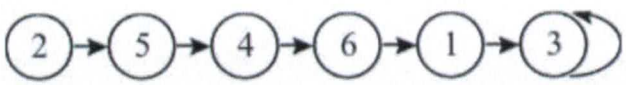
57. Distribuované a paralelní algoritmy - algoritmy nad seznamy, stromy a grafy.

Algoritmy nad seznamy

Tyto algoritmy pracují s vázaným seznamem. Každý prvek má nějakou hodnotu a svého následníka – typicky jej reprezentujeme polem následníků, které budeme označovat Succ. Paralelní práce se seznamy je složitější než práce s poli, protože když se každý procesor stará o svůj prvek, chybí mu v seznamu globální pohled – nevidí, co je v jiných částech seznamu.

- List:



- Pole succ:
  - Succ

3	5	3	6	4	1

  - Index

1	2	3	4	5	6

Nalezení předchůdců

Pokud bychom chtěli namísto následníků znát předchůdce každého prvku seznamu, je možné toho dosáhnout jednoduchým paralelním algoritmem:

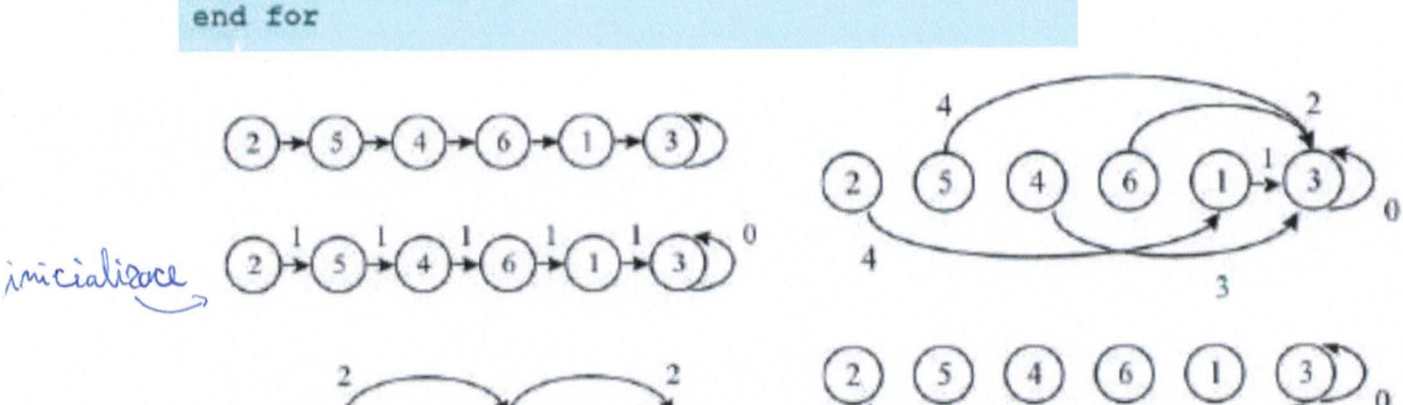
```
for i = 1 to n do in parallel
        if i != Succ[i] then
                Pred[Succ[i]] = i
```

V zásadě pro každý prvek určíme paralelně jeho předchůdce. Časová složitost je O(1), cena je tedy O(n).

List ranking

Chceme najít pořadí prvků v seznamu, tzn. vzdálenost od konce. Sekvenční algoritmus má složitost O(n). Naivní paralelní řešení by stále mělo složitost O(n), což není žádoucí. Jako vylepšení se používá technika zdvojování cesty (path doubling), kdy kromě počítání ranku je seznam v průběhu výpočtu upravován (jsou upravovány ukazatele). Ta funguje na principu zdvojnasobování zkoumané délky:

```
Algorithm
Input: array Succ[1..n]
Output: array Rank[1..n]

for i=1 to n do in parallel
        if Succ[i]=i    then Rank[i] = 0
                        else Rank[i] = 1
        for k = 1 to log n do
                Rank[i] = Rank[i] + Rank[Succ[i]]
                Succ[i] = Succ[Succ[i]]
        end for
end for
```



Vzhledem k exponenciálnímu růstu zkoumané cesty je časová složitost pouze O(log n), cena je tedy O(n log n), což není optimální. Vylepšení ceny je možné dosáhnout podobnými technikami jako je ukázáno u paralelní sumy sufixů.

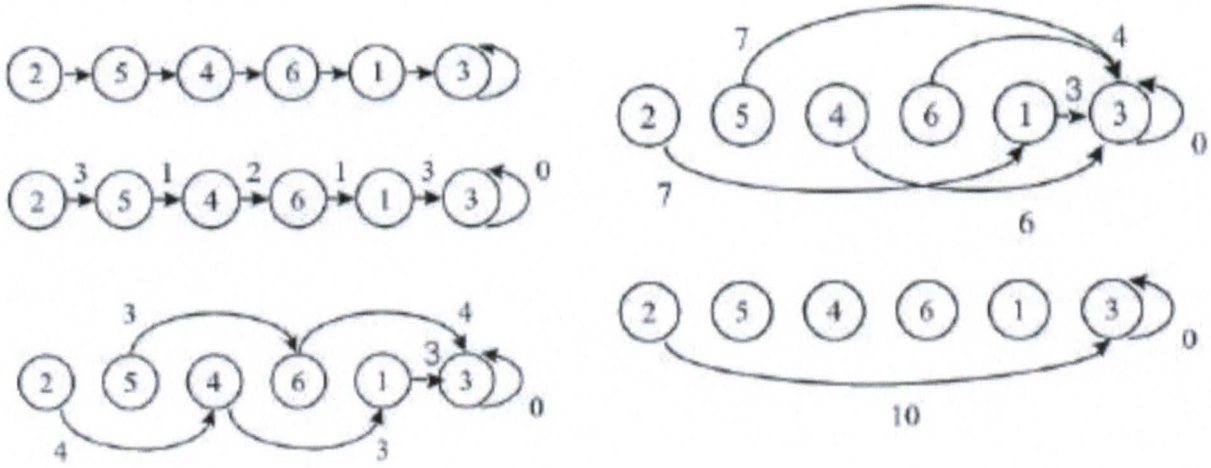
list ranking je speciální případ sumy sufixů, kde v poli V jsou samé 1, operátor je +

Suma suffixů

Je podobná jako suma prefixů prezentovaná v otázce 56, ovšem jdeme od konce, tzn. počítáme součet (resp. obecně asociativní operaci) nad podseznamy od konce seznamu. Algoritmus je téměř totožný s algoritmem list ranking, opět se využívá technika zdvojování cesty. Namísto sčítání se využívá obecný operátor a místo 0 nastavujeme neutrální prvek operace do posledního prvku. Nakonec je také potřeba přičíst do každého prvku poslední hodnotu, která byla přepsána neutrálním prvkem. Časová složitost a cena je tedy stejná s list ranking.

vstupy: pole Succ velikost V, binární asociativní operátor ⊕

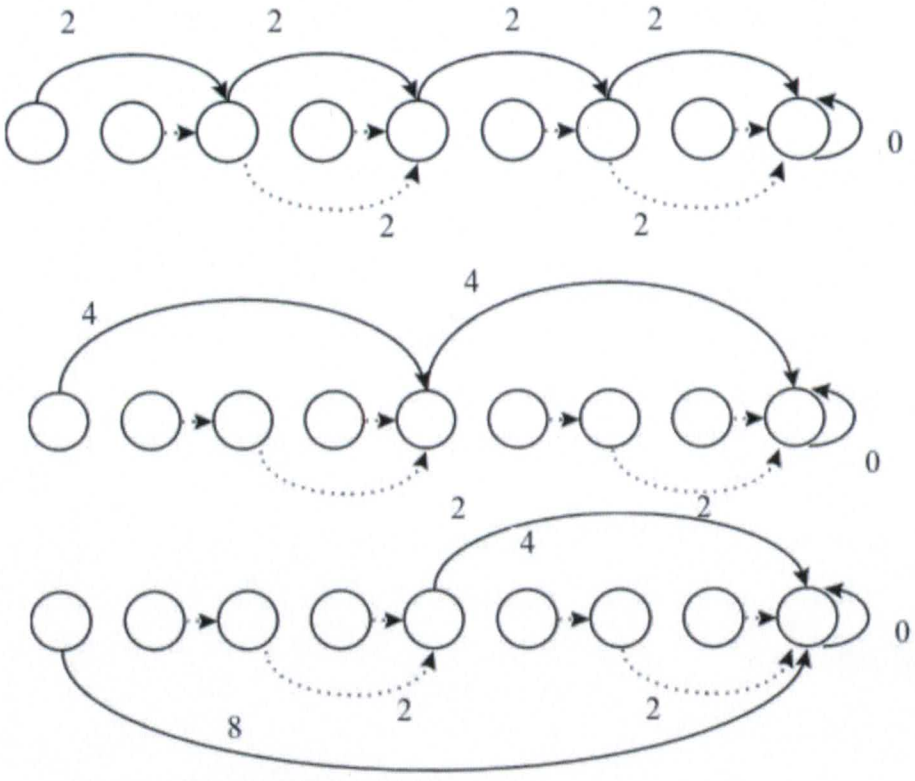
```
Algorithm
V = [v_{n-1}, ... v_1, 0]
for i = 1 to n do in parallel
        if Succ[i] = i then Val[i] = 0 /* neboli neutrální prvek operace ⊕ */
                       else Val[i] = v_i
        for k = 1 to log n do
                Val[i] = Val[i] ⊕ Val[Succ[i]]
                Succ[i] = Succ[Succ[i]]
        end for
        if Val[last] <> 0 then Val[i] = Val[i] ⊕ Val[last]

end for
```



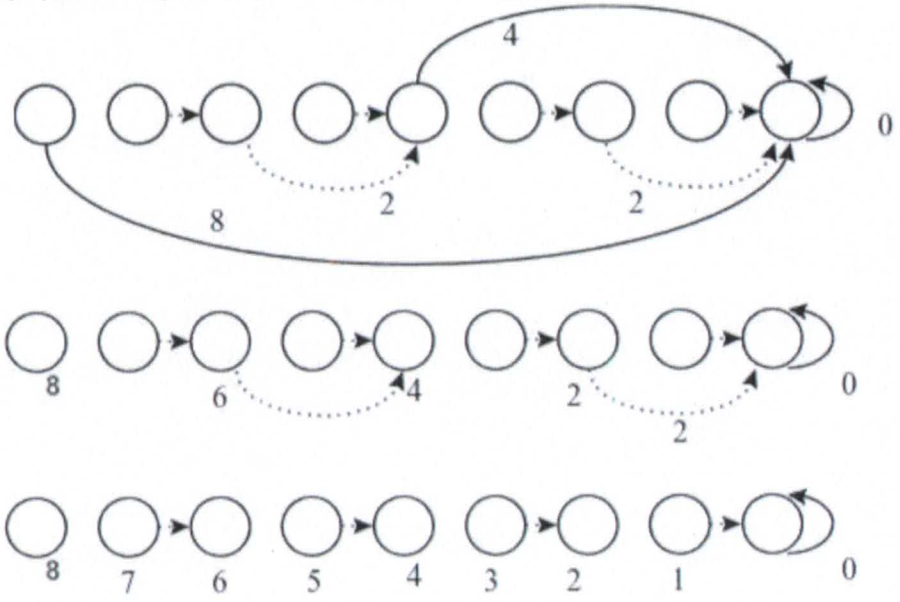
Optimalizace ceny list ranking/sumy suffixů

Cena základních implementací je neoptimální, protože se dělá zbytečná práce. Například v příkladech výše ve druhém kroku (skok o 2) druhý procesor zleva dělá zbytečnou operaci. Namísto práce by mohl čekat a až v dalším kroce se podívat na výsledek svého pravého souseda, který již má dopočítanou hodnotu a pouze k němu přičíst 1 (u list rankingu), resp. připočítat svoji vlastní hodnotu (suma suffixů). Řešením tohoto problému je, že v každém kroku vždy polovinu procesorů uspíme. Algoritmus se pak skládá ze dvou fází:

- jumping fáze – postupně jsou měněny ukazatele jako výše, v každém kroku však polovina procesorů usne. Končí v momentě, kdy poslední (levý) procesor doskákal na konec seznamu a všechny ostatní procesory spí.



- rekonstrukční fáze – poslední procesor má dopočítanou správnou hodnotu, ostatní procesory však mají špatné hodnoty, protože byly uspány. V této fázi se procesory začnou budit v opačném pořadí než usínaly a ke své hodnotě si přičtou (resp. přioperují, v případe sumy sufixů) hodnotu svého následníka.



4/15

Problémem tohoto přístupu je, jak procesor pozná, kdy se má uspat. Pro ideální implementaci by procesor musel vědět, zda je lichý nebo sudý – k tomu bychom potřebovali provést list ranking, což samozřejmě není žádoucí – pro řešení list ranking nechceme využívat list ranking.

Uvedenému problému, kdy procesor kvůli absenci globálního pohledu na celý seznam o sobě nemá jak jednoduše zjistit určitou informaci (například příznak, zda je sudý, nebo lichý), obecně říkáme *problém symetrie*. Procesory jsou v symetrické situaci, neboť každý si kupříkladu může myslet, že je sám sudý. K takzvanému *rozbití symetrie* slouží některé níže uvedené algoritmy jako Random mating, List coloring nebo Ruling set.

Random mating

Pro řešení výše uvedeného problému se používá algoritmus random mating, což je pravděpodobnostní algoritmus. Každý procesor si hodí korunou a přiřadí si pohlaví Male/Female. V každém kroku, pokud je procesor female a jeho následník male (pravděpodobnost ¼), pak se prvek přeskočí a procesor uvolní. Tím dosáhneme podobného chování jako bylo popsáno výše, ovšem bez nutnosti přesně spočítat rank.

```
procedure RandomMate
  for i = 1 to n do in parallel
        rank[i] = 1
        active[i] = True   /*if True - processor is working, False - processor is waiting */
  end for
  t = 1  /* global variable */
  while succ[head] <> tail do in parallel
        if active[i] and succ[i] <> tail then
                sex[i] = Random(M, F)
                if sex[i] = F and sex[succ[i]] = M then
                        time[succ[i]] = t
                        active[succ[i]] = False
                        rank[i] = rank[i] + rank[succ[i]]
                        succ[i] = succ[succ[i]]
                end if
                t = t + 1
        end if
  end while
  /* end of jumping phase */
  /* reconstuction phase */     while t > 0 do in parallel
                if time[i] = t and succ[i] <> tail then
                        rank[i] = rank[i] + rank[succ[i]]
                end if
                t = t - 1
        end while
end
```

Díky tomuto přístupu je cena c(n) = n + (¾)n + (¾)^2 n + … = O(n), tj. optimální za předpokladu, že dokážeme využít spící procesory k něčemu užitečnému. Abychom toto nemuseli řešit,

používá se pevný počet procesorů, konkrétně n/log n, každá procesor pak vykonává práci log n původních procesorů

List coloring (Obarvení seznamu)

V rámci k-obarvení seznamu chceme každému prvku seznamu přiřadit barvu z množiny {1, 2, … k} takovou, že pro každý vrchol platí C[x] != C[succ(x)], tzn. sousedi mají jiné barvy. Pro k=2log n barev dokážeme obarvení spočítat efektivně následujícím algoritmem (uf, tak trochu černá magie):

```
Algorithm
Input: linked list, array Index with indices of processors
Output: 2.log n  - coloring ⇒ array Color with colors of vertices

for every vertex v do in parallel
        k - least significant position in which
            Index[v] and Index[succ(v)] disagree
        Color[v] - 2.k + k-th bit of Index[v]
end for
```

Příklad:

Index	Index (binary)	k	Color
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	…	…
	3 2 1 0		

pro $k = 2 \log n$ je $t(n) = O(1)$, $h(n) = n$

Ruling set (množina oddělovačů)

Máme seznam s vrcholy V = {v1, v2, … vn}. Pak podmnožina S množiny vrcholů V je k-ruling set, pokud:

- žádné dva vrcholy v S spolu nesousedí
- vzdálenost mezi nevybraným vrcholem k následujícímu vybranému je nejvíce k

Např. **v1** v2 v3 **v4** v5 **v6** podtržené a tučné vrcholy jsou 2-ruling set, protože od vrcholu v2 do nejbližšího oddělovače v4 je vzdálenost 2, podobně pak pro v3 a v5

2k-ruling set

Pokud máme již spočtené k-obarvení, je možné spočítat 2k-ruling set v konstantním čase tím, že hledáme vždy "lokální minimum v barvách" a to označíme jako oddělovač. Tento algoritmus staví na principu, že mezi dvěma sousedícími lokálními minimy (údolími) může být maximálně 2k barev – v nejhorším případě vystoupáme z údolí po jedné na nejvyšší barvu a pak opět klesneme zpět na nejnižší barvu.

```
Algorithm
Input: k-coloring-array Color
Output: 2.k-ruling set - array Ruler

init Ruler for Falses
for each vertex v do in parallel
        if pred(v) ≠ head
           and Color[pred(v)]>Color[v]
           and Color[succ(v)]>Color[v]
                then Ruler[v] = True
endfor
Ruler[head] = True
```

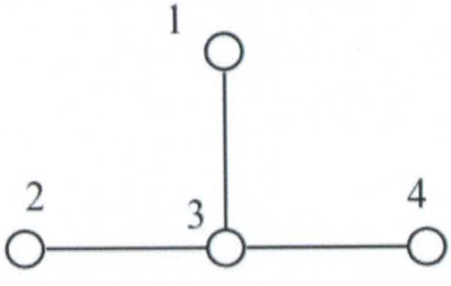
2-ruling set

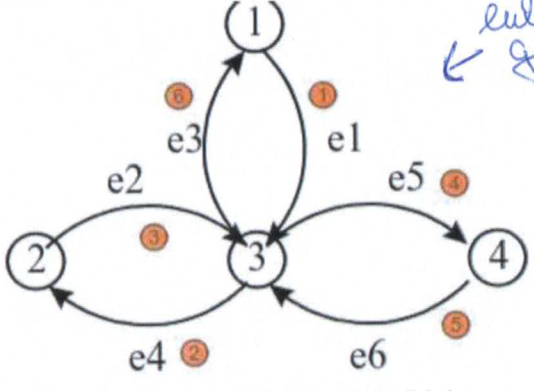
Toto rozdělení odpovídá přibližně rozdělení prvků na sudé a liché. Nejprve spočítáme 2logn obarvení a poté jdeme přes všechny barvy sekvenčním cyklem (tzn. logaritmická složitost) a pro všechny uzly, které mají danou barvu a nejsou oddělovači a zároveň ani jejich předek a následník není oddělovačem, nastavíme daný uzel jako oddělovač:

```
Algorithm
Input: linked list, array Index with indices of processors
Output: array Ruler - 2-ruling set

(1) Compute 2.log n coloring
(2) for Col - 0 to 2.log n do
        if Color[i] = Col
                then if not (Ruler[i] or Ruler[pred(i)]
                        or Ruler[succ(i)])
                        then Ruler[i] = True
                end if
        end if
```

Algoritmy nad stromy

V rámci algoritmů nad stromy zavádíme tzv. Eulerovský průchod, což je obecný průchod binárním stromem – běžně využívané průchody, např. preorder, inorder a postorder jsou pouze speciálními případy Eulerovského průchodu. Eulerovský průchod začíná v kořeni, projde všemi hranami stromu a skončí opět v kořeni, jedná se o tzv. Eulerovu kružnici. Pro potřeby algoritmů nahradíme neorientované hrany dvojicí orientovaných hran.





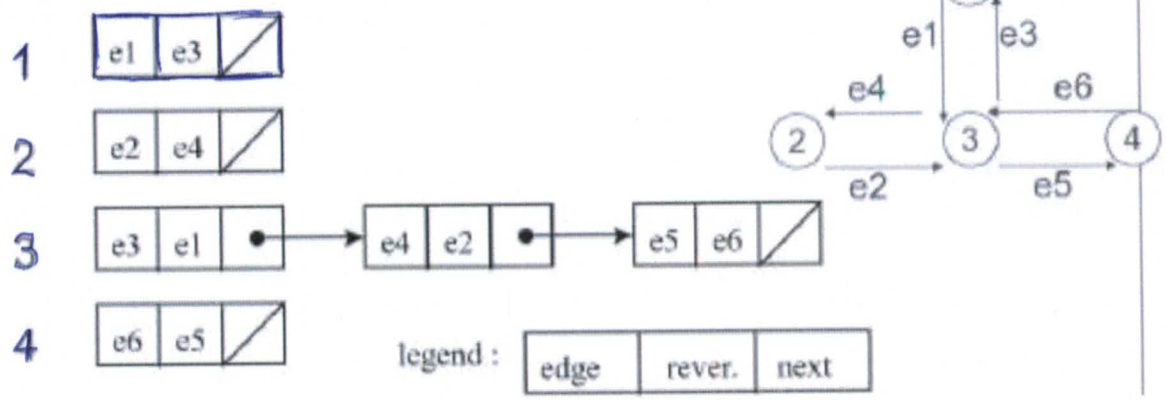
eulerovský graf

Eulerovu kružnici reprezentujeme funkcí následníka Etour (pole následníků, podobně jako u seznamu), která každé hraně přiřazuje hranu, která následuje po dané hraně v Eulerovském průchodu. Samotný strom pak můžeme reprezentovat seznamem sousednosti. Pro každý uzel máme seznam sousedů, který obsahuje pro každého souseda hranu a zpětnou hranu, která k danému sousedovi vede:

eulerovský graf obsahuje eulerovskou kružnici, která prochází každou hranou právě jednou

Seznam sousedů

- Vrchol

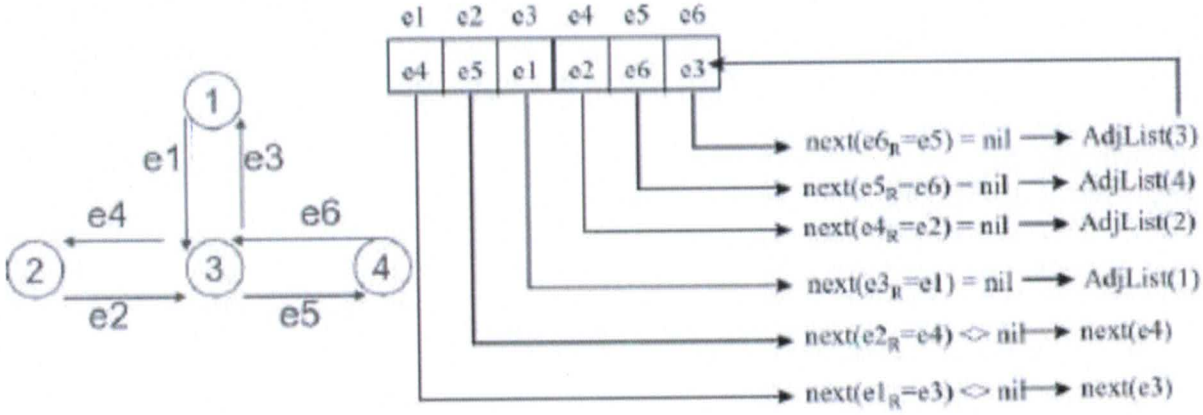


Vytvoření Eulerovy cesty

Na zadaném grafu je možné Eulerovu cestu vytvořit následujícím algoritmem. Ten funguje v konstantním čase – paralelně každý procesor spočítá jeden prvek z výstupního pole Etour. Etour zadané hrany se spočítá následovně:

- pokud následník reverzní hrany není nil, pak dalším prvkem v Etour je právě tento následník
- jinak je dalším prvkem v Etour první hrana v seznamu sousedů cílového uzlu hrany

```
Algorithm
- constructing Euler tour
Input: adjacency list of T
Output: Array Etour with 2n-2 entries
        Etour(e) is a edge following e

for i = 1 to 2n-2 do in parallel        { e_i = (u, v) }
  if next(e_i) <> nil   then Etour(e) - next(e_i)
    else Etour(e) - AdjList(v)          {first item of adj. list of vertex v}
  endif
endfor
```



Zavedení kořene

Když chceme do Eulerovské cesty zavést kořen, označme jej vrchol r, zkonstruujeme nejprve Eulerovu cestu (viz algoritmus výše) a následně přiřadíme pro hranu e vedoucí do kořene r: Etour(e) = e, tzn. vytvoříme tam smyčku. Tím se z Eulerovské kružnice stává seznam.

Obecný postup

Většina algoritmů nad stromem se řeší následujícím způsobem:

- Vytvoříme Eulerovu cestu O(1)
- Vytvoříme nějaké pole hodnot O(1)
- Spočteme nad ním sumu suffixů O(log n)
- Provedeme korekci, abychom spočítali, co chceme O(1)

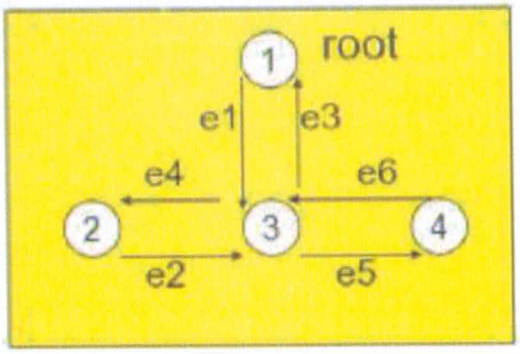
- postorder
- preorder
- inorder
- počet následníků
- úroveň vrcholu

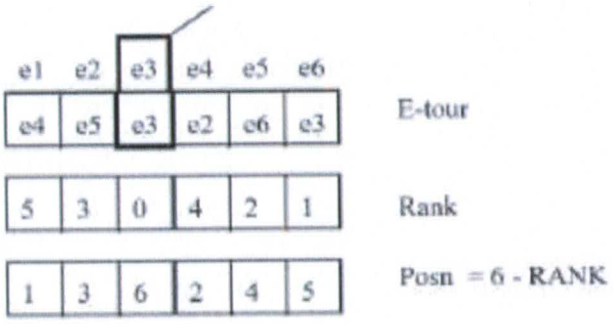
Výpočet Etour je možný v konstantním čase, inicializace pole hodnot taktéž. Suma suffixů má logaritmickou časovou složitost a korekce výsledku se typicky dá provést taktéž v konstantním (každý procesor paralelně upraví výsledek sumy suffixů pro jeden prvek). Celková složitost je tedy O(log n), cena pak závisí od implementace SuffixSums. V případě použití algoritmu jako random mating dostáváme cenu O(n).

Výpočet pozice hran v Eulerově cestě

Pro určení pozice každé hrany v Eulerovské cestě zkombinujeme několik předchozích algoritmů:

1. Spočítáme Etour O(c)
2. Přiřadíme Etour(e) = e pro hranu e vedoucí do kořene O(1)
3. Spočítáme ListRanking nad Etour O(log n)
4. Paralelně spočteme pozici jako posn(e) = 2n - 2 - Rank(e) – korekce, abychom dostali pořadí od začátku namísto od konce O(1)





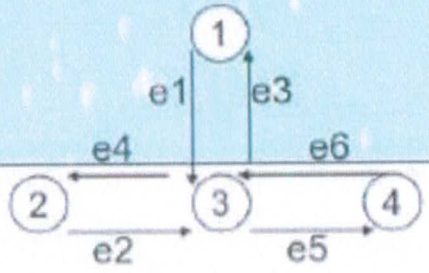
Nalezení rodičů ve stromě

Tento algoritmus využívá výpočet pozice hran v Eulerově cestě. Podle nich pak pozná, zda je hrana zpětná nebo dopředná – dopředná hrana má pozici menší než zpětná. Pokud je hrana (u, v) dopředná, pak ve stromu je u rodičem v.

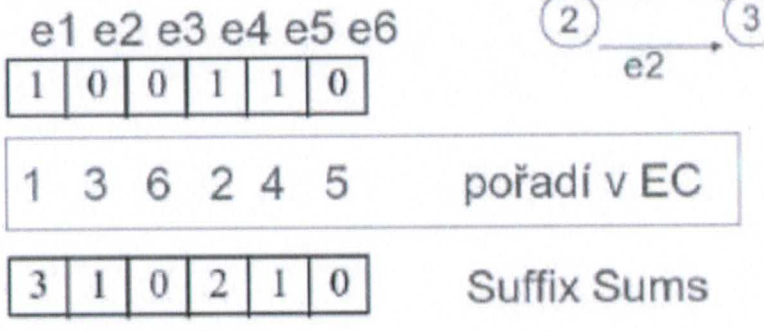
Přiřazení preorder pořadí vrcholům

Pořadí preorder vrcholu ve stromě je 1 + počet dopředných hran, kterými jsme prošli po cestě k vrcholu. Pro řešení tohoto problému tedy přiřadíme dopředným hranám váhu 1, zpětným hranám váhu 0 a spočítame nad Eulerovou cestou sumu suffixů. Nakonec provedeme korekci:

```
Algorithm
1) for each e do in parallel
       if e is forward edge  then weight = 1
             else weight = 0
       endif
2) weight = SuffixSums(Etour, Weight)
3) for each e do in parallel
       if e=(u, v) is forward edge then
         preorder(v) = n - weight(e) + 1
       endif
   preorder(root) = 1
```



- Příklad



Počet následníků vrcholu

Počet následníků vrcholu je roven počtu dořpedných hran v podstromu, který má kořen v daném vrcholu + 1 (aby byl započítán i samotný "kořenový" vrchol). Výpočet je opět podobný, musíme však v rámci sumy suffixů odečíst váhu (počet dopředných hran), která se nachází mimo současný podstrom:

```
Algorithm - number of descendants
1) for each e do in parallel
       if e is forward edge  then weight = 1
             else weight = 0
       endif
2) weight = SuffixSums(Etour, Weight)
3) for each e do in parallel
       if e=(u, v) is forward edge then
         desc(v) = weight(u, v) - weight(v, u)
       endif
   desc(root) = n
```

Výpočet úrovně vrcholu

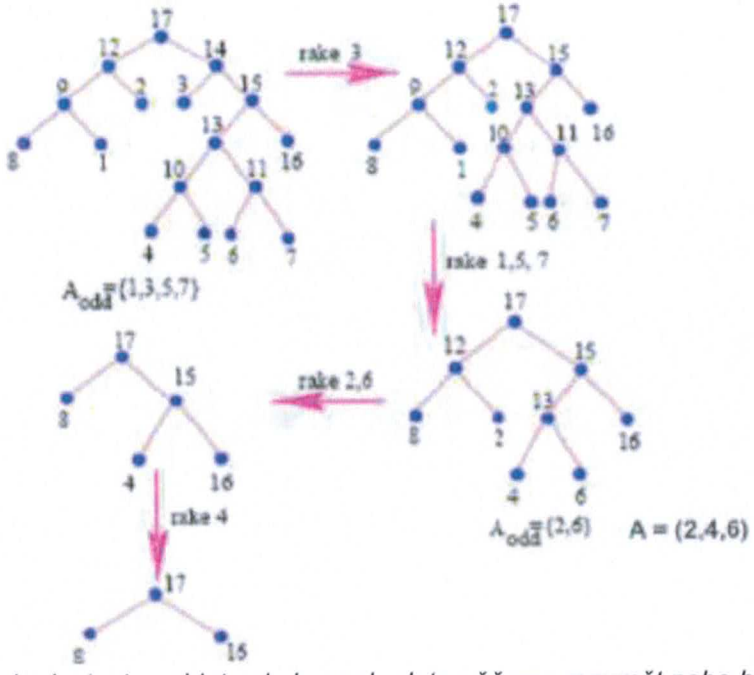
Výška vrcholu v je rozdíl počtu zpětných a dopředných hran na zbytku Eulerovy cesty od vrcholu v až do konce Eulerovy cesty. Algoritmus tedy může pracovat takto:

```
Algorithm: - level of vertex
1) for each e do in parallel
       if e  is forward edge  then weight(e) = -1
                              else weight(e) = +1
       end if
   end for
2) weight = SuffixSums(Etour, weight)
3) for each e do in parallel
       if e = (u, v) is forward edge then level(v) = weight(e) + 1
       endif
   endfor
   level(root) = 0
```

- 0) Spočtení Etour O(c)
- 1) Inicializace weight O(c)
- 2) Výpočet SuffixSums O(log n)
- 3) Korekce výsledku O(c)

Kontrakce stromu

Některé problémy nad stromem nejde efektivně paralelizovat pomocí Eulerovy cesty, např. pokud máme zadán aritmetický výraz stromem (např. Abstraktní Syntaktický Strom) a chceme vyhodnotit hodnotu konstantního výrazu, tzv. kontrakce stromu. Jedním ze způsobů řešení je operace RAKE, která spojuje listové vrcholy s rodičem. Paralelně tedy aplikujeme operaci RAKE na několik vrcholů a tím postupně zmenšujeme hloubku stromu, až se dostaneme pouze k jedinému uzlu. Problémem je, že nesmíme aplikovat paralelně RAKE na uzly, jejichž rodiče spolu sousedí. Pokud algoritmus implementujeme efektivně, dokážeme v každé iteraci zmenšit počet listů na polovinu, výsledná složitost kontrakce je tedy O(log n).



Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele Fifinas.