FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Ověřená implementace Dinitzova algoritmu
Student:	Michal Patera
Vedoucí:	doc. RNDr. Dušan Knop, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Algoritmy pro toky v sítích se studují od počátků informatiky jako vědy a mají dalekosáhlé aplikace v konkrétních úlohách z praxe.

* Nastudujde a popište problematiku toků v orientovaných grafových sítích.
* Nastudujte a popište Dinitzův algoritmus.
* Vhodně implementujte Dinitzův algoritmus v jazyce C.
* Nastudujte framework pro ověřování implementací v jazyce C (Frama-C) a popište jeho partie potřebné pro ověření vaší implementace Dinitzova algoritmu.
* Ověřte vaši implementaci Dinitzova algoritmu, případně diskutujte, proč nebylo možné některé jeho aspekty zcela ověřit.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 13. února 2023 v Praze.

Bakalářská práce

OVĚŘENÁ IMPLEMENTACE DINITZOVA ALGORITMU

Michal Patera

Fakulta informačních technologií
Katedra informační bezpečnosti
Vedoucí: doc. RNDr. Dušan Knop, Ph.D.
11. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat doc. RNDr. Dušan Knop, Ph.D za všechen čas, který věnoval mě a mojí práci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona

V Praze dne 11. května 2023 .....

Abstrakt

Tato práce se zabývá implementaci Dinitzova algoritmu pro hledání maximálního toku v síti a formálnímu důkazu jednotlivých použitích funkcí. Teoretická část seznámí čtenáře s teorií toků v sítích a myšlenkou Dinitzova algoritmu. Představí čtenáři také framework Frama-C pro formální analýzu kódu v programovacím jazyce C a anotační jazyk ACSL, který frameworku pomáhá s verifikací. Praktická část seznámí čtenáře s použitými metodami pro implementaci Dinitzova algoritmu a také objasní použité anotace pro verifikaci algoritmu.

Klíčová slova Frama-C, WP, RTE, ACSL, Dinitzův algoritmus, ověřená implementace

Abstract

This thesis deals with the implementation of the Dinitz algorithm for finding maximum flow in the network and the formal proof of individual functions, that it uses. The theoretical part introduces the reader to the theory of flows in networks and the idea of the Dinitz algorithm. It will also introduce the reader to the Frama-C framework for formal code analysis in the C programming language and the ACSL annotation language, which helps the framework with the verification. The practical part introduces the reader to the methods used to implement the Dinitz algorithm and also clarifies the annotations used for the algorithm verification.

Keywords Frama-C, WP, RTE, ACSL, Dinitz algorithm, verified implementation

Seznam zkratek

Frama-C	Framework for Modular Analysis of C programs
ACSL	ANSI/ISO C Specification Language
RTE	Runtime Error
WP	Weakest Precondition

Úvod

V dnešní době se programy staly nedílnou součástí naší společnosti, řídí naší infrastrukturu počínaje od základních lidských potřeb, například dostupnost pitné vody, zemního plynu až po provoz jaderných elektráren. Stejně tak programy spravují mnoho dalších systémů, které společnost využívá, mezi jinými peněžní systémy jako jsou platby, bankovní účty, daně nebo také dopravní systémy, ať už jsou to semafory na silnicích nebo řízení vlaků, lodí či letadel.

Ve zkratce se programy vyskytují všude kolem nás a o to větší je poté problém, když v některém z těchto programů nastane chyba. Tyto chyby mohou zavinit peněžní ztráty od desítek, nebo stovek tisíc korun až po trvalé následky, či dokonce ztráty na životech. A proto se s každým rokem klade větší důraz na prověřování těchto programů, abychom se chybám vyhnuli.

Jednou z nejčastěji používaných metod je testování programu. Základní myšlenkou tohoto testování je vytvoření velkého množství vstupů, které jsou náhodné a poté menší část vstupů, které byly vytvořeny programátorem nebo testerem a jsou zaměřené na krajní či zranitelnější části programu. Tento postup nám dává jistotu, že ve většině případů program bude fungovat správně, ale nezaručuje nám bezchybnost daného programu a jak už bylo zmíněno i jedna malá chyba může vést na katastrofální následky.

Z tohoto důvodu se tato práce bude zabývat statickou a hlavně formální analýzou programu [1]. Na rozdíl od testování, statická analýza kontroluje program bez nutnosti spuštění a dokáže verifikovat, zda program je korektní. Kvůli nerozhodnutelnosti některých problémů, ne všechny chyby lze označit za vyřešené, nebo nemožné a tedy malé procento chyb zůstává možnou hrozbou, ale dosahujeme zde dostatečných procent korektnosti a bezchybnosti, abychom mohli daný program využívat v důležitých místech.

Formální analýza je rozšířenou statickou analýzou, která využívá matematických operací a tvrzení, aby vyřešila složité problémy v programu. Účelem formální analýzy je tedy dokázat, ať už pomocí vlastních vygenerovaných důkazů, nebo programátorem dodaných, že program je korektní. V této práci budeme využívat formální analýzu k testování korektnosti implementace Dinitzova algoritmu.

Dinitzův algoritmus je jedním z postupů, jak získat maximální tok v sítích. Maximální tok nám představuje množství média např. vody, elektřiny nebo i lidí, které dokážeme přenést či převést z bodu A do bodu B. Mnoho situací z našeho života lze tedy přeformulovat na problém nalezení maximálního toku, jak už bylo zmíněno doručení média, přeprava lidí nebo řízení leteckého provozu.

Hlavním cílem práce je vytvořit implementaci Dinitzova algoritmu, pro hledání maximálního toku v orientované grafové síti, napsáném v programovacím jazyce C a poté ověření implementace Dinitzova algoritmu pomocí anotačního jazyka ACSL a prostředí Frama-C.

Záměrem teoretické části práce je seznámení čtenáře se zakládanými pojmy z oblasti toků v orientovaných grafových sítích a dále popsání a vysvětlení Dinitzova algoritmu.

Účelem praktické části práce je následné popsání a vysvětlení konceptů využitých pro implementaci, ověření Dinitzova algoritmu a diskuze, které části nebo aspekty nebylo možné ověřit.

Kapitola 1
Toky v sítích

V této kapitole uvedeme základní pojmy a definice teorie grafů a toků v sítích, které budou další kapitoly využívat. Poté představíme problém maximálního toku, způsoby řešení a situace z reálného života, které můžeme převést na problém maximálního toku. Hlavním zdrojem informací jsou materiály z předmětu BI-AG2 [2] a kniha *Průvodce labyrintem algoritmů* [3]

1.1 Teorie a definice

▶ **Definice 1.1** (Orientovaný graf [3])**.** Orientovaný graf G *je uspořádaná dvojice* (V, E)*, kde* V *je neprázdná konečná množina vrcholů a* $E \subseteq V \times V$ *je množina orientovaných hran.* Orientovaná hrana $(u, v) \in E$ *je uspořádaná dvojice různých vrcholů* $u, v \in V$*, někdy ji zkráceně značíme* uv*.*

▶ **Definice 1.2** (Síť [2])**.** Sítí *nazveme čtveřici* (G, z, s, c)*, kde* $G = (V, E)$ *je orientovaný graf,* z *a* s *dva různé vrcholy grafu* G *(říkáme jim zdroj a stok) a kapacita* $c\colon E \to \mathbb{R}_Q^+$ *je funkce ohodnocující hrany nezápornými reálnými čísly.*

▶ **Definice 1.3** (Tok v síti [2])**.** Tok v síti (G, z, s, c) *je každá funkce* $f\colon E \to \mathbb{R}_0^+$ *splňující:*

- *Pro každou hranu* $e \in E$ *platí* $0 \leq f(e) \leq c(e)$*.*
- *Pro každý vrchol* $u \in V$ *mimo zdroj a stok platí:*

$$\sum_{(x,u)\in E} f(x, u) = \sum_{(u,y)\in E} f(u, y)$$

▶ **Definice 1.4** (Velikost toku [2])**.** Velikost toku *je:*

$$w(f) = \sum_{(z,x)\in E} f(z, x) - \sum_{(x,z)\in E} f(x, z)$$

Pozor na to, že u orientovaných grafů se pojem zdroj používá pro libovolný vrchol, do nějž nevedou žádné hrany. Zdrojem v síti se označuje právě jeden vrchol a hrany do něj dovolujeme. Analogicky pro stok. [3]

Síť si lze představovat jako soustavu jednosměrných vodovodních trubek, kde každá má svojí šířku, která určuje kolik vody lze danou trubkou transportovat. Každá trubka tedy může být různě veliká a propouštět různé množství vody, zároveň několik trubek se může setkat v jednom místě, takovému místu budeme říkat vrchol. Vrchol má tedy za úkol pouze trubky spojit tak, aby žádná voda nevytekla a sám žádnou vodu neudržoval, síť tedy „těsní"[1]. Dále má dvě vyznačená

[1] Také známe jako Kirchhoffův zákon.

místa – zdroj a stok. Ve zdroji pouštíme vodu dovnitř a ve stoku voda vytéká ven. Tok je poté rozvržení, kterými trubkami voda proudí a jakou intenzitou.

▶ **Definice 1.5** (Průtok v síti [3])**.** *Každé hraně uv přiřadíme její* průtok *f*(uv) = f(uv) − f(vu).*

▶ Pozorování 1.6. Průtoky mají následující vlastnosti:

1. f*(uv) = −f*(vu)
2. f*(uv) ≤ c(uv)
3. f*(uv) ≥ −c(uv)
4. pro všechny vrcholy v, kde v ≠ z,s platí:

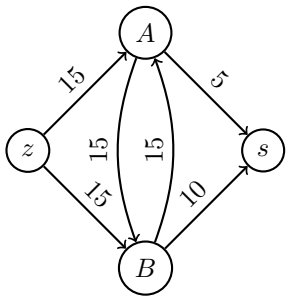
$$\sum_{u:uv\in E} f^*(uv) = 0$$

▶ **Definice 1.7** (Rezerva hrany [3])**.** Rezerva hrany *uv k toku f v síti S = (G, z, s, c) je číslo r(uv) := c(uv) − f(uv) + f(vu). Hraně s nulovou rezervou budeme říkat* nasycená*, hraně s kladnou rezervou* nenasycená*. O cestě řekneme, že je nasycená, pokud je nasycená alespoň jedna její hrana; jinak mají všechny hrany kladné rezervy a cesta je nenasycená.*

▶ **Definice 1.8** (Síť rezerv [3])**.** Síť rezerv *k toku f v síti S = (G, z, s, c) je síť R(S, f) := (G, z, s, r), kde r(e) je rezerva hrany e při toku f.*

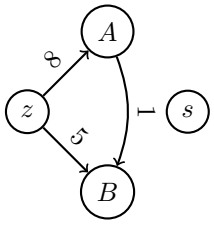
1.2 Problém maximálního toku

Problém maximálního toku si tedy můžeme zadefinovat jako nalezení způsobu jak nejlépe směrovat médium ze zdroje do stoku, tak abychom dosáhli maximálního toku v dané síti. Všimněme si, že může existovat několik způsobů jak maximálního toku dosáhnout obrázek 1.1. Zároveň maximální tok se může rovnat nule, pokud neexistuje cesta složená z hran o kladné kapacitě ze zdroje do stoku obrázek 1.2.



První z možných maximálních toků: z→A(15), A→s(5), A→B(10), B→s(10)
Druhý z možných maximálních toků: z→B(15), B→s(10), B→A(5), A→s(5)

Obrázek 1.1 Ukázka sítě s více možnými maximálními toky



Obrázek 1.2 Ukázka sítě s maximálním tokem o velikosti 0

1.3 Algoritmy pro nalezení maximálního toku

- Ford–Fulkersonův algoritmus [4]

  Ford–Fulkersonův algoritmus je založen na jednoduché myšlence, mějme nulový tok a zlepšujme ho, dokud je to možné a tím dostaneme maximální tok. Jinými slovy tento algoritmus vyhledává nenasycenou cestu z-s, přidá minimální tok, který tuto cestu nasytí a poté proces opakuje dokud další cestu z-s nenajde. To ale přímo nezaručuje správnost, může se stát, že zasytí cestu, která v jiném pořadí by zvětšila aktuální tok a je zapotřebí pro nalezení maximálního toku. Algoritmus tento problém řeší pomocí jednoduchého triku, a to možnost zmenšovat aktuální tok při průchodu proti směru hrany. Neboli přidáme do naší sítě rezervy hrany podle definicí 1.7 a 1.8.

- Edmonds-Karpův algoritmus [5]

  Edmonds-Karpův algoritmus je velice podobný Ford–Fulkersonovu algoritmu, začíná stejně s nulovým tokem a hledá zlepšující cesty, jenže tentokrát nevybírá náhodnou cestu, ale cestu nejkratší.

- Dinitzův algoritmus

  Dinitzovu algoritmu se budeme detailněji zabývat v další kapitole 2, proto ho zde přeskočíme.

- Goldbergův algoritmus [6]

  Goldbergův algoritmus na začátku nastaví všem vrcholům „výšku"a přebytek, výšku si můžeme představit jako třetí osu pro naší síť, tedy doopravdy si můžeme představit síť jako 3D objekt, výška všech vrcholů se ze začátku rovná počtu vrcholů. Přebytek znázorňuje kolik média aktuálně vrchol v sobě drží. Algoritmus běží dokud existuje vrchol s přebytkem. V průběhu algoritmus posouvá přebytek po hranách které nevedou do vrcholů které jsou výš.

- a mnoho dalších jako jsou například algoritm tří indů (MKM) [7], KRT [8] nebo algoritmus Kathuria-Liu-Sidford [9].

Název algoritmu	Časová složitost	Paměťová složitost
Ford-Fulkerson	$O(E \cdot \text{max\_tok})$	$O(V + E)$
Edmonds-Karp	$O(V \cdot E^2)$	$O(V + E)$
Dinitz's	$O(V^2 \cdot E)$	$O(V^2 + E)$
Push-relabel	$O(V^2 \cdot E)$	$O(V^2)$
Push–relabel with FIFO vertex selection	$O(V^3)$	$O(V^2)$
Karp-Stein (capacity scaling)	$O(V \cdot E \cdot \log C)$	$O(V + E)$
FIFO Preflow Push	$O(V^2 \cdot \sqrt{E})$	$O(V^2)$
Boykov-Kolmogorov	$O(V \cdot E^2)$	$O(V^2 + E)$
Malhotra-Kumar-Maheshwari	$O(V^3)$	$O(V^2)$
Ahuja-Orlin	$O(V^2 \cdot E)$	$O(V^2)$
Goldberg-Rao	$O(\min(V^{\frac{2}{3}}, E^{\frac{1}{2}}) \cdot E \cdot \log C)$	$O(V + E)$
Cheriyan-Maheshwari-Mehlhorn	$O(E \cdot \log^2 V)$	$O(V + E)$
King-Rao-Tarjan	$O(V^{\frac{3}{2}} \cdot \log V)$	$O(V^2)$

Tabulka 1.1 Porovnání algoritmů pro řešení problému maximálního toku

Poznámka: E značí počet hran v grafu, V značí počet vrcholů v grafu a C značí kapacitu hran v grafu.

Kapitola 2
Dinitzův algoritmus

Dinitzův algoritmus, také znám jako Dinicův algoritmus je efektivním způsobem pro řešení problému maximálního toku v síti S = (G, z, s, c). Tento algoritmus byl vyvinut Yefimem Dinitzem v roce 1970 a je založen na myšlence používání blokujících toků a pročištěné sítě.

▶ **Definice 2.1** (Blokující tok)**.** Blokujícím tokem *(ang. Blocking Flow) nazýváme tok, který na každé cestě uv obsahuje alespoň jednu hranu nasycenou, neboli můžeme blokující tok také brát jako tok o minimální velikosti, který nasytí cesty uv z definice 1.7.*

▶ **Definice 2.2** (Pročištěná síť [3])**.** Pročištěnou sítí *(ang. Layered network) budeme nazývat síť T = (H, z, s, c), kde H je podgrafem grafu G sítě S = (G, z, s, c) a podgraf H obsahuje pouze vrcholy a hrany, které leží na nejkratší cestách zs.*

Jak tedy Dinitzův algoritmus využívá blokující toky a pročištěné sítě? Pokud bychom chtěli najít blokující tok na celé síti, tak podle definice najdeme tok, který nasytí určité hrany tak, aby neexistovala žádná další nenasycená cesta ze zdroje do stoku, neboli našli bychom maximální tok v síti. Z tohoto důvodu nebudeme hledat blokující tok na celé síti, ale na pročištěné síti, pokud se zase zamyslíme co to znamená, tak zjistíme, že dostaneme tok o velikosti rovné součtu kapacit hran tak, aby všechny nejkratší cesty zs byly nasyceny.

Pokud znova aplikujeme pročištění sítě s novými hranami, které byly nasyceny, tak dostaneme další pročištěnou síť s novými nenasycenými nejkratšími cestami, na které znova najdeme blokující tok a tento postup opakujeme dokud existuje nenasycená cesta zs v síti. Pomocí těchto blokujících toků poté zlepšujeme maximální tok.
Postup Dinitzova algoritmu [3]:
Vstup: Síť $S = (G, z, s, c)$

- $f \leftarrow$ nulový tok
- Opakujeme:

1. Sestrojíme pročištěnou síť H ze sítě S.
2. Najdeme nejkratší cesty v síti H.
3. Pokud žádná taková cesta neexistuje, zastavíme se a vrátíme tok f.
4. $g \leftarrow$ blokující tok v R.
5. Zlepšíme tok f pomocí g.
6. Upravíme aktuální síť S podle toku g.

Výstup: Maximální tok f.

Dovysvětlíme ještě čištění sítě, jak už jsme zmínili, jedná se o síť tvořenou podgrafem H z grafu G. Způsob jakým můžeme jednoduše pročištěnou síť vytvořit, se zakládá na myšlence rozdělení vrcholů sítě do skupin podle vzdálenosti od zdroje. Mějme tedy rozdělené vrcholy do vhodných vrstev, poté odeberme všechny vrstvy, které mají vzdálenost větší než vzdálenost stoku. Dále odeberme všechny hrany které vedly do těchto vrcholů, odeberme také všechny hrany, které vedou mezi vrcholy stejné vrstvy a hrany které vedou do vrstev bližších ke zdroji. Na konec odeberme všechny vrcholy, z kterých nevede žádná hrana.[1]

2.1 Implementace

Existuje několik způsobů implementace Dinitzova algoritmu a volba konkrétní implementace může záviset na programovacím jazyce, použitých datových strukturách a konkrétních požadavcích problému.

1. *Výběr datových struktur*: Pro efektivní implementaci Dinicova algoritmu je důležité vybrat vhodné datové struktury pro reprezentaci grafu, kapacit hran a toku. Běžně se používají matice sousednosti, seznamy sousedů nebo seznamy hran.

2. *Algoritmus pro hledání nejkratší cesty*: Pro konstrukci vrstevnicového grafu je potřeba najít nejkratší cesty od zdroje k ostatním vrcholům v síti. Můžeme použít algoritmy, jako je BFS [10] nebo Dijkstrův algoritmus [11], v závislosti na našich požadavcích a omezeních.

3. *Hledání blokujícího toku*: Pro hledání blokujícího toku v pročištěné síti lze použít DFS [12] nebo BFS [10] s vhodnými úpravami. V některých implementacích se používají také augmentační cesty nebo push-relabel metoda.

4. *Aktualizace toku a kapacity hran*: Po nalezení blokujícího toku je potřeba aktualizovat tok v síti a kapacity hran. To lze provést pomocí jednoduchých cyklů nebo vektorizovaných operací, v závislosti na použitých datových strukturách a programovacím jazyce.

[1] Nasycené hrany, budeme považovat za smazané.

2.2 Použití

Dinitzův algoritmus má širokou škálu praktických aplikací v reálném světě. Následující příklady ilustrují některé z jeho současných použití:

1. *Optimalizace dopravních sítí [13]*: Algoritmus můžeme použit k optimalizaci dopravních sítí, například pro řízení dopravy ve velkých městech nebo zajištění efektivního rozdělování zásob. Algoritmus může pomoci najít nejlepší rozložení zdrojů a cest, aby byl zajištěn maximální tok zboží či dopravy s minimálními náklady a zpožděním.

2. *Řízení vodních zdrojů*: Algoritmus lze aplikovat na řízení vodních zdrojů, jako jsou řeky, přehrady a zavlažovací systémy. Algoritmus může pomoci určit optimální rozdělení vody mezi různé oblasti, aby byla zajištěna maximální účinnost a minimalizace ztrát.

3. *Telekomunikace [14]*: V oblasti telekomunikací se algoritmus používá pro optimalizaci přenosu dat v sítích a pro řízení zdrojů, jako je šířka pásma nebo kapacita spojení. Algoritmus může napomoci při navrhování sítí tak, aby bylo zajištěno maximální propustnost a minimální zpoždění.

4. *Plánování a analýza sítě [15]*: Algoritmus můžeme použít pro plánování a analýzu sítě v různých oborech, jako je například výroba, logistika nebo energetika. Algoritmus může pomoci identifikovat úzká hrdla, optimalizovat zdroje a zlepšit celkovou efektivitu sítě.

5. *Optimalizace zpracování informací[16]*: V oblasti zpracování informací může být algoritmus použit pro optimalizaci paralelního zpracování úkolů, distribuci zdrojů nebo plánování procesů. Algoritmus může napomoci při navrhování systémů tak, aby bylo dosaženo maximálního výkonu a minimálního času zpracování.

Kapitola 3

Verifikační prostředí Frama-C

Frama-C je sofistikovaný analytický nástroj založený na frameworku, který je navržen pro analýzu zdrojového kódu napsaného v jazyce C. Tento nástroj byl nejdříve vyvinut pro zlepšení bezpečnosti a provozu jaderných elektráren společností Inria, francouzským národním institutem pro výzkum v oblasti informatiky a matematického modelování, a jeho hlavním cílem dnes je zlepšit kvalitu, bezpečnost a spolehlivost softwaru. Frama-C poskytuje sadu modulárních a rozšiřitelných pluginů, které umožňují provádět různé druhy analýz, jako je formální analýza, statická analýza, abstraktní interpretace, kontrola běhových chyb nebo generování testovacích dat. My budeme framework Frama-C využívat hlavně pro statickou a formální analýzu programu napsaného v jazyce C, budeme také využívat dva z mnoha možných pluginů – plugin WP a plugin RTE.

3.1 Plugin WP

Plugin WP (Weakest Precondition) [17] je součástí Frama-C a slouží k formální verifikaci zdrojového kódu napsaného v jazyce C. WP využívá techniku nejslabších předpokladů (ang. weakest preconditions) pro ověření, zda daný program splňuje formální specifikace zapsané v jazyce ACSL (ANSI/ISO C Specification Language). Tato technika spočívá ve vytváření matematických důkazů, které zachycují chování programu a umožňují ověřit splnění požadavků specifikovaných v ACSL.

WP podporuje různé druhy nástrojů pro automatické i interaktivní důkazy, jako jsou SMT (Satisfiability Modulo Theories) solvery nebo interaktivní důkazové asistenty, například Coq. Plugin WP je navržen tak, aby byl modulární a rozšiřitelný, což umožňuje snadno přidávat podporu pro další důkazové nástroje nebo techniky.

3.2 Plugin RTE

Plugin RTE (Runtime Error) je další komponenta Frama-C, která se zaměřuje na kontrolu běhových chyb v programu napsaném v jazyce C. Tento plugin analyzuje zdrojový kód a automaticky generuje ACSL anotace pro kontrolu běhových chyb, jako jsou přetečení čísel, dělení nulou, neplatné ukazatele nebo přístup mimo rozsah pole.

RTE [18] využívá Frama-C framework pro propagaci těchto anotací v kódu a ověření jejich splnění pomocí dostupných analýz a pluginů, jako je WP [17] nebo E-ACSL [19]. Plugin RTE umožňuje identifikovat potenciální běhové chyby a nebezpečné chování programu, což přispívá ke zlepšení kvality, bezpečnosti a spolehlivosti softwaru.

Ověření programu potřebuje nejen framework Frama-C, zvolené pluginy, ale i takzvané řešiče (ang. solver) založené na teorii splnitelnosti modulo teorie (SMT – satisfiability modulo theories). Tyto řešiče umožňují automatické dokazování vlastností programu, které vycházejí z anotací ve zdrojovém kódu a matematických teorií. Frama-C podporuje několik SMT řešičů, jako jsou Alt-Ergo, CVC4 a Z3. Každý řešič má své specifické vlastnosti a výhody, což nám umožňuje si vybrat nejvhodnější řešič pro konkrétní úlohu ověřování. My budeme v naší práci používat právě zmíněné tři řešiče Alt-Ergo, CVC4 a Z3.

3.3 Solver Alt-Ergo

Alt-Ergo [20] je open-source automatický řešič teorií (SMT solver) vyvinutý ve francouzském výzkumném institutu INRIA. Je založen na Satisfiability Modulo Theories (SMT) a je navržen tak, aby podporoval různé datové typy a struktury, včetně celočíselných a reálných aritmetik, polí a typů výčtů. Řešič je napsán v OCaml a lze jej snadno integrovat do ověřovacích platforem softwaru, jako je Frama-C. Používá se v různých aplikacích, jako je například verifikační software Why3 [21], programovací jazyk SPARK [22] nebo software pro vývoj programů Atelier-B [23].

3.4 Solver CVC4

CVC4 [24] je široce používaný open-source SMT řešič vyvinutý týmem výzkumníků ze Stanfordské university a Iowa univerzity. Je čtvrtým v řadě řešitelů Cooperating Validity Checker (CVC). CVC4 byl navržen se zaměřením na výkon, škálovatelnost a robustnost a byl úspěšně aplikován v různých projektech ověřování softwaru a formálních metod, mezi jinými byl použit pro ověřovač programů Dafny vyvinutý společností Microsoft Research [25] nebo pro ověření hardwaru sadou nástrojů SymbiYosys [26].

3.5 Solver Z3

Z3 [27] je vysoce výkonný řešič vyvinutý společností Microsoft Research. Řešič je známý svou efektivitou a snadnou integrací, díky čemuž je oblíbenou volbou pro mnoho ověřovacích úloh. Z3 je široce používán při ověřování softwaru, analýze programů a kontrole modelů a byl integrován do mnoha nástrojů a rámců, např. ověřovací infrastruktura Viper [28] nebo Metatheory of Algebraic Process Theories (Maude) system [29], který se používá ke specifikaci a analýze široké škály formálních modelů.

Kapitola 4
Anotační jazyk ACSL

Jedním z klíčových aspektů Frama-C je jeho schopnost provádět formální analýzu pomocí specifikace ACSL (ANSI/ISO C Specification Language) [30]. ACSL je bohatý a výkonný jazyk pro formální analýzu, který umožňuje popsat chování programu a jeho očekávané vlastnosti na základě matematických konstrukcí a logických výrazů. Frama-C dokáže analyzovat zdrojový kód spolu se specifikací ACSL a ověřit, zda kód splňuje dané požadavky. Tímto způsobem je možné odhalit potenciální chyby, nekonzistence nebo nebezpečné chování, které by mohly vést k selhání softwaru nebo bezpečnostním problémům.

4.1 Anotace jazyka ACSL

Jazyk ACSL umožňuje popsání očekávaného chování programu pomocí kontraktů. Kontrakty se skládají z předpodmínek (ang. preconditions), popisující očekávaný stav před voláním funkce a z následujících podmínek (ang. postconditions), které popisují očekávaný stav po dokončení funkce. Frama-C tak může analyzovat a ověřovat správnost programu, ale i správnost jednotlivých funkcí.

Jazyk ACSL používá mnoho různých klíčových slov (ang. keywords), kterými kategorizuje tvrzení, která se za nimi nachází. Rychle zde uvedeme a lehce vysvětlíme některá klíčová slova, které budeme používat v našem ověření Dinitzova algoritmu používat.

- *@requires*: Specifikuje předpoklady, které musí být splněny před zavoláním funkce. Tato klauzule se používá k omezení stavů, ve kterých může být funkce bezpečně volána. Předpoklady mohou zahrnovat podmínky na vstupních hodnotách nebo stavech objektů.
- *@ensures*: Popisuje podmínky, které musí být splněny po úspěšném ukončení funkce. Tyto podmínky mohou zahrnovat očekávané výsledky nebo změny stavu.
- *@assigns*: Udává, jaké proměnné mohou být změněny funkcí. Tato klauzule je důležitá pro omezení účinků funkce na její okolí.
- *@allocates*: Popisuje paměť, kterou funkce alokuje. Tato klauzule je užitečná pro sledování alokace paměti a její správné uvolnění.
- *@frees*: Popisuje paměť, kterou funkce uvolňuje. Tato klauzule je důležitá pro zajištění, že všechna alokovaná paměť je správně uvolněna.
- *@loop_invariant*: Popisuje podmínky, které musí být splněny před a po každé iteraci smyčky. Pokud bychom vzali ukázkový kód 4.1, tak jako invariant cyklu bychom napsali $0 \leq a < 10$, tedy celkem jako /*@ loop invariant 0 <= a <= 10; @*/. Jak tedy vidíme, invarianta musí také obsahovat hodnotu i po skončení cyklu.

- *@loop_assigns*: Udává, jaké proměnné mohou být změněny smyčkou. Z ukázkového kódu 4.1, by se jednalo o dvě proměnné: *a* a *i*.
- *@loop_variant*: Udává funkci, která klesá při každé iteraci smyčky a je nezáporná. Pro náš ukázkový kód 4.1 máme více možností, např.
  /*@ loop variant 10 - i;@*/ nebo /*@ loop variant 10 - a;@*/
- *@axiomatic*: Definuje soubor axiómů, které popisují logické vlastnosti, na kterých lze stavět důkazy. Tato klauzule umožňuje vytvářet složitější logické konstrukty pro dokazování vlastností kódu.
- *@axiom*: Definuje jednotlivý axiom, který je součástí axiomatického bloku a popisuje logickou vlastnost.
- *@predicate*: Definuje logickou funkci, která může být použita jako nástroj pro ověřování. Predikáty se používají k popisu vlastností nebo stavů, které jsou buď pravdivé nebo nepravdivé.
- *@assert*: Kontroluje, zda je daná podmínka splněna, a pokud ne, generuje chybu ověřování. Tato klauzule je užitečná pro ověřování, že kód v určitých bodech splňuje očekávané podmínky.
- *@logic*: Definuje logickou funkci, která může být použita v axiomatických bloků, predikátů a dalších logických výrazů. Tyto logické funkce mohou být použity k vytváření komplexnějších logických výrazů pro ověřování vlastností kódu.
- *@reads*: Určuje, které proměnné nebo paměťové oblasti může logická funkce číst.
- *@old*: Klíčové slovo se používá v kontextu klauzule @ensures pro označení hodnoty proměnné před zavoláním funkce. To umožňuje srovnání stavu před a po volání funkce.
- *@Post*: Klíčové slovo Post se používá v klauzuli @ensures pro označení podmínek, které by měly platit po skončení funkce.
- *@at*: Klíčové slovo at se používá k označení hodnoty proměnné v určitém bodě programu. Je to užitečné pro popisování chování programu v čase, například při sledování hodnot proměnných v různých částech smyčky.

Výpis kódu 4.1 Ukázkový for cyklus v jazyce C

```
int a = 0;
for(int i = 0; i < 10; ++i) {
    a++;
}
```

Kapitola 5

Implementace a ověření algoritmu

V této kapitole si popíšeme implementaci a ověření Dinitzova algoritmu pro nalezení maximálního toku v síti, který jsme v minule kapitole 2 popsali z teoretické stránky a nyní ho představíme z praktické. Náš Dinitzův algoritmus jsme implementovali v programovacím jazykem C a to z důvodu pozdější verifikace frameworkem Frama-C, který toto vyžaduje.

Naši implementaci jsme rozdělili do tří souborů. Soubor main.c, který obsahuje načítání vstupu od uživatele a uložení dat do správné struktury, s kterou dále náš algoritmus pracuje. Hlavičkový soubor dinitz.h, který obsahuje definice pomocných metod a pomocných struktur a jako poslední máme soubor dinitz.c, ve kterém je kód, který implementuje dříve definované funkce z dinitz.h.

5.1 Implementace

Nyní si ukážeme kód implementace a přesněji popíšeme rozhodnutí, které vedly k vybraným technikám, které jsme využili. Začneme od souboru main.c, abychom si uvedli jaké data náš algoritmus přijímá a s jakými podmínkami pracujeme.

5.1.1 Main.c

Nejdůležitější části souboru main.c je funkce readInput z výpisu kódu 5.1, tato funkce čte uživatelem zadané data, a poté je ukládá do pro nás vhodné struktury – matice sousednosti. Zároveň tato funkce hlídá námi předurčené požadavky, neboli chceme aby graf měl alespoň 2 vrcholy, abychom mohli mít zdroj a stok jako dva různé vrcholy.[1]

Dále jsme si stanovili, že budeme uvažovat grafy pouze o maximálním počtů vrcholů rovném, nebo menším než 300 000. Tuto velikost grafů jsme si vybrali, aby tento kód měl stále využití v reálném světe, ale zároveň aby doba běhu programu byla stále akceptovatelná.

Zároveň nám, ale toto rozhodnutí přináší jiné následky. Znamená to pro nás využití dynamicky alokované paměti, abychom mohli graf takové velikosti uložit. S tím se váže nepříjemná vlastnost frameworku Frama-C a to, že klíčová slova pro ověřování frees a alocates z kapitoly 4.1 nejsou implementovány. A proto, jak ještě uvidíme části kódu spojené, nebo úzce spojené s dynamickou alokací paměti nelze plně ověřit.

Soubor main.c zajišťuje výpis výsledku našeho algoritmu a dále obsahuje funkci cleanUp, která uvolní paměť.

[1] Vždycky považujeme první vrchol jako zdroj a poslední vrchol jako stok.

Výpis kódu 5.1 Soubor main.c

```
int readInput( double ***graph)
{
    int nodes;
    // Reading the amount of nodes and allocating
    printf("Insert the number of nodes.\n");
    if(scanf("%d", &nodes) != 1){
        printf("Invalid number of nodes.\n");
        return -1;
    }
    if (nodes < 2 || nodes > 300000){
        printf("Invalid number of nodes.\n");
        return -1;
    }

    *graph = (double**) malloc((nodes) * sizeof(double *));
    if (*graph == NULL){
        return -1;
    }

    for (int i = 0; i < (nodes); ++i)
    {
        (*graph)[i] = (double*) calloc((nodes), sizeof(double));
        if ((*graph)[i] == NULL){
            return -1;
        }
    }

    // Reading all the edges and saving them to the 2D array
    int start, end;
    double cap;
    printf("Insert the edges of the graph.\n");
    while (scanf("%d %d %lf", &start, &end, &cap) == 3){
        if (start < 0 || end < 0 || cap <= 0 || start >= (nodes)
        || end >= (nodes)){
            printf("This edge is invalid.\n");
        }
        else if ((*graph)[start][end] != 0.0){
            printf("This edge already exists.\n");
        }
        else{
            (*graph)[start][end] = cap;
        }
    }
    return nodes;
}
```

5.1.2 Dinitz.h/Dinitz.c

Nyní si popíšeme hlavní soubory našeho programu. Začneme od nejdůležitější funkce dinitzMaxFlow 5.2 a budeme pomalu představovat další pomocné funkce. Naše hlavní funkce jako první připraví další pole určené pro pamatování si úrovní vrcholů a zavolá BFS 5.3, kterému toto pole předá.

BFS má za úkol prohledat do šířky zadaný graf, nastavit úrovně všech dostupných vrcholů a tedy i zjistit zda existuje cesta zs. BFS také využívá naší pomocnou strukturu Node 5.5 a funkce insertHead 5.7 a popHead 5.6, které společně simulují frontu pomocí spojového seznamu.

Po návratu BFS, pokud existuje cesta zs, tak opakovaně voláme funkci DFS 5.4. Naše prohledávání do hloubky je upravené, dovolujeme DFS cestovat pouze po hranách, které vedou z vrcholu o nižší úrovni do vrcholu s úrovni o jedna vyšší. Zároveň si pamatujeme cestu, kterou jsme přišli, navštívené vrcholy a velikost toku, který se rovná nejmenší kapacitě hrany na cestě zs. DFS poté upravuje aktuální graf a maže hrany, které jsou nasyceny.

V naší implementaci tedy nevytváříme pročištěnou síť, ale pouze přiřazujeme úrovně a mažeme nasycené hrany. Pro DFS to znamená, že může prohledávat delší dobu, protože nemažeme slepé uličky, ale zvolili jsme tento postup, abychom se vyhnuli dalším alokacím dynamické paměti, které vytváří problémy s ověřovacím frameworkem.

Výpis kódu 5.2 kód funkce Dinitz Max Flow

```
double dinitzMaxFlow(int nodes, double **graph){
    double total = 0;
    int *vertexLevel = (int*) malloc(nodes * sizeof(int));
    int *visited;
    // Run BFS to check if flow is possible and set the levels of nodes
    while (BFS(0, nodes - 1, graph, vertexLevel))
    {
        visited = (int*) calloc(nodes, sizeof(int));
        double flow;
        while (1)
        {
            flow = DFS(0, nodes - 1, graph, vertexLevel, visited, DBL_MAX);
            if (flow == 0.0)
            {
                break;
            }
            else
            {
                if (DBL_MAX - total < flow) {
                    total = DBL_MAX;
                    break;
                } else {
                    total += flow;
                }
            }
            // Reset visited array
            memset(visited, 0, nodes * sizeof(int));
        }
        free(visited);
    }
    free(vertexLevel);
    // Return max flow
    return total;
}
```

Výpis kódu 5.3 kód funkce BFS

```
int BFS(int start, int end, double **graph, int *levels){
    for (int i = 0; i <= end; ++i) {
        levels[i] = -1;
    }
    levels[start] = 0;
    Node* queueHead = NULL;
    insertHead(&queueHead, start);
    int cur;
    while(queueHead != NULL)
    {
        cur = queueHead->data;
        popHead(&queueHead);
        for (int i = 0; i <= end; ++i)
        {
            if(graph[cur][i] > 0)
            {
                if(levels[i] < 0){
                    levels[i] = levels[cur] + 1;
                    insertHead(&queueHead, i);
                }
            }
        }
    }
    if (levels[end] < 0){
        return 0;
    }
    else {
        return 1;
    }
}
```

Výpis kódu 5.4 kód funkce DFS

```
double DFS (int cur, int end, double **graph,
int *levels, int *visited, double flow){
    if (cur == end){
        return flow;
    }
    visited[cur] = 1;
    for (int i = 0; i <= end; ++i) {
        if (!visited[i] && graph[cur][i] > 0
        && levels[i] == levels[cur] + 1){
            double augmentingFlow = DFS(i, end, graph,
                                        levels, visited,
                                        min(flow, graph[cur][i]));
            if (augmentingFlow > 0){
                graph[cur][i] -= augmentingFlow;
                graph[i][cur] += augmentingFlow;
                return augmentingFlow;
            }
        }
    }
    return 0;
}
```

Výpis kódu 5.5 kód pomocné struktury Node

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;
```

Výpis kódu 5.6 kód funkce popHead

```
void popHead(Node** head_ref) {
    Node* nextHead = (*head_ref)->next;
    free(*head_ref);
    (*head_ref) = nextHead;
}
```

Výpis kódu 5.7 kód funkce insertHead

```
void insertHead(Node** head_ref, int new_data) {
    Node* next = (*head_ref);
    (*head_ref) =  (Node*) malloc(sizeof(Node));
    (*head_ref)->data = new_data;
    (*head_ref)->next = next;
}
```

5.2 Ověření

Nyní už máme naší implementaci kódu a chceme jí ověřit, zavedeme tedy do kódu anotace jazyka ACSL, takové anotace uvádíme pomocí znaků /*@ a @*/. Jedná se o blokové komentáře programovacího jazyka C doplněné o znak @. Pokud anotace jsou na více řádků tak nemusíme na začátek každého řádku uvádět @, ale my to tak budeme dělat, aby bylo jasné kde máme anotace pro framework Frama-C.

Nyní už začneme s uváděním funkcí a jejich kontraktů, začneme od menších funkcí, kde kontrakty bývají jednodušší a budeme se posouvat k funkcím složitějším, které často závisí na předešlých funkcích.

5.2.1 Funkce insertHead

Výpis kódu 5.8 anotace insertHead

```
/*@ requires \valid(head_ref) && (*head_ref) != \null;
  @ ensures *head_ref != \null && (*head_ref)->data == new_data
  && (*head_ref)->next == \old(*head_ref) && \fresh(*head_ref, sizeof(Node))
  && \separated(*head_ref, \old(*head_ref));
  @ allocates *head_ref;
  @ assigns *head_ref, (*head_ref)->data, (*head_ref)->next;
  @*/
void insertHead(Node **head_ref, int new_data){
...
}
```

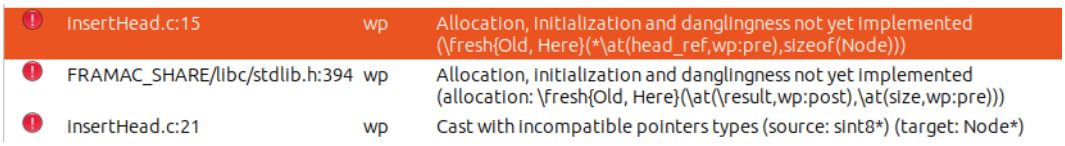
Popišme anotace funkce insertHead, funkce požaduje, aby ukazatel head_ref byl validní a zároveň se nerovnal null, to pro nás znamená, že chceme, aby paměť, na kterou ukazatel ukazuje byla validní ke čtení a zapisování.

Dále funkce zajišťuje, že hodnota head_ref nebude null a zároveň, že složka data v Node, na který ukazuje se bude rovnat parametru new_data, který tato funkce dostala a složka next se bude rovnat staré hodnotě *head_ref, neboli bude to ukazatel na další prvek v seznamu.

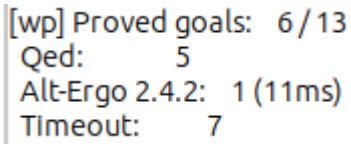
Klíčová slova fresh a separated nám pouze říkají, že head_ref ukazuje na nově alokovaný blok o velikosti Node, a že bloky paměti na které ukazují head_ref a starý head_ref se nepřekrývají.

Dále máme informaci o tom, že tato funkce alokuje paměť, na kterou ukazuje head_ref a jako poslední máme klauzuli assigns, která nám říká, že tato funkce manipuluje s pamětí, v tomto případě proměnných *head_ref, (*head_ref)->data a (*head_ref)->next.

Pokud se pokusíme ověřit pouze tuto část programu tak dostaneme upozornění z obrázku 5.1 a část našeho kódu nebude možné z tohoto důvodu ověřit. 5.2



Obrázek 5.1 Problémy s dynamickou alokací paměti ve Frama-C



Obrázek 5.2 Ověření insertHead

5.2.2 Funkce popHead

Výpis kódu 5.9 anotace popHead

```
/*@ predicate is_valid_linked_list(Node* head) =
  @   head == \null || (is_valid_linked_list(head->next) && \valid(head));
  @*/

/*@ requires \valid(head_ref) && *head_ref != \null
    && is_valid_linked_list(*head_ref);
  @ assigns *head_ref;
  @ ensures \at(*head_ref,Post) == \old((*head_ref)->next)
    && !\valid(\old(*head_ref));
  @*/
void popHead(Node** head_ref) {
...
}
```

Nyní si ukažme anotace funkce popHead, na první pohled vidíme část anotací, které se v minulé funkci neobjevily. Jedná se o anotace predicate, jak už jsme zmínili dříve v kapitole 4.1 tato anotace nás informuje o funkci, která může být použitá k ověřování.

V tomto případě se jedná o funkci, která dostane hlavičku naší fronty v podobě spojového seznamu a rekurzivně se volá na další prvky dokud nenarazí na null a v každém kroku kontroluje, zda je možné číst a zapisovat do daného prvku fronty.

Dále máme již nám známe anotace ohledně požadování validních vstupních parametrů funkce a použití zmíněné predicate funkce ke kontrole fronty. Anotace nám také říkají, že funkce mění proměnnou *head_ref, uvolňuje paměť na kterou ukazovala dřívější hodnota *head_ref.

V části anotací, kde máme, co funkce zaručuje dostáváme nové klíčové slovo at, které nám říká, že se koukneme na hodnotu dané proměnné až po daném bodě v kódu, který označíme pomocí štítku, v tomto případě Post.

Slovo at, ale už tak doopravdy známe, pokud se podíváme na slovo old(x), tak se jedná pouze o zkrácenou formu at(x, Old). Slovo old už jsme vysvětlili, že se jedná o hodnotu proměnné před provedením kódu funkce a slovíčko Post je jejím opakem, tedy hodnota po skončení funkce. Celkově tedy zaručujeme, že poté co funkce skončí tak hodnota *head_ref se rovná ukazateli na další prvek z minulé hlavičky.

A jako poslední, že paměť, na kterou ukazovala stará hodnota *head_ref nyní už není validní, neboli nemůžeme jí číst ani do ní psát. V programovacím jazyce C, všechny tyto vlastnosti vyjadřujeme funkcí free.

Nyní už můžeme ověřit naší funkci popHead obrázek 5.3, očekáváme podobné výsledky jako s funkcí insertHead, protože nadále pracujeme s dynamickou alokací paměti. Zároveň si ukážeme jak Frama-C transformuje námi dodané anotace a také jak plugin RTE 3.2 přidává vlastní ověřovací anotace obrázek 5.4.

```
[wp] Proved goals:   6 / 10
 Qed:             3
 Alt-Ergo 2.4.2:  3 (13ms-18ms-25ms)
 Timeout:         4
```

Obrázek 5.3 Ověření popHead

```
/*@
predicate is_valid_linked_list{L}(Node *head) =
  \at(head ≡ \null ∨ (is_valid_linked_list(head->next) ∧ \valid(head)),L);

*/
/*@ requires
      \valid(head_ref) ∧ *head_ref ≢ \null ∧
      is_valid_linked_list(*head_ref);
    ensures
      \at(*\old(head_ref),Post) ≡ \old((*head_ref)->next) ∧
      ¬\valid(\old(*head_ref));
    assigns *head_ref;
 */
void popHead(Node **head_ref)
{
  /*@ assert rte: mem_access: \valid_read(head_ref); */
  /*@ assert rte: mem_access: \valid_read(&(*head_ref)->next); */
  Node *nextHead = (*head_ref)->next;
  /*@ assert \valid(*head_ref); */ ;
  /* preconditions of free:
     requires
       freeable: (void *)*head_ref ≡ \null ∨ \freeable((void *)*head_ref); */
  /*@ assert rte: mem_access: \valid_read(head_ref); */
  free((void *)*head_ref);
  /*@ assert rte: mem_access: \valid(head_ref); */
  *head_ref = nextHead;
  return;
}
```

Obrázek 5.4 Generovaný kód Frama-C pro funkci popHead

Na obrázku 5.4 je vidět jak předpoklady pro funkci jsou brané jako správné, ale nebyly dokazované, protože aktuálně ověřujeme osamocený kód. V těle funkce bylo dokázáno, že vstupní

parametr head_ref ukazuje na paměť, kterou lze číst a také do ní zapisovat. To nám vychází z předpokladů, ale jen co se dostaneme na ověření uvolnění paměti, tak nám Frama-C oznamuje, že se o ověření pokusilo, ale nedokáže vrátit úspěch ani neúspěch. A tedy ani další notace závisející na správnosti těchto výroků nemohou býti dokázány.

5.2.3 Funkce DFS

Výpis kódu 5.10 anotace DFS

```
/*@ requires 0 <= cur <= end && end >= 0 && end < 300000;
  @ requires 0 <= flow <= DBL_MAX;
  @ requires graph != \null && \valid(levels + (0..end))
  && \valid(visited + (0..end));
  @ requires \forall int i; 0 <= i <= end ==> \valid(graph[i] + (0..end));
  @ assigns visited[0..end], graph[0..end][0..end];
  @ ensures \result >= 0.0 && (\result > 0.0 ==>
    (\forall int i; 0 <= i <= end
        ==> graph[cur][i] <= \old(graph[cur][i])));
  @*/
double DFS (int cur, int end, double **graph, int *levels, int *visited,
           double flow){
...
 visited[cur] = 1;
    /*@ loop invariant 0 <= i <= end + 1;
      @ loop invariant 0 <= flow <= DBL_MAX;
      @ loop assigns i, graph[0..end][0..end], visited[0..end];
      @ loop variant end - i;
      @*/
    for (int i = 0; i <= end; ++i) {
...
}
```

Začínáme se dostávat ke složitějším funkcím, jednou z nich je DFS výpis kódu 5.4, která je volaná z naší hlavní funkce dinitzMaxFlow. Začneme od požadavků funkce, DFS od nás vyžaduje aby index aktuálního vrcholu byl menší, nebo roven indexu[2] stoku, který má být menší než 300 000,což byla námi stanovena horní hranice počtu vrcholů v grafu. Funkce také očekává nezáporný tok a validní matici sousednosti, která zobrazuje náš graf, validní pole držící úrovně vrcholů a také validní pole již navštívených vrcholů.

Dále říkáme, že funkce DFS pouze mění obsah pole s naším grafem a pole navštívených vrcholů. DFS v našem případě zaručuje, že výsledek bude nezáporný a pokud je větší jak nula, tak pro všechny hrany z aktuálního vrcholů platí, že jejich hodnota bude menší nebo rovná minulé. Což v kontextu našeho algoritmu znamená, že pokud nalezneme zlepšující tok zs, tak rezervy hran vrcholů, kterými tento tok vede, se může pouze zmenšit.

Nyní nám už v této funkci zbývají pouze anotace pro cyklus. Cykly bývají pro framework Frama-C velice těžké na ověření, proto přidáváme anotace, které s tím pomáhají.

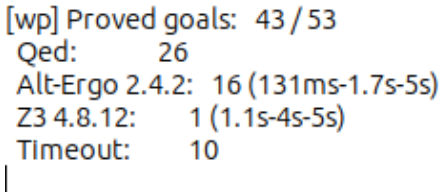
Prvně uvádíme proměnné, které se během cyklu mohou měnit nebo musí splňovat jisté podmínky, tyto podmínky budou kontrolovány vždy na začátku a na konci jednoho cyklu. Zde uvádíme proměnnou i, která je řídící proměnnou cyklu a proměnnou flow, od které požadujeme aby byla nezáporná.

Dále říkáme jaké proměnné cyklus může měnit a na závěr uvádíme číselnou hodnotu, která se s každým cyklem bude zmenšovat a je rovná, nebo větší maximálnímu počtu iterací daného cyklu. My jsme si zde vybrali počet vrcholů, protože je to přesný odhad, ale stejně tak bychom

[2]Protože pole se čísluje od 0, tak pro nás maximální index se bude rovnat 299 999.

mohli říci, že varianta cyklu bude 300 000 - i, protože víme, že naše implementace nebude mít více jak 300 000 vrcholů a tento cyklus prochází všechny hrany daného vrcholu.

Ověřme tedy naší implementaci funkce DFS obrázek 5.5 a zjistíme, že většina této funkce bude úspěšně ověřená, stalo se tak, protože ve funkci DFS pracujeme velice minimálně s proměnnými, které jsou dynamicky alokované.



Obrázek 5.5 Ověření DFS

5.2.4 Funkce BFS

Výpis kódu 5.11 anotace BFS

```
/*@ requires start >= 0 && end >= 0 && end < 300000;
  @ requires graph != \null && \valid(levels + (0..end));
  @ requires \forall int i; 0 <= i < end + 1  ==>
                              \valid(graph[i] + (0..end));
  @ assigns levels[0..end];
  @ ensures \result == 0 || \result == 1;
  @ ensures \forall int i; 0 <= i <= end ==> levels[i] >= -1
                                        && levels[i] <= end;
  @ ensures \forall int i; 0 <= i <= end && levels[i] == -1
  ==> (\forall int j;0 <= j <= end && levels[j] != -1
  ==> graph[j][i] == 0.0 && graph[i][j] == 0.0);
  @ ensures levels[end] > 0 ==>
  (\forall int i; \exists int j; \exists int k; 0 <= i < levels[end]
  && 0 <= j <= end && 0 <= k <= end && (levels[j] == i
  && levels[k] == i + 1) && graph[j][k] > 0);
  @*/
int BFS(int start, int end, double **graph, int *levels){
...
  /*@ loop invariant 0 <= i <= end + 1;
      @ loop invariant \forall int j; 0 <= j < i ==> levels[j] == -1;
      @ loop assigns i, levels[0..end];
      @ loop variant end - i;
      @*/
    for (int i = 0; i <= end; ++i) {
        levels[i] = -1;
    }
...
  /*@ loop invariant \valid(queueHead) || queueHead == \null;
      @ loop invariant \forall int i; 0 <= i <= end ==> levels[i] >= -1
                                                    && levels[i] <= end;
      @ loop assigns cur, queueHead, levels[0..end];
      @*/
    while(queueHead != NULL){
        cur = queueHead->data;
        popHead(&queueHead);
```

```
        /*@ loop invariant \forall int i; 0 <= i <= end ==> levels[i] >= -1
                                                &&& levels[i] <= end;
          @ loop assigns queueHead, levels[0..end];
          @*/
        for (int i = 0; i <= end; ++i) {
            if(graph[cur][i] > 0){
                if(levels[i] < 0){
                    levels[i] = levels[cur] + 1;
                    insertHead(&queueHead, i);
                }
            }
        }
    }
...
}
```

Funkci BFS už určitě musíme rozebrat po částech, začněme od vrchu, funkce BFS má menší požadavky než funkce DFS, požaduje pouze, aby startovní vrchol, neboli zdroj, měl menší index než index stoku, a aby stok byl menší než 300 000. Vyžadujeme validní graf a pole kam můžeme ukládat úrovně vrcholů.

Naše BFS pouze upravuje hodnoty pro úrovně vrcholů a poté hlavní funkcionalitou je, co vše nám BFS zaručuje. Začneme jednoduše, BFS nám vrací 0 pro nenalezení cesty zs, nebo 1 pokud jsme cestu našli. Druhá vlastnost, kterou nám BFS dává je, že každý vrchol bude mít nastavenou úroveň od -1 až po počet vrcholů - 1.

Třetí vlastnost nám říká, že pro každý vrchol, pokud jeho úroveň je -1 po skončení funkce, tak neexistuje hrana mezi vrcholem, který má úroveň vyšší než -1. Neboli BFS nám v teorii grafů zaručuje, že pokud nějaký vrchol nebyl nalezen, tak neexistuje cesta mezi zdrojem a daným vrcholem.

A poslední vlastnost nám říká, že pokud stok má úroveň vyšší než -1, tak pro každou úroveň existuje vrchol který má hranu, která vede z tohoto vrcholu do vrcholu, který má úroveň o jedna vyšší. Jinými slovy tato vlastnost nám zaručuje, že pokud stok má úroveň vyšší než -1 tak existuje cesta o délce úrovně stoku ze zdroje do stoku.

Tedy na rozdíl od předchozího tvrzení, říkáme nejenom, že cesta existuje, ale také tvrdíme, že je dané délky a v tomto případě to také bude délka nejkratší cesty.

Protože ve funkci BFS podle výpisu kódu 5.3 využíváme také funkce popHead z výpisu kódu 5.6 a insertHead z výpisu kódu 5.7, tak už nemůžeme prověřit BFS jako osamocený kód, ale musíme ho prověřit společně s těmito funkcemi. Většinu kódu bude možné verifikovat jak vidíme na obrázku 5.6, ale jejich neověřené části se budou propagovat do pozdějších části BFS a znemožňovat další verifikaci.

```
[wp] Proved goals:  60 / 78
 Qed:          48
 Alt-Ergo 2.4.2:  12 (12ms-30ms-54ms)
 Timeout:      18
```

Obrázek 5.6 Ověření BFS

Jak jsme již viděli na obrázku 5.4, tak i zde Frama-C byla schopná ověřit vše, až do bodu, kdy se střetne s alokací dynamické paměti obrázek 5.7 a poté nemá možnost ověřit naše tvrzení a přiřazování, protože si není jistá, zda paměť lze číst a modifikovat.

```
/*@ requires start ≥ 0 ∧ end ≥ 0 ∧ end < 300000;
    requires graph ≠ \null ∧ \valid(levels + (0 .. end));
    requires
      ∀ int i; 0 ≤ i < end + 1 ⇒ \valid(*(graph + i) + (0 .. end));
    ensures \result ≡ 0 ∨ \result ≡ 1;
    ensures
      ∀ int i;
        0 ≤ i ≤ \old(end) ⇒
        *(\old(levels) + i) ≥ -1 ∧ *(\old(levels) + i) ≤ \old(end);
    ensures
      ∀ int i;
        0 ≤ i ≤ \old(end) ∧ *(\old(levels) + i) ≡ -1 ⇒
        (∀ int j;
           0 ≤ j ≤ \old(end) ∧ *(\old(levels) + j) ≠ -1 ⇒
           *(*(\old(graph) + j) + i) ≡ 0.0 ∧
           *(*(\old(graph) + i) + j) ≡ 0.0);
    ensures
      *(\old(levels) + \old(end)) > 0 ⇒
      (∀ int i; ∃ int j;
        ∃ int k;
          0 ≤ i < *(\old(levels) + \old(end)) ∧ 0 ≤ j ≤ \old(end) ∧
          0 ≤ k ≤ \old(end) ∧ *(\old(levels) + j) ≡ i ∧
          *(\old(levels) + k) ≡ i + 1 ∧ *(*(\old(graph) + j) + k) > 0);
    assigns *(levels + (0 .. end));
  */
int BFS(int start, int end, double **graph, int *levels)
{
  int __retres;
  int cur;
  {
    int i = 0;
    /*@ loop invariant 0 ≤ i ≤ end + 1;
        loop invariant ∀ int j; 0 ≤ j < i ⇒ *(levels + j) ≡ -1;
        loop assigns i, *(levels + (0 .. end));
        loop variant end - i;
    */
    while (i <= end) {
      {
        /*@ assert rte: mem_access: \valid(levels + i); */
        *(levels + i) = -1;
      }
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
    }
  }
  /*@ assert rte: mem_access: \valid(levels + start); */
  *(levels + start) = 0;
  Node *queueHead = (Node *)0;
  insertHead(& queueHead,start);
  /*@ loop invariant \valid(queueHead) ∨ queueHead ≡ \null;
      loop invariant
        ∀ int i;
          0 ≤ i ≤ end ⇒ *(levels + i) ≥ -1 ∧ *(levels + i) ≤ end;
      loop assigns cur, queueHead, *(levels + (0 .. end));
  */
  while (queueHead != (Node *)0) {
    /*@ assert rte: mem_access: \valid_read(&queueHead->data); */
    cur = queueHead->data;
    popHead(& queueHead);
    {
      int i_0 = 0;
      /*@ loop invariant
            ∀ int i;
              0 ≤ i ≤ end ⇒
              *(levels + i) ≥ -1 ∧ *(levels + i) ≤ end;
          loop assigns queueHead, *(levels + (0 .. end));
      */
      while (i_0 <= end) {
        {
          /*@ assert rte: mem_access: \valid_read(graph + cur); */
          /*@ assert rte: mem_access: \valid_read(*(graph + cur) + i_0); */
```

Obrázek 5.7 Generovaný kód Frama-C pro funkci BFS

5.2.5 Funkce dinitzMaxFlow

Výpis kódu 5.12 anotace dinitzMaxFlow

```
/*@ axiomatic SumCapacity {
  @   logic double sum_capacity(double **graph, integer nodes,
  integer index) reads graph[0][0..nodes-1];
  @   axiom sum_capacity_base:
  @     \forall double **graph, integer nodes;
  @       0 <= nodes ==> sum_capacity(graph, nodes, 0) == 0;
  @   axiom sum_capacity_step:
  @     \forall double **graph, integer nodes, integer index;
  @       0 <= index < nodes - 1 ==>
  @         sum_capacity(graph, nodes, index + 1) == sum_capacity
            (graph, nodes, index) + graph[0][index];
  @ }
  @*/

/*@ predicate valid_flow(double result, double **graph, integer nodes) =
      result >= -1 && result <= sum_capacity(graph, nodes, nodes - 1);
  @*/

/*@ requires nodes > 0 && nodes <= 300000;
  @ requires graph != \null && \forall integer i; 0 <= i < nodes
  ==> \valid(graph[i] + (0..nodes-1));
  @ ensures valid_flow(\result, graph, nodes);
  @ assigns graph[0..nodes-1][0..nodes-1];
  @*/

double dinitzMaxFlow(int nodes, double **graph)
    ...
    // Run BFS to check if flow is possible and set the levels of nodes
    /*@ loop invariant 0 <= total && total <= sum_capacity
                                (graph, nodes, nodes - 1);
      @ loop assigns total, vertexLevel[0..nodes-1],
      *visited, graph[0..nodes-1][0..nodes-1];
      @ loop variant nodes - 1 - vertexLevel[nodes-1];
      @*/
    while (BFS(0, nodes - 1, graph, vertexLevel))
    {
        // Initialize visited array
        visited = (int*) calloc(nodes, sizeof(int));
        double flow;
        /*@ loop invariant 0 <= total;
          @ loop assigns flow, total, visited[0..nodes-1],
          graph[0..nodes-1][0..nodes-1];
          @ loop variant (int)(DBL_MAX - total) * 1000;
          @*/
        while (1)
        {
            flow = DFS(0, nodes - 1, graph, vertexLevel, visited, DBL_MAX);
            ...
        }
        free(visited);
    }
    ...
}
```

Poslední funkcí, kterou musíme popsat je naše hlavní funkce dinitzMaxFlow, začneme nejdřív axiomem, který později využíváme dál. Axiomatic SumCapacity nám definuje nový axiomatický blok s názvem SumCapacity, poté definujeme logickou funkci sum_capacity, která bere tři argumenty: dvojrozměrné pole, které udržuje náš graf, číslo nodes, které nám říká počet vrcholů a jako poslední index. Tato logická funkce čte hodnoty z našeho grafu od indexu 0 až po nodes - 1.

Nyní již definujeme axiom sum_capacity_base, který nám dává základní krok pro funkci sum_capacity, samotný axiom říká, že pokud nodes je větší nebo rovno nule, tak sum_capacity pro index 0 je rovno 0. Druhý axiom sum_capacity_step, nám definuje rekurzivní krok pro funkci sum_capacity, kde tvrdíme, že pro každý index, který je mezi 0 a nodes - 1, tak sum_capacity pro index + 1 je rovná sum_capacity pro index + hodnota hrany v grafu z vrcholu o indexu *0* do vrcholu o indexu *index*.

Pokud tento axiom spojíme s naší reprezentací grafu pomocí matice sousednosti a také toho, že pro nás vrchol o indexu 0 je vždy zdroj, tak tento axiom nám dá součet kapacit všech hran vycházejících ze zdroje. Axiomy můžeme také ověřit jak je vidět na obrázku 5.8.

Dále tento součet používáme v námi definovaném predikátu valid_flow, kde říkáme, že výsledek se musí nacházet mezi -1 a součtem kapacit hran včetně z obou stran. Tento predikát poté používáme jako vlastnost kterou dinicMaxFlow funkce splňuje.

Zbylé anotace už jsou oproti minulé triviální, pouze požadujeme, aby počet vrcholů byl větší než 0 a menší než 300 000 a také požadujeme aby graf byl validní. Nakonec říkáme, že tato funkce může změnit obsah matice udržované v proměnné graph.

Teď nám už zbývají pouze anotace u cyklů, začneme anotacemi vnitřního cyklu. Nejdříve uvádíme podmínku na proměnnou total, jedná se o náš výsledek, který by měl být vždycky nezáporný. Poté říkáme, že tento cyklus může měnit hodnoty proměnných flow a total, prvky pole visited a prvky matice graph.

Jako poslední chceme určit hodnotu varianty grafu, tedy jak omezit shora iterace cyklu, v tomto případě jsme se rozhodli o odečítání aktuálního celkového toku od maximální hodnoty typu double zvětšená o 1000. Dovolili jsme totiž hrany, které nemusí být celá čísla a proto chceme zachytávat i tyto malé změny. Zároveň si myslíme, že tato hodnota bude s každým cyklem klesat, protože DFS najde pokaždé hodnotu o kterou zvětší náš tok, nebo vrátí nulu, která cyklus ukončí.

Od vnějšího cyklu vyžadujeme, aby proměnná flow byla před a po každé iteraci větší, nebo rovno nule a menší, nebo rovno sumě kapacit hran vycházejících ze zdroje. Tento cyklus také může měnit proměnnou total a prvky polí vertexLevel, graph a visited[3].

Teď už nám zbývá pouze variant vnějšího cyklu, který tentokrát bude celkový počet vrcholů - 1 - úroveň stoku. Pokud se nad tímto zamyslíme, tak v nejhorším případě naše nejkratší cesty se budou s každou iterací prodlužovat o jeden vrchol a to až do doby kdy všechny vrcholy budou potřeba k nalezení nejkratší cesty zs. V této iteraci naše DFS nasytí i tyto cesty a nezbudou již žádné cesty zs a algoritmus skončí. Tedy variant se s každou iterací zmenší alespoň o jedna a bude vždy vetší nebo roven 0.

Zbývá nám poslední verifikace a to celého kódu obrázek 5.9, protože dinitzMaxFlow využívá všech ostatních funkcí.

[3]Pole visited zde ještě není alokované, takže nemůžeme určit rozměry stejně jako u ostatních polí.

```
/*@
axiomatic SumCapacity {
  logic double sum_capacity{L}(double **graph, ℤ nodes, ℤ index)
    reads \at(*(*(graph + 0) + (0 .. nodes - 1)),L);

  axiom sum_capacity_base{L}:
    ∀ double **graph, ℤ nodes;
      0 ≤ nodes ⇒ sum_capacity(graph, nodes, 0) ≡ 0;

  axiom sum_capacity_step{L}:
    ∀ double **graph, ℤ nodes, ℤ index;
      0 ≤ index < nodes - 1 ⇒
      sum_capacity(graph, nodes, index + 1) ≡
      sum_capacity(graph, nodes, index) + *(*(graph + 0) + index);

  }

*/
```

Obrázek 5.8 Ověření Axiomů

```
[wp] Proved goals:  132 / 190
 Qed:          102
 Alt-Ergo 2.4.2:  29 (12ms-1.6s-5s)
 Z3 4.8.12:        1 (1.5s-4.9s-5s)
 Timeout:         58
```

Obrázek 5.9 Ověření dinitzMaxFlow

Závěr

V této práci jsme se věnovali problematice formálního ověření a implementace Dinitzova algoritmu s dynamicky alokovanou pamětí, pro hledání maximálního toku v orientované grafové síti. Podařilo se nám plně implementovat Dinitzův algoritmus v programovacím jazyce C s ACSL anotacemi a také ověřit část kódu, která nepotřebuje prozatím neimplementované výrazy frameworku Frama-C týkajících se dynamické alokace paměti.

V prvních dvou kapitolách jsme také čtenáře seznámili s oblastí toků v orientovaných grafech, představili jsme jim mnoho základních pojmů a definic a také plně popsali myšlenku Dinitzova algoritmus pro hledání maximálního toku. Poskytli jsme jim také další zdroje a práce, pokud by se chtěli v tomto tématu dále vzdělávat.

Později jsme také uvedli některé možnosti, z mnoha dalších, frameworku Frama-C a doprovázejícímu mu anotačnímu jazyku ACSL, který jsme zde představili. Zavedli jsme si nástroje, které jsme používali v naší práci, mimo frameworku Frama-C, také pluginy WP, RTE a také řešiče Alt-Ergo, CVC4 a Z3. Ke každému z nich jsme dodali krátký popisek a také projekty v kterých byly využity.

V praktické části jsme se již plně zaměřili na popsání naší implementace Dinitzova algoritmu a objasnění našich rozhodnutí ohledně výběrů prostředků pro funkčnost našeho algoritmu. Poté jsme pokračovali vysvětlením a zařazením anotací jazyka ACSL do tématu a problématiky nalezení maximálního toku v sítích. Řekli jsme si zde také problémy svázané s použitím dynamicky alokované paměti a pozdější verifikací pomocí frameworku Frama-C.

Rozšířením této práce by mohlo být porovnání rychlostí ověřeného Dinitzova algoritmu pro hledání maximálního toku a neověřeného Dinitzova algoritmu s využitím programovacích taktik, sloužícím ke zrychlení algoritmu. Takové taktiky často nemůžeme při verifikaci používat, protože je velice obtížné, nebo i nemožné je poté ověřit.

Bibliografie

1. COUSOT, Patrick; COUSOT, Radhia. Modular Static Program Analysis. In: HORSPOOL, R. Nigel (ed.). *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, [online]. 2002 [cit. 2023-04-30], s. 159–179. ISBN 978-3-540-45937-8. Dostupné z DOI: 10.1007/3-540-45937-5_13.
2. SUCHÝ, Ondřej; VALLA, Tomáš. *Sítě, toky v sítích, Fordův-Fulkersonův algoritmus*. [online prezentace]. 2020 [cit. 2023-04-28]. Dostupné také z: https://courses.fit.cvut.cz/BI-AG2/media/lectures/bi-ag2-p03.pdf.
3. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. 2. vydání. Praha: CZ.NIC, z.s.p.o., 2022. ISBN 978-80-88168-66-9.
4. NETO, Euclides; CALLOU, Gustavo. *An Approach Based on Ford-Fulkerson Algorithm to Optimize Network Bandwidth Usage*. [online]. 2015 [cit. 2023-05-02]. Dostupné z DOI: 10.1109/SBESC.2015.21.
5. MALLICK, Kalyan; KHAN, Aminur; AHMED, M.; AREFIN, Md; UDDIN, Md. Modified EDMONDS-KARP Algorithm to Solve Maximum Flow Problems. *Open Journal of Applied Sciences*. [online]. 2016 [cit. 2023-05-02], roč. 6, s. 131–140.
6. GOLDBERG, Andrew V.; TARJAN, Robert E. A New Approach to the Maximum-Flow Problem. *J. ACM*. [online]. 1988 [cit. 2023-05-02], roč. 35, č. 4, s. 921–940. ISSN 0004-5411. Dostupné z DOI: 10.1145/48014.61051.
7. MALHOTRA, Vishv; KUMAR, M.Pramodh; MAHESHWARI, S.N. An O(V3) Algorithm for Finding Maximum Flows in Networks. *Information Processing Letters*. [online]. 1978 [cit. 2023-05-02], roč. 7, s. 277–278. Dostupné z DOI: 10.1016/0020-0190(78)90016-9.
8. KING, Valerie; RAO, Satish; TARJAN, Robert. A Faster Deterministic Maximum Flow Algorithm. In: [online]. 1992 [cit. 2023-05-02], s. 157–164.
9. KATHURIA, Tarun. *A Potential Reduction Inspired Algorithm for Exact Max Flow in Almost $\widetilde{O}(m^{4/3})$ Time*. [online]. 2020 [cit. 2023-05-02]. Dostupné z arXiv: 2009.03260 [cs.DS].
10. HOLDSWORTH, Jason. The Nature of Breadth-First Search. [online]. 1999 [cit. 2023-05-03]. Dostupné také z: https://www.researchgate.net/publication/273264449_Understanding_Dijkstra_Algorithm.
11. JAVAID, Adeel. Understanding Dijkstra Algorithm. *SSRN Electronic Journal*. [online]. 2013 [cit. 2023-05-03]. Dostupné z DOI: 10.2139/ssrn.2340905.
12. PUTRI, Sheila Eka; TULUS, Tulus; NAPITUPULU, Normalina. Implementation and Analysis of Depth-First Search (DFS) Algorithm for Finding The Longest Path. In: [online]. 2011 [cit. 2023-05-03]. Dostupné z DOI: 10.13140/2.1.2878.2721.

13. BHADRA, Somasree; KUNDU, Anirban; KHATUA, Sunirmal. Optimization of Road Traffic Congestion in Urban Traffic Network Using Dinic's Algorithm. In: [online]. 2021 [cit. 2023-05-03], s. 372–379. ISBN 978-3-030-73602-6. Dostupné z DOI: 10.1007/978-3-030-73603-3_34.
14. HAQUE, Md; ISLA, Md. Traffic model of LTE using maximum flow algorithm with binary search technique. In: [online]. 2020 [cit. 2023-05-03].
15. GHAFFARI, Mohsen; KARRENBAUER, Andreas; KUHN, Fabian; LENZEN, Christoph; PATT-SHAMIR, Boaz. *Near-Optimal Distributed Maximum Flow*. [online]. 2015 [cit. 2023-05-03]. Dostupné z arXiv: 1508.04747 [cs.DS].
16. SHILOACH, Yossi; VISHKIN, Uzi. An O(n2 log(n)) Parallel Max-Flow Algorithm. *J. Algorithms*. [online]. 1982 [cit. 2023-05-03], roč. 3, s. 128–146. Dostupné z DOI: 10.1016/0196-6774(82)90013-X.
17. *WP Plug-in Manual: Frama-C 26.1* [online]. [cit. 2023-05-05]. Dostupné z: https://frama-c.com/download/frama-c-wp-manual.pdf.
18. *Frama-C's RTE plug-in: for Frama-C 26.1* [online]. [cit. 2023-05-05]. Dostupné z: https://frama-c.com/download/frama-c-rte-manual.pdf.
19. *E-RTE plug-in Manual: for Frama-C 26.1* [online]. [cit. 2023-05-05]. Dostupné z: https://frama-c.com/download/e-acsl/e-acsl-manual.pdf.
20. *Alt-Ergo Documentation* [online]. [cit. 2023-05-05]. Dostupné z: https://ocamlpro.github.io/alt-ergo/.
21. *Why3 Documentation* [online]. [cit. 2023-05-05]. Dostupné z: https://why3.lri.fr/#documentation.
22. *Spark About Page* [online]. [cit. 2023-05-05]. Dostupné z: https://www.adacore.com/about-spark.
23. *Atelier B Main Page* [online]. [cit. 2023-05-05]. Dostupné z: https://www.atelierb.eu/en/.
24. *CVC4 Documentation* [online]. [cit. 2023-05-05]. Dostupné z: https://cvc4.github.io/documentation.html/.
25. *Dafny Reference Manual* [online]. [cit. 2023-05-05]. Dostupné z: https://dafny.org/latest/DafnyRef/DafnyRef.
26. *SymbiYosis Documentation* [online]. [cit. 2023-05-05]. Dostupné z: https://symbiyosys.readthedocs.io/en/latest/index.html.
27. *Z3 Theorem Prover* [online]. [cit. 2023-05-05]. Dostupné z: https://github.com/z3prover/.
28. *Viper Research* [online]. [cit. 2023-05-05]. Dostupné z: https://www.pm.inf.ethz.ch/research/viper.html.
29. ACETO, Luca; GORIAC, Eugen-Ioan; INGOLFSDOTTIR, Anna. Meta SOS - A Maude Based SOS Meta-Theory Framework. *Electronic Proceedings in Theoretical Computer Science*. 2013, roč. 120, s. 93–107. Dostupné z DOI: 10.4204/eptcs.120.8.
30. *ACSL: ANSI/ISO C Specification Language: Version 1.18* [online]. [cit. 2023-05-05]. Dostupné z: https://frama-c.com/download/frama-c-acsl-implementation.pdf.

Obsah přiloženého média