National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science and Informatics

Optimal Offline Fully Dynamic 2-Edge Connectivity
*– Survey & Implementation**

Ioannis Rizas

July 14, 2025

*based on *R.Peng*, *B.Sandlund* and *D.D.Sleator* paper [10]

Supervisors

My Supervisors:

- D. Fotakis, N.T.U.A. (National Technical University of Athens)
- L. Georgiadis, U.I. (University of Ioannina)

Overview

Previous Work on Dynamic Connectivity

- offline MST: $O(\log n)$ — Eppstein et al. (1994) [2]
- online MST: $O(\log^4 n)$ — Holm et al. (2015) [3]
- offline updates & online edge connectivity queries : $O(\log n)$ — Karczmarz et al. (2015) [8]
- offline updates & online 2-edge connectivity queries : $O(\log n)$ — Karczmarz et al. (2015) [8]
- online connectivity: $\tilde{O}(logn)$ — Huang et al.(2022) [7]
- online 2-edge connectivity: $\tilde{O}(log^2 n)$ — Holm et al.(2018) [6]
- online 2-vertex connectivity: $\tilde{O}(log^2 n)$ — Holm et al.(2025) [4]
- online 3-edge connectivity: $O(n^{2/3})$ — Eppstein et al.(1997) [1]
- online 3-vertex connectivity: $O(n)$ — Eppstein et al.(1997) [1]

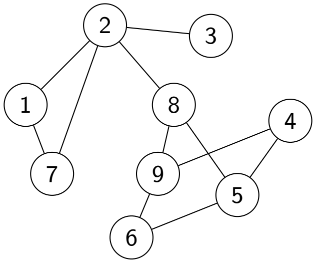
*$\tilde{O}(\cdot)$ polylogarithmic factors—typically powers of log n—in asymptotic complexity bounds. Formally, for a function $f(n)$, we write $\tilde{O}(f(n)) = O(f(n) \cdot \log^k n)$ for some constant $k \geq 0$.

Definitions

What is given as input?

Initial Graph:

- $G = (V, E)$: an **undirected** and **unweighted** graph
- V: the initial set of vertices
- E: the initial set of edges connecting pairs of vertices



Offline Events List:

- $I(a, b)$: **Insertion** of an edge between vertices a and b
- $D(a, b)$: **Deletion** of an edge between vertices a and b
- $Q(a, b)$: **Query** regarding the 2-edge connectivity between vertices a and b

1	2	3	4	...	60	61	...
I(5,6)	D(4,3)	Q(2,1)	I(5,4)	...	D(8,9)	I(2,5)	...

Definitions (1/3) – Online vs Offline

Online vs Offline Problems

In the **offline** setting, the entire sequence of operations is known in advance, allowing pre-processing and global optimization.

In contrast, the **online** setting processes operations one-by-one, without any knowledge of future operations, requiring immediate response to each.

- Offline algorithms provide a strong lower bound benchmark for evaluating the performance of online algorithms.
- Online algorithms often offer efficient greedy or approximation strategies for tackling the offline problem.
- This work focuses exclusively on the offline setting, as it constitutes the primary subject of our study and contributions.

Definitions (2/3) – k-edge connectivity

Query(a,b) for k-edge connectivity

Given an undirected graph $G = (V, E)$, two vertices $a, b \in V$ are said to be k-**edge connected** if and only if

$$\lambda(a, b) = \min_{\substack{S \subseteq E \\ a \not\leftrightarrow b \text{ in } G-S}} |S| \geq k,$$

where $\lambda(a, b)$ denotes the minimum number of edges whose removal disconnects a from b, and $G - S$ is the graph after removing edges in set S. The query $Q(a, b)$ returns `true` if and only if vertices a and b are k-edge connected, i.e., $\lambda(a, b) \geq k$.

This work focuses on 2-edge connectivity.Thus, for simplicity, every query Q(a,b) implicitly refers to checking 2-edge connectivity between vertices a and b

Definitions (3/3) – What's the problem we're tackling?

Optimal Offline (Fully) Dynamic 2-Edge Connectivity Problem

Given an initial graph $G = (V, E)$ and a sequence of update and query operations known in advance, the goal is to answer all queries of 2-edge connectivity correctly and efficiently. The optimal total runtime is $\mathcal{O}(t \cdot \log n)$, where:

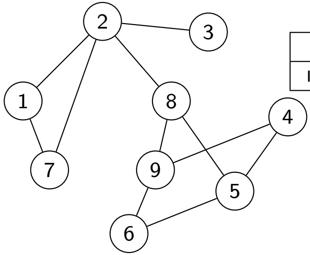
- $n = |V|$: number of vertices
- t: total number of operations (insertions, deletions & queries)

Note: This matches the proven lower bound of $\Omega(\log n)$ per operation by Patracscu & Demaine [9], for both the online and offline settings.

Events Processing – Subroutines

Processing Events - Modified input (Part 1)

Initial Graph:



Initial Operations:

1	2	3	4	...	60	61	...
I(5,6)	D(4,3)	Q(2,1)	I(5,4)	...	D(8,9)	I(2,5)	...

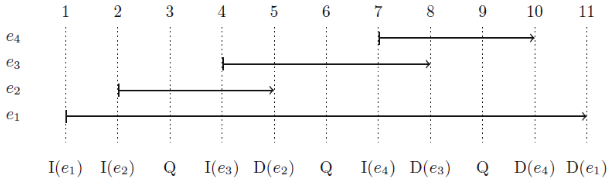
Total sequence of operations

1	2	3	...	11	12	13	14	...	71	72	...
I(1,2)	I(3,2)	I(4,5)	...	I(8,9)	I(5,6)	D(4,3)	Q(2,1)	...	D(8,9)	I(2,5)	...

From now on, the input will be a finite sequence of operations that must be performed on the graph.

Processing Events - Edge Lifespan

Each edge e has a unique lifetime interval $[I(e), D(e)]$, denoting its insertion and deletion times.


1 2 3 4 5 6 7 8 9 10 11
e_4
e_3
e_2
e_1
I(e_1) I(e_2) Q I(e_3) D(e_2) Q I(e_4) D(e_3) Q D(e_4) D(e_1)

Given a subinterval $[l, r]$ of the operation timeline, we classify edges as:

- **Permanent Edges:** $I(e) \leq l \leq r \leq D(e)$
  Edge e is active throughout the interval $[l, r]$.
- **Non-Permanent Edges:** $I(e) \in (l, r)$ or $D(e) \in (l, r)$
  Edge e's lifetime overlaps with but does not fully span the interval.

Processing Events – Active Vertices

Definition

Given a time interval of operations $[s, t]$, all vertices that participate in at least one operation within this interval are called **active**. All other vertices are considered **inactive** during this period.

Computation Strategy: The simplest and most efficient method is a linear pass over the operations within the given interval. Every vertex involved in an operation is marked as active. After this pass, all unmarked vertices are classified as inactive for the interval.

Processing Events – Permanent Edges

Definition

Permanent edges in a time interval of operations are those for which there is no activity of any kind (insertion or deletion) within that interval. In other words, for a given interval of operations, these edges are considered *stable*, as they do not change during that period.

Computation Strategy:
Given a time interval $[s, t]$, identifying the permanent edges for any sub-interval $[i, j] \subseteq [s, t]$ requires selecting those edges $e_{x,y}$ such that: $t_{I(x,y)} < i \wedge t_{D(x,y)} > j$
Computing permanent edges directly is neither straightforward nor efficient without appropriate pre-processing

Processing Events – Permanent Edges: Computation Strategy

For each operation at any given time, we want:

- Immediately determine the deletion time corresponding to an `Insert` operation.
- Immediately determine the insertion time corresponding to a `Delete` operation.

With this information, all permanent edges in any sub-interval can be identified in a single linear pass.

Events Pre-processing Strategies

1. **Sorting Algorithm:**
   An algorithm with time complexity $O(tlogt)$, which sorts the operations in a specific order to facilitate efficient processing.

2. **Hash-Based Algorithm:**
   An amortized $O(t)$ algorithm that employs a hash map to enable constant-time access to insertion and deletion times.

Processing Events – Modified Input (Part 2)

From now on, the final augmented sequence of events will include lifespan information.

Events list:

1	2	3	...	11	12	13	14	...	71	72	...
I(1,2)	I(3,2)	I(4,5)	...	I(8,9)	I(5,6)	D(4,3)	Q(2,1)	...	D(8,9)	I(2,5)	...

Final Augmented Events

1	2	...	11	12	13	14	......
I(1,2, ∞)	I(3,2, 50)	...	I(8,9, ∞)	I(5,6, 53)	D(4,3, 5)	Q(2,1, -)	......

Graph Processing – Subroutines

Graph Processing: Sparsification Overview

- **Bridge Tree Construction** — Identifies 2-edge-connected components by contracting bridges in a static graph.
- **Bridge Tree with Active Vertices** — Extends the bridge tree by incorporating vertex activity, marking components as active if they contain at least one active vertex.
- **Pruned Bridge Tree** — Further simplifies the bridge tree by removing inactive components, retaining only the necessary structure for answering queries efficiently.
- **Identical Graph Construction** — Integrates the above subroutines to construct a sparsified graph that is functionally equivalent to the original. This reduced graph preserves all necessary information to correctly answer the full sequence of dynamic graph queries.

Graph Processing: Sparsification Overview

Motivation:
Given a sequence of operations:

- Each vertex may be active or inactive
- Active vertices are those involved in update operations (delete or insert)
- The objective is to construct an *identical graph* that retains mostly the active parts of the original graph

Goal:
Efficiently transform a dynamic graph into a sparser, equivalent representation that preserves only the information necessary to answer queries over a specific sequence of events.

**All algorithms operate on static graphs, or more precisely, on a static configuration extracted from a dynamic graph.*

2-edge connectivity (Static Graphs)

The standard method for 2-edge connectivity constructs the **Bridge Tree**, a tree representing the graph's 2-edge connected components.

Bridge Tree Algorithm – 2-edge connected components

Steps:

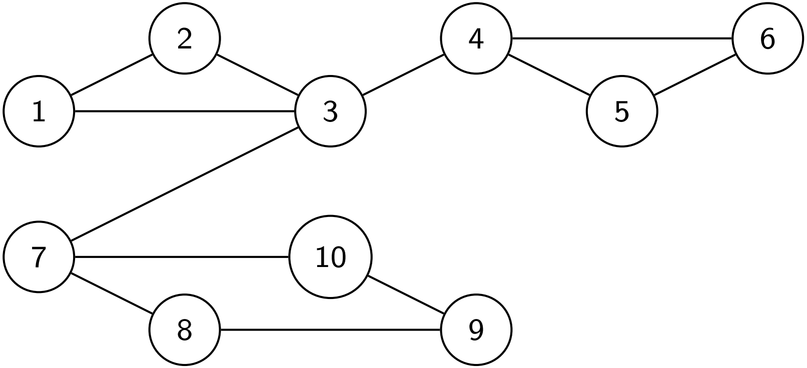
1. **Bridge Identification:** Find all bridges (critical edges whose removal disconnects the graph) in $\mathcal{O}(N + M)$ time using Tarjan's algorithm based on DFS.
2. **Component Construction:** Run DFS again, ignoring the identified bridges, to group vertices into 2-edge connected components.
3. **Labeling Components:** Assign a unique component ID to each connected group found in the previous step.

Time complexity: $\mathcal{O}(N + M)$

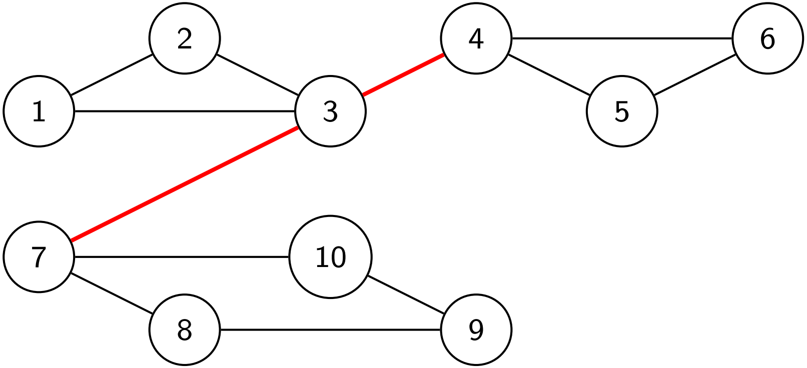
2-edge connectivity (Static Graphs)

- To answer 2-edge connectivity queries, simply check if the vertices belong to the same component in the Bridge Tree.
- This algorithm is both simple and optimal, running in linear time relative to the graph size. It is fundamental to sparsification techniques covered later

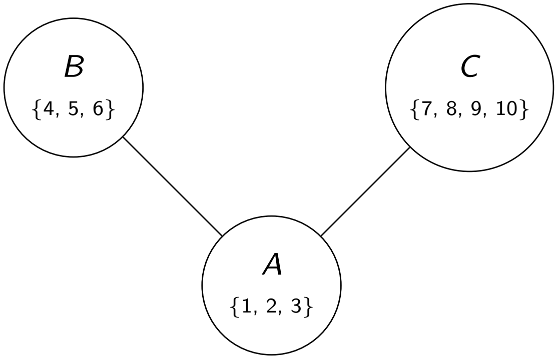
Bridge Tree Algorithm - Example (1/3)



Bridge Tree Algorithm - Example (2/3)



Bridge Tree Algorithm - Example (3/3)



Bridge Tree Algorithm with Active Vertices

A refined version of the classical bridge tree algorithm that accounts for *vertex activity*. This modification is fundamental to our framework, particularly within the context of graph sparsification.

Definition (Bridge Tree with Active Vertices)

Let $G = (V, E)$ be an undirected graph, and let $\mathit{activity} : V \to \{\text{true}, \text{false}\}$ be a boolean array indicating the activity status of each vertex.

The **bridge tree** of G is defined as in the standard setting, with the extension that a 2-edge-connected component is considered **active** if it contains at least one vertex $v \in V$ such that $\mathit{activity}(v) := \text{true}$.

Bridge Tree Algorithm with Active Vertices

Bridge Tree Algorithm with active vertices

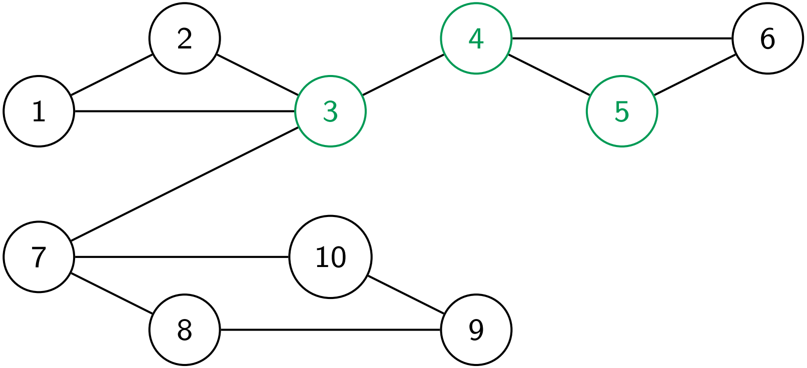
Steps:

1. **Bridge Identification** (same as before)
2. **!Component Construction:** Same as before and if at least one vertex within a 2-edge-connected component is *active* the entire component is considered *active*.
3. **Labeling Components** (same as before)

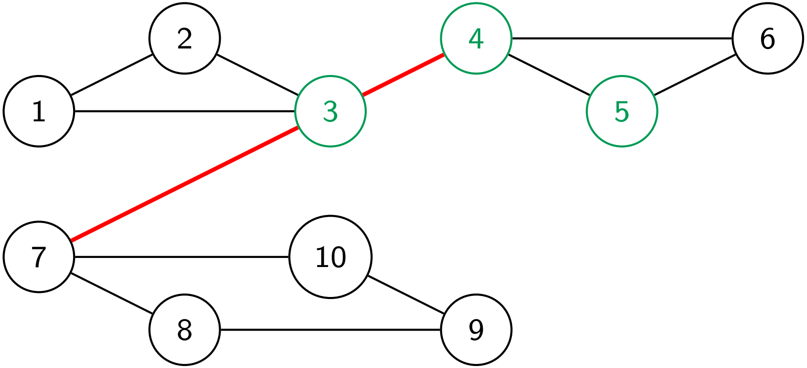
Time complexity: $\mathcal{O}(N + M)$

- Active components are the ones affected by update events within the time interval
- The activity status of vertices can be efficiently tracked using a boolean array

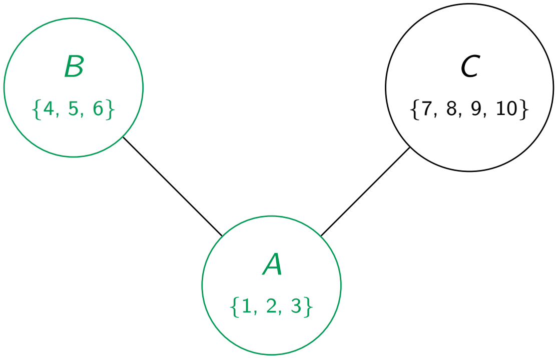
Bridge Tree Algorithm with Active Vertices- Example (1/3)



Bridge Tree Algorithm with Active Vertices- Example (2/3)



Bridge Tree Algorithm with Active Vertices- Example (3/3)



Pruned Bridge Tree

Given a bridge tree $\mathcal{T}(V, E)$, we can further simplify (prune) it according to the following rules:

Pruning rules

1. If a vertex is *inactive* and has degree 0 or 1, it is removed
2. If a vertex is *inactive* and has degree 2, we contract it by adding an edge between its two neighbors and then remove the vertex

Computation Strategy:
A linear-time algorithm is employed to efficiently perform the pruning operation on the bridge tree while preserving its structural correctness.

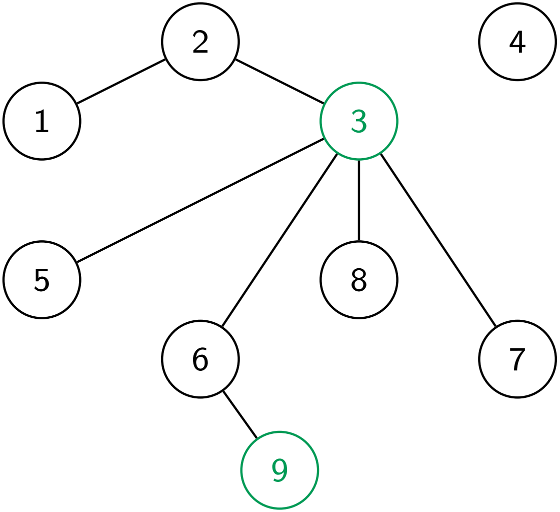
Pruning Bridge Tree - Algorithm

Linear-Time Pruning Algorithm

The algorithm prunes inactive vertices from the bridge tree in three phases:

- **Pruning Inactive Leaves (Degree 1):**
  - Identify all inactive vertices of degree 1 and place them in a queue.
  - While the queue is not empty:
    - Remove the current vertex from the graph.
    - Decrease the degree of its neighbor.
    - If the neighbor becomes a degree-1 inactive vertex, add it to the queue.
  - *Time Complexity:* $O(N + M)$ (Each vertex is processed at most once).
- **Pruning Inactive Isolated Vertices (Degree 0):**
  - Remove all inactive vertices with no edges in a single pass.
  - *Time Complexity:* $O(N + M)$
- **Pruning Inactive Internal Vertices (Degree 2):**
  - For each inactive vertex of degree 2, connect its neighbors directly and remove the vertex.
  - This operation does not affect the degree of unrelated vertices.
  - *Time Complexity:* $O(N + M)$

Pruning Bridge Tree – Example (1/6)

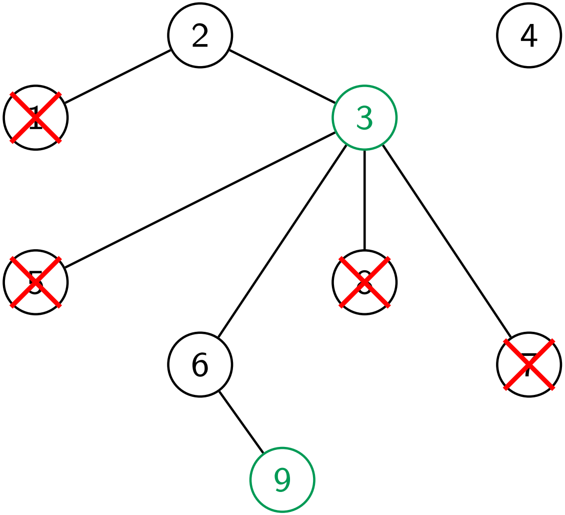


Bridge Component Map for vertices:

Original	Current
1	1
2	2
3	3
4	4
5	4
6	4
7	5
8	6
9	7
10	8
11	9

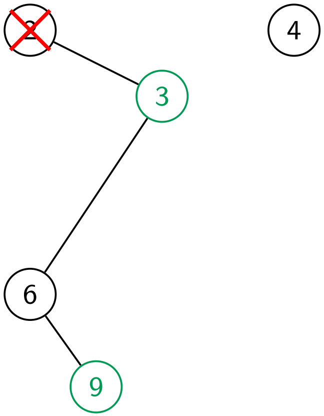
Pruning Bridge Tree - Example (2/6)

Remove inactive vertices with degree 1:



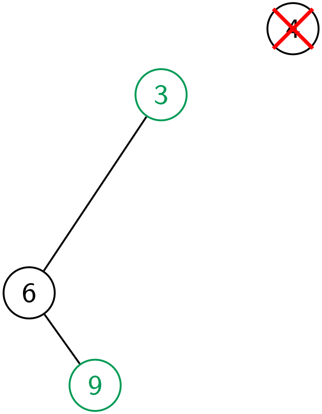
Pruning Bridge Tree - Example (3/6)

Remove inactive vertices with degree 1:



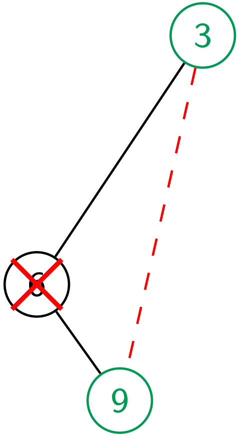
Pruning Bridge Tree - Example (4/6)

Remove inactive vertices with degree 0:

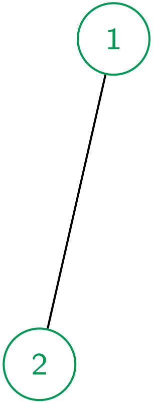


Pruning Bridge Tree - Example (5/6)

Remove inactive vertices with degree 2:



Pruning Bridge Tree – Example (6/6)



All Component Maps for vertices:

Original	**Br.Tree Map**	**Pruned Map**
3	3	1
11	9	2

Graph Processing – Equivalent Graph

Main subroutine

Equivalent Graph – Strategy Overview

Equivalent Graph Construction

Input:

- graph representation
- representatives (component) mapping
- activity status of original vertices
- activity status of the graph vertices

Execution Steps:

1. **Bridge Detection Stage** — Identify all bridges in the original graph, which act as critical edges separating 2-edge-connected components.
2. **Bridge Tree Construction Stage** — Contract the graph along its bridges to form the bridge tree, where each node represents a 2-edge-connected component.
3. **Pruning Stage** — Simplify the bridge tree by removing inactive components based on predefined pruning rules, retaining only relevant structure for efficient query resolution.
4. **Representative (Component) Mapping Update Stage** — Update the representative mapping of the original vertices according to the final pruned structure, ensuring consistency for subsequent operations.

Representatives Mapping Update

After the construction of the pruned bridge tree, the representative (component) mapping must be updated to reflect the final structure.

$$\begin{tikzcd}
\boxed{\text{Initial Vertices}} \arrow[r, "\textbf{initM}"] & \boxed{\text{Current Vertices}} \arrow[r, "\textbf{bridgetreeM}"] & \boxed{\text{Bridge Tree Vertices}} \arrow[r, "\textbf{pruningM}"] & \boxed{\text{Pruned Tree Vertices}}
\end{tikzcd}$$

Update of Representative Mapping (Initially Active Vertices Only)

$$\text{For each } v \in V_{\text{active}}^{\text{original}} : \mathit{initM}[v] \leftarrow \mathit{pruningM}\big(\mathit{bridgeTreeM}(\mathit{initM}[v])\big)$$

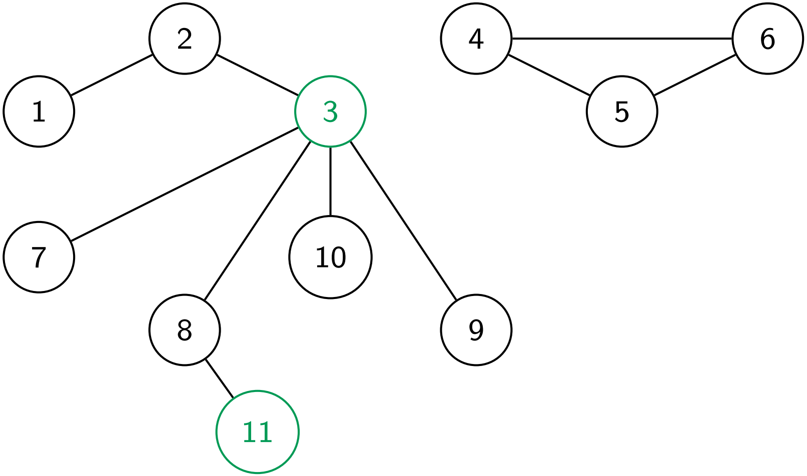
where:

- $V_{\text{active}}^{\text{original}}$ is the set of initially active vertices,
- *initM* maps original vertices to current graph vertices,
- *bridgeTreeM* maps components after bridge tree construction,
- *pruningM* maps components after pruning.

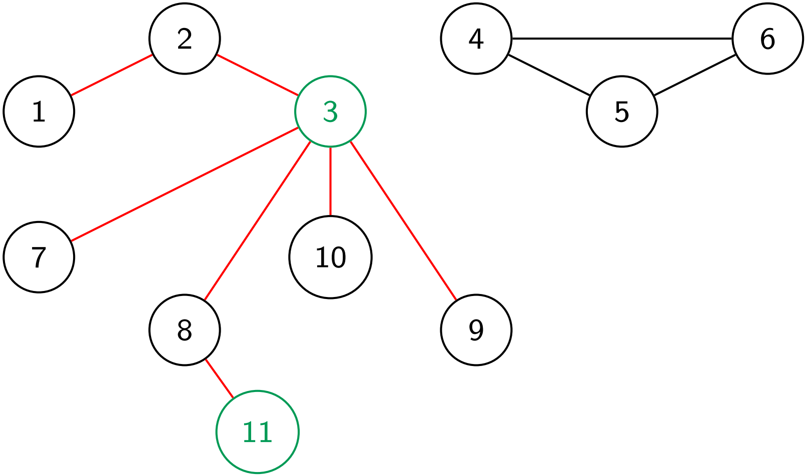
Equivalent Graph - Analysis

- **Time Complexity:** $O(N + M)$
- **Graph Output Size:** $O(K)$, where K denotes the number of active vertices, bounded in the worst case by the number of vertices involved in the current set of operations.

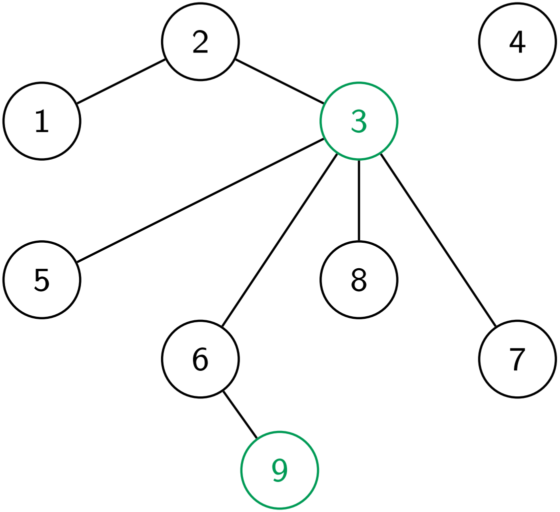
Equivalent Graph- Example (1/9)



Equivalent Graph - Example (2/9)



Equivalent Graph – Example (3/9)

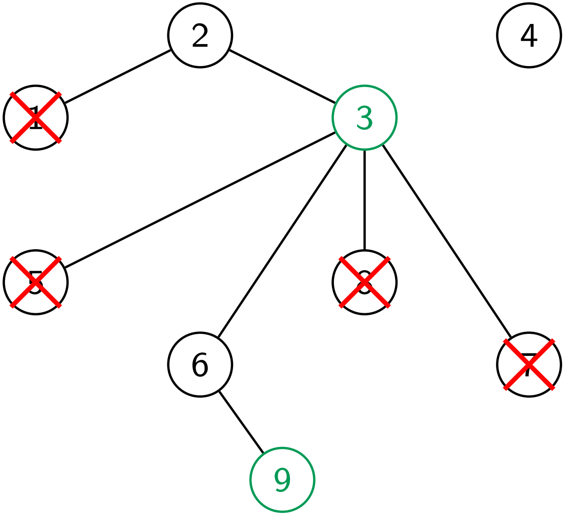


Bridge Component Map for vertices:

Original	**Current**
1	1
2	2
3	3
4	4
5	4
6	4
7	5
8	6
9	7
10	8
11	9

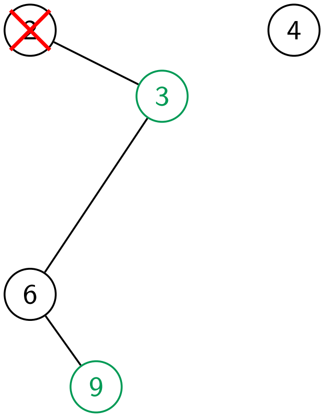
Equivalent Graph - Example (4/9)

Remove inactive vertices with degree 1:



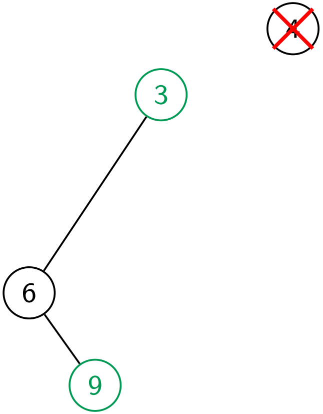
Equivalent Graph - Example (5/9)

Remove inactive vertices with degree 1:



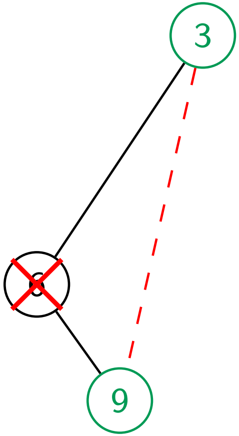
Equivalent Graph - Example (6/9)

Remove inactive vertices with degree 0:

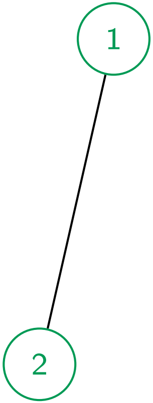


Equivalent Graph - Example (7/9

Remove inactive vertices with degree 2:



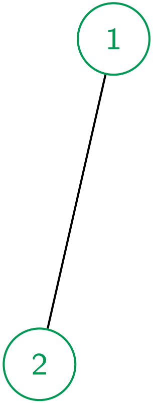
Equivalent Graph – Example (8/9)



All Component Maps for vertices:

Original	**Br.Tree Map**	**Pruned Map**
3	3	1
11	9	2

Equivalent Graph – Example (9/9)



New Mapping for vertices:

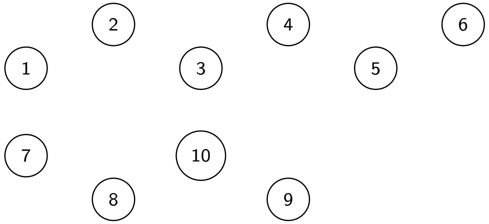
Original	**Current**
3	1
11	2

Optimal Offline Fully Dynamic 2-Edge Connectivity Algorithm

by R. Peng, B. Sandlund, and D. D. Sleator

What is given as Input?

Initial Empty Graph:



Initial (Augmented) Operations List:

1	2	...	11	12	13	14	......
I(1,2, ∞)	I(3,2, 50)	...	I(8,9, ∞)	I(5,6, 53)	D(4,3, 5)	Q(2,1, -)	......

Recursive Algorithm Architecture (1/2)

Two Edge Connectivity Algorithm - Input

- an initially empty graph $G(V, E = \emptyset)$
- an initially identity representatives (components) map:

$$\forall v \in V : \mathrm{comp}[v] = v$$

- a sequence of *operations*, with insertions of initial edges appearing first.

Recursive Algorithm Architecture (2/2)

Two Edge Connectivity Algorithms - Execution Steps

Base Case:

If $|\texttt{operations}| == 1$:

- If operation is a $\texttt{query}(x, y)$, then:

$$\texttt{answer} := \texttt{comp}[x] == \texttt{comp}[y]$$

- Else, it's not a query, `return`.

Steps:

Else $|\texttt{operations}| > 1$:

1. Split the operations interval in half.
2. Identify the **active vertices** within each subinterval.
3. Determine and insert the **permanent edges** for each part into its respective graph.
4. Compute the **equivalent graph**
5. Recursively invoke the procedure with:
    - the equivalent graph,
    - new representatives mapping,
    - the corresponding subinterval of operations.

Time Complexity Analysis — Recurrence Relation

Let t denote the number of operations. The recurrence for the total runtime is given by:

$$T(t) = T\left(\frac{t}{2}\right) + O(t) \Rightarrow T(t) = O(t \log t)$$

This complexity holds under the assumption that at each recursive level:

- The size of the equivalent graph and its representative array remain proportional to t.
- The auxiliary algorithms used for graph processing and sparsification run in linear time with respect to their input size.

***sparsification techniques** are employed to significantly reduce the amount of information required at each recursive call. Through this process, the algorithm achieves the *optimal lower bound* as defined by Patracscu and Demaine [9].

Time Complexity Analysis – Runtime Cases

Let n be the number of initial vertices, and t the number of operations. The algorithm's time complexity depends on the relationship between t and n:

- **Case 1 — Polynomial Regime (Optimal):**
  If $t = O(n^c)$ for some constant c, then the total runtime is:
  $$T(t) = O(t \log n)$$
- **Case 2 — Exponential Regime (Suboptimal):**
  If $t = O(e^n)$, then the runtime becomes:
  $$T(t) = O(t \log t)$$

Remark: In the exponential case, the $O(t \log t)$ runtime is no longer optimal. For comparison, some of the best-known online algorithms addressing this problem achieve runtimes of the form $O(t \cdot \text{polylog}(n))$ [5], which are significantly more efficient in practice.

Time Complexity Analysis — Extended Strategy

Optimized Blocking Strategy

Rather than recursively dividing the operation sequence in half, the operations can be partitioned into blocks of size n^2. Given that no subgraph within any block exceeds $O(n^2)$ in size, each block can be processed independently, resulting in a total runtime of:

$$T(t) = O(t \log n)$$

- This refined strategy enables the algorithm to remain efficient and scalable, even when handling extremely large sequences of operations.
- The implementation of this optimization lies beyond the scope of this thesis. Therefore, in our experimental evaluation, we assume that the number of operations t is polynomially bounded with respect to n.

Brute Force Baseline — Overview

Brute Force Algorithm

The brute-force method handles each operation independently and reconstructs the bridge tree for every query. The procedure involves:

- **Bridge Detection**
- **Bridge Tree Construction**
- **Result Evaluation:** Two vertices are 2-edge-connected if they belong to the same component in the bridge tree.

Operation Time Complexities:

- `insert(u, v)`: $O(1)$
- `delete(u, v)`: $O(N)$
- `query(u, v)`: $O(N + M)$

The overall time complexity for t operations is:

$$T(t) = O(t \cdot (N + M)), \quad \text{empirical input size: } t \approx 10^4$$

Experiments & Results

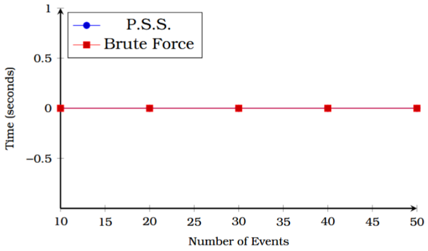
Experiments & Results - Input Generator

- Initial simple graphs generated with no self-loops and multiple edges, satisfying $N < M < \frac{N^2}{2}$ to ensure density.
- Created 7 graphs, each progressively larger than the previous.
- From graph 6 onward, the brute force baseline fails to keep pace with our optimized algorithm.
- For each graph, generated 5 event sets with sizes scaling linearly: the i^{th} event set has size $i \times V$.
- Each experiment consists of one graph combined with its 5 event sets, enabling comparative evaluation of the two algorithms.

Experiments & Results - Diagrams (1/4)

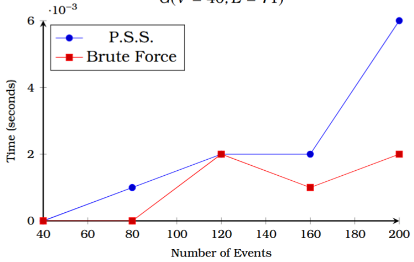
Each experiment consists of one graph combined with its 5 event sets, enabling comparative evaluation of the two algorithms.

$G(V = 10, E = 19)$



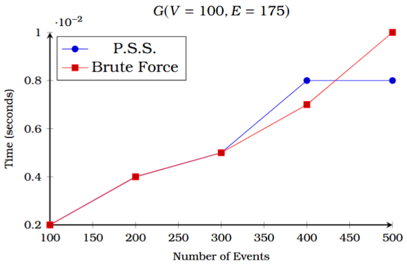
events	P.S.S.	B.F.
10	0	0
20	0	0
30	0	0
40	0	0
50	0	0

$G(V = 40, E = 71)$

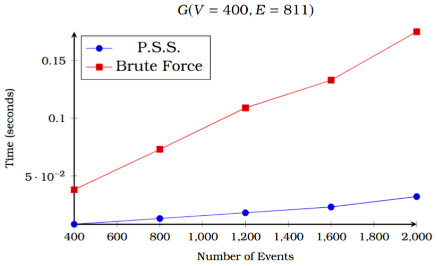


events	P.S.S.	B.F.
40	0	0
80	0.001	0
120	0.002	0.002
160	0.002	0.001
200	0.006	0.002

Experiments & Results - Diagrams (2/4)

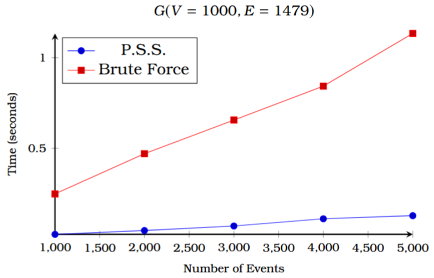


events	P.S.S.	B.F.
100	0.002	0.002
200	0.004	0.004
300	0.005	0.005
400	0.008	0.007
500	0.008	0.01

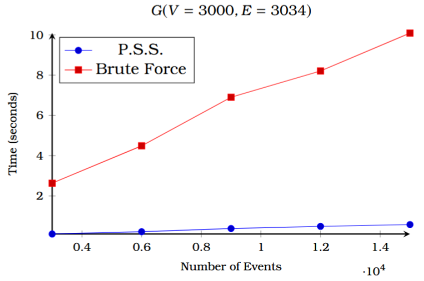


events	P.S.S.	B.F.
400	0.008	0.038
800	0.013	0.073
1200	0.018	0.109
1600	0.023	0.133
2000	0.032	0.175

Experiments & Results - Diagrams (3/4)

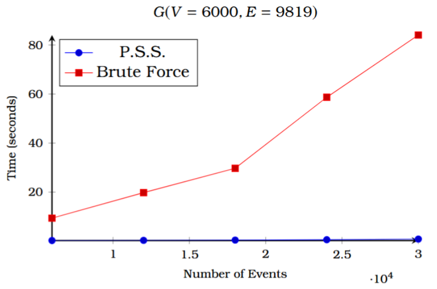


events	P.S.S.	B.F.
1000	0.025	0.248
2000	0.046	0.47
3000	0.071	0.656
4000	0.111	0.843
5000	0.128	1.134

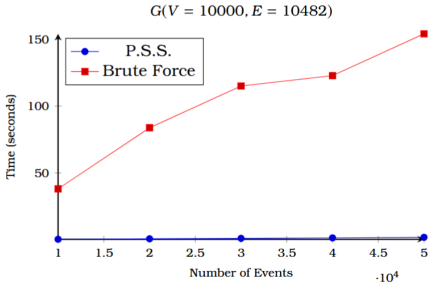


events	P.S.S.	B.F.
3000	0.11	2.633
6000	0.222	4.489
9000	0.38	6.904
12000	0.488	8.208
15000	0.577	10.091

Experiments & Results - Diagrams (4/4)

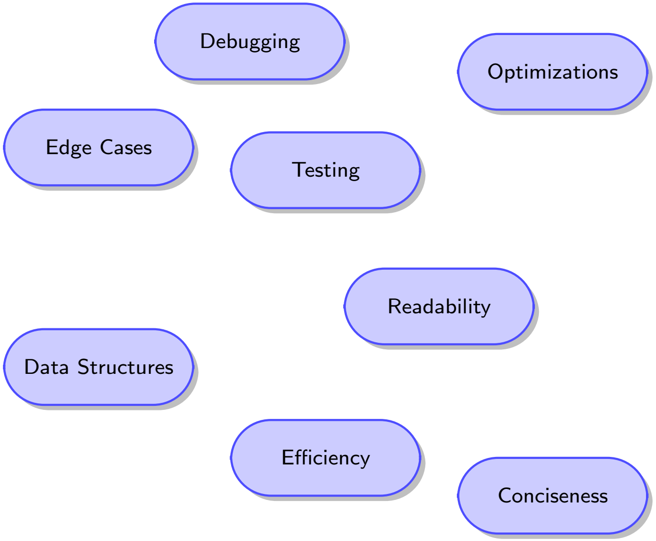


events	P.S.S.	B.F.
6000	0.265	9.377
12000	0.33	19.8
18000	0.441	29.717
24000	0.59	58.688
30000	0.86	84.064



events	P.S.S.	B.F.
10000	0.407	38.09
20000	0.789	83.755
30000	1.113	114.954
40000	1.5	122.731
50000	1.882	153.965

Key Implementation Challenges of the Algorithm



Code Repository

The implementation in C++ is available at:

`https://github.com/dmiw189/offline-2-edge-connectivity`

Possible Changes and Extensions

- **Algorithmic Enhancements:** Further development and implementation can focus on:
  1. Adapting the algorithm to handle multi-edge graphs by slightly modifying the event pre-processing routines. The existing graph processing components remain compatible with such structures.
  2. complete the current algorithm to enhance its performance with implementing its extended version
- **Comprehensive Testing Framework:** Broaden test coverage by incorporating diverse and large-scale datasets, including rigorous stress tests and challenging edge cases.
- **Expanded Applicability:** Adapt the framework to address related connectivity problems, such as:
  - 2-vertex
  - 3-edge
  - 3-vertex
  - potentially 4-edge/vertex

Any questions
or
comments?

Thank you!

References I

David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig.
Sparsification—a technique for speeding up dynamic graph algorithms.
J. ACM, 44(5):669–696, 1997.

David Eppstein et al.
Offline algorithms for dynamic minimum spanning tree problems.
J. Algorithms, 17(2):237–250, 1994.

J. Holm, E. Rotenberg, and C. Wulff-Nilsen.
Faster fully-dynamic minimum spanning forest.
In Nikhil Bansal and Irene Finocchi, editors, *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA 2015)*, volume 9294 of *Lecture Notes in Computer Science*, pages 742–753. Springer, Heidelberg, 2015.
https://doi.org/10.1007/978-3-662-48350-3_62.

Jacob Holm, Wojciech Nadara, Eva Rotenberg, and Marek Sokołowski.
Fully dynamic biconnectivity in $\tilde{O}(\log^2 n) time$, 2025.
https://arxiv.org/abs/2503.21733.

Jacob Holm, Eva Rotenberg, and Mikkel Thorup.
Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time.
In *Symposium on Discrete Algorithms (SODA)*, 2018.

References II

Jakob Holm, Eva Rotenberg, and Mikkel Thorup.
Dynamic bridge-finding in $\tilde{O}(log^2 n)$ amortized time.
In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 35–52. SIAM, 2018.
https://doi.org/10.1137/1.9781611975031.3.

Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie.
Fully dynamic connectivity in $\tilde{O}(logn)$ amortized expected time, 2017.
https://epubs.siam.org/doi/10.1137/1.9781611974782.32.

Adam Karczmarz and Jakub Łacki.
Fast and simple connectivity in graph timelines.
In Frank Dehne, J.-R. Sack, and Ulrike Stege, editors, *Proceedings of the 14th International Symposium on Algorithms and Data Structures (WADS 2015)*, volume 9214 of *Lecture Notes in Computer Science*, pages 458–469. Springer, Cham, 2015.
https://doi.org/10.1145/502090.502095.

Mihai Patrascu and Erik D Demaine.
Lower bounds for dynamic connectivity.
In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing (STOC)*, pages 546–553, 2004.

References III

Richard Peng, Bryce Sandlund, and Daniel D. Sleator.
Optimal offline dynamic 2,3-edge/vertex connectivity, 2019.
https://arxiv.org/abs/1708.03812.