Aplikace grafových algoritmů v abstraktní interpretaci

Tomáš Dacík

11. února 2021

1 Úvod

Abstraktní interpretace je jednou z metod pro statickou analýzu programů, která se dnes i přes svůj teoretický základ uplatňuje i v průmyslové sféře. Jedná se o obecný rámec, který umožňuje definovat řadu různých analýz jako je např. analýza možných hodnot proměnných, analýza vlastněných zámků pro detekci uváznutí nebo analýza horních omezení počtů iterací smyček pro automatickou analýzu složitosti. Výpočet abstraktní interpretace je realizován jako iterativní řešení soustavy rovnic, při kterém je navíc potřeba zavést jistou aproximaci pro zajištění terminace, případně zrychlení celého výpočtu. Tento proces lze také vnímat jako výpočet nad grafem, kdy je v každém kroku vybrán vrchol grafu a na základě příchozích hran vypočítána hodnota s ním asociované proměnné, dokud nedojde k ustálení hodnot všech vrcholů.

Tato práce se zabývá pojmem *slabého topologického uspořádání*, které umožňuje provést dekompozici a uspořádání grafů takových soustav, určit vhodné pořadí aplikace jednotlivých rovnic a dosáhnout tak větší efektivity i přesnosti celého výpočtu. Jedná se o zobecnění pojmu topologické uspořádání, které umožňuje za určitých podmínek i existenci hran, které jdou „proti uspořádání“. Algoritmy pro výpočet se pak typicky snaží počet takových hran (přesněji počet vrcholů, do kterých vedou takové hrany) minimalizovat.

V práci je nejprve stručně představena abstraktní interpretace a způsob, jak na její výpočet nahlížet z grafového pohledu. Poté je popsána formální definice slabého topologického uspořádání a algoritmy pro jeho výpočet. Na závěr je shrnuto několik případů použití, další možná rozšíření a také aplikace mimo oblast analýzy programů.

1.1 Abstraktní interpretace

Abstraktní interpretaci představili Patrick Cousot a Radhia Cousot v článku [5]. Metoda je založena na myšlence nahrazení konkrétní sémantiky programu, jejíž vlastnosti jsou typicky nerozhodnutelné, abstraktní sémantikou. Abstraktní sémantika zachycuje pouze vlastnosti programu relevantní v kontextu jeho analýzy, a to takovým

způsobem, že jsou tyto vlastnosti rozhodnutelné díky použité nad-aproximaci. Abstraktní interpretace poskytuje obecný rámec, pomocí kterého je možné definovat konkrétní analýzy a dokázat jejich korektnost. Formální definici abstraktní interpretace včetně podmínek, které jsou kladeny na jednotlivé její komponenty lze nalézt v [11]. Těmito komponentami jsou:

- **Abstraktní doména** je množina abstraktních kontextů, které tvoří nad-aproximace konkrétních stavů programu. Jednoduchým příkladem je intervalová doména, která umožňuje reprezentovat hodnotu jedné proměnné v daném místě programu jako interval jejích možných hodnot. Zobecněním pak lze získat doménu konvexních polyhedrů (convex polyhedra domain), která umožňuje reprezentovat možné hodnoty n proměnných jako konvexní útvary v n-rozměrném prostoru.
- **Abstraktní transformery** jsou funkce pro jednotlivé instrukce programu, které modelují jejich vliv na abstraktní kontexty.
- **Operátor spojení** ($\sqcup$) definuje, jakým způsobem jsou zkombinovány abstraktní kontexty přicházející ze dvou různých větví programu v místě jejich spojení. Abstraktní doména spolu s uspořádáním $\sqsubseteq$ ($x \sqsubseteq y \Leftrightarrow x \sqcup y = y$) tvoří úplný svaz.
- **Operátor widening** ($\triangledown$) zavádí nad-aproximaci, která zajišťuje terminaci analýzy cyklů v programu. I v případě, kdy je abstraktní doména konečná, se tento operátor často používá pro zrychlení výpočtu.

Na základě těchto komponent je možné převést program na soustavu rovnic s proměnnými X_i, které reprezentují kontrolní body programu a nabývají hodnot z abstraktní domény:

$$\begin{aligned} X_1 &= \Phi_1(X_1, X_2, \ldots, X_n) \\ X_2 &= \Phi_2(X_1, X_2, \ldots, X_n) \\ &\vdots \\ X_n &= \Phi_n(X_1, X_2, \ldots, X_n) \end{aligned}$$

Funkce Φ_i přiřazují hodnoty na základě závislostí mezi kontrolními body programu. Tyto hodnoty jsou získány pomocí abstraktních transformerů, které modelují instrukce programu, a sloučeny pomocí operátoru spojení. Soustavu lze také reprezentovat jako graf, jehož vrcholy reprezentují kontrolní body programu a hrany reprezentují tokovou závislost mezi dvěma body. Takový graf typicky odpovídá grafu toku řízení analyzovaného programu, ale abstraktní interpretaci lze provádět také nad grafy volání funkcí nebo grafy vytváření vláken.

```
1 int x = 0;
2 while (*) {
3     x = x + 1;
4 }
5 return x;
```

$$X_0 = [\ ,\ ] \qquad X_0 = [\ ,\ ]$$
$$X_1 = [0, 0] \qquad X_1 = [0, 0]$$
$$X_2 = X_1 \sqcup X_3 \qquad X_2 = X_1 \bigtriangledown X_3$$
$$X_3 = X_2 + [1, 1] \qquad X_3 = X_2 + [1, 1]$$
$$X_5 = X_2 \qquad X_5 = X_2$$

Obrázek 1: Ukázka programu a jeho reprezentace jako soustavy rovnic s využitím intervalové domény. Pravá soustava rovnic obsahuje aproximaci pro proměnnou X_2 nutnou pro terminaci analýzy cyklu v programu. Symbol $*$ značí nedeterministickou podmínku, která může modelovat například závislost na vstupu programu.

Příklad 1. Příklad programu a soustavy rovnic pro intervalovou doménu [11] je uveden v obrázku 1. Doména a operace nad ní použité v příkladě jsou definovány následovně:

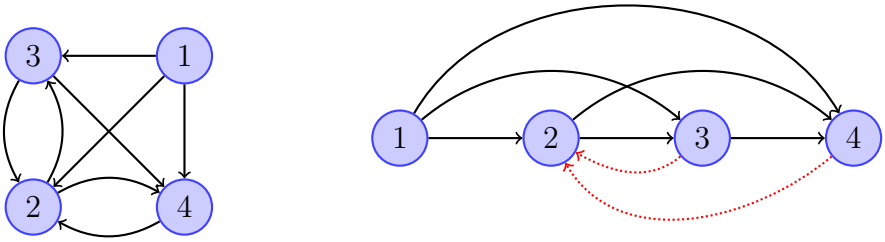
- Abstraktní kontext proměnné x je reprezentován jako interval $[a, b];\ a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$ a značí možné hodnoty $a \leq x \leq b$.
- Operátor spojení je definován jako $[a, b] \sqcup [c, d] = [min(a, c), max(b, d)]$.
- Abstraktní sémantika pro $+$ je definována jako $[a, b] + [c, d] = [a + c, b + d]$.
- Operátor widening je definován následujícím způsobem:

  $$[a, b] \bigtriangledown [c, d] = [\text{pokud } c < a \text{ pak } -\infty \text{ jinak } a, \text{pokud } d > b \text{ pak } +\infty \text{ jinak } b].$$

Řešení soustavy na úrovni grafu probíhá následujícím způsobem. Nejprve je potřeba zvolit množinu $W \subseteq V$ vrcholů pro aproximaci, která je realizována nahrazením operátoru spojení za operátor widening (viz rovnice pro X_2 v obrázku 1). Tato množina musí splňovat podmínku, že každý cyklus v grafu obsahuje alespoň jeden vrchol z W. Nalezení přesného řešení je ekvivalentní s problémem *minimum feedback vertex set* a jedná se o NP-úplný problém [8]. Poté jsou hodnoty jednotlivých proměnných nastaveny na nejmenší hodnotu abstraktní domény a postupně jsou voleny vrcholy grafu, jejichž hodnota je na základě příslušné rovnice aktualizována. Pořadí, ve kterém se vrcholy volí, se nazývá *iterační strategie*. Její vhodnou volbou lze dosáhnout jak větší efektivity, tak přesnosti celého výpočtu. Problémy, které je potřeba vyřešit jsou tedy následující:

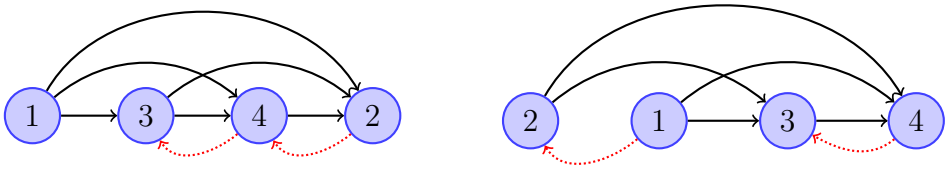
- volba (ideálně co nejmenší) množiny W,
- nalezení efektivní iterační strategie.

Pro oba tyto problémy existují triviální řešení. Aproximaci lze zavést ve všech vrcholech a na základě vlastností definice abstraktní interpretace lze ukázat, že libovolná strategie zahrnující všechny vrcholy grafu bude konvergovat. Takové řešení samozřejmě není ani zdaleka optimální.



1 (2 3 4)

Obrázek 2: Orientovaný graf a znázornění jeho optimálního slabého topologického uspořádání. Červenou tečkovanou čarou jsou vyznačeny hrany zpětné vazby.



1 (3 (4 2))

(2 1)(3 4)

Obrázek 3: Alternativní způsoby slabého topologického uspořádání grafu z obrázku 2. Tato uspořádání obsahují větší počet komponent než je nutné.

Pokud bychom uvažovali pouze acyklické grafy, problém by se stal mnohem jednodušším. Řešením by bylo nalezení topologického uspořádání grafu a následně postupná aplikace jednotlivých rovnic v pořadí daném topologickým uspořádáním. Každá rovnice by tedy byla aplikována právě jednou a nebylo by potřeba zavádět žádnou aproximaci. Pro práci s obecnými orientovanými grafy navrhl François Bourdoncle v článku *Efficient chaotic iteration strategies with widenings* zobecnění topologické uspořádání – slabé topologické uspořádání (WTO), pomocí kterého lze provést dekompozici a uspořádání obecných grafů.

2 Slabé topologické uspořádání

Definice 1. *Hierarchické uspořádání* množiny M je dobře uzávorkovaná permutace M bez dvou po sobě jdoucích "(".

Každé hierarchické uspořádání definuje totální uspořádání $\preceq$ na M, které lze získat odstraněním závorek. Prvky nacházející se mezi dvěma odpovídajícími závorkami se nazývají *komponenty* (pro odlišení od silně souvislých komponent někdy také Bourdoncleho komponenty). Každá komponenta je jednoznačně identifikována svým nejlevějším prvkem, který se nazývá *hlava komponenty*. Pro prvek $m \in M$ definujeme $\omega(m)$ jako množinu hlav všech komponent, které obsahují m a hloubku prvku m jako $\delta(m) = |\omega(m)|$.

Příklad 2. Mějme množinu $M = \{1, 2, 3, 4, 5\}$. Následující posloupnosti jsou příklady jejích hierarchických uspořádání:

$$1\ (2\ 3)\ (4\ 5)$$
$$1\ 2\ 3\ 4\ 5$$
$$(1\ (2\ (3\ (4\ (5)))))$$
$$1\ (2\ (3\ 4))\ 5$$

Na těchto příkladech lze vidět, že definice umožňuje libovolné zanořování komponent nebo jejich skládání za sebe. Je také možné komponenty zcela vynechat a získat tak klasické uspořádání. Přestože definice explicitně nezakazuje prázdné komponenty, typicky se s jich vznikem nepočítá. Podmínka neexistence dvou po sobě jdoucích otevíracích závorek zajišťuje, že každou komponentu lze jednoznačně identifikovat pomocí její hlavy. Například pro poslední zmíněné uspořádání platí, že se skládá ze dvou komponent – komponenty $c_1 = (2\ (3\ 4))$, která obsahuje vnořenou komponentu $c_2 = (3\ 4)$. Tyto komponenty jsou identifikovány svými hlavami 2 a 3. Pro prvek 4 platí $\omega(4) = \{2, 3\}$ a jeho hloubka je tedy $\delta(4) = |\{2, 3\}| = 2$. Pro prvek 5 platí $\delta(5) = |\varnothing| = 0$, protože není obsažen v žádné komponentě.

Definice 2. *Slabé topologické uspořádání (WTO)* grafu $G = (V, E)$ je hierarchické uspořádání na množině V takové, že pro každou hranu $(u, v) \in E$ platí[1]:

$$u \prec v \vee (v \preceq u \wedge v \in \omega(u))$$

Neformálně definice říká, že každá hrana je orientovaná v souladu s uspořádáním (stejně jako u klasického topologického uspořádání), nebo proti jeho směru, ale pouze za podmínky, že cíl hrany v je hlavou některé z komponent, ve které se nachází její zdroj u. Taková hrana se nazývá *hrana zpětné vazby* (feedback edge). Klasické topologické uspořádání je speciálním případem WTO, který neobsahuje hrany zpětné vazby a není tedy potřeba zavádět v něm komponenty. Dalším speciálním případem je triviální WTO, ve kterém každý vrchol definuje vlastní komponentu. Triviální WTO lze vytvořit pro každý orientovaný graf a každou permutaci jeho vrcholů $v_1, v_2, \ldots, v_n$ jako:

$$(v_1(v_2 \ldots (v_{n-1}(v_n)) \ldots))$$

Idea důkazu je následující: pro každou hranu (v_i, v_j) platí jedna z následujících možností: (a) $i < j$, potom hrana respektuje uspořádání, (b) $i \geq j$, potom hrana nerespektuje uspořádání, ale v_i je určitě hlavou některé komponenty, ve které leží v_j.

Příklad 3. Příklad grafu a jeho WTO je ukázán na obrázku 2. Na obrázku 3 jsou znázorněna dvě alternativní uspořádání, která jsou považována za méně vhodná, protože obsahují více komponent. Jak bude vidět v následující kapitole, optimální WTO by mělo mít co nejméně co nejmenších komponent.

[1] Původní definice z [4] uvádí jako podmínku $(u \prec v \wedge v \notin \omega(u)) \vee (v \preceq u \wedge v \in \omega(u))$, lze však ukázat, že tyto podmínky jsou ekvivalentní.

2.1 Využití WTO v abstraktní interpretaci

Pomocí WTO lze získat řešení obou problémů nastíněných v úvodu. Jak již bylo zmíněno, potřebujeme najít množinu $W \subseteq V$ vrcholů pro aproximaci tak, aby každý cyklus v grafu obsahoval alespoň jeden vrchol z této množiny. Lze jednoduše ukázat, že zvolením množiny hlav všech komponent je tento požadavek splněn. Tvrzení vyplývá z faktu, že po odstranění hlav komponent a s nimi spojených hran, zůstávají v grafu pouze hrany (u, v), pro které platí $u \prec v$. Jedná se tedy o topologické uspořádání a takový graf musí být nutně acyklický.

Každé WTO také definuje dvě možné iterační strategie. Takzvanou *iterativní strategii*, která postupně iteruje přes vrcholy v komponentách s hloubkou jedna a její vylepšení *rekurzivní strategii*, která postupně stabilizuje nejvíce vnořené komponenty. Například pro WTO 1 2 (3 (4 5)) (6 7) 8 pak tyto strategie vypadají následovně:

- iterativní strategie: 1 2 [3 4 5]* [6 7]* 8,
- rekurzivní strategie: 1 2 [3 [4 5]*]* [6 7]* 8,

kde $[\cdot]^*$ reprezentuje operátor „iteruj, dokud nedojde ke stabilizaci komponenty“. Stabilizaci komponenty, tedy stav, kdy již žádná iterace nezmění její hodnotu lze detekovat jako stabilizaci hlavy této komponenty. V praxi je nejčastěji využívána rekurzivní strategie [4].

3 Algoritmy pro výpočet WTO

V této kapitole je představeno několik algoritmů pro výpočet slabých topologických uspořádání. Nejprve jednoduchý algoritmus založený na prohledávání do hloubky, který ovšem často vrací nevhodné výsledky. Poté je popsán algoritmus použitelný pro speciální třídu tzv. redukovatelných grafů. Nakonec je představen algoritmus pro obecné orientované grafy založený na rekurzivním rozkladu na silně souvislé komponenty.

Netriviální WTO lze získat v lineárním čase jednoduchou modifikací algoritmu DFS, která je popsána v algoritmu 1. Za jeho běhu je WTO konstruováno následujícím způsobem: kdykoliv je analyzována hranu (u, v), pro kterou platí, že v bylo navštíveno dříve než u, je v vloženo do množiny hlav. Problémem tohoto algoritmu je, že neví, kdy je možné danou komponentu uzavřít a všechny komponenty jsou tak uzavřeny až za posledním vrcholem. Symbolický zápis na řádku 9 reprezentuje konstrukci WTO podle pořadí prvních navštívení vrcholů uložených v poli d tak, že otevře závorku před každou nalezenou hlavou a zavře všechny komponenty za posledním vrcholem. V důsledku tedy algoritmus prodlužuje všechny komponenty a vrací tak méně efektivní iterační strategie. Výsledek je také silně závislý na volbě počátečního vrcholu.

Algoritmus 1: Výpočet WTO založený na prohledávání do hloubky

Vstup : Orientovaný graf $G = (V, E)$, počáteční vrchol $v_0 \in V$
Výstup: WTO grafu G

```
1  DFS-WTO(G)
2      foreach v ∈ V do
3          d[v] ← 0
4          color[v] ← white
5      end
6      time ← 0
7      heads ← ∅
8      DFS-visit(v_0)
9      return (d, heads) as WTO
10
11 DFS-visit(u)
12     color[u] ← black
13     d[u] ← time ← time + 1
14     foreach v ∈ Adj[u] do
15         if d[v] ≤ d[u] then
16             heads ← heads ∪ {v}
17         if color[v] = white then
18             DFS-visit(v)
19     end
```

3.1 Redukovatelné grafy

Redukovatelné grafy (reducible graphs, používá se také pojem dobře strukturované grafy) intuitivně odpovídají grafům toků řízení programů napsaných strukturovaným způsobem [7]. Takové programy buď vůbec nevyužívají konstrukce typu `goto`, nebo je alespoň používají „rozumným způsobem“. Příkladem nestrukturovaného chování je například skok dovnitř cyklu. Přesná definice redukovatelných grafů vyžaduje nejprve představení několika pomocných pojmů. Definice v této kapitole jsou převzaty z [3, 13], kde jsou představeny v kontextu analýzy datového toku (data flow analysis). Podobně jako abstraktní interpretaci, lze tuto analýzu chápat jako řešení soustavy tokových rovnic. V této kapitole jsou dále uvažovány pouze grafy toku řízení – grafy se speciálním vrcholem značeným v_0, pro který platí, že všechny ostatní vrcholy jsou z něj dosažitelné.

Definice 3. Vrchol d dominuje vrcholu v ($d \gg v$), jestliže všechny cesty vedoucí ze vstupního vrcholu v_0 do v vedou přes d. Pro každý vrchol v platí $v \gg v$.

Množinu vrcholů, které dominují vrcholu v značíme $Dom(v)$ a tyto vrcholy nazýváme dominátory vrcholu v. Tyto množiny lze spočítat jako největší (ve smyslu množinové inkluze) řešení následující soustavy rovnic:

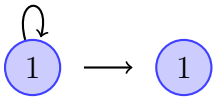
$$\begin{aligned} \mathrm{Dom}(v_0) &= \{v_0\} \\ \mathrm{Dom}(v) &= \{v\} \cup \Big(\bigcap_{p \in preds(v)} \mathrm{Dom}(p) \Big) \end{aligned}$$

kde v_0 je počáteční vrchol. Pro ostatní vrcholy množina obsahuje daný vrchol sjednocený s průnikem dominátorů všech jeho předchůdců. Hranu (u, v), pro kterou platí $v \gg u$ nazýváme v kontextu relace dominance *zpětná hrana*. Tento pojem částečně odpovídá pojmu zpětné hrany v kontextu klasifikace hran na základě prohledávání do hloubky, protože $v \gg u$ zaručuje, že každý průchod do hloubky s počátkem v v_0 nejprve navštíví vrchol v a až po něm vrchol u a klasifikuje hranu (u, v) jako zpětnou. Relace dominance je částečným uspořádáním a jejím zúplněním a doplněním potřebných závorek lze získat WTO. Z tohoto principu vychází i dále popisovaný algoritmus 2.

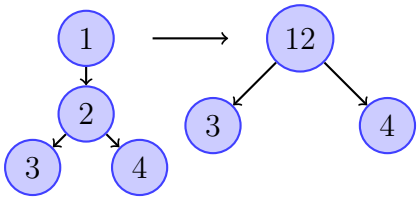
Definice 4. Orientovaný graf $G = (V, E)$ je redukovatelný, jestliže lze množinu hran E rozdělit na dvě disjunktní podmnožiny B a F, pro které platí $B \cup F = E$ a:

- hrany F tvoří acyklický graf, jehož každý vrchol je dosažitelný z v_0,
- pro každou hranu $(u, v) \in B$ platí $v \gg u$ (jedná se tedy o zpětnou hranu v kontextu dominance).

Ekvivalentně lze redukovatelný graf definovat jako graf, který lze transformovat na jediný vrchol pomocí následujících dvou pravidel označovaných jako transformace T_1 a T_2:



Transformace T_1 – odstranění smyčky na libovolném vrcholu grafu

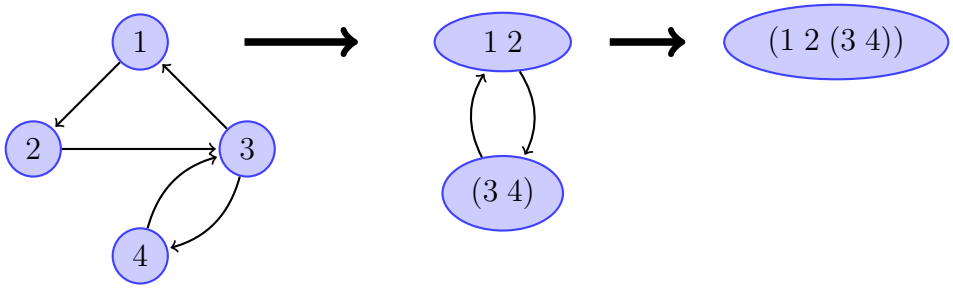


Transformace T_2 – pokud má vrchol jediného předchůdce, je s tímto předchůdcem sloučen

Takto vzniklý graf se nazývá *limitní tokový graf*. Pro redukovatelné grafy konvergence jeho výpočtu nezáleží na pořadí aplikace transformací. Pro neredukovatelné grafy je potřeba pro dosažení limitního grafu zavést další transformaci štěpící vrcholy [3]. V případech redukovatelných grafů lze na konstrukci limitního grafu založit algoritmus pro výpočet WTO. Aplikace transformací je v tomto algoritmu řešena pomocí *intervalů* a *intervalových grafů*.

Definice 5. Nechť $G = (V, E)$ je graf a v_0 jeho vstupní vrchol. *Interval vrcholu* $v \in V$ $(I(v))$ je definován následujícím způsobem:

- $v \in I(v)$,
- pokud všichni předchůdci některého vrcholu $u \neq v_0$ jsou v $I(v)$, pak $u \in I(v)$,
- žádné další vrcholy do $I(v)$ nepatří.



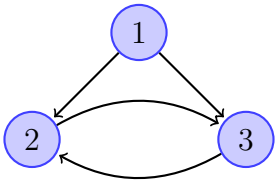
Obrázek 4: Ilustrace výpočtu WTO pomocí konstrukce limitního grafu

Interval $I(v)$ je reprezentován jako posloupnost vrcholů začínající vrcholem v. Pro všechny vrcholy $u \in I(v)$ platí $v \gg u$. Sloučením vrcholů grafu do intervalů je získán intervalový graf. Opakování tohoto procesu pro redukovatelné grafy vede ke vzniku limitního tokového grafu.

Algoritmus 2 v každé iteraci rozdělí graf na disjunktní intervaly a vytvoří z nich intervalový graf. Funkce `construct-interval` konstruuje pro daný vrchol interval přesně podle představené definice. Na řádku 6 je provedena kontrola, zda pro daný interval existuje zpětná hrana a pokud ano, tvoří daný interval komponentu. Funkce `compute-interval-graph` rozdělí graf na intervaly a vrací příslušný intervalový graf. Jakmile se podaří celý graf sloučit do jednoho vrcholu, je takový graf vrácen jako výsledné WTO. Vhodnou implementací tohoto algoritmu lze dosáhnout časové složitosti $\mathcal{O}(|B| \cdot |E| \cdot \alpha(|E|))$, kde B je množina zpětných hran z definice redukovatelného grafu a α značí inverzní funkci k Ackermannově funkci (jedná se o funkci, která roste velmi pomalu a je tedy často považována za konstantu) [4].

Příklad 4. Na obrázku 4 je znázorněn běh popsaného algoritmu. Nejprve je zkonstruován interval pro vstupní vrchol 1: $I(1) = \{1, 2\}$. Vrchol 2 je přidán, protože vrchol 1 je jeho jediný předchůdce. Vrchol 3 již nelze přidat, protože má předchůdce mimo $I(1)$ a je tak pro něj vytvořen samostatný interval $I(3) = \{3, 4\}$. Protože tento interval obsahuje zpětnou hranu $(4, 3)$, je nutné jej uzávorkovat jako komponentu. K dalšímu uzávorkování dojde při druhé iteraci algoritmu. Výsledné WTO (1 2 (3 4)) není optimální, příkladem vhodnějšího uspořádání je (3 1 2 4), které minimalizuje počet komponent. Konstrukcí limitního grafu ovšem toto uspořádání nelze získat, protože konstrukce respektuje relaci dominance a například vrchol 1 musí být vždy v uspořádání první, protože dominuje všem ostatním vrcholům.

Typickým grafem, který není redukovatelný je následující graf:

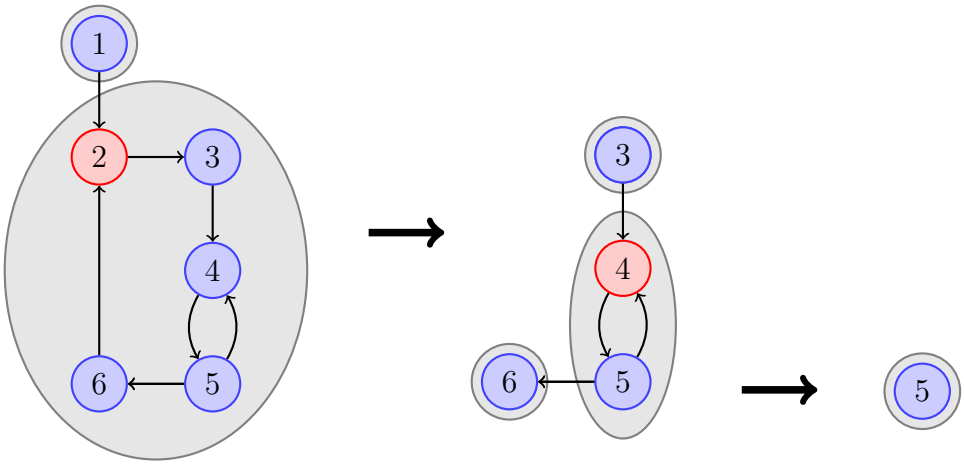


Algoritmus pro tento graf nebude konvergovat, protože interval pro každý z jeho tří vrcholů bude obsahovat vždy pouze daný vrchol a nikdy nedojde k jejich sloučení. Graf obsahuje nestrukturovaný cyklus mezi vrcholy 2 a 3, který nemá jediný vstupní vrchol, který by dominoval druhému vrcholu cyklu (neplatí $2 \gg 3$ ani $3 \gg 2$).

Algoritmus 2: Výpočet WTO založený na konstrukci limitního grafu

Vstup : Redukovatelný graf $G = (V, E)$ a počáteční vrchol $v_0 \in V$
Výstup: WTO grafu G

```
1  function construct-interval(v)
2      I(v) ← {v}
3      while ∃u ∈ V : u ≠ v_0 ∧ ∀p ∈ preds(u) : p ∈ I(v) do
4          I(v) ← I(v) ∪ {u}
5      end
6      if ∃u ∈ V : (u, v) ∈ E then
7          return Component(I(v))
8      else
9          return I(v)
10
11 function compute-interval-graph(G)
12     foreach v ∈ V do
13         selected[v] ← false
14     end
15     I(v_0) ← construct-interval(v_0)
16     foreach v ∈ I(v_0) do
17         selected[v] ← true
18     end
19     while ∃v ∈ V : selected[v] = false ∧ ∃p ∈ preds(v) : selected[p] = true do
20         I(v) ← construct-interval(v)
21         foreach u ∈ I(v) do
22             selected[u] ← true
23         end
24     end
25     return I as interval graph
26
27 function compute-WTO(G)
28     while |V| > 1 do
29         G ←compute-interval-graph(G)
30     end
31     return G as WTO
```



Obrázek 5: Znázornění běhu algoritmu 3. Šedou barvou jsou znázorněny silně souvislé komponenty, červeně jsou označeny vrcholy zvolené jako hlavy při rekurzivním rozkladu komponenty. Výsledné WTO je 1 (2 3 (4 5)) 6.

Algoritmus 3: Rekurzivní rozklad na silně souvislé komponenty

Vstup : Orientovaný graf $G = (V, E)$
Výstup: WTO grafu G

```
1  function recursive-scc(G)
2      C ←compute-sccs(G)
3      components ← ∅
4      foreach c ∈ C do
5          if |c| = 1 then
6              components ← components ∪ {c}
7          else
8              h ← pick some v ∈ c
9              components ← components ∪ {recursive-scc(c \ {h})}
10     end
11     return topological-sort(components)
```

3.2 Rekurzivní rozklad na silně souvislé komponenty

Algoritmus pro obecné orientované grafy představený v [4] je založen na rekurzivním rozkladu na silně souvislé komponenty. Jeho schématická verze je ukázána v algoritmu 3. Ten nejprve pomocí některého ze známých algoritmů nalezne silně souvislé komponenty grafu. Triviální silně souvislé komponenty (tvořené jediným vrcholem) jsou ponechány beze změny, protože takové vrcholy je už na dané úrovni možné uspořádat. Netriviální komponenty je potřeba dále rozkládat – nejprve je pomocí vhodné heuristiky zvolena hlava komponenty a na graf tvořený zbytkem vrcholů je algoritmus rekurzivně zavolán. Rekurzivní volání končí v okamžiku, kdy je možné graf topologicky uspořádat, tedy ve chvíli, kdy je rozložen pouze na triviální silně souvislé komponenty. Při návratu z rekurze je na dané úrovni vždy provedeno (klasické) topologické uspořádání komponent. Ilustrace tohoto postupu je na obrázku 5.

Algoritmus 4: Výpočet WTO založený na Tarjanově algoritmu [4]

Vstup : Orientovaný graf $G = (V, E)$, počáteční vrchol $v_0 \in V$
Výstup: WTO grafu G

```
1  function compute-WTO(G, v_0)
2      foreach u ∈ V do
3          d[u] ← 0
4      end
5      time ← 0
6      wto ← List.empty()
7      visit(v_0, wto)                          ▷ parametr wto je předáván odkazem
8      return wto
9
10 function visit(v, wto)
11     S.push(v)                                ▷ S je globální zásobník
12     head ← d[v] ← time ← time + 1
13     loop ← false
14     foreach u ∈ Adj[v] do
15         if d[u] = 0 then
16             min ← visit(u, wto)
17         else
18             min ← d[u]
19         if min ≤ head then
20             head ← min
21             loop ← true
22         end
23     end
24     if head = d[v] then
25         d[v] ← +∞
26         element = S.pop()
27         if loop then
28             while element ≠ v do
29                 d[element] ← 0
30                 element = S.pop()
31             end
32             wto ←List.prepend(component(v), wto)
33         end
34         else
35             wto ← List.prepend(v, wto)
36         end
37     end
38     return head
39
40 function componenent(v)
41     wto ← List.empty()
42     foreach u ∈ Adj[v] do
43         if d[u] = 0 then
44             visit(u, wto)
45         end
46     end
47     return List.prepend(v, wto)
```

Implementace rekurzivního rozkladu založená na Tarjanově algoritmu

Konkrétní implementace algoritmu 3 je pak často založena na modifikaci Tarjanova algoritmu [14], který hledá silně souvislé komponenty v jejich topologickém pořadí a který sám je založen na prohledávání do hloubky. Jako hlava komponenty je vždy volen první vrchol dané komponenty navštívený při DFS průchodu. Jako datová struktura pro uložení výsledného WTO je typicky použit vnořený seznam. Podle [9] má algoritmus v nejhorším případě kubickou časovou složitost, přesto je v kontextu analýzy programů považován za efektivní.

Původní implementace z článku [4] je popsána v algoritmu 4. Algoritmus nejprve provede v rámci funkce `compute-WTO` inicializaci procházení do hloubky a zavolá funkci `visit` na první vrchol. V ní je aktuálně zpracovávaný vrchol `v` uložen na zásobník a zároveň je na řádku 12 nastaven jako potenciální hlava komponenty. Poté je na řádcích 14 až 22 volán průchod všech jeho sousedů, během kterého je zároveň detekována případná zpětná hrana.

Po skončení průchodu je v proměnné `min` uložena časová známka navštívení prvního vrcholu silně souvislé komponenty, ve které leží `v`. Pokud tato hodnota neodpovídá `d[v]` (řádek 24) je jeho prohledávání ukončeno. Pokud odpovídá (`v` je tedy prvním navštíveným vrcholem nějaké komponenty), mohou nastat dvě situace. Pokud nebyl nalezen cyklus, `v` tvoří triviální silně souvislou komponentu, kterou není nutné rozkládat. Vrchol je tedy na řádku 35 zařazen na začátek konstruovaného WTO.

Druhým případem je situace, kdy byl nalezen cyklus, a tedy existuje netriviální silně souvislá komponenta, kterou je potřeba dále rozkládat. Tato komponenta je tvořena právě vrcholy nalezeného cyklu a vzhledem k charakteru průchodu do hloubky leží tyto vrcholy postupně na vrcholu zásobníku až po vrchol v. Cyklus na řádku 28 postupně provádí jejich vyjmutí ze zásobníku a nulovaní jejich časových známek jako inicializaci pro vnořené DFS. Rozložení komponenty je realizováno voláním funkce `component`, která (vzájemně rekurzivně) volá funkci `visit` na všechny následníky `v` a zahajuje tak vnořené DFS. Po dokončení rozkladu je na řádku 32 výsledné WTO rozkládané komponenty zařazeno jako komponenta WTO celého grafu.

4 Aplikace a další rozšíření

Implementaci algoritmů pro výpočet WTO lze najít v moderních nástrojích implementujících obecný rámec pro abstraktní interpretaci jako jsou Facebook Infer [1] nebo Frama-C [12]. V obou případech implementace vycházejí z algoritmu 4, ale odstraňují vzájemnou rekurzi mezi funkcemi `visit` a `component` na řádcích 32 a 44, která může vést k přetečení zásobníku při výpočtech nad rozsáhlými grafy[2]. Ve Frama-C je slabé topologické uspořádání využito i při řešení tokových rovnic v rámci analýzy datového toku. Obě metody nekonstruují rovnice, ale provádí výpočty přímo nad grafy, které procházejí podle vypočítaných WTO.

[2] `https://github.com/facebook/redex/commit/6bbf8a5ddb`

V článku [9] je představena axiomatická definice WTO a rozšíření konceptů představených v této práci na pojmy částečné hierarchické uspořádání a na jeho základě slabé částečné topologické uspořádání (WPO). S jejich pomocí lze pak implementovat paralelní abstraktní interpretaci pomocí paralelních iteračních strategií. V tomto článku jsou také představeny algoritmy pro výpočet WPO i WTO v *téměř lineárním* čase. Další práce od stejných autorů se pak zabývá i problémem paměťové náročnosti výpočtu [10].

Mimo oblast analýzy programů je koncept WTO zmiňován v souvislosti s analýzou a syntézou obvodů [2], diskrétní simulací [15] nebo obecnými nástroji pro řešení soustav rovnic [6].

Reference

[1] Facebook Infer. `https://fbinfer.com/`.

[2] Agarwal, V., Kankani, K., Rao, R., Bhardwaj, S. and Wang, J. An Efficient Combinationality Check Technique for the Synthesis of Cyclic Combinational Circuits. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*

[3] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] Bourdoncle, F. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and Their Applications.* Springer Berlin Heidelberg, 1993.

[5] Cousot, P. and Cousot, R. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, NY, 1977.

[6] Fecht, Ch. and Seidl, H. A Faster Solver for General Systems of Equations. *Science of Computer Programming*, 1999.

[7] Hecht, M. S. and Ullman, J. D. Flow Graph Reducibility. STOC '72, New York, NY, USA, 1972. Association for Computing Machinery.

[8] Karp, R. M. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations.* Springer US, 1972.

[9] Kim, S. K., Venet, A. J. and Thakur, A. V. Deterministic Parallel Fixpoint Computation. *Proceedings of the ACM on Programming Languages*, 2019.

[10] Kim, S. K., Venet, A. J. and Thakur, A. V. Memory-Efficient Fixpoint Computation. In *Static Analysis.* Springer International Publishing, 2020.

[11] Nielson, F., Nielson, H. and Hankin, Ch. *Principles of Program Analysis*. 1999.

[12] Signoles, J., Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V. and Yakobowski, B. Frama-C: a Software Analysis Perspective. In *Formal Aspects of Computing*, 2012.

[13] Srikant, Y. N. Course on Compiler Design, Department of Computer Science and Automation, Indian Institute of Science. Available online at `https://www.iith.ac.in/~ramakrishna/fc5264/control-flow-analysis.pdf`.

[14] Tarjan, R. Depth-first Search and Linear Graph Algorithms. In *12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*, 1971.

[15] Vahidi, A. *Efficient Analysis of Discrete Event Systems*. PhD thesis, Chalmers University of Technology, 2004.