Abstract

With the increasing size of datasets and demand for real time response for interactive applications, improving runtime for algorithms with excessive computational requirements has become increasingly important. Many different algorithms combining efficient priority queues with various helper structures have been proposed for computing grey-weighted distance transforms. Here we compare the performance of popular competitive algorithms in different scenarios to form practical guidelines easy to adopt. The label-setting category of algorithms is shown to be the best choice for all scenarios. The hierarchical heap with a pointer array to keep track of nodes on the heap is shown to be the best choice as priority queue. However, if memory is a critical issue, then the best choice is the Dial priority queue for integer valued costs and the Untidy priority queue for real valued costs.

Key words: Grey-weighted distance, Geodesic time, Geodesic distance, Fuzzy distance, Algorithms, Region growing

1. Introduction

Image analysis measurements are generally performed on binary representations of the objects. However, when images are acquired, grey levels have specific meanings. Binarisation of such images results in a loss of information and neither the internal intensities nor the borders of the resulting regions represent the imaged objects very well. This can be due to limited resolution, high noise levels, or that the border is a compound of objects. Because of this, measurements are increasingly done directly on grey-level images [31]. Fuzzy theory [41], where an image element has a membership value describing its belongingness to a certain (fuzzy) object, has emerged as a framework for addressing these problems [3, 15].

Distance calculations are widely used to extract shape and size information [3, 15]. This is an area where measurements in grey-level images have become increasingly popular [1, 13, 23, 26, 27, 32, 33]. The applications for content-based distance measures are many, e.g., grey-level morphology and minimal path detection [32], segmentation [21], clustering [13], and solving the Eikonal equation [17, 29, 35]. With the expanding size of datasets and demand for real time response for both automatic and interactive applications, improving memory efficiency and runtime for algorithms with excessive computational requirements has become a focus of greater importance [8, 21, 22]. Due to hardware limitations, the first efficient methods for computing distance transforms were based on the classic raster scan approach [28]. This approach works well for distance calculations on binary images, where a complete distance transform only needs two passes through the image. But for grey-level images, where the domain is generally not convex, the number of passes through the image becomes dependent on content. To improve runtime, propagation using graph-search techniques have become popular [11, 14, 35]. Most of the methods are versions of the well-known, theoretically optimal Dijkstra’s algorithm [10]. The wealth of data structures available for these algorithms makes analysing the computational complexity of all different combinations nontrivial. Even if it was trivial, implementations of lowest complexity might still not be the fastest due to practical implications. For example, the work by Luengo [19] shows the impact of current computer hardware on different priority queues. Also, special situations that often arise in image analysis problems, such as spatial homogeneity in images or overhead of complex structures when working on small problem domains, can also be a factor to why implementations with higher complexity might perform better than ones with low complexity.

Kimmel et al. [17] calculated the grey-weighted distance by solving the Eikonal equation using the Fast Marching method (FMM) [29], which is an efficient numerical scheme for solving the continuous boundary value problems. Here we focus on discrete distance definitions, covered in Section 2, and compare algorithms aimed to find the shortest path in a network with prescribed weights for each link between nodes. Numerical methods for approximating the solutions of a continuous problems are out of scope of this paper.

We put different implementations of the most popular grey-weighted distance transform algorithms, which we cover in Section 3 in a comparative test, under settings representative of common situations in image analysis, in Section 4. The work by Nyul et al. [22] presents a similar study on algorithms for fuzzy-connected image segmentation. However, it is important to point out that the results do not apply to grey-weighted distance transforms due to the different properties of fuzzy connectedness. Since our work relates to the same subject we have chosen to use similar terminology. We incorporate all algo-
gorithms and data structures used by Nyul et al. [22] and also include the data structures introduced by Yatziv et al. [40] and Luengo [19]. The conclusions in Section 5 should be seen as practical guidelines for selecting grey-weighted distance algorithms in different scenarios. To keep the adherence of the guidelines from being overly complex and off-putting, we have chosen competitive algorithms of low programming complexity and use containers from the C++ Standard Template Library (STL) [16] where applicable. We focus on local sequential algorithms, all parallel algorithms are out of scope for this article.

2. Discrete grey-weighted distances

The geodesic distance between two points included in a set is the length of the shortest paths or geodesics [28] linking these points and included in the set. The set is referred to as a geodesic mask, and when calculating grey-weighted distance the grey-scale geodesic mask is usually the same as the input image. In this paper we define a discrete grey-scale image \( f : \mathbb{Z}^n \rightarrow \mathbb{R}^+ \) as an application of a subset of the n-dimensional discrete space \( \mathbb{Z}^n \) into the set \( \mathbb{R}^+ \) of non-negative real numbers. The neighbourhood relations between the points in a discrete image are defined by a graph. We use an 8-connected graph for 2D square grids, and a 26-connected graph for 3D cubic grids. We define a discrete path \( P \) of length \( l \) − 1 going from node \( p \) to node \( q \) as a l-tuple \( (x_1, \ldots, x_l) \) of nodes such that \( x_1 = p, x_i = q, \) and \( (x_i, x_{i+1}) \) defines adjacent nodes for all \( i = 1, \ldots, l - 1 \). The grey-weighted distance \( d(p, q) \) represents the sum of all arc weights \( c_i \) along \( P \). This assumes that the arc weight \( c_i \) represents the cost of travelling from a node \( x_i \) to node \( x_{i+1} \). The grey-weighted distance \( d(p, q) \) then consists of finding the path with the lowest sum of arc weights \( c_i \) along all possible paths linking \( p \) to \( q \). If the set \( P_{pq} \) consists of all possible paths from \( p \) to \( q \), we have

\[
d(p, q) = \min_{P \in P_{pq}} \left( C(P) \right), \quad C(P) = \sum_{i=1}^{l-1} c_i(x_i, x_{i+1}),
\]

where the arc weight \( c_i \), also referred to as cost or local cost, is calculated by a cost function on the geodesic mask \( f \). Although the definition is general for \( n \) dimensions we refer to image elements as pixels (or nodes when utilising the graph analogy) unless we operate on 3D images, where we refer to them as voxels.

Rutovitz first proposed a grey-weighted distance where the arc weight is equal to the grey level of the destination pixel of each step along the path [24]. Levi and Montanari extended this definition when they defined a grey-weighted medial axis transform (GRAYMAT) by weighting the grey levels with the distance between adjacent pixels along the path [18]. In their definition, the length of a path is defined as the discretisation of the integral of the pixel values along the path, and the arc weight is defined as

\[
c_i = \frac{1}{2} (f(x_i) + f(x_{i+1})) \cdot ||x_i - x_{i+1}||, \quad (1)
\]

where \( || \cdot || \) refers to the spatial distance between two adjacent nodes in the image graph. Saha et al. proposed a theoretical framework for the GRAYMAT definition when applied to fuzzy sets [27]. Soille also defined a geodesic measure for fuzzy sets inspired by Levi and Montanari’s definition [32]. For more distance definitions on fuzzy sets we refer to [33].

Toivanen proposed two definitions for arc weights where the path between two points is defined as a path lying on the hyperplane defined by the grey levels [33]. The first is the distance on curved space (DOCS),

\[
c_i = |f(x_i) - f(x_{i+1})| + ||x_i - x_{i+1}||, \quad (2)
\]

and the second is the weighted distance on curved space (WDocs),

\[
c_i = \sqrt{|f(x_i) - f(x_{i+1})|^2 + ||x_i - x_{i+1}||^2}. \quad (3)
\]

While GRAYMAT propagates fast for low grey levels, DOCS and WDOCS account for the changes in height of the ‘height map’ and represent the minimal amount of ascents and descents to be travelled to reach a neighbouring pixel. DOCS performs the distance calculation with integer numbers while each sub-distance along the path for WDOCS is euclidean.

Figure 1 shows the different grey-weighted distance function behaviours. The top row shows the grey-weighted distance transforms when using a gradient image as geodesic mask. The GRAYMAT transform progresses rapidly across the area of low grey levels in the top left corner while DOCS and WDOCS have their fastest progression when travelling normal to the gradient direction. The bottom row shows the transforms on a sinusoidal image with saturated intensities. Once again it is clear that GRAYMAT moves fast in areas of low grey level (the black rings) and slow in areas of high grey level (the white rings). DOCS and WDOCS, on the other hand, move fast in both areas with high and low grey level but move slower through the transitions between two uniform areas, where the difference in grey levels results in increased costs. A more detailed analysis of the behaviour of the different transforms is presented in [12].

3. Algorithms

When referring to a grey-weighted distance computation it can be in one of four settings: (i) grey-weighted transform (or transform for short), (ii) seeded grey-weighted transform (or seeded transform for short), (iii) grey-weighted dilation (or dilation for short), or (iv) route. In the seeded (or marker-based) distance transform, each pixel is the grey-weighted distance of the lowest cost path from a set \( S \) of predefined pixels generally referred to as seeds, markers or features. The other three settings, (i), (ii), and (iv) are all special cases of the seeded transform. In the distance transform, each pixel is the grey-weighted distance from the background. Generally the background is defined as \( B = \{ x \mid f(x) = 0 \}, \) i.e., all pixels with grey level zero in the geodesic mask, which is the same as a seeded grey-weighted transform where \( S = \{ x \mid f(x) = 0 \}. \) Dilation is a seeded transform where the seeds represent the region
to be dilated and the calculation is stopped when a predefined grey-weighted distance is reached. The result is analogous to a morphological grey-level dilation of $S$ where the structuring element is defined by the cost function. The route is a single source shortest path problem calculating the grey-weighted distance from a single pixel $p$ to a single pixel $q$. This is done by seeded transform from the single seed point $p$ and terminating the transform once $q$ is reached.

The seeded transform offers better options for experimental setups than the regular transform by facilitating multiple runs on the same image using different seeds. However, the evaluation can be used as a guideline for choosing an algorithm for both the seeded and unseeded transform since the algorithms are the same. The runtime of a dilation or a route is naturally lower than that of calculating the transform of an entire image. However, the aim of this work is not to illustrate the low computational costs of various methods but to compare various implementations of grey-level–based distance computations, which is shown more clearly for complete (or near complete) image transforms than for dilations and routes.

The first method to calculate grey-weighted distance transforms was to use the chamfer scan approach, e.g., see [18]. The algorithm uses a window containing a weight mask (chamfer mask), and is slided across the image, updating the central pixel at each position. The scan consists of a forward pass and a backward pass. Figure 2 shows the masks used for the forward and backward passes for both 2D and 3D images. The chamfer weights are typically $w_1 = 3$, $w_2 = 4$, and $w_3 = 5$. In contrast with distances from binary images, the domain is usually not convex. Therefore, the chamfer algorithm for grey-weighted distance is an iterative process and has to be repeated until no updates are made in the distance map.

The chamfer algorithm can be considered an algorithm of the label-correcting kind. A grey-weighted distance label is assigned to a pixel at each step; the grey-weighted distance labels are estimates (i.e., a upper bounds on) the grey-weighted distance of the lowest cost path from the source to the individual pixels. What characterises a label-correcting algorithm is that all labels are considered temporary until the final step, when they all become permanent.

A different approach from iterative raster scan in the chamfer algorithm is the graph search approach. Two simple graph search approaches are the depth-first search (DFS) [7] (referred to as recursive propagation) and the breadth-first search (BFS) [7] (referred to as ordered propagation) used by Silvela et al. for distance transform computations [30]. The algorithm for both approaches is listed in Algorithm 1. They are both label-correcting algorithms and the difference between them comes from how pixels are added and removed from the list $L$. For recursive propagation, $L$ is a last-in-first-out (LIFO) list,
i.e., a list where the last pixel added is the first to be removed. For ordered propagation, $L$ is a first-in-first-out (FIFO) list, i.e., a list where the pixel added first is the first to be removed.

Algorithm 1 Depth/Breadth-first search (label-correcting)

**Require:** Seed map $S$, geodesic mask $f$, and empty list $L$.

**Ensure:** A grey-weighted distance map $G$ of $f$.

1: set all elements of $G$ to $\infty$ except $S$ which is set to 0;
2: put all pixels adjacent to $S$, not on $S$, in a list $L$;
3: while $L$ is not empty do
4: remove a pixel $x$ from $L$;
5: find $d_{\text{min}} = \min_{n\in\text{adj}(x)}(G(x), G(n) + c(n, x))$;
6: if $d_{\text{min}} < G(x)$ then
7: set $G(x) = d_{\text{min}}$;
8: put all pixels adjacent to $x$ on $L$;
9: end if
10: end while

The opposite of label-correcting algorithms are algorithms of the label-setting kind. A label-setting algorithm assigns one label as permanent (optimal) at each iteration. The algorithms of this group are basically a variety of implementations of Dijkstra’s well known algorithm [10], first proposed for grey-weighted distance transforms as the uniform cost algorithm [36]. In graph search terminology it is referred to as best-first search since the label-setting algorithms are much more efficient than label-correcting algorithms, but they are applicable only to special situations like region growing scenarios. Recently much work on grey-weighted distance computations has been done using various label-setting implementations [11, 14, 15]. Even though all of them use the theoretically optimal Dijkstra’s algorithm, they differ in what data structures they use. The label-setting algorithm used is listed in Algorithm 2.

#### 3.1. Local cost computation

The structure of the distance map will depend on the image connectivity and spatial distance between adjacent nodes. Common choices for spatial distance in a local neighbourhood are city block, chessboard, 3–4–5, or one of the optimal chamfer weights designed to approximate the Euclidean distance over large distances [4, 5, 7]. However, the relative efficiency of the various algorithms is unlikely to vary with different choices. We choose to only use the common 3–4–5 chamfer weights, since one of the algorithms is only applicable to integer costs, and, as previously mentioned, 8-connectivity for two-dimensional images and 26-connectivity for three-dimensional images.

#### 3.2. Implementations

This section presents the details of the different strategies and data structures used for the test cases in Section 4. The methods are labelled using capital letters with subscripts representing the properties of the method/data structure (see Table 1 for a summary). These labels will be used throughout Section 4.

Algorithm 2 Best-first (label-setting)

**Require:** Seed map $S$, geodesic mask $f$, empty priority queue $Q$, and empty set $E$ for expanded nodes.

**Ensure:** A grey-weighted distance map $G$ of $f$.

1: set all elements of $G$ to $\infty$ except $S$ which is set to 0;
2: put all pixels in $S$ on the queue $Q$;
3: while $Q$ is not empty do
4: remove a pixel $x$ from $Q$ for which $G(x)$ is minimal;
5: add $x$ to $E$;
6: for each $n$ adjacent to $x$ not in $E$ do
7: find $d_{\text{min}} = \min(G(n), G(x) + c(n, x))$;
8: if $d_{\text{min}} < G(n)$ then
9: set $G(n) = d_{\text{min}}$;
10: if $n$ is already on $Q$ then
11: update position of $n$ in $Q$;
12: else
13: put $n$ on $Q$;
14: end if
15: end if
16: end for
17: end while

#### 3.2.1. Label-correcting algorithms

The implementation of the chamfer algorithm does not give much room for variation. The chamfer method iterates over the image through raster scans using the chamfer mask shown in Figure 2 and is given the label C.

For recursive and ordered propagation we use variations of Algorithm [1]. Step 8 puts all neighbours of $x$ on the list if $x$ is updated. This will result in lots of duplicates on the list and unnecessary pop operations. To improve the speed of the algorithm, a pointer array can be used to keep track of whether a pixel is already on the list. If a pixel is already on the list, it need not be duplicated. The propagation algorithms are given the label P with the subscript L for LIFO list (recursive propagation) and F for FIFO list (ordered propagation). The subscript A is used if a pointer array is used to keep track of which pixels are on the list.

#### 3.2.2. Label-setting algorithms using d-heap

Roughness in images was computed by Ikonen et al. [14] using the DOCS transform with a binary heap [7] (d-ary heap with $d = 2$). The algorithm used is the same as Algorithm 2 but without Step 10, i.e., no check whether the neighbour $n$ is already on the queue $Q$. This results in duplicates on the queue, which leads to unnecessary pop operations. Here we represent the priority queue $Q$ in Algorithm 2 by a $d$-heap, both without any helper structures, as Ikonen in [14], and with some helper structures to keep track of the pixels on the queue. The key of a pixel $x$ in $Q$ is the grey-weighted distance $d(S, x)$ at the time it is inserted into $Q$. Since low grey-weighted distance values have priority over high values, all priority queues used in this work are minimum priority queues, i.e., the root stores the element with the smallest key. Step 4 is the extract-min operation, which finds the smallest key and removes it from the
heap, and the update in Step 11 is the \( \text{decrease-key} \) operation, which increases the node priority.

We use \( d = 2 \) for all \( d \)-heaps in the experiments. See Section 4.3 for a more detailed discussion on selecting \( d \). In the first implementation using \( d \)-heap, we always insert a new instance of \( n \) in Step 11 and 13, like in [14], even if it means duplication. This algorithm is labelled H (for heap). In another implementation (labelled H\(_A\)) we use a pointer array, which for every pixel \( x \) stores the position of \( x \) in the heap or NULL indicating that \( x \) is not on the heap. The pointer array is used in Step 10 to check if a pixel is on the queue, and in Step 11 to update the priority (\( \text{decrease-key} \)). The final group using \( d \)-heaps implement hash tables instead of a pointer array to use less memory. They use hash tables with various hash functions and various table sizes to keep track of the positions of the pixels in the heap. However, keeping a hash table requires additional computation, and hash tables work differently depending on the distribution of the key values assigned to the input data (i.e., depending on the grey-weighted distance values and on the geometric structures of objects in the image).

We use the same geometric hash functions used by Nyul et al. for 3D images [22]. They assign a key value to a heap pixel \( x \) using the following equations:

\[
\begin{align*}
\text{key}(x) &= \left((c_1 \cdot \text{height} + c_2) \cdot \text{width} + c_3\right) \mod H, \tag{4} \\
\text{key}(x) &= (c_1 + c_2 + c_3) \mod H, \tag{5} \\
\text{key}(x) &= \left((c_1 \cdot c_2 \cdot c_3) \mod H\right), \tag{6} \\
\text{key}(x) &= \left((c_2 \oplus c_3) \mod H\right), \tag{7}
\end{align*}
\]

where \( c_1, c_2, c_3 \) are the coordinates of the voxel, height and width are the dimensions of a slice, \( H \) is the size of the hash table, and \( \oplus \) is the bit-wise exclusive or operation. The corresponding hash functions for 2D are acquired by simply removing all instances of \( c_3 \) and ‘height’ from the above equations.

For Equations (4) and (6) we use a hash table size of 512 in the 2D case and 8191 in the 3D case. The range of possible hash values is quite large and these sizes are reasonably small and result in a fairly uniform distribution of hash keys in our application. For Equation (5) we use a hash table size of 512 for the 2D case and 768 for the 3D case. For Equation (7) we use a size of 256. This gives us a separate hash bin for each combination of 8-bit coordinates. We use labels H\(_{\text{IN}}\), H\(_{\text{SUM}}\), H\(_{\text{PROD}}\), and H\(_{\text{XOR}}\), respectively, for the algorithms corresponding to Equations (4)-(7).

3.2.3. Label-setting algorithms using Fibonacci heap

The Fibonacci heap [17] is a more complicated structure than the \( d \)-heap. While the \( d \)-heap supports in \( O(\log n) \) worst-case time the operations insert, extract-min, decrease-key and delete, the Fibonacci heap supports the same operations but have the advantage that operations that do not involve deleting an element run in \( O(1) \) amortised time [17]. This means that insert (Step 13) and decrease-key (Step 11) on average run in constant time. In theory the Fibonacci heap is especially desirable when the number of extract-min and delete is small relative to the number of other operations performed. However, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary \( d \)-heaps for most applications [7]. Despite the high programming complexity of the Fibonacci heap we have chosen to incorporate it for comparison purposes.

Like for \( d \)-heaps, the first version (labelled F for Fibonacci) does not keep track on whether a pixel is already on the heap, and we do not perform a search in the heap for a pixel already stored. This means that we always insert a new instance of the pixel, even if it means duplication. In another version (labelled F\(_A\)) we use a pointer array to keep track of pixels on the heap. The final version (labelled F\(_{\text{SUM}}\)) use the hash table that performed best among Equations (4)-(7) in Section 4.5 (Equation (5)), with the same table size as was used for the \( d \)-heap.

3.2.4. Label-setting algorithms using circular bucket queues

Falcao et al. used an efficient implementation of Dijkstra’s shortest path algorithm for grey-weighted distance [11]. It was the circular bucket queue data structure for integer costs introduced by Dial [9]. The priority queue is represented by a circular array where every position (bucket) in the array holds a doubly linked list of all nodes with equal path cost. This is illustrated in Figure 3(a), where the priority queue is represented as an array of \( B = C_m + 1 \) buckets containing the nodes in \( G \), with \( C_m \) being the maximal possible arc weight. Each bucket \( k \) stores a list of all nodes whose path cost is equal to \( k \). The drawback with Dial’s queue is that it only works with integer values, i.e., in our case we can only run it with the DOCS cost function. We test Dial’s bucket queue with both LIFO and FIFO lists, labelled D\(_L\) and D\(_F\) respectively. The LIFO version is also implemented using a pointer array to keep track of whether a pixel is on the queue or not. The pointer array implementation is labelled D\(_{\text{LA}}\).

Yatziv et al. have proposed the Untidy queue [40]. It is a circular bucket queue like Dial’s, but the number of buckets can be chosen freely and each bucket stores all nodes whose
path cost fall within a certain interval. If the number of buckets (referred to as bucket size) is \( B = B_m + 1 \), with \( B_m \) being the highest bucket index, then the bucket number \( k \) is determined by

\[
k = \text{floor}\left(\frac{d(p)}{C_m} B_m\right).
\]

(8)

where \( d(p) \) is the path cost of node \( p \) and \( C_m \) is the maximal possible arc weight. It follows that each bucket \( k \) will hold all nodes whose path cost is on the interval

\[
d_k(p) = \left[k \frac{C_m}{B_m}, (k + 1) \frac{C_m}{B_m}\right).
\]

(9)

This means that the Untidy priority queue works on floating point values as well as integer values. However, since the queue can put nodes with different costs in the same bucket, the Untidy queue will not necessarily return the node with the lowest path cost from a find-min operation. This introduces a rounding error which can be bound by choosing a large bucket size. See [24] for a more detailed analysis of the rounding error. Here we choose the bucket size 261 for DOCS and 2551 for GRAYMAT and WDOCS to ensure a low error. See Section 4.6.3 for a more detailed discussion on selecting bucket sizes. The labels for the Untidy queue implementations are \( U_L, U_F, \) and \( U_{L,A} \), where the subscripts refer to the same properties as for the Dial queue implementations.

The final circular bucket queue implementation used is the hierarchical heap proposed by Luengo [19]. The circular array functions the same way as for the Untidy queue, i.e., each bucket holds pixels with costs on a given interval. However, instead of representing each bucket with a list the hierarchical heap uses a \( d \)-heap at each bucket. This introduces the extra runtime of the heap compared to a list, but it functions properly with real valued costs, i.e., it will always return the node with the lowest path cost from a find-min operation. This also memory consuming. Since the static array needs to be the dynamic allocation data structure. We have used containers from the C++ Standard Template Library (STL) [14] where applicable to ensure a low programming complexity. This will make the algorithms easier to implement and, thus, encourage image analysts to adopt to the guidelines provided. The depth-first and breadth-first lists use the STL list container. The \( d \)-ary heap and hierarchical heap use the STL vector container (the STL heap does not support the update in Step 11 of Algorithm 2). Figure 3 (b) illustrates how an array is used to store a heap. Dial’s bucket queue and the Untidy bucket queue are both implemented using an STL list container

for each bucket. When a pointer array is used to keep track of nodes on the queue, the queue needs to allow for random access to facilitate updates (Step 11 of Algorithm 2). Standard list containers do not support random access, which means that updating a node on the Dial or Untidy queue is associated with a search through the list occupied by the node. This search will most certainly result in higher runtime compared to using a customised list implementation allowing random access.

The execution time can be cut by implementing custom containers using static arrays or intelligent reallocation based on application-specific memory usage. However, such custom implementations are memory consuming and have high programming complexity, and algorithms based on such implementations are not easy to adopt. For example, implementing the Dial queue using a static array, which was done by Falcao et al. [11], is not only nontrivial compared to using STL lists, but also memory consuming. Since the static array needs to be the same size as the image and contain two pointers per element, the static array will use at least twice the memory of the distance map. See [8] for a more detailed discussion on the memory issue for static arrays in region growing algorithms. Because of both the high programming complexity and the large memory requirements, static array implementations are out of scope for this article. However, we include runtimes for static array implementations of the Dial queue and the Untidy queue in Section 5.6.3 to give an indication of the tradeoff between dynamically allocated priority queues and priority queues using static arrays.

Table 1: Summary of the various methods used for comparison and their associated labels

| Label | Method | Data structure | P. array | Hashing |
|-------|--------|----------------|---------|---------|
| C     | Chamfer|                |         |         |
| P_L   | Depth-first | LIFO          | Yes     |         |
| P_F   | Breadth-first | FIFO          |         |         |
| P_LA  | Depth-first | LIFO          | Yes     |         |
| P_FA  | Breadth-first | FIFO          | Yes     |         |
| H     | Best-first | d-ary heap    |         |         |
| H_A   | Best-first | d-ary heap    | Yes     |         |
| H_LIN | Best-first | d-ary heap    | Eq. (8) |         |
| H_SUM | Best-first | d-ary heap    | Eq. (8) |         |
| H_PROD| Best-first | d-ary heap    | Eq. (8) |         |
| H_KOR | Best-first | d-ary heap    | Eq. (8) |         |
| F     | Best-first | Fibonacci heap|         |         |
| F_A   | Best-first | Fibonacci heap| Yes     |         |
| F_SUM | Best-first | Fibonacci heap|         |         |
| D_L   | Best-first | Dial’s/LIFO   | G-w. dist. |       |
| D_F   | Best-first | Dial’s/FIFO   | G-w. dist. |       |
| D_LA  | Best-first | Dial’s/LIFO   | G-w. dist. |       |
| U_L   | Best-first | Untidy/LIFO   | G-w. dist. |       |
| U_F   | Best-first | Untidy/FIFO   | G-w. dist. |       |
| U_LA  | Best-first | Untidy/LIFO   | G-w. dist. |       |
| HH    | Best-first | H-heap        | G-w. dist. |       |
| HH_A  | Best-first | H-heap        | Yes     | G-w. dist. |
4. Tests and results

In this section we do comparative tests of the algorithms described in Section 3 on both 2D and 3D data with varying image properties. All tests were carried out on an Intel Xeon CPU 3.60GHz 64-bit Dual core computer with 4 GB RAM running Red Hat Enterprise Linux 5 update 2. The algorithms were compiled with gcc v3.4 using the -O3 optimisation flag and no advantage was taken of the multiple core architecture of the CPU.

4.1. Datasets

The algorithms were evaluated on images of various complexity to mimic the conditions common in image analysis problems. In the 2D case, the performance of the algorithms was compared using the 8-bit grey-level images seen in Figure 4. The image in Figure 4(a) represents noisy images with high complexity and low spatial correlation between pixels; Figure 4(b) is the image pout.tif from MATLAB® (The MathWorks, Natick, MA, USA) and represents images of varying complexity with some spatial correlation; Figure 4(c) represents low complexity images with large uniform areas and, thus, high spatial correlation. Each image in Figure 4 was 960×1164 pixels and are referred to as NOISE, POUT, and BALL, respectively. A test point grid was used for seeded grey-weighted distance transforms, see the black dots shown in Figure 3(c). The test point grid contains 49 points spread evenly, but not symmetrically, over the entire image area. In the 3D case, the performance of the algorithms was compared on an 8-bit gradient magnitude image of the 256×256×100 computed tomography (CT) image covering the liver region of an abdomen, shown in Figure 5(a), and a 512×512×154 contrast enhanced magnetic resonance angiography (CE-MRA) image of an abdomen, shown in Figure 5(b).

The CT image was used by Vidholm et al. [38] to segment livers semi-automatically by seeded FMM. The CT images are abdominal contrast enhanced venous phase CT images of a patient with either carcinoid or pancreatic neoplasms. The images were acquired with a Siemens Sensation 16 CT scanner. The CE-MRA image was used by Vidholm et al. [39] for semi-automated segmentation with haptic guided seeding. The image was acquired from a 1.5T Gyroscan Intera (Philips Medical Systems) using the standard body coil and a specially built table top extender. The sequence was a 3DRF spoiled gradient echo with TR/TE/flip angle=2.6/1.0/30°. The dataset consisted of four subvolumes: the head and upper thorax, the lower thorax and abdomen, the pelvis and upper legs, and the lower legs.

4.2. Precomputed arc weights

When calculating multiple transforms on the same image and the local cost function is complex, it might be favourable to precompute all arc weights to decrease the computational cost of each step. In these cases, precomputing arc weights can generally increase the efficiency at the cost of memory. If the arc weights are bi-directional, the look-up table of a 26-connected image will require roughly 13 times as much memory as the image. However, for grey-weighted distances the cost function is not very complex. Even if there is sufficient memory, the overhead from accessing the look-up table might be of the same magnitude as evaluating the cost function, in part because of cache misses. Table 2 shows a sample of runtimes for two images when using both precomputed weights and when computing the weights on-the-fly.

In Table 2, we see that the total runtime is higher for all cases where precomputed arc weights are used. For the three-dimensional case, the transform itself takes even longer to compute when using precomputed arc weights. There is apparently an overhead associated with using a (large) lookup table for the arc weight which is greater than computing the arc weight on-the-fly. Since the lookup table is memory consuming and the numbers in Table 2 indicate that there is no gain in computation speed, we do not recommend precomputing arc weights and all experiments in Section 4 were calculated the arc weights on-the-fly.

4.3. Different d-heaps

Table 3 lists the runtimes for d-heaps with different d. It is apparent that the binary heap (d = 2) performs best. It should be noted that the implementation differs slightly for the two cases d = 2 and d > 2. For the binary case (d = 2), each element a[i] has children a[2i], a[2i+1], and parent a[⌊(i−1)/2⌋] (the root is a[1] and tree elements are a[1] ... a[n]). See Figure 3(b) for an illustration. For general d-heaps the child nodes are a[d*i+(i−1)+2] through a[d*(i+1)+1], and the parent a[⌊floor((i−2)/d)+1⌋]. Because the d-heap gets shallower with increasing d the insert operation will be faster but the delete-min operation will be more expensive due to the need of d−1 comparisons to find the smallest child node. It appears that for grey-weighted distance transforms, the slower delete-min operations, in combination with the extra multiplications and divisions by d for the parent/child index calculation, makes the binary heap the preferred choice over any d > 2.

4.4. Bucket sizes

Bucket sizes should be chosen so that a good tradeoff is reached between the number of buckets and the number of pixels in each bucket. Dial’s queue defaults to bucket size B = Cm + 1 as mentioned in Section 3.2.4. This assigns one
Figure 4: Test images and their histograms for 2D transforms. (a) NOISE. A noisy image with uniform grey level. (b) POUT. A photograph. (c) BALL. A uniform image with the test point grid.

Figure 5: 3D test images. (a) CT abdomen image of the liver region. (b) CE-MRA abdomen image of the aorta.
bucket to each possible path cost. For the Untidy queue and the hierarchical heap, which both can handle floating point path costs, the right amount of buckets comes from a combination of grey-weighted distance definition, chamfer weight and actual runtime. To ensure a unique bucket for each possible path cost, we have from Equation (9) that the bucket size \( B = B_m + 1 \) must be chosen as,

\[
B = \frac{C_m}{\min(\Delta d)} + 1,
\]

where \( C_m \) is the maximum possible arc weight and \( \min(\Delta d) \) is the minimum possible difference in path cost. Table 4 lists the required number of buckets for each of the grey-weighted distance definition when using 8-bit images and 3-4-5 chamfer weights.

### 4.4.1. Untidy queue

The runtime for different bucket sizes is shown in Figure 6. The DOCS column shows the runtimes for bucket sizes \( B \in [1, 301] \), the GRAYMAT column for bucket sizes \( B \in [1, 11, 21, \ldots, 3001] \), and the WDOCS column for bucket sizes \( B \in [1, 50, 100, \ldots, 16000] \). Each value is the mean runtime over 5 transforms from the same seed.

For GRAYMAT and DOCS we can choose the bucket size required for unique costs (see Table 4) and remain confident that the runtime will be kept low. For WDOCS, the required bucket size is more than 18000. Figure 6 shows that WDOCS has a slight increase in runtime for increasing bucket size. A good choice for bucket size should, thus, be lower than 18000 when considering the runtime. If accuracy of the grey-weighted distance map is not a vital issue, then selecting the number of buckets required for WDOCS can be simplified by quantising the arc weights produced by Equation (3). Here we limit our analysis to non-quantised arc weights. Using fewer buckets than required for unique costs might introduce rounding errors. However, to result in a rounding error two spatially close nodes with different path costs have to end up in the same bucket and get extracted in the wrong order. The probability of this happening is likely to be quite low and depends on both the grey-weighted distance definition and image properties. Figure 7 shows plots of the percent of erroneous values we get in the distance map for different bucket sizes. The plots only show one transform, but it gives an indication that most errors drop off quite fast. To get a more reliable measure on how few buckets can be used without getting an erroneous distance map, we used 5 different seeds from the seed map and recorded the largest bucket size which resulted in an error for each seed. Table 5 lists the mean and standard deviation over the five seeds for each setting. For DOCS we used bucket sizes \( B \in [1, 261] \), and for GRAYMAT and WDOCS we used \( B \in [1, 3, 5, \ldots, 2551] \). The table indicates that bucket sizes much smaller than 18000 can be chosen for WDOCS without introducing errors in the distance transform. Since Figure 6 shows that the choice bucket size for WDOCS is robust with respect to runtime, we use the same number of buckets as was chosen for GRAYMAT (2551) for simplicity.

### 4.4.2. Hierarchical heap

Unlike the Untidy queue, the hierarchical heap always returns the node with the lowest path cost from a find-min operation, independent of bucket size. Therefore, choosing the bucket size for the hierarchical queue can be based on runtime alone. Using less buckets than required for unique path costs will lead to larger heaps since nodes with different costs might have to share buckets. Using more buckets will introduce excessive buckets which cannot get any nodes assigned to them. This will only make the circular array longer and take more time to sift through. The runtime for different bucket sizes is shown in Figure 8.

The DOCS definition shows a decline in runtime which levels out around 250, which makes 261, the bucket size required for unique path costs, the most reasonable choice of bucket size for the hierarchical heap when using DOCS. The GRAYMAT definition behaves similarly to the DOCS definition for NOISE and POUT, decreasing in runtime and levelling out when getting close to the bucket size required for unique path costs. For

### Table 3: Runtimes of d-heaps with different d.

| Dataset | Alg. | Runtime (s) |
|---------|------|-------------|
|         |      | \(d = 2\)  | \(d = 3\)  | \(d = 4\)  | \(d = 5\)  | \(d = 6\)  | \(d = 7\)  |
| NOISE   | H    | 2.35  | 2.40  | 2.43  | 2.44  | 2.45  | 2.46  |
| H 255   |      | 4.08  | 4.34  | 4.31  | 4.19  | 4.20  | 4.21  |
| H 1275  |      | 2.35  | 2.40  | 2.43  | 2.44  | 2.45  | 2.46  |
| BALL    | H    | 2.10  | 2.26  | 2.25  | 2.27  | 2.29  | 2.31  |
| H 255   |      | 3.51  | 3.65  | 3.63  | 3.65  | 3.66  | 3.68  |
| H 1275  |      | 3.16  | 3.31  | 3.28  | 3.31  | 3.32  | 3.32  |

### Table 4: Properties of grey-weighted distance definitions for calculating bucket sizes when using 8-bit images and 3-4-5 chamfer weights.

| Dataset | Alg. | Grayweight | Distances | Buckets (unique cost) |
|---------|------|------------|-----------|-----------------------|
|         |      | \(B_m\)    | \(\Delta d\) | \(B\) |
| NOISE   | H    | 1275       | 0.5       | 2551 |
| H 255   |      | 260        | 1         | 261 |
| H 1275  |      | -255.05    | -0.0137   | >18000 |
| POUT    | H    | 2455(60)   | 0.5       | 2149(290) |
| H 255   |      | 52(0)      | 1         | 52(0) |
| H 1275  |      | 78(0)      | 1         | 78(0) |
| BALL    | H    | 2149(290)  | 0.5       | 2149(290) |
| H 255   |      | 41(13)     | 1         | 41(13) |
| H 1275  |      | 6(7)       | 1         | 6(7) |

### Table 5: Largest bucket size resulting in an erroneous grey-weighted distance transform. The value is the mean of five transforms and the bracketed value is the standard deviation.

| Image   | Alg. | Bucket size |
|---------|------|-------------|
| NOISE   | U    | 254(84)     | 52(0)     | 79(2) |
| POUT    | U    | 2455(60)    | 52(0)     | 85(0) |
| BALL    | U    | 2149(290)   | 52(0)     | 85(0) |

Table 5 shows plots of the percent of erroneous values we get in the distance map for different bucket sizes. The plots only show one transform, but it gives an indication that most errors drop off quite fast. To get a more reliable measure on how few buckets can be used without getting an erroneous distance map, we used 5 different seeds from the seed map and recorded the largest bucket size which resulted in an error for each seed. Table 5 lists the mean and standard deviation over the five seeds for each setting. For DOCS we used bucket sizes \( B \in [1, 261] \), and for GRAYMAT and WDOCS we used \( B \in [1, 3, 5, \ldots, 2551] \). The table indicates that bucket sizes much smaller than 18000 can be chosen for WDOCS without introducing errors in the distance transform. Since Figure 6 shows that the choice bucket size for WDOCS is robust with respect to runtime, we use the same number of buckets as was chosen for GRAYMAT (2551) for simplicity.
Figure 6: Runtimes for grey-weighted distance transforms when using different bucket sizes for the Untidy LIFO queue (top) and FIFO queue (bottom). Note the different scales on the x-axis for the different grey-weighted distance definitions.
BALL, the runtime seems to increase slightly with bucket size. However, the difference in runtime for small versus large bucket sizes is so low that the behaviour for NOISE and POUT is more important. Therefore, the bucket size required for unique path costs (2551) is a good choice for the hierarchical heap when using GRAYMAT. Since Figure 8 indicates that the choice of bucket size for WDOCS is robust with respect to runtime, except for NOISE where it increases slightly for high bucket sizes, we use the same number of buckets as was chosen for GRAYMAT (2551) for simplicity.

4.4.3. Cost spread

Another interesting aspect of the bucket queues is the spread of the queue over time. The spread shows how many nodes occupy each bucket at a specific time during the calculation of the grey-weighted distance transform. Figure 9 shows the spread for a transform, using bucket size $B = 600$, for each image type and each grey-weighted distance definition. The queue was sampled at every 10’000th iteration, and each row represents the state of the queue at the time (from top to bottom), with every pixel being one bucket. The values ranges from dark blue (empty bucket) to red (maximum number of nodes). The $\max$, $\min$, $\text{mean}$, and $\text{std}$ percentages are statistics on the number of non-empty buckets. The figure also shows the mean spread as a bar diagram for each image.

It is clear that the noisier image result in a wider range of arc weights, giving rise to a wider spread, while both POUT and BALL give a low spread for DOCS and WDOCS. This means that for DOCS and WDOCS, a large increase in bucket size will only have a small effect on the number of buckets used, which can be one of the explanations why the runtime does not vary much between different bucket sizes.

The thin spread for DOCS and WDOCS can be utilised by removing the unused part of the circular queue to get an increased spread. A wider spread will likely lead to shallower heaps for small bucket sizes in the hierarchical heap, and decreased rounding errors for small bucket sizes in the Untidy queue, which in turn can lead to shorter runtimes. Note however that a more thorough statistical analysis of the spread needs to be done to draw any conclusions on how much the circular array can be shortened for the different cases. Such an analysis is out of scope for this article.

4.5. Tests on 2D images

Each of the 49 points in the test point grid in Figure 4(c) was used as a seed in a seeded grey-weighted distance transform of each of the 2D images. Table 6 shows the average runtimes in seconds. When running the experiments, it became apparent that the recursive propagation algorithm (labels $P_L$ and $P_{L\lambda}$) expands pixels in an order which is extremely ineffective for grey-weighted distance computations. The pixel removed in Step 4 in Algorithm 1 is always the pixel most recently added to the list $L$. This has the effect that the algorithm starts by expanding pixels along a path from the seed throughout the image, calculating numerous grey-weighted values based on the one seed neighbour at the beginning of the path. Since grey-weighted distances is monotonically increasing from the seed
and depend on the local neighbourhood, expanding its neighbours (which is done at some point when the recursive propagation passes by on its way back for \( P_L \) or has expanded all subsequent pixels in the image for \( P_{LA} \)) will most likely make the initial path calculation redundant. For example, the recursive propagation algorithm \( P_L \) took 4031 seconds to compute one GRAYMAT on a 240\( \times \)291 version of NOISE, visiting each pixel more than 37000 times on average. The chamfer algorithm, in comparison, took 8.16 seconds on average (standard deviation \( \sigma = 1.59 \)) and visited each pixel an average of 42.6 times. Because of this, the recursive propagation algorithms were excluded from any further experiments and do not show up in the table.

In Table 8 we see that the overall best performers were the label-setting algorithm using the binary heap with pointer array (\( H_S \)), the Dial queue (\( D_L \) and \( D_F \)), the Untidy queue (\( U_L \) and \( U_F \)), and the hierarchical heap with pointer array (\( HH_{LA} \)). It is apparent that the label-correcting algorithms are a poor choice due to their habit of visiting each node multiple times. The binary heap with pointer array was the fastest in two out of the nine cases; when calculating the WDOCS transform on the NOISE and POUT images. The hierarchical heap without pointer array was the fastest in two cases; when calculating the GRAYMAT and DOCS transform on the BALL image. The hierarchical heap with pointer array was the fastest in the rest of the cases and tied with the binary heap with pointer array when calculating the WDOCS transform in the NOISE image. The hash functions clearly have too much overhead and the algorithms with hash tables perform worse than they do without any helper structures.

It is apparent that the Fibonacci heap performs worse than the binary heap. The lower, amortised, time complexity of the Fibonacci heap does not beat the binary heap for grey-weighted distance computations. This indicates that the number of \textit{extract-min} and \textit{delete} operations is high relative to the number of other operations performed on the heap.

The differences in runtime between the Dial queue and the Untidy queue are small, which was anticipated considering their similar structure. The performance of the Dial queue is slightly better than the Untidy queue, most likely due to the extra computation required by the Untidy queue to determine which bucket to put a node (Equation (8)). We can conclude that a LIFO list in the Dial or Untidy queue results in a slightly better runtime than a FIFO list. This is likely to stem from the fact that requires a LIFO list requires fewer operations than a FIFO list.

4.6. 

Tests on 3D images

4.6.1. CT

Here we used the CT image with size 256\( \times \)256\( \times \)100, shown in Figure 5 (a), to compare the fastest algorithms from Section 4.5. This was done by computing grey-weighted distance transforms from manually placed seeds in a gradient magnitude image. The gradient magnitude image can be considered a low complexity image with large uniform areas and has high spatial correlation between voxels, i.e., properties similar to the BALL image in Section 4.5. Two transforms were calculated, one from seeds placed outside the liver and one from seeds placed inside the liver, and the average runtime for each setup is shown in Table 7.

Here it becomes apparent that the Dial queue and Untidy queue do not work well with the pointer array since updating a value is associated with a search through a list, which can be quite large due to the limited spread discussed in Section 4.4.3. For any of the grey-weighted distance transforms on the gradient magnitude of a CT image, the fastest algorithm is the hierarchical heap with the pointer array, but only marginally so for the DOCS definition. It is also noteworthy that the binary heap with pointer array performs better than the Untidy queue for both the GRAYMAT and WDOCS definitions.

4.6.2. CE-MRA

Here we used the ‘lower thorax and abdomen’ subvolume with size 512\( \times \)512\( \times \)154, shown in Figure 5 (b), to compare the
Figure 9: Bucket queue spread for each grey-weighted distance definition during a transform when using 600 buckets. Each pixel row shows a snapshot of how many nodes occupy each bucket at every 10000th iteration (from top to bottom). The first pixel in every row is the bucket with the lowest cost nodes, and the colour ranges from dark blue (empty bucket) to red (bucket with the highest number of nodes). The bar diagrams show the mean spread in number of nodes per bucket for each image and grey-weighted distance definition.
Table 6: Running time (in seconds) for grey-weighted distance transforms on the NOISE, POUT, and BALL images. The best times are marked with black colour, times within 10% of the best with dark grey, and times within 40% of best with light grey.

|       | NOISE          |          |          | POUT          |          |          | BALL          |          |          |
|-------|----------------|----------|----------|---------------|----------|----------|---------------|----------|----------|
|       | GRAYMAT        | DOCS     | WDOCS    | GRAYMAT       | DOCS     | WDOCS    | GRAYMAT       | DOCS     | WDOCS    |
| C     | 697.01         | 683.31   | 703.59   | 223.24        | 166.77   | 289.24   | 22.02         | 60.42    | 246.48   |
| P_F   | 23.09          | 19.81    | 21.86    | 156.62        | 14.73    | 13.98    | 27.16         | 12.33    | 12.90    |
| P_FA  | 107.14         | 80.43    | 93.48    | 229.80        | 16.62    | 10.95    | 46.08         | 4.01     | 4.14     |
| H     | 3.05           | 3.54     | 3.89     | 3.19          | 2.58     | 3.32     | 2.36          | 2.09     | 2.74     |
| H_A   | 2.54           | 2.54     | 2.65     | 2.50          | 2.33     | 2.50     | 2.33          | 2.24     | 2.39     |
| H_LIN | 4.02           | 4.23     | 4.35     | 3.93          | 3.62     | 3.67     | 3.41          | 3.37     | 3.41     |
| H_SUM | 3.92           | 4.14     | 4.26     | 3.71          | 3.49     | 3.50     | 3.25          | 3.13     | 3.22     |
| H_PROD | 4.42          | 4.69     | 4.82     | 4.26          | 3.91     | 3.93     | 3.74          | 3.64     | 3.72     |
| H_XOR | 4.55           | 4.88     | 5.10     | 4.31          | 3.95     | 3.93     | 3.61          | 3.59     | 3.61     |
| F     | 4.47           | 5.08     | 5.72     | 4.57          | 3.53     | 4.79     | 3.33          | 2.82     | 3.91     |
| F_A   | 3.55           | 3.46     | 3.74     | 3.42          | 3.09     | 3.45     | 3.18          | 2.94     | 3.27     |
| F_SUM | 5.12           | 5.02     | 5.32     | 4.66          | 4.35     | 4.59     | 4.22          | 3.93     | 4.15     |
| D_L   | -              | 3.22     | -        | -             | 2.43     | -        | -             | 1.98     | -        |
| D_F   | -              | 3.29     | -        | -             | 2.49     | -        | -             | 2.02     | -        |
| D_LA  | -              | 2.65     | -        | -             | 2.58     | -        | -             | 2.13     | -        |
| U_L   | 2.82           | 3.28     | 3.51     | 2.96          | 2.48     | 3.07     | 2.28          | 2.02     | 2.60     |
| U_F   | 2.83           | 3.29     | 3.52     | 2.96          | 2.54     | 3.09     | 2.27          | 2.05     | 2.60     |
| U_LA  | 2.47           | 2.74     | 2.68     | 2.49          | 2.63     | 2.62     | 2.27          | 2.17     | 2.47     |
| HH    | 2.75           | 3.14     | 3.57     | 2.88          | 2.35     | 3.20     | 2.20          | 1.89     | 2.63     |
| HH_A  | 2.39           | 2.38     | 2.65     | 2.39          | 2.18     | 2.51     | 2.21          | 2.05     | 2.36     |

Table 7: Runtime (in seconds) for grey-weighted distance transforms on the CT and CE-MRA images. The best times are marked with black colour, times within 10% of the best with dark grey, and times within 40% of best with light grey.

|       | CT                       |          |          | CE-MRA                  |          |          |
|-------|--------------------------|----------|----------|-------------------------|----------|----------|
|       | GRAYMAT                  | DOCS     | WDOCS    | GRAYMAT                 | DOCS     | WDOCS    |
| H     | 50.86                    | 54.13    | 86.28    | 257.97                  | 183.30   | 385.54   |
| H_A   | 37.49                    | 49.23    | 54.77    | 152.28                  | 163.12   | 232.13   |
| F     | 55.62                    | 62.25    | 104.59   | 306.65                  | 218.62   | 499.10   |
| F_A   | 39.36                    | 54.31    | 65.54    | 169.56                  | 188.18   | 263.24   |
| D_L   | -                        | 44.70    | -        | -                       | 180.81   | -        |
| D_F   | -                        | 44.48    | -        | -                       | 189.98   | -        |
| D_LA  | -                        | 563.30   | -        | -                       | 6067.59  | -        |
| U_L   | 37.69                    | 43.81    | 61.84    | 191.38                  | 183.69   | 278.04   |
| U_F   | 37.56                    | 42.52    | 60.84    | 193.33                  | 190.76   | 281.31   |
| U_LA  | 2111.36                  | 541.08   | 287.86   | 270.88                  | 6603.72  | 3012.72  |
| HH    | 38.45                    | 44.22    | 69.22    | 195.96                  | 184.31   | 315.37   |
| HH_A  | 31.84                    | 42.19    | 49.03    | 127.88                  | 167.81   | 212.52   |
4.6.3. Dynamic vs static priority queues

In addition to the dynamic versions of the Dial and Untidy heap with pointer array, that use STL List containers, both priority queues were also implemented using static arrays to give an indication of the tradeoff between static and dynamic priority queues. As mentioned in Section 4.6.1, a static array implementation has a high programming complexity compared to using the STL List. It also requires at least twice as much memory as the dynamic map while a dynamic priority queue only requires enough space to accommodate the front wave (sometimes referred to as the ”narrow band”). Table 8 lists the runtimes for the binary heap with pointer array (H), Dial queue (D), Dial queue using static array (DS), Untidy queue (U), Untidy queue using static array (US), and hierarchical heap with pointer array (HH). The static array implementations of the Dial and Untidy queue show an increase in performance by ~ 5-33% compared to their dynamic implementations but only ~ 0-11% compared to the fastest algorithms with dynamic priority queues. Note that when calculating the DOCS transform on the CE-MRA, the static array implementations ran slower than both the binary heap with pointer array and the hierarchical heap with pointer array. Considering the drawbacks with implementing static array priority queues, these runtimes indicate that dynamic priority queues can provide a fair tradeoff between performance and programming complexity/memory usage.

5. Conclusion

The performance variations do not motivate a different choice of algorithm for different grey-weighted distance definitions, i.e., if an algorithm performs well for one definition, it is likely to perform well for the other two. The same holds for the difference in image properties. For 2D images, the hierarchical heap using pointer array shows to be the best choice. The same choice also proves to be the best for 3D images (~ 0-16% faster than the second best). However, if memory is a critical issue, i.e., if the pointer array helper structure is not an option, then the Dial queue is the best option if we work with integer costs, and the Untidy queue if we work with real valued costs. It is noteworthy that the popular binary heap with pointer array performs quite well compared to the more sophisticated priority queues (~ 0-19% slower than the fastest algorithm).

6. Acknowledgements

The Department of Radiology, Uppsala University Hospital, is acknowledged for providing the CT and CE-MRA datasets. Stina Svensson, Robin Strand, Cris Luengo, and Filip Malmberg, Centre for Image Analysis, Uppsala, Sweden, are acknowledged for scientific support. Magnus Gedda is financially supported by the Swedish Research Council (project 621-2005-5540).

References

[1] I. B. Bloch. Geodesic balls in a fuzzy set and fuzzy geodesic mathematical morphology. Pattern Recognition, 33(6):897–906, 2000.
[2] I. Bloch. On fuzzy distances and their use in image processing under imprecision. Pattern Recognition, 32(11):1873–1895, 1999.
[3] G. Borløfors. Applications using distance transforms. In Aspects of Virtual Form Processing, pages 83–108, 1994.
[4] G. Borløfors. Distance transformations in digital images. Computer Vision, Graphics, and Image Processing, 34:344–371, 1986.
[5] G. Borløfors. On digital distance transforms in three dimensions. Computer Vision and Image Understanding, 64(3):368–376, 1996.
[6] C. M. Carvalho, G. T. F. Gomez, and T. Y. Kong. Algorithms for fuzzy segmentation. Pattern Analysis and Applications, 2:73–81, 1999.
[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. The MIT Press, Cambridge, Massachusetts, USA, 1990.
[8] E. Coto, S. Grimm, and D. Williams. O-Buffer based IFT watershed from markers for large medical datasets. Computers & Graphics, 31(6):848–863, 2007.
[9] R. B. Dial. Algorithm 360: shortest-path forest with topological ordering. Communications of the ACM, 12(11):632–633, November 1969.
[10] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269–271, 1959.
[11] A. X. Falco, J. K. Udupa, and F. K. Miyazawa. An ultra-fast user-steered image segmentation paradigm: Live wire on the fly. IEEE Transactions on Medical Imaging, 19(1):55–62, January 2000.
[12] C. Fouard and M. Gedda. An objective comparison between gray weighted distance transforms and weighted distance transforms on curved spaces. In Proceedings of the 13th International Conference on Discrete Geometry for Computer Imagery, volume 4245 of Lecture Notes in Computer Science, pages 259–270. Springer, October 2006.
[13] M. Gedda and S. Svensson. Fuzzy distance based hierarchical clustering calculated using the A* algorithm. In Proceedings of the 11th International Workshop on Combinatorial Image Analysis, volume 4040 of Lecture Notes in Computer Science, pages 101–115. Springer, 2006.
[14] L. Ikonen. Pixel queue algorithm for geodesic distance transforms. In Proceedings 12th International Conference on Discrete Geometry for Computer Imagery, pages 228–239, 2005.
[15] M.W. Jones, J.A. Baerentzen, and M. Sramek. 3D distance fields: a survey of techniques and applications. IEEE Transactions on Visualization and Computer Graphics, 12(4):581–599, 2006.
[16] N. Josuttis. The C++ Standard Library. Addison Wesley, 1999.
[17] R. Kimmel, N. Kiryati, and A. M. Bruckstein. Sub-pixel distance maps and weighted distance transforms. Journal of Mathematical Imaging and Vision, 6:223–233, 1996.
[18] G. Levi and U. Montanari. A grey-weighted skeleton. Information and Control, 17:62–91, 1970.
[19] C. L. Luceno Hendriks. Revisiting priority queues for image analysis. Submitted for publication, January 2009.
[20] F. Malmberg, E. Vidyholm, and I. Nyström. A 3D live-wire segmentation method for volume images using haptic interaction. In Proceedings International Conference on Discrete Geometry for Computer Imagery, volume 4245, pages 663–673, 2006.
[21] F. Meyer. Topographical distance and watershed lines. Signal Processing, 38:113–125, 1994.
[22] L. Nyul, A. X. Falcao, and J. K. Udupa. Fuzzy-connected 3D image segmentation at interactive speeds. Graphical Models, 64:259–281, 2003.
[23] P. Preuteux and N. Merlet. New concepts in mathematical morphology: the topographical distance functions. In Proceedings SPIE, volume 1568, pages 66–77, 1991.
[24] C. Rasch and T. Satzger. Remarks on the O(N) implementation of the fast marching method. IMA Journal of Numerical Analysis, 29(3):806–813, 2009.
Table 8: Runtime (in seconds) for priority queues using static arrays (grey rows) compared to using dynamically allocated queues.

|          | CT              |              |              | CE-MRA           |              |              |
|----------|-----------------|--------------|--------------|------------------|--------------|--------------|
|          | GRAYMAT DOCS    | WDOCS        |              | GRAYMAT DOCS    | WDOCS        |              |
| H_A      | 37.49 49.23     | 54.77        | 152.28       | 163.12 232.13    |              |
| D_L      | - 44.70 -       | - - 180.81   | - - 169.10   |                  |
| D SL     | - 41.62 -       | - - 169.10   | - - 169.10   |                  |
| U_L      | 37.69 43.81     | 61.84        | 191.38       | 183.69 278.04    |              |
| U SL     | 30.34 41.67     | 44.36        | 127.38       | 171.11 191.77    |              |
| HH_A     | 31.84 42.19     | 49.03        | 127.88       | 167.81 212.52    |              |

[25] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the Association for Computing Machinery*, 13(4):471–494, October 1966.

[26] D. Rutovitz. Data structures for operations on digital images. In *Pictorial Pattern Recognition*, pages 105–133, Washington, 1968. Thompson.

[27] P. K. Saha, F. W. Wehrf, and B. R. Gomberg. Fuzzy distance transform: theory, algorithms, and applications. *Computer Vision and Image Understanding*, 86(3):171–190, 2002.

[28] J. Serra. Mathematical morphology for boolean lattices. *Image Analysis and Mathematical Morphology*, 2:37–58, 1988.

[29] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996.

[30] J. Silvela and J. Portillo. Breadth-first search and its application to image processing problems. *Image Processing, IEEE Transactions on*, 10(8):1194–1199, Aug 2001.

[31] N. Sladoje and J. Lindblad. High-precision boundary length estimation by utilizing gray-level information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(2):357–363, 2009.

[32] P. Soille. Generalized geodesy via geodesic time. *Pattern Recognition Letters*, 15(12):1235–1240, 1994.

[33] P. J. Toivonen. New geodesic distance transforms for grey-scale images. *Pattern Recognition Letters*, 17(5):437–450, 1996.

[34] J. K. Udupa and S. Samarasekera. Fuzzy connectedness and and object definition: Theory, algorithms, and applications in image segmentation. *Graphical Models and Image Processing*, 58:246–261, 1996.

[35] B. H. Verwer and B. J. H. Verwer. Shading from shape, the eikonal equation solved by grey-weighted distance transform. *Pattern Recognition Letters*, 11:681–690, October 1990.

[36] B. H. Verwer, P. Verbeek, and S. Dekker. An efficient uniform cost algorithm applied to distance transforms. *IEEE Transactions of Pattern Analysis and Machine Intelligence*, 11(4):425–429, April 1989.

[37] B. H. Verwer. Local distances for distance transformations in two and three dimensions. *Pattern Recognition Letters*, 12:671–682, 1991.

[38] E. Vintholm, S. Nilsson, and I. Nyström. Fast and robust semi-automatic liver segmentation with haptic interaction. In *Proceedings of MICCAI 2006 Lecture Notes in Computer Science*, volume 4191, pages 774–781, 2006.

[39] E. Vintholm, X. Tizon, I. Nyström, and E. Bengtsson. Haptic guided seeding of mra images for semi-automatic segmentation. In *Proceedings 2nd IEEE International Symposium on Biomedical Imaging*, pages 288–291, 2004.

[40] L. Yatziv, A. Bartesaghi, and S. Guillermo. O(n) implementation of the fast marching algorithm. *Journal of Computational Physics*, 212:393–399, 2006.

[41] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.