Lecture notes for Jan 30, 2023
Finding blocks

Chun-Hung Liu

February 5, 2023

Let G be a graph. A *cut-vertex* of G is a vertex v of G such that $G - v$ has more components than G. A *block* of G is a maximal connected subgraph H of G with no cut-vertex of H. Note that every block of G is an induced subgraph of G. Also note that every block of G either is an isolated vertex of G, or is a maximal 2-connected subgraph of G, or consists of an edge that is not contained in any 2-connected subgraph of G. (Recall that a graph is *2-connected* if it contains at least three vertices and cannot be made disconnected by deleting at most one vertex.) Usually problems on a graph can be reduced to its blocks by inductive arguments.

The *block structure* (or *block tree*) of a connected graph G is a tree T that has a bipartition $\{A, B\}$ such that there exist a bijection f from A to the set of cut-vertices of G and a bijection g from B to the set of blocks of G such that for every edge ab of T with $a \in A$ and $b \in B$, $f(a)$ is contained in $g(b)$. The existence of the block structure of a connected graph can be proved in various ways, and it can be constructed by using depth-first-search.

Now we show how to modify DFS to find all blocks and cut-vertices. Note that loops and parallel edges play no rule in the definition of blocks, cut-vertices and block structure. So we may assume that graphs are simple in this subsection for simplicity.

==========

An algorithm for finding all blocks
Input: A connected simple graph G with $|V(G)| \geq 2$.
Output: The set of all blocks of G.
Procedure:

Step 0: Pick a vertex r of $V(G)$.

Step 1: Set T to be the rooted tree $(\{r\}, \emptyset)$ rooted at r. Set all edges of G as "unmarked". Set S to be the sequence (r). Set $B_r = \{r\}$ and $\mathcal{B} = \emptyset$. Set $f(r) = 1$ and $g(r) = 1$.

(It is helpful to keep in mind that S and T will be constructed in the same say as they were constructed in DFS, and the following statements hold during the entire process:

(i) For every vertex v of G, once $f(v)$ is defined, the value $f(v)$ will never be changed; but $g(v)$ might be frequently updated.

(ii) f is a bijection from the current $V(T)$ to $[|V(T)|]$ indicating the ordering of the vertices of G added into T.

(iii) g is the function with domain $V(T)$ such that if v is the last entry of the current S, then either

- $g(v) = f(v)$ and no back edge for the current tree T incident with v is marked, or
- $g(v)$ equals the smallest i satisfying that there exists a back edge for the current tree T between a descendant of v in the current T and the vertex u with $f(u) = i$, and this back edge is marked.

(iv) For every vertex v of G, all vertices in the current B_v are descendants of v in the current tree T.

We will prove that the algorithm really preserves the above properties later.)

Step 2: Terminate the algorithm and output $\mathcal{B}$ if S has no entry. So we may assume that S is non-empty. Say the last entry of S is v. If there exists an unmarked edge of G incident with v, then do Step 2-1; otherwise, do Step 2-2.

Step 2-1 : Pick an unmarked edge e of G incident with v and mark e. Let u be the end of e other than v.

- If $u \notin V(T)$, then add e into T, add u into S as the last entry, set $B_u = \{u\}$, define $f(u)$ to be the minimum positive integer that is not in the image of f, define $g(u) = f(u)$, and then repeat Step 2.

* If $u \in V(T)$, then redefine $g(v)$ to be $\min\{g(v), f(u)\}$, and then repeat Step 2.

Step 2-2:
* If $v = r$, then remove v from S and repeat Step 2.
* Otherwise, let p be the parent of v in T.
  - If $g(v) = f(v)$, then add $\{v, p\}$ into $\mathcal{B}$, and then remove v from S and repeat Step 2.
  - If $g(v) = f(p)$, then add $B_v \cup \{p\}$ into $\mathcal{B}$, and then remove v from S and repeat Step 2.
  - If $g(v) < f(p)$, then set $B_p = B_p \cup B_v$, redefine $g(p)$ to be $\min\{g(p), g(v)\}$, and then remove v from S and repeat Step 2.
    (Note that $f(v) \neq g(v) > f(p)$ is possible due to (ii), (iii) and the known fact that there is no crossing edge for a DFS tree T.)

==========

Now we prove the correctness of this algorithm. Clearly, properties (i), (ii) and (iv) hold during the entire process.

Lemma 1 *Property (iii) holds during the entire process.*

Proof. Let v be the currently last entry in S. If v is just added into S, then $g(v) = f(v)$, and no back edge incident with v is marked. In particular, (iii) holds at the end of Step 1 and holds at the round of executing Step 2 when v is just added into S. So now we may assume that we are at the round of executing Step 2 when v is already in S in the previous round.

Assume that Step 2-1 is executed in this round. Let u be the corresponding vertex mentioned in Step 2-1. If $u \notin V(T)$, then u becomes the last entry and property (iii) holds. So we may assume $u \in V(T)$. Then the edge $e = uv$ is a back edge between v and u, and this edge becomes marked at this moment. Note that v is a descendant of v, so uv is an edge between a descendant of v and u. So if $f(u) < g(v)$, then $g(v)$ is redefined to be $f(u)$, as desired; if $f(u) \geq g(v)$, then $g(v)$ remains unchanged, as desired.

Hence we may assume that Step 2-2 is executed in this round. If $v = r$, then S becomes empty and the algorithm stops. So we may assume $v \neq r$. Hence the parent p of v exists. Recall that the entries of S form a path in T from r to v. So once v is removed, p becomes the last entry of S. Since

property (iii) is preserved so far, we know either $g(v) = f(v)$, or $g(v)$ is the minimum i such that some marked back edge is between a descendant of v and the vertex z with $f(z) = i$. For the former or the latter with $g(v) = f(p)$, the current property (iii) preserved so far implies that there is no marked back edge between a descendant of v and a proper ancestor of p, so there is no need to update $g(p)$. For the latter with $g(v) \neq f(p)$, we know that $g(v)$ is defined by a marked back edge between a descendant of v (and hence a descendant of p) and a proper ancestor of p, so $g(p)$ is updated to be $\min\{g(p), g(v)\}$, as desired. (Note that the contribution of the back edge between a descendant of a child c of p other than v and a proper ancestor of p has been attributed to $g(p)$ when c was removed from S and p became the last entry of S in the past.) This proves the lemma. ∎

Lemma 2 *Let v be a vertex of G. Let B be a block of G containing a proper ancestor of v in the final T and a proper descendant of v in the final T. Then the final B_v contains $\{x \in V(B) : x$ is a descendant of $v\}$ and is contained in B.*

Proof. We prove this lemma by induction on $|V(T)| - f(v)$. This lemma holds when v has no children. In particular, the lemma holds when $|V(T)| - f(v) = 0$. So the induction base holds, and we may assume that the lemma holds when $|V(T)| - f(v)$ is smaller.

Note that every proper descendant of v is a descendant of some child of v. Let c be an arbitrary child of v such that B contains a descendant of c. Since B contains a proper ancestor of v and a descendant of c, $B - v$ contains a path between a proper ancestor of v and a descendant of c. Since there exists no crossing edge, there exists a back edge e_c in B between a descendant of c and a proper ancestor of v contained in B. Since property (iii) is preserved, we know $g(c) < f(v)$ when c is about to be removed from S. So the final B_v contains the final B_c according to the algorithm. Since $f(c) > f(v)$, we know $|V(T)| - f(c) < |V(T)| - f(v)$. So by the induction hypothesis, if B contains a proper descendant of c, then the final B_c contains $\{x \in V(B) : x$ is a descendant of $c\}$ and is contained in B; if B does not contain a proper descendant of c, then c is the unique vertex in $\{x \in V(B) : x$ is a descendant of $c\}$, so the final B_c contains $\{c\} = \{x \in V(B) : x$ is a descendant of $c\}$ and is contained in B. Hence the final $B_v \supseteq B_c$ contains $\{x \in V(B) : x$ is a descendant of $c\}$, and $\{x \in B_v : x$ is a descendant of $c\} \subseteq V(B)$.

Since c is an arbitrary child of v such that B contains a descendant of c, we have the final $B_v \supseteq \{x \in V(B) : x$ is a proper descendant of $v\}$. Since

$v \in B_v$ by definition, the final B_v contains $\{x \in V(B) : x$ is a descendant of $v\}$.

Recall there exists a back edge e_c in B between a descendant of c and a proper ancestor of v contained in B. Since e_c and the tree path connecting the ends of e_c forms a cycle containing $e_c \in E(B)$ and the edge pv, where p is the parent of v, we know that this cycle is in B. In particular, $v \in V(B)$.

Suppose that the final B_v is not contained in B. Since $v \in V(B)$ and property (iv) is preserved, there exists a child c' of v such that B does not contain any descendant of c' but $B_{c'}$ is merged into B_v when c' is about to be removed from S. It implies that $g(c') < f(v)$, so property (iii) implies that there exists a back edge from a descendant of c' to a proper ancestor of v. Hence there exists a cycle C containing pvc' and sharing pv with B, so C is contained in B, contradicting that B does not contain a descendant of c'. So the final B_v is contained in B. ∎

Lemma 3 *The vertex-set of every block of G is a member of $\mathcal{B}$, and every member of $\mathcal{B}$ is the vertex-set of a block of G.*

Proof. We first prove that the vertex-set of every block of G is a member of $\mathcal{B}$. Let B be a block of G. Since $|V(G)| \geq 2$ and G is connected, $|V(B)| \geq 2$. Let v^* be the vertex with $f(v^*) = \min_{x \in V(B)} f(x)$. Since $|V(B)| \geq 2$ and there exists no crossing edge, there exists a unique child c of v^* such that B contains a descendant of c. If B contains a proper descendant of c, then Lemma 2 implies that the final B_c contains $\{x \in V(B) : x$ is a descendant of $c\} = V(B) - \{v^*\}$ and is contained in B, and property (iii) implies that $g(c) = f(v^*)$ when c is about to be removed from S, so $V(B) = B_c \cup \{v^*\}$ is added into $\mathcal{B}$ when c is about to be removed from S. If B does not contain a proper descendant of c, then $V(B) = \{v^*, c\}$, so $g(c) = f(c)$ when c is about to be removed from S, and hence $V(B) = \{v^*, c\}$ is added into $\mathcal{B}$ when c is about to be removed from S. This proves that the vertex-set of any block of G is a member of $\mathcal{B}$.

Now we prove that every member of $\mathcal{B}$ is the vertex-set of a block of G. Let M be a member of $\mathcal{B}$. Let v be the vertex of G such that M is added into $\mathcal{B}$ when v is about to be removed from S. By the algorithm, M contains v and its parent p. Note that there exists a unique block B_M containing vp. So $\{v, p\} \subseteq M \cap V(B_M)$. Since B_M is a block of G, $V(B_M)$ is a member of $\mathcal{B}$. Let y be the vertex of G such that $V(B_M)$ is added into $\mathcal{B}$ when y is about to be removed from S. If $y = v$, then $M = V(B_M)$ and we are done. So we

may assume $y \neq v$. Since property (iv) is preserved, every vertex in B_M is either a descendant of y or the parent of y. Since $p \in M \cap V(B_M) \subseteq V(B_M)$, p is a descendant of the parent of y, so v is a descendant of y. Since $y \neq v$, v is a proper descendant of y. Since property (iv) is preserved, if the final B_p contains v, then B_p has to be updated to be $B_p \cup B_v$ when v is about to be removed from S. But M is added into $\mathcal{B}$ when v is about to be removed, so B_p is not updated at that moment. So v is not in the final B_p. This implies that v is not in the final B_a for every proper ancestor a of v. In particular, v is not in the final B_y. So $v \notin V(B_M)$, a contradiction. ■

Theorem 4 *The blocks of the input graph G can be found in linear time.*

Proof. By Lemma 3, the vertex-sets of all blocks are output by our algorithm. So we can find all blocks by simply taking the subgraphs induced by those sets. And it is easy to see that this algorithm runs in linear time. ■

We remark that the cut-vertices of G are exactly the vertices of G contained in at least two blocks. So we can find all cut-vertices of G in linear time after we know all the blocks. In fact, it is not hard to see that z is a cut-vertex of G if and only if either $z = r$ and r has at least two children, or z equals the vertex p whenever we add a member into $\mathcal{B}$ during the algorithm. Moreover, since we know all the cut-vertices and blocks, we can construct the block structure of G in linear time.