Fast Analysis of Upstream Features on Spatial Networks (GIS Cup)

Salles Viana Gomes Magalhães
Universidade Federal de Viçosa
Viçosa, MG, Brazil
salles@ufv.br

W. Randolph Franklin
Rensselaer Polytechnic Institute
Troy, NY
mail@wrfranklin.org

Ricardo dos Santos Ferreira
Universidade Federal de Viçosa
Viçosa, MG, Brazil
ricardo@ufv.br

ABSTRACT

We present a fast linear time algorithm that uses a block-cut tree for identifying upstream features from a set of starting points in a network. Our implementation has been parallelized and it can process a dataset with 32 million features in less than 8 seconds on a 8-core workstation. This problem is the 2018 ACM SIGSPATIAL CUP challenge and presents several applications mainly on the field of utility networks. Our code is freely available for nonprofit research and education at https://github.com/sallesviana/FastUpstream

CCS CONCEPTS

• **Mathematics of computing** → *Graph algorithms*;

KEYWORDS

Graph algorithms, GISCUP, parallel programming, spatial networks

ACM Reference Format:
Salles Viana Gomes Magalhães, W. Randolph Franklin, and Ricardo dos Santos Ferreira. 2018. Fast Analysis of Upstream Features on Spatial Networks (GIS Cup). In *26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18), November 6–9, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3274895.3276474

1 INTRODUCTION

The recent increase of the availability of big datasets has motivated the creation of a variety of applications to analyze this information. While the quality of the analysis typically improves as the amount of available information increases, the computational cost of processing this data becomes a challenge.

An example of datasets are the spatial networks, which can be employed to represent utility networks [1]. Analyzing these datasets is useful, for example, to determine the reliability of the network (if all the electricity of a region passes through a single transformer, any problem in this transformer may create a power outage), to analyze how the resources flow, etc.

The GISCUP [3] is a yearly contest that encourages the innovation and the development of efficient algorithms for the field of Geographical Information Science. The objective is typically to perform efficient analysis on big datasets. In 2018, the challenge is the creation of an algorithm for analyzing the flow of resources in utility networks. This paper describes our solution to this problem.

SIGSPATIAL '18, November 6–9, 2018, Seattle, WA, USA

ACM ISBN 978-1-4503-5889-7/18/11…$15.00
https://doi.org/10.1145/3274895.3276474

1.1 The problem

The utility network is modeled as an graph G where the edges (always bidirectional) represent the medium (e.g. pipes, wires, etc.) where resources (e.g. water, gas, electricity, etc.) flow and vertices represent connections. Also, G contains some special vertices called controllers (like power plants or water stations) that are the source of the resources.

Given a utility network and a set of vertices and edges (the *starting points*), the upstream identification problem consists in finding all the features (vertices and edges) that are in a simple path between a controller and a starting point (in the rest of this paper we call these output features simply *upstream features*). An example of application is to find devices (cables, transformers, etc) that are between power stations (controllers) and important electricity consumers (starting points) in an electricity network. This analysis could be employed to find features that are critical for providing a reliable source of electricity to hospitals, factories, etc.

Thus, the input of the problem is a graph G composed of a set of vertices V and edges E, a set of controllers C (which is a subset of V) and a set of starting points S (which is a subset of the edges and vertices). In the rest of this paper we call vertices/edges that are either controllers or starting points *important* vertices/edges.

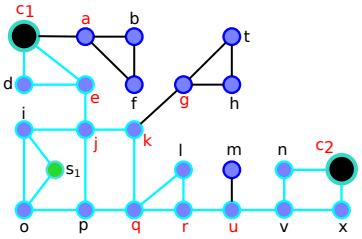
Figure 1a presents an example of utility network (represented by the graph G_1) and detaches (on light blue) all the vertices and edges on a simple path between controllers (vertices on black) and starting points (the vertex on green).

1.2 Block-cut trees

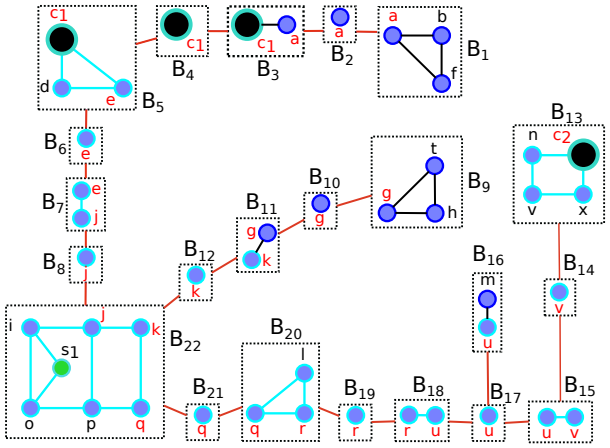
The key technique employed in our algorithm is to use the block-cut tree [2] of the network to find the upstream features. Given a graph G, the block-cut tree of G (shortened to BC-tree, or $BC(G)$) is a tree where the vertices are the blocks (biconnected components) and articulations of G. There is an edge in BC between each vertex representing a block and the vertices representing its articulations.

The leaves of a block-cut tree $BC(G)$ are associated to biconnected components of the original graph and any path in $BC(G)$ alternates vertices associated to biconnected components with vertices associated to articulations.

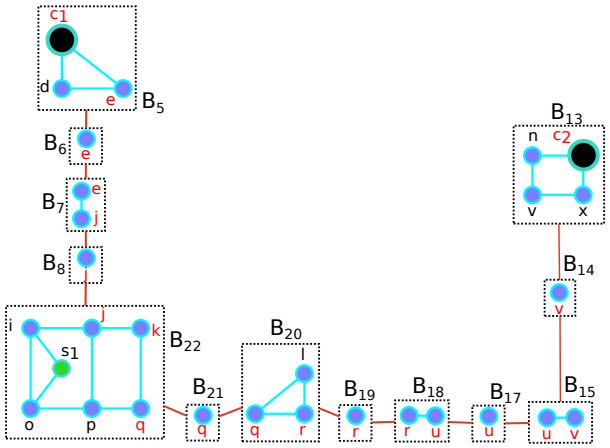
Figure 1b presents the block-cut tree BC_1 of G_1. Each vertex (represented by square rectangles) i of BC_1 is labeled with B_i and the graph features (vertices and edges) that generated this vertex are drawn inside the corresponding rectangle (in the rest of this paper we consider these features to be *inside* the BC-tree vertex) .



(a) Graph G_1 of the network



(b) Block-cut tree BC_1 of G_1



(c) After iteratively removing non-important leaves from (b)

Figure 1: Example of utility network. The vertex s_1 (on green) represents a starting point, and c_1 and c_2 (on black) are the controllers. Edges and vertices detached on light blue are upstream from the starting point.

2 THE SOLUTION

2.1 Analyzing the network

We claim that the upstream features can be identified without explicitly enumerating all the paths between starting points and controllers (this is typically unfeasible because the number of such paths could be exponential on the size of the graph).

For simplicity, let us assume the network is connected (otherwise the problem can be solved independently for each connected component), that it contains both controllers and starting points vertices (otherwise the output should be empty), its vertices do not contain loops and that a vertex cannot be a controller and a starting point simultaneously. Also, assume starting points can only be vertices. The special cases where some of these assumptions do not hold will be considered in Section 2.2.

Because of space limitations, we will present the proofs related to the process of finding upstream vertex features (the proofs for edge features are similar).

Claim 1. *Let BG be a biconnected graph. Any vertex of BG is on a simple path between two vertices u, v of BG.*

Proof: Since BG is connected, there is at least one simple path between u and v. Also, given any vertex w ($w \neq u, v$) in BG there are at least two paths Pu (between w and u) and Pv (between w and v) such that Pu and Pv do not share a vertex (other than w) and, thus, w is in a u-v path. If all paths between w and u and between w and v shared a vertex x (s.t. x!=w), then removing x would disconnect BG and, thus, it could not be biconnected. ■

Claim 2. *Let A and C be two vertices of a a block-cut tree $BC(G)$, where both A and C contain important vertices and at least one of these important vertices is a controller and another one is a starting point. If vertex B of $BC(G)$ ($B \neq A, C$) is in the unique simple path connecting A to C, then any vertex of G in B is an upstream feature.*

Proof: This is a result of Claim 1. For example, in Figure 1b vertex B_{20} is in the unique path between B_{13} (containing a controller) and B_{22} (containing a starting point). Thus, any vertex in B_{20} is an upstream feature. ■

Claim 3. *Let A and C be the two endpoints of a path P in a block-cut tree $BC(G)$, where A and C contain important vertices and at least one of these important vertices is a controller and another one is a starting point. If both A and C contain at least one important vertex which is not an articulation in P, then all features of G in vertices of P are upstream features.*

Proof: Because of Claim 2, we only have to prove that the vertices in A and in C are upstream features. A (the proof for B is identical) has at least one important vertex v that is not the articulation a, which is also present in the neighbor of A in the path. Assume v is a controller (if v is a starting point the proof is similar). If another vertex in A is a starting point, because of Claim 1 any vertex in A will be upstream. Otherwise, a vertex b of B is a starting point and, since there is a path between b and a and any vertex in A is in a simple path between v and a, then all vertices in A are upstream.

For example, any vertex of G_1 in path $B_{22}B_8B_7B_6B_5$ in Figure 1b is an upstream vertex. On the other hand, vertex a in path $B_3B_4B_5$ is not upstream (the only important vertex in B_3 is an articulation represented by vertex B_4 in the path) ■

Claim 4. *Let T be a connected subgraph of a block-cut tree $BC(G)$. Assume T contains both controllers and starting points, that all leaves of T represent biconnected components and that each leaf of T contains at least one important vertex that is not an articulation still in a vertex of T. Then, all vertices of T are in a simple path between a vertex containing a controller and one containing a starting point.*

Proof: If all leaves contain only controllers (resp. starting points), then at least one internal vertex V contain a starting point (resp. controller). Since T is a tree, any vertex will be in a path between V (containing a starting point) and a leaf (containing a controller).

If some leaves contain controllers and others contain starting points, any vertex of T will be in a simple path between a leaf containing a controller and a leaf containing a starting point.

Because of Claim 3, any vertex of G in T will be upstream. For example, all vertices of G_1 (Figure 1a) in Figure 1c are upstream. ■

Claim 5. *Let L be a leaf of a block-cut tree $BC(G)$, $V(L)$ be the set of vertices in L and* a *be the articulation whose associated vertex A of $BC(G)$ is adjacent to L. If no vertex of $V(L)-$*a *is an important vertex, then, except possibly for* a*, no vertex in L is upstream.*

Proof: Suppose a vertex $v \neq a$ in L was on a simple path p between two important vertices. Since the only vertex of L that may be important is a, then p would contain a twice and, therefore, p could not be a simple path. For example, in Figure 1b B_1 does not have any important vertex and, thus, no simple path connecting two important vertices could contain a vertex (other than the articulation a) in B_1. ■

Claim 6. *Let L be a leaf of a connected subgraph T of a block-cut tree $BC(G)$, $V(L)$ be the set of vertices in L and* a *be the articulation whose associated vertex in T is adjacent to L. If no vertex of $V(L)-$*a *is an important vertex, then, the upstream vertices in T are also in $T - L$.*

Proof: According to Claim 5, $V(L) - a$ does not contain upstream vertices. Also, even though a may be an upstream vertex L may be removed because a is an articulation and, thus, a neighbor of L in T contains a copy of a. If T has upstream vertices it is guaranteed that L has a neighbor (otherwise T would not have both kinds of important vertices).

For example, consider Figure 1b. $BC_1 - B_1$, $BC_1 - B_{16}$ and BC contain all the upstream vertices of the original graph. Observe that, even though B_{16} has an upstream feature (u), there is a copy of this feature in other vertices (since u is an articulation). ■

These claims suggest an algorithm (Algorithm 1) to compute the upstream features in a graph. Observe that after the loop on lines 2 and 5, according to Claim 4 all returned features are upstream from the starting points. Also, because of Claim 6, the loops on lines 2 and 5 do not remove any feature that should be in the output.

Figure 1c illustrates the state of the block-cut tree BC_1 (Figure 1b) after some leaves are removed by the loops on lines 2 and 5.

The block-cut tree can be created in linear time on the size of the input (using Tarjan's algorithm [4] to find the biconnected components). Also, pruning the tree is linear (on the size of the tree, which in the worst case is similar to the size of the input graph) and, thus, in the worst case the cost of the pruning step is similar to the cost of the creation of the tree. Therefore, the overall time complexity of the algorithm is linear on the size of the graph.

2.2 Special cases

Instead of separately treating the special cases where edges are starting points, we decided to reduce a dataset with this special case to one without it. This is accomplished by replacing each starting point edge $e = (u, v)$ with two artificial edges $e_1 = (u, s)$ and $e_2 = (s, v)$, where s is an "artificial" starting point vertex. This process is illustrated in Figure 2. Then, each occurrence of the artificial edges or vertices in the output obtained from the modified dataset is replaced with the corresponding original starting point edges. It is clear that the output of this modified input is the same as the expected solution for the original dataset.

Algorithm 1 Computes the upstream features in a graph G

1: $BC \leftarrow$ Create the block-cut tree of G
2: **while** BC has a leaf L without important vertices **do**
3: Remove L from BC
4: **end while**
5: **while** BC has a leaf L where L has exactly one important vertex v and v is an articulation still in BC **do**
6: Remove L from BC
7: **end while**
8: return the vertices and edges of G inside the vertices of BC

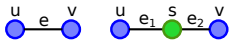


Figure 2: Using artificial edges and vertex (right) to represent a starting point edge e (left).

Another special case arises when a vertex has a loop. Loops cannot be in simple paths and, thus, they are ignored by the algorithm. However, in the GISCUP a path may start at an edge and, thus, a starting point edge that is a loop can be in a path to a controller (since the path would start on the edge and, thus, won't include the loop vertex twice). This does not have to be explicitly handled by the algorithm since the creation of the artificial edges/vertices (as described above) replaces the loop with a cycle and, thus, the algorithm behaves as desired. This process is illustrated in Figure 3.

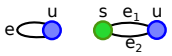


Figure 3: Loop starting point e being replaced with two artificial edges and a starting point s.

In the GISCUP, vertices were allowed to be simultaneously controllers and starting points. We claim (without proof) that if a connected component contains at least two different important vertices and at least one of the vertices is simultaneously a starting point and a controller, then no special treatment is necessary. The only special case that needs to be handle separately happens when the connected component has only one important vertex v which is simultaneously a controller and a starting point. In this situation the only output feature generated from this component will be v.

2.3 Implementation details

The algorithm described in this paper has been implemented in C++, and several optimization techniques were employed to improve its performance. The performance advantage of each design choice was carefully evaluated during the implementation process.

First, in the GISCUP contest the global identifier of each feature was always represented using a 32-digit string (where each digit is in hexadecimal) which can be encoded using two 64-bit integers. By employing this kind of representation we were able to not only

Table 1: Running times (in seconds) of key steps of the algorithm considering datasets with varying sizes. Columns BC-tree, Remove leaves, Select output and Total represent, respectively, the times to create the block-cut tree, iteratively remove leaves without upstream features, select the upstream features for the output and the total time (excluding the parsing and I/O).

Dataset	Number of					Time (s)					
	Vertices	Edges	Biconnected Components	Articulations	Output features	Graph creation	Tarjan	BC-tree	Remove leaves	Select output	Total
EsriNaperville	8465	8302	7859	4677	34	0.001	0.000	0.001	0.000	0.000	0.002
Graph1	1M	16M	1	0	17000000	0.780	0.565	0.013	0.000	0.292	1.650
Graph2	3M	3.3M	1866755	1228925	2566493	0.221	0.599	0.209	0.062	0.057	1.148
Graph3	3M	5M	435857	395566	7128288	0.303	0.769	0.105	0.011	0.091	1.279
Graph4	5M	8M	843737	753966	11312528	1.227	1.603	0.276	0.023	0.193	3.322
Graph5	13M	16M	5414204	4096497	18171594	0.939	3.047	0.842	0.163	0.449	5.440
Graph6	16M	16M	15999979	7998574	39	1.020	2.937	1.216	0.536	0.117	5.826
Graph6_100	16M	16M	15999979	7998574	1142	1.028	2.939	1.445	0.533	0.112	6.057
Graph6_1M	16M	16M	16500369	8498964	7336103	1.064	3.055	2.658	0.444	0.753	7.974

reduce the memory footprint of the algorithm, but also to accelerate operations like comparing the ids of two features.

Also, the articulation points and biconnected components were computed using Tarjan's algorithm [4]. However, instead of reusing existing implementations of Tarjan's algorithm (such as the one provided by the Boost Graph Library), we decided to reimplement it for performance purposes. Since Tarjan's algorithm has to perform a DFS traversal in the entire graph, our custom implementation saves time by using this traversal to also identify connected components (*CCs*), ignoring *CCs* not containing controllers or starting points, determining whether or not a biconnected component has a controller (and/or a starting point), etc.

Furthermore, a custom parser has been created to read the JSON files representing the network. However, all the modules of our implementation have been designed to be loosely coupled and, thus, in situations where code reusability and maintainability is more important than performance, one can easily adapt the implementation to reuse a regular JSON parser instead of the custom one.

Finally, the implementation has been profiled and the bottlenecks were parallelized using OpenMP.

3 EXPERIMENTAL RESULTS

The algorithm has been evaluated on a workstation with a 8-core Intel Xeon CPU E5-2630 v3, 64 GB of RAM, and Linux operating system. Experiments have been performed on the sample datasets provided by the GISCUP organizers [3], and we always used 8 threads. Also, we created a variety of random connected graphs with different number of edges, vertices, starting points and controllers.

Table 1 presents the running times for some of the aforementioned datasets. Dataset *EsriNap.* is the sample network "Esri's Naperville Electric Network Dataset" provided by the GISCUP organizers while the other 8 ones were randomly created.

Graph6_100 and Graph6_1M contain, respectively, 100 and 10^6 controllers and starting points. Their network is equal to Graph6's. The other inputs contain one controller and one starting point.

Observe that, in general, the most time-consuming step of the algorithm is the computation of the biconnected components (Tarjan's algorithm). Since this step performs a DFS on the graph (which is hard to parallelize efficiently because of data dependency) it was not implemented in parallel. The creation of the graph and of the block-cut tree, on the other hand, were performed in parallel (for example, considering Graph6_1M these steps took, respectively, 5 and 2 times longer to run sequentially).

As expected, there is little variation in the time for graphs with similar sizes. The variation on the time for graphs with different amounts of starting point and controllers is also small: Graph6_1M takes 37% longer to be processed than Graph6.

Besides performing experiments to evaluate the running time of the algorithm, we also manually created a variety of small graphs to evaluate the correctness of the algorithm. Special cases evaluated include disconnected graphs, graphs with loops, multiple edges, graphs where the important vertices are articulations, graphs where the starting point and controllers were the same vertices, etc.

4 CONCLUSIONS

We presented a fast algorithm for finding features in a network that are on a simple path between starting points and controllers. This algorithm presents a running time that is linear on the number of vertices and edges (its asymptotic running time does not depend on the number of starting points or controllers). Its efficiency makes it suitable, for example, for interactive applications where the user wants to quickly visualize the result of a modification on the network topology.

This research was partially supported by FAPEMIG, CAPES and CNPq.

REFERENCES

[1] Petko Bakalov, Erik G. Hoel, and Sangho Kim. 2017. A Network Model for the Utility Domain. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'17)*. ACM, New York, NY, USA, Article 32, 10 pages.
[2] Frank Harary. 1969. *Graph Theory.* Addison-Wesley, Reading, MA.
[3] Dev Oliver, Bo Xu, and Yuanyuan Pao. 2018. GISCUP - ACM SIGSPATIAL CUP 2018. (2018). http://sigspatial2018.sigspatial.org/giscup2018 (accessed Aug–2018).
[4] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.