Dynamic DFS in undirected graphs

Baswana, Chaudhury, Choudhary and Khan

Tim Ungerhofer

Table of contents

Definitions

Definition (DFS traversal)

Let $G = (V, E)$ be an undirected connected graph with $|V| = n$ and $|E| = m$. A DFS traversal from any node $r \in V$ produces a rooted spanning tree, called DFS tree, with root r.

Definitions

Definition (DFS traversal)

Let $G = (V, E)$ be an undirected connected graph with $|V| = n$ and $|E| = m$. A DFS traversal from any node $r \in V$ produces a rooted spanning tree, called DFS tree, with root r.

Given any rooted spanning tree of G, non-tree edges can be classified in two categories:

- Back edge
  $(u, v) \in E$: u is an ancestor of v in the tree, or vice versa.
- Cross edge
  $(u, v) \in E$: (u, v) is not a back edge.

Definition (DFS tree)

A rooted spanning tree is a DFS tree where any non-tree edges are back edges.

Dynamic environments

Real-world graph applications may deal with changing graphs.

Definition (Graph update)

A graph update is either the insertion or deletion of vertices or edges.

Dynamic environments

Real-world graph applications may deal with changing graphs.

Definition (Graph update)

A graph update is either the insertion or deletion of vertices or edges.

Classification of dynamic graph algorithms:

- Partially dynamic
  1. Incremental (insertions)
  2. Decremental (deletions)
- Fully dynamic (both)

What problem is solved?

Algorithmic Graph Problem (dynamic setting)

Given an online sequence of updates, efficiently update an existing solution after each update.

What problem is solved?

Algorithmic Graph Problem (dynamic setting)

Given an online sequence of updates, efficiently update an existing solution after each update.

Specifically:

Solution updates must be faster than the best static algorithm for the problem.

What problem is solved?

Algorithmic Graph Problem (dynamic setting)

Given an online sequence of updates, efficiently update an existing solution after each update.

Specifically:

Solution updates must be faster than the best static algorithm for the problem.

Problem statement

Given an undirected G, and a set of updates U, maintain a DFS tree efficiently in any dynamic environment.

Challenges in dynamic DFS

Reif ['85] showed that the ordered DFS tree problem is *P*-Complete.
This seemed to imply that the computation of any DFS tree is inherently sequential.
Reif ['87] and later, Miltersen et al ['94] showed that *P*-Completeness of a problem also implies hardness of the problem in the dynamic setting.
This means that maintaining an ordered DFS tree is among the hardest problems in *P*.
Although only for the ordered class of DFS trees, these results stifled research into all dynamic DFS problems.

Existing results for maintaining a dynamic DFS tree

Incremental algorithm in a DAG $\mathcal{O}(mn)$, Franciosa, Gambosi, and Nanni['97].
Decremental algorithm in a DAG $\mathcal{O}(mnlogn)$, Baswana and Choudhary ['15].
Incremental algoritm in undirected Graph $\mathcal{O}(n^2)$, Baswana and Khan ['14].

Existing results for maintaining a dynamic DFS tree

Incremental algorithm in a DAG $\mathcal{O}(mn)$, Franciosa, Gambosi, and Nanni['97].

Decremental algorithm in a DAG $\mathcal{O}(mnlogn)$, Baswana and Choudhary ['15].

Incremental algoritm in undirected Graph $\mathcal{O}(n^2)$, Baswana and Khan ['14].

Partially dynamic environments, none achive $o(m)$ update time. The $\mathcal{O}(m)$ barrier persists and the general DFS is still as hard as the ordered DFS problem in the dynamic environment.

Questions and Answers

1. Does there exist any nontrivial fully dynamic algorithm for maintaining a DFS tree?

Questions and Answers

1. Does there exist any nontrivial fully dynamic algorithm for maintaining a DFS tree?
2. Is it possible to achieve worst case o(m) update time for maintaining a DFS tree in a dynamic environment?

Questions and Answers

1. Does there exist any nontrivial fully dynamic algorithm for maintaining a DFS tree?
2. Is it possible to achieve worst case o(m) update time for maintaining a DFS tree in a dynamic environment?

For undirected graphs, both questions can be answered in the affirmative.

Questions and Answers

1. Does there exist any nontrivial fully dynamic algorithm for maintaining a DFS tree?
2. Is it possible to achieve worst case o(m) update time for maintaining a DFS tree in a dynamic environment?

For undirected graphs, both questions can be answered in the affirmative.
The result handles both edge and vertex updates (which are considered harder), making it widely applicable for many dynamic graph problems.

Main Idea

Observation

Throughout this presentation, let U be the set of generalized updates (insertions/deletions of vertices/edges).

Let $G + U$ denote the undirected Graph G after performing updates U.
Let T be a DFS tree of G.

Main Idea

Observation

Throughout this presentation, let U be the set of generalized updates (insertions/deletions of vertices/edges).

Let $G + U$ denote the undirected Graph G after performing updates U.
Let T be a DFS tree of G.
To compute a DFS tree of $G + U$, the main idea is to reuse T to preprocess G to build a data structure D.

Preprocessing G

First, add a dummy vertex r to G and connect it to all elements of V. The algorithm starts with any arbitrary DFS tree rooted at r, which is maintained throughout all updates.

Observation

Each subtree rooted at any child of r is a DFS tree of a connected component of $G + U$.

Preliminaries

Notation used throughout this presentation:

- $T(x)$: The subtree of T rooted at vertex x.
- $path(x, y)$: the path from vertex x to vertex y.
- $par(x)$: the parent of vertex x.
- $T*$: The DFS tree rooted at r.
- $LCA(x, y)$: the least common ancestor of x and y

Preliminaries

Notation used throughout this presentation:

- $T(x)$: The subtree of T rooted at vertex x.
- $path(x, y)$: the path from vertex x to vertex y.
- $par(x)$: the parent of vertex x.
- $T*$: The DFS tree rooted at r.
- $LCA(x, y)$: the least common ancestor of x and y

Unless stated otherwise, *path* always refers to an ancestor-descendant path.

Definition (Ancestor-descendant path)

A path p in a DFS tree T is said to be an ancestor-descendant path if its endpoints have an ancestor-descendant relationship in T.

Preliminaries

The data structure D supports the following operations:

1. $Query(w, x, y)$: among all the edges from w that are incident on path(x, y) in G + U , return the edge that is incident nearest to x on path(x, y).
2. $Query(T(w), x, y)$: among all the edges from T (w) that are incident on path(x, y) in G + U , return an edge that is incident nearest to x on path(x, y).

Leveraging DFS traversal flexibility

During DFS traversal at any vertex $v \in V$, any unvisited neighbor of v may be visited next.

This flexibility, the original DFS tree T and the components property of a DFS traversal are used to efficiently compute the DFS tree for $G + U$.

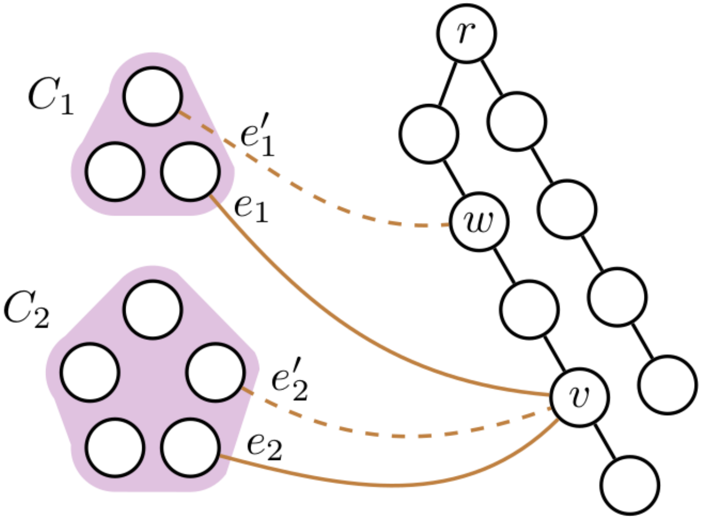
Leveraging DFS traversal flexibility

During DFS traversal at any vertex $v \in V$, any unvisited neighbor of v may be visited next.
This flexibility, the original DFS tree T and the components property of a DFS traversal are used to efficiently compute the DFS tree for $G + U$.

Lemma (Components property)

Let $T*$ be the partially grown DFS tree and v be the vertex currently being visited. Let C be any connected component in the subgraph induced by the unvisited vertices. Suppose two edges e and $e\prime$ from C are incident, respectively, on v and some ancestor (not necessarily proper) w of v in $T*$.
Then it is sufficient to consider only e during the rest of the DFS traversal; i.e., $e\prime$ need not be scanned.

Components property of DFS traversal

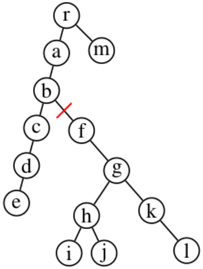


Components property of DFS traversal

To highlight the importance of the components property and to motivate the use of the data structure D, consider handling a single update, the failure of an edge e.

Handling a single update: Edge failure

Consider this example of the edge (b, f) failing.

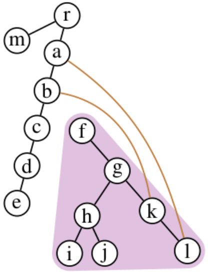


Observation

During DFS traversal at b, the unvisited $T(f)$ is a single connected component.

Handling a single update: Edge failure

Now, assume that the following edges exist in G:



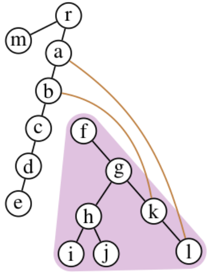
By the components property, we only need to process the edge closest to b, (k, b). The DFS traversal thus enters $T(f)$ at k, turning k into the new root of this subtree.
Rebuilding the DFS tree after edge failure thus reduces to finding the lowest edge from $T(f)$ to $path(b, r)$.

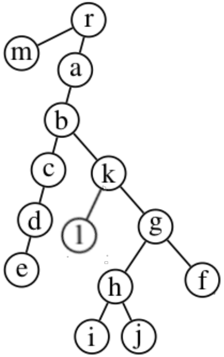
Rerooting a DFS tree

Procedure Reroot($T(r_0), r'$): Reroots the subtree $T(r_0)$ of T to be rooted at the vertex $r' \in T(r_0)$.

```
1  foreach $(a, b)$ on $path(r_0, r')$ do  /* $a = par(b)$ in original tree $T(r_0)$. */
2      $par(a) \leftarrow b$;
3      foreach child $c$ of $b$ not on $path(r_0, r')$ do
4          $(u, v) \leftarrow Query(T(c), r_0, b)$ ;   /* where $u \in path(r_0, r')$ and $v \in T(c)$. */
5          if $(u, v)$ is nonnull then
6              $Reroot(T(c), v)$;
7              $par(v) \leftarrow u$;
8          end
9      end
10 end
```

Result of $\mathit{Reroot}(T(f), k)$





Reroot run time analysis

The total time required is proportional to the number of edges processed by the procedure.
These edges include tree edges and added edges.
The number of tree edges is bounded by $\mathcal{O}(|T\prime|)$, $T\prime$ being the original subtree.
The number of added edges is also bounded by $|T\prime|$.
This results in the total runtime of $\mathcal{O}(|T\prime| * query)$ where query is the time taken by the data structure to answer each query.

Reducing all edge updates to reroot

1. Deletion of an edge (u, v):
   In case (u, v) is a back edge in T , simply delete it from the graph. Otherwise, let u = par(v) in T. The algorithm finds the lowest edge $(u\prime\ , v\prime)$ on the $path(u, r)$ from $T(v)$, where $v\prime \in T(v)$. The subtree $T(v)$ is then rerooted at its new root $v\prime$ and appended to $u\prime$ using $(u\prime, v\prime)$ in the final tree $T*$.
2. Insertion of an edge (u,v):
   In case (u, v) is a back edge, simply insert it in the graph. Otherwise, let w be the LCA of u and v in T and $v\prime$ be the child of w such that $v \in T(v\prime)$. The subtree $T(v\prime)$ is then rerooted at its new root v and appended at u using (u, v) in the final tree $T*$.

Data structure D

Recall

For any three vertices $w, x, y \in T$ with $path(x, y)$, D must answer the queries:

1. $Query(w, x, y)$: among all the edges from w that are incident on path(x, y) in G + U , return the edge that is incident nearest to x on path(x, y).
2. $Query(T(w), x, y)$: among all the edges from T (w) that are incident on path(x, y) in G + U , return an edge that is incident nearest to x on path(x, y).

Data structure D

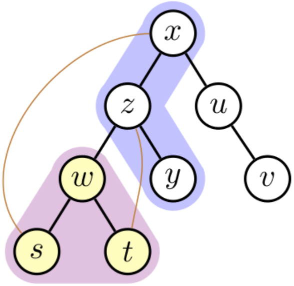
D is built using two techniques called heavy-light composition and suitable augmentation of a binary tree (segment tree) as follows:

1. Perform a preorder traversal of tree T with the following restriction: Upon visiting a vertex $v \in T$, the child of v that is visited first is the one storing the largest subtree.
   Let L be the list of vertices ordered by this traversal.

Data structure *D*

D is built using two techniques called heavy-light composition and suitable augmentation of a binary tree (segment tree) as follows:

1. Perform a preorder traversal of tree T with the following restriction: Upon visiting a vertex $v \in T$, the child of *v* that is visited first is the one storing the largest subtree.
   Let *L* be the list of vertices ordered by this traversal.



This produces *L* = [x, z, w, s, t, y, u, v]

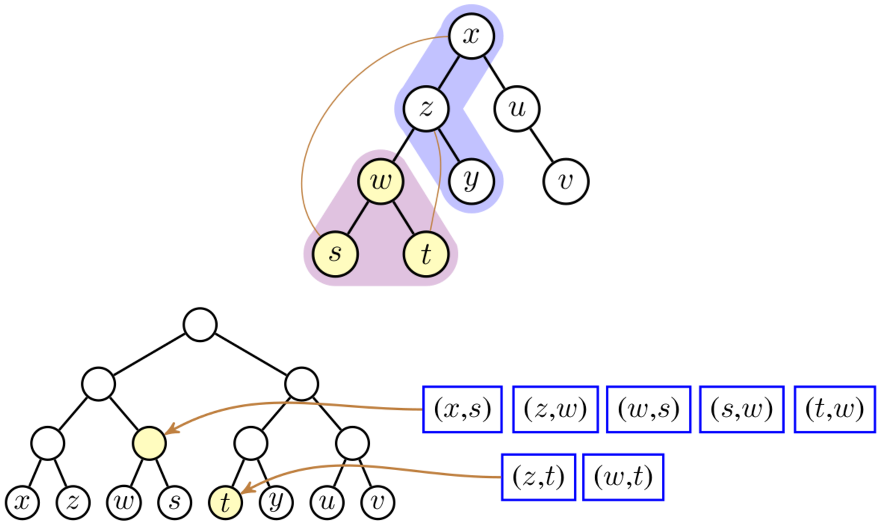
Data structure D

2. Build a segment tree T_B whose leaf nodes from left to right represent the vertices in L.

Data structure *D*

2. Build a segment tree T_B whose leaf nodes from left to right represent the vertices in L.
3. Augment each node $z \in T_B$ with a binary search tree $Bin(z)$, storing all edges $(u, v) \in E$ where u is a leaf node in the subtree rooted at z in T_B. These edges are sorted according to the position of the second endpoint in L

Data structure *D*



L = [x, z, w, s, t, y, u, v]

Data structure D properties

- For $Query(T(w), x, y)$, the subtree $T(w)$ is present in L as an interval of subtrees. This interval can be expressed as a union of $\mathcal{O}(logn)$ disjoint subtrees in T_B.
- The $path(x, y)$ can be divided into $\mathcal{O}(logn)$ subpaths.
- To find the edge closest to $path(x, y)$, a single predecessor or successor query on $Bin(z)$ suffices. This query can be answered in $\mathcal{O}(logn)$.

Data structure D properties

- For $Query(T(w), x, y)$, the subtree $T(w)$ is present in L as an interval of subtrees. This interval can be expressed as a union of $\mathcal{O}(logn)$ disjoint subtrees in T_B.
- The $path(x, y)$ can be divided into $\mathcal{O}(logn)$ subpaths.
- To find the edge closest to $path(x, y)$, a single predecessor or successor query on $Bin(z)$ suffices. This query can be answered in $\mathcal{O}(logn)$.

Therefore, queries can be answered by D in $\mathcal{O}(log^3 n)$.
D requires $\mathcal{O}(mlogn)$ space, as each edge is stored at $O(logn)$ levels in T_B.

Data structure D properties

- For $Query(T(w), x, y)$, the subtree $T(w)$ is present in L as an interval of subtrees. This interval can be expressed as a union of $\mathcal{O}(logn)$ disjoint subtrees in T_B.
- The $path(x, y)$ can be divided into $\mathcal{O}(logn)$ subpaths.
- To find the edge closest to $path(x, y)$, a single predecessor or successor query on $Bin(z)$ suffices. This query can be answered in $\mathcal{O}(logn)$.

Therefore, queries can be answered by D in $\mathcal{O}(log^3 n)$.
D requires $\mathcal{O}(mlogn)$ space, as each edge is stored at $O(logn)$ levels in T_B.T_B can be built in linear time and $Bin(z)$ can be built in time linear in the number of edges in $Bin(z)$ for each node z.

Data structure D properties

- For $Query(T(w), x, y)$, the subtree $T(w)$ is present in L as an interval of subtrees. This interval can be expressed as a union of $\mathcal{O}(logn)$ disjoint subtrees in T_B.
- The $path(x, y)$ can be divided into $\mathcal{O}(logn)$ subpaths.
- To find the edge closest to $path(x, y)$, a single predecessor or successor query on $Bin(z)$ suffices. This query can be answered in $\mathcal{O}(logn)$.

Therefore, queries can be answered by D in $\mathcal{O}(log^3 n)$.
D requires $\mathcal{O}(mlogn)$ space, as each edge is stored at $O(logn)$ levels in T_B.T_B can be built in linear time and $Bin(z)$ can be built in time linear in the number of edges in $Bin(z)$ for each node z.
This results in the total build time of $\mathcal{O}(mlogn)$ for D.

Data structure D properties

- For $Query(T(w), x, y)$, the subtree $T(w)$ is present in L as an interval of subtrees. This interval can be expressed as a union of $\mathcal{O}(logn)$ disjoint subtrees in T_B.
- The $path(x, y)$ can be divided into $\mathcal{O}(logn)$ subpaths.
- To find the edge closest to $path(x, y)$, a single predecessor or successor query on $Bin(z)$ suffices. This query can be answered in $\mathcal{O}(logn)$.

Therefore, queries can be answered by D in $\mathcal{O}(log^3 n)$.
D requires $\mathcal{O}(mlogn)$ space, as each edge is stored at $O(logn)$ levels in T_B.T_B can be built in linear time and $Bin(z)$ can be built in time linear in the number of edges in $Bin(z)$ for each node z.
This results in the total build time of $\mathcal{O}(mlogn)$ for D.
Furthermore, this means the runtime of procedure Reroot can now be finalized to $\mathcal{O}(|T\prime|log^3 n)$

Handling multiple updates

For a single update, the DFS tree can be recomputed in $\tilde{\mathcal{O}}(n)$, by reduction to procedure Reroot.
For multiple updates, however, the same is not true, as Reroot is dependent on D, which is built from the original tree T. After an update, e.g. deletion of an edge, D must be rebuilt for the updated tree.

Handling multiple updates

For a single update, the DFS tree can be recomputed in $\tilde{\mathcal{O}}(n)$, by reduction to procedure Reroot.
For multiple updates, however, the same is not true, as Reroot is dependent on D, which is built from the original tree T. After an update, e.g. deletion of an edge, D must be rebuilt for the updated tree.
To avoid the $\mathcal{O}(mlogn)$ build time after each update, D will be reused across multiple updates as follows.

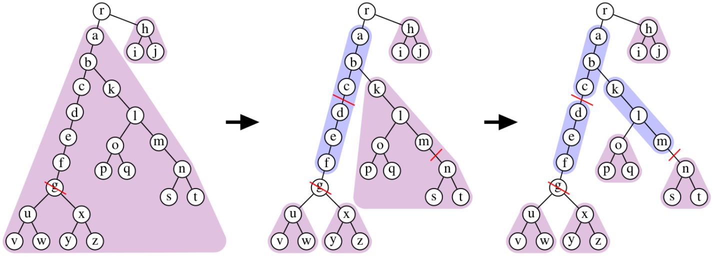
Handling multiple updates

Observation

To handle $G+U$, D needs to ensure that all queried paths and subtrees do not contain failed edges.

For any set U, compute a partitioning of T into a collection of paths $\mathcal{P}$ and subtrees $\mathcal{T}$, where all paths and subtrees are disjoint.

Disjoint tree partitioning



Procedure

- For every failed vertex $v \in$ subtree of $\mathcal{T}$, add the path from $par(v)$ to the root of the subtree to $\mathcal{P}$, store the resulting subtrees in $\mathcal{T}$.
- If $v \in$ path of $\mathcal{P}$: Split the path at v, remove the path from $\mathcal{P}$ and add the two resulting paths to $\mathcal{P}$.

Dynamic DFS

Procedure Static-DFS(G, r): Static algorithm to compute a DFS tree of G rooted at r.

```
Stack $S \leftarrow \emptyset$;
$Push(r)$;
$status(r) \leftarrow visited$;
while $S \neq empty$ do
    $w \leftarrow Top(S)$;
    if $N(w) = \emptyset$ then $Pop(w)$;
    else
        $u \leftarrow$ First vertex in $N(w)$;
        Remove $u$ from $N(w)$;
        if $status(u) = unvisited$ then
            $par(u) \leftarrow w$;
            $status(u) \leftarrow visited$;
            $Push(u)$;
        end
    end
end
```

Procedure Dynamic-DFS(G, U, r): Algorithm for updating the DFS tree T rooted at r for the graph $G + U$.

```
Stack $S \leftarrow \emptyset$;    $(\mathcal{T}, \mathcal{P}) \leftarrow Partition(T, U)$;
$Push(r)$;
$status(r) \leftarrow visited$;  $L(r) \leftarrow N(r)$;
while $S \neq empty$ do
    $w \leftarrow Top(S)$;  $u_0 \leftarrow w$;
    if $L(w) = \emptyset$ then $Pop(w)$;
    else
        $u \leftarrow$ First vertex in $L(w)$;
        Remove $u$ from $L(w)$;
        if $status(u) = unvisited$ then
            if INFO$(u) = tree$ then
                $\{u_1, ..., u_t\} \leftarrow$ DFS-in-Tree$(u)$;
            else if INFO$(u) = path$ then
                $\{u_1, ..., u_t\} \leftarrow$DFS-in-Path$(u)$;
            end
            for $i = 1$ to $t$ do
                $par(u_i) \leftarrow u_{i-1}$;
                $status(u_i) \leftarrow visited$;
                $Push(u_i)$;
            end
        end
    end
end
```

Procedures for in-tree and in-path DFS

```
if $status(u) = unvisited$ then
    if INFO$(u) = tree$ then
        $\{u_1, ..., u_t\} \leftarrow$ DFS-in-Tree$(u)$;
    else if INFO$(u) = path$ then
        $\{u_1, ..., u_t\} \leftarrow$DFS-in-Path$(u)$;
    end
    for $i = 1$ to $t$ do
        $par(u_i) \leftarrow u_{i-1}$;
        $status(u_i) \leftarrow visited$;
        $Push(u_i)$;
    end
end
```

DFS-in-tree(u)

Adds the path from u to the root of its subtree to $\mathcal{P}$, appends resulting subtrees to $\mathcal{T}$ and updates $L(w)$ for all vertices w on this path.

Procedures for in-tree and in-path DFS

```
if $status(u) = unvisited$ then
    if $\text{INFO}(u) = tree$ then
        $\{u_1, ..., u_t\} \leftarrow$ DFS-in-Tree($u$);
    else if $\text{INFO}(u) = path$ then
        $\{u_1, ..., u_t\} \leftarrow$DFS-in-Path($u$);
    end
    for $i = 1$ to $t$ do
        $par(u_i) \leftarrow u_{i-1}$;
        $status(u_i) \leftarrow visited$;
        $Push(u_i)$;
    end
end
```

DFS-in-path(u)

Scans the path from *u* to the far end of a given *path*(x, y), let *y* be the far end. Replaces *path*(x, y) with *path*(u, y) in $\mathcal{P}$, computes $L(w)$ for all vertices w *path*(u, y).

Correctness

Lemma (Adapted components property $*$)

When a path p is attached to the partially constructed DFS tree $T*$ during the algorithm, for every edge (x, y), where $x \in p$ and y belongs to the unvisited graph, the following condition holds:
Either y is added to L(x) or $y\prime$ is added to $L(x\prime)$ for some edge $(x\prime, y\prime)$, where $x\prime$ is a descendant (not necessarily proper) of x in p and $y\prime$ is connected to y in the unvisited graph.

- Invariant 1:
  The sequence of vertices in the stack from bottom to top constitutes an ancestor-descendant path from r in the DFS tree computed.
- Invariant 2:
  For each vertex v that is popped out, all vertices in the set N (v) have already been visited.

Extension to a fully dynamic algorithm

Dynamic-DFS can be further extended to include insertions of vertices and edges and by scheduling the rebuilding of D correctly, the algorithm can be used to solve problems such as the dynamic subgraph connectivity problem and the dynamic biconnectivity and dynamic 2-edge connectivity problem with the following run times.

Results

For dynamic subgraph connectivity:

References	Update time	Query time
Frederickson [27] (1985), Eppstein et al. [24] (1997)	$O(n\sqrt{n})$	$O(1)$
Holm, de Lichtenberg, and Thorup [37] (2001)	$\tilde{O}(n)$ amortized	$\tilde{O}(1)$
Chan [12] (2006)	$\tilde{O}(m^{0.94})$ amortized	$\tilde{O}(m^{1/3})$
Chan, Patrascu, and Roditty [13] (2008)	$\tilde{O}(m^{2/3})$ amortized	$\tilde{O}(m^{1/3})$
Duan [22] (2010)	$\tilde{O}(m^{4/5})$	$\tilde{O}(m^{1/5})$
Kapron, King, and Mountjoy [41] (2013)	$\tilde{O}(n)$	$\tilde{O}(1)$ (Monte Carlo)
New	$\tilde{O}(\sqrt{mn})$	$O(1)$

Results

For dynamic biconnectivity ($*$) and dynamic 2-edge connectivity ($\dagger$):

References	Update time	Query time
Frederickson [27] (1985), Eppstein et al. [24] (1997)	$O(n\sqrt{n})$	$O(1)$
Holm, de Lichtenberg, and Thorup [37] (2001)	$\tilde{O}(n)$ amortized	$\tilde{O}(1)$
Chan [12] (2006)	$\tilde{O}(m^{0.94})$ amortized	$\tilde{O}(m^{1/3})$
Chan, Patrascu, and Roditty [13] (2008)	$\tilde{O}(m^{2/3})$ amortized	$\tilde{O}(m^{1/3})$
Duan [22] (2010)	$\tilde{O}(m^{4/5})$	$\tilde{O}(m^{1/5})$
Kapron, King, and Mountjoy [41] (2013)	$\tilde{O}(n)$	$\tilde{O}(1)$ (Monte Carlo)
New	$\tilde{O}(\sqrt{mn})$	$O(1)$

Thank you for your attention!