Optimal decremental connectivity in planar graphs

Jakub Łącki
University of Warsaw

Piotr Sankowski
University of Warsaw

31.01.2015

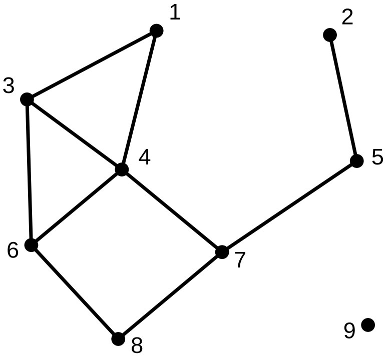
Dynamic connectivity

Input: an undirected graph G

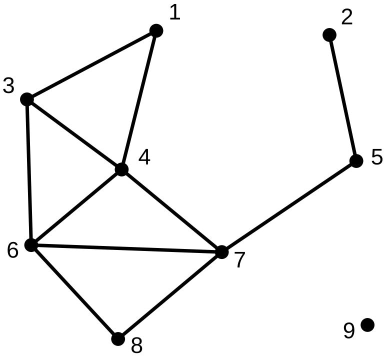
Update: Add/remove an edge.

Query: Are vertices u and v connected with a path?

Example

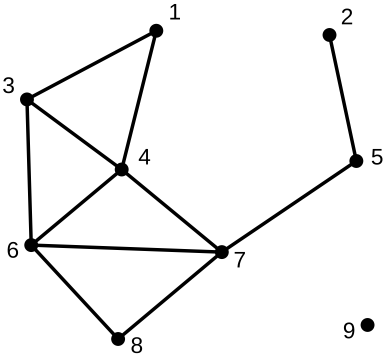


Example



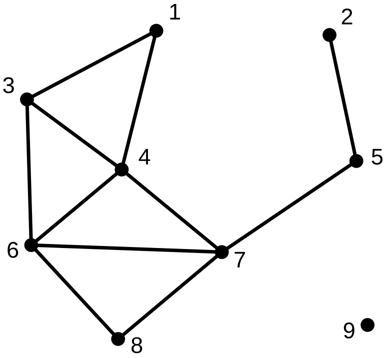
add(6,7)

Example



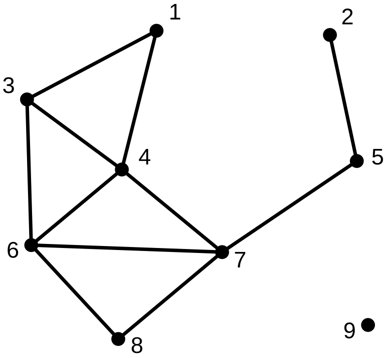
`query(1,6)?`

Example



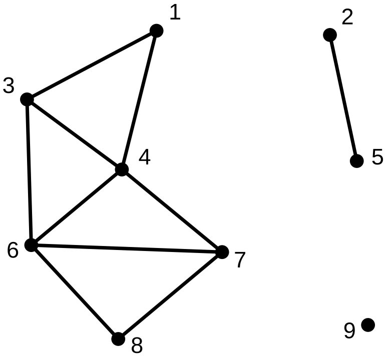
query(1,6)? YES

Example



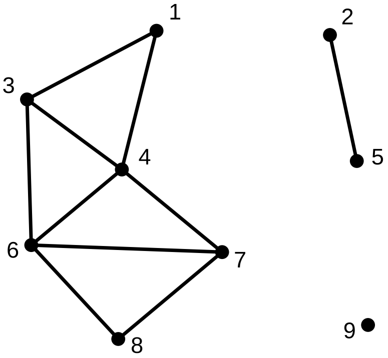
`remove(5,7)`

Example



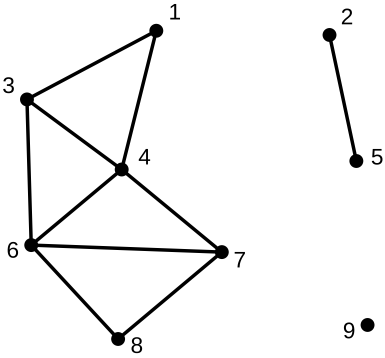
`remove(5,7)`

Example



`query(2,8)?`

Example



query(2,8)? NO

Three variants

Dynamic graph problems (in particular connectivity) come in three variants:

- *incremental* – edges can only be added
- *decremental* – edges can only be deleted
- *fully dynamic* – both edge insertions and deletions are allowed

Dynamic connectivity

Incremental connectivity (Tarjan 1975)

Updates and queries in $O(\alpha(n))$ amortized time.

Dynamic connectivity

Incremental connectivity (Tarjan 1975)

Updates and queries in $O(\alpha(n))$ amortized time.

Fully dynamic connectivity (Thorup 2000)

Updates in $O(\log n(\log\log n)^3)$ amortized time and queries in $O(\log n/\log\log n)$ time.

Dynamic connectivity

Incremental connectivity (Tarjan 1975)

Updates and queries in $O(\alpha(n))$ amortized time.

Fully dynamic connectivity (Thorup 2000)

Updates in $O(\log n(\log\log n)^3)$ amortized time and queries in $O(\log n/\log\log n)$ time.

Decremental connectivity (Thorup 1999)

For $m = \Omega(n(\log n \log\log n)^2)$ any sequence of deletions can be handled in $O(m \log n)$ time. Queries answered in $O(1)$ time.

Fully dynamic connectivity in planar graphs

Planar graphs (Eppstein et al. 1996)

Updates in $O(\log^2 n)$ worst-case time and queries in $O(\log n)$ time.

Fully dynamic connectivity in planar graphs

Planar graphs (Eppstein et al. 1996)

Updates in $O(\log^2 n)$ worst-case time and queries in $O(\log n)$ time.

Plane graphs (Eppstein et al. 1992)

Updates and queries in $O(\log n)$ amortized time, provided that the embedding does not change over time.

Fully dynamic connectivity in planar graphs

Planar graphs (Eppstein et al. 1996)

Updates in $O(\log^2 n)$ worst-case time and queries in $O(\log n)$ time.

Plane graphs (Eppstein et al. 1992)

Updates and queries in $O(\log n)$ amortized time, provided that the embedding does not change over time.

Corollary

There exists a decremental connectivity algorithm for planar graphs that handles all updates in $O(n \log n)$ time.

Dynamic connectivity in trees

Fully dynamic connectivity in trees (Sleator and Tarjan 1982)

Updates and queries in $O(\log n)$ worst-case time.

Dynamic connectivity in trees

Fully dynamic connectivity in trees (Sleator and Tarjan 1982)

Updates and queries in $O(\log n)$ worst-case time.

Decremental connectivity in trees (Alstrup et al. 1997)

Any sequence of edge deletions can be handled in $O(n)$ time and queries can be answered in $O(1)$ time.

Dynamic connectivity in trees

Fully dynamic connectivity in trees (Sleator and Tarjan 1982)

Updates and queries in $O(\log n)$ worst-case time.

Decremental connectivity in trees (Alstrup et al. 1997)

Any sequence of edge deletions can be handled in $O(n)$ time and queries can be answered in $O(1)$ time.

Both in general graphs and trees there exists a decremental connectivity algorithm, which is faster then the fully dynamic one.

Our result

Theorem

There exists a decremental connectivity algorithm for planar graphs that supports updates in $O(n)$ total time and answers queries in constant time.

Overview

1. Introduction
   - Dynamic connectivity
   - Related work
   - Our result
2. Our algorithms
   - Critical deletions
   - O(n log n) time algorithm
   - O(n log log n) time algorithm
   - O(n log log log n) time algorithm
   - O(n) time algorithm

Preliminaries

- We work with a planar graph $G = (V, E)$, subject to edge deletions
- G has n vertices and $O(n)$ edges
- We may assume that G is initially connected, and the degree of every vertex is at most 3

Critical deletion

Definition

An edge deletion is called *critical* if it increases the number of connected components in a graph.

Critical deletion

Definition

An edge deletion is called *critical* if it increases the number of connected components in a graph.

We first show how to detect critical deletions.

Theorem (Euler's formula)

Let $G = (V, E)$ be a planar graph, $v = |V|, e = |E|$, f be the number of faces, and c be the number of connected components of G. Then

$$v - e + f = c + 1$$

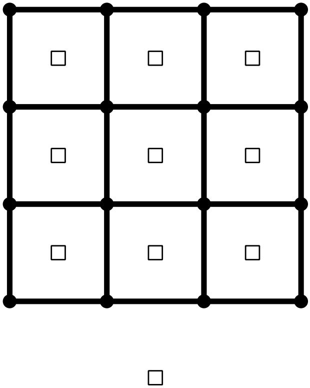
Detecting critical deletions

- If we know the number of vertices, edges and faces, we can obtain the number of connected components.
- Thus, we need to maintain the number of faces of G.

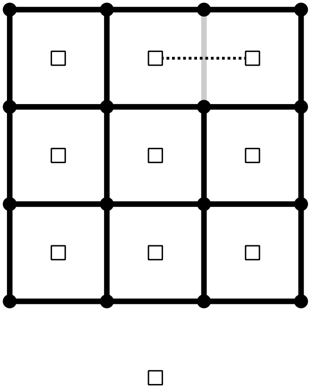
Detecting critical deletions

- If we know the number of vertices, edges and faces, we can obtain the number of connected components.
- Thus, we need to maintain the number of faces of G.
- When an edge is deleted, we merge faces on both sides.

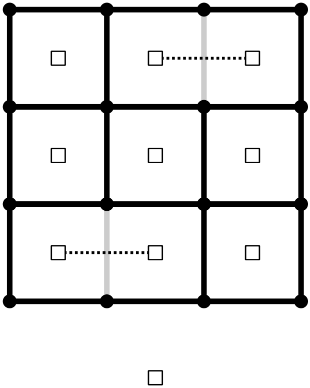
Example



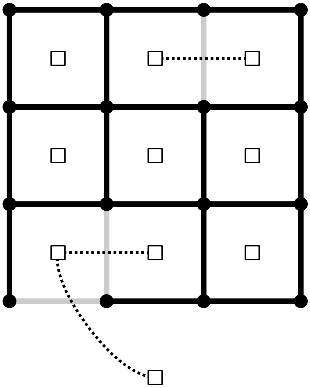
Example



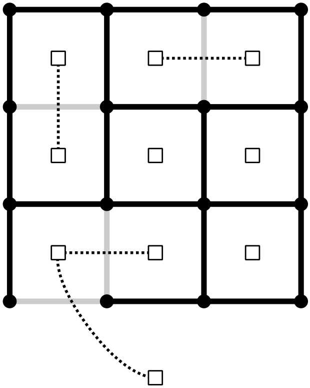
Example



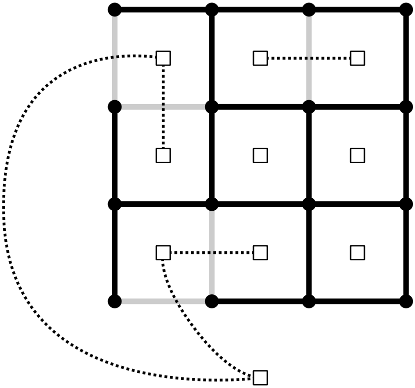
Example



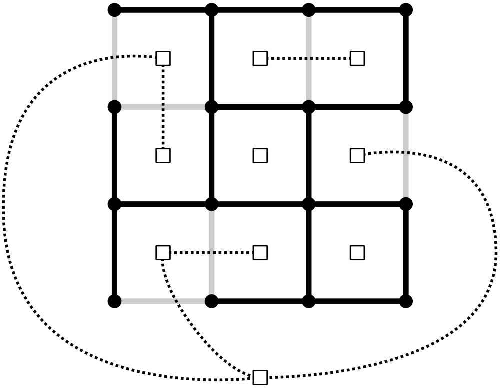
Example



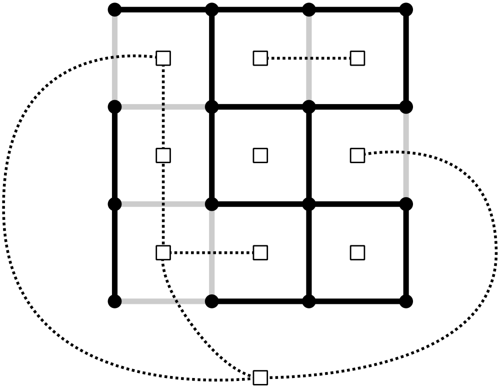
Example



Example



Example



Detecting critical deletions

- Maintaining the number of faces of G is the union-find problem.

Detecting critical deletions

- Maintaining the number of faces of G is the union-find problem.
- In addition, the set of allowed union operations forms a planar graph.
- As shown by Gustedt (1998), in this variant in union-find all updates can be processed in $O(n)$ time.

Detecting critical deletions

- Maintaining the number of faces of G is the union-find problem.
- In addition, the set of allowed union operations forms a planar graph.
- As shown by Gustedt (1998), in this variant in union-find all updates can be processed in $O(n)$ time.

We can detect critical deletions with $O(n)$ overhead.

O(n log n) time algorithm

The algorithm, for every vertex v maintains a *cc-identifier*, that is a unique identifier of its connected component. Thus

```
query(u, w):
return true iff cc-identifier[u] = cc-identifier[w]
```

O(n log n) time algorithm

The algorithm, for every vertex v maintains a *cc-identifier*, that is a unique identifier of its connected component. Thus

```
query(u, w):
return true iff cc-identifier[u] = cc-identifier[w]
```

Queries are answered in constant time.

O(n log n) time algorithm

The algorithm, for every vertex v maintains a *cc-identifier*, that is a unique identifier of its connected component. Thus

```
query(u, w):
return true iff cc-identifier[u] = cc-identifier[w]
```

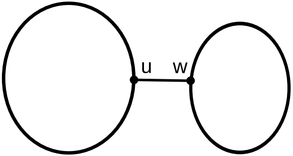
Queries are answered in constant time.

How to update the cc-identifiers after an edge deletion?

O(n log n) time algorithm

After an edge *uw* is deleted:

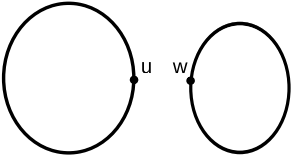
- If the deletion is not critical, do nothing.
- Otherwise, start two parallel DFS searches from *u* and *w*



O(n log n) time algorithm

After an edge uw is deleted:

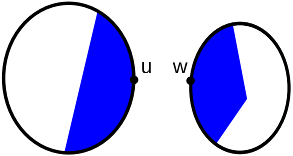
- If the deletion is not critical, do nothing.
- Otherwise, start two parallel DFS searches from u and w



O(n log n) time algorithm

After an edge *uw* is deleted:

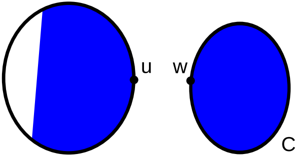
- If the deletion is not critical, do nothing.
- Otherwise, start two parallel DFS searches from *u* and *w*



O(n log n) time algorithm

After an edge *uw* is deleted:

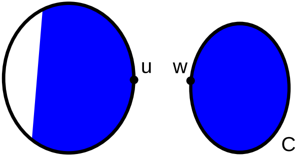
- If the deletion is not critical, do nothing.
- Otherwise, start two parallel DFS searches from u and w, and stop them once the first one finishes and finds a connected component C. This takes $O(|C|)$ time.



O(n log n) time algorithm

After an edge uw is deleted:

- If the deletion is not critical, do nothing.
- Otherwise, start two parallel DFS searches from u and w, and stop them once the first one finishes and finds a connected component C. This takes $O(|C|)$ time.



C is the *smaller* among the two new connected components.

O(n log n) time algorithm

We assign a new cc-identifier to every vertex of C

O(n log n) time algorithm

We assign a new cc-identifier to every vertex of C

Analysis

- The running time is proportional to the number of changes of cc-identifiers.

O(n log n) time algorithm

We assign a new cc-identifier to every vertex of C

Analysis

- The running time is proportional to the number of changes of cc-identifiers.
- Every time a vertex changes its cc-identifier, the size of its connected component halves.

O(n log n) time algorithm

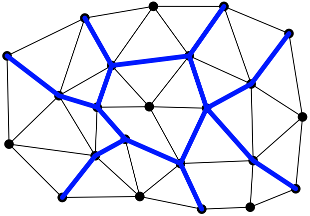
We assign a new cc-identifier to every vertex of C

Analysis

- The running time is proportional to the number of changes of cc-identifiers.
- Every time a vertex changes its cc-identifier, the size of its connected component halves.
- All updates are processed in $O(n \log n)$ time.

O(n log log n) time algorithm

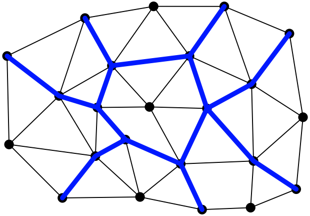
To obtain a faster algorithm we use an *r-division*.



It is a division of a graph into *regions* of size at most r. Each region has at most $\sqrt{r}$ vertices on its *boundary*.

O(n log log n) time algorithm

To obtain a faster algorithm we use an *r-division*.



It is a division of a graph into *regions* of size at most r. Each region has at most $\sqrt{r}$ vertices on its *boundary*.
We use $r = \log^2 n$.

O(n log log n) time algorithm

Overall idea

- Build an r-division for $r = \log^2 n$.
- Run the $O(n \log n)$ algorithm inside every region.
- Use a similar idea to maintain cc-identifiers of boundary vertices (there are $O(n/\log n)$ of them).

This way we obtain an $O(n \log \log n)$ time algorithm.

O(n log log log n) time algorithm

Overall idea

- Build an r-division for $r = \log^2 n$.
- Run the $O(n \log \log n)$ algorithm inside every region.
- Use a similar idea to maintain cc-identifiers of boundary vertices (there are $O(n/\log n)$ of them).

This way we obtain an $O(n \log \log \log n)$ time algorithm.

How to speed it up to O(n)?

The only slow part of the $O(n \log\log\log n)$ time algorithm is maintaining connectivity in graphs on $O(\log^2 \log n)$ vertices.

How to speed it up to O(n)?

The only slow part of the $O(n \log\log\log n)$ time algorithm is maintaining connectivity in graphs on $O(\log^2 \log n)$ vertices.

- There are $\leq 2^{s^2}$ graphs (including non-planar) on s vertices, and $2^{s^2} = 2^{O(\log^4 \log n)} = n^{o(1)}$.

How to speed it up to O(n)?

The only slow part of the $O(n \log\log\log n)$ time algorithm is maintaining connectivity in graphs on $O(\log^2 \log n)$ vertices.

- There are $\leq 2^{s^2}$ graphs (including non-planar) on s vertices, and $2^{s^2} = 2^{O(\log^4 \log n)} = n^{o(1)}$.
- Each such graph can be encoded as a binary string of length $s^2 = o(\log n)$, which fits in a machine word.

How to speed it up to O(n)?

The only slow part of the $O(n \log\log\log n)$ time algorithm is maintaining connectivity in graphs on $O(\log^2 \log n)$ vertices.

- There are $\leq 2^{s^2}$ graphs (including non-planar) on s vertices, and $2^{s^2} = 2^{O(\log^4 \log n)} = n^{o(1)}$.
- Each such graph can be encoded as a binary string of length $s^2 = o(\log n)$, which fits in a machine word.
- We can precompute connected components of all small graphs in $o(n)$ time!

Main result

Theorem

There exists a decremental connectivity algorithm for planar graphs that supports updates in $O(n)$ total time and answers queries in constant time.

Main result

Theorem

There exists a decremental connectivity algorithm for planar graphs that supports updates in $O(n)$ total time and answers queries in constant time.

Thank you!

Questions?