Dynamic Bottleneck Optimization for 2-Vertex and Strong Connectivity [⋆]

Orestis A. Telelis   Vassilis Zissimopoulos
{telelis,vassilis}@di.uoa.gr

Department of Informatics and Telecommunications
University of Athens
Panepistimioupolis, Ilissia, 157 84 Athens, Hellas (Greece)

Abstract. On a complete weighted graph that changes dynamically by edge weight updates, we consider the problem of maintaining efficiently a minimum value b, such that the set of edges with weights less than b induces a 2-vertex connected graph (in the undirected case) and a strongly connected graph (in the directed case) on the same vertex set. These problems find application in minimizing power consumption of wireless networks. We design a dynamic algorithm of $O(n\alpha(n)\log n)$ complexity per edge weight update for the first problem, and a dynamic algorithm for the second one, whose experimental analysis shows its appropriateness for use in practice.
Keywords: dynamic graph algorithms, biconnectivity, strong connectivity

1 Introduction

We consider the following two problems on a complete weighted graph K_n: determine the minimum weight value b (referred to as the *bottleneck* value) such that there is a set of edges $\{e \in K_n | w(e) \leq b\}$ (a *bottleneck subgraph*) inducing:

– a 2-vertex connected (*biconnected*) spanning subgraph on the initial set of vertices, when the input graph is undirected,
– a strongly connected spanning subgraph on the initial set of vertices, when the input graph is directed.

These problems find direct application in minimizing energy consumption of wireless adhoc networks [1, 2] and are solvable in polynomial time by well known greedy algorithms. In this work we consider their dynamic versions, where we have to re-evaluate the bottleneck value b efficiently after an edge weight has been updated. Our target is to develop algorithms that re-evaluate b in less time than the time required to solve the problem from scratch.

Our work falls in the field of dynamic graph algorithms. A dynamic graph algorithm maintains a graph property when the underlying graph changes by edge insertions and deletions (or equivalently edge weight decreases and increases). Dynamic maintenance of graph connectivity properties has been extensively dealt with (see e.g. [3]). Dynamic

[⋆] Partially supported by the Greek Ministry of Education under the program PYTHAGORAS II

network optimization problems include minimum cost spanning tree (MST) [4], shortest paths tree [5], and all-pairs shortest paths. Dynamic transitive closure (which is related to strong connectivity) on digraphs has recently seen progress [6].

We design dynamic algorithms for the biconnectivity (section 2) and strong connectivity (section 3) bottleneck. In section 4 we present some experimental results for our algorithms and conclude. Throughout the paper graphs are represented with their edge set and set operations over graphs are with respect to their edge sets, unless otherwise stated. Subgraphs are also edge-induced unless otherwise stated.

2 Biconnectivity

A graph is biconnected if removal of 1 vertex (along with its incident edges) does not disconnect the graph. Alternative characterization stems from Menger's theorem: the graph is biconnected iff there are 2 (internally) vertex-disjoint paths connecting every pair of vertices. It is known that a bottleneck biconnected subgraph (if it exists) can be found in linear time with respect to the input graph's edges [7]. We review here a simple static algorithm for obtaining the biconnectivity bottleneck. This algorithm, although of superlinear complexity, it will be useful for our purposes.

A Static Algorithm. The algorithm "grows" a bottleneck biconnected subgraph of a complete weighted graph K_n by first finding a minimum weight spanning tree (MST) of K_n, and subsequently augmenting it with additional edges to biconnect it:

```
1. Find a MST $T$ of $K_n$, and set $B = T$.
2. foreach $(u, v) \in K_n - B$ in order of non-decreasing weight do:
3.    if $u$ and $v$ not in the same biconnected component, insert $(u, v)$ in $B$.
4. return $\max_{e \in B} w(e)$.
```

This algorithm is referred to with `b-biconnect`. The final subgraph B produced by `b-biconnect` is sparse, i.e. $|B| = O(n)$. Indeed, the MST T contains exactly $n - 1$ trivial biconnected components (the MST edges), and since each edge insertion in a second phase always merges at least two different biconnected components into one, at most $O(n)$ insertions take place. The algorithm is implemented using the incremental algorithm of [8] for maintaining the biconnected components under edge insertions: each insertion incurs $O(\alpha(n))$ amortized time and a query of whether two vertices are in the same biconnected component also incurs $O(\alpha(n))$ time. α denotes the inverse of Ackermann's function. A simple implementation of `b-biconnect` incurs $O(n^2 \log n)$ complexity (due to sorting the edges). We introduce a definition:

Definition 1. *A* Bottleneck Spanning Biconnector *(BSB) of a weighted (not necessarily biconnected) graph G is an edge subgraph $B \subseteq G$, such that for every edge $(u, v) \notin B$ there are two internally vertex-disjoint paths in B connecting u to v, and for every edge e on these paths $w(e) \leq w(u, v)$.*

We show that if G is biconnected, a BSB $B \subseteq G$ contains a bottleneck biconnected subgraph of G. If G is biconnected, B is also biconnected (otherwise the definition is contradicted). Let G have biconnectivity bottleneck b. Let $B^+ \subseteq B$ contain edges with weight $> b$ and assume that, $B - B^+$ is not biconnected. Then there are some edges $B^- \subseteq (G - B)$, with weight $\leq b$, such that $(B - B^+) \cup B^-$ is biconnected. Edges in B^- weigh less than edges in B^+, thus there is at least one edge $e \in B^- \subseteq (G - B)$ with $w(e)$ weighing less than some edge on one of the two vertex-disjoint paths biconnecting the endpoints of e in B: a contradiction to the BSB definition.

Lemma 1. *Algorithm* `b-biconnect` *produces a BSB of its input graph.*

Proof. It is an MST property that for every edge $(u, v) \notin T$ there is a $u - v$ path in T with $w(e) \leq w(u, v)$ for all edges e in the path. Since u is not biconnected to v in T, it must be the case that a set of edges in $B - T$ picked prior to examination of (u, v) biconnected u to v. Examination of edges in non-decreasing weight order gives the result. $\square$

The dynamic algorithm. The complete weighted graph K_n is encoded in a *sparsification tree* [3] in the following way. For each $e \in K_n$ the tree has a leaf node x_e, which represents the graph $G(x_e)$ of n vertices and of a single edge e. Each internal node x has two children, say y and z, and it represents the graph $G(x) = G(y) \cup G(z)$. The root node r of the tree represents the complete weighted graph $G(r) = K_n$. Since at some level of the tree there might be an odd number of nodes, at most one of the nodes in the upper level may eventually have three children. In the sequel we consider nodes with two children, without affecting generality.

Let x be a node of the sparsification tree and y, z be its children. Store at x the edge subgraph $C(x) \subseteq G(x)$, which is the output of `b-biconnect` when executed on $C(y) \cup C(z)$. For leaf nodes x_e set $C(x_e) = G(x_e)$. It is shown that $C(r)$ is a sparse bottleneck biconnected subgraph of $G(r) = K_n$:

Lemma 2. *(***Monotonicity of BSBs***) Let $G = G_1 \cup G_2$ be an arbitrary edge partition of a weighted (not necessarily biconnected) graph G, and $C_1 \subseteq G_1$, $C_2 \subseteq G_2$, be BSBs of G_1, G_2 respectively. Then $C_1 \cup C_2$ is a BSB of G.*

Proof. For every edge $e = (u, v) \in G - C_1 \cup C_2$ there exist two vertex-disjoint paths, either in C_1 or in C_2, connecting u to v, and for every edge e' of these paths $w(e') \leq w(e)$, by definition. $\square$

This implies that $C(r)$, being the output of `b-biconnect` is a BSB of $G(r)$, and clearly a bottleneck biconnected subgraph of K_n, by lemma 1. Consider updating the weight of an edge e. The dynamic algorithm is as follows: let x_e be the leaf node that contains e. Starting from x_e and following the path from x_e towards the root r of the sparsification tree recompute $C(x)$ for all nodes x on the path. Then $C(r)$ is a novel bottleneck biconnected subgraph of K_n.

Theorem 1. *The biconnectivity bottleneck can be maintained in $O(n\alpha(n) \log n)$ time per edge weight update.*

Proof. The sparsification tree is of $O(\log n)$ height. Algorithm `b-biconnect` outputs edges sorted in order of non-decreasing weight. Thus merge-scanning the BSB edges stored at a nodes's children provides them in order of non decreasing weight. *Union-find* data structures are used for computing the MST, while augmentation towards biconnectivity is performed using the incremental algorithm of [8]. Since at most $O(n)$ edges are merge-scanned for each tree node, the result follows. $\square$

The sparsification tree has $O(n^2)$ nodes, and finding $C(x)$ for each node x incurs $O(n\alpha(n))$ complexity. Bottom-up construction incurs $O(n^3\alpha(n))$ complexity. Furthermore, the dynamic algorithm is of $O(n^3)$ space, since each node stores $O(n)$ edges. One intriguing question is whether lower complexity can be achieved per edge update. In the context of maintaining explicitly a bottleneck biconnected subgraph the proposed algorithm is optimal up to polylogarithic factor: *there is a complete graph with appropriately set edge weights and an infinite sequence of updates such that the bottleneck biconnected subgraph is sparse and unique and each update causes it to change by $O(n)$ edges.* We omit a formal proof of this due to lack of space.

3 Strong Connectivity

For the strong connectivity bottleneck we design a lazy dynamic version of a well known static algorithm. A lazy algorithm encodes the execution of a static algorithm in an appropriate data structure, and when an update of the input data occurs, it efficiently invalidates encoded information relevant to the updated portion of the input data. The static algorithm is then executed to complete a partial solution. The lazy algorithmic approach does not generally improve on complexity, but does improve on running time in practice. There are partial arguments against existence of improved complexity dynamic algorithms for strong connectivity [9]. Such a lazy algorithm is the only known alternative for dynamic shortest paths [5, 10].

We assume a complete weighted digraph K_n with arc weights $w(a) \geq 0$. For each arc $a = (u, v)$, $t(a) = u$ is the tail vertex of a, while $h(a) = v$ is its head vertex. *Weakly/strongly connected components* are abbreviated to WCC/SCC. For a vertex subset S let $\delta(S) = \{a \in K_n | h(a) \in S, t(a) \notin S\}$ (exactly the "in" cut-set of S).

A static algorithm. A *contraction* algorithm, also used for calculating a minimum weight directed minimum spanning forest [11, 12] is discussed. A contraction operation applies to a directed cycle of vertices and replaces the cycle by a single vertex (called the *contraction vertex*), maintaining all arcs incident to cycle vertices if one their endpoints do not belong in the cycle. The algorithm is as follows:

```
1. $H \leftarrow \emptyset$
2. while $K_n$ is not a single vertex do:
   (a) pick a vertex $v$ with $\delta(v) \cap H = \emptyset$ and $\delta(v) \neq \emptyset$.
   (b) let $a^\star \leftarrow \arg\min_{a \in \delta(v)} w(a)$ and insert $a^\star$ in $H$.
   (c) if a directed cycle has occurred, contract the cycle into a single vertex.
3. return $\max_{a \in H} w(a)$
```

This algorithm is referred to with `b-str_connect`. One can easily verify that it produces a bottleneck strongly connected H. At most $O(n)$ contractions take place, hence $|H| = O(n)$. Every contraction vertex in some iteration corresponds to a SCC of H having emerged in that iteration, as a directed cycle between smaller SCCs. Hence

a contraction vertex corresponds to a set of vertices of the initial graph, having formed a SCC with respect to the current H of some iteration. We refer to these sets as *active sets*. The vertices of the initial graph are trivial SCCs and trivial active sets. In each iteration of `b-str_connect` an active set S (represented with a contraction vertex) is associated with a *unique* arc, which is denoted with $a(S)$. Accordingly let $S(a)$ be the active set which caused insertion of a in H. Implementation of `b-str_connect` is discussed in [13, 12, 11], and for a complete digraph its complexity is $O(n^2)$.
Data Structure. A natural tree data structure, namely the *Active Sets Tree* (AST), encodes the contractions performed by `b-str_connect`. Its vertices are referred to as 'nodes' in order to distinguish them from the digraph's vertices. The AST is defined as follows:

– Each contraction vertex (also an active set) is represented with a single AST node S. Each initial digraph vertex v is also represented with a 'trivial' node denoted with $\{v\}$.
– An AST node S stores: $a(S)$, $children(S)$, $parent(S)$.
– If S is not trivial $children(S)$ holds pointers to AST nodes representing contraction vertices which formed a directed cycle and were contracted to the vertex represented by S.
– $parent(S)$ is a pointer to an AST node R which represents a vertex to which the vertex represented by S was contracted. If S is not contracted $parent(S) = \{t(a(S))\}$.

These definitions imply a tree data structure, i.e. the AST, because $parent(S)$ is unique: the S-vertex was either contracted to a unique vertex, or not, in which case $a(S)$, hence $\{t(a(S))\}$ is unique. The AST has $O(n)$ nodes. From each node S the active set S can be constructed by a breadth first search (BFS) starting from S towards its trivial descendants. Implicit access to active supersets of S is given by following a path from S towards the AST root. T_S will denote the AST-subtree rooted at node S. A straightforward augmentation of `b-str_connect` can provide the AST along with H: each time an arc incoming to some vertex (with AST node representation S) is selected, set $a(S)$ and $parent(S) = \{t(a(S))\}$. When a directed cycle is contracted place a new AST node S, and make it the parent node of the cycle's nodes, while appropriately setting $children(S)$.

3.1 The dynamic algorithm

Let H be a bottleneck strongly connected subgraph. Then, increasing the weight of an arc at a value less than the current bottleneck b or if the arc does not belong in the maintained bottleneck subgraph, the optimum bottleneck is not affected. This is the case also when decreasing the weight of an arc at a value greater than the current bottleneck. The overall structure of the dynamic algorithm is the following:

```
increase($a, w'(a)$)
1. if $a \in H$ and $w'(a) > b$ do:
2.    invalidate part of $H$ relevant to $a$
3.    update $w(a)$ to $w'(a)$
4.    update data structures
5.    execute b-str_connect
```

```
decrease($a, w'(a)$)
1. if $w'(a) < b$ and $a$ replaces $a' \in H$ do:
2.    invalidate part of $H$ relevant to $a'$
3.    update $w(a)$ to $w'(a)$
4.    update data structures
5.    execute b-str_connect
```

We discuss the important parts of the two operations, namely *invalidating part of H* with respect to an arc a (for **increase**), the *replacement test* performed by **decrease**, which decides whether $a \notin H$ should replace some arc of H, and usage and update of data structures. Note that **increase** and **decrease** differ only in the *replacement test*.

Invalidating H. Invalidation of part of H with respect to an arc $a \in H$, consists of removing from H $a(R)$ for all $R \supseteq S$, and deleting the corresponding AST nodes for all $R \supset S$. Deletion of the AST nodes results in a re-arrangement of the AST. In particular, starting from $S(a)$ and following a path towards the AST root, the following operations are performed:

1. Remove R from the AST and $a(R)$ from H.
2. All children C of R change their parent: set $parent(C) = \{t(a(R))\}$ and add C to $children(\{t(a(C))\})$.

This process is precisely the reverse of the one followed for the construction of the AST in the previous paragraph. It should also be noted that S is not entirely invalidated: it only loses its $parent(S)$ and $a(S)$, but it is subject to processing from `b-str_connect` when it is re-executed (i.e. it will be a SCC vertex set of the updated H). One can verify that this process rearranges the AST in such a way, that S becomes its new root. Furthermore the AST does not become disconnected: all nodes changing their parent are hanged under the subtree T_S, by definition of contractions (a more formal proof is omitted due to lack of space). During invalidation each AST node is touched at most once. Hence invalidation is of $O(n)$ complexity.

Replacement Test. Consider an arc $a \notin H$, decreasing its weight to $w'(a) < b$. This may cause the replacement of an arc $a' \in H$. This happens if an active set S with $a(S) = a'$ has $w'(a) < w(a')$ and $a \in \delta(S)$. We care for the *earliest* (lower lying in the AST) such active set S, which should replace its $a(S)$.

Such an active set is identified by following a path from $\{h(a)\}$ towards the AST root and checking for each node S visited whether $w(a(S)) > w(a)$. If no such node S is found the test fails. Otherwise $a \in \delta(S)$ is tested: for this purpose S is constructed by a BFS on T_S as explained previously. If this test is positive also, then AST information related to $a(S) = a'$ is cancelled, data structures are updated, and `b-str_connect` is executed. Identifying a candidate active set and constructing it takes $O(n)$ time.

Notably, the strategy of re-executing `b-str_connect` from scratch per arc weight decrease (below the current bottleneck) has no way of performing a *replacement test*, because active sets are not maintained.

Data Structures and Complexity. All known implementations of `b-str_connect` [13, 11, 12] handle contraction using *union-find* disjoint set representations of SCCs and WCCs of H. For each (contracted) SCC S unionable priority queues implement $\delta(S)$ [11]. For a complete digraph the achieved complexity is optimum $O(n^2)$. These data structures must be initialized so as to reflect the state of H after invalidation related to some arc a. As explained previously, H invalidation results in a tree rooted at $S(a) = S$, hence invalidated H remains weakly connected (a single WCC). A single $O(n)$ time BFS of the AST can provide all the SCCs with respect to the current H set. Simply store for each S node of the AST its trivial descendants. Hence disjoint sets representation of

Gain in Execution Time per Weight Update		
Sequence Type	**Biconnectivity**	**Strong Connectivity**
Decrease Only		$90\% - 92\%$
Mixed	$\tilde{O}(n)$ vs. $O(n^2)$	$62\% - 65\%$
Increase Only		$39\% - 41\%$

Table 1. Summary of results.

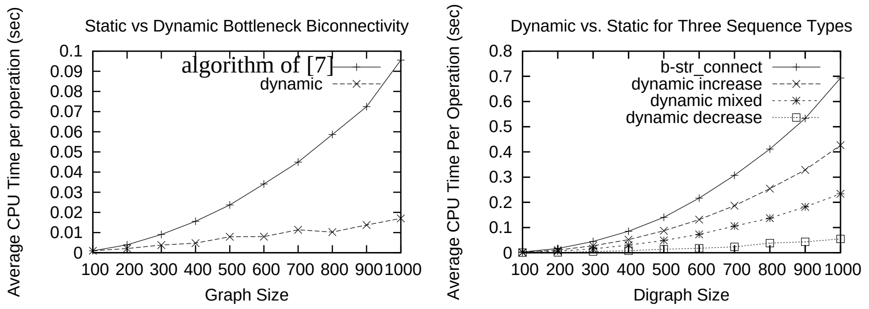

Fig. 1. The proposed dynamic algorithms against executing the static ones per edge update. Biconnectivity is examined on the left, while strong connectivity on the right diagram.

the SCCs can be found in $O(n)$ time. A priority queue is then initialized for each such SCC, by scanning the arcs incoming to its vertices in a total of $O(n^2)$ time.

4 Experimental Evaluation

Both dynamic algorithms were evaluated experimentally against re-execution of the static ones per weight update. Implementations were grown in standard C++, using the `gcc 3.3.2` compiler. `CPU` time was acquired by the `getrusage()` system call on a `P4 2.4GHz, 512MB` machine under `Linux Kernel 2.6.11`. Average `CPU` time per weight update was measured over 100 sequences with 10000 weight updates, of differing initial complete graphs of 100 to 1000 vertices and edge weights drawn uniformly in $1 \ldots 10000$. A weight update was decided to be an *increase* or *decrease* with $\frac{1}{2}$ probability. Weight increase raises the weight of a randomly selected edge belonging in the maintained subgraph beyond the bottleneck value b, while weight decrease was performed on a randomly selected edge not belonging in the maintained subgraph and below b. Hence the algorithms could not ignore an update (without executing at all). Table 1 summarizes the gain in execution time per weight update.

The dynamic algorithm for the biconnectivity bottleneck was compared against the algorithm of [7] with $O(n^2)$ complexity per update on sequences of mixed weight increase/decrease operations selected with probability $\frac{1}{2}$. The graph on the left of fig. 1 depicts the average `CPU` time per operation taken by each strategy and confirms the asymptotic superiority of our dynamic algorithm.

The dynamic algorithm for the strong connectivity bottleneck handles weight increases and decreases in an asymmetric manner, because it employs an additional *replacement test* for weight decreases, before invalidating the data structure. For this reason the algorithm was tested on three types of sequences: increase-only, decrease-only, and mixed operations. Average `CPU` times per operation are compared on the right of fig. 1 for each type of sequence against the average execution time of `b-str_connect` (it was roughly the same for all three sequence types, hence an average is depicted). As shown in table 1 an impressive stability of gain in `CPU` time over all graph sizes is evident. The discussed replacement test for weight decrease operations amplifies the performance of the algorithm and contributes to the average of mixed sequences.

References

1. Ramanathan, R., Rosales-Hain, R.: Topology control of multihop wireless networks using transmit power adjustment. In: Proc. of IEEE INFOCOM'00. (2000) 404–413
2. Li, N., Hou, J.: Topology Control in Heterogeneous Wireless Networks: Problems and Solutions. In: Proc. of the 23rd Conf. of the IEEE Communications Society, INFOCOM'04. (2004)
3. Eppstein, D., Galil, Z., Italiano, G.F., Nissenzweig, A.: Sparsification - a technique for speeding up dynamic graph algorithms. Journal of the ACM **44**(5) (1997) 669–696
4. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Journal of the ACM **48** (2001) 723–760
5. Ramalingam, G., Reps, T.W.: On the Computational Complexity of Dynamic Graph Problems. Theoretical Computer Science **158**(1& 2) (1996) 233–277
6. Sankowski, P.: Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract). In: Proc. of the IEEE Conf. on Foundations of Computer Science, FOCS'04. (2004) 509–517
7. Manku, G.S.: A linear time algorithm for the Bottleneck Biconnected Spanning Subgraph problem. Information Processing Letters **59**(1) (1996) 1–7
8. Westbrook, J., Tarjan, R.E.: Maintaining bridge-connected and biconnected components online. Algorithmica **7**(5&6) (1992) 433–464
9. Khanna, S., Motwani, R., Wilson, R.H.: On Certificates and Lookahead in Dynamic Graph Problems. In: ACM-SIAM Symposium on Discrete Algorithms, SODA'96. (1996) 222–231
10. Roditty, L., Zwick, U.: On Dynamic Shortest Paths Problems. In: Proc. of the 12th Annual European Symposium on Algorithms, ESA'04. (2004) 580–591
11. Gabow, H.N., Galil, Z., Spencer, T.H., Tarjan, R.E.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica **6**(2) (1986) 109–122
12. Mendelson, R., Thorup, M., Zwick, U.: Meldable RAM priority queues and minimum directed spanning trees. In: Proc. of the ACM-SIAM Symposium on Discrete Algorithms, SODA'04. (2004) 40–48
13. Tarjan, R.E.: Finding optimum branchings. Networks **7** (1977) 25–35