Technische Universität Berlin

FACHBEREICH 3
MATHEMATIK

MEMORY MANAGEMENT FOR UNION-FIND ALGORITHMS

by

CHRISTOPHE FIORIO[1] JENS GUSTEDT[2]

No. 523/1996

Memory Management for Union-Find Algorithms

Christophe Fiorio[1] and Jens Gustedt[2]

Technische Universität Berlin, Sekr. MA 6-1, D-10623 Berlin, Germany

Abstract. We provide a general tool to improve the real time performance of a broad class of *Union-Find* algorithms. This is done by minimizing the random access memory that is used and thus to avoid the well-known von Neumann bottleneck of synchronizing CPU and memory. A main application to image segmentation algorithms is demonstrated where the real time performance is drastically improved.

1 Introduction

A main obstacle for really efficient implementations of random access data structures on todays computers is still the so-called von Neumann bottleneck. It addresses the fact that background memory of such a computer usually has an access time that is much larger than the cycle rate of the CPU. In particular it states that a computation that randomly accesses data elements over and over again (eg by pointer jumping) mainly has to **wait** and so the CPU will be idle most of the time – `nop` is certainly the assembler instruction that is among the most executed ones.

Modern architectures try to circumvent this problem by introducing a hierarchical memory model, consisting of registers, cache, RAM and disk. In particular cache that has a cycle rate comparable with the CPU is used to hold those data elements that are suspected to be accessed in the near future. Whereas most programmers seem to be sensible to the problem that occurs when data located in virtual memory must be fetched from disk they seem to be less aware of the problem that arises when the data transfer between RAM, cache and CPU must be handled.

Clearly such a hierarchical architecture is useless if it is not supported by appropriate software that allows easy estimates on what elements to load into cache and/or registers. Our goal here is to provide such a scheme for *Union-Find* (UF) data structures and algorithms.

The best way to control which data elements are going to be used next by an algorithm is when data is accessed sequentially, i.e when data is consecutively read or written into an array or (pseudo) file. We model this in distinguishing to different kinds of memory access, *random* and *sequential*, and give a tool to reduce the memory that is accessed randomly to a neglectable portion.

For a UF algorithm $\mathcal{A}$ we say that an element v of the groundset is **open** from the moment in time it is accessed for the first time until $\mathcal{A}$ *knows* that it is accessed for the last time. For an instance I, by $\mathit{Width}_{\mathcal{A}}(I)$ we denote the maximum cardinality of the set of open elements dur-

[1] Supported by DIMANET. Current address: LIRMM, F-34392 Montpellier Cedex 5, France

[2] Supported by the IFP “Digitale Filter”

ing the run of $\mathcal{A}$ on I. For an integer n let $Width_{\mathcal{A}}(n)$ denote the maximum of $Width_{\mathcal{A}}(I)$ taken over all instances I of size n. Our main result is given by the following theorem.

Main Theorem. *Any UF algorithm $\mathcal{A}$ may be implemented in such a way that the amount of random access memory needed during the execution of $\mathcal{A}$ is bounded by $O(Width_{\mathcal{A}}(n))$.*

The main idea is an improvement of the techniques found in [2], namely to delete elements that are not further needed for a UF algorithm and to reuse the memory that was occupied by them. For practical purposes any access of arrays using only *pointer increments* (or decrements) and no other pointer arithmetics can be subsumed under being a sequential access. We use that fact to save those deleted elements into a (pseudo) *file*, that allows us to reconstruct the final partition of the ground set after having done all necessary *Union* operations. This technique applies to a broad class of algorithms for which we show how to improve them in a straight forward manner.

We tackle the UF problems by modeling their access to individual elements by a *graph*; the elements of the set are identified with vertices of the graph and edges represent permissible *Union* operations. We assume that an algorithm that uses a UF data structure has an estimation of a time interval for each element in which this element might be accessed. This is equivalent of saying that an interval supergraph (or **path decomposition**) of such graph is given as well. This is introduced and developed in Section 3.

Path decompositions are then used in Section 4 to prove the Main Theorem. In addition we also give a technique that after all *Union*s being performed allows a reconstruction of the final partition of the groundset.

Our main application, introduced in Section 2.3 and developed in Section 5, are UF algorithms for image segmentation. Here we are able to combine and extend results of [2, 3, 6] to achieve linear time algorithms with a very low random access memory demand that already have proven to perform very well in practice, [4].

2 Basic Definitions and Facts

2.1 Basics of Union-Find

Union-Find algorithms solve the disjoint set union problem. It can be stated as follows: let S be a set of elements that form one-element subsets at the beginning, perform a sequence of ***Union*** operations on these subsets; ***Find*** operation identifies for one element the set it belongs to (for a more general presentation see e.g [8, 5, 16]). This must be done in a more efficiently way, but algorithms that solve the *Union-Find* problem in the general case are not known to have a linear time solution. The best complexity known has been obtained by an algorithm of McIllroy and Morris that has been shown to perform in $O(\alpha(m,n)m)$ by Tarjan, [13], where α is a very slowly growing function and $n < m$ are the amount of calls to an *Union* and *Find* operation respectively. This bound has been proven to be sharp for some classes of pointer machines [14, 1, 15] and extended to general pointer machines by La Poutré in [7].

Efficient implementations represent sets by trees, the root of a tree being the unique representative of the set. This can, e.g, easily be done by giving each element e a pointer to another element in the same set, its parent in the tree, denoted e.`par`. The *Union* of two sets is realized by linking

Algorithm 1: *FindComp(p)*

```
if p is the root of the tree then
    return p
else
    p.par ← FindComp(p.par )
    return p.par
```

the root of one tree to the root of the other one. *Find* identifies the root of the tree, i.e the unique representative of the set, by an iterative pointer search. In the following we will always assume that any algorithm doing UF will work with such a representation, we will refer to such an algorithm as tree *Union-Find* algorithm, **TUF** for short. Clearly the cost of a TUF algorithm is dominated by the number of pointer jumps of *Find* operations. In *Union* operations, the choice of which root to link and which to remain a root has an influence on the number of pointer jumps to be done for future *Find* operations and so on the overall complexity of the algorithm. Usually one links the smaller tree under the bigger one, the so-called "*weighted union rule*".

There is also an commonly used refinement of the *Find* operation, see Algorithm 1. Clearly after *FindComp(p)*, all elements on the path from p to the root have direct access to the root, i.e are linked directly to the root. Let S_0 be an arbitrary subset and let *Impl*(S_0) be S_0 together with those elements that lye on the path from any element $s \in S_0$ to its root. We denote by *Flatten*(S_0) the operation that consists of applying *FindComp* to each element of S_0. We get the following statements.

Remark 2.1. *After Flatten(S_0) all elements of S_0 have direct access to their root and $|Impl(S_0)| \leqslant 2 \cdot |S_0|$.*

Proposition 2.2. *Flatten(S_0) performs with at most $2 \cdot |Impl(S_0)|$ pointer jumps.*

Proof. Observe that any element $s \in$ *Impl*(S_0) is used for pointer jumps in two different roles:

(a) *FindComp* is called directly on s. This occurs at most once.
(b) s is found on the path of some element to its root. This can occur as often as s is parent pointer for other elements *at the beginning*.

Clearly that each of (a) and (b) sums up in total to at most $|Impl(S_0)|$. □

In this paper, we will show that, given an *Union-Find* algorithm, we can improve its practical efficiency by a better memory management thanks to *Flatten*. This will be verified with an application to image processing. In fact, we consider a particular type of *Union-Find* problem: the so-called *Union-Find* with graphical restrictions.

2.2 Union-Find with graphical restrictions

Not only that UF appears as a subproblem of many algorithmic graph problems, see [9], graphs can also be quite useful to *model* algorithmic features of UF problems, as we intend to prove in this paper.

Union-Find problem with graphical restriction can be defined as follows: given is a graph G, vertices are element of the set S and *Union*'s can only be done according to the edges, i.e at each step of the algorithm, subsets are obtained form connected subgraphs of G. For such a problem, the **maximal sets** are the connected subgraphs obtained after several *Union* and *Find* operations when the process of UF is considered to be terminated.

In addition we assume that each demand for an *Union* operation is explicitly given by an edge joining the corresponding sets. The problem of eventually finding such an edge is not part of the UF problem itself.

This problem has been studied in [6] and shown to be linear for several classes of graphs, trees and partial k-trees, for any fixed parameter k, d-dimensional grids for fixed d and 8-neighborhood graphs of a 2-dimensional grid, and planar graphs.

2.3 Image segmentation and Union-Find

Image segmentation can be seen as an attempt to capture the essential features of a scene, i.e an image. One way to do that is to extract significant regions from an image. One possible technique of extraction is **region growing**, first described in [10]. It consists of starting with the smallest possible regions, i.e pixels, 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. Clearly the specification of such oracles is a matter of its own rights and can not be the subject of this paper.

As has already been observed by Dillencourt et al. in [2], region growing as defined above *leads naturally* to the *Union-Find* problem. In addition to usual UF the problem of image segmentation requires also that each set is connected. UF with graphical restriction easily models that situation: choose an appropriate grid as underlying graph, where vertices of the grid are pixels of the image and edges denote the adjacency relation. The maximal sets represent the regions of the image. Moreover, due to the *Find* operation, we are able to tell for each pixel the region it belongs to. So in terms of image processing a "connected component labeling" has

Algorithm 2: Image Segmentation with Union-Find

Data : A grid *bm* of size $w \times l$

Result : A segmented image with connected components labeling

```
special treatment of the first line
for i = 2 to l do
    Flatten(line i − 1 of bm)
    special treatment of the first pixel
    for j = 2 to w do
        left = FindComp(bm[j, i − 1])
        up = FindComp(bm[j − 1, i])
        current = FindComp(bm[j, i])
        if Oracle(left,current) then
            Union(left,current)
        if Oracle(up,current) then
            Union(up,current)
```

been realized in the same time as the segmentation by the UF algorithm.

In [3] segmentation algorithms using *Union-Find*, and as well an extension to a restricted version of the *Union-Find* problem on planar graphs, have been shown to perform in linear time. In Algorithm 2 we give an implementation of *Union-Find* applied to segmentation of 2-dimensional image. As already said above, *Oracle* is a criterion to make the decision whether or not to merge the two sets (regions in this case). This algorithm uses a special rule to decide which tree must be linked to the other one when an *Union* is realized: the sets seen the first on a given line is always linked under the one seen later on the line. For a proof of the linearity see [3].

The image processed can be very large, so there is a lot of data (about 1 million elements for a 1024×1024 image), and an efficient memory management has a direct influence on the practical efficiency of the algorithm. In Algorithm 2 *Flatten* is necessary to achieve the linear complexity, but as we will see in Section 3 it is also useful for a better memory management. In the general case we will show that applying *Flatten* on some particular elements and at some particular moments allows a better memory management without increasing the complexity of *Union-Find* algorithm and so gives a better practical efficiency. These results are emphasized by some experimental results in image processing given in Section 5.

3 Memory Management by Path Decompositions

3.1 Random versus Sequential Memory Access

As already addressed in the introduction our main issue is to avoid the von Neumann bottleneck for UF algorithms. There we must provide a tool to ease an estimation for compilers and operating systems which data elements are going to be used next.

The best way to control which data elements are going to be used next by an algorithm is when data is accessed sequentially, i.e when data is consecutively read or written into an array or (pseudo) file. If done so, modern operation systems and compilers perform quite well in optimizing the performance since they are capable to shuffle entire blocks of memory between cache, RAM and disk.

For the context of this paper we propose thus a distinction between two types of access to data and thus to the memory that is used to store it:

RAM random access memory, a part of the memory that may be accessed in a unpredictable way, and

SAM sequential access memory, a part of the memory that we only access sequentially in a predictable way, i.e e.g as a *stack* or a *file*.

For practical purposes any access of arrays using only *pointer increments* (or decrements) and no other pointer arithmetics can be subsumed under the SAM model.

Our goal in this paper is to optimize the use of memory under these aspects and thus improve the real time performance of TUF algorithms.

3.2 UF in phases

To allow minimization of RAM we assume that a virtual algorithm $\mathcal{A}$ requiring TUF operations proceeds in phases, $1, \ldots, \ell$ say. In fact the main idea is, for a given phase, to recycle as much memory as possible that was used previously.

We will assume that the amount of memory that is potentially accessed randomly in each phase is the resource that we want to minimize. In fact such an approach is not a restriction to the TUF algorithms that are to be used since we may introduce a phase for each *Union* and *Find* operation. On the other hand it allows to combine consecutive *groups of such operations* in order to improve the behavior of certain algorithms.

We make the assumption that for each phase i a set V_i' of **active** elements is known, i.e a set of elements for which phase i will possibly do *Find* operations[3]. In our example, Algorithm 2, the set of active elements are e.g. pairs of consecutive lines in the image. We also assume that each individual element must only be created once and freed later on. To cover that we call an element $v \in V$ **open** in phase i if

[3] Including those *Find*'s that are needed to perform a *Union*.

there are $j \leqslant i \leqslant k$ such that v is active for both phase j and k. Note that every active element is open as well. By V_i we denote the set of open elements for phase i.

V_i is the least set of elements that any TUF algorithm $\mathcal{A}$ with the same sets of active elements has to administrate at phase i if in addition it is only allowed to create and free elements once. So $\max_{1 \leq i \leq \ell} |V_i|$ measures the minimal amount of RAM needed by each such algorithm.

3.3 Path Decompositions

Since our overall goal is to solve UF problems with graphical restrictions we now develop a tool that turns out to combine the underlying graph with the idea of doing TUF in phases: *path decompositions*. These originally have been developed in the framework of the Graph Minor Project of Robertson and Seymour, see [11, 12] for the original definition.

Two key observations lead us there; the first is that for TUF running in phases a set V_i forms a separator of the graph between those elements that already have been processed and those that are not yet touched. The second is that for every element $v \in V$ the indices i such that $v \in V_i$ are consecutive numbers.

Formally a path decomposition $V_1, \ldots, V_\ell$ of a graph $G = (V, E)$ is given by the following requirements

(1) $\bigcup_{1 \leq i \leq \ell} V_i = V$
(2) For all $e \in E$ there is i such that $e \subseteq V_i$.
(3) For all $v \in V$ and all $j \leq i \leq k$ with $v \in V_j$ and $v \in V_k$ we also have that $v \in V_i$.

In our context Requirement (2) covers the fact that the endpoints of an edge that might be used for a *Union* operation must be active simultaneously in some phase.

The notion of path decomposition is closely related to interval extensions of the graph G: blowing up each set V_i to a clique defines such an extension. On the other hand a consecutive clique arrangement of an interval extension is easily checked to verify the necessary conditions of a path decomposition. For the graphical TUF problem in phases such an interval extension thus adds those edges to the graph that still would give rise to exactly the same sequence of sets of open elements as G.

The ***Width*** [4] of a path decomposition is $\max_{1 \leq i \leq \ell} |V_i| - 1$. For a TUF algorithm $\mathcal{A}$ we denote with $\mathit{Width}(\mathcal{A})$ the width of the path decomposition corresponding to the sequence of open sets. As we have seen above this parameter is of particular interest in our context — it measures the least amount of elements that our algorithm $\mathcal{A}$ must keep in RAM. Below, Corollary 3.2, we will see that in fact it always can be realized up to a (small) constant factor.

A lot of efforts are made for several theoretical and practical applications to keep the width of path decompositions as small as possible, i.e to chose a particular path decomposition of a graph that minimizes the width. Often one aims to bound this parameter by a constant for graph classes of particular interest. But our situation is much better: we are not seeking to minimize it but only to keep it inside certain bounds in terms of the size of the graph.

3.4 Bounding the Memory Requirement

Our algorithms use tree data structures to represent the current subsets of the UF process. So if we don't want to exceed $\mathit{Width}(\mathcal{A})$ by more than a linear factor we have to be careful how many elements are used as internal nodes of some UF tree.

[4] The "-1" is included historical reasons only.

For $\mathcal{A}$ call *Impl*$_{\mathcal{A}}(V_i)$ the total set of elements that are either open for i or accessed during a possible *Find* operation of one of the open elements. Clearly, for a given phase i, we only need to keep the elements of *Impl*$_{\mathcal{A}}(V_i)$ in RAM. So all the elements that don't belong to *Impl*$_{\mathcal{A}}(V_i)$ can be deleted and the memory space they used can be recycled.

In fact an easy estimation for $\mathcal{A}$ is given by the following proposition.

Proposition 3.1. *Let $\mathcal{A}$ be a TUF strategy that runs in ℓ phases with open sets $V_1,\ldots,V_\ell$ such that before each phase $i > 1$ Flatten(V_{i-1}) is invoked. Then*

$$|Impl_{\mathcal{A}}(V_i)| \leq 2 \cdot |V_i|$$

and the total amount of additional work introduced by the calls to Flatten is linear in $\sum_{1\leq i\leq \ell} |V_i|$.

Proof. For (3.1) just observe that when starting phase i after the Flatten every active element, i.e element of V_i, has direct access to the root of its tree[5] and so the elements that are accessed during phase i itself are at most the active ones and these roots.

The estimation of the work now follows easily with Proposition 2.2. □

If we assume that for each phase i we are able to delete all elements not in *Impl*$_{\mathcal{A}}(V_i)$ Proposition 3.1 leads us to the following corollary:

Corollary 3.2. *Let $\mathcal{A}$ be as in Proposition 3.1. Then the amount of RAM used by $\mathcal{A}$ for the UF data structure is linear in Width($\mathcal{A}$).*

[5] All new elements of $V_i \setminus V_{i-1}$ introduced after *Flatten*(V_{i-1}) are one-element sets and so have direct access to their root, i.e themselves

Observe that estimating the additional work in terms of the *open* elements instead of the *active* ones may already overshoot the budget given by the complexity of $\mathcal{A}$. So for a particular algorithm that produces much more open elements than active ones it might be interesting to refine the ideas given so far to call *Flatten* on active elements only. We will see an example where this is possible in Section 5.

4 Union-Find Algorithm With Memory Management

From Corollary 3.2 we know that the memory needed by a TUF in phase algorithm $\mathcal{A}$ can be linear in *Width*($\mathcal{A}$). This corollary holds only if, for a given phase i, we are able to delete all elements not in *Impl*(V_i). In the following we present an implementation of such an algorithm. Then in Section 4.2 we show that the data what is freed can in fact be mapped into a SAM in order to be able to reconstruct the maximal sets. Then in Section 4.3 we prove a linear time bound for the additional work introduced by the memory management. Thus the global complexity of final Algorithm 5 remains.

4.1 Implementation of a low memory consuming TUF

To achieve our goal we must be able to delete all elements not in *Impl*(V_i), for a given phase i. This problem can be decomposed into two sub-problems:

1. How to know that a given element *does not belong* to *Impl*(V_i).
2. How to ensure that *all* such elements are deleted.

In order to solve the first point we provide each element e with a counter, denoted e.`impl`.

Algorithm 3: *Decr(e,n)*

Input: an element e and an integer n.

$e.\texttt{impl} \leftarrow e.\texttt{impl} - n$
if $e.\texttt{impl} = 0$ **then**
  delete e

Invariant 1. *For a given phase i and for a given element e, $e.\texttt{impl}$ records the number of elements, below it in the tree , including itself, that belong to Impl(V_i).*

Now with Invariant 1 we know that, for a given phase i, a given element e does not belong to *Impl(V_i)* if $e.\texttt{impl}$ is equal to 0.

Invariant 2. *An element e with $e.\texttt{impl} = 0$ is immediately deleted.*

Provided Invariant 1 and Invariant 2 are guaranteed we get easily the following invariant:

Invariant 3. *For a given phase i, all elements that do not belong to Impl(V_i) are deleted.*

Note that Invariant 2 is verified if each time a value is subtracted from `impl` we check its value and delete the corresponding element if $\texttt{impl} = 0$. So we add the new function ***Decr*** , see Algorithm 3.

If we only use *Decr* to subtract a value from `impl`, Invariant 2 is verified.

In order to verify Invariant 1, we must check that:

1. every time trees are modified,
2. and every time an element will not be active anymore,

`impl` is well maintained.

The first point occurs only during *Union-Find* operations, i.e *Union*, *FindComp* or when a new element is created. For *Union* it is sufficient to add to the new root the `impl` value of the root linked to it. When a new element is created we just have to initialize `impl` to the value of 1. The only difficulty comes from *FindComp* which messes up the tree. Algorithm 4 shows an appropriate update of *FindComp*.

Algorithm 4: *FindComp(p)* with update of `impl`

1 **if** *p is the root of the tree* **then**
  return p
else
2.1 $\texttt{origin} \leftarrow p.\texttt{par}$
2.2 $p.\texttt{par} \leftarrow$ *FindComp($p.\texttt{par}$)*
2.3 **if** $p.\texttt{par} \neq \texttt{origin}$ **then**
    Decr($\texttt{origin}$,$p.\texttt{impl}$)
3 **return** $p.\texttt{par}$

It is easy to see that if Invariant 1 is verified before a *FindComp*, it remains after. Indeed the only place where the tree can be modified is at line 2.2. At line 2.3 we check if this really happens. In fact if the parent has changed, the old parent has "lost" all the elements of *Impl(V_i)* below p. This number of elements is recorded by $p.\texttt{impl}$ since Invariant 1 is assumed to be verified before doing the *FindComp*. So we just have to subtract this value to the value of `impl` of the old parent. This is realized by the instruction *Decr($\texttt{origin}$,$p.\texttt{impl}$)*.

Observe that by now we already have shown that *Flatten* also maintains `impl` properly if this new version of *FindComp* is used.

So we have proven that every time trees are modified, `impl` is well maintained. Now we must update `impl` of elements that will no more be active. This happens only during the transition between two phases. Clearly, all such elements are in $V_{i-1} \setminus V_i$. So at the beginning of a given

Algorithm 5: TUF with optimizing memory

let $\mathcal{A}_i$ be the part of $\mathcal{A}$ done during in phase i

1 $\mathcal{A}_1$
2 **for** $i = 2 \cdots \ell$ **do**
3   *Flatten(V_{i-1})*
4   **foreach** $e \in V_{i-1} \setminus V_i$ **do**
5   **if** *e is not the root of the tree* **then**
6   *Decr(e.*`par`*,1)*
7   *Decr(e,1)*
8     $\mathcal{A}_i$

phase i we must decrement the value of `impl` of all these elements by one. Of course this operation will be done by a call to *Decr*. But changing the value of `impl` of a given element implies to update the value of `impl` of all the elements above in the tree. After *Flatten(V_{i-1})* we know that all elements of $V_{i-1} \setminus V_i$ have direct access to its root. Thus for a given element $e \in V_{i-1} \setminus V_i$, in the same time we decrement e.`impl`, we just have to decrement the value of `impl` of e.`par`. This is realized by lines 4, 5, 6 and 7 of Algorithm 5.

We just have proven that Invariant 1 remains true along the algorithm, Invariant 2 remains true too, since we always call *Decr* to reduced the value of `impl` of a given element. Algorithm 5 implements these solutions and thus respects Invariant 3. So it is an implementation of $\mathcal{A}$ according to Corollary 3.2.

Since the amount of necessary RAM is reduced, one can expect that all the data is allocated consecutively, or at least on the same page of memory. So this reduces the number of page faults of large scale applications drastically. But minimizing the amount of necessary RAM is not only useful for such large applications: for medium sized applications a great part of the data now fits into the memory cache of the CPU. So the practical efficiency will then be greatly improved for such applications as well.

4.2 Mapping data into sequential memory

Algorithm 5 presented above allows to minimize the amount of RAM necessary to a TUF algorithm by deleting elements that are not necessary anymore, and by reusing the memory that was freed. But UF is generally a part of a more global process and after running an *Union-Find* algorithm, one may need to access the maximal sets. We now present a way to reconstruct the maximal sets while keeping the amount of necessary RAM linear in *Width($\mathcal{A}$)*.

The principle of the method is to save informations about the deleted elements on a stack. Clearly a stack can be implemented on a SAM and so it doesn't use RAM. At the end of the process, unstacking the information saved should allow to reconstruct the maximal sets. The only information we need is the element itself and the current root of its tree. We must also ensure that this root is always saved after all elements below it. So it will be unstacked first and the tree will be easily reconstruct when unstacking the elements.

This can easily realized in Algorithm 5. Note that elements are deleted, either in a *FindComp*, or when updating elements of $V_{i-1} \setminus V_i$. In a *FindComp* the root of the tree is known, and updating elements of $V_{i-1} \setminus V_i$ occurs just after *Flatten(V_{i-1})*, so either the element itself or its parent is a root.

So when deleting an element we know the root of the tree. We just have to check if roots are always deleted after elements below them and so delete operation can be replaced by a stack-in (push) operation. In *FindComp* of Algorithm 4 we never call *Decr* on a root element, so a root will never be deleted during a *FindComp*. But when updating elements of $V_{i-1} \setminus V_i$, no assumption is done on the order the elements are processed. In particular a root can be deleted before elements below it. So to guarantee that roots will be stacked after leaves we just have to check the elements two times: the first time we update an element only if it is a leaf, the second time we update the roots. The delete operation can thus be replaced by a push operation which saves the element and the name of its root on a stack (SAM).

The efficiency of the future memory accesses to the maximal sets can be improved if elements of the same set are consecutive in RAM. This can be easily achieved. Indeed the number of elements of a set can be recorded in the root. So when saving a root, one can stack also this number. Thus the necessary amount of memory can be allocated in a whole, when reconstructing the maximal sets.

4.3 Complexity of the additional work

First of all, let us recall that the total amount of additional work introduced by the calls to *Flatten* is linear in *Width*($\mathcal{A}$), ie in $\sum_{1\leq i\leq \ell} |V_i|$, see Proposition 3.1. It is easy to see that operation *Decr* is done in constant time, even after replacing delete by push. Additional work added in *Union-Find* functions are also constant time operations since we only change value of `impl` and make call to *Decr*. The process of updating elements of $V_{i-1} \setminus V_i$ is not realized in constant time, but clearly is done in linear time of $|V_{i-1} \setminus V_i|$ so it doesn't cost more than the additional work introduced by *Flatten*.

Thus the complexity of the additional work introduced for the memory management is bounded by $O(\mathit{Width}(\mathcal{A}))$.

5 Application to Image Segmentation

As already mentioned in Section 2.3, *Union-Find* is well suited to implement region growing segmentation in image processing. But a major drawback is the space consuming data structure. Indeed we need at least one pointer for each element, and for such an application as region segmentation, some additional data are needed in order to help the *Oracle* function to take its decision. For example in our application, an element requires a structure of at least 16 bytes. So for an image of about 1024×1024, i.e about 1 million of element, we need 16 MBytes to keep the entire *Union-Find* data structure in RAM. In such a situation our approach of using the main part of this memory only sequentially pays off in a qualitative improvement of the running times. If we go even further and try to treat 3-dimensional images we reach – by analogous computations – a memory demand of 16GB, something nobody is currently capable to install as RAM for a reasonable price. So it is clear that a part of the structure must be kept on a SAM to improve efficiency of the memory accesses, and Algorithm 5 is well-suited for such a purpose.

In fact Algorithm 2 follows already the scheme of Algorithm 5. Indeed the algorithm proceeds line by line, and only two are involved in the same time. So we have a phase decomposition where each V_i is the set of pixels of two consecutive lines. Moreover *Flatten* is done at the end of the phase and this is the same as doing it at the beginning of the following phase. So we just have to add the process of updating elements of $V_{i-1} \setminus V_i$ after Flatten and

Algorithm 6: MergeSquare

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

```
1  if k = 0 then return
2  hm ← 2^{k-1} − 1; hp ← 2^{k-1}
3  for dir=NM to SE do
4      MergeSquare(bm[dir], k − 1)
5  for i = 0 to 2^k do
6      left ←Find(bm[i, hm])
7      right ←Find(bm[i, hp])
8      if Oracle(left,right) then
9          Union(left,right)
10 for i = 0 to 2^k do
11     up ←Find (bm[hm, i])
12     down ←Find (bm[hp, i])
13     if Oracle(up,down) then
14         Union(up,down)
```

to modify *Union-Find* operations accordingly to Section 4.1. Note that $V_{i-1} \setminus V_i$ is in fact the line $j-1$.

In [3] a second segmentation algorithm based on UF is presented, see Algorithm 6. Assuming the image is a square of size $n \times n$, this algorithm proceeds recursively by dividing the image into 4 sub-squares of size $n/2 \times n/2$. After coming up the recursion, the regions in the 4 sub-squares are merged together along the common boundary. For a proof of the linearity of this algorithm we again refer to [3].

For the formulation of Algorithm 6, $bm[NW]$ denotes the northwestern submatrix of bm, $bm[NE]$ the northeastern, etc... Here again, the decomposition in phases is trivial, and to assure a better memory management we just need to add the necessary stuffs before line 3 of Algorithm 6. But in the case of this algorithm the sets of open elements can be very large, e.g all the image at the lower level of the recursion. So the cost of the additional works will completely overshoot the linear time complexity of this algorithm. But here, the process can be refined by doing *Flatten* only on active elements. Indeed, due to the recursion and to the particular decomposition of the bitmap, we are sure that open elements which do not belong to the current bitmap (i.e bm in Algorithm 6) cannot be part of a tree of an active element. Then we only need to do *Flatten* on active elements of bm (medians of bm), i.e $bm[0 \cdots 2^k, hm]$, $bm[0 \cdots 2^k, hp]$, $bm[hm, 0 \cdots 2^k]$ and $bm[hp, 0 \cdots 2^k]$. So additional work introduced by *Flatten* is not too much costly.

Note that these two algorithms have been implemented and gives short running times: about $2s$ for a 512×512 image.

In addition in [4] we have presented a method to segment 3-dimensional images using this principle. Using such a strategy, we have reduced the need of RAM from 710MB to only 10MB. So the algorithm can be executed on a common workstation.

6 Conclusion

We have presented a general tool to minimize the random access memory that is used of a broad class of *Union-Find* algorithms. Not only this allows to use *Union-Find* strategy on large application, but also this improves the real time performance. Indeed the well-known von Neumann bottleneck of synchronizing CPU and memory is thus avoided. This method has been successfully implemented for the image segmentation problem.

Acknowledgment

We like to thank Thomas Rehm for successfully implementing parts of the strategies presented in this paper.

References

1. 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), pp. 59–65.
2. M. B. DILLENCOURT, H. SAMET, AND M. TAMMINEN, *A general approach to connected-component labeling for arbitrary image representations*, J. Assoc. Comput. Mach., 39 (1992), pp. 253–280. Corr. p. 985-986.
3. C. FIORIO AND J. GUSTEDT, *Two linear time Union-Find strategies for image processing*, Theoret. Comput. Sci., 154 (1996), pp. 165–181.
4. ———, *Volume segmentation of* 3*-dimensional images*, Tech. Rep. 515/1996, Technische Universität Berlin, 1996.
5. Z. GALIL AND G. F. ITALIANO, *Data structures and algorithms for disjoint set union problems*, ACM Computing Surveys, 23 (1991), pp. 319–344.
6. J. GUSTEDT, *Efficient union-find for planar graphs and other sparse graph classes*, in Graph-Theoretic Concepts in Computer Science, 22nd International Workshop WG '96, Ausiello et al., eds., Lecture Notes in Computer Science, Springer-Verlag, 1996. *to appear*.
7. J. A. LA POUTRÉ, *New techniques for the union-find problem*, in Proceedings of the first annual ACM-SIAM Symposium on Discrete Algorithms, A. Aggarwal et al., eds., Society of Industrial and Applied Mathematics (SIAM), 1990, pp. 54–63.
8. K. MEHLHORN, *Data Structures and Algorithms 1: Sorting and Searching*, Springer, 1984.
9. K. MEHLHORN AND A. TSAKALIDIS, *Handbook of Theoretical Computer Science*, vol. A, Algorithms and Complexity, Elsevier Science Publishers B.V., Amsterdam, 1990, ch. 6, Data Structures, pp. 301–314.
10. J. MUERLE AND D. ALLEN, *Experimental evaluation of techniques for automatic segmentation of objects in a complex scene*, in Pictorial Pattern Recognition, G. C. Cheng et al., eds., Thompson, Washington, 1968, pp. 3–13.
11. N. ROBERTSON AND P. SEYMOUR, *Graph minors I, excluding a forest*, J. Combin. Theory Ser. B, 35 (1983), pp. 39–61.
12. ———, *Graph minors II, algorithmic aspects of tree-width*, J. Algorithms, 7 (1986), pp. 309–322.
13. R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.
14. ———, *A class of algorithms which require non-linear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.
15. R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.
16. M. J. VAN KREVELD AND M. H. OVERMARS, *Union-copy structures and dynamic segment trees*, J. of the Association for Computing Machinery, 40 (1993), pp. 635–652.

Reports from the group

"Algorithmic Discrete Mathematics"

of the Department of Mathematics, TU Berlin

533/1996 *Andreas S. Schulz, and Martin Skutella:* Randomization Strikes in LP–Based Scheduling — Improved Approximations for Min–Sum Criteria

530/1996 *Ulrich H. Kortenkamp, Jürgen Richter-Gebert, Aravamuthan Sarangarajan, and Günter M. Ziegler:* Extremal Properties of 0/1-Polytopes, to appear in Discrete & Computational Geometry

524/1996 *Elias Dahlhaus, Jens Gustedt and Ross McConnell:* Efficient and Practical Modular Decomposition

523/1996 *Jens Gustedt and Christophe Fiorio:* Memory Management for Union-Find Algorithms

520/1996 *Rolf H. Möhring, Matthias Müller-Hannemann, and Karsten Weihe:* Mesh Refinement via Bidirected Flows: Modeling, Complexity, and Computational Results

519/1996 *Matthias Müller-Hannemann and Karsten Weihe:* Minimum Strictly Convex Quadrangulations of Convex Polygons

517/1996 *Rolf H. Möhring, Markus W. Schäffter, and Andreas S. Schulz:* Scheduling Jobs with Communication Delays: Using Infeasible Solutions for Approximation

516/1996 *Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein:* Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms

515/1996 *Christophe Fiorio and Jens Gustedt:* Volume Segmentation of 3-dimensional Images

514/1996 *Martin Skutella:* Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem, to appear in Proceedings of The Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, January 1997

509/1996 *Soumen Chakrabarti, Cynthia A. Phillips, Andreas S. Schulz, David B. Shmoys, Cliff Stein, and Joel Wein:* Improved Scheduling Algorithms for Minsum Criteria

508/1996 *Rudolf Müller and Andreas S. Schulz:* Transitive Packing

506/1996 *Rolf H. Möhring and Markus W. Schäffter:* A Simple Approximation Algorithm for Scheduling Forests with Unit Processing Times and Zero-One Communication Delays

505/1996 *Rolf H. Möhring and Dorothea Wagner:* Combinatorial Topics in VLSI Design: An Annotated Bibliography

504/1996 *Uta Wille:* The Role of Synthetic Geometry in Representational Measurement Theory

502/1996 *Nina Amenta and Günter M.Ziegler:* Deformed Products and Maximal Shadows of Polytopes

500/1996 *Stephan Hartmann and Markus W. Schäffter and Andreas S. Schulz:* Switchbox Routing in VLSI Design: Closing the Complexity Gap

498/1996 *Ewa Malesinska, Alessandro Panconesi:* On the Hardness of Allocating Frequencies for Hybrid Networks

496/1996 *Jörg Rambau:* Triangulations of Cyclic Polytopes and higher Bruhat Orders

489/1995 *Eva Maria Feichtner and Dmitry N. Kozlov:* On Subspace Arrangements of Type $\mathcal{D}$

483/1995 *Rolf H. Möhring and Markus W. Schäffter:* Approximation Algorithms for Scheduling Series-Parallel Orders Subject to Unit Time Communication Delays

478/1995 *Sven G. Bartels:* The complexity of Yamnitsky and Levin's simplices algorithm

477/1995 *Jens Gustedt, Michel Morvan and Laurent Viennot:* A compact data structure and parallel algorithms for permutation graphs, appeared in : Nagl et al., editors, *Graph-Theoretic Concepts in Computer Science*, Proccedings of the 20th International Workshop WG '95.

476/1995 *Jens Gustedt:* Efficient Union-Find for Planar Graphs and other Sparse Graph Classes, to appeare in : Ausiello et al., editors, *Graph-Theoretic Concepts in Computer Science*, Proccedings of the 21st International Workshop WG '96 .

475/1995 *Ross McConnell and Jeremy Spinrad:* Modular decomposition and transitive orientation

474/1995 *Andreas S. Schulz:* Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP–Based Heuristics and Lower Bounds

473/1995 *Günter M. Ziegler:* Shelling Polyhedral 3-Balls and 4-Polytopes

472/1995 *Martin Henk, Jürgen Richter-Gebert and Günter M. Ziegler:* Basic Properties of Convex Polytopes

471/1995 *Jürgen Richter-Gebert and Günter M. Ziegler:* Oriented Matroids

465/1995 *Ulrich Betke and Martin Henk:* Finite Packings of Spheres

463/1995 *Ulrich H. Kortenkamp:* Every simplicial polytope with at most $d+4$ vertices is a quotient of a neighborly polytope, to appear in Discrete & Computational Geometry

462/1995 *Markus W. Schäffter:* Scheduling with Forbidden Sets

461/1995 *Markus W. Schäffter:* Drawing Graphs on Rectangular Grids with at most 2 Bends per Edge

458/1995 *Ewa Malesińska:* List Coloring and Optimization Criteria for a Channel Assignment Problem

447/1995 *Martin Henk:* Minkowski's second theorem on successive minima

441/1995 *Andreas S. Schulz, Robert Weismantel, Günter M. Ziegler:* 0/1–Integer Programming: Optimization and Augmentation are Equivalent, appeared in Paul Spirakis (ed.): Algorithms – ESA '95, Lecture Notes in Computer Science 979, Springer: Berlin, 1995, pp. 473-483

440/1995 *Maurice Queyranne, Andreas S. Schulz:* Scheduling Unit Jobs with Compatible Release Dates on Parallel Machines with Nonstationary Speeds, appeared in Egon Balas and Jens Clausen (eds.): Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science 920, Springer: Berlin, 1995, pp. 307-320

439/1995 *Rolf H. Möhring, Matthias Müller-Hannemann:* Cardinality Matching: Heuristic Search for Augmenting Paths

436/1995 *Andreas Parra, Petra Scheffler:* Treewidth Equals Bandwidth for AT-Free Claw-Free Graphs

432/1995 *Volkmar Welker, Günter M. Ziegler, Rade T. Živaljević:* Comparison Lemmas and Applications for Diagrams of Spaces

431/1995 *Jürgen Richter-Gebert, Günter M. Ziegler:* Realization Spaces of 4-Polytopes are Universal, to appear in *Bulletin of the American Mathematical Society*, October 1995.

430/1995 *Martin Henk:* Note on Shortest and Nearest Lattice Vectors

429/1995 *Jörg Rambau, Günter M. Ziegler:* Projections of Polytopes and the Generalized Baues Conjecture

428/1995 *David B. Massey, Rodica Simion, Richard P. Stanley, Dirk Vertigan, Dominic J. A. Welsh, Günter M. Ziegler:* Lê Numbers of Arrangements and Matroid Identities

408/1994 *Maurice Queyranne, Andreas S. Schulz:* Polyhedral Approaches to Machine Scheduling

407/1994 *Andreas Parra, Petra Scheffler:* How to Use the Minimal Separators of a Graph for Its Chordal Triangulation

401/1994 *Rudolf Müller, Andreas S. Schulz:* The Interval Order Polytope of a Digraph, appeared in Egon Balas and Jens Clausen (eds.): Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science 920, Springer: Berlin, 1995, pp. 50-64

396/1994 *Petra Scheffler:* A Practical Linear Time Algorithm for Disjoint Paths in Graphs with Bounded Tree-width

394/1994 *Jens Gustedt:* The General Two-Path Problem in time $O(m \log n)$, extended abstract

393/1994 *Maurice Queyranne:* A Combinatorial Algorithm for Minimizing Symmetric Submodular Functions

392/1994 *Andreas Parra:* Triangulating Multitolerance Graphs

390/1994 *Karsten Weihe:* Maximum (s,t)–Flows in Planar Networks in $O(|V| \log |V|)$ Time

386/1994 *Annelie von Arnim, Andreas S. Schulz:* Facets of the Generalized Permutahedron of a Poset, to appear in *Discrete Applied Mathematics*

383/1994 *Karsten Weihe:* Kurzeinführung in C++

377/1994 *Rolf H. Möhring, Matthias Müller-Hannemann, Karsten Weihe:* Using Network Flows for Surface Modeling

376/1994 *Valeska Naumann:* Measuring the Distance to Series-Parallelity by Path Expressions

375/1994 *Christophe Fiorio, Jens Gustedt:* Two Linear Time Union-Find Strategies for Image Processing

374/1994 *Karsten Weihe:* Edge–Disjoint (s,t)–Paths in Undirected Planar graphs in Linear Time

373/1994 *Andreas S. Schulz:* A Note on the Permutahedron of Series–Parallel Posets, appeared in *Discrete Applied Mathematics* 57 (1995), pp. 85-90

371/1994 *Heike Ripphausen–Lipa, Dorothea Wagner, Karsten Weihe:* Efficient Algorithms for Disjoint Paths in Planar Graphs

Reports may be requested from: S. Marcus
Fachbereich Mathematik, MA 6–1
TU Berlin
Straße des 17. Juni 136
D-10623 Berlin – Germany

e-mail: Marcus@math.TU-Berlin.DE

Reports are available via anonymous ftp from: ftp.math.tu-berlin.de
cd pub/Preprints/combi
file Report–<number>–<year>.ps.Z