ELSEVIER
Theoretical Computer Science 203 (1998) 123–141
Theoretical Computer Science

Efficient Union-Find for planar graphs and other sparse graph classes [1]

Jens Gustedt *
TU Berlin, Sekr. MA 6-1, D-10623 Berlin, Germany

Abstract

We solve the Union-Find Problem (UF) efficiently for the case the input is restricted to several graph classes, namely partial k-trees for any fixed k, d-dimensional grids for any fixed dimension d and for planar graphs. The result on grids allows us to perform region growing techniques that are used for image segmentation in linear time. For planar graphs we develop a technique of decomposing such a graph into small subgraphs, patching, that might be useful for other algorithmic problems on planar graphs, too.

By efficiency we do not only mean linear time in a theoretical setting but also a practical reorganization of memory such that a dynamic data structures for UF is allocated consecutively.

Keywords: Graph algorithms; Union-find; Planar graphs; Image Segmentation

1. Introduction and overview

In this paper we present a new idea for the Union-Find Problem (*UF*) that extends the approach of Gabow and Tarjan [6]. In this paper the authors efficiently solve the UF problem where the `Unions` that are allowed form a tree: in linear time on a **RAM**.

Here we attack this in a more general way. Given is a graph G, the objective is to perform `Unions` and `Finds` on the vertices $V(G)$ such that for every step the actual subsets that are obtained form connected subgraphs of G.

Clearly, if $G = K_n$ we are back to the usual UF problem with no restrictions at all; at least if we ignore complexity issues for a moment. In general, we will have some family $\mathcal{G}$ of graphs and we will call the problem that the graph G might be arbitrarily chosen from $\mathcal{G}$ the *$\mathcal{G}$-graphical Union-Find Problem*, $\mathcal{G}$UF. To avoid the lower bounds for the complexity of UF, see [8, 13], we have to assume that such an instance G is given explicitly:

[1] Supported by the IFP "Digitale Filter".
* E-mail: gustedt@math.tu.berlin.de.

0304-3975/98/$19.00
PII S0304-3975(97)00291-0

General Assumption. *The input is at least as large as the number of edges of G.*

The method we use (and develop in Sections 2 and 3) is an extension of the one given by Gabow and Tarjan [6], namely by

(i) solving the problem for "small" sets in a preprocessing,
(ii) then dividing the instance by a "neglectable" portion of the graph, a *skeleton*, into such small sets, the *clusters*, and
(iii) giving a technique how to perform UF as a combination of local information in a cluster and global information in the skeleton.

This technique can also be seen as a method of *reorganizing memory* in order to reduce memory faults or delay and thus improve the real time behavior of UF data structures.

The real world bottleneck for `Union-Find` is the use of a dynamic data structure. The usual path compression data structure makes no guarantee at all where in memory the next item to fetch might be located. So if we are doing `Unions` more or less arbitrary the *real* processing time of the algorithm is dominated by loading data, either from memory into cache or – even worse – from disk into memory. A discussion of this reorganization aspect is given at the end in Section 7.

Since for such practical considerations the distinction between random access machines (RAM) and pointer machines (PM) is rather academic we restrict ourselves to the RAM.

The progress made here in this work for this class of problems is summarized by the following theorem.

Main Theorem. *$\mathcal{G}$UF is solvable on a RAM in time proportional to the number of* `Find`*s for $\mathcal{G}$ any of the following classes of graphs:*

(i) *Trees and partial k-trees, for any fixed parameter k.*
(ii) *d-dimensional grids for fixed d and 8-neighborhood graphs of two-dimensional grids.*
(iii) *Planar graphs.*

Just to give an example of an application of this theorem consider the problem of computing minimum spanning trees. Suppose we are in a situation that we have graph with given edge weights such that in addition the sort order of the edges is known (or easy to compute). With Kruskal's algorithm our Main Theorem then immediately gives linear time bounds if we restrict ourselves to the graph classes in question.

For the proof of the Main Theorem the three parts are handled in Sections 4, 5 and 6. For the scope of this introduction we briefly describe the parts of the Main Theorem in the following three paragraphs.

Partial k-trees: Graphs of treewidth at most k, the so-called partial k-trees, are a quite popular generalization of trees, see Bodlaender [2]. These are graphs that have a tree-decomposition of width k, i.e. such a graph can be separated by sets of size at

most $k+1$ in a "tree-like" fashion. For partial k-trees the result obtained as well as the methods used are straight generalizations of Gabow and Tarjan [6]. It is primarily chosen as a first illustration of the power of this approach.

Grids and UF for image segmentation: An important application will be that the underlying graph G is, e.g. a grid as it appears in image segmentation. In image segmentation the goal is to group a digital image into homogeneous *connected* regions, the so-called *segments*. An important technique to do this is region growing: starting from one-point segments and gluing together neighboring segments if appropriate. Clearly, such a technique involves UF as a data structure: segments are subsets of the set of pixels and gluing them together involves a `Union` of two such sets. But the `Union`'s that are permissible are strongly restricted by the requirement that segments always should be connected.

Until now there are only two special cases where the complexity of this approach is known to be linear, see [4, 5]. These special cases strongly restrict the order in which `Unions` may be performed, the so-called *scanning order*, see [4], and the neighborhood definition that is used for the connectivity property. One of our goals here is to extend this to *arbitrary scanning orders*, to *other neighborhood definitions* on digital images and even to three-dimensional images.

Planar graphs: For planar graphs we develop a technique, called patching, of decomposing such a graph into small subgraphs that might be useful for other algorithmic problems on planar graphs, too. A patching is a separator of the graph of negligible size, i.e. smaller than $n/\mathtt{slow}(n)$ for some growing function `slow`, that separates the graph into small components.

2. Basic definitions and facts

2.1. Notations

Graphs are simple and without loops or isolated vertices. For a graph G, $V(G)$ and $E(G)$ denote the vertex and edge sets. The degree of a vertex v is denoted by $\deg_G(v)$, $\deg_G$ is the maximum degree over all vertices. $d_G(u,w)$ denotes the distance between two vertices u and w, i.e. the number of edges on a shortest path between u and w. The notation $d_G(U,W)$ is the obvious extension to arbitrary vertex sets U and W. For some vertex set U and some value p we define the *p-neighborhood* of U as

$$N_G^{\mathrm{p}}(U) = \{v \in V \mid d_G(v,U) \leqslant \mathrm{p}\}. \tag{1}$$

We will always assume that graphs that are given as instances are given *explicitly* as lists of edges, say. So in particular we always have an input size that is proportional to the number of edges.

The following problem certainly is one of the most important (and famous) in the theory of data structures and algorithms and already appeared very early as a real-world subproblem in algorithms dealing with any kind of set operations:

Problem 2.1 (*Union-Find Problem, UF*).
Instance: *A set V and $V_1,\ldots$ a partition of V.*
Task: *Perform a sequence of $n < |V|$* `Union`*s and $m \geqslant |V|$* `Find`*s on V.*

Here `Union` operations are considered to unify two existing subsets of the partition and creating thus a new partition of V. `Find` operations are used to identify for each element $v \in V$ the subset of the partition it currently belongs to. Usually, this identification in a `Find` is thought to output some designated element of that subset, its *representative*.

A commonly used restriction of the UF problem is the case that the partition that is given in the instance are just the singleton subsets of V. Indeed, this restriction is not too severe since we may simply assume that we perform the necessary `Union` operations to obtain the partition $V_1,\ldots$ in a preprocessing.

The problem we are dealing with is a restriction of the general UF problem and given by the following specification.

Problem 2.2 (*Graphical Union-Find Problem, GUF*).
Instance: *A graph G.*
Task: *Perform a sequence of $n < |V(G)|$* `Union`*s and $m \geqslant |V(G)|$* `Find`*s on $V(G)$ that respect G.*

Here a sequence of `Union`s and `Find`s *respects* a graph G if after each `Union` every subset created induces a connected subgraph of G. This is equivalent of saying that every `Union` can be realized as an edge contraction in G.

A graph H is said to be a *minor* of another graph G if H (or its isomorphic image) can be obtained by a sequence of edge or vertex deletions and edge contractions. Thus, the actual state of a UF process is always represented by a minor H of G. So we easily obtain the following remark.

Remark 2.3. Let G^- be a minor of G^+ such that a sequence of edge contractions and deletions that lead from G^+ to G^- is explicitly given. Suppose GUF is solvable for G^+ in time $t(n,m)$, then any sequence of $n^- < |V(G^-)|$ `Union`s and $m^- \geqslant |V(G^-)|$ `Find`s on $V(G^-)$ that respects G^- is solvable in time $t(n^- + |V(G^+)|, m^-)$.

Observe that this means in particular that if the size of G^+ is linear in the size of G^-, GUF may be solved on G^- in the same complexity as on G^+.

We will investigate the GUF problem restricted to certain graph classes. If $\mathcal{G}$ is a class of graphs $\mathcal{G}$UF refers to GUF restricted to $\mathcal{G}$. To warm up, let us consider the problem $\mathcal{G}^3$UF, where $\mathcal{G}^3$ is the class of graphs of degree bounded by 3.

Remark 2.4. GUF is solvable in linear time iff $\mathcal{G}^3$UF is solvable in linear time.

Proof. "$\Longrightarrow$" is trivial. For "$\Longleftarrow$" replace in instance G any vertex of degree higher than 3 by an appropriate binary tree:

Let V be such a vertex and $d = \deg(v)$ its degree. Let T_v be (newly created) binary tree on $\lceil d/2 \rceil$ leaves. Delete v from G and connect the leaves of T_v to the former neighbors of v in an appropriate way.

By that we easily obtain a graph G' that fulfills the requirements of Remark 2.3 and is at most twice as large as G. □

To simplify the discussion in the rest of the paper a bit we will always assume that $n = |V(G)| - 1$. We will also assume that the demand for a `Union` is presented by pointing out an edge of G for which the components/subsets of the endpoints should be glued into one; *the question whether or not two current subsets may be united or not is not part of the problem specification.*

2.2. Slowly growing functions

We will reduce the problem so that the number of edges in the underlying graph must not be too large compared to the number of vertices. Therefore, throughout the following we use the notation of $\mathtt{slow}(n)$ for a *slow*ly growing function. By that we mean a monotone function that at least fulfills

$$\mathtt{slow}(n) \leqslant \tfrac{1}{18} \log \log(n) \tag{2}$$

and is dominating α, the inverse of the Ackerman function. We use a definition for that function α that turns out to be basically (in O-notation) the same as the traditional one but is a bit simpler to handle:

$$\alpha(m,n) = \min\{x \mid A(x, \lceil m/n \rceil) > n\}, \tag{3}$$

where A is the Ackermann function given by the usual recursion

$$A(i,0) = 1, \tag{4}$$

$$A(0,x) = 2x, \tag{5}$$

$$A(i+1,x+1) = A(i,A(i+1,x)). \tag{6}$$

Observe that α is increasing in the second argument but decreasing in the first:

$$\alpha(m,n+1) \geqslant \alpha(m,n), \tag{7}$$

$$\alpha(m+1,n) \leqslant \alpha(m,n). \tag{8}$$

We require for `slow` that there is a constant c such that $A(c, \lceil \mathtt{slow}(n) \rceil) > n$ for all n. In particular, this means that

$$\alpha(\mathtt{slow}(n)n, n) \leqslant c. \tag{9}$$

Observe that there are many commonly used functions that fulfill requirements (2) and (9), e.g. (almost[2]) any iterated log-function or $\log^*$. For these functions choosing c

[2] Almost for the magic constant $\frac{1}{18}$ in (2) that is needed for an estimation later on.

to be 2 can be easily seen to do the job. In the rest of the paper we will assume that such a function `slow` with corresponding constant c is *chosen* and *fixed*.

2.3. Known bounds for the Union-Find Problem

UF is one of the few problems for which a lower bound (i.e. $\alpha(m,n)m$) on the complexity is known. This lower bound has been given by Tarjan [13] for some restricted version of a pointer machine and by La Poutre [8] for unrestricted pointer machines. Whether or not this bound also applies to the RAM is an open problem.

By the work of Tarjan [12] and Banachowski [1] we also know that this bound is sharp:

Theorem 2.5. *Any UF problem can be solved in time* $O(\alpha(m,n)m)$.

This is mainly achieved by two ideas:

(1) The elements of V are maintained in a rooted forest such that each component corresponds to a actual subset and is represented by the root of that component. `Find` operations are done by following the path from an element upward to its root. `Union` operations are done by linking one of the two roots of the components in question to the other one.
(2) During each `Find` operation the corresponding tree is updated to optimally use the information collected when following a path upward. This is done by linking each intermediate element on the path directly to the root, so-called *path compression*.

For an overview over UF problems see the article [11]. For more details and legible proofs see the book of Melhorn [10].

As a corollary from that we easily obtain:

Corollary 2.6. *Let G be an instance of GUF that is given explicitly as input and $n = |V(G)|$ with $|E(G)| > \alpha(n,n)n$. Then any sequence of* `Union`*s and* `Find`*s that respects G can be performed in linear time.*

Together with Remark 2.4 this shows in particular that it makes sense to restrict our discussion to classes of sparse graphs.

Until now there are several approaches known to solve restricted version of the UF problem in linear time. All are in fact specializations of GUF. The most commonly know work in that direction is the one of Gabow and Tarjan (1984). It solves the problem if the class of graphs is restricted to trees.

Less commonly known are two papers that are motivated by UF as it appears in image processing, Dillencourt et al. [4] and Fiorio and Gustedt [5]. They restrict the permissible graphs to two-dimensional grids (and also to planar graphs in [5]) but in addition pose severe restrictions on the order in which `Union`s between sets may be performed to achieve their linear-time bounds.

Our goal here is to merge these two types of results to widen the classes of graphs that are tractable and to eliminate all other restrictions on the order of `Union` operations.

2.4. Micro-encoding of small sets

A basic method will be to encode UF problems on small subsets of the ground set by bit-vectors, so called *micro-encoding*. Here by "small" we mean sets that are smaller than a function ℓ with[3]

$$\ell(n) \leqslant \sqrt[3]{\log n}. \tag{10}$$

Such micro-encodings will then be used to do UF on small subsets of the groundset. Therefore, let V with $|V| = n$ be a set and $V_1, V_2, \ldots$ a partition of V. We say that a sequence of `Unions` and `Finds` on V *respects* the partition if for any subset U produced by those `Unions` there is an i such that $U \subseteq V_i$. Our aim is the following lemma.

Lemma 2.7. *Let V with $|V| = n_0$ be a set and $V_1, V_2, \ldots$ a partition of V such that $|V_i| \leqslant \ell(n_0)$ for all i. Then any sequence of n* `Union`*s and m* `Find`*s on V that respects the partition can be done in time $O(n_0 + n + m)$.*

Proof. To have all operations in constant time we maintain tables for the `Union` and the `Find` operation on sets smaller than $\ell(n_0)$. We generate these tables in a preprocessing that runs in time $O(n_0)$.

We assume that we always know a part $V_{i(v)}$ of V to which v belongs and the unique "name" $0 \leqslant j(v) < |V_{i(v)}|$ of v in $V_{i(v)}$. Now operation `Find`(v) identifies the *state* s of $V_{i(v)}$ and `Find`'s the current identifier of the subset of v by means of s and $j(v)$.

Operation `Union`(v, w) does its job in a similar way and in addition updates the state of $V_{i(v)}$ to the new value representing the new family of subsets of $V_{i(v)}$ that results.

So a *state* has to represent all information needed for such problems of sets of size less than $\ell(n_0)$. This can be done by representing each state by $\ell(n_0)$ numbers of size $\ell(n_0)$. Thus, each such state may then be encoded in $\ell(n_0)\log \ell(n_0)$ bits which can be bounded as follows:

$$\ell(n_0)\log \ell(n_0) \leqslant \tfrac{1}{3} \log \log(n_0) \sqrt[3]{\log(n_0)} \leqslant \tfrac{1}{3} \log(n_0), \tag{11}$$

if n is large enough.

It is then easy to encode such a UF tree in a bit vector. Since $\ell(n_0)\log \ell(n_0)$ is sufficiently small we may represent each such tree (=state) by an integer and thus we achieve constant time per `Union` and `Find` .

To obtain the right complexity we have to show that the tables are not too large. Therefore, observe that the number of possible states is less than

$$2^{1/3 \log(n_0)} \leqslant \sqrt[3]{n_0}. \tag{12}$$

So clearly the size of such tables for `Union` and `Find` can be bounded by

$$\sqrt[3]{n_0} \log^2 \ell(n_0) \tag{13}$$

[3] ℓ stands for large, since it will be much larger than `slow` .

which in turn is dominated by n_0 if n_0 is suitable large. The preprocessing to build up all tables consistently is also easily seen to lay in that bound. □

Observe that a concrete choice of ℓ has not been necessary for the proof, we only needed the upper bound (10). In the following it will only be important that ℓ is a function that is unbounded. So if the reader feels uncomfortable by assuming a logarithmic word size of our RAM she or he may choose her or his growth function for the word size and easily work out an appropriate choice of ℓ. Any growth function for the word size that dominates $\alpha(n,n)$ would do the trick.

3. Union-Find by clustering

For a graph $G = (V,E)$ and $n = |V(G)|$ a *skeleton*, `skel` $=$`skel`(G), is a vertex set with cardinality bounded by $n/$`slow`(n). A *cluster* C of G with respect to `skel` is then a component of $G\setminus$ `skel`. The *boundary* $\partial(C)$ of C are those $v \in$ `skel` that have an edge to a vertex in C. The *closure* $\Delta(C)$ of C is $\Delta(C) = C \cup \partial(C) = N_G^1(C)$.

Denote by $\mathscr{C}_{\texttt{skel}}$ the *clustering* w.r.t. `skel`, i.e. the family of components of $G\setminus$`skel`. A clustering $\mathscr{C}$ is *valid* if in addition $|\Delta(C)| \leqslant \ell(n)$ for all $C \in \mathscr{C}$. Clearly `skel` can be reconstructed from $\mathscr{C}_{\texttt{skel}}$, and we will omit the subscript whenever possible. On the other hand it will be sometimes necessary to denote the skeleton of a clustering $\mathscr{C}$ explicitly by $\texttt{skel}_{\mathscr{C}}$.

Because every edge either joins two elements of `skel` or is inside exactly one $\Delta(C)$ for some cluster C and every such $\Delta(C)$ is connected we have that

$$\sum_{C\in\mathscr{C}} |\Delta(C)| \leqslant |E(G)| \tag{14}$$

and thus we may encode any clustering with an additional overhead that is linear in $|E(G)|$.

We are now able to formulate a UF data structure for that context, see the procedures `Find` above and `Union` on the next page. We assume that for each $\Delta(C)$ we maintain a local (micro-encoded) UF data structure and a global (path compression) one on `skel`. For the later we assume that if $v \neq w$ are the representatives of their sets and we perform `Union`(v,w) the representative of the newly created set will be v, i.e. the first argument to `Union`.

`Find`

Input: Vertex $v \in V(G)$.

Output: Root $r \in V(G)$ a unique identifier for the current subset of v.

1 **if** $v \notin$ `skel` **then** $v :=$ `Find`$_{\Delta(C(v))}v$;

2 **else return** `Find`$_{\texttt{skel}}\ v$;

3 **if** $v \in$ `skel` **then return** `Find`$_{\texttt{skel}}\ v$;

4 **else return** v;

Union

```
   Input: Edge {v,w} ∈ E(G).
1  if v ∉ skel then v := Find_{Δ(𝒞(v))} v;
2  if w ∉ skel then w := Find_{Δ(𝒞(w))} w;
3  if v ∈ skel then v := Find_skel v;
4  if w ∈ skel then w := Find_skel w;
5  if v ≠ w then
6  |  if w ∉ skel then Union_{Δ(𝒞(w))}(v,w);
7  |  else
8  |  |  if v ∉ skel then Union_{Δ(𝒞(v))}(w,v);
9  |  |  else Union_skel(v,w)
```

First observe that Find for some subset S indeed gives a unique identifier $r = r(S) \in S$. Because of the interchange of v and w in line 8 of Union it is also easy to see that the identifier of a subset is an element of skel whenever this is possible.

Remark 3.1. For any subset S created by Unions with $S \cap \mathtt{skel} \neq \emptyset$ we have that $r(S) \in \mathtt{skel}$.

Lemma 3.2. *Let $G = (V,E)$ be an instance of GUF such that a valid clustering $\mathscr{C}$ of G is given. Then the above data structure solves this problem for any valid sequence of $n < |V|$ Unions and $m \geqslant |V|$ Finds in time $O(n+m)$ on a RAM.*

Proof. Correctness follows directly from Remark 3.1. It remains to show the complexity. We have $O(n+m)$ calls to the operations of the local UF data structure and an over all $O(n)$ preprocessing time, so the local data structures poses no problems.

In addition, we have $m + 2n$ Finds on the global data structure. This global data structure has at most $n' = n/\mathtt{slow}(n)$ elements and in particular we cannot perform more than n' Unions here. We find that

$$m \geqslant n > n' = n/\mathtt{slow}(n) \tag{15}$$

and

$$m/n' \geqslant n/n' = \mathtt{slow}(n) \geqslant \mathtt{slow}(n') \tag{16}$$

which amounts to

$$m \geqslant \mathtt{slow}(n')n'. \tag{17}$$

So the complexity of the global Unions and Finds is bounded by

$$\alpha(m+2n,n')m \leqslant \alpha(m,n')m \leqslant \alpha\left(\mathtt{slow}(n')n',n'\right)m. \tag{18}$$

Since $\mathtt{slow}(n')$ is suitably growing, by (9) we have that the right-hand side is bounded by $c \cdot m$, and we are done. □

```
                    Tree Partition
Input: A binary rooted tree G, with root r and n = V(G).
Output: skel, a valid skeleton for G.
1  skel := ∅;
2  foreach v ∈ V(G) do c_v = 0;
3  foreach v ∈ V(G) from leaves up to r do
4  |  foreach child w of v do c_v += c_w;
5  |  if c_v ≥ 2slow(n) then
6  |  |  skel := skel ∪ {v};
7  |  |  c_v = 0;
```

4. Partial k-trees

Let us now prove part (i) of the Main Theorem. Since the ideas are just straight forward extensions of the ideas of Gabow and Tajan [6], we give the proof in an informal way.

Let us first concentrate on the case that is settled by Gabow and Tarjan, namely trees. The key observation is to restrict ourselves to binary trees. This can easily be achieved as described for Remark 2.4 since the local replacements that were done for vertices of high degree are indeed binary themselves.

If we now root such an instance given by a binary tree at an arbitrary vertex r we may easily collect subtrees as clusters of size s with $\mathtt{slow}(n) \leqslant s < 2\mathtt{slow}(n)$ from leaves to r and cut of such subtrees by their root, see procedure `TreePartition` above. So for a choice of $\ell(n) = 2\mathtt{slow}(n)$ Lemma 3.2 easily applies to that case.

To prove the theorem for partial k-trees, recall that k is assumed to be a fixed constant. From Bodlaender [3] we know that a tree decomposition of width k for such a graph can be found in linear time and it is also well known that such a decomposition may be easily adapted such that the underlying tree is binary. Since every vertex of the decomposition tree represents a separator of size at most $k+1$ (and k is fixed) the ideas for trees immediately extend: collect subgraphs of size s with $\mathtt{slow}(n) \leqslant s < 2\mathtt{slow}(n) + k$ corresponding to subtrees of the decomposition tree and cut them of the graph by the separator corresponding to the root of the subtree.

5. d-dimensional grids and relatives

Recall from the introduction that one of our goals is to provide UF algorithms for image segmentation. This in fact is easily modeled as graphical UF problem restricted to two- or 3-dimensional grids. Recall also that we want to be able to attack arbi-

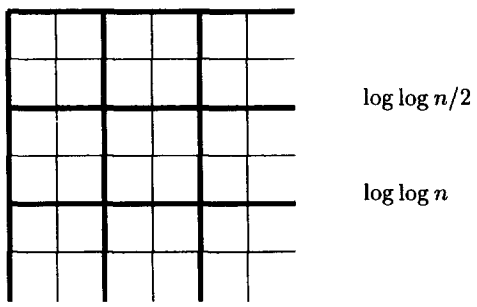


Fig. 1. The skeleton in a two-dimensional grid.

trary sequences of UF operations (as long as they respect the grid) with generalized neighborhood relations inside the grid.

Besides the usual neighborhood definition on grids another one that also is widely used is the so called 8-neighborhood which connects a pixel to all 8 pixels surrounding it: left, right, up, down, left-up, right-up, left-down, right-down. The arguments used in [4, 5] to show linear time complexity of certain UF algorithms heavily rely on the fact that the underlying graph is planar which is not the case for the 8-neighborhood any more. So there is no hope to extend this to the 8-neighborhood.

Proof of Main Theorem, part (ii). Let d be some fixed integer and the instance of the GUF G be a d-dimensional grid. Assume that the vertices are identified with their position $(p_1, \ldots, p_d)$ in the grid. An appropriate skeleton of G can easily be defined:

$$\texttt{skel}(G) = \{(p_1, \ldots, p_d) \in V(G) \mid \text{there is } i \text{ s.t } p_i = 0 \bmod \log\log(n)/d\} \quad (19)$$

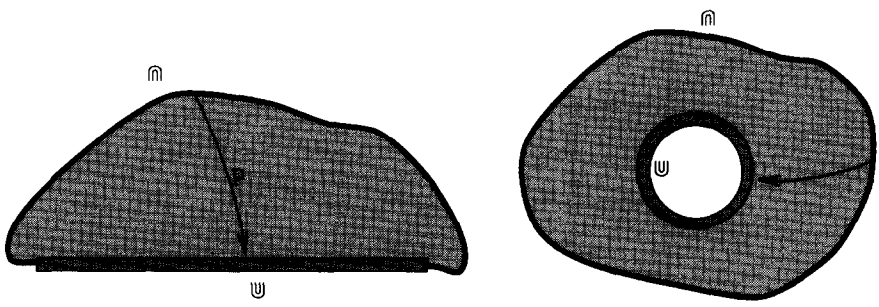
We easily obtain that $|\texttt{skel}(G)|$ is bounded by $d \cdot n/\log\log(n)$ and that for every connected component K of $G \setminus \texttt{skel}(G)$.

$$|\Delta K| \leqslant (1 + \log\log(n)/d)^d. \quad (20)$$

So if we choose $\ell(n)$ to be $(1+\log\log(n)/d)^d$ the part of the statement for grids follows easily with Lemma 3.2. To solve the problem such two paths on the 8-neighborhood in a two-dimensional grid (or a higher-dimensional analogue) observe that the skeleton given above still is a separator and so there is nothing new to prove. □

6. Planar graphs

A natural approach to apply our technique to planar graphs would be to use the Planar Separator Theorem, see [9], by dividing the graph into two halves and going on recursively in each of the halves until we remain with parts that are small enough. But for two reasons a straightforward application of that theorem does not lead to the desired result. First the running time guaranteed by that approach would only lead to



(a) A single boundary face (b) Two boundary faces

Fig. 2. The two possibilities for p-patches.

$O(n \log n)$ which is much too bad for our purposes. Second, it is not easy to establish an appropriate bound on the size of the skeleton; by a single application of that theorem we obtain a separator of size $\sqrt{n}$ and a partition into unequal halves. So we are only able to show that each level l of recursion in total contributes $\sqrt{D^l}\sqrt{n}$ for some $D > 2$ to the skeleton. But this is only leads to a super-linear bound on the size of the skeleton.

So we follow another way to solve GUF if the input graph is planar. What we would like to do is to proceed analogously as for two-dimensional grids. But this is also not as straight forward as one could hope. We have to introduce some technical definitions (p-patches) that will play the (key) role that the tiling squares of size $\log\log(n)/2$ played for the grids.

6.1. p-*patches*

For some positive value p a *p-patch* is a planar graph $G = (V,E)$ together with either

(a) a face $F = F_G$ the *boundary*, partitioned into two intervals $\Cap = \Cap_G$, the *upper boundary*, and $\Cup = \Cup_G$, the *lower boundary*, or

(b) two distinct faces $\Cap = \Cap_G$ the *upper boundary*, and $\Cup = \Cup_G$, the *lower boundary*, such that

(i) all faces but $\Cap$ and $\Cup$ (resp. F) are triangles,

(ii) $\deg_G \leqslant 8$,

(iii) $d_G(v, \Cup) \leqslant p$ for all $v \in V$ and

(iv) $|\Cap| \leqslant 3p \cdot 5^p$.

See Fig. 2 for an illustration of the two different cases (a) and (b). The constants 3, 5 and 8 appearing in this definition are more or less arbitrary and chosen to make the estimations easy. An p-patch is *narrow* if (iv) is strengthened to $|\Cap_G| \leqslant 3p$. We will omit the subscript G at F, $\Cup$ and $\Cap$ whenever possible.

Let G and G' be p-patches. G' is a p-*subpatch* of G if it is a subgraph of G and if $\Cup_{G'} = \Cup_G|_{V(G)}$. Observe that if $G' \neq G$ then also $F_{G'} \neq F_G$. Important properties of p-patches are reflected by the following lemmas.

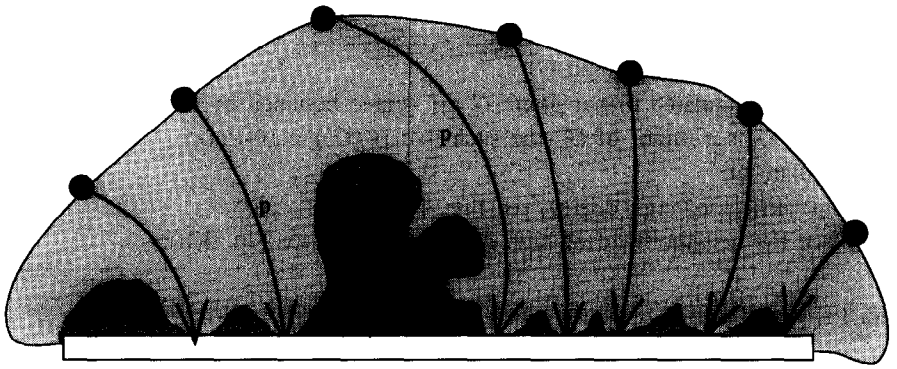


Fig. 3. The p-neighborhood of $\Cap$.

Lemma 6.1. *Let G be an* p*-patch then* $|N_G^{\mathrm{p}}(\Cap)| \leqslant 15\mathrm{p} \cdot 5^{2\mathrm{p}}$.

Proof. This is easy to see because of properties (i) and (ii). The (not so important) constant 5 comes from the fact that when coming from $\Cap$ at least one edges is needed as back edge and at least two are needed as side edges because of the triangulation, enforced by (i). So the size of $N_G^i(\Cap) \leqslant 5N_G^{i-1}(\Cap)$. □

Lemma 6.2. *Let G be an* p*-patch and K be a connected component of $G \setminus N_G^{\mathrm{p}}(\Cap)$ and $G' = N_G^1(K)$. Then G' with $\cup_{G'} = \cup_G|_{V(G')}$ is* p*-subpatch of G.*

Proof. Consider Fig. 3. Here the vertical bar represents $\cup_G$, the filled circles represent elements of $\Cap_G$ and the dashed line represents the boundary of the connected components of $G \setminus N_G^{\mathrm{p}}(\Cap)$.

We have to show that $\Cap_{G'}$ is not too large. Because (iii) of the definition we have that for every $v \in \Cap$ a path P from v to $\cup$ is included in $N_G^{\mathrm{p}}(\Cap)$. Since G is planar, such two such paths of neighbors on $\Cap$ separate G and thus $G \setminus N_G^{\mathrm{p}}(\Cap)$ into two different components. So the component K under investigation is cut of by two such paths P, one on the left and one on the right. Thus, there is $V' \subseteq \Cap$ with $|V'| \leqslant 2$ such that K can already be found as $G \setminus N_G^{\mathrm{p}}(V')$.

But this means also that $\Cap_{G'} \subseteq N_G^{\mathrm{p}}(V')$ and thus because of the degree constraints that $|\Cap_{G'}| \leqslant 2 \cdot 5^{\mathrm{p}}$. □

Lemma 6.3. *There is an algorithm that, given an integer* p *and a narrow* p*-patch G as input, runs in linear time and finds a skeleton skel G of size $\leqslant |V(G)|/\mathrm{p}$ such that every cluster is of size at most* $15\mathrm{p} \cdot 5^{2\mathrm{p}}$.

Proof. Compute a BFS-forest growing from $\Cap$ and let L_i (*level i*) be the set of vertices of distance i from $\Cap$. For each $j = 0, \ldots, \mathrm{p}-1$ we sum up the cardinalities of all levels

Bound Degree

Input: arbitrary planar graph G.
Output: Triangulated planar graph G' of degree bounded by 8 such that G is a minor of G' and such that $|V(G')| \leqslant 10|V(G)|$.
1 Triangulate G;
2 Isolate vertices of high degree, see Fig. 4(a);
3 Replace the neighborhood of high degree vertices, see Fig. 4(b);

i with $i \equiv j \bmod \mathrm{p}$:

$$s_j = \sum_{i \equiv j \bmod \mathrm{p}} |L_i|. \tag{21}$$

Since the levels form a partition of the vertex set there exists a j_0 such that

$$s_{j_0} \leqslant |V(G)|/\mathrm{p}. \tag{22}$$

The union of all corresponding levels

$$S = \bigcup_{i \equiv j_0 \bmod \mathrm{p}} L_i \tag{23}$$

gives our desired skeleton.

Because G is narrow, level j_0 itself has at most $3\mathrm{p} \cdot 5^{j_0} \leqslant 3\mathrm{p} \cdot 5^{\mathrm{p}}$ elements and thus $N_G^{j_0}(\Cap)$ and $G \setminus N_G^{j_0}(\Cap)$ are both p-patches. Now applying Lemma 6.2 iteratively shows that $G \setminus N_G^{j_0}(\Cap)$ breaks down into p-patches and Lemma 6.1 shows that these have the appropriate sizes. □

Observe also, that the multiplicative constant hidden in the linear time bound does not depend on p.

6.2. *Patching an arbitrary planar graph*

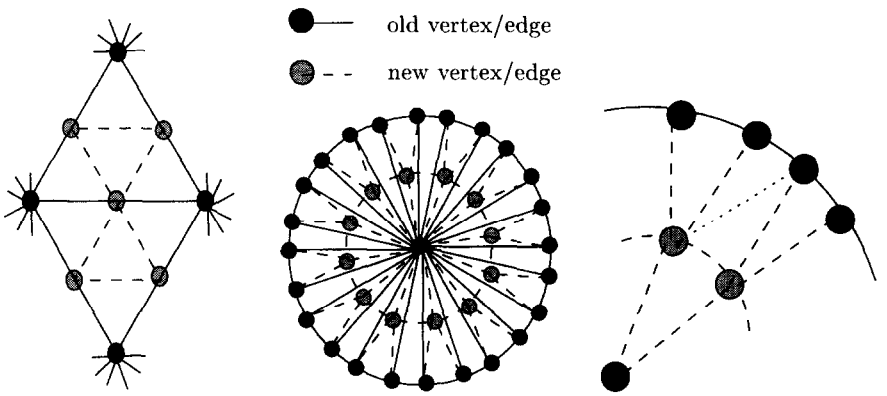
To apply these methods to an arbitrary planar graph we first have to ensure that we always may find a major that is triangulated and has bounded degree, see procedure Bound Degree on the current page.

Clearly, triangulation poses no problem at all. *Isolation* of vertices of high degree is done in such a way that afterwards we have

(i) all vertices v with $\deg(v) > 8$ only have neighbors of degree 6 or less
(ii) every vertex w has at most 2 neighbors v with $\deg(v) > 8$.

This can easily be achieved by dividing each edge by a new vertex and joining all these new vertices to the other 4 new vertices on the neighboring facets. It is clear that now

(i) every old vertex only has new vertices as neighbors,
(ii) every new vertex has degree 6,



(a) Isolating vertices of high degree

(b) Replacement at a vertex of high degree

(c) Replacement of 4 neighbors by 2 new ones

Fig. 4. Bounding the degree.

(iii) every such new vertex has 2 old vertices as neighbors and
(iv) the resulting graph is triangulated again.
See Fig. 4(a) for an illustration.

Replacement at vertices of high degree is done as follows. Let v be a vertex of degree larger than 8 and w and w' two neighbors that form a triangle together with v. Then
(i) delete the edges vw and vw' from G
(ii) introduce a new vertex v' and join it to v, w and w'.

Do so for all adequate pairs of edges of v. These are $\lfloor \deg(v)/2 \rfloor$. If $\deg(v)$ is odd divide the remaining edge by yet another new vertex. Now join all new vertices obtained in this step to a cycle, see Fig. 4(b). Up to now all new vertices have degree at most 5 and all old vertices (but v) have their degree unchanged. To triangulate this newly obtained graph observe that the only facets that are not triangles are those with two new vertices and two former neighbors of v. These squares can be triangulated by diagonals that all go in the same direction, to the right, say, see Fig. 4(c) Now all new vertices have degree 6 and all former neighbors of v have their degree increased at most by one.

We iterate this procedure until $\deg(v) \leqslant 8$. Then all new vertices introduce have degree at most 7.

We then loop for all vertices of high degree. Any vertex w that originally had degree at most 6 is involved into such a replacement for at most 2 vertices of high degree. So the new degree of w is at most 8. To summarize we state

Proposition 6.4. *Let G be a planar graph; then there is a triangulated planar graph G' of degree not larger than* 8 *such that*

```
                                Patch
```

Input: Triangulated planar graph G of bounded degree 8, $n = |V(G)|$, vertex $v_0 \in V(G)$.

Output: $S \subset V(G)$ of size $|S| \leqslant n/\mathrm{p}$ and such that every component of $G \setminus S$ but the one of v_0 is a p-patch.

```
 1  Grow a BFS-tree from v_0;
 2  for i = 1,..., do collect the cardinalities c(i) of all levels L(i);
 3  Group levels according to their index i mod p;
 4  There is such a group i_0 of levels that in total has cardinality ⩽|V(G)|/p;
 5  foreach i = i_0 mod p do
 6      if c(i) > 3p then
 7          for j = 1,..., ⌊c(i)/p⌋ do
 8              Collect equidistant elements w(i,j);
 9              Let P(i,j) be the path from w(i,j) down to level L(i − p);
10  S = ⋃_{i=i_0 mod p} L(i) ∪ ⋃_{i=i_0 mod p, j=1,...⌊c(i)/p⌋} P(i,j)
```

(i) *G is a minor of G',*
(ii) $|V(G')| \leqslant 10|V(G)|$,
and such a graph G' can be found in linear time.

Proof of Main Theorem, part (iii). If we chose

$$\mathrm{p} = \mathrm{p}(n) = \mathtt{slow}(n) \tag{24}$$

and

$$\ell(n) = 15\mathtt{slow}(n)5^{2\mathtt{slow}(n)}, \tag{25}$$

we have by (2) that

$$\ell(n) \leqslant \tfrac{5}{6} \cdot \log\log(n) 5^{(1/9)\cdot\log\log(n)} \leqslant \tfrac{5}{6} \cdot \log\log(n) \log(n)^{\log 5/9} < \sqrt[3]{\log(n)} \tag{26}$$

and thus Lemmas 3.2 and 6.3 show the claim for narrow p(n)-patches. With Proposition 6.4 it remains to show how to obtain narrow p-patches from a triangulated planar graph of bounded degree 8. We mainly use the same idea that was used to split narrow p-patches themselves, see procedure `Patch` above.

First we compute a BFS-tree from an arbitrary vertex v_0 and partition the vertex set V into the levels L_i of distance i. Then we look for a value i_0 such that the union of the levels with $i \equiv i_0(\mathrm{p})$ is small enough. For every such i, we then look at the component that remains when we eliminate levels L_i and $L_{i-\mathrm{p}}$. If it is not a narrow p-patch we have to divide it further by paths of length p that join L_i and $L_{i-\mathrm{p}}$.

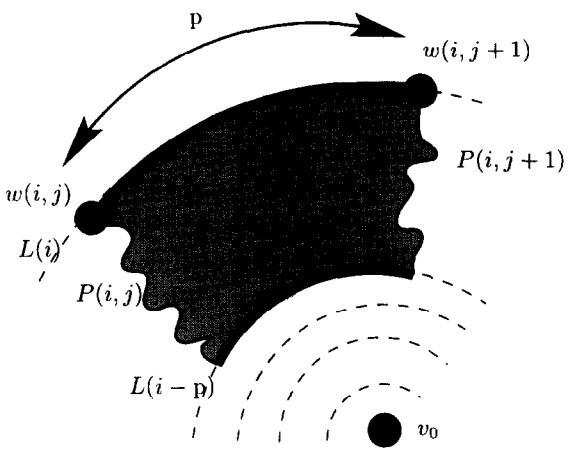


Fig. 5. Patch on level i.

Now every connected component of $G \setminus S$ (but the one of v_0) is a narrow p-patch, see Fig. 5. The only non-p-patch maybe the component of v_0 but this is of size at most 5^{p}, so it is already an appropriate cluster.

By line 6 we ensure that a level is only split into multiple patches if this is necessary. Observe that otherwise we possibly would collect too much vertices into the paths $P(i,j)$. So now S as chosen above is not too large. In fact, its cardinality can be estimated as follows:

$$|S| \leqslant \sum_{i \equiv i_0(\mathrm{p})} |L(i)| + \sum_{\substack{i \equiv i_0(\mathrm{p}) \\ c(i) > 3\mathrm{p}}} \sum_{j=1}^{\lfloor c(i)/\mathrm{p} \rfloor} |P(i,j)|. \tag{27}$$

The second term then is bounded by

$$\sum_{\substack{i \equiv i_0(\mathrm{p}) \\ c(i) > 3\mathrm{p}}} \sum_{j=1}^{\lfloor c(i)/\mathrm{p} \rfloor} \mathrm{p} \leqslant |V(G)|/\mathrm{p}. \tag{28}$$

So all together $|S|$ is bounded by $\leqslant 2|V(G)|/\mathrm{p}$ and we are done. □

7. Perspectives

7.1. Practical considerations: Reorganization of memory

Clearly, one disadvantage of our approach is the intensive use of a RAM as computational model. It is often claimed that using micro-encoding to handle "small" subproblems would only pay off in cases where the overall problem size is so huge, that we never expect it to occur in our limited universe. Therefore, there have been intensive studies to avoid such use of the RAM and to restrict the computational model to a *pointer machine*, PM, see e.g. [8].

On the other hand it is also often argued that α is such a slowly growing function that the factor is completely neglectable for practical purposes. So some people claim, that avoiding α-factors is a useless theoretical game.

Both criticisms go to short, since the real-world bottleneck for doing UF on large data as it occurs, e.g. in image processing is the *von Neumann bottleneck*, i.e. the different speed main memory and CPU have in modern computers. According to the book by [7, ch. 5], nowadays the factor for an access to main memory commonly is between 40 and 100 clock cycles, and, seen the difference in the development of CPU speed and memory access times, this factor is increasing. Modern computer architectures work against this with a hierarchical organization of memory, in particular by the introduction of caches.

Seen this, the bottleneck for `Union-Find` is the use of a dynamic data structure. The usual technique of path compression makes no guarantee at all where in memory the next item might be located: suppose we have to follow a chain of elements $v_1, v_2, \ldots$ and by bad luck v_1 is not present in cache. Then we have to wait 40 clock cycles, say, until we even know which element is next. If our elements are randomly spread in memory the probability that we do not have to wait is only

$$p_{\text{hit}} = \frac{\text{cache size}}{\text{data size}} \tag{29}$$

which for large data is a neglectable small number.

If we are doing `Unions` more or less arbitrary we may create chains that link memory items arbitrarily and then the real performance of the algorithm is dominated by loading data from memory. Clearly, that the inability to reflect this real world restriction applies to both, to the PM as well as to the RAM. So from that point of view, none of these models is preferable.

On the other hand the approach of clusters as presented in this paper easily allows a *reorganization of memory* that hopefully may cooperate with the strategies of modern architectures. Since here the data is fetched in larger blocks, the necessary wait is much less dramatic if memory access goes to "close" locations several times.

For a conventional UF data structure, it is very unlikely that dereferencing a parent pointer of a UF element stays on the same page or even on a small group of pages. In contrast to that, with a given valid clustering we can allocate each cluster continuously on subsequent pages of memory and do the same for the skeleton as a whole, this behavior changes drastically. Then a `Find` may either

(i) stay inside a cluster and thus on the pages allocated for it or

(ii) lead into the skeleton and thus stay on the pages allocated for the skeleton.

Thus, only a sublinear number of pages are potentially accessed. This is so, even if we do not impose any other restriction for the sequence of UF than to respect the underlying graph. So the probability p_{hit} that we can *guarantee* is much better than without the reorganization, namely

$$p_{\text{hit}} = \frac{\text{cache size}}{\text{sublinear function in the data size}}. \tag{30}$$

7.2. *Possible generalizations*

For graph algorithms people have in many cases been able to avoid the α-bottleneck of UF by looking closely the graphical structure occurring in their particular problem. We have the impression that there must be a common pattern behind all these attempts. In regard to Remark 2.4 and by hoping on the best of all worlds we conclude with the following conjecture.

Conjecture. *GUF is solvable in linear time on a RAM.*

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 (2) (1980) 59–65.
[2] H.L. Bodlaender, A tourist guide through treewidth, Acta Cybernet. 11 (1993) 1–23.
[3] H.L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, SIAM J. Comput. 25 (1996) 1305–1317.
[4] M.B. Dillencourt, H. Samet, M. Tamminen, A general approach to connected-component labeling for arbitrary image representations, J. Assoc. Comput. Mach. 39 (2) (1992) 253–280. Corr. pp. 985–986.
[5] C. Fiorio, J. Gustedt, Two linear time Union-Find strategies for image processing, Theoret. Comput. Sci. 154 (2) (1996) 165–181.
[6] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1984) 209–221.
[7] J.L. Hennessy, D.A. Patterson, Computer Architecture, A Quantitative Approach, 2nd ed., Morgan-Kaufmann, Los Altos, CA, 1996.
[8] J.A. La Poutré, Lower bounds for the Union-Find and the Split-Find problem on pointer machines, J. Comput. System Sci. 52 (1) (1996) 87–99.
[9] R.J. Lipton, R. E. Tarjan, A separator theorem for planar graphs, SIAM J. Appl. Math. 36 (1979) 177–189.
[10] K. Melhorn, Data Structures and Algorithms, vol 1: Sorting and Searching, Springer, Berlin, 1984.
[11] K. Mehlhorn, A. Tsakalidis, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. A, Algorithms and Complexity, ch. 6, Data Structures, Elsevier, Amsterdam, 1990, pp. 301–314.
[12] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. Assoc. Comput. Mach. 22 (1975) 215–225.
[13] R.E. Tarjan, A class of algorithms which require non-linear time to maintain disjoint sets, J. Comput. System Sci. 18 (1979) 110–127.