Rezumat:Studiu teoretic si practic al algoritmilor DFS incrementali[1]

Nicoleta Ciauşu

December 4, 2022

1 Introducere

Parcurgerea în adâncime (i.e. Depth first search sau DFS) este o tehnică de baza folosită pentru a parcurge un graf. Utilitatea acestei tehnici este indiscutabilă, ea fiind folosită în componența a mai multori algoritmi ce produc rezultate folositoare (de exemplu: sortare topologică, componente conexe, și altele).

Articolul ales de mine se concentrează pe următoarea problema: în practică se lucrează foarte rar cu grafuri statice. In marea majoritate a situațiilor în care folosim grafuri, ele vor suferi modificări precum adăugare/eliminare de noduri sau muchii. În această situație, un algoritm obișnuit, offline, ar trebui să fie rerulat pe noul input pentru a afla rezultatul modificat. Ideal, în acest caz, ne-am dori să folosim un algoritm ce poate procesa modificările asupra grafului cu o complexitate mai bună decât simpla recalculare pe întregul graf.

Acești algoritmi sunt numiți în mod generic algoritmi dinamici (i.e. dynamic algorithms). Un algoritm incremental este un caz special de algoritm dinamic, ce permite doar adăugarea de muchii. Practic, ne putem imagina că primim secvențial muchiile unui graf, iar algoritmul construiește soluția treptat.

În cazul articolului de față, discuția se centrează în jurul menținerii în memorie a arborelui generat de parcurgerea DFS a unui graf. Deși este un concept de baza, există un număr limitat de algoritmi incrementali care permit construcția acestui arbore. Articolul trece în revista acești algoritmi, explicând modul în care funcționează, determinând atât complexitatea teoretică, cât și performanțele în practică, pe diverse input-uri.

În urma acestor experimente, autorii observă o proprietate a formei pe care arborii DFS o iau, și plecând de la această observație, ei propun un nou algoritm incremental, cu complexitate similară, dar performanță practică mai bună, decât cei existenți deja.

S-a ales explorarea doar a algoritmilor incrementali și nu a celor dinamici din două motive: 1. majoritatea covârșitoare a operațiilor pe grafuri în situații reale sunt operații de insert [1] și 2. există un singur algoritm DFS dinamic, deci nu ar fi avut sens comparațiile.

2 Preliminarii[2]

1. Pentru a putea obtine un DFS Tree ce parcurge intregul graf, grafului original G ii va fi adaugat un nod s conectat la toate celelalte nodurile din graf. Ruland algoritmul DFS incepand din s, DFS Tree-ul obtinut va avea radacina in s, iar toti descendentii directi ai lui s vor fi radacini pentru componentele conexe ale lui G.

2. T este arborele DFS asociat grafului G.

3. $T(x)$ este subarborele DFS ce are radacina in x.

4. $LCA(x, y)$ este cel mai apropiat stramos comun al lui x si y in T.

5. $path(x, y)$ este lantul de la nodul x la nodul y in T

6. Un *back-edge* este o muchie care leaga un nod de catre un stramos al acestuia in arbore.

7. Un *cross-edge* este o muchie care leaga doua noduri ce nu au ca LCA pe unul dintre cele doua noduri. Insertia acestui tip de muchie duce la recalcularea anumitor parti din arborele DFS. In cazul grafurilor orientate, apare o restrictie in plus legata de sens: muchia este numita cross-edge daca in reprezentarea arborelui DFS ea conecteaza un subarbore aflat din dreapta cu unul din stanga.

8. Un *anti cross edge* este o muchie care in reprezentarea arborelui DFS a unui graf orientat conecteaza un subarbore aflat din stanga cu unul din dreapta.

9. Observatie: distinctia intre muchii *cross-edge* si *anti cross-edge* apare doar in cazul grafurilor G orientate. Pentru cele neorientate, se foloseste doar denumirea *cross-edge*.

10. In lucrare, avem doua tipuri de definiri ale grafurilor aleatoare:

    (a) $G(n, p)$ unde n este numarul de noduri, iar p este probabilitatea ca o muchie sa existe in graf. Practic, fiecare muchie a grafului complet cu n noduri are sansa p sa apara in graful aleator $G(n, p)$.

    (b) $G(n, m)$ unde n este numarul de noduri, iar m este lista primelor m muchii dintr-o permutare aleatoarea a tuturor muchiilor posibile dintr-un graf complet cu n noduri.

3 Algoritmi DFS Incrementali[3]

În prima parte a articolului, sunt prezentate modurile de funcționare ale algoritmilor incrementali cunoscuți în mod curent pentru generarea arborelui DFS.

3.1 SDFS si SDFS-Int

Primul, și cel mai evident algoritm, presupune că la adăugarea unei muchii noi, să rulăm algoritmul obișnuit de DFS pe graful actualizat, și să înlocuim arborele vechi cu cel nou obținut. Acest procedeu este numit în continuare SDFS (Static DFS).

O variantă a acestui algoritm, SDFS-Int (Static DFS with interrupt), obține performanțe mult mai bune pe grafurile aleatoare decât SDFS. Singura diferență dintre SDFS-Int este că cel de-al doilea se oprește în momentul în care toate nodurile grafului au fost marcate ca fiind vizitate.

3.2 FDFS

Următorul algoritm țintește menținerea eficientă a DFS Tree-ului pe directed acyclic graphs (DAG).

Pentru a putea funcționa, FDFS reține în arborele pe care îl generează (notat T) numerotarea parcurgerii post-order a nodurilor sale, numită în continuare DFN. La adăugarea unei muchii noi (x, y) în graful G, FDFS consultă această numerotare pentru a verifica dacă muchia nouă este *anti-cross edge* (muchie de la stânga la dreapta în arbore, deci dacă $DFN[x] < DFN[y]$).

Dacă nu este, atunci nu vor surveni modificări majore ale arborelui, iar FDFS poate să actualizeze graful cu ușurință, fără recalculari.

Dacă muchia (x, y) este *anti-cross edge*, atunci FDFS va recalcula o parte din arbore. Pe scurt, se scot din arbore toate nodurile cu scorul DFN între valorile DFN ale lui x și y. Aceste noduri sunt cele afectate de către adăugarea muchiei (x, y). Practic, vor fi selectate nodurile aflate în stânga $path(LCA(x, y), y)$, sau dreapta $path(LCA(x, y), x)$. În continuare, se va recalcula folosind DFS acest arbore parțial, iar rezultatul său va fi atașat nodului y din T prin muchia (x, y). DFN-urile nodurilor afectate vor fi, de asemenea, recalculate, pentru a reflecta noile modificări și a pregăti arborele pentru inserțiile următoare.

Deși original gândit pentru directed acyclic graphs, acest algoritm poate fi modificat ușor astfel încât să funcționeze pe orice graf orientat, modificarea având loc la pasul de recalculare a arborelui parțial la întâlnirea unui anti-cross edge. Pe scurt, mulțimea nodurilor ce sunt candidați pentru recalculare este extinsă astfel încât să includă tot subarborele din care face nodul y parte, nu doar nodurile cu valorile DFN între $DFN[x]$ și $DFN[y]$. Pentru acest algoritm nu este cunoscut un bound mai bun decât $O(m^2)$.

3.3 ADFS

ADFS este un algoritm incremental DFS care functioneaza pe grafuri neorientate. Acesta mentine in paralel o structura de date care poate raspunde eficient la query-uri de tip LCA, respectiv level ancestor(LA) (unde $LA(x, k)$ returneaza parintele nodului x aflat la nivelul k in arbore). Procedeul de functionare se inspira din FDFS: la inserarea unei muchii (x, y), ADFS incearca sa determine daca muchia este cross-edge prin calcularea $w = LCA(x, y)$.

Daca $w \neq x$ si $w \neq y$, atunci avem de a face cu o muchie cross-edge, asadar o parte din graf va trebui reconstruita. Algoritmul de reconstructie este urmatorul:

1. Fie u si v copii ai lui w astfel incat $x \in T(u)$ si $y \in T(v)$ (adica: u si v sunt radacinile subarborilor din care fac parte x si y).

2. Fie nivelul lui x in graf mai mare decat nivelul lui y. (x este mai jos decat y in arbore)

3. ADFS va reconstrui $T(v)$ si il va atasa la (x, y) astfel:

    (a) Va inversa $path(y, v)$, fapt ce transforma multe muchii back-edge din $T(v)$ in cross-edge.

    (b) Va colecta aceste muchii cross-edge nou obtinute si le va insera in graf folosind acelasi procedeu.

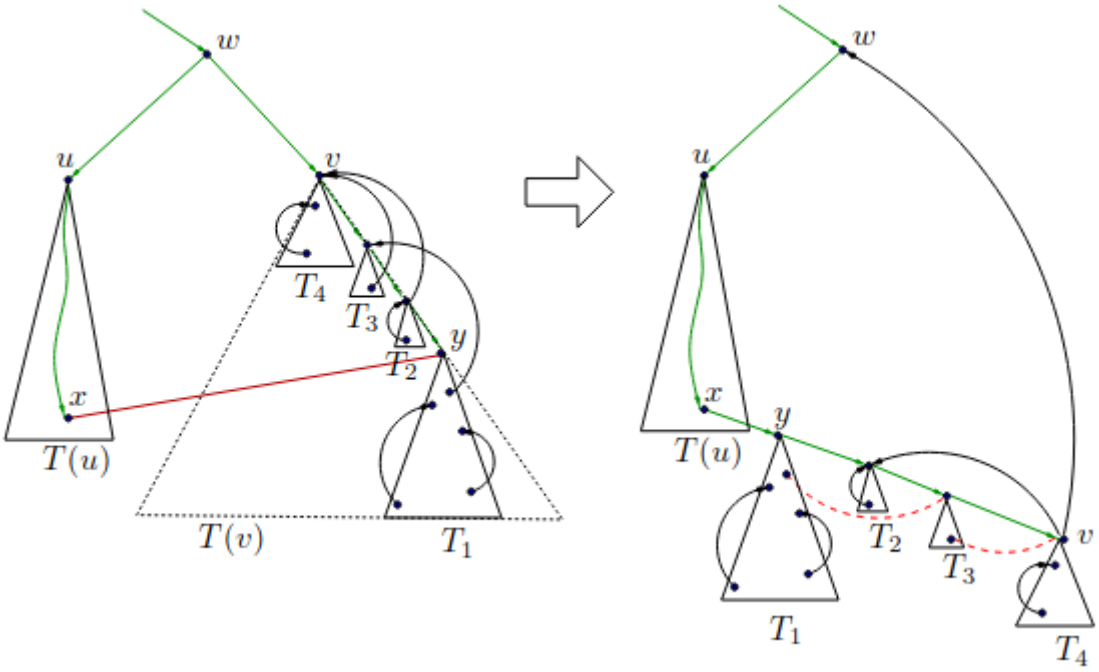

Figure 1: Pasul de "inversare" aplicat pe $path(v, y)$ transforma drumul in graf in (y, v) dupa ce y devine radacina.[11] Muchiile punctate cu rosu devin la randul lor cross edges iar algoritmul va trebui aplicat si pe acestea, obtinandu-se un efect de cascada

Putine detalii despre pasul de "inversare", gasite in paper-ul care propune algoritmul acesta[10]:

- Se bazeaza pe un principiu pe care ei il numesc "monotonic fall": la adaugarea unei muchii noi (x, y), algoritmul garanteaza ca toate nodurile afectate de restructurarea arborelui DFS, fie vor ramane pe nivelul lor original in arbore, fie vor "coborî" (le va creste nivelul).
- Prin "inversarea" $path(y, v)$, se intelege rearanjarea subarborelui $T(v)$ astfel incat radacina sa să devina y.

Exista doua variante ale ADFS, numite ADFS1 si ADFS2, singura diferenta intre cele doua fiind ordinea in care sunt procesate cross edge-urile generate de algoritm. ADFS1 le proceseaza arbitrar, in timp ce ADFS2 le mentine intr-o structura de date non-triviala, prioritizand muchiile cu noduri aflate la un nivel mai inalt in graf.

Algorithm	**Graph**	**Update time**	**Total time**
SDFS [53]	Any	$O(m)$	$O(m^2)$
SDFS-Int [29]	Random	$O(n \log n)$ expected	$O(mn \log n)$ expected
FDFS [19]	DAG	$O(n)$ amortized	$O(mn)$
ADFS1 [6]	Undirected	$O(n^{3/2}/\sqrt{m})$ amortized	$O(n^{3/2}\sqrt{m})$
ADFS2 [6]	Undirected	$O(n^2/m)$ amortized	$O(n^2)$
WDFS [4]	Undirected	$O(n \log^3 n)$	$O(mn \log^3 n)$

Figure 2: Tabel cu complexitatile teoretice ale algoritmilor comparati[4]

3.4 WDFS

In timp ce toti ceilalti algoritmi de mai sus au worst-case update time tot de $O(m)$, WDFS ofera garantia unui update time de $O(nlog^3n)$, prin mentinerea unei structuri de date care permite reconstruirea eficienta a DFS tree-ului dupa o operatie de update.

Cum construirea acestei structuri se face in $O(m)$ timp, ea este reconstruita periodic, la un interval de m/n operatii, nu dupa fiecare update.

Acest algoritm este relevant din perspective teoretice, fiind singurul care ofera un worst-case update time mai bun decat $O(m)$. In practica, el ofera performante mai slabe overall, chiar si decat SDFS.

4 Analiza experimentala [5]

In aceasta sectiune, autorii au comparat performantele reale ale algoritmilor SDFS, SDFS-Int, ADFS si WDFS. Au fost folosite atat grafuri generate aleator, cat si dataset-uri cu exemple de grafuri reale. Metrica pentru masurarea performantei folosita este numarul de muchii procesate, nu timpul.

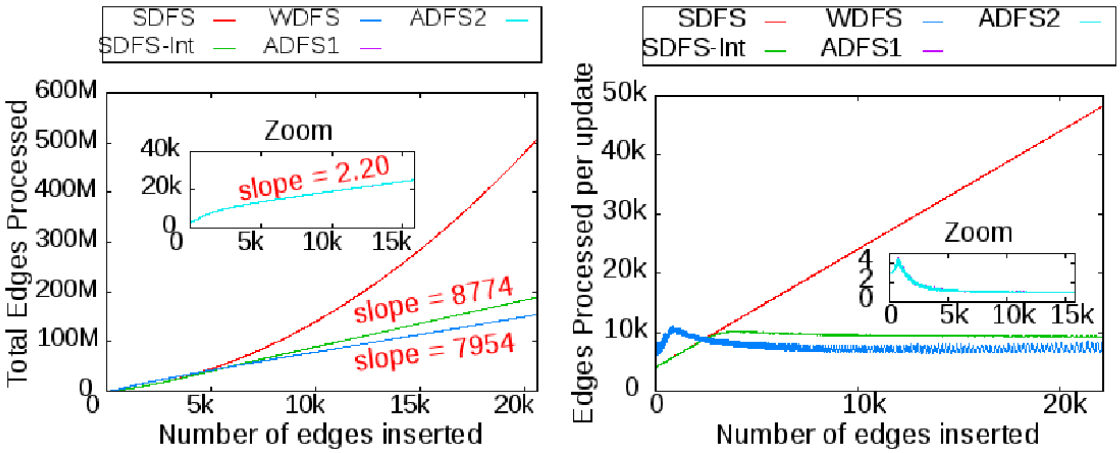

Figure 3: Comparatii ale algoritmilor DFS incrementali [6]

In urma experimentarii cu diferite grafuri aleatoare, autorii au observat ca algoritmii SDFS, SDFS-Int si WDFS au performante similare cu complexitatile lor teoretice. In schimb, in cazul ADFS1 si ADFS2, s-a observat o anomalie: ambii algoritmi au performante similare in practica, in ciuda complexitatilor diferite ($O(n^{3/2}\sqrt{m})$ versus $O(n^2)/m$). Totodata, acesti algoritmi sunt mult mai rapizi decat ceilalti cu care sunt comparati. Cea mai surprinzatoare observatie este rezultata din graficul care compara raportul dintre cate muchii sunt inserate in graf cu cate muchii sunt procesate per update. In cazul ADFS, cu cat graful devine mai dens, cu atat numarul de muchii procesate per operatie de update scade, tinzand chiar asimptotic spre zero.

5 Forma arborelui DFS [7]

In continuare, autorii se concentreaza pe a incerca sa determine ce se intampla mai exact in cazul ADFS, si descopera o proprietate a formei arborelui DFS pe care ei o numesc "broomstick property".

Prima observatie pe care ei o fac este ca dintre toti cei patru algoritmi comparati, ADFS reconstruieste arborele doar la inserarea unui cross-edge. In continuare, ei noteaza cu p_c probabilitatea ca intr-un graf aleator, o muchie nou inserata sa fie cross-edge, si studiaza experimental cum se schimba p_c in functie de numarul de muchii din graf.

Se observa experimental ca p_c scade cu cat graful devine mai dens, si se propune teoria ca ADFS are performante mult mai bune decat ceilalti algoritmi din acest motiv.

5.1 Broomstick property

Se observa ca in arborele DFS asociat unui graf aleator exista un drum fara ramificari (adica: fiecare nod al drumului are un singur nod copil). Cu putina imaginatie, acest drum poate fi vazut ca fiind "coada" unei maturi (broomstick), celelalte noduri si muchii din graf constituind "perii" (bristles).

Fie l_s lungimea drumului "cozii de matura". In continuare, autorii au incercat sa determine in mod probabilist lungimea l_s in functie de numarul de muchii prezente in graf.

Pana in momentul in care graful devine conex (deci, dupa insertia a aprox. $nlogn$ muchii), l_s este 0, caci arborele DFS va avea radacina virtuala s care va fi conectata printr-o muchie fiecarei componente conexe a grafului.

Dupa aceea, este demonstrat ca pentru un graf aleator $G(n,p)$, cu $p = (logn_0 + c)/n_0$, pentru orice $n_0 < n$ si $c \geq 1$, exista un drum fara ramificatii de lungime minim $n - n_0$ in G cu probabilitate macar $1 - 2e^{-c}$.

Ce am inteles eu de aici e ca intai se calculeaza un bound mai putin strict pentru lungimea "cozii", apoi se cauta unul mai tight, obtinandu-se rezultatul ca probabilitatea ca arborele DFS al grafului $G(n,p)$ are forma de matura, si are coada de lungime $\geq n - n_0$, este de macar $1 - 3e^{-c}$, aceasta probabilitate tinzand catre 1 pentru orice functie monoton crescatoare $c(n)$, cu $c(n) > 1$ pentru orice n.

In final, se mentioneaza ca rezultatul obtinut pe $G(n,p)$ se aplica si pe $G(n,m)$.

Prin analiza experimentala, se confirma analiza teoretica a autorilor, observandu-se ca partea de "coada" apare dupa inserarea a aprox $nlogn$ muchii (din felul in care este definit graful aleator). Coada ajunge la 90% din lungimea ei maxima dupa inserarea a aprox $3nlogn$ muchii, dupa care cresterea in dimensiuni stagneaza, atingand lungimea maxima dupa aprox $O(n^2)$ insertii.

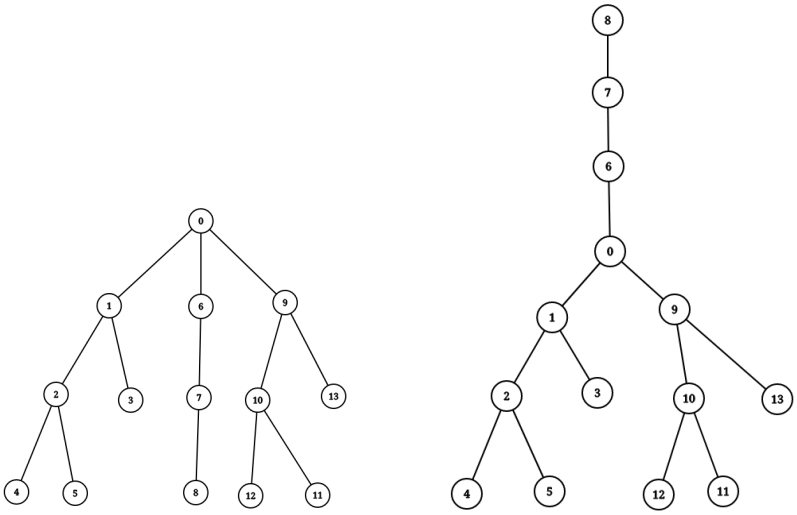

Figure 4: Cel mai lung drum in arborele DFS fara ramificatii devine "coada" maturii, restul fiind "perii"

Observatia ca arborele ia forma unei maturi este relevanta deoarece vazand arborele astfel, orice muchie nou-adaugata care are macar unul dintre capete un nod al cozii este in mod evident back-edge.

Cum muchiile cross-edge apar doar intre "perii" arborelui, iar multimea de muchii ce constituie perii arborelui descreste cu cat graful devine mai dens, cu atat devine mai usoara actualizarea arborelui DFS in cazul algoritmului ADFS, deoarece acesta nu face modificari asupra "cozii".

Sunt identificate doua motive pentru care ADFS are performanta mai buna decat ceilalti algoritmi, ce rezulta ca si consecinta a "broomstick property":

- P1. ADFS reconstruieste arborele DFS doar la adaugarea unui cross-edge.
- P2. ADFS modifica doar "perii" structurii, lasand "coada" intacta.

6 Un nou algoritm pentru grafurile aleatoare[8]

In urma observarii acestor proprietati, autorii propun o modificare a algoritmului SDFS care sa includa cele doua observatii cruciale ce fac ADFS sa aiba performante asa de bune, cu o implementare mai simpla.

Prima varianta, SDFS2, propune modificarea urmatoare: atunci cand se adauga o noua muchie in graf, daca este cross-edge, nodurile care fac parte din "perii" grafului vor fi marcati ca nevizitiati si algoritmul DFS va fi rerulat, cu punctul de start in ultimul nod al "cozii" (primul nod din care incep "perii"). Totodata, se va avea grija ca muchiile care leaga nodurile din "perii" arborelui de "coada" in graful G sa nu fie luate in calcul (fiind back-edges).

Acest algoritm simplu de implementat atinge performante mai bune decat SDFS, si comparabile cu ADFS, pe grafurile aleatoare, avand o complexitate de $O(n^2 log^2 n)$.

7 Teste pe grafuri reale[9]

Toate studiile de mai sus au fost facute pe grafuri aleatoare, de forma $G(n,m)$ sau $G(n,p)$. In practica, grafurile nu sunt aleatoare, asadar este important sa studiem performanta algoritmilor pe cazuri concrete, precum retele de prietenii, retele de calculatoare etc.

Cum grafurile care tind sa apara in aplicatii practice nu sunt foarte dense, iar SDFS2 reconstruieste intregul arbore pana cand acesta incepe sa capete structura de matura pentru care este optimizat, in practica, pe grafuri reale, ADFS are performante mult mai bune decat SDFS2. Autorii propun o alta varianta, SDFS3, care sa imbunatateasca performanta pentru grafurile reale.

SDFS3 adauga o restrictie suplimentara: la adaugarea unei noi muchii cross-edge (x, y) (deci $w = LCA(x, y)$, $w \neq x$, $w \neq y$, $T(x)$ subarborele din care face parte x, $T(y)$ subarborele din care face parte y), SDFS3 va reconstrui doar subarborele cu numar mai mic de noduri. La fel, nodurile acestuia vor fi marcate ca nevizitate, se va rerula DFS, iar arborele rezultat va fi atasat la muchia (x, y).

Aceasta euristica imbunatateste semnificativ performanta algoritmului SDFS2, SDFS3 fiind de 10 ori mai rapid decat acesta in practica. Totusi, nu ajunge la performantele lui ADFS, in cazul grafurilor neorientate. In cazul grafurilor orientate, SDFS2/3 ofera performante similare cu FDFS.

8 Concluzii

In urma investigatiilor autorilor, au fost obtinute noi informatii despre importanta structurii de forma de matura (broomstick structure) a arborilor DFS, si a implicatiilor acesteia. Au fost propusi doi algoritmi noi, SDFS2 si SDFS3, cu performanta foarte buna in cazul grafurilor orientate. Desi in cazul grafurilor neorientate, ADFS este superior din punct de vedere al performantei, SDFS3 se remarca prin simplitatea implementarii, si este valoros prin acest lucru.

References

[1] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018.

[2] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 55. SIAM, 2018.

[3] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 56. SIAM, 2018.

[4] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 54. SIAM, 2018.

[5] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 57. SIAM, 2018.

[6] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 57. SIAM, 2018.

[7] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 58–60. SIAM, 2018.

[8] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 61. SIAM, 2018.

[9] S. Baswana, A. Goel, and S. Khan. Incremental dfs algorithms: a theoretical and experimental study. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 63–64. SIAM, 2018.

[10] S. Baswana and S. Khan. Incremental algorithm for maintaining a dfs tree for undirected graphs. *Algorithmica*, 79(2):466–483, 2017.

[11] S. Baswana and S. Khan. Incremental algorithm for maintaining a dfs tree for undirected graphs. *Algorithmica*, 79(2):472, 2017.