Wüpstream: Efficient Enumeration of Upstream Features (GIS Cup)

Thomas C. van Dijk
Lehrstuhl für Informatik I
Würzburg University
thomas.van.dijk@uni-wuerzburg.de

Tobias Greiner
Lehrstuhl für Informatik I
Würzburg University
tobias.greiner@uni-wuerzburg.de

Bas den Heijer
ORTEC
Zoetermeer
bas.denheijer@ortec.com

Nadja Henning
Lehrstuhl für Informatik I
Würzburg University
nadja.henning@stud-mail.uni-wuerzburg.de

Felix Klesen
Lehrstuhl für Informatik I
Würzburg University
felix.klesen@stud-mail.uni-wuerzburg.de

Andre Löffler
Lehrstuhl für Informatik I
Würzburg University
andre.loeffler@uni-wuerzburg.de

ABSTRACT

This short paper describes Wüpstream, an efficient code for enumerating upstream features in undirected graphs. It uses a linear-time algorithm based on block-cut trees. We describe this algorithm and discuss some performance considerations in the C++ implementation. Code is available at: https://github.com/tcvdijk/wupstream.

CCS CONCEPTS

• **Information systems → Geographic information systems**;
• **Theory of computation → Graph algorithms analysis**;

KEYWORDS

graph algorithms, network analysis, software engineering

ACM Reference Format:
Thomas C. van Dijk, Tobias Greiner, Bas den Heijer, Nadja Henning, Felix Klesen, and Andre Löffler. 2018. Wüpstream: Efficient Enumeration of Upstream Features (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/10.1145/3274895.3276475

1 INTRODUCTION

This short paper describes Wüpstream, our submission to the 2018 ACM SIGSPATIAL GIS Cup. The competition concerns a problem on spatial utility networks (e.g. water or electricity): reporting all features of the network that lie on a simple path between any *starting points* and any *controller* features indicated in the data. Such features are called *upstream*.

Considering the geospatial nature of such networks, we are provided with various kinds of domain-appropriate data such as the geometry of the network connections. However, properties such as crossing line features are not relevant in the problem statement and there is no guaranteed geometric structure such as planarity; the problem only involves the (abstract) network topology.

In Section 2 we specify a formal Upstream Enumeration problem. To the best of our knowledge, the problem as described on the contest website (see http://sigspatial2018.sigspatial.org/giscup2018) and clarified on the official contest forum[1] is correctly captured by this problem, after the following transformation. Every `row` in the input file represents a single edge of an undirected graph, where the labels of the vertices are given by `fromId` and `toId`, and vertices with the same label are identified. The label of the edge is given by `viaId`. No other data from the `rows` array is used.

Section 3 describes a linear-time algorithm for Upstream Enumeration. It is based on an (augmented) *block-cut tree* [4]. Section 4 describes the implementation of this algorithm in Wüpstream. We conclude with some suggestions for future work.

2 PROBLEM STATEMENT

Our algorithm solves the following problem, which is slightly generalised from the contest problem. Let $G = (V, E)$ be a graph and $id\colon V \cup E \to L$, where L is some set of *labels*. When notationally convenient, we use a label to stand for the things it labels. Finally, let $A, B \subseteq L$.

For any $a \in A, b \in B$, and any $x \in V \cup E$ on a simple path from a to b, the label $id(x)$ is called an *upstream feature*. (This "path" can start and end at an edge.) The Upstream Enumeration problem is then: enumerate all upstream features.

3 ALGORITHM

3.1 Upstream Features in Trees

First we consider the Upstream Enumeration problem in trees with $A, B \subseteq V$. The advantage of trees is that the path between any two vertices is uniquely defined, which simplifies the concept of upstream features. The case that A or B contains edges can be handled by a simple extension. The algorithm proceeds in two passes through the tree.

MarkTowardA: Step one marks both directions of each edge with the following information: does any element with a label in

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

ACM ISBN 978-1-4503-5889-7/18/11.
https://doi.org/10.1145/3274895.3276475

[1] https://groups.google.com/forum/#!forum/giscup-2018

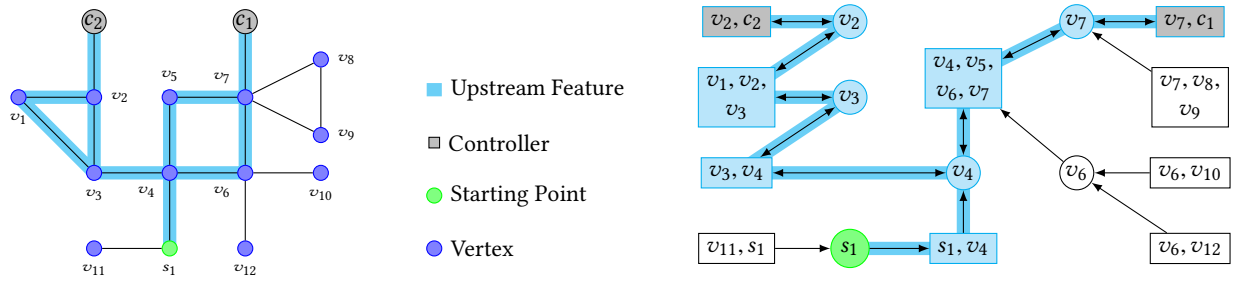

Figure 1: Left: solution to an example instance from the contest website. Right: the corresponding block-cut tree (omitting edge labels to save space). Rectangles are block nodes and discs are cut nodes; arrows on the edges mark toward the controllers.

A occur in that direction? Initialise this to false everywhere and then, for every vertex $a \in A$, start a depth-first traversal and mark the backward direction on the edges: since we just came from that direction, there is a label of A in that direction. Do not traverse edges whose backward direction is already marked: going into that subtree would only repeat what an earlier traversal (from a different a) has already marked. For linear runtime, remove traversed *directions* of edges from the adjacency lists so that every edge is traversed at most twice.[2] We call this the mark step.

FloodFromB: It would be possible to do this traversal also for B. Then a vertex is "upstream" if and only if:

- it is in B and has an edge with outgoing mark "toward A", or
- it is in A and has an edge with outgoing mark "toward B", or
- it has an edge with outgoing mark "toward A" and a different edge with outgoing mark "toward B."

Upstream edges can be recognised similarly. A third pass through the tree could check these conditions and output the label of each such element. Instead, we note that these conditions can be checked locally *during* the "mark toward B" traversals without actually marking the tree; the upstream features are emitted immediately. We call this the flood step.

Note that in a (connected) tree, the cardinality of A and B is largely inconsequential for the runtime since later traversals must terminate quickly. The runtime may be influenced in a forest, where components with neither A nor B get ignored.

3.2 The Block-Cut Tree (BC-Tree)

Upstream features in general graphs are highly related to the concepts of *blocks* (biconnected components) and *cut vertices* [4]. Recall that a graph is biconnected if it is connected and remains so after the removal of any single vertex; a cut vertex is one whose removal results in more connected components than before. A block is a maximal set of vertices that induces a biconnected graph. We use the linear-time algorithm by Hopcroft & Tarjan [6] to enumerate these blocks and cut vertices; meanwhile, we incrementally construct an undirected tree of *block nodes* and *cut nodes* – every block node is connected to the cut node for each cut vertex it contains. This is the *block-cut tree (BC-tree)*; see Figure 1 for an example. We say a block node *contains* all the vertices and edges of its block; a cut node contains the cut vertex and no edges. Notice that a cut vertex is contained in multiple block nodes.

For any simple path going through a block, there exists a detour using any other vertex in that block – and therefore if a block node is upstream (in the sense of Section 3.3), then all vertices and edges in that block are upstream as well.

Lemma 3.1. *Let $G = (V, E)$ be a biconnected graph and $x, y, z \in V$ be three distinct vertices. There exists a simple path from x to z via y.*

Proof (sketch). Replace every edge by two arcs (one in each direction) and consider the maximum flow from sources x and z to sink y, with *vertex* capacity 1 everywhere, including the sources. Since G is biconnected, by definition its minimum vertex cut is at least 2. Hence, by flow/cut duality, we have flow at least 2. In fact, we have flow exactly 2, since we only have two sources, each with capacity 1. Furthermore, there exists an integer flow with this value, since all capacities are integer [2]. This gives us vertex-disjoint paths from x to y and from z to y, and by concatenation: from x to z via y. □

To show that a detour through any edge is possible, subdivide the edge, let y be the newly created vertex, and apply the lemma.

3.3 Upstream Features in BC-Trees

The algorithm for Upstream Enumeration on general graphs G works by using the algorithm for trees on the BC-tree of G. Run the algorithm with the following set A' (and ditto for B').

- Cut nodes are in A' if and only if the contained vertex has a label in A.
- Block nodes are in A' if and only if they contain a non-cut vertex that has a label in A or an edge with a label in A.

When the tree algorithm would output a node's label, instead we output the labels of all vertices and edges it contains (with one exception). Always doing this would be almost correct: by Lemma 3.1, any path through the BC-tree would imply a simple path through any vertex and any edge contained in any of the nodes. However, the lemma does not apply to blocks that contain exactly two vertices. (Nor does it apply to nodes that contain a single vertex – such as all cut nodes – but if that vertex is upstream then trivially all vertices in the node are upstream.)

For nodes containing exactly two vertices, a case analysis reveals the following procedure: always output the edge label, and for each of the two vertices, output its label if that label is in A.

[2] Wüpstream does not do this and as a result has runtime $\Theta(n + m + \Delta(|A| + |B|))$, where Δ is the maximum degree in the graph. For realistic A, B, and Δ, it is probably not worth it in practice to dynamically change the graph.

4 ENGINEERING

We now describe several technical aspects of our implementation. Since the task is solved by a fairly simple linear-time algorithm, implementation details become significant for the overall runtime. Not all of our techniques are advisable in a general context; in such cases we comment on responsible alternatives.

This short report lacks the space for an extensive experimental evaluation. Any quantitative claims are for illustration purposes only and refer to a single instance: the Naperville Electric Network instance from the contest website.[3] It consists of 8465 vertices and 9101 edges represented by approximately six megabytes of JSON; the sets A and B are singleton. Runtimes refer to an Intel® Core™ i5 Haswell-DT processor at 3.00GHz.

Our initial prototype in Java achieved a runtime of approximately 3 seconds on the Naperville instance, with the overwhelming majority of the time spent on I/O and parsing. Wüpstream is a performance-conscious reimplementation in C++11 and runs in approximately 50ms; some of its implementation details are described below. Note that this is in no way a comparison of the relative performance of Java and C++: the first implementation was a proof of concept, the latter was written for speed.

4.1 Overview

Our program performs the following steps in sequence, roughly following the description in Section 3.

Step		Time (ms)
1.	Read complete file to memory	3
2.	Parse JSON string, construct DOM in-situ	31
3.	Build graph representation	15
4.	Construct block-cut tree	8
5.	Mark toward controllers	0
6.	Flood from starting points and write output	0

Note that given the BC-tree, solving the Upstream Enumeration problem takes zero milliseconds. In a use case where upstream features are enumerated for multiple sets A and B, the BC-tree should be persisted between queries.

4.2 JSON Parsing

Our submitted program uses RapidJSON [8] for parsing the input network. This is an open source, high performance, highly compliant JSON parser. It has a convenient, modern DOM interface and comes highly recommended. Particularly relevant for performance are its SIMD-vectorised parser (using SSE4.2) and in-situ parsing.[4] With a baseline parsing time of 70ms, turning on the former saves 18ms and the latter saves another 21ms, dropping the total to 31ms. These gains rely solely on (open source) library code, but demonstrate that SIMD vectorisation and memory management can have a significant effect on runtime.

We experimented with a bespoke parser that integrates the graph-building step and spends a total of 18ms rather than (31 + 15)ms, resulting in an effective parsing time of 3ms. This code relies on the specifics of the contest file format and jumps over much of the file, looking at unneeded information as little as possible and ignoring the validity of the JSON. This strategy is extremely brittle to changes in the file format and can lead to silent parse errors that are nearly impossible to diagnose in the file itself. However, it does show that there is room for improvement and that we certainly pay a runtime cost for the file format. To properly handle situations where only a small fraction of a JSON DOM is relevant, one might consider approaches like Mison [7].

4.3 Vertex Identification

The input file format identifies vertices by 38-character strings. For the rest of the program, we represent vertices by objects (with address identity), so in order to construct the graph we maintain a dictionary from strings to our vertex objects. We use a hashmap and profiling shows that about 7% of the total runtime is spent on computing hash values. In our tests, the C++ compilers used state-of-the-art vectorised string hash implementations, so we do not expect much runtime can be saved on the hashing itself.

We have used `std::unordered_map` for our hashmap. Due to some performance-hostile specifics of the C++ standard, it may be expected that a faster hashmap implementation is possible,[5] but hashmap performance is a subtle matter that involves both code and data. We have not investigated, but suggest that Wüpstream may be improved here. Somewhat interestingly, using `std::map` (typically implemented using a search tree) is only 3ms slower.

4.4 Graph Representation

The connectivity of a vertex is represented by a vector of pointers to its neighbours. There is no object representation of edges: that would lead to two indirections to traverse an edge, rather than one; the label of each edge is stored redundantly at both its vertices.

The BC-tree is also represented using an object per node, each containing a vector of pointers to its neighbors. It is built incrementally during a single (augmented) depth-first search through the input graph. During this traversal, each node accumulates the set of vertex and edge labels it contains. When, during the flood step, the node is found to be upstream, we can output all the labels it represents without searching the original graph again. (Beware the exception for block nodes that contain exactly two vertices.)

4.5 Memory Management

Our code makes many small allocations (particularly the vertices of the graph and the nodes of the BC-tree), which makes memory management a performance-critical aspect of the program. In the context of the GIS Cup evaluation, our program gets invoked, outputs a solution, and is then discarded; at termination, the OS frees any memory and other resources we hold.

> *"The building is being demolished. Don't bother sweeping the floor and emptying the trash cans and erasing the whiteboards. (...) All you're doing is making the demolition team wait for you to finish these pointless housecleaning tasks."* – Raymond Chen [1]

Taking this advice to its logical extreme, we don't even bring a broom. Standard memory managers operate on the assumption

[3] http://sigspatial2018.sigspatial.org/giscup2018/download

[4] In-situ parsing uses the memory of the string itself to store the DOM objects. This renders the string unusable, but obviates dynamic memory allocation.

[5] Folklore suggests that an open-addressing flat hashmap would be faster than the bucket lists (practically) mandated by the standard.

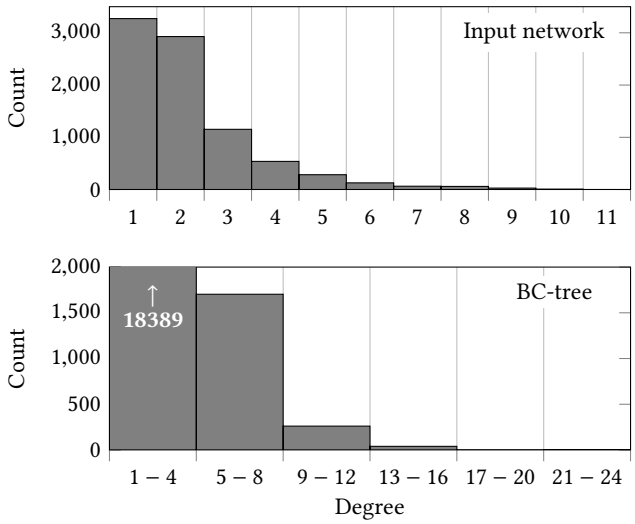

Figure 2: Histogram of degrees in the Naperville instance. The input network has maximum degree 11; the BC-tree has a small number of high-degree vertices, with a maximum 88.

that it must be possible to free memory in arbitrary patterns. Since we know that the building will get demolished behind us (so to speak), we make no effort to track anything.

It would be interesting to know the runtime cost of doing everything properly with modern smart pointers instead, but we have not done such a reference implementation. Regardless, for a high-performance implementation we advise a scoped arena allocator (see e.g. [3]): we expect that to have nearly identical performance to our leaky allocator.

4.6 Small-Vector Optimisation

In real-world networks we may expect many vertices of low degree, in the input graph as well as its BC-tree. For example, trees in the graph result in many degree-2 block nodes. Figure 2 shows the degree distributions of the aforementioned Naperville instance. We note that 93% of vertices and 90% of BC-nodes have degree 4 or less. We expect similar degree distributions for other real-world instances.

Wüpstream uses *small-vector optimisation* for all adjacency lists, with inline capacity 4. This technique (commonly applied to strings) stores a small number of elements "inline" within the vector object itself rather than in a buffer on the heap; if the vector contains more than this fixed number of elements, the implementation falls back to normal vector behaviour with an exponentially-growing buffer. For small vectors, where the number of elements is below the threshold, this saves the memory allocation for the buffer and saves an indirection when accessing the elements. The small-vector optimisation saves 2ms overall, compared to a `std::vector` baseline.

5 CONCLUSION

Our approach to solving the Upstream Enumeration problem relies on algorithm design as well as software engineering. When it comes to solving real world problems efficiently in practice, neither aspect can be ignored. A bad algorithm will not scale well with instance size, while a poor implementation might drown in overhead. When designing an algorithm, it is important to be aware of the rich literature of graph algorithms and data structures: the main algorithmic tool used in Wüpstream (the block-cut tree) can be traced back to at least 1969 [4]. On the other hand, knowledge of the performance characteristics of the implementation language and modern hardware is indispensable.

We finish by discussing some aspects of the algorithm and its implementation that may be improved. First of all, Wüpstream is "fighting code" for a competition. To be generally useful as a library, it would need to be cleaned up. As discussed in Section 4, we do not expect this to come at a significant cost in runtime since proper engineering techniques exist for each corner we have cut.

In fact, the current implementation is evaluated end-to-end, from an input file to an output file. When used in the context of a larger geographic information system, the graph is likely to be in main memory, cutting most of our runtime. Almost all the remaining time is spent on constructing the BC-tree, which can be reused between multiple upstream queries on the same graph, leaving only the mark and flood steps which take hardly any time. If the graph changes often, and these changes are interspersed with upstream queries, it may be beneficial to dynamically keep track of the BC-tree. Advanced data structures are available for this (e.g. [5]).

Building the BC-tree and marking toward controllers always processes the entire graph, even when the set of upstream features is small. This seems unavoidable in the purely abstract graph setting, but maybe the additional domain information (such as geometry) can be used to make an output-sensitive algorithm. For example, in some settings we may expect that all upstream features are near the starting points or have a guarantee that the network is planar.

Lastly, note that Wüpstream is single-threaded. The mark and flood steps should be fairly straightforward to parallelise, but this would only help for large numbers of controllers or starting points.

ACKNOWLEDGMENTS

We thank Johannes Blum and Steven Chaplick for some helpful discussions. Thomas C. van Dijk is supported by the German Research Foundation (DFG), grant number Di 2161/2-1 in the context of the priority programme *Volunteered Geographic Information.*

REFERENCES

[1] Raymond Chen. 2012. When DLL-PROCESS-DETACH tells you that the process is exiting, your best bet is just to return without doing anything. https://blogs.msdn.microsoft.com/oldnewthing/20120105-00.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.
[3] David R. Hanson. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience* 20, 1 (1990), 5–12. https://doi.org/10.1002/spe.4380200104
[4] F. Harary. 1969. *Graph theory.* Addison-Wesley Pub. Co.
[5] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic Deterministic Fully-dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-edge, and Biconnectivity. *J. ACM* 48, 4 (July 2001), 723–760. https://doi.org/10.1145/502090.502095
[6] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.
[7] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. In *The 43rd International Conference on Very Large Data Bases (VLDB 2017).*
[8] Milo Yip and THL A29 Limited, a Tencent company. 2015. RapidJSON. http://rapidjson.org/.