Porovnání algoritmů hledání cyklů v grafech

Bc. Jan Bíl
Bc. Michal Šedý

15. listopadu 2022

Obsah

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 páce 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 ověřování. 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[1] jsou uvedeny v kapitole 8.

[1] 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 $v \in \mathit{Adj}[u] \iff (u, v) \in E$. 2) jako matici souslednosti Adj_M, kde $\mathit{Adj}_M[u][v] = 1 \iff (u, v) \in E \land \mathit{Adj}_M[u][v] = 0 \iff (u, v) \notin E$. 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 $d_+ : V \to \mathbb{N}_0$, která udává počet přechodu vstupujících do uzlu.*

Definice 2.1.4 ***Vstupní stupeň uzlu*** *je dán funkcí $d_- : V \to \mathbb{N}_0$, 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 obsažný 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 vstupních 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* π *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: π, 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             DFS–VISIT(v)
 7         end
 8     end
 9 end
10 for u ∈ V do
11     color[u] ← WHITE
12     π[u] ← NIL
13 end
14 time ← 0
15 for u ∈ V do
16     if color[u] = WHITE then
17         DFS–VISIT(u)
18     end
19 end
20 return π, 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 $\mathcal{O}(|V| + |E|)$. □

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

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

Teorém 2.3.2 *Protože výpočet topologického uspořádání využívá pouze DFS v časovou složitostí* $\mathcal{O}(|V| + |E|)$ *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_{simply}

Procedure Pruning($G_p := (V_p, E_p)$)
    for $u \in$ Topological-sort(G_p) **do**
        if $d_{p+}[u] = 0$ **then**
            for $v \in Adj_p[u]$ **do**
                $d_{p+}[v] \leftarrow d_{p+}[v] - 1$
            end
            $V_p.remove(u)$
        end
    end
end
$G_t \leftarrow G^T$ `// Transponujeme graf` G
Pruning(G_t) `// Smažeme zanedbatelné stavy v grafu` G_t
$G_{simply} \leftarrow G_t^T$ `// Transponujeme graf` G_t
Pruning(G_{simply}) `// Smažeme zanedbatelné stavy v grafu` G_{simply}
return G_{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 $\mathcal{O}(|V|+|E|)$. □

Kapitola 3

Brute-force

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

Algorithm 4: Dummy Algorithm.

```
Do something.;
for k = 0, 1, 2, 3, ... do
    Do something.;
    if x ≥ y then
        Do something.;
    else
        Do something.;
        for j = 1, ..., 10 do
            Do something.;
        end
        THIS IS THE SPOT WHERE I NEED THE PAGE BREAK.;
        Do something.;
    end
end
```

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

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

Důkaz. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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 nabyl 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}

```
Procedure CONCAT(isRecursion, Path)
    // Inicializace lokálních proměnných
    cycleTails ← EmptyList
    toAddSave ← EmptyList
    toAddToControl ← EmptyList
    added ← EmptyList
    v ← END(Path)

    for cycle ∈ L_cycles do
        tail ← TAIL(v, cycle)
        if tail = ∅ ∨ tail ∈ L_cycles then
            continue
        end
        cycleTails.append(tail)
        if ∃v_k ∈ tail : v_k ∈ Path then
            continue
        end
        cycleEnd ← END(cycle)
        if S_v[cycleEnd] = 2 then
            toAddToControl.extend(CONCAT(True, Path + tail))
            continue
        else
            newCycle ←⟨cycleEnd⟩ + TAIL(cycleEnd, TT)+
                            Path + TAIL(END(Path), cycle)
            if isRecursion then
                toAddToControl.append(newCycle)
            else
                toAddSave.append(newCycle)
            end
        end
    end

    if isResursion then
        return toAddToControl
    else
        L_cycles.extend(toAddSave)
        added.extedn(toAddSave)
        for cycle ∈ toAddToControl do
            if cycle ∉ added then
                L_cycles.append(cycle)
                added.append(cycle)
            end
        end
    end
end
```

```
Procedure EXAMINE(v)
    if S_v[v] = 0 then
        S_v[v] ← 1
        TT.append(v)
    else if S_v[v] = 1 then
        L_cycles.append(⟨v⟩ + TAIL(v, TT) + ⟨v⟩)
    else
        CONCAT(False, [v])
end

Procedure EXTEND
    while TT ≠ ∅ do
        u ← END(TT)
        possible_v ← {v ∈ V | (u, v) ∈ E ∧ S_e[u][v] = 0}
        if possible_v = ∅ then
            S_v[u] ← 2
            TT.removeLast()
        else
            v ← PickOne(possible_v)
            S_e[u][v] ← 2
            EXAMINE(v)
        end
    end
end

// Inicializace globálních proměnných
TT ← EmptyList
S_e[0 . . . n − 1][0 . . . n − 1] ← 0                // nulová matice n × n
S_v[0 . . . n − 1] ← 0

for v ∈ V do
    if S_v[v] = 0 then
        S_v[v] ← 1
        TT.append(v)
        EXTEND()
    end
end
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 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 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 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* 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 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 $\mathcal{O}((|V| + |E|) * (c + 1))$, 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 $CONCAT$ $\mathcal{O}(|V| + |E|)$.

Nerekurzivní 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). Nerekurzivní volání procedury CONCAT má časovou složitost $\mathcal{O}(c * |E|)$. Procedura CONCAT může být volána rekurzivně pro každý vrchol nejvýše jednou kvůli podmínce (17) $S_v[cycleEnd] = 2$. Rekurzivní volání procedury CONCAT má časovou složitost $\mathcal{O}(c * |V|)$.

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

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[1], 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. □

[1] V případě úplného grafu je počet cyklů rovem 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 porovnáni 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 jejichž 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 $k \in \mathbb{N}_0$ be z 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 $v \in \{v \in Adj[END(P)] \,|\, v \notin P \wedge ord(v) > ord(HEAD(P))\}$ 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 ← 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 $\mathcal{O}(2^{|V|})$.*

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 $\mathcal{O}(2^{|V|})$. □

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 $\mathcal{O}(2^{|V|})$. $\square$

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 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

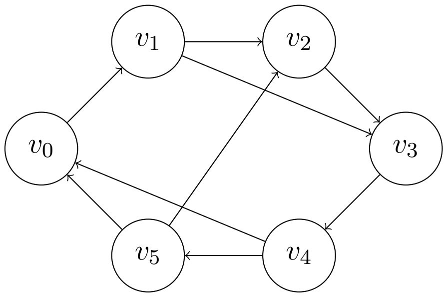
Třída grafů disponuje základními metodami pro práci s grafy, jakými jsou přidávání uzlů (`add_vertex`), přidávání hran (`add_edge`), transponování (`transpose`), získání indukovaného grafu (`get_induced_graph`), výpočet dfs lesa (`gfs`) nebo získání topologického uspořádání (`get_topological_sort`).

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.

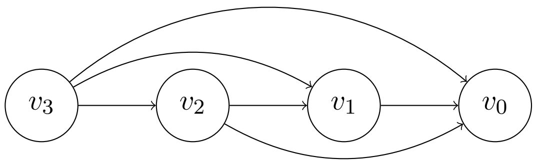
Definice 6.1.1 *Nechť $G = (V, E)$ je orientovaný graf. G se nazývá **úplný orientovaný graf**, pokud $E = V \times V$.*

Definice 6.1.2 *Nechť $G = (V, E)$ je orientovaný graf. G se nazývá **multi-cyclický 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 $E = \{(u, v) \mid ((u, m), (v, n)) \in (V \triangledown \mathbb{N})^2 \land n < m\}$, kde $\triangledown$ je diagonální produkt*[1].



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 dvojici 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 přípo, ale přez volání funkce `get_cycles`, která na základě specifikovaného prametru `algo` vybere, zda bude použit algoritmus z knihovny Networkx (`algo="nx"`), brute-force algorithmus (`algo="bf"`), Herbert Weinbalttův algoritmus (`algo="wein"`), nebo algoritmus Hongo Liu a Jiaxin Wanga (`algo="hj"`).

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

6.2 Použité knihovny

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

Dále je využívána knihovna Argparse, pro zpracování argumentů příkazové řádky, která je 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. Na konci kapitoly jsou uvedeny vybrané příklady jednotlivých možností spuštění.

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 dále nachází soubor `requirements.txt` obsahující využívané knihovny a jejich verze.

7.2 Instalace závislostí

Potřebné knihovny využívané programem lze instalovat pomocí vástroje 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 spuštení 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 Wanga, nebo algoritmem Herberta Weinblatta. Musí být zadán právě jeden přepínač určující algoritmus.

Dále lze zpracovávat graf zadaný vstupním souborem, vyhegerovaným úplným, multicyklickým, nebo neted grafem. Právě jeden 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 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:
 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 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 inplementována (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 přiklad 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).

Herberta 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).

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 používané jako hlavní měřítko efektivity algoritmů pro výčet cyklů.

Dále jsou algoritmy testovány na nested grafech, které neobsahují žádný cyklus, ale jsou velice husté a obsahují velké množství otevřených cest. Tento typ grafu je použit z důvodu skutečnosti, že mnoho časová i prostorová složitost elektivních algoritmů závisí na počtu cyklů v grafu. Proto vyvstává otázka, jak moc jsou jednotlivé algoritmy efektivní, pokud se počet cyklů v grafu blíží nule.

Posledním typem grafu, na kterém budou algoritmy testovány, je multi-cyklický graf. Tento graf simuluje prohledávání nad řidkými 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.

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.