

# Porovnání algoritmů hledání cyklů v grafech

Bc. Jan Bíl  
Bc. Michal Šedý

15. listopadu 2022

# Obsah

<b>1</b>	<b>Úvod</b>	<b>3</b>
<b>2</b>	<b>Prerekvizity</b>	<b>4</b>
2.1	Orientovaný graf . . . . .	4
2.2	Prohledávání do hloubky . . . . .	5
2.3	Topologické uspořádání . . . . .	6
2.4	Zanedbání stavů . . . . .	6
<b>3</b>	<b>Brute-force</b>	<b>8</b>
3.1	Popis algoritmu . . . . .	8
3.2	Časová složitost . . . . .	9
3.3	Prostorová složitost . . . . .	9
<b>4</b>	<b>Herbert Weinbaltt</b>	<b>11</b>
4.1	Popis algoritmu . . . . .	11
4.2	Časová složitost . . . . .	14
4.3	Prostorová složitost . . . . .	14
<b>5</b>	<b>Hongbo Liu a Jiaxin Wang</b>	<b>16</b>
5.1	Popis algoritmu . . . . .	16
5.2	Časová složitost . . . . .	17
5.3	Prostorová složitost . . . . .	17
<b>6</b>	<b>Návrh programu</b>	<b>19</b>
6.1	Moduly . . . . .	19
6.1.1	digraph . . . . .	19
6.1.2	parser . . . . .	20
6.1.3	algorithms . . . . .	20
6.2	Použité knihovny . . . . .	21
<b>7</b>	<b>Použití programu</b>	<b>22</b>
7.1	Struktura projektu . . . . .	22
7.2	Instalace závislostí . . . . .	22
7.3	Spuštění programu . . . . .	22
7.4	Formát vstupu . . . . .	23
7.5	Formát výstupu . . . . .	23
7.6	Příklady spuštění . . . . .	24
7.7	Automatické testy . . . . .	24

<b>8 Experimenty</b>	<b>25</b>
8.1 Úplné grafy . . . . .	25
8.2 Multicyklické grafy . . . . .	26
<b>9 Závěr</b>	<b>27</b>
<b>Literatura</b>	<b>28</b>

# Kapitola 1

## Úvod

Orientovaný graf je struktura popisující množinu bodů (uzlů), jenž 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  $\dots u_m u_n \dots$  platí, že existuje orientovaná hrana vedoucí z uzlu  $u_m$  do uzlu  $u_n$ . Pato práce se zabývá popisem algoritmů pro získání seznamu všech existujících cyklů v zadaném grafu.

Vyhledávání všech (výčet) cyklů v grafu je využíváno v mnoha odvětvích teorie grafů. Tato informace je používána k optimalizaci počítačových programů [1], při analýze booleovských sítí využívaných pro modelování biologických sítí nebo sítí genových regulátorů [8], při návrhu, vývoji [7] nebo ověření spolehlivosti a fault-tolerance komunikačních systému [6], atd.

Tato práce porovnávající tři algoritmy pro výčet všech cyklů v grafu byla vytvořena v rámci projektu "Porovnání - Hledání cyklů" do předmětu GAL (grafové algoritmy). Text na úvod definuje potřebné pojmy dále využívané v algoritmech. V následujících kapitolách jsou uvedeny jednotlivé implementované algoritmy. Kapitola 3 popisuje přímočarý algoritmus [2, str. 287], který postupně generuje různé kandidáty cest, a ti jsou následně ověřováni. Algoritmus, který navrhl Herbert Weinblatt využívající zpětné navrácení [10] je uveden v kapitole 4. Kapitola 5 popisuje algoritmus Hongbo Liu a Jiaxin Wangův algoritmus využívající frontu [5]. Tyto algoritmy byly implementovány v jazyce Python3. Popis návrhu implementace aplikace a její používání jsou uvedeny v kapitolách 6 a 7. Experimenty porovnávající efektivitu jednotlivých postupů výčtu všech cyklů včetně grafové knihovny Networkx<sup>1</sup> jsou uvedeny v kapitole 8.

---

<sup>1</sup>Dostupné z <https://networkx.org/>

## Kapitola 2

# Prerekvizity

Tato kapitola poskytuje základní definice pro orientované grafy, jakými jsou základní definice grafu, sledu, cesty a cyklů. Dále jsou popsány základní algoritmy pro práci s grafy, kterými jsou prohledávání do hloubky (DFS) a topologické uspořádání, které jsou využívány pro zjednodušení výčtu cyklů grafů. 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).*

Orientovaný graf  $G = (V, E)$  je možno v algoritmech reprezentovat dvěma způsoby. Nechť  $u, v \in V$ . 1) jako pole  $Adj$  seznamů sousedů, pro které platí  $v \in Adj[u] \iff (u, v) \in E$ . 2) jako matici souslednosti  $Adj_M$ , kde  $Adj_M[u][v] = 1 \iff (u, v) \in E \wedge Adj_M[u][v] = 0 \iff (u, v) \notin E$ . Pro účely této práce byl zvolen první přístup, kterým je pole seznamů sousedů.

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

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

**Definice 2.1.4** ***Vstupní stupeň uzlu** je dán funkcí  $d_- : V \rightarrow \mathbb{N}_0$ , která udává počet přechodu vstupujících do 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, pro každý stav obsažený v cyklu musí platit, že jeho vstupní i výstupní stupeň je nenulový. Tyto uzly s nulovým stupněm mohou být v části přípravy algoritmů pro výčet cyklů zanedbány (odstraněny). Toto zanedbání uzlu může snížit hodnotu vstupních nebo výstupních stupňů uzlů incidentních s uzlem  $u$  na nulu. V takovém případě jsou dále rekurzivně 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 (otevřený cesta) je sled, ve kterém se neopakují uzly.*

**Definice 2.1.7** *Cyklus je cesta, ve které 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 strom prohledávání do hloubky.

**Definice 2.2.1** *Nechť  $G = (V, E)$  a  $\pi$  pole předchůdců, kde  $u \in \pi[v] \implies (u, v) \in E$ . Strom prohledávání do hloubky je  $G_\pi = (V, E_\pi)$ , 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    $color[v] \leftarrow GRAY$ 
3    $d[v] \leftarrow time \leftarrow time + 1$ 
4   for  $v \in Adj$  do
5     if  $color[v] = WHITE$  then
6       DFS-VISIT( $v$ )
7     end
8   end
9 end

10 for  $u \in V$  do
11    $color[u] \leftarrow WHITE$ 
12    $\pi[u] \leftarrow NIL$ 
13 end

14  $time \leftarrow 0$ 
15 for  $u \in V$  do
16   if  $color[u] = WHITE$  then
17     DFS-VISIT( $u$ )
18   end
19 end

20 return  $\pi, d, f$ 

```

---

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

*Důkaz.* Inicializační část 10–13 má časovou obtížnost  $\mathcal{O}(|V|)$ . Hlavní cyklus 15–19 je prováděn maximálně  $|V|$ -krát, tedy časová obtížnost je  $\mathcal{O}(|V|)$ . Funkce DFS-VISIT je spouště na pouze pro bílé uzly, tedy  $|V|$ -krát a cyklus v proceduře 4–8 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  $\mathcal{O}(|V| + |E|)$ .  $\square$

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

## 2.4 Zanedbání stavů

---

**Algorithm 3:** Zanedbání stavů

---

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

**Output:**  $G_{\text{simply}}$

- 1 **Procedure** Pruning( $G_p := (V_p, E_p)$ )
  - 2     **for**  $u \in \text{Topological-sort}(G_p)$  **do**
  - 3         **if**  $d_{p+}[u] = 0$  **then**
  - 4             **for**  $v \in \text{Adj}_p[u]$  **do**
  - 5                  $d_{p+}[v] \leftarrow d_{p+}[v] - 1$
  - 6             **end**
  - 7              $V_p.\text{remove}(u)$
  - 8         **end**
  - 9     **end**
  - 10 **end**
  - 11  $G_t \leftarrow G^T$  // Transponujeme graf  $G$
  - 12 Pruning( $G_t$ ) // Smažeme zanedbatelné stavy v grafu  $G_t$
  - 13  $G_{\text{simply}} \leftarrow G_t^T$  // Transponujeme graf  $G_t$
  - 14 Pruning( $G_{\text{simply}}$ ) // Smažeme zanedbatelné stavy v grafu  $G_{\text{simply}}$
  - 15 **return**  $G_{\text{simply}}$
-

Jak již bylo dříve řečeno, stavy, 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  $d_+$  tak, že budou objeveny nové stavy s nulovým vstupním nebo výstupním stupněm. K jejich kompletní eliminaci slouží následující algoritmus.

**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 prochází  $|V|$ -krát a vnitřní cyklus 4–6 se prochází  $|Adj[u]|$ -krát. Časová složitost procedury Pruning je tedy  $\mathcal{O}(|V| + |E|)$ , z čehož plyne, že časová složitost algoritmu zanedbání stavů je  $\mathcal{O}(|V| + |E|)$ .  $\square$



## Kapitola 3

# Brute-force

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

### 3.1 Popis algoritmu

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque.

---

**Algorithm 4:** Dummy Algorithm.

---

```
1 Do something.;
2 for  $k = 0, 1, 2, 3, \dots$  do
3   Do something.;
4   if  $x \geq y$  then
5     Do something.;
6   else
7     Do something.;
8     for  $j = 1, \dots, 10$  do
9       Do something.;
10    end
11    THIS IS THE SPOT WHERE I NEED THE PAGE BREAK.;
12    Do something.;
13  end
14 end
```

---

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

## 3.2 Časová složitost

**Teorém 3.2.1** *Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque.*

*Důkaz.* Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. □

## 3.3 Prostorová složitost

**Teorém 3.3.1** *Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque.*

*Důkaz.* Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. □

## Kapitola 4

# Herbert Weinbaltt

Tento algoritmus byl publikován v roce 1972 Herbertem Weinblattem [10]. Jeho základem je Tiernanův algoritmus [9] publikovaný o dva roky dříve, který vyhodnocuje každý cyklus pouze jednou, jednalo se tedy o teoreticky nejefektivnější algoritmus, ovšem za cenu vyšších paměťových nároků. Sam 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 jedno, 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 samotným popisem algoritmu se 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 nabyt v seznamu *TT*, uzel  $v$  se momentálně nachází v seznamu *TT*, nebo že se uzel  $v$  již nenachází v *TT*. 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, respektive 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  $P_2 = \langle v_3 v_4 \rangle$  budeme využívat 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  $\leftarrow$  EmptyList
3  toAddSave  $\leftarrow$  EmptyList
4  toAddToControl  $\leftarrow$  EmptyList
5  added  $\leftarrow$  EmptyList
6   $v \leftarrow \text{END}(\textit{Path})$ 
7  for  $\textit{cycle} \in L_{cycles}$  do
8     $\textit{tail} \leftarrow \text{TAIL}(v, \textit{cycle})$ 
9    if  $\textit{tail} = \emptyset \vee \textit{tail} \in L_{cycles}$  then
10     | continue
11   end
12    $\textit{cycleTails.append}(\textit{tail})$ 
13   if  $\exists v_k \in \textit{tail} : v_k \in \textit{Path}$  then
14     | continue
15   end
16    $\textit{cycleEnd} \leftarrow \text{END}(\textit{cycle})$ 
17   if  $S_v[\textit{cycleEnd}] = 2$  then
18     |  $\textit{toAddToControl.extend}(\text{CONCAT}(\text{True}, \textit{Path} + \textit{tail}))$ 
19     | continue
20   else
21     |  $\textit{newCycle} \leftarrow \langle \textit{cycleEnd} \rangle + \text{TAIL}(\textit{cycleEnd}, TT) +$ 
22       |  $\textit{Path} + \text{TAIL}(\text{END}(\textit{Path}), \textit{cycle})$ 
23     | if isRecursion then
24       | |  $\textit{toAddToControl.append}(\textit{newCycle})$ 
25     | else
26       | |  $\textit{toAddSave.append}(\textit{newCycle})$ 
27     | end
28   end
29   if isResursion then
30     | return toAddToControl
31   else
32     |  $L_{cycles.extend}(\textit{toAddSave})$ 
33     |  $\textit{added.extedn}(\textit{toAddSave})$ 
34     | for  $\textit{cycle} \in \textit{toAddToControl}$  do
35       | | if  $\textit{cycle} \notin \textit{added}$  then
36         | | |  $L_{cycles.append}(\textit{cycle})$ 
37         | | |  $\textit{added.append}(\textit{cycle})$ 
38       | | end
39     | end
40   end
41 end
```

---

---

```

42 Procedure EXAMINE( $v$ )
43   if  $S_v[v] = 0$  then
44      $S_v[v] \leftarrow 1$ 
45      $TT.append(v)$ 
46   else if  $S_v[v] = 1$  then
47      $L_{cycles}.append(\langle v \rangle + \text{TAIL}(v, TT) + \langle v \rangle)$ 
48   else
49      $\text{CONCAT}(\text{False}, [v])$ 
50 end

51 Procedure EXTEND
52   while  $TT \neq \emptyset$  do
53      $u \leftarrow \text{END}(TT)$ 
54      $possible\_v \leftarrow \{v \in V \mid (u, v) \in E \wedge S_e[u][v] = 0\}$ 
55     if  $possible\_v = \emptyset$  then
56        $S_v[u] \leftarrow 2$ 
57        $TT.removeLast()$ 
58     else
59        $v \leftarrow \text{PickOne}(possible\_v)$ 
60        $S_e[u][v] \leftarrow 2$ 
61       EXAMINE( $v$ )
62     end
63   end
64 end

// Inicializace globálních proměnných
65  $TT \leftarrow \text{EmptyList}$ 
66  $S_e[0 \dots n-1][0 \dots n-1] \leftarrow 0$  // nulová matice  $n \times n$ 
67  $S_v[0 \dots n-1] \leftarrow 0$ 

68 for  $v \in V$  do
69   if  $S_v[v] = 0$  then
70      $S_v[v] \leftarrow 1$ 
71      $TT.append(v)$ 
72     EXTEND()
73   end
74 end
75 return  $L_{cycles}$ 

```

---

Před spuštěním vlastního prohledávání grafu jsou všechny uzly, které mají hodnotu  $d_+$  nebo  $d_-$  rovnou nule a jejich hrany 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 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 uzel

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 se již byl v minulosti vyhodnocen, pak před zpětným navrácením zkontroluje, zda některá z doposud vyhodnocovaných hran tvoří nový cyklus. Pokud se uzel  $v$  nachází v  $TT$ , pak existuje právě jeden cyklus, který je tvořen sjednocením  $v$  s  $TAIL\ TT$  s respektem z uzlu  $v$ . Pokud se uzel  $v$  již nenachází v  $TT$ , pak může cyklus existovat pouze pokud byly již nějaké cykly obsahující  $v$  objeveny. V takovém případě je spuštěna rekursivní 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  $TAIL\ TT$  s respektem k  $u$ . (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  $v_3$ . A takovém případě je konkatencí 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  $\mathcal{O}((|V| + |E|) * (c + 1))$ , kde  $c$  je počet cyklů v grafu.*

*Důkaz.* 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 přechod  $(u, v) \in E$ , pro který je  $S_e[u][v] = 0$ . Pro takovýto přechod je následně nastaveno  $S_e[u][v]$  na hodnotu 2. Časová složitost bez procedury  $CONCAT$   $\mathcal{O}(|V| + |E|)$ .

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

Celková složitost algoritmu je tedy  $\mathcal{O}(|V| + |E| + c * |V| + c * |E|) = \mathcal{O}((|V| + |E|) * (c + 1))$ .  $\square$

## 4.3 Prostorová složitost

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

*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* můžeme

zanedbat, protože jejich data jsou obsažena v listu všech detekovaných cyklů  $L_{cycles}$ . Prostorová složitost listu  $L_{cycles}$  je  $\mathcal{O}(c * |V|)$ , protože obsahuje  $c$  cyklů<sup>1</sup>, kde délka každého cyklu může být až  $|V| + 1 \simeq |V|$ .

Pokud by prohledávaný graf neobsahoval žádný cyklus, potom budou vždy inicializovány globální pomocné struktury s prostorovou složitostí  $\mathcal{O}(|V|^2)$ .

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

---

<sup>1</sup>V případě úplného grafu je počet cyklů roven mohutnosti symetrické grupy  $|S_{|V|}| = |V|!$ .



## Kapitola 5

# Hongbo Liu a Jiaxin Wang

V roce 2006 spolu Hongo Liu a Jiaxin Wang publikovali algoritmus [5] pro výčet všech cyklů v grafu. Tento algoritmus pracuje v exponenciální časové složitosti oproti algoritmu Jonsona [3], Tiernana [9] nebo zmíněného Weinblattova algoritmu [10]. V porovnání s nimi 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  $k \in \mathbb{N}_0$  bez nutnosti nalezení všech cyklů.

### 5.1 Popis algoritmu

Algoritmus využívá pomocné funkce *END*, která již byla 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  $ord(v) > ord(END(P))$ .

Před spuštěním prohledávání grafu jsou všechny uzly, které mají hodnotu  $d_+$  nebo  $d_-$  rovnou nule a jejich hrany 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  $(END(P), HEAD(P))$ . Dále z cesty  $P$  na základě přechodů vedoucích z  $END(P)$  generuje nové cesty  $P + \langle v \rangle$  pro  $v \in \{v \in Adj[END(P)] \mid v \notin P \wedge ord(v) > ord(HEAD(P))\}$  a vkládá je 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  $Q \leftarrow EmptyQueue$ 
2 for  $v \in V$  do
3    $ENQUEUE(Q, \langle v \rangle)$ 
4 end

5 while  $Q \neq \emptyset$  do
6    $P \leftarrow DEQUEUE(Q)$ 
7    $head \leftarrow HEAD(P)$ 
8    $end \leftarrow END(P)$ 
9   for  $v \in Adj[end]$  do
10    if  $v = head$  then
11       $L_{cycles}.append(P + \langle v \rangle)$ 
12    else if  $ord(v) > ord(head)$  then
13      if  $v \notin P$  then
14         $ENQUEUE(Q, P + \langle v \rangle)$ 
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  $\mathcal{O}(2^{|V|})$ .

*Důkaz.* Časová složitost zanedbání uzlů je  $\mathcal{O}(|V| + |E|)$ . Inicializace a naplnění fronty (1–4)  $Q$  má časovou složitost  $\mathcal{O}(|V|)$ . V hlavním cyklu (5–18) jsou postupně vygenerovány všechny cesty  $P$ , pro které platí  $\forall v \in P : ord(v) \geq ord(HEAD(P))$ . Takovýchto otevřených cest je až  $\sum_{k=1}^{n:=|V|} \binom{n}{k} = 2^n - 1$ . Celková časová složitost algoritmu je tedy  $\mathcal{O}(2^{|V|})$ .  $\square$

## 5.3 Prostorová složitost

**Teorém 5.3.1** Prostorová složitost algoritmu, který publikovali Hongo Liu a Jiaxin Wang je  $\mathcal{O}(2^{|V|})$ .

*Důkaz.* Prostorová složitost výstupního seznamu  $L_{cycles}$  je  $\mathcal{O}(c * |V|)$ , protože délka cyklu může být až  $|V| + 1 \simeq 1$ . Do fronty  $Q$  se postupně ukládají všechny cesty v grafu, kde pro každý uzel  $v$  otevřené cesty  $P$  musí platit  $ord(v) \geq HEAD(P)$ . Algoritmem bude

celkem zpracováno až  $\sum_{k=1}^{n:=|V|} \binom{n}{k}$  otevřených cest. Hodnota této funkce dosahuje maxima pro  $k = \lceil n/2 \rceil =: k_{0.5}$ . Tedy prostorová složitost fronty  $Q$  je  $\mathcal{O}(\binom{n}{k_{0.5}}) \in \mathcal{O}(2^{|V|})$ .

Celková prostorová složitost algoritmu je  $\mathcal{O}(2^{|V|})$ . □

## Kapitola 6

# Návrh programu

Tato kapitola popisuje jednotlivé komponenty, z nichž sestává program demonstrující zmíněné algoritmy pro výčet všech cyklů v grafu. Veškerá implementace byla provedena v jazyce Python 3 (verze 3.8+).

### 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

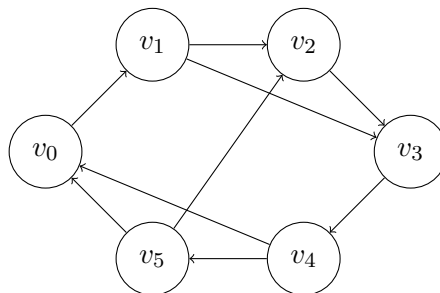
Třída grafů disponuje základními metodami pro práci s grafy, jakými jsou přidávání uzlů (`add_vertex`), přidávání hran (`add_edge`), transponování (`transpose`), získání indukovaného grafu (`get_induced_graph`), výpočet dfs lesa (`dfs`) nebo získání topologického uspořádání (`get_topological_sort`).

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 nested grafů (`create_nested_graph`). Tyto generované grafy budou použity pro získání experimentálních výsledků.

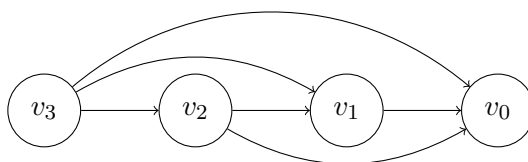
**Definice 6.1.1** *Nechť  $G = (V, E)$  je orientovaný graf.  $G$  se nazývá **úplný orientovaný graf**, pokud  $E = V \times V$ .*

**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  $|C| = |V| + 1$ .*



Obrázek 6.3: Multi-cyklický 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<sup>1</sup>.*



Obrázek 6.5: Nested 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 jeden přechod. Ten je reprezentován dvojicí vrcholů oddělených mezerou. Jména vrcholů mohou obsahovat pouze alfanumerické znaky.

Příklad formátu grafu:

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

### 6.1.3 algorithms

Modul `algorithms` obsahuje implementace jednotlivých algoritmů pro výčet cyklů v grafu. Pro porovnání je zde také použita knihovna `Networkx`. Jednotlivé implementace nejsou k dispozici přímo, ale přes volání funkce `get_cycles`, která na základě specifikovaného parametru `algo` vybere, zda bude použit algoritmus z knihovny `Networkx` (`algo="nx"`), brute-force algoritmus (`algo="bf"`), Herbert Weinbalttův algoritmus (`algo="wein"`), nebo algoritmus Hongo Liu a Jiaxin Wanga (`algo="hj"`).

<sup>1</sup>Produkce pouze dvojice prvků ležících na diagonále množinového produktu.

## 6.2 Použité knihovny

Program využívá knihovny Networkx (verze 2.8.7) pro porovnávání efektivity implementovaných algoritmů pro výčet všech cyklů v grafu, knihovnu Bidict (verze 0.22.0) implementující obousměrný slovník využíváný pro obousměrný překlad jmen uzlů na jejich pořadové čísla a knihovnu Numpy (verze 1.23.3) k reprezentaci matic. Tyto závislosti lze nainstalovat pomocí nástroje pip3.

Dále je využívána knihovna Argparse, pro zpracování argumentů příkazové řádky, která je součástí standardních 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<sup>1</sup>, Networkx<sup>2</sup> a Numpy<sup>3</sup>) využívaných programem. Kapitola také uvádí náповědu 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 podporovaných programem 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 jejich verze. Skript s automatickými testy pro modul `gal` je umístěn ve složce `/tests`.

### 7.2 Instalace závislostí

Potřebné knihovny využívané programem lze instalovat pomocí nástroje `pip3` do virtuálního prostředí následovně.

```
python3 -m venv gal-env
source ./gal-env/bin/activate
pip3 install -r requirements.txt
```

### 7.3 Spuštění programu

Při spuštění programu lze přepínači `znolit` 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 Wang, nebo algoritmem Herberta Weinblatta. Musí být zadán právě jeden přepínač určující algoritmus.

Dále lze zpracovávat graf zadaný vstupním souborem, vyhegerovaným úplným, multi-cyklickým, nebo neted grafem. Právě jeden graf musí být zadán.

---

<sup>1</sup>Bidict dostupné z: <https://bidict.readthedocs.io>

<sup>2</sup>Networkx dostupné z: <https://networkx.org>

<sup>3</sup>Numpy dostupné z: <https://www.numpy.org>

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

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

poziční argumenty:

<code>input</code>	Vstupní soubor s grafem.
--------------------	--------------------------

přepínače:

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

## 7.4 Formát vstupu

Vstupní soubor s příponou `.grph` obsahuje orientovaný graf. Každý řádek souboru reprezentuje jeden přechod. 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 posloupností vrcholu 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 příklady spuštění programu.

### Networkx + vstupní soubor

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

Program provede výčet všech 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  $|E| \rightarrow 10 + 5$  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 neted grafem  $G_n = (V, E)$ , kde  $|V| = 9$  s využitím Herbert Weinblattova algoritmu (5).

## 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  $|V| \in \langle 1, 6 \rangle$ , multy-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 vyšší přehlednosti je během testování tištěna informace o počtu provedených testů pomocí knihovny `tqdm`<sup>4</sup> (verze 4.64.1). Tato knihovna se souběžně instaluje se závislostmi pro program `enum_cycles.py`.

---

<sup>4</sup>Tqdm je 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, které jsou často v důsledku své vysoké obtížnosti používány jako hlavní měřítko efektivity algoritmů pro výčet cyklů.

Během implementace programu se ukázalo, že nested grafy jsou nevhodné pro porovnávání algoritmu (všechny algoritmy byly stejně efektivní). Z tohoto důvodu byly nested grafy použity pouze pro testování.

Posledním typem grafu, na kterém budou algoritmy testovány, je multi-cyklický graf. Tento graf simuluje prohledávání nad řídkými grafy obsahujícími cykly. Hlavním cílem tohoto porovnávání bude, jak se bude měnit efektivita jednotlivých algoritmu s rostoucí hustotou (počtem přechodů) grafu pro fixní počet uzlů.

Měření byla prováděna na procesoru AMD Ryzen 7 5800H (3.2 GHz) s 8 jádry a 16 vláky s 16 GB RAM.

V tabulkách jsou uvedeny popuze zkratky pro jednotlivé algoritmy (NX = algoritmus knihovny networkx, BF = Brute-Force algoritmus, WEIN = Herberta Weinblattův algoritmus, HJ = algoritmus Hongbo Liu a Jiaxin Wanga).

### 8.1 Úplné grafy

—		Čas [s]				Paměť [KB]			
V	# cyklů	NX	BF	WEIN	HJ	NX	BF	WEIN	HJ
1	—	—	—	—	—	—	—	—	—
2	—	—	—	—	—	—	—	—	—
3	—	—	—	—	—	—	—	—	—
4	—	—	—	—	—	—	—	—	—
5	—	—	—	—	—	—	—	—	—
6	—	—	—	—	—	—	—	—	—
7	—	—	—	—	—	—	—	—	—
8	—	—	—	—	—	—	—	—	—
9	—	—	—	—	—	—	—	—	—
10	—	—	—	—	—	—	—	—	—

Tabulka 8.1: Spotřebovaný čas a paměť při prohledávání úplných grafů s různou velikostí.

## 8.2 Multicyklické grafy

—		Čas [s]				Paměť [KB]			
$ E $	# cyklů	NX	BF	WEIN	HJ	NX	BF	WEIN	HJ
200	—	—	—	—	—	—	—	—	—
205	—	—	—	—	—	—	—	—	—
210	—	—	—	—	—	—	—	—	—
215	—	—	—	—	—	—	—	—	—
220	—	—	—	—	—	—	—	—	—
225	—	—	—	—	—	—	—	—	—
230	—	—	—	—	—	—	—	—	—
235	—	—	—	—	—	—	—	—	—
240	—	—	—	—	—	—	—	—	—
245	—	—	—	—	—	—	—	—	—
250	—	—	—	—	—	—	—	—	—

Tabulka 8.2: Maximální časový limit byl 120 sekund.

## Kapitola 9

### Závěr

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# 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] 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>.
- [8] 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.
- [9] 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>.
- [10] 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>.