Grafy a hry

Teorie grafů a grafové hry

Sedláček Pavel

2024-11-04

EDUcanet

—

Table of contents

— Table of contents

Table of contents

— Table of contents

Úvod

Úvod

—

Hodnocení

Podle dokumentu v `Google classroom`.

Projití je podmíněno:

- Odevzdáváním malých dílčích úkolů v průběhu roku.
  - Postupně si naimplementujeme klíčové vlastnosti
  - Úkoly budou hodnoceny 10% nebo 20% podle náročnosti
- Odevzdáním pololetní větší práce
  - Bude se jednat o úlohu složenou z těch malých
  - Úloha bude hodnocena 100%

Další známky bude možné získat aktivitou na hodině.

Úvod
— Hodnocení

Výsledná známka

tabulka

Úvod
— Výsledná známka

Jazyky, úkoly

Jazyky, úkoly

—

Úkoly můžete odevzdávat v *normálním* jazyku dle volby[1]. Uvažujte v potaz, že některé jazyky svojí povahou udělají úkoly jinak náročné, než jak byly uvažovány. Pokud si nejste jistí, tak se radši zeptejte.

Na hodinách budeme používat Javu. Úryvky kódů budou psané i v pseudokódu.

K vypracování úloh neni povoleno používat cizí knihovny ani frameworky. Cokoliv si napíšete sami samozřejmě používat můžete, pokud není řečeno jinak.

[1] tím se chápou běžně používané jazyky (C, JS, C#, Python, …). Pokud jazyk nemá first-class podporu od JetBrains, tak vám ho asi neuznám.

Úkoly

Úkoly se budou věnovat tématům probíraným v hodinách. Zpravidla se bude jednat o implementaci více teoretické látky nebo rozšíření nějaké předneseného algoritmu.

Úkoly se budou odevzdávat do `Google Classroom` přes verzovací systém git a to prostřednictvím `GitHub`[1] nebo `GitLab`. Úkoly odevzdané prostřednictvím `Google Drive` nebo v archivech nebudou přijaty.

Na vypracování úkolů bude vždy *dost času* a úkoly *nebudou* časově *náročné*.

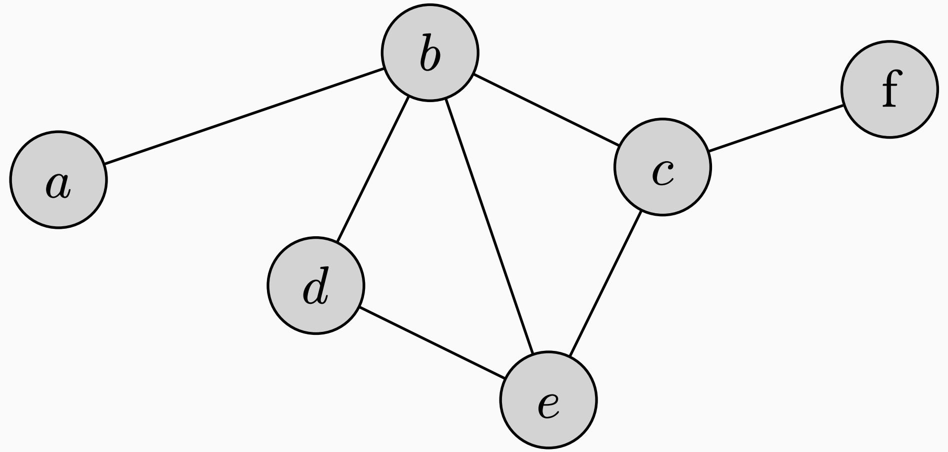
[1] https://github.com/

Jazyky, úkoly
— Úkoly

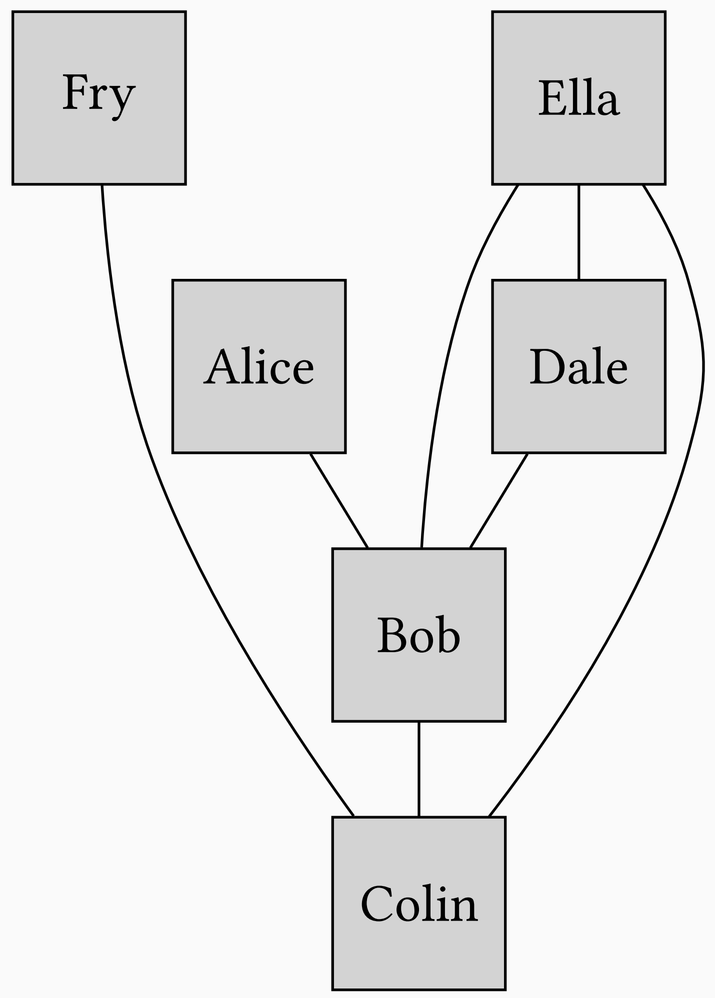
Grafy

Co to je graf?

Vrcholy / *body* / *hodnoty* mezi kterými existují *hrany* / *čáry* / *vztahy*.



Grafy
— Grafy



Můžeme se ptát:

- Jaká největší skupina se může sejít, tak aby se všichni znali?
- Kdo je pojící prvek všech lidí?
- Zná každý někoho, kdo zná Ellu?
- Kdo má nejvíce přátel?

Řídké
Grafy, které mají významně malý počet hran vůči počtu vrcholů.

Husté
Grafy, které mají významně velký počet hran vůči počtu vrcholů.

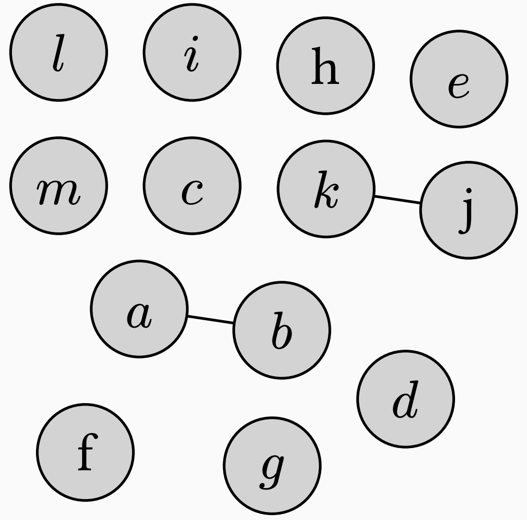
Řídké grafy umožňují levnější znázornění v paměti, můžeme využívat hloupější algoritmy, atd ...
Husté grafy většinou mají optimalizovanější algoritmy, skoro konstantní vyhledávací časy, atd ...

Řídké grafy jsou takové grafy, které mají relativně nízký počet hran vůči vrcholům. Obecně neexistuje konkrétní hranice mezi normálním a řídkým grafem. Extrémním případem jsou grafy bez hran (množina všech hran je prázdná).

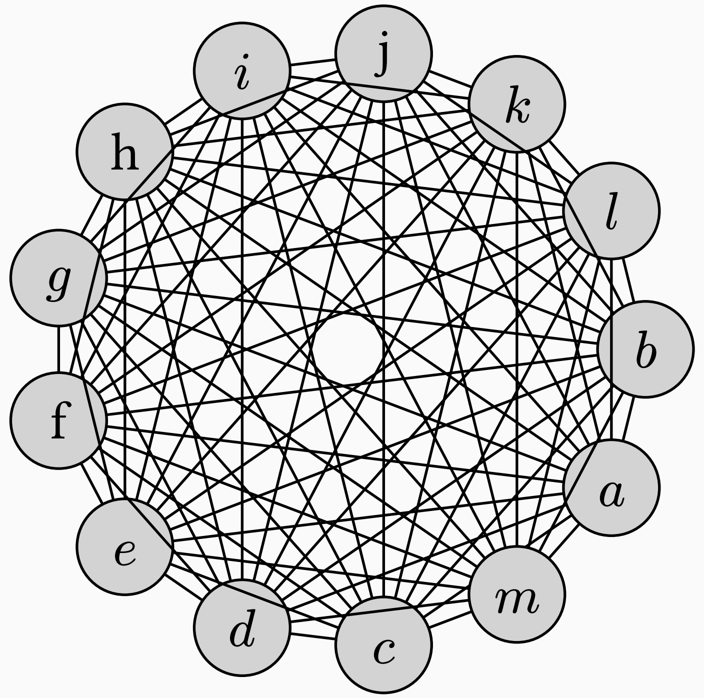
Husté grafy jsou protikladem grafů řídkých. Vyznačují se tím, že existuje hrana mezi *většinou* vrcholů. Extrémním případem jsou plné grafy, kde existuje cesta mezi *každými* dvěma vrcholy. Množina hran pak obsahuje $n(n-1)$ prvků.

Řídké a husté grafy

Řídký



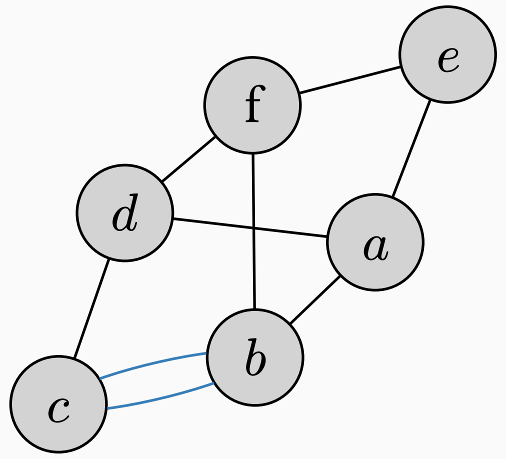
Hustý



Grafy

— Řídké a husté grafy

Vážené grafy každé hraně přiřadí váhu (hodnotu). Hodnota většinou znázorňuje cenu cesty nebo hodnotu znázorněné relace.

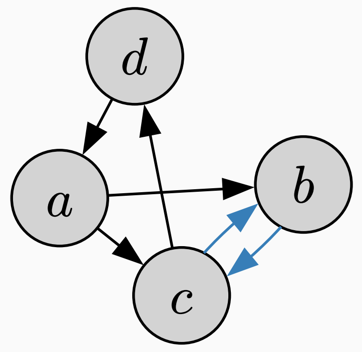


Vážené grafy jsou jednou z nejpoužívanějších variací grafů, protože umožňují řešit praktické problémy (hledání *nejkratších* cest, optimalizace výkonu operací, ...)

Orientované (směrové) grafy - directed graph

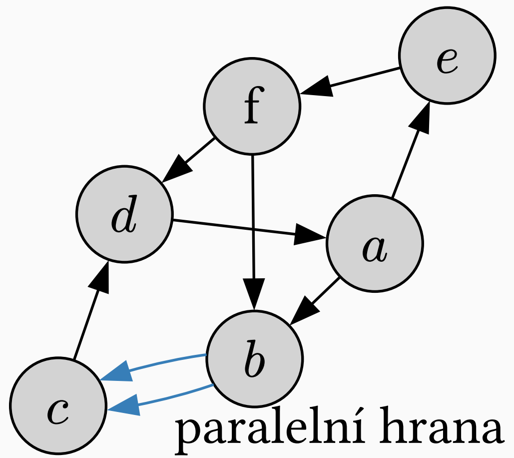
Směrové grafy navíc každé hraně přiřadí směr. Hranu pak lze využít pouze pro *přesun* po směru hrany.

- Orientované grafy značíme šipkou na konci čar.
- Můžeme kombinovat s váženým i *obousměrným*
  - Některé kombinace jsou velmi nepraktické
  - Některé naopak využívané skoro všude



Všechny typy grafů můžeme libovolně kombinovat, ale výrazně se nám tím limitují algoritmy, které na takovém grafu budou schopné operovat.

Hrana je paralelní, pokud existuje alespoň další jedna hrana ze stejného vrcholu do stejného cílového vrcholu.



Nyní můžeme smysluplně zadefinovat paralelní hrany, tedy hrany, které jsou ze stejného počátečního vrcholu do stejného koncového vrcholu.

Každá taková hrana může mít vlastní váhu.

Stupeň vrcholu přímo odpovídá počtu hran k/od vrcholu.

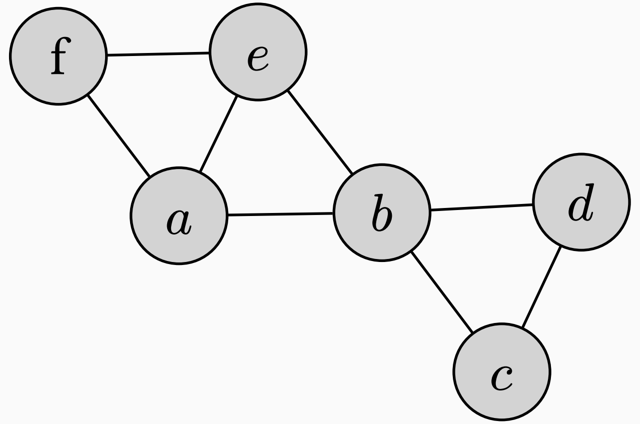
Poznámka

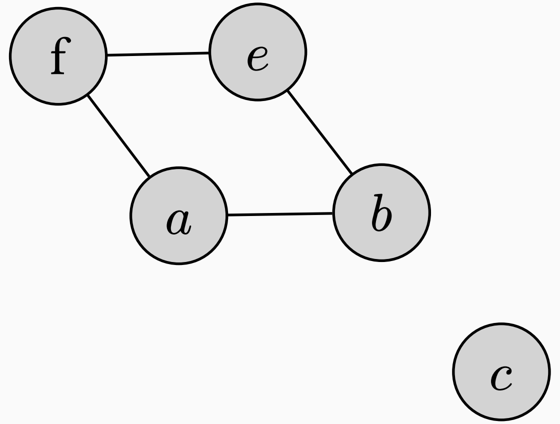
Stupeň vrcholu vypovídá o *důležitosti* vrcholu v grafu.

Výstupní stupeň
Hodnotu omezuje pouze na hrany, které směřují ***z*** tohoto vrcholu.

Vstupní stupeň
Hodnotu omezuje pouze na hrany, které směřují ***do*** tohoto vrcholu.

Podgraf grafu je graf, který neobsahuje žádné vrcholy a hrany, které nejsou v původním grafu.





Grafy
— Podgrafy - subgraph

Kreslení grafů

Graf je abstraktní struktura. Pro každý graf existuje nekonečně mnoho *nakreslení* - způsobů jak graf graficky znázornit. Většinou je cílem graf nakreslit tak, aby měl co ***nejméně*** (ideálně nula) **překřížení** hran.

Topologicky rozvrhnout kreslení grafů je *relativně* obtížná úloha a nebudeme se jí zde zabývat. Grafy budeme případně kreslit ručně, kde to lze vyřešit intuitivně.

- https://github.com/dagrejs/dagre
- https://eclipse.dev/elk/reference/algorithms.html
- https://github.com/terrastruct/d2
- https://gitlab.com/graphviz/graphviz/-/tree/main/lib/dotgen

Mřížky jako grafy - Grid

Problém, který se snažíme vyřešit:

- Známe (*nebo alespoň znát budeme*) celou řadu algoritmů, které umí pracovat na grafech, ale my bychom potřebovali, aby je bylo možné spustit na mřížce
- Přirozeně se s mřížkou setkáme častěji než s grafem

Řešení, která se nabízí:

- Umíme upravit algoritmy na to, aby fungovaly i na mřížce
- Umíme nakreslit mřížku jako graf

Representace grafů v paměti

Representace grafů v paměti

—

Základní třída

Budeme uvažovat třídu

```java
class Label {
  String name;

  public Label(String name) {
    this.name = name;
  }
}
```

s přepsanou metodou `equals` na porovnání atributu `name`.

přepsaná funkce `equals` a `hash`

```java
@Override
public boolean equals(Object other) {
  return other instanceof Label &&
           ((Label) other).name = this.name;
}
@Override
public int hashCode() {
  return this.name.hashCode();
}
```

Seznamem hran

Java

```java
class Edge {
    // Pro orientované grafy je lepší `from` a `to`
    // Případně můžeme využít nějaký enum `Orientation`
    Label a;
    Label b;

    // Pokud chceme vážený graf
    float weight;
}
```

Seznamem hran

Java

```java
Label a = new Label("a");
   ...
Label d = new Label("f");

Label[] vertices = { a, b, c, d};
Edge[] edges = { new Edge(a, b), new Edge(b, c),
                 new Edge(c, d), new Edge(b, d) };
```


Representace grafů v paměti
— Seznamem hran

Spojovým seznamem

Potřebujeme dodefinovat novou třídu:

Java

```java
class Vertex {
  Label label;
  Label[] connectedLabels;
}
```

- Mohli bychom využít `Pair<Label, Label[]>`, ale kvůli čitelnosti a vyhnutí se zbytečné generice je lepší takto explicitní třída.
- Místo pole můžeme využít `List`; případně `Set`, pokud víme, že graf nepovoluje paralelní hrany.

Representace grafů v paměti
— Spojovým seznamem

Java

```java
Label a = new Label("a");
 ...
Label d = new Label("f");

List<Vertex> graph = new ArrayList<>();

graph.add(new Vertex(a, { b } ));
graph.add(new Vertex(b, { c, d } ));
graph.add(new Vertex(c, { d } ));
```


Representace grafů v paměti

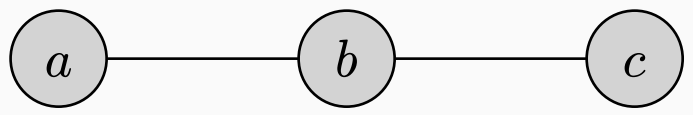
— Spojovým seznamem - Implementace

Maticí

```java
float[][] graph = { { 0, 1, 0 },
                    { 1, 0, 1 },
                    { 0, 1, 0 } };
```

Java

- Místo `float` můžeme využít libovolný typ, který znázorňuje váhu hrany, nebo jednoduchý `boolean`.
- Musíme si sami nastavit pravidla, co která hodnota značí.
- Dejme tomu, že předchozí příklad znázorňuje:



Representace grafů v paměti
— Maticí

Několik důležitých poznámek:

- Pro neorientované grafy je matice symetrická podle diagonály
- Pro grafy bez smyček je diagonála nulová
- Pro řídké grafy dostaneme řídkou matici
  - Matici s hodně nulami

Algoritmická složitost

Algoritmická složitost

—

Složitost je metrika, která určuje kolik zdrojů algoritmus zabere na vykonání pro určitou velikost vstupu.

Poznámka

Většinou nás zajímá *čas* a *paměť*.

Časová náročnost vymezuje čas, který algoritmus A potřebuje ke zpracování vstupu V o velikosti n. Náročnost se vyjadřuje v počtu operací procesoru vůči velikosti vstupu.

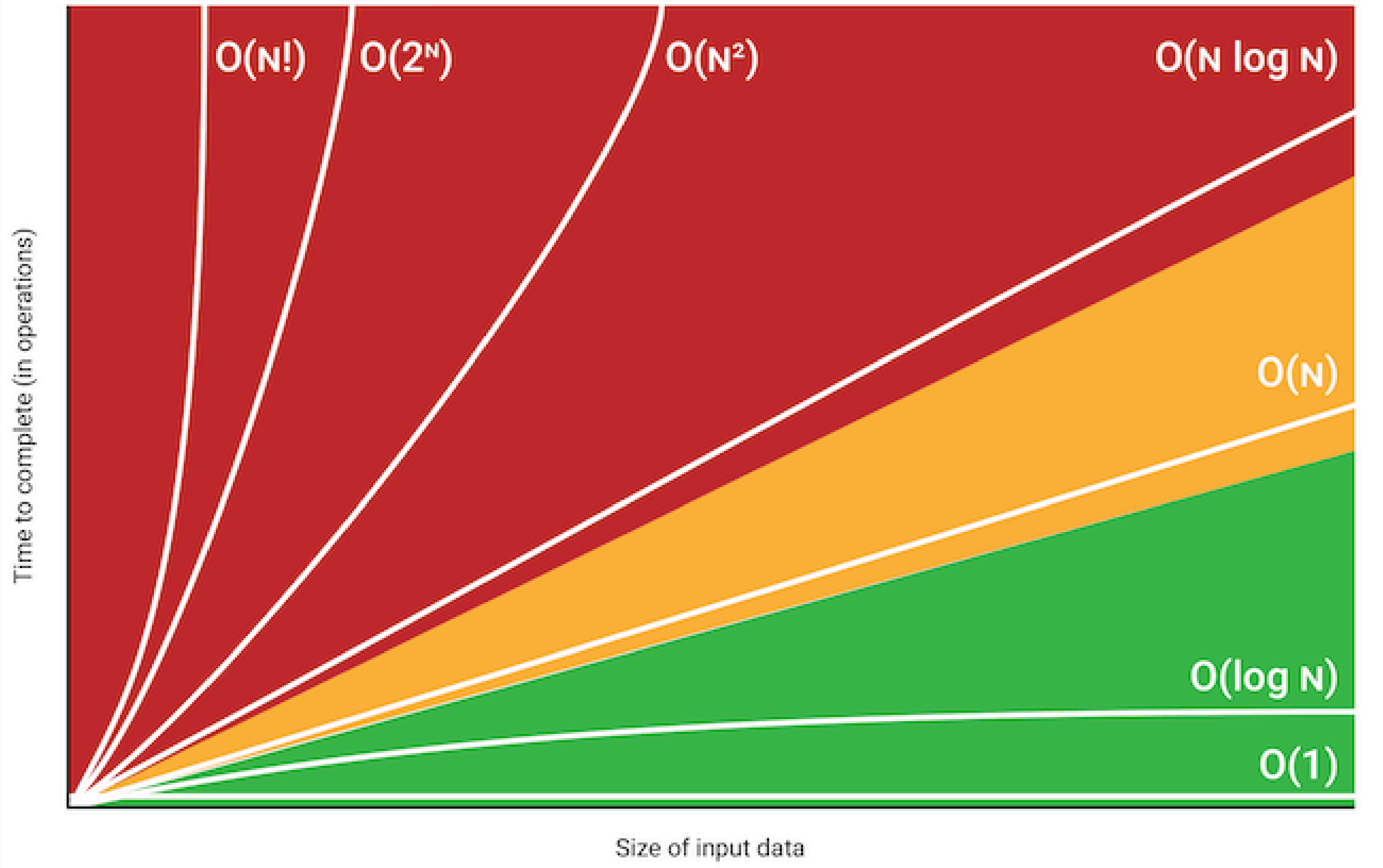
Třídy složitosti

- 1
  - Konstatní čas
  - Sečtení prvních dvou prvků
- $\log n$
  - Logaritmnický čas
  - Kdykoliv něco dělíme na půlky
- n
  - Lineární čas
  - Procházení pole
- $n \log n$
  - Linearitmický čas
  - Vkládání do haldy, většina třídících algoritmů
- n^2
  - Kvadratický čas
  - Procházení pole v poli

Algoritmická složitost
— Třídy složitosti

- 2^n
  - Exponenciální čas
  - Rekursivní dělící algoritmy
- $n!$
  - Faktoriálový čas
  - Permutace, Variace, ...

Chování složitostí



Asymptotické chování algoritmů

Asymptotické chování = *chování v nekonečnu*

Cesty v grafech

Sled - Walk

Sled je posloupnost vrcholů a hran.

Tah - Trail

Tah je posloupnost vrcholů a hran, ve které je každá hrana maximálně jednou.

Cesta - Path

Cesta je posloupnost vrcholů a hran, ve které je každý vrchol maximálně jednou

Cesty v grafech
— Sled, Tah a Cesta

Sled, tah nebo cesta jsou uzavřené, pokud začínají a končí ve stejném vrcholu.

Kružnice / cyklus - Cycle

Kružnice je uzavřený tah, kde pouze první a poslední vrcholy jsou stejné. (Nejsou žádná jiná *překřížení*).

Orientovaná kružnice

Kružnice ve směrovém grafu v alespoň jednom *směru*.

Komponenty (souvislosti) grafu jsou největší podgrafy, pro něž platí, že každé dva vrcholy jsou spojené cestou.

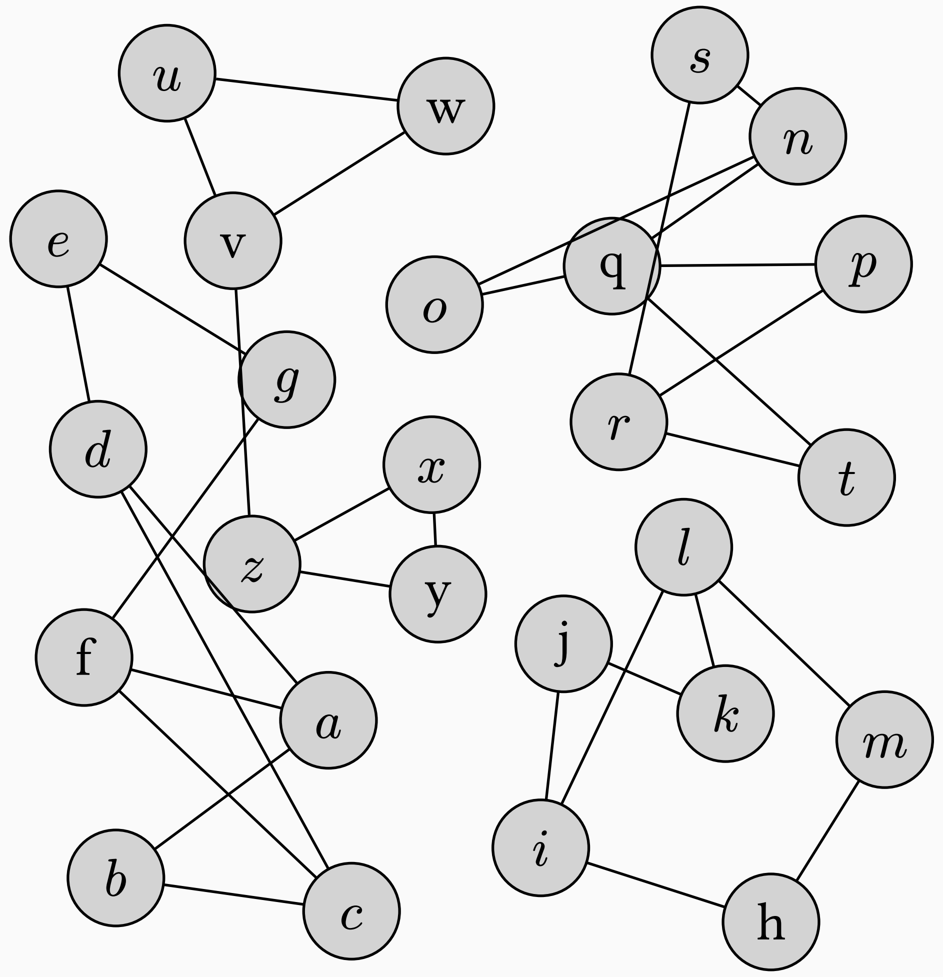
i.e.: nespojené části grafu

- Stromy mají pouze jednu (hlavní) komponentu souvislosti.
- Jiné grafy mohou mít až tolik, kolik je v nich vrcholů.

Silné komponenty - Strongly connected component

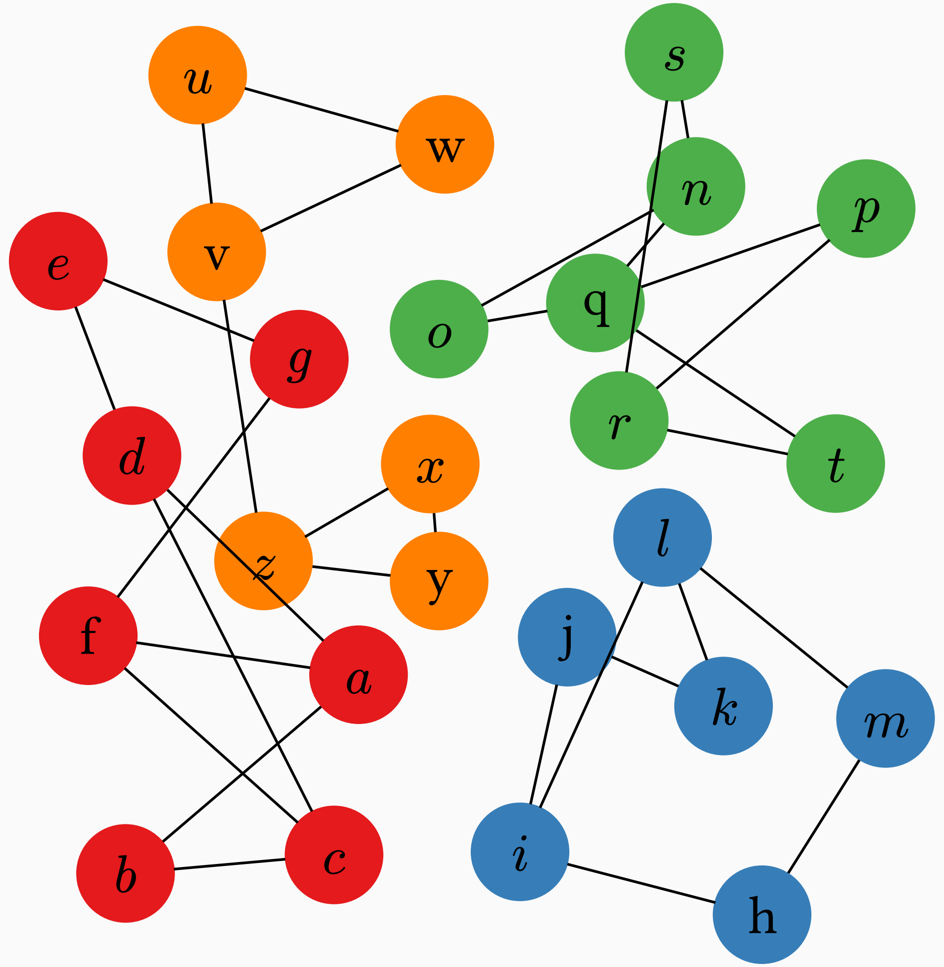
Silné komponenty jsou obdobou pro orientované grafy. Komponenta je omezená na vrcholy, mezi kterými vede orientovaná cesta.

Komponenty tohoto grafu?



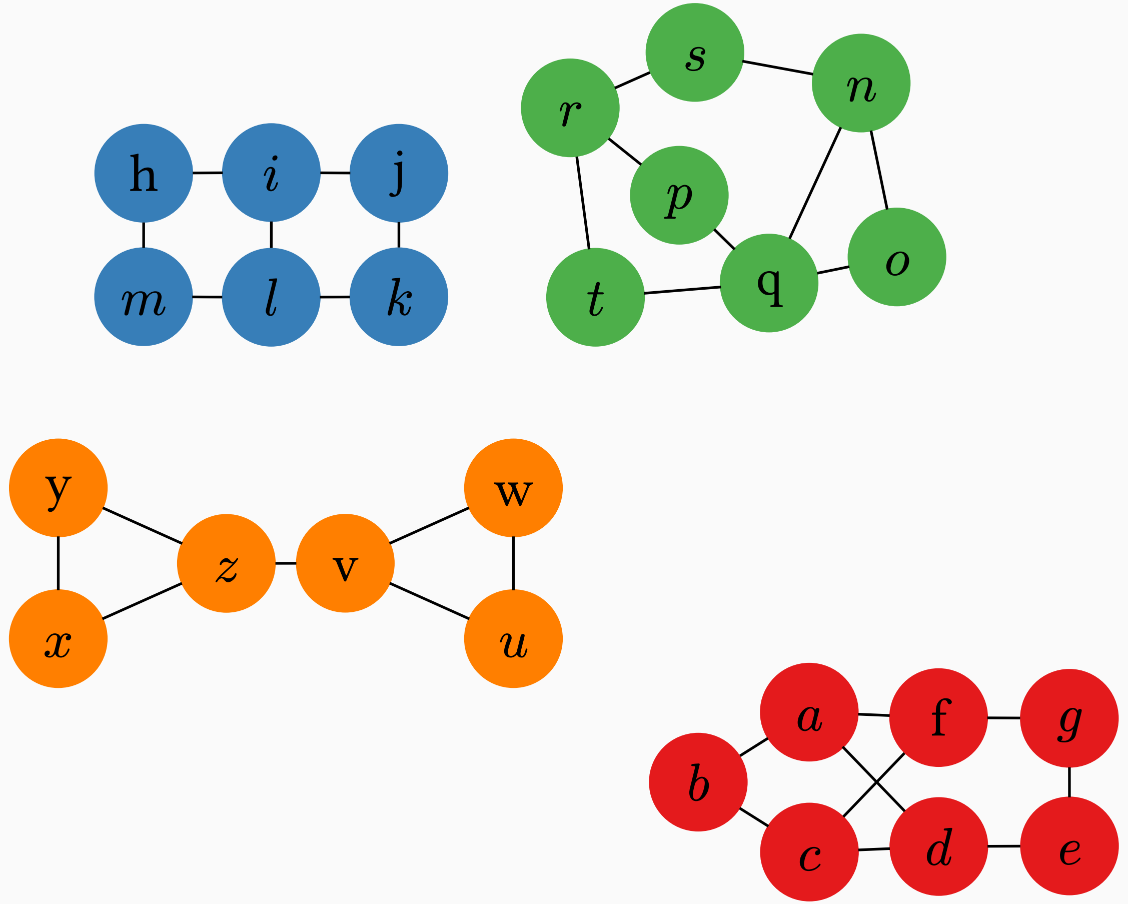
Cesty v grafech
— Komponenty tohoto grafu?

Komponenty tohoto grafu?

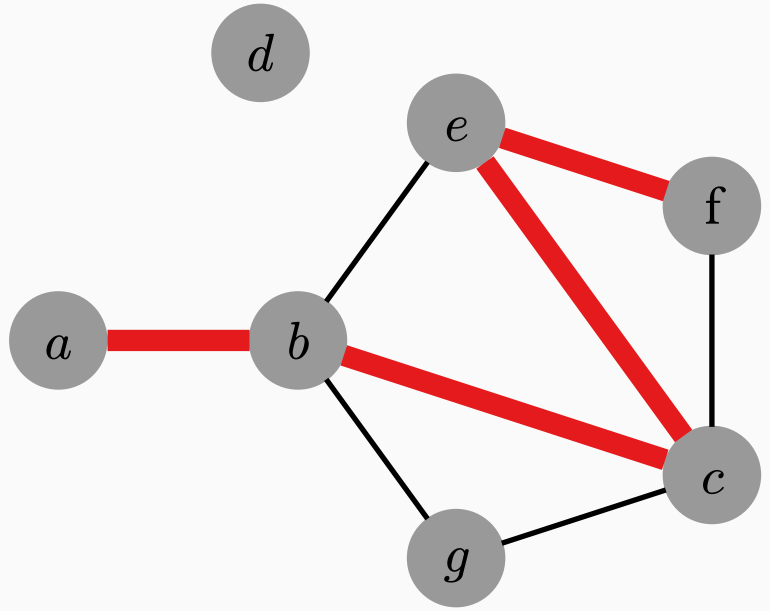


Cesty v grafech
— Komponenty tohoto grafu?

Protože je identický s:



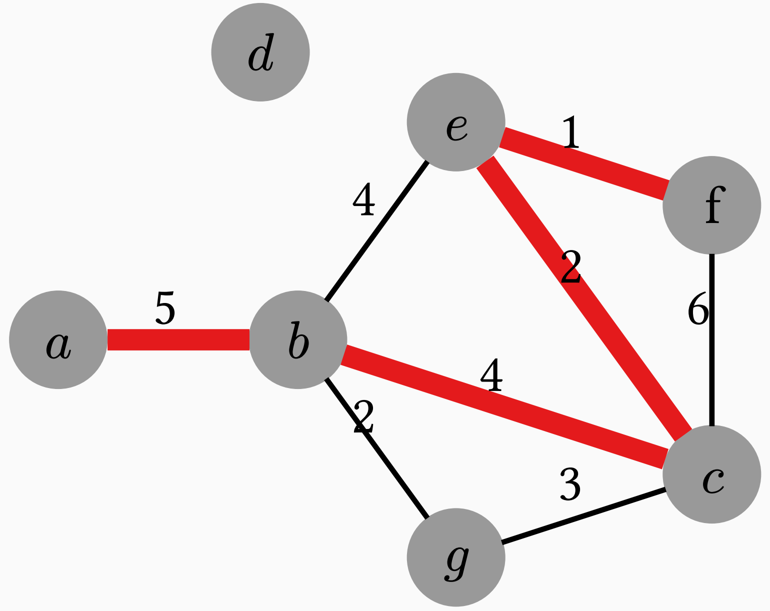
= Počet hran v tahu



Vzdálenost a od f je 4.

Vážená vzdálenost

= Součet vah hran v tahu



Vzdálenost $\boldsymbol{a}$ od $\boldsymbol{f}$ je $5 + 4 + 2 + 1 = 12$.

Cesty v grafech
— Vážená vzdálenost

Graf prohledáváme *přednostně* do hloubky.

1. ***L**IFO* fronta a vložíme počáteční vrchol
2. Vybereme vrchol z fronty
3. Vložíme do fronty všechny jeho potomky[1]
4. opakujeme 2. a 3. krok, dokud:
   1. není fronta prázdná
   2. nenajdeme cílový bod

Pokud chceme znát cestu, pak při přidávání vrcholu do fronty k němu přiložíme jeho předky.

[1] Organizovaně, ideálně topologicky, ie. nejdřív levé potomky a pak pravé

Cesty v grafech
— DFS

Graf prohledáváme *přednostně* do šířky.

1. ***F**IFO* fronta a vložíme počáteční vrchol
2. Vybereme vrchol z fronty
3. Vložíme do fronty všechny jeho potomky[1]
4. opakujeme 2. a 3. krok, dokud:
   1. není fronta prázdná
   2. nenajdeme cílový bod

Pokud chceme znát cestu, pak při přidávání vrcholu do fronty k němu přiložíme jeho předky.

[1] Organizovaně, ideálně topologicky, ie. nejdřív levé potomky a pak pravé

Graf prohledáváme chamtivě (*greedy algorithm*).

Dijkstrův algoritmus se snaží přednostně prohledávat zatím nejlevnější cesty.

1. *Prioritní* fronta a vložíme počáteční vrchol
2. Vybereme vrchol z fronty
3. Vložíme do fronty jeho potomky. Jako klíč použijeme cenu cesty do daného bodu.
4. opakujeme 2. a 3. krok, dokud:
   1. není fronta prázdná
   2. nenajdeme cílový bod

Několik poznámek k Dijkstrově algoritmu

- Algoritmus nám ohodnotí vzdálenost každého vrcholu od počátku prohledávání
- Dijkstra selže na většině grafech se zápornou váhou

A* je rozšířením Dijkstrova algoritmu, který přidává heuristické ohodnocení každému vrcholu. Tím můžeme vybrat lepší příští vrchol na prozkoumání.

1. Prioritní fronta a vložíme počáteční vrchol
2. Vybereme vrchol fronty
3. Vložíme do fronty jeho potomky. Jako klíč použijeme cenu cesty do daného vrcholu + vypočítaná heuristika pro vrchol.
4. opakujeme 2. a 3. krok, dokud:
   1. není fronta prázdná
   2. nenajdeme cílový bod

Několik poznámek k A* algoritmu

- Pokud by naše ohodnocovací funkce byla *perfektní*, pak by A* nenavštívil žádný vrchol, který není na nejkratší cestě
  - Perfektní funkce, by byla taková, která každému vrcholu přiřadí skutečnou vzdálenost od cíle.
- Pokud ohodnocovací funkce bude konstantní, pak se A* bude chovat stejně jako Dijkstra
- Čím komplexnější heuristická funkce, tím lepších výsledků algoritmus dosáhne.

- Pro grafy znázorňující reálné prostředí je vhodnou metrikou euklidovská vzdálenost. Rychlá na vypočítání v $O(1)$ a pro otevřené terény dává skoro perfektní výsledky.

Stromy

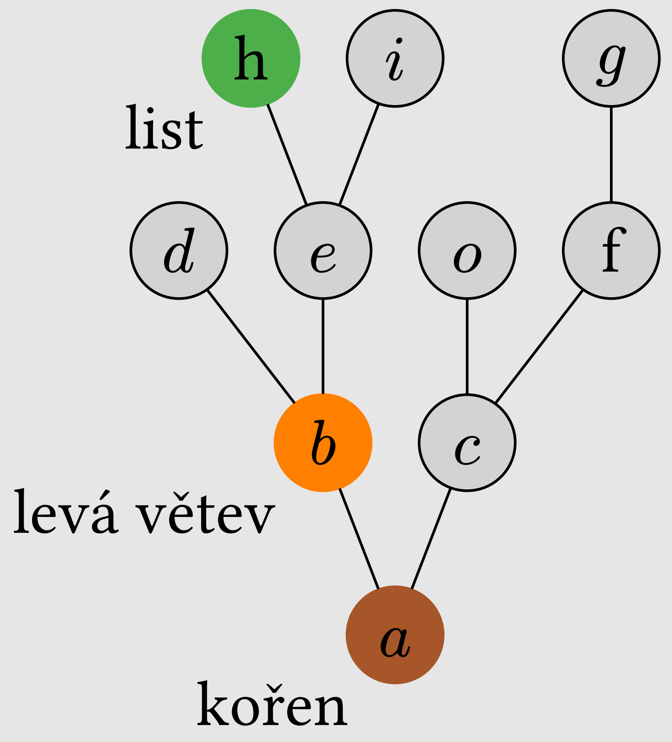
Co to je strom?

Strom je graf. Aby graf byl považován za strom, tak musí splňovat libovolnou podmínku:

- Nesmí obsahovat kružnice a musí být ***souvislý***
- Nesmí obsahovat kružnice a $|E| = |V| - 1$[1]
- ***Mezi každou dvojicí vrcholů vede právě jedna cesta***
- Přidání *libovolné* hrany vytvoří cyklus
- každý vrchol je potomkem jediného vrcholu

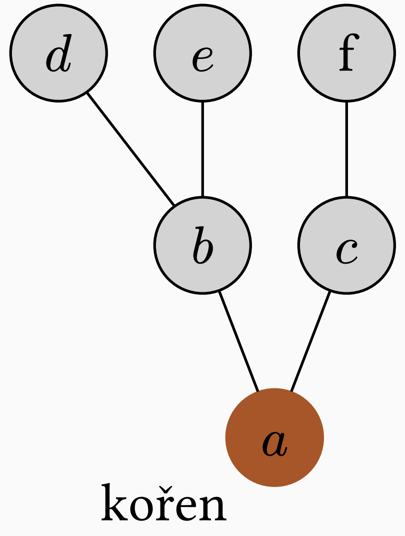
[1]počet hran je počet vrcholů bez jedné

Pojem „*strom*“ je odvozen od vzhledu grafu, pokud jeden z jeho vrcholů budeme považovat za „*kořen*“ a zbytek se vrcholů se z něj „*větví*“.



- Stromy se kreslí *naopak* (vzhůru nohama)
- Volba kořenu je čistě na nás
- Stromy jsou nejpraktičtější formou grafů
-

Stromové grafy se zpravidla *zakořeňují*. Vybereme jeden vrchol, který prohlásíme za kořen a všechny ostatní vrcholy se stanou jeho potomky a vytvoří tak jasnou hierarchii.



Stromy

— Zakořenění stromu - Rooted tree

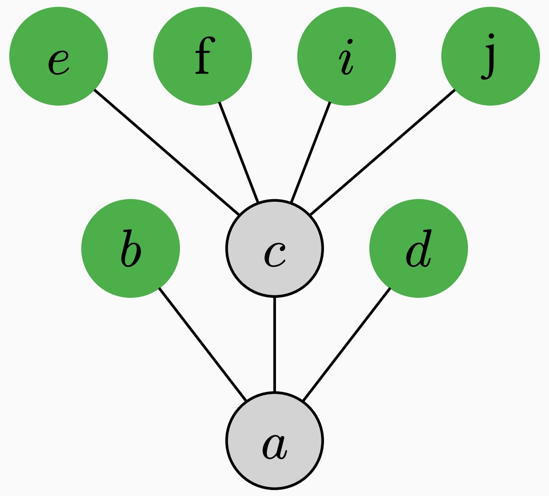
Aby se se stromy lépe pracovalo, tak definujeme kořen stromu, tedy vrchol který nemá žádné předky a všechny ostatní vrcholy jsou jeho potomky.

Navzdory názvu se kořen kreslí na vrch grafu.

Listy zakořeněného stromu - Leaves

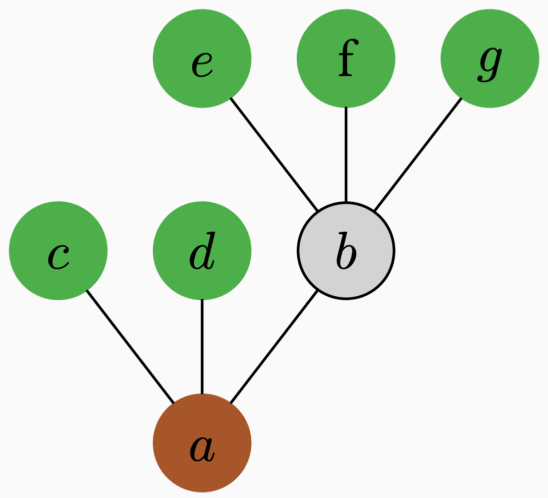
Listy jsou vrcholy, které nemají žádné potomky.

- Jsou to vrcholy na *konci* stromu
- Listy zpravidla obsahují data

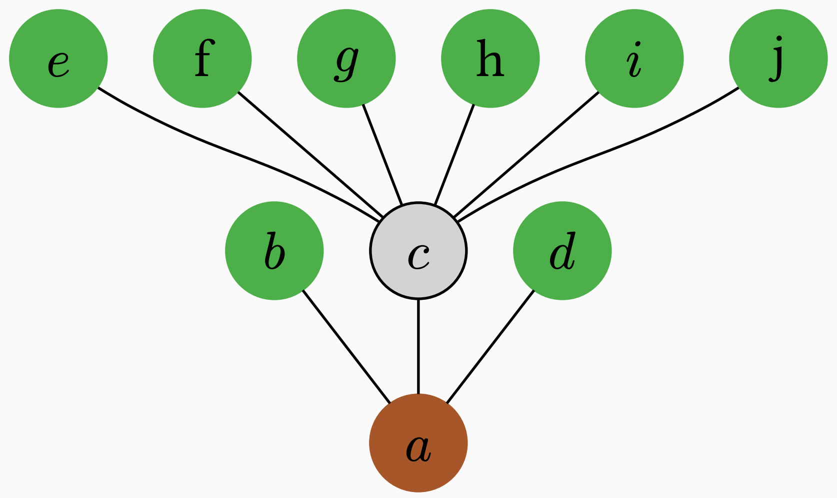


N-ární stromy omezují navíc maximální počet potomků všech vrcholů. Například ***binární*** strom může mít pro každý vrchol ≤ 2, *ternární* ≤ 3, atd...

Ternární(3) strom

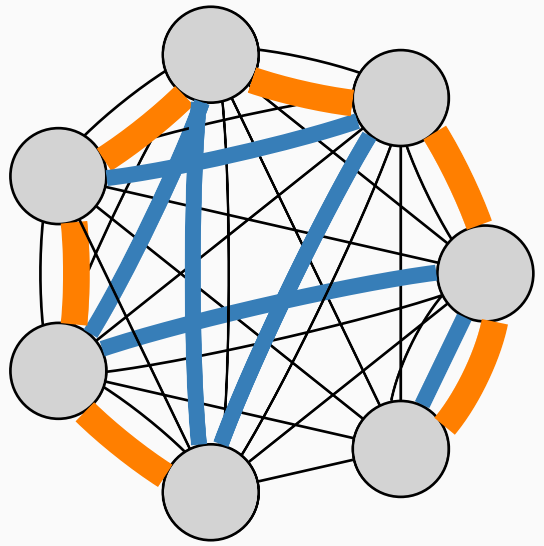


Oktární(8) strom

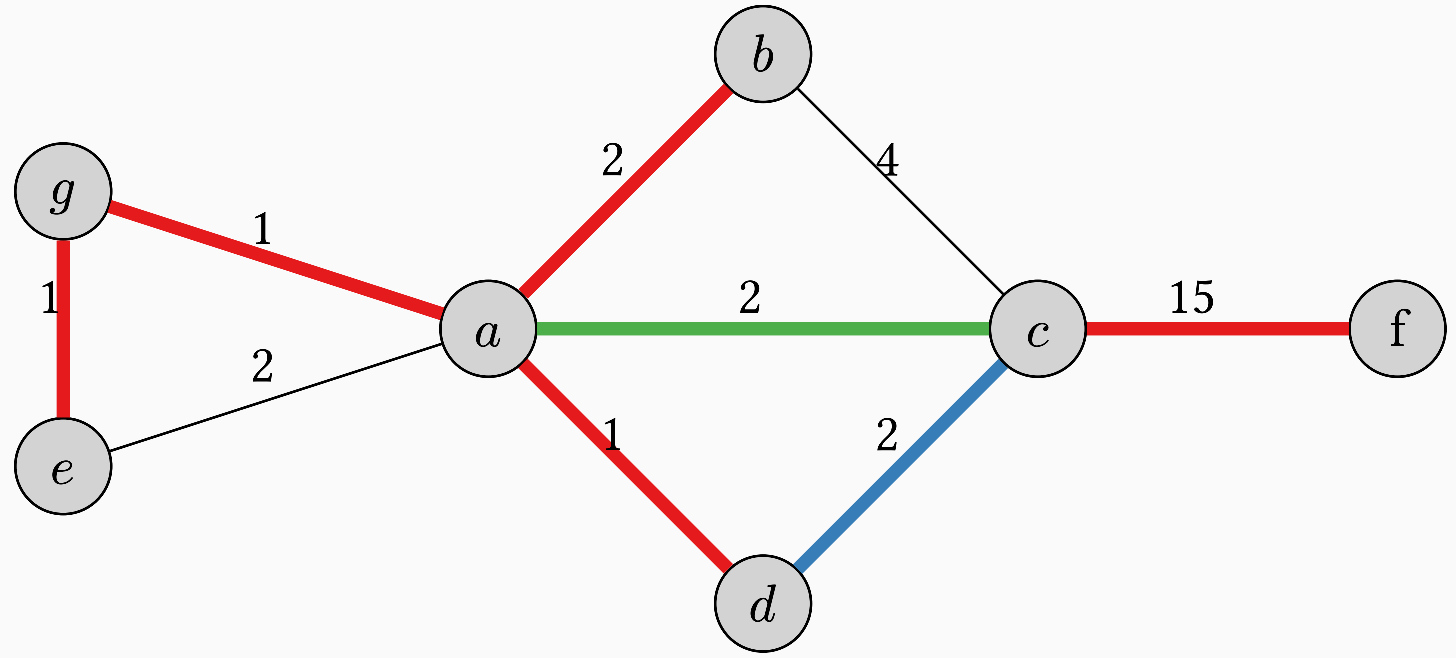


Kostra grafu je strom vytvořený z hran a vrcholů grafu tím, že se zruší kružnice. Výsledkem je graf (strom), který má *nejmenší* možný počet hran tak, aby propojil všechny vrcholy.

Koster grafu může být více.



Minimální kostra je kostra váženého grafu s nejmenší možnou kumulativní váhou.



Co to je les?

Poměrně logická odpověd: les je více stromů pohromadě.

Les jako celek můžeme považovat za graf. Pak platí poměrně pěkné rovnosti.

- # souvislých komponent = $|V| - |E|$ = # stromů v lese
  - Každý strom má o jeden vrchol více než hran
  - …
- …

Kruskal

1. Setřídit všechny hrany
2. Postupě od nejlevnější přidat každou hranu, která přidáním nevytvoří cyklus
3. Skončit při $|V| - 1$ přidaných hranách

Stromy
— Kruskal

Kruskal - pseudo

Pseudokód

```
def kruskal(graph)
  tree = Graph.with_nodes(graph.nodes)
  for edge in graph.edges.sorted()
    if not tree.with(edge).has_cycle()
      tree.add(edge)
    if tree.length() + 1 == vertices.length()
      return tree
  retrun Error("Graf nema kostru")
```


Stromy
— Kruskal - pseudo

Halda

Halda

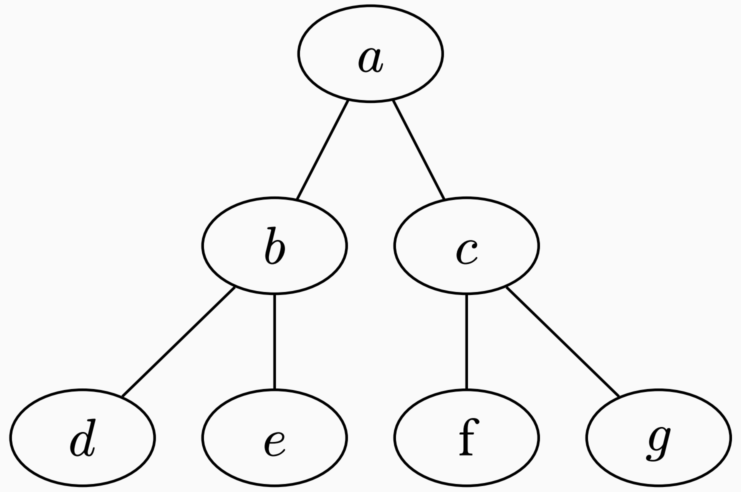
—

Halda je jednou z nejpoužívanějších struktur v informatice. Jedná se o binární strom, který navíc splňuje tyto podmínky:

- Vrcholy jsou topologicky uspořádané
  - Pořádá se podle úrovní a zleva do prava
- Je plný
  - Každý vrchol vyjma posledního má právě dva *potomky*
- Největší / nejmenší prvek je na vrcholu
  - menší hodnoty jsou na levo
  - větší na pravo

! Je třeba si dát pozor, že v informatice se pojem halda využívá pro dva zcela odlišné koncepty. Jedním je grafová halda, kterou si zde vysvětlujeme, druhá je část operační paměti určená k dynamickému alokování.

Haldu znázorníme prostým polem. Pro indexování použijeme jednoznačnost topologického uspořádání stromu.[1]



Index	0	1	2	3	4	5	6
Data	a	b	c	d	e	f	g

[1]Připomeňme si: to pro binární strom znamená číslování odshora zleva počínaje kořenem. Chybějící potomky počítáme také!

Haldy o známém počtu potomků se znázorňují velmi lehce a *levně*.

Vztah potomků a předků

Explicitní hrany/vztahy pro binární haldu můžeme dopočítat přes:

Předek	Levý potomek	Pravý potomek
$\frac{i-1}{2}$	$2i+1$	$2i+2$

Obdobné vzorce můžeme najít pro libovolně *n-ární* haldy.

```java
int parent(int child) { return (child - 1) / 2; }
int leftChild(int parent) { return 2 * parent + 1; }
int rightChild(int parent) { return 2 * parent + 2; }
```

Vkládání do haldy

Jediné *prázdné* místo v haldě, na které máme ukazatel, je její konec[1].

Algoritmus

1. Vložit prvek na konec haldy
2. *Heapifikovat* haldu podle minima/maxima
   1. Porovnat prvek s předkem
   2. Pokud je předek menší/větší, tak je vyměníme
   3. Opakujeme dokud není předek větší/menší
3. Prvek je vložený v grafu a graf je *validní* halda

[1]Teoreticky máme ještě ukazatel na kořen/počátek, ale tam již nějaký prvek je

Vkládání do haldy

MinHeap.java

```java
void insert(int element) {
  this.raw[this.tail] = element;
  int pos = this.tail;
  while (this.raw[parent(pos)] < element) {
      this.swap(parent(pos), pos);
      pos = parent(pos);
  }
  this.tail += 1;
}
```


- Ideální by bylo mít funkci `heapify` která by uměla haldu sestavit z nějaké posice směrem dolu i nahoru.

Prioritní fronta by měla umět:

- **Vkládat** prvek
  - Umíme v logaritmickém čase
- **Odebrat** první prvek
  - Umíme v logaritmickém čase
- **Změnit prioritu** prvku
  - Zatím neumíme
- Podívat se na největší prvek
- Zjistit počet prvků

Halda
— Halda jako prioritní fronta

```java
class Node<T> {
	int priorityKey;
	T my_data;
}
```

Java

nebo

```java
class Node {
  int priorityKey;
  int data_index;
}
```

Java

Logaritmický *update key*

- Naivní způsob: dostat klíč na vrch haldy, vyhodit ho a přidat nový
  - $O(1 + 2 \log n)$
- Rozumný způsob: znevalidnit starý klíč a přidat nový.
  - $O(1 + 2 \log n)$[1]
- Chytrý způsob: změnit starý klíč a vytvořit haldu *z posice*
  - $O(\log n)$

Nutné je si povšimnout, že *asymptotické* chování všech algoritmů je stejné, ale pro *rozumně* konečné n je rozdíl $\sim$ 2x

[1] Jeden $\log n$ amortizovaný v odebírání dalších prvků

Halda
— Logaritmický *update key*

- Rozumný způsob:
  1. nalezení klíče $O(1)$
  2. zinvalidování $O(1)$
  3. přidání nového klíče $O(\log n)$.

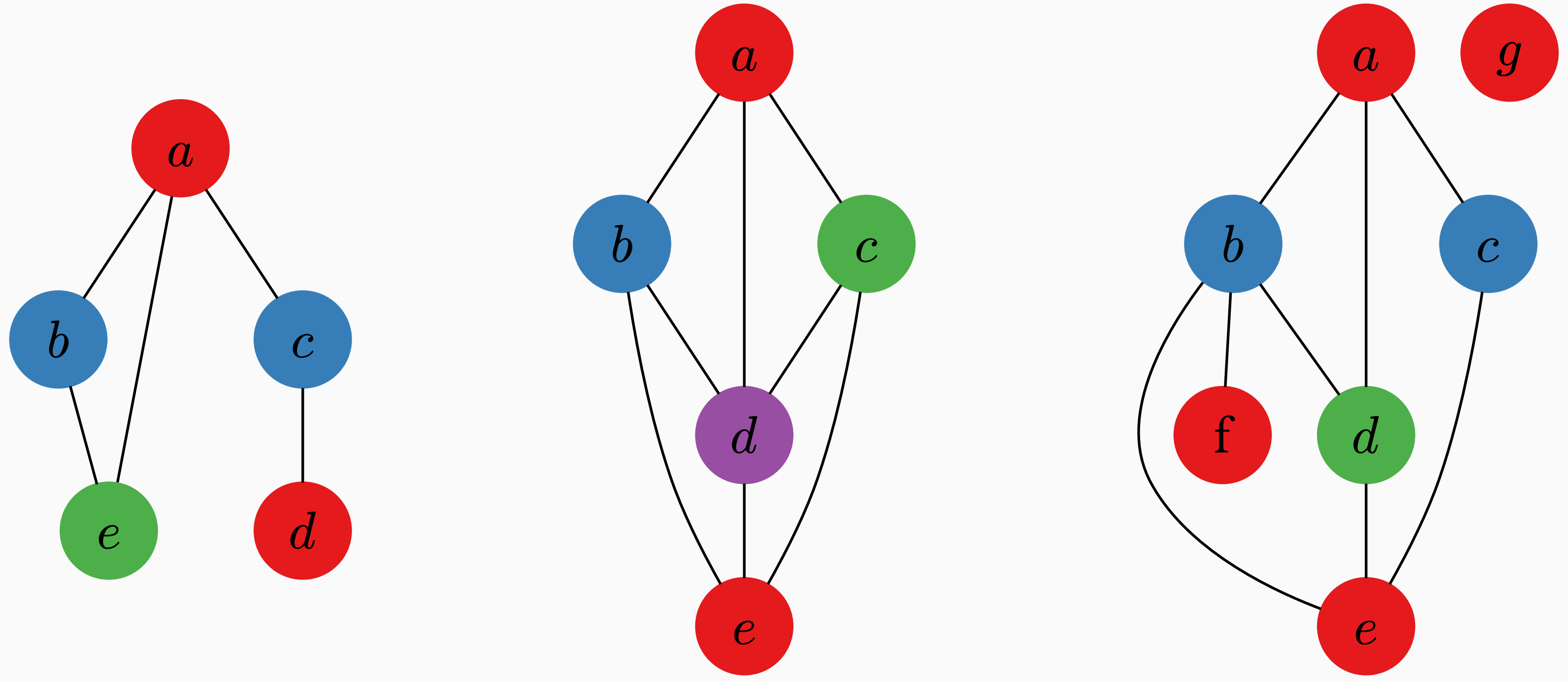
Tak v čem je problém? Při odebrání prvku, pokud se na vrch haldy dostane nevalidní prvek, musíme haldu *heapifikovat* dvakrát.

- Chytrý způsob
  1. nalezení klíče $O(1)$
  2. změna klíče $O(1)$
  3. heapifikace $O(\log n)$ z posice

Barevnost

—

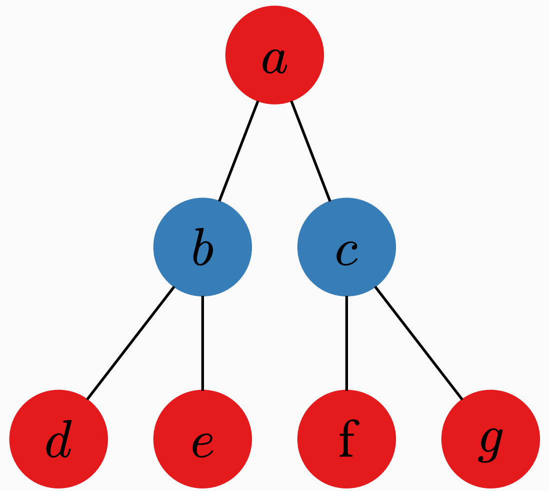
Obarvení grafu se říká přiřazení barev každému vrcholu tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu.

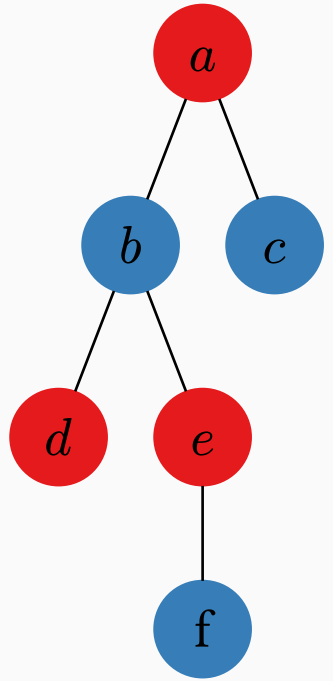


Barevnost

— Obarvení grafu - Graph coloring

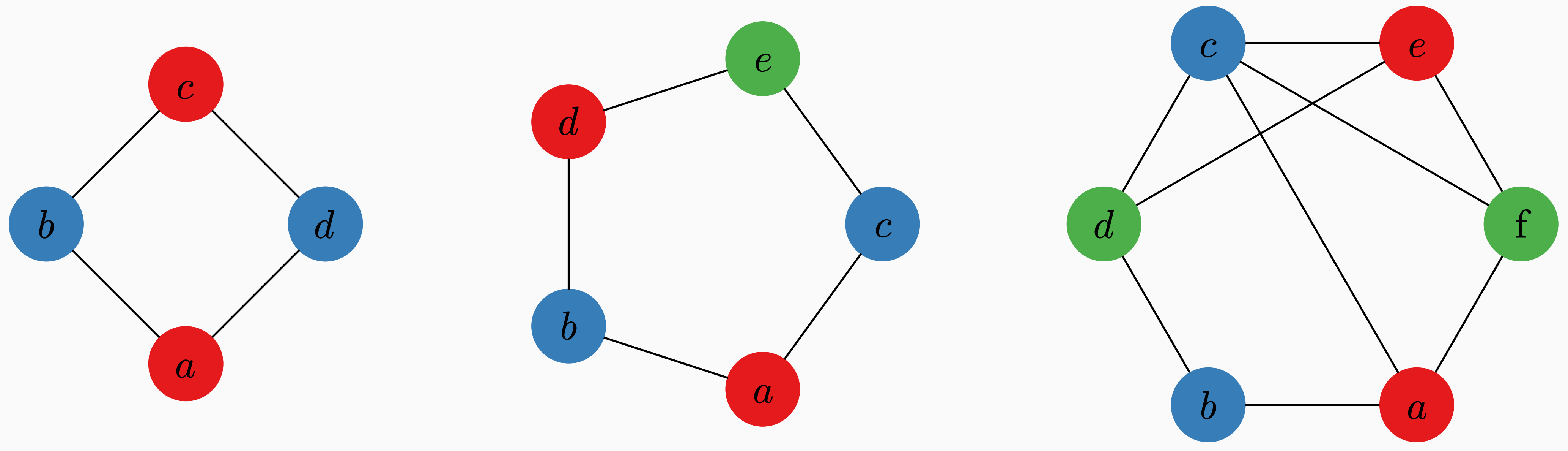
***Každý* strom má barevnost dva**, ale …





Cyklické grafy

Grafy s kružnicí mají barevnost ≥ 3, pokud je kružnice liché délky, ≥ 2 pokud je sudé délky.



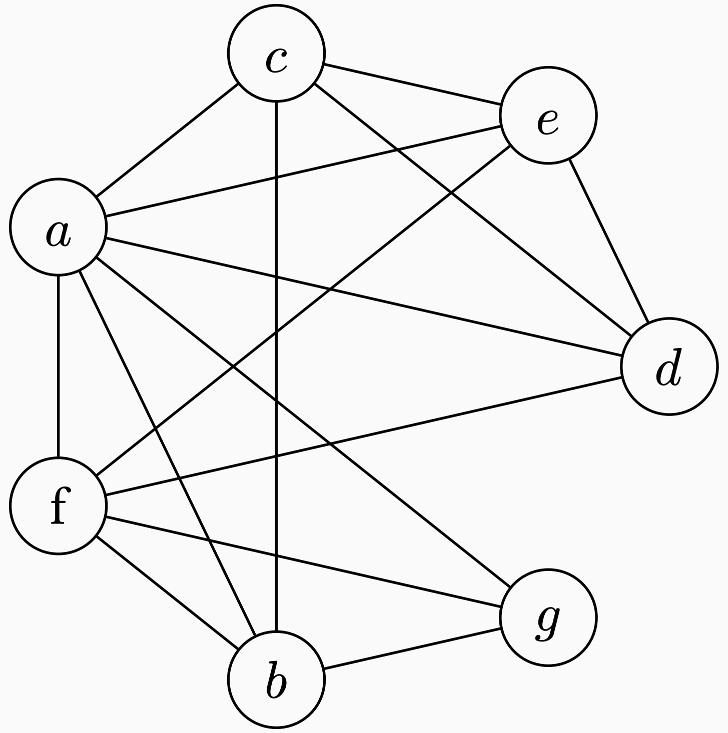
Barevnost
— Cyklické grafy

Jakou barevnost má tento graf?

Barevnost

—

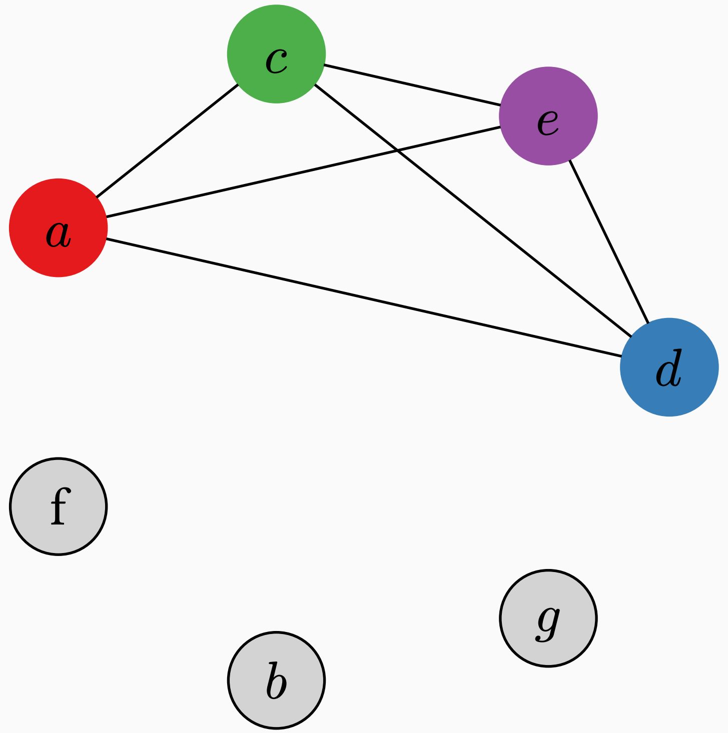
Jakou barevnost má tento graf?



Barevnost
— Jakou barevnost má tento graf?

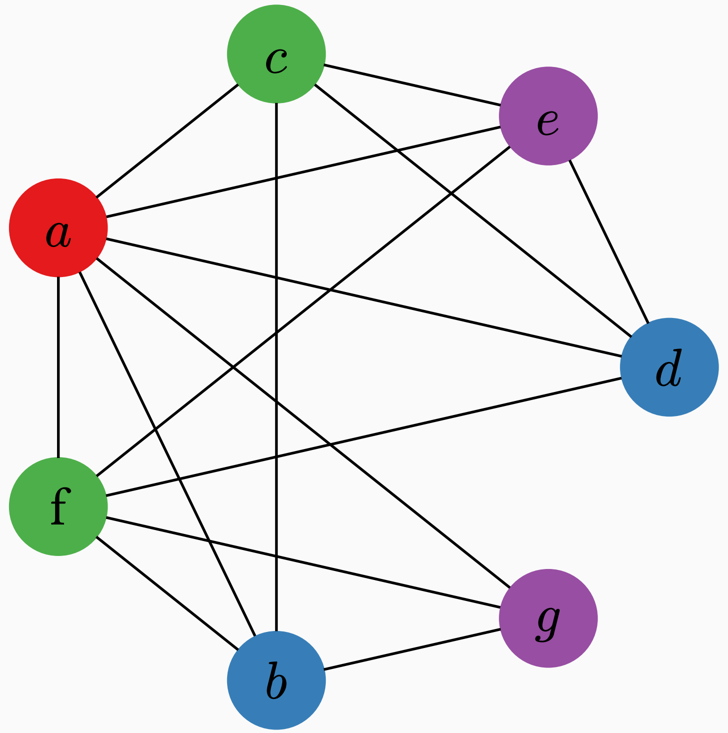
Jakou barevnost má tento graf?

Omezující je:



Barevnost
— Jakou barevnost má tento graf?

Barevnost 4



Barevnost

— Jakou barevnost má tento graf?

K čemu barevnost je?

- Návrh síťové infrastruktury
- Řešení her (Sudoku)
- Návrh rozvrhu
- Barvení map ve hrách
- Data mining
- Plánování procesů a přístup ke zdrojům
- Přejmenování registrů
- ...

Barevnost
— K čemu barevnost je?

- Uvažujeme vrcholy seřazené podle stupně
- Každému vrcholu přiřadíme *nejmenší* barvu, kterou nemá žádný jeho soused

Jedná se o **greedy** algoritmus, které obecně **nenajdou optimální** obarvení. Správnost obarvení záleží na zvoleném pořadí vrcholů. Welsh-Powell (řazení podle stupně) zaručuje obarvení **nanejvýše** o jednou barvou více, než je optimální.

Pseudokód

```
def welshPowell(graph)
  coloring = graph.nodes.map_to(Color.None)
  vertices = graph.nodes.sort_by(Node.Degree)
  for vertex in vertices
    neighbours = vertex.neighbours()
    color = Color.first_not_present_in(neighbours)
    coloring[vertex] = color

  return coloring
```

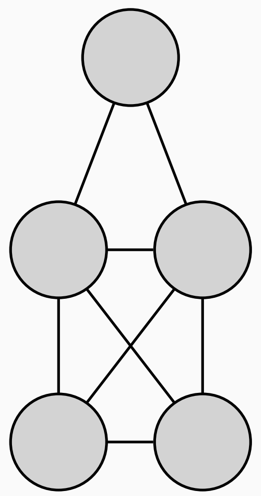

Barevnost

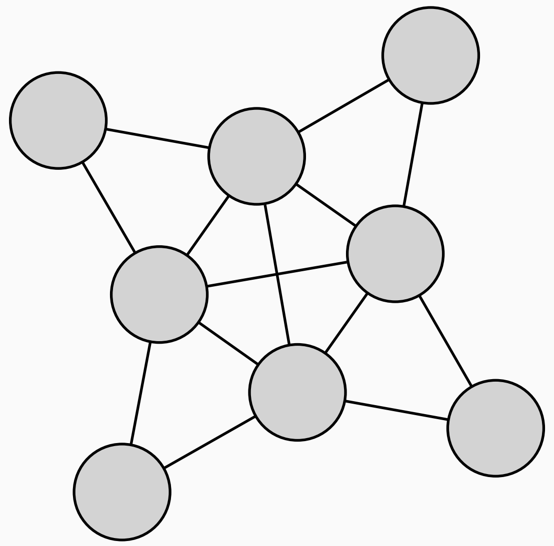
— Welsh-Powell - Implementace

Eulerovské grafy

Eulerovské grafy

—





Eulerovské grafy
— Graf jedním tahem

Storytime

https://cs.wikipedia.org/wiki/Sedm_most%C5%AF_m%C4%9Bsta_Kr%C3%A1lovce

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

https://www.matweb.cz/sedm-mostu/

https://wikisofia.cz/wiki/Teorie_s%C3%ADt%C3%AD_a_nov%C3%A1_m%C3%A9dia

https://www.britannica.com/science/Konigsberg-bridge-problem

Graf jedním tahem

- Aby graf byl nakreslitelný jedním tahem, tak musí:
  - Být souvislý
  - Mít 0 nebo 2 vrcholů s lichým stupněm
    - Pokud máme lichý stupeň, pak musíme začít tah v něm

Eulerovské grafy
— Graf jedním tahem

Algoritmy a implementace

Algoritmy a implementace

—

Prvky necháme seřadit haldou

- Vložit všechny prvky do minimální haldy
  - Halda se postará, aby nejmenší prvek byl navrchu

- Vybrat všechny prvky z haldy
  - Při každém odebrání se další nejmenší prvek dostane na vrch

Algoritmy a implementace
— Heapsort

Heapsort - Implementace

Pseudokód

```
def heapsort(list)
  heap = Heap()
  for element in list heap.insert(element)
  sorted = []
  while not heap.empty()
    sorted += heap.remove()
  return sorted
```


Algoritmy a implementace
— Heapsort - Implementace

Quicksort

Algoritmus vždy třídí okolo nějakého *pivotu*

- Vybereme *náhodný*[1] pivot
- Procházím pole zleva / zprava směrem k pivotu
- Pokud je prvek menší / větší prvek s pivotem prohodím
- Skončím když se levý a pravý konec setkají
  - Menší prvky jsou nalevo a větší napravo od pivotu
- Stejný algoritmus provedeme na levé a pravé polovině

[1] Na volbě prvního pivotu velmi záleží, protože nám určí jak dobře se prvky rozdělí

Algoritmy a implementace
— Quicksort

Mergesort

Algoritmus metody *Divide-and-Conquer*

1. Rozdělíme kolekci na jednotlivé prvky
2. Pod dvojicích je setříděně spojíme
3. Spojíme dvě dvojce
   - Prvky ve dvojci jsou seřazené
   - Vezmeme první prvek z obou a porovnáme, menší přidáme
   - Přidáme všechny čtyři prvky
4. Opakujeme krok 3 s o stupeň většími kolekcemi
5. Máme setříděný seznam

Algoritmy a implementace
— Mergesort

Mergesort - Implementace

Pseudokód

```
def sort(input)
  if input.length() <= 2
    merge(input[0], input[1])
  remerge(sort(input.split()[0]), sort(input.split()[1]))
```

Pseudokód

```
def merge(a, b)
  res = []
  while not a.empty() || not b.empty():
    res += min(a.peek(), b.peek()).consume()
  res
```


Algoritmy a implementace
— Mergesort - Implementace

Časová náročnost

- Heapsort
  - Worst / Average: $O(n \log n)$
  - Best: $O(n)$
- Quicksort
  - Worst: $O(n^2)$
  - Average: $O(n \log n)$
  - Best: $O(n)$
- Mergesort
  - Worst / Average: $O(n \log n)$
  - Best: $O(n)$

Algoritmy a implementace
— Časová náročnost

Vyhledávání

- Snažíme se najít konkrétní prvek v poli
- Obecně umíme v lineárním čase - $O(n)$
- BVS umožní průměrnou složitost snížit na $O(\log n)$
- Binární strom

- Prvky postupně vkládáme do stromu:
  - Začneme vkládat od kořene
  - Menší a stejné hodnoty se pokoušíme vložit do levého podstromu
  - Větší prvky vkládáme do pravého podstromu
- Při mazání nahradíme nejmenším prvkem v pravém podstromu, nebo největším prvkem v levém podstromu
- Při vyhledávání procházíme od kořene
  - Pokud hledáme menší hodnotu, rekurzivně prohledáme levý podstrom
  - Pokud hledáme větší hodnotu, prohledáme pravý podstrom

Algoritmy a implementace
— Binární vyhledávací strom

```java
void insert(int key) {
  if (value <= key) {
    if (left == null)
      left = new Node(key);
    else
      left.insert(key);
  } else {
    if (right == null)
      right = new Node(key);
    else
      right.insert(key);
  }
}
```

Java

Algoritmy a implementace
— BVS - insert

Java

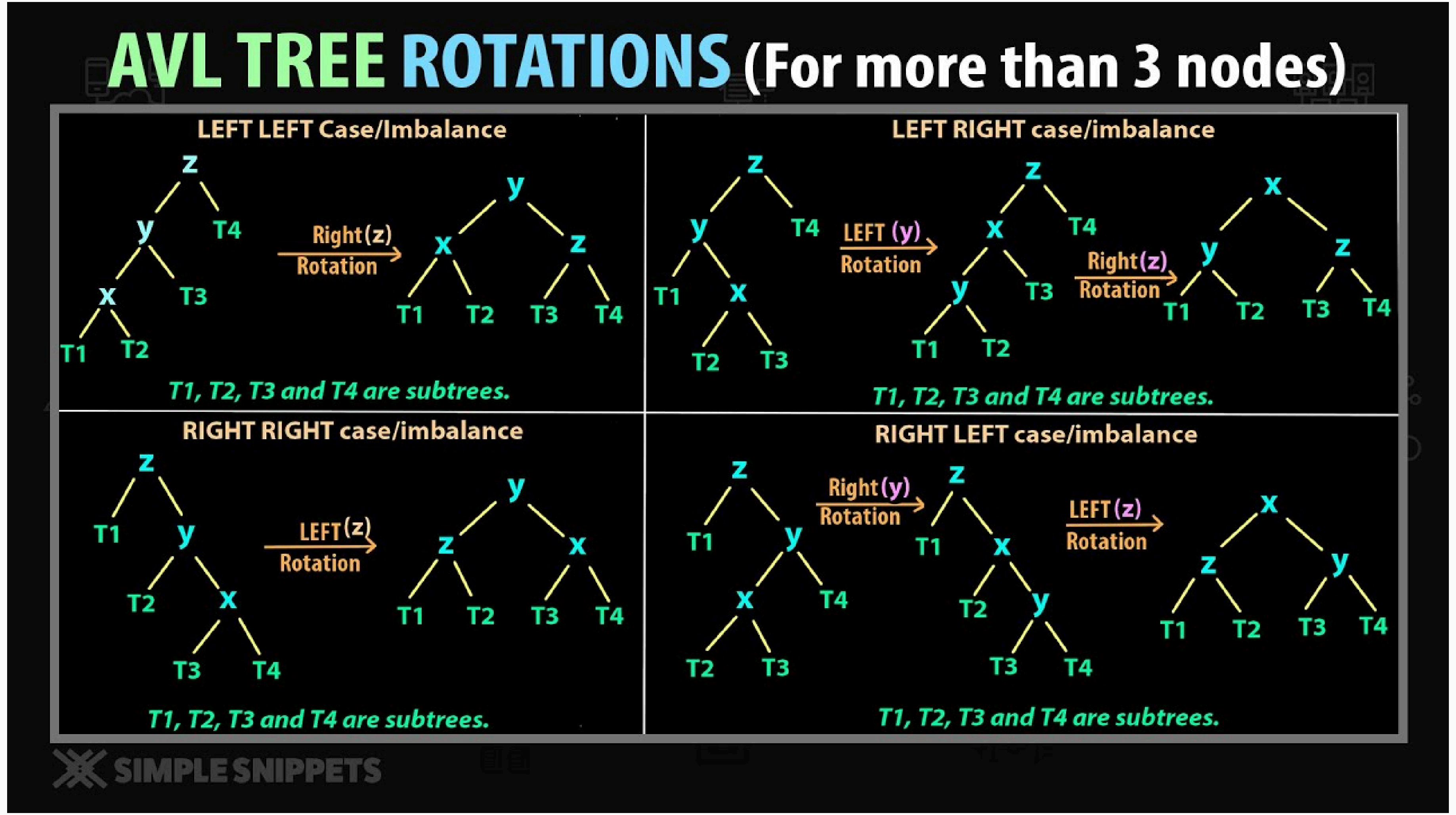
```java
Integer find(int key) {
  if (value == key)
    return value;

  if (value < key && left != null)
    return left.find(key);
  else if (value > key && right != null)
    return right.find(key);

  return null;
}
```

- BVS mohou být nevyvážené
  - vznikne nám *„šňůra“*
- AVL při vkládání a odebírání provádí kontrol vyváženosti
  - od přidaného / mazeného vrcholu se posouváme ke kořenu stromu
  - V každém uzlu zkontrolujeme vyváženost
    - hloubka pravého a levého podstromu se může lišit maximálně o jedna
  - Pokud je strom nevyvážený, provedeme rotaci
    - Left rotace: Pravý potomek uzlu se stane novým kořenem podstromu
    - Right rotace: Levý potomek uzlu se stane novým kořenem podstromu
    - Left right rotace: Levá rotace v potomkovi, pravá rotace v kořeni
    - Right left rotace: Pravá rotace v potomkovi, levá rotace v koření

Algoritmy a implementace
— AVL



Algoritmy a implementace
— AVL

Hry

Hry

—

Několik omezení

- Hra má jasně definované
  - Stavy
  - Přechody mezi stavy
  - Výherní podmínku
- Fixní počet hráčů

Můžeme analyzovat strategie například pro:

- Piškvorky
- Šachy
- Mariáš

Grafově znázorněné hry mají jasně danou výherní strategii, proč tedy nevíme jak přesně vyhrát každou hru šachu?

Paměť

Graf hry šachu bude *hodně* velký. Dobrým odhadem je $10^{65} \sim 2^{215}$ možných stavů hry... To je přibližně hodně!

Výpočet

S výpočetním výkonem to je obdobné.

Proč potřebujeme všechny stavy najednou?

Abychom mohli najít jádro grafu, tak **potřebujeme** znát všechny hrany a vrcholy v grafu hry. Některé stavy a přechody můžeme spojit (*kondensace grafu*)

Ideální jsou hry s málo možnými stavy.[1] Ukázkovou hrou bývá tahání sirek z hromádky:

- Na počátku máte 7 sirek
- Střídají se dva hráči
- V každém tahu může vzít 1 nebo 2 sirky
- Kdo vezme poslední sirku prohrává

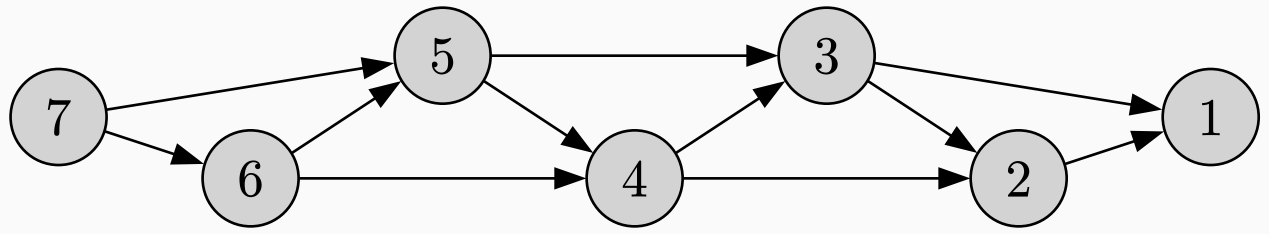
Možných stavů máme pouze 7 (jednotlivé počty sirek) a přechodů 11.

Jaká je optimální strategie?

[1] **málo** je opravdu důležité, třeba 3x3 piškvorky mají 5477 platných stavů

Hry
— K čemu to vlastně tedy je?

Sirky



Hry
— Sirky

Stavový automat je graf, kde vrcholy znázorňují nějaký stav (prostředí, programu, výpočtu, ...) a vrcholy možné přechody mezi jednotlivými stavy.

Hry
— Stavový automat

Jádro grafu

Hry
— Jádro grafu

Turnamenty

Turnamenty

—

Turnamenty

—

Superstromy

Binomické stromy

Superstromy
— Binomické stromy

Binomická halda

Superstromy
— Binomická halda

B-tree

B-tree je sebe balancující stromová struktura. Všechny běžné operace jsou v logaritmickém čase[1].

B-tree stupně m je strom, který:

- Každý vrchol má nanejvýše m potomků
- každý vrchol, který není list nebo kořen, má nejméně $\frac{m}{2}$ potomků
- všechny listy jsou ve stejné hloubce
- ne-listy s k potomky mají $k - 1$ záznamů

Každý vrchol je seřazený seznam hodnot.

[1] mluvíme o amortizovaném čase

B-tree - kořen

B-tree - listy

Superstromy
— B-tree - listy

B-tree - vnitřní vrcholy

Superstromy
— B-tree - vnitřní vrcholy

f

f

f

f

Superstromy
— Implementace - Odebírání