IN2010 - Algoritmer (pseudokode)

Yrjar Vederhus
yrjarv@ifi.uio.no

Contents

Introduksjon

Rett-frem søk

Traverserer en liste fra start til slutt, for å finne ut om et element er i lista.

Kjøretidskompleksitet: `O(n)`

```
Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner true ellers false

function search(A, x)
    for i = 0 to |A| - 1; do
        if A[i] == x; then
            return true
    return false
```

Binærsøk

Fungerer kun på sorterte lister. Går til midten av lista, sjekker om elementet der er større eller mindre enn mål-elementet. Går deretter til midten av sub-lista som ligger i riktig retning.

Kjøretidskompleksitet: `O(log n)`

```
Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returnerer true ellers false

function binarySearch(A, x)
    low = 0
    high = |A| - 1
    while low < high; do
        i = lower((low + high) / 2)
        if A[i] == x; then
            return true
        if A[i] < x; then
            low = i + 1
            continue
        if A[i] > x; then
            high = i - 1
            continue
    return false
```

Trær

Dybde

Finner dybden til et tre rekursivt. Dybden til en node er 1 mer enn dybden til foreldrenoden. Roden har alltid dybde 0. Siden vi tillater et tomt tre gir vi forelderen til roten dybde -1.

Kjøretidskompleksitet: `O(h)` der `h` er høyden til treet

```
Input: En node v
Output: Dybden av noden

function depth(v)
    if v == null; then
        return -1
    return 1 + depth(v.parent)
```

Høyde

Høyden av et tre er den største avstanden til en etterkommer av rotnoden - altså dybden til den dypese løvnoden.

Algoritmen går gjennom alle barn og endrer høyden avhengig av hvor dypt det dypeste barnet så langt er.

Kjøretidskompleksitet: `O(n)` der `n` er antall noder i treet

```
Input: En node v
Output: Høyden av noden

function height(v)
    max_height = -1
    if v == null; then
        return max_height
    for child in v.children:
        h = max(max_height, height(child))
    return 1 + max_height
```

Preorder

Traverserer et tre og gjør en operasjon `O` på hvert element.

Kjøretidskompleksitet: `O(n)` der `n` er antall noder i treet.

```
Input: En node v (som ikke er null), en funksjon O som skal utføres på hvert
       element i treet
Output: Ingen

function preorder(v, O)
```

```
    if v == null; then
        return
    O(v)
    for child in v.children; do
        preorder(child)
```

Postorder

Traverserer et tre og gjør en operasjon `O` på hvert element.

Kjøretidskompleksitet: `O(n)` der `n` er antall noder i treet.

```
Input: En node v (som ikke er null), en funksjon O som skal utføres på hvert
       element i treet
Output: Ingen

function postorder(v, O)
    if v == null; then
        return
    for child in v.children; do
        postorder(child)
    O(v)
```

Innsetting i binært søketre

Setter inn et element på riktig plass i et binært søketre.

Kjøretidskompleksitet: `O(h)` der `h` er høyden til treet. Hvis treet er balansert, blir det `O(log n)`

```
Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

function insert(v, x)
    if v == null; then
        return new Node(x)
    if x < v.element; then
        v.left = insert(v.left, x)
        return v
    else if x > v.element; then
        v.right = insert(v.right, x)
    return v
```

Oppslag

Finner og returnerer noden som inneholder et element i et binært søketre hvis det finnes, eller null hvis det ikke finnes i treet.

Kjøretidskompleksitet: Samme som ved innsetting

```
Input: En node v og et element x
Output: Dersom x forekommer i en node u som en etterkommer av v, returnerer u,
        ellers null

function search(v, x)
    if v == null; then
        return null
    if v.element == x; then
        return v
    if x < v.element; then
        return search(v.left, x)
    if x > v.element; then
        return search(v.right, x)
    return null
```

Finne minste element i et binært søketre

Går til venstre hele tiden, siden det minste elementet er nederst til venstre i det binære søketreet.

Kjøretidskompleksitet: `O(h)`, der `h` er høyden til treet

```
Input: En node v
Output: Noden som inneholder den minste etterkommeren av v

function findMin(v)
    while v.left != null; do
        v = v.left
    return v
```

Sletting i binære søketrær

Søker etter noden som skal slettes, bytter ut denne noden sin verdi med verdien til den minste noden i dens høyre subtre, og sletter (rekursivt) ovennevnte minste node.

Kjøretidskompleksitet: `O(h)`

```
Input: En node v og et element x
Output: Et binært søketre hvor noden som har verdien x er fjernet

function remove(v, x)
    if v == null; then
        return null
    if x < v.element; then
        v.left = remove(v.left, x)
        return v
    if x > v.element; then
```

```
        v.right = remove(v.right, x)
        return v
    if v.left == null; then
        return v.right
    if v.right == null; then
        return v.left

    # This is only reached if x == v.element
    smallest_child = findMin(v.right)
    v.element = smallest_child.element
    v.right = remove(v.right, child.element)
```

Balanserte søketrær

Venstrerotasjon av et binært søketre

```
   1        ->      3
 /   \            /   \
0     3          1     5
     / \        / \   / \
    2   5      0   2 4   6
       / \
      4   6
```

"Henger opp" treet etter det høyre barnet av roten

Kjøretidskompleksitet: `O(1)`

```
Input: En node v
Output: Et rotert tre sin rotnode

function leftRotate(v)
    new_root = v.right
    middle_child = new_root.left

    new_root.left = v
    v.right = middle_child

    setHeight(v)
    setHeight(new_root)

    return new_root
```

Høyrerotasjon av et binært søketre

```
       7    ->      5
     /   \        /   \
```

```
    5     8      3     7
   / \          / \   / \
  3   6        1   4 6   8
 / \
1   4
```

"Henger opp" treet etter det venstre barnet av roten

Kjøretidskompleksitet: `O(1)`

```
Input: En node v
Output: Et rotert tre sin rotnode

function rightRotate(v)
    new_root = v.left
    middle_child = new_root.right

    new_root.right = v
    v.left = middle_child

    setHeight(v)
    setHeight(new_root)

    return new_root
```

Balansefaktor

Bruker `height` til å finne differansen mellom høydene til barna til en node.

Kjøretidskompleksitet: `O(1)` hvis `height(v)` er `O(1)`

```
Input: En node v (rot til treet)
Output: Høydeforskjellen på v sitt venstre- og høyrebarn

function balanceFactor(v)
    if v == null; then
        return 0
    return height(v.left) - height(v.right)
```

Balansering av AVL-tre

Balanserer et AVL-tre ved å dobbelt-rotere ved behov, og ellers rotere som nødvendig.

Kjøretidskompleksitet: `O(1)`

```
Input: En node v
Output: En rotnode til et balansert AVL-tre
```

```
function balance(v)
    if balanceFactor(v) < -1; then              # Right heavy
        if balanceFactor(v.right) > 0; then     # Right subtree is left heavy
            v.right = rightRotate(v.right)      # 1st part of double rotation
        return leftRotate(v)
    if balanceFactor(v) > 1; then               # Left heavy
        if balanceFactor(v.left) < 0; then      # Left subtree is right heavy
            v.left = leftRotate(v.left)         # 1st part of double rotation
        return rightRotate(v)

    # It is balanced now
    return v
```

Innsetting i AVL-tre

Setter inn rekursivt (til “bunnen” av AVL-treet), og balanserer det.

Kjøretidskompeksitet: `O(h)`

```
Input: En node v og et element x
Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

function insert(v, x)
    v = bst.insert(v, x)
    setHeight(v)
    return balance(v)
```

Sletting i AVL-trær

Sletter rekursivt og balanserer.

```
Input: En node v og et element x
Output: Et balansert AVL tre uten en node hvor element er x

function remove(v, x)
    v = bst.remove(v, x)
    setHeight(v)
    return balance(v)
```

Innsetting i binære heaps

Setter et element på den første ledige plassen, og bobler opp ved behov.

```
Input: Et array A som representerer en heap, og et element x
Output: Et array som representerer en heap, som inneholder x

function insert(a, x)
    a[a.numElements] = x
```

```
    index = a.numElements
    a.numElements += 1

    while index > 0 && a[index] < a[parentOf(index)]; do
        swap(a[index], a[parentOf(index)])
        index = parentOf(index)
```

Fjerning av minste element i binære heaps

Flytter det siste elementet først, og flytter det nedover til det er på rett plass. Dette gjøres ved å alltid swappe med det venstre barnet helt til både høyre og venstre barn er større enn elementet. Hvis høyre barn er mindre enn venstre barn, swapper vi i stedet med det høyre barn.

Dette betyr at ned-flyttinga skjer ved å alltid swappe med det minste barnet, så lenge begge barna er større enn elementet selv.

Kjøretidskompleksitet: `O(h)`

```
Input: Et array a som representerer en ikke-tom heap
Output: Et array som representerer en heap, men hvor det minste elementet fra a
        er fjernet, og elementet som ble fjernet

function removeMin(a)
    smallest_element = a[0]
    a[0] = a[a.numElements - 1]

    working_index = 0
    while leftOf(working_index) < a.numElements - 1; do # In "bounds"
        to_be_swapped_index = leftOf(working_index)
        if (rightOf(working_index) < a.numElements - 1 # In "bounds"
            && a[rightOf(working_index)] < a[to_be_swapped_index]
            # Righ child < left child
           ); then
            to_be_swapped_index = rightOf(working_index)
        if a[working_index] <= a[to_be_swapped_index]; then # Correctly ordered
            break
        swap(a[working_index], a[to_be_swapped_index])
        working_index = to_be_swapped_index

    return a, smallest_element
```

Bygge Huffman-trær

Legger til alle tegn i en prioritetskø (“heap”), og popper to og to tegn, og legger til en ny node som har de to gamle nodene som barn.

Kjøretidskompleksitet: `O(n log n)`

```
Input: En mengde characters med par (symbol, frequency)
Output: Et Huffman-tre

function huffman(characters)
    queue = new PriorityQueue()
    for pair in characters:
        insert(queue, new Node(symbol, frequency, null, null))
    while queue.numElements > 1; do
        node1 = removeMin(queue)
        node2 = removeMin(queue)
        sum_frequencies = node1.frequency + node2.frequency
        insert(queue, new Node(null, sum_frequencies, v1, v2))
    return removeMin(queue)
```

Sortering

Bubble sort

Sammenligner par av elementer og bytter plass på dem hvis de er i feil rekkefølge. Gjør det `n` ganger.

Kjøretidskompleksitet: `O(n ** 2)`

```
Input: Et array a med n elementer
Output: Et sortert array med de samme elementene

function bubbleSort(a)
    for i in {0 ... n-2}; do
        for j in {0 ... n-i-2}; do
            if a[j] > a[j + 1]; then
                swap(a[j], a[j + 1])
    return a
```

Kan optimaliseres med early return:

```
Input: Et array a med n elementer
Output: Et sortert array med de samme elementene

function bubbleSort(a)
    for i in {0 ... n-2}; do
        if !has_had_swaps; then
            return a
        has_had_swaps = false

        for j in {0 ... n-i-2}; do
            if a[j] > a[j + 1]; then
                swap(a[j], a[j + 1])
```

```
            has_had_swaps = true
    return a
```

Selection sort

Finner det minste elementet i den usorterte delen av arrayen, og flytter det til bakerst i den sorterte delen av arrayen.

Kjøretidskompleksitet: `O(n ** 2)`

```
Input: Et array a med n elementer
Output: Et sortert array med de samme elementene

function selectionSort(a)
    for num_sorted in {0 ... n-1}; do
        index_of_smallest = num_sorted
        for j in {num_sorted+1 ... n-1}; do
            if a[j] < a[index_of_smallest]; then
                index_of_smallest = j
        if num_sorted != index_of_smallest; then
            swap(a[num_sorted], a[index_of_smallest])
    return a
```

Insertion sort

Flytter det minste elementet i den usorterte delen av arrayet til den første ledige plassen til venstre i arrayet.

Kjøretidskompleksitet: `O(n ** 2)`

```
Input: Et array a med n elementer
Output: Et sortert array med de samme elementene

function insertionSort(a)
    for i in {1 ... n-1}; do
        j = i
        while j > 0 && a[j - 1] > a[j]; do
            swap(a[j], a[j - 1])
            j--
    return a
```

Bubble down

Brukes for å bygge en max-heap i heapsort

Kjøretidskompleksitet: `O(log n)`

```
Input: En (uferdig) heap a med n elementer der root er indeks til roten
Output: En mindre uferdig heap
```

```
function bubbleDown(a, root, n)
    largest = root
    left = 2*root + 1
    right = 2*root + 2

    if left < n && a[largest] < a[left]; then        # Left child is larger
        largest = left

    if right < n && a[largest] < a[right]; then      # Right child is larger
        largest = right

    if root != largest; then                            # A child is larger
        swap(a[root], a[largest])
        bubbleDown(a, largest, n)         # Recursive call from the larger child
```

Heapsort

Bygger en max-heap, og popper fra den for å lage en sortert array bakfra.

Kjøretidskompleksitet: `O(n log n)`

```
Input: Et array a med n elementer
Output: Et sortert array med de samme elementene

function heapsort(a)
    for root_index in {lower(n/2) ... 0}; do
        bubbleDown(a, root_index, n)

    for index_to_fill in {n-1 ... 0}; do
        swap(a[0], a[index_to_fill])
        bubbleDown(a, 0, index_to_fille)
```

Merge

Fletter to arrayer sammen.

Kjøretidskompleksitet: `O(n)`, der `n` er lengden til de to input-arrayene kombinert

```
Input: To sorterte arrayer a1 og a2, og et array a der |a| = |a1| + |a2|
Output: Et sortert array med elementene fra a1 og a2

function merge(a1, a2, a)
    i1 = 0
    i2 = 0

    while i1 < |a1| && i2 < |a2|; do
        if a1[i1] <= a2[i2]; then
```

```
        a[i1 + i2] = a1[i1]
        i++
    else
        a[i1 + i2] = a2[i2]
        j++
while i1 < |a1|; do
    a[i1 + i2] = a1[i1]
    i++
while i2 < |a2|; do
    a[i1 + i2] = a2[i2]
    a2++

return a
```

Merge sort

Divide and conquer. Splitter arrayet i 2, merge-sorterer de to delene, og kombinerer igjen.

Kjøretidskompleksitet: `O(n log n)`

```
Input: Et array a med n elementer
Output: Et sortert array med de samme elementene

function mergeSort(a)
    if n <= 1; then
        return a
    i = lower(n/2)
    a1 = mergeSort(a[0 ... i-1])
    a2 = mergeSort(a[i ... n-1])
    return merge(a1, a2, a)
```

Quicksort

Vi antar at vi har en funksjon `choosePivot` som velger en pivot-indeks basert på en `O(1)` algoritme, og som gir en effektiv sortering.

Alle elementene i arrayet som er mindre enn pivoten, plasseres før pivoten, og tilsvarende for større elementer som plasseres etter pivoten. Dette gjøres da rekursivt.

For å sortere et array `a` kaller vi `quicksort(a, 0, n-1)`.

Kjøretidskompleksitet: `O(n ** 2)` (verste tilfelle, beste tilfelle er `n log n`)

```
Input: Et array a med n elementer, og indeksene low og high
Output: Et sortert array med de samme elementene

function quicksort(a, low, high)
```

```
    if low >= high; then  # The boundaries have met or crossed
        return a

    pivot_index = choosePivot(a, low, high)
    swap(a[pivot_index], a[high])

    pivot = a[high]
    left = low
    right = high - 1

    while left <= right; do
        while left <= right && a[left] <= pivot; do
            left++
        while right >= left && a[right] >= pivot; do
            right--
        if left < right; then
            swap(a[left], a[right])
    swap(a[left], a[high])

    a = quicksort(a, low, left - 1)
    a = quicksort(a, left + 1, high)

    return a
```

Bucket sort

Vi sorterer alle elementene inn i bøtter (lenkelister) som ligger i et array. Merk at denne ikke er in-place.

Kjøretidskompleksitet: `O(n)`

```
Input: Et array a med n elementer, og antallet N bøtter
Output: Et array med de samme elementene sortert etter nøkler

function bucketSort(a)
    buckets = new Array(N)
    for i in {0 ... n-1}; do
        key = (the key associated with a[i])
        buckets[key].append(a[i])

    i = 0
    for key in (possible keys); do
        for x in buckets[key]; do
            a[i] = x
            i++

    return a
```

En god hashfunksjon på strenger

Faktoren `31` er valgt siden det er et primtall som man også kan multiplisere tall med ved å bitshifte tallene 5 til venstre og trekke fra tallet.

```
Input: En streng string og et positivt heltall max_hash
Output: Et heltall mellom 0 og max_hash

function hash(string, max_hash)
    hash = 0
    for character in string; do
        hash = 31*h + charToInt(character)
    return hash MOD n
```

Grafer

Dybde-først-søk (rekursivt)

Bruker call-stacken til å lagre hvilke veier man skal prøve.

Kjøretidskompleksitet: `O(|V| + |E|)`

```
Input: En graf G = (V, E), en startnode u, og en mengde visited med besøkte
       noder
Output: Ingenting

function dfsVisit(G, u, visited)
    visited.add(u)
    for (u, v) in E; do
        if v in visited; then
            continue
        dfsVisit(G, v, visited)
```

Dybde-først-søk (iterativt)

Bruker en stack til å lagre hvilke veier man skal prøve.

Kjøretidskompleksitet: `O(|V| + |E|)`

```
Input: En graf G = (V, E), en startnode u, og en mengde visited med besøkte
       noder
Output: Ingenting

function dfsVisit(G, u, visited)
    stack = new Stack()
    stack.add(u)

    while |stack| > 0; do
```

```
        u = stack.pop()
        if u in visited; then
            continue
        visited.add(u)
        for (u, v) in E; do
            stack.push(v)
```

Bredde-først-søk

Bruker en kø til å besøke utover i en “ring” fra startnoden.

Kjøretidskompleksitet: `O(|V| + |E|)`

```
Input: En graf G = (V, E), en startnode s, og en mengde visited med besøkte
       noder
Output: Ingenting

function bfsVisit(G, s, visited)
    visited.add(s)
    queue = new Queue()
    queue.enqueue(s)

    while |queue| > 0; do
        u = queue.dequeue()
        for (u, v) in E; do
            if v in visited; then
                continue
            visited.add(v)
            queue.enqueue(v)
```

Topologisk sortering

En topologisk sortering putter rotnodene lengst mulig framme i et array, og de nederste nodene helt på slutten i arrayet.

Kjøretidskompleksitet: `O(|V| + |E|)`

```
Input: En rettet graf G = (V, E)
Output: En array med en topologisk sortering av nodene i G

function topSort(G)
    stack = new Stack()
    output = new List()

    for v in V; do
        if v.indegree == 0; then
            stack.push(v)
```

```
    while |stack| > 0; do
        u = stack.pop()
        output.append(u)
        for (u, v) in E; do
            E.remove(u, v)
            v.indegree--
            if v.indegree == 0; then
                stack.push(v)
    if |output| < |V|; then
        error "G contains a cycle"
    return output
```

Topologisk sortering med DFS

```
Input: En rettet asyklisk graf G = (V, E)
Output: En array med en topologisk sortering av nodene i G

function dfsTopSort(G)
    stack = Stack()
    visited = Set()
    for u in V; do
        if u in visited; then
            continue
        stack = dfsVisit(G, u, visited, stack)
    return stack

function dfsVisit(G, u, visited, stack)
    visited.add(u)
    for (u, v) in E; do
        if v in visited; then
            continue
        stack = dfsVisit(G, u, visited, stack)
    stack.push(u)
    return stack
```

Vektede grafer

Dijkstra (tradisjonell)

BFS-aktig men bruker en prioritetskø, og endrer prioritet basert på total avstand til denne noden. Gir en tabell med korteste distanse fra startnode til alle noder.

Kjøretidskomplektsitet: `O(|E| log |V|)` hvis grafen er sammenhengende, `O((|V| + |E|) log |V|)` hvis ikke.

```
Input: En vektet graf G = (V, E) med vektfunksjon weight og en startnode s
```

```
Output: Et map som angir lengden på korteste vei fra s til alle noder i G

function dijkstra(G, s)
    queue = new Queue()
    distances = new Map()

    for v in V; do
        distances[v] = infinity
        insert(queue, v) with priority infinity

    distances[s] = 0
    decreasePriority(queue, s, 0)

    while |queue| > 0; do
        u = removeMin(queue)
        for (u, v) in E; do
            new_distance = distances[u] + weight(u, v)
            if new_distance < distances[v]; then
                distances[v] = new_distance
                decreasePriority(queue, v, new_distance)

    return distances
```

Dijkstra (uten `decreasePriority`)

Som tradisjonell Dijkstra, men …

Kjøretidskomplektsitet: `O(|E| log |V|)` hvis grafen er sammenhengende, `O((|V| + |E|) log |V|)` hvis ikke.

```
Input: En vektet graf G = (V, E) med vektfunksjon weight og en startnode s
Output: Et map som angir lengden på korteste vei fra s til alle noder i G

function dijkstra(G, s)
    queue = new Queue()
    dist = new Map()
    visited = new Set()

    distances[s] = 0
    insert(queue, s) with priority 0

    while |queue| > 0; do
        u = removeMin(queue)
        if u in visited; then
            continue
        visited.add(u)
        for (u, v) in E; do
```

```
            new_distance = distances[u] + weight(u, v)
            if new_distance < distances[v]; then
                distances[v] = new_distance
                insert(queue, v) with priority new_distance

    return distances
```

Bellman ford

Takler negative vekter og hindrer sykler, i motsetning til Dijkstra.

Kjøretidskompleksitet: `O(|V| * |E|) = O(|V| ** 3)`

```
Input: En vektet graf G = (V, E) med vektfunksjon weight og en startnode s
Output: Et map som angir lengden på korteste vei fra s til alle noder i G

function bellmanFord(G, s)
    distances = new Map() with infinity as default value
    distances[s] = 0

    repeat |V| - 1 times; do
        for (u, v) in E; do
            new_distance = distances[u] + weight(u, v)
            if new_distance < distances[v]; then
                distances[v] = new_distance

    for (u, v) in E; do
        new_distance = distances[u] + weight(u, v)
        if new_distance < distances[v]; then
            error "Contains negative cycle"

    return distances
```

Korteste stier i en DAG

Ved å bruke topsort, kan vi lett finne korteste vei: Det er alltid bare en vei til hvert element.

Kjøretidskompleksitet: `O(|V| + |E|)`

```
Input: En vektet, rettet, asyklisk graf G = (V, E) med vektfunksjon weight og en
       startnode s
Output: Et map som angir lengden på korteste vei fra s til alle noder i G

function dagShortestPaths(G, s)
    distances = new Map() with infinity as default value
    distances[s] = 0
```

```
    for u in topSort(G); do
        for (u, v) in E; do
            new_distance = distances[u] + weight(u, v)
            if new_distance < distances[v]; then
                distances[v] = new_distance

    return distances
```

Prims algoritme for MST

Bruker en prioritetskø, "BFS" fra en tilfeldig node i midten og utover.

Kjøretidskomplektsitet: `O(|E| log |V|)`

```
Input: En sammenhengende, vektet, urettet graf G = (V, E) med vektfunksjon
       weight
Output: Et minimalt spenntre for G

function prim(G)
    queue = new PriorityQueue()
    parents = new Map()

    s = V.popRandom()
    insert(queue, (null, s)) with priority 0

    while |queue| > 0; do
        (parent, u) = removeMin(queue)
        if u not in parents; then
            parents[u] = p
            for (u, v) in E; do
                insert(queue, (u, v)) with priority weight(u, v)

    return parents
```

Kruskals algoritme for MST

Kruskal lager ikke nødvendigvis ett spenntre, men kan lage flere ("spennskog"). Dette gjøres ved å sortere kantene i stigende rekkefølge i en PQ. Deretter popper man dem en etter en, og legger dem til i MST-et hvis ikke det allerede finnes en sti mellom nodene kanten sammenkobler.

Pseudokode ble ikke oppgitt.

Boruvkas

Basically det samme som Kruskal.

Sammenhengende grafer

Finne separasjonsnoder

```
Input: En sammenhengende graf G = (V, E)
Output: Alle separasjonsnoder i G

function separationNodes(G)
    depth, low = 2 * new Map()
    separation_nodes = new Set()

    start = V.chooseRandom()
    depth[start], low[start], children = 0

    for (start, node) in E; do
        if node in depth; then
            continue
        separationNodesRecursive(G, start, node, 1)
        children++

    if children > 1; then
        separation_nodes.add(start)


function separationNodesRecursive(G, parent, node, depth_value)
    depth[node] = depth_value
    low[node] = depth_value

    for (node, other_node) in E; do
        if other_node == parent; then
            continue
        if other_node in depth; then
            low[node] = min(low[node], depth[other_node])
            continue

        separationNodeRecursive(G, node, other_node, depth_value + 1)
        low[node] = min(low[node], low[other_node])
        if d <= low[other_node]; then
            separation_nodes.add(u)
```

Sterkt sammenhengende komponenter

```
Input: En rettet graf G = (V, E)
Output: De sterkt sammenhengende komponentene til

function stronglyConnectedComponents(G)
```

```
stack = dfsTopSort(G)
reversedG = reverseGraph(G)
visited, components = 2 * new Set()

while |stack| > 0; do
    u = stack.pop()
    if u in visited; then
        continue;
    component = new Set()
    DFSVisit(reversedG, u, visited, component)
    components.add(component)

return components
```