Karlsruhe Institute of Technology

Finding maximum-weight consistent digitally convex regions

Studienarbeit
von

Moritz v. Looz

An der Fakultät für Informatik
Institut für Theoretische Informatik

Gutachterin:	Prof. Dr. Dorothea Wagner
Betreuender Mitarbeiter:	Dr. rer. nat. Martin Nöllenburg
Externer Betreuer:	Prof. Takeshi Tokuyama

Bearbeitungszeit: 01. Juni 2010 – 07. Februar 2011

Acknowledgements

I wish to thank Professor Tokuyama and the Tohoku University for their hospitality and for pointing out the open question this work addresses. Further, I wish to express my thanks to Professor Wagner for setting up the partnership between the Tohoku University and the University of Karlsruhe and making my stay in Sendai possible. It is impossible to thank Dr. Martin Nöllenburg enough for his advice and his patience. Finally, I would like to thank Natsuda Kaothanthong, who asked just the right kind of questions during the concept phase and gave several ideas for the second algorithm.

Declaration

I declare I have written this thesis by myself and have not used any sources or assistance other than those listed.

Karlsruhe, 14. Februar 2011

Abstract

We describe an algorithm to find maximum-weight digitally convex regions in gray-scale images. After Christ et al. [CPS10] gave a system for consistent digital line systems, the question how to transfer the concept of convexity to digital regions naturally appeared. The regions computed using this definition are weight-maximal, the computational complexity of our algorithm amounts to $O(n^5 \cdot \log n)$. With additional input data constraining the possible solution, this bound can be improved to $O(n^3 \cdot \log n)$. We further describe a hybrid approximation algorithm that first generates suitable constraints by executing the $O(n^5 \cdot \log n)$-algorithm on a downscaled version of the input image and then uses these constraints to achieve the $O(n^3 \cdot \log n)$ bound.

Zusammenfassung

Wir beschreiben und implementieren einen Algorithmus, um digital-konvexe Regionen maximalen Gewichts in Graustufenbildern zu finden. Nachdem Christ et al. [CPS10] ein System für konsistente digitale Liniensegmente konstruierten, stellte sich die Frage nach entsprechenden digital-konvexen Regionen. Die vom Algorithmus berechneten Regionen sind gewichtsmaximal, der Zeitaufwand liegt bei $O(n^5 \cdot \log n)$. Mit zusätzlichen Eingabedaten zur Einschränkung des Suchraums kann diese Schranke zu $O(n^3 \cdot \log n)$ verbessert werden. Weiterhin beschreiben wir einen hybriden Approximationsalgorithmus, der sinnvolle Einschränkungen erzeugt indem der $O(n^5 \cdot \log n)$-Algorithmus auf einer verkleinerten Version des Eingabebildes ausgeführt wird und anschließend diese Einschränkungen verwendet um die $O(n^3 \cdot \log n)$-Schranke zu erreichen.

Contents

1. Introduction **1**

1.1. Motivation 1

1.2. Related work 1

1.3. Outline 2

2. Preliminaries **3**

2.1. Terminology 3

2.2. Definitions 3

2.2.1. Line segment definitions 3

2.2.2. Convexity requirements 4

2.3. Slopes 5

2.3.1. Grid paths in images 5

2.3.2. Total orders 6

2.3.3. Non-convexity witness paths 7

2.3.4. Slope constraints 8

2.3.4.1. Ascending upper boundary 8

2.3.4.2. Descending upper boundary 8

2.3.4.3. Descending lower boundary 9

2.3.4.4. Ascending lower boundary 10

2.4. Region definitions 10

2.4.1. Potential 10

2.4.2. Weight 10

3. Algorithm **11**

3.1. Outline 11

3.2. Input and output data 11

3.3. Complete algorithm 12

3.3.1. Region definitions 12

3.3.2. Data structure 12

3.3.3. Overview 12

3.4. Create new regions in column i 13

3.5. Extend existing regions to column $i + 1$ 14

3.5.1. Expanding boundary 14

3.5.2. Contracting boundary 15

3.5.3. Invariants 15

3.6. Trickle 16

3.6.1. Invariants 16

3.7. Update slope constraints 17

3.8. Anomalies 17

3.9. Consolidate constructed regions 18

3.10. Reconstructing the optimal region 19

3.11. Correctness 19
3.11.1. Returned region is convex 19
3.11.2. Returned region is weight-maximal 20
3.12. Computational and space complexity 20
3.12.1. Space complexity 20
3.12.2. Naive implementation 20
3.12.3. Improvements 20
3.13. Baseline-convex algorithm 21
3.13.1. Introduction 21
3.13.2. Additional input 21
3.13.3. Half-convex regions 22
3.13.4. Data structure 23
3.13.5. Algorithm 23
3.13.6. Space complexity 23
3.13.7. Computational complexity 23
3.13.8. Limitations 24
3.14. Hybrid algorithm 24
3.14.1. Computational complexity 25
3.14.2. Iterating 25

4. Evaluation **27**
4.1. Anomalies 27
4.2. Experimental setup 27
4.3. Experimental results 28
4.3.1. Limitations 28
4.3.2. Output 28
4.3.3. Time and occupancy measurements 30
4.3.4. Approximation quality 30
4.4. Possible optimizations 31
4.4.1. Choice of data structure 31
4.4.2. Remove obsolete old regions 33
4.4.3. Remove dominated regions 33
4.4.4. Possible parallelizations 33
4.4.5. Implementation details 33

5. Conclusion **35**
5.1. Findings 35
5.2. Future work 35
5.2.1. Performance improvements 35
5.2.2. Building pyramids 35

Bibliography **37**

Appendix **39**
A. Source code 39

1. Introduction

1.1. Motivation

The question how to find a good digital representation of a Euclidean line has long been a problem in digital geometry and image recognition. Rasterizing an analogue line and choosing the pixels as nodes in a grid is straightforward and gives a grid path close to the Euclidean line. Unfortunately, it does not yield the same consistency properties valid for Euclidean line segments. For example, the intersection of two rasterized lines might not be connected. Some alternative systems for digital lines were compiled by Klette and Rosenfeld [KR04]. Convexity in pixel grids was equally not consistent, as two digitized conventionally convex shapes might have a non-convex intersection. However, finding convex shapes as a problem of image segmentation has many applications. One example is medical imaging, where knots, lumps or even tumors are often convex.

Christ et al. [CPS10] gave a system of consistent digital line segments, which can be used to define convexity on a grid. For image recognition purposes, we are interested in weight-maximal digitally convex regions in a two dimensional grid, using the line segment definition of Christ et al. [CPS10].

Finding the optimal digitally convex region in an $N \times N$-image turned out to be computationally very expensive with a time complexity of $O(n^5 \log n)$. When a line L from the leftmost contained pixel to the rightmost contained pixel is given, the complexity improves to $O(n^3 \log n)$. We call this constrained set of regions *baseline-convex* on L.

1.2. Related work

How to find shapes with certain properties in images is a known problem in computer science. In medical imaging, convex or similar shapes are often of interest. Kobatake and Murakami [KM96] described a method to find tumors by searching round convex shapes and presented the iris filter to achieve this.

Different approaches for digital straightness were developed, over which Klette and Rosenfeld [KR04] gave a survey. Chun et al. [CKNT08] described a system of consistent digital rays, which achieved the desired consistency properties for some paths. Using the digital rays, they gave an algorithm to efficiently find star-shaped regions in images. The digitally convex regions we discuss are a proper subset of the family of star-shaped regions. The family of regions bounded by two x-monotone curves are another superset of the digitally convex regions and were considered by Asano et al. [ACKT01], who proved the NP-completeness of finding a weight-maximal connected shape within an image. Christ et al [CPS10] expanded upon the work of Chun et al. [CKNT08] and constructed a system of digital line segments which guarantee a small Hausdorff distance to the Euclidean line segment. With this foundation, a consistent definition of digitally convex regions appears. Our work draws from the definition of digital line segments and computes the maximum-weight digitally convex region in input images. This work is primarily based on the work of Christ et al [CPS10]. Our main dynamic programming algorithm is similar to and inspired by one described by Chun et al [CST03].

1.3. Outline

We present the theoretical groundwork in Chapter 2, proceeding from the line segment definitions to convexity requirements of digital regions in a grid. We transfer the notion of slopes to digital line segments, resulting in the slope constraints, which are central to our approach. The definitions of the computed regions conclude the chapter.

In Chapter 3, we present the main algorithm to find maximum-weight digitally convex regions. After evaluating the space and time complexity, we present a second algorithm which compromises optimality for the sake of performance. A hybrid version allows fine-grained adjustments to balance performance against quality.

We evaluate the performance and results of the algorithms in Chapter 4. After discussing the behavior and output of the implementation, we give some possible improvements.

Chapter 5 concludes our work and presents further open questions.

2. Preliminaries

In this chapter, the boundaries of digitally convex regions will be discussed, as they are fundamental for the algorithm proper presented in Chapter 3. We start with the line segment properties defined by Christ et al. [CPS10], construct a notion of slopes on a pixel grid and conclude with the constraints defining the boundaries of a digitally convex region and their construction.

2.1. Terminology

Since this thesis uses different orders on $\mathbb{Z}$, some notations may not be unambiguous any more. Unless mentioned otherwise, min, max, $\leq$ and related symbols are assumed to be based on the natural order. If a minimum is based on another total order $\prec$, it will be marked as $\min^{\prec}$. The same holds for the maximum $\max^{\prec}$ and other notations based on orders. Since the coordinate sum, or *metric* of a pixel position p is central to the construction of line segments, we define $m(p) := (p_x + p_y)$ as an abbreviation.

2.2. Definitions

2.2.1. Line segment definitions

A line segment is a grid path between two points. Given points $p = (p_x, p_y)$, $q = (q_x, q_y)$ on a two dimensional grid and a total order $\prec$ on $\mathbb{Z}$, the line segment between them is contained in the bounding box spanned by p and q. We construct the line segment $S(p, q)$ as a path on the two-dimensional grid, where neighboring nodes are connected. Each node has four neighbors.

Without loss of generality, we assume $p_x < q_x$ and construct the path $S(p, q)$ from left to right. Let $r = (r_x, r_y)$ be the current node of the construction. $p_x \leq r_x \leq q_x$ holds.

Christ et al. [CPS10] divided two cases:

- case $p_y \leq q_y$:
  - if $m(r)$ is among the $(q_y - p_y)$ greatest elements in $[m(p), m(q))$ according to $\prec$, we continue the path with the upper neighbor of r.
  - if $m(r)$ is not among the $(q_y - p_y)$ greatest elements according to $\prec$, we continue the path with the right neighbor of r.

- case $p_y > q_y$:
  $S(p, q)$ is the mirror reflection of $S((-q_x, q_y), (-p_x, p_y))$ along the y-axis.

Continuing a line segment with the right neighbor of the current rightmost point will be called *going to the right*, adding the upper neighbor will be called *going up*. The other directions follow analogously for the mirrored version.

This system of line segments has several interesting properties:

1. Grid path property: For all $p, q \in \mathbb{Z}^2$, $S(p, q)$ is the vertex set of a path from p to q in the grid graph.
2. Symmetry property: For all $p, q \in \mathbb{Z}^2$, $S(p, q) = S(q, p)$
3. Subsegment property: For all $p, q \in \mathbb{Z}^2$ and every $r \in S(p, q)$, we have $S(p, r) \subseteq S(p, q)$
4. Prolongation property: For all $p, q \in \mathbb{Z}^2$, there exists $r \in \mathbb{Z}^2$, such that $r \notin S(p, q)$ and $S(p, q) \subset S(p, r)$.
5. Monotonicity property: If both $p, q \in \mathbb{Z}^2$ lie on a line that is either horizontal or vertical, then the whole segment $S(p, q)$ belongs to this line.

2.2.2. Convexity requirements

For a region $R \subset \mathbb{Z}^2$ to be convex, every line segment between any two points has to be contained within the region. If the line segment $S(p, q)$ between two points p, $q \in R$ is not contained in R, we call $S(p, q)$ a *non-convexity witness path*. In a region R containing k pixels, at most $(k^2)/2$ possible witness paths exist. We call a point in R having at least one neighbor not contained in R a *border point*.

Within the set of border points, we define two subsets:

- The *upper boundary* of a region R consists of the pixels in R whose upper neighbor is not contained in R.
- The *lower boundary* of a region R consists of the pixels in R whose lower neighbor is not contained in R.

These boundaries may overlap if the height of R is only one pixel at some columns. The union of upper and lower boundary may not contain all border points of R, but uniquely defines R. Given upper boundary ub and lower boundary lb, a point p is contained in the resulting region iff there are points $u \in ub$ and $l \in lb$ so that $q_x = u_x = l_x$ and $l_y \leq q_y \leq u_y$. The boundaries are not connected in most cases. We define a *boundary path* of a region R as a connected set of pixels in R, each having one neighbor not contained in R. The *upper boundary path* consists of the upper boundary and those border points necessary to connect them. Similarly, the *lower boundary path* consists of the lower boundary and the border points necessary to connect them.

When deciding whether a given region is convex, we only have to consider possible non-convexity witness paths involving border points. Assume a non-convex grid region R and two points a, b contained in R, so that their line segment $S(a, b)$ is not contained in R. Then a point $r \in S(a, b)$ exists with $r \notin R$. $S(a, r) \subset S(a, b)$ due to the subsegment property mentioned in Section 2.2.1.

As line segments are connected, a border point t is contained in $S(a, r)$, and a border point s is contained in $S(r, b)$. The line segment $S(t, s)$ is a subsegment of $S(a, b)$, since t and s are both contained in $S(a, b)$. Due to the subsegment property mentioned in Section 2.2.1, $S(t, s)$ contains every point of $S(a, b)$ between t and s, including r. Since it includes $r \notin R$, and $t, s \in R$, $S(t, s)$ is a non-convexity witness path for R. We have now constructed a non-convexity witness path between two border points for an arbitrary non-convexity witness path. Hence, it is sufficient to consider only border points of R, since every non-convexity witness path of R contains a witness path between two border points.

Due to the monotonicity property, non-convexity witness paths cannot go around a region R. Every non-convexity witness path between a point u of the upper boundary path and a point l of the lower boundary path will have to pass through R and contain at least one other boundary path point t to reach u respectively l from outside R. The border point t must be contained in either the upper or lower boundary path, resulting in a non-convexity witness path within the upper or lower boundary path. Hence, it is sufficient to concentrate on pairs of points within the upper respective lower boundary path.

Some border points are not contained in either boundary path. These are points of the rightmost and leftmost column of R, which are between the upper and lower boundary and therefore neither in the upper boundary path nor in the lower boundary path, as those only contain additional points to make them connected. However, no non-convexity witness path includes these points as the only border points. Such a path would need to contain a point right of the rightmost column of R or left of the leftmost column, which is prevented by the monotonicity property.

The upper boundary path of a digitally convex region first rises monotonously and then falls monotonously. Otherwise, a horizontal non-convexity witness path could be constructed due to the monotonicity property. Likewise, the lower boundary first falls, then rises monotonously.

2.3. Slopes

2.3.1. Grid paths in images

Regarding a convex Euclidean shape in $\mathbb{R}^2$, the slope of the upper boundary is not allowed to increase, the slope of the lower boundary is not allowed to decrease.

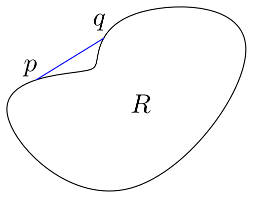


Figure 2.1.: Non-convexity in a Euclidean shape

In Figure 2.1, the slope of the upper boundary of a Euclidean shape increases between p and q, which leads to a non-convexity witness path between them. We want to find a similar notion of slope for the system of consistent digital line segments by Christ et al. [CPS10]. For this, a definition of regions on a pixel grid is necessary. The definition discussed in Section 2.2.2 is useful for a grid path, where the nodes have an area of zero. The pixels of the input image have a non-empty area, which may lead to ambiguity when computing the weight of an enclosed region. To resolve this ambiguity, we define the boundary paths to go through the pixel centers and the resulting region to include every pixel which is passed. The dashed boundary paths shown in Figure 2.2 enclose the region R. The upper boundary path consists of the nodes $(0, 3), (1, 3), (1, 4), (2, 4), (2, 3)$ and $(3, 3)$. The lower boundary path consists of the nodes $(0, 0), (1, 0), (1, 1), (2, 1)$ and $(3, 1)$.

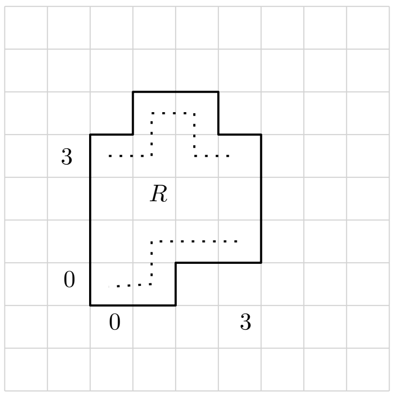


Figure 2.2.: A region defined by it's boundary paths

Some extensions of these boundary paths will result in a digitally convex region, others will not. We identify the *slope of a path* with the possible extensions preserving the convexity. Since the possible extensions of a boundary path depend on the metrics of previous pixels contained in it, $2(A+B)$ possible slopes exist in an image of size $A \times B$. This stems from the $A+B$ possible values for $m(.)$ for positive and negative slopes, respectively. Not all slopes can end on all pixels. For example, no path with a steep negative slope can exist on the upper border.

2.3.2. Total orders

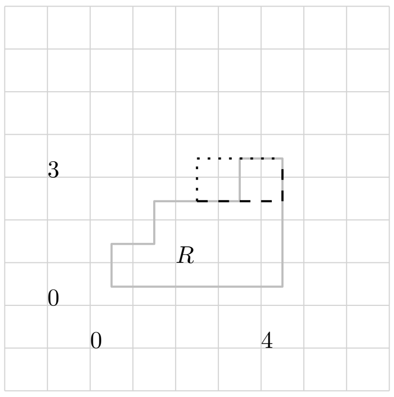


Figure 2.3.: Non-convexity in digital regions

In Figure 2.3, the region R is bounded by a staircase path as an upper boundary and a horizontal line as a lower boundary. Whether this region is considered convex depends on the total order used to construct the system of digital line segments. If the natural order is used, the line segment from $(2,2)$ to $(4,3)$ consists of the points $(2,2),(3,2),(4,2)$ and $(4,3)$, which are all in R. This line segment is represented as the dashed line. Considering a system of line segments based on the natural order, this region is indeed convex. However, using the natural order causes the resulting digitally convex regions to be staircase regions. Figure 2.4 shows an example of such a region. The lower boundary is simply a horizontal line, while the upper boundary has monotonicity as it's only constraint. These regions do not fit our intuitive concept of convexity very well. Apart from that, they could be computed faster, as the family of regions convex with respect to the natural order is a subset of the family of point-stabbed regions discussed by Chun et al [CST03]. Their algorithm would only needed to be modified slightly to compute regions digitally convex

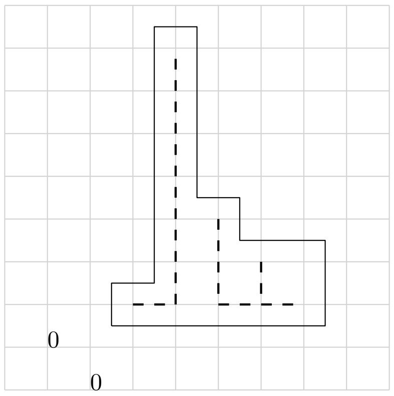


Figure 2.4.: A region R, digitally convex with respect to the natural order

with respect to the natural order. For these reasons, we concentrate on a different total order used by Christ et al. [CPS10].

Since a line segment constructed between points p and q with the natural order consists of the bottom and right border of the bounding box of p and q, the Hausdorff distance between the line segment $S(p,q)$ and the euclidean segment $\overline{pq}$ has a bound of $\Theta(\sqrt{n})$ for a euclidean distance of n between p and q. Because of this, Christ et al. [CPS10] used another total order based on powers of two, which improves the distance bound to $\Theta(\log(n))$.

Definition 1. Let $|k|_2 = \sup\{m : 2^m | K\}$ be the number of times k is divisible by 2.
We define the total order $\prec$ by saying $a \prec b$ if and only if there exists a non-negative integer i such that $|a-i|_2 \leq |b-i|_2$ and for all $j \in \{0,..,i-1\}$ we have $|a-j|_2 = |b-j|_2$.

From now on, we will refer to $\prec$ as the *canonical total order*. While the algorithm presented in Chapter 3 can operate with any total order $\prec$ on $[-n, n+m] \subseteq \mathbb{Z}$, we only used the canonical total order for the experimental evaluation in Chapter 4.

Using $\prec$, the line segment from $(2,2)$ to $(3,3)$ in Figure 2.3 becomes the dotted line, as $|4|_2 \geq |5|_2$, and therefore $5 \prec 4$. Since the point $(2,3)$ is not contained in R, the line segment between $(2,2)$ and $(3,3)$ is a non-convexity witness and R is not convex *with respect to* $\prec$. Thus, the same region can be convex with respect to one total order and not convex with respect to another.

2.3.3. Non-convexity witness paths

To generalize this example, we consider non-convexity witness paths in increasing upper boundaries: Let R be a region and p, q be two points in the upper boundary of R, whose line segment $S(p,q)$ is not contained in R. Without loss of generality, let $p_x \leq q_x$ hold. We call the path between p and q along the upper boundary the *boundary path b*. The boundary path and $S(p,q)$ have at least p and q in common. Since both the starting point and the end point of $S(p,q)$ are within R, $S(p,q)$ has to start at p in R, lead out of it and re-enter it to reach q. We follow $S(p,q)$ from p to q and observe the point where it diverges from the boundary path and therefore from R. We consider an increasing upper boundary, so the only way to diverge from R is for $S(p,q)$ to go up from a border point where the boundary path goes to the right. $S(p,q)$ may diverge from the boundary path by going to the right when the boundary path goes up, but it does not constitute a non-convexity witness path. As we are only interested in these, we focus on the first point where the boundary path goes to the right and $S(p,q)$ goes up. Naturally, it is a border point. Let $r = (r_x, r_y)$ be this border point and $s = (r_x, r_y + 1)$ it's upper neighbor, so that $r \in R \cap S(p,q)$ and $s \in S(p,q) \backslash R$. According to $\prec$, $m(r)$ must be one of the $q_y - r_y$ biggest numbers in $[m(r), m(q))$ to cause $S(p,q)$ to go up at r. Both paths go up $q_y - p_y$ times. Since the boundary path went to the right at r, it still has to go up $q_y - r_y$ times to reach q, one more time than $S(p,q)$. Since $m(r)$ was one of the $q_y - r_y$ biggest numbers in $[m(r), m(q))$, the boundary path only has $q_y - r_y - 1$ pixels u_i left for which $m(u_i)$ is among the $q_y - r_y$ biggest numbers. Therefore, it has to go up at another point t right of r, despite $m(t)$ not being one of the $q_y - r_y$ biggest numbers in $[m(r), m(q))$. The inequality $m(t) \prec m(r)$ holds.

We have now an easy criterion to check if any path between two points p, q ($p_x < q_x, p_y < q_y$) is a suitable upper boundary path for a digital region. If there is a non-convexity witness path, we can find the corresponding pixel where the boundary path goes to the right and $S(p,q)$ goes up, leaving the region. If there are pixels $r = (r_x, r_y)$, $t = (t_x, t_y)$ with $m(t) \prec m(r)$ and $r_x < t_x$ and the boundary path goes to the right at r and up at t, we can construct a non-convexity witness path $S(r, (t_x, t_y + 1))$.

Using this criterion, we can construct suitable upper boundary paths by ensuring no step up is made on a pixel t, if $m(t) \prec m(r)$ and the path went to the right at r. To formalize the possible extensions of boundary paths which leave it's suitability to be a boundary path unchanged, we define the *slope constraint* of a boundary. The slope constraint is an integer which determines the possible extensions of a boundary path.

2.3.4. Slope constraints

2.3.4.1. Ascending upper boundary

Let *ubp* be a valid ascending upper boundary path with the rightmost x-coordinate i. For an extension of *ubp* to column $i + 1$ to be valid, no step up can be made on a pixel whose value is smaller than any value of a pixel where *ubp* went to the right. Otherwise, a non-convexity witness path could be constructed as described in Section 2.3.3. Since the boundary path goes to the right once in every column, we only have to consider the pixels of the upper boundary. The metric of every pixel in column $i + 1$ where *ubp* goes up must be larger than the *upper slope constraint*, according to $\prec$.

Definition 2. The slope constraint uc(*ubp*) of an upper boundary path *ubp* is the maximum of all metrics where *ubp* went to the right.

$$\mathrm{uc}(up) := \max^{\prec}_{\substack{b \in ubp \\ ubp \text{ went to the right at } b}} m(b)$$

The slope constraint of an upper ascending boundary path in an image of dimensions $n \times m$ can be computed in $O(n)$, since only one value for every column has to be considered. If the slope constraint is updated while constructing a boundary path, the constant complexity per column will be dominated by the construction of the boundary path, as long as $a \prec b$ can be resolved in constant time. Since the possible values for a and b are independent of the actual picture, they can be computed beforehand and retrieved in constant time. $O(n + m)$ space is sufficient to store all possible values of $m(p)$ sorted according to $\prec$.

2.3.4.2. Descending upper boundary

Line segments with a negative slope are created by mirroring along the y-axis, as defined in Section 2.2.1. Let *ubp* be a valid descending upper boundary path and let q be the

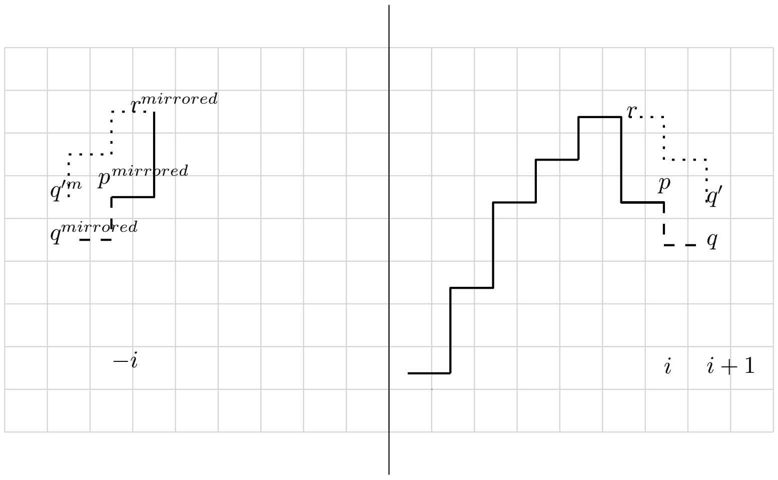


Figure 2.5.: Extending an upper boundary with negative slope through mirroring the line segments

uppermost pixel to which *ubp* can be extended. The metric of mirrored pixel $q^{mirrored}$ has to be lower than the value of any mirrored pixel $r^{mirrored}$, if *ubp* went downwards to r. Otherwise, a line segment from the selected pixel would go up and leave the region, as seen with q' in Figure 2.5.

Definition 3. The slope constraint of a descending upper boundary path *ubp* is the minimum of the metrics of all pixels to which *ubp* went downwards to

$$\mathrm{uc}(ubp) := \min^{\prec}_{\substack{b \in ubp \\ ubp \text{ went downwards to } b}} m(b^{mirrored})$$

$m(q^{mirrored}) \prec \mathrm{uc}(ubp)$ holds for every valid extension q. For the constraints of descending upper boundaries, several pixels per column must be considered, but only one per row. Similar to the ascending boundary, we can update the slope constraints in amortized constant time. Definition 2 is used for upper boundaries with positive slope, definition 3 is used for upper boundaries with negative slope. As argued in Section 2.2.2, an upper boundary will first have a positive slope, then the rightmost part of it will have a negative slope. Technically, a boundary path will not have a slope due to not being a straight line segment. However, we consider a boundary path ending in column i to have "negative slope" if the associated pixel in column i has a lower y-coordinate than the one in column $i-1$.

2.3.4.3. Descending lower boundary

Similar to the process applied to the descending upper boundary, the pixels in question are mirrored along the y-axis. The resulting constraints are similar to those used for the ascending upper boundary, mirrored along the y-axis as well. Let *lbp* be a valid descending lower boundary path and let q be the lowermost pixel to which *lbp* can be extended. No step down should be made to a pixel t, if *lbp* went to the right at s and $m(t^{mirrored}) \prec m(s^{mirrored})$ holds.

Definition 4. The slope constraint of a lower boundary path *lbp* is the maximum of the metrics of pixels $b^{mirrored}$, if *lbp* went to the right at b.

$$\mathrm{lc}(lbp) := \max^{\prec}_{\substack{b \in lbp \\ lbp \text{ went right at } b}} m((b_x+1, b_y)^{mirrored})$$

2.3.4.4. Ascending lower boundary

The process has some similarities with the slope constraint calculations of the upper ascending and the upper descending boundary. Let *lbp* be an ascending lower boundary path and let $q = (q_x, q_y)$ be the lowermost pixel *lbp* can be extended to. The metric of every pixel where *lbp* went up has to be greater than $m(q_x - 1, q_y)$.

Definition 5. The slope constraint of an ascending lower boundary *lbp* is the minimum of the metrics of pixels b where *lbp* went up.

$$\mathrm{lc}(lp) := \min^{\prec}_{\substack{b \in lbp \\ lbp \text{ went up at } b}} m(b)$$

2.4. Region definitions

We define three notations for use in the main algorithm.

2.4.1. Potential

Definition 6. A region A is said to have *higher potential* than a region B iff

- the rightmost column of A equals the rightmost column of B and
- the largest possible extension of B into the next column is a true subset of the largest possible extension of A.

This does not constitute a total order on the family of convex regions, since many regions are not comparable.

2.4.2. Weight

Definition 7. The *weight* of a region weight(R) is defined as the summed up weight of all pixels contained in it, i.e. $\mathrm{weight}(R) = \sum_{p \in R} \mathrm{weight}(p)$

Definition 8. A region A is said to *dominate* a region B iff

- A has a higher potential than B and
- $weight(A) \geq weight(B)$.

3. Algorithm

3.1. Outline

We describe two algorithms to find the weight-maximal digitally convex shape in an image. The first one requires only the input image and the total order used to define convexity. We call this the *complete algorithm*. Unfortunately, it has a space complexity of $O(n^5)$ and a computational complexity of $O(n^5 \cdot \log n)$, rendering it impractical for usual inputs. The second algorithm requires additional input but achieves a space complexity of $O(n^3)$ and a time complexity of $O(n^3 \cdot \log n)$. We call this second algorithm the *baseline-convex algorithm*. We will then proceed to combine both algorithms into a hybrid approach by scaling the input image to achieve an approximation with an overall running time that is dominated by the baseline-convex algorithm. Both algorithms are very similar in structure. In Sections 3.2 and 3.3 we give a high-level description of the complete algorithm, in Sections 3.4 to 3.10 detailed parts are discussed until the baseline-convex algorithm in Section 3.13 and the outline of the hybrid approach in Section 3.14 conclude the chapter. The sub-algorithms are shared between the complete and the baseline-convex algorithm with only minor adaptions. Therefore they are only discussed in-depth for the complete algorithm.

3.2. Input and output data

The algorithm to find a maximum-weight digitally convex shape in an $n \times m$ pixel image requires the following inputs:

- the image I, which can be interpreted as a matrix of pixel weights, $I \in \mathbb{Z}^{n \times m}$
- a total order $\prec$ on $[-n, n+m] \subseteq \mathbb{Z}$
- a threshold t

The pixel weights of I are constructed by using the brightness values of these pixels and subtracting t. The brightness values are usually positive while the pixel weights include positive and negative elements. Without the threshold, the weight-maximal region would trivially contain the whole image. The output consists of the upper and lower boundary defining the maximum-weight digitally convex region in I.

3.3. Complete algorithm

3.3.1. Region definitions

Definition 9. The region $R(x, uy, ly, uc, lc)$ is defined as the weight-maximal digitally convex region fulfilling the following constraints:

- x is the rightmost column of R
- uy is the upper boundary pixel of R in column x
- ly is the lower boundary pixel of R in column x
- uc is the upper slope constraint
- lc is the lower slope constraint

3.3.2. Data structure

The main data structures are three five-dimensional arrays of integers. The five dimensions are modeled after Definition 9.

1. $W(x, uy, ly, uc, lc)$ – the summed region weights
2. previousUpperY(x, uy, ly, uc, lc) – previous upper y-coordinate
3. previousLowerY(x, uy, ly, uc, lc) – previous lower y-coordinate
4. previousUpperC(x, uy, ly, uc, lc) – previous upper slope constraint
5. previousLowerC(x, uy, ly, uc, lc) – previous lower slope constraint

The first array stores the weight of the computed regions and is central to the algorithm. The second and third array are used to calculate the slope constraints and to backtrack the shape of the maximum-weight region after the execution of the main algorithm. The fourth and fifth array are not strictly necessary, they simplify reconstructing the maximum-weight region. A region R is stored by writing the corresponding values in the respective arrays. Often, we use W to describe a step in the algorithm and omit write operations in the other arrays, since they are exactly analogous. Due to the sparseness of the arrays in practice, the actual implementation uses nested maps. Since the array notation is clearer, it will be kept in the algorithm description.

3.3.3. Overview

Algorithm 1 processes the image I column by column from left to right. For every column i, the maximum-weight regions having their rightmost pixel in i are either created as single columns or constructed from the maximum-weight regions having their rightmost pixel in column $i - 1$.

Invariant 1 holds true for all x, uy, ly, uc, lc at each start of the outer for loop. Each value stored in W is the weight of the maximum-weight digitally convex region ending on the respective coordinates. We initialize W with $-\infty$ to indicate an empty field.

Invariant 1. $W(x, uy, ly, uc, lc) = -\infty$ or $W(x, uy, ly, uc, lc) = weight(R(x, uy, ly, uc, lc))$

At the first execution of the outer loop, W is completely empty. Invariant 2 holds for $x \leq i$ and all uy, ly with $ly \leq uy$ at every step $i = 0, .., n$ of Algorithm 1.

Invariant 2. $\exists uc, lc \; W(x, uy, ly, uc, lc) = weight(R(x, uy, ly, uc, lc))$

In experimental data, the set of regions $R(i, uy, ly, uc, lc)$ for all uy, ly, uc, lc in a given column i turned out to contain a high amount of redundancy. The majority of regions was dominated by other regions, i.e. their weight was less than or equal to the weight of another region with higher potential.

For this reason, we decided to reverse the direction in the implementation. Instead of computing the optimal region for every possible combination of uy, ly, uc, lc, we expand existing regions ending in column $i - 1$ to column i and then remove dominated and duplicate regions.

Algorithm 1: High-level description of the complete algorithm

```
Input: image $I$ as a matrix of pixel weights, total order $\prec$, (threshold $k$)
Output: maximum-weight digitally convex region in $I$
$M := \emptyset$ ;
for $i = 0$ to $n$ do
    create new regions in column $i$ and add them to $M$;
    foreach region $R \in M$ ending in column $i$ do
        extend $R$ to region $R'$ ending in column $i + 1$ so resulting area is maximized;
        add $R'$ to $M$;
    end
    foreach region $R' \in M$ ending in column $i + 1$ do
        if $R'$ contains more than one pixel in column $i + 1$ then
            trickle $R'$ to smaller regions $R'_{smaller}$;
            add $R'_{smaller}$ to $M$;
            /* $R'$ is not deleted                                    */
        end
    end
    foreach region $R' \in M$ ending in column $i + 1$ do
        update slope constraints of $R'$;
        if $R'$ is invalid due to anomaly then
            remove $R'$ from $M$
        end
    end
    consolidate regions;
end
return weight-maximal region in $M$
```

3.4. Create new regions in column i

A single vertical line is a valid digitally convex region. In every column, a new region is created for every combination of *upperY* and *lowerY* as shown in Algorithm 2. After this

Algorithm 2: Creating initial regions

```
Input: image $I$, column $x$
for $upperY = 0$ to $m$ do
    for $lowerY = 0$ to $upperY$ do
        $W(x, upperY, lowerY, x + upperY, -x + lowerY) = \sum_{i=lowerY}^{upperY} I(x, i)$
    end
end
```

step, Invariant 2 holds for $x \leq i$ and every uy, ly with $ly \leq uy$. Since the regions created in this step are digitally convex, Invariant 1 still holds. The initial slope constraints are the metrics of the uppermost and lowermost pixel, as these are the first pixels contained in the respective boundary paths. Newly created regions always have the highest potential.

3.5. Extend existing regions to column $i + 1$

When extending an existing digitally convex region ending in column i to column $i+1$, we have to ensure the extended region's convexity. The paths between pixels contained in the given region will not change, and a single column cannot contain any non-convex paths due to the monotonicity property, as discussed in Section 2.2.1. For that reason, we only have to consider possible paths between the previous region and the new column. Specifically, based on Section 2.3.3 and using the slope constraints constructed in Section 2.3.4, we calculate the highest possible extension of the upper boundary and the lowest possible extension of the lower boundary. Given a region $R = R(i, uy, ly, uc, lc)$ ending in column i, we create a region $R' = R(i + 1, newUpperY, newLowerY, uc, lc)$ such that $R \subset R'$ and $newUpperY - newLowerY$ is maximized. The possible extensions of both boundaries enclosing the maximum area can be computed almost independently, since they depend only on the region up to column i. As their only interaction, the lower boundary gives a lower bound for the upper boundary. Extensions where this constraint is violated are invalid and will be discarded. Extending region R to a single R' has a constant computational complexity. We focus on the extension of the upper boundary, as the lower boundary's extension works similarly. Since the upper ascending and the lower descending boundary both increase the distance between them, we call them *expanding* boundaries. The lower ascending and the upper descending boundary are called *contracting* boundaries.

3.5.1. Expanding boundary

In this section, we assume the upper boundary has a positive slope, we will discuss the other case in Section 3.5.2. Let $p = (p_x, p_y)$ be the last pixel of the upper boundary, $p_x = i$. Since we assume the boundary to be expanding, we use the first definition of the upper slope constraint to find a valid pixel to extend it to. This pixel will be called $q = (q_x, q_y)$. Naturally, q_x is $i + 1$. To maximize the enclosed area, q_y should be as high as possible. We find q_y by selecting the lowest pixel whose metric is smaller than uc. If it were not, we could go up at q_y and select q_{y+1} as the extension, so q_y would not be the highest possible. Since a horizontal extension of an ascending upper boundary is always valid, p_y is a suitable lower bound for q_y.

$$q_y = \min_{y \geq p_y,\ (p_x+1)+y \prec uc} y$$

where $\prec$ is the total order and uc is the upper slope constraint, as defined in Section 2.3.4.1. If $uc \prec (p_x + 1) + y$ holds true for every $y \geq p_y$, every y-coordinate would be a valid extension and we could select an arbitrary one. To maximize the area, we choose $q_y = m$, with m being the height of our image. Since $n + m$ pixel metrics exist in an $n \times m$-image, we can store possible minimal values for every slope constraint in $O((n + m)^2)$ space. Using precalculated values, we can find the y-coordinate for a given boundary extension in constant time.

In Figure 3.1, a given upper boundary is extended from column i to column $i + 1$. In this example, q is the highest possible extension, marked as a dashed line. The path from r to q', marked as a dotted line, is above p. If we took q' as the highest point of column $i + 1$, the path $S(r, q')$ would be a non-convexity witness, rendering the resulting region invalid.

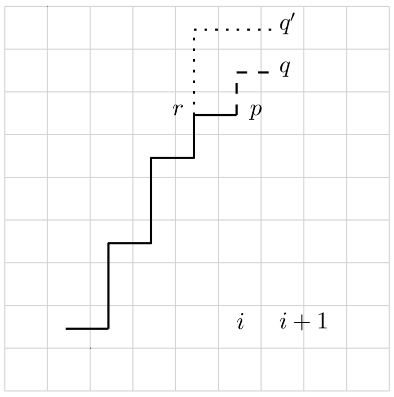


Figure 3.1.: Extending the upper boundary. $uc = r_x + r_y$

3.5.2. Contracting boundary

According to the line segment system definition of Christ et al. [CPS10], line segments with negative slope are constructed by mirroring the definition for positive slopes on the y-axis. For descending upper boundaries, these mirrored definitions result in the following constraints:

$$q_y = \max_{y \leq p_y, (-(q_x)+y \prec uc)} y$$

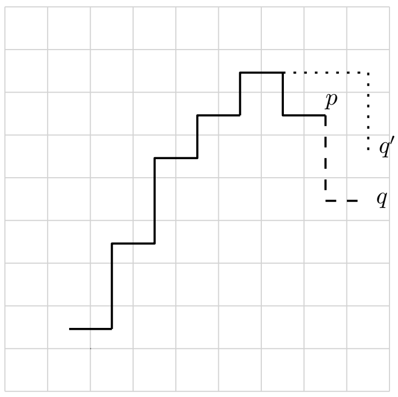


Figure 3.2.: Extending the upper boundary with a negative slope

The lower boundary of a region can be extended in a similar way. Given a region R ending in column i, we can now construct a region R' ending in column $i + 1$ by extending both boundaries. The resulting region is well-defined, as considered in Section 2.2.2.

If we use precalculated values for $\prec$, the operations described in this section can be executed in constant time.

3.5.3. Invariants

We showed in Section 2.3.3 that every non-convexity witness path necessitates a violation of the slope constraints. In the current step of the algorithm, we chose the extensions of the boundary paths so that they fulfill the slope constraints. Therefore, no non-convexity witness paths exist and the resulting region is convex.

3.6. Trickle

The region R' constructed by extending the boundaries of a previous region is valid, but not necessarily optimal. We need to consider all possible continuations of R into column $i+1$. In Algorithm 3, we “trickle inwards” both boundaries of R' to attain all possible extensions of R into column $i+1$. The updating of arrays *previousUpperY* and *previousLowerY* happens as well, but is omitted in the description for the sake of brevity. In Figure 3.3, an

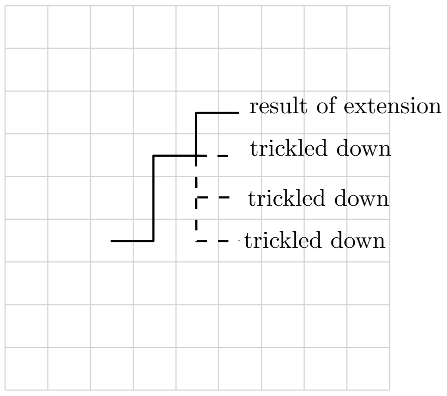


Figure 3.3.: Obtaining several upper boundaries by trickling down one

upper boundary is trickled down to obtain y new regions. This process is repeated until the upper and lower boundary overlap, as shown in Algorithm 3. In this phase, the slope

Algorithm 3: Trickling a single region

```
Input: region $R$
for $uy = upperY(R) - 1$ to $lowerY(R)$ do
    /* create new region with the upper boundary lowered and add it to
       $M$                                                          */
    $W(i+1, uy, lowerY, uh, lh) = W(i+1, uy+1, lowerY, uh, lh) - I(i+1, uy+1)$;
    for $ly = lowerY(R) + 1$ to $uy$ do
        /* create new region with the lower boundary raised and add it to
           $M$                                                      */
        $W(i+1, uy, ly, uh, lh) = W(i+1, uy, ly-1, uh, lh) - I(i+1, ly-1)$;
    end
end
```

constraints were only copied and may or may not be valid for the newly created regions. This change will be taken care of in Section 3.7. To avoid overwriting dominating regions, the resulting regions are not in fact written directly to W, but stored until the entire column has been considered. After this step, $(newUpperY - newLowerY + 1)^2/2$ trickled regions are created from R. Repeating this algorithm for every of the $O(n^5)$ regions stored in W would result in an overall computational complexity of $O(n^7)$, which is the reason another approach was developed. The details will be discussed in Section 3.12.3.

3.6.1. Invariants

The created regions with expanding boundaries are convex as well, since the vertical part of a line segment connecting a point in column i and column $i+1$ is contained in column $i+1$ and is now only shortened. Regions with contracting boundaries may or may not be valid, the invalid ones are discarded in Section 3.8.

3.7. Update slope constraints

As the slope constraints of a region R depend on its boundary paths, they change when the boundary paths change. Re-evaluating the formulas presented in Section 2.3.4 would necessitate considering the entire boundary path so far, which results in linear computational complexity. To keep the complexity for one path update constant, we only compare the previous slope constraints with the metrics of the boundary path pixels in the rightmost column. As shown in Algorithm 4, the metrics of pixels where the boundary path went up or down also influence the new slope constraint. For this calculation, we need the minimum, according to $\prec$, of an interval, defined with the natural order. Using precalculated values and $O((n+m)^2)$ space, we can retrieve these minimal values in constant time. Let *newUpperY* and *newLowerY* be the rightmost y-coordinate of the upper respective lower boundary in column $i+1$. The slope constraints are uc and lc, the y-coordinates in column i are *previousUpperY* and *previousLowerY*. Algorithm 4 has a constant computational

Algorithm 4: Update slope constraints

```
Input: previous slope constraints $uc$ and $lc$, column $i$, previous and current
       $y$-coordinates $newUpperY$, $newLowerY$, $previousUpperY$ and
       $previousLowerY$
Output: new slope constraints $uc$ and $lc$
/* upper boundary                                                  */
if upper boundary has positive slope then
    if $uc \prec i + 1 + newUpperY$ then
        $uc = i + 1 + newUpperY$;
    end
    else
        /* leave $uc$ unchanged                                    */
    end
end
else
    $uc = \min^{\prec}(uc, \min^{\prec}[-(i) + newUpperY, -(i) + previousUpperY))$;
    /* minimum over interval is used                               */
end
/* lower boundary                                                  */
if lower boundary has positive slope then
    $lc = \min^{\prec}(lc, \min^{\prec}[i + previousLowerY, i + newLowerY))$;
end
else
    if $lc \prec -(i+2) + newLowerY$ then
        $lc = -(i+2) + newLowerY$;
    end
    else
        /* leave $lc$ unchanged                                    */
    end
end
```

complexity, as long as the minimum of an interval according to $\prec$ can be found in constant time. Like before, this can be done with cached values in $O((n+m)^2)$ space.

3.8. Anomalies

Since the lower ascending and the upper descending boundaries share some properties, we call both *contracting* boundaries. In case of a contracting boundary, the regions derived by trickling a valid region inwards may not always be valid as well. For some total orders, *anomalies* can occur on upper boundaries with negative slope or lower boundaries with positive slope. The anomalies occur when two numbers A and B exist with $A \leq B$ and $B \prec A$. Naturally, using the natural order will prevent anomalies from occurring, but we will loose the bound for the Hausdorff distance.

In Figure 3.4, we have a region whose lower boundary has a positive slope. The point $q = (5, 1)$ is a valid extension of the lower boundary, according to $\prec$. The point $q' = (5, 4)$, even though it lies inwards of q, is not. The path between $(4, 0)$ and q' consists of $(4, 0), (4, 1), (4, 2), (4, 3), (5, 3), (5, 4)$, according to $\prec$. If we continue the lower boundary with q', $(5, 3)$ is not contained in the resulting region, which is therefore non-convex.

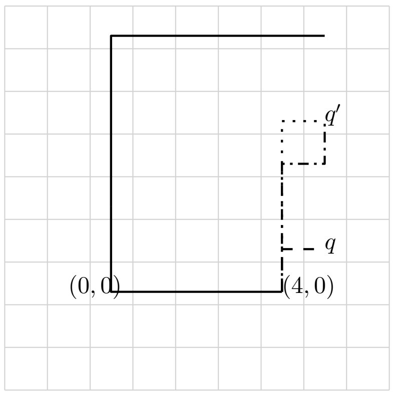


Figure 3.4.: Invalid region derived through trickling

After trickling contracting boundaries, we have to remove these invalid regions. If any of these two conditions holds true, the region will be deleted:

- $uc \prec -(i+1) + upperY$
- $lc \prec (i+1) - 1 + lowerY$

After the regions fulfilling the above conditions are deleted, the remaining ones are digitally convex and can be stored in W without violating Invariant 1.

3.9. Consolidate constructed regions

In the previous steps, we constructed the expansions of one region R ending in column i to a set of regions ending in R'. The descendants of a single region have unique values for $upperY$ and $lowerY$.

After repeating this constructions for all regions ending in i, we have created about $n^6/2$ regions. Since W only holds n^4 paths per column, some regions share coordinates. To get the optimal region $R(i+1, \cdot, \cdot, \cdot, \cdot)$, every constructed region CR is stored in the array W, only if there was no other region with a higher weight already stored for these coordinates. In both cases, the region with the lower weight is discarded. After this step, the Invariant 1 holds true. W does contain an empty value, indicated by $-\infty$, for some indices for which valid regions exist to reduce redundancy. An empty value indicates the existence of another region which dominates this one. For those indices, the associated region could be constructed by choosing the maximum-weight region under those with higher potential. In Section 3.4, one region consisting only of one column was created in each column i for every uy, ly with $ly \leq uy$. If we continue both boundaries horizontally, the resulting region is rectangular and therefore digitally convex. This region will only be deleted if a dominating region with the same i, uy, ly is found. For that reason, Invariant 2 holds true for all uy, ly with $ly \leq uy$.

3.10. Reconstructing the optimal region

After the main algorithm has finished, the boundaries of the maximum-weight region are stored implicitly in the arrays W, $previousUpperY$ and $previousLowerY$. We first identify the last column of the maximum-weight region by iterating over the values in W and pick the maximum. With the end points identified, we follow the boundaries backwards in Algorithm 5. Since the region is unambiguously characterized by the upper and lower boundary, as shown in Section 2.2.2, the output of Algorithm 5 is sufficient. We assume

Algorithm 5: following paths backwards

```
Input: $x, uy, ly, uc, lc$
Output: upper and lower boundary defining the maximum-weight region
        $R(x, uy, ly, uc, lc$
weight = $W(x, uy, ly, uc, lc)$;
/* if the weight of the region currently being backtracked is the same
   as the current column, we are already at the leftmost column      */
while $weight \geq \sum_{y=ly}^{uy} I(x, y)$;
do
    upperBoundary.prepend($x, uy$);
    lowerBoundary.prepend($x, ly$);
    previousUY = previousUpperY($x, uy, ly, uc, lc$);
    previousLY = previousLowerY($x, uy, ly, uc, lc$);
    previousUC = previousUpperC($x, uy, ly, uc, lc$);
    previousLC = previousLowerC($x, uy, ly, uc, lc$);
    weight = $W(x, uy, ly, uc, lc)$ - $\sum_{y=ly}^{uy} I(x, y)$;
    $x = x - 1$;
    $uy$ = previousUY;
    $ly$ = previousLY;
    $uc$ = previousUC;
    $lc$ = previousLC;
end
upperBoundary.prepend($x, uy$);
lowerBoundary.prepend($x, ly$);
return upperBoundary, lowerBoundary
```

Invariant 1 holds before the execution of Algorithm 5 and W is not empty for the input coordinates given. The associated region $R(x, uy, ly, uc, lc)$ is recreated by backtracking the values stored in the five arrays. According to Invariant 1, this region is indeed digitally convex.

The operations within the while loop take constant time. Since x is decreased with every loop repetition, the loop body will be executed at most n times. The computational complexity of Algorithm 5 is therefore $O(n)$.

3.11. Correctness

3.11.1. Returned region is convex

The initial shapes created in Section 3.4 are convex, since all line segments between two points sharing a x-coordinate only contain points within that column, see Section 2.2.1. The regions computed by Algorithm 1 are digitally convex if Invariant 1 holds at the end of the outer loop and Algorithm 5 returns a digitally convex region if Invariant 1 holds. The former is discussed in the respective sections, the latter in Section 3.10.

3.11.2. Returned region is weight-maximal

Given an image I, let R_{opt} be the maximum-weight digitally convex region on I. Let x_{opt} be the rightmost column of R_{opt} and $R_{predecessor}$ be region R_{opt} without x_{opt}. $R_{predecessor}$ is the maximum-weight region among those with the potential to be extended to R_{opt}. Otherwise, we could extend another region R' ending in $x_{opt} - 1$ to include the intersection of x_{opt} and R_{opt} so that the resulting region $R'_{extended}$ had a higher weight than R_{opt}. This is a contradiction to the initial assumption of R_{opt} being the maximum-weight digitally convex region on I. This argumentation can be extended to show that every predecessor R_i, part of R_{opt} up to region i, is the maximum-weight region among those ending in column i with the same constraints. The possible extensions of a region are controlled by the y-coordinates and slope constraints.

We now have to show that these regions are actually constructed. In the first step of Algorithm 1, all possible regions consisting of a single column, including the leftmost column of R_{opt}, are created. Since it is the maximum-weight region among those of it's coordinates and constraints, it will not be dominated by any other region and will be written to W.

In the phase of extending and trickling, all possible extensions of regions stored in one column i of $W(i, \cdot, \cdot, \cdot, \cdot)$ are created. Since R_i is stored in W, R_{i+1} is among the possible extensions. Since R_{i+1} is weight-maximal among the regions with the same potential, it will be saved to $W(i + 1, \cdot, \cdot, \cdot, \cdot)$. In accordance with Invariant 1, this continues until R_{opt} is computed.

3.12. Computational and space complexity

3.12.1. Space complexity

Each of the five five-dimensional arrays has a space complexity of $O(n \cdot m^2 \cdot (n + m)^2)$. In the extension and the slope constraint update phases, arrays with precalculated values are used. These use $O((n + m)^2)$ space each, their influence on the total space complexity is dominated by the five storage arrays. This results in an overall space complexity of $O(n \cdot m^2 \cdot (n + m)^2)$ or $O(n^5)$ for a quadratic input image.

3.12.2. Naive implementation

The algorithm steps described in Section 3.4, Section 3.8, Section 3.5 and Section 3.7 have a constant computational complexity for one region. The final part of retrieving the computed region, described in Section 3.10 has a computational complexity of $O(n)$. The naive implementation of the trickling step, described in Section 3.6, creates up to $m^2/2$ regions for every region in W. Using this implementation results in an overall time complexity of $O(n \cdot m^2 \cdot (n + m)^2 \cdot m^2 \cdot 1) = O(n \cdot m^4 \cdot (n + m)^2)$ or $O(n^7)$ for a quadratic input image.

3.12.3. Improvements

A simple speedup consists in consolidating the expanded regions first and trickling them afterwards in place in W. This results in a constant computational complexity for the trickle step and an overall complexity of $O(n^5)$. Unfortunately, the previous y-coordinates are necessary for the slope constraint update, as seen in Section 3.7. Before the boundaries are trickled inwards, the potential of a region depends on the previous y-coordinate as well as on the slope constraints. If, for example, $newUpperY \leq previousUpperY$, the upper boundary cannot rise again. Before the trickling and the slope constraint update are

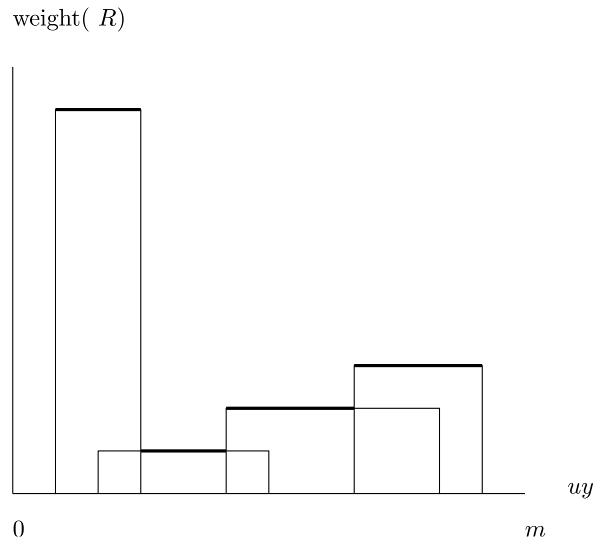


Figure 3.5.: Dominating regions modeled as a skyline problem.

completed, a lower value for *previousUpperY* results in a higher potential. Removing regions earlier could result in one region R overwriting another region R', if $uc(R) = uc(R')$, $lc(R) = lc(R')$, $uy(R) = uy(R')$ and $ly(R) = ly(R')$ and $weight(R) \geq weight(R')$, despite R' having a lower value for *previousUpperY* and therefore possibly having more potential than R. These coordinates are only important in the range between the most outward lying extension and *previousUpperY* respective *previousLowerY*. This can be constructed as a skyline problem. The x-axis of the skyline problem corresponds to uy or ly, the weight of the regions is represented as the height of the skyline shapes, as shown in Figure 3.5. The resulting skyline problem can be solved by sorting the uppermost and lowermost y-coordinate of the $O(m \cdot (n+m)^2)$ shapes, which results in a time complexity of $O(m \cdot (n+m)^2 \cdot \log(m \cdot (n+m)))$ for one skyline model. The sections marked in bold designate the regions written to W for each set of coordinates. The skyline modeling has to be repeated m-times per column, resulting in a complexity of $O(m^2 \cdot (n+m)^2 \cdot log(m \cdot (n+m)))$ per column. With this improvement, the stated complexity of $O(n^5 \cdot \log n)$ can be reached. Algorithm 6 implements this for the expanding upper boundaries and is inspired by work of Börzsönyi et al. [BKS01]. This problem does not appear for contracting boundaries, as the previous y-coordinate is of no significance in this case and they can be trickled in place. After the execution of Algorithm 6, the slope constraints are updated as usual, described in Section 3.7.

3.13. Baseline-convex algorithm

3.13.1. Introduction

With some additional input, we can reduce the computational complexity and space complexity to $O(n^3 \log n)$ and $O(n^3)$, respectively. We will first discuss the nature of these inputs in Section 3.13.2, then discuss a family of regions in Section 3.13.3 and the main algorithm in Section 3.13.5, to conclude with the space and time complexity and limitations in Sections 3.13.7 and 3.13.8.

3.13.2. Additional input

In addition to the inputs of the complete algorithm, as defined in Section 3.2, the baseline-convex algorithm requires two inputs:

Algorithm 6: Solving the influence of *previousY* as a skyline problem

```
Input: column $i$
extended = $\emptyset$;
foreach $R \in M$ ending in column $i$ do
    extend $R$ and add the extension $R'$ to extended;
end
for $ly = 0$ to $m$ do
    for $uc = 0$ to $(n + m)$ do
        for $lc = 0$ to $(n + m)$ do
            tempset = regions $R'(i + 1, \cdot, ly, uc, lc)$ in extended;
            activeset = $\emptyset$;
            sort the $y$-coordinates $uy$ and $previousUpperY$ of regions in tempset;
            /* skyline algorithm                                    */
            for $u = m$ down to $ly$ do
                add region $R' \in$ tempset to activeset if $uy(R') = u$;
                remove region $R'$ from activeset if $previousUpperY(R') = u$;
                $W(i + 1, u, ly, uc, lc) = \max_{R' \in activeset} weight(R')$;
            end
        end
    end
end
repeat the above for the lower boundary;
trickle the contracting boundaries in place;
```

- an array d of length n, containing one y-coordinate $d[i]$ for every column i
- the starting column *xstart*

The algorithm finds the maximum-weight region R fulfilling two additional constraints:

- *xstart* is the leftmost column whose intersection with R is non-empty.
- If R contains a pixel in column i, it contains $(i, d[i])$ as well.

The array d serves as the baseline, dividing the upper and lower boundary and allowing to compute optimal boundaries for the upper respective lower part independently. This reduces the space complexity to $O(n^3)$ and the time complexity to $O(n^5)$ or $O(n^3 \log n)$, depending on implementation choice, see Section 3.12.

3.13.3. Half-convex regions

To use the additional information of the baseline d, we define *half-convex regions.*

Definition 10. A half-convex region is a region H where either

- every non-convexity witness path contains pixels of the lower boundary, or
- every non-convexity witness path contains pixels of the upper boundary.

These types of half-convex regions are called *upper half-convex region* and *lower half-convex regions*, respectively. In Figure 3.6, U is an upper half-convex region and L is a lower half-convex region. Since they share the baseline (represented by the pixels filled in gray) and their leftmost and rightmost x-coordinate, they can be combined to form a digitally convex region. A subset of the baseline is the fixed lower boundary for the upper half-convex region and the upper boundary for the lower half-convex region. An upper and a lower boundary need to share the leftmost and rightmost x-coordinate to unambiguously define a region. The leftmost x-coordinate is given as *xstart*.

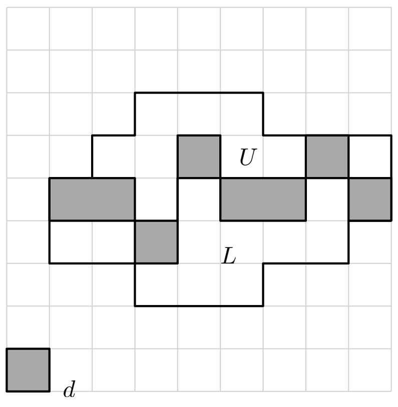


Figure 3.6.: Half-convex regions U and L

3.13.4. Data structure

Since we compute the upper and lower boundaries independently, we can store them in two three-dimensional arrays.

- UW(x, uy, uc) – weight of the upper half-convex regions
- UPreviousY(x, uy, uc) – path tracking of the upper half-convex regions
- LW(x, ly, lc) – weight of the lower half-convex regions
- LPreviousY(x, ly, lc) – path tracking of the lower half-convex regions

Since no upper boundaries end below the dividing line, the first two arrays are empty for all (x, y)-positions in the lower half. The same holds for the lower boundaries and the latter two arrays. Due to this and the overall sparseness, map structures were used in the implementation, similar to the complete algorithm.

3.13.5. Algorithm

The overall structure is outlined in Algorithm 7. It is very similar to the previous Algorithm 1. We proceed column by column in the direction of increasing x-values. The upper and lower half-convex regions are computed independently. To allow this, the leftmost column of each region is fixed as $xstart$. No completely new regions are created after the first step, which is a major difference to the first algorithm. After the execution of the main algorithm, the maximum-weight pair of half-convex regions is located and joined. Since the leftmost column is fixed, for every column i, a pair of half-convex regions with their rightmost column i can be combined to form a digitally convex region. After the maximum-weight pair of half-convex regions are joined for every column, we only have to iterate over all columns to pick the overall weight-maximal region.

3.13.6. Space complexity

Similar to Section 3.12, the space complexity is dominated by the storage arrays. The baseline-convex algorithm uses four three-dimensional arrays, which results in a space complexity of $O(n \cdot m^2)$.

3.13.7. Computational complexity

$O(n \cdot m^2)$ regions are considered. The extension, update of the slope constraints and removing of invalid regions have a constant complexity for each region. As in Section 3.12, the trickle step dominates the complexity for each region, resulting in an overall complexity of $O(n \cdot m^2 \log m)$

Algorithm 7: Computing a region baseline-convex on d

```
Input: total order $\prec$, image $I$, array $d$, int $xstart$
Output: maximum-weight region baseline-convex on $d$ starting on $xstart$
$UM = \emptyset$;
$LM = \emptyset$;
create upper half-convex regions in column $xstart$ and add them to $UM$;
create lower half-convex regions in column $xstart$ and add them to $LM$;
for $x = xstart$ to $n$ do
    foreach upper half-convex region $R$ in $UM$ ending in column $x$ do
        create $R'$ ending in column $x + 1$ from $R$;
        add $R'$ to $UM$;
    end
    foreach lower half-convex region $R$ in $LM$ ending in column $x$ do
        create $R'$ ending in column $x + 1$ from $R$;
        add $R'$ to $LM$;
    end
    foreach half-convex region $R' \in UM \cup LM$ that ends in column $x + 1$ do
        if $R'$ contains more than one pixel in column $x + 1$ then
            trickle $R'$ inwards to $R_{smaller}$;
            add $R_{smaller}$ to $UM$ or $LM$;
        end
    end
    foreach half-convex region $R' \in UM \cup LM$ that ends in column $x + 1$ do
        update slope constraints of $R'$;
        if $R'$ is invalid due to anomaly then
            remove $R'$ from $M$
        end
    end
    remove dominated regions;
end
return maximum-weight matching pair of half-convex regions in $UM$, $LM$
```

3.13.8. Limitations

The family of digitally convex regions recognized by Algorithm 7 is equal to the one recognized by Algorithm 1, if given the right input parameters. The quality of the result depends on the input parameters. If the optimal region R_{opt} and the dividing line do not intersect, the optimal region will not be found. Hence, it is vital to obtain suitable input parameters. A simple approach consists of calculating the summed weight for every row and choose the maximum row as the dividing line. After we chose the baseline d, the starting point $xstart$ could be obtained by choosing the leftmost point of the maximum-weight segment in d. This is feasible in $O(n + m)$. Unfortunately, even a simple diagonal shape, though convex, would elude this approach.

3.14. Hybrid algorithm

To solve the problem of obtaining good input parameters for the baseline-convex algorithm, we propose to use Algorithm 1 on a scaled down version I' of the input image I. The resulting weight-maximum region R_{small} is scaled up to fit the original image. From this result, the input parameters for Algorithm 7 are extracted. The baseline d is constructed by taking the arithmetic mean between *upperY* and *lowerY* of the up-scaled region in every column. We note that imperfections due to rounding errors occur. The leftmost column of R_{small} is scaled up and used as $xstart$. Again, this choice is not unambiguous, since several columns of the original image correspond to one column of R_{small}. Despite these limitations, the hybrid approach is able to find convex shapes R_{hybrid} of any form and orientation. From now on, the *baseline-convex algorithm* designates Algorithm 7 with the naive input parameters, as described in Section 3.13.8.

3.14.1. Computational complexity

The computational complexity of the hybrid algorithm depends on the scaling factor. If we choose $\sqrt{n} \times \sqrt{m}$ as the dimensions for I', the resulting computational complexity will be dominated by the complexity of the baseline-convex algorithm: $O(n^{5/2} \log^2 n + n^3 \log n) = O(n^3 \log n)$. The size of the draft image to be computed by the complete algorithm can be varied to balance output quality against performance.

3.14.2. Iterating

The input parameters obtained by the hybrid approach are not necessarily optimal. Two properties of the resulting convex region R_{hybrid} indicate the potential for improvements:

1. If the total weight of the leftmost column of R_{hybrid} is smaller than or equal to zero, $xstart$ should be increased to avoid including it.
2. If the total weight of the top row or the bottom row of R_{hybrid} is smaller than or equal to zero, d needs to be adapted. This adaption does not apply if the top or bottom row with negative weight is the only row intersecting R_{hybrid} in some columns. In other cases, the row in question should be omitted from the region and is only included in the current solution because an outlier in d forces the algorithm to do so.

The first step will be executed at most $width(I)/width(I')$ times and the second step at most $height(I)/height(I')$ times, which are the number of possible columns respective rows in I corresponding to one column respective row in I.

4. Evaluation

4.1. Anomalies

The shapes computed by our algorithms share the limitations of the underlying system of line segments. As Christ et al. [CPS10] discuss, the line segments and therefore our regions are not invariant under translation, rotation and mirroring, as the construction depends on the sum of absolute coordinates. Apart from that, some anomalies inconsistent with an intuitive understanding of convexity occur, as noted in Section 3.8.

4.2. Experimental setup

To evaluate the time requirements of the algorithms and the sparsity of the data structures, we generated random diamonds on top of white noise to use them as input images. An example can be seen in Figure 4.2. These images varied in size to examine scalability. Additionally we used the implementation to find digitally convex regions in images derived by magnetic resonance imaging, as shown in Figure 4.1. That was not part of the runtime test, instead served to illustrate a possible application. For the MRI-images, we chose a fixed threshold of 190 within a range of 0 to 255 for brightness values. In randomized images, the threshold was chosen randomly as well, between 0 and 255. We used the algorithms described in Sections 3.3, 3.13.5 and 3.14 to find convex regions in the aforementioned images. The scaling factor used for the hybrid algorithm's implementation was

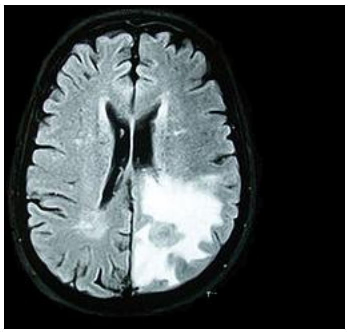


Figure 4.1.: 256×256-image of a brain tumor, derived by MRI. This image was released under the Creative Commons BY-SA license by Wikipedia user Bobjgalindo at `http://commons.wikimedia.org/wiki/File:MRI_brain_tumor.jpg`

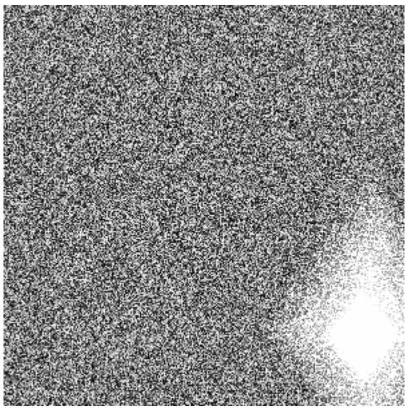


Figure 4.2.: 300 × 300-image of a diamond on top of white noise

$\sqrt{n}$ in an image of dimensions $n \times n$. When not called the *hybrid* algorithm, the *baseline-convex* algorithm denotes Algorithm 7 with the input derived by a scanline approach, as described in Section 3.13.8. The algorithms were implemented in Java 1.6.0 and executed with the OpenJDK Runtime Environment (IcedTea6 1.9.5) on Linux 2.6.34. The hardware consisted of an 8-core machine equipped with 32GB of random access memory and Intel E5430 processors, clocked at 2.66GHz. Since the parallelizations proposed in Section 4.4.4 were not implemented, only a single core was used.

4.3. Experimental results

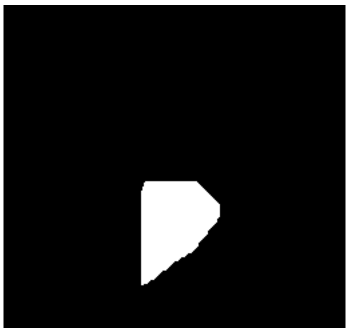
4.3.1. Limitations

Memory consumption was the limiting factor for each algorithm. The testing machine was equipped with 32GB of random access memory. With this, the complete algorithm was able to reliably compute convex regions in images with a width and height of 70 pixels. Regions in quadratic images whose width was in the range of $80 - 90$ pixels could sometimes be computed, since the memory consumption varies with the actual image. For images larger than 100×100, the complete algorithm was unable to obtain a result with the specified memory. The baseline-convex algorithm reliably operated on images consisting of up to 2000×2000 pixels. Regions in images larger than 4000×4000 could not be computed. The results for inputs between these marks varied and depended on the actual number of regions created.

4.3.2. Output

Figure 4.1 was too large for the complete algorithm, so we used the baseline-convex and the hybrid algorithm. The resulting regions are shown in Figure 4.3a and Figure 4.3b. The weights of the returned regions are 53810 and 54235, for the baseline-convex and hybrid algorithm, respectively. Obviously, the solution of the baseline-convex region is not optimal. While the hybrid algorithm's solution is very likely not optimal either, it shows one weakness of the other approach: The shape is slightly diagonal, therefore unobtainable by the baseline-convex algorithm, as discussed in Section 3.13.8.

Despite this advantage, the hybrid algorithm does not always yield better results than the baseline-convex one. An example for this would be Figure 4.2 and the results shown in Figure 4.4a and Figure 4.4b. In this example, the hybrid result is inferior due to the rounding errors mentioned in Section 3.14. Executing the algorithm again with a different value for xstart would achieve a better result.


(a) Result of the baseline-convex algorithm, weight of 53810.

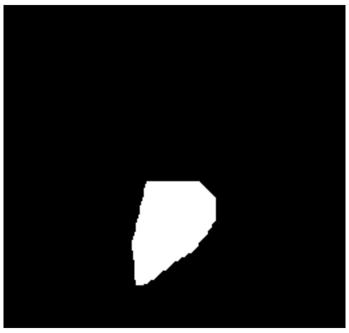
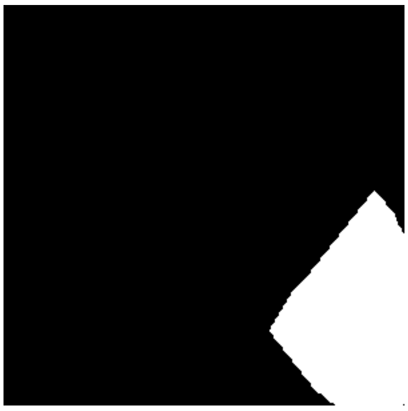

(b) Result of the hybrid algorithm, weight of 54235.

Figure 4.3.: Regions in Figure 4.1


(a) Result of the Baseline-convex algorithm, weight of 591095.

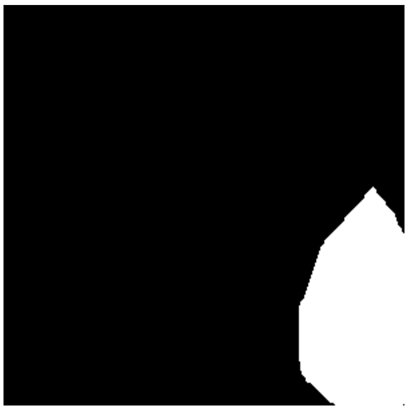

(b) Result of the hybrid algorithm, weight of 585579.

Figure 4.4.: Regions in Figure 4.2

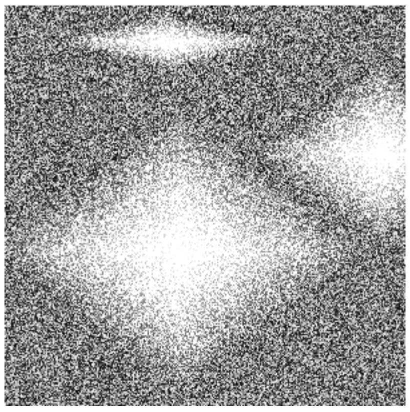


Figure 4.5.: Test image for occupancy measurements

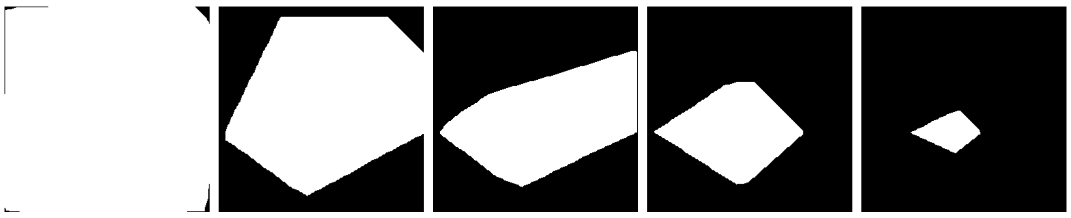


Figure 4.6.: Solutions computed by the baseline-convex algorithm on Figure 4.5 for thresholds 120, 140, 160, 180 and 240

4.3.3. Time and occupancy measurements

As noted in Section 3.3.2, only very few of the possible $O(n^5)$ regions are created. We call the ratio of occupied cells to total cells the *occupancy rate*. The occupancy rate does not correlate with image size, instead it depends on the brightness of the input image. While regions of negative weight cannot be discarded immediately, they are still much more likely to be redundant. An image brighter with respect to the threshold will result in more candidate regions and higher occupancy. The running time correlates strongly with the number of computed candidate regions. We executed the baseline-convex and the hybrid algorithm with different thresholds on a random input image, shown in Figure 4.5. The computed regions for some of the thresholds are shown in Figure 4.6. The ratio of occupied cells to total cells decreased with a rising threshold. The decrease was minor at first and became significant as the threshold passed the average pixel weight, see Figure 4.7.

The time requirements rose as expected from the asymptotic complexity. Figures 4.8 and 4.9 show the average time in seconds for 50 executions of the baseline-convex and complete algorithm, respectively.

4.3.4. Approximation quality

Since the complete algorithm turned out to be prohibitively computationally expensive for large images, we developed the baseline-convex algorithm as an approximation. To evaluate the quality of these approximations, we executed all three algorithms on ten test images of size 50×50. As apparent in Table 4.1, the approximation quality varies. The poor results of the hybrid algorithm can be attributed to the small image size: The draft image to generate suitable input data only had dimensions of 7×7, therefore rounding errors could influence the result substantially. Unfortunately, due to the limitations discussed in Section 4.3.1, we were unable to compare the hybrid and complete algorithms on suitably large images.

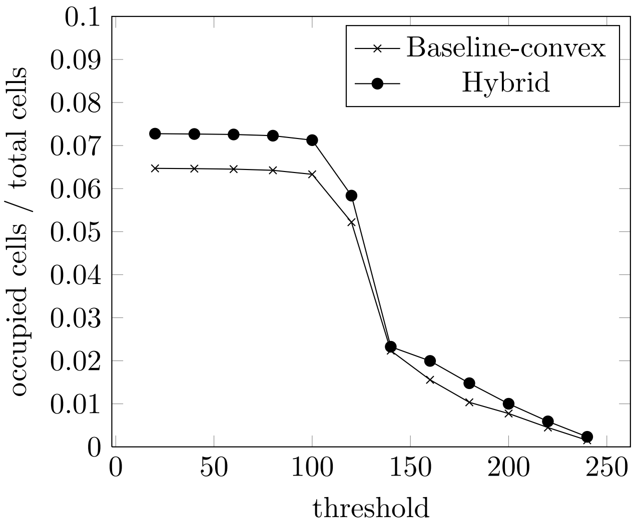


Figure 4.7.: Ratio of occupied cells for Figure 4.5 and different thresholds

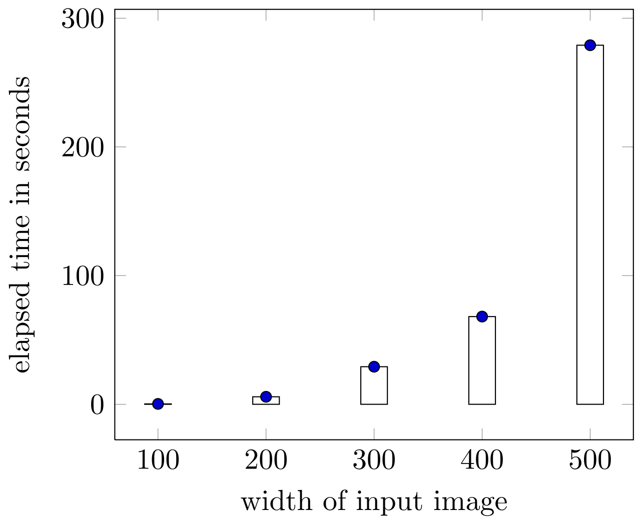


Figure 4.8.: Average time for the execution of 50 instances of the baseline-convex algorithm

4.4. Possible optimizations

We finally describe some optimizations to improve the performance on test data which, however, leave the asymptotic computational complexity unchanged. For this reason, we did not discuss them in Chapter 3. From the improvements described in this Section, only the custom data structure is used. We implemented the removal of dominated regions mentioned in Section 4.4.3, but with mixed results. The performance on some test images improved, on others hardly any regions were removed and the performance worsened due to the overhead.

4.4.1. Choice of data structure

In the algorithm description of chapter 3, we always used arrays in five and three dimensions, respectively. This conforms to the five possible parameters a region has in Algorithm 1. In experimental data, the arrays turned out to be filled very sparsely, as seen in Figure 4.7. To use less memory, we used maps to store the weight and previous y-coordinates of the computed regions. The tuple of (x, uy, ly, uc, lc) serves as a key. Among

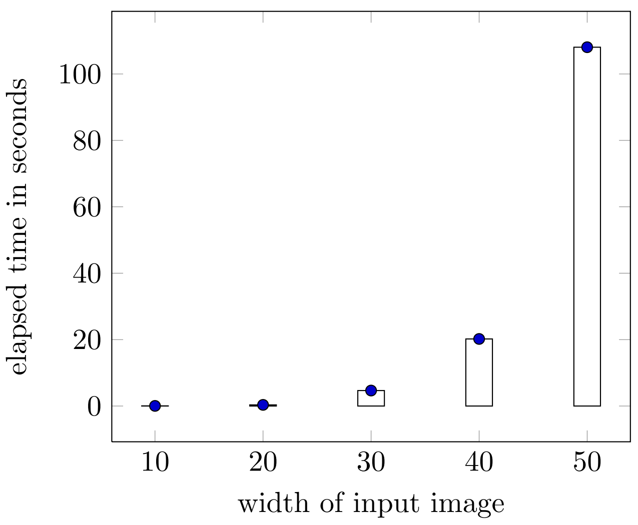


Figure 4.9.: Average time for the execution of 50 instances of the complete algorithm

	Complete	Baseline	Hybrid	ratio Baseline	ratio Hybrid
	808	579	234	70%	28%
	6042	4466	4182	73%	69%
	976	579	102	59%	10%
	3763	3626	3043	96%	80%
	660	446	584	67%	88%
	29832	24451	26886	81%	90%
	6171	6143	4973	99%	80%
	912	580	301	63%	33%
	695	552	217	79%	31%
	976	704	95	72%	9%
Average				75%	51%

Table 4.1.: weights of the three algorithms' solutions on 10 random input images

the possible data structures are a flat map and a nested map, with one map for every tuple of (x, uy, ly). Inspired by Invariant 2, the nested map for (x, uy, ly) is the smallest map guaranteed to be non-empty. Even though the primary purpose is to conserve memory, the performance improves, too. In the steps described in Sections 3.5, 3.6 and 3.7, iterations over empty cells can be omitted, therefore the computational complexity depends on the number of actual regions. The complexity to enumerate every region in the current column is linear for nested maps and may or may not be linear for flat maps, depending on map implementation. In the baseline-convex implementation, a nested map is used.

4.4.2. Remove obsolete old regions

To be relevant in the search for a maximum-weight region, any region computed has to be of maximum weight itself or be the predecessor of the maximum-weight region. Regions with a weight lower than the maximum encountered so far can therefore be deleted, if they do not have any non-overwritten successors. All regions in the current column would have to be retained, since it is not yet known whether any successors will be saved in W. Removing obsolete old regions does not improve the asymptotic space complexity, since in the worst case, every of the $O(m^2 \cdot (n+m)^2)$ regions in the current column i has a number of predecessors linear in n, leading again to a space requirement of $O(n \cdot m^2 \cdot (n+m)^2)$. Still, some memory consumption improvements could be attained. If the retrieval costs of the data structure in use depend on the total number of regions stored, this results in performance improvements.

4.4.3. Remove dominated regions

If a region R dominates another region R' (see Definition 8), R' can be removed without interfering with the result. In Section 3.9, R' is removed only if it shares the values i, uy, ly, uc, lc with R. We could compare all regions in (i, uy, ly) in pairs to find additional dominated regions and remove them. This is best done using a two-dimensional skyline algorithm, which leads to a computational complexity dominated by the sorting step.

4.4.4. Possible parallelizations

The algorithmic steps described in Sections 3.4, 3.5, 3.6 and 3.7 could be done in parallel if the architecture permits concurrent reading. With $O(m^2)$ processors, the naive implementation outlined in Section 3.12.2 can be implemented with a time complexity of $O(m^2 \cdot (n+m)^2)$ per column, which results in an overall complexity of $O(n \cdot m^2(n+m)^2)$. This could be achieved by computing every $O(m^2)$ possible extensions of a single region in parallel.

4.4.5. Implementation details

When using map structures to store the computed regions, as discussed in Section 4.4.1, many key objects are created and discarded. The use of object pools or customized data structures would result in some performance gains. We used Java for our implementation, switching to another language optimized for speed will very likely improve performance as well.

5. Conclusion

5.1. Findings

We designed and implemented an algorithm for the computation of weight-maximal digitally convex regions in arbitrary input images and interchangeable total orders. The optimal solution can be found in $O(n^5 \log n)$ time and approximations in $O(n^3 \log n)$ for images of size $n \times n$. There are applications in image recognition, for example medical imaging where convex regions are often of interest. While the rather high space and time complexity limit the algorithms usefulness in practical applications, our approach has some potential to be expanded upon.

5.2. Future work

5.2.1. Performance improvements

Some straightforward improvements are mentioned in Section 4.4. Implementing these, especially the parallelization, will result in noticeable performance gains. Additional improvements could be achieved by finding a suitable heuristic for the potential to estimate a lower bound for the weight-maximal region and discard unsuitable regions earlier. As we did not prove a lower bound for the computational complexity, there might be room for improvement as well.

5.2.2. Building pyramids

Apart from the maximum-weight region for a given threshold, a *pyramid* of regions for a set of thresholds is often of interest, for example to approximate mountains in noisy input images. Chun et al. [CST03] discussed the pyramid problem for point-stabbed regions, which are similar to and a superset of the digitally convex regions. The naive approach would be to execute our algorithm for every desired threshold and compose the resulting regions to a pyramid. While this is straightforward, one could also build a pyramid by modifying the data structure and algorithm. Since a different threshold is only a constant difference, a modified algorithm could generate regions for several thresholds in parallel and create a pyramid in this process.

Bibliography

[ACKT01] Tetsua Asano, Danny Z. Chen, Naoki Katoh, and Takeshi Tokuyama. Efficient algorithms for optimization-based image segmentation. *International Journal of Computational Geometry and Applications*, 11:145–166, 2001.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[CKNT08] Jinhee Chun, Matias Korman, Martin Nöllenburg, and Takeshi Tokuyama. Consistent digital rays. In *SCG '08: Proceedings of the twenty-fourth Annual Symposium on Computational Geometry*, pages 355–364, New York, NY, USA, 2008. ACM.

[CPS10] Tobias Christ, Dömötör Pálvölgyi, and Miloš Stojaković. Consistent digital line segments. In *SoCG '10: Proceedings of the 2010 Annual Symposium on Computational Geometry*, pages 11–18, New York, NY, USA, 2010. ACM.

[CST03] Jinhee Chun, Kunihiko Sadakane, and Takeshi Tokuyama. Efficient algorithms for constructing a pyramid from a terrain. In Jin Akiyama and Mikio Kano, editors, *Discrete and Computational Geometry*, volume 2866 of *Lecture Notes in Computer Science*, pages 108–117. Springer Berlin / Heidelberg, 2003.

[KM96] H. Kobatake and M. Murakami. Adaptive filter to detect rounded convex regions: Iris filter. *Pattern Recognition, International Conference on*, 2:340–344, 1996.

[KR04] Reinhard Klette and Azriel Rosenfeld. Digital straightness–a review. *Discrete Applied Mathematics*, 139(1-3):197 – 230, 2004. The 2001 International Workshop on Combinatorial Image Analysis.

Appendix

A. Source code

The source code of the implementation is available under the GPLv3 and can be found at `http://gitorious.org/digital-convex-regions`. The current commit at the time of writing is 31f1e08921ffddf1d3f3215c95fab68edd45762a.