ELSEVIER
Theoretical Computer Science 154 (1996) 165–181
Theoretical Computer Science

Two linear time Union–Find strategies for image processing

Christophe Fiorio[a,*], Jens Gustedt[b,1]

[a] *LIRMM, UMR 9928 Université Montpellier II/CNRS, 161 rue Ada, 34392 Montpellier Cedex 5, France*
[b] *Technische Universität Berlin, Sekr. MA 6-1, Strasse des 17. Juni 136, D-10623 Berlin, Germany*

Received June 1994; revised October 1994
Communicated by M. Nivat

Abstract

We consider Union–Find as an appropriate data structure to obtain two linear time algorithms for the segmentation of images. The linearity is obtained by restricting the order in which Union's are performed. For one algorithm the complexity bound is proven by amortizing the Find operations. For the other we use periodic updates to keep the relevant part of our Union–Find-tree of constant height. Both algorithms are generalized and lead to new linear strategies for Union–Find that are neither covered by the algorithm of Gabow and Tarjan (1984) nor by the one of Dillencourt et al. (1992).

1. Introduction and overview

An important problem in image processing is to capture the essential features of a scene. One way to do that is to extract (hopefully) significant regions from the image. The technique for extraction used in this paper is *region growing* first described in [13]. It consists of starting with the smallest regions (i.e. pixels or points of the image) and merging them until they are considered to be optimal. The merging criterion is some oracle that should guarantee the significance of the newly created region. The specification of such oracles is not the subject of this paper – for practical purposes we have chosen some classical threshold function.

As has already been observed by Dillencourt et al. [3], region growing as defined above leads naturally to the *disjoint set union problem*, Union–Find for short. Union–Find in general is not known to have a linear time solution. The best complexity known has been first obtained by an algorithm of Tarjan, see [15], that has been shown to perform in $O(\alpha(n,m)m)$ where α is a very slowly growing function

* Corresponding author. Email: fiorio@lirmm.fr.
[1] Supported by a postdoctoral grant of the Graduiertenkolleg Algorithmische Diskrete Mathematik. Part of this work was done during a visit to the LIRMM of the second author that was funded by the French government. Email: gustedt@math.tu-berlin.de.

0304-3975/96/$09.50
SSDI 0304-3975(94)00262-2

and $n < m$ are the amounts of calls to a Union and Find operation, respectively. In [16, 17, 2] it has been proven that this bound is sharp for some classes of pointer machines and recently this has been generalized to general pointer machines by La Poutré in [6]. Whether or not an algorithm with better complexity on a random access machine might exist is not known until now. If the sequence of Union and Find operations is restricted there are algorithms due to Gabow and Tarjan [4] and Dillencourt et al. [3] that perform in linear time.

Both types of algorithms are not well suited for our purposes: the first – apart from being nonlinear – has tremendous constants of proportionality in the known bound on the complexity; the algorithms of the second type are either too restrictive or do not leave room for generalizations.

In this paper we consider two different variants of Union–Find that solve region growing and then give generalizations of them. For both we give an algorithm that *performs in linear time*. They use classical scanning strategies as used for example in [12] for a preprocessing step. The first algorithm scans the image line by line. For each line, we examine each pixel and we see if we can merge it with the two regions to the left and above. After we have processed a particular line we rescan it in a post-process to maintain our data structures accordingly. The second algorithm, in its recursive variant, assumes that the image is an $(\sqrt{n} \times \sqrt{n})$-square and proceeds by dividing it into 4 subsquares of size $\sqrt{n}/2 \times \sqrt{n}/2$. After coming up from recursion the regions in the 4 subsquares are merged together along the common boundary; i.e. for every pair of neighboring pixels that belong to different subsquares we perform a Union on the corresponding regions if our decision oracle tells us so. This algorithm leads easily to a parallelization.

The linear time complexity of the first algorithm is due to the fact that we are able to keep the tree of our data structure that is constructed for each region flat. The linearity of the second is proven by amortizing the Find operation. From both it is possible to deduce a generic scheme of algorithms that solve restricted Union–Find's in linear time. The first generalizes to a so-called IntervalUnionFind where the sets that are allowed for Union and Find operations form antichains of an interval order. This scheme is applied to solve a similar problem on planar graphs in linear time, too. The second generalizes to EquilibratedUnionFind where certain restrictions on the size of the sets obtained are required. It leads to linear algorithms for data of higher dimensionality, e.g. spatial bitmaps.

Both algorithms have been implemented for two-dimensional bitmaps. The theoretical efficiency translates very well into *short running times*; in fact we achieve practical *real time interpretation* of the image on today's workstation, and as shown in Fig. 2 the results are well suitable even with the simple oracle chosen.

2. Basics of Union–Find

The general Union–Find problem, or more precisely the disjoint set union problem, can be formulated as follows. Given is a set S, the groundset, of elements, pixels in our

application, that form one-element subsets at the beginning. The goal is to perform arbitrary sequences of Union and Find operations in the best time complexity possible. Here a Union works on two disjoint subsets fusing them into one; a Find identifies the subset a certain element belongs to. For an introduction and overview to Union–Find see e.g. [10]; for recent results see [9].

In the following we will only assume a straightforward implementation of Union–Find that could easily be implemented on an arbitrary pointer machine. In fact there exist versions of Union–Find that are much more sophisticated, see e.g. [15] or [7], that perform in time $O(\alpha(n, m)m)$ and are thus optimal on pointer machines for the general case where no restrictions to the Union's or Find's apply. There is also a version that performs in linear time on a special case, first shown to work well when implemented on a random access machine, see [4], and then generalized to pointer machines in [9].

For these algorithms it is necessary to determine a tree of the elements in advance such that all subsets form connected subtrees of that tree at any time of the algorithm. For the application considered here this is not adequate because e.g. the number of pairs of elements that may form two-element subsets would only be $n - 1$ where it should be about $4n$ when considering all neighboring pairs of the matrix.

For our purposes it is sufficient that every set is represented by a rooted tree of the members, the root being the unique representative of the set. This can e.g. be done by giving each element a pointer to another element, the parent in the tree. Find identifies the root of the set by an iterative pointer search. The Union of two sets is done by linking the root of one set to the root of the other one. The choice of which element to link and of which to remain a root will be specified differently for each algorithm.

The cost of both operations, Union and Find, is dominated by the number of pointer jumps of a Find operation. We say that an element has direct access to its region if it is linked directly to the root of the tree.

2.1. Flattening the Union–Find-tree

We give a simple refinement of the Find operation that will be helpful for some special cases; for an example see Fig. 1.

Algorithm 1. FindCompress(p)

(1) **if** *isTop*(p) **then return** p

(2) **else return** $p.parent :=$ FindCompress($p.parent$)

We have

(2.1) After a call FindCompress(p) all elements on the path from p to the root have direct access to the root.

(2.2) FindCompress(p) performs with at most l pointer jumps where l is the length of the path from p to the root.

Suppose now that we have an arbitrary subset S_0 of the groundset such that

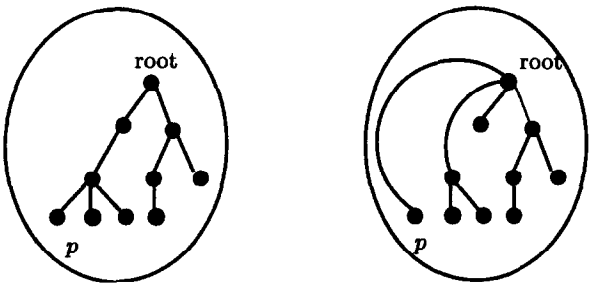

Fig. 1. Updating the representation of a set.

(2.3) every $p \in S_0$ has direct access to the root of its corresponding tree.

Suppose in addition that we perform some arbitrary Find and Union operations exclusively on the set S_0. Clearly after several Union operations (2.3) might be violated. But then we may perform FindCompress(p) for all $p \in S_0$ which we denote by Flatten(S_0) and we get:

(2.4) After *Flatten*(S_0) all elements of S_0 have direct access to their region.

(2.5) *Flatten*(S_0) performs with at most $2|S_0|$ pointer jumps.

3. Image segmentation by merging regions

A major problem in image processing and particularly in scene analysis is to describe information compactly and to capture the essential feature of a scene. An approach is segmentation, e.g. dividing the image into regions (see Fig. 2). Several techniques for image segmentation have been described, see e.g. [5] for an overview. In this paper we are concerned with region growing. This approach searches for areas of the image presenting some homogeneous features.

In Fig. 2 we show two examples of segmented images each produced with the algorithms presented here. The images on the left-hand side show the original images, the middle ones show the borders of the regions obtained and the left ones show the images that result if we replace the original grey tone of each individual pixel with the average value of its particular region.

There are two dual approaches to region segmentation: the split and the merge, see e.g. [14]. In this paper we are working with the merge technique. It consists of starting with the smallest regions (i.e. pixels or points of the image) and merging them until they are optimal. This scheme is also called Region Growing.

The result and the complexity of this grouping depends much on the order in which the merging operations are done. Some criteria we want all grouping strategies to fulfill are the following:

(3.1) Every pair of neighboring pixels should only be considered at least once but at most a bounded number of times.

Fig. 2. Examples of segmentation.

This is to guarantee a linear number of questions to our grouping criteria and to ensure that no artificial borders between objects remain.

(3.2) The size of the regions should be equilibrated during the algorithm.

This is to avoid that certain regions dominate artificially before others had a chance to constitute themselves. This requirement excludes some simple graph searching techniques as e.g. depth-first search.

A commonly used representation is the region adjacency graph (RAG), proposed by Zucker [19]. This representation associates a vertex to each region and links two vertices with an edge if the two corresponding regions are adjacent. So region growing is the process of joining neighboring vertices into one, subject to some conditions as the predicate Oracle. For practical purposes this has the disadvantage that a relatively complicated data structure for the RAG must be maintained. This in general leads to algorithms with nonlinear complexity as in [12]. Our approach is a little different. We consider a region as a set of pixels and instead of grouping two vertices into one, we group them into a set of vertices. Thus we are led to the Union–Find problem.

3.1. Incorporating the oracles

For the overall complexity of a segmentation algorithm that uses Union–Find it will not only be important to perform Union's and Find's efficiently but also to guarantee that the oracle used will increase the complexity only by some factor. For

the simple oracles that we used this is easily achieved; they are threshold functions on

(1) the absolute difference between the average colors,[2]

(2) the difference between the minimum and maximum color of a potentially created region,

(3) the variance of the color values of a potentially created region,

and any combinations of these. Such oracles can be calculated in constant time per call if at every Union operation the minima, maxima, sums of the color values and sums of the squares of the color values are maintained properly.

4. A line by line strategy

In the following we will describe an algorithm that we denote ScanLine. A similar algorithm for a related problem, namely finding the connected components in a black and white image, also running in linear time has been developed by Dillencourt et al. [3]. Besides that it uses a quiet involved data structure for Union–Find, it does not lead to the same generalization as ours, namely planar graphs, as will be given below.

4.1. Scanning a raster image

ScanLine scans the image line by line and applies Union–Find on the encountered regions. For the following let us assume that the image is a $w \times l$ rectangle.

Algorithm 2. ScanLine

Input: A bitmap *bm* of size $w \times l$

```
(1)  special treatment of the first line
(2)  for i := 2 to l do begin
(3)      special treatment of the first pixel of line i
(4)      for j := 2 to w do begin
(5)          left := FindCompress(bm[i, j − 1]);
(6)          up := FindCompress(bm[i − 1, j]);
(7)          this := FindCompress(bm[i, j]);
(8)          if Oracle(left, this) then this := Union(left, this);
(9)          if Oracle(up, this) then Union(up, this);
(10)         end
(11)     Flatten(line i)
(12)     end
```

[2] All definitions given here are formulated for grey-scaled images. It is easy to see that they can be generalized to real color images when considering e.g. each color plane separately.

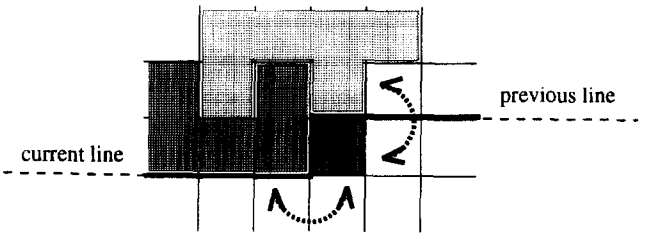

Fig. 3.

At the beginning all the regions consist of exactly one pixel. Initially we process on the first line, then line by line regarding the previous one (see Fig. 3). For each line, we examine each pixel and see if we can merge it with the two regions corresponding to the pixels to the left and above. Clearly in the first line we only deal with the pixel to the left. After we have processed a particular line we rescan it and make a call to Flatten for the set of pixels of this line.

To guarantee the overall complexity, Union links the region that occurred first on the line to the later one. This can easily be realized by a counter that is incremented for each new region. Since Union is done by linking one region to the root of the other we may assume that Union is performed in constant time. As a result we have Theorem 4.1.

Theorem 4.1. *Algorithm ScanLine touches every pair of neighboring pixels and performs in linear time.*

It is easy to see that every pair of pixels is touched. To prove the complexity we need Proposition 4.2 and Lemma 4.3.

Proposition 4.2. *At the beginning of the process on a line, each pixel of the previous line has direct access to its region.*

Proof. This is guaranteed by invariant (2.4) of Flatten. □

With Proposition 4.2 we are able to prove the next lemma:

Lemma 4.3. *For each FindCompress realized when processing a line we have to do at most* 4 *pointer jumps.*

Proof. At the beginning all the pixels of the line and the previous one have direct access to their regions. When one of the regions consists simply of one pixel and a Union is necessary, we only have to add the pixel to the region. Things get more complicated when we need to realize the Union of two regions each including more than one pixel since the depth of the tree increases.

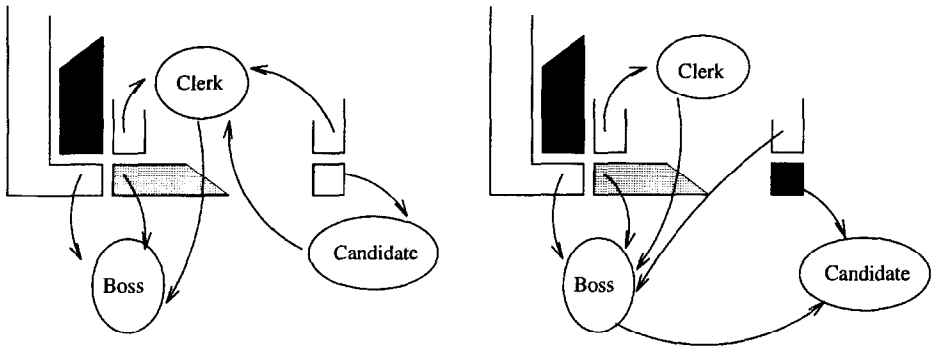

Fig. 4. The two possibilities to have 3 links.

Now we show that we will never meet a pixel which needs more than one extra pointer jump to find the root of its region. Therefore let us suppose that we just merged two regions, denote them *Boss* and *Clerk*. If we have to do a Union between this newly formed region and another one, *Candidate* say, we will risk creating a chain of length 3 in the tree. As we see in Fig. 4, there are only two possibilities[3] to do that: either linking *Candidate* directly to *Clerk* or linking the *Boss* directly to *Candidate*.

The first case is impossible. Indeed, before doing a Union we always do a FindCompress and perform a Union operation only on the root of the regions. Since these are *Boss* and *Candidate*, *Clerk* will never be involved directly in such a Union. In the second case all pixels which are linked directly to *Clerk* need 3 pointer jumps to retrieve the root of their region, i.e. *Candidate*. But we will never meet such pixels when continuing on this particular line: since *Boss* was linked to *Candidate*, the later occurred first on the line. Furthermore *Candidate* is connected, so it is surrounding *Boss* and *Clerk*, see Fig. 5. Therefore we will never meet any pixel linked directly to the other two regions. So with invariant (2.2) we have at most 4 pointer jumps for each FindCompress. □

Proof of Theorem 4.1. We will assume that the cost of the algorithm is dominated by the number of pointer jumps. First we scan the line and perform 2 FindCompress's for each pixel: one for the pixel above and one for the pixel to the left. Moreover we perform at most 2 Unions, but these are realized in constant time and do not use pointer jumps.

We will now compute the total number of pointer jumps. Flatten is repeated on each line, so with invariant (2.5) we can compute its total cost: $2 \cdot w \cdot l = 2n$. For each pixel we make at most 2 pointer jumps for each of the two FindCompress's, so in total $4 \cdot w \cdot l = 4n$. Overall the number of pointer jumps is $6n$. □

[3] Note that you can reverse *Boss* and *Clerk* in Fig. 4 without changing the argumentation.

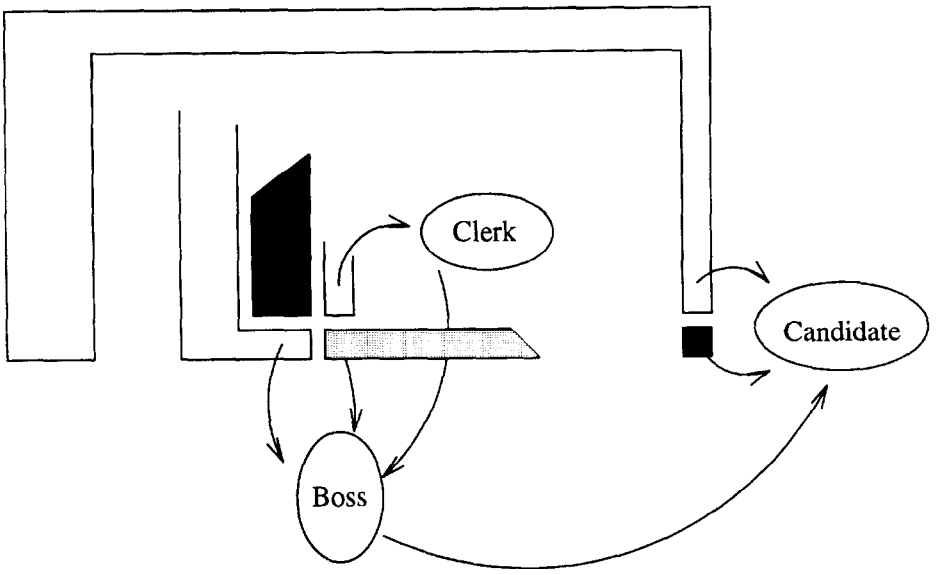

Fig. 5.

4.2. A generalization to planar graphs

In our algorithm it was important that we did not increase the distance of elements still to come on a line from their regions, and that we were able to update the whole line in a second run via a call to Flatten. A generic algorithm IntervalUnionFind that captures these features is

(4.1) **for** $i := 1$ **to** l **do** *Generate* E_i; Flatten $X(E_i)$; *Process* E_i.

Here every E_i is a set of pairs of elements subject to a possible Union and $X(E_i)$ denotes the set of elements involved in these pairs. We require

(4.2) $X(E_i) \cap X(E_j) \subseteq X(E_k)$ for all $i < k < j$ and

(4.3) *Process* E_i performs Union and Find operations exclusively on the Union–Find sets of the elements in $X(E_i)$.

Observe that (4.2) means that the sets $X(E_i)$ a particular element belongs to appear consecutively. Thus we may associate an interval to each element that represents the period in our algorithm during which we have the right to access it. Invariants (2.4) and (2.5) then translate into:

(4.4) At the beginning of *Process* E_i in IntervalUnionFind each element of $X(E_i)$ has direct access to the root of its region.

(4.5) The running time for all calls to Flatten in IntervalUnionFind is $O(\sum_{i=1}^{l} |X(E_i)|)$.

Now suppose we have a planar graph $G = (V, E)$ that is equipped with some data on the vertices and where we want to perform a similar task as segmentation, i.e. where we want to cluster vertices into connected regions according to some

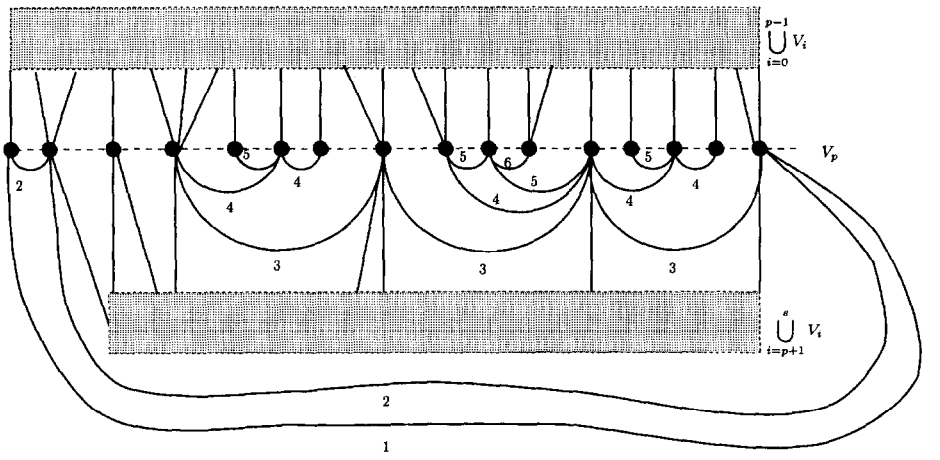

Fig. 6.

homogeneity criterion on the data. We assume that the graph is given together with a combinatorial embedding and has a designated outer face. We may then find a shelling $V_0, \ldots, V_s$ of the graph as follows. Let V_0 be some consecutive part of the outer face and $V_p = N_G(V_{p-1}) \setminus \bigcup_{j=1}^{p-1} V_j$, i.e. the sets of equal distance to V_0. We may assume that the V_p are given as lists in the natural order prescribed by the embedding.

Algorithm 3. PlanarShelling

Input: Planar Graph $G = (V, E)$, shelling $V_0, \ldots, V_s$

(1) **for** $p := 1$ **to begin**
(2) **for all** $v \in V_p$ **do** Process all edges joining v and $\bigcup_{j=0}^{p-1} V_j$
(3) Process internal edges of V_p
(4) **end**

Here phase (2) connects the vertices of V_p to $\bigcup_{j=0}^{p-1} V_j$ in the natural order. This part is a direct generalization of ScanLine to this situation and does not need further explanation. Now in phase (3) we have to process all internal edges of V_p, i.e. that have both endpoints in V_p. Let E_v denote the set of such internal edges with one endpoint being vertex $v \in V_p$, see Fig. 6. We may assume that all vertices of V_p lie on a line, all vertices yet processed lie above that line, and all vertices still to come lie below. Observe also that there are no internal edges crossing that line since everybody is connected to the part above the line, so V_p is outerplanar.

Now let E_1 be the lower cover of the set of internal edges, i.e. that are visible from below. Remove E_1 and obtain a new lower cover E_2. Repeat this procedure until no internal edges remain and collect the edges in sets $E_1, \ldots, E_l$ and let $V(E_i)$ be the set of vertices being endpoint of an edge in E_i.

In the figure the numbers at the edges indicate the set E_i they belong to. The E_i have the following properties:

(4.6) Every E_i is a collection of paths.

(4.7) For every $v \in V_p$ there is an interval $[l_v, r_v]$ s.t. $E_v \cap E_i \neq \emptyset \Leftrightarrow l_v \leqslant i \leqslant r_v$.

Clearly the interval in (4.7) may also be empty. Now we may process the internal edges by starting with E_l and proceeding with E_{l-1} and so on.

Algorithm 4. ProcessInternalEdges

Input: Sets $E_1, \ldots, E_l$ of edges that fulfill (4.6) and (4.7).

(1) **for** $i := l$ **downto** 1 **do begin**
(2) Flatten($V(E_i)$)
(3) scan E_i from left to right
(4) **End**

Theorem 4.4. *PlanarShelling runs in linear time.*

Proof. With what is said above it is clear that all phases (2) of PlanarShelling together run in linear time.

Each particular ProcessInternalEdges fulfills the requirements for IntervalUnion-Find and, moreover, the same topological argument as above ensures that we do not have to follow long chains of references to find a root of a particular region. So provided we are able to generate the sets E_i of edges in linear time each such phase also runs in linear time. But this is easy to achieve, since the internal edges of V_p may be seen as a system of parentheses and the levels edges belong to can be found by a scan from left to right. □

5. A divide and conquer strategy

Now we are going to present an algorithm that will also perform in linear time, but has the additional feature that it allows a straightforward parallelization. To get a good upper bound of its complexity it will be necessary to amortize the Find operation over the complete run of the algorithm; the number of pointer jumps for a particular Find might well be logarithmic and not constant any more. To achieve logarithmic time for every Find we use a variant of the Union operation, the so-called weighted union rule, that always links the smaller region to the larger one. Because of that choice we have the following invariant, see e.g. [10], that we will need later:

(5.1) Every Find operation can be done with $\log s$ pointer jumps
where s is the cardinality of the set in question.

5.1. The recursive algorithm

For the following algorithm we assume that the image is an $(\sqrt{n} \times \sqrt{n})$-square, $\sqrt{n}$ a power of 2, and proceed recursively by dividing it into 4 subsquares of size $\sqrt{n}/2 \times \sqrt{n}/2$. After coming up from recursion the regions in the 4 subsquares are

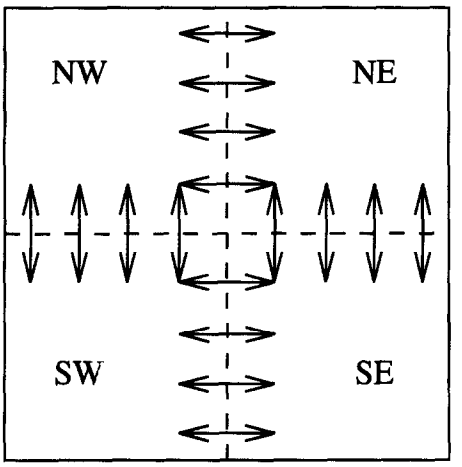

Fig. 7.

merged together along the common boundary; i.e. for every pair of neighboring pixels that belong to different subsquares we perform a Union on the corresponding regions if our oracle tells us so.

For the following formulation of the algorithm we assume that we have easy access to the four subsquares of our bitmap (see Fig. 7). If the bitmap is called *bm* we denote by $bm[NW]$ the northwestern submatrix, by $bm[NE]$ the northeastern, etc.

Algorithm 5. MergeSquares

Input: An integer k and a bitmap *bm* of size $2^k \times 2^k$

```
(1)  if k = 0 then return;
(2)  h⁻ := 2^(k-1) − 1; h⁺ := 2^(k-1);
(3)  for DIR := NW to SE do MergeSquares(bm[DIR], k − 1);
(4)  for i := 0 to 2^k do begin
(5)      left := Find(bm[i, h⁻]); right := Find(bm[i, h⁺]);
(6)      if Oracle(left, right) then Union(left, right);
(7)      end
(8)  for i := 0 to 2^k do begin
(9)      up := Find(bm[h⁻, i]); down := Find(bm[h⁺, i]);
(10)     if Oracle(up, down) then Union(up, down);
(11)     end
```

Theorem 5.1. *MergeSquares touches all neighboring pairs of pixels of the bitmap and performs in total in linear time.*

Proof. It is easy to see that MergeSquares visits all neighboring pairs exactly once. For the complexity let us analyze a call to MergeSquares for the size of the square being $2^k \times 2^k$. We have 4 recursive calls and $2 \times 2^k = 2^{k+1}$ possible merging operations.

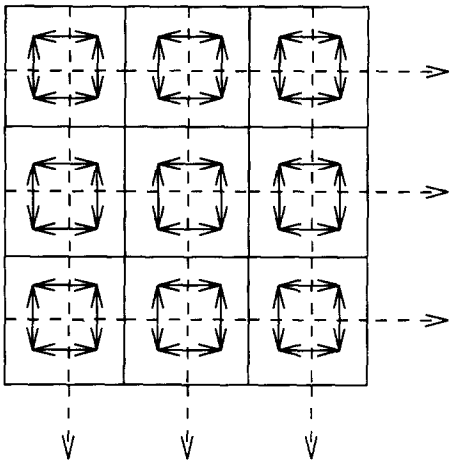

Fig. 8.

We will assume that the cost of such a merging operation is dominated by the length of the pointer jumps to perform when looking for the roots of the corresponding sets via two Find's.

The regions that might be merged together have a size bounded by $2^k \times 2^k$. So by (5.1) we know that each such merging operation needs at most $2 \log 2^{2k} = 4k$ pointer jumps. So in total we perform with at most $8k2^k$ pointer jumps.

If we fix k for a moment, this is done $n/(2^{2k})$ times for subsquares of size $2^k \times 2^k$. So for all such subsquares we need at most

$$8k2^k n/(2^{2k}) = 8(k/2^k)n \tag{5.2}$$

pointer jumps. In total all pointer jumps are now bounded by

$$\sum_{k=1}^{\log\sqrt{n}} 8(k/2^k)n < 8n \sum_{k=1}^{\infty} k\tfrac{1}{2}^k. \tag{5.3}$$

For $-1 < x < 1$ we have the well-known identity

$$\frac{x}{(1-x)^2} = \sum_{k=1}^{\infty} kx^k. \tag{5.4}$$

This can e.g. easily be seen when expanding the function $x/(1-x)^2$ in a Taylor series at 0. Thus the right-hand side of (5.3) evaluates to $16n$ which is linear in the size n of our bitmap. □

5.2. *An iterative formulation*

If we look at all merging operations that are done on a specific recursion level l we see a characteristic pattern (see Fig. 8). That is if we cover the whole bitmap with all $2^l \times 2^l$ subsquares we see that the pairs of pixels that are possibly subject to a Union

operation on level l are found in only some specific rows and columns; it is easy to see that all Unions to do on level l are for pairs of the form $((i,j),(i+1,j))$ where $i = r\cdot 2^l + 2^{l-1}$ for $r = 0,\ldots,\sqrt{n}/2^l$ and $j = 1,\ldots,n$ and of the form $((i,j),(i,j+1))$ where $i = 1,\ldots,n$ and $j = r\cdot 2^l + 2^{l-1}$ for $r = 0,\ldots,\sqrt{n}/2^l$. So we may perform all these Unions in an arbitrary order. In fact by starting with level $l = 1$ we may proceed iteratively and obtain an algorithm that works on arbitrary rectangular bitmaps.

5.3. *An adequate memory mapping for parallelization*

For an efficient processing of the image it might be important to parallelize the algorithm MergeSquares. From the definition of the algorithm we get easily

Proposition 5.2. *MergeSquares can be implemented efficiently on a PRAM with $p < 2^k$ processors such that it runs on a bitmap of size $2^k \times 2^k$ in $O(2^{2k}/p)$ time.*

To implement MergeSquares efficiently it is important that

(1) we do not copy the data needed for a recursive call,

(2) a routine on a lower level of recursion should not have to take care of the global size of the data, and

(3) we map the matrix linearly into memory such that every call has its data in a consecutive part.

Then all recursive calls can be performed completely independent of each other.

This can be realized by a recursive "quad tree" definition of the matrix: first mapping submatrix NE, then NW, etc., see above. What we get in fact by such a recursive definition is that pixel (i,j) is mapped in position $i_{k-1}j_{k-1}\cdots i_0 j_0$ if the binary representations of i and j are $i_{k-1}\cdots i_0$ and $j_{k-1}\cdots j_0$, respectively, i.e. the index of a particular matrix element in the linear memory is given by alternating the bits of i and j. For an example of a (4×4)-matrix A mapping into a vector v of length 16 consider Fig. 9. Here we map e.g. the element $(2,2) = (10b, 10b)$ of matrix A (indices starting with 0) onto the element $1100b = 12$ of vector v. Now some juggling with bit-operations makes it easy to perform the merging phase of the algorithm sequentially.

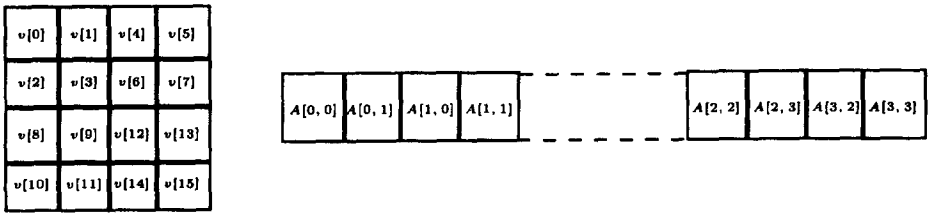

Fig. 9. Mapping a matrix to a vector.

5.4. A generalization to higher dimensions

For MergeSquares it was important that we had the guarantee that the regions would not grow too fast, and that the number of Find's to perform was small compared to the possible size of the regions. This is best generalized by introducing a logarithmic number of phases. If each phase i guarantees

(1) no subset is larger than $O(2^i)$ and

(2) no phase makes more than $O(2^{\log n - i})$ Find operations,

then the complexity is bounded by

$$O\left(\sum_{i=1}^{\log n} \log 2^i \cdot 2^{\log n - i}\right) = O\left(\sum_{i=1}^{\log n} i \cdot 2^{-i} \cdot n\right) = O(n). \tag{5.5}$$

We call such a strategy EquilibratedUnionFind. One application of EquilibratedUnionFind could be the case where the elements are considered to be vertices of a planar graph G of bounded maximum degree. Then balanced separator techniques could be used to obtain again a linear time Union–Find strategy. We do not go into further details since this problem is already covered by PlanarShelling. Another more important application is to bitmaps of higher dimensions, e.g. spatial data. If we denote the natural generalization of MergeSquares to dimension d by MergeOctants_d we easily get

Theorem 5.3. *Let d be some fixed dimension, then MergeOctants_d runs in linear time.*

6. Notes on implementation

Both algorithms have been implemented straightforwardly in C^{++}. They show suprisingly good results both in the quality of the segmentation as well as in time performance. Compiled with $g++$, the C^{++} compiler of the GNU project, we achieve a running time of about 12 μs per pixel. For example, for a small image with 256×256 pixels like the boat in Fig. 2 we had a processing time of 0.8 s. Certainly these times will improve when the implementation becomes more sophisticated or if the algorithms are realized on an appropriate hardware.

Even more surprising for the authors than the running time has been the quality of the segmentation. In order to reduce the data to be considered both algorithms originally were thought to form a preprocessing step to some other treatment. But seeing that the output is already competitive we believe now that they can be immediately followed by an interpretation step that tries e.g. to group regions into objects. One indication that our approach reaches the limits of what can be achieved with segmentation by itself is that iterating the algorithms does not change the picture very much.

For example, if we apply the divide and conquer algorithm on the boat in Fig. 2 several times we obtain the regions shown in Fig. 10. On the left we see what is given

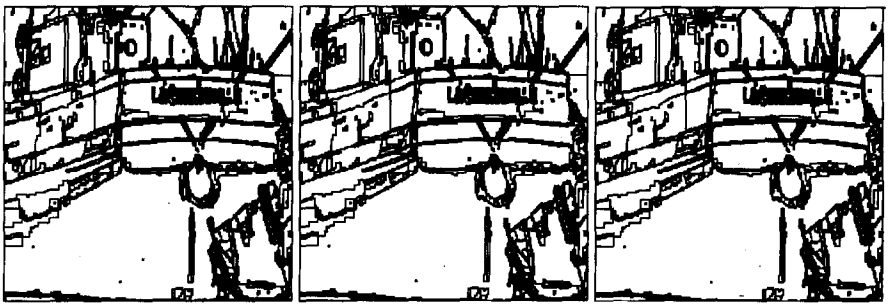

Fig. 10. Three iterations of the segmentation algorithm.

by the first iteration (3223 regions), in the middle the second (2622 regions) and on the right the fourth (2574 regions). Then in any further iteration no additional improvement is made and the situation stabilizes with that number of regions. So the most important reduction of the complexity from 65 536 regions (=number of pixels) to 3223 is already done in the first iteration.

References

[1] A. Aggarwal et al., eds., *Proc. 1st Ann. ACM–SIAM Symp. on Discrete Algorithms* (SIAM, Philadelphia, PA, 1990).

[2] L. Banachowski, A complement to Tarjan's result about the lower bound on the complexity of the set union problem, *Inform. Process. Lett.* **11** (1980) 59–65.

[3] M.B. Dillencourt, H. Samet and M. Tamminen, A general approach to connected-component labeling for arbitrary image representations, *J. ACM* **39** (1992) 253–280, Corr. pp. 985–986.

[4] H.N. Gabow and R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *J. Comput. System Sci.* **30** (1984) 209–221.

[5] R.M. Haralick and L.G. Shapiro, Survey: image segmentation techniques, *Comput. Vision Graphics Image Process.* **29** (1985) 100–132.

[6] J.A. La Poutré, Lower bounds for the union–find and the split–find problem on pointer machines, RUU-CS-89-21, 1989.

[7] J.A. La Poutré, New techniques for the union–find problem, RUU-CS-89-19, 1989.

[8] J.A. La Poutré, New techniques for the union–find problem, [1] (1990) 54–63, extended abstract of [7].

[9] J.A. La Poutré, Dynamic graph algorithms and data structures, Ph.D. Thesis, Rijksuniversiteit te Utrecht, 1991.

[10] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching* (Springer, Berlin, 1984).

[11] K. Mehlhorn and A. Tsakalidis, Data structures, in [18], Ch. 6, pp. 301–314.

[12] O. Monga and B. Wrobel-Dautcourt, Segmentation d'images: vers une méthodologie, *Traitement du Signal* **4** (1987) 169–193.

[13] J.L. Muerle and D.C. Allen, Experimental evaluation of techniques for automatic segmentation of objects in a complex scene, in: G.C. Cheng et al., eds., *Pictorial Pattern Recognition* (Thompson, Washington, 1968) 3–13.

[14] M. Popovic, F. Chantemargue, R. Canals and P. Bonton, Several approaches to implement the merging step of the split and merge region segmentation, in: F.H. Post and W. Barth, eds., *EUROGRAPHICS'91* (Elsevier, Amsterdam, 1991) 399–412.

[15] R.E. Tarjan, Efficiency of a good but not linear set union algorithm *J. ACM* **22** (1975) 215–225.

[16] R.E. Tarjan, A class of algorithms which require non-linear time to maintain disjoint sets, *J. Comput. System Sci.* **18** (1979) 110–127.

[17] R.E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. ACM* **31** (1984) 245–281.

[18] J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity* (Elsevier, Amsterdam, 1990).

[19] S.W. Zucker, Survey: region growing: childhood and adolescence, *Comput. Graphics Image Process.* **5** (1976) 382–399.