Masarykova univerzita
Fakulta informatiky

Detekce cyklů v dynamických grafech

Bakalářská práce

Jaroslav Bendík

Brno, 2014

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Jaroslav Bendík

Vedoucí práce: prof. RNDr. Ivana Černá, CSc.

Shrnutí

Práce se zabývá problémem dosažitelnosti akceptujících cyklů v dynamických grafech, tedy v grafech, které jsou postupně modifikovány tak, že se do nich přidá, nebo se naopak odebere některá z existujících hran, a problém dosažitelnosti akceptujícího cyklu je rozhodován po každé takovéto změně. V práci je podrobně popsán a formálně definován tento problém. Dále je zde ukázáno triviální řešení pomocí algoritmu zanořeného prohledávání do hloubky (NDFS), kdy se po každé změně v grafu graf vždy znovu prohledá a problém dosažitelnosti akceptujícího cyklu je znovu rozhodnut. Postupně je poukazováno na různé příčiny neefektivnosti tohoto řešení a jsou navrhovány a zdůvodňovány možnosti, proč a jak je možné toto řešení optimalizovat. Ve výsledku jsou představeny dvě mírně odlišné optimalizované verze algoritmu NDFS pro řešení zkoumaného problému, jejichž vzájemná efektivita se liší podle typu zpracovávaného grafu a typu prováděných modifikací grafu.

Klíčová slova

dynamický graf, akceptující cykly v grafu, dosažitelnost v grafu, algoritmus prohledávání do hloubky, verifikace

Obsah

1 Úvod **1**
Práce na podobné téma 2

2 Formální definice problému a výchozí algoritmy **3**
2.1 Problém akceptujícího cyklu 3
2.2 Problém akceptujícího cyklu nad dynamickým grafem 3
2.3 Prohledávání grafu do hloubky 4
2.4 Zanořené prohledávání grafu do hloubky 8
2.5 Základní algoritmus 11

3 Optimalizovaný algoritmus **13**
3.1 Vliv operace *zpracování* 14
3.2 Navázání na předchozí běh prohledávání 14
3.3 Označení akceptujícího cyklu 20
3.4 Anotovaný graf 21
3.5 Operace *zpracování* nad anotovaným grafem 22
3.6 Výsledný optimalizovaný algoritmus 28

4 Alternatvní přístupy **31**
4.1 NDFS bez obnovy hodnot proměnných 31
4.2 Využití Tarjanova algoritmu 34

5 Závěr **35**

Literatura **36**

1 Úvod

Grafové algoritmy mají ve světě rozsáhlé využití a i když většina lidí ani netuší co to graf z matematického pohledu je, jelikož si pod tímto pojmem obvykle představí pouze kus koláče, tak jejich služby využívají každodenně aniž by si to uvědomovali. Typickým příkladem je využití interaktivní mapy na webu k nalezení nejkratší cesty mezi dvěma městy, jelikož se na pozadí při jejím hledání ve skutečnosti řeší problém nalezení nejkratší cesty v grafu, kde vrcholy v grafu představují města a hrany, jenž tyto vrcholy spojují, představují silnice. Stačí tedy pouze vymyslet, jak se dá problémová situace z reálného světa abstrahovat do podoby grafového problému a můžeme místo zdánlivě složité úlohy řešit mnohdy jednoduchý grafový problém.

Cílem této bakalářské práce bylo vymyslet efektivní algoritmus pro řešení problému dosažitelnosti akceptujících cyklů v dynamickém grafu, což je něco, co můžeme v reálném světě využít například při tvorbě umělé inteligence v odvětví robotiky. Problém dosažitelnosti cyklů v grafu se již několik let využívá v odvětví softwarové verifikace při ověřování, že systém, který má pevnou množinu možných stavů a množinu přechodů mezi těmito stavy, funguje správně. Pokud rozšíříme tento problém na prohledávání dynamického grafu, tedy grafu, ve kterém jsou postupně přidávány a nebo odebírány hrany, tak můžeme řešením tohoto problému ověřovat i správnost fungování dynamicky se měnícího systému, tedy právě například správnost chování robota s umělou inteligencí, který s každým svým pohybem rozvíjí své znalosti o okolním světe.

Vedoucí mé bakalářské práce mi bylo určeno, abych jako základ navrhovaného algoritmu využil algoritmus zanořeného prohledávání do hloubky [2], jenž se dá mimo jiné využít právě pro detekci akceptujících cyklů v grafu. Triviální způsob, jakým se dá tento algoritmus využít i pro práci s dynamickými grafy, je opakovaně pomocí něj prohledávat graf po každé změně, která je v něm provedena. Jelikož však každá takováto změna představuje pouze změnu v existenci jedné hrany, tak se zřejmě průběh prohledávání nově vzniklého grafu nemusí moc lišit od průběhu prohledání pro předchozí graf a mělo by být možné nové prohledávání urychlit díky znalosti informací získaných při předchozích prohledávání. Při návrhu algoritmu jsem tedy vycházel z tohoto triviálního způsobu a hledal jsem možnosti jak ho optimalizovat.

V první fázi této bakalářské práce jsem zkoumal existující grafové algoritmy zabývající se dosažitelností akceptujících cyklů nebo alespoň částí tohoto problému, tedy dosažitelností vrcholů či existenci cyklů v grafu, a hledal nějaké zajímavé vlastnosti grafů, které bych při návrhu svého algoritmu mohl využít.

Během druhé fáze jsem formálně zadefinoval řešený problém, sestrojil pseudokód a popsal fungování zadaného triviálního řešení, a zkoumal možnosti, jak toto triviální řešení vylepšit.

Následovalo vytvoření dvou mírně odlišných variant optimalizovaného algoritmu, jejichž vzájemná časová náročnost se liší v závislosti na počtu a podobě prováděných

úprav v grafu. Uvedl jsem, a v případně potřeby dokázal, vlastnosti grafů, které byly nezbytné k ukázání korektnosti těchto optimalizací. Poslední fáze byla zaměřena na nástin dalších možných řešení zadaného problému, uvedení jejich výhod i nevýhod a porovnání s mnou navrženým algoritmem.

Hlavním cílem této práce bylo upravit zadaný algoritmus do takové podoby, která by byla svou časovou náročností efektivnější a zároveň by měla prostorovou složitost vyšší maximálně o konstantní nárůst a toto se povedlo. Výsledný algoritmus je až na extrémní případy rychlejší než původní triviální řešení.

Struktura písemné práce je rozdělena do tří základních částí. V první části je formálně definován řešený problém, je popsán algoritmus zanořeného prohledávání do hloubky, na jehož modifikaci je tato práce založena, a jsou uvedeny teoretické základy, ze kterých jsem vycházel. V druhé části je představen triviální algoritmus řešící zadaný problém, je poukázáno na hlavní příčiny jeho neefektivity a jsou postupně představeny kroky potřebné pro jeho optimalizaci; kód výsledného optimalizovaného algoritmu je uveden na konci této části. Závěrečná část práce obsahuje další dvě alternativní řešení, přičemž jedno je mírnou modifikací výsledného řešení a druhé nastiňuje možný přístup k problému bez využití vnořeného prohledávání do hloubky.

Práce na podobné téma

Při řešení problému dosažitelnosti akceptujícího cyklu v grafu se vlastně řeší dva podproblémy: problém dosažitelnosti a problém existence akceptujícího cyklu. Speciálně problémem dosažitelnosti nad dynamickým grafem se zabývali pánové Liam Roditty a Uri Zwick [7], kteří přišli s algoritmem, jehož amortizovaná časová náročnost na provedení aktualizace grafu patří do $\mathcal{O}(m + n * log(n))$ a časová složitost dotazu v nejhorším případě do $\mathcal{O}(n)$, kde m je současný počet hran a n je počet vrcholů v grafu.

Problém dosažitelnosti akceptujícího cyklu je často využíván při verifikování systému metodou ověřování modelu (model checking) s použitím lineární temporální logiky (LTL) [1], kdy se problém existence nesprávného chování systému dá převést právě na problém existence dosažitelného akceptujícího cyklu v grafu. Řešení tohoto problému ve spojitosti s LTL verifikací společně zkoumali Jaco Geldenhuys a Antti Valmari [6], namísto algoritmu zanořeného prohledávání do hloubky však využívají Tarjanův algoritmus [8] pro detekci silně souvislých komponent v grafu. Možnost využití tohoto přístupu je krátce diskutována v závěrečné kapitole této práce.

2 Formální definice problému a výchozí algoritmy

2.1 Problém akceptujícího cyklu

Je dán orientovaný graf $G = (V, E)$, počáteční vrchol v_0 a rozklad množiny vrcholů V na akceptující a neakceptující vrcholy. Akceptující cyklus v grafu je takový cyklus, který obsahuje alespoň jeden akceptující vrchol. Problém akceptujícího cyklu představuje rozhodnutí, zda graf obsahuje akceptující cyklus dosažitelný z počátečního vrcholu v_0.

2.2 Problém akceptujícího cyklu nad dynamickým grafem

Nechť $G(V, E)$ je orientovaný graf a nechť $H(h_0^{\pm}, ...h_n^{\pm})$ je posloupnost prvků ve tvaru $(a, b) : a, b \in V$, přičemž prvky s indexem h^+ budeme nazývat aditivními prvky a prvky s indexem h^- prvky subtraktivními.

Zpracováním prvku h z posloupnosti H budeme označovat transformaci grafu $G(V, E)$ na graf $\overline{G}(V, \overline{E})$, přičemž pro aditivní prvky platí $\overline{G} = (V, E \cup \{h\})$ a pro subtraktivní prvky platí $\overline{G} = (V, E - \{h\})$.

Řekneme, že pro graf G a posloupnost H je graf G v k-té iteraci, pokud proběhlo zpracování prvních k prvků z posloupnosti H. Graf G, který je v k-té iteraci, budeme dále označovat G_k.

Problémem akceptujícího cyklu nad dynamickým grafem rozumíme rozhodnutí, zda pro čtveřici $(G(V, E), v_0 \in V, H, i)$ na vstupu graf G_i obsahuje akceptující cyklus dosažitelný z vrcholu v_0.

2.3 Prohledávání grafu do hloubky

Prohledávání grafu do hloubky, zkráceně **DFS** [3] (z anglického Depth First Search) je jeden ze základních algoritmů pro prohledávání grafu. Pro zadaný graf $G = (V, E)$ a počáteční bod v_0 algoritmus systematicky nalezne všechny vrcholy, které jsou z v_0 dosažitelné.

V tomto algoritmu jsou vždy zkoumány hrany vedoucí z právě zpracovávaného vrcholu v a vybere se vždy taková, která vede do zatím neobjeveného vrcholu w a začne se jeho zpracovávání. Tento postup se opakuje tak dlouho, dokud z právě zpracovávaného vrcholu vede nějaká hrana do vrcholu, který ještě nebyl objeven. Jakmile z vrcholu w již žádná taková hrana nevede, tak je zpracovávání vrcholu w ukončeno a algoritmus pokračuje s výpočtem ve vrcholu, ze kterého byl vrchol w objeven. Celý výpočet začíná v počátečním vrcholu v_0 a končí ve chvíli, kdy z vrcholu v_0 nevedou žádné hrany do doposud neobjevených vrcholů. V tu chvíli jsou objeveny všechny vrcholy dosažitelné z v_0.

V každém okamžiku běhu prohledávání je vrchol označen buď za bílý, šedý anebo černý. Bílé vrcholy jsou takové, které se zatím nezačaly zpracovávat (nebyly tedy ještě objeveny). Šedé jsou takové, které již byly objeveny, ale jejich zpracovávání ještě nebylo ukončeno. Černé jsou takové vrcholy, jejichž zpracovávání bylo již ukončeno.

Každému vrcholu je ve chvíli, kdy je objeven, přiřazena časová známka, která určuje pořadí vrcholů, v jakém byly objeveny. Nejmenší časová známka je 0 a při objevení každého dalšího vrcholu se vždy zvětší o jedna. Platí tedy, že pokud je časová známka vrcholu v menší, než časová známka vrcholu w, tak byl vrchol v objeven dříve, než vrchol w.

Kromě přidělení časové známky je pro každý vrchol při jeho objevení určena relace rodič-potomek. Rodičem vrcholu w je vrchol v, ze kterého byl w objeven; zároveň povíme, že w je potomkem v. Rodiče vrcholu v budeme ukládat v proměnné $\pi[v]$, v případě, že rodič vrcholu není zatím určený, tak $\pi[v] = NIL$. Tranzitivní uzávěr této relace tvoří relaci předek-následník.

Algoritmus DFS také vytváří "depth-first tree" [3] , zkráceně **DFT**, s kořenem v_0, který obsahuje všechny vrcholy dosažitelné z v_0. DFT reprezentuje relaci rodič-potomek pro každý dosažitelný vrchol. Formálně řečeno, pro graf $G = (V, E)$ a počátení vrchol v_0, definujeme DFT grafu G jako jeho podgraf $G_\pi = (V_\pi, E_\pi)$, kde:

$$V_\pi = \{v \in V \ : \pi[v] \neq NIL\} \cup v_0$$
$$E_\pi = \{(\pi[v], v) : v \ \in V_\pi - \{v_0\}\}$$

Průběh prohledávání a tedy i podoba příslušného DFT je závislá na tom, v jakém pořadí prozkoumáváme jednotlivé hrany. Pokud by výběr hran probíhal náhodně, tak by pro graf G mohlo existovat několik navzájem neizomorfních DFT. Z tohoto

důvodu předpokládáme, že máme dané uspořádání na množině hran i na množině vrcholů a hrany pro prozkoumání volíme v závislosti na tomto uspořádání. Pro různé běhy algoritmu DFS na grafu G nám proto vzniknou vždy, až na izomorfismus, shodné DHT.

```
DFS(G(V,E), v_0)
1  for each vertex v ∈ V
2        do v.color ← white
3           π[v] ← NIL
4  timestamp ← 0
5  DFS-Visit(v_0)

DFS-Visit(v)
1  v.color ← gray
2  v.timestamp ← timestamp
3  timestamp ← timestamp + 1
4  for each (v,w) ∈ E
5        do if w.color = white
6              then π[w] ← v
7                   DFS-Visit(w)
8  v.color ← black
```

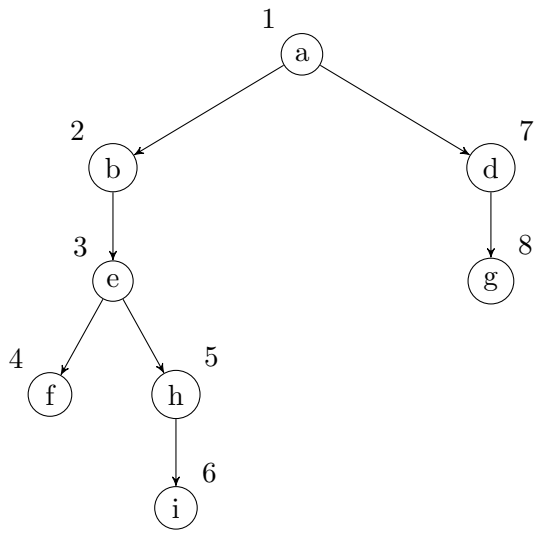
Obrázek 2.1 Základní algoritmus Prohledávání grafu do hloubky (DFS)

Procedura DFS pracuje následovně. Řádky 1 až 3 obarví všechny vrcholy na bílé a nastaví jim jako rodiče hodnotu NIL. Řádek 4 nastaví hodnotu globální proměnné pro časovou známku na 0. Řádek 5 spustí prohledávání pomocí procedury DFS-VISIT.

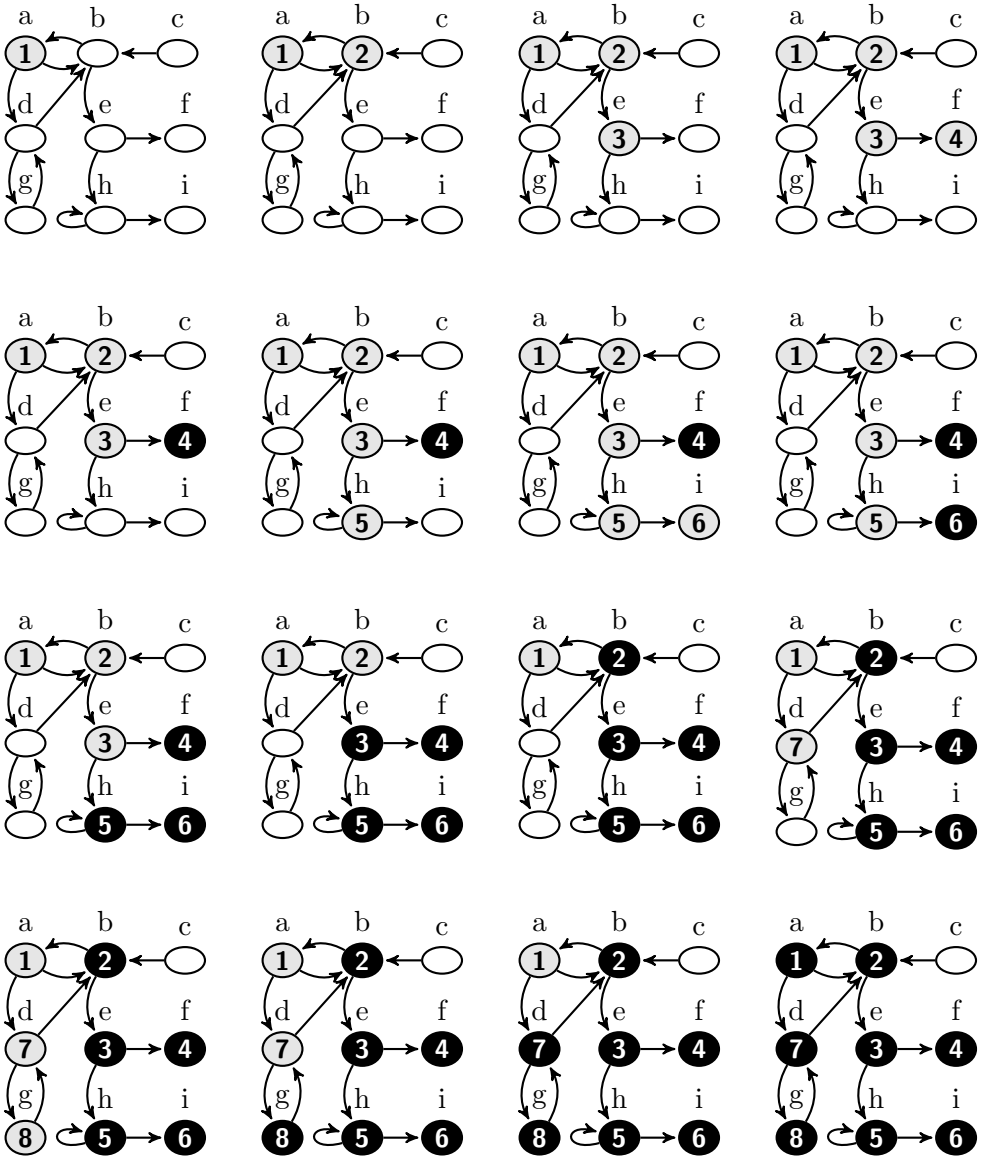
Procedura DFS-VISIT pracuje následovně. Řádek 1 obarví právě zpracovávaný vrchol na šedou barvu. Řádek 2 přiřadí vrcholu časovou známku. Řádek 3 inkrementuje hodnotu globální časové známky. Řádky 4 až 7 postupně zkoumají všechny hrany vedoucí z právě zpracovávaného vrcholu a pokud hrana vede do doposud neobjeveného (bílého) vrcholu, tak mu řádek 6 přiřadí rodiče a řádek 7 na něm spustí vnořenou proceduru DFS-VISIT. Řádek 8 obarví vrchol na černo ve chvíli, kdy končí jeho zpracovávání. Průběh prohledávání orientovaném grafu ukazuje obrázek 2.3.

Pro určení časové složitosti algoritmu si ještě jednou projděme postupně celý pseudokód. Řádky 1-2 v proceduře DFS zaberou čas $\mathcal{O}(V)$ a řádky 3-4 jsou konstantní operace. Procedura DFS-VISIT je prováděna pro každý navštívený vrchol právě jednou, jelikož se spouští pouze na bílých vrcholech a po začátku zpracování vrcholu ho hned obarví na šedou. Řádky 4-7 jsou pro každý vrchol zpracovány právě tolikrát, kolik z daného vrcholu vychází hran, celkově tedy všechna zpracování těchto řádků zabere čas $\mathcal{O}(E)$. Celková časová složitost tohoto algoritmu tak náleží do $\mathcal{O}(V + E)$.

Důkaz korektnosti algoritmu lze najít v [3].



Obrázek 2.2 Depth-first tree znázorňující průběh algoritmu DFS na grafu z obrázku 2.2. Čísla nad vrcholy ukazují časové známky vrcholů a tedy i čas, kdy byly přidány do DFT. Z obrázku je vidět, že pokud má vrchol více potomků, tak jsou seřazeny zleva doprava, v závislosti na jejich časové známce. Tohoto přístupu se budeme dále držet.



Obrázek 2.3 Průběh algoritmu prohledávání do hloubky na orientovaném grafu o devíti vrcholech, $\{a, b, ..., i\}$, s abecedním uspořádáním podle jejich názvu. Na hranách je definováno uspořádání po složkách (vrcholech). Nenavštívené vrcholy jsou znázorněné bílou barvou, vrcholy, které se již začaly zpracovávat, šedou barvou a vrcholy, jejichž zpracování bylo již ukončeno, jsou znázorněny černou barvou. Časové známky, určující pořadí, v jakém byly vrcholy objeveny, jsou vepsány dovnitř vrcholů.

2.4 Zanořené prohledávání grafu do hloubky

Pokud potřebujeme detekovat dosažitelné akceptující cykly v grafu, můžeme například využít tuto strategii: nejprve najdeme v grafu dosažitelný akceptující vrchol a poté zjistíme, jestli je tento vrchol součástí nějakého cyklu. Pro účely první fáze, tedy detekce akceptujícího vrcholu, můžeme využít již zmíněný algoritmus DFS. Ten totiž umožňuje projít všechny dosažitelné vrcholy v grafu, a proto je pomocí něj i možné najít všechny dosažitelné akceptující vrcholy. Pro účely druhé fáze, tedy ověření, zda je akceptující vrchol v obsažen v nějakém cyklu, lze využít opět algoritmus DFS. Tentokrát budeme zjišťovat, jestli je vrchol v dosažitelný sám ze sebe, což odpovídá tomu, býti součástí cyklu.

Budeme proto prohledávat graf pomocí algoritmu DFS a u každého vrcholu, těsně předtím, než ukončíme jeho zpracovávání, ověříme, zda-li se jedná o akceptující vrchol a pokud ano, tak spustíme vnořenou proceduru DFS a zjistíme, jestli je tento vrchol součástí akceptujícího cyklu.

Algoritmus zanořeného prohledávání do hloubky, zkráceně **NDFS** (z anglického Nested Depth First Search) pro zadaný orientovaný graf $G = (V, E)$, počáteční vrchol v_0 a rozklad množiny V na akceptující a neakceptující vrcholy, rozhodne, jestli G obsahuje akceptující cyklus dosažitelný z vrcholu v_0. Pokud ano, tak vrátí hodnotu *true*, v opačném případě hodnotu $false$. Jeho pseudokód ukazuje obrázek 2.4.

Funkce ACC-CYCLE-DETECTION pracuje následovně. Řádek 1 do návratové globální proměnné *detected* přiřadí hodnotu $false$, jelikož jsme žádný akceptující cyklus zatím nedetekovali. Řádky 2 až 5 obarví všechny vrcholy na bílé, nastaví jim jako rodiče hodnotu NIL a nastaví indikátor *.nested* pro vnitřní DFS proceduru. Řádek 6 nastaví hodnotu globální časové známky na 0. Řádek 7 spustí prohledávací proceduru NDFS-VISIT a řádek 8 vrátí výsledek tohoto prohledávání.

Procedura NDFS-VISIT pracuje obdobně jako procedura DFS-VISIT, která byla popsána na obrázku 2.1. Jediným rozdílem jsou řádky 8 a 9, které kontrolují, jestli je právě zpracovávaný vrchol akceptující a pokud ano, tak spustí proceduru na detekci cyklu.

```
ACC-CYCLE-DETECTION(G(V, E), v_0)
1  detected ← false
2  for each vertex v ∈ V
3        do v.color ← white
4           π[v] ← NIL
5           v.nested ← false
6  timestamp ← 0
7  NDFS-VISIT(v_0)
8  return detected

NDFS-VISIT(v)
 1  v.color ← gray
 2  v.timestamp ← timestamp
 3  timestamp ← timestamp + 1
 4  for each (v, w) ∈ E
 5        do if w.color = white
 6              then π[w] ← v
 7                   NDFS-VISIT(w)
 8  if v.isAcc
 9     then DETECT-CYCLE(v)
10  v.color ← black

DETECT-CYCLE(v)
1  v.nested ← true
2  for each (v, w) ∈ E
3        do if w.color = gray
4              then detected ← true
5                   EXIT()
6           elseif w.nested ≠ true
7           DETECT-CYCLE(w)
```

Obrázek 2.4 Algoritmus zanořeného prohledávání do hloubky (NDFS)

Procedura DETECT-CYCLE kontroluje, jestli je vrchol součástí cyklu a pracuje na již popsaném principu prohledávání do hloubky. Řádek 1 označí vrchol v v proměnné *v.nested* za navštívený. Řádky 2 až 7 postupně prozkoumají hrany (v, w) vedoucí z vrcholu v, řádek 3 zkontroluje, jestli byl nalezen cyklus, a pokud ano, tak upraví globální proměnnou *detected* a pomocí volání EXIT() **ukončí celé rekurzivní volání** procedury NDFS-VISIT. Pokud hrana vedoucí do vrcholu w neznamenala detekci akceptujícího cyklu, a pokud w ještě touto procedurou nebyl navštíven, tak pokračujeme jeho zpracováním.

Pozorování

(a) V průběhu procházení grafu vnější procedurou DFS se každý vrchol grafu nachází v danou chvíli právě v jednom ze tří stavů:

- Doposud nenavštívený, tedy bílý vrchol
- Vrchol, který již začal být zpracováván, tedy šedý vrchol
- Vrchol, jehož zpracování je již ukončeno, tedy černý vrchol

(b) Pro každý vrchol v každou chvíli platí, že má maximálně jednoho šedého potomka, jelikož potomci každého vrcholu jsou zpracováváni postupně

(c) Pro každý šedý vrchol platí, že jeho rodič je také šedý, jelikož zpracovávání vrcholu je ukončeno až ve chvíli, kdy je ukončeno zpracovávání všech jeho potomků

(d) Ve chvíli, kdy se kontroluje, jestli je právě zpracovávaný vrchol akceptující, tak už tento vrchol nemá žádné šedé potomky, jelikož bylo jejich zpracování již ukončeno

(e) Z (b) a (c) vyplývá, že v každou chvíli všechny aktuálně šedé vrcholy tvoří právě jednu cestu v grafu, počínající ve vrcholu v_0,

(f) a z (d) vyplývá, že ve chvíli, kdy se kontroluje, jestli je právě zpracovávaný vrchol v akceptující, tak tento vrchol tvoří poslední článek této cesty. Platí tedy, že z každého šedého vrcholu různého od v nutně existuje cesta do v

Důsledek na algoritmus Při prohledávání ve vnořené proceduře DETECT-CYCLE stačí k detekci cyklu pouze nalezení libovolného šedého vrcholu.

Pozorování

(g) Po skončení procedury NDFS-VISIT(v), volané v průběhu zpracovávání NDFS-VISIT(v_0), je zpracování všech následníků vrcholu v již ukončeno a nebyl nalezen žádný cyklus (kdyby byl, tak již byl výpočet ukončen). Platí tedy, že černý podgraf dosažitelný z vrcholu v neobsahuje akceptující cyklus.

(h) Jestliže podgraf dosažitelný z vrcholu v neobsahuje akceptující cyklus, tak žádný akceptující cyklus dosažitelný z vrcholu u, který leží mimo tento podgraf, neprochází vrcholem dosažitelným z v.

Důsledek na algoritmus Ve volání vnitřní procedury DETECT-CYCLE lze omezit zkoumání na takové vrcholy, které ještě nebyly zkoumány v žádném předchozím volání procedury DETECT-CYCLE.

Časová složitost inicializační funkce ACC-CYCLE-DETECTION jsme již určovali u algoritmu DFS, jediný rozdíl je přidání řádků 1 a 7, které mají konstantní časovou složitost. Složitost této procedury tedy patří do třídy $\mathcal{O}(V + E)$. Procedura NDFS-VISIT je prováděna pro každý vrchol maximálně jednou, jelikož se spouští pouze pro bílé vrcholy a každý takový vrchol je hned po spuštění procedury obarvený. Řádky 4 až 7 tvoří konstantní operace anebo volání další procedury a jsou pro každý vrchol zpracovány právě tolikrát, kolik hran z daného vrcholu vede, celkově tedy pro všechny hrany v grafu zabere toto zpracování čas v $\mathcal{O}(E)$. Procedura DETECT-CYCLE je prováděna pro každý akceptující vrchol právě jednou, řádek 1 má konstantní složitost, řádky 2 až 7 jsou konstantní operace a jsou spuštěny pro každou hranu v grafu maximálně jednou. Zpracování této procedury tedy zabere čas v $\mathcal{O}(E)$. Celková časová složitost tohoto algoritmu tedy patří do třídy $\mathcal{O}(V + E + E) = \mathcal{O}(V + E)$.

2.5 Základní algoritmus

Podle definice problému akceptujícího cyklu nad dynamickým grafem, kterou jsme uvedli v kapitole 2.2, máme na vstupu problému graf $G = (V, E)$, počáteční vrchol v_0, rozklad množiny V na akceptující a neakceptující vrcholy, posloupnost H a index i a chceme rozhodnout, zda graf G_i obsahuje akceptující cyklus dosažitelný z vrcholu v_0. V kapitole 2.4 jsme si ukázali algoritmus ACC-CYCLE-DETECTION, který řeší detekci akceptujících cyklů v G dosažitelných z vrcholu v_0. Potřebujeme tedy ještě algoritmus, který by *zpracovával* jednotlivé prvky posloupnosti H a dokázal tak po zpracování prvních i prvků transformovat graf $G \equiv G_0$ na graf G_i. Z definice víme, že *zpracování* prvku h buď tento prvek přidá, nebo odebere z množiny E a to v závislosti na tom, zda-li se jedná o aditivní, nebo subtraktivní prvek. Pseudokód procedury ALTER-GRAPH, která řeší *zpracování* jednoho prvku z posloupnosti H ukazuje obrázek 3.1. Jde na první pohled vidět, že tato procedura má konstantní časovou složitost.

```
ALTER-GRAPH(G(V, E), h)
1  if h.isSubtractive
2     then E ← E − h
3     else E ← E ∪ h
```

Obrázek 3.1 Procedura ALTER-GRAPH pro zpracování prvků posloupnosti

Nyní už máme k dispozici vše potřebné proto, abychom mohli představit algoritmus BASIC-DYM-ACC-CYCLE-DETECTION, který řeší problém akceptujícího cyklu nad dynamickým grafem. Vstupem algoritmu je trojice$(G(V, E), v_0, H)$, kde G je orientovaný graf zadaný množinou vrcholů V, s určeným rozkladem na akceptující a neakceptující vrcholy, a množinou hran E, v_0 představuje počáteční vrchol

a H je posloupnost prvků pro *zpracování*. Algoritmus nejprve pomocí funkce ACC-CYCLE-DETECTION rozhodne, zda iniciální graf G_0 obsahuje akceptující cyklus a následně začne zpracovávat jednotlivé prvky posloupnosti H, přičemž po zpracování každého prvku znova spustí funkci ACC-CYCLE-DETECTION. Výsledky volání této funkce ukládá do pole S, které slouží jako návratová hodnota po ukončení algoritmu. Platí tedy, že graf G_i obsahuje akceptující cyklus právě tehdy, když pole S na i-té pozici obsahuje hodnotu *true*.

```
BASIC-DYM-ACC-CYCLE-DETECTION(G(V,E), v_0, H)
1  S ← array()
2  S_0 ← ACC-CYCLE-DETECTION(G(V,E), v_0)
3  for (i = 1; i ≤ H.length; i = i + 1)
4        do ALTER-GRAPH(G(V,E), h_{i-1})
5           S_i ← ACC-CYCLE-DETECTION(G(V,E), v_0)
6  return S
```

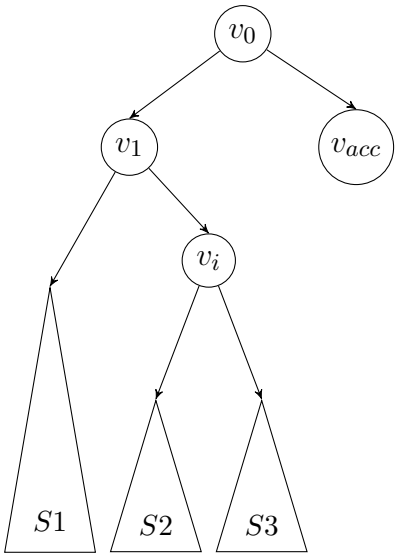
Obrázek 3.2 Algoritmus BASIC-DYM-ACC-CYCLE-DETECTION pro detekci akceptujících cyklů v dynamickém grafu

Časovou složitost BASIC-DYM-ACC-CYCLE-DETECTION určíme jednoduše, jelikož je z větší části složen z algoritmů, pro které jsme si již časovou složitost určili dříve. Řády 1 a 6 mají konstantní časovou složitost a o ALTER-GRAPH jsme si již ukázali, že má také konstantní časovou složitost. O ACC-CYCLE-DETECTION víme, že má složitost v $\mathcal{O}(V + E)$, přičemž na řádku 2 je vykonán jednou a na řádcích 3-5 tolikrát, kolik prvků má posloupnost H. Výsledná časová složitost algoritmu je tedy v $\mathcal{O}((k+1) * (V + E))$, kde k je počet prvků posloupnosti H.

Taková práce s dynamickým grafem má i dopad na prostorovou složitost. Jelikož algoritmus NDFS při svém průběhu mění informace o grafu s řídícím charakterem, jako jsou například barvy vrcholů či relace π, tak je třeba při začátku nového prohledávání tyto informace obnovit do iniciálního stavu. A jelikož by tato obnova byla časově náročná, tak budeme předpokládat, že máme zvlášť uloženou čistou kopii grafu s iniciálním ohodnocením těchto údajů, a pokaždé, když se v této práci budeme odkazovat na začátek nového prohledávání, tak tím budeme myslet prohledávání na této čisté kopii grafu.

3 Optimalizovaný algoritmus

V této kapitole si představíme efektivní algoritmus, který dokáže rozhodovat Problém akceptujícího cyklu v dynamickém grafu. Naivní algoritmus BASIC-DYM-ACC-CYCLE-DETECTION, představený v kapitole 2.5, pracuje tak, že pro každou iteraci začíná nové prohledávání grafu G. Intuitivně je však jasné, že přidáním, anebo odebráním, jedné hrany v grafu G_i, se průběh prohledávání pro nově vzniklý graf G_{i+1} nemusí moc lišit od průběhu prohledávání grafu G_i a že se tyto dva průběhy začnou lišit až ve chvíli, kdy má ona hrana poprvé nějaký vliv na prohledávání. Základní ideou optimalizace tohoto algoritmu je neopakovat zbytečně části prohledávání, které mají stejný průběh jako v předchozí iteraci, ale prohledávat třeba jen jistý podgraf celého grafu.



Obrázek 3.3 DHT z prohledávání grafu G_i, které skončilo nalezením akceptujícího cyklu při zkoumání vrcholu v_{acc}. Pokud byl v další iteraci tento cyklus narušen, tak bychom rádi například navázali na prohledávání grafu G_{i+1} až v bodě v_i a podgraf $S1$ znova neprohledávali.

Pro účely rychlejšího rozhodnutí o tom, zda graf G v i-té iteraci obsahuje akceptující cyklus, využijeme informace získané v předchozích iteracích; budeme si tedy kromě množin V, s rozkladem na akceptující a neakceptující vrcholy, a E, jenž určují graf v právě zpracovávané iteraci, a posloupnosti H pamatovat i nějaké další informace navíc. Je však nutné zvolit optimální množství těchto informací, protože pokud by jich bylo mnoho, tak by algoritmus sice mohl mít lepší časovou složitost než algoritmus BASIC-DYM-ACC-CYCLE-DETECTION, ale jeho prostorová náročnost by mohla být neadekvátně větší.

3.1 Vliv operace *zpracování*

Při pohledu na algoritmus DFS je zřejmé, že možný vliv na to, v jakém pořadí budou zpracovávány vrcholy při procházení grafu, může mít pouze pořadí, v jakém prozkoumáváme hrany vedoucí z právě zpracovávaných vrcholů a volba počátečního vrcholu prohledávání. Ostatní operace prováděné při prohledávání mají pouze značkovací charakter (obarvování vrcholů, označovaní vrcholů časovými známkami a ukládání rodičů jednotlivých vrcholů). My však máme počáteční vrchol pevně daný a v kapitole 2.3 jsme si uvedli, že požadujeme uspořádání na množině hran a že se tímto uspořádáním řídíme při výběru hran k prozkoumání; tento výběr tedy probíhá zcela deterministicky. Platí tedy, že průběh algoritmu DFS je vždy stejný.

Algoritmus zanořeného prohledávání do hloubky (NDFS) v takové podobě, jaké jsme ho uvedli, se od algoritmu DFS liší pouze přidáním operací, které nemají řídící charakter a tedy ani vliv na průběh algoritmu, a vnořené procedury DETECT-CYCLE. Tato procedura je však opět mírně modifikovanou verzí algoritmu DFS, byla do ní přidána kontrola barvy dosažitelného vrcholu a možnost okamžitého ukončení této procedury včetně celého rekurzivního volání procedury NDFS-VISIT, což je sice operací řídícího charakteru, ale nepřináší nedeterministické chování. Platí tedy, že i průběh algoritmu NDFS má při opakovaném spuštění na jednom grafu vždy stejný průběh.

Zpracování jednoho prvku $h = (u, v)$, které graf G_i upraví na graf G_{i+1}, už však může ovlivnit průběh algoritmu NDFS a pro nás je podstatné, v jakém okamžiku se tato úprava projeví. Je zřejmé, že absence anebo přítomnost nové hrany, může mít vliv až ve chvíli, kdy algoritmus s takovou hranou poprvé nějak pracuje a to nastává v okamžiku, kdy je zpracováván vrchol u mající časovou známku k. Do toho okamžiku je průběh prohledávání grafu G_{i+1} zcela identický tomu, který byl pro graf G_i, tedy zpracování vrcholů s časovou známkou menší než k proběhlo stejně. Naším cílem je navázat na předchozí prohledávání právě v okamžiku, kdy začal být zpracováván vrchol u anebo nějaký předek vrcholu u. Takový vrchol budeme nazývat **iniciální vrchol**; omezení na něj kladené a způsob jeho výběru ještě upřesníme později.

3.2 Navázání na předchozí běh prohledávání

Stav prohledávání je jednoznačně dán ohodnocením proměnných a zásobníkem volání funkcí [4], který uchovává informaci o aktuálním zanoření funkčního volání. V případě procedury ACC-CYCLE-DETECTION jsou těmito proměnnými indikátor existence akceptujícího cyklu *detected*, čítač časových známek *timestamp*, pole uchovávající relaci rodič-potomek pro jednotlivé vrcholy π, barvy, časové známky a indikátory *.nested* jednotlivých vrcholů.

Proto, abychom navázali na výpočet ve chvíli, kdy začal být zpracováván iniciální vrchol, nastavíme proměnné následovně:

- *detected* na $false$, jelikož ještě nebyl objeven akceptující cyklus

- čítač *timestamp* na časovou známku iniciálního vrcholu
- barvu na *white* u všech vrcholů, které mají větší časovou známku než iniciální vrchol
- indikátor *.nested* na $false$ u všech vrcholů, které mají větší časovou známku než iniciální vrchol
- časové známky těchto vrcholů a pro ně příslušné hodnoty v π upravovat nemusíme, jelikož je zaktualizujeme jakmile tyto vrcholy začneme znova zpracovávat a opětovné zpracování je umožněno výše uvedenou změnou barvy vrcholů. Abychom si však toto mohli dovolit, tak budeme muset omezit využívání těchto informací pouze na nebílé vrcholy.

První dvě úpravy jsme schopni provést v konstantním čase. Ty zbývající však mohou být časově náročnější a trochu komplikovanější. Všimněme si následujícího pozorování:

Pozorování

(i) Pokud je v šedý vrchol s časovou známkou k, tak všechny vrcholy s časovou známkou větší než k jsou z v dosažitelné a jsou jeho potomky

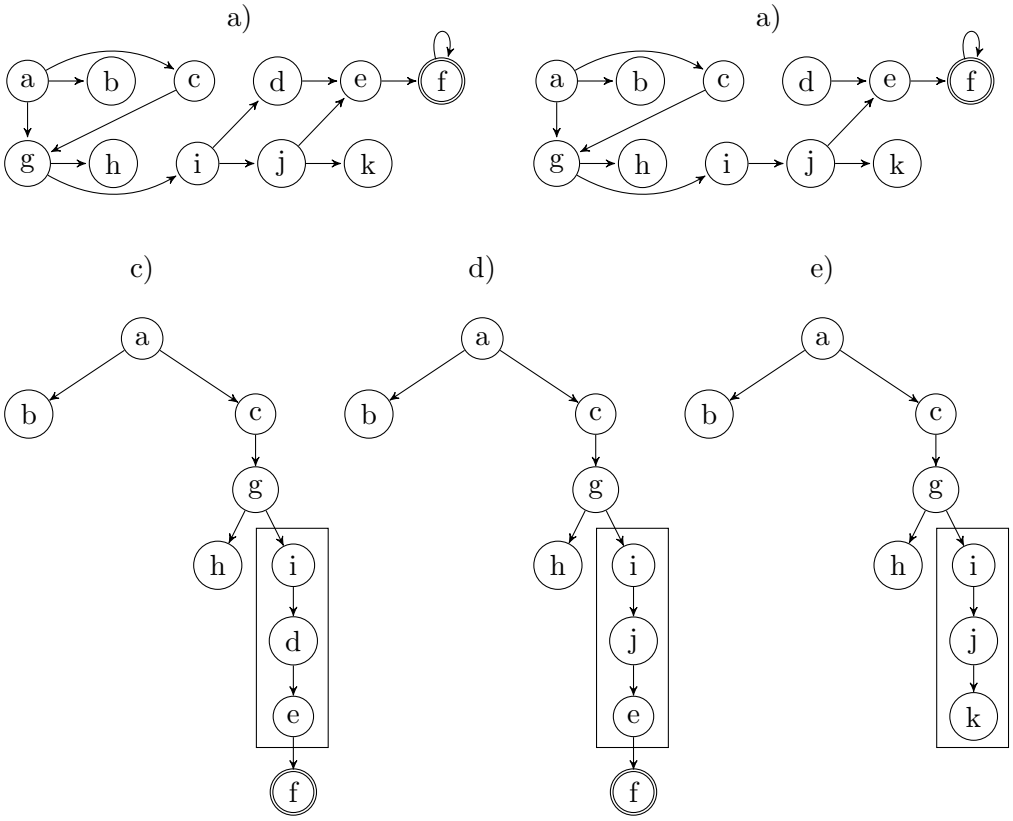
A my bychom právě chtěli upravit všechny vrcholy, které mají časovou známku větší než nějaké l, kde l je časová známka iniciálního vrcholu, a kvůli i) by pro nás bylo výhodné, aby všechny takové vrcholy byly dosažitelné z iniciálního vrcholu. Proto budeme vyžadovat, aby iniciální vrchol byl šedý vrchol anebo shodný s počátečním vrcholem v_0.

S tímto omezením jsme schopni najít všechny vrcholy, mající časovou známku větší nebo rovno než l, pomocí prohledávání do hloubky, s počátkem v iniciálním vrcholu, a každý vrchol při jeho zpracování náležitě upravit. Obrázek 3.4 ukazuje kód procedury CLEAN-VERTICES, která tyto úpravy provede.

```
CLEAN-VERTICES(v)
1  v.color ← white
2  π[v] ← NIL
3  for each (v, w) ∈ E
4        do if w.color ≠ white ∧ w.timestamp > v.timestamp
5              then CLEAN-VERTICES(w)
```

Obrázek 3.4 Procedura CLEAN-VERTICES pro vstup v prohledá podgraf vrcholu v a každému vrcholu z tohoto podstromu nastaví barvu na šedou a rodiče na NIL. Rozeznání vrcholů, které je třeba touto procedurou navštívit, probíhá pomocí časové známky a barvy vrcholu, přičemž časová známka vymezuje všechny vrcholy, které chceme upravit a barva v kombinaci s časovou známkou rozlišuje ty vrcholy, které už touto procedurou byly upraveny.

Časová náročnost této obnovy hodnot proměnných ovšem může být občas velmi vysoká, v nejhorším případě se může stát, že bychom obnovovali hodnoty všech vrcholů v grafu. Občas je tedy výhodnější rozhodnout se vůbec na předchozí prohledávání nenavazovat a začít prohledávat graf úplně znovu. Toto rozhodnutí založíme na tom, jaký je poměr mezi počtem vrcholů, jejichž hodnoty bychom takto obnovovali, a počtem vrcholů, které předcházely při minulém prohledávání iniciálnímu vrcholu. Počet vrcholů, které předcházeli iniciálnímu vrcholu, je rovný časové známce iniciálního vrcholu. Jelikož požadujeme, aby iniciální vrchol byl šedý vrchol a všechny vrcholy, jejichž hodnotu chceme upravit, byli jeho potomky, tak je počet těchto vrcholů roven rozdílu mezi hodnoty čítače *timestamp* − 1 (což je časová známka posledního objeveného vrcholu) a časovou známkou iniciálního vrcholu. Jestliže počet vrcholů k upravení převyšuje počet vrcholu předcházejících, tak začneme úplně nové prohledávání, v opačném případě navážeme na předchozí prohledávání.



Obrázek 3.5 ukazuje graf ve dvou po sobě následujících iteracích a příslušné DHT. Navíc ukazuje problém, který by vznikl, kdybychom navázali na průběh minulého prohledávání a neobnovili ohodnocení příslušných proměnných.

a) je graf G_i, b) je graf G_{i+1} vzniklý odebráním hrany (i, d), vrchol f tvoří akceptující cyklus. c) je DHT grafu G_i, d) je DHT grafu G_{i+1}. e) vznikne, pokud si jako iniciální vrchol zvolíme i,

mající časovou známku k, a nezahodíme změny provedené na vrcholech s vyšší časovou známkou než k. Vrcholu e tak totiž zůstane šedá barva a nebude již znova zpracován.

Obnovení zásobníku volání funkcí

Zásobník volání umožňuje, aby se z právě zpracovávané procedury α mohlo zavolat zpracování vnořené procedury β, vykonat celou proceduru β a po jejím ukončení se navrátit ke zpracovávání procedury α. Zásobník volání si proto tedy pamatuje dvě věci: vnější proceduru, ze které bylo zavoláno zpracování vnořené procedury, a instrukci ve vnější proceduře, kterou se mělo pokračovat.

V případě našeho algoritmu vzniká zanořené volání uvnitř vnější procedury NDFS-VISIT, v momentě, když se zkoumají dosažitelné vrcholy z právě zpracovávaného vrcholu. Pokud se najde nějaký dosažitelný vrchol, který ještě nebyl objeven, tak se na něj zavolá vnořená procedura NDFS-VISIT a o návrat a pokračování ve vnější proceduře se normálně stará zásobník volání. Pokud však začneme s naším výpočtem až zpracováváním iniciálního vrcholu, tak žádný zásobník volání existovat nebude a nebudeme schopni po ukončení zpracování iniciálního vrcholu pokračovat ve zpracovávání jeho předků.

Zásobník volání je systémová záležitost a my ho nemůžeme upravovat. Proto jeho činnost nasimulujeme. Řekli jsme, že zásobník volání si pamatuje proceduru, do které se vrátit a instrukci, kterou pokračovat, po ukončení vnořené procedury; stačí tedy zajistit tyto dvě věci.

Proceduru, do které se máme vrátit po ukončení vnoření procedury, zjistíme pomocí hodnoty v poli π. Ta totiž pro každý objevený vrchol v (vyjma v_0) uchovává informaci o tom, kdo je jeho rodič a tedy i to, při zpracovávání kterého vrcholu došlo k zavolání vnořené procedury pro jeho zpracování. Platí tedy, že NDFS-VISIT($\pi[v]$) zavolala proceduru NDFS-VISIT(v).

Instrukce v rámci vnější procedury, kterou máme pokračovat po ukončení zanořené procedury NDFS-VISIT, je vždy stejná, jelikož vnitřní proceduru voláme vždy na stejném místě. Jedná se o místo, ve kterém se zkoumají odchozí hrany z právě zpracovávaného vrcholu.

Jenomže zásobník volání si nejen pamatuje proceduru, do které se vrátit, a instrukci, kterou pokračovat, ale navíc "umožňuje" se k ní vrátit. To my ale neumíme; mohli bychom sice spustit znovu proceduru NDFS-VISIT($\pi[v]$), ale neumíme ji spustit až od určité instrukce. Proto místo této procedury zavoláme jinou, pomocnou proceduru, která obsahuje pouze ty instrukce, které se ještě měli vykonat.

Procedura COMPLETE-CALL pro vrchol $\pi[v]$ na vstupu naváže na zpracovávání vrcholu $\pi[v]$ ve chvíli, kdy by se normálně pokračovalo ve výpočtu při ukončení vnořené procedury pro vrchol v. Její kód je tvořen výňatkem z kódu procedury NDFS-VISIT a ukazuje ho obrázek 3.6.

Procedura CALL-STACK pro vrchol v na vstupu zpracuje v pomocí procedury NDFS-VISIT a dokončí zpracovávání všech předků vrcholu v a to za pomoci pos-

tupného volání procedury COMPLETE-CALL, její kód ukazuje obrázek 3.7. Pro nás bude tato procedura sloužit k navázání na průběh předchozího vyhledávání, na vstupu tedy bude iniciální vrchol.

```
COMPLETE-CALL(v)
1  for each (v, w) ∈ E
2        do if w.color = white
3              then π[w] ← v
4                   NDFS-VISIT(w)
5  if v.isAcc
6     then DETECT-CYCLE(v)
7  v.color ← black
```

Obrázek 3.6 Procedura COMPLETE-CALL navazuje na průběh procedury NDFS-VISIT v okamžiku, kdy se zkoumají odchozí hrany z právě zpracovávaného vrcholu.

```
CALL-STACK(v)
1  NDFS-VISIT(v)
2  v ← π[v]
3  while v ≠ NIL
4        do COMPLETE-CALL(v)
5           v ← π[v]
```

Obrázek 3.7 Procedura CALL-STACK nejprve zpracuje vstupní vrchol v a poté, pokud má vrchol v nějaké předky, tak postupně dokončí i jejich zpracovávání. Výpočet končí, když je dokončeno zpracovávání prvku, který nemá žádného rodiče, což platí pro počáteční vrchol v_0. Platí sice, že žádného rodiče nemají taky zatím neobjevené vrcholy, ale ty se v průběhu této procedury neobjeví, jelikož pracujeme pouze s vrcholy z pole π a to obsahuje pouze objevené vrcholy.

Obnovení hodnot *.nested* u vrcholů

Než si povíme, jak obnovit hodnoty *.nested* u jednotlivých vrcholů do stavu, v jakém byly při začátku zpracovávání iniciálního vrcholu, tak ještě upravíme způsob, jak budeme tyto hodnoty nastavovat. Doposud probíhalo nastavení iniciální hodnoty $false$ v rámci procedury ACC-CYCLE-DETECTION, my toto iniciální nastavení přesuneme až do procedury NDFS-VISIT. Pro zdůvodnění, proč to můžeme udělat, se podívejme na následující pozorování.

Pozorování

(j) Při přebarvování vrcholu v na černo v rámci procedury NDFS-VISIT jsou všechny vrcholy z něj dosažitelné buď označeny za jeho potomky a přebarveno na černo (pozorování (d)) anebo cesta k nim vede přes nějakého šedého předka vrcholu v, takže tato cesta, od šedého vrcholu dále, není v rámci zpracovávání vrcholu v zkoumána (tyto vrcholy jsou potomky některého předka vrcholu v)

(k) Vnitřní procedura DETECT-CYCLE(v) prohledává celý podgraf dosažitelný z vrcholu v, ale jelikož její průběh je ukončen ve chvíli, kdy narazí na první šedý vrchol (anebo neobjeví žádný šedý vrchol), tak platí, že zpracovává, kromě akceptujícího vrcholu, pouze černé vrcholy.

Jelikož podle (k) DETECT-CYCLE(v) pracuje pouze s černými vrcholy a černé vrcholy jsou obarvovány v rámci procedury NDFS-VISIT, tak nastavení výchozí hodnoty *.nested* pro tyto vrcholu můžeme provést až při jejich zpracovávání. To nám ušetří trochu času, jelikož doposud jsme tuto hodnotu nastavovali i pro nedosažitelné vrcholy a vrcholy, které možná ani nebudou zpracovány. Má to ale ještě jednu výhodu, nejprve se však podívejme na další pozorování.

(l) Z (k) plyne, že akceptující cyklus prochází maximálně dvěma šedými vrcholy (akceptujícím vrcholem a vrcholem, jehož nalezením je ukončeno hledání cyklu, což může být i onen akceptující cyklus). Ten z nich, který má nižší časovou známku, budeme označovat h.

(m) Akceptující cyklus je tvořen pouze následníky vrcholu h. Pokud by to tak nebylo a akceptující cyklus by obsahoval i nějaký černý vrchol c, který není následníkem vrcholu h, tak by musel mít c nutně vyšší časovou známku než h a h by byl dosažitelný z c, jelikož jsou oba součástí jednoho cyklu. Pak by ale h musel nutně být objeven v rámci zpracovávání c a tedy i jeho následníkem a jelikož je h šedý, tak by i c (podle (c)) musel být šedý a to je spor (s (k)).

Ve chvíli, kdy navazujeme na průběh předchozího prohledávání, můžeme vrcholy, jenž mají hodnotu *.nested* rovnu *true* (tedy ty vrcholy, které byly zpracovány v rámci nějakého běhu procedury DETECT-CYCLE), rozdělit na dvě skupiny: vrcholy, které jsou součástí akceptujícího cyklu a vrcholy, které jeho součástí nejsou. Vrcholy, které jsou sočástí akceptujícího cyklu jsou podle (m) následníky nějakého vrcholu h a toho můžeme využít. Kdyby vždy platilo, že iniciálním vrcholem je vrchol h, a nebo nějaký jeho předek, tak bychom při obnově procedurou CLEAN-VERTICES přebarvili všechny tyto vrcholy na bílou barvu. A jelikož během procedury DETECT-CYCLE pracujeme pouze s černými vrcholy, tak nemusíme pro vrcholy, které tvořily akceptující cyklus, obnovovat hodnoty *.nested*; ty budou obnoveny ve chvíli, kdy bude vrchol znovu navštíven procedurou NDFS-VISIT.

Položíme tedy další omezení na iniciální vrchol:

- Iniciální vrchol musí být vždy buď vrchol h z pozorování (l) anebo nějaký jeho předek

Vrcholy, které nebyly součástí akceptujícího cyklu, můžeme opět rozdělit na dvě skupiny: vrcholy, které jsou následníky vrcholu h a ty, jenž nejsou. Pro vrcholy, které jsou potomky vrcholu h platí výše uvedené v souvislosti s vrcholy, které byly součástí akceptujícího cyklu; jejich hodnoty tedy není třeba obnovovat. Pro vrcholy, které nejsou následníky vrcholu h platí, že při jejich zpracování procedurou DETECT-CYCLE z nich nevedla cesta k žádnému šedému vrcholu, jelikož kdyby vedla, tak by při jejich zpracování byl nutně detekován akceptující cyklus a to nebyl. Navíc platí, že ani v žádném okamžiku při prohledávání vnější procedurou, který následoval po zpracování takovýchto vrcholů, z těchto vrcholů nevedla cesta k šedému vrcholu (pozorování (h)). Tato cesta by mohla vzniknout pouze v případě, že by do grafu byla přidána hrana k níž (respektive od níž) vede cesta z (respektive k) takovému vrcholu. Jelikož jsou však tyto vrcholy černé a my požadujeme, aby iniciální vrchol byl šedý a zároveň aby byl buď shodný s výchozím vrcholem takto přidané hrany anebo s nějakým předkem tohoto vrcholu, tak by byly nutně námi diskutované vrcholy následníky iniciálního vrcholu a platí pro ně opět výše zmíněné; není třeba u nich obnovovat hodnotu *.nested*, jelikož ta bude obnovena při jejich opětovném navštívení vnější procedurou.

3.3 Označení akceptujícího cyklu

V případě, kdy mám v grafu již nalezený akceptující cyklus a přidáme do něj další hranu, tak zřejmě tento akceptující cyklus nemůže zaniknout. Proto při zpracování aditivního prvku v takovém případě nebudeme nikdy spouštět prohledávání grafu, pouze zkontrolujeme, jestli toto zpracování nemohlo změnit iniciální vrchol. Pokud však máme graf, ve kterém již existuje dosažitelný akceptující cyklus, a my z něj odebereme hranu, tak můžeme tento cyklus (anebo cestu k němu) narušit. Cestu k akceptujícímu cyklu jsme schopni rozeznat, jelikož je tvořena šedými vrcholy (pozorování (e) a (f)), vrcholy tvořící akceptující cyklus však již schopni rozeznat nejsme. Z toho důvodu nějak označíme i tyto vrcholy, abychom při zpracování subtraktivního prvku byli schopni říct, zda byl narušen již existující akceptující cyklus (anebo cesta k němu) a je potřeba začít prohledávat graf.

Toto označení budeme provádět v rámci běhu vnitřní procedury pro detekci akceptujícího cyklu DETECT-CYCLE(v) a to tak, že každý vrchol při začátku jeho zpracovávání vhodně označíme a při ukončení jeho zpracovávání toto označení zase odstraníme. Platí totiž, že tato procedura buď zpracuje všechny vrcholy dosažitelné z vrcholu v a nenalezne akceptující cyklus (takže i ukončí zpracovávání všech vrcholů dosažitelných z v), anebo nalezne akceptující cyklus a okamžitě ukončí průběh procedury. Zpracovávání některých vrcholů tedy nebude ukončeno a jelikož tato pro-

cedura pracuje na principu prohledávání do hloubky, tak platí, že právě tyto vrcholy tvoří akceptující cyklus (pozorování (e)).

Vrcholy budeme označovat pomocí čísla aktuální iterace, abychom dokázali rozlišit aktuální akceptující cyklus od akceptujících cyklů, které byly narušeny v předchozích iteracích. V rámci našeho programu proto zavedeme další dvě proměnné, čítač aktuální iterace *iteration* a číslo iterace, ve které byl naposledy prohledáván graf *lastSearch*. Kód modifikované procedury DETECT-CYCLE ukazuje obrázek 3.8.

```
DETECT-CYCLE(v)
1  v.nested ← true
2  v.accIteration ← iteration
3  for each (v, w) ∈ E
4        do if w.color = gray
5              then detected ← true
6                   EXIT()
7           elseif w.nested ≠ true
8           DETECT-CYCLE(w)
9  v.accIteration ← −1
```

Obrázek 3.8 Modifikovaná procedura DETECT-CYCLE. Řádek 2 na začátku zpracovávání každého vrcholu v uloží do $v.accIteration$ číslo aktuální iterace a označí tak tento vrchol jako potenciální součást akceptujícího cyklu. V případě, že vrchol není součástí akceptujícího cyklu, tak je na řádku 9 toto označení nastaveno na hodnotu -1 (což zřejmě není číslo žádné iterace).

Pozorování

(o) Po skončení prohledávání grafu může nastat pouze jedna z těchto dvou možností

- Byl nalezen akceptující cyklus, který je tvořen právě takovými vrcholy v, pro které je hodnota $v.accIteration$ rovna hodnotě $lastSearch$
- Nebyl nalezen akceptující cyklus a pro žádný vrchol v neplatí, že hodnota $v.accIteration$ je rovna hodnotě $lastSearch$

3.4 Anotovaný graf

Již jsme si uvedli všechny dodatečné informace, které si za účelem zefektivnění výpočtu budeme muset pamatovat. Graf rozšířený o tyto informace budeme nazývat anotovaným grafem a nyní si pro zopakovaní a upřesnění uvedeme jeho formální definici.

Anotovaný graf $\mathbb{G} = (V, E, v_0, init, iteration, lastSearch, detected)$ se skládá z množiny vrcholů V, množiny hran E, počátečního vrcholu v_0, iniciálního vrcholu

init, aktuální iterace, ve které se graf nachází *iteration*, číslo iterace *lastSearch*, ve které naposledy došlo k prohledávání grafu a hodnoty *detected*, která udržuje informaci o existenci dosažitelného akceptujícího cyklu v tomto grafu.

- $v \in V \ = (color, acc, timestamp, ancestor, nested, accIteration)$
  - $color \in \{white, gray, black\}$ je barva vrcholu
  - $acc \in \{true, false\}$ rozlišuje, zda-li se jedná o akceptující vrchol
  - $timestamp \in \mathbb{N}$ je časová známka
  - $ancestor \in V$ je rodič tohoto vrcholu (z čehož mimo jiné pro nebílé vrcholy vyplývá, že v grafu existuje hrana $e = (ancestor, v)$)
  - $accIteration \in \mathbb{N}$ určuje poslední iteraci, ve které byl vrchol označen za součást akceptujícího cyklu
- $e \in E = (a, b); a, b \in V$
- $v_0, init \in V$
- $iteration \in \mathbb{N}$ je číslo iterace, ve které se graf momentálně nachází, tedy počet již zpracovaných prvků z množiny H
- $lastSearch \in \{0..iteration\}$
- $detected \in \{true, false\}$

Dříve jsme v pseudokódech použitých funkcí používali pro graf značení $G(V, E)$, jelikož jsme v této funkci tyhle možiny využívali. Pro anotovaný graf však budeme v zájmu přehlednosti používat pouze označení G, jelikož již používáme mnohem více proměnných, než jen V a E. Názvy příslušných proměnných použitých v kódu funkce jsou převzaty z výše uvedené definice a jejich význam by tak měl být zřejmý.

3.5 Operace *zpracování* nad anotovaným grafem

V kapitole 2.2 jsme si definovali operaci *zpracování* pro graf $G = (V, E)$, pro práci s anotovaným grafem však budeme muset tuto definici změnit.

Zpracováním prvku h z posloupnosti H budeme označovat transformaci anotovaného grafu $\mathbb{G} = (V, E, v_0, init, iteration, lastSearch, detected)$ na anotovaný graf $\overline{\mathbb{G}} = (\overline{V}, \overline{E}, v_0, \overline{init}, \overline{iteration}, \overline{lastSearch}, \overline{detected})$. Pokud v důsledku zpracování prvku nedojde k prohledávání grafu, tak bude $\overline{V}$ shodné s V, v opačném případě bude $\overline{V}$ určeno průběhem prohledávání. Dále pak pro aditivní prvky platí $\overline{E} = E \cup \{h\}$ a pro substraktivní prvky $\overline{E} = E - h$. Hodnota $\overline{iteration}$ je rovna $iteration + 1$. Hodnota $\overline{lastSearch}$ je závislá na tom, zda-li se při zpracování prvku prohledává graf či nikoliv. Pokud ano, tak $\overline{lastSearch}$ má hodnotu $\overline{iteration}$, pokud ne, zůstává tato hodnota stejná jako v grafu $\mathbb{G}$. Ohodnocení *detected* je buď závislé na výsledku

prohledávání grafu, pokud bylo uskutečněno, anebo tato hodnota zůstává stejná jako pro graf $\mathbb{G}$.

Hodnota $\overline{init}$ je dána existencí dosažitelného akceptujícího cyklu v $\mathbb{G}$, dále na tom, jestli zpracováváme aditivní, či subtraktivní prvek $h = (u, v)$, a také na vlastnostech vrcholu u a v. Musíme proto rozlišit několik různých situací; postup určení hodnoty $\overline{init}$ je uveden níže.

Řekneme, že pro anotovaný graf $\mathbb{G}$ a posloupnost H je $\mathbb{G}$ v k-té iteraci, pokud proběhlo zpracování prvních k prvků z posloupnosti H. Anotovaný graf $\mathbb{G}$, který je v k-té iteraci, budeme dále označovat $\mathbb{G}_k$.

NGA vrcholu

Pokud zpracováváme hranu $h = (u, v)$, tak bychom občas chtěli, aby se u stal novým iniciálním vrcholem. Jelikož však jsem si položení omezení, že iniciální vrchol musí být šedý, tak to nemusí být možné. Z toho důvodu v těchto situacích místo vrcholu u použijeme nějakého jeho předka.

Pod pojmem NGA (z anglického *Nearest Gray Ancestor*) vrcholu v budeme označovat toho šedého předka vrcholu v, který má nejvyšší časovou známku. Jelikož si při zpracovávání libovolného vrcholu vždy zapamatujeme, ze kterého vrcholu byl tento vrchol navštíven, tak jsme schopni iterativně pomocí této vazby pro každý vrchol NGA spočítat. Nutnou podmínkou je, aby každý vrchol měl alespoň jednoho šedého předka, ale ta je automaticky splněna, jelikož každý vrchol má jako předka minimálně vrchol v_0 a ten je šedý vždy. Pro vrchol v_0 a bílé vrcholy není NGA definován.

Zpracování prvku pro graf, který neobsahuje dosažitelný akceptující cyklus

To, že graf $\mathbb{G}_i$ neobsahuje dosažitelný akceptující cyklus znamená, že byl celý prohledán a akceptující cyklus nebyl při prohledávání nalezen. To také znamená, že zpracování všech vrcholů bylo při prohledávání ukončeno a graf tak neobsahuje ani jeden šedý vrchol. A jelikož jsme si na iniciální vrchol položili omezení, že musí být buď šedý anebo shodný s počátečním vrcholem, tak hodnota *init* v grafu $\mathbb{G}_i$ musí být nutně v_0. Zaveďme si tři následující pravidla:

- Jestliže prohledávání grafu nenalezne akceptující cyklus, tak se iniciálním vrcholem stává počáteční vrchol v_0
- Jestliže bychom měli navázat na předchozí prohledávání a iniciální vrchol je shodný s vrcholem v_0, tak navazovat nebudeme, ale začneme úplně nové prohledávání pomocí ACC-CYCLE-DETECTION.

- Jestliže prohledávání grafu skončí nalezením akceptujícího cyklu, tak se iniciálním vrcholem stává šedý vrchol ležící na akceptujícím cyklu, který má nejmenší časovou známku (vrchol h z pozorování (l)).

Zřejmě platí, že jestliže graf $\mathbb{G}_i$ neobsahuje dosažitelný akceptující cyklus, tak tím, že z něj odebereme libovolnou hranu, na tomto faktu nic nezměníme. Proto nebudeme spouštět nové prohledávání a ani nebudeme měnit hodnotu iniciálního vrcholu.

Přidáním nové hrany však již vznik takového cyklu zapříčinit můžeme a budeme proto muset upravený graf prohledávat. Nová hodnota iniciálního vrcholu je dána výše uvedenými pravidly.

Zpracování prvku pro graf, který obsahuje dosažitelný akceptující cyklus

Jestliže graf $\mathbb{G}_i$ obsahuje dosažitelný akceptující cyklus a my do něj přidáme libovolnou hranu, tak je zřejmé, že existenci tohoto cyklu nijak neovlivníme a z toho důvodu ani nebudeme spouštět prohledávání grafu. Můžeme však zapříčinit vznik dalšího dosažitelného akceptujícího cyklu a v takovém případě musíme zvážit upravení iniciálního vrcholu. Následuje výčet jednotlivých možností, jak může vypadat přidaná hrana do grafu a jaký dopad bude mít tato změna na iniciální vrchol.

1. (černý vrchol, černý vrchol), předpokládáme zde a i níže pojmenování této dvojice vrcholů (u, v)
   Může způsobit vznik nového akceptujícího cyklu. Platí, že pokud je časová známka vrcholu u menší než časová známka iniciálního vrcholu, tak se novým iniciálním vrcholem stává NGA vrcholu u.

2. (šedý vrchol, černý vrchol)
   Může způsobit vznik nového dosažitelného akceptujícího cyklu, ale jelikož zpracování vrcholu u ještě nebylo ukončeno, tak bude tato hrana algoritmem ještě zpracována. Iniciální vrchol se nemění.

3. (černý vrchol, šedý vrchol)
   Může způsobit vznik nového akceptujícího cyklu, platí že pokud je časová známka vrcholu u menší než časová známka iniciálního vrcholu, tak se NGA vrcholu u stává novým iniciálním vrcholem.

4. (šedý vrchol, šedý vrchol)
   Může způsobit vznik nového akceptujícího cyklu, ale jelikož zpracování vrcholu u ještě nebylo ukončeno, bude tato hrana algoritmem teprve zpracována. Iniciální vrchol se nemění.

5. (černý vrchol, bílý vrchol)
   Může způsobit vznik nového akceptujícího cyklu. Platí, že pokud je časová známka vrcholu u menší než časová známka iniciálního vrcholu, tak se novým iniciálním vrcholem stává NGA vrcholu u.

6. (bílý vrchol, černý vrchol)
   Může způsobit vznik nového akceptujícího cyklu, ale jelikož vrchol u teprve bude algoritmem zpracován, tak se iniciální vrchol nemění.

7. (šedý vrchol, bílý vrchol)
   Může způsobit vznik nového akceptujícího cyklu, ale jelikož zpracování vrcholu u ještě nebylo ukončeno, tak bude tato hrana algoritmem ještě zpracována. Iniciální vrchol se nemění.

8. (bílý vrchol, šedý vrchol)
   Může způsobit vznik nového akceptujícího cyklu, ale jelikož vrchol u teprve bude algoritmem zpracován, tak se iniciální vrchol nemění.

9. (bílý vrchol, bílý vrchol)
   Může způsobit vznik nového akceptujícího cyklu, ale jelikož vrchol u teprve bude algoritmem zpracován, tak se iniciální vrchol nemění.

Z výše uvedených devíti možností vyšlo pouze pro tři z nich, že přidání hrany takového typu může mít vliv na iniciální vrchol a jsou to právě hrany vedoucí z černých vrcholů. Tento výsledek je ale zcela očekávatelný, jelikož pokud hrana vede z bílého vrcholu, tak vede z místa, do kterého se výpočet, na nějž chceme navázat, vůbec nedostal, a proto zde ani nemůžeme nalézt žádný vhodný iniciální vrchol. U hran, které vedou z šedých vrcholů, platí, že tyto hrany mohou být prozkoumány v pokračujícím běhu prohledávání, a proto pokud by přidání takové hrany mohlo způsobit vznik nového dosažitelného akceptujícího cyklu, tak to, že nezměníme iniciální vrchol, dosažitelnost takového cyklu neovlivní. Problémem je však to, že na hranách máme definované uspořádání, které je založené na uspořádání vrcholů, a z toho důvodu bychom mohli tuto hranu "přeskočit". Proto tohle uspořádání upravíme následovně: v nulté iteraci grafu je uspořádání na hranách dáno opět v závislosti na uspořádání vrcholů a pokud v dalších iteracích přidáme do grafu nějakou hranu, tak ji v tomto uspořádání vždy zařadíme na poslední místo. To nám zaručí, že nově přidanou hranu vždy budeme moci zpracovat.

V případě, kdy graf obsahoval dosažitelný akceptující cyklus a my z něj odebereme hranu, tak můžeme tento cyklus rozpojit anebo rozpojit cestu k němu. Pokud jedno z těchto rozpojení nastane, tak v případě potřeby změníme iniciální vrchol a vždy budeme prohledávat graf. Jak jsme si uvedli již dříve, cesta k akceptujícímu cyklu je tvořena šedými vrcholy a akceptující cyklus je označen pomocí proměnných *.accIteration*, kontrolu na rozpojení proto budeme provádět pomocí těchto označení.

Jestliže odebereme hranu, která není ani součástí cesty k akceptujícímu cyklu a ani cyklu samotného, tak nebudeme provádět prohledávání, jelikož v grafu zůstává již objevený cyklus a nebudeme ani měnit iniciální vrchol, jelikož odebrání hrany nemohlo zapříčinit vznik nového dosažitelného akceptujícího cyklu. Následuje výčet pro nás zajímavých možností odebrání hrany.

1. $(u, v) : u.color = gray \wedge v.color = gray$
   Znamená porušení cesty k dosažitelnému akceptujícímu cyklu. Platí, že pokud je časová známka vrcholu u menší než časová známka iniciálního vrcholu, tak se novým iniciálním vrcholem stává vrchol u.

2. $(u, v) : u.accIteration = lastSearch \wedge v.accIteration = lastSearch$
   Znamená rozpojení akceptujícího cyklu, který byl nalezen při posledním prohledávání a tedy toho aktuálního. Iniciální vrchol se nemění.

Ve dvou výše uvedených příkladech tedy musíme prohledávat graf. Ve všech ostatních případech můžeme v konstantním čase prohlásit, že řešení problému akceptujícího cyklu v grafu $\mathbb{G}_{i+1}$ je shodné s řešením pro graf $\mathbb{G}_i$.

Funkce PROCESS-ELEMENT pro anotovaný graf G_i a prvek $h(u, v)$ na vstupu provede zpracování prvku $h(u, v)$, tedy převede G_i na G_{i+1}, a vrátí hodnotu $true$ v případě, že G_{i+1} obsahuje dosažitelný akceptující cyklus, v opačném případě hodnotu $false$. Výpočet je větven podle toho, zda graf G_i obsahuje dosažitelný akceptující cyklus či nikoliv (dle hodnoty proměnné *detected*) a pak dále podle toho, jestli zpracováváme subtraktivní ($h.isSubtractive = true$), či aditivní ($h.isSubtractive = false$) prvek . Kód a popis této funkce ukazuje obrázek 3.9.

```
PROCESS-ELEMENT(G, h(u, v))
 1  iteration ← iteration + 1
 2  if detected
 3     then
 4          if h.isSubtractive
 5             then E ← E − h
 6                  if u.color = gray ∧ v.color = gray
 7                     then if u.timestamp < init.timestamp
 8                               then init ← u
 9                          return CONTINUE(G, init)
10                  elseif u.accIteration = lastSearch ∧
                           v.accIteration = lastSearch
11                     then return CONTINUE(G, init)
12                     else return true
13             else E ← E ∪ {h}
14                  if u.color = black
15                     then if u.timestamp < init.timestamp
16                               then init ← NGA(u)
17                  return true
18     else
19          if h.isSubtractive
20             then E ← E − h
21                  return false
22             else E ← E ∪ {h}
23                  init ← v0
24                  detected ← ACC-CYCLE-DETECTION(G, v0)
25                  lastSearch ← iteration
26                  return detected
```

Obrázek 3.9 Kód a popis funkce PROCESS-ELEMENT, která provede zpracování jednoho prvku. Na řádcích 3 a 23 je provedeno větvení v závislosti na tom, jestli G_i obsahuje dosažitelný akceptující cyklus či nikoliv. Pokud jdeme první větví, tak je dále na řádcích 4 a 17 rozlišeno o jaký prvek se jedná a hned je v grafu buď přidána, anebo odebrána hrana. V případě odebrání hrany rozlišujeme na řádcích 7 a 12 ty případy, kdy mohla být rozpojena cesta k akceptujícímu cyklu, respektive rozpojen akceptující cyklus, upravujeme iniciální vrchol, pokud je to třeba, pomocí procedury CLEAN-UP provádíme obnovu stavu výpočtu, jaký byl při začátku zpracování iniciálního vrcholu, a poté spouštíme funkci CALL-STACK, která naváže na předchozí výpočet z iniciálního vrcholu *init*. Výsledek této funkce ukládáme do proměnné *detected* a tu pak následně vracíme jako výsledek volání PROCESS-ELEMENT. Pokud odebrání hrany nemohlo rozpojit akceptující cyklus a ani cestu k němu, tak na řádku 16 vracíme hodnotu $false$ a žádné další změny neprovádíme.

Pokud jdeme druhou větví, tak je na řádcích 25 a 27 opět rozlišeno, zda se jedná o aditivní, či subtraktivní prvek a v grafu je opět hned přidána, respektive odebrána hrana. Pokud v grafu neexistoval dosažitelný akceptující cyklus a my z něj odebrali hranu, tak na řádku 26 vracíme hodnotu $false$ a další změny neprovádíme. V případě, že jsme do grafu přidali hranu, tak

mohl vzniknout dosažitelný akceptující cyklus, a proto začneme nové prohledávání grafu pomocí ACC-CYCLE-DETECTION, do proměnné *detected* nastavíme výsledek tohoto volání a vrátíme ho i jako výsledek volání PROCESS-ELEMENT. Po každém spuštění výpočtu vždy zaktualizujeme proměnnou *lastSearch*, která uchovává informaci o poslední iteraci, ve které bylo provedeno prohledávání.

3.6 Výsledný optimalizovaný algoritmus

Máme již popsané všechny kroky, které budeme muset během rozhodování problému akceptujícího cyklu nad dynamickým grafem provést a uvedli a vysvětlili jsme si všechny funkce, které budeme při tomto rozhodování využívat. Můžeme tedy uvést výsledný algoritmus, který tento problém rozhoduje. Je složený pouze z procedur, které jsme již dříve uvedli a pracuje na principu, který zde byl popsán; na následujících stranách je ukázán jeho pseudokód.

DYM-ACC-CYCLE-DETECTION(G, v_0, H)

```
1  S ← array()
2  S_0 ← ACC-CYCLE-DETECTION(G, v_0)
3  for (i = 1; i ≤ H.length; i ← i + 1)
4        do S_i ← PROCESS-ELEMENT(G, h_{i−1})
5  return S
```

ACC-CYCLE-DETECTION(G, v_0)

```
1  detected ← false
2  for each v ∈ V
3        do v.color ← white
4           π[v] ← NIL
5  timestamp ← 0
6  NDFS-VISIT(v_0)
7  return detected
```

NDFS-VISIT(v)

```
 1  v.color ← gray
 2  v.nested ← false
 3  v.timestamp ← timestamp
 4  timestamp ← timestamp + 1
 5  for each (v, w) ∈ E
 6        do if w.color = white
 7              then π[w] ← v
 8                   NDFS-VISIT(w)
 9  if v.isAcc
10     then DETECT-CYCLE(v)
11  v.color ← black
```

```
DETECT-CYCLE(v)
 1  v.nested ← true
 2  v.accIteration ← iteration
 3  for each (v, w) ∈ E
 4        do if w.color = gray
 5              then detected ← w.nested ← true
 6                   init ← w
 7                   EXIT()
 8           elseif w.nested ≠ true
 9              then DETECT-CYCLE(w)
10  v.accIteration ← −1
```

```
PROCESS-ELEMENT(G, h(u, v))
 1  iteration ← iteration + 1
 2  if detected
 3     then
 4          if h.isSubtractive
 5             then E ← E − h
 6                  if u.color = gray ∧ v.color = gray
 7                     then if u.timestamp < init.timestamp
 8                               then init ← u
 9                          return CONTINUE(G,INIT)
10                  elseif u.accIteration = lastSearch ∧
                           v.accIteration = lastSearch
11                     then return CONTINUE(G,INIT)
12                     else  return true
13             else  E ← E ∪ {h}
14                  if u.color = black
15                     then if u.timestamp < init.timestamp
16                               then init ← NGA(u)
17                  return true
18     else
19          if h.isSubtractive
20             then E ← E − h
21                  return false
22             else  E ← E ∪ {h}
23                  init ← v0
24                  detected ← ACC-CYCLE-DETECTION(G, v0)
25                  lastSearch ← iteration
26                  return detected
```

```
CONTINUE(G, v)
1  if v.timestamp > (timestamp − v.timestamp)
2     then timestamp ← init.timestamp
3          CLEAN-VERTICES(init)
4          detected ← CALL-STACK(v)
5     else detected ← ACC-CYCLE-DETECTION(G, v)
6  lastSearch ← iteration
7  return detected

CLEAN-VERTICES(v)
1  v.color ← white
2  for each (v, w) ∈ E
3        do if w.color ≠ white ∧ w.timestamp > v.timestamp
4              then CLEAN-VERTICES(w)

NGA(u)
1  while u.color ≠ gray
2        do u ← π[u]
3  return u

CALL-STACK(v)
1  NDFS-VISIT(v)
2  v ← π[v]
3  while v ≠ NIL
4        do COMPLETE-CALL(v)
5           v ← π[v]
6  return detected

COMPLETE-CALL(v)
1  for each (v, w) ∈ E
2        do if w.color = white
3              then π[w] ← v
4                   NDFS-VISIT(w)
5  if v.isAcc
6     then DETECT-CYCLE(v)
7  v.color ← black
```

Obrázek 3.10 Výsledný optimalizovaný algoritmus pro rozhodování problému akceptujícího cyklu v dynamickém grafu.

4 Alternativní přístupy a srovnání

V této kapitole si nastíníme další možné přístupy, kterými by mohl jít problém dosažitelnosti akceptujícího cyklu v dynamickém grafu řešit, stručně zmíníme jejich možné výhody a nevýhody a porovnáme je s algoritmem, který jsme v této práci představili.

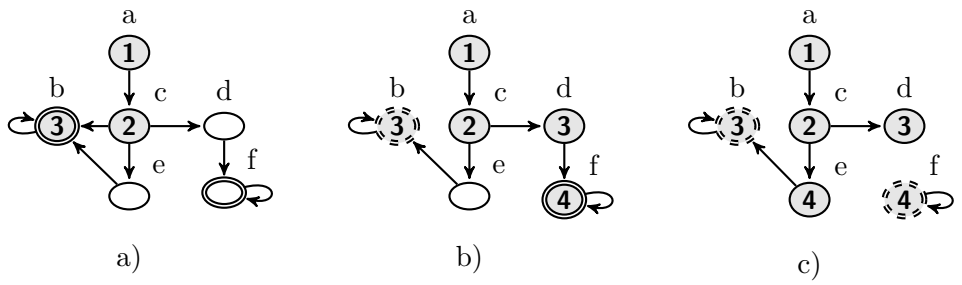
4.1 NDFS bez obnovy hodnot proměnných

V námi zvoleném přístupu provádíme vždy před navázáním na průběh předchozího prohledávání obnovení hodnot proměnných do takového stavu, v jakém byly při začátku zpracovávání iniciálního vrcholu. Uvedli jsme si, proč je nezbytné tuto obnovu provést a na obrázku 3.5 jsme demonstrovali co by se mohlo stát, kdybychom tuto obnovu neprovedli. Řekli jsme si však, že tato obnova může být časově náročná a že v jistých případech ji ani nebudeme provádět a začneme raději úplně nové prohledávání. Místo snahy o obnovu ohodnocení proměnných bychom se však mohli pokusit pouze tyto informace označit za neaktuální a předpokládat iniciální ohodnocení vrcholů. Ukázali jsme si, že vyjma pár hodnot, které lze obnovit v konstantním čase, potřebujeme obnovit pouze barvy vrcholů a to z toho důvodu, abychom je mohli znovu navštívit. Pojďme si tedy naznačit postup, jakým bychom mohli v konstantním čase označit informace o barvách některých vrcholů za neaktuální.

Řekli jsme si, že obnovujeme hodnoty právě takových vrcholů, které mají vyšší časovou známku než má iniciální vrchol. Mohli bychom tedy v navazujícím prohledávání umožnit navštěvovat kromě bílých vrcholů také ty vrcholy, které mají vyšší časovou známku než iniciální vrchol. Tento přístup by však okamžitě selhal ve chvíli, kdybychom narazili na nějaký cyklus; prohledávání by se zacyklilo, jelikož bychom tyto vrcholy mohli navštěvovat pořád dokola. Zkusme proto tedy tuto podmínku zpřísnit tak, že bychom mohli kromě bílých vrcholů navštěvovat i vrcholy, které mají vyšší časovou známku než iniciální vrchol a zároveň byly naposledy navštíveny v některé z předchozích iterací. Takto jsme již schopni tyto vrcholy znovu navštívit a vyhneme se zacyklení.

Výše uvedený postup pro umožnění opětovného navštívení daných vrcholů však funguje pouze v té iteraci, v níž označíme informace o těchto vrcholech za neaktivní a navážeme na prohledávání grafu. Jestliže bychom navazovali na prohledávání v i-té a následně v j-té iteraci a iniciální vrchol pro j-tou iteraci by měl vyšší časovou známku než iniciální vrchol pro tu i-tou, tak by už nebylo možné návštívit ty vrcholy, které mají časovou známku v rozmezí časových známek těchto dvou iniciálních vrcholů; na příkladu tento problém demonstruje obrázek 4.1.

Pokud bychom tedy chtěli tento postup využít, tak bychom mohli navazovat na předchozí prohledávání jen v těch případech, kdy by iniciální vrchol neměl větší časovou známku než iniciální vrchol při předchozím navázání. V opačném případě



Obrázek 4.1 Ukazuje graf ve třech po sobě následujících iteracích, kdy byly postupně odebrány hrany (c, b) a (d, f). Barvy vrcholů ukazují stav, ve kterém skončil výpočet, čísla uvnitř vrcholů jsou příslušné časové známky, vrcholy b a f jsou akceptující a vrcholy, které jsou nakresleny přerušovanou čarou, jsou vrcholy, jejichž informace byly zneaktuálněny pomocí podmínky na poslední zpracování v jedné z předešlých iterací a vyšší časovou známku než má iniciální vrchol.

Na obrázku a) byl nalezen akceptující cyklus tvořený vrholem b a prohledávání bylo ukončeno. Obrázek b) znázorňuje iteraci, kdy byla odebrána hrana (c, b), vrchol c se stal novým iniciálním vrcholem a byl nalezen akceptující cyklus tvořený vrcholem f. Poslední obrázek pak zobrazuje iteraci, kdy byla odebrána hrana (d, f), vrchol d se stal novým iniciálním vrcholem a prohledávání došlo do vrcholu e, kde by správně mělo navštívit vrchol b a detekovat akceptující cyklus, ale jelikož b nemá vyšší časovou známku než iniciální vrchol (3), tak se tak nestane.

bychom museli začít nové prohledávání. Také bychom si však v těchto případech mohli pro každou iteraci zvlášť pamatovat množinu vrcholů, jejichž informace byly zneaktuálněny a pak při každé kontrole na navštívenost vrcholu kontrolovat, jestli daný vrchol nepatří do jedné z těchto množin. A jelikož nám k vyznačení takové množiny stačí, jak již bylo uvedeno, pamatovat si pouze dvojici (časová známka iniciálního vrcholu, číslo aktuální iterace), tak nás pamatování si těchto informací nemusí stát ani moc místa; kolik takových množin si během výpočtu budeme muset zapamatovat závisí na vlastnostech prvků z množiny H, které budeme zpracovávat. V nejhorším případě si pro každý zpracovaný prvek budeme muset zapamatovat jedno takové vyznačení množiny.

Pro implementaci tohoto řešení do našeho algoritmu je třeba upravit proceduru CONTINUE tak, jak ukazuje obrázek 4.2, dále je nutné na začátku procedury NDFS-VISIT pomocí instrukce $v.iteration \leftarrow iteration$ u vrcholu poznačit, v jaké iteraci byl naposledy zpracován, a je potřeba upravit podmínku na navštívenost vrcholu následovně:

if $w.color = white \vee \exists(t, i) \in outdated$ such as $(w.timestamp > t \wedge w.iteration \leq i)$

```
CONTINUE(G, v)
1  if init.timestamp > lastSearchInit.timestamp
2      then outdated ← outdated ∪ {(init.timestamp, iteration)}
3  detected ← CALL-STACK(v)
4  lastSearch ← iteration
5  lastSearchInit ← init
6  return detected
```

Obrázek 4.2 Kód upravené procedury CONTINUE

Poslední změnou je úprava formální definice anotovaného grafu:
Anotovaný graf $\mathbb{G} = (V, E, v_0, init, iteration, lastSearch, detected, outdated)$

- $v \in V \ = (color, acc, timestamp, ancestor, nested, accIteration)$
  - $color \in \{white, gray, black\}$ je barva vrcholu
  - $acc \in \{true, false\}$ rozlišuje, zda-li se jedná o akceptující vrchol
  - $timestamp \in \mathbb{N}$ je časová známka
  - $ancestor \in V$ je předchůdce tohoto vrcholu (z čehož mimo jiné vyplývá, že v grafu existuje hrana $e = (ancestor, v)$)
  - $accIteration \in \mathbb{N}$ určuje poslední iteraci, ve které byl vrchol označen za součást akceptujícího cyklu
  - $iteration$ je číslo iterace, ve které byl vrchol naposledy navštíven
- $e \in E = (a, b); a, b \in V$
- $v_0, init \in V$
- $iteration \in \mathbb{N}$ je číslo iterace, ve které se graf momentálně nachází, tedy počet již zpracovaných prvků z množiny H
- $lastSearch \in \{0..iteration\}$
- $detected \in \{true, false\}$
- *outdated* je množina prvků tvaru (t, i): $t \in \mathbb{N}$, $i \in \{0, .., iteration\}$, která vymezuje vrcholy, jejichž ohodnocení bylo prohlášeno za neaktuální

Tato varianta je oproti řešení, které jsme uvedli ve třetí kapitole rychlejší ve chvíli, kdy se navazuje na předchozí průběh prohledávání, jelikož není potřeba vykonávat proceduru CLEAN-VERTICES, která má lineární časovou složitost vzhledem k počtu vrcholů v grafu, ale stačí pouze v konstantním čase rozhodnout, zda-li je třeba upravit množinu *outdated* a pokud ano, tak do ní v konstantním čase přidat jeden prvek. Nevýhodou je složitější podmínka pro ověření, zda je možno navštívit vrchol, kdy se kromě barvy vrcholu zkoumá i jeho příslušnost do množiny *outdated*, jejíž počet prvků může narůst až do velikosti množiny zpracovávaných prvků H.

4.2 Využití Tarjanova algoritmu

Tarjanův algoritmus je algoritmus postavený na již zmíněném prohledávání do hloubky (DFS) a využívá se k rozkladu grafu na silně souvislé komponenty, což jsou takové podgrafy zadaného grafu, ve kterých jsou z každého vrcholu dosažitelné všechny vrcholy, které patří do stejné komponenty. A to tedy znamená, že každé dva vrcholy, které patří do stejné komponenty, jsou součástí jednoho cyklu. Tarjanův algoritmus se proto dá využít pro řešení problému dosažitelnosti akceptujícího cyklu a to tak, že při odhalení každé silně souvislé komponenty zkontrolujeme, jestli je mezi vrcholy, které tuto komponentu tvoří, obsažen akceptující vrchol. Pokud ano, tak je tento vrchol nutně součástí akceptujícího cyklu a jelikož je Tarjanův algoritmus postaven na algoritmu DFS a prohledávání grafu začíná v daném počátečním vrcholu v_0, tak i víme, že takto nalezený cyklus je dosažitelný z vrcholu v_0. Bližší popis Tarjanova algoritmu lze najít v [8].

Pro účely této práce by bylo možné využít i tento algoritmus, navíc jsou oba, jak Tarjanův algoritmus, tak i NDFS postaveny na prohledávání do hloubky, takže pracují velice podobně a optimalizační techniky, které byly v této práci představeny, by se dali až na mírné modifikace aplikovat i na Tarjanův algoritmus. V praxi je však pro řešení problému dosažitelného akceptujícícho cyklu v grafu využíváno spíše NDFS [5] a to byl hlavní důvod, proč bylo jeho využití v této práci upřednostněno.

Přestože jsme uvedli mnohé optimalizační techniky, jak lze urychlit triviální řešení, kdy po každé modifikaci grafu modifikovaný graf znovu prohledáme, tak výsledné optimalizované řešení asymptoticky patří do stejné časové třídy jako toto triviální řešení, jelikož v nejhorším případě musíme vždy znovu prohledávat celý graf úplně od začátku. Posun do jiné časové složitostní třídy se však ani nepředpokládal.

5 Závěr

V této práci jsem se zabýval dynamickými orientovanými grafy, tedy grafy, které jsou postupně modifikovány tak, že se do nich postupně přidávají, nebo se z nich naopak odebírají hrany. Konkrétně se práce zaměřuje na problém dosažitelnosti akceptujícího cyklu a to po každé provedené modifikaci grafu. Možným jednoduchým řešením je po každé provedené změně v grafu prohledat celý graf algoritmem zanořeného prohledávání do hloubky (NDFS) a pomocí něj problém rozhodnout; v této práci je navržen optimalizovaný algoritmus vycházející z tohoto řešení.

Na úvod jsem formálně definoval pojem dynamického grafu, problém dosažitelnosti akceptujícího cyklu nad dynamickým grafem a podrobně a názorně jsem popsal chování algoritmu zanořeného prohledávání do hloubky a jeho vlastnosti, které lze využít při řešení zkoumaného problému. Následně jsem poukázal na hlavní nevýhody zadaného jednoduchého řešení a ukázal jsem způsoby, jak se těchto problémových částí vyvarovat. Hlavním přínosem práce je možnost vyhnout se zbytečnému opakování počáteční části prohledávání, která je pro modifikovaný graf stejná jako pro graf původní. Navrhl jsem způsob, jakým lze navázat na průběh předchozího prohledávání právě ve chvíli, kdy by se průběhy těchto dvou prohledávání začaly lišit, takže není nutné počátek výpočtu opakovat. To, jak velká část výpočtu je tímto ušetřena, se pro každou konkrétní modifikaci grafu liší. Další provedené optimalizace většinou vycházejí z charakteru prováděných úprav v grafu a často jsme schopni po provedené úpravě rozhodnout problém dosažitelnosti akceptujícího cyklu v konstantním čase. V průměrném případě by proto měl být optimalizovaný algoritmus výrazně efektivnější než původní řešení, v krajních případech na tom bude vždy přinejhorším asymptoticky stejně.

Jako další možné pokračování v této práci vidím potřebu implementace navržených řešení, která by umožnila otestovat je v praxi a porovnat jejich výkonnost s dalšími algoritmy, jenž řeší tento problém. V případě příznivých výsledků by se dalo uvažovat například o využití vzniklého algoritmu v oblasti robotiky a práce s umělou inteligencí, což byla jedna z motivací při zadávání tohoto tématu.

Literatura

[1] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[2] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[4] Edsger W Dijkstra. Recursive programming. *Numerische Mathematik*, 2(1):312–318, 1960.

[5] Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on büchi automata. *arXiv preprint arXiv:0910.3766*, 2009.

[6] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly ltl verification more efficient. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 205–219. Springer, 2004.

[7] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 184–191. ACM, 2004.

[8] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.