# Testing connectivity of faulty networks in sublinear time[☆]

Tomáš Dvořák[a], Jiří Fink[a], Petr Gregor[a], Václav Koubek[a], Tomasz Radzik[b]

[a] *Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic*
[b] *Department of Computer Science, King's College London, United Kingdom*

**Abstract**

Given a set $F$ of vertices of a connected graph $G$, we study the problem of testing the connectivity of $G - F$ in polynomial time with respect to $|F|$ and the maximum degree $\Delta$ of $G$. We present two approaches. The first algorithm for this problem runs in $O\left(|F|\Delta^2\varepsilon^{-1}\log(|F|\Delta\varepsilon^{-1})\right)$ time for every graph $G$ with vertex expansion at least $\varepsilon > 0$. The other solution, designed for the class of graphs with cycle basis consisting of cycles of length at most $l$, leads to $O\left(|F|\Delta^{\lceil l/2\rceil}\log(|F|\Delta^{\lfloor l/2\rfloor})\right)$ running time. We also present an extension of this method to test the biconnectivity of $G - F$ in $O\left(|F|\Delta^l\log(|F|\Delta^l)\right)$ time.

*Keywords:* biconnectivity, connectivity, cycle basis, faulty vertex, interconnection network, vertex expansion

## 1. Introduction

In the study of reliability of interconnection networks, one may ask whether a given network preserves its functionality even if some of its nodes become overloaded or unavailable. Typically, the network is large and faults occur dynamically. Hence it may not be affordable to search the complete network. However, it seems to be reasonable to allow for searching up to some bounded distance from the faulty nodes.

In that way, some properties may be tested easily; for example, consider the minimal degree. When a faulty node is removed, it suffices to check all its neighbors to find out whether some of them has the (new) degree smaller than the (old) minimal degree. However, in this paper we are interested in testing global properties, though very basic. It is natural to assume that each reasonable interconnection network should be (bi)connected. If some nodes of the network become faulty, will this property still hold? If the network is modeled as a simple undirected graph $G$ and $G - F$ denotes the graph obtained from $G$ by removing a set $F$ of faulty vertices, our problem may be formulated as follows: Given a set $F$ of vertices of a (bi)connected graph $G$, is $G - F$ still (bi)connected?

Although there are notorious algorithms for these problems which are linear with respect to the size of the graph $G$ to be tested, our aim is to design algorithms that search $G$ only locally; their time complexity then would be sublinear in the size of $G$. More precisely, we require the running time to be bounded by a polynomial with respect to the number of faulty vertices and the maximum degree $\Delta(G)$ of the graph $G$. It follows that the network itself cannot be given as a part of the input. Instead, we use a so called *neighborhood oracle*. A neighborhood oracle for a graph $G$ is a function $n_G$ which, given a vertex $v$, returns the set of all neighbors of $v$ in $G$. Algorithmically, it can be viewed as a procedure from an external library. We assume that each call to $n_G(v)$ takes time proportional to the number of neighbors of $v$ in $G$.

Is there is an algorithm which, given a set $F$ of faults and a neighborhood oracle $n_G$, decides whether $G - F$ is connected in polynomial time with respect to $|F|$ and $\Delta(G)$? If the tested graph $G$ may be arbitrary, it is easy to see that no such general algorithm exists. Indeed, consider a cycle containing a vertex

[☆] This research was partially supported by the Czech Science Foundation Grant 201/08/P298 and by the project 1M0545 of the Czech Ministry of Education. This paper extends our conference paper [3].

$u$ and a path with a middle vertex $v$. A deletion of $u$ and $v$ results in a connected and disconnected graph, respectively. To distinguish these two cases, the algorithm has to search $G$ far from $u$ and $v$, which is beyond the time restriction if the cycle and path are large enough, respectively.

Therefore, it is necessary to restrict ourselves to some proper subclass of connected graphs. In [3] we studied this problem for the class of hypercubes, one of the classical architectures for interconnection networks [8].

In this paper, we present two methods of solution to our problem. The first approach is designed for graphs with good vertex-expansion properties. The resulting algorithm, described in Section 3, runs in $O\left(|F|\Delta(G)^2\varepsilon^{-1}\log(|F|\Delta(G)\varepsilon^{-1})\right)$ time where $\varepsilon > 0$ is a lower bound on the vertex expansion of $G$. The other approach is based on local properties of a class $\mathcal{G}(l)$ of connected graphs that possess a cycle basis consisting of cycles of length at most $l$. In Section 4 we describe an algorithm for testing the connectivity of $G - F$ for any graph $G \in \mathcal{G}(l)$ that runs in $O\left(|F|\Delta(G)^{\lceil l/2\rceil}\log(|F|\Delta(G)^{\lfloor l/2\rfloor})\right)$ time. Section 5 presents an extension of our method which leads to an algorithm for testing the biconnectivity of $G - F$ in $O\left(|F|\Delta(G)^l\log(|F|\Delta(G)^l)\right)$ time. We also suggest a possible generalization towards testing the connectivity of a higher order.

## 2. Preliminaries

In this paper, we consider only finite undirected graphs without loops or multiple edges. As usual, $V(G)$ and $E(G)$ denote the vertex and edge sets of a graph $G$. The degree of a vertex $v$ in $G$ se denoted by $\deg_G(v)$, the subscript being omitted if no ambiguity may arise. We use $\Delta(G)$ to denote the maximum degree of $G$. For a set $F \subseteq V(G)$, $G - F$ denotes the subgraph of $G$ induced by the set $V(G) \setminus F$. To simplify the notation, instead of $G - \{v\}$ we simply write $G - v$. For a subset of vertices $W \subseteq V(G)$, let $\partial W$ denote the set of *neighbors* of $W$, that is, the set of vertices which are not in $W$ but are connected by single edges with $W$:

$$\partial W = \{v \in V(G) - W \mid \ vw \in E(G) \text{ for some } w \in W\}.$$

If $W = \{w\}$, we simply write $\partial w$ instead of $\partial\{w\}$.

A *walk* in a graph $G$ (of length $k-1$) is a sequence $W = (v_1, v_2, \ldots, v_k)$ of vertices of $G$ such that $v_iv_{i+1}$ is an edge of $G$ for all $1 \le i < k$. A *path* is a walk in which all vertices are pairwise distinct. A *uv-walk* is a walk that starts with $u$ and ends with $v$. A *closed u-walk* is a $uu$-walk. A *cycle* is a closed walk in which all vertices but the endvertices are distinct. Note that in particular, the sequence $(u)$ for a vertex $u$ is considered to be a path (of length 0), and the sequence $(u, v, u)$ where $uv$ is an edge of $G$ is considered to be a cycle (of length 2). We use $E(C)$ to denote the set of edges of a cycle $C$.

For a $uv$-walk $W_1 = (u = u_1, u_2, \ldots, u_k = v)$ and a $vw$-walk $W_2 = (v = v_1, v_2, \ldots, v_l = w)$, let $(W_1, W_2)$ denote the $uw$-walk $(u = u_1, u_2, \ldots, u_k = v = v_1, v_2, \ldots, v_l = w)$ formed by a concatenation of $W_1$ and $W_2$. If a walk $W'$ is a contiguous subsequence of a walk $W$, we say that $W'$ is a *subwalk* of $W$. Given a walk $W = (w_1, w_2, \ldots, w_n)$, the reverse of $W$ is denoted by $W^R = (w_n, w_{n-1}, \ldots, w_1)$.

A graph $G$ is *connected* if there is a $uv$-path for an arbitrary pair $u, v$ of vertices of $G$. A maximal connected subgraph of $G$ is called a *component* of $G$. A vertex $v$ of $G$ is called a *cutvertex* if $G - v$ is has more components than $G$. A connected graph on at least 3 vertices with no cutvertex is called *biconnected*. More generally, a set $X \subseteq V(G)$ is a *separator* if $G - X$ has more components than $G$. We say that $G$ is *k-connected* if $G$ is connected, $|V(G)| > k$ and $G$ has no separator of size less than $k$. The maximum $k$ such that $G$ is $k$-connected is the *connectivity* $\kappa(G)$ of $G$.

### *2.1. Cycle space*

The set $2^E$ of all subsets of $E$ forms an $|E|$-dimensional vector space over GF(2) with vector addition $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$ and scalar multiplication $1 \cdot X = X$ and $0 \cdot X = \emptyset$ for all $X, Y \in 2^E$. A set $X \subseteq E$ is called *Eulerian* if every vertex of $(V, X)$ has even degree. The set $\mathcal{C}(G)$ of all Eulerian subsets of $E$ forms a vector subspace of $2^E$, called the *cycle space* of $G$. The dimension of $\mathcal{C}(G)$ equals $|E| - |V| + k$ where $k$ the number of components of $G$; see [1].

Let $L(B)$ be the length of the longest cycle of a basis $B$ of the cycle space $\mathcal{C}(G)$. Let $L(G)$ be the minimum of $L(B)$ over all bases $B$ of $\mathcal{C}(G)$. For an integer $l \geq 2$ we define the class of graphs

$$\mathcal{G}(l) = \{G \mid G \text{ is connected and } L(G) = l\} \ .$$

Note that $\mathcal{G}(2)$ is the class of trees, and every graph belongs to $\mathcal{G}(l)$ for some $l \geq 2$. In Section 4 we describe an algorithm which tests whether graph $G - F$ is connected provided that $G \in \mathcal{G}(l)$.

When studying $L(G)$, it may be of interest to note the relationship to the smallest possible sum of the lengths of all cycles included in a basis of $\mathcal{C}(G)$. Let $S(B)$ be the sum of lengths of all cycles of a basis $B$. A *shortest basis* of $\mathcal{C}(G)$ is a basis with the smallest $S(B)$ over all bases $B$ of $\mathcal{C}(G)$. Chickering et al. [2] showed that every shortest basis $B$ of $\mathcal{C}(G)$ satisfies $L(B) = L(G)$. This result has significant consequences, as a shortest cycle basis of a given graph with $n$ vertices and $m$ edges may be found in polynomial time. There are a number of algorithms to this problem, ranging from a classical $O(m^3n)$-time solution due to Horton [5] up to a more recent $O(m^2n)$-time algorithm of Kavitha et al. [7]. It follows that the value of $L(G)$ may be determined within the same time complexity.

Finally, let us mention that Imrich and Stadler [6] proved that

$$L(G \,\square\, H) = \max\{L(G), L(H), 4\}$$

where $\square$ stands for the Cartesian product. If follows that for example the $n$-dimensional hypercube $Q_n$ satisfies $L(Q_n) = 4$ for every $n \geq 2$.

### *2.2. Model of computation*

Recall that the purpose of this paper is to describe an algorithm for testing the connectivity of a graph $G - F$ for a connected graph $G$ with a set $F$ of faulty vertices. Since we require the running time to be bounded by a polynomial with respect to $|F|$ and $\Delta(G)$, it follows that $G$ itself cannot be given as a part of the input.

We therefore assume that the input of our algorithms consists of a set $F$ of faulty vertices and a neighborhood oracle $n_G$. A neighborhood oracle for a graph $G$ is a function $n_G$ which, given a vertex $v$, returns the set $\partial v$ of all neighbors of $v$. We assume that each call to $n_G(v)$ takes $O(|\partial v|)$ time. Moreover, we work in a commonly used $O(1)$-time unit model; that is, we assume that each elementary operation on vertices (such as their comparison) takes $O(1)$ time.

In each of the algorithms described below, we apply $n_G$ to construct a certain auxiliary subgraph of $G$. To that end, we need a data structure to maintain both the set $F$ and the set of all vertices that have been generated by the oracle so far. Note that we cannot use an array indexed by $V(G)$ as usual in standard graph algorithms, since this would impose an unacceptable $\Omega(|V(G)|)$ bound on the working space. We therefore employ a dictionary data structure $D$ for this purpose. Consequently, even though we assume that oracle $n_G$ needs only constant time to generate a vertex $v$, each such operation is accompanied by an attempt to access a record on $v$ in the dictionary $D$, which requires $O(\log |D|)$ time.

## 3. Expansion approach

We start with an approach designed for graphs with good vertex-expansion properties. We consider an undirected connected graph $G$ with the set of vertices $V = V(G)$ and the set of edges $E = E(G)$. Recall that $\partial W$ denotes the set of neighbors of $W \subseteq V$. The (*vertex*) *expansion* of $G$ is defined as

$$\delta(G) = \min\left\{\frac{|\partial W|}{|W|} : \ W \subseteq V(G), 1 \leq |W| \leq |V(G)|/2\right\}.$$

For two subsets of vertices $W$ and $U$, let $E(W, U)$ denote the set of edges between $W$ and $U$:

$$E(W, U) = \{wu \in E(G) : \ w \in W, u \in U\}.$$

For a given subset of vertices $F \subseteq V(G)$, we check if $G - F$ is connected by searching the graph from the vertices in $F$. We obviously can stop the search as soon as we have identified a connected component of $G - F$. But if we keep searching the graph and do not see a connected component, do we have to continue until the whole graph is inspected? We may be able to stop earlier, if we assume something about set $F$ or the structure of graph $G$ that implies existence of a small connected component in $G - F$, if $G - F$ is not connected. If a subset of vertices $Y \subseteq V - F$ is a connected component of $G - F$, then $\partial Y \subseteq F$. Moreover, if $|Y| \leq |V|/2$ (and $G - F$ must have a component of size at most $|Y|/2$, if $G - F$ is not connected), then $|\partial Y| \geq \delta(G)|Y|$. Therefore, if the size of a connected component is at most $|V|/2$, then its size is actually at most $|F|/\delta(G)$.

Thus if we know a lower bound $\varepsilon > 0$ on $\delta(G)$, then to decide whether $G - F$ is connected, it suffices to look for a connected component of size at most $|F|/\varepsilon$. Such a search can be performed in the following straightforward way. For each vertex $v \in \partial F$, search $G - F$ from $v$ using a systematic efficient graph search strategy like the depth-first search or the breadth-first search, until either all reachable vertices are processed or $|F|/\varepsilon$ vertices are processed. We say that a vertex is *processed*, if the list of its adjacent vertices has already been examined during this search. The previous paragraph implies that at the end of the computation the set of processed vertices must contain all connected components of $G - F$ of size at most $|V|/2$. This is enough information to decide whether $G - F$ is connected. Regarding the running time, we process first the vertices in $F$ to find $\partial F$, and then perform $|\partial F| \leq |F|\Delta$ searches and process at most $|F|/\varepsilon$ vertices during each search. Processing one vertex takes $O(\Delta)$ time, where $\Delta = \Delta(G)$ is the maximum degree of a vertex. Hence the total running time is

$$O\left(\frac{|F|^2\Delta^2}{\varepsilon}\right).$$

In the remaining part of this section we show how the above bound can be improved by a more cautious search of the graph. Consider searching $G - F$ starting from a vertex $v \in \partial F$. Each vertex discovered during this search is either *waiting* to be processed or is already processed. Let $X$ and $Z \subseteq X$ be the set of vertices discovered so far during this search and the set of vertices already processed during this search, respectively. The main idea of the improved algorithm is that we can stop this search as soon as $|X| > |E(X, F)|/\varepsilon$. This condition may be satisfied much earlier than the condition that $|Z| \geq |F|/\varepsilon$, which was used as the stopping criteria in our first approach.

Another aspect of the new algorithm, which is important for efficient performance, is merging the sets of vertices discovered in different searches. When we search from a vertex $u \in \partial F$ and encounter a vertex which has been earlier discovered during the search from another vertex $v \in \partial F$, then we "merge" the set $X_u$ of the vertices discovered so far during the search from $u$ with the set $X_v$ of the vertices discovered so far during the search from $v$. In our algorithm we allow different searches, which have started from different vertices in $\partial F$, to progress in an interleaving way, rather than starting the next search only when the previous one has been completed. However, this interleaving way of progressing the searches is not essential and we use it only because it simplifies description of the overall algorithm. We proceed now to the detailed description of the algorithm.

The pseudocode of algorithm Balanced-Search is given below. The input is an undirected graph $G$ given by an oracle $n_G$ and assumed to be connected, a lower bound $\varepsilon > 0$ on the expansion $\delta(G)$ of graph $G$, and a subset of vertices $F \subseteq V = V(G)$. The output is the information whether graph $G - F$ is connected. The algorithm maintains the set $D \subseteq V - F$ of all discovered vertices in $G - F$, partitioned into the set $P$ of vertices already processed and the set $W$ of vertices waiting to be processed. During the initial part of the computation, lines 1–6, the algorithm first processes the vertices in $F$ (line 1), and then initializes $P \leftarrow \emptyset$ and $W \leftarrow \partial F$.

Throughout the computation the set of known (discovered) edges in $G - F$ is $H = E(P, V)$, that is, the set of edges adjacent to the processed vertices in $V - F$. Thus the known subgraph of $G - F$ is the graph $(D, H)$. The algorithm maintains the family $\mathcal{COMP}$ of connected components of this graph, and we can think of each $C \in \mathcal{COMP}$ as corresponding to the set of vertices discovered during one search process. A component $C \in \mathcal{COMP}$ is *active* if there is at least one waiting vertex in $C$ (that is, $C \cap W \neq \emptyset$) and $|C| \leq |E(C, F)|/\varepsilon$. Initially $D = \partial F$, $H = \emptyset$, and all components in $\mathcal{COMP}$ are singletons, representing the

**Algorithm 1:** BALANCED-SEARCH

**Input**: Oracle $n_G$ for graph $G$, $\varepsilon$ such that $0 < \varepsilon \leq \delta(G)$, $F \subseteq V = V(G)$.
**Output**: Information whether $G - F$ is connected.

1 Reveal all edges in $E(F, V)$ by calling oracle $n_G$ for each $v \in F$;
2 $W \leftarrow \partial F$ ; // discovered waiting vertices in $V - F$
3 $P \leftarrow \emptyset$ ; // discovered processed vertices in $V - F$
4 $D \equiv P \cup W$ ; // all discovered vertices in $V - F$
5 $H \equiv E(P, V)$ ; // all discovered edges in $G - F$
6 $\mathcal{COMP} \leftarrow$ the family of connected components of graph $(D, H)$;
7 **while** $C \cap W \neq \emptyset$ *for each* $C \in \mathcal{COMP}$, *and* $|C| \leq |E(C, F)|/\varepsilon$ *for some* $C \in \mathcal{COMP}$ **do**
8 $C \leftarrow$ a component in $\mathcal{COMP}$ such that $|C| \leq |E(C, F)|/\varepsilon$;
9 $w \leftarrow$ a vertex in $C \cap W$;
10 reveal all neighbors of $w$ using the oracle $n_G$;
11 $W \leftarrow W - \{w\}$; $P \leftarrow P \cup \{w\}$;
12 **foreach** $x \in \partial w$ **do**
13 **if** $x \notin F$ **then**
14 **if** $x \notin D$ **then**
15 add $x$ to $W$ and to component $C$;
16 **else** // $x \in D$
17 $C' \leftarrow$ component in $\mathcal{COMP}$ containing $x$;
18 **if** $C' \neq C$ **then**
19 merge $C$ and $C'$ into one component: $C'$ is removed from $\mathcal{COMP}$ and added to $C$ ($C \leftarrow C \cup C'$);
20 **if** $|\mathcal{COMP}| \geq 2$ *and exists* $C \in \mathcal{COMP}$ *such that* $C \cap W = \emptyset$ **then**
21 // such $C$ is a proper connected component of $G - F$
22 **ouput:** graph $G - F$ is not connected;
23 **else**
24 **ouput:** graph $G - F$ is connected.

starting vertices for searching $G - F$. Check also that all these singleton components are active.

The main part of the algorithm, lines 7–19, is the iterative process of taking one (arbitrary) active component $C \in \mathcal{COMP}$ and processing one (arbitrary) waiting vertex $w$ in $C$. The processing of vertex $w$ amounts to moving $w$ from $W$ to $P$ (that is, changing the status of $w$ from "waiting" to "processed"), asking oracle $n_G$ for the list of neighbors of $w$ in $G$, and performing the following computation for each neighbor $x$ of $w$. If $x \in F$, then we do not do anything. If $x \notin F \cup D$, then $x$ is a newly discovered vertex, which is added to component $C$ and marked as a waiting vertex. If $x \in D$ and $x$ is in the same component $C$ where vertex $w$ is, then we do not need to do anything: edge $wx$ joins two vertices which are already known to be in the same connected component of $G - F$. If $x \in D$ and $x$ is in a component $C' \in \mathcal{COMP}$ other than $C$, then components $C$ and $C'$ are merged. Formally, $C'$ is removed from $\mathcal{COMP}$ and $C$ becomes $C \cup C'$.

The main part of the algorithm ends either when one component in $\mathcal{COMP}$ does not have any waiting vertices left, or when no active component is left in $\mathcal{COMP}$. In the final part of the algorithm, lines 20–23, the decision is made, and output, whether $G - F$ is connected. If there are at least two components left in $\mathcal{COMP}$ and at least one of them does not contain any waiting vertices, then the algorithm declares that $G-F$ is not connected. Otherwise, the algorithm declares that $G-F$ is connected. We prove in Theorem 3.1 that these decisions are correct. We remark that the decision that $G - F$ is not connected is the easier one to justify. If a component $C$ contains only processed vertices (no waiting vertices), then for each edge $vx$ with $v \in C$, vertex $x$ cannot be undiscovered and cannot be in another component (those two components would have merged earlier), so it has to be either in $C$ or in $F$. Thus in this case $C$ is disconnected in $G-F$ from the other components in $\mathcal{COMP}$, so $G - F$ is not connected.

**Theorem 3.1.** *For a connected graph $G$ given by a neighborhood oracle $n_G$, a lower bound $\varepsilon > 0$ on the vertex expansion of $G$, and a subset of vertices $F \subseteq V = V(G)$, algorithm* Balanced-Search *terminates and correctly identifies whether $G - F$ is connected.*

*Proof.* The algorithm always terminates because each iteration of the "while" loop increases by one the number of processed vertices, and a processed vertex remains processed throughout the computation.

We consider now the conditions checked in line 20, after the termination of the "while" loop. Throughout the computation, $\partial F \subseteq \bigcup C \equiv \bigcup_{C \in \mathcal{COMP}} C$ (initially, in line 6, $\partial F = \bigcup C$, and $\bigcup C$ can only grow). Thus if at the end of the computation $|\mathcal{COMP}| = 1$, then $\partial F$ is inside a connected component of $G - F$, implying that $G - F$ is connected, so the algorithm is correct in this case.

Assume now that at the end of the computation, $|\mathcal{COMP}| \geq 2$. If $C \cap W = \emptyset$ for some $C \in \mathcal{COMP}$, then $C$ is a proper subset of $V - F$ and $\partial C \subseteq F$, so removal of $F$ from $G$ disconnects $C$ from the other components in $\mathcal{COMP}$. Thus in this case the algorithm correctly decides that $G - F$ is not connected.

It remains to consider the case when at the end of the computation there are at least two components in $\mathcal{COMP}$ and $C \cap W \neq \emptyset$, for each $C \in \mathcal{COMP}$. In this case, the condition for the termination of the "while" loop implies that $|C| > |E(C, F)|/\varepsilon$, for each $C \in \mathcal{COMP}$. We take $Y \subset V - F$ as an arbitrary proper subset of $V - F$ such that $|Y| \leq |V|/2$, and show that $\partial Y$ is not contained in $F$, so $Y$ is not a connected component of $G - F$. If $\emptyset \neq Y \cap C \neq C$ for some $C \in \mathcal{COMP}$, then $\partial Y$ contains a vertex from $C$, so it cannot be a subset of $F$. Otherwise, if $Y \cap C = \emptyset$ or $Y \cap C = C$, for each $C \in \mathcal{COMP}$, then let $C_1, C_2, \ldots, C_r$ be the components in $\mathcal{COMP}$ which are contained in $Y$. If $r = 0$, then $\partial Y \cap F = \emptyset$, so $\partial Y$ is not a subset of $F$. If $r \geq 1$, then we have

$$\begin{aligned} |Y| &\geq |C_1| + |C_2| + \cdots + |C_r| \\ &> (|E(C_1, F)| + |E(C_2, F)| + \cdots + |E(C_r, F)|)/\varepsilon \\ &= |E(Y, F)|/\varepsilon \;\geq\; |E(Y, F)|/\delta(G). \end{aligned}$$

On the other hand, $\delta(G)|Y| \leq |\partial Y|$, so $|\partial Y| > |E(Y, F)| \geq |\partial Y \cap F|$, implying that $\partial Y$ is not a subset of $F$. Thus we have shown that there is no connected component in $G - F$ of size at most $|V|/2$, so the algorithm correctly decides that in this case $G - F$ is connected. □

The next theorem, and its proof, show that algorithm Balanced-Search has an efficient implementation.

**Theorem 3.2.** *Algorithm* Balanced-Search *can be implemented in such a way that for a graph $G$ given by a neighborhood oracle $n_G$, a lower bound $\varepsilon$ on the vertex expansion of $G$, and a subset of vertices $F \subseteq V = V(G)$, the running time is*

$$O\left(|F|\Delta^2\varepsilon^{-1}\log(|F|\Delta\varepsilon^{-1})\right), \tag{3.1}$$

*where $\Delta$ is the maximum degree of a vertex in $G$.*

*Proof.* We can show by induction on the steps of the algorithm that throughout the computation, for each $C \in \mathcal{COMP}$, the number of processed vertices in $C$ is

$$|C \cap P| \leq |E(C, F)|/\varepsilon. \tag{3.2}$$

These inequalities hold at the beginning of the computation, when $P = \emptyset$ and each component $C \in \mathcal{COMP}$ is a singleton. Before the processing of the waiting vertex $w$ in the selected component $C$ starts in line 10,

$$|C \cap P| \;\leq\; |C - \{w\}| \;\leq\; |C| - 1 \;\leq\; |E(C, F)|/\varepsilon - 1,$$

where the last inequality follows from the condition of the "while" loop in line 7. Hence, after changing the status of vertex $w$ from waiting to processed, Inequality (3.2) remains true for the selected component $C$. ($C$ has not changed while $|C \cap P|$ increased by 1). The only other time when a component or the number of processed vertices in a component changes is when two components $C$ and $C'$ are merged in line 19. The

number of processed vertices in the new component is $|C \cap P| + |C' \cap P|$ and the number of edges between the new component and set $F$ is $E(C, F) + E(C', F)$. Thus if (3.2) holds for components $C$ and $C'$ (the inductive hypothesis), then it also holds for the new component.

For any step during the computation, we bound, using Inequalities (3.2), the number of vertices which are not in $F$ and are already processed by this step:

$$|P| = \sum_{C \in \mathcal{COMP}} |C \cap P| \leq \sum_{C \in \mathcal{COMP}} |E(C, F)|/\varepsilon = |E(V - F, F)|/\varepsilon \leq |F|\Delta/\varepsilon.$$

Thus the total number of vertices processed during the main part of the algorithm is at most $|F|\Delta/\varepsilon$. This implies that the total number of iterations of the inner loop in lines 12–19 is $O(|F|\Delta^2\varepsilon^{-1})$. Moreover, the number of discovered vertices is also $O(|F|\Delta^2\varepsilon^{-1})$.

We maintain the family of components $\mathcal{COMP}$ in a find-union data structure with logarithmic amortized time per operation. In addition, for each component $C \in \mathcal{COMP}$ we keep a list of the vertices in $C \cap W$ to facilitate the checking of the condition in line 7 and the selecting of a waiting vertex $w$ from the chosen component $C$ in line 9. Since we do not need to perform any "find" operations on the sets $C \cap W$, $C \in \mathcal{COMP}$, but only the union operation (when two components are merged in line 19), adding a new vertex (line 15) and removing an arbitrary vertex (line 9), then a simple list implementation gives constant time per each operation.

We also need to store numbers $|E(C, F)|$ for $C \in \mathcal{COMP}$, and to update these numbers when two components merge, but this can be easily done in constant time per update and query. Note that the components $\mathcal{COMP}$ are pairwise disjoint, so the sets $E(C, F)$, for $C \in \mathcal{COMP}$ are also pairwise disjoint. We keep the list of the components $C$ for which $|C| \leq |E(C, F)|/\varepsilon$. This list may need to be updated when a component $C$ changes in line 15 and when components $C$ and $C'$ merge in line 19, with constant time per one update. This list enables the checking of the condition of the main loop and the selection of a component in constant time (lines 7 and 8). Observed that checking the condition whether $C \cap W \neq \emptyset$ for each component reduces to checking only whether $C \cap W \neq \emptyset$ for the component considered in the previous iteration. The other components which are still in $\mathcal{COMP}$ have not changed in any way during the previous iteration. In particular, the waiting vertices in these components remain the same as they were at the beginning of the previous iteration.

Summarizing, there are $O(|F|\Delta/\varepsilon)$ iterations of the main loop, and each iteration takes $O(\Delta)$ time plus the time of $O(\Delta)$ iterations of the inner loop. One iteration of the inner loop takes $O(\log(|F|\Delta\varepsilon^{-1}))$ amortized time, since it is dominated by a constant number of find-union operations on sets of size $O(\log(|F|\Delta^2\varepsilon^{-1}))$. Thus the main part of the algorithm takes $O\left(|F|\Delta^2\varepsilon^{-1}\log(|F|\Delta\varepsilon^{-1})\right)$ time. The initial part of the algorithm takes $O(|F|\Delta)$ time (discovering the edges adjacent to the vertices in $F$ and initializing the data structures). The final part of the algorithm takes constant time. Hence the running time of the whole algorithm is as given in the statement of the theorem.

We conclude the proof with a remark why we use a logarithmic-time find-union data structure, rather than a faster one with $O(\log^*)$ amortized time per operation. Normally a find-union data structure uses an array of pointers indexed by all elements in the domain. This array allows finding a given element in the current data structure in constant time. We cannot use such an array, since it would need $O(|V|)$ time and space. Instead of an array, we use a dictionary data structure to maintain the set of discovered vertices and the pointers to the locations of these vertices in the current find-union data structure. The number of discovered vertices is $O(|F|\Delta^2\varepsilon^{-1})$, so each find operation is preceded by an $O(\log(|F|\Delta\varepsilon^{-1}))$-time access operation on the dictionary data structure. This means that a find-union data structure with the amortized time per one operation better than logarithmic would not improve the bound on the total running time of the algorithm. □

## 4. Local connectedness

In this section we present an algorithm for testing the connectivity of $G - F$ for any graph $G \in \mathcal{G}(l)$. Since this approach is based on local properties of the graph $G$, first we need to study the class $\mathcal{G}(l)$ in detail.

Let $G \in \mathcal{G}(l)$ for some $l \geq 2$ and $F \subseteq V(G)$. An *$l$-ball* $B_l(v)$ centered at a vertex $v \in V(G)$ is the subgraph of $G$ induced by all the edges $xy \in E(G)$ with $d(v,x) + d(v,y) < l$. Note that if $l$ is odd (even), $B_l(v)$ is actually the subgraph induced by vertices (edges) at distance at most $\lceil l/2 \rceil - 1$ from $v$. In particular, $B_l(v)$ is connected for every $v \in V(G)$. An *$l$-ball graph* of $F$ is $B_l(F) = \bigcup_{v \in F} B_l(v)$. Note that $B_l(F)$ preserves all cycles from $G$ of length at most $l$ that contain some vertex of $F$. Moreover, $B_l(F)$ has at most $O(|F|\Delta(G)^{\lfloor l/2 \rfloor})$ vertices and $O(|F|\Delta(G)^{\lceil l/2 \rceil})$ edges.

Let $D$ be a component of $B_l(F) - F$, and let $W = (w_0, w_1, \ldots, w_n)$ be a walk in $G$. An edge between a vertex of $D$ and a vertex of $F$ is called a *$(D,F)$-edge*. Let $p(D,W)$ be the number of indices $i \in \{1, 2, \ldots, n\}$ such that $w_{i-1}w_i$ is a $(D,F)$-edge. Note that each $(D,F)$-edge with multiple occurrences on $W$ is counted in $p(D,W)$ as many times as it appears on $W$.

In the next four propositions we assume that $l \geq 2$, $G \in \mathcal{G}(l)$, $F \subseteq V(G)$, and $D$ is an arbitrary component of $B_l(F) - F$.

**Proposition 4.1.** *If $C, C_1, \ldots, C_k$ are cycles in $G$ such that*

$$E(C) = E(C_1) \,\Delta\, E(C_2) \,\Delta \ldots \Delta\, E(C_k),$$

*then the numbers $p(D,C)$ and $\sum_{i=1}^{k} p(D,C_i)$ are of the same parity.*

*Proof.* Observe that an edge $e$ of $G$ belongs to $E(C)$ iff the number $n(e) = |\{i \mid e \in E(C_i)\}|$ is odd. Consequently, denoting the set of all $(D,F)$-edges of $G$ by $E(D,F)$, we have

$$\left(\sum_{i=1}^{k} p(D,C_i)\right) \bmod 2 = \left(\sum_{e \in E(D,F)} n(e)\right) \bmod 2 = p(D,C) \bmod 2 \ .$$

□

**Proposition 4.2.** *For every $uv$-walk $W$ in $B_l(F)$, the numbers $p(D,W)$ and $|V(D) \cap \{u\}| + |V(D) \cap \{v\}|$ are of the same parity.*

*Proof.* We argue by induction on $p(D,W)$. If $p(D,W) = 0$, then $u \in D$ iff $v \in D$ and the statement follows. Otherwise it must be the case that $W = (W_1, (x,y), W_2)$ for suitable $ux$-walk $W_1$, $yv$-walk $W_2$ and $(D,F)$ edge $xy$. Assuming without loss of generality that $x \in V(D)$ and $y \in F$, the induction hypothesis implies that

- $p(D,W_1) \bmod 2 = (|V(D) \cap \{u\}| + 1) \bmod 2$,
- $p(D,(x,y)) = 1$,
- $p(D,W_2) \bmod 2 = |V(D) \cap \{v\}|$.

Since $p(D,W) = p(D,W_1) + p(D,W_2) + p(D,W_3)$, the conclusion follows. □

**Proposition 4.3.** *For every closed walk $W$ in $G$, $p(D,W)$ is even.*

*Proof.* First settle the case that $W$ is a cycle of length at most $l$. If the intersection of $W$ with $F$ is empty, then $p(D,W) = 0$. Otherwise $W$ must be fully contained in $B_l(F)$ and therefore $p(D,W)$ is even by Proposition 4.2.

Second, let $W$ be a cycle whose length exceeds $l$. Since $G \in \mathcal{G}(l)$, we have

$$E(C) = E(C_1) \,\Delta \ldots \Delta\, E(C_k)$$

where cycles on the righ-hand side are of length at most $l$. The statement now follows from the previous case and Proposition 4.1.

Otherwise it must be the case that $W = (W_1, C, W_2)$ where $C$ is a cycle and $(W_1, W_2)$ is a closed walk of shorter length than $W$. Since $p(D,W) = p(D,C) + p(D,(W_1,W_2))$, the statement follows by induction on the length of $W$. □

**Proposition 4.4.** *Let $W_1$ and $W_2$ be $uv$-walks in $G$. For every component $D$ of $B_l(F) - F$, the numbers $p(D, W_1)$ and $p(D, W_2)$ have the same parity.*

*Proof.* Since $W = (W_1, W_2^R)$ is a closed walk, $p(D, W)$ must be even by Proposition 4.3. As $p(D, W) = p(D, W_1) + p(D, W_2)$, both summands must be of the same parity and the statement follows. □

Now we have derived all the properties necessary to prove the theorem which serves as a cornerstone for the algorithm presented at the end of this section.

**Theorem 4.5.** *Let $G \in \mathcal{G}(l)$ for some $l \geq 2$ and $F \subseteq V(G)$. Then $G - F$ is connected if and only if $H - F$ is connected for every component $H$ of $B_l(F)$.*

*Proof.* Assume that $H - F$ is connected for every component $H$ of $B_l(F)$. Let $u, v \in V(G) \setminus F$ and $W$ be an arbitrary $uv$-walk in $G$ containing some vertices of $F$. Then $W$ contains a subwalk $S = (x, y_1, \ldots, y_n, z)$ such that $x, z \in V(G) \setminus F$ and $y_1, \ldots, y_n \in F$. Since $S$ entirely belongs to some component $H$ of $B_l(F)$ and $H - F$ is connected, there is an $xz$-walk $T$ in $H - F$. By replacing $S$ with $T$ in $W$, we obtain a $uv$-walk with less vertices from $F$. We can repeat this process until $W$ contains no vertex of $F$. Hence, the vertices $u$ and $v$ are connected in $G - F$.

On the other hand, assume that $H - F$ is disconnected for some component $H$ of $B_l(F)$. Let $u$ and $v$ be vertices from different components $C_u$ and $C_v$ of $H - F$, respectively. Since $H$ is connected, it contains a $uv$-walk $W_1$. Since $u$ and $v$ belong to different components of the disconnected graph $H - F$, Proposition 4.2 implies that the numbers $p(C_u, W_1)$ and $p(C_v, W_1)$ are odd. Hence by Proposition 4.4, every $uv$-walk $W_2$ in $G$ has $p(C_u, W_2)$ and $p(C_v, W_2)$ also odd, and consequently, it contains some vertex of $F$. Therefore, the vertices $u$ and $v$ are disconnected also in $G - F$. □

Armed with this result, we are ready to describe an algorithm for testing whether $G - F$ is a connected graph. Note that as described below, it includes a construction of an auxiliary subgraph $B_l(F)$ and testing its properties using standard graph algorithms. Since we already described a similar type technique in detail in Section 3, here we omit the pseudocode and concentrate on the merits of our algorithm instead.

**Theorem 4.6.** *There is an algorithm which, given a neighborhood oracle $n_G$ for a graph $G \in \mathcal{G}(l)$, $l \geq 2$, and a set $F \subseteq V(G)$, decides whether $G - F$ is connected in*

$$O\left(|F|\Delta^{\lceil l/2 \rceil} \log(|F|\Delta(G)^{\lfloor l/2 \rfloor})\right)$$

*time, where $\Delta = \Delta(G)$.*

*Proof.* First we need to construct an auxiliary graph $B_l(F)$. Note that for each $f \in F$, the construction of $B_l(f)$ requires $O(\Delta(G)^{\lceil l/2 \rceil - 1})$ calls to $n_G$. Each call to $n_G(v)$ takes $O(\deg_G(v)) = O(\Delta(G))$ time, and a generation of each neighbor of $v$ is followed by an access to dictionary $D$, which requires $O(\log |D|)$ time (cf. Section 2.2). Since $D$ contains no more than $|V(B_l(F))| = O(|F|\Delta(G)^{\lfloor l/2 \rfloor})$ vertices, it follows that the construction of $B_l(F)$ may be performed in $O(|F|\Delta(G)^{\lceil l/2 \rceil} \log(|F|\Delta(G)^{\lfloor l/2 \rfloor}))$ time.

Given $B_l(F)$, we find its components, test the connectedness of $H - F$ for each component $H$ of $B_l(F)$, and apply Theorem 4.5. Each of these steps can be performed in $O(|E(B_l(F))|) = O\left(|F|\Delta(G)^{\lceil l/2 \rceil}\right)$ time using standard algorithms. □

We already observed that the $n$-dimensional hypercube $Q_n$ satisfies $L(Q_n) = 4$. Since for an arbitrary set $F \subseteq V(Q_n)$ we have $\log(|F|\Delta(Q_n)^2) = O(\log |V(Q_n)|) = O(n)$, we obtain the following result, originally presented in [3].

**Corollary 4.7.** *There is an algorithm which decides whether $Q_n - F$ is connected in $O(|F|n^3)$ time.*

## 5. Local biconnectedness

In this section, we extend the method described in Section 4 to allow for testing the connectivity of a higher order. Recall that a connected graph on at least 3 vertices with no cutvertex is called biconnected. A *leaf* is a vertex of degree one. Note that if $v$ is a cutvertex of $G$ contained in a subgraph $H$ of $G$, then $v$ is a cutvertex, leaf or isolated vertex of $H$.

**Lemma 5.1.** *Let $G \in \mathcal{G}(l)$ for some $l \geq 2$. Then $G$ is biconnected if and only if $B_l(v) - v$ is connected for every $v \in V(G)$.*

*Proof.* Since $B_l(v)$ is connected, if $v$ is a cutvertex in $G$, it is also a cutvertex in $B_l(v)$. On the other hand, if $B_l(v) - v$ is disconnected for some $v \in V(G)$, then $\deg_G(v) \geq 2$ and $G$ has no cycle of length at most $l$ that contains $v$. Since $G \in \mathcal{G}(l)$, it follows that $G$ has no (larger) cycle containing $v$, and consequently, $v$ is a cutvertex of $G$. □

**Theorem 5.2.** *Let $G \in \mathcal{G}(l)$ for some $l \geq 2$ be biconnected and let $F \subseteq V(G)$ be such that $G - F$ is a connected graph with at least 3 vertices. Then $G - F$ is biconnected if and only if no vertex of $B_l(F) - F$ is a cutvertex of $B_l(F \cup C) - F$, where $C$ is the set of all cutvertices and leaves of $B_l(F) - F$.*

*Proof.* Assume that $G - F$ is not biconnected; that is, it contains some cutvertex $v$. Since $B_l(v) - v$ is connected by Lemma 5.1, the vertex $v$ is in $B_l(F) - F$; otherwise, $B(v)$ contains no vertex of $F$ and consequently, $v$ is not a cutvertex of $G - F$. Since $B_l(F) - F$ is a subgraph of $G - F$ and $v$ is a cutvertex of $G - F$, we deduce that $v$ is a cutvertex or leaf of $B_l(F) - F$; that is, $v \in C$. Hence $v$ is not a leaf of $B_l(F \cup C) - F$. Since $B_l(F \cup C) - F$ is a subgraph of $G - F$, it follows that $v$ is a cutvertex of $B_l(F \cup C) - F$ as well.

On the other hand, assume that some vertex $v$ of $B_l(F) - F$ is a cutvertex of $B_l(F \cup C) - F$. Then $v \in C$ and $v$ is also a cutvertex of $B_l(F \cup \{v\}) - F$. Let $H$ be the component of $B_l(F \cup \{v\})$ containing $v$. Since $H - (F \cup \{v\})$ is disconnected, it follows from Theorem 4.5 that $G - (F \cup \{v\})$ is disconnected as well. Therefore, $G - F$ is not biconnected. □

**Theorem 5.3.** *There is an algorithm which, given a neighborhood oracle $n_G$ for a biconnected graph $G \in \mathcal{G}(l)$, $l \geq 2$, and a set $F \subseteq V(G)$, decides whether $G - F$ is biconnected in*

$$O\left(|F|\Delta^l \log(|F|\Delta^l)\right)$$

*time, where $\Delta = \Delta(G)$.*

*Proof.* First, we test whether the graph $G - F$ is connected using the algorithm of Theorem 4.6. Then we apply a standard method to find the set $C$ of all cutvertices and leaves of $B_l(F) - F$ in linear time with respect to the number of edges of the tested graph. Both these steps take $O\left(|F|\Delta(G)^{\lceil l/2 \rceil}\right)$ time. Finally, we find all cutvertices of $B_l(F \cup C) - F$, and we apply Theorem 5.2. Since the graph $B_l(F \cup C)$ is a subgraph of $B_{2l}(F)$ it follows that the total running time is bounded by $O\left(|E(B_{2l}(F))| \log |V(B_{2l}(F))|\right) = O\left(|F|\Delta(G)^l \log(|F|\Delta(G)^l)\right)$. □

We conclude this paper with a brief suggestion of how to generalize the above ideas to decide whether the graph $G - F$ is $k$-connected.

Let $k > 2$ be a natural number. Assume that $G \in \mathcal{G}(l)$ is a graph such that $B_l(v) - v$ is $k$-connected for every vertex $v \in V(G)$. Analogously as in the proof of Theorem 5.2 we obtain that if $X \subseteq V(G) \setminus F$ is a minimal separator of $G - F$ with respect to inclusion of size at most $k$ then $X \subseteq V(B_l(F))$. Hence we obtain the analogous statement as Theorem 5.2: $G - F$ is $k$-connected if and only if no minimal separator $X$ of $B_l(F) - F$ of size at most $k$ is a separator of $B_l(F \cup C) - F$ where $C$ is the union of all leaves and all minimal separators of $B_l(F) - F$ of size at most $k$. Thus if $f(n)$ is a function such that there exists an algorithm which for a graph $G$ on $n$-element set constructs in $O(f)$ all minimal separators of size at most $k$, then there exists an algorithm which for a graph $G \in \mathcal{G}(l)$ and $F \subseteq V(G)$ such that $B_l(v) - v$ is $k$-connected for all vertices $v$ of $G$ decides whether $G - F$ is $k$-connected in time $O(f(|F|\Delta(G)^l))$.

There is an algorithm for finding $\kappa(G)$ that requires $O(\kappa(G)|V(G)|^{1.5}|E(G)|)$ time [4]. This algorithm is based on the algorithm for finding maximal flow. Thus for a fixed $k$ there exists a polynomial algorithm (of order $k$) constructing all minimal separators of size at most $k$.

## References


[1] B. Bollobás, *Modern Graph Theory*, Springer, 1998.

[2] D. M. Chickering, D. Geiger, D. Heckerman, *On finding a cycle basis with a shortest maximal cycle*, Inf. Process. Lett. **54** (1995), 55–58.

[3] T. Dvořák, J. Fink, P. Gregor, V. Koubek, T. Radzik, *Efficient connectivity testing of hypercubic networks with faults*, Lect. Notes Comput. Sci. **6460** (2011), 181–191.

[4] S. Even, R. E. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Comput. **4** (1975), 507–518.

[5] J. D. Horton, *A polynomial-time algorithm to find the shortest cycle basis of a graph*, SIAM J. Comput. **16** (1987), 358–366.

[6] W. Imrich, P. F. Stadler, *Minimum Cycle Bases of Product Graphs*, Australas. J. Combin. **26** (2002), 233–244.

[7] T. Kavitha, K. Mehlhorn, D. Michail, and K. E. Paluch, *A Faster Algorithm for Minimum Cycle Basis of Graphs*, Lect. Notes Comput. Sci. **3142** (2004), 846–857.

[8] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA 1992.