a cost with a calculation graph we have to choose a particular implementation. In this report we choose to implement each merging operation using an instance of Batcher's merging network, because Batcher's merging network is the most efficient, known, network for merging two sorted lists. Also, there is a simple formula to calculate the number of extrema operations used in such a network. Note that this approach differs from most of the literature, where a combination of the optimum sorting networks and Batcher's merging networks is used. However, in Chapter 6 we show that we improve upon the results existing in the literature, despite the fact that we do not use the optimum sorting networks.

The formula for the number of compare and swap operations in the network can be found in [7]. However, recall that we measure the cost of our calculation in extrema operations. The cost function in [7] measures cost in C&S operations, so we had to adapt our cost function to produce its result in extrema

operations. Also, since each tile \mathbf{T} represents only an interval $[\text{low}(\mathbf{T}).. \text{up}(\mathbf{T})]$ of the resulting list, we do not need the entire merging network. We only need that part of the network that has an effect on the desired outputs, and we can prune the network to get rid of the rest of the extrema operations.

Pruning is a relatively simple optimization that can be done automatically. In the case of a software implementation it is called "dead-code elimination" and in circuit design it is called "unused logic elimination", which is performed automatically by the compiler or synthesizer respectively. So it is by no means a new optimization, but in this section we introduce (to the best of our knowledge) the first cost function for such networks.

The function $\text{PB}(s_1, s_2, l_0, u_0)$ for $0 \leq l_0 \leq u_0 < s_1 + s_2$ returns the number of extrema operations required by a pruned version of Batcher's merging network which takes sorted lists, say $\overline{\mathbf{T}}_1$ and $\overline{\mathbf{T}}_2$, of size $s_1 = |\mathbf{T}_1|$ and $s_2 = |\mathbf{T}_2|$ respectively and produces the interval $[l_0..u_0]$ of the sorted list $\mathbf{T}_1 \cup \mathbf{T}_2$. This function PB satisfies the following recurrence relation:

$$\text{PB}(s_1, s_2, l_0, u_0) = \begin{cases} 0 & (s_1 = 0) \vee (s_2 = 0) \vee \\ & (u_0 = -1) \vee (l_0 = s_1 + s_2) \\ (u_0 \downarrow 1) - (l_0 \uparrow 0) + 1 & (s_1 = s_2 = 1) \wedge \\ & (u_0 \neq -1) \wedge (l_0 \neq s_1 + s_2) \\ \text{PB}(\lceil \frac{s_1}{2} \rceil, \lceil \frac{s_2}{2} \rceil, \lceil \frac{l_0}{2} \rceil, \lceil \frac{u_0}{2} \rceil) + & \text{otherwise} \\ \text{PB}(\lfloor \frac{s_1}{2} \rfloor, \lfloor \frac{s_2}{2} \rfloor, \lfloor \frac{l_0}{2} \rfloor - 1, \lfloor \frac{u_0}{2} \rfloor - 1) + & \\ (u_0 \downarrow (2 \lfloor \frac{s_1 + s_2 - 1}{2} \rfloor)) - (l_0 \uparrow 1) + 1 & \end{cases} \quad (3.9)$$

Note that this recurrence relation has a larger domain than $0 \leq l_0 \leq u_0 < s_1 + s_2$. We expanded the domain to $l_0 \leq u_0 \wedge -1 \leq u_0 \wedge l_0 \leq s_1 + s_2$ to simplify the relation's notation. The extra range of values introduced for l_0 and u_0 does not really affect the outcome of the function, since we have that $\text{PB}(s_1, s_2, l_0, u_0) = \text{PB}(s_1, s_2, l_0 \uparrow 0, u_0 \downarrow (s_1 + s_2 - 1))$.

The recurrence relation is a modification of the cost function for Batcher's merging networks without pruning given in [7]. The first case of (3.9) are simple; first of all if $s_1 = 0, s_2 = 0, u_0 = -1$ or $l_0 = s_1 + s_2$, then there is no merging necessary, since either one of the lists is empty, or we need an empty interval from the resulting list.

The second case occurs when two lists of length 1 are merged. In this case the cost can also be easily calculated. If only one member of the resulting list is needed ($l_0 = u_0$) then we need 1 extrema operation to determine it. If both members of the resulting list are needed ($l_0 < u_0$) then we need 2 extrema operations.

The third case is the most complicated, but the example in Figure 3.3 might help in understanding. In this figure a merging network is depicted by a box. The numbers in the upper left (s_1) and upper right corner (s_2) denote the size of the lists to be merged. The box contains a merging network that accomplishes that operation. This merging network consists of three parts, the first two are recursively merging the even and odd sequences, which have length $\lceil \frac{s_1}{2} \rceil + \lceil \frac{s_2}{2} \rceil$ and $\lfloor \frac{s_1}{2} \rfloor + \lfloor \frac{s_2}{2} \rfloor$ respectively. In the figure we see this recursion as the nesting of two boxes in the main box, in equation (3.9) we see this as the two recursive calls.

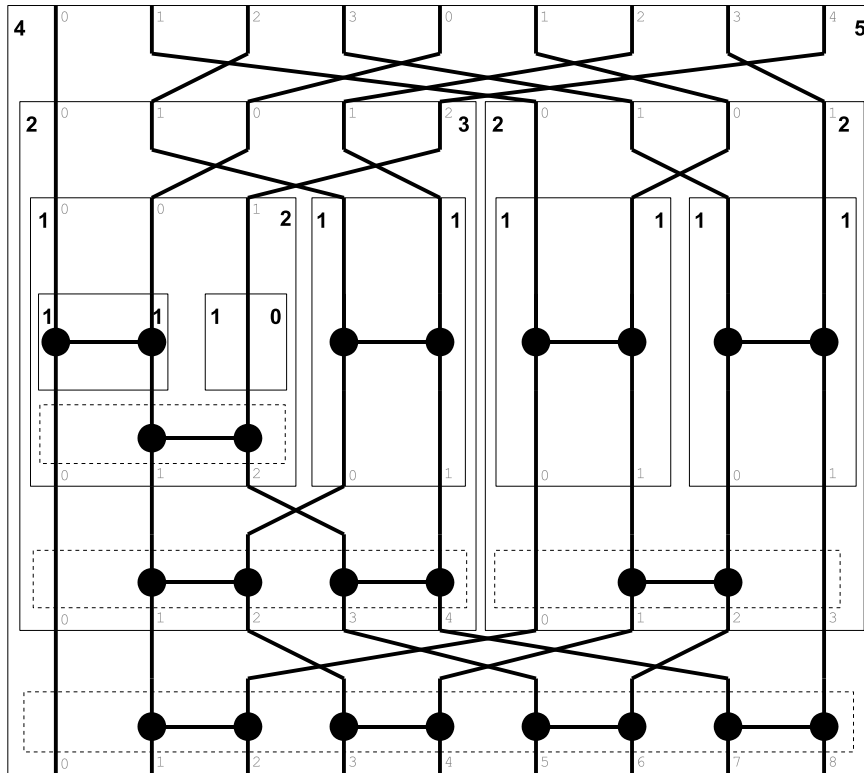


Figure 3.3: Batcher's merging network for a 4-5 merge

The third part of the merging network consist of a network that merges the results of the other two (odd and even) parts. In the figure this part has been marked by a dashed box. Its structure is relatively simple; it pairwise compares and swaps elements from the even and the odd list, starting from element 1 from the even list and element 0 from the odd list. This is illustrated in the picture by the small grey numbers, which indicate the indices of the elements in the sorted lists.

This pair-merging network allows us to derive the lower and upper bounds of the ranks for the recursive calls; from the even sequence we need $\lceil \frac{l_0}{2} \rceil$ and $\lceil \frac{u_0}{2} \rceil$ as lower and upper bound respectively. This, and the structure of the pair-merging network implies that we need $\lceil \frac{l_0}{2} \rceil - 1$ through $\lceil \frac{u_0}{2} \rceil - 1$ from the odd sequence.

The cost of the pair-merging network is expressed by the last term of the third case of (3.9). This term looks daunting, but we can understand it by noting that every element passing through the pair-merging network takes part in one extrema operation, except for the element with index 0 and possibly the element with the highest index. So, we simply count the number of elements that pass through the pair-merging network to get the number of extrema operations.

The first element that passes through the pair-merging network and participates in an extrema operation is $l_0 \uparrow 1$. The last element that participates is

u_0 when $u_0 < s_1 + s + 2$. Otherwise the last element is $s_1 + s_2$ if $s_1 + s_2$ is odd, or $s_1 + s_2 - 1$ if $s_1 + s_2$ is even. Hence when pruning takes place the last element that participates in an extrema operation is the element with index $u_0 \downarrow 2 \lfloor \frac{s_1+s_2-1}{2} \rfloor$.

3.5 Mirroring

Since Batcher's merging networks are the best merging networks known, we might expect that their pruned versions are also the best. This is, however, not the case.

This is caused by the fact that when the range of desired ranks is not symmetric around the center of the window the amount of pruning depends on the order in which the extrema operators are arranged in the network. As an example let $\overline{\mathbf{T}}_1$ be a list of size $|\mathbf{T}_1| = 1$ and let $\overline{\mathbf{T}}_2$ be a list of size $|\mathbf{T}_2| = 2$ and let $\overline{\mathbf{T}} = \overline{\mathbf{T}}_1 \cup \overline{\mathbf{T}}_2$ be a list of size 3. When $\overline{\mathbf{T}}[0..2]$, i.e. all ranks, is required Batcher's merging network requires 4 extrema operations (see Figure 3.4(a)). When $\overline{\mathbf{T}}[1]$, i.e. the median, is required a pruned version of the network containing 2 extrema operations suffices (see Figure 3.4(b)). When $\overline{\mathbf{T}}[2]$, i.e. the maximum, is required a pruned version of the network also containing 2 extrema operations suffices (see Figure 3.4(c)). However, when $\overline{\mathbf{T}}[0]$, i.e. the minimum, is required a pruned version of the network containing only 1 extrema operation suffices (see Figure 3.4(d)).

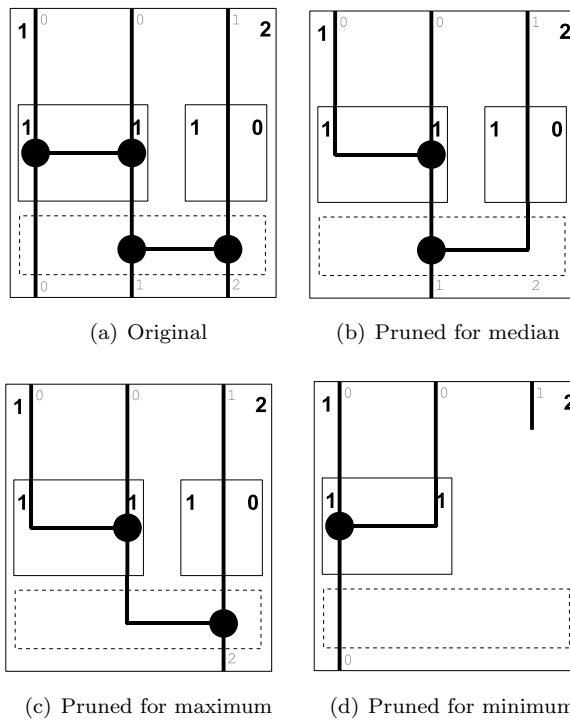


Figure 3.4: Batcher's merging network for a 1-2 merge

This example suggest that in case of pruning we can come up with another automatic optimization which involves varying the layout of the network by varying the sorting order. We either sort in the standard ascending order, or we switch to a descending order if that is less costly. Changing the sorting order amounts to taking the mirror image of a sorting network. Hence we obtain the following cost function:

$$\begin{aligned}
\text{MPB}(s_1, s_2, l_0, u_0) &= \\
&\text{MPB}'(s_1, s_2, l_0, u_0) \downarrow \text{MPB}'(s_1, s_2, s_1 + s_2 - 1 - u_0, s_1 + s_2 - 1 - l_0) \\
\text{MPB}'(s_1, s_2, l_0, u_0) &= \\
&\left\{ \begin{array}{ll} 0 & (s_1 = 0) \vee (s_2 = 0) \vee \\ & (u_0 = -1) \vee (l_0 = s_1 + s_2) \\ (u_0 \downarrow 1) - (l_0 \uparrow 0) + 1 & (s_1 = s_2 = 1) \wedge \\ & (u_0 \neq -1) \wedge (l_0 \neq s_1 + s_2) \\ \text{MPB}(\lceil \frac{s_1}{2} \rceil, \lceil \frac{s_2}{2} \rceil, \lceil \frac{l_0}{2} \rceil, \lceil \frac{u_0}{2} \rceil) + \\ \text{MPB}(\lfloor \frac{s_1}{2} \rfloor, \lfloor \frac{s_2}{2} \rfloor, \lfloor \frac{l_0}{2} \rfloor - 1, \lfloor \frac{u_0}{2} \rfloor - 1) + & \text{otherwise} \\ (u_0 \downarrow (2 \lfloor \frac{s_1 + s_2 - 1}{2} \rfloor)) - (l_0 \uparrow 1) + 1 & \end{array} \right. \tag{3.10}
\end{aligned}$$

This function is almost identical to (3.9), but at each point in the recursion we now consider whether the pruned version of the original, or the pruned version of the mirror network, minimizes the cost.

This new optimization method can also be applied to networks other than pruned versions of Batcher's merging networks. Optimal sorting networks that are pruned can also be mirrored prior to the pruning operation to see if this leads to a decrease in cost. In at least one case (namely in the solution presented in [8]) this is the case.

We are now in a position to define the cost function MC from (3.8). Using function MPB (3.10) and functions low and up as defined in (3.4) we define MC as:

$$\begin{aligned}
\text{MC}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}) &= \text{MPB}(\\
&\text{up}(\mathbf{T}_1) - \text{low}(\mathbf{T}_1) + 1, \\
&\text{up}(\mathbf{T}_2) - \text{low}(\mathbf{T}_2) + 1, \\
&\text{low}(\mathbf{T}) - \text{low}(\mathbf{T}_1) - \text{low}(\mathbf{T}_2), \\
&\text{up}(\mathbf{T}) - \text{low}(\mathbf{T}_1) - \text{low}(\mathbf{T}_2)) \tag{3.11}
\end{aligned}$$

Chapter 4

Generator Graphs

A calculation graph is an infinite graph that represents the infinite calculation performed by a filter. The filter itself, however, being an algorithm implemented either in hardware or software, is a finite object. The gap between infinite calculation graphs and finite filter implementations is bridged by so-called *generator graphs*. A generator graph is a finite representation of a filter that exploits the regularities present in the calculations performed by such a filter. On the one hand the generator graph, as its name suggests, can be used to generate a calculation graph. On the other hand, a generator graph can serve as the starting point for a concrete filter implementation.

In this chapter generator graphs are defined and it is shown how calculation graphs can be generated from them. Finding a generator graph for a specific filter is the subject of Chapter 5. However, various heuristics presented in that chapter share a common approach, which we explain in this chapter as well. Deriving parallel implementations from generator graphs is the subject of Chapter 7.

4.1 Preliminaries

Although a calculation graph contains infinitely many tiles there are only a finite number of ways in which a tile of a certain shape is constructed. This suggests that one can condense the calculation graph such that only the different construction methods are retained. Consider all tiles of a specific shape in the calculation graph. We group these tiles according to their construction method and represent this group by a single node in the generator graph.

Recall that we have already introduced the function `tile` (2.1) as a way of generating regular sets of tiles. The argument of this function is a triple consisting of a shape, a phase and a period:

$$T = (\mathbf{S}_T, \mathbf{f}_T, \mathbf{p}_T)$$

We will use these triples as the nodes of our generator graph $G_g = (V_g, E_g)$. Hence it follows that a node represents a tile-set $\mathcal{T} = \text{tile}(\mathbf{S}_T, \mathbf{f}_T, \mathbf{p}_T) = \text{tile}(T)$. Note that the phase and periods can be used to identify the various construction methods for tiles of the same shape.

So, the node set of a generator graph V_g is a subset of $(\mathbb{P}(\mathbb{Z}^D) \times \mathbb{Z}^D \times \mathbb{Z}^D)$, with each node representing a tile-set. Since distinct triples can describe the same tile set we introduce the function $\mathbf{norm}(\mathbf{T})$ that rewrites these triples to a unique canonical form that satisfies:

$$\mathbf{tile}(\mathbf{norm}(\mathbf{T})) = \mathbf{tile}(\mathbf{T}) \quad (4.1)$$

$$\mathbf{0}_D \leq \mathbf{f}_{\mathbf{norm}(\mathbf{T})} \leq \mathbf{P}_{\mathbf{norm}(\mathbf{T})} \quad (4.2)$$

$$(\forall \mathbf{g} : \mathbf{g} \in \mathbf{S}_{\mathbf{norm}(\mathbf{T})} : \mathbf{g} \geq \mathbf{0}_D) \quad (4.3)$$

$$(\forall \mathbf{t} : \mathbf{t} \in \mathbb{N}^D \wedge \mathbf{t} \neq \mathbf{0}_D : (\exists \mathbf{g} : \mathbf{g} \in (\mathbf{S}_{\mathbf{norm}(\mathbf{T})} - \mathbf{t}) : \mathbf{g} < \mathbf{0}_D)) \quad (4.4)$$

The canonical form describes the same tile set (4.1), but its phase lies between zero and the period (4.2). Also, the set of grid locations forming its shape are positioned in the positive hyper-quadrant (4.3) in such a way that there is no tile with the same shape in the positive hyper-quadrant that lies closer to the origin (4.4). All the nodes in the generator graph adhere to this canonical form.

The nodes are used to group tiles that have the same construction method and the edges of the generator graph are used to capture these construction methods. To this end the edges of the generator graph mimic those of the calculation graph. Since a tile containing more than one grid location is constructed from two other tiles the generator graph also has this property:

$$(\forall \mathbf{T} : \mathbf{T} \in V_g \wedge |\mathbf{S}_{\mathbf{T}}| > 1 : |\{(\mathbf{U}, \mathbf{T}) \mid (\mathbf{U}, \mathbf{T}) \in E_g\}| = 2)$$

However, the tiles \mathbf{T}_1 and \mathbf{T}_2 , used in the construction of a tile \mathbf{T} , may be contained in the same tile-set and thus the same node of the generator graph. This means that the generator graph is a multi-graph and thus the set of edges is a multi-set. From our definition of the nodes of the generator graph it then follows:

$$(\forall \mathbf{T}, \mathbf{T}_1, \mathbf{T}_2 : (\mathbf{T}_1, \mathbf{T}) \in E_g \wedge (\mathbf{T}_2, \mathbf{T}) \in E_g \wedge \mathbf{T}_1 \neq \mathbf{T}_2 : \mathbf{tile}(\mathbf{T}) \subseteq \{[\mathbf{T}_1] \bowtie [\mathbf{T}_2] \mid \mathbf{T}_1 \in \mathbf{tile}(\mathbf{T}_1) \wedge \mathbf{T}_2 \in \mathbf{tile}(\mathbf{T}_2)\}) \quad (4.5)$$

$$(\forall \mathbf{T}, \mathbf{T}_1 : (\mathbf{T}_1, \mathbf{T}) \in E_g \wedge (\forall \mathbf{T}_2 : (\mathbf{T}_2, \mathbf{T}) \in E_g : \mathbf{T}_1 = \mathbf{T}_2) : \mathbf{tile}(\mathbf{T}) \subseteq \{[\mathbf{T}_1] \bowtie [\mathbf{T}_2] \mid \mathbf{T}_1, \mathbf{T}_2 \in \mathbf{tile}(\mathbf{T}_1) \wedge \mathbf{T}_1 \neq \mathbf{T}_2\}) \quad (4.6)$$

These restrictions express that for every pair of edges describing the construction of a tile set \mathcal{T} , there are two other tile-sets \mathcal{T}_1 and \mathcal{T}_2 in which the necessary tiles can be found. Equation (4.5) expresses this for cases where the composing tiles come from distinct tile-sets and (4.6) expresses it for cases where the composing tiles come from the same tile-set.

Consider the example generator graph in Figure 4.1. For ease of reading we have chosen to denote shapes in a graphical manner. Each shape is a set of grid locations, and in the one dimensional case we can represent this set as a linear sequence of squares. When a square is black (■) this means that that specific grid location is in the set, when the square is blank (□) that specific grid location is not in the set. We start the numbering of grid locations at 0 because of the normalization requirements. So sequence “■■■■” denotes the set $\{0, 1, 2, 3\}$, and “■□□■” denotes the set $\{0, 3\}$.

Figure 4.1 shows us an example of why the generator graph is a directed *multi-graph*, whereas the calculation graph is not a multi-graph. A tile \mathbf{T} from the tile-set $\mathbf{tile}(\mathbf{■■■■}, 1, 2)$ is formed by taking two tiles \mathbf{T}_1 and \mathbf{T}_2 from the

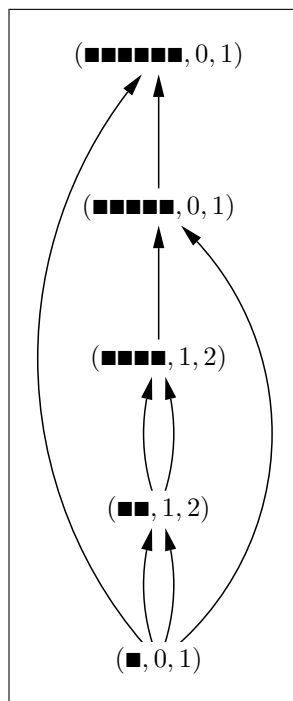


Figure 4.1: Example generator graph G_g for $\mathcal{W} = \text{tile}(\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$

same tile-set $\text{tile}(\blacksquare\blacksquare, 1, 2)$. In the generator graph nodes represent tile-sets and we might need two tiles from the same tile-set to generate a tile, hence we need the generator graph to be a multi-graph. In the calculation graph each tile is represented by a single node and hence we only need at most one edge between nodes.

4.2 Canonical generator graphs

Using the preliminaries from the previous section we can now define the relationship between the nodes of the calculation graph (V) and the generator graph (V_g) more formally:

$$V = (\cup \mathbf{T} : \mathbf{T} \in V_g : \text{tile}(\mathbf{S}_{\mathbf{T}}, \mathbf{f}_{\mathbf{T}}, \mathbf{p}_{\mathbf{T}}))$$

The set of edges of the calculation graph describe the construction method for the tiles/nodes of the calculation graph. These edges can also be derived from the generator graph. Using restriction (4.5) and (4.6) we specify:

$$(\forall \mathbf{T}, \mathbf{T}_1 : \mathbf{T} \in \text{tile}(\mathbf{T}) \wedge (\mathbf{T}_1, \mathbf{T}) \in E_g : (\exists \mathbf{T}_2 : \mathbf{T}_2 \in \text{tile}(\mathbf{T}_1) : (\mathbf{T}_2, \mathbf{T}) \in E))$$

Note that since E_g is a multi-set, a duplicate edge in E_g means that there are two (distinct) edges in E .

This, however, does not describe exactly which tiles $\mathbf{T}_1 \in \mathcal{T}_1$ and $\mathbf{T}_2 \in \mathcal{T}_2$ should be used in the construction of a tile $\mathbf{T} \in \mathcal{T}$, only that a construction

method exists for some \mathbf{T}_1 and \mathbf{T}_2 . However, there are at most two combinations of tiles \mathbf{T}_1 and \mathbf{T}_2 that partition the tile \mathbf{T} . This can be explained as follows: take a grid location $\mathbf{g} \in \mathbf{T}$. We now have either $\mathbf{g} \in \mathbf{T}_1$ or $\mathbf{g} \in \mathbf{T}_2$, i.e. the tile containing \mathbf{g} comes either from tile-set \mathcal{T}_1 or \mathcal{T}_2 , hence there are two options. One might argue that there could be multiple tiles $\mathbf{T}_1 \in \mathcal{T}_1$ such that $\mathbf{g} \in \mathbf{T}_1$, however there is only one \mathbf{T}_1 capable of forming \mathbf{T} with one other tile \mathbf{T}_2 from the other tile-set because the shape of the tiles is fixed for \mathcal{T}_1 , \mathcal{T}_2 and \mathcal{T} . The same reasoning can, of course, be used to show that there is only one suitable $\mathbf{T}_2 \in \mathcal{T}_2$.

When multiple options exist a choice must be made. This choice does not influence the cost of the resulting calculation graph. The cost of merging, after all, depends only on the size of the tiles, which is not influenced by the choice. The choice, however, must be made in a consistent manner. This ensures that always the same calculation graph is generated from a generator graph. This, in turn, ensures that the implementation based on the generator graph will correspond to the calculation graph.

To make such a consistent choice we introduce an ordering on the grid locations and the tile-sets. The grid locations are ordered by sorting them in ascending order on their first coordinate, then on their second coordinate, etc. The tile-sets are ordered first on the size of the shape then on their phase and subsequently on their period. Whenever a choice has to be made we make it such that the tile containing the grid location with the lowest rank of the resulting tile ($\mathbf{g} \in \mathbf{T}$) comes from the tile-set with the lowest rank. In most cases it is possible to make a consistent choice, but the example of Section 4.1 shows a case where this is not possible: $\text{tile}(\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$. Consider the lowest coordinates of the tiles from $\text{tile}(\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$ if this coordinate is even the tile can only be formed by taking a tile containing this coordinate from $\text{tile}(\blacksquare, 0, 1)$, but when this coordinate is odd the tile can only be formed by taking a tile containing this coordinate from $\text{tile}(\blacksquare\blacksquare\blacksquare, 1, 2)$.

To solve this problem we introduce a canonical form such that all choices during the generation of a calculation graph from generator graphs in canonical form can be made in a consistent manner. The canonical form entails nothing more than that all the nodes in the generator graph should have the same period and that this period should be as small as possible. We call this period the meta-period of the graph and it can be calculated by taking the least common multiple of the periods of all the nodes of a generator graph (that has minimum periods for all its tile-sets). This meta-period also plays an important role in actually implementing the design in Chapter 7.

Note that it is always possible to get the canonical form of a generator graph since nodes that do not have the required period can simply be duplicated a number of times. We have done this for the generator graph from Figure 4.1 and the result is shown in Figure 4.2.

A major benefit of this canonical form is that the the choices in generating the calculation graph can now be made consistently. For example the tiles from node $(\blacksquare\blacksquare\blacksquare\blacksquare, 0, 2)$ in Figure 4.2 are always constructed by a tile from $(\blacksquare, 0, 2)$ followed by a tile from $(\blacksquare\blacksquare\blacksquare, 1, 2)$. Conversely the tiles from node $(\blacksquare\blacksquare\blacksquare\blacksquare, 1, 2)$ are always constructed by a tile from $(\blacksquare\blacksquare\blacksquare, 1, 2)$ followed by a tile from $(\blacksquare, 1, 2)$. In contrast the tiles from the original graph in Figure 4.1 had to alternate between these construction methods and thus did not allow a consistent choice.

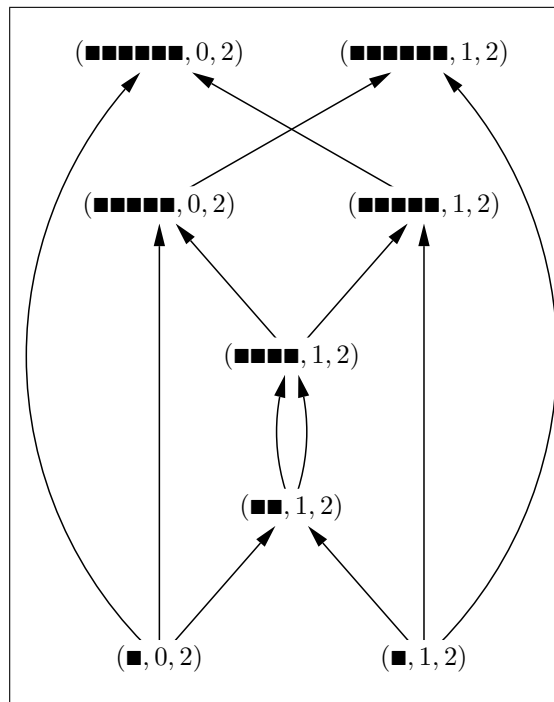


Figure 4.2: Figure 4.1 in canonical form

4.3 Cost calculation

We can calculate the cost of a calculation graph generated by a generator graph without actually generating the calculation graph itself. Take Figure 4.2 for example; the fact that node $(\blacksquare\blacksquare, 1, 2)$ has incoming edges from $(\blacksquare, 0, 2)$ and $(\blacksquare, 1, 2)$ means that it represents the merging of two tiles of size 1 and that this merging occurs with a period of 2. To calculate the cost of merging two tiles of size 1 we need to know the range of ranks needed from the resulting tile. This range of ranks can be calculated using (3.4) and, for the sake of simplicity, we look at the window sorting problem here, where all ranks are needed. Then the merging of two tiles of size 1 requires 2 extrema operations. Because the period of the windows is 1 and the period of these mergings is 2 we know that this cost of the merging operation can be amortized over 2 windows. Hence the contribution of this merging operation to the total is 1 epw. In a similar manner we can calculate the contribution of all the nodes in the generator graph to the total cost. For Figure 4.2 this has been done in Table 4.1.

Note that, besides not having to actually generate the calculation graph to calculate the cost, the generator graph has another advantage; we are able to avoid taking limits as required by definition (3.8).

| Node | Cost per operation | Contribution |
|-----------------------------|--------------------|--------------|
| (■■■■■■■, 0, 2) | 10 | 5 |
| (■■■■■■■, 1, 2) | 10 | 5 |
| (■■■■■, 0, 2) | 8 | 4 |
| (■■■■■, 1, 2) | 8 | 4 |
| (■■■■, 1, 2) | 6 | 3 |
| (■■, 1, 2) | 2 | 1 |
| (■, 0, 2) | 0 | 0 |
| (■, 1, 2) | 0 | 0 |
| Total cost: (in epw) | | 22 |

Table 4.1: Cost of the generator graph in Figure 4.2

4.4 Constructing generator graphs

Chapter 5 describes, in detail, several heuristic algorithms for constructing generator graphs. In this section we discuss the framework used by some of the algorithms from that chapter and indicate how other heuristic algorithms can be designed.

The algorithms for constructing a generator graph start with an empty generator graph (\emptyset, \emptyset) and add nodes and edges until a generator graph is formed such that the calculation graph derived from it is valid.

To add a node and edges to the generator graph the **add** function is used, which returns the updated graph:

$$\text{add}(G_g, \mathbb{T}, \mathbb{T}_1, \mathbb{T}_2) = (V_g \cup \{\mathbb{T}\}, E_g \cup \{(\mathbb{T}_1, \mathbb{T}), (\mathbb{T}_2, \mathbb{T})\}) \quad (4.7)$$

This adds the node \mathbb{T} to the set of nodes and adds incoming edges (from \mathbb{T}_1 and \mathbb{T}_2) to the graph. Note that for the resulting graph to be valid the construction of the tiles in \mathcal{T} from the tiles in \mathcal{T}_1 and \mathcal{T}_2 must be possible, that \mathbb{T}_1 and \mathbb{T}_2 must be in the set of nodes, and that all the triples have to be in canonical form. Moreover, recall that E_g is a multi-set, so in case $\mathbb{T}_1 = \mathbb{T}_2$ two edges are added.

Some algorithms create a generator graph in a top down fashion. These algorithms start by defining the construction of the windows (specified by the ROFDP) and then recursively define the construction method of the window's components.

The top down approach is based on two functions. The first one is a function that specifies how tiles from an existing tile-set should be split into two sub-tiles from (possibly different) tile-sets and which (possibly other) function specifies the construction method of these new tile-sets. This function forms the core of the heuristic algorithm and we therefore call it the heuristic function. The second function used by this approach is the function that recursively calls the heuristic function and adds the results to the generator graph.

The method of construction of a tile set is defined by the heuristic function, say \mathbf{f} . The function differs for each algorithm, but all functions return the same kind of tuple: $\mathbf{f}(\mathbb{T}) = ((\mathbb{T}_1, \mathbf{f}_1), (\mathbb{T}_2, \mathbf{f}_2))$, where $\mathbb{T}_1 = \text{norm}(\mathbb{T}_1)$ and $\mathbb{T}_2 = \text{norm}(\mathbb{T}_2)$. The function effectively specifies that the tiles in tile set $\text{tile}(\mathbb{T})$ should be constructed from tiles in $\text{tile}(\mathbb{T}_1)$ and $\text{tile}(\mathbb{T}_2)$. It also specifies that the tile-sets $\text{tile}(\mathbb{T}_1)$ and $\text{tile}(\mathbb{T}_2)$ should be constructed in the manner

defined by the heuristic functions \mathbf{f}_1 and \mathbf{f}_2 respectively. For basic algorithms we have $\mathbf{f} = \mathbf{f}_1 = \mathbf{f}_2$, but more sophisticated algorithms switch heuristics at a certain point.

Although this first function differs per algorithm, there are enough similarities between algorithms to introduce the so-called **split** function. This function is used to split a tile at a specific point into two sub-tiles:

$$\mathbf{split}(\mathbb{T}, \mathbf{c}, \mathbf{p}_1, \mathbf{f}_1, \mathbf{p}_2, \mathbf{f}_2) = ((\mathbf{cut}(\mathbb{T}, \mathbf{c}, \mathbf{p}_1), \mathbf{f}_1), (\mathbf{rem}(\mathbb{T}, \mathbf{c}, \mathbf{p}_2), \mathbf{f}_2)) \quad (4.8)$$

The result returned by the **split** function adheres to the format for the heuristic function and is often used as a basis for such a function.

To split a tile into two sub-tiles the **split** function uses the **cut** and **rem** functions which respectively return the part of a tile closest to the origin and the remainder of the tile. More formally:

$$\mathbf{cut}(\mathbb{T}, \mathbf{c}, \mathbf{p}_{new}) = \mathbf{norm}((\mathbb{S}_{\mathbb{T}} \cap \mathbf{hyperbar}(\mathbf{c}), \mathbf{f}_{\mathbb{T}}, \mathbf{p}_{new})) \quad (4.9)$$

$$\mathbf{rem}(\mathbb{T}, \mathbf{c}, \mathbf{p}_{new}) = \mathbf{norm}((\mathbb{S}_{\mathbb{T}} \setminus \mathbf{hyperbar}(\mathbf{c}), \mathbf{f}_{\mathbb{T}}, \mathbf{p}_{new})) \quad (4.10)$$

$$(4.11)$$

The functions $\mathbf{cut}(\mathbb{T}, \mathbf{c}, \mathbf{p}_{new})$ and $\mathbf{rem}(\mathbb{T}, \mathbf{c}, \mathbf{p}_{new})$ therefore return normalized triples based on \mathbb{T} . The difference between the two functions is in the shape of the returned triple. The shape of the original triple is effectively cut in two pieces based on the cut-off point described by vector \mathbf{c} . The function **cut** returns the part closest to the origin, while the function **rem** returns the remainder. The triples returned by these functions have a period indicated by the vector \mathbf{p}_{new} .

The **hyperbar** function is used by the **cut** and **rem** functions to partition a tile. As its name implies the function does nothing more than returning a tile in the shape of a hyper-bar with dimensions indicated by vector \mathbf{s} :

$$\mathbf{hyperbar}(\mathbf{s}) = \{\mathbf{g} | \mathbf{0}_D \leq \mathbf{g} < \mathbf{s}\} \quad (4.12)$$

$$(4.13)$$

Let us examine $\mathbf{split}((\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1), 2, 1, \mathbf{f}_1, 2, \mathbf{f}_2)$ as an example. This formula specifies that the tiles from $(\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 2)$ have to be formed from tiles of size 2 and 4, with periods 1 and 2 respectively:

$$\begin{aligned} & \mathbf{split}((\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 2), 2, 1, \mathbf{f}_1, 2, \mathbf{f}_2) \\ = & \quad \{\text{Definition of } \mathbf{split}\} \\ & ((\mathbf{cut}((\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 2), 2, 1), \mathbf{f}_1), (\mathbf{rem}((\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 2), 2, 2), \mathbf{f}_2)) \\ = & \quad \{\text{Definition of } \mathbf{cut}, \mathbf{rem} \text{ and } \mathbf{hyperbar}\} \\ & ((\mathbf{norm}((\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare \cap \blacksquare\blacksquare, 0, 1)), \mathbf{f}_1), (\mathbf{norm}((\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare \setminus \blacksquare\blacksquare, 0, 2)), \mathbf{f}_2)) \\ = & \quad \{\text{Set arithmetic}\} \\ & ((\mathbf{norm}((\blacksquare\blacksquare, 0, 1)), \mathbf{f}_1), (\mathbf{norm}((\square\square\blacksquare\blacksquare\blacksquare, 0, 2)), \mathbf{f}_2)) \\ = & \quad \{\text{Rewrite to canonical form}\} \\ & (((\blacksquare\blacksquare, 0, 1), \mathbf{f}_1), ((\blacksquare\blacksquare\blacksquare\blacksquare, 2, 2), \mathbf{f}_2)) \end{aligned}$$

Note that the **split** function does nothing to ensure correctness. We could also have used the example:

$\text{split}(\text{■■■■■■}, 0, 1), 2, 3, \mathbf{f}_1, 2, \mathbf{f}_2) = (((\text{■■}, 0, 3), \mathbf{f}_1), ((\text{■■■■}, 2, 2), \mathbf{f}_2))$, which would result in a generator graph that is in violation of (4.5). So `split` is only a notational shorthand for our algorithms. That an algorithm provides us with a correct generator graph has to be shown for each algorithm individually.

The other major part of the framework for top-down algorithms is the function that recursively calls the heuristic function until the construction method of every tile has been specified. This function is called `TD`:

$$\text{TD}(G_g, \mathbb{T}, \mathbf{f}) = \begin{cases} G_g & \mathbb{T} \in V_g \\ (V_g \cup \{\mathbb{T}\}, E_g) & \mathbb{T} \notin V_g \wedge |\mathbf{S}_{\mathbb{T}}| = 1 \\ \text{TD}(\text{TD}(\text{add}(G_g, \mathbb{T}, \mathbb{T}_1, \mathbb{T}_2), \mathbb{T}_1, \mathbf{f}_1), \mathbb{T}_2, \mathbf{f}_2) & \text{otherwise} \end{cases}$$

where $((\mathbb{T}_1, \mathbf{f}_1), (\mathbb{T}_2, \mathbf{f}_2)) = \mathbf{f}(\mathbb{T})$ (4.14)

This function returns an updated generator graph by taking a generator graph G_g by adding the triple \mathbb{T} to it (we assume that $\mathbb{T} = \text{norm}(\mathbb{T})$). This explains the first two cases in (4.14); in the first case the triple \mathbb{T} is already part of the generator graph and thus nothing has to be done. In the second case the triple \mathbb{T} is not yet part of the generator graph, so it has to be added, and since it is of size 1 it requires no incoming edges to describe its construction. The last case is more complicated, not only must the triple \mathbb{T} be added as a node, also two edges need to be added that describe how the tiles in the tile set (\mathbb{T}) must be constructed. The particular manner in which this construction takes place is described by the heuristic function \mathbf{f} , which (for most algorithms) is based on the `split` function described earlier.

Chapter 5

Heuristic Algorithms

In the previous chapter the problem of designing an efficient ROF has been reduced to the problem of finding a generator graph with low cost. In this chapter we introduce several heuristic algorithms capable of finding such low-cost generator graphs. Finding optimal ROFs of small size can be done by a brute force algorithm, just as optimal sorting networks have been found for small sorting problems [7], but for larger problems the number of possible generator graphs explodes. Therefore we resort to heuristic algorithms.

In the next few sections we develop several heuristic algorithms for (mainly one dimensional) ROFDPs using the method sketched in Chapter 4. This means that we introduce various ways of splitting tile-sets. We also show how an algorithm designed for a one dimensional ROFDP can be generalized to multiple dimensions.

5.1 Preliminaries

In the following sections several heuristic algorithms to construct generator graphs are presented. The way these algorithms work is illustrated using two running examples. The first example concerns the design of a 1-dimensional window sorter for window size 6, i.e. window-set $\mathcal{W} = \text{tile}(\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$ and $l = 0$ and $u = 5$. The second example concerns a 1-dimensional window sorter for window size 7, i.e. window-set $\mathcal{W} = \text{tile}(\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$ and $l = 0$ and $u = 6$. Furthermore we introduce three functions that are frequently used in the definition of the heuristics. They are:

$$\text{pow2}(a) = (\exists i : i \in \mathbb{N} : 2^i = a) \quad (5.1)$$

$$\text{msb}_1(a) = (\uparrow i : 2^i < a : 2^i) \quad (5.2)$$

$$\text{lsb}_1(a) = (\uparrow i : 2^i \mid a : 2^i) \quad (5.3)$$

These functions respectively return: a boolean value that indicates whether a is a power of two and the power of two represented by the most-significant 1-bit of a and the least-significant 1-bit of a .

An operation which is used often by the algorithms is selecting the triple describing tiles with the largest or smallest shape from a set. To indicate these

we use the following notation:

$$\begin{aligned} \downarrow Q & \text{ for an element of } Q \text{ such that } |\mathbf{S}_{(\downarrow Q)}| = (\downarrow q : q \in Q : |\mathbf{S}_q|) \\ \uparrow Q & \text{ for an element of } Q \text{ such that } |\mathbf{S}_{(\uparrow Q)}| = (\uparrow q : q \in Q : |\mathbf{S}_q|) \end{aligned} \quad (5.4)$$

5.2 Divide and Conquer

This algorithm is based on the observation that, when using Batcher’s merging networks, the cheapest way to obtain a sorted list of size s is by merging sorted lists of size: $\lceil \frac{s}{2} \rceil$ and $\lfloor \frac{s}{2} \rfloor$. This is a simple divide and conquer approach leading to the construction:

$$G_g = \text{TD}((\emptyset, \emptyset), \text{norm}(W), \text{dac})$$

where the heuristic function **dac** is specified by:

$$\begin{aligned} G_g &= \text{TD}((\emptyset, \emptyset), \text{norm}(W), \text{dac}) \\ \text{dac}(T) &= \text{split}(T, \lceil \frac{|\mathbf{S}_T|}{2} \rceil, \mathbf{p}_T, \text{dac}, \mathbf{p}_T, \text{dac}) \end{aligned}$$

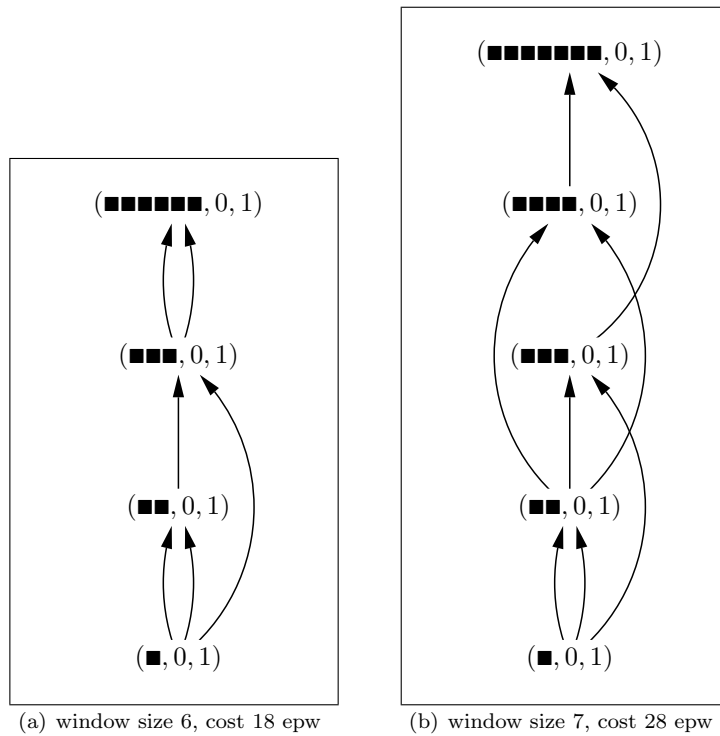
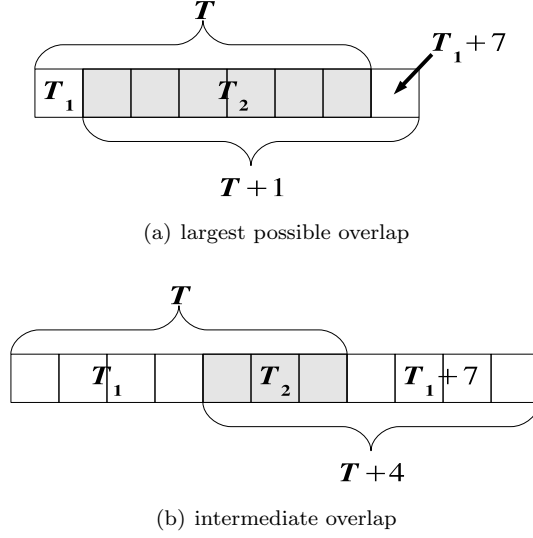


Figure 5.1: Generator graphs produced by the **dac**-heuristic

The generator graphs produced by this algorithm for our running examples are shown in Figure 5.1. The resulting rank order filters require 18 epw (extrema operations per window) and 28 epw for sorting windows of size of 6 and 7 respectively. Since each tile-set has a period equal to the period of the windows the **dac**-heuristic does not reuse any tiles.


 Figure 5.2: Example of overlapping for $T = (\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$

5.3 Recursive Tiling

The divide and conquer algorithm from the previous section provides generator graphs in which no reuse takes place. Before we introduce an algorithm that generates generator graphs with reuse, let us examine a possible way in which reuse can take place.

A solid tile $T \in \mathcal{T}$ can be formed by combining two tiles T_1 and T_2 with $(\forall \mathbf{g}_1, \mathbf{g}_2 : \mathbf{g}_1 \in T_1 \wedge \mathbf{g}_2 \in T_2 : \mathbf{g}_1 < \mathbf{g}_2)$. When $|T_1|$ is a multiple of \mathbf{p}_T then T_2 can be reused in the construction of $T + |T_1|$ like so: $[T + |T_1|] = [T_2] \bowtie [T_1 + |T|]$. Two examples of this are shown in Figure 5.2.

Note that the part that could be reused in the divide and conquer algorithm is relatively small; for any T the reusable part would have size $\lfloor \frac{|T|}{2} \rfloor$. So, instead of adopting the divide and conquer algorithm so that it exhibits reuse, we immediately turn to an algorithm that maximizes the size of the reused parts.

This algorithm simply splits a tile in a remainder and in an overlapping part. In the first step the remainder will be of size one and the overlapping part of size $|\mathbf{S}_W| - 1$. This overlapping part then has a period of 2, so this remainder is split into a part of size 2 and a part of size $|\mathbf{S}_W| - 3$. More formally this algorithm is described by the heuristic function \mathbf{rts} :

$$\mathbf{rts}(c, m)(T) = \begin{cases} \mathbf{split}(T, c, \mathbf{p}_T, \mathbf{dac}, (2\mathbf{p}_T) \downarrow m, \mathbf{rts}(2c, m)) & c < |\mathbf{S}_T| \\ \mathbf{dac}(T) & c = |\mathbf{S}_T| \\ \mathbf{rtm}(T) & c > |\mathbf{S}_T| \end{cases}$$

$$\mathbf{rtm}(T) = \mathbf{split}(T, \mathbf{msb}_1(|\mathbf{S}_T|), \mathbf{p}_T, \mathbf{dac}, \mathbf{p}_T, \mathbf{rtm})$$

The \mathbf{rts} function has two parameters: c and m . The parameter c equals the size of the tile that is split of. The algorithm starts with $c = 1$ and, since c can

only be doubled, this means that c is always a power of two. Conceptually the period of the overlapping part will double in each recursive step. However, to make sure that the generator graph produced by the algorithm is valid we need to limit the period at a certain value. This limit is indicated by the parameter m . Since a higher period allows for more reuse this parameter will be set as high as possible, but low enough to ensure that a valid generator graph is produced. This leads to $m = 2\text{lsb}_1(|\mathbf{S}_W| + 1)$.

The `rts` function recursively calls itself to split the tiles, but at a certain stage c will be larger or equal to the size of the tile that is under construction. In these cases we have to switch heuristics. The `dac` heuristic is a possible choice, but while using the `rts` heuristic a lot of tiles with sizes that are a power of two have been generated. We can reuse these tiles by switching to the correct heuristic. When c , which is a power of two, equals the tile size the `dac` heuristic will reuse them automatically since all the tiles it generates will be powers of two. In the case where c is larger than the tile size we switch to the `rtm` heuristic to make sure that we construct the tile from tiles whose size are a power of two.

This heuristic performs well for odd window sizes, however for even window sizes we have $m = 2\text{lsb}_1(|\mathbf{S}_W| + 1) = 2$, which severely limits reuse. We therefore take a special step in the case of even window sizes, defined by the `rtf` heuristic:

$$\text{rtf}(\mathbf{T}) = \begin{cases} \text{split}(\mathbf{T}, 1, \mathbf{p}_T, -, \mathbf{p}_T, \text{rts}(1, 2\text{lsb}_1(|\mathbf{S}_T|))) & |\mathbf{S}_T| \bmod 2 = 0 \\ \text{rts}(1, 2\text{lsb}_1(|\mathbf{S}_T| + 1))(\mathbf{T}) & |\mathbf{S}_T| \bmod 2 = 1 \end{cases}$$

The generator graph produced by the recursive tiling algorithm for one dimensional problems is therefore defined by:

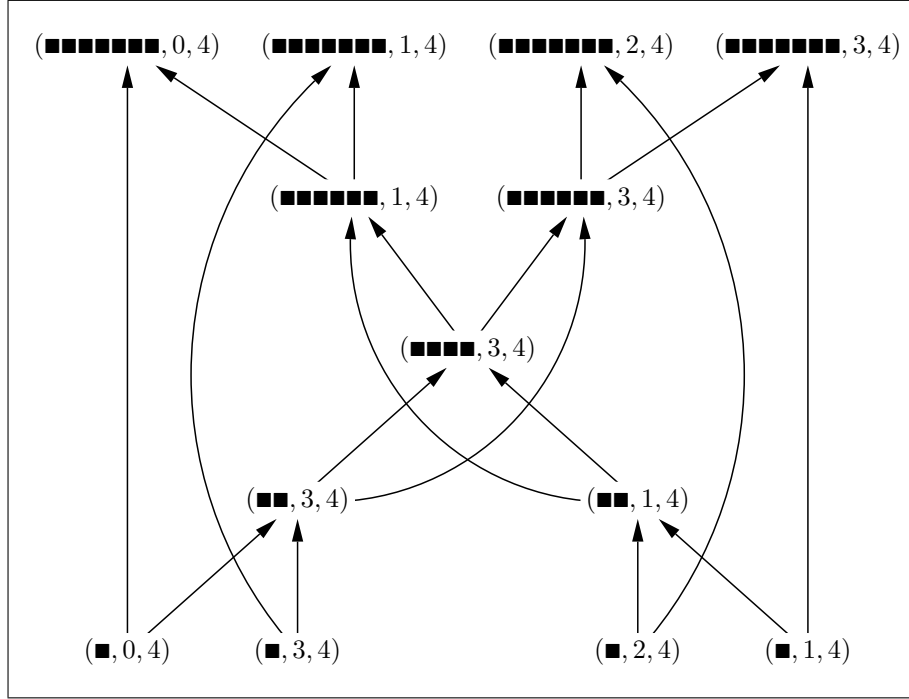
$$G_g = \text{TD}((\emptyset, \emptyset), (\text{norm}(W), \text{rtf}))$$

The result of this algorithm for our running examples can be seen in figures 4.2 and 5.3. The cost for both solutions is respectively 22 and 20.5 epw for window sizes 6 and 7. So for windows of size 7 this algorithm is better than the divide and conquer algorithm. In fact, for most problems this algorithm performs better than the divide and conquer algorithm. The reason for this is the reuse; for the problem with window size 7 the tiles of size 2, 4 and 6 can be reused, with the tile of size 4 even being used for 4 different windows. It is this amount of reuse that allows the recursive tiling algorithm to improve upon the divide and conquer algorithm. Unfortunately, only the tiles of size 2 and 4 can be reused for the problem with window size 6 and this reuse is not enough to improve upon the divide and conquer algorithm.

5.4 Bit-pattern Tiling

Our experiments with the heuristic from the previous section shows that power of two tiles (i.e. tiles with a size that is a power of two) can be created efficiently using the `dac`-heuristic. In this section we therefore present an algorithm that produces windows based on such power of two tiles. Since the type of power of two tiles depends on the pattern of bits representing the window size, we have named this heuristic "Bit-pattern tiling".

This algorithm comes in two variants; one that splits off the least significant power of two from an existing tile, and one that splits of the most significant

Figure 5.3: G_g for the Recursive Tiling algorithm for window size 7 (20.5 epw)

power of two. In effect we either iterate through the bit-pattern from left to right or from right to left.

The first variant of the algorithm is called "Least Significant Bit Tiling" or "LSB Tiling" and is defined as follows:

$$G_g = \text{TD}((\emptyset, \emptyset), \text{norm}(W), \text{lsbt})$$

where the heuristic function lsbt is specified by:

$$\text{lsbt}(T) = \begin{cases} \text{dac}(T) & \text{pow2}(|S_T|) \\ \text{split}(T, \text{lsb}_1(|S_T|), p_T, \text{lsbt}, 2p_T, \text{lsbt}) & \neg \text{pow2}(|S_T|) \wedge p_T = \text{lsb}_1(|S_T|) \\ \text{split}(T, \text{lsb}_1(|S_T|), p_T, \text{lsbt}, p_T, \text{lsbt}) & \text{otherwise} \end{cases}$$

The second variant is called "Most Significant Bit Tiling" or "MSB Tiling":

$$G_g = \text{TD}((\emptyset, \emptyset), \text{norm}(W), \text{msbt})$$

where the heuristic function msbt is specified by:

$$G_g = \text{TD}((\emptyset, \emptyset), (\text{norm}(W), \text{msbt}))$$

$$\text{msbt}(T) = \begin{cases} \text{dac}(T) & \text{pow2}(|S_T|) \\ \text{split}(T, \text{msb}_1(|S_T|), p_T, \text{msbt}, 2p_T, \text{msbt}) & \neg \text{pow2}(|S_T|) \wedge p_T = \text{msb}_1(|S_T|) \\ \text{split}(T, \text{msb}_1(|S_T|), p_T, \text{msbt}, p_T, \text{msbt}) & \text{otherwise} \end{cases}$$

Both algorithms consists of three cases. The first case applies only to tiles that have a size that is a power of two. Those tiles are constructed using the `dac`-heuristic. The other two cases are quite similar; a piece of the tile is split off and the size of this piece is the least- and most-significant bit of the tile's size for `lsbt` and `msbt` respectively. The only difference between the two cases is the period of the part that is split off. If possible, i.e. if we still end up with a valid construction, this period is double the period of the original tile since it can be reused in that case.

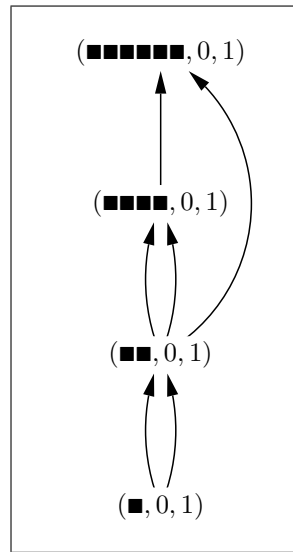


Figure 5.4: G_g for the Bit-Pattern Tiling 1 and 2 algorithms for window size 6 (20 epw)

Both algorithms produce the same generator graph for our running example with window size 6 (see Figure 5.4), which costs 20 epw. However, there is very little reuse in this solution, so although the new algorithms improve upon the recursive tiling algorithm, they do not improve upon the divide and conquer algorithm. The LSB tiling algorithm produces, for windows of size 7, the same graph as the recursive tiling algorithm (shown in Figure 5.3) and thus has the same cost (20.5 epw). The MSB tiling algorithm also produces an already familiar graph for windows of size 7, namely the same one produced by the divide an conquer algorithm (Figure 5.1(b)) which has a cost of 28 epw.

5.5 Split Tiling

The fact that power of two tiles can be formed relatively cheaply is exploited in the heuristic from the previous section, but not to its full extent. In this section we propose a heuristic that will form each window from a set of power of two tiles, just like the Bit-pattern tiling heuristic from the previous section, but there will be more (possible) orders in which these power of two tiles are combined. Most notably we make use of the fact that the merge operation is

commutative (see Section 3.1), which allows us to construct tiles which are split by "holes". Hence we call this heuristic "Split Tiling".

This heuristic assumes that the period of the windows is one ($\mathbf{p}_W = 1$) and that the shape of the tile is solid to begin with, i.e. $\mathbf{S}_W = \{0, \dots, |\mathbf{S}_W| - 1\}$.

The algorithm works by first creating all tiles with a size that is a power of two and that have a period equal to their size. This can be done at a very low cost.

$$\begin{aligned} G &= \text{TD}((\emptyset, \emptyset), (\{0, \dots, \text{msb}(|\mathbf{S}_W|) - 1\}, 0, \text{msb}_1(|\mathbf{S}_W|), \text{sth})) \\ \text{sth}(\mathbb{T}) &= \text{split}(\mathbb{T}, |\mathbf{S}_\mathbb{T}|/2, \mathbf{p}_\mathbb{T}/2, \text{sth}, \mathbf{p}_\mathbb{T}/2, \text{sth}) \end{aligned}$$

This only generates part of the final generator graph. The second step is to use the tiles in this partial generator graph to form the window. Since the meta-period of the graph created so far is $\text{msb}_1(|\mathbf{S}_W|)$, there are $\text{msb}_1(|\mathbf{S}_W|)$ different combinations of tiles that form a window; one combination for each phase in the period $\text{msb}_1(|\mathbf{S}_W|)$. The combination of tiles that form the window at phase i is expressed by $\text{comb}(i)$:

$$\begin{aligned} \text{comb}(f) &= \text{cr}(|\mathbf{S}_W|, f) \\ \text{cr}(s, f) &= \text{ct}(\text{gt}(\text{msb}_1(s), f)) \cup \text{cr}(s - \text{gt}(\text{msb}_1(s), f), f + \text{gt}(\text{msb}_1(s), f)) \\ \text{ct}(s, f) &= \{(\{0, \dots, s - 1\}, f, s)\} \\ \text{gt}(s, f) &= \begin{cases} s & f \bmod s = 0 \\ \text{gt}(s/2, f) & f \bmod s \neq 0 \end{cases} \end{aligned}$$

The function gt returns the size of the largest tile smaller than or equal to s that starts at phase f . This function is then used by the recursive function cr to find the set of triples describing the tiles that will form the window. The function cr returns the set of tiles that form a tile of size s at phase f . The function ct is used to construct the actual triple, based on its size. Note that this triple is not normalized. This is done to make sure that the set returned by cr contains one triple for each tile used in the construction of the window. For example: constructing the tiles from $\text{tile}(\blacksquare\blacksquare\blacksquare\blacksquare, 1, 4)$ requires tiles from $\text{tile}(\blacksquare, 1, 1)$, $\text{tile}(\blacksquare\blacksquare, 2, 2)$ and $\text{tile}(\blacksquare\blacksquare, 4, 2)$. If we would normalize the triples we would end up with only two distinct triples, and the implicit information in the phase would be destroyed too.

Now, the order in which those tiles are combined can be varied, therefore we have made three variants of this heuristic. These variants are named "Split Tiling 1", "Split Tiling 2" and "Split Tiling 3" respectively. Each of the split tiling algorithm variants generates a partial generator graph G_i based on $\text{comb}(i)$ for $0 \leq i < \text{msb}_1(|\mathbf{S}_W|)$. We then combine all the partial generator graphs to create the final generator graph like so:

$$G_g = G_b \cup (\cup i : 0 \leq i < \text{msb}_1(|\mathbf{S}_W|) : G_i)$$

To generate the partial graphs G_i all the variants of this heuristic have to merge the tiles in $\text{comb}(i)$ until the entire window is formed. This bottom up

approach can be expressed as follows:

$$\begin{aligned} \text{ST}(G_i, S, \mathbf{f}) &= \begin{cases} G_i & |S| = 1 \\ \text{ST}(\text{ma}(G_i, \mathbb{T}_1, \mathbb{T}_2), \text{sup}(S, \mathbb{T}_1, \mathbb{T}_2), \mathbf{f}_1) & \text{otherwise} \end{cases} \\ &\text{where } (\mathbb{T}_1, \mathbb{T}_2, \mathbf{f}_1) = \mathbf{f}(S) \\ \text{ma}(G_i, \mathbb{T}_1, \mathbb{T}_2) &= \text{add}(G_i, \text{norm}(\text{mt}(\mathbb{T}_1, \mathbb{T}_2)), \text{norm}(\mathbb{T}_1), \text{norm}(\mathbb{T}_2)) \\ \text{mt}(\mathbb{T}_1, \mathbb{T}_2) &= (\text{ms}(\mathbb{T}_1, \mathbb{T}_2), \mathbf{f}_{\mathbb{T}_1}, \text{msb}_1(|\mathbf{S}_{\mathbb{W}}|)) \\ \text{ms}(\mathbb{T}_1, \mathbb{T}_2) &= \mathbf{S}_{\mathbb{T}_1} \cup (\mathbf{S}_{\mathbb{T}_2} + \mathbf{f}_{\mathbb{T}_2} - \mathbf{f}_{\mathbb{T}_1}) \\ \text{sup}(S, \mathbb{T}_1, \mathbb{T}_2) &= (S \setminus \{\mathbb{T}_1, \mathbb{T}_2\}) \cup \{\text{mt}(\mathbb{T}_1, \mathbb{T}_2)\} \end{aligned}$$

The ST function generates the partial generator graph G_i by repeatedly taking two tiles from the set S , merging them and adding them to G_i . This will eventually lead to a single tile in the set S (the window at phase i), which indicates that the partial graph is finished. If the recursion is not yet finished the ST function uses the function ma to merge two tiles and add the new tile and its construction method to G_i . It does this by using mt which merges the tile, which in turn uses ms to merge the shape of the tiles. The function sup is used to update the set of tiles that still have to be merged.

The selection of the tiles from the set S is determined by the selection function \mathbf{f} , which differs per variant of the split tiling algorithm. The function $\mathbf{f}(S) = (\mathbb{T}_1, \mathbb{T}_2, \mathbf{f}_1)$ returns a triple with $\mathbb{T}_1 \in S$ and $\mathbb{T}_2 \in S$ indicating the tiles to be merged and \mathbf{f}_1 indicating a (possibly) new selection function for the next step in the algorithm.

So, in contrast to the algorithms discussed so far the "Split Tiling" approach is bottom-up; given some tiles it constructs the windows. Whereas the other algorithms are top-down approaches; given the window they decide in which way it should be split.

The differences between the variants of the "Split Tiling" algorithm lie in the selection function and for "Split Tiling 1" we have $G_i = \text{ST}((\emptyset, \emptyset), \text{comb}(i), \text{st1})$:

$$\text{st1}(S) = (\downarrow S, \downarrow(S \setminus \{\downarrow S\}), \text{st1})$$

So "Split Tiling 1" simply keeps on merging the smallest tiles in the set, until the entire window has been constructed.

For "Split Tiling 2" we have $G_i = \text{ST}((\emptyset, \emptyset), \text{comb}(i), \text{st2}(i))$:

$$\begin{aligned} \text{st2}(i)(S) &= \begin{cases} \text{st2a}(i)(S) & (\exists \mathbb{T}_1, \mathbb{T}_2 : \mathbb{T}_1, \mathbb{T}_2 \in (S \cap \text{comb}(i)) \wedge \\ & \mathbb{T}_1 \neq \mathbb{T}_2 : |\mathbf{S}_{\mathbb{T}_1}| = |\mathbf{S}_{\mathbb{T}_2}|) \\ \text{st1}(S) & \text{otherwise} \end{cases} \\ \text{st2a}(i)(S) &= (\mathbb{T}_1, \mathbb{T}_2, \text{st2}) \text{ such that:} \\ |\mathbf{S}_{\mathbb{T}_1}| &= |\mathbf{S}_{\mathbb{T}_2}| \\ \mathbb{T}_1, \mathbb{T}_2 &\in (S \cap \text{comb}(i)) \\ \mathbb{T}_1 &\neq \mathbb{T}_2 \\ |\mathbf{S}_{\mathbb{T}_1}| &= (\downarrow \mathbb{T}_3, \mathbb{T}_4 : \mathbb{T}_3, \mathbb{T}_4 \in (S \cap \text{comb}(i)) \wedge \\ & \mathbb{T}_3 \neq \mathbb{T}_4 \wedge |\mathbf{S}_{\mathbb{T}_3}| = |\mathbf{S}_{\mathbb{T}_4}| : |\mathbf{S}_{\mathbb{T}_3}|) \end{aligned}$$

This variant first merges all the initial tiles of the same size, and only after that it starts merging the smallest tiles until the entire window is formed. The idea

is that Batchner's merging networks are more efficient when lists of the same size are merged.

For "Split Tiling 3" we have $G_i = ST((\emptyset, \emptyset), \text{comb}(i), \text{st3})$:

$$\begin{aligned} \text{st3}(S) &= \begin{cases} \text{st3a}(S) & (\exists T_1, T_2 : T_1, T_2 \in S \wedge T_1 \neq T_2 : |S_{T_1}| = |S_{T_2}|) \\ \text{st3b}(S) & \text{otherwise} \end{cases} \\ \text{st3a}(S) &= (T_1, T_2, \text{st3}) \text{ such that} \\ |S_{T_1}| &= |S_{T_2}| \wedge T_1, T_2 \in S \wedge T_1 \neq T_2 \\ |S_{T_1}| &= (\downarrow T_3, T_4 : T_3, T_4 \in S \wedge T_3 \neq T_4 \wedge |S_{T_3}| = |S_{T_4}| : |S_{T_3}|) \\ \text{st3b}(S) &= ((\uparrow S, \uparrow(S \setminus \{\uparrow S\})), \text{st3b}) \end{aligned}$$

This variant first merges all the tiles of the same size in S . If all the tiles have different sizes it simply starts merging the largest two tiles.

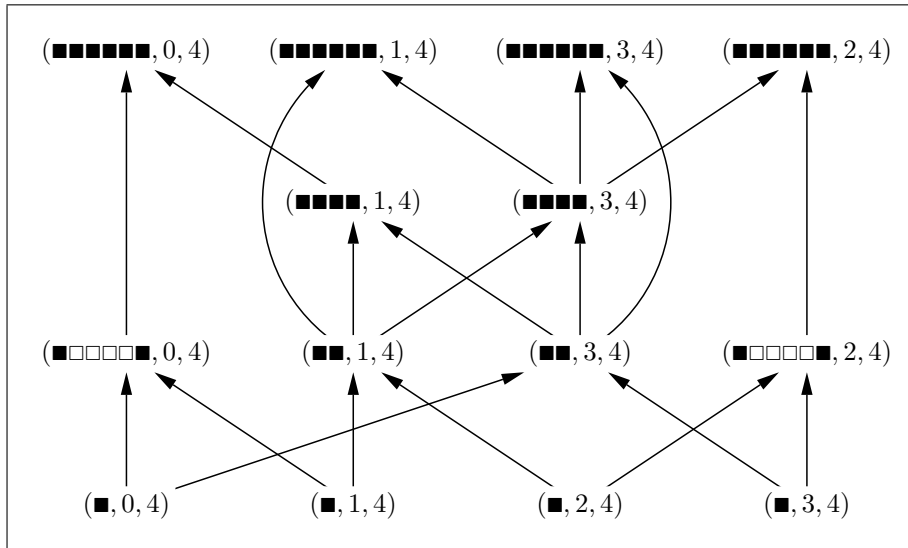


Figure 5.5: G_g for the Split Tiling 1,2 and 3 algorithms for window size 6 (17 epw)

For windows of size 6 all three variants of the algorithm produce the same generator graph (depicted in Figure 5.5), with a cost of 17 epw. This is an improvement over all the algorithms we have seen so far. This is not surprising since this is the first design for this problem in which tiles of size 4 are reused. For windows of size 7 the first variant produces the generator graph in Figure 5.6 with a cost of 22.5 epw and the other two variants produce the same graph as the recursive tiling algorithm (Figure 5.3) with a cost of 20.5 extrema operations.

5.6 Overlap Recursive Mirroring

So far our algorithms have attempted to maximize reuse by following some algorithm based on the shape of the windows. However, the cost of the generator

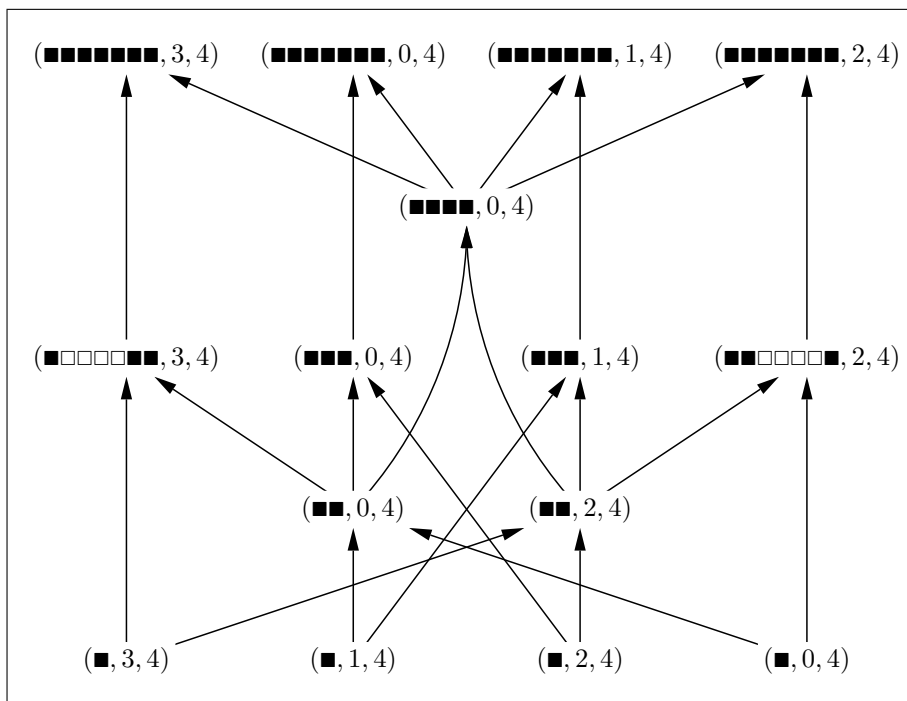


Figure 5.6: G_g for the Split Tiling 1 algorithm for window size 7 (22.5 epw)

graph also depends on l and u and we can obtain better results by taking a look at the possible ways of reuse and selecting the best one based on the cost function. This is what the "Overlap Recursive Mirroring" algorithm does.

For every tile it has to construct the heuristic checks whether that tile is a hyper-bar and whether it overlaps with the next (periodic) version of itself. If this is the case the overlapping part is a good candidate for an intermediate tile. Of course, a tile may overlap with multiple versions of itself. The overlapping part of multiple versions is smaller than the overlapping part of two single versions, but the former can be reused multiple times. This is a difficult trade-off and the "ORM" algorithm makes this trade-off by simply trying a number of possibilities and selecting the one with the lowest cost.

To keep the calculation time under control we introduce the parameter R which indicates the maximum number of times an overlap is re-used. So, for $R = 0$ the algorithm only considers constructing the tile without making use of any overlap. For $R = 1$ it also considers constructing a tile by re-using the overlap between two consecutive occurrences of a tile. For $R = 2$ it considers all the possibilities of $R = 1$ and the possibility of re-using the overlap between three consecutive occurrences of a tile. And so forth for larger values of R .

The function $ovl(T)$ returns the set of possible manners in which tiles from $tile(T)$ can be overlapped with periodic versions of itself, given that the tile is a hyper-bar.

$$ovl(R, T) = \{g | g \in hyperbar(R, 1_D) \wedge (S_T \cap (g \cdot p_T + S_T)) \neq \emptyset\}$$

So each element of the returned set is a vector. This vector no longer represents a coordinate on the grid, but a way in which the tiles overlap with other tiles from the same tile-set. So $(0, 0)$ means no overlapping, $(1, 0)$ means that a tile overlaps with the next tile in the first dimension, but that there is no overlapping in the second dimension. This function is used to generate the set of overlaps considered by the algorithm. Note that the parameter R is used here to keep the number of possibilities in check.

Note that we only apply the `ovl` function when \mathbf{S}_T is a hyper bar. We do this because this makes it possible to implement the `ovl` function in a fast manner in practice. Whenever the tile is not a hyper-bar we only consider the overlaps represented by the set $\{\mathbf{0}_D\}$.

Other functions that deal with the overlapping part are:

$$\begin{aligned} \text{os}(T, \mathbf{o}) &= (\cap \mathbf{g} : \mathbf{g} \in \text{hyperbar}(\mathbf{o} + \mathbf{1}_D) : \mathbf{S}_T + \mathbf{g} \cdot \mathbf{p}_T) \\ \text{op}(T, \mathbf{o}) &= (\mathbf{o} + \mathbf{1}_D) \cdot \mathbf{p}_T \\ \text{ot}(T, \mathbf{o}) &= \text{norm}(\text{os}(T, \mathbf{o}), \mathbf{f}_T, \text{op}(T, \mathbf{o})) \end{aligned}$$

In these functions the parameter \mathbf{o} is an element of the set produced by the `ovl` function that indicates the manner of overlap under consideration. The function `os` returns the shape of the overlapping part of the tiles, `op` the period and `ot` returns the normalized triple representing the tile-set of overlapping parts.

The generator graph created by the function `orm` heuristic:

$$\begin{aligned} G_g &= \text{orm}(R, (\emptyset, \emptyset), \text{norm}(W)) \\ \text{orm}(R, G_g, T) &\in \{\text{ov}(G_g, T, \mathbf{o}) \mid \mathbf{o} \in \text{ovl}(R, T)\} \\ \text{such that } \mathcal{C}(\text{orm}(R, G_g, T)) &= (\downarrow \mathbf{o} : \mathbf{o} \in \text{ovl}(R, T) : \mathcal{C}(\text{ov}(G_g, T, \mathbf{o}))) \end{aligned}$$

Note that we abuse the notation here slightly; the cost function \mathcal{C} should receive a calculation graph instead of a generator graph as its parameter. We allow this, however, since the cost can be calculated from the generator graph directly (see Section 4.3).

The function `orm` simply selects the best generator graph generated by the `ov` function. This `ov` function generates a generator graph by using the given overlapping pattern \mathbf{o} for the tile T .

$$\text{ov}(G_g, T, \mathbf{o}) = \begin{cases} \text{nov}(G_g, T) & \mathbf{o} = \mathbf{0}_D \\ \text{ovt}(G_g, T, \mathbf{o}) & \text{otherwise} \end{cases}$$

The generator graph in which no overlap for tiles in T is used is generated by `nov` and the generator graph in which overlap is used is generated by `ovt`.

An example: The tiles defined by the triple $T = (\blacksquare\blacksquare\blacksquare\blacksquare, 0, 1)$ overlap each other. For $R = 0$ the algorithm constructs the tile without exploiting any overlap in the tiles. This method is explained later on in this section. For $R = 1$ the algorithm also considers constructing $(\blacksquare\blacksquare, 1, 2)$ as an intermediate result, since these tiles are the overlapping part of two consecutive tiles from T . For $R = 2$ the algorithm considers all the previous possibilities and also the construction of $(\blacksquare, 2, 3)$, since those tiles are the overlapping part of three consecutive tiles from T .

So, that is where the "Overlap" part of the algorithm's name comes from. The "Recursive" part is because this algorithm is applied recursively to the resulting intermediate tiles. And the "Mirroring" part is because the shape of tiles that have to be constructed to form the original tiles in combination with the overlap are mirror symmetric. In the last case from our example we construct $(\blacksquare\blacksquare, 2, 3)$ as the overlap and we would need $(\blacksquare\blacksquare, 0, 3)$, $(\blacksquare\square\blacksquare, 1, 3)$ and $(\blacksquare\blacksquare, 4, 3)$ to be able to construct \mathbb{T} .

This is all done by the `ovt` function:

$$\begin{aligned} \text{ovt}(G_g, \mathbb{T}, \mathbf{o}) &= \text{pq}_1(G_g, \mathbb{T}, \text{ot}(\mathbb{T}, \mathbf{o}), \text{enq}(\mathbb{T}, \mathbf{o}), \{\text{ot}(\mathbb{T}, \mathbf{o})\}) \\ \text{rt}(\mathbb{T}, \mathbf{o}, \mathbf{g}) &= \text{norm}((\mathbf{S}_{\mathbb{T}} + \mathbf{g} \cdot \mathbf{p}_{\mathbb{T}}) \setminus \text{os}(\mathbb{T}, \mathbf{o}), \mathbf{f}_{\mathbb{T}} + \mathbf{g} \cdot \mathbf{p}_{\mathbb{T}}, \text{op}(\mathbb{T}, \mathbf{o})) \\ \text{enq}(\mathbb{T}, \mathbf{o}) &= \{\text{rt}(\mathbb{T}, \mathbf{o}, \mathbf{g}) \mid \mathbf{g} \in \text{hyperbar}(\mathbf{o} + \mathbf{1}_D)\} \end{aligned}$$

The function `rt` calculates the triple representing the tile-sets describing the remaining (non-overlapping) parts at phase \mathbf{g} for an overlapping pattern \mathbf{o} . The function `enq` uses this function in turn to create a set the remaining parts for each phase. This set, or queue, is then processed by the following function to add the construction method (overlapping part and the remaining part) for the tiles of \mathbb{T} to the generator graph for each of the phases:

$$\text{pq}_1(G_g, \mathbb{T}, \mathbf{O}, Q_1, Q_2) = \begin{cases} \text{pq}_2(G_g, Q_2) & Q_1 = \emptyset \\ \text{pq}_1(\text{add}(G_g, (\mathbf{S}_{\mathbb{T}}, \mathbf{p}_{\mathbb{T}}, \mathbf{f}_{\mathbb{T}}), \mathbb{T}, \mathbf{O}, Q), \mathbf{O}, \\ \quad Q_1 \setminus \{Q\}, Q_2 \cup \{Q\}) \text{ with } Q \in Q_1 & \text{otherwise} \end{cases}$$

The `pq2` function is then used to make sure that the tiles used in the construction are actually constructed themselves using the `orm` heuristic:

$$\text{pq}_2(G_g, Q) = \begin{cases} G_g & Q = \emptyset \\ \text{pq}_2(\text{orm}(G_g, \downarrow Q), Q \setminus \{\downarrow Q\}) & \text{otherwise} \end{cases}$$

This is how the `ovt` function is used to construct tiles with overlap. The `nov` is used when the algorithm decides to consider the possibility of constructing a tile without overlap (either as one of the possibilities, or because the tile simply does not overlap). The function searches in the tiles generated so far for any tile that overlaps with the tile that is to be constructed. At the very least it finds $(\blacksquare, 0, 1)$, but perhaps it can find something bigger. If the tile it finds is of "reasonable" size, it is used in the construction. Otherwise the tile to be constructed is simply created by splitting it into two parts. This splitting is done in one of two ways: if the tile is not a hyper-bar (i.e. it contains gaps) then the largest possible hyper-bar is found and split from the tile. If the tile is a hyper-bar we take the dimension in which it is largest (say size x) and split the tile along that dimension in a tile of size $\text{lsb}_1(x)$ and $x - \text{lsb}_1(x)$.

The algorithm decides what a "reasonable" size is by estimating the amount of work that still has to be done after the available tile has been reused. It does this by taking the size of the tile and subtracting the size of the available tile. It then estimates the amount of work that has to be done if the tile is split. It does this by taking the size of the largest of the two components in which the tile should be split. The idea here is that the smaller of the two parts is reused in the larger part, so that no extra costs are incurred for the smaller part. These

two estimates are compared and the one that is estimated to lead to the lowest cost is used.

So, the construction of tiles without reusing the overlap is done as follows:

$$\begin{aligned} \text{nov}(G_g, \mathbb{T}) &= \text{orm}(\text{orm}(\text{add}(G_g, \mathbb{T}, \mathbb{T}_1, \mathbb{T}_2), \mathbb{T}_1), \mathbb{T}_2) \\ &\quad \text{where } (\mathbb{T}_1, \mathbb{T}_2) = \text{ch}(V_g, \mathbb{T}) \end{aligned}$$

The function `nov` adds the construction method specified by the function `ch` to the generator graph and calls `orm` to add the composing tile sets to the generator graph. The function `ch` makes the choice between either by reusing available tiles, or by splitting the tile without reuse:

$$\begin{aligned} \text{ch}(V_g, \mathbb{T}) &= \begin{cases} \text{reu}(V_g, \mathbb{T}) & (|\mathbb{S}_{\mathbb{T}}| - |\mathbb{S}_{\mathbb{T}_2}|) \leq (|\mathbb{S}_{\mathbb{T}_3}| \uparrow |\mathbb{S}_{\mathbb{T}_4}|) \\ \text{spl}(\mathbb{T}) & \text{otherwise} \end{cases} \\ &\quad \text{where } (\mathbb{T}_1, \mathbb{T}_2) = \text{reu}(V_g, \mathbb{T}) \wedge (\mathbb{T}_3, \mathbb{T}_4) = \text{spl}(\mathbb{T}) \end{aligned}$$

Splitting depends on whether the shape is a hyper-bar or not:

$$\text{spl}(\mathbb{T}) = \begin{cases} \text{spl}_1(\mathbb{T}) & \mathbb{S}_{\mathbb{T}} \text{ is a hyperbar} \\ \text{spl}_2(\mathbb{T}) & \text{otherwise} \end{cases}$$

If the shape happens to be a hyper-bar we split it along the largest dimension:

$$\begin{aligned} \text{spl}_1(\mathbb{T}) &= (\text{norm}(\mathbb{S}_{\mathbb{T}_1}, \mathbf{f}_{\mathbb{T}}, \mathbf{p}_{\mathbb{T}}), \text{norm}(\mathbb{S}_{\mathbb{T}_2}, \mathbf{f}_{\mathbb{T}}, \mathbf{p}_{\mathbb{T}})) \text{ such that} \\ \mathbb{S}_{\mathbb{T}} &= \text{hyperbar}(\mathbf{q}) \\ \mathbb{S}_{\mathbb{T}_1} &= \text{hyperbar}(\mathbf{r}) \\ \mathbb{S}_{\mathbb{T}_2} &= \mathbb{S}_{\mathbb{T}} \setminus \mathbb{S}_{\mathbb{T}_1} \\ \mathbf{r}(e) &= \text{lsb}_1(\mathbf{q}(e)) \\ &(\forall d : 0 \leq d < D \wedge d \neq e : \mathbf{r}(d) = \mathbf{q}(d)) \\ &(\forall d : 0 \leq d < D : \mathbf{q}(d) \leq \mathbf{q}(e)) \end{aligned}$$

Otherwise we just extract the largest hyper-bar from the shape:

$$\begin{aligned} \text{spl}_2(\mathbb{T}) &= (\text{norm}(\mathbb{S}_{\mathbb{T}_1}, \mathbf{f}_{\mathbb{T}}, \mathbf{p}_{\mathbb{T}}), \text{norm}(\mathbb{S}_{\mathbb{T}_2}, \mathbf{f}_{\mathbb{T}}, \mathbf{p}_{\mathbb{T}})) \text{ such that} \\ \mathbb{S}_{\mathbb{T}_2} &= \mathbb{S}_{\mathbb{T}} \setminus \mathbb{S}_{\mathbb{T}_1} \\ \mathbb{S}_{\mathbb{T}_1} &= \text{the largest hyper-bar in } \mathbb{S}_{\mathbb{T}} \end{aligned}$$

The `reu` function returns, given an original tile, a triple containing an already existing tile that can be reused and the rest of the original tile.

$$\begin{aligned} \text{reu}(V_g, \mathbb{T}) &= (\mathbb{T}_1, \text{norm}((\mathbb{S}_{\mathbb{T}} + \mathbf{f}_{\mathbb{T}}) \setminus (\mathbb{S}_{\mathbb{T}_1} + \mathbf{g} \cdot \mathbf{p}_{\mathbb{T}_1} + \mathbf{f}_{\mathbb{T}_1}), 0, \mathbf{p}_{\mathbb{T}})) \text{ such that} \\ \mathbf{g} &\in \mathbb{Z}^D \\ \mathbb{T}_1 &= \uparrow \{ \mathbb{T}_2 \mid \mathbb{T}_2 \in V_g \wedge (\mathbf{p}_{\mathbb{T}} \bmod \mathbf{p}_{\mathbb{T}_2} = 0) \wedge \\ &\quad (\mathbb{S}_{\mathbb{T}_2} + \mathbf{g} \cdot \mathbf{p}_{\mathbb{T}_2} + \mathbf{f}_{\mathbb{T}_2}) \subseteq (\mathbb{S}_{\mathbb{T}} + \mathbf{f}_{\mathbb{T}}) \} \end{aligned}$$

Where the set from which \mathbb{T}_1 is selected is the set containing all the triples describing tile set of which all the tiles are a subset of a tile described by \mathbb{T} .

Furthermore, this set is constructed in such a way that the tiles occur at a regular position in \mathbb{T} , which is expressed by the variable \mathbf{g} .

Note that an increase in R generally results in lower cost generator graphs, since the algorithm considers more generator graphs. However, it does not necessarily mean that it considers a superset. Therefore it is not guaranteed that an increase in R leads to a reduction in cost. The two dimensional median filtering is a good example of this (see Appendix A.2.2). Consider the following example; the function `orm` selects the best generator graphs generated by the `ov` function. For $R = 1$ the best graph it encounters is G_1 . For $R = 2$ it also encounters G_1 , but also the slightly better G_2 . So the $R = 1$ algorithm selects G_1 and the $R = 2$ algorithm selects G_2 . However, if this happens to be an intermediate step in the generation of the final graph, then there are still some tiles that have to be added to the selected graph. The cost of adding tiles depends on the chosen graph, since tiles already in the chosen graph might be reused. It is possible that some of the tiles which are to be added can be constructed with more reuse in G_1 than in G_2 . The net effect of this is that the final calculation graph produced by the $R = 1$ algorithm can be less costly than the $R = 2$ algorithm, because the $R = 1$ algorithm based its solution on the (at first sight slightly more costly) G_1 graph.

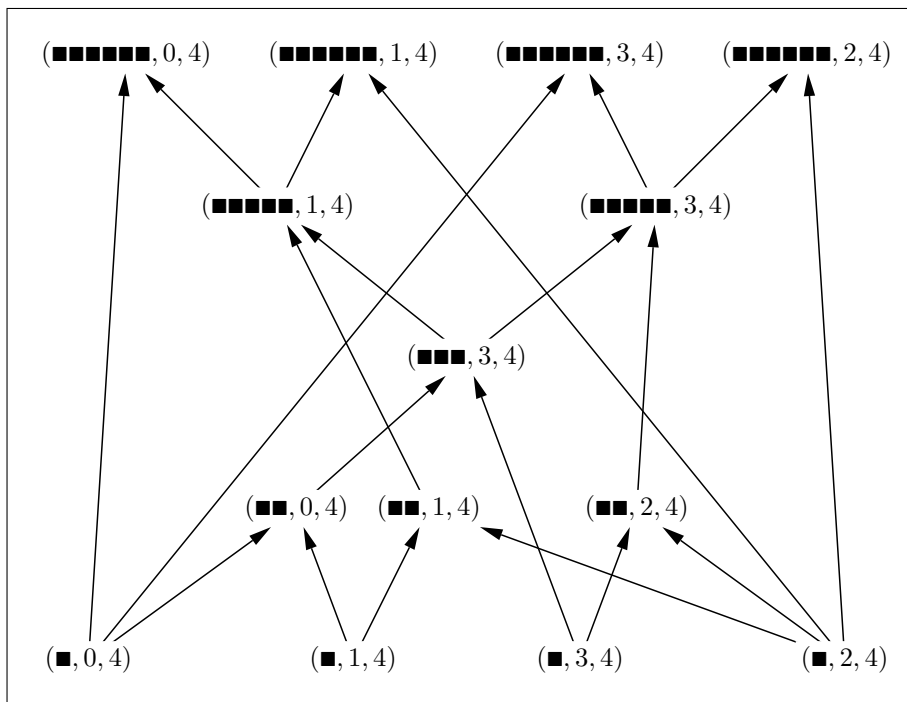


Figure 5.7: G_g for the ORM 1 algorithm for window size 6 (17.5 epw)

The generator graph produced by this algorithm for our running example with window sizes 6 for $R = 1$ is shown in Figure 5.7. This generator graph has a cost of 17.5 epw, which is slightly more expensive than our best solution so far. But when the parameter R is increased to 2 we get the generator graph

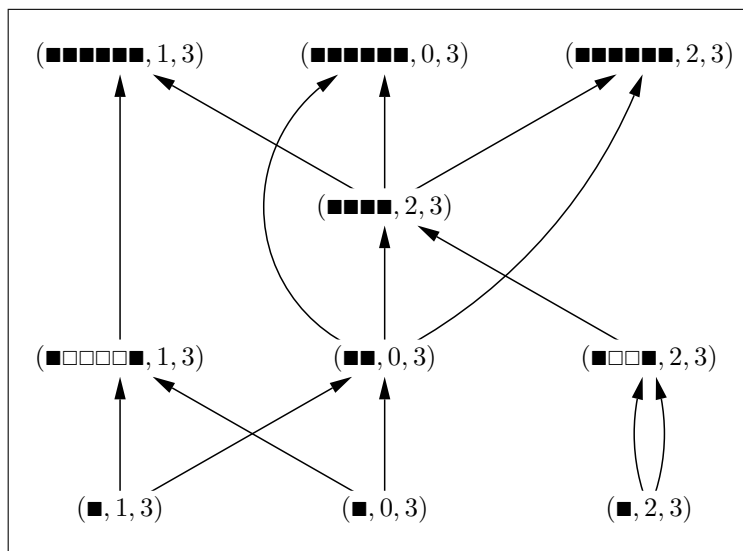


Figure 5.8: A generator graph $G_g(V_g, E_g)$ for $\mathcal{W} = \text{tile}(\text{■■■■■■}, 0, 1)$ (16 epw)

from Figure 5.8. The reuse of the tile of size 4 gives this solution its efficiency and results in a cost of 16 epw, which is the lowest cost found by our algorithms. For the running example with window sizes 7 the ORM algorithm produces the same calculation graphs as the recursive tiling algorithm (depicted in Figure 5.3), with a cost of 20.5 extrema operations, which is also the lowest cost found by our algorithms.

5.7 Multiple Dimensions

Most of our algorithms (except for Overlap Recursive Mirroring) are designed for one dimensional problems. However, they can be generalized to algorithms for multi-dimensional problems using the method described in this section.

This method is only applicable for a certain class of problems, namely the class of problems whose windows have the shape of a hyper-bar.

$$(\exists s : s \in \mathbb{Z}^D : \mathbf{S}_{\mathcal{W}} = \text{hyperbar}(s)) \tag{5.5}$$

The generalized algorithm then functions as follows: We first use the one-dimensional algorithm to produce a solution for windows of size $\mathbf{s}(0)$. This solution shows us how we can construct those windows from tiles of size 1. We then take this solution and extend all the generated tiles into another dimension. All the tiles get a size of $\mathbf{s}(1)$ in this dimension. The construction methods we have for all the tiles are still valid, except for the tiles of size $1 \times \mathbf{s}(1)$. We never generated a construction method for them because they used to be of size 1 in the one-dimensional problem.

So now we execute the algorithm again, but take the tiles with size $1 \times \mathbf{s}(1)$ as the new windows and generate a way to constructing these tiles. Since these tiles only have a size larger than one in the second dimension, they can be considered

one dimensional in practice and the algorithm can be executed normally on them.

We repeat this process as often as necessary to get a solution in all the dimensions.

5.8 Maximum reuse

Upon reviewing this report prior to publication we were able to devise a new heuristic which we call the "Maximum Reuse" heuristic. This heuristic is for one dimensional windows, but can be generalized for multi dimensional windows as described in the previous section.

The algorithm works like the ORM 1 algorithm, but instead of choosing between no overlap and reusing the overlap between two successive windows it always chooses for the reuse. Hence the name of this heuristic, since this method results in that the largest possible tile being reused. But only the overlapping tile is fed back into the maximum reuse heuristic, all other tiles, and tiles with no overlap are created using the `msbt` heuristic.

This new heuristic provides solutions that are close (in cost) to those of the ORM 1 algorithm. Since it is not an improvement upon the ORM algorithm we have not updated the tables and graphs in this report. However, the new heuristic deserves to be mentioned here because its running time is much lower than that of any of the ORM algorithms.

Chapter 6

Results

In the previous chapters we have presented several heuristic algorithms that find generator graphs for specific ROFs. Although we have not yet discussed how an abstract representation in the format of a generator graph can be turned into a concrete implementation, we can already determine the performance of such an abstract filter in terms of extrema operations per window. In this chapter we use this performance measure for two purposes. First we compare the heuristic algorithms from Chapter 5. Second we compare our best abstract designs to designs found in the literature.

6.1 Filter classes

In this section we define the three filter classes that we use to evaluate the heuristic algorithms. These three classes are chosen such that they are representative for filters commonly found in literature and provide sufficient variety to illustrate the differences between the heuristic algorithms.

Although our implementations could easily generate calculation graphs for ROFDPs with an arbitrary number of dimensions, we restrict our attention to one- and two-dimensional cases, where in the latter case we only consider square windows. Although for each window size there are many rank order filters that select a single rank and many more that select an interval of ranks, we restricted our attention to the following three classes:

Window sorters Since in a window sorter all the ranks are obtained, this class gives us an upper-bound for any filter which has to select one or more ranks.

Median filters This rank order filter is interesting because the median is the most costly single rank that we could wish to obtain.

Minimum filters This rank order filter is interesting because the minimum is the least costly single rank that we could wish to obtain. Note that the minimum filter, on grounds of symmetry, cost the same as the maximum filter.

6.2 Heuristic algorithm evaluation

In this section we compare the heuristic algorithms from Chapter 5 by studying their performance on the various filter classes defined in the previous section.

The results of the heuristic algorithms for the classes are plotted in Figures 6.1, 6.2, 6.3, 6.4, 6.5 and 6.6. In all figures the vertical axis shows the cost of the resulting calculation graph in extrema operations per window. The horizontal axis contains the window size. The results of the different heuristic algorithms have been plotted using distinct symbols and colors.

The graphs (sub-figures (a)) show us that the same algorithm can exhibit a quite different trend when it is applied to a different class of filters. All algorithms, for example, display roughly the same, approximately linear, trend when applied to window sorting (except for LSB tiling). However, when those same algorithms are applied to the median filtering algorithm they show different trends; the divide and conquer algorithm shows a linear trend, while the ORM algorithms seem to indicate a logarithmic curve. This difference also holds for the minimum filtering problem, where the divide and conquer algorithm appears to be logarithmic, but the ORM algorithms seem to have an almost constant upper-bound.

Despite the difference in trends the ORM algorithm produces the best filter in each of the three classes. Specifically it is the ORM 4 algorithm (displayed by the small dots) that provides the design with the lowest cost, compared to the other algorithms.

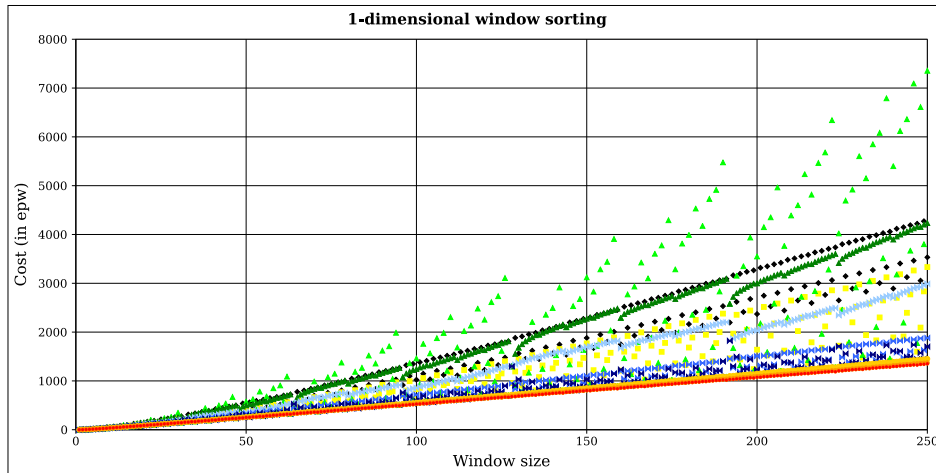
Since we are mainly interested in these best designs, we limit our examination of the trends to the trend of the ORM algorithm. To this end the figures also contain a graph (sub-figures (b)) that contains the results of the ORM algorithms for a specific R , with a curve fitted to the data points. For the 1-dimensional cases these graphs contain the results for $R = 4$. However, for the 2-dimensional cases the original graphs contain too few data points to examine the trend effectively. This was solved by taking larger window sizes and generating additional designs. However, the running time of the ORM algorithm for larger window sizes is impractical for larger R and therefore we opted to use ORM 1 algorithm instead.

The curve fitting for both the window sorting and median filtering for 1-dimensional problems is quite satisfactory. The window sorting seems to follow a linear curve, while the median filtering problem exhibits a logarithmic curve. The 2-dimensional variants are fitted with the same curve as the 1-dimensional variants. Even though the curve fits well the predictive quality of the 2-dimensional curves is questionable.

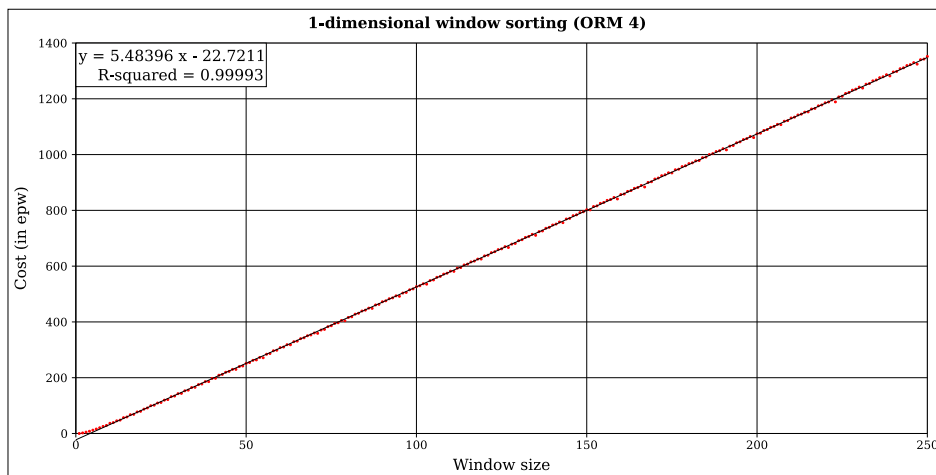
It is possible that the curves would fit better if we had fitted them to the results of the ORM algorithm with a larger parameter R . After all, we see that the ORM algorithms produce approximately the same results for smaller window sizes, but for larger windows the ORM algorithms with a higher R produce better results.

The minimum filtering problem seems to be in another category entirely. Fitting a curve to it proves difficult and we have not been able to find an appropriate one. It seems that the one dimensional variant requires between three and four extrema operations for all the window sizes we examined. The 2-dimensional variant seems to require between four and eight extrema operations.

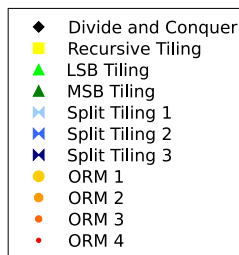
The cost of the minimum filter increases with the size of the window, but this



(a) All algorithms

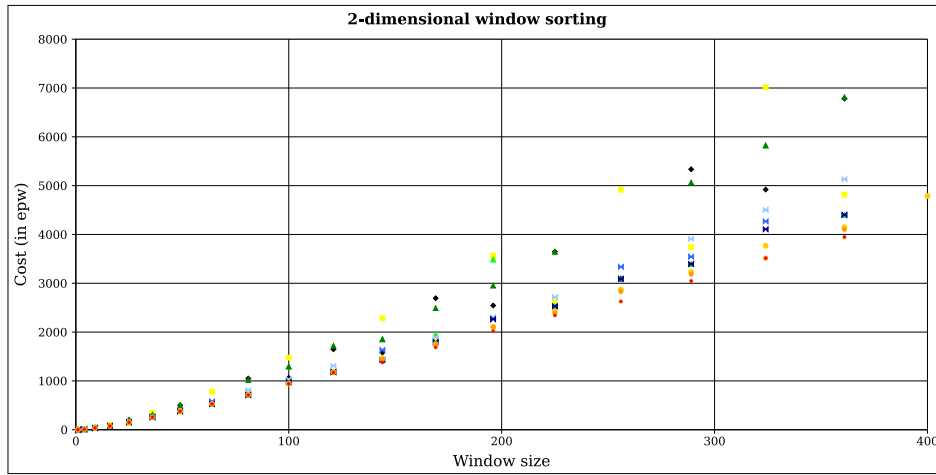


(b) ORM 4

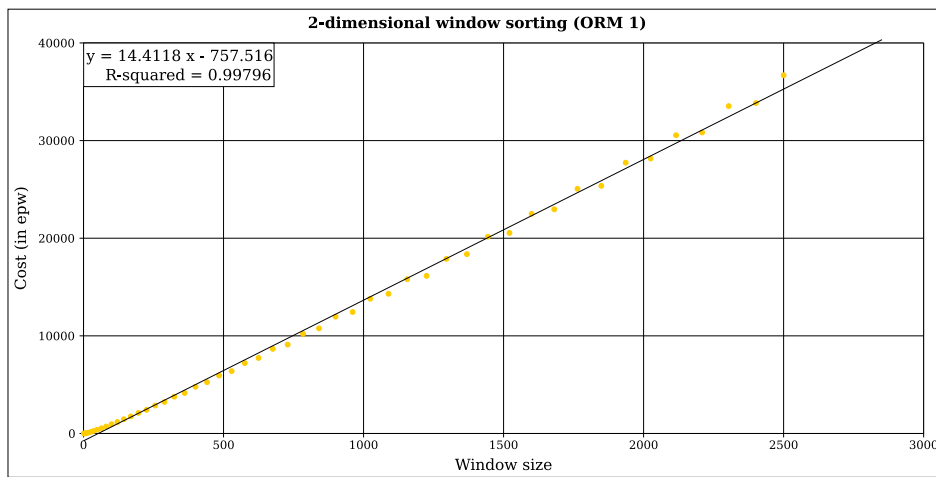


(c) Legend

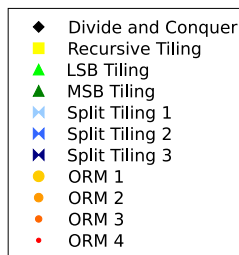
Figure 6.1: 1-Dimensional Window Sorting



(a) All algorithms

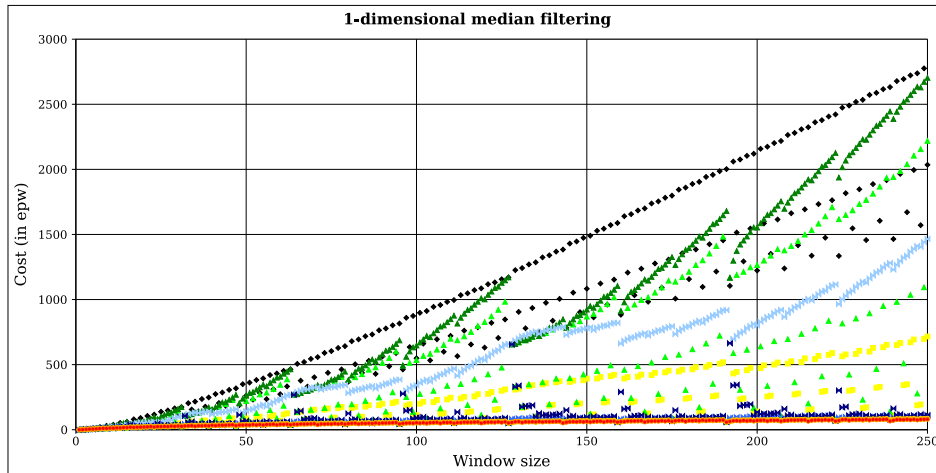


(b) ORM 1

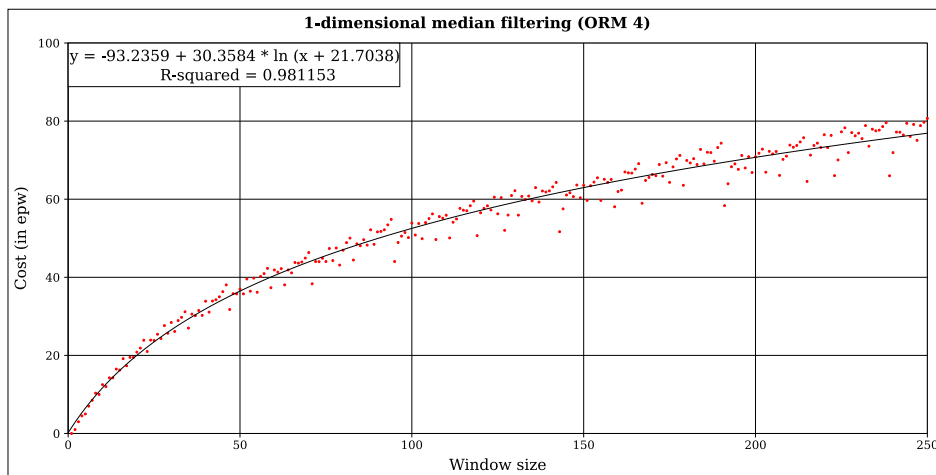


(c) Legend

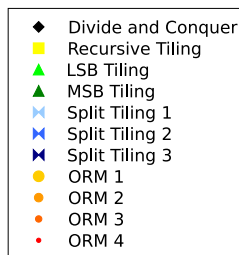
Figure 6.2: 2-Dimensional Window Sorting



(a) All algorithms

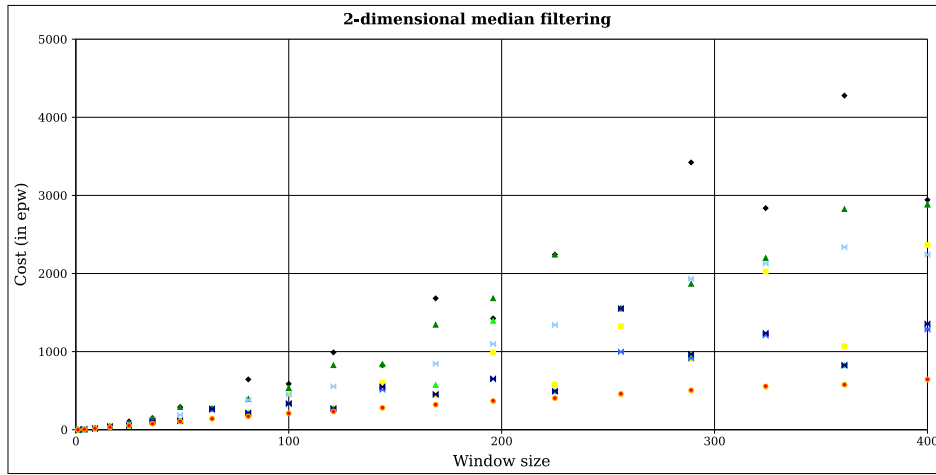


(b) ORM 4

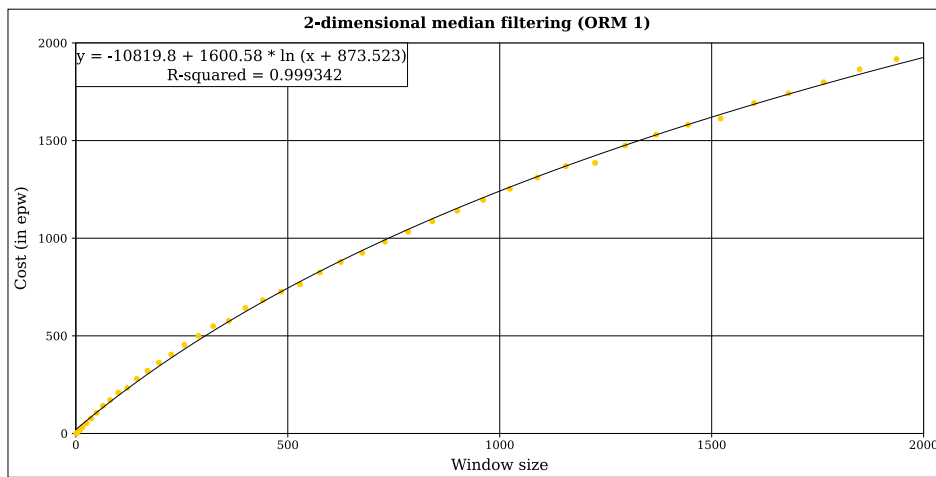


(c) Legend

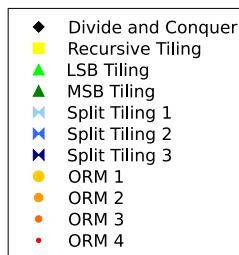
Figure 6.3: 1-Dimensional Median Filter



(a) All algorithms

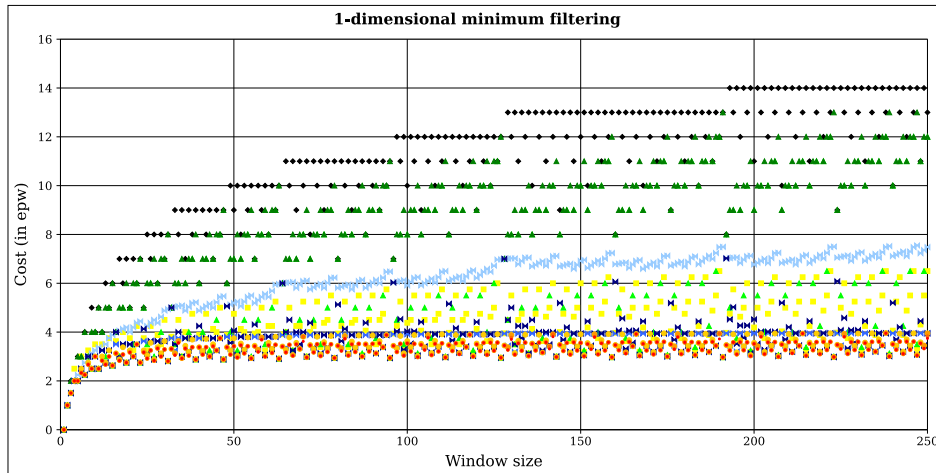


(b) ORM 1

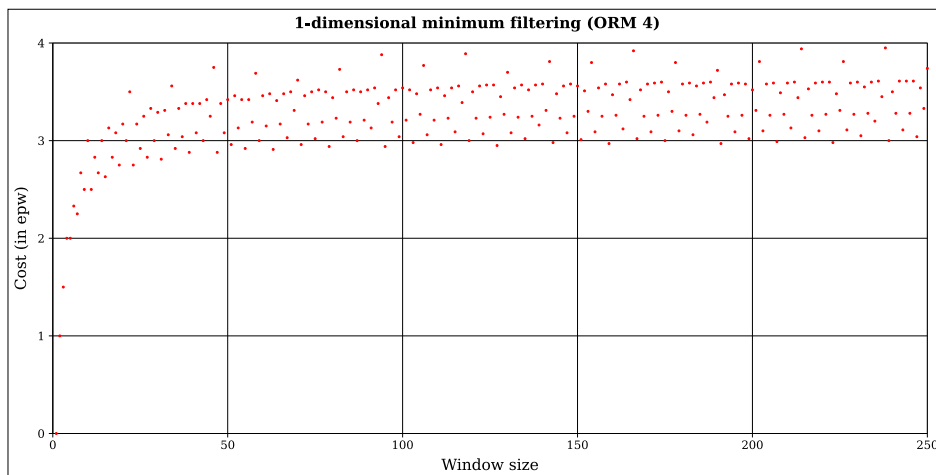


(c) Legend

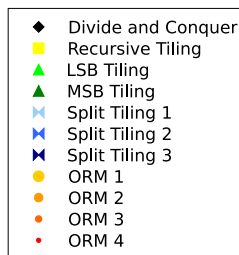
Figure 6.4: 2-Dimensional Median Filter



(a) All algorithms

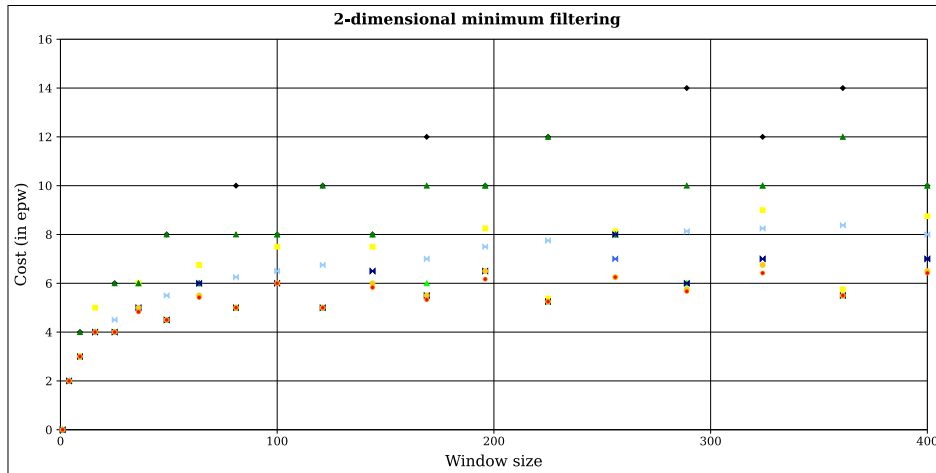


(b) ORM 4

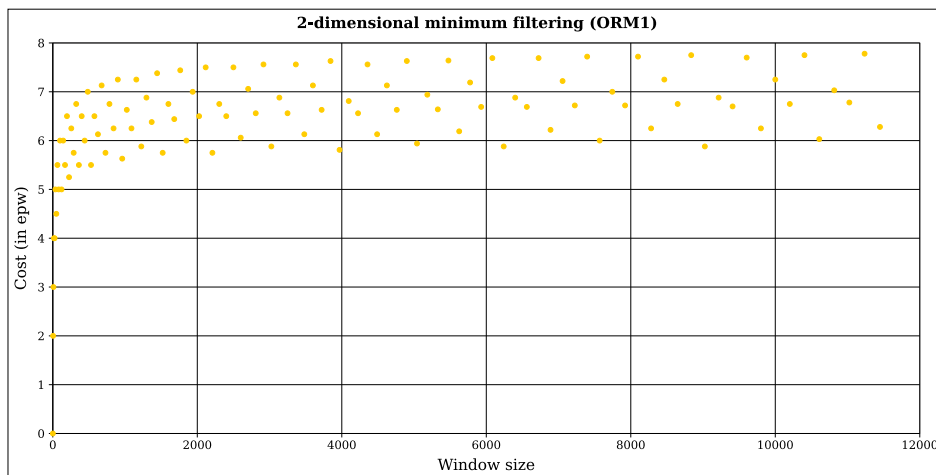


(c) Legend

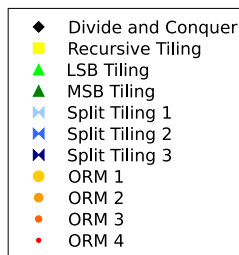
Figure 6.5: 1-Dimensional Minimum Filter



(a) All algorithms



(b) ORM 1



(c) Legend

Figure 6.6: 2-Dimensional Minimum Filter

increase is extremely small. So it is safe to say that implementing a minimum (or maximum) filter requires very few extrema operations, regardless of the number of dimensions and the window size. This is interesting since the best solutions for minimum filter are actually the best solutions for any problem where the cost function always returns a cost of 1 operation. This is discussed in more detail in Chapter 8.

6.3 Comparison with known filters

In the previous section we compared the heuristic algorithms. In this section we compare the results found by our best algorithm with filters taken from the literature. Whenever a publication contains multiple filters for the same ROFDP we have compared our solution with the best one in the publication. These comparisons are based on the cost of the filter in epw. Note that for some publications this measure requires the conversion of the cost in C&S operations (as presented in the publication) to the cost in extrema operations. We do this using the rule that 1 C&S operation costs 2 extrema operations. This conversion is not entirely fair; it might be possible to prune some of those extrema operations, but this can only be done when the design is known. However, most publications only mention their design method and the cost of the resulting filters, but not the filters themselves. Therefore we mark all the costs that have been converted with "*", to indicate that the cost might be lower if extrema operations are pruned.

6.3.1 Window sorters

In this section we survey the first class of problems: the window sorters. We first examine the one-dimensional cases, followed by the two-dimensional cases.

| Window | ORM 4 | Lucke&Parhi[9] |
|--------|-------|----------------|
| 3 | 5.00 | 5.00* |
| 5 | 12.00 | 12.00* |
| 6 | 16.00 | 15.00* |
| 7 | 20.50 | 20.88* |
| 9 | 29.00 | 29.00* |

Table 6.1: Cost of known solutions for 1-dimensional window sorting

Table 6.1 contains the costs of one-dimensional window sorters. Lucke and Parhi examine such window sorters, which they call merge sorters (because of the merging used), in the first part of their paper [9]. Their method is quite similar to our ORM algorithm, in that it exploits overlap between successive windows. However, instead of working with infinite input streams they optimize for a fixed number of overlapping windows, which they call the block size. For all block sizes considered the results shown in Table 6.1 are the best results reported by them.

In one specific instance in Table 6.1, indeed in anything we found in the literature, there is a better solution than the one found by the ORM algorithm. The 1-dimensional window sorter for $|\mathbf{S}_W| = 6$ by Lucke and Parhi [9] requires

only 15 epw, compared to the 16 required by the solution of the ORM algorithm. Unfortunately we cannot explain this anomaly since we have been unable to reconstruct the design of Lucke and Parhi using the algorithm described in their article.

| Window | ORM 4 | Kolte et al.[8] | Lucke&Parhi[9] |
|---------|----------|-----------------|----------------|
| 3 × 3 | 35.00 | 50.00 | 37.50* |
| 4 × 4 | 76.00 | 128.00 | |
| 5 × 5 | 152.00 | 288.00 | 180.44* |
| 6 × 6 | 254.50 | 520.00 | |
| 7 × 7 | 379.00 | 820.00 | |
| 8 × 8 | 526.67 | 1,232.00 | |
| 9 × 9 | 711.00 | 1,804.00 | |
| 10 × 10 | 940.71 | 2,522.00 | |
| 11 × 11 | 1,169.13 | 3,346.00 | |
| 12 × 12 | 1,379.83 | 4,400.00 | |
| 13 × 13 | 1,688.85 | 5,572.00 | |
| 14 × 14 | 2,017.88 | 7,050.00 | |
| 15 × 15 | 2,339.65 | 8,494.00 | |
| 16 × 16 | 2,627.45 | 10,348.00 | |

Table 6.2: Known solutions for 2-dimensional window sorting

Lucke and Parhi also examined 2-dimensional window sorters [9] and so did Kolte, Smith and Su in their paper [8]. The results from these papers are shown in Table 6.2 and compared to our ORM 4 algorithm. Again the ORM 4 algorithm performs best, but the results of Lucke and Parhi come very close. Although the focus of Kolte et al. lies on finding implementations that are easy to implement on a SIMD processor, they nevertheless manage to get good results.

6.3.2 Median filter

In this section we survey the second class of problems: the median filters. We first examine the one-dimensional cases, followed by the two-dimensional cases.

| Window | ORM 4 | Chakrabarti[2] | Lucke&Parhi[9] |
|--------|-------|----------------|----------------|
| 3 | 3.00 | 5.00* | |
| 5 | 5.00 | 11.20* | 31.00* |
| 7 | 8.50 | 19.66* | |
| 9 | 10.00 | 32.00* | |

Table 6.3: Known solutions for 1-dimensional median filtering

In the article [9] Lucke and Parhi also make an excursion to the median filter. Their approach is to specify the median filter as a stack filter in a max-min structure and then applying their method for implementing stack filters. However, as shown in Table 6.3 the resulting solution is not very efficient. In

fact, it is less costly to implement a window sorter according to their own method (see Table 6.1) than it is to implement a median filter (for $|\mathbf{S}_W| = 5$).

Chakrabarti focusses entirely on 1-dimensional median filters in his paper [2] and achieves better results. In a later publication with Wang [3] he also considers 2-dimensional median filtering and again obtains good results (see Table 6.4). Still, the ORM 4 algorithm produces even better results.

| Window | ORM 4 | Chakrabarti & Wang [3] | Kolte et al.[8] |
|----------------|--------|------------------------|-----------------|
| 3×3 | 16.00 | 26.00* | 17.00 |
| 5×5 | 53.00 | 61.00* | 107.00 |
| 7×7 | 105.75 | | 317.00 |
| 9×9 | 171.00 | | 685.00 |
| 11×11 | 232.75 | | 1254.00 |
| 13×13 | 321.25 | | 2036.00 |
| 15×15 | 404.38 | | 3018.00 |

Table 6.4: Known solutions for 2-dimensional median filtering

6.3.3 Minimum filter

In this section we examine the third and final class of filters, namely the minimum filters. There has been little attention for the minimum filter in the literature, but Harrington [5] has examined the problem and has patented a novel method for both 1 and 2-dimensional minimum filters. The 1-dimensional method proposed by Harrington requires just under 3 epw for any window size $|\mathbf{S}_W|$. In most cases this is better than the solutions found by our algorithms. Our algorithms only obtain solutions with such a similar low cost for $|\mathbf{S}_W| = 2^i - 1$ for $i \in \mathbb{Z}$.

The two dimensional approach specified in [5], however, is less efficient. The proposed method for 2-dimensional rectangular windows requires just under 11 epw, which is more expensive than the solutions found by our algorithms. Furthermore, unlike our approach and the 1-dimensional approach from [5], the two dimensional approach in [5] is not suited for every associative operation. The method presented in the patent relies on combining overlapping tiles and thus it is only suited for idempotent operators like minimum and maximum.

6.4 Summary

In the previous chapter we presented several heuristic algorithms that generate a generator graph and thus a design for a ROF. The so-called "Overlap Recursive Mirroring" or ORM algorithm is the best algorithm we designed. The only method we found in the literature that is capable of obtaining better results than our ORM algorithm is by Harrington [5], but only for 1-dimensional minimum filtering. The approach by Lucke and Parhi [9] is capable of obtaining similar results, but only for the window sorting problem. For the median filtering problem our ORM algorithm is the best so far.

The ORM algorithm has a parameter R that gives an indication of the search space covered by the algorithm. As R increases more and more possible designs are considered, and better designs are found. For $R = 4$ the ORM algorithm finds the better solutions than for $R = 3$, however in the majority of cases up to $|\mathbf{S}_W| \leq 250$ the ORM algorithm with $R = 3$ already provides the best solution. The solutions provided by $R = 4$ are only better in a handful of cases.

An overview of the costs of the designs generated by our algorithms can be found in Appendix A. Furthermore the detailed designs of several of the smaller solutions for window sorting and median filtering can be found in Appendix B.

Chapter 7

Parallel ROF Implementations

In this chapter we present two kinds of parallel ROF implementations, viz. implementations in the form of VLSI-circuits and implementations in the form of parallel programs suited for SIMD machines like the EVP_{16} . We discuss the scalability of the resulting implementations and for some specific 2-dimensional filters we provide the number of clock cycles per output required by the EVP_{16} solution. As an intermediate step towards a parallel implementation we first show how to obtain a sequential program and in passing we also show how to handle finite input streams.

7.1 Sequential Implementation

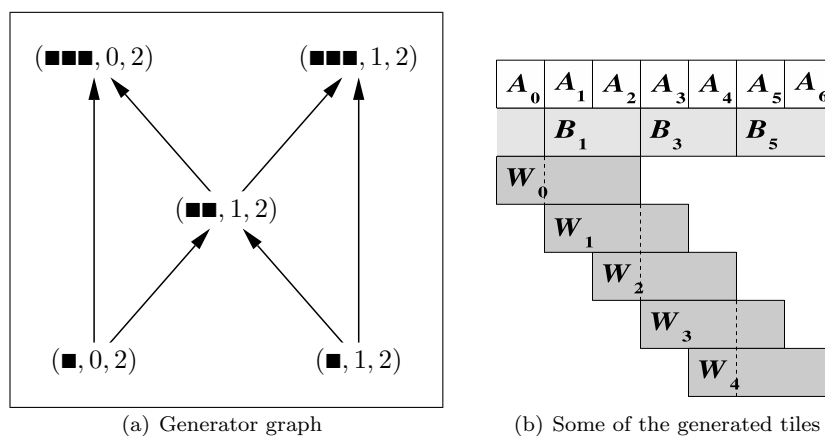
So far we have used calculation graphs as abstract representations of ROFs. Recall that a calculation graph is an acyclic graph whose nodes represent merging operations and whose edges represent the dependencies those operations. By sorting the nodes topologically we obtain an order in which to perform the merge operations and thereby a sequential program.

Unfortunately, because the calculation graph is an infinite graph the program is infinite as well. We can overcome this problem by introducing loops that capture the repetitive nature of the calculation graph.

This repetitive behavior is captured by the generator graph. Note that any two tiles (or windows) from the same tile-set in the generator graph are calculated in the same way and that these tiles are separated by a meta-period. Therefore we can obtain a loop in our program that handles one meta-period per iteration.

The number of windows in a single meta-period is $I = \frac{\text{meta-period}}{Pw}$, and this is therefore the number of windows which are calculated in a single iteration of the sequential program. This differs from common filtering programs which usually calculate only one result per iteration of the loop.

Because a single iteration of our program calculates I new windows it follows that all tiles needed for the calculation of those windows also need to be “obtained”. We use the word *obtain* and not *calculate*, since there are two ways

Figure 7.1: A solution for $\mathcal{W} = \text{tile}(\blacksquare\blacksquare\blacksquare, 0, 1)$

in which the tile can be obtained. On the one hand it can, of course, be calculated by merging the tiles on which it depends. These tiles, in turn, have to be obtained and the recursion ends in the tiles of size one, which form the actual inputs. On the other hand the memoization technique discussed in Section 3.3 can be used. Whenever a tile has been computed that is needed in a future iteration that tile is stored in memory, so that in future iterations the tile only needs to be retrieved, not calculated.

Note that tiles need not be stored indefinitely; by examining the calculation graph we can determine when tiles are no longer needed and remove them from memory. In fact the regularity of the calculation graph allows us to simply introduce a fixed number of buffers which are read and written in a regular pattern.

Whereas the number of windows per iteration is fixed at I , the phasing of the windows within the meta-period can be selected freely. This phase determines which of the actual windows are calculated in a single iteration. Therefore this choice affects the implementation, since it determines which tiles need to be buffered and therefore the size of the buffer.

As an illustration consider the ROFDP with $\mathcal{W} = (\blacksquare\blacksquare\blacksquare, 0, 1)$ for which a solution is given in Figure 7.1. This figure shows the generator graph with meta-period 2 and some tiles of the three tile sets $\text{tile}(\blacksquare\blacksquare\blacksquare, 0, 1)$, $\text{tile}(\blacksquare\blacksquare, 1, 2)$ and $\text{tile}(\blacksquare, 0, 1)$. The construction of these individual tiles can be derived from the generator graph:

$$\begin{aligned} \mathbf{B}_{2i+1} &= \mathbf{A}_{2i+1} \bowtie \mathbf{A}_{2i+2} \\ \mathbf{W}_{2i} &= \mathbf{A}_{2i} \bowtie \mathbf{B}_{2i+1} \\ \mathbf{W}_{2i+1} &= \mathbf{B}_{2i+1} \bowtie \mathbf{A}_{2i+3} \end{aligned}$$

If we choose phase 0, then \mathbf{W}_{2i} and \mathbf{W}_{2i+1} (for $i \in \mathbb{Z}$) would fall into the same meta-period. So the iteration that calculates \mathbf{W}_0 and \mathbf{W}_1 needs to obtain tiles \mathbf{A}_0 , \mathbf{B}_1 , \mathbf{A}_3 and, recursively, \mathbf{A}_1 and \mathbf{A}_2 . Of these tiles \mathbf{A}_0 and \mathbf{A}_1 can be read from memory, because they were also needed in the previous iteration, and

tiles \mathbf{A}_2 and \mathbf{A}_3 need to be stored in memory because they will be needed in the next iteration. Choosing phase of 0 therefore requires the buffering of two tiles of size 1 between each iteration.

Next, choose the phase to be 1, then \mathbf{W}_{2i+1} and \mathbf{W}_{2i+2} would fall into the same meta-period. So the iteration that calculates \mathbf{W}_1 and \mathbf{W}_2 needs to obtain tiles \mathbf{B}_1 , \mathbf{A}_3 , \mathbf{A}_2 , \mathbf{B}_3 and, recursively, \mathbf{A}_1 , \mathbf{A}_2 , \mathbf{A}_3 and \mathbf{A}_4 . Note that the tiles \mathbf{A}_2 and \mathbf{A}_3 have to be obtained twice in this iteration. Of the tiles to be obtained the tiles \mathbf{B}_1 and \mathbf{A}_2 can be read from memory, because they were needed in the previous iteration, and tiles \mathbf{B}_3 and \mathbf{A}_4 need to be stored in memory because they will be needed in the next iteration (the one that calculates \mathbf{W}_3 and \mathbf{W}_4). Choosing a phasing of 1 therefore requires the buffering of one tile of size 1 and one tile of size 2 between each iteration.

It seems that a phase of 1, for our example, requires the largest amount of buffering. However, keep in mind that the size of a tile and the size of the interval of ranks required from a tile are not necessarily the same (see Section 3.1). If Figure 7.1 is a solution for a minimum or maximum filter, for example, then we only need one item from the tile of size 2. So the buffer cost for both phases would be 2 data items, in that case, therefore, the phase would not matter.

Finding the best phase can be done with a simple exhaustive search, since the meta-period for a dimension is an upper bound for the number of phases in that dimension. Note that choosing the correct phase can diminish the amount of buffering, but that it does not adversely affect the number of extrema operations that have to be performed.

```

do true →
  {buf0 =  $\mathbf{A}_{2i}$ } {buf1 =  $\mathbf{A}_{2i+1}$ }
  a2 := read( $\mathbf{A}_{2i+2}$ );
  {a2 =  $\mathbf{A}_{2i+2}$ }
  b := buf1  $\bowtie$  a2;
  {b =  $\mathbf{B}_{2i+1} = \mathbf{A}_{2i+1} \bowtie \mathbf{A}_{2i+2}$ }
   $\mathbf{W}_{2i} :=$  buf0  $\bowtie$  b;
  { $\mathbf{W}_{2i} = \mathbf{A}_{2i} \bowtie \mathbf{B}_{2i+1}$ }
  buf0 := a2;
  {buf0 =  $\mathbf{A}_{2i+2}$ }
  a3 := read( $\mathbf{A}_{2i+3}$ );
  {a3 =  $\mathbf{A}_{2i+3}$ }
   $\mathbf{W}_{2i+1} :=$  b  $\bowtie$  a3;
  { $\mathbf{W}_{2i+1} = \mathbf{B}_{2i+1} \bowtie \mathbf{A}_{2i+3}$ }
  buf1 := a3;
  {buf1 =  $\mathbf{A}_{2i+3}$ }
  i := i + 1;
od

```

Figure 7.2: Pseudo-code for the example in Figure 7.1

Once the phase has been chosen the sequential program can be written. The program executes an infinite loop that calculates I windows per iteration and

between each iteration the program buffers several data items, in accordance with the explanation above. Effectively, we have chosen a part of the calculation graph and implemented it as a number of statements. Because the calculation graph can be formed by replication of the chosen part we can implement the filter by repeating those statements in a loop.

A phase of zero for our example leads to the pseudo-code in Figure 7.2. Note that the pseudo-code resembles the generator graph. The number of merge operations equals the number of tile-sets with tiles larger than one and the number of read operations equals the number of tile-sets with tiles of size one in the generator graph. It is, in fact, possible to generate the program code for a rank order filter automatically from the generator graph.

In Section 7.3 we will show how to deal with an input stream that is finite in one or more dimensions. This will effectively limit the number of iterations for the loops that are related to those finite dimensions. Using this we can generalize our approach to multi-dimensional problems where the program will consist of several nested loops, one loop for each dimension. The approach will work as long as there is at most one dimension in which the inputs are infinite; the loop related to that dimension will become the outermost loop.

7.2 Parallel Implementation

The parallel implementations in this section are based on one iteration of the sequential program. The VLSI implementation, for example, will be based on components that implement one such iteration. Furthermore, although there are data dependencies between consecutive iterations, it is also possible to implement each iteration on a processing element of a SIMD machine since these dependencies can be resolved.

7.2.1 VLSI Circuit

Implementing the rank order filter in parallel in a VLSI circuit is quite straightforward.

First we create a component that performs the same operation as a single iteration of the sequential program. This component has two sets of inputs and two sets of outputs. One set of inputs is for the tiles of size one, the actual inputs of the filter. The other set of inputs is for the tiles the component needs to read from a buffer from a previous iteration. Similarly the outputs are divided into two sets; those for the complete windows and those for the tiles that need to be buffered for a next iteration.

This component is depicted in Figure 7.3. Note that we used boxes with the symbol "⊗" to denote the merging networks. One such component can be used to get an implementation that calculates I outputs in parallel. To do this we simply connect the components buffer outputs to a buffer and connect the buffer to the components buffer inputs, as is depicted in Figure 7.4(a).

To obtain a circuit that calculates $L = IP$ windows in parallel we simply use P components and connect them in a similar manner. For $P = 4$ this has been done in Figure 7.4(b). In this way any number of components may be connected to increase the number of results calculated by the circuit.

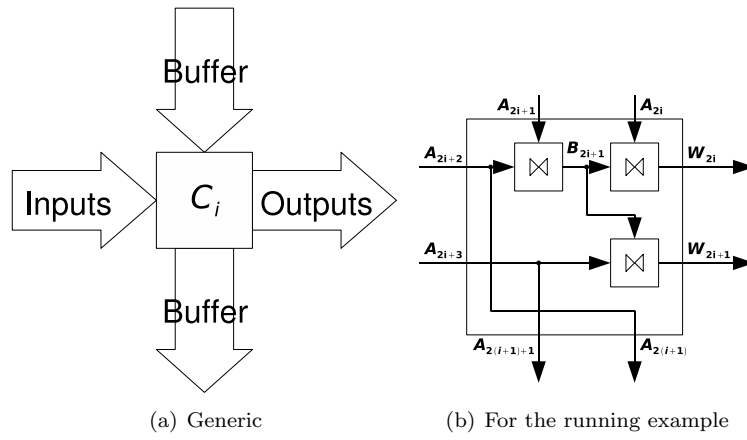


Figure 7.3: Basic component for VLSI implementation

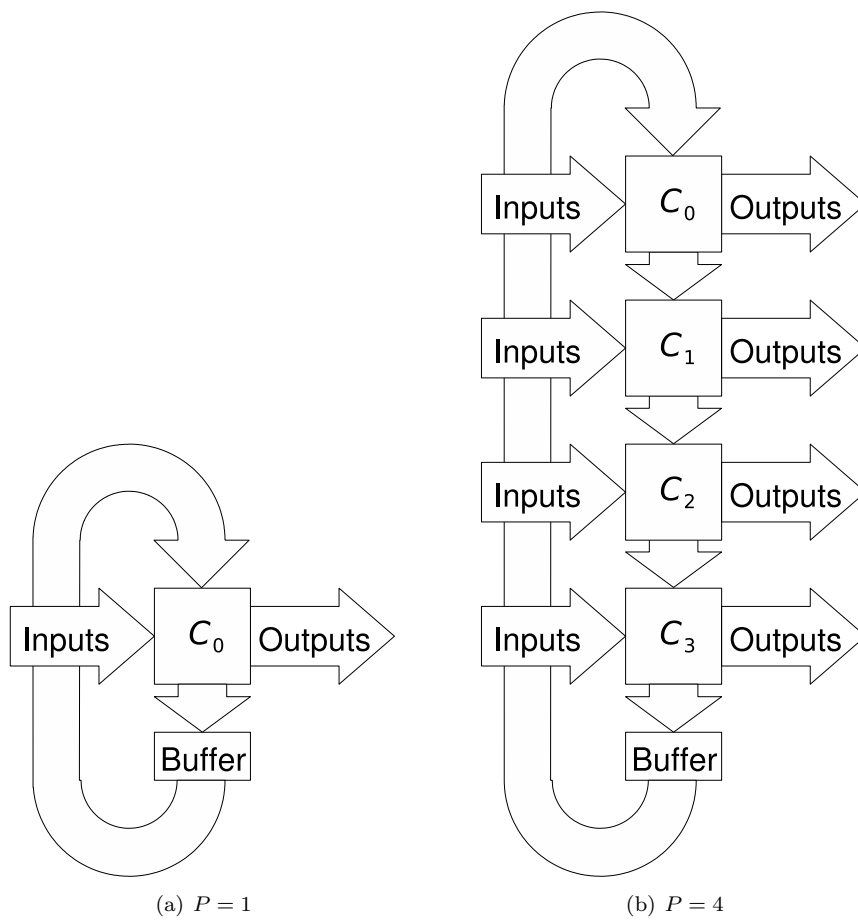


Figure 7.4: Parallel rank order filtering circuit

The actual cycle time of the circuit is independent of P . There is a dependency between components since tiles that are calculated in one component may be needed (also) in the others, but it is the depth of the calculation graph (from tiles of size 1 to the windows) and the calculation time of the merging networks that determine the cycle time. Neither of these depend on P , therefore the cycle time is independent of the amount of parallelism, and an arbitrary throughput can be obtained. This throughput is $\mathcal{O}(L)$ and therefore it scales linearly with the size of the circuit.

The circuit may even be pipelined to obtain a smaller cycle time, since it is essentially a feed forward network. The calculation graph is an directed acyclic graph and this is reflected in the circuit; smaller tiles are fed into the rest of the circuit to calculate larger tiles. Although there appears to be a cycle in the circuit (see Figure 7.4) a detailed examination of the circuit will show that there are no cycles, only some feed forward paths that contain one or (possibly) more buffers.

A note on optimization: a component that performs a compare and swap operation is cheaper (in terms of the number of gates) than two components that perform an extrema operation. Therefore a good optimization step when implementing the merging networks is to replace all the pairs of extrema components that actually perform a C&S operation by a C&S component.

7.2.2 SIMD Processor

In an SIMD processor with P processing elements (PEs) there are more restrictions to implementing an algorithm than in hardware. However we can calculate P iterations of the sequential program in one iteration of the SIMD program. We can also look at it from a different angle and consider one iteration of the SIMD program as the simulation of one cycle of the VLSI implementation (with P components).

The problem in vectorizing the sequential program is that there is a data dependency between the iterations; the tiles depend on the tiles in the buffers. The reading from and writing to these buffers needs to be synchronized such that a read from a buffer is immediately followed by a write to that buffer. In this way the communication of the tile stored in the buffer can be implemented by a shift operation. During the shift operation each of the PEs receives a tile from its neighbor that is working on the preceding iteration (the read from the buffer) and sends a tile to its neighbor that is working on the next iteration (the write to the buffer).

The calculation of a tile that has to be written to a buffer will never depend on the contents of that same buffer (tiles can only depend on smaller tiles), it is therefore always possible to rearrange the statements in such a way that the read from and write to a buffer follow each other immediately.

Note that during a shift operation the tile received by PE 0 has to come from a buffer and the element send by PE $P - 1$ has to be placed in that buffer, and that the contents of this buffer need to be kept for the next iteration of the SIMD program. Most SIMD processor have some sort of functionality to handle the data items at the edge of the PE array during a shift operation, so this is not a problem.

There is one last problem that we have to tackle before we have our SIMD implementation. This concerns the reading of the inputs and the writing of the

outputs. Since all the tile-set in the generator graph have a period equal to the meta-period the implementation requires the inputs and produces the outputs with that same period. Consider our running example; the sequential program calculates first the next even and then the next odd window in an iteration. The SIMD program, however, first calculates the next P even windows and then the next P odd windows. The same reasoning holds for the reading of the inputs.

This is no problem if the inputs and outputs happen to be in memory in that particular order. However in most cases the inputs are in memory in consecutive order and the outputs are also desired in a consecutive order. Therefore we can use so-called strided memory access to implement our algorithm, as shown in Figure 7.5.

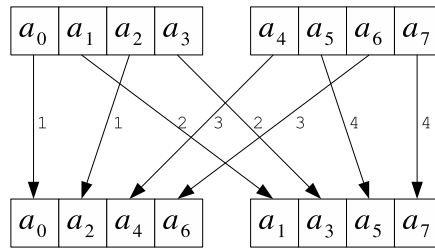
```

do true →
  {buf0 =  $\mathbf{A}_{2i}$ }
  {buf1 =  $\mathbf{A}_{2i+1}$ }
  a2_vec := strided_read( $\mathbf{A}_{2i+2}$ , 2);
  {a2_vec[j] =  $\mathbf{A}_{2(i+j)+2}$  for  $0 \leq j < P$ }
  a3_vec := strided_read( $\mathbf{A}_{2i+3}$ , 2);
  {a3_vec[j] =  $\mathbf{A}_{2(i+j)+3}$  for  $0 \leq j < P$ }
  t1_vec, buf1 := shift(buf1, a3_vec);
  {t1_vec[j] =  $\mathbf{A}_{2(i+j)+1}$  for  $0 \leq j < P \wedge \text{buf1} = \mathbf{A}_{2(i+P)+1}$ }
  b_vec := t1_vec  $\otimes$  a2_vec;
  {b_vec[j] =  $\mathbf{B}_{2(i+j)+1} = \mathbf{A}_{2(i+j)+1} \otimes \mathbf{A}_{2(i+j)+2}$  for  $0 \leq j < P$ }
  t0_vec, buf0 := shift(buf0, a2_vec);
  {t0_vec[j] =  $\mathbf{A}_{2(i+j)}$  for  $0 \leq j < P \wedge \text{buf0} = \mathbf{A}_{2(i+P)}$ }
   $\mathbf{W}_{2i}$  := t0_vec  $\otimes$  b_vec;
  { $\mathbf{W}_{2(i+j)} = \mathbf{A}_{2(i+j)} \otimes \mathbf{B}_{2(i+j)+1}$  for  $0 \leq j < P$ }
   $\mathbf{W}_{2i+1}$  := b_vec  $\otimes$  a3_vec;
  { $\mathbf{W}_{2(i+j)+1} = \mathbf{B}_{2(i+j)+1} \otimes \mathbf{A}_{2(i+j)+3}$  for  $0 \leq j < P$ }
  i := i + P;
od

```

Figure 7.5: SIMD pseudo-code for the example in Figure 7.1

Not many SIMD processors support strided memory access, but another solution to this problem is possible if the processor is capable of shuffling the vectors. A shuffle operation is used to reorder the elements of a vector. So, after the inputs have been read in consecutive order we use several (masked) shuffling operations to distribute them to the correct PEs. This process is depicted in Figure 7.6 for an SIMD processor with 4 PEs. On the top the figure shows the inputs as they read from memory, at the bottom it shows how two vectors have been created, both containing the inputs with a period of two, but the left-most vector with a phase of 0, and the right-most vector with a phase of 1. The numbers at the arrows indicate the number of the shuffle operation in which the movement of the input to the destination vector is accomplished. The figure shows that 4 shuffle operations are needed, assuming that each shuffle operation can have only one source and one destination vector. A similar shuffling pattern can be used to get the outputs back into a consecutive order.

Figure 7.6: Shuffling consecutive inputs for $P = 4$ and a meta-period of 2

Note that the number of shuffling operations depends only on the meta-period, not on the number of processing elements (P). Therefore the running time of an iteration of the SIMD program is still independent of P and we end up with an SIMD implementation of which the throughput ($\mathcal{O}(L) = \mathcal{O}(IP)$) scales linearly with the number of processing elements.

A note on optimization: a SIMD processor is usually capable of the required extrema operations. However should it be capable of the C&S operation then using that instruction wherever possible in the implementation would be a good idea. A C&S (or extrema) operation can even be simulated by comparing two values and using the result as a mask to some assignment operations, as shown by the pseudo-code in Table 7.1.

| Option 1: | Option 2: |
|------------------------------|---|
| <code>t := a;</code> | <code>mask := compare_gte(a, b);</code> |
| <code>a := min(a, b);</code> | <code>t := a;</code> |
| <code>b := max(t, b);</code> | <code>a := masked_move(b, mask);</code> |
| | <code>b := masked_move(t, mask);</code> |

Table 7.1: Two options for implementing a C&S operation

Clearly option 1 requires less instructions, and by using dead code elimination it can be automatically removed from the program if necessary. However, option 2 might be faster than simple extrema operations if the processor also provides VLIW parallelism. Although option 2 clearly requires more operations, in a VLIW processor there might be multiple functional units capable of a move operation (addition of zero, shifting over a distance of zero, etc.), while there is usually only one functional unit capable of extrema operations. So option 2 implementation might require less clock cycles for some processors. For an example of such a processor, see Section 7.4.

7.3 Finite input streams

So far we have considered input streams that are infinite. In practice, however, this is not the case. How do our implementations cope with this?

The answer is relatively simple: the finite input streams are extended to contain a number of elements that is a multiple of L . In that way all of our

implementations can simply make an integer number of iterations. Extending the input stream with dummy inputs is something that happens routinely in image processing to prevent artifacts at the edge of a picture.

The only problem with this solution is that the implementation now has some overhead compared to the theoretical solution found by the calculation graph. However, this overhead is $\mathcal{O}(L)$ for the entire input stream, so it becomes less and less important for the larger input streams.

This solution can be extended to multiple-dimensions. If there are $d \leq D$ dimensions in which the input stream is finite then this introduces an overhead of $\mathcal{O}(L^d)$. Again this overhead becomes less and less important as the size of the input stream increases.

7.4 Philips EVP₁₆

The Philips EVP₁₆ is an SIMD processor with $P = 16$ processing elements [1]. The EVP₁₆ is a processor that supports extrema operations and shuffle operations. So, although it does not support C&S operations or strided memory access, it is possible to implement our solution. We wrote a program that, given a generator graph, automatically produces the program code for the EVP₁₆.

The EVP₁₆ also supports VLIW parallelism and masked operations. The extrema operations and compare operations are only supported on one unit¹, while assignment or move operations are supported on 4 units². This means that the alternative implementation of the C&S operation, i.e. option 2 from Table 7.1, is possible and might even be an improvement.

Option 2 would also take two clock cycles (one for the compare and move operation and the other for the two remaining masked move operations). So there seems to be no gain. However, with this implementation we can start a new C&S operation in each clock cycle. Assuming, of course, that the second of the subsequent C&S operations does not require the outputs of the first as input. This is something to take into account.

Another thing we must take into account is that the compiler also needs to schedule some other operations, like the memory accesses and the shuffling of vectors. These operations can only be performed on the same units that are used by option 2. So producing the most efficient program code is an entirely new optimization problem.

We have not examined this problem extensively, because with (for example) N C&S operations in a program, and two options for implementing each C&S operation, there would be 2^N configurations of the program in total. Needless to say, it is impractical to examine all these possibilities and select the best one.

To find a good program we use two steps. The first step is to find all the C&S operations of which one of the extrema operations can be pruned. For these C&S operations the best implementation is option 1, since that implementation reduces to a single extrema operation and thus requires only one clock cycle on the ALU unit.

After this first step there may still be C&S operations left for which we can choose between the two alternative implementations. So in the second step we use a simple heuristic algorithm to find good choices for these remaining C&S

¹VALU

²VLSU, VMAC, VALU and VSHU

operations. The choices for these operations can be represented as a bit vector. We start by using the bit vector representing that all C&S operations are implemented as option 2. We then examine the neighborhood of the current configuration for a new configuration that is better. The neighborhood of a configuration consists of all configurations with a Hamming distance of one to the current configuration, i.e. all the configurations that have a different option for a single one of the C&S operations. The best configuration in this neighborhood is taken and we apply the same algorithm again and again, until there is no more improvement. The performance of the resulting implementations after such an optimization can be found in Table 7.2. However, solving this optimization problem with a more sophisticated algorithm may improve these results.

| Window: | 3×3 | | 5×5 | |
|---------------------|--------------|--------|--------------|--------|
| Clock cycles per: | iteration | output | iteration | output |
| Using only option 1 | 84 | 1.31 | 295 | 4.61 |
| Using only option 2 | 77 | 1.20 | 345 | 5.39 |
| After optimization | 71 | 1.11 | 241 | 3.77 |

Table 7.2: Clock cycles for median filtering on the EVP_{16}

Table 7.2 presents the information on two 2-dimensional median filters implemented on the EVP_{16} . The implementations produce $L = IP$ outputs per iteration. The number of windows per iteration happens to be $I = 4$ for both problems and the processor has $P = 16$ PEs, therefore the number of clock cycles per output is simply the number of clock cycles per iteration divided by $L = 64$. The rows of the table indicate the number of clock cycles for different implementation strategies. The row marked "Using only option 1" and "Using only option 2" shows the results when step 2 of our optimization is skipped and all remaining C&S operations are implemented using option 1 or option 2 respectively. The row marked "After optimization" shows the results when the neighborhood search algorithm is used to optimize the implementation.

Note that the 3×3 and 5×5 median filters require 1.15 and 6.6 clock cycles per output respectively on the Altivec, according to [8]. Our implementation on the EVP_{16} with 1.11 and 3.77 clock cycles respectively is an improvement on those numbers, especially for the latter filter.

Table 7.2 also shows that median filters can be implemented very efficiently. No sequential algorithm is capable of producing outputs at the rate shown in the table. There are sequential algorithms that have a better complexity than our solutions, but the oblivious approach allows for easy parallelism. For practical window sizes this parallelism compensates small differences in complexity and provides better overall performance.

Chapter 8

Conclusion

In this report we have presented a mathematical framework for representing both the rank-order filter design problem (ROFDP) and the solutions to that problem, i.e. the rank-order filters (ROF) themselves. We have also introduced several heuristic algorithms for solving the ROFDP and have compared the filters found by those algorithms to filters found in the literature. Furthermore, we have shown how our designs can be used to obtain scalable, parallel implementations of ROFs as VLSI circuits or as programs on a SIMD machine like the EVP_{16} .

A novelty of our designs is the mirroring of merging and/or sorting networks prior to the pruning (or "dead-code elimination"). We have not encountered this optimization in the literature so far, but it improves results significantly. Furthermore we present a cost function for pruned merging networks, both for the standard versions and for the versions with the mirroring optimization applied.

The major result of our research, however, we consider the mathematical framework that uses concepts like tiling, the calculation graph and the generator graph. This framework enables us to formalize the reuse that takes place in a ROF. In addition it allows us to make a clear distinction between the calculation to be performed by the ROF and the (parallel) implementation itself. This separation of concerns allows us to design the solution first and choose the implementation method and the amount of parallelism at a later stage. In all cases considered this approach has resulted in designs that require less extrema operations per window than the ones reported in the literature so far, except for 1-dimensional minimum filtering [5] and a 1-dimensional window sorter for windows of size 6[9].

Since we have not been unable to provide an upper bound for the cost for arbitrary large window sizes. The asymptotic complexity of our designs is unknown, just like the designs presented in [2, 3, 8, 9]. Although the cost of our ROF designs is very low, we know that the complexity is not as low some of the best algorithms found in the literature [6, 10], due to the fact that we resort to oblivious sorting methods. Because our designs rely on oblivious sorting methods they have two major advantages over these algorithms. First of all they allow for scalable parallelism; the throughput of the filter scales linearly with the amount of parallelism. Secondly, the constant factor for our designs is very low, which means that our designs perform better for practical

(i.e. relatively small) window sizes.

The methods presented in this report are widely applicable. Any problem consisting of a sliding window and an aggregation operator that is both associative and commutative can benefit from our approach. The cost of an implementation using an aggregation function with a cost of one (like the addition, multiplication, minimum and maximum operators) is equal to the cost of the minimum filter. The only difference is that in the first two cases the cost is measured in addition operations per window and multiplications operations per window respectively, instead of extrema operations per window.

Other problems for which our methods can be adapted are weighted rank order filters. In these filters the set of coordinates forming the windows become multi-sets, and the weight of a coordinate is represented by the number of times it is in the multi-set.

Although the generator graphs found by our algorithms are already quite good there is still room for improvement. The largest improvement, however, can be achieved by improving on the implementation of our designs on SIMD processors. Generating efficient code that can be readily optimized by the compiler for such machines is a hard problem that we have not examined fully yet. All these improvements, however, are possible topics for future research.

In summary, our research has shown that the computational gap between linear filters and rank order filters is not as large as was previously believed. The improvements made in this report are sufficient to make the use of rank order filters in real-time systems attractive.

Bibliography

- [1] C.H. van Berkel, Frank Heinle, Patrick P.E. Meuwissen, Kees Moerman, and Matthias Weiss. Vector processing as an enabler for software-defined radio in handsets from 3G+WLAN onwards. In *Proceedings of the 2004 Software Defined Radio Technical Conference*, volume B, pages 125–130, November 2004.
- [2] Chaitali Chakrabarti. Sorting network based architectures for median filters. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 40(11):723–727, November 1993.
- [3] Chaitali Chakrabarti and Li-Yu Wang. Novel sorting network-based architecture for rank order filters. *IEEE Transactions on VLSI Systems*, 2(4):502–507, December 1994.
- [4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.
- [5] Steven J. Harrington. Us patent #7019760: Method and apparatus for fast computation of associative operations over fixed size regions of a digital image, March 2006. Xerox Corporation.
- [6] Martti Juhola, Jyrki Katajainen, and Timo Raita. Comparison of algorithms for standard median filtering. *IEEE Transactions on Signal Processing*, 39(1):204–208, January 1991.
- [7] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1968.
- [8] Priyadarshan Kolte, Roger Smith, and Wen Su. A fast median filter using altivec. In *International Conference on Computer Design (ICCD)*, pages 384–391, 1999.
- [9] Lori E. Lucke and Keshab K. Parhi. Parallel processing architectures for rank order and stack filters. *IEEE Transactions on Signal Processing*, 42(5):1178–1189, May 1994.
- [10] Ben Weiss. Fast median and bilateral filtering. *ACM Transactions on Graphics (TOG)*, 25(3):519–526, July 2006.

Appendix A

Algorithm Results

The next sections of this appendix present tables containing the results of the different algorithms described in Chapter 5. Each row of such a table corresponds to a specific window size W , and each column to a specific algorithm. For the sake of brevity, we have adopted the following numbering scheme to identify the algorithms:

0. Divide and Conquer (Section 5.2)
1. Recursive Tiling (Section 5.3)
2. LSB Tiling (Section 5.4)
3. MSB Tiling (Section 5.4)
4. Split Tiling 1 (Section 5.5)
5. Split Tiling 2 (Section 5.5)
6. Split Tiling 3 (Section 5.5)
7. Overlap Recursive Mirroring 1 (Section 5.6 with $R = 1$)
8. Overlap Recursive Mirroring 2 (Section 5.6 with $R = 2$)
9. Overlap Recursive Mirroring 3 (Section 5.6 with $R = 3$)
10. Overlap Recursive Mirroring 4 (Section 5.6 with $R = 4$)

The last column of these tables indicates which of the algorithms provided a solution with the minimum cost.

A.1 One Dimensional Problems

In this section we present the results of our algorithms for one-dimensional ROFDPs.

- $D = 1$
- $\mathcal{W} = \text{tile}(\text{hyperbar}(W), 1, 0)$

The following subsections consider various choices for l and u .

A.1.1 Window Sorting

This section considers the problem of sorting the entire window. Hence $l = 0$ and $u = W-1$.

| W | 0) | 1) | 2) | 3) | 4) | 5) | 6) | 7) | 8) | 9) | 10) | Best strategies |
|----|---------|--------|---------|---------|--------|--------|--------|--------|--------|--------|--------|----------------------------------|
| 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 2 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 3 | 6.00 | 5.00 | 5.00 | 6.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 1, 2, 4, 5, 6, 7, 8, 9, 10 |
| 4 | 8.00 | 11.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 0, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 5 | 16.00 | 12.00 | 12.00 | 16.00 | 13.50 | 12.00 | 12.00 | 12.00 | 12.00 | 12.00 | 12.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 6 | 18.00 | 22.00 | 20.00 | 20.00 | 17.00 | 17.00 | 17.00 | 17.50 | 16.00 | 16.00 | 16.00 | 8, 9, 10 |
| 7 | 28.00 | 20.50 | 20.50 | 28.00 | 22.50 | 20.50 | 20.50 | 20.50 | 20.50 | 20.50 | 20.50 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 8 | 26.00 | 34.50 | 26.00 | 26.00 | 26.00 | 29.00 | 26.00 | 26.00 | 25.33 | 25.33 | 25.33 | 8, 9, 10 |
| 9 | 46.00 | 30.50 | 29.00 | 42.00 | 35.75 | 29.00 | 29.00 | 29.00 | 29.00 | 29.00 | 29.00 | 2, 5, 6, 7, 8, 9, 10 |
| 10 | 42.00 | 48.50 | 50.00 | 50.00 | 41.25 | 38.00 | 38.00 | 36.75 | 36.67 | 36.67 | 36.67 | 8, 9, 10 |
| 11 | 60.00 | 39.00 | 39.00 | 60.00 | 47.75 | 39.00 | 39.00 | 39.00 | 39.00 | 39.00 | 39.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 12 | 52.00 | 61.00 | 60.00 | 60.00 | 46.50 | 53.50 | 46.50 | 47.00 | 45.33 | 45.33 | 45.33 | 8, 9, 10 |
| 13 | 80.00 | 51.50 | 54.00 | 74.00 | 58.00 | 49.50 | 49.50 | 50.75 | 48.00 | 48.00 | 48.00 | 8, 9, 10 |
| 14 | 70.00 | 77.50 | 96.00 | 82.00 | 65.50 | 62.50 | 62.50 | 58.00 | 56.67 | 56.67 | 56.67 | 8, 9, 10 |
| 15 | 94.00 | 59.25 | 59.25 | 94.00 | 72.75 | 59.25 | 59.25 | 59.25 | 59.25 | 59.25 | 59.25 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 16 | 76.00 | 89.25 | 76.00 | 76.00 | 76.00 | 82.00 | 76.00 | 68.00 | 67.33 | 67.33 | 67.33 | 8, 9, 10 |
| 17 | 122.00 | 77.00 | 70.00 | 108.00 | 91.63 | 74.00 | 70.00 | 70.00 | 68.67 | 68.67 | 68.67 | 8, 9, 10 |
| 18 | 106.00 | 111.00 | 124.00 | 124.00 | 100.88 | 91.00 | 87.00 | 79.38 | 77.17 | 77.17 | 77.17 | 8, 9, 10 |
| 19 | 140.00 | 81.50 | 79.50 | 138.00 | 110.38 | 79.50 | 79.50 | 79.50 | 79.50 | 79.50 | 79.50 | 2, 5, 6, 7, 8, 9, 10 |
| 20 | 112.00 | 119.50 | 142.00 | 142.00 | 110.50 | 106.00 | 91.00 | 90.25 | 87.33 | 87.33 | 87.33 | 8, 9, 10 |
| 21 | 164.00 | 103.00 | 111.00 | 158.00 | 123.88 | 94.00 | 94.00 | 94.13 | 91.83 | 91.83 | 91.60 | 10 |
| 22 | 140.00 | 145.00 | 202.00 | 168.00 | 131.63 | 115.00 | 115.00 | 101.25 | 100.00 | 99.50 | 99.50 | 9, 10 |
| 23 | 182.00 | 101.50 | 101.50 | 182.00 | 138.88 | 101.50 | 101.50 | 101.50 | 101.50 | 101.50 | 101.50 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 24 | 142.00 | 147.50 | 166.00 | 166.00 | 122.25 | 138.25 | 122.25 | 112.00 | 109.17 | 109.17 | 109.17 | 8, 9, 10 |
| 25 | 212.00 | 129.00 | 131.00 | 190.00 | 140.00 | 122.25 | 112.25 | 116.38 | 111.33 | 111.33 | 111.33 | 8, 9, 10 |
| 26 | 180.00 | 179.00 | 238.00 | 204.00 | 151.50 | 147.25 | 137.25 | 123.75 | 119.56 | 119.56 | 119.56 | 8, 9, 10 |
| 27 | 230.00 | 126.75 | 130.00 | 220.00 | 164.00 | 123.75 | 123.75 | 125.75 | 121.67 | 121.67 | 121.67 | 8, 9, 10 |
| 28 | 180.00 | 180.75 | 264.00 | 224.00 | 166.75 | 162.25 | 139.25 | 135.25 | 132.75 | 132.00 | 132.00 | 9, 10 |
| 29 | 254.00 | 160.00 | 188.00 | 244.00 | 184.25 | 142.25 | 142.25 | 139.63 | 134.17 | 134.17 | 134.17 | 8, 9, 10 |
| 30 | 214.00 | 218.00 | 348.00 | 256.00 | 195.75 | 171.25 | 171.25 | 147.00 | 142.89 | 142.75 | 142.75 | 9, 10 |
| 31 | 272.00 | 145.63 | 145.63 | 272.00 | 203.88 | 145.63 | 145.63 | 145.63 | 143.88 | 143.88 | 143.88 | 9, 10 |
| 32 | 206.00 | 207.63 | 206.00 | 206.00 | 206.00 | 199.50 | 206.00 | 156.63 | 154.33 | 153.00 | 153.00 | 9, 10 |
| 33 | 312.00 | 193.50 | 167.00 | 270.00 | 228.94 | 175.50 | 167.00 | 161.38 | 157.42 | 155.38 | 155.38 | 9, 10 |
| 34 | 264.00 | 259.50 | 302.00 | 302.00 | 243.56 | 208.50 | 200.00 | 169.13 | 165.33 | 164.75 | 164.75 | 9, 10 |
| 35 | 334.00 | 178.00 | 168.00 | 324.00 | 257.94 | 173.00 | 168.00 | 168.00 | 165.67 | 165.67 | 165.67 | 8, 9, 10 |
| 36 | 260.00 | 248.00 | 336.00 | 336.00 | 261.25 | 223.50 | 187.50 | 180.88 | 175.22 | 175.22 | 175.22 | 8, 9, 10 |
| 37 | 364.00 | 227.50 | 240.00 | 356.00 | 278.81 | 195.50 | 190.50 | 185.19 | 177.44 | 177.44 | 177.44 | 8, 9, 10 |
| 38 | 306.00 | 301.50 | 444.00 | 370.00 | 288.94 | 232.50 | 227.50 | 192.38 | 185.89 | 185.89 | 185.89 | 8, 9, 10 |
| 39 | 386.00 | 192.25 | 189.75 | 388.00 | 298.44 | 189.75 | 189.75 | 189.75 | 189.17 | 186.00 | 186.00 | 9, 10 |
| 40 | 290.00 | 270.25 | 376.00 | 376.00 | 282.50 | 254.50 | 218.50 | 203.63 | 198.33 | 197.50 | 197.50 | 9, 10 |
| 41 | 422.00 | 259.50 | 268.00 | 402.00 | 303.06 | 222.50 | 200.50 | 208.06 | 198.00 | 198.00 | 198.00 | 8, 9, 10 |
| 42 | 354.00 | 341.50 | 496.00 | 418.00 | 315.69 | 263.50 | 241.50 | 215.56 | 209.44 | 209.00 | 209.00 | 9, 10 |
| 43 | 444.00 | 229.25 | 238.50 | 436.00 | 329.06 | 216.00 | 216.00 | 217.50 | 212.89 | 212.25 | 212.00 | 10 |
| 44 | 342.00 | 315.25 | 538.00 | 442.00 | 331.25 | 278.50 | 239.50 | 227.00 | 221.00 | 219.63 | 219.63 | 9, 10 |
| 45 | 474.00 | 295.50 | 357.00 | 464.00 | 349.19 | 242.50 | 242.50 | 231.19 | 223.83 | 222.75 | 222.75 | 9, 10 |
| 46 | 396.00 | 385.50 | 670.00 | 478.00 | 359.81 | 287.50 | 287.50 | 238.38 | 232.00 | 230.00 | 230.00 | 9, 10 |
| 47 | 496.00 | 235.75 | 235.75 | 496.00 | 366.94 | 235.75 | 235.75 | 235.75 | 232.75 | 230.00 | 230.00 | 9, 10 |
| 48 | 368.00 | 329.75 | 432.00 | 432.00 | 310.13 | 319.63 | 310.13 | 249.06 | 242.28 | 239.92 | 239.92 | 9, 10 |
| 49 | 538.00 | 322.50 | 312.00 | 474.00 | 335.75 | 279.63 | 255.13 | 253.81 | 244.44 | 242.00 | 242.00 | 9, 10 |
| 50 | 450.00 | 420.50 | 576.00 | 498.00 | 353.00 | 328.63 | 304.13 | 261.56 | 253.61 | 251.25 | 251.25 | 9, 10 |
| 51 | 560.00 | 280.25 | 280.50 | 520.00 | 370.50 | 269.13 | 256.13 | 263.81 | 255.25 | 254.42 | 254.42 | 9, 10 |
| 52 | 430.00 | 382.25 | 626.00 | 530.00 | 376.63 | 343.63 | 283.63 | 273.63 | 264.94 | 262.33 | 262.33 | 9, 10 |
| 53 | 590.00 | 363.50 | 417.00 | 554.00 | 398.00 | 299.63 | 286.63 | 277.75 | 263.78 | 263.78 | 263.78 | 8, 9, 10 |
| 54 | 492.00 | 469.50 | 782.00 | 570.00 | 411.75 | 352.63 | 339.63 | 285.44 | 274.78 | 272.75 | 272.75 | 9, 10 |
| 55 | 612.00 | 285.88 | 290.00 | 590.00 | 425.00 | 281.88 | 281.88 | 284.63 | 278.50 | 271.42 | 271.42 | 9, 10 |
| 56 | 454.00 | 395.88 | 682.00 | 578.00 | 412.38 | 374.63 | 318.63 | 296.50 | 286.67 | 284.17 | 284.17 | 9, 10 |
| 57 | 648.00 | 401.50 | 453.00 | 610.00 | 438.13 | 326.63 | 292.63 | 301.31 | 290.00 | 286.75 | 286.75 | 9, 10 |
| 58 | 540.00 | 515.50 | 850.00 | 630.00 | 455.63 | 383.63 | 349.63 | 309.19 | 297.83 | 296.58 | 296.58 | 9, 10 |
| 59 | 670.00 | 337.50 | 371.00 | 652.00 | 474.13 | 312.13 | 312.13 | 311.38 | 298.56 | 298.56 | 298.56 | 8, 9, 10 |
| 60 | 512.00 | 455.50 | 908.00 | 660.00 | 480.88 | 398.63 | 343.63 | 321.06 | 308.00 | 307.83 | 307.83 | 9, 10 |
| 61 | 700.00 | 442.50 | 574.00 | 686.00 | 504.38 | 346.63 | 346.63 | 325.38 | 310.00 | 310.00 | 310.00 | 8, 9, 10 |
| 62 | 582.00 | 564.50 | 1088.00 | 702.00 | 519.88 | 407.63 | 407.63 | 332.69 | 318.67 | 318.33 | 318.33 | 9, 10 |
| 63 | 722.00 | 327.81 | 327.81 | 722.00 | 527.94 | 327.81 | 327.81 | 327.81 | 324.61 | 318.06 | 318.06 | 9, 10 |
| 64 | 528.00 | 453.81 | 528.00 | 528.00 | 445.75 | 528.00 | 341.44 | 331.61 | 329.08 | 329.08 | 329.08 | 9, 10 |
| 65 | 776.00 | 483.00 | 392.00 | 656.00 | 559.53 | 389.75 | 392.00 | 346.44 | 333.17 | 331.13 | 331.13 | 9, 10 |
| 66 | 648.00 | 613.00 | 720.00 | 720.00 | 580.97 | 454.75 | 457.00 | 354.44 | 342.83 | 339.96 | 339.96 | 9, 10 |
| 67 | 802.00 | 398.50 | 360.50 | 758.00 | 601.84 | 371.25 | 360.50 | 356.88 | 346.83 | 343.13 | 343.13 | 9, 10 |
| 68 | 614.00 | 532.50 | 786.00 | 786.00 | 610.00 | 469.75 | 396.00 | 366.75 | 354.72 | 350.33 | 350.33 | 9, 10 |
| 69 | 838.00 | 530.00 | 529.00 | 814.00 | 633.72 | 409.75 | 399.00 | 371.25 | 358.22 | 353.33 | 353.33 | 9, 10 |
| 70 | 698.00 | 668.00 | 990.00 | 836.00 | 648.16 | 478.75 | 468.00 | 378.75 | 366.94 | 360.38 | 360.38 | 9, 10 |
| 71 | 864.00 | 387.00 | 374.00 | 862.00 | 661.97 | 380.00 | 374.00 | 374.00 | 364.61 | 359.17 | 359.17 | 9, 10 |
| 72 | 638.00 | 529.00 | 858.00 | 858.00 | 648.63 | 500.75 | 418.75 | 390.38 | 374.44 | 370.92 | 370.92 | 9, 10 |
| 73 | 908.00 | 571.00 | 573.00 | 888.00 | 674.59 | 436.75 | 384.75 | 395.19 | 376.94 | 373.29 | 373.29 | 9, 10 |
| 74 | 756.00 | 717.00 | 1074.00 | 908.00 | 690.78 | 509.75 | 457.75 | 403.06 | 386.33 | 382.92 | 382.92 | 9, 10 |
| 75 | 934.00 | 458.75 | 471.00 | 930.00 | 707.59 | 414.25 | 408.25 | 405.19 | 390.83 | 386.33 | 386.33 | 9, 10 |
| 76 | 712.00 | 608.75 | 1148.00 | 940.00 | 711.50 | 524.75 | 447.75 | 414.75 | 397.44 | 394.58 | 394.58 | 9, 10 |
| 77 | 970.00 | 614.00 | 726.00 | 966.00 | 732.66 | 456.75 | 450.75 | 418.91 | 396.89 | 396.89 | 396.89 | 8, 9, 10 |
| 78 | 806.00 | 768.00 | 1376.00 | 984.00 | 744.84 | 533.75 | 527.75 | 426.06 | 408.67 | 405.33 | 405.33 | 9, 10 |
| 79 | 996.00 | 422.88 | 419.88 | 1006.00 | 753.47 | 419.88 | 419.88 | 419.88 | 412.50 | 404.17 | 404.17 | 9, 10 |
| 80 | 724.00 | 580.88 | 946.00 | 946.00 | 696.63 | 563.75 | 522.25 | 437.34 | 420.30 | 416.08 | 416.08 | 9, 10 |
| 81 | 1046.00 | 648.00 | 633.00 | 990.00 | 725.97 | 491.75 | 435.25 | 442.09 | 424.61 | 418.38 | 418.38 | 9, 10 |
| 82 | 870.00 | 810.00 | 1186.00 | 1016.00 | 745.28 | 572.75 | 516.25 | 449.88 | 433.22 | 427.88 | 427.88 | 9, 10 |
| 83 | 1072.00 | 516.75 | 521.00 | 1040.00 | 764.66 | 465.25 | 436.25 | 452.09 | 434.83 | 431.08 | 431.08 | 9, 10 |
| 84 | 816.00 | 682.75 | 1268.00 | 1052.00 | 771.13 | 587.75 | 479.75 | 462.03 | 444.06 | 439.13 | 439.13 | |

| | | | | | | | | | | | | |
|-----|---------|---------|---------|---------|---------|---------|---------|--------|--------|--------|--------|-------|
| 91 | 1204.00 | 580.00 | 651.50 | 1188.00 | 868.66 | 508.25 | 508.25 | 499.38 | 480.47 | 475.69 | 475.69 | 9, 10 |
| 92 | 914.00 | 762.00 | 1710.00 | 1198.00 | 873.63 | 642.75 | 555.75 | 508.97 | 488.31 | 483.06 | 483.06 | 9, 10 |
| 93 | 1240.00 | 786.00 | 1039.00 | 1226.00 | 896.34 | 558.75 | 558.75 | 513.13 | 491.22 | 486.25 | 486.25 | 9, 10 |
| 94 | 1028.00 | 972.00 | 1986.00 | 1244.00 | 909.91 | 651.75 | 651.75 | 520.28 | 499.42 | 493.47 | 493.47 | 9, 10 |
| 95 | 1266.00 | 513.88 | 513.88 | 1266.00 | 915.97 | 513.88 | 513.88 | 513.88 | 500.50 | 491.88 | 491.88 | 9, 10 |
| 96 | 914.00 | 703.88 | 1074.00 | 1074.00 | 762.06 | 693.81 | 762.06 | 530.59 | 510.39 | 503.50 | 503.50 | 9, 10 |
| 97 | 1322.00 | 801.00 | 729.00 | 1150.00 | 797.00 | 605.81 | 578.06 | 535.53 | 512.67 | 505.71 | 505.71 | 9, 10 |
| 98 | 1098.00 | 995.00 | 1362.00 | 1192.00 | 821.63 | 702.81 | 675.06 | 543.47 | 522.28 | 515.13 | 515.13 | 9, 10 |
| 99 | 1348.00 | 631.75 | 609.00 | 1224.00 | 846.00 | 571.31 | 538.56 | 545.88 | 526.75 | 518.38 | 518.38 | 9, 10 |
| 100 | 1024.00 | 829.75 | 1460.00 | 1244.00 | 857.31 | 717.81 | 590.06 | 555.84 | 533.64 | 526.42 | 526.42 | 9, 10 |
| 101 | 1384.00 | 857.00 | 930.00 | 1274.00 | 884.88 | 625.81 | 593.06 | 560.19 | 535.19 | 529.63 | 529.63 | 9, 10 |
| 102 | 1148.00 | 1059.00 | 1760.00 | 1296.00 | 903.00 | 726.81 | 694.06 | 567.81 | 544.92 | 536.79 | 536.79 | 9, 10 |
| 103 | 1410.00 | 587.63 | 586.25 | 1322.00 | 920.50 | 572.06 | 556.06 | 567.28 | 548.86 | 535.38 | 535.38 | 9, 10 |
| 104 | 1032.00 | 793.63 | 1564.00 | 1316.00 | 910.56 | 748.81 | 616.81 | 579.78 | 556.75 | 548.09 | 548.09 | 9, 10 |
| 105 | 1454.00 | 906.00 | 990.00 | 1352.00 | 941.13 | 652.81 | 566.81 | 584.38 | 560.17 | 550.41 | 550.41 | 9, 10 |
| 106 | 1206.00 | 1116.00 | 1876.00 | 1376.00 | 961.75 | 757.81 | 671.81 | 592.13 | 568.14 | 560.00 | 560.00 | 9, 10 |
| 107 | 1480.00 | 700.00 | 759.50 | 1402.00 | 983.13 | 614.31 | 598.31 | 594.03 | 564.22 | 563.31 | 563.31 | 9, 10 |
| 108 | 1122.00 | 914.00 | 1710.00 | 1414.00 | 991.31 | 772.81 | 653.81 | 604.28 | 575.78 | 571.44 | 571.44 | 9, 10 |
| 109 | 1516.00 | 956.00 | 1207.00 | 1444.00 | 1017.25 | 672.81 | 656.81 | 608.44 | 577.70 | 574.75 | 574.75 | 9, 10 |
| 110 | 1256.00 | 1174.00 | 2306.00 | 1464.00 | 1033.88 | 781.81 | 765.81 | 615.81 | 587.26 | 582.03 | 582.03 | 9, 10 |
| 111 | 1542.00 | 612.94 | 618.00 | 1488.00 | 1047.00 | 607.94 | 607.94 | 611.44 | 591.96 | 581.41 | 581.41 | 9, 10 |
| 112 | 1112.00 | 834.94 | 1684.00 | 1428.00 | 994.19 | 811.81 | 738.31 | 626.22 | 598.70 | 592.78 | 592.78 | 9, 10 |
| 113 | 1592.00 | 997.00 | 1066.00 | 1480.00 | 1029.81 | 707.81 | 619.31 | 631.22 | 598.33 | 594.97 | 594.97 | 9, 10 |
| 114 | 1320.00 | 1223.00 | 2020.00 | 1512.00 | 1055.06 | 820.81 | 732.31 | 639.25 | 609.89 | 604.25 | 604.25 | 9, 10 |
| 115 | 1618.00 | 765.00 | 817.50 | 1542.00 | 1080.56 | 665.31 | 620.31 | 641.66 | 613.59 | 607.41 | 607.41 | 9, 10 |
| 116 | 1226.00 | 995.00 | 2134.00 | 1568.00 | 1092.69 | 835.81 | 679.81 | 651.88 | 621.96 | 615.16 | 615.16 | 9, 10 |
| 117 | 1654.00 | 1052.00 | 1299.00 | 1590.00 | 1122.06 | 727.81 | 682.81 | 656.16 | 625.11 | 618.19 | 618.19 | 9, 10 |
| 118 | 1370.00 | 1286.00 | 2482.00 | 1612.00 | 1141.81 | 844.81 | 799.81 | 663.91 | 633.56 | 625.31 | 625.31 | 9, 10 |
| 119 | 1680.00 | 695.25 | 734.50 | 1638.00 | 1161.06 | 662.06 | 662.06 | 663.19 | 636.26 | 625.42 | 625.42 | 9, 10 |
| 120 | 1226.00 | 933.25 | 2254.00 | 1630.00 | 1152.44 | 866.81 | 730.81 | 675.41 | 644.41 | 636.72 | 636.72 | 9, 10 |
| 121 | 1724.00 | 1102.00 | 1367.00 | 1670.00 | 1186.19 | 754.81 | 672.81 | 680.16 | 646.59 | 639.13 | 639.13 | 9, 10 |
| 122 | 1428.00 | 1344.00 | 2614.00 | 1696.00 | 1209.69 | 875.81 | 793.81 | 688.00 | 655.85 | 648.25 | 648.25 | 9, 10 |
| 123 | 1750.00 | 834.25 | 988.00 | 1724.00 | 1234.19 | 708.31 | 708.31 | 690.06 | 660.30 | 651.72 | 651.72 | 9, 10 |
| 124 | 1324.00 | 1080.25 | 2736.00 | 1736.00 | 1244.94 | 890.81 | 771.81 | 699.75 | 667.15 | 659.16 | 659.16 | 9, 10 |
| 125 | 1786.00 | 1153.00 | 1616.00 | 1768.00 | 1274.44 | 774.81 | 774.81 | 703.97 | 662.67 | 662.41 | 662.41 | 9, 10 |
| 126 | 1478.00 | 1403.00 | 3108.00 | 1788.00 | 1293.94 | 899.81 | 899.81 | 711.19 | 677.78 | 669.72 | 669.72 | 9, 10 |
| 127 | 1812.00 | 701.91 | 701.91 | 1812.00 | 1300.97 | 701.91 | 701.91 | 701.91 | 681.70 | 666.84 | 666.84 | 9, 10 |
| 128 | 1298.00 | 955.91 | 1298.00 | 1298.00 | 1298.00 | 948.88 | 1298.00 | 718.81 | 690.04 | 678.94 | 678.94 | 9, 10 |
| 129 | 1882.00 | 1193.50 | 905.00 | 1554.00 | 1339.30 | 828.88 | 905.00 | 723.91 | 693.56 | 681.41 | 681.41 | 9, 10 |
| 130 | 1562.00 | 1451.50 | 1682.00 | 1682.00 | 1368.83 | 957.88 | 1034.00 | 732.00 | 701.81 | 691.16 | 691.16 | 9, 10 |
| 131 | 1912.00 | 904.00 | 777.00 | 1752.00 | 1397.61 | 778.38 | 777.00 | 734.53 | 703.15 | 694.72 | 694.72 | 9, 10 |
| 132 | 1450.00 | 1166.00 | 1812.00 | 1812.00 | 1412.06 | 972.88 | 844.50 | 744.56 | 713.89 | 703.06 | 703.06 | 9, 10 |
| 133 | 1954.00 | 1261.50 | 1170.00 | 1856.00 | 1443.58 | 848.88 | 847.50 | 749.13 | 716.15 | 706.66 | 706.66 | 9, 10 |
| 134 | 1620.00 | 1527.50 | 2208.00 | 1894.00 | 1463.98 | 981.88 | 960.00 | 756.69 | 725.89 | 714.09 | 714.09 | 9, 10 |
| 135 | 1994.00 | 807.75 | 758.25 | 1936.00 | 1483.86 | 771.13 | 758.25 | 756.44 | 730.54 | 710.53 | 710.53 | 9, 10 |
| 136 | 1448.00 | 1077.75 | 1948.00 | 1948.00 | 1474.81 | 1003.88 | 836.00 | 768.75 | 737.39 | 723.72 | 723.72 | 9, 10 |
| 137 | 2036.00 | 1316.50 | 1246.00 | 1986.00 | 1508.27 | 875.88 | 769.00 | 773.59 | 736.35 | 726.13 | 726.13 | 9, 10 |
| 138 | 1688.00 | 1590.50 | 2356.00 | 2014.00 | 1530.05 | 1012.88 | 906.00 | 761.53 | 747.91 | 735.88 | 735.88 | 9, 10 |
| 139 | 2066.00 | 978.25 | 967.50 | 2044.00 | 1552.42 | 821.38 | 808.50 | 763.69 | 751.39 | 739.25 | 739.25 | 9, 10 |
| 140 | 1564.00 | 1256.25 | 2494.00 | 2062.00 | 1560.06 | 1027.88 | 880.00 | 793.47 | 759.98 | 747.53 | 747.53 | 9, 10 |
| 141 | 2108.00 | 1370.50 | 1527.00 | 2096.00 | 1586.73 | 895.88 | 885.00 | 797.78 | 761.93 | 750.91 | 750.91 | 9, 10 |
| 142 | 1746.00 | 1652.50 | 2914.00 | 2122.00 | 1602.64 | 1036.88 | 1024.00 | 805.09 | 770.17 | 758.31 | 758.31 | 9, 10 |
| 143 | 2138.00 | 812.00 | 796.00 | 2152.00 | 1614.98 | 803.00 | 796.00 | 796.00 | 765.92 | 755.92 | 755.92 | 9, 10 |
| 144 | 1536.00 | 1098.00 | 2100.00 | 2100.00 | 1560.13 | 1066.88 | 954.38 | 816.41 | 778.93 | 769.06 | 769.06 | 9, 10 |
| 145 | 2198.00 | 1414.50 | 1338.00 | 2148.00 | 1596.08 | 930.88 | 803.38 | 821.44 | 781.06 | 771.41 | 771.41 | 9, 10 |
| 146 | 1822.00 | 1704.50 | 2532.00 | 2178.00 | 1620.17 | 1075.88 | 948.38 | 829.44 | 790.61 | 780.78 | 780.78 | 9, 10 |
| 147 | 2228.00 | 1046.25 | 1033.50 | 2206.00 | 1644.23 | 872.38 | 804.38 | 831.81 | 795.26 | 784.03 | 784.03 | 9, 10 |
| 148 | 1686.00 | 1340.25 | 2678.00 | 2222.00 | 1653.63 | 1090.88 | 879.88 | 842.03 | 801.96 | 791.84 | 791.84 | 9, 10 |
| 149 | 2270.00 | 1471.50 | 1635.00 | 2252.00 | 1681.05 | 950.88 | 882.88 | 846.31 | 801.48 | 794.94 | 794.94 | 9, 10 |
| 150 | 1880.00 | 1769.50 | 3122.00 | 2274.00 | 1697.64 | 1099.88 | 1031.88 | 853.94 | 812.65 | 802.16 | 802.16 | 9, 10 |
| 151 | 2300.00 | 918.38 | 930.50 | 2300.00 | 1713.55 | 861.13 | 854.13 | 853.22 | 816.17 | 801.75 | 801.75 | 9, 10 |
| 152 | 1674.00 | 1220.38 | 2830.00 | 2294.00 | 1700.56 | 1121.88 | 938.88 | 865.44 | 824.20 | 813.72 | 813.72 | 9, 10 |
| 153 | 2352.00 | 1523.50 | 1719.00 | 2332.00 | 1731.42 | 977.88 | 864.88 | 870.08 | 827.20 | 816.11 | 816.11 | 9, 10 |
| 154 | 1948.00 | 1829.50 | 3286.00 | 2358.00 | 1751.02 | 1130.88 | 1017.88 | 877.80 | 835.31 | 825.19 | 825.19 | 9, 10 |
| 155 | 2382.00 | 1117.50 | 1244.00 | 2386.00 | 1771.42 | 915.38 | 908.38 | 879.78 | 838.11 | 828.45 | 828.45 | 9, 10 |
| 156 | 1800.00 | 1427.50 | 3440.00 | 2400.00 | 1777.25 | 1145.88 | 987.88 | 889.36 | 845.93 | 835.86 | 835.86 | 9, 10 |
| 157 | 2424.00 | 1576.50 | 2032.00 | 2432.00 | 1802.33 | 997.88 | 990.88 | 893.47 | 847.96 | 838.98 | 838.98 | 9, 10 |
| 158 | 2006.00 | 1890.50 | 3908.00 | 2454.00 | 1816.67 | 1154.88 | 1147.88 | 900.58 | 857.20 | 846.17 | 846.17 | 9, 10 |
| 159 | 2454.00 | 893.44 | 889.94 | 2480.00 | 1823.48 | 889.94 | 889.94 | 889.94 | 861.94 | 840.88 | 840.88 | 9, 10 |
| 160 | 1750.00 | 1211.44 | 2292.00 | 2292.00 | 1868.81 | 1193.88 | 1234.13 | 911.70 | 868.72 | 857.00 | 857.00 | 9, 10 |
| 161 | 2520.00 | 1599.50 | 1466.00 | 2370.00 | 1708.33 | 1041.88 | 954.13 | 916.61 | 863.93 | 859.40 | 859.40 | 9, 10 |
| 162 | 2088.00 | 1921.50 | 2772.00 | 2414.00 | 1735.92 | 1202.88 | 1115.13 | 924.53 | 877.72 | 869.10 | 869.10 | 9, 10 |
| 163 | 2550.00 | 1177.25 | 1137.50 | 2448.00 | 1763.14 | 975.38 | 898.63 | 926.91 | 881.48 | 872.60 | 871.13 | 10 |
| 164 | 1928.00 | 1503.25 | 2934.00 | 2470.00 | 1775.75 | 1217.88 | 982.13 | 936.91 | 889.63 | 880.90 | 879.05 | 10 |
| 165 | 2592.00 | 1664.50 | 1795.00 | 2502.00 | 1805.92 | 1061.88 | 985.13 | 941.23 | 893.00 | 884.42 | 882.28 | 10 |
| 166 | 2146.00 | 1994.50 | 3426.00 | 2526.00 | 1825.14 | 1226.88 | 1150.13 | 948.78 | 901.17 | 891.77 | 889.28 | 10 |
| 167 | 2622.00 | 1026.38 | 1026.50 | 2554.00 | 1843.64 | 960.13 | 924.13 | 948.23 | 902.24 | 888.00 | 883.83 | 10 |
| 168 | 1906.00 | 1360.38 | 3102.00 | 2550.00 | 1833.25 | 1248.88 | 1016.88 | 960.86 | 912.54 | 901.02 | 900.05 | 10 |
| 169 | 2674.00 | 1721.50 | 1887.00 | 2588.00 | 1865.98 | 1088.88 | 934.88 | 965.47 | 914.67 | 903.36 | 902.60 | 10 |
| 170 | 2214.00 | 2059.50 | 3606.00 | 2614.00 | 1887.33 | 1257.88 | 1103.88 | 973.25 | 924.19 | 912.91 | 912.38 | 10 |
| 171 | 2704.00 | 1253.50 | 1368.00 | 2642.00 | 1909.33 | 1018.38 | 982.38 | 975.11 | 928.59 | 916.19 | 915.95 | 10 |
| 172 | 2042.00 | 1595.50 | 3776.00 | 2656.00 | 1916.75 | 1272.88 | 1069.88 | 985.22 | 935.26 | 924.22 | 924.20 | 10 |
| 173 | 2746.00 | 1778.50 | 2232.00 | 2688.00 | 1943.14 | 1108.88 | 1072.88 | 989.33 | 936.33 | | | |

| | | | | | | | | | | | | |
|-----|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|-------|
| 190 | 2532.00 | 2378.50 | 5478.00 | 3066.00 | 2197.95 | 1399.88 | 1399.88 | 1091.06 | 1037.41 | 1021.02 | 1021.02 | 9, 10 |
| 191 | 3092.00 | 1079.94 | 1079.94 | 3092.00 | 2201.98 | 1079.94 | 1079.94 | 1079.94 | 1040.02 | 1016.88 | 1016.88 | 9, 10 |
| 192 | 2196.00 | 1461.94 | 2580.00 | 2580.00 | 1822.03 | 1452.91 | 1822.03 | 1100.84 | 1048.48 | 1030.27 | 1030.27 | 9, 10 |
| 193 | 3164.00 | 1968.50 | 1674.00 | 2722.00 | 1867.56 | 1268.91 | 1301.03 | 1105.86 | 1050.72 | 1032.60 | 1032.60 | 9, 10 |
| 194 | 2620.00 | 2354.50 | 3156.00 | 2798.00 | 1901.00 | 1461.91 | 1494.03 | 1113.88 | 1060.33 | 1042.23 | 1042.23 | 9, 10 |
| 195 | 3194.00 | 1438.25 | 1321.50 | 2848.00 | 1933.88 | 1186.41 | 1141.03 | 1116.36 | 1065.07 | 1045.69 | 1045.69 | 9, 10 |
| 196 | 2412.00 | 1828.25 | 3350.00 | 2886.00 | 1952.03 | 1476.91 | 1240.53 | 1126.39 | 1072.20 | 1053.90 | 1053.90 | 9, 10 |
| 197 | 3236.00 | 2047.50 | 2067.00 | 2926.00 | 1987.75 | 1288.91 | 1243.53 | 1130.84 | 1070.43 | 1057.38 | 1057.38 | 9, 10 |
| 198 | 2678.00 | 2441.50 | 3938.00 | 2958.00 | 2012.19 | 1485.91 | 1440.53 | 1138.44 | 1082.46 | 1064.67 | 1064.67 | 9, 10 |
| 199 | 3266.00 | 1241.38 | 1206.50 | 2994.00 | 2036.00 | 1163.16 | 1122.28 | 1138.06 | 1086.20 | 1061.00 | 1061.00 | 9, 10 |
| 200 | 2370.00 | 1639.38 | 3550.00 | 2998.00 | 2030.66 | 1507.91 | 1231.03 | 1150.61 | 1094.44 | 1074.52 | 1074.52 | 9, 10 |
| 201 | 3318.00 | 2112.50 | 2175.00 | 3040.00 | 2068.63 | 1315.91 | 1133.03 | 1155.31 | 1097.70 | 1076.79 | 1076.79 | 9, 10 |
| 202 | 2746.00 | 2514.50 | 4150.00 | 3070.00 | 2094.81 | 1516.91 | 1334.03 | 1163.16 | 1105.78 | 1086.44 | 1086.44 | 9, 10 |
| 203 | 3348.00 | 1522.50 | 1592.00 | 3102.00 | 2121.63 | 1229.41 | 1188.53 | 1165.19 | 1106.52 | 1089.71 | 1089.71 | 9, 10 |
| 204 | 2526.00 | 1928.50 | 4352.00 | 3120.00 | 2133.53 | 1531.91 | 1292.03 | 1175.22 | 1116.41 | 1097.88 | 1097.88 | 9, 10 |
| 205 | 3390.00 | 2175.50 | 2584.00 | 3156.00 | 2164.69 | 1335.91 | 1295.03 | 1179.44 | 1118.46 | 1101.17 | 1101.17 | 9, 10 |
| 206 | 2804.00 | 2585.50 | 4964.00 | 3182.00 | 2184.88 | 1540.91 | 1500.03 | 1186.77 | 1128.22 | 1108.42 | 1108.42 | 9, 10 |
| 207 | 3420.00 | 1207.06 | 1204.13 | 3212.00 | 2201.50 | 1187.03 | 1168.03 | 1182.77 | 1132.87 | 1107.46 | 1107.46 | 9, 10 |
| 208 | 2442.00 | 1621.06 | 3766.00 | 3158.00 | 2150.66 | 1570.91 | 1382.41 | 1198.58 | 1139.74 | 1119.65 | 1119.65 | 9, 10 |
| 209 | 3480.00 | 2228.50 | 2299.00 | 3214.00 | 2192.00 | 1370.91 | 1167.41 | 1203.41 | 1141.15 | 1121.81 | 1121.81 | 9, 10 |
| 210 | 2880.00 | 2646.50 | 4390.00 | 3250.00 | 2221.31 | 1579.91 | 1376.41 | 1211.31 | 1150.81 | 1131.21 | 1131.21 | 9, 10 |
| 211 | 3510.00 | 1599.50 | 1674.00 | 3284.00 | 2250.69 | 1280.41 | 1168.41 | 1213.52 | 1154.52 | 1134.27 | 1134.27 | 9, 10 |
| 212 | 2648.00 | 2021.50 | 4600.00 | 3304.00 | 2265.16 | 1594.91 | 1275.91 | 1223.66 | 1162.46 | 1142.04 | 1142.04 | 9, 10 |
| 213 | 3552.00 | 2294.50 | 2724.00 | 3340.00 | 2298.06 | 1390.91 | 1278.91 | 1227.84 | 1165.78 | 1145.04 | 1145.04 | 9, 10 |
| 214 | 2938.00 | 2720.50 | 5236.00 | 3366.00 | 2319.88 | 1603.91 | 1491.91 | 1235.39 | 1174.07 | 1152.17 | 1152.17 | 9, 10 |
| 215 | 3582.00 | 1361.00 | 1426.75 | 3396.00 | 2341.06 | 1253.16 | 1234.16 | 1234.45 | 1166.28 | 1153.03 | 1153.03 | 9, 10 |
| 216 | 2596.00 | 1791.00 | 4816.00 | 3392.00 | 2333.03 | 1625.91 | 1350.91 | 1247.52 | 1178.41 | 1163.38 | 1163.38 | 9, 10 |
| 217 | 3634.00 | 2354.50 | 2840.00 | 3436.00 | 2369.69 | 1417.91 | 1244.91 | 1252.16 | 1180.48 | 1165.60 | 1165.60 | 9, 10 |
| 218 | 3006.00 | 2788.50 | 5464.00 | 3466.00 | 2394.75 | 1634.91 | 1461.91 | 1259.92 | 1189.91 | 1174.75 | 1174.75 | 9, 10 |
| 219 | 3664.00 | 1678.75 | 1964.50 | 3498.00 | 2420.69 | 1323.41 | 1304.41 | 1261.84 | 1194.54 | 1178.02 | 1178.02 | 9, 10 |
| 220 | 2762.00 | 2116.75 | 5682.00 | 3514.00 | 2431.66 | 1649.91 | 1415.91 | 1271.69 | 1201.17 | 1185.50 | 1185.50 | 9, 10 |
| 221 | 3706.00 | 2414.50 | 3281.00 | 3550.00 | 2462.38 | 1437.91 | 1418.91 | 1275.81 | 1200.69 | 1188.65 | 1188.65 | 9, 10 |
| 222 | 3064.00 | 2856.50 | 6342.00 | 3574.00 | 2481.94 | 1658.91 | 1639.91 | 1283.02 | 1212.61 | 1195.90 | 1195.90 | 9, 10 |
| 223 | 3736.00 | 1275.97 | 1282.00 | 3602.00 | 2494.00 | 1269.97 | 1269.97 | 1274.22 | 1216.33 | 1188.90 | 1188.90 | 9, 10 |
| 224 | 2650.00 | 1721.97 | 4022.00 | 3414.00 | 2344.09 | 1697.91 | 1710.16 | 1292.67 | 1224.69 | 1206.73 | 1206.73 | 9, 10 |
| 225 | 3802.00 | 2445.50 | 2459.00 | 3502.00 | 2391.03 | 1481.91 | 1334.16 | 1297.73 | 1227.93 | 1209.23 | 1209.23 | 9, 10 |
| 226 | 3146.00 | 2895.50 | 4694.00 | 3554.00 | 2425.66 | 1706.91 | 1559.16 | 1305.81 | 1236.50 | 1219.08 | 1219.08 | 9, 10 |
| 227 | 3832.00 | 1746.50 | 1794.00 | 3596.00 | 2460.03 | 1383.41 | 1426.66 | 1308.31 | 1239.46 | 1222.67 | 1222.67 | 9, 10 |
| 228 | 2890.00 | 2200.50 | 4920.00 | 3624.00 | 2479.34 | 1721.91 | 1378.16 | 1318.48 | 1247.89 | 1231.10 | 1231.10 | 9, 10 |
| 229 | 3874.00 | 2519.50 | 2916.00 | 3664.00 | 2516.91 | 1501.91 | 1381.16 | 1322.92 | 1250.24 | 1234.71 | 1234.71 | 9, 10 |
| 230 | 3204.00 | 2977.50 | 5604.00 | 3694.00 | 2543.03 | 1730.91 | 1610.16 | 1330.59 | 1259.65 | 1241.98 | 1241.98 | 9, 10 |
| 231 | 3904.00 | 1477.00 | 1530.75 | 3728.00 | 2568.53 | 1352.16 | 1296.16 | 1330.14 | 1264.28 | 1236.46 | 1236.46 | 9, 10 |
| 232 | 2828.00 | 1939.00 | 5152.00 | 3728.00 | 2564.59 | 1752.91 | 1420.91 | 1343.02 | 1271.20 | 1252.44 | 1252.44 | 9, 10 |
| 233 | 3956.00 | 2584.50 | 3040.00 | 3774.00 | 2605.16 | 1528.91 | 1306.91 | 1347.70 | 1266.89 | 1254.66 | 1254.66 | 9, 10 |
| 234 | 3272.00 | 3050.50 | 5848.00 | 3806.00 | 2633.78 | 1761.91 | 1539.91 | 1355.58 | 1281.61 | 1264.33 | 1264.33 | 9, 10 |
| 235 | 3965.00 | 1830.75 | 2104.50 | 3840.00 | 2663.16 | 1426.41 | 1370.41 | 1357.52 | 1287.94 | 1267.60 | 1267.60 | 9, 10 |
| 236 | 3004.00 | 2300.75 | 6082.00 | 3858.00 | 2677.34 | 1776.91 | 1489.91 | 1367.75 | 1293.17 | 1275.63 | 1275.63 | 9, 10 |
| 237 | 4028.00 | 2648.50 | 3513.00 | 3896.00 | 2711.28 | 1548.91 | 1482.91 | 1371.95 | 1296.33 | 1278.92 | 1278.92 | 9, 10 |
| 238 | 3330.00 | 3122.50 | 6790.00 | 3922.00 | 2733.91 | 1785.91 | 1729.91 | 1379.38 | 1304.56 | 1286.18 | 1286.18 | 9, 10 |
| 239 | 4058.00 | 1413.13 | 1458.25 | 3952.00 | 2753.03 | 1372.03 | 1372.03 | 1375.03 | 1305.76 | 1285.26 | 1285.26 | 9, 10 |
| 240 | 2892.00 | 1891.13 | 5400.00 | 3996.00 | 2704.22 | 1815.91 | 1614.41 | 1390.39 | 1316.78 | 1297.53 | 1297.53 | 9, 10 |
| 241 | 4118.00 | 2703.50 | 3180.00 | 3958.00 | 2749.84 | 1583.91 | 1367.41 | 1395.34 | 1318.83 | 1299.57 | 1299.57 | 9, 10 |
| 242 | 3406.00 | 3185.50 | 6120.00 | 3998.00 | 2763.09 | 1824.91 | 1608.41 | 1403.31 | 1328.96 | 1308.96 | 1307.78 | 9, 10 |
| 243 | 4148.00 | 1909.75 | 2194.50 | 4036.00 | 2816.59 | 1477.41 | 1368.41 | 1405.64 | 1333.81 | 1311.97 | 1311.97 | 9, 10 |
| 244 | 3126.00 | 2395.75 | 6362.00 | 4058.00 | 2834.72 | 1839.91 | 1491.91 | 1415.88 | 1340.80 | 1319.74 | 1319.74 | 9, 10 |
| 245 | 4190.00 | 2772.50 | 3669.00 | 4098.00 | 2872.09 | 1603.91 | 1494.91 | 1420.09 | 1341.94 | 1322.71 | 1322.71 | 9, 10 |
| 246 | 3464.00 | 3262.50 | 7094.00 | 4126.00 | 2897.84 | 1848.91 | 1739.91 | 1427.67 | 1351.91 | 1329.58 | 1329.58 | 9, 10 |
| 247 | 4220.00 | 1598.63 | 1775.00 | 4158.00 | 2923.09 | 1442.16 | 1442.16 | 1426.91 | 1355.59 | 1330.73 | 1324.32 | 10 |
| 248 | 3054.00 | 2092.63 | 6610.00 | 4154.00 | 2918.47 | 1870.91 | 1574.91 | 1439.20 | 1363.78 | 1340.86 | 1340.20 | 10 |
| 249 | 4272.00 | 2834.50 | 3801.00 | 4202.00 | 2960.22 | 1630.91 | 1452.91 | 1443.88 | 1367.11 | 1343.04 | 1342.45 | 10 |
| 250 | 3532.00 | 3332.50 | 7354.00 | 4234.00 | 2989.72 | 1879.91 | 1701.91 | 1451.66 | 1375.56 | 1352.27 | 1351.90 | 10 |

A.1.2 Median Filtering

This section considers the problem of selecting the median of a window. Hence

$$l = u = \lfloor \frac{W}{2} \rfloor.$$

| W | 0) | 1) | 2) | 3) | 4) | 5) | 6) | 7) | 8) | 9) | 10) | Best strategies |
|----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------------------------------|
| 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 2 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 3 | 4.00 | 3.00 | 3.00 | 4.00 | 3.00 | 3.00 | 3.00 | 3.00 | 3.00 | 3.00 | 3.00 | 1, 2, 4, 5, 6, 7, 8, 9, 10 |
| 4 | 5.00 | 6.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 4.50 | 4.50 | 4.50 | 4.50 | 7, 8, 9, 10 |
| 5 | 10.00 | 5.00 | 5.00 | 8.00 | 7.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 6 | 11.00 | 9.50 | 11.00 | 11.00 | 8.50 | 8.50 | 8.50 | 7.00 | 7.00 | 7.00 | 7.00 | 7, 8, 9, 10 |
| 7 | 18.00 | 8.50 | 8.50 | 18.00 | 12.50 | 8.50 | 8.50 | 8.50 | 8.50 | 8.50 | 8.50 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 8 | 15.00 | 13.00 | 15.00 | 15.00 | 15.00 | 15.00 | 15.00 | 10.75 | 10.33 | 10.33 | 8, 9, 10 | |
| 9 | 30.00 | 11.00 | 10.00 | 18.00 | 19.00 | 10.00 | 10.00 | 10.00 | 10.00 | 10.00 | 10.00 | 2, 5, 6, 7, 8, 9, 10 |
| 10 | 25.00 | 16.25 | 23.00 | 23.00 | 19.75 | 14.50 | 14.50 | 12.50 | 12.50 | 12.50 | 12.50 | 7, 8, 9, 10 |
| 11 | 38.00 | 12.00 | 12.00 | 32.00 | 22.25 | 12.00 | 12.00 | 12.00 | 12.00 | 12.00 | 12.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 12 | 29.00 | 17.25 | 31.00 | 31.00 | 19.75 | 19.75 | 19.75 | 14.25 | 14.25 | 14.25 | 14.25 | 7, 8, 9, 10 |
| 13 | 52.00 | 15.00 | 20.00 | 42.00 | 27.50 | 16.00 | 16.00 | 14.25 | 14.25 | 14.25 | 14.25 | 7, 8, 9, 10 |
| 14 | 41.00 | 21.25 | 45.00 | 49.00 | 33.25 | 21.25 | 21.25 | 17.25 | 16.50 | 16.50 | 16.50 | 8, 9, 10 |
| 15 | 60.00 | 16.25 | 16.25 | 60.00 | 38.75 | 16.25 | 16.25 | 16.25 | 16.25 | 16.25 | 16.25 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 16 | 41.00 | 21.50 | 41.00 | 41.00 | 41.00 | 28.00 | 41.00 | 19.38 | 19.17 | 19.17 | 19.17 | 8, 9, 10 |
| 17 | 80.00 | 24.00 | 23.00 | 44.00 | 49.00 | 23.50 | 23.00 | 19.50 | 17.33 | 17.33 | 17.33 | 8, 9, 10 |
| 18 | 63.00 | 29.75 | 51.00 | 51.00 | 51.00 | 29.50 | 28.50 | 21.25 | 19.50 | 19.50 | 19.50 | 8, 9, 10 |
| 19 | | | | | | | | | | | | |

APPENDIX A. ALGORITHM RESULTS

| | | | | | | | | | | | | |
|-----|---------|--------|--------|--------|--------|-------|--------|-------|-------|-------|-------|-------------------------|
| 27 | 148.00 | 26.25 | 32.00 | 128.00 | 75.50 | 26.25 | 26.25 | 25.25 | 24.33 | 24.33 | 24.33 | 8, 9, 10 |
| 28 | 97.00 | 31.88 | 115.00 | 131.00 | 76.38 | 34.88 | 34.88 | 27.63 | 27.63 | 27.63 | 27.63 | 7, 8, 9, 10 |
| 29 | 164.00 | 43.00 | 65.00 | 148.00 | 90.00 | 32.50 | 32.50 | 27.88 | 25.67 | 25.67 | 25.67 | 8, 9, 10 |
| 30 | 123.00 | 50.75 | 141.00 | 159.00 | 99.63 | 39.13 | 39.13 | 30.38 | 28.42 | 28.42 | 28.42 | 8, 9, 10 |
| 31 | 174.00 | 26.13 | 26.13 | 174.00 | 105.88 | 26.13 | 26.13 | 26.13 | 26.13 | 26.13 | 26.13 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 32 | 107.00 | 31.75 | 107.00 | 107.00 | 107.00 | 43.50 | 107.00 | 29.75 | 28.92 | 28.92 | 28.92 | 8, 9, 10 |
| 33 | 204.00 | 56.00 | 56.00 | 110.00 | 120.75 | 40.25 | 56.00 | 30.50 | 29.75 | 29.75 | 29.75 | 8, 9, 10 |
| 34 | 155.00 | 62.25 | 119.00 | 119.00 | 126.06 | 47.25 | 62.50 | 32.31 | 31.17 | 31.17 | 31.17 | 8, 9, 10 |
| 35 | 216.00 | 37.00 | 37.00 | 132.00 | 131.56 | 37.00 | 37.00 | 30.63 | 27.00 | 27.00 | 27.00 | 8, 9, 10 |
| 36 | 141.00 | 42.88 | 131.00 | 131.00 | 127.38 | 45.25 | 44.75 | 33.44 | 30.58 | 30.58 | 30.58 | 8, 9, 10 |
| 37 | 236.00 | 64.00 | 70.00 | 142.00 | 134.50 | 42.25 | 41.00 | 33.06 | 30.17 | 30.17 | 30.17 | 8, 9, 10 |
| 38 | 177.00 | 71.25 | 151.00 | 155.00 | 136.44 | 49.50 | 47.75 | 34.69 | 31.50 | 31.50 | 31.50 | 8, 9, 10 |
| 39 | 248.00 | 31.00 | 30.25 | 172.00 | 139.31 | 30.25 | 30.25 | 30.25 | 30.25 | 30.25 | 30.25 | 2, 5, 6, 7, 8, 9, 10 |
| 40 | 151.00 | 36.81 | 155.00 | 155.00 | 118.50 | 42.63 | 56.13 | 33.88 | 33.88 | 33.88 | 33.88 | 7, 8, 9, 10 |
| 41 | 274.00 | 69.00 | 82.00 | 174.00 | 128.00 | 39.75 | 39.25 | 34.44 | 31.08 | 31.08 | 31.08 | 8, 9, 10 |
| 42 | 205.00 | 77.25 | 175.00 | 191.00 | 132.06 | 46.63 | 45.13 | 35.88 | 33.92 | 33.92 | 33.92 | 8, 9, 10 |
| 43 | 286.00 | 44.25 | 52.50 | 210.00 | 138.44 | 36.50 | 36.50 | 34.25 | 34.25 | 34.25 | 34.25 | 7, 8, 9, 10 |
| 44 | 183.00 | 50.13 | 197.00 | 213.00 | 135.63 | 45.13 | 45.13 | 36.69 | 35.00 | 35.00 | 35.00 | 8, 9, 10 |
| 45 | 306.00 | 79.00 | 106.00 | 230.00 | 146.75 | 42.75 | 42.75 | 36.31 | 36.31 | 36.31 | 36.31 | 7, 8, 9, 10 |
| 46 | 227.00 | 87.25 | 229.00 | 247.00 | 152.94 | 50.13 | 50.13 | 38.06 | 38.06 | 38.06 | 38.06 | 7, 8, 9, 10 |
| 47 | 318.00 | 32.00 | 32.00 | 268.00 | 156.81 | 32.00 | 32.00 | 32.00 | 31.75 | 31.75 | 31.75 | 8, 9, 10 |
| 48 | 189.00 | 37.81 | 203.00 | 203.00 | 113.94 | 49.69 | 113.94 | 35.81 | 35.81 | 35.81 | 35.81 | 7, 8, 9, 10 |
| 49 | 350.00 | 78.00 | 106.00 | 238.00 | 131.63 | 47.25 | 64.00 | 37.19 | 35.75 | 35.75 | 35.75 | 8, 9, 10 |
| 50 | 261.00 | 87.25 | 221.00 | 261.00 | 143.19 | 54.06 | 69.31 | 38.63 | 36.96 | 36.96 | 36.96 | 8, 9, 10 |
| 51 | 362.00 | 50.25 | 63.50 | 282.00 | 155.00 | 44.13 | 43.63 | 37.56 | 35.75 | 35.75 | 35.75 | 8, 9, 10 |
| 52 | 231.00 | 56.13 | 241.00 | 289.00 | 157.19 | 53.19 | 52.19 | 39.56 | 39.56 | 39.56 | 39.56 | 7, 8, 9, 10 |
| 53 | 382.00 | 89.00 | 128.00 | 308.00 | 172.63 | 51.50 | 49.75 | 38.88 | 36.44 | 36.44 | 36.44 | 8, 9, 10 |
| 54 | 283.00 | 100.25 | 271.00 | 325.00 | 182.69 | 59.06 | 56.56 | 41.38 | 39.79 | 39.79 | 39.79 | 8, 9, 10 |
| 55 | 394.00 | 38.63 | 46.00 | 346.00 | 192.25 | 38.63 | 38.63 | 37.38 | 36.17 | 36.17 | 36.17 | 8, 9, 10 |
| 56 | 235.00 | 44.44 | 285.00 | 333.00 | 177.69 | 51.44 | 64.94 | 41.19 | 40.21 | 40.21 | 40.21 | 8, 9, 10 |
| 57 | 420.00 | 100.00 | 150.00 | 360.00 | 197.50 | 49.25 | 48.75 | 40.94 | 40.94 | 40.94 | 40.94 | 7, 8, 9, 10 |
| 58 | 311.00 | 111.25 | 313.00 | 379.00 | 211.19 | 56.19 | 54.69 | 42.88 | 42.29 | 42.29 | 42.29 | 8, 9, 10 |
| 59 | 432.00 | 62.50 | 88.00 | 400.00 | 225.88 | 46.38 | 46.38 | 42.13 | 37.33 | 37.33 | 37.33 | 8, 9, 10 |
| 60 | 273.00 | 68.38 | 345.00 | 407.00 | 230.69 | 55.69 | 55.69 | 44.38 | 41.88 | 41.88 | 41.88 | 8, 9, 10 |
| 61 | 452.00 | 117.00 | 184.00 | 430.00 | 250.25 | 54.25 | 54.25 | 44.25 | 41.33 | 41.33 | 41.33 | 8, 9, 10 |
| 62 | 333.00 | 126.25 | 387.00 | 445.00 | 263.81 | 61.81 | 61.81 | 46.25 | 42.22 | 42.22 | 42.22 | 8, 9, 10 |
| 63 | 464.00 | 38.06 | 38.06 | 464.00 | 269.94 | 38.06 | 38.06 | 38.06 | 38.06 | 38.06 | 38.06 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 64 | 269.00 | 43.88 | 269.00 | 269.00 | 269.00 | 61.75 | 269.00 | 41.88 | 41.88 | 41.88 | 41.88 | 7, 8, 9, 10 |
| 65 | 506.00 | 132.00 | 137.00 | 272.00 | 290.00 | 59.13 | 137.00 | 44.25 | 41.13 | 41.13 | 41.13 | 8, 9, 10 |
| 66 | 377.00 | 138.75 | 283.00 | 283.00 | 300.44 | 66.38 | 144.50 | 45.91 | 43.79 | 43.79 | 43.79 | 8, 9, 10 |
| 67 | 520.00 | 77.00 | 78.50 | 298.00 | 310.59 | 56.75 | 78.50 | 44.00 | 43.63 | 43.63 | 43.63 | 8, 9, 10 |
| 68 | 331.00 | 83.13 | 297.00 | 297.00 | 309.25 | 65.63 | 86.25 | 46.91 | 43.92 | 43.92 | 43.92 | 8, 9, 10 |
| 69 | 544.00 | 146.00 | 153.00 | 308.00 | 320.31 | 63.63 | 62.50 | 46.44 | 44.92 | 44.92 | 44.92 | 8, 9, 10 |
| 70 | 403.00 | 153.75 | 319.00 | 323.00 | 324.06 | 71.50 | 89.75 | 48.03 | 46.33 | 46.33 | 46.33 | 8, 9, 10 |
| 71 | 558.00 | 52.00 | 51.50 | 342.00 | 328.78 | 51.25 | 51.50 | 43.81 | 38.33 | 38.33 | 38.33 | 8, 9, 10 |
| 72 | 331.00 | 57.94 | 325.00 | 325.00 | 308.00 | 63.68 | 77.38 | 47.72 | 44.04 | 44.04 | 44.04 | 8, 9, 10 |
| 73 | 590.00 | 154.00 | 167.00 | 344.00 | 319.88 | 61.38 | 60.50 | 47.58 | 44.00 | 44.00 | 44.00 | 8, 9, 10 |
| 74 | 437.00 | 162.75 | 347.00 | 363.00 | 324.19 | 68.75 | 66.63 | 49.47 | 44.88 | 44.88 | 44.88 | 8, 9, 10 |
| 75 | 604.00 | 87.25 | 96.00 | 384.00 | 330.84 | 59.25 | 58.25 | 47.56 | 44.04 | 44.04 | 44.04 | 8, 9, 10 |
| 76 | 381.00 | 93.38 | 371.00 | 387.00 | 336.44 | 68.25 | 66.88 | 50.09 | 47.38 | 47.38 | 47.38 | 8, 9, 10 |
| 77 | 628.00 | 166.00 | 193.00 | 404.00 | 337.44 | 66.38 | 64.50 | 49.47 | 44.29 | 44.29 | 44.29 | 8, 9, 10 |
| 78 | 463.00 | 174.75 | 405.00 | 423.00 | 341.56 | 76.38 | 72.13 | 50.91 | 47.50 | 47.50 | 47.50 | 8, 9, 10 |
| 79 | 642.00 | 44.00 | 43.13 | 446.00 | 343.66 | 43.13 | 43.13 | 43.13 | 43.13 | 43.13 | 43.13 | 2, 5, 6, 7, 8, 9, 10 |
| 80 | 369.00 | 49.91 | 381.00 | 381.00 | 382.13 | 60.81 | 125.06 | 46.94 | 46.94 | 46.94 | 46.94 | 7, 8, 9, 10 |
| 81 | 680.00 | 166.00 | 195.00 | 416.00 | 297.00 | 58.38 | 75.13 | 50.03 | 48.88 | 48.88 | 48.88 | 8, 9, 10 |
| 82 | 503.00 | 175.75 | 403.00 | 443.00 | 304.19 | 65.44 | 80.69 | 51.28 | 50.04 | 50.04 | 50.04 | 8, 9, 10 |
| 83 | 694.00 | 94.25 | 110.00 | 468.00 | 313.03 | 55.75 | 55.25 | 49.47 | 44.42 | 44.42 | 44.42 | 8, 9, 10 |
| 84 | 437.00 | 100.38 | 427.00 | 475.00 | 311.00 | 64.81 | 63.81 | 51.88 | 48.58 | 48.58 | 48.58 | 8, 9, 10 |
| 85 | 718.00 | 177.00 | 221.00 | 494.00 | 323.56 | 63.13 | 61.38 | 50.97 | 48.08 | 48.08 | 48.08 | 8, 9, 10 |
| 86 | 529.00 | 188.75 | 461.00 | 515.00 | 329.31 | 70.94 | 68.44 | 52.81 | 49.63 | 49.63 | 49.63 | 8, 9, 10 |
| 87 | 732.00 | 80.63 | 70.25 | 540.00 | 335.84 | 50.75 | 50.75 | 48.25 | 48.25 | 48.25 | 48.25 | 7, 8, 9, 10 |
| 88 | 429.00 | 66.56 | 479.00 | 527.00 | 317.13 | 63.56 | 77.06 | 52.16 | 52.16 | 52.16 | 52.16 | 7, 8, 9, 10 |
| 89 | 764.00 | 188.00 | 247.00 | 554.00 | 334.00 | 61.38 | 60.88 | 51.91 | 48.44 | 48.44 | 48.44 | 8, 9, 10 |
| 90 | 563.00 | 199.75 | 513.00 | 579.00 | 343.56 | 68.69 | 67.19 | 53.44 | 51.63 | 51.63 | 51.63 | 8, 9, 10 |
| 91 | 778.00 | 106.50 | 139.50 | 606.00 | 355.28 | 59.25 | 59.25 | 51.75 | 51.75 | 51.75 | 51.75 | 7, 8, 9, 10 |
| 92 | 487.00 | 112.63 | 551.00 | 613.00 | 356.00 | 68.56 | 68.56 | 54.13 | 52.17 | 52.17 | 52.17 | 8, 9, 10 |
| 93 | 802.00 | 205.00 | 287.00 | 636.00 | 372.56 | 67.13 | 67.13 | 53.56 | 53.42 | 53.42 | 53.42 | 8, 9, 10 |
| 94 | 589.00 | 214.75 | 601.00 | 659.00 | 382.19 | 75.19 | 75.19 | 55.06 | 54.83 | 54.83 | 54.83 | 8, 9, 10 |
| 95 | 816.00 | 45.00 | 45.00 | 686.00 | 385.41 | 45.00 | 45.00 | 44.04 | 44.04 | 44.04 | 44.04 | 8, 9, 10 |
| 96 | 463.00 | 50.91 | 493.00 | 493.00 | 276.97 | 68.84 | 276.97 | 48.91 | 48.91 | 48.91 | 48.91 | 7, 8, 9, 10 |
| 97 | 860.00 | 185.00 | 251.00 | 560.00 | 301.94 | 66.63 | 146.00 | 53.22 | 50.54 | 50.54 | 50.54 | 8, 9, 10 |
| 98 | 635.00 | 195.75 | 513.00 | 601.00 | 318.78 | 73.59 | 151.22 | 54.50 | 51.48 | 51.48 | 51.48 | 8, 9, 10 |
| 99 | 874.00 | 106.25 | 137.00 | 632.00 | 335.38 | 63.94 | 84.94 | 53.00 | 50.17 | 50.17 | 50.17 | 8, 9, 10 |
| 100 | 549.00 | 112.38 | 535.00 | 647.00 | 340.72 | 73.22 | 93.47 | 55.06 | 53.90 | 53.90 | 53.90 | 8, 9, 10 |
| 101 | 898.00 | 200.00 | 275.00 | 670.00 | 360.31 | 71.88 | 91.00 | 54.25 | 50.83 | 50.83 | 50.83 | 8, 9, 10 |
| 102 | 661.00 | 214.75 | 567.00 | 693.00 | 372.66 | 79.72 | 97.72 | 56.28 | 53.77 | 53.77 | 53.77 | 8, 9, 10 |
| 103 | 912.00 | 67.63 | 83.25 | 720.00 | 384.38 | 59.69 | 59.44 | 52.16 | 49.88 | 49.88 | 49.88 | 8, 9, 10 |
| 104 | 533.00 | 73.56 | 583.00 | 711.00 | 370.47 | 72.72 | 85.72 | 56.06 | 54.02 | 54.02 | 54.02 | 8, 9, 10 |
| 105 | 944.00 | 212.00 | 299.00 | 740.00 | 393.06 | 70.88 | 69.50 | 55.44 | 55.04 | 55.04 | 55.04 | 8, 9, 10 |
| 106 | 695.00 | 228.75 | 615.00 | 765.00 | 407.91 | 78.22 | 75.47 | 56.84 | 56.23 | 56.23 | 56.23 | 8, 9, 10 |
| 107 | 958.00 | 121.50 | 164.50 | 792.00 | 423.50 | 68.94 | 67.19 | 55.66 | 49.67 | 49.67 | 49.67 | 8, 9, 10 |
| 108 | 599.00 | 127.63 | 651.00 | 801.00 | 427.72 | 78.59 | 76.47 | 58.06 | 55.56 | 55.56 | 55.56 | 8, 9, 10 |
| 109 | 982.00 | 233.00 | 337.00 | 826.00 | 447.69 | 77.63 | 75.00 | 57.63 | 55.13 | 55.13 | 55.13 | 8, 9, 10 |
| 110 | 721.00 | 245.75 | 699.00 | 849.00 | 460.53 | 85.84 | 82.72 | 59.47 | 55.90 | 55.90 | 55.90 | 8, 9, 10 |
| 111 | 996.00 | 53.06 | 62.00 | 876.00 | 469.88 | 53.06 | 53.06 | 51.56 | 50.08 | 50.08 | 50.08 | 8, 9, 10 |
| 112 | 565.00 | 58.97 | 687.00 | 815.00 | 415.09 | 70.97 | 135.22 | 55.47 | 54.10 | 54.10 | 54.10 | 8, 9, 10 |
| 113 | 1034.00 | 240.00 | 351.00 | 860.00 | 442.75 | 68.88 | 85.63 | 57.78 | 54.94 | 54.94 | 54.94 | 8, 9, 10 |
| 114 | 761.00 | 252.75 | | | | | | | | | | |

APPENDIX A. ALGORITHM RESULTS

| | | | | | | | | | | | | |
|-----|---------|--------|---------|---------|--------|--------|--------|-------|-------|-------|-------|-------------------------|
| 126 | 847.00 | 309.75 | 983.00 | 1147.00 | 653.91 | 89.03 | 89.03 | 64.91 | 60.40 | 60.40 | 60.40 | 8, 9, 10 |
| 127 | 1170.00 | 52.03 | 52.03 | 1170.00 | 658.97 | 52.03 | 52.03 | 52.03 | 52.03 | 52.03 | 52.03 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 128 | 655.00 | 67.94 | 655.00 | 655.00 | 655.00 | 82.88 | 655.00 | 55.94 | 55.94 | 55.94 | 55.94 | 7, 8, 9, 10 |
| 129 | 1226.00 | 308.00 | 330.00 | 658.00 | 684.56 | 80.56 | 330.00 | 60.94 | 60.94 | 60.94 | 60.94 | 7, 8, 9, 10 |
| 130 | 905.00 | 315.25 | 671.00 | 671.00 | 701.67 | 87.81 | 338.50 | 62.42 | 62.17 | 62.17 | 62.17 | 8, 9, 10 |
| 131 | 1242.00 | 169.00 | 176.00 | 688.00 | 718.20 | 78.38 | 176.00 | 60.34 | 55.92 | 55.92 | 55.92 | 8, 9, 10 |
| 132 | 779.00 | 175.38 | 687.00 | 687.00 | 721.53 | 87.56 | 183.75 | 63.20 | 60.71 | 60.71 | 60.71 | 8, 9, 10 |
| 133 | 1270.00 | 334.00 | 348.00 | 698.00 | 738.63 | 86.06 | 180.00 | 62.59 | 59.92 | 59.92 | 59.92 | 8, 9, 10 |
| 134 | 935.00 | 342.25 | 711.00 | 715.00 | 746.39 | 94.13 | 187.75 | 64.02 | 60.81 | 60.81 | 60.81 | 8, 9, 10 |
| 135 | 1286.00 | 100.00 | 100.75 | 736.00 | 755.20 | 74.25 | 100.75 | 59.63 | 59.58 | 59.58 | 59.58 | 8, 9, 10 |
| 136 | 749.00 | 106.06 | 719.00 | 719.00 | 736.69 | 87.19 | 126.63 | 63.58 | 62.96 | 62.96 | 62.96 | 8, 9, 10 |
| 137 | 1324.00 | 348.00 | 364.00 | 738.00 | 753.75 | 85.19 | 109.75 | 63.81 | 59.27 | 59.27 | 59.27 | 8, 9, 10 |
| 138 | 975.00 | 357.25 | 743.00 | 759.00 | 761.17 | 92.75 | 116.13 | 65.30 | 62.10 | 62.10 | 62.10 | 8, 9, 10 |
| 139 | 1340.00 | 185.25 | 195.50 | 782.00 | 770.95 | 83.63 | 108.00 | 63.13 | 61.98 | 61.98 | 61.85 | 10 |
| 140 | 837.00 | 191.63 | 769.00 | 785.00 | 767.84 | 93.13 | 116.63 | 65.83 | 62.13 | 62.13 | 62.13 | 8, 9, 10 |
| 141 | 1368.00 | 364.00 | 392.00 | 802.00 | 781.69 | 91.94 | 114.25 | 65.19 | 63.17 | 63.17 | 63.17 | 8, 9, 10 |
| 142 | 1005.00 | 373.25 | 805.00 | 823.00 | 786.77 | 100.31 | 122.13 | 66.58 | 64.29 | 64.29 | 64.29 | 8, 9, 10 |
| 143 | 1384.00 | 69.00 | 68.25 | 848.00 | 789.95 | 67.63 | 68.25 | 59.03 | 51.67 | 51.67 | 51.67 | 8, 9, 10 |
| 144 | 781.00 | 74.97 | 783.00 | 783.00 | 727.75 | 85.44 | 150.19 | 62.98 | 57.52 | 57.52 | 57.52 | 8, 9, 10 |
| 145 | 1430.00 | 367.00 | 396.00 | 818.00 | 745.94 | 83.19 | 100.25 | 65.66 | 61.08 | 61.08 | 61.08 | 8, 9, 10 |
| 146 | 1053.00 | 377.25 | 807.00 | 847.00 | 754.42 | 90.50 | 105.94 | 67.02 | 61.65 | 61.65 | 61.65 | 8, 9, 10 |
| 147 | 1446.00 | 195.25 | 211.50 | 874.00 | 764.64 | 81.13 | 80.63 | 65.00 | 60.67 | 60.67 | 60.67 | 8, 9, 10 |
| 148 | 903.00 | 201.63 | 833.00 | 881.00 | 762.16 | 90.38 | 89.19 | 67.83 | 63.63 | 63.63 | 63.63 | 8, 9, 10 |
| 149 | 1474.00 | 380.00 | 424.00 | 900.00 | 775.75 | 88.94 | 86.75 | 66.97 | 60.35 | 60.35 | 60.35 | 8, 9, 10 |
| 150 | 1083.00 | 392.25 | 869.00 | 923.00 | 780.64 | 97.06 | 93.94 | 68.61 | 63.56 | 63.56 | 63.56 | 8, 9, 10 |
| 151 | 1490.00 | 111.63 | 121.50 | 950.00 | 786.52 | 77.25 | 76.38 | 63.91 | 59.69 | 59.69 | 59.69 | 8, 9, 10 |
| 152 | 863.00 | 117.69 | 889.00 | 937.00 | 765.44 | 90.25 | 102.69 | 67.86 | 63.44 | 63.44 | 63.44 | 8, 9, 10 |
| 153 | 1528.00 | 393.00 | 452.00 | 964.00 | 782.63 | 88.31 | 86.50 | 67.73 | 64.38 | 64.38 | 64.38 | 8, 9, 10 |
| 154 | 1123.00 | 405.25 | 925.00 | 991.00 | 790.67 | 95.94 | 92.94 | 69.09 | 65.48 | 65.48 | 65.48 | 8, 9, 10 |
| 155 | 1544.00 | 209.50 | 243.00 | 1020.00 | 801.14 | 86.88 | 85.13 | 66.97 | 59.69 | 59.69 | 59.69 | 8, 9, 10 |
| 156 | 961.00 | 215.88 | 965.00 | 1027.00 | 798.97 | 96.44 | 94.44 | 69.44 | 65.10 | 65.10 | 65.10 | 8, 9, 10 |
| 157 | 1572.00 | 412.00 | 494.00 | 1050.00 | 814.06 | 95.31 | 93.00 | 68.69 | 64.31 | 64.31 | 64.31 | 8, 9, 10 |
| 158 | 1153.00 | 422.25 | 1017.00 | 1075.00 | 820.52 | 103.75 | 101.19 | 69.97 | 65.06 | 65.06 | 65.06 | 8, 9, 10 |
| 159 | 1588.00 | 59.00 | 58.06 | 1104.00 | 820.95 | 58.06 | 58.06 | 58.06 | 58.06 | 58.06 | 58.06 | 2, 5, 6, 7, 8, 9, 10 |
| 160 | 883.00 | 64.95 | 911.00 | 911.00 | 861.81 | 81.91 | 290.03 | 61.97 | 61.97 | 61.97 | 61.97 | 7, 8, 9, 10 |
| 161 | 1640.00 | 393.00 | 460.00 | 978.00 | 853.50 | 79.69 | 159.06 | 68.58 | 62.33 | 62.33 | 62.33 | 8, 9, 10 |
| 162 | 1207.00 | 404.25 | 935.00 | 1023.00 | 895.48 | 86.78 | 164.41 | 69.73 | 66.98 | 66.98 | 66.98 | 8, 9, 10 |
| 163 | 1655.00 | 210.25 | 243.50 | 1058.00 | 708.67 | 77.25 | 98.25 | 67.72 | 66.75 | 66.75 | 66.75 | 8, 9, 10 |
| 164 | 1033.00 | 216.63 | 961.00 | 1073.00 | 709.28 | 86.53 | 106.78 | 70.08 | 66.69 | 66.69 | 66.69 | 8, 9, 10 |
| 165 | 1684.00 | 408.00 | 488.00 | 1096.00 | 725.56 | 85.19 | 104.31 | 69.14 | 67.71 | 67.71 | 67.71 | 8, 9, 10 |
| 166 | 1237.00 | 423.25 | 997.00 | 1123.00 | 733.08 | 93.16 | 111.16 | 70.70 | 69.10 | 69.10 | 69.10 | 8, 9, 10 |
| 167 | 1700.00 | 119.63 | 137.50 | 1154.00 | 741.36 | 73.25 | 73.00 | 66.17 | 58.96 | 58.96 | 58.96 | 8, 9, 10 |
| 168 | 983.00 | 125.69 | 1017.00 | 1145.00 | 722.75 | 86.28 | 99.28 | 70.13 | 64.81 | 64.81 | 64.81 | 8, 9, 10 |
| 169 | 1738.00 | 420.00 | 516.00 | 1174.00 | 742.00 | 84.44 | 63.06 | 69.88 | 65.58 | 65.58 | 65.58 | 8, 9, 10 |
| 170 | 1277.00 | 437.25 | 1053.00 | 1203.00 | 752.05 | 91.91 | 89.16 | 71.17 | 66.33 | 66.33 | 66.33 | 8, 9, 10 |
| 171 | 1754.00 | 225.50 | 275.00 | 1234.00 | 764.17 | 82.75 | 61.00 | 69.17 | 66.02 | 66.02 | 66.02 | 8, 9, 10 |
| 172 | 1091.00 | 231.88 | 1093.00 | 1243.00 | 763.72 | 92.41 | 90.28 | 71.80 | 68.88 | 68.88 | 68.88 | 8, 9, 10 |
| 173 | 1782.00 | 441.00 | 558.00 | 1268.00 | 780.31 | 91.44 | 88.81 | 71.05 | 65.90 | 65.90 | 65.90 | 8, 9, 10 |
| 174 | 1307.00 | 454.25 | 1145.00 | 1295.00 | 789.39 | 99.78 | 96.66 | 72.55 | 69.35 | 69.35 | 69.35 | 8, 9, 10 |
| 175 | 1798.00 | 79.06 | 90.13 | 1326.00 | 794.23 | 67.13 | 67.13 | 64.31 | 64.31 | 64.31 | 64.31 | 7, 8, 9, 10 |
| 176 | 1007.00 | 85.03 | 1137.00 | 1265.00 | 734.81 | 85.03 | 149.28 | 68.27 | 68.27 | 68.27 | 68.27 | 7, 8, 9, 10 |
| 177 | 1844.00 | 448.00 | 576.00 | 1310.00 | 759.06 | 82.94 | 99.69 | 70.86 | 70.31 | 70.31 | 70.31 | 8, 9, 10 |
| 178 | 1355.00 | 461.25 | 1173.00 | 1347.00 | 773.80 | 90.16 | 105.41 | 72.05 | 71.21 | 71.21 | 71.21 | 8, 9, 10 |
| 179 | 1860.00 | 239.50 | 305.00 | 1382.00 | 789.98 | 80.75 | 80.25 | 70.23 | 63.56 | 63.56 | 63.56 | 8, 9, 10 |
| 180 | 1157.00 | 244.88 | 1213.00 | 1395.00 | 793.53 | 90.16 | 89.16 | 72.66 | 69.92 | 69.92 | 69.92 | 8, 9, 10 |
| 181 | 1888.00 | 464.00 | 618.00 | 1422.00 | 813.50 | 88.94 | 87.19 | 71.73 | 69.29 | 69.29 | 69.29 | 8, 9, 10 |
| 182 | 1385.00 | 481.25 | 1265.00 | 1451.00 | 824.77 | 97.03 | 94.53 | 73.50 | 70.38 | 70.38 | 70.38 | 8, 9, 10 |
| 183 | 1904.00 | 134.25 | 172.25 | 1484.00 | 836.67 | 77.25 | 77.25 | 68.88 | 68.88 | 68.88 | 68.88 | 7, 8, 9, 10 |
| 184 | 1097.00 | 140.31 | 1301.00 | 1475.00 | 821.50 | 90.41 | 103.91 | 72.83 | 72.75 | 72.75 | 72.75 | 8, 9, 10 |
| 185 | 1942.00 | 483.00 | 662.00 | 1510.00 | 845.81 | 88.69 | 88.19 | 72.52 | 69.04 | 69.04 | 69.04 | 8, 9, 10 |
| 186 | 1425.00 | 498.25 | 1353.00 | 1543.00 | 860.86 | 96.28 | 94.78 | 73.92 | 72.02 | 72.02 | 72.02 | 8, 9, 10 |
| 187 | 1958.00 | 256.75 | 352.50 | 1578.00 | 877.98 | 87.25 | 87.25 | 71.94 | 71.94 | 71.94 | 71.94 | 7, 8, 9, 10 |
| 188 | 1215.00 | 263.13 | 1411.00 | 1589.00 | 882.22 | 97.03 | 97.03 | 74.33 | 69.74 | 69.74 | 69.74 | 8, 9, 10 |
| 189 | 1986.00 | 507.00 | 722.00 | 1618.00 | 904.25 | 96.19 | 96.19 | 73.61 | 73.23 | 73.23 | 73.23 | 8, 9, 10 |
| 190 | 1455.00 | 518.25 | 1483.00 | 1647.00 | 917.33 | 104.66 | 104.66 | 74.92 | 74.35 | 74.35 | 74.35 | 8, 9, 10 |
| 191 | 2002.00 | 60.00 | 60.00 | 1680.00 | 918.95 | 60.00 | 60.00 | 60.00 | 58.35 | 58.35 | 58.35 | 8, 9, 10 |
| 192 | 1105.00 | 65.95 | 1167.00 | 1167.00 | 963.98 | 90.92 | 663.98 | 63.95 | 63.95 | 63.95 | 63.95 | 7, 8, 9, 10 |
| 193 | 2060.00 | 432.00 | 588.00 | 1298.00 | 697.53 | 88.81 | 340.00 | 72.20 | 68.31 | 68.31 | 68.31 | 8, 9, 10 |
| 194 | 1515.00 | 444.25 | 1189.00 | 1373.00 | 721.11 | 95.83 | 345.14 | 73.39 | 69.05 | 69.05 | 69.05 | 8, 9, 10 |
| 195 | 2076.00 | 234.25 | 306.50 | 1422.00 | 744.13 | 86.28 | 182.28 | 71.55 | 67.65 | 67.65 | 67.65 | 8, 9, 10 |
| 196 | 1293.00 | 240.63 | 1213.00 | 1453.00 | 754.30 | 95.67 | 190.80 | 73.75 | 71.20 | 71.20 | 71.20 | 8, 9, 10 |
| 197 | 2104.00 | 457.00 | 614.00 | 1484.00 | 780.03 | 94.50 | 188.31 | 72.88 | 68.00 | 68.00 | 68.00 | 8, 9, 10 |
| 198 | 1545.00 | 475.25 | 1247.00 | 1517.00 | 796.61 | 102.45 | 194.95 | 74.55 | 70.89 | 70.89 | 70.89 | 8, 9, 10 |
| 199 | 2120.00 | 133.63 | 168.50 | 1554.00 | 812.56 | 82.59 | 108.34 | 70.25 | 66.83 | 66.83 | 66.83 | 8, 9, 10 |
| 200 | 1223.00 | 139.69 | 1265.00 | 1553.00 | 801.23 | 95.73 | 134.61 | 74.20 | 70.84 | 70.84 | 70.84 | 8, 9, 10 |
| 201 | 2158.00 | 473.00 | 640.00 | 1586.00 | 829.22 | 94.06 | 118.38 | 73.59 | 71.77 | 71.77 | 71.77 | 8, 9, 10 |
| 202 | 1585.00 | 495.25 | 1299.00 | 1617.00 | 847.55 | 101.52 | 124.27 | 74.89 | 72.82 | 72.82 | 72.82 | 8, 9, 10 |
| 203 | 2174.00 | 255.50 | 336.00 | 1650.00 | 866.50 | 92.41 | 115.91 | 73.38 | 66.92 | 66.92 | 66.92 | 8, 9, 10 |
| 204 | 1351.00 | 261.88 | 1337.00 | 1663.00 | 872.42 | 102.23 | 125.17 | 75.61 | 72.26 | 72.26 | 72.26 | 8, 9, 10 |
| 205 | 2202.00 | 500.00 | 680.00 | 1692.00 | 895.59 | 101.50 | 123.69 | 75.03 | 71.60 | 71.60 | 71.60 | 8, 9, 10 |
| 206 | 1615.00 | 516.25 | 1387.00 | 1721.00 | 909.92 | 109.89 | 131.33 | 76.59 | 72.22 | 72.22 | 72.22 | 8, 9, 10 |
| 207 | 2218.00 | 87.06 | 105.13 | 1754.00 | 920.69 | 77.34 | 77.47 | 68.83 | 66.10 | 66.10 | 66.10 | 8, 9, 10 |
| 208 | 1239.00 | 93.03 | 1377.00 | 1697.00 | 865.86 | 95.36 | 159.61 | 72.78 | 70.20 | 70.20 | 70.20 | 8, 9, 10 |
| 209 | 2264.00 | 508.00 | 696.00 | 1744.00 | 897.22 | 93.44 | 110.00 | 75.56 | 71.02 | 71.02 | 71.02 | 8, 9, 10 |
| 210 | 1663.00 | 526.25 | 1411.00 | 1781.00 | 918.67 | 100.64 | 115.52 | 76.55 | 73.86 | 73.86 | 73.86 | 8, 9, 10 |
| 211 | 2280.00 | 271.50 | 3 | | | | | | | | | |

| | | | | | | | | | | | | |
|-----|---------|--------|---------|---------|---------|--------|--------|-------|-------|-------|-------|----------|
| 225 | 2474.00 | 564.00 | 816.00 | 2016.00 | 1001.50 | 91.44 | 170.81 | 77.52 | 77.25 | 77.25 | 77.25 | 8, 9, 10 |
| 226 | 1817.00 | 578.25 | 1649.00 | 2067.00 | 1028.23 | 98.48 | 176.11 | 78.73 | 78.28 | 78.28 | 78.28 | 8, 9, 10 |
| 227 | 2490.00 | 299.50 | 423.00 | 2108.00 | 1054.72 | 88.97 | 109.97 | 77.13 | 71.94 | 71.94 | 71.94 | 8, 9, 10 |
| 228 | 1547.00 | 305.88 | 1685.00 | 2131.00 | 1068.05 | 98.42 | 118.67 | 79.19 | 77.05 | 77.05 | 77.05 | 8, 9, 10 |
| 229 | 2518.00 | 582.00 | 854.00 | 2164.00 | 1097.63 | 97.31 | 116.44 | 78.27 | 76.23 | 76.23 | 76.23 | 8, 9, 10 |
| 230 | 1847.00 | 606.25 | 1733.00 | 2195.00 | 1117.86 | 105.30 | 123.30 | 80.23 | 76.91 | 76.91 | 76.91 | 8, 9, 10 |
| 231 | 2534.00 | 166.25 | 230.25 | 2230.00 | 1137.47 | 85.47 | 85.22 | 76.08 | 75.52 | 75.52 | 75.52 | 8, 9, 10 |
| 232 | 1457.00 | 172.31 | 1765.00 | 2227.00 | 1129.55 | 98.67 | 111.67 | 80.03 | 78.85 | 78.85 | 78.85 | 8, 9, 10 |
| 233 | 2572.00 | 600.00 | 894.00 | 2266.00 | 1162.13 | 97.06 | 95.69 | 79.42 | 73.57 | 73.57 | 73.57 | 8, 9, 10 |
| 234 | 1887.00 | 628.25 | 1813.00 | 2299.00 | 1184.86 | 104.55 | 101.80 | 80.84 | 77.93 | 77.93 | 77.93 | 8, 9, 10 |
| 235 | 2588.00 | 322.75 | 466.50 | 2334.00 | 1208.34 | 95.47 | 93.72 | 79.58 | 77.51 | 77.51 | 77.51 | 8, 9, 10 |
| 236 | 1605.00 | 329.13 | 1867.00 | 2349.00 | 1218.55 | 105.36 | 103.23 | 81.86 | 77.69 | 77.69 | 77.69 | 8, 9, 10 |
| 237 | 2616.00 | 634.00 | 950.00 | 2382.00 | 1246.50 | 104.69 | 102.06 | 81.36 | 78.60 | 78.60 | 78.60 | 8, 9, 10 |
| 238 | 1917.00 | 652.25 | 1935.00 | 2411.00 | 1265.23 | 113.11 | 109.98 | 83.11 | 79.57 | 79.57 | 79.57 | 8, 9, 10 |
| 239 | 2632.00 | 104.13 | 140.00 | 2444.00 | 1280.47 | 80.59 | 80.59 | 75.09 | 66.00 | 66.00 | 66.00 | 8, 9, 10 |
| 240 | 1465.00 | 110.09 | 1941.00 | 2387.00 | 1229.67 | 98.67 | 162.92 | 79.05 | 71.93 | 71.93 | 71.93 | 8, 9, 10 |
| 241 | 2678.00 | 650.00 | 982.00 | 2442.00 | 1267.31 | 96.81 | 113.56 | 81.69 | 77.19 | 77.19 | 77.19 | 8, 9, 10 |
| 242 | 1965.00 | 666.25 | 1987.00 | 2481.00 | 1294.64 | 104.02 | 119.27 | 82.83 | 77.16 | 77.16 | 77.16 | 8, 9, 10 |
| 243 | 2694.00 | 342.75 | 509.50 | 2518.00 | 1322.22 | 94.66 | 94.16 | 81.33 | 76.44 | 76.44 | 76.44 | 8, 9, 10 |
| 244 | 1671.00 | 349.13 | 2039.00 | 2537.00 | 1336.36 | 104.30 | 103.30 | 83.48 | 79.44 | 79.44 | 79.44 | 8, 9, 10 |
| 245 | 2722.00 | 671.00 | 1036.00 | 2572.00 | 1367.75 | 103.38 | 101.63 | 82.63 | 76.07 | 76.07 | 76.07 | 8, 9, 10 |
| 246 | 1995.00 | 693.25 | 2105.00 | 2601.00 | 1389.58 | 111.52 | 109.02 | 84.53 | 79.15 | 79.15 | 79.15 | 8, 9, 10 |
| 247 | 2738.00 | 187.88 | 278.00 | 2634.00 | 1410.91 | 91.84 | 91.84 | 80.09 | 75.07 | 75.07 | 75.07 | 8, 9, 10 |
| 248 | 1571.00 | 193.94 | 2155.00 | 2629.00 | 1404.30 | 105.23 | 118.73 | 84.05 | 78.84 | 78.84 | 78.84 | 8, 9, 10 |
| 249 | 2776.00 | 698.00 | 1094.00 | 2672.00 | 1440.06 | 103.81 | 103.31 | 83.66 | 79.71 | 79.71 | 79.71 | 8, 9, 10 |
| 250 | 2035.00 | 716.25 | 2219.00 | 2703.00 | 1465.61 | 111.42 | 109.92 | 85.14 | 80.69 | 80.69 | 80.69 | 8, 9, 10 |

A.1.3 Minimum Filtering

This section considers the problem of selecting the minimum of a window. Hence $l = u = 0$.

| W | 0) | 1) | 2) | 3) | 4) | 5) | 6) | 7) | 8) | 9) | 10) | Best strategies |
|----|------|------|------|-----|------|------|------|------|------|------|------|----------------------------------|
| 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 2 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 3 | 2.0 | 1.5 | 1.5 | 2.0 | 1.5 | 1.5 | 1.5 | 1.5 | 1.5 | 1.5 | 1.5 | 1, 2, 4, 5, 6, 7, 8, 9, 10 |
| 4 | 2.0 | 2.5 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 0, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 5 | 3.0 | 2.0 | 2.0 | 3.0 | 2.25 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 2.0 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 6 | 3.0 | 3.0 | 3.0 | 3.0 | 2.5 | 2.5 | 2.5 | 2.5 | 2.33 | 2.33 | 2.33 | 8, 9, 10 |
| 7 | 4.0 | 2.25 | 2.25 | 4.0 | 2.75 | 2.25 | 2.25 | 2.25 | 2.25 | 2.25 | 2.25 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 8 | 3.0 | 3.25 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 2.75 | 2.67 | 2.67 | 2.67 | 8, 9, 10 |
| 9 | 5.0 | 2.5 | 2.5 | 4.0 | 3.13 | 2.5 | 2.5 | 2.5 | 2.5 | 2.5 | 2.5 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 10 | 4.0 | 3.5 | 4.0 | 4.0 | 3.25 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 5, 6, 7, 8, 9, 10 |
| 11 | 5.0 | 2.5 | 2.5 | 5.0 | 3.38 | 2.5 | 2.5 | 2.5 | 2.5 | 2.5 | 2.5 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 12 | 4.0 | 3.5 | 4.0 | 4.0 | 3.25 | 3.25 | 3.25 | 3.0 | 2.83 | 2.83 | 2.83 | 8, 9, 10 |
| 13 | 6.0 | 2.75 | 3.0 | 5.0 | 3.5 | 2.75 | 2.75 | 2.75 | 2.67 | 2.67 | 2.67 | 8, 9, 10 |
| 14 | 5.0 | 3.75 | 5.0 | 5.0 | 3.75 | 3.25 | 3.25 | 3.25 | 3.0 | 3.0 | 3.0 | 8, 9, 10 |
| 15 | 6.0 | 2.63 | 2.63 | 6.0 | 3.88 | 2.63 | 2.63 | 2.63 | 2.63 | 2.63 | 2.63 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 16 | 4.0 | 3.63 | 4.0 | 4.0 | 4.0 | 3.5 | 4.0 | 3.13 | 3.13 | 3.13 | 3.13 | 7, 8, 9, 10 |
| 17 | 7.0 | 3.0 | 3.0 | 5.0 | 4.06 | 3.0 | 3.0 | 2.88 | 2.83 | 2.83 | 2.83 | 8, 9, 10 |
| 18 | 6.0 | 4.0 | 5.0 | 5.0 | 4.13 | 3.5 | 3.5 | 3.38 | 3.08 | 3.08 | 3.08 | 8, 9, 10 |
| 19 | 7.0 | 2.75 | 2.75 | 6.0 | 4.19 | 2.75 | 2.75 | 2.75 | 2.75 | 2.75 | 2.75 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 20 | 5.0 | 3.75 | 5.0 | 5.0 | 4.0 | 3.5 | 3.5 | 3.25 | 3.17 | 3.17 | 3.17 | 8, 9, 10 |
| 21 | 7.0 | 3.25 | 3.5 | 6.0 | 4.19 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 5, 6, 7, 8, 9, 10 |
| 22 | 6.0 | 4.25 | 6.0 | 6.0 | 4.38 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 3.5 | 5, 6, 7, 8, 9, 10 |
| 23 | 7.0 | 2.75 | 2.75 | 7.0 | 4.44 | 2.75 | 2.75 | 2.75 | 2.75 | 2.75 | 2.75 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 24 | 5.0 | 3.75 | 5.0 | 5.0 | 4.13 | 3.63 | 4.13 | 3.25 | 3.17 | 3.17 | 3.17 | 8, 9, 10 |
| 25 | 8.0 | 3.25 | 3.5 | 6.0 | 4.25 | 3.13 | 3.13 | 3.06 | 2.92 | 2.92 | 2.92 | 8, 9, 10 |
| 26 | 7.0 | 4.25 | 6.0 | 6.0 | 4.38 | 3.63 | 3.63 | 3.56 | 3.25 | 3.25 | 3.25 | 8, 9, 10 |
| 27 | 8.0 | 2.88 | 3.0 | 7.0 | 4.5 | 2.88 | 2.88 | 2.88 | 2.83 | 2.83 | 2.83 | 8, 9, 10 |
| 28 | 6.0 | 3.88 | 6.0 | 6.0 | 4.38 | 3.63 | 3.63 | 3.38 | 3.33 | 3.33 | 3.33 | 8, 9, 10 |
| 29 | 8.0 | 3.5 | 4.0 | 7.0 | 4.63 | 3.13 | 3.13 | 3.13 | 3.0 | 3.0 | 3.0 | 8, 9, 10 |
| 30 | 7.0 | 4.5 | 7.0 | 7.0 | 4.88 | 3.63 | 3.63 | 3.63 | 3.29 | 3.29 | 3.29 | 8, 9, 10 |
| 31 | 8.0 | 2.81 | 2.81 | 8.0 | 4.94 | 2.81 | 2.81 | 2.81 | 2.81 | 2.81 | 2.81 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 32 | 5.0 | 3.81 | 5.0 | 5.0 | 5.0 | 3.75 | 5.0 | 3.31 | 3.31 | 3.31 | 3.31 | 7, 8, 9, 10 |
| 33 | 9.0 | 3.5 | 3.5 | 6.0 | 5.03 | 3.25 | 3.5 | 3.13 | 3.13 | 3.06 | 3.06 | 9, 10 |
| 34 | 8.0 | 4.5 | 6.0 | 6.0 | 5.06 | 3.75 | 4.0 | 3.63 | 3.63 | 3.56 | 3.56 | 9, 10 |
| 35 | 9.0 | 3.0 | 3.0 | 7.0 | 5.09 | 3.0 | 3.0 | 2.94 | 2.92 | 2.92 | 2.92 | 8, 9, 10 |
| 36 | 7.0 | 4.0 | 6.0 | 6.0 | 4.88 | 3.75 | 3.75 | 3.44 | 3.33 | 3.33 | 3.33 | 8, 9, 10 |
| 37 | 9.0 | 3.75 | 4.0 | 7.0 | 5.03 | 3.25 | 3.25 | 3.19 | 3.04 | 3.04 | 3.04 | 8, 9, 10 |
| 38 | 8.0 | 4.75 | 7.0 | 7.0 | 5.19 | 3.75 | 3.75 | 3.69 | 3.38 | 3.38 | 3.38 | 8, 9, 10 |
| 39 | 9.0 | 2.88 | 2.88 | 8.0 | 5.22 | 2.88 | 2.88 | 2.88 | 2.88 | 2.88 | 2.88 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 40 | 6.0 | 3.88 | 6.0 | 6.0 | 4.88 | 3.75 | 4.25 | 3.38 | 3.38 | 3.38 | 3.38 | 7, 8, 9, 10 |
| 41 | 9.0 | 3.75 | 4.0 | 7.0 | 4.97 | 3.25 | 3.25 | 3.22 | 3.08 | 3.08 | 3.08 | 8, 9, 10 |
| 42 | 8.0 | 4.75 | 7.0 | 7.0 | 5.06 | 3.75 | 3.75 | 3.72 | 3.38 | 3.38 | 3.38 | 8, 9, 10 |
| 43 | 9.0 | 3.13 | 3.25 | 8.0 | 5.16 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 5, 6, 7, 8, 9, 10 |
| 44 | 7.0 | 4.13 | 7.0 | 7.0 | 5.0 | 3.75 | 3.75 | 3.5 | 3.42 | 3.42 | 3.42 | 8, 9, 10 |
| 45 | 9.0 | 4.0 | 4.5 | 8.0 | 5.22 | 3.25 | 3.25 | 3.25 | 3.25 | 3.25 | 3.25 | 5, 6, 7, 8, 9, 10 |
| 46 | 8.0 | 5.0 | 8.0 | 8.0 | 5.44 | 3.75 | 3.75 | 3.75 | 3.75 | 3.75 | 3.75 | 5, 6, 7, 8, 9, 10 |
| 47 | 9.0 | 2.88 | 2.88 | 9.0 | 5.47 | 2.88 | 2.88 | 2.88 | 2.88 | 2.88 | 2.88 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 48 | 6.0 | 3.88 | 6.0 | 6.0 | 5.06 | 3.81 | 5.06 | 3.38 | 3.38 | 3.38 | 3.38 | 7, 8, 9, 10 |
| 49 | 10.0 | 3.75 | 4.0 | 7.0 | 5.13 | 3.31 | 3.56 | 3.25 | 3.08 | 3.08 | 3.08 | 8, 9, 10 |
| 50 | 9.0 | 4.75 | 7.0 | 7.0 | 5.19 | 3.81 | 4.06 | 3.75 | 3.42 | 3.42 | 3.42 | 8, 9, 10 |
| 51 | 10.0 | 3.13 | 3.25 | 8.0 | 5.25 | 3.06 | 3.06 | 3.03 | 2.96 | 2.96 | 2.96 | 8, 9, 10 |
| 52 | 8.0 | 4.13 | 7.0 | 7.0 | 5.06 | 3.81 | 3.81 | 3.53 | 3.46 | 3.46 | 3.46 | 8, 9, 10 |
| 53 | 10.0 | 4.0 | 4.5 | 8.0 | 5.25 | 3.31 | 3.31 | 3.28 | 3.13 | 3.13 | 3.13 | 8, 9, 10 |
| 54 | 9.0 | 5.0 | 8.0 | 8.0 | 5.44 | 3.81 | 3.81 | 3.78 | 3.42 | 3.42 | 3.42 | 8, 9, 10 |
| 55 | 10.0 | 2.94 | 3.0 | 9.0 | 5.5 | 2.94 | 2.94 | 2.94 | 2.92 | 2.92 | 2.92 | 8, 9, 10 |
| 56 | 7.0 | 3.94 | 7.0 | 7.0 | 5.19 | 3.81 | 4.31 | 3.44 | 3.42 | 3.42 | 3.42 | 8, 9, 10 |
| 57 | 10.0 | 4.0 | 4.5 | 8.0 | 5.31 | 3.31 | 3.31 | 3.28 | 3.28 | 3.19 | 3.19 | 9, 10 |
| 58 | 9.0 | 5.0 | 8.0 | 8.0 | 5.44 | 3.81 | 3.81 | 3.78 | 3.78 | 3.69 | 3.69 | 9, 10 |
| 59 | 10.0 | 3.25 | 3.5 | 9.0 | 5.56 | 3.06 | 3.06 | 3.06 | 3.0 | 3.0 | 3.0 | 8, 9, 10 |
| 60 | 8.0 | 4.25 | 8.0 | 8.0 | 5.44 | 3.81 | 3.81 | 3.56 | 3.46 | 3.46 | 3.46 | 8, 9, 10 |
| 61 | 10.0 | 4.25 | 5.0 | 9.0 | 5.69 | 3.31 | 3.31 | 3.31 | 3.15 | 3.15 | 3.15 | 8, 9, 10 |

| | | | | | | | | | | | | |
|-----|------|------|------|------|------|------|------|------|------|------|------|-------------------------|
| 62 | 9.0 | 5.25 | 9.0 | 9.0 | 5.94 | 3.81 | 3.81 | 3.81 | 3.48 | 3.48 | 3.48 | 8, 9, 10 |
| 63 | 10.0 | 2.91 | 2.91 | 10.0 | 5.97 | 2.91 | 2.91 | 2.91 | 2.91 | 2.91 | 2.91 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 64 | 6.0 | 3.91 | 6.0 | 6.0 | 6.0 | 3.88 | 6.0 | 3.41 | 3.41 | 3.41 | 3.41 | 7, 8, 9, 10 |
| 65 | 11.0 | 4.0 | 4.0 | 7.0 | 6.02 | 3.38 | 4.0 | 3.28 | 3.17 | 3.17 | 3.17 | 8, 9, 10 |
| 66 | 10.0 | 5.0 | 7.0 | 7.0 | 6.03 | 3.88 | 4.5 | 3.78 | 3.48 | 3.48 | 3.48 | 9, 10 |
| 67 | 11.0 | 3.25 | 3.25 | 8.0 | 6.05 | 3.13 | 3.25 | 3.06 | 3.06 | 3.03 | 3.03 | 9, 10 |
| 68 | 9.0 | 4.25 | 7.0 | 7.0 | 5.81 | 3.88 | 4.0 | 3.56 | 3.5 | 3.5 | 3.5 | 8, 9, 10 |
| 69 | 11.0 | 4.25 | 4.5 | 8.0 | 5.95 | 3.38 | 3.5 | 3.31 | 3.31 | 3.31 | 3.31 | 7, 8, 9, 10 |
| 70 | 10.0 | 5.25 | 8.0 | 8.0 | 6.09 | 3.88 | 4.0 | 3.81 | 3.81 | 3.81 | 3.62 | 10 |
| 71 | 11.0 | 3.0 | 3.0 | 9.0 | 6.11 | 3.0 | 3.0 | 2.97 | 2.96 | 2.96 | 2.96 | 8, 9, 10 |
| 72 | 8.0 | 4.0 | 7.0 | 7.0 | 5.75 | 3.88 | 4.38 | 3.47 | 3.46 | 3.46 | 3.46 | 9, 10 |
| 73 | 11.0 | 4.25 | 4.5 | 8.0 | 5.83 | 3.38 | 3.38 | 3.31 | 3.17 | 3.17 | 3.17 | 8, 9, 10 |
| 74 | 10.0 | 5.25 | 8.0 | 8.0 | 5.91 | 3.88 | 3.88 | 3.81 | 3.5 | 3.5 | 3.5 | 8, 9, 10 |
| 75 | 11.0 | 3.38 | 3.5 | 9.0 | 5.98 | 3.13 | 3.13 | 3.09 | 3.02 | 3.02 | 3.02 | 8, 9, 10 |
| 76 | 9.0 | 4.38 | 8.0 | 8.0 | 5.81 | 3.88 | 3.88 | 3.59 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 77 | 11.0 | 4.5 | 5.0 | 9.0 | 6.02 | 3.38 | 3.38 | 3.34 | 3.19 | 3.19 | 3.19 | 8, 9, 10 |
| 78 | 10.0 | 5.5 | 9.0 | 9.0 | 6.22 | 3.88 | 3.88 | 3.84 | 3.5 | 3.5 | 3.5 | 9, 10 |
| 79 | 11.0 | 2.94 | 2.94 | 10.0 | 6.23 | 2.94 | 2.94 | 2.94 | 2.94 | 2.94 | 2.94 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 80 | 7.0 | 3.94 | 7.0 | 7.0 | 5.81 | 3.88 | 5.13 | 3.44 | 3.44 | 3.44 | 3.44 | 7, 8, 9, 10 |
| 81 | 11.0 | 4.25 | 4.5 | 8.0 | 5.86 | 3.38 | 3.63 | 3.34 | 3.34 | 3.23 | 3.23 | 9, 10 |
| 82 | 10.0 | 5.25 | 8.0 | 8.0 | 5.91 | 3.88 | 4.13 | 3.84 | 3.84 | 3.73 | 3.73 | 9, 10 |
| 83 | 11.0 | 3.38 | 3.5 | 9.0 | 5.95 | 3.13 | 3.13 | 3.11 | 3.04 | 3.04 | 3.04 | 8, 9, 10 |
| 84 | 9.0 | 4.38 | 8.0 | 8.0 | 5.75 | 3.88 | 3.88 | 3.61 | 3.5 | 3.5 | 3.5 | 8, 9, 10 |
| 85 | 11.0 | 4.5 | 5.0 | 9.0 | 5.92 | 3.38 | 3.38 | 3.36 | 3.19 | 3.19 | 3.19 | 8, 9, 10 |
| 86 | 10.0 | 5.5 | 9.0 | 9.0 | 6.09 | 3.88 | 3.88 | 3.86 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 87 | 11.0 | 3.06 | 3.13 | 10.0 | 6.14 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 10 |
| 88 | 8.0 | 4.06 | 8.0 | 8.0 | 5.81 | 3.88 | 4.38 | 3.5 | 3.5 | 3.5 | 3.5 | 10 |
| 89 | 11.0 | 4.5 | 5.0 | 9.0 | 5.92 | 3.38 | 3.38 | 3.36 | 3.21 | 3.21 | 3.21 | 8, 9, 10 |
| 90 | 10.0 | 5.5 | 9.0 | 9.0 | 6.03 | 3.88 | 3.88 | 3.86 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 91 | 11.0 | 3.5 | 3.75 | 10.0 | 6.14 | 3.13 | 3.13 | 3.13 | 3.13 | 3.13 | 3.13 | 5, 6, 7, 8, 9, 10 |
| 92 | 9.0 | 4.5 | 9.0 | 9.0 | 6.0 | 3.88 | 3.88 | 3.63 | 3.54 | 3.54 | 3.54 | 8, 9, 10 |
| 93 | 11.0 | 4.75 | 5.5 | 10.0 | 6.23 | 3.38 | 3.38 | 3.38 | 3.38 | 3.38 | 3.38 | 5, 6, 7, 8, 9, 10 |
| 94 | 10.0 | 5.75 | 10.0 | 10.0 | 6.47 | 3.88 | 3.88 | 3.88 | 3.88 | 3.88 | 3.88 | 5, 6, 7, 8, 9, 10 |
| 95 | 11.0 | 2.94 | 2.94 | 11.0 | 6.48 | 2.94 | 2.94 | 2.94 | 2.94 | 2.94 | 2.94 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 96 | 7.0 | 3.94 | 7.0 | 7.0 | 6.03 | 3.91 | 6.03 | 3.44 | 3.44 | 3.44 | 3.44 | 7, 8, 9, 10 |
| 97 | 12.0 | 4.25 | 4.5 | 8.0 | 6.06 | 3.41 | 4.03 | 3.34 | 3.19 | 3.19 | 3.19 | 8, 9, 10 |
| 98 | 11.0 | 5.25 | 8.0 | 8.0 | 6.09 | 3.91 | 4.53 | 3.84 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 99 | 12.0 | 3.38 | 3.5 | 9.0 | 6.13 | 3.16 | 3.28 | 3.13 | 3.04 | 3.04 | 3.04 | 8, 9, 10 |
| 100 | 10.0 | 4.38 | 8.0 | 8.0 | 5.91 | 3.91 | 4.03 | 3.63 | 3.54 | 3.54 | 3.54 | 8, 9, 10 |
| 101 | 12.0 | 4.5 | 5.0 | 9.0 | 6.06 | 3.41 | 3.53 | 3.38 | 3.21 | 3.21 | 3.21 | 8, 9, 10 |
| 102 | 11.0 | 5.5 | 9.0 | 9.0 | 6.22 | 3.91 | 4.03 | 3.88 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 103 | 12.0 | 3.06 | 3.13 | 10.0 | 6.25 | 3.03 | 3.03 | 3.02 | 2.98 | 2.98 | 2.98 | 8, 9, 10 |
| 104 | 9.0 | 4.06 | 8.0 | 8.0 | 5.91 | 3.91 | 4.41 | 3.52 | 3.48 | 3.48 | 3.48 | 8, 9, 10 |
| 105 | 12.0 | 4.5 | 5.0 | 9.0 | 6.0 | 3.41 | 3.41 | 3.38 | 3.37 | 3.27 | 3.27 | 9, 10 |
| 106 | 11.0 | 5.5 | 9.0 | 9.0 | 6.09 | 3.91 | 3.91 | 3.88 | 3.67 | 3.77 | 3.77 | 9, 10 |
| 107 | 12.0 | 3.5 | 3.75 | 10.0 | 6.19 | 3.16 | 3.16 | 3.14 | 3.06 | 3.06 | 3.06 | 8, 9, 10 |
| 108 | 10.0 | 4.5 | 9.0 | 9.0 | 6.03 | 3.91 | 3.91 | 3.64 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 109 | 12.0 | 4.75 | 5.5 | 10.0 | 6.25 | 3.41 | 3.41 | 3.41 | 3.21 | 3.21 | 3.21 | 8, 9, 10 |
| 110 | 11.0 | 5.75 | 10.0 | 10.0 | 6.47 | 3.91 | 3.91 | 3.91 | 3.54 | 3.54 | 3.54 | 8, 9, 10 |
| 111 | 12.0 | 2.97 | 3.0 | 11.0 | 6.5 | 2.97 | 2.97 | 2.97 | 2.96 | 2.96 | 2.96 | 8, 9, 10 |
| 112 | 8.0 | 3.97 | 8.0 | 8.0 | 6.09 | 3.91 | 5.16 | 3.47 | 3.46 | 3.46 | 3.46 | 8, 9, 10 |
| 113 | 12.0 | 4.5 | 5.0 | 9.0 | 6.16 | 3.41 | 3.66 | 3.38 | 3.23 | 3.23 | 3.23 | 8, 9, 10 |
| 114 | 11.0 | 5.5 | 9.0 | 9.0 | 6.22 | 3.91 | 4.16 | 3.88 | 3.54 | 3.54 | 3.54 | 8, 9, 10 |
| 115 | 12.0 | 3.5 | 3.75 | 10.0 | 6.28 | 3.16 | 3.16 | 3.14 | 3.14 | 3.09 | 3.09 | 9, 10 |
| 116 | 10.0 | 4.5 | 9.0 | 9.0 | 6.09 | 3.91 | 3.91 | 3.64 | 3.56 | 3.56 | 3.56 | 8, 9, 10 |
| 117 | 12.0 | 4.75 | 5.5 | 10.0 | 6.28 | 3.41 | 3.41 | 3.39 | 3.39 | 3.39 | 3.39 | 7, 8, 9, 10 |
| 118 | 11.0 | 5.75 | 10.0 | 10.0 | 6.47 | 3.91 | 3.91 | 3.89 | 3.89 | 3.89 | 3.89 | 7, 8, 9, 10 |
| 119 | 12.0 | 3.13 | 3.25 | 11.0 | 6.53 | 3.03 | 3.03 | 3.03 | 3.0 | 3.0 | 3.0 | 8, 9, 10 |
| 120 | 9.0 | 4.13 | 9.0 | 9.0 | 6.22 | 3.91 | 4.41 | 3.53 | 3.5 | 3.5 | 3.5 | 8, 9, 10 |
| 121 | 12.0 | 4.75 | 5.5 | 10.0 | 6.34 | 3.41 | 3.41 | 3.39 | 3.23 | 3.23 | 3.23 | 8, 9, 10 |
| 122 | 11.0 | 5.75 | 10.0 | 10.0 | 6.47 | 3.91 | 3.91 | 3.89 | 3.56 | 3.56 | 3.56 | 8, 9, 10 |
| 123 | 12.0 | 3.63 | 4.0 | 11.0 | 6.59 | 3.16 | 3.16 | 3.16 | 3.07 | 3.07 | 3.07 | 8, 9, 10 |
| 124 | 10.0 | 4.63 | 10.0 | 10.0 | 6.47 | 3.91 | 3.91 | 3.66 | 3.57 | 3.57 | 3.57 | 8, 9, 10 |
| 125 | 12.0 | 5.0 | 6.0 | 11.0 | 6.72 | 3.41 | 3.41 | 3.41 | 3.24 | 3.24 | 3.24 | 8, 9, 10 |
| 126 | 11.0 | 6.0 | 11.0 | 11.0 | 6.97 | 3.91 | 3.91 | 3.91 | 3.57 | 3.57 | 3.57 | 8, 9, 10 |
| 127 | 12.0 | 2.95 | 2.95 | 12.0 | 6.98 | 2.95 | 2.95 | 2.95 | 2.95 | 2.95 | 2.95 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 128 | 7.0 | 3.95 | 7.0 | 7.0 | 7.0 | 3.94 | 7.0 | 3.45 | 3.45 | 3.45 | 3.45 | 7, 8, 9, 10 |
| 129 | 13.0 | 4.5 | 4.5 | 8.0 | 7.01 | 3.44 | 4.5 | 3.38 | 3.38 | 3.27 | 3.27 | 9, 10 |
| 130 | 12.0 | 5.5 | 8.0 | 8.0 | 7.02 | 3.94 | 5.0 | 3.88 | 3.88 | 3.77 | 3.7 | 10 |
| 131 | 13.0 | 3.5 | 3.5 | 9.0 | 7.02 | 3.19 | 3.5 | 3.14 | 3.08 | 3.08 | 3.08 | 8, 9, 10 |
| 132 | 11.0 | 4.5 | 8.0 | 8.0 | 6.78 | 3.94 | 4.25 | 3.64 | 3.56 | 3.54 | 3.54 | 9, 10 |
| 133 | 13.0 | 4.75 | 5.0 | 9.0 | 6.91 | 3.44 | 3.75 | 3.39 | 3.24 | 3.24 | 3.24 | 8, 9, 10 |
| 134 | 12.0 | 5.75 | 9.0 | 9.0 | 7.05 | 3.94 | 4.25 | 3.89 | 3.57 | 3.57 | 3.57 | 8, 9, 10 |
| 135 | 13.0 | 3.13 | 3.13 | 10.0 | 7.05 | 3.06 | 3.13 | 3.03 | 3.03 | 3.02 | 3.02 | 9, 10 |
| 136 | 10.0 | 4.13 | 8.0 | 8.0 | 6.69 | 3.94 | 4.5 | 3.53 | 3.53 | 3.52 | 3.52 | 9, 10 |
| 137 | 13.0 | 4.75 | 5.0 | 9.0 | 6.76 | 3.44 | 3.5 | 3.39 | 3.25 | 3.25 | 3.25 | 8, 9, 10 |
| 138 | 12.0 | 5.75 | 9.0 | 9.0 | 6.83 | 3.94 | 4.0 | 3.89 | 3.57 | 3.57 | 3.57 | 8, 9, 10 |
| 139 | 13.0 | 3.63 | 3.75 | 10.0 | 6.9 | 3.19 | 3.25 | 3.16 | 3.16 | 3.16 | 3.16 | 7, 8, 9, 10 |
| 140 | 11.0 | 4.63 | 9.0 | 9.0 | 6.72 | 3.94 | 4.0 | 3.66 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 141 | 13.0 | 5.0 | 5.5 | 10.0 | 6.91 | 3.44 | 3.5 | 3.41 | 3.41 | 3.41 | 3.31 | 10 |
| 142 | 12.0 | 6.0 | 10.0 | 10.0 | 7.11 | 3.94 | 4.0 | 3.91 | 3.91 | 3.91 | 3.81 | 10 |
| 143 | 13.0 | 3.0 | 3.0 | 11.0 | 7.12 | 3.0 | 3.0 | 2.98 | 2.98 | 2.98 | 2.98 | 8, 9, 10 |
| 144 | 9.0 | 4.0 | 8.0 | 8.0 | 6.69 | 3.94 | 5.19 | 3.48 | 3.48 | 3.48 | 3.48 | 8, 9, 10 |
| 145 | 13.0 | 4.75 | 5.0 | 9.0 | 6.73 | 3.44 | 3.69 | 3.39 | 3.23 | 3.23 | 3.23 | 8, 9, 10 |
| 146 | 12.0 | 5.75 | 9.0 | 9.0 | 6.77 | 3.94 | 4.19 | 3.89 | 3.56 | 3.56 | 3.56 | 8, 9, 10 |
| 147 | 13.0 | 3.63 | 3.75 | 10.0 | 6.8 | 3.19 | 3.19 | 3.16 | 3.08 | 3.08 | 3.08 | 8, 9, 10 |
| 148 | 11.0 | 4.63 | 9.0 | 9.0 | 6.59 | 3.94 | 3.94 | 3.66 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 149 | 13.0 | 5.0 | 5.5 | 10.0 | 6.76 | 3.44 | 3.44 | 3.41 | 3.25 | 3.25 | 3.25 | 8, 9, 10 |
| 150 | 12.0 | 6.0 | 10.0 | 10.0 | 6.92 | 3.94 | 3.94 | 3.91 | 3.56 | 3.56 | 3.56 | 8, 9, 10 |
| 151 | 13.0 | 3.19 | 3.25 | 11.0 | 6.96 | 3.06 | 3.06 | 3.05 | 3.01 | 3.01 | 3.01 | 8, 9, 10 |
| 152 | 10.0 | 4.19 | 9.0 | 9.0 | 6.63 | 3.94 | 4.44 | 3.55 | 3.51 | 3.51 | 3.51 | 8, 9, 10 |
| 153 | 13.0 | 5.0 | 5.5 | 10.0 | 6.73 | 3.44 | 3.44 | 3.41 | 3.41 | 3.3 | 3.3 | 9, 10 |
| 154 | 12.0 | 6.0 | 10.0 | 10.0 | 6.83 | 3.94 | 3.94 | 3.91 | 3.91 | 3.8 | 3.8 | 9, 10 |
| 155 | 13.0 | 3.75 | 4.0 | 11.0 | 6.93 | 3.19 | 3.19 | 3.17 | 3.09 | 3.09 | 3.09 | 8, 9, 10 |
| 156 | 11.0 | 4.75 | 10.0 | 10.0 | 6.78 | 3.94 | 3.94 | 3.67 | 3.56 | 3.54 | 3.54 | 9, 10 |
| 157 | 13.0 | 5.25 | 6.0 | 11.0 | 7.01 | 3.44 | 3.44 | 3.42 | 3.25 | 3.25 | 3.25 | 8, 9, 10 |
| 158 | 12.0 | 6.25 | 11.0 | 11.0 | 7.23 | 3.94 | 3.94 | 3.92 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 159 | 13.0 | 2.97 | 2.97 | 12.0 | 7.24 | 2.97 | 2.97 | 2.97 | 2.97 | 2.97 | 2.97 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 160 | 8.0 | 3.97 | 8.0 | 8.0 | 6.78 | 3.94 | 6.06 | 3.47 | 3.47 | 3.47 | 3.47 | 7, 8, 9, 10 |

APPENDIX A. ALGORITHM RESULTS

| | | | | | | | | | | | | |
|-----|------|------|------|------|------|------|------|------|------|------|------|-------------------------|
| 161 | 13.0 | 4.75 | 5.0 | 9.0 | 6.8 | 3.44 | 4.06 | 3.41 | 3.26 | 3.26 | 3.26 | 8, 9, 10 |
| 162 | 12.0 | 5.75 | 9.0 | 9.0 | 6.83 | 3.94 | 4.56 | 3.91 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 163 | 13.0 | 3.63 | 3.75 | 10.0 | 6.85 | 3.19 | 3.31 | 3.17 | 3.17 | 3.12 | 3.12 | 9, 10 |
| 164 | 11.0 | 4.63 | 9.0 | 9.0 | 6.63 | 3.94 | 4.06 | 3.67 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 165 | 13.0 | 5.0 | 5.5 | 10.0 | 6.77 | 3.44 | 3.56 | 3.42 | 3.42 | 3.42 | 3.42 | 7, 8, 9, 10 |
| 166 | 12.0 | 6.0 | 10.0 | 10.0 | 6.92 | 3.94 | 4.06 | 3.92 | 3.92 | 3.92 | 3.92 | 7, 8, 9, 10 |
| 167 | 13.0 | 3.19 | 3.25 | 11.0 | 6.95 | 3.06 | 3.06 | 3.05 | 3.02 | 3.02 | 3.02 | 8, 9, 10 |
| 168 | 10.0 | 4.19 | 9.0 | 9.0 | 6.59 | 3.94 | 4.44 | 3.55 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 169 | 13.0 | 5.0 | 5.5 | 10.0 | 6.68 | 3.44 | 3.44 | 3.43 | 3.25 | 3.25 | 3.25 | 8, 9, 10 |
| 170 | 12.0 | 6.0 | 10.0 | 10.0 | 6.77 | 3.94 | 3.94 | 3.93 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 171 | 13.0 | 3.75 | 4.0 | 11.0 | 6.85 | 3.19 | 3.19 | 3.18 | 3.09 | 3.09 | 3.09 | 8, 9, 10 |
| 172 | 11.0 | 4.75 | 10.0 | 10.0 | 6.69 | 3.94 | 3.94 | 3.68 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 173 | 13.0 | 5.25 | 6.0 | 11.0 | 6.9 | 3.44 | 3.44 | 3.45 | 3.26 | 3.26 | 3.26 | 8, 9, 10 |
| 174 | 12.0 | 6.25 | 11.0 | 11.0 | 7.11 | 3.94 | 3.94 | 3.95 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 175 | 13.0 | 3.03 | 3.06 | 12.0 | 7.13 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | 5, 6, 7, 8, 9, 10 |
| 176 | 9.0 | 4.03 | 9.0 | 9.0 | 6.72 | 3.94 | 5.19 | 3.5 | 3.5 | 3.5 | 3.5 | 7, 8, 9, 10 |
| 177 | 13.0 | 5.0 | 5.5 | 10.0 | 6.77 | 3.44 | 3.69 | 3.42 | 3.42 | 3.3 | 3.3 | 9, 10 |
| 178 | 12.0 | 6.0 | 10.0 | 10.0 | 6.83 | 3.94 | 4.19 | 3.92 | 3.92 | 3.8 | 3.8 | 9, 10 |
| 179 | 13.0 | 3.75 | 4.0 | 11.0 | 6.88 | 3.19 | 3.19 | 3.18 | 3.1 | 3.1 | 3.1 | 8, 9, 10 |
| 180 | 11.0 | 4.75 | 10.0 | 10.0 | 6.69 | 3.94 | 3.94 | 3.68 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 181 | 13.0 | 5.25 | 6.0 | 11.0 | 6.87 | 3.44 | 3.44 | 3.43 | 3.26 | 3.26 | 3.26 | 8, 9, 10 |
| 182 | 12.0 | 6.25 | 11.0 | 11.0 | 7.05 | 3.94 | 3.94 | 3.93 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 183 | 13.0 | 3.25 | 3.38 | 12.0 | 7.1 | 3.06 | 3.06 | 3.06 | 3.06 | 3.06 | 3.06 | 5, 6, 7, 8, 9, 10 |
| 184 | 10.0 | 4.25 | 10.0 | 10.0 | 6.78 | 3.94 | 4.44 | 3.56 | 3.56 | 3.56 | 3.56 | 7, 8, 9, 10 |
| 185 | 13.0 | 5.25 | 6.0 | 11.0 | 6.9 | 3.44 | 3.44 | 3.43 | 3.27 | 3.27 | 3.27 | 8, 9, 10 |
| 186 | 12.0 | 6.25 | 11.0 | 11.0 | 7.02 | 3.94 | 3.94 | 3.93 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 187 | 13.0 | 3.88 | 4.25 | 12.0 | 7.13 | 3.19 | 3.19 | 3.19 | 3.19 | 3.19 | 3.19 | 5, 6, 7, 8, 9, 10 |
| 188 | 11.0 | 4.88 | 11.0 | 11.0 | 7.0 | 3.94 | 3.94 | 3.69 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 189 | 13.0 | 5.5 | 6.5 | 12.0 | 7.24 | 3.44 | 3.44 | 3.44 | 3.44 | 3.44 | 3.44 | 5, 6, 7, 8, 9, 10 |
| 190 | 12.0 | 6.5 | 12.0 | 12.0 | 7.48 | 3.94 | 3.94 | 3.94 | 3.94 | 3.94 | 3.72 | 10 |
| 191 | 13.0 | 2.97 | 2.97 | 13.0 | 7.49 | 2.97 | 2.97 | 2.97 | 2.97 | 2.97 | 2.97 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 192 | 8.0 | 3.97 | 8.0 | 8.0 | 7.02 | 3.95 | 7.02 | 3.47 | 3.47 | 3.47 | 3.47 | 7, 8, 9, 10 |
| 193 | 14.0 | 4.75 | 5.0 | 9.0 | 7.03 | 3.45 | 4.52 | 3.41 | 3.25 | 3.25 | 3.25 | 8, 9, 10 |
| 194 | 13.0 | 5.75 | 9.0 | 9.0 | 7.05 | 3.95 | 5.02 | 3.91 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 195 | 14.0 | 3.63 | 3.75 | 10.0 | 7.06 | 3.2 | 3.52 | 3.17 | 3.09 | 3.09 | 3.09 | 8, 9, 10 |
| 196 | 12.0 | 4.63 | 9.0 | 9.0 | 6.83 | 3.95 | 4.27 | 3.67 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 197 | 14.0 | 5.0 | 5.5 | 10.0 | 6.97 | 3.45 | 3.77 | 3.42 | 3.26 | 3.26 | 3.26 | 8, 9, 10 |
| 198 | 13.0 | 6.0 | 10.0 | 10.0 | 7.11 | 3.95 | 4.27 | 3.92 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 199 | 14.0 | 3.19 | 3.25 | 11.0 | 7.13 | 3.08 | 3.14 | 3.06 | 3.02 | 3.02 | 3.02 | 8, 9, 10 |
| 200 | 11.0 | 4.19 | 9.0 | 9.0 | 6.77 | 3.95 | 4.52 | 3.56 | 3.52 | 3.52 | 3.52 | 8, 9, 10 |
| 201 | 14.0 | 5.0 | 5.5 | 10.0 | 6.84 | 3.45 | 3.52 | 3.42 | 3.42 | 3.31 | 3.31 | 9, 10 |
| 202 | 13.0 | 6.0 | 10.0 | 10.0 | 6.92 | 3.95 | 4.02 | 3.92 | 3.92 | 3.81 | 3.81 | 9, 10 |
| 203 | 14.0 | 3.75 | 4.0 | 11.0 | 7.0 | 3.2 | 3.27 | 3.19 | 3.1 | 3.1 | 3.1 | 8, 9, 10 |
| 204 | 12.0 | 4.75 | 10.0 | 10.0 | 6.83 | 3.95 | 4.02 | 3.69 | 3.58 | 3.58 | 3.58 | 8, 9, 10 |
| 205 | 14.0 | 5.25 | 6.0 | 11.0 | 7.03 | 3.45 | 3.52 | 3.44 | 3.26 | 3.26 | 3.26 | 8, 9, 10 |
| 206 | 13.0 | 6.25 | 11.0 | 11.0 | 7.23 | 3.95 | 4.02 | 3.94 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 207 | 14.0 | 3.03 | 3.06 | 12.0 | 7.25 | 3.02 | 3.02 | 3.01 | 2.99 | 2.99 | 2.99 | 8, 9, 10 |
| 208 | 10.0 | 4.03 | 9.0 | 9.0 | 6.83 | 3.95 | 5.2 | 3.51 | 3.49 | 3.49 | 3.49 | 8, 9, 10 |
| 209 | 14.0 | 5.0 | 5.5 | 10.0 | 6.88 | 3.45 | 3.7 | 3.42 | 3.27 | 3.27 | 3.27 | 8, 9, 10 |
| 210 | 13.0 | 6.0 | 10.0 | 10.0 | 6.92 | 3.95 | 4.2 | 3.92 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 211 | 14.0 | 3.75 | 4.0 | 11.0 | 6.97 | 3.2 | 3.2 | 3.19 | 3.19 | 3.13 | 3.13 | 9, 10 |
| 212 | 12.0 | 4.75 | 10.0 | 10.0 | 6.77 | 3.95 | 3.95 | 3.69 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 213 | 14.0 | 5.25 | 6.0 | 11.0 | 6.94 | 3.45 | 3.45 | 3.44 | 3.44 | 3.44 | 3.44 | 7, 8, 9, 10 |
| 214 | 13.0 | 6.25 | 11.0 | 11.0 | 7.11 | 3.95 | 3.95 | 3.94 | 3.94 | 3.94 | 3.94 | 7, 8, 9, 10 |
| 215 | 14.0 | 3.25 | 3.38 | 12.0 | 7.16 | 3.08 | 3.08 | 3.07 | 3.03 | 3.03 | 3.03 | 8, 9, 10 |
| 216 | 11.0 | 4.25 | 10.0 | 10.0 | 6.83 | 3.95 | 4.45 | 3.57 | 3.53 | 3.53 | 3.53 | 8, 9, 10 |
| 217 | 14.0 | 5.25 | 6.0 | 11.0 | 6.94 | 3.45 | 3.45 | 3.44 | 3.26 | 3.26 | 3.26 | 8, 9, 10 |
| 218 | 13.0 | 6.25 | 11.0 | 11.0 | 7.05 | 3.95 | 3.95 | 3.94 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 219 | 14.0 | 3.88 | 4.25 | 12.0 | 7.16 | 3.2 | 3.2 | 3.2 | 3.1 | 3.1 | 3.1 | 8, 9, 10 |
| 220 | 12.0 | 4.88 | 11.0 | 11.0 | 7.02 | 3.95 | 3.95 | 3.7 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 221 | 14.0 | 5.5 | 6.5 | 12.0 | 7.25 | 3.45 | 3.45 | 3.45 | 3.27 | 3.27 | 3.27 | 8, 9, 10 |
| 222 | 13.0 | 6.5 | 12.0 | 12.0 | 7.48 | 3.95 | 3.95 | 3.95 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 223 | 14.0 | 2.98 | 3.0 | 13.0 | 7.5 | 2.98 | 2.98 | 2.98 | 2.98 | 2.98 | 2.98 | 8, 9, 10 |
| 224 | 9.0 | 3.98 | 9.0 | 9.0 | 7.05 | 3.95 | 6.08 | 3.48 | 3.48 | 3.48 | 3.48 | 8, 9, 10 |
| 225 | 14.0 | 5.0 | 5.5 | 10.0 | 7.08 | 3.45 | 4.08 | 3.43 | 3.43 | 3.31 | 3.31 | 9, 10 |
| 226 | 13.0 | 6.0 | 10.0 | 10.0 | 7.11 | 3.95 | 4.58 | 3.93 | 3.93 | 3.81 | 3.81 | 9, 10 |
| 227 | 14.0 | 3.75 | 4.0 | 11.0 | 7.14 | 3.2 | 3.33 | 3.19 | 3.11 | 3.11 | 3.11 | 8, 9, 10 |
| 228 | 12.0 | 4.75 | 10.0 | 10.0 | 6.92 | 3.95 | 4.08 | 3.69 | 3.59 | 3.59 | 3.59 | 8, 9, 10 |
| 229 | 14.0 | 5.25 | 6.0 | 11.0 | 7.08 | 3.45 | 3.58 | 3.44 | 3.27 | 3.27 | 3.27 | 8, 9, 10 |
| 230 | 13.0 | 6.25 | 11.0 | 11.0 | 7.23 | 3.95 | 4.08 | 3.94 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 231 | 14.0 | 3.25 | 3.38 | 12.0 | 7.27 | 3.08 | 3.08 | 3.07 | 3.07 | 3.05 | 3.05 | 9, 10 |
| 232 | 11.0 | 4.25 | 10.0 | 10.0 | 6.92 | 3.95 | 4.45 | 3.57 | 3.57 | 3.55 | 3.55 | 9, 10 |
| 233 | 14.0 | 5.25 | 6.0 | 11.0 | 7.02 | 3.45 | 3.45 | 3.45 | 3.28 | 3.28 | 3.28 | 8, 9, 10 |
| 234 | 13.0 | 6.25 | 11.0 | 11.0 | 7.11 | 3.95 | 3.95 | 3.95 | 3.6 | 3.6 | 3.6 | 8, 9, 10 |
| 235 | 14.0 | 3.88 | 4.25 | 12.0 | 7.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 3.2 | 7, 8, 9, 10 |
| 236 | 12.0 | 4.88 | 11.0 | 11.0 | 7.05 | 3.95 | 3.95 | 3.7 | 3.61 | 3.61 | 3.61 | 8, 9, 10 |
| 237 | 14.0 | 5.5 | 6.5 | 12.0 | 7.27 | 3.45 | 3.45 | 3.46 | 3.46 | 3.45 | 3.45 | 5, 6, 9, 10 |
| 238 | 13.0 | 6.5 | 12.0 | 12.0 | 7.48 | 3.95 | 3.95 | 3.96 | 3.96 | 3.95 | 3.95 | 5, 6, 9, 10 |
| 239 | 14.0 | 3.06 | 3.13 | 13.0 | 7.52 | 3.02 | 3.02 | 3.02 | 3.0 | 3.0 | 3.0 | 8, 9, 10 |
| 240 | 10.0 | 4.06 | 10.0 | 10.0 | 7.11 | 3.95 | 5.2 | 3.52 | 3.5 | 3.5 | 3.5 | 8, 9, 10 |
| 241 | 14.0 | 5.25 | 6.0 | 11.0 | 7.17 | 3.45 | 3.7 | 3.44 | 3.28 | 3.28 | 3.28 | 8, 9, 10 |
| 242 | 13.0 | 6.25 | 11.0 | 11.0 | 7.23 | 3.95 | 4.2 | 3.94 | 3.61 | 3.61 | 3.61 | 8, 9, 10 |
| 243 | 14.0 | 3.88 | 4.25 | 12.0 | 7.3 | 3.2 | 3.2 | 3.2 | 3.11 | 3.11 | 3.11 | 8, 9, 10 |
| 244 | 12.0 | 4.88 | 11.0 | 11.0 | 7.11 | 3.95 | 3.95 | 3.7 | 3.61 | 3.61 | 3.61 | 8, 9, 10 |
| 245 | 14.0 | 5.5 | 6.5 | 12.0 | 7.3 | 3.45 | 3.45 | 3.45 | 3.28 | 3.28 | 3.28 | 8, 9, 10 |
| 246 | 13.0 | 6.5 | 12.0 | 12.0 | 7.48 | 3.95 | 3.95 | 3.95 | 3.61 | 3.61 | 3.61 | 8, 9, 10 |
| 247 | 14.0 | 3.31 | 3.5 | 13.0 | 7.55 | 3.08 | 3.08 | 3.08 | 3.04 | 3.04 | 3.04 | 8, 9, 10 |
| 248 | 11.0 | 4.31 | 11.0 | 11.0 | 7.23 | 3.95 | 4.45 | 3.58 | 3.54 | 3.54 | 3.54 | 8, 9, 10 |
| 249 | 14.0 | 5.5 | 6.5 | 12.0 | 7.36 | 3.45 | 3.45 | 3.45 | 3.45 | 3.33 | 3.33 | 9, 10 |
| 250 | 13.0 | 6.5 | 12.0 | 12.0 | 7.48 | 3.95 | 3.95 | 3.95 | 3.95 | 3.83 | 3.74 | 10 |

A.2 Two Dimensional Problems

In this section we present the results of our algorithms for two-dimensional ROFDPs with squares windows.

- $D = 2$
- $\mathcal{W} = \text{tile}(\text{hyperbar}((W, W)), \mathbf{0}_D, \mathbf{1}_D)$

The following subsections consider various choices for l and u .

A.2.1 Window Sorting

This section considers the problem of sorting the entire window. Hence $l = 0$ and $u = W \times W$.

| W | 0) | 1) | 2) | 3) | 4) | 5) | 6) | 7) | 8) | 9) | 10) | Best strategies |
|----|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|----------------------------------|
| 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 2 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 8.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 3 | 42.00 | 35.00 | 35.00 | 42.00 | 35.00 | 35.00 | 35.00 | 35.00 | 35.00 | 35.00 | 35.00 | 1, 2, 4, 5, 6, 7, 8, 9, 10 |
| 4 | 76.00 | 104.00 | 76.00 | 76.00 | 76.00 | 76.00 | 76.00 | 76.00 | 76.00 | 76.00 | 76.00 | 0, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 5 | 192.00 | 152.00 | 152.00 | 204.00 | 165.00 | 152.00 | 152.00 | 152.00 | 152.00 | 152.00 | 152.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 6 | 270.00 | 342.00 | 306.00 | 306.00 | 258.00 | 258.00 | 258.00 | 255.25 | 254.50 | 254.50 | 254.50 | 8, 9, 10 |
| 7 | 500.00 | 379.00 | 379.00 | 500.00 | 391.00 | 379.00 | 379.00 | 379.00 | 379.00 | 379.00 | 379.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 8 | 528.00 | 777.50 | 528.00 | 528.00 | 528.00 | 579.00 | 528.00 | 528.00 | 526.67 | 526.67 | 526.67 | 8, 9, 10 |
| 9 | 1048.00 | 744.00 | 711.00 | 1020.00 | 808.00 | 711.00 | 711.00 | 711.00 | 711.00 | 711.00 | 711.00 | 2, 5, 6, 7, 8, 9, 10 |
| 10 | 1062.00 | 1476.50 | 1294.00 | 1294.00 | 1032.75 | 976.00 | 976.00 | 944.63 | 944.83 | 940.71 | 940.71 | 9, 10 |
| 11 | 1648.00 | 1182.00 | 1182.00 | 1720.00 | 1308.50 | 1182.00 | 1182.00 | 1176.75 | 1176.75 | 1169.13 | 1169.13 | 9, 10 |
| 12 | 1580.00 | 2285.00 | 1852.00 | 1852.00 | 1429.00 | 1632.00 | 1429.00 | 1460.28 | 1438.56 | 1379.83 | 1379.83 | 9, 10 |
| 13 | 2694.00 | 1861.00 | 1951.00 | 2492.00 | 1889.00 | 1796.00 | 1796.00 | 1752.13 | 1734.75 | 1688.85 | 1688.85 | 9, 10 |
| 14 | 2544.00 | 3566.50 | 3486.00 | 2952.00 | 2305.00 | 2265.00 | 2265.00 | 2106.66 | 2099.21 | 2017.88 | 2017.88 | 9, 10 |
| 15 | 3646.00 | 2618.75 | 2529.50 | 3646.00 | 2716.50 | 2529.50 | 2529.50 | 2422.31 | 2386.64 | 2345.21 | 2339.65 | 10 |
| 16 | 3092.00 | 4914.25 | 3092.00 | 3092.00 | 3092.00 | 3337.00 | 3092.00 | 2866.41 | 2831.44 | 2627.45 | 2627.45 | 9, 10 |
| 17 | 5336.00 | 3738.00 | 3394.00 | 5064.00 | 3910.75 | 3544.00 | 3394.00 | 3231.00 | 3185.26 | 3047.08 | 3047.08 | 9, 10 |
| 18 | 4920.00 | 7015.00 | 5822.00 | 5822.00 | 4504.63 | 4268.00 | 4106.00 | 3768.94 | 3515.65 | 3513.47 | 3513.47 | 9, 10 |
| 19 | 6784.00 | 4813.50 | 4401.00 | 6812.00 | 5129.25 | 4401.00 | 4401.00 | 4147.88 | 4102.56 | 3947.67 | 3947.67 | 9, 10 |

A.2.2 Median Filtering

This section considers the problem of selecting the median of a window. Hence $l = u = \lfloor \frac{W \times W}{2} \rfloor$.

| W | 0) | 1) | 2) | 3) | 4) | 5) | 6) | 7) | 8) | 9) | 10) | Best strategies |
|----|---------|---------|---------|---------|---------|---------|---------|--------|--------|--------|--------|----------------------------------|
| 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 2 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 3 | 22.00 | 16.00 | 16.00 | 22.00 | 16.00 | 16.00 | 16.00 | 16.00 | 16.00 | 16.00 | 16.00 | 1, 2, 4, 5, 6, 7, 8, 9, 10 |
| 4 | 41.00 | 51.00 | 41.00 | 41.00 | 41.00 | 41.00 | 41.00 | 32.50 | 32.50 | 32.50 | 32.50 | 7, 8, 9, 10 |
| 5 | 106.00 | 53.00 | 53.00 | 88.00 | 72.50 | 53.00 | 53.00 | 53.00 | 53.00 | 53.00 | 53.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 6 | 151.00 | 120.50 | 145.00 | 145.00 | 108.50 | 108.50 | 108.50 | 76.50 | 75.75 | 75.75 | 75.75 | 8, 9, 10 |
| 7 | 290.00 | 113.50 | 113.50 | 290.00 | 188.50 | 113.50 | 113.50 | 105.75 | 105.75 | 105.75 | 105.75 | 7, 8, 9, 10 |
| 8 | 269.00 | 260.00 | 269.00 | 269.00 | 269.00 | 269.00 | 269.00 | 141.63 | 141.29 | 141.29 | 141.29 | 8, 9, 10 |
| 9 | 644.00 | 203.50 | 214.00 | 394.00 | 380.75 | 214.00 | 214.00 | 171.00 | 171.00 | 171.00 | 171.00 | 7, 8, 9, 10 |
| 10 | 587.00 | 464.00 | 533.00 | 533.00 | 451.50 | 334.50 | 334.50 | 209.63 | 209.21 | 209.21 | 209.21 | 8, 9, 10 |
| 11 | 990.00 | 270.50 | 270.50 | 826.00 | 553.25 | 270.50 | 270.50 | 232.75 | 232.75 | 232.75 | 232.75 | 7, 8, 9, 10 |
| 12 | 825.00 | 601.50 | 839.00 | 839.00 | 550.25 | 513.25 | 550.25 | 279.84 | 281.96 | 281.96 | 281.96 | 7 |
| 13 | 1682.00 | 445.50 | 573.00 | 1344.00 | 844.50 | 452.50 | 452.50 | 320.69 | 321.25 | 321.25 | 321.25 | 7 |
| 14 | 1425.00 | 988.00 | 1395.00 | 1683.00 | 1098.25 | 651.25 | 651.25 | 362.94 | 370.79 | 370.79 | 370.79 | 7 |
| 15 | 2242.00 | 577.75 | 492.75 | 2242.00 | 1342.25 | 492.75 | 492.75 | 404.38 | 404.38 | 404.38 | 404.38 | 7, 8, 9, 10 |
| 16 | 1553.00 | 1323.75 | 1553.00 | 1553.00 | 1553.00 | 997.00 | 1553.00 | 454.28 | 461.49 | 461.49 | 461.49 | 7 |
| 17 | 3422.00 | 908.00 | 967.00 | 1868.00 | 1932.63 | 914.50 | 967.00 | 499.94 | 507.23 | 507.23 | 507.23 | 7 |
| 18 | 2837.00 | 2025.50 | 2197.00 | 2197.00 | 2128.00 | 1208.00 | 1236.50 | 549.91 | 558.04 | 558.04 | 558.04 | 7 |
| 19 | 4278.00 | 1068.50 | 825.50 | 2824.00 | 2337.13 | 825.50 | 825.50 | 576.72 | 576.72 | 576.72 | 576.72 | 7, 8, 9, 10 |
| 20 | 2943.00 | 2366.00 | 2883.00 | 2883.00 | 2242.25 | 1285.25 | 1355.25 | 642.53 | 645.32 | 645.32 | 645.32 | 7 |

A.2.3 Minimum Filtering

This section considers the problem of selecting the minimum of a window. Hence $l = u = 0$.

| W | 0) | 1) | 2) | 3) | 4) | 5) | 6) | 7) | 8) | 9) | 10) | Best strategies |
|---|-------|------|------|------|------|------|------|------|------|------|------|----------------------------------|
| 1 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 2 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 2.00 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 3 | 4.00 | 3.00 | 3.00 | 4.00 | 3.00 | 3.00 | 3.00 | 3.00 | 3.00 | 3.00 | 3.00 | 1, 2, 4, 5, 6, 7, 8, 9, 10 |
| 4 | 4.00 | 5.00 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 0, 2, 3, 4, 5, 6, 7, 8, 9, 10 |
| 5 | 6.00 | 4.00 | 4.00 | 6.00 | 4.50 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 4.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 6 | 6.00 | 6.00 | 6.00 | 6.00 | 5.00 | 5.00 | 5.00 | 5.00 | 4.83 | 4.83 | 4.83 | 8, 9, 10 |
| 7 | 8.00 | 4.50 | 4.50 | 8.00 | 5.50 | 4.50 | 4.50 | 4.50 | 4.50 | 4.50 | 4.50 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 8 | 6.00 | 6.75 | 6.00 | 6.00 | 6.00 | 6.00 | 6.00 | 5.50 | 5.42 | 5.42 | 5.42 | 8, 9, 10 |
| 9 | 10.00 | 5.00 | 5.00 | 8.00 | 6.25 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 1, 2, 5, 6, 7, 8, 9, 10 |

| | | | | | | | | | | | | |
|----|-------|------|-------|-------|------|------|------|------|------|------|------|-------------------------|
| 10 | 8.00 | 7.50 | 8.00 | 8.00 | 6.50 | 6.00 | 6.00 | 6.00 | 6.00 | 6.00 | 6.00 | 5, 6, 7, 8, 9, 10 |
| 11 | 10.00 | 5.00 | 5.00 | 10.00 | 6.75 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 1, 2, 5, 6, 7, 8, 9, 10 |
| 12 | 8.00 | 7.50 | 8.00 | 8.00 | 6.50 | 6.50 | 6.50 | 6.00 | 5.83 | 5.83 | 5.83 | 8, 9, 10 |
| 13 | 12.00 | 5.50 | 6.00 | 10.00 | 7.00 | 5.50 | 5.50 | 5.50 | 5.33 | 5.33 | 5.33 | 8, 9, 10 |
| 14 | 10.00 | 8.25 | 10.00 | 10.00 | 7.50 | 6.50 | 6.50 | 6.50 | 6.17 | 6.17 | 6.17 | 8, 9, 10 |
| 15 | 12.00 | 5.38 | 5.25 | 12.00 | 7.75 | 5.25 | 5.25 | 5.25 | 5.25 | 5.25 | 5.25 | 2, 5, 6, 7, 8, 9, 10 |
| 16 | 8.00 | 8.13 | 8.00 | 8.00 | 8.00 | 7.00 | 8.00 | 6.25 | 6.25 | 6.25 | 6.25 | 7, 8, 9, 10 |
| 17 | 14.00 | 6.00 | 6.00 | 10.00 | 8.13 | 6.00 | 6.00 | 5.75 | 5.67 | 5.67 | 5.67 | 8, 9, 10 |
| 18 | 12.00 | 9.00 | 10.00 | 10.00 | 8.25 | 7.00 | 7.00 | 6.75 | 6.42 | 6.42 | 6.42 | 8, 9, 10 |
| 19 | 14.00 | 5.75 | 5.50 | 12.00 | 8.38 | 5.50 | 5.50 | 5.50 | 5.50 | 5.50 | 5.50 | 2, 5, 6, 7, 8, 9, 10 |
| 20 | 10.00 | 8.75 | 10.00 | 10.00 | 8.00 | 7.00 | 7.00 | 6.50 | 6.42 | 6.42 | 6.42 | 8, 9, 10 |

Appendix B

Sample solutions

In this appendix we present the details of some of the smaller ROF we found.

B.1 Representation of the ROFs

The ROFs in the rest of this appendix are presented in a table with a specific format.

The first two rows of such a table describe the windows and the desired interval of ranks from those windows that have to be calculated by the filter algorithm. The filter itself is described by the generator graph. The next two rows give the meta-period of this graph and the number of extrema operations per window.

The remaining rows of the table are used to describe the generator graph itself. Each row describes a single node of the graph. The first column contains the triple associated with the node. The next two columns contain the nodes that have edges leading towards the node in question. The final column contains the contribution of this node to the final cost of the entire filter.

B.2 One Dimensional Problems

This section contains some solutions for one dimensional problems.

B.2.1 Window Sorting

The first problem is that of sorting the entire window.

| | | | |
|-------------------|----------------|-----------|------|
| Window: | (■■■, 0, 1) | | |
| Desired Interval: | [0..2] | | |
| Meta-period: | 2 | | |
| Total Cost: | 5.00 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■, 0, 2) | (■■, 1, 2) | (■, 0, 2) | 2.00 |
| (■■■, 1, 2) | (■■, 1, 2) | (■, 1, 2) | 2.00 |

| | | | |
|-------------------|----------------|------------|------|
| Window: | (■■■■, 0, 1) | | |
| Desired Interval: | [0..3] | | |
| Meta-period: | 1 | | |
| Total Cost: | 8.00 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 1) | | | 0.00 |
| (■■, 0, 1) | (■, 0, 1) | (■, 0, 1) | 2.00 |
| (■■■■, 0, 1) | (■■, 0, 1) | (■■, 0, 1) | 6.00 |

| | | | |
|-------------------|----------------|------------|------|
| Window: | (■■■■■, 0, 1) | | |
| Desired Interval: | [0..4] | | |
| Meta-period: | 2 | | |
| Total Cost: | 12.00 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■■, 1, 2) | (■■, 1, 2) | (■■, 1, 2) | 3.00 |
| (■■■■■, 0, 2) | (■■■■, 1, 2) | (■, 0, 2) | 4.00 |
| (■■■■■, 1, 2) | (■■■■, 1, 2) | (■, 1, 2) | 4.00 |

| | | | |
|-------------------|----------------|--------------|------|
| Window: | (■■■■■■, 0, 1) | | |
| Desired Interval: | [0..5] | | |
| Meta-period: | 3 | | |
| Total Cost: | 16.00 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 3) | | | 0.00 |
| (■, 1, 3) | | | 0.00 |
| (■, 2, 3) | | | 0.00 |
| (■■, 0, 3) | (■, 0, 3) | (■, 1, 3) | 0.67 |
| (■■□□□■, 1, 3) | (■, 1, 3) | (■, 0, 3) | 0.67 |
| (■■□□, 2, 3) | (■, 2, 3) | (■, 2, 3) | 0.67 |
| (■■■■, 2, 3) | (■■□□, 2, 3) | (■, 0, 3) | 2.00 |
| (■■■■■■, 0, 3) | (■■■■, 2, 3) | (■, 0, 3) | 4.00 |
| (■■■■■■, 1, 3) | (■■□□□■, 1, 3) | (■■■■, 2, 3) | 4.00 |
| (■■■■■■, 2, 3) | (■■■■, 2, 3) | (■, 0, 3) | 4.00 |

| | | | |
|-------------------|-----------------|------------|------|
| Window: | (■■■■■■■, 0, 1) | | |
| Desired Interval: | [0..6] | | |
| Meta-period: | 4 | | |
| Total Cost: | 20.50 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 4) | | | 0.00 |
| (■, 1, 4) | | | 0.00 |
| (■, 2, 4) | | | 0.00 |
| (■, 3, 4) | | | 0.00 |
| (■■, 1, 4) | (■, 1, 4) | (■, 2, 4) | 0.50 |
| (■■, 3, 4) | (■, 3, 4) | (■, 0, 4) | 0.50 |
| (■■■■, 3, 4) | (■■, 1, 4) | (■■, 3, 4) | 1.50 |
| (■■■■■■, 1, 4) | (■■■■, 3, 4) | (■, 1, 4) | 3.00 |
| (■■■■■■, 3, 4) | (■■■■, 3, 4) | (■, 3, 4) | 3.00 |
| (■■■■■■■, 0, 4) | (■■■■■■, 1, 4) | (■, 0, 4) | 3.00 |
| (■■■■■■■, 2, 4) | (■■■■■■, 3, 4) | (■, 2, 4) | 3.00 |
| (■■■■■■■, 1, 4) | (■■■■■■, 1, 4) | (■, 3, 4) | 3.00 |
| (■■■■■■■, 3, 4) | (■■■■■■, 3, 4) | (■, 1, 4) | 3.00 |

| | | | |
|-------------------|------------------|--|--|
| Window: | (■■■■■■■■, 0, 1) | | |
| Desired Interval: | [0..7] | | |
| Meta-period: | 6 | | |
| Total Cost: | 25.33 | | |

| V_g | Incoming edges | | Cost |
|-----------------|----------------|----------------|------|
| (■, 0, 6) | | | 0.00 |
| (■, 1, 6) | | | 0.00 |
| (■, 2, 6) | | | 0.00 |
| (■, 3, 6) | | | 0.00 |
| (■, 4, 6) | | | 0.00 |
| (■, 5, 6) | | | 0.00 |
| (■■, 0, 6) | (■, 0, 6) | (■, 1, 6) | 0.33 |
| (■■, 3, 6) | (■, 3, 6) | (■, 4, 6) | 0.33 |
| (■■, 2, 6) | (■, 2, 6) | (■, 3, 6) | 0.33 |
| (■■, 5, 6) | (■, 5, 6) | (■, 0, 6) | 0.33 |
| (■□□□□■, 1, 6) | (■, 1, 6) | (■, 2, 6) | 0.33 |
| (■□□□□■, 4, 6) | (■, 4, 6) | (■, 5, 6) | 0.33 |
| (■■■, 2, 6) | (■■, 3, 6) | (■, 2, 6) | 0.67 |
| (■■■, 5, 6) | (■■, 0, 6) | (■, 5, 6) | 0.67 |
| (■■■■■, 2, 6) | (■■■, 2, 6) | (■■■, 5, 6) | 2.00 |
| (■■■■■, 5, 6) | (■■■, 2, 6) | (■■■, 5, 6) | 2.00 |
| (■■■■■■■, 0, 6) | (■■■■■, 2, 6) | (■, 0, 6) | 3.00 |
| (■■■■■■■, 3, 6) | (■■■■■■, 5, 6) | (■■, 3, 6) | 3.00 |
| (■■■■■■■, 1, 6) | (■□□□□■, 1, 6) | (■■■■■■, 2, 6) | 3.00 |
| (■■■■■■■, 4, 6) | (■□□□□■, 4, 6) | (■■■■■■, 5, 6) | 3.00 |
| (■■■■■■■, 2, 6) | (■■■■■■, 2, 6) | (■, 2, 6) | 3.00 |
| (■■■■■■■, 5, 6) | (■■■■■■, 5, 6) | (■, 5, 6) | 3.00 |

| | |
|-------------------|-------------------|
| Window: | (■■■■■■■■■, 0, 1) |
| Desired Interval: | [0..8] |
| Meta-period: | 2 |
| Total Cost: | 29.00 |

| V_g | Incoming edges | | Cost |
|-----------------|----------------|--------------|------|
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■, 1, 2) | (■■, 1, 2) | (■■, 1, 2) | 3.00 |
| (■■■■, 1, 2) | (■■■, 1, 2) | (■■■■, 1, 2) | 9.00 |
| (■■■■■■, 0, 2) | (■■■■■■, 1, 2) | (■, 0, 2) | 8.00 |
| (■■■■■■■, 1, 2) | (■■■■■■, 1, 2) | (■, 1, 2) | 8.00 |

B.2.2 Median Filtering

This is the median filtering problem.

| | |
|-------------------|-------------|
| Window: | (■■■, 0, 1) |
| Desired Interval: | [1..1] |
| Meta-period: | 2 |
| Total Cost: | 3.00 |

| V_g | Incoming edges | | Cost |
|-------------|----------------|-----------|------|
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■, 0, 2) | (■■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■, 1, 2) | (■■, 1, 2) | (■, 1, 2) | 1.00 |

| | |
|-------------------|---------------|
| Window: | (■■■■■, 0, 1) |
| Desired Interval: | [2..2] |
| Meta-period: | 2 |
| Total Cost: | 5.00 |

| V_g | Incoming edges | | Cost |
|--------------|----------------|------------|------|
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■, 1, 2) | (■■, 1, 2) | (■■, 1, 2) | 2.00 |
| (■■■■, 0, 2) | (■■■■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■■, 1, 2) | (■■■■, 1, 2) | (■, 1, 2) | 1.00 |

| | | | |
|-------------------|------------------|------------|------|
| Window: | (■■■■■■■■, 0, 1) | | |
| Desired Interval: | [3..3] | | |
| Meta-period: | 4 | | |
| Total Cost: | 8.50 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 4) | | | 0.00 |
| (■, 1, 4) | | | 0.00 |
| (■, 2, 4) | | | 0.00 |
| (■, 3, 4) | | | 0.00 |
| (■■, 1, 4) | (■, 1, 4) | (■, 2, 4) | 0.50 |
| (■■, 3, 4) | (■, 3, 4) | (■, 0, 4) | 0.50 |
| (■■■■, 3, 4) | (■■, 1, 4) | (■■, 3, 4) | 1.50 |
| (■■■■■■, 1, 4) | (■■■■, 3, 4) | (■■, 1, 4) | 2.00 |
| (■■■■■■, 3, 4) | (■■■■, 3, 4) | (■■, 3, 4) | 2.00 |
| (■■■■■■■■, 0, 4) | (■■■■■■■■, 1, 4) | (■, 0, 4) | 0.50 |
| (■■■■■■■■, 2, 4) | (■■■■■■■■, 3, 4) | (■, 2, 4) | 0.50 |
| (■■■■■■■■, 1, 4) | (■■■■■■■■, 1, 4) | (■, 3, 4) | 0.50 |
| (■■■■■■■■, 3, 4) | (■■■■■■■■, 3, 4) | (■, 1, 4) | 0.50 |

| | | | |
|--------------------|--------------------|--------------|------|
| Window: | (■■■■■■■■■■, 0, 1) | | |
| Desired Interval: | [4..4] | | |
| Meta-period: | 2 | | |
| Total Cost: | 10.00 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■■, 1, 2) | (■■, 1, 2) | (■■, 1, 2) | 3.00 |
| (■■■■■■■■, 1, 2) | (■■■■, 1, 2) | (■■■■, 1, 2) | 4.00 |
| (■■■■■■■■■■, 0, 2) | (■■■■■■■■■■, 1, 2) | (■, 0, 2) | 1.00 |
| (■■■■■■■■■■, 1, 2) | (■■■■■■■■■■, 1, 2) | (■, 1, 2) | 1.00 |

B.2.3 Minimum Filtering

This is the minimum filtering problem.

| | | | |
|-------------------|----------------|-----------|------|
| Window: | (■■■■, 0, 1) | | |
| Desired Interval: | [0..0] | | |
| Meta-period: | 2 | | |
| Total Cost: | 1.50 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■■■, 0, 2) | (■■, 1, 2) | (■, 0, 2) | 0.50 |
| (■■■■, 1, 2) | (■■, 1, 2) | (■, 1, 2) | 0.50 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 0.50 |

| | | | |
|-------------------|----------------|------------|------|
| Window: | (■■■■■, 0, 1) | | |
| Desired Interval: | [0..0] | | |
| Meta-period: | 1 | | |
| Total Cost: | 2.00 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 1) | | | 0.00 |
| (■■■■, 0, 1) | (■■, 0, 1) | (■■, 0, 1) | 1.00 |
| (■■, 0, 1) | (■, 0, 1) | (■, 0, 1) | 1.00 |

| | | | |
|-------------------|----------------|--|--|
| Window: | (■■■■■■, 0, 1) | | |
| Desired Interval: | [0..0] | | |
| Meta-period: | 2 | | |
| Total Cost: | 2.00 | | |

| V_g | Incoming edges | | Cost |
|----------------|----------------|------------|------|
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■■■■■, 0, 2) | (■■■■■■, 1, 2) | (■, 0, 2) | 0.50 |
| (■■■■■■, 1, 2) | (■■■■■■, 1, 2) | (■, 1, 2) | 0.50 |
| (■■■■■■, 1, 2) | (■■, 1, 2) | (■■, 1, 2) | 0.50 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 0.50 |

| | |
|-------------------|-----------------|
| Window: | (■■■■■■■, 0, 1) |
| Desired Interval: | [0..0] |
| Meta-period: | 2 |
| Total Cost: | 2.50 |

| V_g | Incoming edges | | Cost |
|-----------------|-----------------|------------|------|
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■■■■■■, 0, 2) | (■■■■■■■, 1, 2) | (■, 0, 2) | 0.50 |
| (■■■■■■■, 1, 2) | (■■■■■■■, 1, 2) | (■, 0, 2) | 0.50 |
| (■■■■■■■, 1, 2) | (■■■■■■, 0, 2) | (■, 1, 2) | 0.50 |
| (■■■■■, 0, 2) | (■■, 0, 2) | (■■, 0, 2) | 0.50 |
| (■■, 0, 2) | (■, 0, 2) | (■, 1, 2) | 0.50 |

| | |
|-------------------|------------------|
| Window: | (■■■■■■■■, 0, 1) |
| Desired Interval: | [0..0] |
| Meta-period: | 4 |
| Total Cost: | 2.25 |

| V_g | Incoming edges | | Cost |
|------------------|------------------|------------|------|
| (■, 0, 4) | | | 0.00 |
| (■, 1, 4) | | | 0.00 |
| (■, 2, 4) | | | 0.00 |
| (■, 3, 4) | | | 0.00 |
| (■■■■■■■■, 0, 4) | (■■■■■■■■, 1, 4) | (■, 0, 4) | 0.25 |
| (■■■■■■■■, 2, 4) | (■■■■■■■■, 3, 4) | (■, 2, 4) | 0.25 |
| (■■■■■■■■, 1, 4) | (■■■■■■■■, 1, 4) | (■, 3, 4) | 0.25 |
| (■■■■■■■■, 3, 4) | (■■■■■■■■, 3, 4) | (■, 1, 4) | 0.25 |
| (■■■■■■■■, 1, 4) | (■■■■■■, 3, 4) | (■, 1, 4) | 0.25 |
| (■■■■■■■■, 3, 4) | (■■■■■■, 3, 4) | (■, 3, 4) | 0.25 |
| (■■, 1, 4) | (■, 1, 4) | (■, 2, 4) | 0.25 |
| (■■, 3, 4) | (■, 3, 4) | (■, 0, 4) | 0.25 |
| (■■■■■, 3, 4) | (■■, 1, 4) | (■■, 3, 4) | 0.25 |

| | |
|-------------------|-------------------|
| Window: | (■■■■■■■■■, 0, 1) |
| Desired Interval: | [0..0] |
| Meta-period: | 4 |
| Total Cost: | 2.75 |

| V_g | Incoming edges | | Cost |
|-------------------|-------------------|------------|------|
| (■, 0, 4) | | | 0.00 |
| (■, 1, 4) | | | 0.00 |
| (■, 2, 4) | | | 0.00 |
| (■, 3, 4) | | | 0.00 |
| (■■■■■■■■■, 0, 4) | (■■■■■■■■■, 1, 4) | (■, 0, 4) | 0.25 |
| (■■■■■■■■■, 2, 4) | (■■■■■■■■■, 3, 4) | (■, 2, 4) | 0.25 |
| (■■■■■■■■■, 1, 4) | (■■■■■■■■■, 1, 4) | (■, 0, 4) | 0.25 |
| (■■■■■■■■■, 3, 4) | (■■■■■■■■■, 3, 4) | (■, 2, 4) | 0.25 |
| (■■■■■■■■■, 1, 4) | (■■■■■■■■, 2, 4) | (■, 1, 4) | 0.25 |
| (■■■■■■■■■, 3, 4) | (■■■■■■■■, 0, 4) | (■, 3, 4) | 0.25 |
| (■■■■■■■■, 0, 4) | (■■■■■■, 2, 4) | (■■, 0, 4) | 0.25 |
| (■■■■■■■■, 2, 4) | (■■■■■■, 2, 4) | (■, 2, 4) | 0.25 |
| (■■, 0, 4) | (■, 0, 4) | (■, 1, 4) | 0.25 |
| (■■, 2, 4) | (■, 2, 4) | (■, 3, 4) | 0.25 |
| (■■■■■, 2, 4) | (■■, 0, 4) | (■■, 2, 4) | 0.25 |

| | | | |
|-------------------|------------------|--------------|------|
| Window: | (■■■■■■■■, 0, 1) | | |
| Desired Interval: | [0..0] | | |
| Meta-period: | 2 | | |
| Total Cost: | 2.50 | | |
| V_g | Incoming edges | | Cost |
| (■, 0, 2) | | | 0.00 |
| (■, 1, 2) | | | 0.00 |
| (■■■■■■■■, 0, 2) | (■■■■■■■■, 1, 2) | (■, 0, 2) | 0.50 |
| (■■■■■■■■, 1, 2) | (■■■■■■■■, 1, 2) | (■, 1, 2) | 0.50 |
| (■■■■■■■■, 1, 2) | (■■■■, 1, 2) | (■■■■, 1, 2) | 0.50 |
| (■■■■, 1, 2) | (■■, 1, 2) | (■■, 1, 2) | 0.50 |
| (■■, 1, 2) | (■, 1, 2) | (■, 0, 2) | 0.50 |

B.3 Two Dimensional Problems






















































In this section we present some solutions for two dimensional problems.

We extend our notation for a set of coordinates to two dimensions by drawing matrices of squares. The upper left square of such a matrix corresponds to the coordinate (0,0) the square to the right corresponds to (1,0), etc.

B.3.1 Window Sorting
















The first problem is that of sorting the entire window.
















































| | | | |
|------------------------|-----------------------|-----------------------|------|
| Window: | (■■■, (0, 0), (1, 1)) | | |
| Desired Interval: | [0..8] | | |
| Meta-period: | (2,2) | | |
| Total Cost: | 35.00 | | |
| V_g | Incoming edges | | Cost |
| (■, (0, 0), (2, 2)) | | | 0.00 |
| (■, (0, 1), (2, 2)) | | | 0.00 |
| (■, (1, 0), (2, 2)) | | | 0.00 |
| (■, (1, 1), (2, 2)) | | | 0.00 |
| (■, (0, 1), (2, 2)) | (■, (0, 1), (2, 2)) | (■, (0, 0), (2, 2)) | 0.50 |
| (■, (1, 1), (2, 2)) | (■, (1, 1), (2, 2)) | (■, (1, 0), (2, 2)) | 0.50 |
| (■■, (0, 0), (2, 2)) | (■, (0, 0), (2, 2)) | (■■, (0, 1), (2, 2)) | 1.00 |
| (■■, (0, 1), (2, 2)) | (■, (0, 1), (2, 2)) | (■■, (0, 1), (2, 2)) | 1.00 |
| (■■, (1, 0), (2, 2)) | (■, (1, 0), (2, 2)) | (■■, (1, 1), (2, 2)) | 1.00 |
| (■■, (1, 1), (2, 2)) | (■, (1, 1), (2, 2)) | (■■, (1, 1), (2, 2)) | 1.00 |
| (■■■, (1, 0), (2, 2)) | (■■, (1, 0), (2, 2)) | (■■■, (0, 0), (2, 2)) | 3.00 |
| (■■■, (1, 1), (2, 2)) | (■■, (1, 1), (2, 2)) | (■■■, (0, 1), (2, 2)) | 3.00 |
| (■■■■, (0, 0), (2, 2)) | (■■■, (1, 0), (2, 2)) | (■■■, (0, 0), (2, 2)) | 6.00 |
| (■■■■, (0, 1), (2, 2)) | (■■■, (1, 1), (2, 2)) | (■■■, (0, 1), (2, 2)) | 6.00 |
| (■■■■, (1, 0), (2, 2)) | (■■■, (1, 0), (2, 2)) | (■■■, (1, 0), (2, 2)) | 6.00 |
| (■■■■, (1, 1), (2, 2)) | (■■■, (1, 1), (2, 2)) | (■■■, (1, 1), (2, 2)) | 6.00 |

| | | | |
|--|--|--|-------|
| Window: |  $(0, 0), (1, 1)$ | | |
| Desired Interval: | [0..24] | | |
| Meta-period: | (2,2) | | |
| Total Cost: | 152.00 | | |
| V_g | Incoming edges | | Cost |
|  , (0, 0), (2, 2) | | | 0.00 |
|  , (0, 1), (2, 2) | | | 0.00 |
|  , (1, 0), (2, 2) | | | 0.00 |
|  , (1, 1), (2, 2) | | | 0.00 |
|  , (0, 1), (2, 2) |  , (0, 1), (2, 2) |  , (0, 0), (2, 2) | 0.50 |
|  , (1, 1), (2, 2) |  , (1, 1), (2, 2) |  , (1, 0), (2, 2) | 0.50 |
|  , (0, 1), (2, 2) |  , (0, 1), (2, 2) |  , (0, 1), (2, 2) | 1.50 |
|  , (1, 1), (2, 2) |  , (1, 1), (2, 2) |  , (1, 1), (2, 2) | 1.50 |
|  , (0, 0), (2, 2) |  , (0, 0), (2, 2) |  , (0, 1), (2, 2) | 2.00 |
|  , (0, 1), (2, 2) |  , (0, 1), (2, 2) |  , (0, 1), (2, 2) | 2.00 |
|  , (1, 0), (2, 2) |  , (1, 0), (2, 2) |  , (1, 1), (2, 2) | 2.00 |
|  , (1, 1), (2, 2) |  , (1, 1), (2, 2) |  , (1, 1), (2, 2) | 2.00 |
|  , (1, 0), (2, 2) |  , (1, 0), (2, 2) |  , (0, 0), (2, 2) | 6.50 |
|  , (1, 1), (2, 2) |  , (1, 1), (2, 2) |  , (0, 1), (2, 2) | 6.50 |
|  , (1, 0), (2, 2) |  , (1, 0), (2, 2) |  , (1, 0), (2, 2) | 17.50 |
|  , (1, 1), (2, 2) |  , (1, 1), (2, 2) |  , (1, 1), (2, 2) | 17.50 |
|  , (0, 0), (2, 2) |  , (1, 0), (2, 2) |  , (0, 0), (2, 2) | 23.00 |
|  , (0, 1), (2, 2) |  , (1, 1), (2, 2) |  , (0, 1), (2, 2) | 23.00 |
|  , (1, 0), (2, 2) |  , (1, 0), (2, 2) |  , (1, 0), (2, 2) | 23.00 |
|  , (1, 1), (2, 2) |  , (1, 1), (2, 2) |  , (1, 1), (2, 2) | 23.00 |

B.3.2 Median Filtering

This is the median filtering problem.

| | | | |
|--|--|--|------|
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 1), (2, 2) | 0.25 |
|  (0, 1), (2, 2) |  (0, 1), (2, 2) |  (0, 0), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 0), (2, 2) | 0.25 |
|  (1, 0), (2, 2) |  (1, 0), (2, 2) |  (0, 0), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (0, 1), (2, 2) | 0.25 |

| | | | |
|--|--|---|------|
| Window: |  (0, 0), (1, 1) | | |
| Desired Interval: | [0..0] | | |
| Meta-period: | (2,2) | | |
| Total Cost: | 4.00 | | |
| V_g | Incoming edges | | Cost |
|  (0, 0), (2, 2) | | | 0.00 |
|  (0, 1), (2, 2) | | | 0.00 |
|  (1, 0), (2, 2) | | | 0.00 |
|  (1, 1), (2, 2) | | | 0.00 |
|  (0, 0), (2, 2) |  (1, 0), (2, 2) |  (0, 0), (2, 2) | 0.25 |
|  (0, 1), (2, 2) |  (1, 1), (2, 2) |  (0, 1), (2, 2) | 0.25 |
|  (1, 0), (2, 2) |  (1, 0), (2, 2) |  (1, 0), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 1), (2, 2) | 0.25 |
|  (0, 0), (2, 2) |  (0, 0), (2, 2) |  (0, 1), (2, 2) | 0.25 |
|  (0, 1), (2, 2) |  (0, 1), (2, 2) |  (0, 1), (2, 2) | 0.25 |
|  (1, 0), (2, 2) |  (1, 0), (2, 2) |  (1, 1), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 1), (2, 2) | 0.25 |
|  (0, 1), (2, 2) |  (0, 1), (2, 2) |  (0, 1), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 1), (2, 2) | 0.25 |
|  (0, 1), (2, 2) |  (0, 1), (2, 2) |  (0, 0), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 0), (2, 2) | 0.25 |
|  (1, 0), (2, 2) |  (1, 0), (2, 2) |  (1, 0), (2, 2) | 0.25 |
|  (1, 1), (2, 2) |  (1, 1), (2, 2) |  (1, 1), (2, 2) | 0.25 |

| | | | |
|--|--|--|------|
| $\left(\begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{array}, (1, 0), (2, 2) \right)$ | $\left(\begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \end{array}, (1, 0), (2, 2) \right)$ | $\left(\begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{array}, (0, 0), (2, 2) \right)$ | 0.25 |
| $\left(\begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{array}, (1, 1), (2, 2) \right)$ | $\left(\begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \end{array}, (1, 1), (2, 2) \right)$ | $\left(\begin{array}{c} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{array}, (0, 1), (2, 2) \right)$ | 0.25 |