Annales Mathematicae et Informaticae
63 (2026) pp. 88–106
DOI: 10.33039/ami.2026.06.002
URL: https://ami.uni-eszterhazy.hu

Fully dynamic strong connectivity and reachability in digraphs*

Gregory Morse, Tamás Kozsik

Eötvös Loránd University
morse@inf.elte.hu
kto@elte.hu

Abstract. Computing strongly connected components (SCCs) and reachability in directed graphs is fundamental in compilers, static analysis, and many graph algorithms. While efficient offline algorithms are well known, maintaining this information dynamically under both edge insertions and deletions remains challenging.

This paper presents a deterministic fully dynamic algorithm that simultaneously maintains SCCs and reachability in directed graphs. The approach combines a union–find structure for efficient merging of SCCs during edge insertions with localized recomputation of SCCs using Nuutila's algorithm when deletions potentially split a component. Reachability information is maintained at the SCC level and propagated through the condensation DAG.

For a current graph with n vertices and m edges, the resulting algorithm supports $\mathcal{O}(1)$ reachability queries while updates have worst-case complexity $\mathcal{O}(m + n^2)$ due to reachability propagation. Although this does not improve the best known theoretical bounds for specialized dynamic algorithms, the method is simple, deterministic, and well suited to sparse graphs such as control flow graphs (CFGs). Experimental evaluation on random graphs and real program CFGs shows that the algorithm significantly outperforms repeated offline recomputation in practical scenarios.

Keywords: fully dynamic algorithm, strongly connected components, reachability, directed graphs, incremental algorithm, decremental algorithm, graph maintenance

2020 *Mathematics Subject Classification:* 68R10, 68W27

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Submitted: March 31, 2026
Accepted: June 30, 2026
Published online: July 17, 2026

1. Introduction

A fully dynamic algorithm maintains graph information under both edge additions and edge deletions. When a graph is being constructed or modified while queries and decisions about its continued construction are needed, a fully dynamic algorithm can perform better than repeated recomputation for the entire graph.

An unweighted directed graph (digraph) is an ordered pair $G = (V, E)$ with its vertex labels in V and ordered pairs of directed edges in E in the form (x, y) where $x \in V$ is the source or predecessor node and $y \in V$ is the destination or successor. Reachability of vertices based on the transitive closure of the edges can be defined as the existence of any path $s \xrightarrow{*} t$ for two nodes s and t. A strongly connected component (SCC) or region in the graph is defined as a maximal set of nodes S where all nodes are reachable from all other nodes in the set as $\{x \xrightarrow{*} y \| x \in S, y \in S\}$. A non-trivial SCC is one where there is strong connectivity, e.g., $|S| > 1$ or a single node x which has a self-loop and thus reach itself given $(x, x) \in E$. This leaves single node components without self-loops as trivial so a node x where $(x, x) \notin E \wedge \nexists y \in V(x \xrightarrow{*} y \wedge y \xrightarrow{*} x)$. If every SCC is collapsed into a single representative node with only its inter-SCC edges remaining, then the resulting graph is a directed acyclic graph (DAG). Any element of an SCC can be selected to be a root node for the SCC, though classically the root is chosen as the first node of an SCC which was visited during a depth first traversal of the graph. For convenience, we say the current number of vertices and edges after any update are $|V| = n$ and $|E| = m$ respectively. It is also easy to see the relationship between the number of edges and vertices as $m \leq n^2$.

Our focus is on deterministic algorithms that provide exact reachability information and constant-time $\mathcal{O}(1)$ queries. Although the worst-case amortized time complexity of the algorithm can seem less than ideal, in practice, the performance is sufficient when dealing with sparse graphs that contain sufficient strong connectivity. Furthermore, optimized variants which use batching so that a group of edges may be either added or deleted simultaneously can often significantly improve update throughput for fully dynamic algorithms. It should be noted that it is trivially shown that the more dense a graph becomes, then SCCs in the graph become more or larger, so the algorithm that will be presented would become very efficient in such types of graphs. However, in the usual contexts where SCCs are computed, these graphs are often control-flow graphs (CFGs) and tend to be sparse. SCCs have also been used as abstraction units in logical encodings, where SCC structure influences properties of SAT representations [14]. Regardless, for dense graphs, it is well known that SCCs are a performant method to compute transitive closure.

Contributions. This paper presents a deterministic fully dynamic algorithm that simultaneously maintains SCCs and all-pairs reachability in directed graphs. The algorithm uses union–find structures to efficiently merge SCCs during edge insertions and recomputes components locally during deletions using Nuutila's algorithm. Reachability information is propagated through the SCC DAG, yielding $\mathcal{O}(1)$ query time and worst-case $\mathcal{O}(m + n^2)$ update complexity.

2. Related work

Tarjan's algorithm is one of the most well-known for computing the SCCs of a digraph [23]. This algorithm uses two interleaved traversals of the graph to identify components as well as a representative node considered to be its entry. The first traversal is a depth first search (DFS) and the second is based on a stack which stores potential component elements that are removed when the component is identified. The time complexity is $\mathcal{O}(m+n)$, the same as for DFS, while the storage complexity is $n(2+5w)$ where w is the machine's word size.

Nuutila and Soisalon-Soininen modified Tarjan's algorithm to yield two new algorithms which reduce the number of nodes stored on the stack and thus a space complexity improvement to $n(1+4w)$ [17]. The first algorithm does not store component root nodes, while the second one stores only possible root nodes for non-trivial components. The interesting aspect of the second algorithm is that it leads to a simple extension to yield transitive closure information found with more detail in Nuutila's later paper [16]. However the addition of transitive closure has the worst case same complexity of its space and time which is $\mathcal{O}(n^2)$. Purdom had previously shown a similar method to compute SCCs and then perform topological sort of the SCC nodes and compute transitive closure based on the simplification provided by the SCC induced DAG to also run in $\mathcal{O}(n^2)$ [19]. The primary difference is that Purdom's algorithm computes the SCCs in advance and did not make use of a stack thus requiring an extra topological sort. Pearce improved the space efficiency of Tarjan's algorithm further to $n(1+3w)$ in [18].

More recently, Haeupler et al. show how to incrementally maintain topological ordering and SCCs in $\mathcal{O}(m^{3/2})$ time using bidirectional search in [8] with the Union-Find data structure. However, though decrementally maintaining topological ordering is possible under their algorithm, they merely discuss that decremental SCCs are a harder problem. Georgiadis et al. give efficient data structures for solving the incremental SCCs case in [6] by maintaining dominators and what they called the hyperloop nesting forest as it is based on the dominators not the depth first spanning tree. Georgiadis et al. also showed a way to compute SCCs upon potential deletion of every edge using the concept of classifying an edge as a strong bridge with $\mathcal{O}(n)$ time complexity in [7]. A strong bridge is an edge whose deletion would increase the number of SCCs. Bernstein et al. show a method to solve only the decremental SCC algorithm in $\mathcal{O}(m)$ time using ES-trees in [1]. This was improving on the previous best known decremental time of $\mathcal{O}(m\sqrt{n})$ in [2]. Karczmarz and Smulewicz give a randomized fully dynamic SCC data structure with conditionally tight $\mathcal{O}(n^{1.529})$ worst-case update time, and also reduce fully dynamic strong connectivity to dynamic reachability [12]. These specialized algorithms give stronger asymptotic bounds for individual dynamic SCC or reachability variants, while the algorithm here emphasizes a simple deterministic structure that maintains SCCs and all-pairs reachability together.

This paper is thereby motivated primarily to analyze not merely a fully dynamic algorithm for strong connectivity or reachability in their isolated sense, but

the simultaneous maintenance of both. Because of their relationship purely based on their mathematical definitions, it makes sense to do this. It should be noted that a DFS-based approach to their maintenance should also be considered and due to advances in algorithms for fully dynamic maintenance of the DFS-tree and its intervals [26], an ideal algorithm could likely look at the exact changes that occurred to the DFS-tree to recompute SCCs very efficiently. This would not only be an elegant algorithm that could be extended to loop nesting forest maintenance but could replace the use of the more expensive maintenance of all-pairs transitive closure. However, there are situations where reachability information is also needed. Thus the fact that SCCs can significantly reduce the amount of storage needed to make reachability queries makes their combination in dynamic algorithms appropriate.

As for transitive closure, the Floyd-Warshall algorithm is the classic algorithm for computing all-pairs shortest paths in a digraph however it is easily adapted to transitive closure but it has $\mathcal{O}(n^3)$ time complexity and $\mathcal{O}(n^2)$ space complexity due to using a matrix in [4]. This will be the worst-case space complexity for the reachability structure used in this paper when every vertex is an SCC. Since the cubic computation in Floyd-Warshall becomes impractically slow very quickly as the number of vertices of the graph increases, a better algorithm is the well known utilization of the DFS or breadth-first search (BFS) starting with every node in the graph which gives a time complexity of $\mathcal{O}(n^2 + nm)$ reducing it to quadratic. For decremental transitive closure maintenance, Even and Shiloach provided a single source $\mathcal{O}(mn)$ algorithm in [22] which uses a BFS structure maintaining depth information sometimes aptly referred to as an ES-tree to efficiently store the information and allow updates. It should be noted that maintaining a BFS tree in a fully dynamic setting is still an open problem and a semi-dynamic algorithm which works in both an isolated incremental or decremental setting is the best that is known [5]. In fact, not maintaining the BFS tree but only the depth or level information in a decremental setting is more straight-forward. However, edge addition still requires a BFS recomputation starting from the target vertex. Furthermore, this data structure solves the all-pairs shortest paths problem and transitive closure is, in fact, a subset of the information provided by shortest paths. It is also used in state-of-the-art decremental SCC as well as single source reachability (SSR) algorithms. Henzinger et al. reduce the time complexity for the single source problem to being sublinear but achieves it with randomization [9]. ES-trees will not be used in this fully dynamic context as no known efficient computation can be done incrementally.

As for fully dynamic transitive closure with $\mathcal{O}(1)$ queries, King presented an $\mathcal{O}(n^2 \log n)$ algorithm in [13]. Demetrescu and Italiano presented a quadratic $\mathcal{O}(n^2)$ algorithm in [3] using matrices with initialization time of $\mathcal{O}(n^3)$. Sankowski improved the $\mathcal{O}(n^2)$ to being worst-case in [21]. Roditty improves the initialization time to $\mathcal{O}(n^2)$ in [20].

It is thus understood that Union-Find has been the basis for all efficient incremental algorithms while ES-trees have been the basis for efficient decremental algorithms regarding SCCs and reachability.

As for connectivity of the digraph, it is not required for SCCs or transitive closure but for simplicity it can also be assumed a virtual root node of the graph which connects implicitly to all nodes is present, thereby reaching all nodes and being a trivial SCC wlog.

3. Algorithm

The Union-Find disjoint set data structure with the usual path compression but rank specified by order of arguments to the union operation will be used as it has been in many similar algorithms due to its efficiency in collapsing sub-graphs which is precisely what will be done with SCCs. However, it is reinitialized for vertices of an SCC when a deletion potentially splits the component; the new SCCs are recomputed using Nuutila's algorithm on the affected subgraph. This operation, referred to as `unmerge`, will require as input a complete disjoint set, and will effectively divide them back into individual set elements. This is needed when a SCC will be removed. As long as the entire set in the tree is passed, this is a very simple operation as it merely resets the parent of all entries in the given set to themselves. Obviously, this improves the time complexity of the incremental case at the expense of the decremental one. The simple justification for such a choice is that in compilers and static analysis, edge addition tends to be much more common than edge deletion, due to a typical initial construction of the CFG. However, certain refactorings and optimizations could still cause significant edge deletions.

Self-loops are considered as though they do not affect SCCs, they do change reachability but only in that very limited aspect. Other trivial cases are when an edge is added between two nodes in an SCC, or when an edge is removed between two nodes in an SCC without affecting the SCC. However, detecting that an edge removal does not affect the SCC requires an offline reachability check. Therefore, strictly adapting the offline algorithm to account for the trivial cases will not be particularly beneficial.

Four update cases are considered: (i) insertion merging SCCs, (ii) insertion without merge, (iii) deletion splitting an SCC, (iv) deletion without SCC change. In all of these cases, the reachability information will need to be updated by propagation. Only the predecessor adjacency lists for the graph are necessary for our part of the algorithm as the successor information is only used in Nuutila's algorithm for forward reachability.

When an edge addition induces a merging of SCCs, it merely requires a query that the destination edge reaching the source edge. By traversing all reachable SCCs of the destination node's SCC which can also reach the source, the new component can be readily identified and constructed. This will not lead to root node of the SCC being the same one that might be identified in an offline algorithm as by using reachability, because the DFS ordering is not maintained. However, the most difficult case is an edge removal within an SCC. The data structures do not contain any information to determine whether the SCC has remained intact, but if it has

not, the reachability information does not help break it into its smaller components with their new reachability information. Therefore, Nuutila's algorithm can be applied on the sub-graph of the SCC which had an edge removal. If the changes in the depth first spanning tree or the loop nesting forest information were available, a more detailed decremental strategy could be employed. In fact, Nuutila's algorithm identifies SCC root nodes according to the DFS order it computes; because it does not know the original DFS order for the sub-graph, these roots may still differ from those typically used in an offline algorithm. This is considered unimportant and the maintenance of DFS though possible would provide only this marginal benefit in the algorithm and certainly add unneeded complexity as maintaining entries of SCCs and deciding when the entry node DFS order has changed would further need to be managed.

The reachability information in the incremental case is propagated backwards through the DAG of SCCs by adding the newly reachable nodes whether the edge addition created a new SCC or merely added a path between SCCs. In the decremental case, the reachability is also backward propagated, however, before removal, reachability must be constructed for all nodes which reach the target node to make sure there is not another pathway to the source node and nodes which reach the source node that potentially need to be removed. Additionally it can only be done by visiting all predecessor nodes in an order such that all successors to a given predecessor have already been processed. The DFS reverse postordering or topological sort is well known to satisfy this property and is trivial to prove since the graph formed by SCCs is acyclic. Other methods to do the propagation would use successor reaching sets to directly rebuild the reaching set or compute the necessary set subtractions, however this leads to an $O(n^3)$ algorithm and would also be slower in practice.

The algorithm uses three data structures, summarized in Table 1. Successor reachability information is maintained only as this is sufficient for any query regarding any edge in a digraph.

Table 1. Maintained data structures.

Structure	Meaning
lp	Union-Find forest with unmerge capability and one tree for each SCC.
scc	Dictionary from SCC root elements to all elements in the SCC.
reach	Dictionary from SCC root elements to all vertices reached from that SCC.

These data structures allow $\mathcal{O}(1)$ queries for locating the SCC a given node x is part of (lp.`find`(x)), enumeration of all nodes in a given SCC for node x (scc[lp.`find`(x)]), as well as "x reaches y" (lp.`find`(y) $\in$ reach[lp.`find`(x)]) or "x reaches" queries (reach[lp.`find`(x)]).

Finally a few convenience dictionaries and functions are given:

- `nuutilaReachSCC` implements Nuutila's algorithm, returns SCCs and reachability information, and is used to recompute SCCs for sub-graphs. The Nuutila implementation is expected to be run only if the reachability between the two nodes in the component actually changed, for practical reasons.

- **pred** returns predecessors of a node such that $\text{pred}[v] \equiv \{x \| (x,y) \in E, y = v\}$, while **succ** returns successors such that $\text{succ}[v] \equiv \{y \| (x,y) \in E, x = v\}$. These are typically stored in arrays with adjacency lists or a matrix.

- `subGraph` returns the sub-graph of nodes and all edges among them:

$$\texttt{subGraph}(S) \equiv (S, \{(x,y) \mid (x,y) \in E, x \in S \wedge y \in S\}).$$

Algorithms 1 and 2 take a graph whose edge (x,y) was just added or removed respectively and update the SCC and reachability information based on that edge. In both cases, self-loops are dealt with first, then skipping of cases which have no effects, followed by merging or collapsing the SCCs if needed, and finally propagation of reachability information.

Algorithm 1 Incremental SCC and reachability update (AddEdgeReachSCC)

Require: Added edge (x,y) with source x and target y
Data: Union–Find lp with `find`/`union`; maps scc, reach, pred
if $x = y$ **then** ▷ self-loop
  if $x \in \text{scc}$ **then** $\text{reach}[x] \leftarrow \text{reach}[x] \cup \{y\}$
  return
$x_c \leftarrow \text{lp.}\texttt{find}(x)$; $y_c \leftarrow \text{lp.}\texttt{find}(y)$
if $x_c = y_c$ **then return** ▷ no SCC change
if $x_c \in \text{reach}[y_c]$ **then** ▷ new SCC discovered
  $C \leftarrow \text{scc}[x_c]$; $R \leftarrow \text{reach}[x_c] \cup \{x_c\}$
  for all $v \in \{\, \text{lp.}\texttt{find}(z) \mid z \in \text{reach}[y_c] \cup \{y_c\} \,\}$ **do**
    if $v \neq x_c \ \wedge \ x_c \in \text{reach}[v]$ **then** ▷ member of new SCC
      $C \leftarrow C \cup \text{scc}[v]$; $R \leftarrow R \cup \text{reach}[v] \cup \{v\}$
      $\text{lp.}\texttt{union}(x_c, v)$; $\text{scc.}\texttt{delete}(v)$; $\text{reach.}\texttt{delete}(v)$
  $\text{scc}[x_c] \leftarrow C$; $\text{reach}[x_c] \leftarrow R$; $y_c \leftarrow x_c$
$P \leftarrow \{x_c\}$; $V \leftarrow \emptyset$; $R \leftarrow \text{reach}[y_c] \cup \{y_c\}$ ▷ back-propagate reachability
while $P \neq \emptyset$ **do**
  remove some v from P
  $\text{reach}[v] \leftarrow \text{reach}[v] \cup R$; $V \leftarrow V \cup \{v\}$
  $P \leftarrow P \cup \{\, \text{lp.}\texttt{find}(z) \mid \exists\, w \in \text{scc}[v] \colon \ z \in \text{pred}[w] \ \wedge \ z \notin \text{scc}[v] \ \wedge \ z \notin V \,\}$

An example illustrating the algorithm is provided in Appendix A. Extensions to support batched edge updates are discussed in Appendix B.

Algorithm 2 Decremental SCC and reachability update (RemoveEdgeReachSCC)

Require: Removed edge (x, y) with source x and target y
Data: maps scc, reach, pred, succ; Union-Find lp; `nuutilaReachSCC`, `subGraph`

1: **if** $x = y$ **then** ▷ self-loop removal
2: **if** $x \in \text{scc} \ \wedge \ \{x\} = \text{scc}[x]$ **then** $\text{reach}[x] \leftarrow \text{reach}[x] \setminus \{y\}$
3: **return**
4: $x_c \leftarrow \text{lp.}\texttt{find}(x)$; $y_c \leftarrow \text{lp.}\texttt{find}(y)$
5: **if** $x_c = y_c$ **then** ▷ rebuild sub-components
6: $S \leftarrow \text{scc}[x_c]$
7: $(\text{subscc}, \text{subreach}) \leftarrow \texttt{nuutilaReachSCC}(\texttt{subGraph}(S))$
8: **if** $|\text{subscc}| = 1$ **then return** ▷ SCC did not change
9: $R \leftarrow \text{reach}[x_c] \cup \{x_c\}$
10: $\text{lp.}\texttt{unmerge}(S)$; $\text{scc.}\texttt{delete}(x_c)$; $\text{reach.}\texttt{delete}(x_c)$
11: $Q \leftarrow \{\, x \mid x \in \text{subscc} \,\}$; $P \leftarrow \emptyset$
12: **while** $Q \neq \emptyset$ **do** ▷ rebuild, propagate subgraph component reach
13: $v \leftarrow$ any of $\{\, w \mid w \in P,\ (\text{reach}[w] \setminus \{w\}) \cap Q = \emptyset \,\}$
14: $\text{scc}[v] \leftarrow \text{subscc}[v]$
15: **for all** $w \in \text{subscc}[v] \setminus \{v\}$ **do** $\text{lp.}\texttt{union}(v, w)$
16: $\text{reach}[v] \leftarrow \text{subreach}[v] \cup \bigcup_{c \in \{\text{lp.}\texttt{find}(z) \mid w \in \text{scc}[v],\ z \in \text{succ}[w]\},\, c \notin S} (\text{reach}[c] \cup \{c\})$
17: $S \leftarrow S \setminus \text{subscc}[v]$; $Q \leftarrow Q \setminus \{v\}$
18: **else**
19: $P \leftarrow \{x_c\}$; $R \leftarrow \text{reach}[y_c] \cup \{y_c\}$
20: $V \leftarrow \emptyset$ ▷ back-propagate reachability with successor checking
21: **while** $P \neq \emptyset$ **do**
22: $v \leftarrow$ any of $\{\, w \mid w \in P,\ (\text{reach}[w] \setminus \{w\}) \cap P = \emptyset \,\}$
23: $\text{reach}[v] \leftarrow \text{reach}[v] \setminus \Big(R \setminus \bigcup_{w \in \text{scc}[v], c \in \{\text{lp.}\texttt{find}(z) \mid z \in \text{succ}[w]\}, c \neq v} (\text{reach}[c] \cup \{c\})\Big)$
24: $P \leftarrow (P \setminus \{v\}) \cup \{\, \text{lp.}\texttt{find}(z) \mid w \in \text{scc}[v],\ z \in \text{pred}[w],\ z \notin \text{scc}[v] \wedge z \notin V \,\}$
25: $V \leftarrow V \cup \{v\}$

4. Correctness and complexity

Firstly self-loops are dealt with as trivial cases. Edge addition which induces a self-loop to a node in a non-trivial component requires no update. For a trivial node, it merely is able to reach itself without having any other effects. Similarly upon removal of a self-loop edge in a non-trivial component causes no changes. In a non-trivial single node component, it merely removes its ability to reach itself. The proof of this is straight-forward from the definition of reachability in SCCs. Self-loop handling is obviously $\mathcal{O}(1)$ for both the incremental and decremental case.

The reachability information always represents a DAG given that any cycle has

been collapsed to a single node via SCCs. Both the strong connectivity and the transitive closure part of the algorithms involves straight-forward formal proofs that can be expressed by the transitive relationship between the edges. The propagation order being correct is proven by the fact that the DFS provides a topological ordering for a DAG [25]. The proofs are straight-forward based on these well-known properties and omitted for brevity. Graph nodes are visited only once when computing SCC adjustments and SCC root nodes are visited a maximum of one time when propagating reachability information in the incremental algorithm. The decremental or batching algorithm can require two visits when determining the new reachability information before adjusting the affected nodes.

Any Union-Find data structure operation takes $\mathcal{O}(m\alpha(n))$ where m here refers to the number of union or find operations and α is the inverse Ackermann function which makes any operation essentially $\mathcal{O}(1)$ [24].

In the incremental case, merging SCCs requires $\mathcal{O}(n)$ for enumerating and combining the SCCs where n is the number of SCCs being merged and worst case $\mathcal{O}(n^2)$ for the reachability set union operations. While for the final step of propagating reachability, the complexity is $\mathcal{O}(m+n)$ for the predecessor traversal and worst case $\mathcal{O}(n^2)$ again for set union operations. This means that given all considerations, the worst case total update time is $\mathcal{O}(m+n^2)$. The algorithm will be expected to perform favorably when edge insertions are expected to be frequent as there is no fixed $\mathcal{O}(m+n)$ DFS traversal.

As for decremental case, determining if an SCC needs to be decomposed requires the worst case of DFS, e.g., $\mathcal{O}(m+n)$ operations. When decomposing an SCC into two or more new SCCs, it requires the time complexity of Nuutila's algorithm. Afterwards, it takes $\mathcal{O}(n)$ time where n in this case is the number of vertices in the SCC being decomposed and the expected worst case $\mathcal{O}(n^2)$ union operations when building the reachFrom dictionary as well as $\mathcal{O}(n^2)$ for finally doing the set subtraction operations. As mentioned Nuutila's algorithm runs only on the sub-graph of the SCC being decomposed. This could be substituted for Tarjan's algorithm as the reachability information obtained only provides an optimization, but there will be a significant performance penalty. Naturally, there are practical time issues due to the set union and subtraction operations used for the reachability in our proposed algorithm and the implementation details of Nuutila. Instead of checking Nuutila to determine that only a single SCC still exists, running a $\mathcal{O}(m+n)$ topological search with early termination is considered an important optimization especially as the graph grows more dense. Nuutila's algorithm with a single SCC has the same complexity but it cannot terminate early and due to all the data structures and reachability construction, it increases the time from doing a topological search by some constant factor of n depending on implementation details. Finally, the reachability propagation in the decremental case is $\mathcal{O}(m+n)$ to traverse the predecessors and compute their DFS postorder. Then enumerating them requires $\mathcal{O}(n)$ yet since we must rebuild the reachability information, it takes worst case $\mathcal{O}(m+n^2)$ to traverse all the predecessors and make appropriate set subtraction operations. This still makes the worst case total update time $\mathcal{O}(m+n^2)$.

This is why it can be worth checking reachability with a DFS before doing the computation, perhaps based on a fixed heuristic based on the density of the graph, or number or size of the components.

When considering the set operations used for reachability, the complexity is mostly dependent on the cost of set union and intersection operations. The classical algorithms and the proposed algorithm are generally intended for sparse graphs. However, dense graphs tend to have more and larger SCCs so in practice it will be very efficient in this context as well. In fact, as far as the SCCs are concerned, the incremental algorithm is reasonably efficient, as we perform only a single propagation, and especially for dense graphs given that m is bounded by n^2 as only unweighted graphs need to be considered. However, the use of Nuutila's algorithm and reachability propagation is the primary drawback. Even though reachability allows for easy incremental computation of SCCs, its own maintenance requires a backwards traversal of the SCC DAG with set union operations in the incremental case and both union and subtraction operations in the decremental case. The worst case will appear when adding an edge from a node which is reached by half of nodes to a node which reaches the other half of the nodes requiring $\frac{n^2}{4}$ set union operations. Deleting that same edge requires $\frac{n^2}{4}$ set subtraction operations.

The worst-case space complexity is $\mathcal{O}(n)$ for the Union-Find and SCC dictionary, while being $\mathcal{O}(n^2)$ for the reachability information. Although for dense graphs, matrices are of course preferred over adjacency lists or sets for storing predecessors and successors as well as reachability, when SCCs are used, in fact, the proposed dictionary would be better for maintaining reachability information to avoid complications from re-labeling. The worst case storage and time complexity thereby occurs in a DAG where given the nodes labeled in ascending numerical order, each node connects to every other node with a larger label.

The algorithm is less effective on nearly acyclic graphs where the problem reduces to transitive closure. It performs best when SCCs are large or numerous, which is typical for control flow graphs. The performance of the algorithm depends on a lot of implementation details, for example, such as that sets or dictionaries might be implemented with hash tables or balanced trees yielding $\mathcal{O}(1)$ vs. $\mathcal{O}(\log n)$ time complexity for additions or look-ups respectively. The time complexity for set union and subtraction operations also could be considered as operations such as simple set operation equivalences e.g., when A is a set and B is a set of sets allows $A - \bigcup_{S \in B} (S) \equiv A - \bigcup_{S \in B} (A \cap S)$ to reduce sizes for union operations.

There is also the idea of maintaining the reverse reachability information so instead of fast lookup for "x reaches" queries, fast look-ups could also occur for "x is reached by" queries. There is insignificant additional time complexity to maintaining the reverse data structure, it is simply updated any time reach is however since the dictionary is maintained as a mapping of components to all reachable nodes and not components to reachable components, some simple reduction of nodes to components and expansion from components to nodes is needed giving nonetheless worst case $\mathcal{O}(n)$.

It should be noted that the worst-case complexity of many fully dynamic algo-

rithms will often be similar to the average-case of the offline algorithm. In other words, any online algorithm should never be worse than linear in terms of nodes and edges for SCC maintenance or quadratic relative to the nodes for transitive closure. Constructing scenarios where the worst case occurs generally relies on pathological cases though and does not actually happen in practice, e.g., a complete DAG whose vertices are totally ordered and whose $\frac{n(n-1)}{2}$ edges all go from earlier to later vertices. The initial construction of a large CFG would still ideally be done via an offline algorithm in the compilation context because of syntax and semantic guarantees of programming languages, however, in the context of decompilation where self-modifying or unreachable code is an issue, the penalty of incremental construction would be a necessity. The average-case complexity is more useful however it depends on a lot of averages of graph properties, such as the average number of successors or predecessors, the graph density, the average size of SCCs, the number of SCCs in proportion to the number of nodes, etc.

5. Experimental results

The testing program was written in carefully optimized Python 3 code using dictionary and set support from the language and executed on a Intel i7-9750H CPU. This is compared with an optimized Nuutila implementation which supports self-loops and also uses the suggested component stack duplicate and sorting modifications. The online and offline algorithms were both modified to utilize a stack data structure and avoid use of recursion which due to interpreter stack depth limitations would limit the number of vertices to around 1000 otherwise. The public Git repository contains the Python 3 implementation, including sccreach.py, the test.py benchmarking harness, CFG extraction support such as gengraphs.idc, benchmark inputs in cfgs/, and rerun instructions [15]. The released CFG inputs are intended to make the reported experiments rerunnable; exact package versions and Windows build numbers for the original binaries were not recorded, and proprietary disassembly has been removed from the public sanitized data.

We first tested on random graphs and these results, provided in Appendix C, are not included for brevity yet they reflect the realistic experiments. We then tested these results by using realistic CFGs extracted with Hex-Rays IDA Freeware 7.0. IDA provides IDC scripting and can export function flow graphs in textual Graph Description Language (GDL) format, which was used by a short extraction script before parsing the graph edges [10, 11]. The corpus contains Linux x86-64 ELF binaries sendmail (1,599 functions), smbd (799), and vsftpd (705), and Windows 10 x64 PE binaries explorer.exe (11,703), kernel32.dll (2,533), and user32.dll (2,651). The private extraction artifacts contain disassembly, so the public repository redistributes only sanitized graph topology and labels sufficient to rerun the benchmark. The results appear in Figure 1.

The results exactly confirm the suspicion that if the graphs have a lot of non-trivial components or large components as in the first experiment, the incremental algorithm will perform exceptionally well. In fact, it is only about a constant a_1

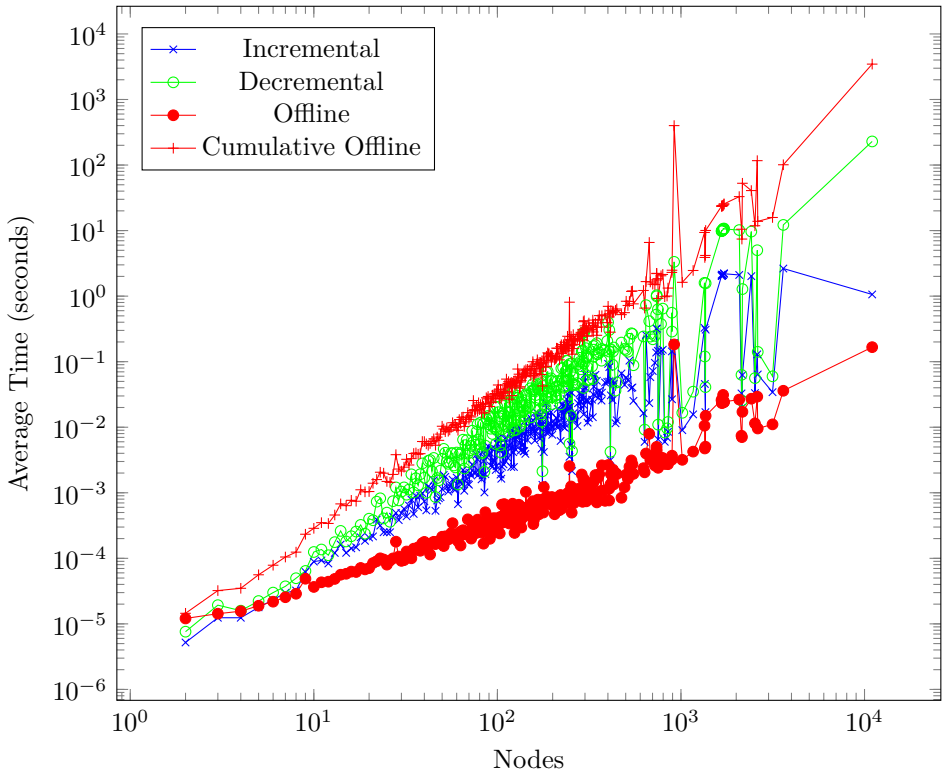

Figure 1. Real Control Flow Graph Edge Addition and Edge Deletion Execution Time.

time slower than doing a single offline computation after all the edges are added. In this context, the deletion does not perform as well but it is only a factor slower based on the number of nodes, not edges or roughly $a_2 n$ times slower than a single offline computation where a_2 is a constant. On the other hand, with the locality and average out-degree causing fewer non-trivial SCCs and closer to a DAG, the quadratic propagation of reachability information becomes the predominant factor. However, realistic CFGs tended to perform more closely to the locality model than the first model, largely because CFGs tend to be far more sparse than the first model, and they are constructed in a way that by the time the SCC is identified, the inter-component edges would have already been added. Unsurprisingly in all the cases, both algorithms outperform cumulative offline computation, the decremental was always slower than the incremental as by design, and the incremental was slower than a single offline computation.

6. Conclusion

This represents only a step towards finding a highly efficient and deterministic fully dynamic algorithm in this area. The challenge of finding the right data structure has left it open so that although worst-case bounds are known for such an algorithm, there is more to be done to make this practical for very large graphs. Specifically a better data structure to maximize time and space efficiency of this algorithm is needed.

As well, most dynamic graph algorithms focus on maintaining only a single type of graph information, often only incrementally or decrementally based on very specialized data structures. This paper has explored the opportunity to combine maintenance of two types of graph information simultaneously where both strong connectivity and reachability are used to lower the complexity and assist with computing the other.

In the future, more complex fully dynamic algorithms which can maintain even more simultaneous types of information are to be expected. This would be a potential basis for a shift in how compilers are constructed if the control flow graph never required explicit re-computation.

Furthermore, fully dynamic algorithms which can compute loop nesting forests for reducible and irreducible graphs are a logical extension to SCC maintenance though requiring techniques based on DFS, dominators or a recursive application of SCCs. Lastly, incremental algorithms for maintaining reducible graphs based on irreducible graphs or single-entry single-exit regions in graphs from arbitrary graphs would be a step forward for representing arbitrary code in high-level programming languages and would benefit from already having efficient reachability queries.

References

[1] A. Bernstein, M. Probst, C. Wulff-Nilsen: *Decremental Strongly-Connected Components and Single-Source Reachability in near-Linear Time*, in: Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 365–376, isbn: 9781450367059, doi: 10.1145/3313276.3316335.

[2] S. Chechik, T. D. Hansen, G. F. Italiano, J. Łącki, N. Parotsidis: *Decremental Single-Source Reachability and Strongly Connected Components in $\tilde{O}(m\sqrt{n})$ Total Update Time*, in: 2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS), 2016, pp. 315–324, doi: 10.1109/FOCS.2016.42.

[3] C. Demetrescu, G. F. Italiano: *Fully dynamic transitive closure: breaking through the $O(n^2)$ barrier*, in: Proceedings 41st Annual Symposium on Foundations of Computer Science, 2000, pp. 381–389, doi: 10.1109/SFCS.2000.892126.

[4] R. W. Floyd: *Algorithm 97: Shortest Path*, Commun. ACM 5.6 (June 1962), p. 345, issn: 0001-0782, doi: 10.1145/367766.368168.

[5] P. G. Franciosa, D. Frigioni, R. Giaccio: *Semi-dynamic breadth-first search in digraphs*, Theoretical Computer Science 250.1 (2001), pp. 201–217, issn: 0304-3975, doi: 10.1016/S0304-3975(99)00132-2.

[6] L. Georgiadis, G. F. Italiano, N. Parotsidis: *Incremental Strong Connectivity and 2-Connectivity in Directed Graphs*, in: LATIN 2018: Theoretical Informatics, ed. by M. A. Bender, M. Farach-Colton, M. A. Mosteiro, Cham: Springer International Publishing, 2018, pp. 529–543, isbn: 978-3-319-77404-6, doi: 10.1007/978-3-319-77404-6_39.

[7] L. Georgiadis, G. F. Italiano, N. Parotsidis: *Strong Connectivity in Directed Graphs under Failures, with Applications*, in: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '17, Barcelona, Spain: Society for Industrial and Applied Mathematics, 2017, pp. 1880–1899, doi: 10.1137/1.9781611974782.123.

[8] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, R. E. Tarjan: *Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance*, ACM Trans. Algorithms 8.1 (Jan. 2012), issn: 1549-6325, doi: 10.1145/2071379.2071382.

[9] M. Henzinger, S. Krinninger, D. Nanongkai: *Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs*, in: Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing, STOC '14, New York, New York: Association for Computing Machinery, 2014, pp. 674–683, isbn: 9781450327107, doi: 10.1145/2591796.2591869.

[10] Hex-Rays: *Announcing version 7.6 for IDA Freeware*, url: https://hex-rays.com/blog/announcing-version-7-6-for-ida-freeware (visited on 06/09/2026).

[11] Hex-Rays: *gen_flow_graph*, url: https://docs.hex-rays.com/9.0/developer-guide/idc/idc-api-reference/alphabetical-list-of-idc-functions/1253 (visited on 06/09/2026).

[12] A. Karczmarz, M. Smulewicz: *On Fully Dynamic Strongly Connected Components*, in: 31st Annual European Symposium on Algorithms (ESA 2023), ed. by I. L. Gørtz, M. Farach-Colton, S. J. Puglisi, G. Herman, vol. 274, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 68:1–68:15, isbn: 978-3-95977-295-2, doi: 10.4230/LIPIcs.ESA.2023.68.

[13] V. King: *Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs*, in: 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039), 1999, pp. 81–89, doi: 10.1109/SFFCS.1999.814580.

[14] G. Kusper, Y. Zijian Győző, B. Nagy: *Using extended resolution to represent strongly connected components of directed graphs*, Annales Mathematicae et Informaticae Accepted manuscript (2023), issn: 1787-6117, doi: 10.33039/ami.2023.08.011.

[15] G. Morse: *dyngraph: Dynamic graph maintenance source code and experiment scripts*, url: https://github.com/GregoryMorse/dyngraph (visited on 06/08/2026).

[16] E. Nuutila: *An Efficient Transitive Closure Algorithm for Cyclic Digraphs*, Information Processing Letters 52 (1994), doi: 10.1016/0020-0190(94)90128-7.

[17] E. Nuutila, E. Soisalon-Soininen: *On finding the strongly connected components in a directed graph*, Information Processing Letters 49.1 (1994), pp. 9–14, issn: 0020-0190, doi: 10.1016/0020-0190(94)90047-7.

[18] D. J. Pearce: *A space-efficient algorithm for finding strongly connected components*, Information Processing Letters 116.1 (2016), pp. 47–52, issn: 0020-0190, doi: 10.1016/j.ipl.2015.08.010.

[19] P. Purdom: *A transitive closure algorithm*, BIT Numerical Mathematics 10.1 (Mar. 1970), pp. 76–94, issn: 1572-9125, doi: 10.1007/BF01940892.

[20] L. Roditty: *A Faster and Simpler Fully Dynamic Transitive Closure*, ACM Trans. Algorithms 4.1 (Mar. 2008), issn: 1549-6325, doi: 10.1145/1328911.1328917.

[21] P. Sankowski: *Dynamic transitive closure via dynamic matrix inverse: extended abstract*, in: 45th Annual IEEE Symposium on Foundations of Computer Science, 2004, pp. 509–517, doi: 10.1109/FOCS.2004.25.

[22] Y. Shiloach, S. Even: *An On-Line Edge-Deletion Problem*, J. ACM 28.1 (Jan. 1981), pp. 1–4, issn: 0004-5411, doi: 10.1145/322234.322235.

[23] R. Tarjan: *Depth first search and linear graph algorithms*, SIAM JOURNAL ON COMPUTING 1.2 (1972), doi: 10.1137/0201010.

[24] R. E. Tarjan: *Data Structures and Network Algorithms, 2. Disjoint Sets*, in: Society for Industrial and Applied Mathematics, 1983, pp. 23–31, doi: 10.1137/1.9781611970265.ch2.

[25] R. E. Tarjan: *Edge-disjoint spanning trees and depth-first search*, Acta Informatica 6.2 (June 1976), pp. 171–185, issn: 1432-0525, doi: 10.1007/BF00268499.

[26] B. Yang, D. Wen, L. Qin, Y. Zhang, X. Wang, X. Lin: *Fully Dynamic Depth-First Search in Directed Graphs*, Proc. VLDB Endow. 13.2 (Oct. 2019), pp. 142–154, issn: 2150-8097, doi: 10.14778/3364324.3364329.

Appendix A: Example

Consider the graph in Figure 2. When the edge $(5, 3)$ is added, it creates a new SCC as seen in Figure 3. By traversing forward through SCCs reachable from the destination node's SCC, the component represented by 4 traverses to component 7 which is the source node component and finally components 8, 9 and 10 are ignored since they do not reach component 4. The two components are collapsed into a new component and their reachability merged by a union operation. Finally the reachability for the predecessors of the new component is propagated backwards as it would be for an edge addition which does not change the SCCs. Likewise, if the edge $(5, 3)$ is then deleted, Nuutila's algorithm computes the SCCs for the sub-graph of the SCC being decomposed as shown in Figure 4. As it can be seen, component 8, 9 and 10 are not in the sub-graph so as the SCCs are rebuilt all of them are included in the new reachability sets. Then the SCCs which reach the target as well as all SCCs the target reaches is computed in a single topological ordered predecessor-based traversal of nodes. Afterwards, predecessor-based propagation starts from the new SCCs, to allow precise subtractive operations to recompute reachability sets affected by the deletion, which is identical to the propagation that occurs for an edge deletion that does not change the components.

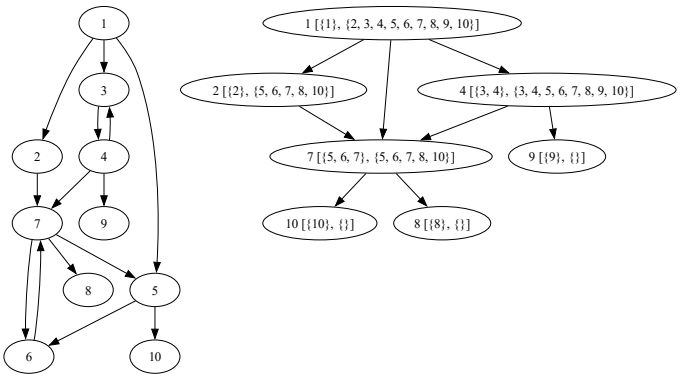

Figure 2. A digraph and its corresponding SCC DAG with its SCCs and reachability shown.

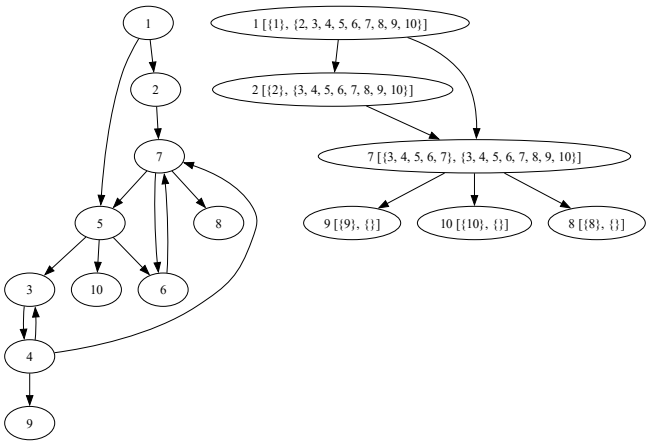

Figure 3. The digraph with edge (5, 3) added.

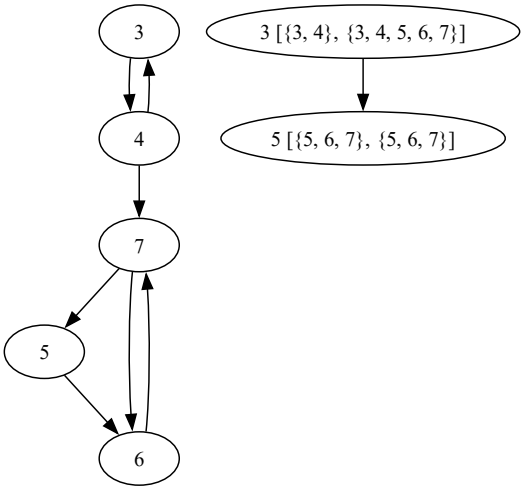

Figure 4. The sub-graph and its corresponding SCC DAG with its SCC and reachability when edge (5, 3) is removed.

Appendix B: Batching

Batching can have special optimizations for for multiple edges which all come from a single source node or go to a single target node in many circumstances including this one. However, we consider only arbitrary edges, rather than various special batching cases. The basic idea is that the time complexity for processing a batch of edges must have the same worst case as the non-batching dynamic algorithms making the number of edges in the batch an additive term and not multiplicative term.

In the incremental context, self-loops and edge additions which are already within the same component should be filtered out. Additionally if this filtering yields only a single edge then it should be processed with the non-batching algorithm, which is a standard optimization. In the decremental context, self-loops should be filtered out and optionally traversal reachability checks can filter out edges within a component where the source still reaches the target via a longer path. The single edge case also applies. After this basic processing, two options are then possible when converting the standard incremental and decremental algorithms to a batching algorithm for both the incremental and decremental case. It should be noted that Nuutila's algorithm will need to be applied regardless in all situations with batching but in different ways.

The first method for the incremental context simply uses Nuutila to recompute the SCCs and reachability for the graph assumed to have the new edges already present but having its existing SCCs collapsed. The results are then directly merged. The second method uses Nuutila on the subgraph induced only by the filtered added edges and any components reachable from source or target nodes of these edges, followed by merging and propagating reachability for these nodes.

For the decremental case, the first method is equivalent in that Nuutila can recompute the SCCs and reachability for the graph assumed to have the deleted edges already removed but having SCCs collapsed except the ones which are being decomposed. The second method only uses Nuutila for the components to be decomposed, and then re-merges the resulting components followed by propagating reachability. In fact, the algorithm as written is very easily adapted to the batching setting. It is the incremental case which requires a different SCC strategy as the detection of merging components via reachability is possible only if the reachability information is continuously updated. One strategy could be to process the edges which are detected as forming an SCC in sub-batches and propagating their reachability, and then repeatedly doing this until no more SCC merges can be done. Yet this would lead to a worst case time complexity equivalent to using the non-batching incremental algorithm and hence inappropriate for batching.

Unfortunately practically speaking there are cases where the proposed batching algorithms would perform slower than the non-batching algorithms, unless the batch sizes are large. In the event that current graph is small and the number of batched edges is large enough, an offline computation would achieve better performance without batching, with a reasonable threshold being a constant factor for the edge insertion case and a factor of n for the edge deletion scenario. Even with carefully optimized batching code, in practice the best performance may occur when small batches are done with non-batching algorithms, large batches are done with offline computation and the batching algorithms are used for a carefully determined range in between.

The most elegant batching algorithm would handle intermixed edge additions and deletions, not just one or the other. However, this can be done efficiently by using a general principle for intermixed batching. Namely, filtering out additions and deletions of identical edges, followed by dividing it into separate sets of edge

additions and edge deletions and using the respective incremental or decremental batching algorithms only once each.

Appendix C: Random graph experiments

Here we build random connectivity maintained graphs with $10 - 200$ nodes by adding $\frac{n^2}{2}$ random edges until the point that the graph becomes dense. This number of edges guarantees at least one non-trivial SCC will be present. Then we delete those same edges in the reverse order to maintain connectivity which gives the results in Figure 5.

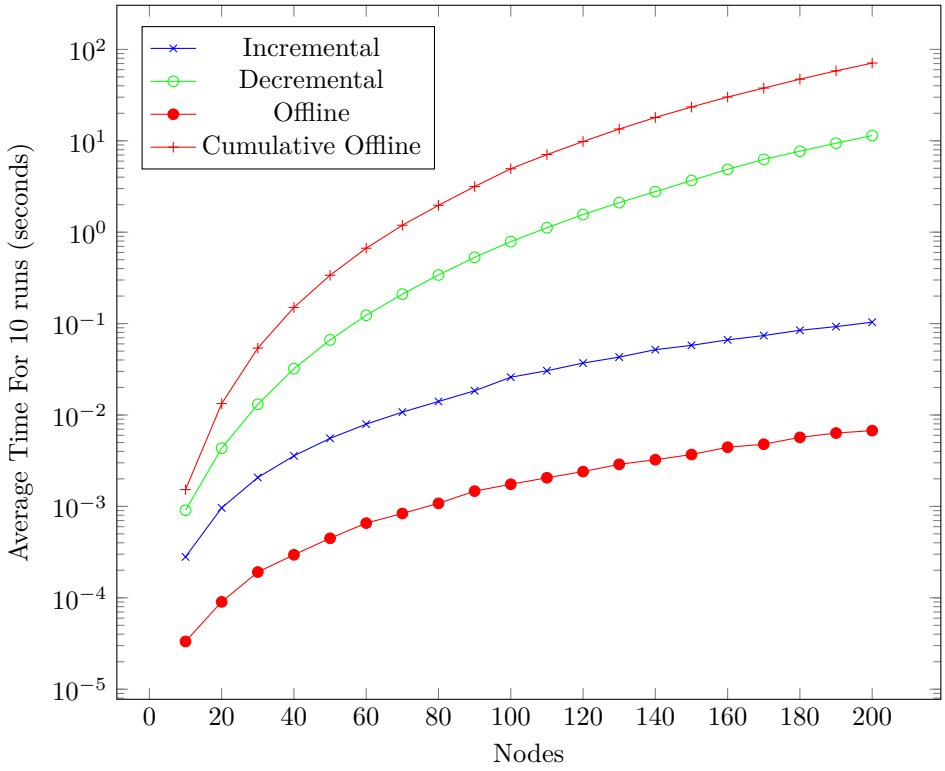

Figure 5. Random Connected Graph Edge Addition and Deletion Performance Execution Time Averaged Over 10 Runs.

The problem with random graphs is that eventually a single large component will emerge and does not necessarily provide a good understanding of how it would perform on actual CFGs. Instead, a better method is to use an out-degree and locality parameter so that there will be many smaller SCCs. However, this is still unrealistic as large loops can and do occur in real programs. So instead we opt for

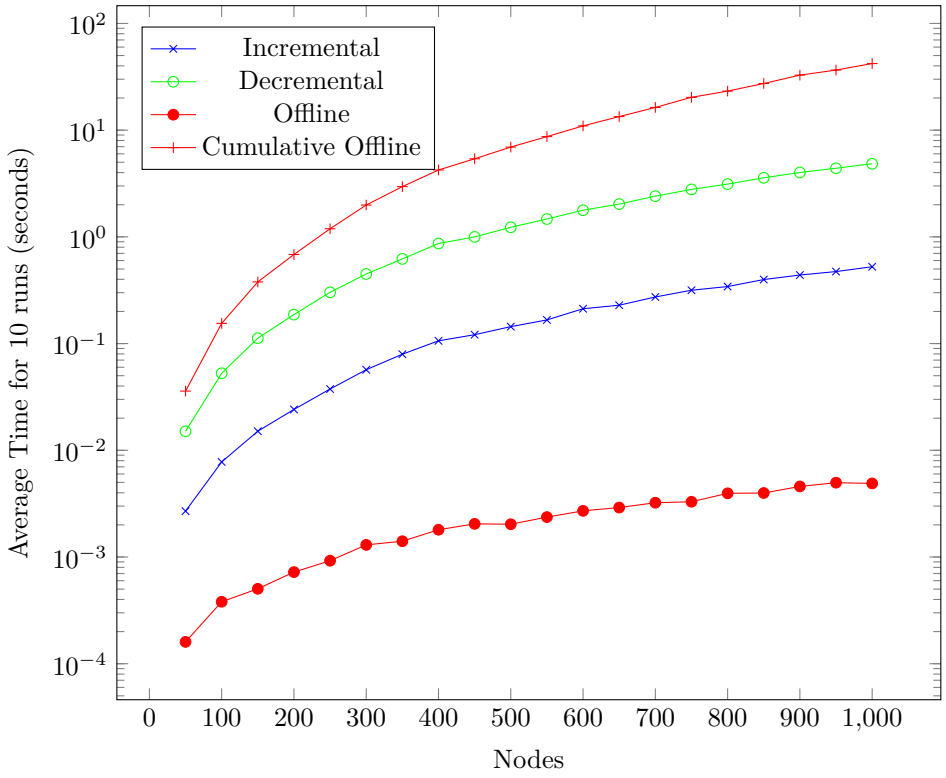

Figure 6. Random Fixed Locality=5 and Average Out-Degree=5 Connected Graph Edge Addition and Deletion Execution Time Averaged Over 10 Runs.

testing with both methods to evaluate the performance in different circumstances. The results for a fixed locality and average out-degree are presented in Figure 6.

Since maintaining connectivity of the graph skews the randomness towards nodes which became part of the graph earlier in the edge addition choices, it should be noted that mathematically the results here are not trying to achieve traditional standards of randomness where there is typically a fixed expected probability for each edge. Given that this algorithm has use cases which are practical in nature, it made more sense to sacrifice the simple probabilities offered by random graphs that do not necessarily have connectivity, for a more context sensitive variant with biases which are understood. A graph is said to be connected if there exists one or more nodes which can reach every other node, and one of these nodes in the CFG context is the entry and termed the root.