České vysoké učení technické v Praze
Fakulta elektrotechnická

Tématické okruhy ke státní závěrečné zkoušce pro magisterský studijní program Otevřená Informatika (OI)

http://www.fel.cvut.cz/cz/education/master/topicsOI.html

OI Mgr - Společné

Vygenerováno: 28. května 2015 13:27

Obsah

1 Amortizovaná složitost. Prioritní fronty, haldy (binární, d-regulární, binomiální, Fibonacciho), operace nad nimi a jejich složitost.

Amortizovaná složitost. Amortizovaná časová složitost označuje časovou složitost algoritmu v sekvenci nejhorších možných vstupních dat. Na rozdíl od průměrné složitosti nevyužívá pravděpodobnosti a je proto zaručená [2].

Asymptotická složitost je **deklarována na základě nejhorší** (nejlepší) možné **instance běhu** algoritmu, což **ale** není vždy vypovídající, protože **i nejhorší sekvence případů může mít výrazně lepší průběh**, než by asymptotická složitost napovídala. Tento zdánlivý paradox je zapříčiněn tím, že operace s vysokou složitostí změní datovou strukturu tak, že takto špatný případ nenastane po nějakou delší dobu - tím se složitá operace amortizuje.

Jako jednoduchý příklad můžeme uvést specifickou implementaci dynamického pole, která zdvojnásobuje velikost pole pokaždé, když dojde k jeho naplnění. V tomto případě je tedy nutná realokace, v nejhorším případě tato operace potřebuje čas až $O(n)$ - což je asymptotická složitost. Samotné vkládání prvků (bez nutnosti realokace) vyžaduje čas $O(1)$, pro n prvků tedy také $O(n)$. Pro vložení n prvků (včetně realokace) je tedy potřeba $O(n)+O(n) = O(n)$, amortizovaný čas na jedno vložení prvku je pak $O(n)/n = O(1)$ [7].

Definice. Je dána datová struktura D, na které postupně provádíme posloupnost stejných operací. Začneme s $D_0 = D$. První operace zavolaná na D_0 upraví datovou strukturu na D_1. Druhá operace zavolaná na D_1 upraví datovou strukturu na D_2. A tak dále. Postupně zavoláme i-tou operaci na D_{i-1} a ta upraví datovou strukturu na D_i. Některá operace může trvat krátce, jiná déle. Průměrný čas doby trvání operace nazveme amortizovanou časovou složitostí. Amortizovanou časovou složitost jedné operace spočítáme tak, že spočteme celkovou časovou složitost posloupnosti operací v nejhorším případě a vydělíme ji počtem operací.

K čemu je amortizovaná časová složitost? Pomůže nám lépe odhadnout časovou složitost některých algoritmů v nejhorším případě [1].

- Účetní metoda
- Metoda potenciálu

Interaktivní struktury: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

1.1 Prioritní fronta *(Priority Queue)*

Prioritní fronta je *abstraktní datový typ*, podobný klasické frontě či zásobníku s tím rozdílem, že každý element má svou „prioritu“. V prioritní frontě je element s vyšší prioritou vybrán dříve než element s nižší prioritou. Pokud dva elementy mají stejnou prioritu, vyberou se v pořadí v jakém byly vloženy.

Operace. Prioritní fronta musí implementovat alespoň následující operace:

- `void push(Element e)` - vloží element do prioritní fronty
- `Element pull()` - vybere z fronty element s nejvyšší prioritou

1.2 Haldy

Halda (minimální) obecně je datová struktura (obvykle stromová) splňující **vlastnost haldy**:

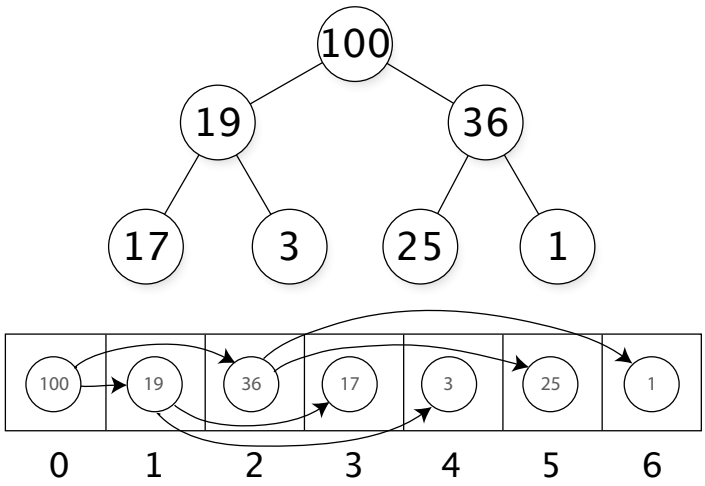
$$\textit{Pokud } B \textit{ je potomek } A\textit{, pak } B \geq A$$

Binární halda (Binary heap)

(Minimální) binární halda je binární strom s dvěma dalšími omezeními:

1. Je to kompletní binární strom krom posledního patra (nemusí být úplné). Elementy posledního patra se plní zleva doprava.
2. Každý element je menší nebo roven vůči jeho potomkům (vlastnost haldy).

Je to jedna z možných implementací prioritní fronty. Reprezentovat ji můžeme pomocí pole (obr. 1), kde prvek na indexu (číslováno od 1) i má potomky na indexech $2i$ a $2i+1$ a rodiče na indexu $i/2$.



Obrázek 1: Reprezentace (maximální) binární haldy v poli

Operace. Operace binární haldy a jejich složitosti:

- `accessMin()` - vrátí hodnotu kořene stromu (typicky první prvek pole)
- `deleteMin(e)` - vrátí element `e`, který je kořenem stromu, na jeho místo vloží nejpravější prvek `y` ze spodního patra a poté, pokud je `y` větší než jeho nejmenší potomek, prohazuji `y` s jeho nejmenším potomkem do té doby, dokud je `y` větší než jeho nově vzniknuvší nejmenší potomek, (tzv. `y` probublává stromem dolů).
- `insert(e)` - přidáme element `e` na konec haldy a dokud je předek větší než `e`, tak je prohazujeme (probublávání směrem nahoru).
- `delete(e)` - podobně jako `deleteMin()`, odeberu element `e` a nechám ho probublat

- `merge(h1, h2)` - sloučí 2 haldy, vytvoří nové pole, kam nakopíruje obsah obou hald a na toto nové pole zavolá proceduru `heapify()`. Ta jede od půlky pole směrem na začátek (navštíví všechny podhaldy) a každý vrchol podhaldy nechá probublat a správně zařadit.
- `decreaseKey(k, v)` - zmenšíme hodnotu elementu s klíčem `k` o `v` a necháme probublat stromem.

operace	čas. složitost	poznámka
`accessMin()`	$\Theta(1)$	přístup k vrcholu haldy
`deleteMin()`	$\Theta(\log(n))$	smazání vrcholu haldy
`insert(e)`	$\Theta(\log(n))$	přidání prvku do haldy
`delete(e)`	$\Theta(\log(n))$	smazání elementu haldy
`merge(h1,h2)`	$\Theta(n_1 + n_2)$	sloučení 2 hald
`decreaseKey(k,v)`	$\Theta(\log(n))$	snížení hodnoty klíče k o v

Tabulka 1: Binární halda - Operace a jejich složitosti

1.2.1 D-regulární halda (D-ary heap)

D-regulární halda je zobecnění binární haldy, kde počet potomků se rovná číslu d, namísto 2 jako je tomu v haldě binární. d nám udává počet štěpení stromu haldy. Operace jsou identické jako v binární haldě. Časová složitost operací je téměř stejná, liší se jen v základu logaritmu (u binární haldy je základ 2, tady d). Pro efektivní implementaci je vhodné zvolit d jako mocninu 2. V tomto případě lze totiž využít bitových posunů - změna indexu při průchodu polem. D-halda běží rychleji než binární pokud velikost haldy převyšuje velikost cache počítače [5].

Operace. Operace d-regulární haldy a jejich složitosti:

operace	čas. složitost	poznámka
`accessMin()`	$\Theta(1)$	přístup k vrcholu haldy
`deleteMin()`	$\Theta(\log_d(n))$	smazání vrcholu haldy
`insert(e)`	$\Theta(\log_d(n))$	přidání prvku do haldy
`delete(e)`	$\Theta(\log_d(n))$	smazání elementu haldy
`merge(h1,h2)`	$\Theta(n_1 + n_2)$	sloučení 2 hald
`decreaseKey(k,v)`	$\Theta(\log_d(n))$	snížení hodnoty klíče k o v

Tabulka 2: D-regulární halda - Operace a jejich složitosti

1.2.2 Binomiální halda (Binomial heap)

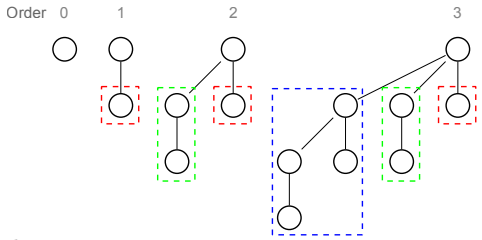
Binomiální halda je kolekce binomiálních stromů stupňů $i = 0 \dots \lfloor \log(n) \rfloor$. Každý řád je zastoupen maximálně 1 stromem.

Binomiální strom je definovám rekurzivně:

- Binomiální strom řádu 0 obsahuje jediný prvek - kořen
- Binomiální strom řádu k má kořenový element, jehož potomci jsou kořeny binomiálních stromů stupňů $k-1, k-2, \ldots, 2, 1, 0$ (v tomto pořadí)

Pro binomiální strom řádu k platí:

- splňuje vlastnost haldy
- hloubka stromu je k
- kořen má k potomků
- obsahuje 2^k prvků



Operace. Operace binomiální haldy a jejich složitosti:

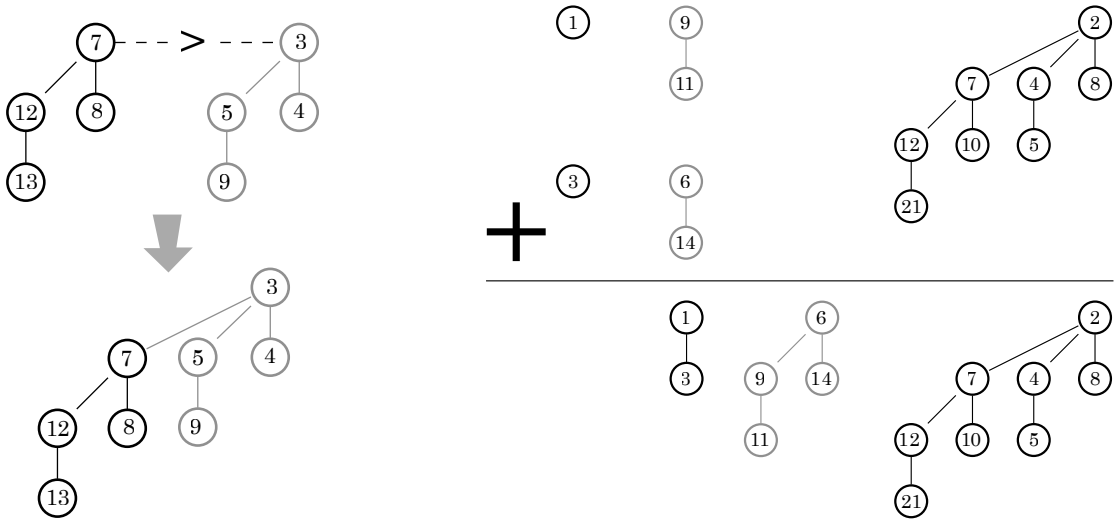
- `accessMin()` - vrátí kořen binomiálního stromu z MIN ukazatele.
- `deleteMin(e)` - vezmou se všechny podstromy, které vznikly odebráním kořene a ty se postupně mergují s ostatními stromy
- `insert(e)` - z vkládaného prvku se vytvoří nová binomiální halda (strom řádu 0) a ta se merguje s původní haldou
- `delete(e)` - sníží se hodnota pomocí `decreaseKey` na $-\infty$ a provede se `deleteMin()`
- `merge(h1, h2)` - Analogie mezi mergeováním dvou hald a binárním sčítáním. Naskládáme si stromy obou hald pod sebe (podle jejich stupňů). Pokud v jedné haldě je strom i-tého řádu a v druhé není, tak ho jen opíši. Pokud v obou haldách existují stromy stejného řadu, pak vzniká přenos do vyššího řádu. Kdykoliv vznikne přenos, merguje-me tyto dva stromy do sebe. Díky struktuře stromů, se provádí merge, porovnáním jejich kořenů, menší z kořenů se stane kořenem nově vzniklého stromu (řádu o 1 vyšší) a druhý strom se stane jeho potomkem.
- `decreaseKey(k, v)` - podobné jako u binární haldy

operace	čas. složitost	poznámka
`accessMin()`	$\Theta(1)$	přístup k vrcholu haldy
`deleteMin()`	$\Theta(\log(n))$	smazání vrcholu haldy
`insert(e)`	$\Theta(\log(n))$ amortizovaně: $\Theta(1)$	přidání prvku do haldy
`delete(e)`	$\Theta(\log(n))$	smazání elementu haldy
`merge(h1,h2)`	$\Theta(\log(n))$	sloučení 2 hald
`decreaseKey(k,v)`	$\Theta(\log(n))$	snížení hodnoty klíče k o v

Tabulka 3: Binomiální halda - Operace a jejich složitosti

1.2.3 Fibonacciho halda

Založena na binomiální haldě. Má relaxovanější strukturu, která umožňuje zlepšené asymptotické složitosti. Fibonacciho halda se nevyužívá v real-time systémech, protože některé operace mají lineární složitost. U operací, které nevyžadují mazání (accesMin, merge, decreaseKey) je amortizovaná složitost $O(1)$.



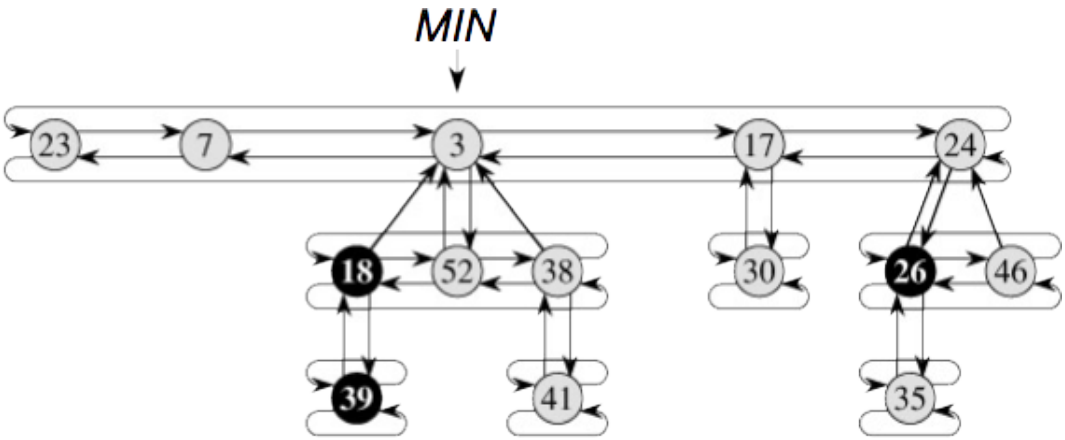
Obrázek 2: Binomiální halda MERGE - 2 příklady

Struktura. Fibonacciho haldu tvoří skupina stromů vyhovující lokální podmínce na uspořádání haldy, která vyžaduje, aby pro každý uzel stromu platilo, že prvek, který reprezentuje, je menší než prvek reprezentovaný jeho potomky. Z této podmínky vyplývá, že minimálním prvkem je vždy kořen jednoho ze stromů. Vnitřní struktura Fibonacciho haldy je v porovnání s binomiální haldou daleko více flexibilní. Jednotlivé stromy nemají pevně daný tvar a v extrémním případě může každý prvek haldy tvořit izolovaný strom nebo naopak všechny prvky mohou být součástí jediného stromu hloubky n. Tato flexibilní struktura umožňuje velmi jednoduchou implementaci operací s haldou. Operace, které nejsou potřebné, odkládáme a vykonáváme je až v okamžiku, kdy je to nevyhnutelné, například spojení nebo vložení nového prvku se jednoduše provede spojením kořenových seznamů (s konstantní náročností) a jednotlivé stromy spojíme až při operaci snížení hodnoty klíče [8].

Implementace. Pro rychlé vymazání a zřetězení se vytváří *obousměrný cyklický spojový seznam* kořenů všech stromů (obr. 3). Pro potomky každého prvku se vytváří podobný seznam. Pro každý uzel se ukládá počet synů a údaj, zda je zvýrazněn. Navíc si uchováváme ukazatel na kořenový prvek s minimální hodnotou klíče (MIN) [8].

- **N** - aktuální počet prvků
- **MIN** - minimum haldy
- **rank** - pomocné pole pří slévání stromů (označení stupně stromů)
- **mark(x)** - boolean hodnota indikující jestli prvek x ztratil potomka od té doby kdy x bylo vytvořeno jako potomek jiného prvku. Nově vytvořené prvky jsou neoznačené, a prvek x se stane neoznačeným, vždy když se stane potomkem jiného prvku.

Operace. Operace Fibonacciho haldy a jejich složitosti:



Obrázek 3: Reprezentace fibonacciho haldy

- `accessMin()` - vrátí kořen z Fibonacciho stromu na který ukazuje MIN pointer
- `deleteMin()` - Operace odstranění minima probíhá ve třech krocích. V prvním odstraníme kořenový prvek s minimální hodnotou klíče. Jeho synové vytvoří kořenové prvky nových stromů. Poté se slévají stromy stejného stupně, a nakonec se upraví ukazatel MIN. Při slévání stromů tvořících jednu haldu postupně sjednotíme kořeny stromů stejných stupňů. Uvažujeme, že počet kořenových prvků na počátku operace je N. Pokud máme dva kořeny U a V stejného stupně, vytvoříme z jednoho z nich syna druhého prvku tak, aby kořenovým zůstal prvek s menší hodnotou klíče. Jeho stupeň se pak zvětší o jedničku. Toto opakujeme, dokud v haldě existují dva stromy se stejným stupněm. K efektivnímu hledání stromů stejného stupně používáme pole ukazatelů, ve kterém uchováváme reference vždy na jeden kořen každého stupně. Pokud je nalezen druhý strom stejného stupně, oba stromy jsou spojeny a příslušný ukazatel v poli je aktualizován
- `insert(e)` - vytvoří se nová halda obsahující jediný element `e`; `mark(e) = false`; Merge s původní haldou - $O(1)$
- `merge(h1, h2)` - prosté spojení seznamů s kořenovými prvky stromů jednotlivých hald a update MIN pointeru

operace	čas. složitost	poznámka
`accessMin()`	$O(1)$	přístup k vrcholu haldy
`deleteMin(e)`	$O(n)$ amortizovaně: $O(\log(n))$	smazání vrcholu haldy
`insert(e)`	$O(1)$	přidání prvku do haldy
`delete(e)`	$O(n)$ amortizovaně: $O(\log(n))$	smazání elementu haldy
`merge(h1,h2)`	$O(1)$	sloučení 2 hald
`decreaseKey(k,v)`	$O(\log(n))$ amortizovaně $O(1)$	snížení hodnoty klíče k o v

Tabulka 4: Fibonacciho halda - Operace a jejich složitosti

deleteMin() Popis operace deleteMin a následné konzolidace haldy.

```
z = MIN;
if (z ≠ null) then {
  foreach x ∈ descendants(z) do
    add x to the root list of the heap;
  remove z from the root list of the heap;
  if (N=1) then {
    MIN = null;
  } else {
    MIN = any pointer to a root from the root list of the heap;
    Consolidate();
  }
  N--;
}
```

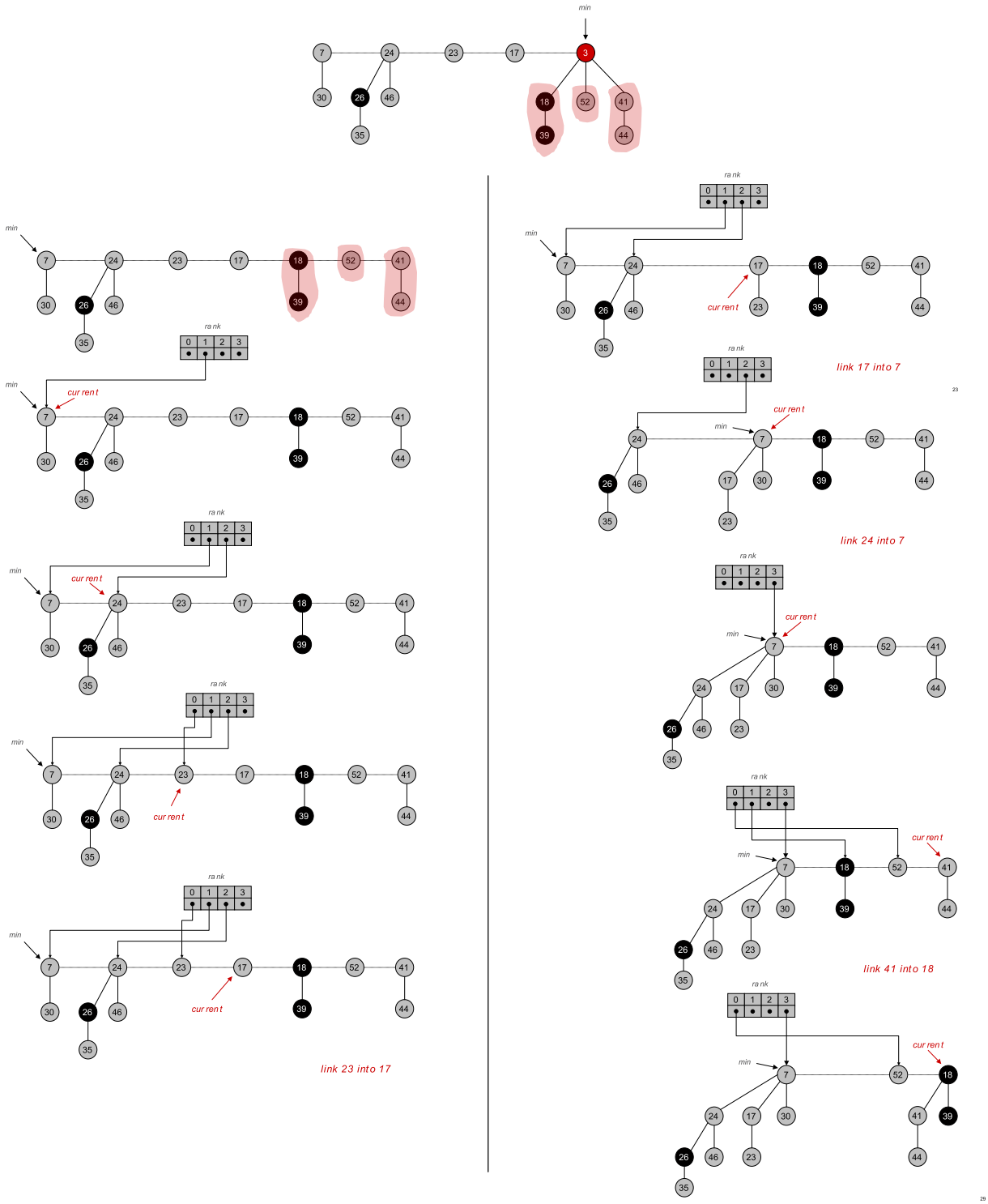
Kód 1: deleteMin ve Fibonacciho haldě

```
foreach w ∈ all trees in the root list of the heap do {
  x = w; d = a degree of the tree w;
  while rank[d] ≠ null do {
    y = rank[d];
    if key(x) > key(y) then swap x and y;
    remove y from the root list of the Heap;
    make y a child of x, incrementing the degree of x;
    mark(y) = false; rank[d] = null; d++;
  }
  rank[d] = x;
}
MIN = null;
for i = 0 to max. degree of a tree in the array A do {
  if rank[i] ≠ null then {
    add rank[i] to the root list of the heap;
    If (MIN = null) or (key(A[i]) < key(MIN)) then MIN = A[i];
  }
}
```

Kód 2: Konzolidace ve Fibonacciho haldě

	binary heap	**d-ary heap**	**binomial heap**	**Fibonacci heap**
accessMin()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
deleteMin()	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n); O(\log(n))$
insert(e)	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n)); O(1)$	$\Theta(1)$
delete(e)	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(n); O(\log(n))$
merge(h1,h2)	$\Theta(n)$	$\Theta(n)$	$O(\log(n))$	$\Theta(1)$
decreaseKey(k,v)	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n)); O(1)$

Tabulka 5: Haldy - srovnání složitostí



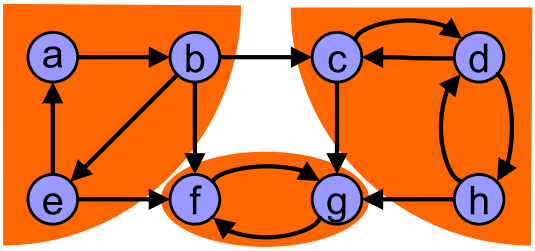
Obrázek 4: Postup při deleleMin operaci

2 Neorientované a orientované grafy, jejich reprezentace. Prohledávání grafu (do hloubky a do šířky), topologické uspořádání, souvislost, stromy, minimální kostra.

Graf je uspořádaná dvojice množiny vrcholů a hran $G = (V, E)$. Dá se reprezentovat mnoha způsoby. Nelze říci, který je lepší, protože se každá reprezentace hodí k trochu jinému typu úloh. Nejčastěji je ale graf reprezentován **maticí sousednosti** (1 kde je hrana), **laplaceovou maticí** (na diagonále stupeň vrcholu, -1 pro hranu), **maticí incidence** (řádky jsou body, sloupce hrany, 1 a -1 značí zdrojový a koncový bod) nebo pomocí **seznamu sousedů** (u každébo bodu list s jeho sousedama).

Silně souvislá komponenta (SCC) Graf je silně souvislý jestliže existuje cesta v obou směrech mezi každými dvěma vrcholy. **SCC** orientovaného grafu je maximální silně souvislý podgraf.

$$SCC(v) = \{u \in V | \text{ there exists a path in } G \text{ from } u \text{ to } v \text{ and a path in } G \text{ from } v \text{ to } u\}$$



2.1 Kosaraju

Používá upravený DFS kde při uzavírání vrcholu ho vloží do stacku S. Po prvním průchodu DFS celým grafem prohodí hrany a vyresetuje označení vrcholů. Dokud S obsahuje nějaký vrchol tak ho vybere ze stacku. Pokud vrchol ještě nebyl navštíven tak opět provede DFS. Množina aktuálně navštívených vrcholů dá SCC obsahující i aktuální vrchol.

Algoritmus provádí 2 kompletní průchody grafem. Pokud je graf reprezentován jako list sousedů pak běží v $\Theta(|V| + |E|)$, pokud jako matice sousedů tak $O(|V^2|)$

2.2 Tarjan

Tarjanův algoritmus vychází z prohledávání do hloubky. Vrcholy se při prohledávání indexují dle pořadí svého nalezení. Při návratu z rekurze se každému vrcholu přiřadí uzel s nejnižším indexem na jaký lze dosáhnout. Všechny vrcholy, které mají totožný cílový uzel (index), jsou ve stejné komponentě.

Tarjanův algoritmus má stejně jako prohledávání do hloubky asymptotickou složitost $\Theta(|V| + |E|)$ (při použití listu sousedů, jinak při matici sousedů $O(|V^2|)$).

```
procedure tarjanAlgorithm(Node node, List scc, Stack s, int index)
  v.index = index
  v.lowlink = index
  index++
  s.push(node) //pridej na zasobnik
  for each Node n in Adj(node) do //pro vsechny potomky
    if n.index == -1 //pokud jeste nebyl uzel objeven
      tarjanAlgorithm(n, scc, s, index) //prohledej
      node.lowlink = min(node.lowlink, n.lowlink) //uprav lowlink otce
    else if stack.contains(n) //pokud komponenta nebyla jiz uzavrena
      node.lowlink = min(node.lowlink, n.index)

  if node.lowlink == node.index //pokud jsme v koreni komponenty
    Node n = null
    List component //seznam uzlu dane komponenty
    do
      n = stack.pop() //vyber uzel ze zasobniku
      component.add(n) //pridej ho do komponenty
    while(n != v) //dokud nejsme v koreni
    scc.add(component) //komponentu pridej do seznamu komponent
```

Kód 3: Tarjan

2.3 Topologické uspořádání

Topologické uspořádání je taková posloupnost uzlů grafu, že pro každou jeho hranu (u, v) platí, že uzel u je zařazen před uzlem v. Topologicky lze proto uspořádat pouze acyklické grafy.

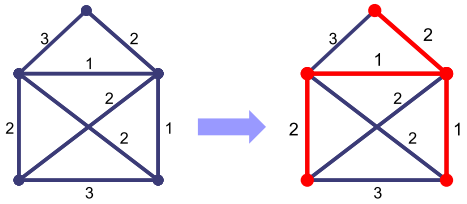
Jinými slovy: Všechny vrcholy grafu G jsou očíslovány tak, že $u \leq v$ platá pro každý pár vrcholů, které mají hranu (u, v)

Topologické uspořádání lze zjistit pomocí upraveného DFS. Lze jím testovat grafovou acyklicitu, konektivitu, hledání souvislých komponent.

Algoritmus topologického uspořádání vychází z procházení grafu do hloubky. Jedná se o pořadí uzavření uzlů opačně orientovaného grafu. Časová složitost tohoto postupu proto $O(|V| + |E|)$.

2.4 Minimální kostra grafu - MST

Kostra grafu G je graf, který má stejný počet vrcholů jako G a je to strom. Minimální kostra grafu je taková kostra, která má součet vah použitých hran nejmenší.

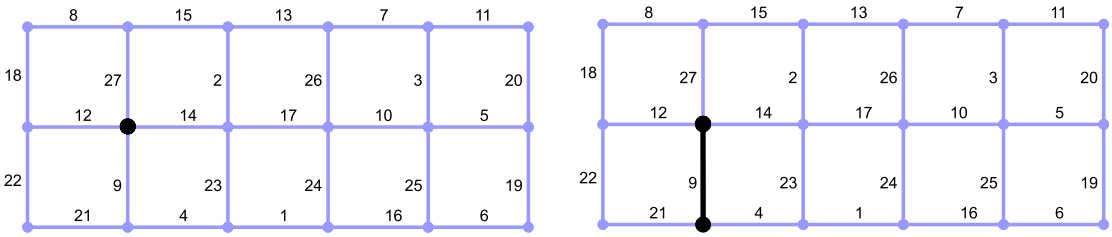


2.4.1 Primův alg.

Nejprve se vybere libovolný vrchol z V a vloží se do výsledné množiny K. K němu se pak v každé iteraci (je jich V) hledají takové hrany $(u, v) \in E$, které mají minimální cenu, u je v K a v není v K

Algorithm 1 Prim alg.

Select an arbitrary vertex $v_0 \in V(G)$
$K = \{v_0\}$
while $|V(K)| \neq |V(G)|$ **do**
  Select edge $\{u, v\} \in E(G)$ where $u \in V(K)$ and $v \notin V(K)$ so that $w(\{u, v\})$ is min
  $K = K + edge\{u, v\}$
end while



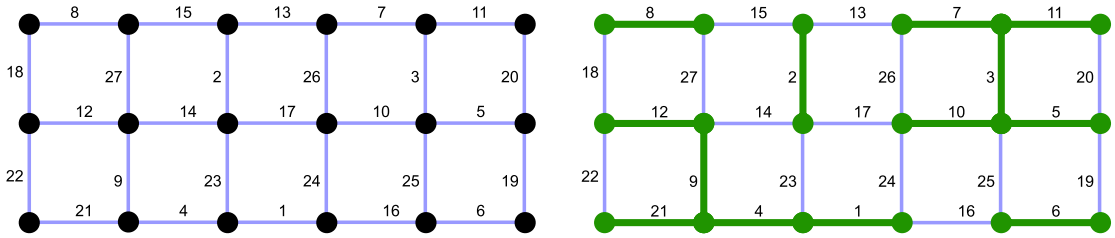
Složitost základní implementace je $O(E \cdot V)$. Jednoduchá implementace s použitím reprezentace grafu pomocí matice sousednosti a prohledáváním pole cen má časovou složitost $O(V^2)$. S použitím binární haldy a seznamu sousedů dosáhneme složitosti $O((V + E) \log(V)) = E \log(V)$.

2.4.2 Borůvkův alg.

Borůvkův algorimus funguje na principu skládání komponent. Na začátku jsou všechny uzly grafu považovány za samostatné komponenty. Algoritmus v každém svém kroku propojí každou komponentu s jinou komponentou pomocí nejkratší možné hrany. Jelikož Borůvkův algoritmus vyžaduje, aby měly všechny hrany unikátní váhu, tak při propojení komponent nikdy nemůže vzniknout cyklus. Dále je zajištěno, že se v každém kroku zmenší počet komponent minimálně na polovinu - tj. algoritmus terminuje v $\lceil \log_2 |V| \rceil$ krocích. Při každém z těchto kroků je třeba najít pro všechny komponenty nejkratší vycházející hranu, což může zabrat až $O(|E|)$ operací. Celková asymptotická složitost Borůvkova algoritmu je tedy $O(|E| \cdot \log_2 |V|)$.

Algorithm 2 Borůvka alg.

$K = (V(G))$
while while K has at least two connected components **do**
  For all components T_i of graph K the ***light incident edge*** t_i is chosen.
  All edges t_i are added to K
end while



2.4.3 Kruskalův („greedy“) alg.

Kruskalův algoritmus nejprve setřídí hrany dle jejich váhy (od nejmenší) a následně přidává hrany do grafu takovým způsobem, aby nevznikl žádný cyklus (tj. procedura terminuje po přidání $|V| - 1$ hran). V každé iteraci je výsledek podgrafem MST.

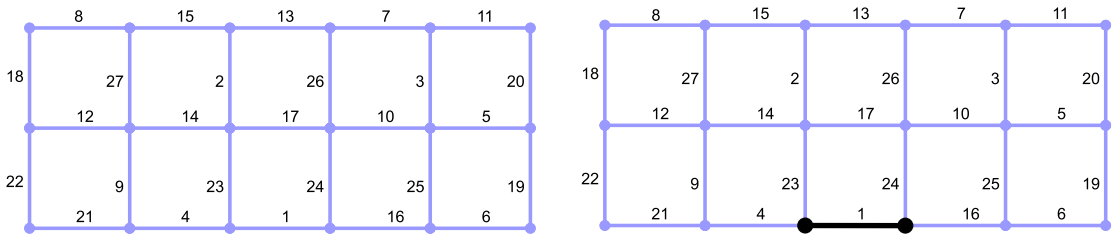
K zajištění acykličnosti si algoritmus pomocí datové struktury disjoint set (union-find) udržuje pro každý uzel informaci o příslušnosti ke komponentě souvislosti. Disjoint set poskytuje dvě operace: union (spojí dvě komponenty souvislosti) a find (zjistí pro daný uzel příslušnost ke komponentě souvislosti).

Struktury se ptáme $|E(G)|$-krát jestli jsou 2 vrcholy ve stejné kmoponentě (operace find) a mergujeme jen $|V(G)| - 1$-krát 2 komponenty do jedné (operace union).

Časová složitost algoritmu je v případě použití řadicího algoritmu založeného na porovnávání $O(|E| \cdot \log |E|)$. Pokud jsou hrany již předřazeny, nebo je možno k jejich seřazení použít řadicí algoritmus s lineární složitostí (např. counting sort), tak je složitost Kruskalova algoritmu rovna $O(|E| \cdot \alpha(|E|))$, kde α je inverzní Ackermannova funkce (odpovídá složitosti operací union a find).

Algorithm 3 Kruskal alg.

Sort all edges $e_1, \ldots, e_{m=|E(G)|} \in E(G)$ so that $w(e_1) \leq \ldots \leq w(e_m)$
$K = (V(G))$
for $i = 1 \ldots m$ **do**
  if K + edge $\{u, v\}$ is an acyclic graph **then**
    $K = K$ + edge $\{u, v\}$
  end if
end for



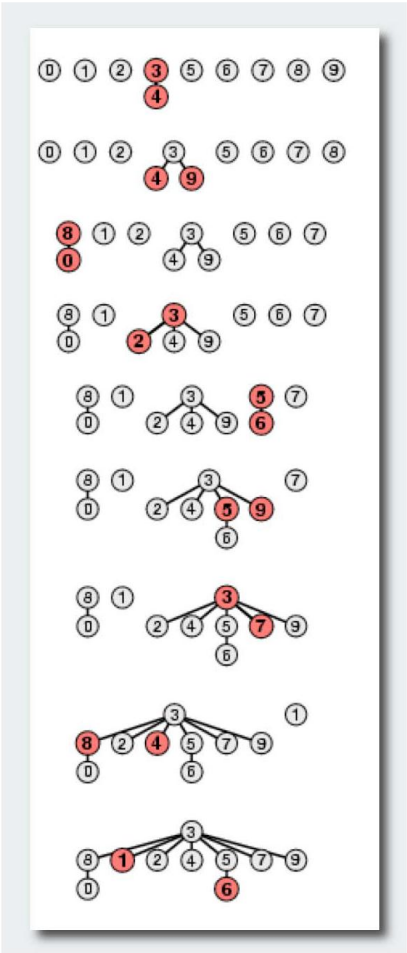
2.4.4 Union-Find problém

Problém, který se řeší v kruskalově algoritmu pro hledání MST, při zjišťování, zda 2 vrcholy leží v jedné komponentě, či nikoli - $O(1)$.

Jednoduché řešení Pole, kde pro každý vrchol udržujem číslo komponenty v jaké je (defaultně obsahuje svoje číslo). Operace find jen vrátí hodnotu na indexu. Union si najde pomocí findu oba prvky a pokud jsou ruzné, tak hodnota jednoho prvků je přepsána na hodnotu druhého (ve všech výskytech) - $O(V)$.

Vylepšené řešení za použití orientovaného stromu Ve findu, pokud se naleznou prvky s různými komponentami, tak kořen menšího stromu je přidán jako potomek většího. Hodnoty v poli vždy ukazují na roota komponenty.

```
3-4   0 1 2 3 3 5 6 7 8 9
4-9   0 1 2 3 3 5 6 7 8 3
8-0   8 1 2 3 3 5 6 7 8 3
2-3   8 1 3 3 3 5 6 7 8 3
5-6   8 1 3 3 3 5 5 7 8 3
5-9   8 1 3 3 3 3 5 7 8 3
7-3   8 1 3 3 3 3 5 3 8 3
4-8   8 1 3 3 3 3 5 3 3 3
6-1   8 3 3 3 3 3 5 3 3 3
```



3 Vyhledávací stromy: B, B+, R-B, 2-3-4, splay a jejich praktické využití. Problematika vyhledávání ve více dimenzích, K-D stromy.

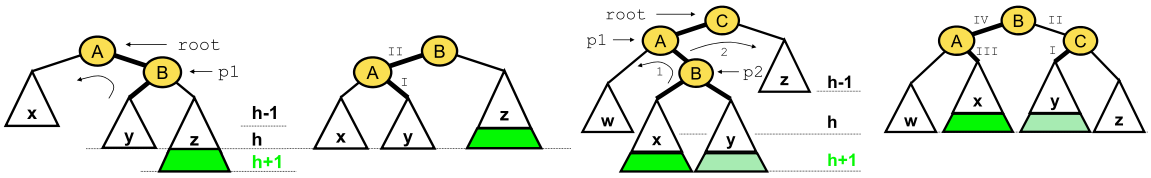
Pro vyhledvání se používají naivní metody: sekvenční prohledávání nebo binární půlení pole (předpoklad seřazeného pole), interpolační hledání (seřazené pole).

3.1 BVS - Binární vyhledávací strom

Strom, kde uzel má max 2 potomky. Leví potomcí jsou vždy menší, praví větší. Minimum ze stromu je nejlevější prvek (maximum nejpravější).

Hledání, maximum, minimum, následník, předchůdce jsou nalezeny v $O(h)$, kde h je výška stromu. Pokud je strom nevyvážený tak $h = n$ a tedy $O(n)$. Pokud je vyvážený tak $h = \log(n)$ a tím pádem i $O(\log(n))$

Vyvažování stromu Při různých operacích (např. vložení) by časová náročnost je $O(n)$ kdy je celý strom jedná dlouhá větev. Pro zlepšení složitosti na $O(\log(n))$ se stromy tzv. vyvažují - tím se snižuje počet pater. S tímto pojmem souvísí **rotace** stromu.



Obrázek 5: Levá a levopravá rotace

AVL jsou výškově vyvážené stromy.

3.2 Splay strom

Splay strom je **samovyvažující** binární vyhledávací strom mající tu vlastnost, že **prvky**, k nimž se **nedávno přistupovalo**, jsou **rychle** znovu **dostupné**. Provádí základní operace jako vkládání, vyhledávání a odstraňování prvků v amortizovaném čase $O(\log n)$ (výška je n ale amortizovaně jsou složitosti logaritmické). Výhodou oproti (např. AVL) je, že nepotřebuje udržovat další informaci (výška nebo barva).

Všechny obvyklé operace na binárních vyhledávacích stromech jsou spojeny s jednou základní, které se říká *splay*. Splay uzlu přeuspořádá strom tak, že se daný uzel dostane do kořene. Způsob, jak toho docílit, je provést standardní vyhledávání daného uzlu v binárním stromu a následně provést speciální rotace stromu takové, aby se uzel dostal do kořene. Každý přístup nebo vložení přendá prvek do rootu. *Zig-Zag* a *Zig-Zig* rotace.

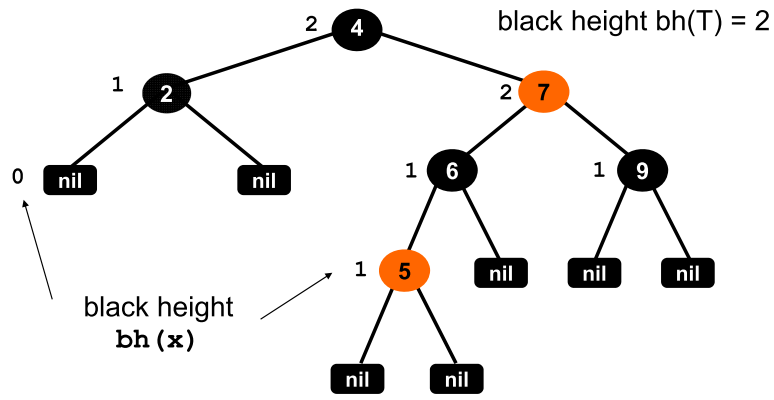
3.3 R-B (Red-Black) strom

Červeno-černý strom je binární vyhledávací strom. Je vyvážený, jeho hloubka je maximálně dvojnásobek hloubky vyváženého stromu. Jedná se o datovou strukturu často používanou pro implementaci asociativního pole.

Červeno-černý strom musí splňovat následující pravidla:

- Každý vrchol je buď červený, nebo černý.
- Kořen je černý.
- Listy (nil) jsou pokládány za černé vrcholy.
- Každý červený vrchol má dva černé syny.
- Každá cesta z jednoho vrcholu do jeho podřízených listů obsahuje stejný počet černých vrcholů.

Černá výška vrcholu `x` je počet černých vrcholů na cestě z `x` k listu (nepočítá se vrchol samotný).



Vkládání $O(\log(n))$ (max 2 rotace) Nový node `x` je červený. Vloží se normálně jako v klasickém BST. Pokud je rodič černý, vše je ok. Pokud je rodič červený:

- jestli je strýc (nodu `x`) červený - přebarvení
- jinak jestli je `x` pravý potomek - dvojitá rotace (první rotací se z toho stane 3. případ) + přebarvení
- jinak jednoduchá rotace + přebarvení

Mazání $O(\log(n))$ (max 3 rotace) Najde se vrchol ke smazání a klasicky se smaže (může mít max 1 potomka, jinak ho zaměníme s s nejbližším předchůdcem a pak ho mažem z nové pozice). Pokud je mazaný prvek červený, můžeme ho v klidu smazat - strom stále zůstane R-B.

Nyní předpokládejte, že mazaný prvek je černý a `X` je označení potomka mazaného (černého) prvku. Pokud je `X` červený, stačí ho obarvit na černo a tím je konec. Pokud je černý, tak nastávají 4 případy:

- pokud je sourozenec červený - levá rotace pravého podstromu, přebarvi sourozence a pokračuj do dalšího případu
- černý sourozenec s 2 černými potomky - obarvi sourozence a jdi (`X`) nahoru (double černá barva obarví rodiče)
- černý sourozenec s min 1 červeným potomkem
  - levý potomek je červený - pravá rotace sourozence (transformuje na následující případ)
  - pravý potomek je červený - levá rotace pravého podstromu

3.4 B strom

B-strom je **zobecněním BST** v tom smyslu, že umožňuje více než 2 potomky. Je specifický tím, že má řád n a limity na maximální (n), i minimální ($\lceil \frac{n}{2} \rceil$) počet potomků vrcholu. B-strom je díky této vlastnosti **vyvážený**, operace přidání, vyjmutí i vyhledávání tedy probíhají v logaritmickém čase. Tato struktura je často používána v aplikacích, kdy není celá struktura uložena v paměti RAM, ale v nějaké sekundární paměti, jako je pevný disk (například **databáze**). Protože přístup do tohoto typu paměti je náročný na čas (hlavně vyhledání náhodné položky), snažíme se minimalizovat počet přístupů do této paměti.

B-strom řádu n je takový strom, který splňuje tyto vlastnosti:

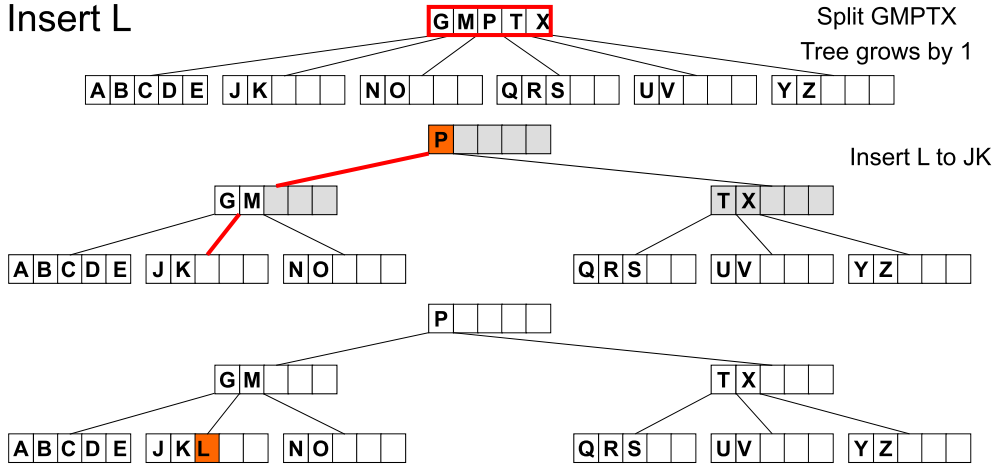
- Všechny listy (tj.uzly které nemají žádné potomky) jsou na stejné úrovni (ve stejné hloubce).
- Všechny uzly kromě kořene mají maximálně n a minimálně $\lceil \frac{n}{2} \rceil$ potomků.
- Kořen má nejvýše n potomků, spodní hranice není omezena.

Princip uložení dat Data jsou ve stromu uložena jako setříděné hodnoty, které rozdělují strom na jednotlivé podstromy. Například pokud nějaký uzel má tři potomky, musí být v tomto uzlu uloženy dva klíče k_1 a k_2, které budou uzel rozdělovat. Všechny hodnoty které jsou menší než k_1 musí být uloženy v levém podstromu, hodnoty které jsou větší než k_1 a menší než k_2 musí být uloženy v prostředním podstromu, a konečně všechny hodnoty větší než k_2 musí být v pravém podstromu. Na tyto podstromy jsou samozřejmě v uzlu uloženy ukazatele.

List tedy obsahuje $\lceil \frac{n}{2} \rceil - 1$ až $n - 1$ klíčů a neobsahuje žádný ukazatel na podstrom. Vnitřní uzel (tj. takový uzel, který není listem ani kořenem) obsahuje stejný počet klíčů k, ale tyto klíče rozdělují potomky tohoto uzlu do $k+1$ podstromů. Kořen má maximálně $n-1$ klíčů a nemusí mít žádné potomky - v tom případě je pak zároveň listem.

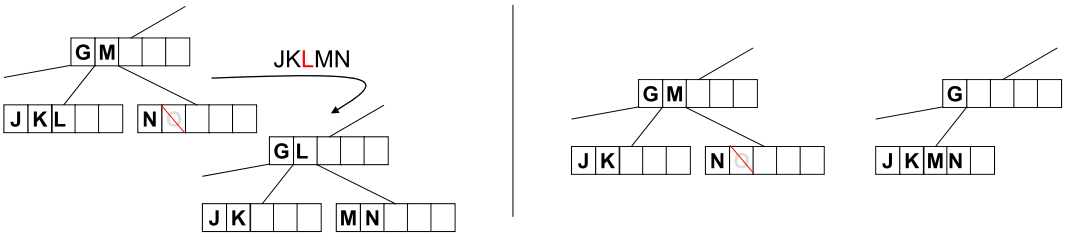
Pokud chceme vložit nebo smazat data (klíče) z uzlu, změní se tím počet potomků tohoto uzlu. Aby se dodržel rozsah daný řádem stromu, vnitřní uzly se v případě potřeby rozdělují či slučují.

Vkládání Multi vs single fáze strategie vkládání



Obrázek 6: singlephase strategie vkládání („avoid the future problems“)

Mazání jen multipass strategie!



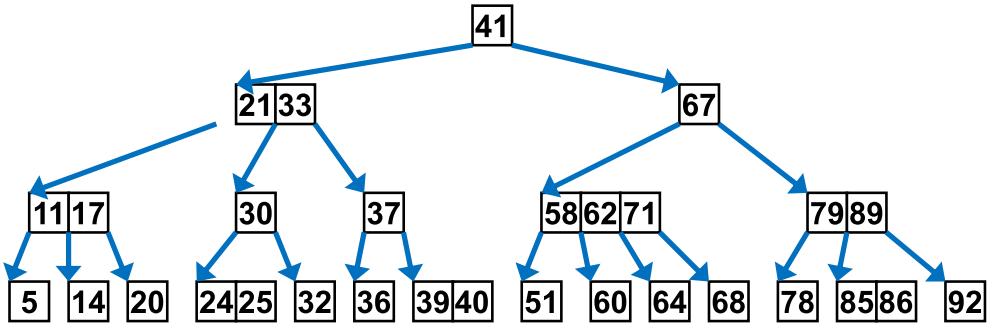
3.5 2-3-4 strom

2-3-4 vyhledávací strom je buď prázdný nebo obsahuje 3 typy prvků:

- **2**-nody s jedním klíčem, levým odkazem na strom s menšími klíči a pravým odkazem na strom s většími klíči.
- **3**-nody s dvěma klíči, levým odkazem na strom s menšími klíči, prostředním odkazem na strom s hodnotami mezi a pravým odkazem na strom s většími klíči.
- **4**-nody s třemi klíči a čtyřmi odkazy na stromy s klíči s hodnotami mezi.

Všechny odkazy na prázdné stromy (např. listy) mají stejnou vzdálenost od kořene - strom je **perfektně vyvážený**. Vkládání (s mírným vylepšením) a mazání je stejné jako v B-stromu.

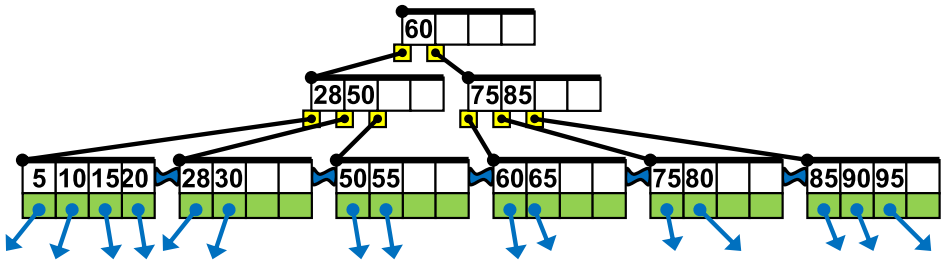
2-3-4 strom je strukturou stejný jako B-strom řádu 4.



3.6 B+ strom

Podobný B-stromu (vždy perfektně vyvážený) s rozdíly: **Hodnoty** jsou uloženy **jen v listech**. Interní prvky obsahují jen vyhledávací klíče a jsou použity jen jako placeholdery k nasměrování hledání.

Listy jsou spolu linkovány ve formě LinkedListu. Hodnoty pak mohou být získány sekvenčně bez přístupu skrze strom.



Find, Insert, Delete - $\Theta(\log_b n)$ - b je řád stromu, n je počet prvků

3.7 Vyhledávání ve více dimenzích

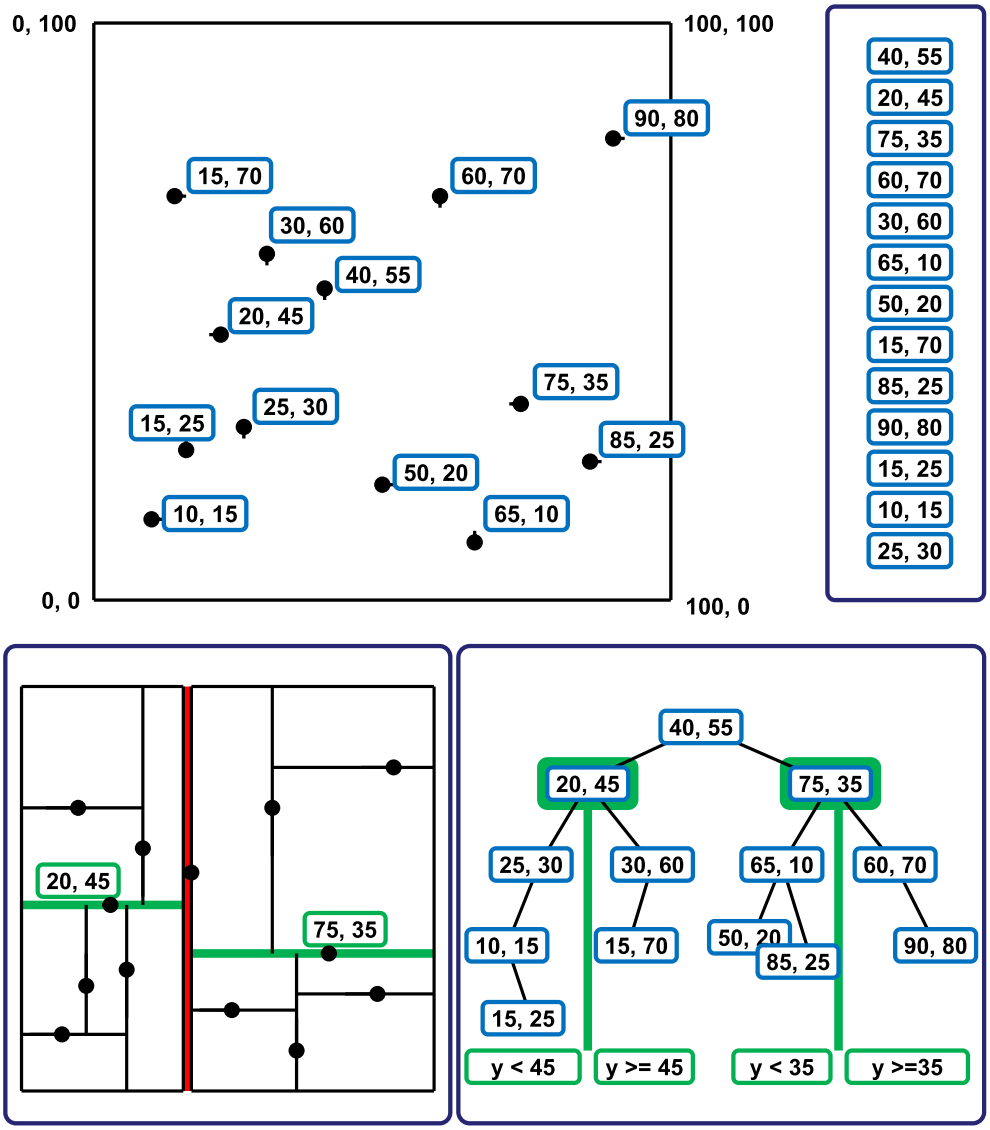
3.8 K-D strom

Je BVS reprezentující obdélníkovou plochu v D-dimenzionálním prostoru. Plocha je rozdělena (a rekurzivně dorozdělena) do obdélníkových buněk. Dimenze jsou značeny podle jejich indexu $0, 1, \ldots, D-1$.

R je kořen stromu (nebo podstromu). Obdélníková D-dimenzionální buňka $C(R)$ (hyperobdelník) je asociován s R. Jsou definovány souřadnice $R[0], R[1], ..$ a hloubka stromu h.

Operace

- **Find(key)** - stejné jako v 1D stromě. Hledá se střídavě podle souřadníc odpovídající hloubce stromu (modulo) a postupně se osekává vyhledávaný prostor.



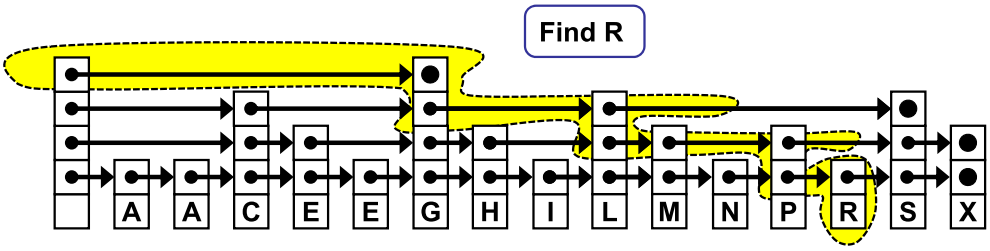
- **Insert(key)** - stejné jako v 1D stromě. Porovnávají se střídavě souřadníce odpovídající hloubce stromu (modulo) a postupně se projde až k listu, kam se nový prvek vloží.
- **FindMin(dim)** - minimální prvek pro určitou dimenzi. Tato operace je provedena jako část operace delete. Nejnáročnější operace (protože delete metoda rapidně mění strukturu stromu) - $O(n^{1-1/d})$.
- **Delete()** - jen listy mohou být smazány. Mazání prvku uvnitř stromu je zajištěno náhradou hodnot, za hodnoty jiného odpovídajícího prvku hlouběji ve stromu. Pokud pravý podstrom R není prázdný, najdi v něm minimum (podle dimenze hledaného prvku). Pokud je prázdný hledej v levém podstromu (podle dimenze hledaného prvku).

3.9 Skip List

Je seřazený linkedlist kde každý prvek obsahuje proměnný počet odkazů. k-tý link implementuje jednoduchý linkedlist, který přeskakuje prvky s menším počtem linků než k.

Je to LinkedList s vyhledávací náročnosti $O(\log(n))$. Problém má navazujícími insert/delete operacemi - ty ničí „správný“ tvar listu a tím. Řešením je vytvořit náhodný tvar podobný tomu optimálnímu, malé náhodné deviace v dlouhém běhu využití struktury nám tolik nevadí.

- **find** - postupně se prochází top-level listy a postupně se míří níže.
- **insert** - opět se postupně prochází níže až se vloží nový prvek a náhodně se vygeneruje k pro nově vzniklý prvek - doplní se linky na ostatní prvky.

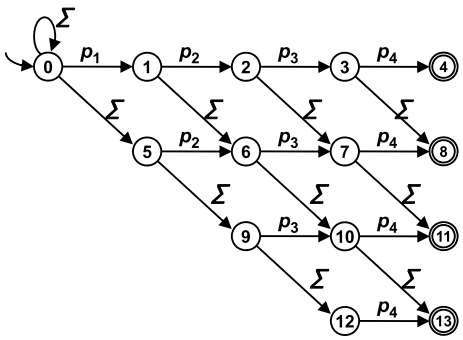


4 Přesné a přibližné hledání množin vzorků v textu, Hammingova a Levenshteinova vzdálenost. Efektivní algoritmy hledání založené na využití konečných automatů. Klasické hledání v textu (naivní, Boyer-Moore). Slovníkové automaty.

- **Abeceda:** Konečná množina znaků. Značí se A
- **Text:** Posloupnost znaků nad danou abecedou. Symboly textu se značí $t_1, \ldots, t_n$
- **Vzorek:** Posloupnost znaků nad stejnou abecedou, jejich výskyt se hledá v daném textu. Text bývá řádově delší než vzorek. Symboly vzorku se značí $p_1, \ldots, p_m$

4.1 Hammingova vzdálenost

Hammingova vzdálenost $k \geq 0$, je minimální číslo takové, že změnou symbolů na k různých pozicích v jednom z řetězců získáme druhý řetězec. Symboly nelze vypouštět nebo přidávat, Hammingova vzdálenost je **definována** jen pro **řetězce stejné délky**. V praxi se implementuje pomocí dynamického programování podobně jako Levenshteinova vzdálenost. Využití při vyhledávání podřetězců.



Obrázek 7: NFA, který přijímá slovo s Hammingovou vzdáleností max 3 od vzoru $p_1p_2p_3p_4$

4.2 Levenshteinova vzdálenost

Levenshteinova vzdálenost je minimální počet operací **vkládání, mazání a substituce** takových, aby po jejich provedení byly zadané řetězce totožné. Používá se dynamické programování (předpočítaná tabulka).

Vyplnění tabulky:

1. v nultém řádku a sloupcy tabulky A jsou postupně inkrementované čísla od 0
2. pokud jsou znaky na pozici (i, j) shodné, vložíme na pozici (i, j) hodnotu $A(i-1, j-1)$
3. pokud se znaky liší, na pozici (i, j) vložíme minimum z těchto tří hodnot:

(a) $A(i, j-1)+1$ - odpovídá operaci odstranění znaku

(b) $A(i-1, j)+1$ - odpovídá operaci vložení znaku

(c) $A(i-1, j-1)+1$ - odpovídá operaci substituce znaku

4. Výsledky najdeme v posledním řádku tabulky. Zajímají nás sloupce, kde je hodnota menší rovna požadované Levenshteinově vzdálenosti. Čteme zprava doleva.

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1	**0**	**1**	**2**	3	4	5	6	7
u	2	1	1	2	**2**	3	4	5	6
n	3	2	2	2	3	**3**	4	5	6
d	4	3	3	3	3	4	**3**	4	5
a	5	4	3	4	4	4	4	**3**	4
y	6	5	4	4	5	5	5	4	**3**

Tabulka 6: Levenshteinova vzdálenost slova „Sunday“ a „Saturday“ = 3

4.3 Hledání v textu pomocí konečných automatů

Deterministický konečný automat (DFA) je automat, který z každého stavu může přejít do maximálně jednoho cílového stavu.

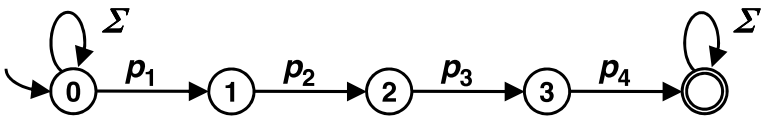
Nedeterministický konečný automat (NFA) je automat, který z každého stavu může přejít do libovolného počtu cílových stavů. Po přečtení jednoho symbolu ze vstupu přejde současně do všech cílových stavů a ze všech těchto stavů pokračuje čtením dalšího vstupu. V přechodové tabulce NKA je navíc sloupeček pro prázdný vstup, označovaný ϵ (prázdné slovo; $\epsilon \sum$). Epsilon-přechody automat provádí neustále bez čtení symbolu ze vstupu.

DFA i NFA jsou definovány jako pětice A, Q, q_0, F, δ, kde A je vstupní konečná abeceda, Q je množina vnitřních stavů, q_0 je počáteční, F je neprázdná množina koncových stavů, δ je přechodová funkce.

Přechodová funkce:

- V DFA je $\delta : Q \times A \to Q$
- V NFA je $\delta : Q \times A \to P(Q)$, kde P je potenční množina

Převod NFA do DFA Opisuju všechny stavy + přidávám nově vzniklé „multistavy“. Tabulka může narůst až na 2^n stavů.



Obrázek 8: NFA, který přijímá jakékoliv slovo se substringem $p_1p_2p_3p_4$ (kdekoliv)

Operace s automaty: S tím souvisí ϵ-přechod. To je takový přechod, který se děje neustále, bez přečtení vstupu.

- **union** - vytvoří se nový start stav a do obou start stavů sjednocovaných automatů se vytvoří ϵ-přechod
- **concatenation** - z finálních stavů A se vytvoří ϵ-přechody do start stavů B
- **iteration** - nový start stav s ϵ-přechodem do startu A; z finálních stavů A ϵ-přechody do startů A.
- **intersection** - kartézský součin stavů z obou automatů

4.4 Naivní hledání

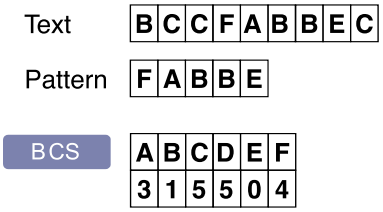
1. Přiložíme vzorek k začátku textu.
2. Dokud znaky vzorku a textu souhlasí, posunujeme se ve vzorku kupředu.
3. Když narazíme na neshodu, posuneme celý vzorek o jednu pozici kupředu, ve vzorku se nastavíme na začátek a jdeme na 2.
4. Když dojdeme za konec vzorku nebo vzorek přesáhne za konec textu, ohlásíme výsledek a případně postupujeme dále jako ve 3.

Složitost tohoto postupu je $O(m \cdot n)$, kde m je délka vzoru a n je délka textu.

4.5 Boyer-Moore

1. Vzorek přiložíme k textu a testujeme shodu vzorku odzadu.
2. Když dojde k neshodě, je šance, že vzorek lze posunout o více pozic dopředu, mnohdy o celou délku vzorku. Čím delší vzorek, tím rychlejší hledání!

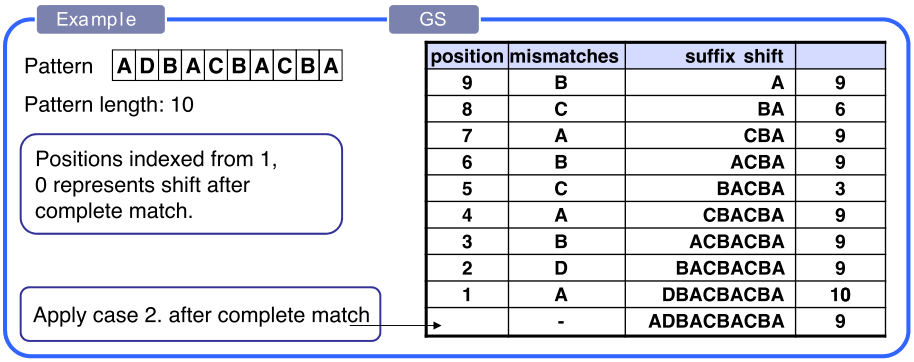
Kolize na poslední pozici vzorku Dopomáhá nám tabulka BCS (bad character shift). BCS je tabulka indexovaná znaky abecedy značící vzdálenost znaků od konce vzorku. Když tam znak není, je vzdálenost délka vzorku.



Kolize po částečné shodě na konci vzorku Nastávají 3 případy:

- Přípona p se ve vzorku vyskytuje, a to tak, že jí předchází jiný znak než právě ve vzorku kolidující. Pak musíme vzorek posunout tak, aby se tato další nejbližší instance přípony kryla s textem, tj. o vzdálenost mezi těmito instancemi přípony.
- Některá přípona vzorku stejně dlouhá nebo kratší než p se vyskytuje také na začátku vzorku. Uvažme nejdelší takovou příponu, označme její výskyt na začátku vzorku symbolem q. Vzorek pak musíme posunout o vzdálenost mezi p a q.

Tabulka GSS (Good Suffix Shift) obsahuje přípony vzorku všech možných délek od 1 do m, kde m je délka vzorku a k těmto příponám počet pozic, o které se má vzorek posunout v případě kolize.



position	mismatches	suffix shift	
9	B	A	9
8	C	BA	6
7	A	CBA	9
6	B	ACBA	9
5	C	BACBA	3
4	A	CBACBA	9
3	B	ACBACBA	9
2	D	BACBACBA	9
1	A	DBACBACBA	10
	-	ADBACBACBA	9

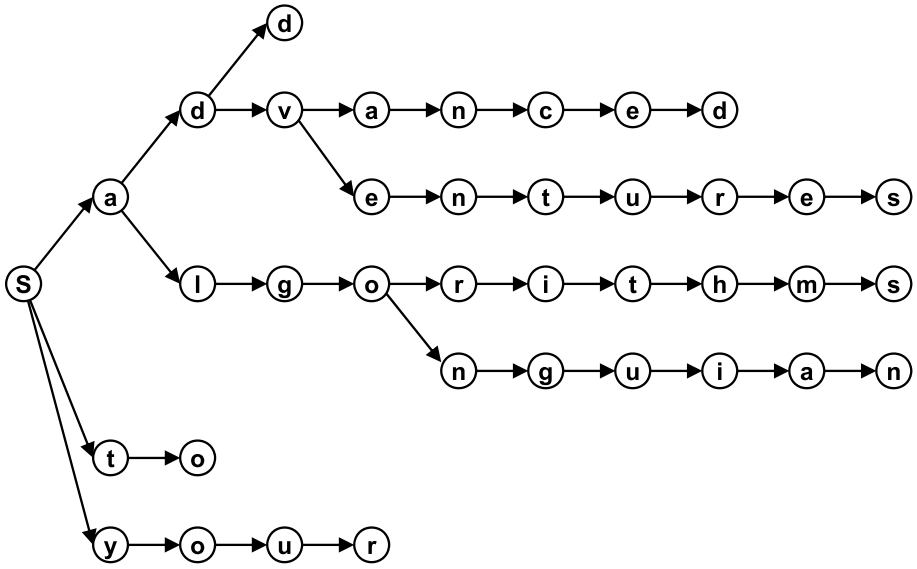
4.6 Slovníkové automaty

Slovník nad abecedou A je konečná množina řetězců (patternů) z A*. Slovníkový automat hledá text pro jakýkoliv řetězec ve slovníku.

- Abeceda: A = {a, c, d, e, g, h, i, l, m, n, o, q, r, s, t, u, v, y}
- Slovník: D = {add, advanced, algorithms, to, your, algonqiuan, adventures}

Stejné prefixy se dají mergovat do společných stavů.

Vylepšení: Identické suffixy lze mergovat do jednoho stavu, např. „add“ - lze zrušit jeden stav (koncové „d“) a dát přechod do jiného koncového „d“ (např. u slova „advanced“).



Obrázek 9: Slovníkový automat (bez suffix vylepšení)

5 Algoritmus, správnost algoritmu, složitost algoritmu, složitost úlohy, třída $\mathcal{P}$, třída $\mathcal{NP}$.

Algoritmus. *Algoritmem* rozumíme dobře definovaný proces, tj. posloupnost výpočetních kroků, který přijímá hodnoty (zadání, vstup) a vytváří hodnoty (řešení, výstup).

Řekneme, že algoritmus $\mathcal{A}$ *řeší* úlohu $\mathcal{U}$, jestliže pro každý vstup (každou instanci problému $\mathcal{U}$) vydá správné řešení.

Správnost algoritmu K ověření správnosti algoritmu je třeba ověřit 2 věci:

1. algoritmus se na každém vstupu zastaví
2. algoritmus po zastavení vydá správný výstup - řešení

Variant Důkaz faktu, že se algoritmus na každém vstupu zastaví, je založen na nalezení tzv. *variantu*. Variant je hodnota udaná přirozeným číslem, která se během práce algoritmu snižuje až nabude nejmenší možnou hodnotu (a tím zaručuje ukončení algoritmu po konečně mnoha krocích).

Invariant *Invariant*, též *podmíněná správnost algoritmu*, je tvrzení, které:

- platí před vykonáním prvního cyklu algoritmu, nebo po prvním vykonání cyklu
- platí-li před vykonáním cyklu, platí i po jeho vykonání
- při ukončení práce algoritmu zaručuje správnost řešení

5.1 Složitost algoritmu

Složitost algoritmu udává, jak je daný algoritmus rychlý (kolik provede elementárních operací) vzhledem k množině vstupních dat. Ke klasifikaci algoritmů se obvykle používá tzv. asymtotická složitost, což je rozdělení algoritmů do tříd složitostí, u kterých platí, že od určité velikosti dat, je algoritmus dané třídy vždy pomalejší než algoritmus třídy předchozí, bez ohledu na to, jestli je některý z počítačů c-násobně výkonnější (c je konstanta).

Algoritmy lze rozdělit do několika tříd složitosti na základě času a paměti, jež potřebují ke svému vykonání na různých typech Turingových strojů. [3]

Časovou složitost algoritmu udáváme jako asymptotický odhad $T(n)$ času potřebného pro vyřešení každé instance velikosti n.

5.1.1 Asymptotický růst funkcí

Definujeme několik symbolů (množin).

Symbol $\mathcal{O}$. Je dána nezáporná funkce $g(n)$. Řekneme, že nezáporná funkce $f(n)$ je $\mathcal{O}(g(n))$, jestliže existuje kladná konstanta c a přirozené číslo n_0 tak, že

$$f(n) \leq c \cdot g(n) \text{ pro všechny } n \geq n_0$$

$\mathcal{O}(g(n))$ můžeme též chápat jako třídu všech nezáporných funkcí $f(n)$:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c > 0, n_o \text{ tak, že } f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

Další symboly:

- $\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_o \text{ tak, že } f(n) \geq c \cdot g(n)\ \forall n \geq n_0\}$
- $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, n_o \text{ tak, že } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall n \geq n_0\}$
- $o(g(n)) = \{f(n) \mid \forall c > 0\ \exists n_o \text{ tak, že } 0 \leq f(n) < c \cdot g(n)\ \forall n \geq n_0\}$
- $\omega(g(n)) = \{f(n) \mid \forall c > 0\ \exists n_o \text{ tak, že } 0 \leq c \cdot g(n) < f(n)\ \forall n \geq n_0\}$

Tranzitivita $\mathcal{O}, \Omega, \Theta$. Máme dány tři nezáporné funkce $f(n), g(n), h(n)$

- Jestliže $f(n) \in \mathcal{O}(g(n))$ a $g(n) \in \mathcal{O}(h(n))$, pak $f(n) \in \mathcal{O}(h(n))$
- Jestliže $f(n) \in \Omega(g(n))$ a $g(n) \in \Omega(h(n))$, pak $f(n) \in \Omega(h(n))$
- Jestliže $f(n) \in \Theta(g(n))$ a $g(n) \in \Theta(h(n))$, pak $f(n) \in \Theta(h(n))$

Reflexivita $\mathcal{O}, \Omega, \Theta$. Pro všechny nezáporné funkce $f(n)$ platí:

- $f(n) \in \mathcal{O}(f(n))$
- $f(n) \in \Omega(f(n))$
- $f(n) \in \Theta(f(n))$

5.1.2 Master Theorem

Používá se pro určení asymptotického časového odhadu u rekurentních vztahů.
Jsou dána přirozená čísla $a \geq 1, b \geq 1$ a funkce $f(n)$. Předpokládejme, že funkce $T(n)$ je dána na přirozených číslech rekurentním vztahem

$$T(n) = aT\left(\tfrac{n}{b}\right) + f(n), \text{ kde } \tfrac{n}{b} \text{ znamená buď } \lfloor \tfrac{n}{b} \rfloor \text{ nebo } \lceil \tfrac{n}{b} \rceil.$$

1. Jesltiže $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ pro nějakou konstantu $\epsilon > 0$, pak $T(n) \in \Theta(n^{\log_b a})$.
2. Jesltiže $f(n) \in \Theta(n^{\log_b a})$, pak $T(n) \in \Theta(n^{\log_b a} \lg n)$.
3. Jesltiže $f(n) \in \Omega(n^{\log_b a + \epsilon})$ pro nějakou konstantu $\epsilon > 0$ a jestliže $af(\frac{n}{b}) \leq cf(n)$ pro nějakou konstantu $c < 1$ pro všechna dostatečně velká n, pak $T(n) \in \Theta(f(n))$.

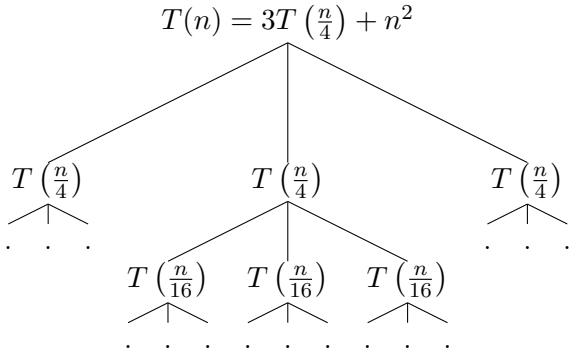
MT nepokrývá všechny případy.

► **Příklad**

$T(n) = 6T\left(\frac{n}{4}\right) + n^2.\lg(n)$ // 3. případ $n^2 \lg(n) \in \Omega(n^{\log_4 6})$
$6\left(\frac{n}{4}\right)^2 \lg\left(\frac{n}{4}\right) \leq c \cdot n^2 \lg(n)$ // roznásobení
$\frac{6}{16} n^2 \lg\left(\frac{n}{4}\right) \leq c \cdot n^2 \lg(n)$ // $\lg\left(\frac{n}{4}\right)$ si "zvětším"na $\lg(n)$
$\frac{6}{16} n^2 \lg(n) \leq c \cdot n^2 \lg(n)$ // pokrátím (vydělím) $n^2 \lg(n)$
$\frac{6}{16} \leq c$ // $c < 1$, platí
$\rightarrow T(n) = \Theta(n^2 \lg(n))$

5.1.3 Řešení rekurzivních vztahů pomocí rekurzivních stromů

► **Příklad**



Vytvoříme si jednotlivé hladiny stromu, který popisuje rekurzivní výpočet funkce $T(n)$. V nulté hladině máme pouze $T(n)$ a hodnotu n^2, kterou potřebujeme k výpočtu $T(n)$ (známe-li $T\left(\frac{n}{4}\right)$. V první hladině se nám výpočet $T(n)$ rozpadl na tři výpočty $T(n)$. K tomu potřebujeme hodnotu $3 \cdot (\frac{n}{4})^2 = \frac{3}{16} n^2$. Při přechodu z hladiny i do hladiny $i+1$ se každý vrchol rozdělí na tři a každý přispěje do celkové hodnoty jednou šestnáctinou předchozího. Proto je součet v hladině i roven $(\frac{3}{16})^i n^2$. Poslední hladina má vrcholy označené hodnotami $T(1)$ a tím rekurze končí. Počet hladin odpovídá $\log_4 n$. V poslední hladině je $3^{\log_4 n} = n^{\log_4 3}$ hodnot $T(1)$. Proto platí

$$T(n) = \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i n^2 + \Theta(n^{\log_4 3})$$

Odtud

$$T(n) < n^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) = n^2 \frac{1}{1 - \frac{3}{16}} + \Theta(n^{\log_4 3}) = \frac{16}{13} n^2 + \Theta(n^{\log_4 3})$$

Proto $T(n) \in \Theta(n^2)$

5.2 Složitost úlohy

Složitost úlohy je složitost nejlepšího algoritmu řešícího danou úlohu.

5.3 Turingův stroj (Turing machine - TM)

Je teoretický model počítače, který se skládá z:

- **řídící jednotky**, která se může nacházet v jednom z konečně mnoha stavů
- potenciálně **nekonečné pásky** (nekonečné na obě strany) rozdělené na jednotlivé pole
- **čtecí hlavy**, která umožňuje číst obsah polí a přepisovat obsah polí pásky

Na základě symbolu X, který čte hlava na pásce, a na základě stavu q, ve tkerém se nachází řídící jednotka, se řídící jednotka Turingova stroje přesune do stavu p, hlava přepíše obsah čteného pole na Y a přesune se buď doprava nebo doleva (tato akce je popsána tzv. přechodovou funkcí).

Je dán sedmicí $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, kde

- Q je konečná množina stavů
- Σ je konečná množina vstupních symbolů
- Γ je konečná množina páskových symbolů, přitom $\Sigma \subset \Gamma$
- B - je prázdný symbol (*blank*), jedná se o páskový symbol, který není vstupním symbolem (tj. $B \in \Gamma \setminus \Sigma$)
- δ je přechodová funkce, tj. parciální zobrazení z množiny $(Q \setminus F) \times \Gamma$ do množiny $Q \times \Gamma \times L, R$, (zde L znamená pohyb hlavy o jedno pole doleva, R pohyb doprava)
- $q_0 \in Q$ je počáteční stav
- $F \subset Q$ je množina koncových stavů

Nedeterministický TM Je takový Turingův stroj, u kterého připustíme, aby v jedné situaci mohl provézt několik různých kroků.

Přijímaný a rozhodovaný jazyk TM. Vstupní slovo $w \in \Sigma^*$ je *přijato* Turingovým strojem M právě tehdy, když se Turingův stroj na slově w úspěšně zastaví. Množinu slov $w \in \Sigma^*$, které Turingův stroj přijímá, se nazývá *jazyk přijímaný* M a značíme ho $L(M)$.

Turingův stroj *rozhoduje* jazyk L, jestliže tento jazyk přijímá a navíc se na každém vstupu zastaví.

5.4 Třída složitosti - $\mathcal{P}$

Třída $\mathcal{P}$ Řekneme, že rozhodovací úloha $\mathcal{U}$ leží ve třídě $\mathcal{P}$, jestliže **existuje deterministický** Turingův stroj, který **rozhodne** jazyk $L_{\mathcal{U}}$ a pracuje v **polynomiálním čase**; tj. funkce $T(n)$ je $\mathcal{O}(p(n))$ pro nějaký polynom $p(n)$.

Příklady $\mathcal{P}$ úloh:

- **Minimální kostra v grafu**. Je dán neorinetovaný graf G s ohodnocením hran c. Je dáno číslo k. Existuje kostra grafu ceny menší nebo rovno k?
- **Nejkratší cesty v acyklickém grafu**. Je dán acyklický graf s ohodnocením hran a. Jsou dány vrcholy r a c. Je dáno číslo k. Existuje orientovaná cesta z vrcholu r do vrcholu c délky menší nebo rovno k?
- **Toky v sítích**. Je dána síť s horním omezením c, dolním omezením l, se zdrojem z a spotřebičem s. Dále je dáno číslo k. Existuje přípustný tok od z do s velikosti alespoň k?
- **Minimální řez**. Je dána síť s horním omezením c, dolním omezením l. Dále je dáno číslo k. Existuje řez, který má kapacitu menší nebo rovnu k?

5.5 Třída složitosti - $\mathcal{NP}$

Třída $\mathcal{NP}$ Řekneme, že rozhodovací úloha $\mathcal{U}$ leží ve třídě $\mathcal{NP}$, jestliže **existuje nedeterministický** Turingův stroj, který **rozhodne** jazyk $L_{\mathcal{U}}$ a pracuje v **polynomiálním čase**.

Příklady $\mathcal{NP}$ úloh:

- **Kliky v grafu**. Je dán neorinetovaný graf G a číslo k. Existuje klika v grafu G o alespoň k vrcholech?
- **Nejkratší cesty v obecném grafu**. Je dán orientovaný graf s ohodnocením hran a. Jsou dány vrcholy r a v. Je dáno číslo k. Existuje orientovaná cesta z vrcholu r do vrcholu c délky menší nebo rovno k?
- **k-barevnost**. Je dán neorientovaný graf G. Je graf G k-barevný?
- **Knapsack**. Je dáno n předmětů $1, 2, \ldots, n$. Každý předmět i má cenu c_i a váhu w_i. Dále jsou dána čísla A a B. Je možné vybrat předměty tak, aby celková váha nepřevýšila A a celková cena byla alespoň B?

Otázka obsahuje texty, úryvky a definice z [6].

6 $\mathcal{NP}$-úplné a $\mathcal{NP}$-těžké úlohy, Cookova věta, heuristiky na řešení $\mathcal{NP}$-těžkých úloh, pravděpodobnostní algoritmy.

Než nadefinujeme třídu $\mathcal{NPC}$, musíme definovat (polynomiální) *redukci úloh.*

Redukce a polynomiální redukce úloh. Jsou dány dvě rozhodovací úlohy $\mathcal{U}$ a $\mathcal{V}$. Řekneme, že úloha $\mathcal{U}$ se *redukuje* na úlohu $\mathcal{V}$, jestliže existuje algoritmus (program pro RAM, Turingův stroj) M, který pro každou instanci I úlohy $\mathcal{U}$ zkonstruuje instanci I' úlohy $\mathcal{V}$ a to tak, že

$$I \text{ je ANO-instance } \mathcal{U} \text{ iff } I' \text{ je ANO-instance } \mathcal{V}$$

Fakt, že úloha $\mathcal{U}$ se redukuje na úlohu $\mathcal{V}$ značíme

$$\mathcal{U} \lhd \mathcal{V}.$$

Jestliže navíc, algoritmus M pracuje v polynomiálním čase, říkáme, že $\mathcal{U}$ se *polynomiálně* redukuje na $\mathcal{V}$ a značíme

$$\mathcal{U} \lhd_p \mathcal{V}.$$

6.1 Třída složitosti - $\mathcal{NPC}$ ($\mathcal{NP}$-complete, $\mathcal{NP}$-úplná)

$\mathcal{NP}$ úplné úlohy. Řekneme, že rozhodovací úloha $\mathcal{U}$ je $\mathcal{NP}$ *úplná* , jestliže

1. $\mathcal{U}$ je ve třídě $\mathcal{NP}$
2. každá $\mathcal{NP}$ úloha se polynomiálně redukuje na $\mathcal{U}$.

Příklady $\mathcal{NPC}$ úloh:

- **SAT** Splnitelnost formulí v konjunktivním normálním tvaru.
- **3 - CNF SAT**
- **3-barevnost**
- **ILP**

6.2 Třída složitosti - $\mathcal{NP}$-hard ($\mathcal{NP}$-těžká)

$\mathcal{NP}$ obtížné úlohy. Jestliže o některé úloze $\mathcal{U}$ pouze víme, že se na ní polynomiálně redukuje některá $\mathcal{NP}$ úplná úloha, pak říkáme, že $\mathcal{U}$ je $\mathcal{NP}$ těžká, nebo též $\mathcal{NP}$ obtížná. Poznamenejme, že to vlastně znamená, že $\mathcal{U}$ je alespoň tak těžká jako všechny $\mathcal{NP}$ úlohy.

6.3 Cookova věta

Dle Cookovy věty lze převést v polynomiálním čase libovolný nedeterministický Turingův stroj na problém splnitelnosti booleovských formulí v konjunktivním normálním tvaru (CNF SAT).

Důsledkem této věty je vymezení skupiny úloh, které jsou nejtěžší v rámci všech problémů třídy NP. O těchto úlohách, na které lze převést v polynomiálním čase libovolnou jinou úlohu z NP, říkáme, že jsou $\mathcal{NP}$-úplné ($\mathcal{NP}$-complete).

Důkaz. Není těžké se přesvědčit že úloha SAT je ve třídě $\mathcal{NP}$. První fáze nedeterministického algoritmu vygeneruje ohodnocení logických proměnných a na základě tohoto ohodnocení jsme schopni v polynomiálním čase ověřit, zda je v tomto ohodnocení formule pravdivá nebo ne.

Druhá část důkazu spočívá v popisu práce TM formulí výrokové logiky. Načrtneme si základní myšlenku tohoto popisu.

Je dán NTM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Předpokládejme, že M přijímá slovo w a potřebuje při tom $p(n)$ kroků.

Zavedeme logické proměnné:

- $h_{i,j},\ i = 0, 1, \ldots, p(n); j = 1, 2, \ldots, p(n);$
  - $h_{i,j}$ je rovna 1, pokud hlava TM v čase i čte j-té pole pásky.
- $s_i^q,\ i = 0, 1, \ldots, p(n); q \in Q$
  - s_i^q je rovna 1, pokud TM v čase i je ve stavu q.
- $t_{i,j}^A,\ i = 0, 1, \ldots, p(n); j = 1, 2, \ldots, p(n); A \in \Gamma$
  - $t_{i,j}^A$ je rovna 1, pokud v čase i v j-té poli pásky je páskový symbol A.

Nyní je třeba formulemi popsat následující fakta:

1. V každém okamžiku je TM v právě jednom stavu.
2. V každém okamžiku čte hlava TM právě jedno pole vstupní pásky.
3. V každém okamžiku je na každém poli pásky TM právě jeden páskový symbol.
4. Na začátku práce (tj. v čase 0) je TM ve stavu q_0, hlava čte první pole pásky a na pásce je na prvních n polích vstupní slovo, ostatní pole pásky obsahují B.
5. Krok TM je určen přechodovou funkcí, tj. stav stroje, obsah čteného pole a poloha hlavy v čase $i + 1$ je dána přechodovou funkcí.
6. V polích pásky, které v čase i hlava nečte, je obsah v čase $i + 1$ stejný jako v i.
7. Na konci práce TM , tj. v čase $p(n)$, je stroj ve stavu q_f.

Ukážeme jak utvořit formule pro body **1**, **4**, **5**, **6** a **7**.

Bod **1**. V okamžiku i je TM v aspoň jednom stavu:

$$\bigvee_{q \in Q} s_i^q.$$

V okamžiku i TM není ve dvou různých stavech:

$$\bigwedge_{q \neq q'} (\neg s_i^q \vee \neg s_i^{q'}).$$

Nyní fakt, že TM je v okamžiku i právě v jednom stavu je konjunkce obou výše uvedených formulí:

$$(\bigvee_{q \in Q} s_i^q) \wedge \bigwedge_{q \neq q'} (\neg s_i^q \vee \neg s_i^{q'}).$$

Bod **4**. Na začátku práce (tj. v čase 0) je TM ve stavu q_0, hlava čte první pole pásky a na pásce je na prvních n polích vstupní slovo $a_1 a_2 \dots a_n$, ostatní pole pásky obsahují B.

$$s_0^{q_0} \wedge h_{0,1} \wedge t_{0,1}^{a_1} \wedge \dots \wedge t_{0,n}^{a_n} \wedge t_{0,n+1}^{B} \wedge \dots \wedge t_{0,p(n)}^{B}.$$

Bod **5**. Jestliže TM je v čase i ve stavu q, hlava je na j-tém poli pásky, hlava čte páskový symbol A a $\delta(q, A)$ se skládá z trojic (p, C, D) (zde $D = 1$ znamená posun hlavy doprava, $D = -1$ znamená posun hlavy doleva), pak formule má tvar:

$$\bigwedge_j \bigwedge_{A \in \Gamma} ((s_i^q \wedge h_{i,j} \wedge t_{i,j}^A) \Rightarrow \bigvee (s_{i+1}^p \wedge t_{i+1,j}^C \wedge h_{i+1,j+D})).$$

Bod **6**. Obsah polí kromě j-tého zůstává v čase $i + 1$ stejný:

$$\bigwedge_j \bigwedge_{A \in \Gamma} ((\neg h_{i,j} \wedge t_{i,j}^A) \Rightarrow t_{i+1,j}^A).$$

Bod **7**. Na konci práce TM, tj. v čase $p(n)$ je stroj ve stavu q_f:

$$s_{p(n)}^{g_f}.$$

Výslednou formuli dostaneme jako konjunkci všech dílčích formulí pro všechny časové okamžiky $i = 0, 1, \dots, p(n)$.

6.4 Heuristiky na řešení $\mathcal{NP}$-těžkých úloh

Jestliže je třeba řešit problém, který je $\mathcal{NP}$ úplný, musíme pro větší instance opustit myšlenku přesného nebo optimálního řešení a smířit se s tím, že získáme „dostatečně přesné“ nebo „dostatečně kvalitní“ řešení. K tomu se používají heuristické algoritmy pracující v polynomiálním čase. Algoritmům, kde umíme zaručit „jak daleko“ je nalezené řešení od optimálního, se také říká **aproximační** algoritmy.

Trojúhelníková nerovnost. Řekneme, že instance obchodního cestujícího splňuje trojúhelníkovou nerovnost, jestliže pro každá tři města i, j, k platí:

$$d(i,j) \leq d(i,k) + d(k,j).$$

6.4.1 2-aproximační algoritmus

Jestliže instance I obchodního cestujícího splňuje trojúhelníkovou nerovnost, pak existuje polynomiální algoritmus $\mathcal{A}$, který pro I najde trasu délky D, kde $D \leq 2OPT(I)$ ($OPT(I)$ je délka optimální trasy v I).

Slovní popis algoritmu. Instanci I považujeme za úplný graf G s množinou vrcholů $V = \{1, 2, \ldots, n\}$ a ohodnocením d.

1. V grafu G najdeme minimální kostru (V, K).
2. Kostru (V, K) prohledáme do hloubky z libovolného vrcholu.
3. Trasu T vytvoříme tak, že vrcholy procházíme ve stejném pořadí jako při prvním navštívení během prohledávání grafu. T je výstupem algoritmu.

Zřejmě platí, že délka kostry K je menší než $OPT(I)$. Ano, vynecháme- li z optimální trasy některou hranu, dostaneme kostru grafu G. Protože K je minimální kostra, musí být délka K menší než $OPT(I)$ (předpokládáme, že vzdálenosti měst jsou kladné). Vzhledem k platnosti trojúhelníkové nerovnosti, je délka T menší nebo rovna dvojnásobku délky kostry K.

6.4.2 Christofidesův algoritmus

Jestliže instance I obchodního cestujícího splňuje trojúhelníkovou nerovnost, pak následující algoritmus najde trasu T délky D takovou, že $D \leq \frac{3}{2}OPT(I)$.

Instanci I považujeme za úplný graf G s množinou vrcholů $V = \{1, 2, \ldots, n\}$ a ohodnocením d.

1. V grafu G najdeme minimální kostru (V, K).
2. Vytvoříme úplný graf H na množině všech vrcholů, které v kostře (V, K) mají lichý stupeň.
3. V grafu H najdeme nejlevnější perfektní párování P[1].
4. Hrany P přidáme k hranám K minimální kostry. Graf $(V, P \cup K)$ je eulerovský graf. V grafu $(V, P \cup K)$ sestrojíme uzavřený eulerovský tah.
5. Trasu T získáme z eulerovského tahu tak, že vrcholy navštívíme v pořadí, ve kterém jsme do nich poprvé vstoupili při tvorbě eulerovského tahu.

Platí, že délka takto vzniklé trasy je maximálně $\frac{3}{2}$ krát větší než délka optimální trasy.

[1] Párování grafu je v teorii grafů taková podmnožina hran grafu, že žádné dvě hrany z této množiny nemají společný vrchol. (Idea je taková, že vrcholy grafu dáváme do párů. Pár může vzniknout jen tam, kde byla hrana. Přitom každý vrchol může být jen v jednom páru.) Perfektní párování grafu je párování, které pokrývá všechny vrcholy grafu [9].

6.5 Pravděpodobnostní algoritmy

Randomizovaný Turingův stroj (RTM). RTM je, zhruba řečeno, Turingův stroj M se dvěma nebo více páskami (pásky > 2 obsahují B), kde první páska má stejnou roli jako u deterministického Turingova stroje, ale druhá páska obsahuje náhodnou posloupnost 0 a 1, tj. na každém políčku se 0 objeví s pravděpodobností $\frac{1}{2}$ a 1také s pravděpodobností $\frac{1}{2}$.

Třída $\mathcal{RP}$. Jazyk L patří do třídy $\mathcal{RP}$ právě tehdy, když existuje RTM M takový, že:

1. Jestliže $w \notin L$, stroj M se ve stavu q_f zastaví s pravděpodobností 0.
2. Jestliže $w \in L$, stroj M se ve stavu q_f zastaví s pravděpodobností, která je alespoň rovna $\frac{1}{2}$.
3. Existuje polynom $p(n)$ takový, že každý běh M (tj. pro jakýkoli obsah druhé pásky) trvá maximálně $p(n)$ kroků, kde n je délka vstupního slova.

Příklady $\mathcal{RP}$ úloh:

- **Miller-Rabinův test prvočíselnosti**

TM typu Monte-Carlo. RTM splňující podmínky 1 a 2 z definice $\mathcal{RP}$ se nazývá TM typu *Monte-Carlo* (obecně nemusí pracovat v polynomiálním čase).

6.5.1 Třída $\mathcal{ZPP}$

Jazyk L patří do třídy $\mathcal{ZPP}$ právě tehdy, když existuje RTM M takový, že:

1. Jestliže $w \notin L$, stroj M se ve stavu q_f zastaví s pravděpodobností 0.
2. Jestliže $w \in L$, stroj M se ve stavu q_f zastaví s pravděpodobností 1.
3. Střední hodnota počtu kroků M v jednom běhu je $p(n)$, kde $p(n)$ je polynom a n je délka vstupního slova.

To znamená: M neudělá chybu, ale nezaručujeme vždy polynomiální počet kroků při jednom běhu, pouze střední hodnota počtu kroků je polynomiální.

TM typu Las-Vegas. RTM splňující podmínky z definice $\mathcal{ZPP}$ se nazývá TM typu *Las-Vegas*.

Otázka obsahuje texty, úryvky a definice z [6].

7 Turingovy stroje, rekurzivní a rekurzivně spočetné jazyky, algoritmicky neřešitelné úlohy.

Turingovy stroje viz předchozí otázky. TODO: možná nějaký příklad TM

7.1 Rekurzivní jazyky

Řekneme, že jazyk L je *rekurzivní*, jestliže existuje Turingův stroj M, který rozhoduje jazyk L.

Připomeňme, že Turingův stroj M rozhoduje jazyk L znamená, že jej přijímá a na každém vstupu se zastaví (buď úspěšně nebo neúspěšně).

7.2 Rekurzivně spočetné jazyky

Řekneme, že jazyk L je *rekurzivně spočetný*, jestliže existuje Turingův stroj M, který tento jazyk přijímá.

Jinými slovy, M se pro každé slovo w, které patří do L, úspěšně zastaví a pro slovo w, které nepatří do L se buď zastaví neúspěšně nebo se nezastaví vůbec.

Jazykům, které **nejsou rekurzivní**, také říkáme, že jsou *algoritmicky neřešitelné* nebo *nerozhodnutelné*. Obdobně mluvíme o úlohách, které jsou nerozhodnutelné nebo **algoritmicky neřešitelné**.

- Diagonální jazyk
- Univerzální jazyk

8 Metoda větví a mezí. Algoritmy pro celočíselné lineární programování. Formulace optimalizačních a rozhodovacích problémů pomocí celočíselného lineárního programování. Toky a řezy. Multi-komoditní toky.

8.1 Metoda větví a mezí (Branch and Bound)

Prozkoumávání stavového stromu všech možností. Uzly představují částečná řešení problému. Listy jsou konečná řešení. Během procházení stromu lze odřezávat celé větve, které jsou buď nepřípustné, nebo nejsou lepší než dosud nalezené řešení.

8.2 ILP - celočíselné lineární programování

Úloha celočíselného lineárního programování (LP) je zadána maticí $\mathbf{A} \in \mathbf{R}^{m \times n}$ a vektory $b \in \mathbf{R}^m, c \in \mathbf{R}^n$. Cílem je najít takový vektor $x \in \mathbf{Z}^n$, že platí $\mathbf{A} \cdot x \leq b$ a $c^T \cdot x$ je maximální.

Obvykle se celočíselné lineární programování zapisuje ve tvaru:

$$\begin{aligned} \max \quad & \mathbf{c}^T \cdot \mathrm{x} \\ & \mathbf{A} \cdot \mathrm{x} \leq \mathbf{b} \end{aligned}$$

Pokud bychom takovou úlohu řešili pomocí lineárního programování s tím, že bychom výsledek zaokrouhlili, nejenom že bychom neměli zaručeno že výsledné řešení bude optimální ale ani to, zda bude přípustné. Zatímco úloha LP je řešitelná v polynomiálním čase, úloha ILP je tzv. NP-těžká (NP-hard), neboli není znám algoritmus, který by vyřešil libovolnou instanci této úlohy v polynomiálním čase. Protože prostor řešení ILP není konvexní množina, nelze přímo aplikovat metody konvexní optimalizace.

8.3 Algoritmy pro celočíselné lineární programování

1. Výčtové metody (Enumerative Methods)
2. Metoda větví a mezí (Branch and Bound)
3. Metody sečných nadrovin (Cutting Planes Methods)

8.3.1 Výčtové metody (Enumerative Methods)

Výpočet je založen na prohledávání oblasti zahrnující všechna přípustná řešení (Vyzkouším všechna celá čísla v dané množině). Vzhledem k celočíselnému omezení proměnných je počet těchto řešení konečný, ale jejich počet je extrémně vysoký. Proto je tato metoda vhodná pouze pro malé problémy s omezeným počtem diskrétních proměnných. Postup je možno zobecnit na úlohu MIP (mixed IP) tak, že ke každé kombinaci diskrétních proměnných je vyřešena úloha LP kde jsou diskrétní proměnné považovány za konstanty [4].

8.3.2 Branch & Bound

Spočítáme LP řešení - pokud je výsledek celočíselný, accept. Jinak jednu z proměnných zaokrouhlíme dolů a zkoumáme případ $\leq$ zaokrouhlení a $>$ zaokrouhlení. Znovu se spouští LP pro menší oblasti, dokud není vše celočíselné. Odřezávám nepřípustná řešení a horší než zatím nejlepší.

8.3.3 Metody sečných nadrovin (Cutting Planes Methods)

Další skupinou algoritmů jsou metody sečných nadrovin (cutting plane methods), založené podobně jako metoda větví a mezí na opakovaném řešení úlohy LP. Výpočet je prováděn iterativně tak, že v každém kroku je přidána další omezující podmínka zužující oblast přípustných řešení. Každá nová omezující podmínka musí splňovat tyto vlastnosti:

1. Optimální řešení nalezené pomocí LP se stane nepřípustným.
2. Žádné celočíselné řešení přípustné v předchozím kroku se nesmí stát nepřípustným.

Nové omezení splňující tyto vlastnosti je přidáno v každé iteraci. Vzniklý ILP program je vždy znovu řešen jako úloha LP. Proces je opakován, dokud není nalezeno přípustné celočíselné řešení. Konvergence takovéhoto algoritmu potom závisí na způsobu přidávání omezujících podmínek. Mezi nejznámější metody patří Dantzigovi řezy (Dantzig cuts) a Gomoryho řezy (Gomory cuts) [4].

8.3.4 Formulace optimalizačních a rozhodovacích problémů pomocí ILP.

2-partition problem Je $n \in Z^+$ bankovek a jejich hodnoty $p_1, \ldots, p_n$. Existuje taková podmnožina bankovek S, která rozdělí celkovou hodnotu bankovek na půl? Matematicky zapsáno $S \subseteq \{1, \ldots, n\}$, kde $\sum_{i \in S} p_i = \sum_{i \notin S} p_i$.

$$
\begin{aligned}
\text{min} \quad & 0 \\
\text{omezení:} \quad & \sum_{i \in 1..n} x_i \cdot p_i = 0,5 \cdot \sum_{i \in 1..n} p_i \\
\text{parametry:} \quad & n \in Z^+, p_{\in 1..n} \in Z^+ \\
\text{proměnné:} \quad & x_{i \in 1..n} \in \{0, 1\}
\end{aligned}
$$

Ze slidů:

- nemovitosti
- trika a kalhoty
- big M

8.4 Toky a řezy

Toky v síti, kde síť je pětice (G, l, u, s, t), kde:

- G je orientovaný graf
- l je dolní omezení hran
- u je horní omezení hran
- s je zdroj a t spotřebič

Tok je takové ohodnocení hran, kde pro každý vrchol platí Kirchhoffův zákon: co tam vteče taky vyteče (a tok je v mezích $\langle l, u \rangle$), kromě zdroje a spotřebiče.

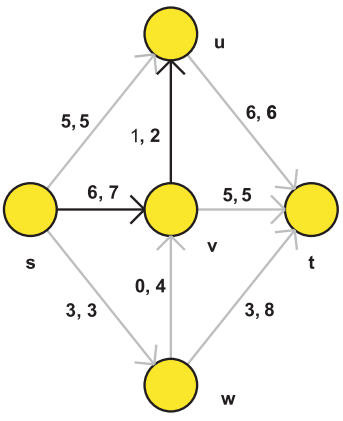
8.4.1 Ford-Fulkerson

Ford-Fulkerson hledá **maximální tok** v síti. Začnu libovolným přípustným tokem a postupně hledám zlepšující se cesty. Když taková cesta existuje, zvednu tok na té cestě. Když neexistuje, máme maximální tok. Zvedání toku se dělá tak, že zvednu tok na hraně vpřed a snížím tok na hraně vzad o rozdíl do (horní, dolní) kapacity cesty.

Jak najít zlepšující se cestu: Na začátku všechny hrany označím FALSE, zdroj TRUE. Když najdu hranu TRUE → FALSE a jde navýšit (hrana vpřed) tak označím druhou také TRUE. Když najdu FALSE → TRUE a jde snížit (hrana vzad), tak označím první TRUE.

Když naleznu spotřebič TRUE mám zlepšující cestu, tu navýším a mohu začít hledat novou zlepšující cestu.

Řez je množina vrcholů, ve které je zdroj ale není spotřebič Minimální řez je řez s minimální kapacitou, který odděluje zdroj a spotřebič.



Spočítá se Ford-Fulkensonem - vezmu TRUE vrcholy, když už nemůžeme najít zlepšující se cestu. Sečtu toky hran, které mi znemožnili zlepšení (tzn. ty které jsou nasycené, i ty zpětné!). **Kapacita řezu** je **rovna** velikosti **maximálního toku**.

V příkladě na obrázku chceme najít minimální řez. O něm víme, že jeho vrcholy získáme pomocí značkovací metody Ford-Fulkensona. Ta nejdřív označí zdroj, pak označí vrchol v, protože jinam nemůže (všude je nasyceno). Následně označí vrchol u a z něho dál již nelze jít (všude je opět nasyceno). Tím jsou určeny vrcholy řezu $(\{s, v, u\})$.

Kapacita min. řezu je určena jako součet toků hran, které zabránili zlepšení (a nevedou uvnitř řezu). Tzn. hrana 6/6, 5/5, 0,4 a 3/3. Toky hran které „odchází“ sečteme, a toky zpětných hran (které přichází) odečteme → $6 + 5 - 0 + 3 = 14$. Kapacita řezu je rovna 14, stejně tak jako hodnota maximálního toku.

Párování je množina hran, kde žádné dvě nemají společný vrchol. Nebo také rozdělení vrcholů na dvě skupiny kde uvnitř skupiny nejsou hrany. Používá se k přiřazování lidí k taskům. Jde převést na toky nebo Maďarským algoritmem.

Tokama lze řešit velké množství problémů:

- transport zásob
- zaokrouhlování v excelu
- multiprocesorové rozvrhování

8.5 Multi-komoditní toky

Tok, kde teče víc komodit. Řeší se pomocí LP nebo ILP - nutné nadefinovat Kirchhoffův zákon pro každou komoditu.

9 Nejkratší cesty. Úloha obchodního cestujícího. Heuristiky a aproximační algoritmy. Metoda dynamického programování. Problém batohu. Pseudo-polynomiální algoritmy.

9.1 Problém nejkratší cesty

Nejkratší cesta je problém nalezení nejkratší cesty z x do y. V obecném grafu se zápornými cykly je to NP-úplná úloha.

Bellmanova rovnice: Polynomiální řešení problému nejkratších cest je založeno na Bellmanovu principu optimality, který říká, že **pokud** graf **neobsahuje cyklus záporné délky**, tak pro každé tři vrcholy x, y, z platí:

$$u(x, y) = \min_{x \neq y}(u(x, z) + a(z, y))$$

Kde $u(x, y)$ značí délku cesty z vrcholu x do vrcholu y a $a(z, y)$ značí vzdálenost vrcholu z od vrcholu y (tj. délku nejkratší hrany, která tyto vrcholy spojuje).

Z Bellmanovy rovnice jednoduše řečeno vyplývá, že se každá **nejkratší cesta skládá z nejkratších cest** – tj. nejkratší cesta $[a, \cdots, x, \cdots, c]$ mezi uzly a a b obsahuje také nejkratší cestu mezi uzly a, x a x, c.

Polynomiální algoritmy pro nejkratší cesty:

Dijkstra: $O(E \cdot \log V)$ Funguje **pokud nejsou záporné hrany**. Algoritmus má uzly v prioritní frontě podle vzdálenosti od zdroje. Na začátku zdroj 0, ostatní nekonečno. Vždy odebere uzel s nejmenší vzdáleností a provede tzv. relaxaci - podívá se na všechny sousedy ještě ve frontě a ověří, jestli se tam přes tento uzel nejde dostat rychleji. Pokud ano, sníží prioritu. Když odebereme cílový uzel, algoritmus končí. Je dobré si zaznamenávat předchůdce.

Bellman-Ford: $O(E \cdot V)$ Podobný Dijkstrovi, ale funguje pro hrany záporné délky. Může také **detekovat záporný cyklus**.

Základem Bellman-Fordova algoritmu je operace relaxace. Do této operace vstupují dva uzly a hrana, která mezi nimi vede. Pokud je vzdálenost zdrojového uzlu sečtená s délkou hrany menší než aktuální vzdálenost cílového uzlu, tak se za předchůdce cílového uzlu na nejkratší cestě označí zdrojový uzel (a vzdálenost cílového uzlu se přepočítá). V případě nesplnění nerovnosti tato hrana cestu nezkracuje a neprovádí se proto žádné změny.

Délka cesty ze zdrojového do každého z cílových uzlů může být dlouhá maximálně |V|-1 hran (protože by jinak musela obsahovat cyklus). Proto pokud pustíme operaci relaxace na všechny hrany grafu |V|-1 krát, tak již musí být nalezeny všechny nejkratší cesty. Toto ověříme ještě jedním spuštěním relaxací všech hran. Pokud dojde k nějaké relaxaci, tak graf obsahuje cyklus záporné délky, pokud k relaxaci nedojde, algoritmus může vrátit výsledek.

Floyd-Warshall: $O(E^2 \cdot V)$ Najde nejkratší cesty od všech uzlů ke všem, detekuje záporné cykly. Nejdřív matice kdo s kým sousedí, iteruju $k = 1..n$. Procházím všechny prvky a dívám se co protínám v k-tém řádku a sloupci. Sečtu ty prvky. Když je součet menší, nahradím. Při náhradách je dobré vést matici předchůdců. Když bude < 0 na diagonále - záporný cyklus.

Existence hamiltonovské kružnice Je neorientovaný graf a máme rozhodnout, zda existuje HC (kružnice která navštíví každý vrchol přesně jednou). Je to **NP-úplný problém**.

9.2 Úloha obchodního cestujícího (TSP)

Problém je NP-úplný a silně NP-obtížný, což znamená, že pokud platí $P \neq NP$, pak pro problém obchodního cestujícího **neexistuje** žádný polynomiální **k-aproximační** algoritmus - neexistuje polynomiální algoritmus, který by našel libovolné řešení, které je nejhůře k-násobkem optimálního řešení.

Důkaz NP-hard: Redukcí z existence Hamiltonovské kružnice.

Mějme neorientovaný graf G, kde rozhodujeme jestli tam je hamiltonovská kružnice. Vytvoříme instanci TSP tak, že vytvoříme uplný graf K. Každý vrchol z G je přiřazen jednomu vrcholu v K. Cena hran $\{i, j\}$ v K je rovna:

$$c(\{i,j\}) = \begin{cases} 1 & \text{pokud hrana } \{i,j\} \text{ je v } G \\ 2 & \text{pokud hrana } \{i,j\} \text{ není v } G \end{cases}$$

G má hamiltonovskou kružnici právě tehdy, když optimální řešení TSP se rovná n. $\Rightarrow$ TSP je silně NP-hard.

9.2.1 Metrický obchodní cestující

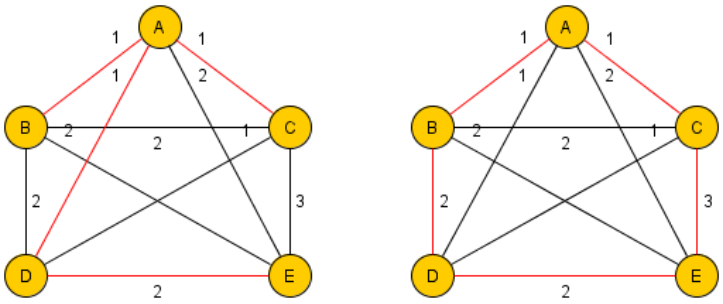
Variantou tohoto problému je problém metrického obchodního cestujícího, ve kterém vzdálenosti na grafu splňují trojúhelníkovou nerovnost. Toto zjednodušení odpovídá velkému množství reálných problémů (např. hledání na mapě), a zároveň umožňuje konstrukci aproximačních algoritmů.

2-aproximační algoritmus: Nalezne se kostra Kruskalem a vypíše se první výskyt uzlů při procházení do hloubky.

Algoritmus nejprve zkonstruuje minimální kostru grafu. Z definice kostry plyne, že cena(kostra) $\leq$ cena(optimum) protože kostra obsahuje $|V| - 1$ minimálních hran, zatímco kružnice jich obsahuje $|V|$.

V druhém kroku projde algoritmus kostru z libovolného uzlu do hloubky a poznamená si všechny průchody přes vrcholy - protože se jedná o průchod do hloubky, budou zde některé uzly zpracovány vícekrát.

V posledním kroku - zkrácení cest - algoritmus tento seznam projde a vynechá všechny duplicity (zanechá pouze první výskyty uzlů). Tímto dojde k vytvoření samotné kružnice.



Obrázek 10: 2-aproximační alg pro TSP: např. A,B,D,E,C

3/2-aproximační algoritmus (Christofidesův): Christofidesův algoritmus řeší problém metrického obchodního cestujícího tak, že je výsledná trasa v nejhorším případě dlouhá 3/2 délky trasy optimálního řešení. Toto zlepšení je ovšem vykoupeno výrazně obtížnější implementací, a zároveň se na reálných datech ukazuje, že výsledek není v průměrném případě o mnoho lepší než při použití 2-aproximačního algoritmu uvedeného výše.

Christofidesův algoritmus nejprve zkonstruuje **minimální kostru grafu**. Poté kostru **projde** z libovolného uzlu **do hloubky** a **vybere** ty **uzly**, jež mají **lichý stupeň** a zkonstruuje na nich **úplný graf** G. Na grafu G nalezne **nejlevnější perfektní párování** P. Hrany z P přidá do minimální kostry. Graf $K \cup P$ je nyní **eulerovský** (tzn. existuje v něm tah, který obsahuje všechny hrany grafu). Algoritmus nyní nalezne eulerovský tah – výsledná trasa odpovídá pořadí prvních návštěv uzlů při konstrukci tohoto tahu.

9.3 Batoh (Knapsack)

Problém batohu (Knapsack) řeší problém, které předměty dát do batohu tak, aby nebyla překročena kapacita W a celková cena C byla maximální.

2-aproximační alg $O(n^2)$ Předpokladem je, že každý z předmětů má nižší váhu než je kapacita batohu W (těžší můžeme vypustit). A součet těchto vah je naopak větší než kapacita batohu (kdyby měli menší nebo roven, tak již máme optimální řešení).

1. Seřadíme sestupně předměty podle jejich poměru cena/váha $\frac{c_i}{w_i}$
2. Z takto seřazených předmětů vezmeme nejmenší část předmětů h, která přeleze přes kapacitu W, tzn. $h = \min\{j \in \{1, \dots n\} : \sum_{i=1}^{j} w_i > W\}$
3. Nakonec se vezme lepší ze dvou řešení $\{1, \dots, h-1\}$ nebo $\{h\}$

9.4 Dynamické programování

Algoritmus, kde jsou udržovány mezivýsledky (něco jako cache na výsledky), které jsou dále využívány.

Dynamické programování na problém batohu Dynamické programování může vyřešit pseudo-polynomiálně. Mám tabulku číslo rozhodnutí x váha. Vždy se větvím na dva - přidám nebo nepřidám. Posunu se dolů o 1 a vpravo kolik tím přibyde váhy. Když překročím kapacitu, ořez. Do políček zapisuju celkovou cenu. Když už na políčku něco je, nahrazuji jenom menší cenu. Nejvíc vpravo dole bude optimální řešení.

x_i \ Σ	0	1	2	3	4	5	6	7	8	9	10	11	12
0 (v/w)	0												
1 (3/3)	0			3									
2 (5/6)	0			3		6			~~9~~				
3 (3/4)	0			3		6	7		~~10~~				
4 (1/1)	0	1		3	4	6	7	8					

Obrázek 11: Batoh o kapacitě 8 a předměty s váhami: $[3, 6, 4, 1]$, hodnotami $[3, 5, 3, 1]$

9.5 Pseudo-polynomiální algoritmy

Pseudo-polynomiální algoritmy mají složitost $O(c \cdot n)$, kde c nezáleží na velikosti instance (n) a může být hodně velké, až exponenciální. Např. algoritmus dynamického programování pro Knapsack.

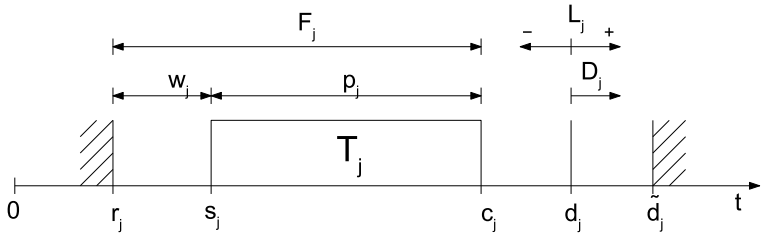
10 Rozvrhování na jednom procesoru a na paralelních procesorech. Rozvrhování projektu s časovými omezeními. Programování s omezujícími podmínkami.

10.1 Rozvrhování

Rozvhrování obecně je přiřazení úloh zdrojům v čase. Na vstupu je typicky množina úloh k rozvržení, každá z nich má své parametry (délka zpracování, termín dokončení apod.). Dále jsou v problému zdroje (typicky nějaké stroje nebo lidská síla), ty jsou určeny počtem (kapacitou) a každý jednotka zdroje může mít v čase přiřazen max. jednu úlohu. Cílem je rozvrhnout (přiřadit) všechny úlohy v čase při dodržení všech omezení. Přičemž rozvrh by měl být v nějakém smyslu optimální - to určuje kriteriální funkce (např. minimální délka rozvrhu).

Některé parametry:

- **release time** r_j - čas, kdy úlohu lze nejdřív rozvrhnout
- **process time** p_j - doba zpracování
- **due date** d_j - termín, kdy by úloha měla být dokončena
- **deadline** $\widetilde{d_j}$ - termín, kdy úloha musí být dokončena
- **start čas** s_j, **čas dokončení** c_j
- **čas čekání** $w_j = s_j - r_j$
- **lateness** L_j - rozdíl doby dokončení od due datu $c_j - d_j$
- **tardiness** D_j - zpoždění = doba dokončení po due datu $\max\{c_j - d_j, 0\}$



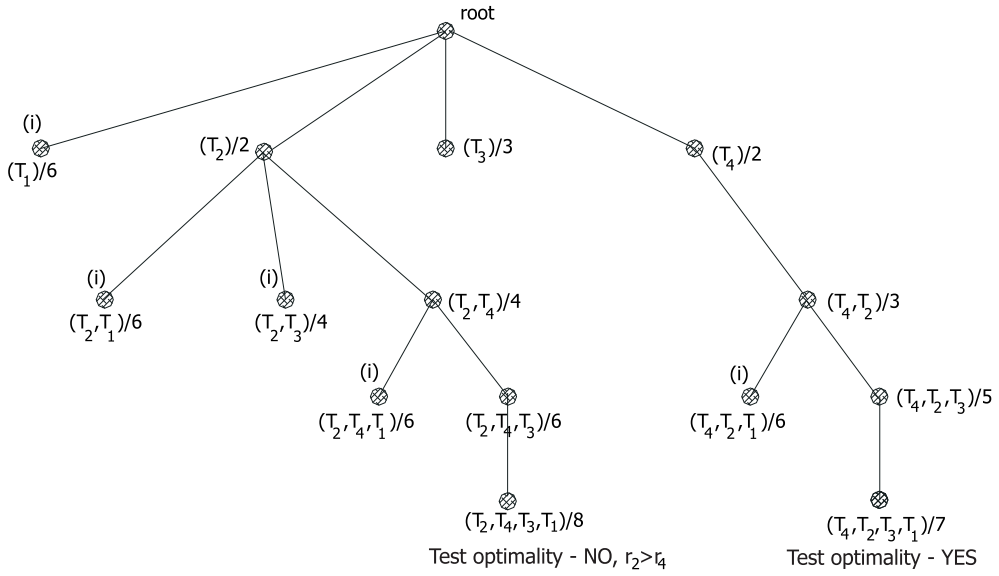
Notace rozvrhovacích problémů: $\alpha|\beta|\gamma$ (příklad: $1|r_j, d_j|C_{max}$)

- α - popisuje zdroje, jejich počet a typ (paralelní,uniformní, unrelated, job shop, atd)
- β - popisuje omezení jobů (precedence, preempce, due dates, deadlines, process times)
- γ - popisuje kritérium rozvrhu - délka rozvrhu (C_{max}), lateness, tardiness

10.2 Rozvrhování na jednom procesoru

Mnoho „easy“ problémů ($1|prec|C_{max}$, $1||C_{max}$, $1|r_j|C_{max}$, $1|\widetilde{d_j}|C_{max}$) - jen seřadíme úlohy např. podle $\widetilde{d_j}$. Kombinace $1|r_j, \widetilde{d_j}|C_{max}$ už není tak jednoduchá (NP-hard).

Bratleyův B&B alg Řeší $1|r_j, \widetilde{d_j}|C_{max}$ které je **NP-hard**. Klasický branch&bound, když překročíme deadline, můžeme odříznout i bratry. Když narazíme na řešení, zkusíme test optimality - má první úloha nejmenší release time? Jestli jo, break jinak jedeme dál.



Obrázek 12: Bratleyův alg pro: $r = [4, 1, 1, 0], p = [2, 1, 2, 2], \widetilde{d} = [8, 5, 6, 4]$

$1|prec|\sum wC$ se řeší branch&bound s LP. Definujeme problém jako LP (rozhodovací proměnná $x_{ij} = 1$ iff úloha i je předchůdce j), což nám dá hodnotu zbylých úloh. Ořezáváme, pokud řešení + zbylé úlohy > dosud nejlepší řešení.

10.3 Rozvrhování na paralelních procesorech

McNaughton $O(n)$ Řeší $P|pmtn|C_{max}$. Spočítám si C_{max} jako maximum ze součtu časů/počet procesorů a času nejdelšího úkolu:

$$C^*_{max} = \max\{\frac{1}{R}\sum_{i=1}^{n} p_i, \max_{i=1,\cdots,n} p_i\}$$

Nyní, když je již známá délka rozvrhu, může algoritmus přistoupit k samotnému plánování. Algoritmus iteruje postupně přes všechny úkoly a vyplňuje matici od levého horního rohu po řádcích. První úkol umístí do levého horního rohu, druhý úkol těsně za něj a tak dále. V okamžiku, kdy se některý z úkolů již nevejde celý na jeden řádek (zdroj), tak jej algoritmus přeruší (nechá na k-tém zdroji vykonat pouze koncovou část úkolu), a jeho první část rovrhne na zdroj následující.

- $P2||C_{max}$ NP-hard - lze převést z 2-partition problému.
- $P|pmtn, r, d|C_{max}$ jde formulovat jako úlohu maximálního toku: Udělám si intervaly pro všechny release timy a deadliny. Každý interval bude jeden vrchol. Každá úloha bude vrchol. Od zdroje k úlohám budou mít hrany hodnotu doby trvání. Od úloh do intervalových vrcholů budou mít hrany kapacitu velikosti intervalu. Od vrcholu do zdroje je kapacita velikost intervalu * počet procesorů.

List Scheduling (LS) $P|prec|C_{max}$ Aproximační alg. Nejdřív dám do listu úlohy bez předchůdců. Postupně přiřazuju z listu na procesory a když to někde doběhne, dám tam další a na konec listu dám následovníky té co doběhnula. Aproximační faktor $r_{LS} = 2 - \frac{1}{R}$ (R je počet zdrojů).

Longest processing time first (LPT) $P||C_{max}$ Vylepšení pro LS v podobě vhodného řazení úloh. Funguje stejně jako LS ale řadím list podle processing timu. Aprox faktor $r_{LPT} = \frac{4}{3} - \frac{1}{3R}$. Tento problém se dá řešit ještě pomocí pseudopolynomiálního dynamického programování Rothkopf (R-rozměrné pole).

Rothkopf řeší dynamickým programováním pseudopolynomiálně P||Cmax. Mám tabulky časů pro každý procesor, počet rozměrů stejně jako procesorů a zapisuju tam všechny možnosti. Na konci minimalizuju rozdíl časů na jednotlivých procesorech.

Úrovňový algoritmus $P|pmtn, prec|C_{max}$ řeší $P|pmtn, prec|C_{max}$ v (n^2). Nejdřív ohodnotím graf následností úrovněmi.

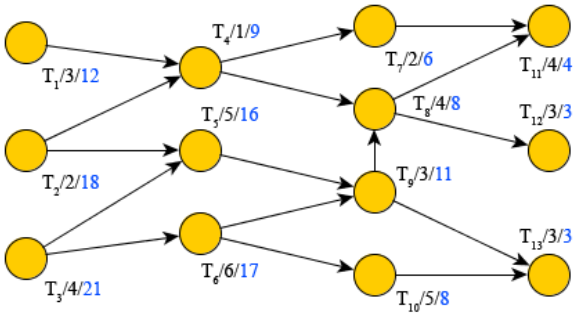
Algoritmus nejprve zkonstruuje graf závislostí, v němž jsou úkoly vyjádřené pomocí uzlů a relace následností pomocí orientovaných hran (hrana vede vždy z předka do následníka). Dále algoritmus ohodnotí všechny uzly dle následujícího schématu (S(x) značí všechny následníky uzlu x):

$$level(j) = \max_{s \in S(j)} \{level(s)\} + p_J$$

Úroveň se počítá jako délka zpracování + maximální úroveň z následníků. Proto to je vhodné počítat zprava. Samotné rozvrhování probíhá v časových kvantech. Na každý zdroj je umístěn úkol s nejvyšší úrovní. Po obsazení všech zdrojů je spuštěno zpracování úkolů na zdrojích na časové kvantum T, které odpovídá času do zpracování nejkratšího z úkolů. Po vykonání tohoto kvanta je zpracovaný úkol odstraněn a u částečně zpracovaných úkolů je snížena jejich úroveň o T (úroveň se skládá také z délky úkolu a ta je nyní kratší). Na zdroje jsou znovu umístěny úkoly s nejvyšší úrovní (jejichž všichni předci již byli zpracováni), což ale nemusí být nutně ty, které byly částečně zpracovány v minulé iteraci. Algoritmus terminuje v okamžiku zpracování všech úkolů.

10.4 Rozvrhování projektu (Project scheduling)

Je scheduling s temporálními omezeními. $PS|temp|C_{max}$ - NP-obtížný problém. Temporální omezení jsou relace následností s váhami na hranách. Když váha $= p_j$ předchozí, můžu začít když ta předchozí skončí. Když váha $> p_j$, musím čekat. Když $0 <$ váha $< p_j$, musím začít během vykonávání na jiném procesoru. Řeší se pomocí ILP - binární nebo celočíselná formulace.



Čas	Zdroj 1	Zdroj 2	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}
0	T_3	T_2	**3/12**	**2/18**	**4/21**	1/9	5/16	6/17	2/6	4/8	3/11	5/8	4/4	3/3	3/3
2	T_3	T_1	**3/12**		**2/19**	1/9	5/16	6/17	2/6	4/8	3/11	5/8	4/4	3/3	3/3
4	T_6	T_5	**1/10**			1/9	**5/16**	**6/17**	2/6	4/8	3/11	5/8	4/4	3/3	3/3
9	T_6	T_1	**1/10**			1/9		**1/12**	2/6	4/8	3/11	5/8	4/4	3/3	3/3
10	T_9	T_4				**1/9**			2/6	4/8	**3/11**	**5/8**	4/4	3/3	3/3
11	T_9	T_{10}							**2/6**	4/8	**2/10**	**5/8**	4/4	3/3	3/3
13	T_8	T_7, T_{10}							**2/6**	**4/8**		**3/6**	4/4	3/3	3/3
17	T_{10}	T_{11}										**1/4**	**4/4**	**3/3**	3/3
18	T_{11}, T_{12}, T_{13}												**3/3**	**3/3**	**3/3**
22,5															

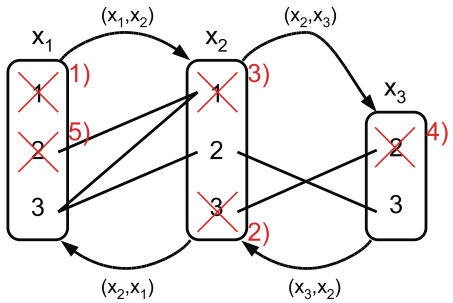
Obrázek 13: Úrovňový algoritmus

10.5 Programování s omezujícími podmínkami (Constraint Programming)

Je podobný jako ILP, ale můžeme definovat víc druhů podmínek (ILP jenom nerovnice, CSP libovolné relace). Kromě **omezení** se definují domény **pro** každou **proměnnou** - např. **sudoku** 1-9.

Postup: nejdřív udělám úvodní propagaci - aplikuji podmínky na obor hodnot. Potom jedu b&b, postupně zkouším jednotlivé data a propaguji podmínky. Až se dostanu k nějakému řešení. Hranová konzistence znamená, že hrana s daným řešením splňuje všechna omezení.

Jeden z algoritmů pro řešení je AC-3. Při **AC3** si udržujeme **frontu hran**, které je potřeba **revidovat**. Na začátku jsou tam všechny hrany a postupně je odebíráme. Pozor, revizí nějaké hrany se může znevalidnit už validovaná hrana. Algoritmus běží, dokud nejsou všechny hrany konzistentní. Vždy reviduju přechody nejdříve v jednom směru a pak v druhém. pokud se mi změní doména (odeberu číslo), tka musím opět dát do fronty tu hranu, které se doména týká.



Obrázek 14: AC-3: $x_1 > x_2, x_2 \neq x_3, x_2 + x_3 > 4; D_1 = \{1,2,3\}, D_2 = \{1,2,3\}, D_3 = \{2,3\}$

Reference

[1] Ing. Jakub Černý, PhD. Amortizovaná časová složitost. Dostupné z: http://algoritmy.eu/zga/casova-slozitost/amortizovana/.

[2] Ing. Pavel Mička. Amortizovaná složitost, . Dostupné z: http://www.algoritmy.net/article/3024/Amortizovana-slozitost.

[3] Ing. Pavel Mička. Třídy složitosti a Turingovy stroje, . Dostupné z: http://www.algoritmy.net/article/5774/Tridy-slozitosti.

[4] Ing. Přemysl Šůcha, Ph.D. *Celočíselné lineární programování* [online]. 2004. Dostupné z: http://support.dce.felk.cvut.cz/pub/hanzalek/_private/ref/sucha_ilp.pdf.

[5] Jiří Vyskočil, Radek, Mařík, Marko Berezovský. *Slidy k přednáškám PAL* [online]. 2013. Dostupné z: https://cw.felk.cvut.cz/wiki/courses/a4m33pal/prednasky.

[6] Prof. RNDr. Marie Demlová, CSc. *Slidy k přednáškám TAL* [online]. 2014. Dostupné z: http://math.feld.cvut.cz/demlova/teaching/tal/predn_tal.html.

[7] Přispěvovatelé wikipedie. Asymptotická složitost, . Dostupné z: http://cs.wikipedia.org/wiki/Asymptotickásložitost.

[8] Přispěvovatelé wikipedie. Fibonacciho halda, . Dostupné z: http://cs.wikipedia.org/wiki/Fibonacciho_halda.

[9] Přispěvovatelé wikipedie. Párování grafu, . Dostupné z: http://cs.wikipedia.org/wiki/Párování_grafu.