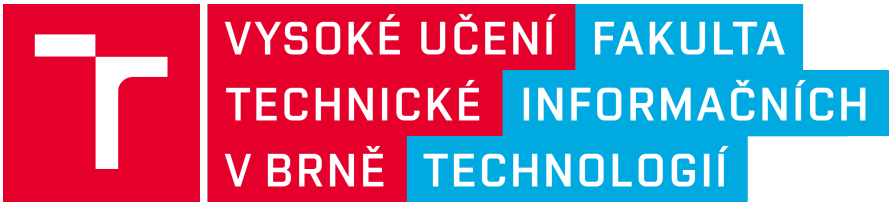



# Porovnání algoritmů výčtu cyklů v grafech

Bc. Jan Bíl
Bc. Michal Šedý

7. prosince 2022

# Obsah

# Kapitola 1

# Úvod

Orientovaný graf je struktura popisující množinu bodů (uzlů), které jsou mezi sebou propojeny orientovanými hranami. Cyklus v orientovaném grafu představuje takovou spojitou posloupnost uzlů, že se žádný uzel s výjimkou prvního a posledního v sekvenci neopakuje, a zároveň pro dvojici sousedících uzlů v posloupnosti $\ldots u_m u_n \ldots$ platí, že existuje orientovaná hrana vedoucí z uzlu $u_m$ do uzlu $u_n$. Tato práce se zabývá porovnáním algoritmů pro získání seznamu všech existujících cyklů v zadaném orientovaném grafu.

Vyhledávání (výčet) všech cyklů v grafu je využíváno v mnoha oblastech teorie grafů. Tato informace je používána k optimalizaci počítačových programů [1], při analýze booleovských sítí využívaných pro modelování biologických sítí nebo sítí genových regulátorů [9], při návrhu a vývoji komunikačních systému [8] nebo ověření jejich spolehlivosti a fault-tolerance [6], atd.

Tato páce byla vytvořena v rámci projektu "Porovnání - Hledání cyklů" do předmětu GAL (grafové algoritmy) a zkoumá efektivitu tři algoritmů pro výčet všech cyklů v orientovaných grafech. Na úvod jsou definovány základní pojmy a algoritmy týkající se orientovaných grafů. V následujících kapitolách jsou pak uvedeny jednotlivé algoritmy. Kapitola 3 popisuje přímočarý algoritmus [2, str. 287], který postupně prochází graf do hloubky a ověřuje, zda nebyl objeven cyklus. Algoritmus, který navrhl Herbert Weinblatt využívající zpětné navrácení [11] je uveden v kapitole 4. Kapitola 5 popisuje algoritmus Hongbo Liu a Jiaxin Wanga [5], který využívá frontu. Všechny tyto algoritmy byly implementovány v jazyce Python 3. Popis návrhu aplikace pro výčet cyklů s využitím uvedených algoritmů, včetně jejího používání jsou uvedeny v kapitolách 6 a 7. Experimenty porovnávající efektivitu jednotlivých postupů výčtu všech cyklů jsou uvedeny v kapitole 8. Mimo již zmíněných algoritmů byl také testován Johnsonův algoritmus [3] poskytován knihovnou Networkx[1].

[1] Dostupné z https://networkx.org

# Kapitola 2

# Prerekvizity

Tato kapitola poskytuje definice pojmů spojených s orientovanými grafy, jakými jsou definice orientovaného grafu, sledu, cesty a cyklu. Dále jsou popsány základní algoritmy prohledávání do hloubky (DFS) a topologického uspořádání, které jsou využívány pro zjednodušení grafů při vyhledávání cyklů. Tato kapitola je převzata z [4].

## 2.1 Orientovaný graf

**Definice 2.1.1** ***Orientovaný graf*** *je uspořádaná dvojice $G = (V, E)$, kde $V$ je množina uzlů grafu a $E \subseteq V \times V$ je množina orientovaných hran, kde hrana $(u, v) \in E$ znamená, že v grafu $G$ vede hrana z uzlu $u$ do uzlu $v$ (uzly $u, v$ jsou incidentní).*

Mějme orientovaný graf $G = (V, E)$ a nechť $u, v \in V$, pak lez graf $G$ reprezentovat v algoritmech dvěma různými způsoby. 1) jako pole $Adj$ seznamů sousedů, pro které platí $v \in Adj[u] \iff (u, v) \in E$. 2) jako incidentní matici $Adj_M$, kde $Adj_M[u][v] = 1 \iff (u, v) \in E$ a zároveň $Adj_M[u][v] = 0 \iff (u, v) \notin E$. Pro účely této práce byl zvolen první přístup, kterým je pole seznamů sousedů.

**Definice 2.1.2** *Nechť $G = (V, E)$ je orientovaný graf.* ***Transponovaný graf*** *grafu $G$ je $G^T = (V, E^T)$, kde $E^T = \{(v, u) \mid (u, v) \in E\}$.*

**Definice 2.1.3** ***Vstupní stupeň uzlu*** *je dán funkcí $d_+ : V \to \mathbb{N}_0$, která udává počet přechodu vstupujících do daného uzlu.*

**Definice 2.1.4** ***Výstupní stupeň uzlu*** *je dán funkcí $d_- : V \to \mathbb{N}_0$, která udává počet přechodu vystupujících z daného uzlu.*

Lze snadno ukázat, že pokud má uzel $u \in V$ hodnotu $d_-(u) = 0$ nebo $d_+(u) = 0$, pak nemůže být součástí žádného cyklu, protože pro každý stav obsažný v cyklu musí platit, že jeho vstupní i výstupní stupeň je nenulový, jinak by nemohl být cyklus "uzavřen". Tyto uzly s nulovým stupněm mohou být v části přípravy algoritmů pro výčet cyklů zanedbány (odstraněny). Toto zanedbání uzlu může snížit hodnotu vstupních nebo vstupních stupňů uzlů s ním incidentních na nulu. V takovém případě mohou být dále zanedbány také tyto uzly.

**Definice 2.1.5** ***Sled*** *je posloupnost vrcholů $\langle v_0 \dots v_n \rangle$, kde $n \in \mathbb{N}$, $v_i \in V$ pro $0 \leq i \leq n$, a $(v_{j-1}, v_j) \in E$ pro $1 \leq j \leq n$.*

**Definice 2.1.6** ***Cesta*** *(otevřená) je sled, ve kterém se neopakují uzly.*

**Definice 2.1.7** ***Cyklus*** *je cesta, ve které se shodují první a poslední uzel.*

## 2.2 Prohledávání do hloubky

Algoritmus prohledávání do hloubky (DFS) je základním algoritmem pro práci s grafy. DFS postupně prochází všechny uzly grafu $G = (V, E)$ a vytváří les prohledávání do hloubky.

**Definice 2.2.1** *Nechť $G = (V, E)$ je orientovaný graf a $\pi$ je pole předchůdců, kde $u \in \pi[v] \implies (u, v) \in E$.* ***Les prohledávání do hloubky*** *je orientovaný graf $G_\pi = (V, E_\pi)$, kde $E_\pi = \{(u, v) \in E \mid u = \pi[v]\}$.*

Během výpočtu se vytváří pole barev uzlů $color[u] \in \{WHITE, GRAY, BLACK\}$, pole časů prvního prozkoumání $d[u] \in \mathbb{N}$, pole časů dokončení prozkoumávání seznamu sousedů $f[u] \in \mathbb{N}$ a pole předchůdců $\pi[u] \subseteq V$.

**Algorithm 1:** DFS

**Input:** $G := (V, E)$
**Output:** $\pi, d, f$

```
1  Procedure DFS–VISIT(v)
2      color[u] ← GRAY
3      d[u] ← time ← time + 1
4      for v ∈ Adj do
5          if color[v] = WHITE then
6              π[v] ← u
7              DFS–VISIT(v)
8          end
9      end
10     f[u] ← time ← time + 1
11     color[u] ← BLACK
12 end
13 for u ∈ V do
14     color[u] ← WHITE
15     π[u] ← NIL
16 end
17 time ← 0
18 for u ∈ V do
19     if color[u] = WHITE then
20         DFS–VISIT(u)
21     end
22 end
23 return π, d, f
```

**Teorém 2.2.2** *Časová složitost algoritmu DFS je* $\mathcal{O}(|V| + |E|)$.

*Důkaz.* Inicializační část (13–16) má časovou složitost $\mathcal{O}(|V|)$. Hlavní cyklus (18–22) bude proveden maximálně $|V|$-krát, tedy časová obtížnost je $\mathcal{O}(|V|)$. Procedura DFS-VISIT je spouštěna pouze pro bílé uzly, tedy $|V|$-krát a cyklus (4–8) v proceduře je proveden maximálně $|Adj[v]|$-krát. Protože $\sum_{v \in V} |Adj[v]| = |E|$, je časová obtížnost cyklu 4–8 $\mathcal{O}(|E|)$. Celková složitost je tedy $\mathcal{O}(|V| + |E|)$. □

## 2.3 Topologické uspořádání

**Definice 2.3.1** ***Topologické uspořádání*** *orientovaného grafu* $G = (V, E)$ *je lineární uspořádání všech uzlů tak, že pokud* $(u, v) \in E$*, pak uzel* $u$ *předchází uzel* $v$ *v daném uspořádání.*

Pokud graf $G$ obsahuje cykly, poté není možné určit topologické uspořádání. Nicméně algoritmus lze spustit. Výsledkem bude *pseudo-topologické uspořádání*, ve kterém bude platit, že pokud $(u, v) \in E$ a zároveň se $u$ nenachází v žádném cyklu, pak $u$ předchází $v$ v daném uspořádání.

**Algorithm 2:** Topological-sort

**Input:** $G := (V, E)$
**Output:** $L$

1 zavolej DFS($G$) pro výpočet hodnot $f[v]$
2 každý dokončený uzel zařaď na začátek seznamu uzlů $L$
3 **return** $L$

**Teorém 2.3.2** *Časová složitost algoritmu topologického uspořádání je* $\mathcal{O}(|V| + |E|)$.

*Důkaz.* Protože výpočet topologického uspořádání využívá pouze DFS v časovou složitostí $\mathcal{O}(|V|+|E|)$ a operaci vložení na začátek seznamu má konstantní časovou složitost, je časová složitost topologického uspořádání $\mathcal{O}(|V| + |E|)$. □

## 2.4 Zanedbání stavů

Jak již bylo dříve řečeno, uzly, jejichž vstupní nebo výstupní stupeň je nulový, nemohou být součástí žádného cyklu, a proto mohou být při výčtu všech cyklů grafu zanedbány (odstraněny). Při zanedbání těchto uzlů se ale mohou změnit hodnoty funkcí $d_-$ a $d_+$ uzlů s nimi incidentními tak, že budou objeveny nové uzly s nulovým vstupním nebo výstupním stupněm. Pro jejich kompletní eliminaci slouží algoritmus 3.

**Teorém 2.4.1** *Časová složitost algoritmu zanedbání stavů je* $\mathcal{O}(|V| + |E|)$.

*Důkaz.* Transponování grafu má časovou složitost $\mathcal{O}(|V| + |E|)$. Topologické uspořádání má časovou složitost $\mathcal{O}(|V| + |E|)$. V proceduře Pruning se hlavní cyklus (2–9) provádí $|V|$-krát a vnitřní cyklus (4–6) se provádí $|Adj[v]|$-krát. Protože $\sum_{v \in V} |Adj[v]| = |E|$, je časová složitost procedury Pruning $\mathcal{O}(|V|+|E|)$, z čehož plyne, že časová složitost algoritmu zanedbání stavů je $\mathcal{O}(|V| + |E|)$. □

**Algorithm 3:** Zanedbání stavů

**Input:** $G := (V, E)$
**Output:** $G_{simply}$

```
1  Procedure Pruning(G_p := (V_p, E_p))
2      for u ∈ Topological-sort(G_p) do
3          if d_{p+}[u] = 0 then
4              for v ∈ Adj_p[u] do
5                  d_{p+}[v] ← d_{p+}[v] − 1
6              end
7              G_p.remove(u)
8          end
9      end
10 end
11 G_t ← G^T                                   // Transponujeme graf G
12 Pruning(G_t)                                // Smažeme zanedbatelné stavy v grafu G_t
13 G_simply ← G_t^T                            // Transponujeme graf G_t
14 Pruning(G_simply)                           // Smažeme zanedbatelné stavy v grafu G_simply
15 return G_simply
```

# Kapitola 3

# Brute-force

Tento algoritmus prvně publikoval S. M. Roberts a B. Flores [7] a následně byl upraven Tiernanen [10]. Finální podoba je popsána v knize od N. Deo [2]. Algoritmus prohledáváním prochází všechny možnosti, jeho časová náročnost je tedy velmi vysoká a je zde prostor pro různé optimalizace.

## 3.1 Popis algoritmu

Vrcholy vstupního grafu G jsou reprezentovány celými čísly $1, 2 \dots n$. Algoritmus začíná ve vrcholu $p_1$ a prohledáváním následovníků vytváří orientovanou cestu $P = \langle p_1 p_2 \dots p_k \rangle$. Když již cestu není možné rozšířit, algoritmus zkontroluje, zda existuje hrana $(p_k, p_1) \in E$. Pokud ano, je zaznamenán cyklus $\langle p_1 p_2 \dots p_k p_1 \rangle$. Poté se algoritmus vrátí o jeden vrchol zpět (do vrcholu $p_{k-1}$) a snaží se z tohoto vrcholu cestu rozšířit jinou hranou, pokud taková existuje. Vrchol $p_k$ je zakázán pro další prohledávání z vrcholu $p_{k-1}$ aby se zamezilo procházení stejných cest. Tento proces je opakován, dokud se výpočet nevrátí až k vrcholu $p_1$. V případě, že již byli všichni následníci $p_1$ zakázáni (byly nalezeny všechny cykly začínající z vrcholu $p_1$), je zvolen nový počáteční vrchol cesty $P$ a proces vyhledávání je opakován. Z důvodu zamezení zbytečným operacím při vyhledávání cyklů musí být zavedeny následující opatření:

- Během rozšiřování orientované cesty musí být zajištěna její korektnost (žádný vrchol se nesmí opakovat).
- Nesmí být zaznamenány žádné cyklické permutace cyklů (začínaly by pouze z jiného vrcholu). To je zajištěno podmínkou, že žádný vrchol $i \leq p_1$ není možné prozkoumat, pokud budovaná orientovaná cesta začíná ve vrcholu $p_1$.
- Během vyhledávání se nesmí žádná cesta procházet více než jednou. To je zajištěno aktualizováním binární matice $n \times n : H[h_{ij}]$. Pokud $h_{ij} = 1$, pak nesmí být použita hrana $(i, j) \in E$ pro rozšíření cesty $P$ (byla již jednou použita). Hodnota 0 informuje, že daná hrana může být použita pro rozšíření cesty. Řádek $i$ matice $H$ je nulován při zpětném navrácení z vrcholu $i$. Při prohledávání z nového počátečního vrcholu je matice $H$ nulová.

V algoritmu se používá indexace pomocí záporných čísel. Tato indexace má následující význam: index -1 značí poslední prvek listu, index -2 předposlední a tak dále. Pro sjednocení dvou cest $P_1 = \langle v_1 v_2 \rangle$ a $P_2 = \langle v_3 v_4 \rangle$ bude využíván operátor +, tedy $P_1 + P_2 = \langle v_1 v_2 v_3 v_4 \rangle$.

**Algorithm 4:** Bruteforce algorithm

**Input:** $G := (V, E), n := |V|$
**Output:** $L_{cycles}$

```
1  L_cycles ← EmptyList
2  H[0...n−1][0...n−1] ← 0                          // nulová matice n × n
3  for p_s ∈ V do
4      P ← EmptyList
5      P.append(p_s)
6      p_curr ← p_s
7      while True do
8          foundSuccessor ← False
9          for (p_curr, p_next) ∈ E do
10             if H[p_curr][p_next] = 0 ∧ p_next ∉ P ∧ p_next > p_s then
11                 P.append(p_next)
12                 p_curr ← p_next
13                 foundSuccessor ← True
14                 break
15             end
16         end
17         if foundSuccessor then
18             continue
19         else
20             if (p_curr, p_s) ∈ E then
21                 L_cycles.append(P + ⟨p_s⟩)
22             end
23             if p_curr = p_s then
24                 break
25             end
26             H[P[−2]][p_curr] ← 1
27             H[p_curr][0...n−1] ← 0
28             p_curr ← P[−2]
29             P.removeLast()
30         end
31     end
32 end
33 return L_cycles
```

## 3.2 Časová složitost

**Teorém 3.2.1** *Časová složitost Brute-Force algoritmu je* $\mathcal{O}(d^{|V|-d} \cdot d!)$*, kde* $d$ *je maximální výstupní stupeň v grafu.*

*Důkaz.* Na prohledávání cest se můžeme dívat jako na prohledávání stromu, kde každý vnitřní uzel má až $d$ potomků. Hloubka takového stromu může dosahovat až $|V| - 1 \approx |V|$. Ovšem je zřejmé, že v hloubce větší než $|V| - d$, začíná počet možných následníků klesat. V hloubce $|V|-1$ již není možné cestu rozšířit. V hloubce $|V|-2$ máme právě jednu možnost.

V hloubce $|V| - d$ máme $d - 1$ možností. Pro počet prohledaných uzlů rozdělíme strom na 2 části: v první má každý uzel $d$ následovníků až do hloubky $|V| - d$. Počet uzlů je tedy $d^{|V|-d}$. Druhá část stromu obsahuje $(d-1) \cdot (d-2) \cdot (d-3) \cdot \ldots \cdot 1 = (d-1)!$ uzlů, tedy celý strom má mohutnost $d^{|V|-d} \cdot (d-1)! \in \mathcal{O}(d^{|V|-d} \cdot d!)$. Bylo dokázáno, že časová složitost algoritmu je $\mathcal{O}(d^{|V|-d} \cdot d!)$. □

## 3.3 Prostorová složitost

**Teorém 3.3.1** *Prostorová složitost Brute-Force algoritmu je* $\mathcal{O}(|V|^2)$.

*Důkaz.* Matice $H$, která obsahuje $|V| \times |V|$ prvků má prostorovou složitost $\mathcal{O}(|V|^2)$. Dále si algoritmus musí pamatovat cestu P, která nabývá délky maximálně $|V| - 1 \in \mathcal{O}(|V|)$. Celková prostorová složitost je tedy $\mathcal{O}(|V|^2)$. □

# Kapitola 4

# Herbert Weinbaltt

Tento algoritmus byl publikován v roce 1972 Herbertem Weinblattem [11]. Jeho základem je Tiernanův algoritmus [10] publikovaný o dva roky dříve, který vyhodnocuje každý cyklus pouze jednou, jednalo se tedy o teoreticky nejefektivnější algoritmus, ovšem za cenu vyšších paměťových nároků. Sám Tiernan ale naznačil, že pro průměrně husté grafy s více než 100 hranami by bylo použití tohoto algoritmus neprakticky pomalé. Herbert Weinblatt ve svém článku popisuje nový přístup, který stejně jako Tiernanův vyhodnocuje každý cyklus pouze jednou, ale nově také minimalizuje množství prozkoumaných hran nutných k objevení cyklu. V důsledku toho dokázal algoritmus implementovaný v experimentálním jazyce Snobol3 na počítači IBM 7094 objevit všech 44 cyklů v grafu s 194 uzly a 294 hranami za méně než sedm sekund.

## 4.1 Popis algoritmu

Před vlastním popisem algoritmu je potřeba definovat pomocné funkce *END* a *TAIL*.

**Definice 4.1.1** ***END*** *je unární funkce, které pro cestu* $\langle v_0 v_1 \dots v_n \rangle$*, vrací poslední uzel cesty* $v_n$*.*

**Definice 4.1.2** ***TAIL*** *je binární funkce, která pro dvojici uzlu* $v_k$ *a otevřenou cestu* $\langle v_0 v_1 \dots v_k v_{k+1} \dots v_n \rangle$*, respektive cyklus* $\langle v_0 v_1 \dots v_k v_{k+1} \dots v_0 \rangle$ *vrací podcestu* $\langle v_{k+1} \dots v_n \rangle$*, respektive* $\langle v_{k+1} \dots v_0 \rangle$ *s respektem k uzlu* $v_k$*. V případě cyklu* $\langle v_k v_{k+1} \dots v_k \rangle$ *vrací prázdnou cestu* $\langle\rangle$ *délky 0.*

V průběhu výpočtu využívá algoritmus seznam $TT$, který reprezentuje aktuální zkoumanou cestu. Dále jsou udržovány dvě pomocné struktury $S_v$ a $S_e$. Kde $S_v$ je pomocné pole udržující informaci, zda již byl uzel $v \in V$ v seznamu $TT$. $S_v[v]$ může nabývat hodnot 0, 1, nebo 2, což indikuje, že uzel $v$ ještě nebyl v seznamu $TT$, uzel $v$ se momentálně nachází v seznamu $TT$, nebo že uzel $v$ byl v $TT$, ale již se nenachází. Obdobná informace je udržována pro hrany. $S_e$ je matice informující, zda již byla hrana $(u, v) \in E$ v seznamu $TT$. $S_e[u][v]$ může nabývat pouze dvou hodnot, a to 0, nebo 2, což indikuje, že hrana $(u, v)$ ještě nebyla v $TT$, respektive že hrana $(u, v)$ již byla v $TT$ a stále může být.

Pro sjednocení dvou cest $P_1 = \langle v_1 v_2 \rangle$ a $P_2 = \langle v_3 v_4 \rangle$ bude využíván operátor $+$, tedy $P_1 + P_2 = \langle v_1 v_2 v_3 v_4 \rangle$.

**Algorithm 5:** Herbert Weinbalttův algoritmus

**Input:** $G := (V, E), n := |V|$
**Output:** $L_{cycles}$

```
1  Procedure CONCAT(isRecursion, Path)
       // Inicializace lokálních proměnných
2      cycleTails ← EmptyList
3      toAddSave ← EmptyList
4      toAddToControl ← EmptyList
5      added ← EmptyList
6      v ← END(Path)
7      for cycle ∈ L_cycles do
8          tail ← TAIL(v, cycle)
9          if tail = ∅ ∨ tail ∈ L_cycles then
10             continue
11         end
12         cycleTails.append(tail)
13         if ∃v_k ∈ tail : v_k ∈ Path then
14             continue
15         end
16         cycleEnd ← END(cycle)
17         if S_v[cycleEnd] = 2 then
18             toAddToControl.extend(CONCAT(True, Path + tail))
19             continue
20         else
21             newCycle ←⟨cycleEnd⟩ + TAIL(cycleEnd, TT)+
                                Path + TAIL(END(Path), cycle)
22             if isRecursion then
23                 toAddToControl.append(newCycle)
24             else
25                 toAddSave.append(newCycle)
26             end
27         end
28     end
29     if isResursion then
30         return toAddToControl
31     else
32         L_cycles.extend(toAddSave)
33         added.extend(toAddSave)
34         for cycle ∈ toAddToControl do
35             if cycle ∉ added then
36                 L_cycles.append(cycle)
37                 added.append(cycle)
38             end
39         end
40     end
41 end
```

```
42 Procedure EXAMINE(v)
43     if S_v[v] = 0 then
44         S_v[v] ← 1
45         TT.append(v)
46     else if S_v[v] = 1 then
47         L_cycles.append(⟨v⟩ + TAIL(v, TT) + ⟨v⟩)
48     else
49         CONCAT(False, [v])
50 end
51 Procedure EXTEND
52     while TT ≠ ∅ do
53         u ← END(TT)
54         possible_v ← {v ∈ V | (u, v) ∈ E ∧ S_e[u][v] = 0}
55         if possible_v = ∅ then
56             S_v[u] ← 2
57             TT.removeLast()
58         else
59             v ← PickOne(possible_v)
60             S_e[u][v] ← 2
61             EXAMINE(v)
62         end
63     end
64 end
   // Inicializace globálních proměnných
65 TT ← EmptyList
66 S_e[0 . . . n − 1][0 . . . n − 1] ← 0                    // nulová matice n × n
67 S_v[0 . . . n − 1] ← 0
68 for v ∈ V do
69     if S_v[v] = 0 then
70         S_v[v] ← 1
71         TT.append(v)
72         EXTEND()
73     end
74 end
75 return L_cycles
```

Před spuštěním prohledávání grafu jsou všechny uzly, pro které se může hodnota $d_+$ nebo $d_-$ rovnou nule zanedbány (odstraněny) algoritmem 3. Cílem tohoto kroku je eliminovat uzly, které nemohou tvořit cykly a tím snížit velikost grafu nad kterým bude prohledávání prováděno.

Algoritmus náhodně vybere jeden uzel grafu (*počáteční uzel cesty*) a začne prozkoumávat všechny cesty vycházející z tohoto uzlu. Pokud se během vytváření cesty některý uzel navštíví vícekrát, je tato podcesta označena za cyklus a konstrukce cyklu se navrátí k předchozímu uzlu, pro který existují doposud neprozkoumaní následníci. Pokud již žádný

takový následník neexistuje, zvolí algoritmus nový, doposud nezvolený, *počáteční uzel cesty*. V případě, že takový uzel neexistuje, algoritmus skončí.

V průběhu výpočtu je cesta vedoucí z *počátečního uzlu cesty* reprezentovaná seznamem $TT$ ("trial thread"). Při návratu je odstraněn poslední uzel (nejvzdálenější od *počátečního uzlu*) cesty $TT$.

Když algoritmus dospěje k uzlu $v$, který již byl v minulosti vyhodnocován, pak před zpětným navrácením provede kontrolu, zda některá z doposud vyhodnocovaných hran netvoří nový cyklus. Pokud se uzel $v$ nachází v $TT$, pak existuje právě jeden cyklus, který je tvořen sjednocením $v$ s *TAIL* $TT$ s respektem k uzlu $v$. Pokud se uzel $v$ již nenachází v $TT$, pak může cyklus existovat pouze tehdy, pokud již byly dříve nějaké cykly obsahující $v$ objeveny. V takovém případě je spuštěna rekurzivní procedura CONCAT, která se snaží nalézt cestu, která začíná na uzlu $v$ a končí na některém uzlu $u$, který je stále na $TT$. Z každé takto nalezené cesty je vytvořen cyklus sjednocením této cesty s *TAIL* $TT$ respektujícím $u$. (Nechť $C_1 = \langle v_1 v_2 v_1 \rangle$ a $C_2 = \langle v_2 v_3 v_2 \rangle$ jsou dva již objevené cykly, $TT = \langle v_1 \rangle$ a algoritmus objeví již jednou zpracovaný uzel $v_3$. V takovém případě je konkatenací podcesty $\langle v_1 \rangle$ z $TT$, podcesty $\langle v_3 v_2 \rangle$ z $C_2$ a podcesty $\langle v_1 \rangle$ z $C_1$ vytvořen nový cyklus $\langle v_1 v_3 v_2 v_1 \rangle$.)

## 4.2 Časová složitost

**Teorém 4.2.1** *Časová složitost Herbert Weinblattova algoritmu pro výčet všech cyklů v orientovaném grafu je* $\mathcal{O}(|V|^2 + |E| \cdot |V| \cdot (c+1))$*, kde* $c$ *je počet cyklů v grafu.*

*Důkaz.* Imicializace glogálních proměnných (65–67) má časovou složitost $\mathcal{O}(|V|^2)$. Časová složitost zanedbání uzlů je $\mathcal{O}(|V| + |E|)$. Hlavní cyklus (68–74) a zavolání procedury EXTEND bude provedeno nejvýše $|V|$-krát. Cyklus (62–63) v proceduře EXTEND a vykonání procedury EXAMINE bude provedeno nejvýše $|E|$-krát, protože podmínkou pro neprázdnost *possible_v* je, že existuje hrana $(u, v) \in E$, pro který je $S_e[u][v] = 0$. Pro vybranou hranu $(u, v)$ je následně nastaveno $S_e[u][v]$ na hodnotu 2. Časová složitost bez procedury CONCAT je $\mathcal{O}(|V|^2 + |E|)$.

Nerekurzivní volání procedury CONCAT (49) je uskutečněno maximálně $|E|$-krát, protože je provedeno z procedury EXAMINE. Hlavní cyklus (7–28) procedury CONCAT stejně jako cyklus pro odstranění duplikátů (34–39) je proveden pro každý cyklus. Procedura CONCAT může být volána rekurzivně pro každý vrchol nejvýše jednou kvůli podmínce (17) $S_v[cycleEnd] = 2$. Volání procedůry CONCAT tak má časovou složitost $\mathcal{O}(c \cdot |E| \cdot |V|)$.

Celková složitost algoritmu je tedy $\mathcal{O}(|V|^2 + |E| \cdot |V| \cdot (c+1))$. Hodnota $c+1$ je použita pro případy, kdy $c = 0$. I v takových případech totiž bude provedeno prohledávání. □

## 4.3 Prostorová složitost

**Teorém 4.3.1** *Prostorová složitost Herbert Weinblattova algoritmu pro výčet všech cyklů v orientovaném grafu je* $\mathcal{O}(|V|^2)$.

*Důkaz.* Algoritmus využívá tři globální pomocné struktury. Prostorová složitost $TT$ je $\mathcal{O}(|V|)$ protože maximální délka cyklu je shora omezena na $|V|+1$. Prostorová složitost matice $S_e$ je $\mathcal{O}(|V|^2)$. A prostorová složitost pole $S_v$ je $\mathcal{O}(|V|)$. Prostorovou složitost lokálních pomocných proměnných *added*, *cycleTails*, *toAddSave* a *toAddToControl* nikdy nemůže překročit $\mathcal{O}(|V|^2)$, protože v proceduře CONCAT bude vytvořeno maximálně $|V|^2$ cyklů.

Bylo dokázáno, že prostorová složitost Herbert Weinblattova algoritmu pro výčet všech cyklů v orientovaném grafu je $\mathcal{O}(|V|^2)$. □

# Kapitola 5

# Hongbo Liu a Jiaxin Wang

V roce 2006 spolu Hongo Liu a Jiaxin Wang publikovali algoritmus [5] pro výčet všech cyklů v grafech. Tento algoritmus není tak efektivní jako algoritmus Johnsona [3]. V porovnání s ním je ale jednodušší pro porozumění, čímž jsou minimalizovány chyby v implementaci způsobené chybnou interpretací. Na druhou stranu Liuův a Wangův přístup je neefektivní pro velké grafy. Nicméně existují případy, pro které je jejich řešení efektivnější.

Algoritmus využívá frontu zkoumaných cest $P_0 \dots P_n$ pro $n \in \mathbb{N}_0$, kde pro délky cest ve frontě platí $|P_n| \leq |P_0| + 1$ a zároveň $|P_{i-1}| \leq |P_i|$. Díky tomu jej lze snadno využít pro výčet všech cyklů délek maximálně $k \in \mathbb{N}$ bez nutnosti nalezení všech cyklů.

## 5.1 Popis algoritmu

Algoritmus využívá pomocné funkce *END* (byla již definována dříve 4.1.1) a *HEAD*.

**Definice 5.1.1** ***HEAD*** *je unární funkce, které pro cestu* $\langle v_0 v_1 \dots v_n \rangle$*, vrací první uzel cesty* $v_0$*.*

Pro sjednocení dvou cest $P_1 = \langle v_1 v_2 \rangle$ a $P_2 = \langle v_3 v_4 \rangle$ budeme využívat operátor $+$, tedy $P_1 + P_2 = \langle v_1 v_2 v_3 v_4 \rangle$.

Aby se zabránilo duplicitní detekci cyklů, které byly generovány z jiných počátečních uzlů, ale jinak jsou si zcela ekvivalentní, je každému uzlu $v$ přiřazena různá hodnota $ord(v) \in \mathbb{N}_0$, kde pro $u, v \in V$ platí $ord(u) = ord(v) \iff u = v$. Při prohledávání cesty $P$ v grafu pak může být uzel $v$ připojen na konec cesty $P$ pouze pokud $ord(v) > ord(HEAD(P))$.

Před spuštěním prohledávání grafu jsou všechny uzly, pro které se může hodnota $d_+$ nebo $d_-$ rovnou nule zanedbány (odstraněny) algoritmem 3. Cílem tohoto kroku je eliminovat uzly, které nemohou tvořit cykly a tím snížit velikost grafu nad kterým bude prohledávání prováděno.

Při inicializaci proměnných (1–4) jsou všechny cesty délek 0 vloženy do fronty cest $Q$. V hlavním cyklu (5–19) se algoritmus snaží vytvořit cykly délky $k$ pro cesty délek $k - 1$. Cyklus délky $k$ je tvořen sjednocením cesty $P$ délky $k - 1$ a hrany $(END(P), HEAD(P))$. Z cesty $P$ jsou dále na základě přechodů vedoucích z $END(P)$ vygenerovány nové cesty $P + \langle v \rangle$ pro $v \in \{v \in Adj[END(P)] \mid v \notin P \wedge ord(v) > ord(HEAD(P))\}$, které jsou následně vloženy do fronty cest $Q$.

Pokud je fronta cest $Q$ prázdná (nebylo již možné vygenerovat další cesty), je algoritmus ukončen a všechny objevené cykly jsou vráceny v seznamu $L_{cycles}$.

**Algorithm 6:** Liuův a Wangův algoritmus

**Input:** $G := (V, E)$
**Output:** $L_{cycles}$

```
   // Inicializace
 1 Q ← EmptyQueue
 2 for v ∈ V do
 3 |   ENQUEUE(Q, ⟨v⟩)
 4 end
 5 while Q ≠ ∅ do
 6 |   P ← DEQUEUE(Q)
 7 |   head ← HEAD(P)
 8 |   end ← END(P)
 9 |   for v ∈ Adj[end] do
10 |   |   if v = head then
11 |   |   |   L_cycles.append(P + ⟨v⟩)
12 |   |   else if ord(v) > ord(head) then
13 |   |   |   if v ∉ P then
14 |   |   |   |   ENQUEUE(Q, P + ⟨v⟩)
15 |   |   |   end
16 |   |   end
17 |   end
18 end
19 return L_cycles
```

## 5.2 Časová složitost

**Teorém 5.2.1** *Časová složitost algoritmu, který publikovali Hongo Liu a Jiaxin Wang je* $\mathcal{O}(d^{|V|-d} \cdot d!)$, *kde* $d = max_{v \in V}(d_-(v))$ *je maximální výstupní stupeň v grafu.*

*Důkaz.* Časová složitost zanedbání uzlů je $\mathcal{O}(|V| + |E|)$. Inicializace a naplnění fronty (1–4) $Q$ má časovou složitost $\mathcal{O}(|V|)$. Halvní cyklus (5–18) bude proveden tolikrát, kolik existuje cest $P$ odpovídajících podmínce $\forall v \in P : ord(v) \geq ord(HEAD(P))$. Takových cest může být až $X$, kde

$$X = \sum_{n=1}^{|V|} (d-1)^{\alpha} \cdot (\beta - 1)!$$

$\alpha = \frac{|n-d|+n-d}{2}$ a $\beta = min(d, n)$. Pokud se $d$ blíží $|V|$, pak $X \approx |V|!$. Pokud se $d$ blíží 1, pak $X \approx d^{|V|}$. Časová složitost algoritmu je tedy $\mathcal{O}(d^{|V|-d} \cdot d!)$.

□

## 5.3 Prostorová složitost

**Teorém 5.3.1** *Prostorová složitost algoritmu, který publikovali Hongo Liu a Jiaxin Wang je* $\mathcal{O}(d^{|V|-d} \cdot d!)$*, kde* $d = max_{v \in V}(d_{-}(v))$ *je maximální výstupní stupeň v grafu.*

*Důkaz.* Do fronty $Q$ se postupně ukládají všechny cesty v grafu. Do fronty může být celkem uloženo až $X$ cest, kde

$$X = \sum_{n=1}^{|V|} (d-1)^{\alpha} \cdot (\beta - 1)!$$

$\alpha = \frac{|n-d|+n-d}{2}$ a $\beta = min(d, n)$. Součet všech délek všech $X$ cest je

$$Y = \sum_{n=1}^{|V|} (d-1)^{\alpha} * (\beta - 1)! \cdot n$$

Obsah sumy dosahuje maxima pro $n = |V|$. Pokud se $d$ blíží $|V|$, pak $Y \approx |V|!$. Pokud se $d$ blíží 1, pak $Y \approx d^{|V|}$. Lze vidět, že prostorová složitost algoritmu je definována prostorovou složitostí fronty. Prostorová složitost algoritmu je tedy $\mathcal{O}(d^{|V|-d} \cdot d!)$. □

# Kapitola 6

# Návrh programu

Tato kapitola popisuje jednotlivé komponenty, z nichž sestává program demonstrující výčet všech cklů v orientovaných grafech. Veškerá implementace byla provedena v jazyce Python 3 (verze 3.8+) s využitím knihoven (Bidict[1], Networkx[2] a Numpy[3]).

## 6.1 Moduly

Za účelem zapouzdření jsou veškeré algoritmy a třídy umístěny do balíku `gal`. Ten obsahuje modul `digraph`, ve kterém se nachází definice třídy orientovaných grafů `DiGraph`, modul `parser` s třídou `Parser`, která slouží pro získání orientovaného grafu ze vstupního souboru a modul `algorithms`, který obsahuje definice jednotlivých algoritmů pro výčet všech cyklů.

### 6.1.1 digraph

Třída grafů disponuje základními metodami pro práci s grafy, jakými jsou přidávání uzlů (`add_vertex`), přidávání hran (`add_edge`), transponování (`transpose`), získání indukovaného grafu (`get_induced_graph`), výpočet DFS lesa (`dfs`) nebo získání topologického uspořádání (`get_topological_sort`).

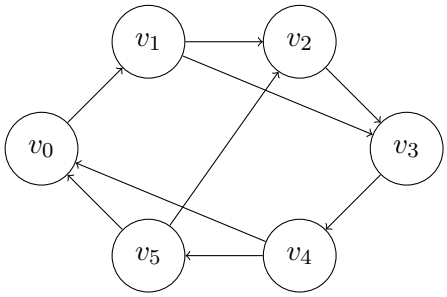
Pro zjednodušení interní reprezentace uzlů a pokrytí funkce *ord* využívané v algoritmech jsou uzly reprezentovány číselnou hodnotou, kde pro dva uzly $u, v \in V$ platí $ord(u) = ord(v) \iff u = v$. Informace o původním jménu uzlu je uložena v obousměrném slovníku `vertex_cname`.

Třída `DiGraph` poskytuje možnost generování úplných grafů (`create_complete_graph`), multi-cyklických grafů (`create_multicycle_graph`) a v poslední řadě takzvaných nested grafů (`create_nested_graph`). Tyto generované grafy byly použity pro testování a při získávání experimentálních výsledků.

**Definice 6.1.1** *Nechť $G = (V, E)$ je orientovaný graf. $G$ se nazývá **úplný orientovaný graf**, pokud $E = V \times V$.*

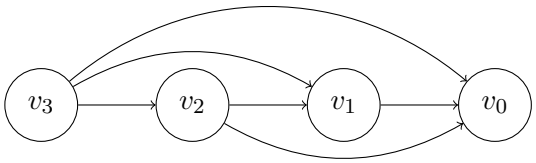
**Definice 6.1.2** *Nechť $G = (V, E)$ je orientovaný graf. $G$ se nazývá **multi-cyklický orientovaný graf**, pokud $|E| > |V|$ a zároveň existuje v grafu cyklus $C = \langle v_1 v_2 \dots v_n \rangle$, kde $|C| = |V| + 1$.*

[1] Bidict dostupné z: https://bidict.readthedocs.io
[2] Networkx dostupné z: https://networkx.org
[3] Numpy dostupné z: https://www.numpy.org




Obrázek 6.3: Multi-cyklický orientovaný graf.

**Definice 6.1.3** *Nechť $G = (V, E)$ je orientovaný graf. $G$ se nazývá **nested orientovaný graf**, pokud $E = \{(u, v) \mid ((u, m), (v, n)) \in (V \nabla \mathbb{N})^2 \wedge n < m\}$, kde $\nabla$ je diagonální produkt*[4].




Obrázek 6.5: Nested orientovaný graf.

### 6.1.2 parser

Třída `Parser` poskytuje statickou metodu pro načítání grafů ze vstupních souborů. Tato metoda vrací instanci třídy `DiGraph`. Vstupní soubor obsahuje na každém řádku jednu hranu. Hrana je reprezentován dvojici vrcholů oddělených mezerou. Jména vrcholů mohou obsahovat pouze alfanumerické znaky.

Příklad formátu grafu $G = (\{v_0, v_1, v_2\}, \{(v_0, v_1), (v_1, v_2), (v_2, v_0)\})$:

```
v0 v1
v1 v2
v2 v0
```

### 6.1.3 algorithms

Modul `algorithms` obsahuje implementace jednotlivých algoritmů pro výčet cyklů v grafech. Pro porovnání je zde také použita knihovna Networkx. Jednotlivé implementace nejsou k dispozici přípo, ale přez volání funkce `get_cycles`, která na základě specifikovaného prametru `algo` vybere, zda bude použit algoritmus z knihovny Networkx [3] (`algo="nx"`), brute-force algorithmus [2, str. 287] (`algo="bf"`), Herbert Weinbalttův algoritmus [11] (`algo="wein"`), nebo algoritmus Hongo Liu a Jiaxin Wanga [5] (`algo="hj"`).

[4] Produkuje pouze dvojice prvků ležící na diagonále množinového produktu.

## 6.2 Použité knihovny

Program využívá knihovnu Networkx (verze 2.8.7) pro porovnávání efektivity implementovaných algoritmů, knihovnu Bidict (verze 0.22.0) implementující obousměrný slovník využívaný pro obousměrný překlad jmen uzlů na jejích pořadová čísla a knihovnu Numpy (verze 1.23.3) k reprezentaci matic. Tyto závislosti lza nainstalovat pomocí nástroje pip3, nebo do virtuálního prostředí `gal-env` pomocí skriptu `build.sh` unístěného v kořenové složce projektu.

Dále je využita knihovna Argparse, pro zpracování argumentů příkazové řádky, která je ale součásti stendartních knihoven python již od verze 3.2.

# Kapitola 7

# Použití programu

V této kapitole je popsána struktura složky projektu. Dále je uveden postup pro instalaci knihoven (Bidict[1], Networkx[2] a Numpy[3]) využívaných programem. Kapitola také uvádí nápovědu pro spuštění programu pomocí interpretu Python 3 včetně vzorového formátu vstup a výstupu, po kterých následují příklady jednotlivých možností spuštění. Na závěr kapitoly jsou popsány automatické testy, který byly při implementaci použity.

## 7.1 Struktura projektu

Na kořenové úrovni projektu se nachází tři složky. Složka `/src` obsahuje modul `gal` a spustitelný program `enum_cycles.py`. Příklady grafů spolu s nulticyklickými grafy využitými při experimentech se nachází ve složce `/graphs`. Tato dokumentace je umístěná ve složce `/doc`. Na kořenové úrovni projektu se také nachází soubor `requirements.txt` obsahující využívané knihovny a skript `build.sh`, který je nainstaluje do virtuálního prostření `gal-env`. Skrip s automatickými testy pro modul `gal` je umístěn ve složce `/tests`.

## 7.2 Instalace závislostí

Potřebné knihovny využívané programem lze instalovat pomocí vástroje pip3 do virtuálního prostředí `gal-env` spuštěním skriptu `build.sh`. Spuětění vistuálního prostředí lze provést příkazen `source ./gal-env/bin/activate` v kořenové úrovni projektu.

## 7.3 Spuštění programu

Při spuštěni programu lze přepínači zvolit algoritmus, který bude použit pro výčet všech cyklů v zadaném grafu. Lze vybírat mezi algoritmem z knihovny Networkx, brute-force algoritmem, algoritmem Hongbo Liu a Jiaxin Wanga, nebo algoritmem Herberta Weinblatta. Musí být zadán přávě jeden přepínač určující algoritmus.

Dále lze zpracovávat graf zadaný vstupním souborem, nebo vygenerovaný (úplný, multicyklický, nebo nested). Právě jeden vstupní graf musí být zadán.

[1] Bidict dostupné z: https://bidict.readthedocs.io
[2] Networkx dostupné z: https://networkx.org
[3] Numpy dostupné z: https://www.numpy.org

Program `enum_cycles.py` umístěný ve složce `/src` lze po úspěšné instalaci závislostí spustit ve virtuálním prostředí `gal-env` s využitím interpretu python3 (verze 3.8+):

```
enum_cycles.py (--nx | --bf | --hj | --wein) [-c N] [-m N M] [-n N] [input]

poziční argumenty:
 input                       Vstupní soubor s grafem.

přepínače:

 -h,      --help             Zobrazí tuto nápovědu.
 --nx                        Použije algoritmus z knihovny Networkx.
 --bd                        Použije brute-force algoritmus.
 --hj                        Použije algoritmus Hongbo Liu a Jiaxin Wanga.
 --wein                      Použije algoritmus Herberta Weinblatta.
 -c N,    --complete N       Spustí výčet cyklů nad úplným grafem s N uzly.
 -m N M,  --multicycle N M   Spustí výčet cyklů nad multi-cyklickým grafem
                             s N uzly a X hranami, kde X se blíží N+M.
 -n N,    --nested N         Spustí výčet cyklů nad nested grafem s N uzly.
```

## 7.4 Formát vstupu

Vstupní soubor s příponou `.grph` obsahuje orientovaný graf. Každý řádek souboru reprezentuje jednu hranu. Uzel z kterého hrana vystupuje a uzel, do kterého vede jsou na řádku odděleny mezerou. Jména uzlu mohou sestávat z alfanumerických znaků. V tomto formátu jsou brány v úvahu pouze ty uzly grafu, pro které existuje hrana. Podpora pro izolované uzly bez hran není implementována (tyto uzly nikdy nemohou být součástí cyklu).

Příklad vstupního grafu:

```
v0 v0
v0 v1
v1 v2
v2 v0
```

## 7.5 Formát výstupu

Všechny nalezené cykly jsou vypsány na `STDOUT`. Na každém řádku se nachází právě jeden cyklus. Cyklus je reprezentován posloupnosti vrcholů oddělených mezerou. Poslední uzel cyklu, který je shodný s počátečním uzlem není uváděn (po vzoru knihovny Networkx).

Následující příklad výstupu je výsledkem prohledávání grafu uvedeného v předchozí sekci 7.4.

Příklad výstupu:

```
v0
v0 v1 v2
```

## 7.6 Příklady spuštění

Tato sekce uvádí vybrané přiklady spuštění programu.

### Networkx + vstupní soubor

```
python3 enum_cycles.py --nx g.grph
```

Program provede výčet včech cyklů nad grafem uvedeném v souboru `g.grph` s využitím algoritmu z knihovny Networkx.

### Brute-Force + úplný graf

```
python3 enum_cycles.py --bf -c 6
```

Program provede výčet všech cyklů nad úplným grafem $G_c = (V, E)$, kde $|V| = 6$ s využitím Brute-Force algoritmu (3).

### Hongbo Liu a Jiaxin Wang + multi-cyklický graf

```
python3 enum_cycles.py --hj -m 10 5
```

Program provede výčet všech cyklů nad multi-cyklickým grafem $G_m = (V, E)$, kde $|V| = 10$ a $|E| \to 10 + 5$ s využitím Hongbo Liu a Jiaxin Wangovým algoritmem (4).

### Herbert Weinblatt + nested graf

```
python3 enum_cycles.py --bf -n 9
```

Program provede výčet všech cyklů nad nested grafem $G_n = (V, E)$, kde $|V| = 9$ s využitím Herbert Weinblattova algoritmu (5). Výsledkem tohoto prohledávání musí být vždy 0 cyklů.

## 7.7 Automatické testy

Při vývoji modulu `gal` byl použit skript `run_tests.py` s automatickými testy, který je umístěn ve složce `/tests`. Testy jsou prováděny na úplných grafech pro $|V| \in \langle 1, 6 \rangle$, multi-cyklických grafech pro kombinace $|V| \in \langle 1, 20 \rangle$ a $|E| \in \langle |V|, |V| + 20 \rangle$ a na nested grafech pro $|V| \in \langle 1, 100 \rangle$. Za referenční výsledky jsou zvoleny cykly, které jsou objeveny pomocí knihovny Networkx.

Za účelem vyžší přehlednosti je během testování tištěna informace o počtu provedených testů pomocí knihovny `tqdm`[4] (verze 4.64.1). Tato knihovna je zahrnuta v souboru závislostí `requirements.txt`.

[4] Tqdm dostupné z: https://tqdm.github.io

# Kapitola 8

# Experimenty

Tato kapitola popisuje výsledky experimentálních měření pro studované algoritmy výčtu cyklů v orientovaných grafech. Za účelem testování byly zvoleny úplné grafy a mulit-cyklické grafy. Během implementace programu se ukázalo, že nested grafy nejsou vhodné pro porovnávání algoritmu (všechny algoritmy byly stejně efektivní). Z tohoto důvodu byly nested grafy použity pouze pro testování.

Úplné grafy byly vybrány z důvodu své vysoké komplexnosti, navíc jsou často používané jako hlavní měřítko efektivity algoritmů pro výčet cyklů.

Multi-cyklické grafy slouží k simulování prohledávání nad řidkými grafy obsahujícími cykly. Hlavním cílem tohoto porovnávání je zjistit, jak se bude měnit efektivita jednotlivých algoritmu s rostoucí hustotou (počtem hran) grafu pro fixní počet uzlů (200).

Měření byla prováděna na procesoru AMD Ryzen 7 3800XT (3.9 GHz) s 8 jádry a 16 vláky a 32 GB RAM.

Experimenty se zaměřovaly na množství času a paměti spotřebované jednotlivými algoritmy. Tyto hodnoty jsou zobrazeny v následujících tabulkách, kde jsou uvedeny pouze zkratky pro jednotlivé algoritmy (NX pro algoritmus knihovny Networkx, BF pro Brute-Force algoritmus, WEIN pro Herbert Weinblattův algoritmus, HJ pro algoritmus Hongbo Liu a Jiaxin Wanga).

Pokud zkoumaný algoritmus nedokázal vyhledat všechny cykly v zadaném orientovaném grafu do 3 minut, byl ukončen a další testování na obtížnějších grafech již nebylo prováděno (prázdné buňky tabulek).

## 8.1 Úplné grafy

V této sekci jsou porovnány reálné časové a paměťové nároky algoritmů výčtu cyklů pro úplné grafy. Úplné grafy představují nejkomplexnější grafovou strukturu a proto jsou často používané jako hlavní měřítko efektivity algoritmů pro výčet cyklů. Úplný orientovaný graf o $|V| =: n$ vrcholech obsahuje $\sum_{i=1}^{n} \frac{n!}{(n-i)!\cdot i}$ cyklů.

Lze vidět, že čas spotřebovaný algoritmem BF prudce roste pro úplné grafy s více než 10-ti vrcholy. Pro graf o 11-ti vrcholech by algoritmus BF potřeboval více než 180 sekund k nalezení všech cyklů. Naměřená paměťová náročnost tohoto algoritmu je pro úplné grafy ekvivalentní s ostatními algoritmy.

Časová složitost algoritmu WEIN roste v porovnání s ostatními algoritmy mnohem rychleji. Úplné grafy obsahující 8 vrcholů již není možné algoritmem WEIN prohledat v časovém limitu tří minut.

Z tabulky lze vidět, že časová složitost NX a HJ je pro úplné grafy podobná, i když poměř časů HJ a NX pro úplný graf o 5-ti vrcholech je 0.48, ale pro graf s 11-ti vrcholy již 0.66, z čehož lze pozorovat, že od určitého počtu vrcholů/cyklů začíná časová složitost algoritmu HJ růst rychleji než NX. Obdobné tvrzení lze vyvodit i pro paměťové nároky algoritm HJ.

| | | Čas [s] | | | | Paměť [MiB] | | | |
|---|---|---|---|---|---|---|---|---|---|
| $\|V\|$ | # cyklů | NX | BF | WEIN | HJ | NX | BF | WEIN | HJ |
| 1 | 1 | 0.23 | 0.11 | 0.11 | 0.11 | 68.6 | 40.0 | 40.0 | 40.1 |
| 2 | 3 | 0.23 | 0.11 | 0.11 | 0.11 | 68.3 | 39.3 | 40.0 | 39.9 |
| 3 | 8 | 0.23 | 0.11 | 0.11 | 0.11 | 68.4 | 39.3 | 39.9 | 39.9 |
| 4 | 24 | 0.23 | 0.11 | 0.11 | 0.11 | 68.7 | 40.0 | 40.0 | 39.9 |
| 5 | 89 | 0.23 | 0.12 | 0.11 | 0.11 | 68.5 | 40.0 | 40.0 | 39.9 |
| 6 | 415 | 0.23 | 0.12 | 0.14 | 0.11 | 68.4 | 40.0 | 39.6 | 39.9 |
| 7 | 2'372 | 0.25 | 0.16 | 1.54 | 0.13 | 68.6 | 40.3 | 40.1 | 40.2 |
| 8 | 16'072 | 0.35 | 0.45 | $> 180$ | 0.19 | 70.1 | 42.2 | $> 42.6$ | 41.9 |
| 9 | 125'673 | 1.25 | 3.12 | | 0.77 | 85.8 | 59.2 | | 59.8 |
| 10 | 1'112'083 | 10.02 | 29.1 | | 6.56 | 228.1 | 236.6 | | 239.6 |
| 11 | 10'976'184 | 104.52 | $> 180$ | | 69.04 | 1'741.6 | $> 1'656.8$ | | 2'158.7 |

Tabulka 8.1: Spotřebovaný čas a paměť při prohledávání úplných grafů s rostoucím počtem vrcholů. Výpočty probíhající více než 180 sekund byly ukončeny.

## 8.2 Multi-cyklické grafy

Pro testování algoritmů vyhledávání všech cyklů v řídkých orientovaných grafech byly vygenerované multi-cyklické grafy s o 200 vrcholech a 200 až 250 hranách (simulace rostoucí hustoty grafu). Tyto grafy jsou k dispozici ve složce `/graphs/multicycle`.

| | | Čas [s] | | | | Paměť [MiB] | | | |
|---|---|---|---|---|---|---|---|---|---|
| $\|E\|$ | # cyklů | NX | BF | WEIN | HJ | NX | BF | WEIN | HJ |
| 200 | 1 | 0.24 | 0.20 | 0.12 | 0.17 | 69.2 | 40.6 | 40.6 | 40.8 |
| 205 | 7 | 0.26 | 0.18 | 0.12 | 0.16 | 68.8 | 40.7 | 41.0 | 40.8 |
| 210 | 30 | 0.24 | 0.29 | 0.12 | 0.24 | 69.1 | 40.9 | 40.7 | 41.0 |
| 215 | 54 | 0.25 | 0.32 | 0.12 | 0.26 | 69.0 | 40.4 | 40.9 | 41.2 |
| 220 | 143 | 0.28 | 0.71 | 0.14 | 0.56 | 69.2 | 40.3 | 40.8 | 42.6 |
| 225 | 350 | 0.37 | 1.36 | 0.27 | 1.03 | 69.2 | 40.9 | 41.1 | 45.6 |
| 230 | 2'494 | 0.69 | 3.67 | 9.88 | 2.78 | 71.1 | 42.9 | 45.4 | 55.9 |
| 235 | 6'027 | 0.95 | 4.85 | 46.23 | 3.73 | 75.0 | 46.7 | 52.7 | 61.6 |
| 240 | 7'339 | 1.98 | 11.27 | 137.43 | 8.91 | 76.1 | 47.9 | 55.6 | 88.4 |
| 245 | 17'809 | 5.74 | 30.97 | $> 180$ | 24.12 | 91.3 | 65.3 | $> 64.0$ | 181.8 |
| 250 | 119'589 | 23.59 | $> 180$ | | $> 180$ | 177.6 | $> 96.9$ | | $> 1'441.5$ |

Tabulka 8.2: Spotřebovaný čas a paměť při prohledávání multi-cyklických grafů pro $|V| = 200$ s rostoucím počet hran. Výpočty probíhající déle než 180 sekund byly ukončeny.

Poměr spotřebovaného času algoritmem BF k ostatním algoritmům byl pro multi-cyklické grafy podobný jako pro úplné grafy. Z tabulky lze vypozorovat, že paměťová obtížnost algoritmu BF rostla pomaleji než u algoritmů WEIN a HJ, ale rychleji než pro algoritmus NX.

Znovu bylo potvrzeno, že časová složitost algoritmu WEIN roste rychleji než u ostatních algoritmů. Pro multi-cyklický graf s 200 vrcholy, 225 hranami a 350 cykly byl čas výpočtu algoritmu WEIN 0.27 sekund a algoritmu NX 0.37 sekund, ale pro graf s 200 vrcholy, 240 hranami a 7'339 cykly byl již čas výpočtu algoritmu WEIN 137.43 sekund a algoritmu NX pouhých 1.98 sekund.

Pro úplné grafy, sice časová složitost algoritmu HJ rostla rychleji než NX, ale pro grafy do 11-ti vrcholů byl algoritmus HJ rychlejší. Toto tvrzení pro multi-cyklické grafy, které simulují řidké grafy, neplatí. Zde rostou časové nároky algoritmu HJ velice rychle. Pro porovnání: algoritmus NX nalezne všechny cykly v multi-cyklickém grafu o 200 vrcholech, 235 hranách a 6'027 cyklech za 0.95 sekund a algoritmus HJ za 3.73 sekund, ale pro 200 vrcholů, 250 hran a 119'589 cyklů, které algoritmus NX nalezne za 23.59 sekund, již algoritmus HJ nestihne doběhnout v časovém limitu tři minut. Tento problém algoritmu HJ pramení z využívání fronty, ve které jsou uloženy všechny rozpracované cesty, což vede na opakované prohledávání jednotlivých vrcholů. Další nevýhodou fronty je její vysoká (zbytečná) paměťová náročnost v případě nízkéno počtu cyklů v poměru k počtu vrcholů. Problém vysokých paměťových nároků lze vidět v případě multi-cyklického grafu s 200 vrcholy, 250 hranami a 119'589 cykly. V tomto případě algoritmus NX využil při hledání cyklů maximálně 177.6 MiB paměti, ale algoritmus HJ, který ani nestačil doběhnout, již spotřeboval 1'441.5 MiB paměti.

# Kapitola 9

# Závěr

Tento projekt implementuje a porovnává tři různé algoritmy pro výčet všech cyklů v orientovaných grafech.

Nejpřímočařejší Brute-Force (BF) algoritmus prohledávající celý graf do hloubky má časovou složitost $\mathcal{O}(d^{|V|-d} \cdot d!)$ a prostorovou složitost $\mathcal{O}(|V|^2)$.

Algoritmus Herberta Weinblatta (WEIN) vyhledávající cykly pomocí kombinace již nalezených cyklů má časovou složitost $\mathcal{O}(|V|^2 + |E| \cdot |V| \cdot (c+1))$ a prostorovou složitost $\mathcal{O}(|V|^2)$, kde $c$ je počet cyklů.

Posledním algoritmem je algoritmus, kerý publikovali Hongbo Liu a Jiaxin Wang (HJ). Tento algoritmus využívá frontu, do které generuje všechny cesty v grafu. Časová složitost tohoto algoritmus je $\mathcal{O}(d^{|V|-d} \cdot d!)$ a prostorová složitost je $\mathcal{O}(d^{|V|-d} \cdot d!)$, kde $d$ je maximální výstupní stupeň v grafu.

Během experimentálních měření byly algoritmy porovnány s Johnsonovým přístupen implementovaným v knihovně Networkx. Ukázalo se, že pro úplné grafy je algoritmus HJ nejefektivnějším. Zato pro multi-cyklické grafy byl algoritmus HJ "mnohonásobně"pomalejší než Johnsonův. Tím bylo potvrzeno, že Johnsonův algoritmus je opravdu nejefektivnějším a nejvšestrannějším algoritmem pro vyhledávání cyklů.

# Literatura


[1] Allen, F. E. Program optimization. *Research Report RC-1959*. IBM Watson Research Center, Yorktown Heights, N.Y. april 1966.

[2] Deo, N. *Graph Theory with Applications to Engineering and Computer Science*. Dover Publications, 2017. ISBN 9780486820811. Dostupné z: https://books.google.cz/books?id=DSBMDgAAQBAJ.

[3] Johnson, D. B. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*. 1975, sv. 4, č. 1, s. 77–84. DOI: 10.1137/0204007. Dostupné z: https://doi.org/10.1137/0204007.

[4] Křivka, Z. a Masopust, T. *Grafové algoritmy*. VUT Brno, Fakulta informačních technologií, 2018. Dostupné z: http://www.fit.vutbr.cz/study/courses/GAL/public/gal-slides.pdf.

[5] Liu, H. a Wang, J. A new way to enumerate cycles in graph. In: *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*. 2006, s. 57–57. DOI: 10.1109/AICT-ICIW.2006.22.

[6] Médard, M. a Lumetta, S. Network Reliability and Fault Tolerance. In:. Duben 2003. DOI: 10.1002/0471219282.eot281. ISBN 9780471219286.

[7] Roberts, S. M. a Flores, B. Systematic Generation of Hamiltonian Circuits. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. sep 1966, sv. 9, č. 9, s. 690–694. DOI: 10.1145/365813.365842. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/365813.365842.

[8] Rozenfeld, H. D., Kirk, J. E., Bollt, E. M. a Avraham, D. ben. Statistics of cycles: how loopy is your network? *Journal of Physics A: Mathematical and General*. IOP Publishing. may 2005, sv. 38, č. 21, s. 4589–4595. DOI: 10.1088/0305-4470/38/21/005. Dostupné z: https://doi.org/10.1088%2F0305-4470%2F38%2F21%2F005.

[9] Rushdi, A. a Alsogati, A. Matrix Analysis of Synchronous Boolean Networks. *International Journal of Mathematical, Engineering and Management Sciences*. Duben 2021, sv. 6, s. 598–610. DOI: 10.33889/IJMEMS.2021.6.2.036.

[10] Tiernan, J. C. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. dec 1970, sv. 13, č. 12, s. 722–726. DOI: 10.1145/362814.362819. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/362814.362819.

[11] Weinblatt, H. A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph. *J. ACM*. New York, NY, USA: Association for Computing Machinery. jan 1972, sv. 19, č. 1, s. 43–56. DOI: 10.1145/321679.321684. ISSN 0004-5411. Dostupné z: https://doi.org/10.1145/321679.321684.