FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Ověřená implementace struktury Union-Find
Student:	Jakub Bartoň
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

Úkoly práce jsou následující:
1) Seznamte se s datovou strukturou Union-Find a možnostmi její implementace.
2) Zhodnoťte a porovnejte vybrané implementace, včetně jejich kladů a záporů.
3) Jednu konkrétní metodu naimplementujte v programovacím jazyku C.
4) Prostudujte framework Frama-C.
5) V rámci možností poskytovaných aktuální verzí Frama-C proveďte verifikaci Vaší implementace.
6) Proveďte analýzu, do jaké míry lze v tomto frameworku, jehož současná implementace nedostatečně podporuje ověřování práce s dynamickou pamětí, ověřit správu dynamické paměti při běhu algoritmů nad touto strukturou.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 30. ledna 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Ověřená implementace struktury Union-Find

Jakub Bartoň

Katedra počítačových systémů
Vedúci práce: doc. RNDr. Dušan Knop, Ph.D.

6. mája 2023

Poďakovanie

Touto cestou by som rád poďakoval doc. RNDr. Dušanovi Knopovi, Ph.D. za jeho podporu, pomoc a užitočné rady behom celého procesu tvorby tejto práce. Ďalej by som rád poďakoval svojim rodičom a priateľom za podporu, ktorú mi poskytli behom tvorby tejto práce a behom ťažkých chvíľ v priebehu celého štúdia.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

Odkaz na túto prácu

Bartoň, Jakub. *Ověřená implementace struktury Union-Find.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Union-Find je dátová štruktúra používaná v úlohách, ktoré vyžadujú množinovú operáciu zjednotenia a identifikáciu, do akej množiny prvok patrí. Programátori často tieto štruktúry neimplementujú sami, ale vyhľadávajú ich implementácie online. V takom prípade je problémom overenie, že implementácia funguje správne a že vykonáva iba to čo má. Táto práca sa preto zameriava na analýzu dátovej štruktúry Union-Find, možnosťami jej implementácie, verifikáciou a porovnaním výkonnosti jednotlivých implementácií.

Klíčová slova dátová štruktura, verifikácia, Union-Find, Frama-C, symbolic engine, benchmark

Abstract

Union-Find is a data structure that can be used for tasks that require set union and checking to which set an element belongs. Software developers do not implement those data structures on their own. They prefer to look for them online. The problem is, that it is difficult to verify the correctness of the implementation. This thesis focuses on analyzing the Union-Find data structure, ways of optimized implementation, verification, and performance comparison of selected implementations.

Keywords data structure, verification, union-find, Frama-C, symbolic engine, benchmark

Obsah

Zoznam obrázkov

Úvod

V dnešnej dobe veľká časť programátorov v rámci svojej práce implementuje riešenia na konkrétne problémy, ale nie dátové štruktúry pre to potrebné. V moment keď narazia na potrebu použitia dátovej štruktúry využívajú v minulosti použité implementácie alebo si implementáciu nájdu online (napríklad na stránkach ako je github[1] alebo stackoverflow[2]).

Sú však tieto implementácie správne? Môžeme sa naozaj spoľahnúť na to, že neobsahujú žiadne chyby, ktoré by mohli viesť k nesprávnym výsledkom alebo v horšom prípade eskalovať k väčším problémom? Na tieto stránky predsa môže prispievať ktokoľvek. Kód, ktorý možno na týchto stránkach nájsť nie je žiadnym spôsobom regulovaný, jediný spôsob pre zistenie či sa jedná o dobrý zdroj je popularita autora (počet ľudí sledujúcich užívateľa na githube alebo reputácia v prípade stackoverflow). Aby sme však zaistili, že kód spĺňa naše požiadavky je potreba ho poriadne otestovať.

Testovanie možno vykonať napríklad pomocou jednotkových testov alebo testom komponenty. Prípadne je možné siahnuť po nástrojoch slúžiacich pre analýzu kódu ako je napríklad *symbolic execution engine* (o ňom sa dozvieme niektoré základné informácie v podkapitole 4.1). *Symbolic execution engine* netestuje správnosť výsledkov jednotlivých operácií, ale objavuje skôr chyby poškodzujúce pamäť – chyby vznikajúce nesprávnou prácou s pamäťou (napríklad dereferencia pamäti, ku ktorej by sme nemali mať prístup, či už sa jedná o pamäť zásobníku, alebo haldy).

Existuje však nejaké vhodnejšie riešenie? Našťastie odpoveď znie áno. Tomuto riešeniu sa hovorí verifikácia kódu. Jedná sa o proces, ktorý nám umožňuje formálne dokázovať, že náš kód funguje správne a vykonáva práve to, čo od neho očakávame a to za pomoci software a dôkladne špecifikovaných požiadavkov ako by mali vyzerať parametry funkcie, ako by mal vyzerať výstup funkcie a tak ďalej. Verifikáciu možno vykonať pomocou rôznych nástrojov alebo

[1] https://github.com/
[2] https://stackoverflow.com/

frameworkov, v prípade programovacieho jazyka C je to napríklad framework *Frama-C*, o ktorom budeme pojednávať v rámci kapitoly 2.

Cieľom tejto práce je oboznámiť sa s dátovou štruktúrou Union-Find a zistiť aké sú možnosti jej implementácie pre dosiahnutie zrýchlenia, všetky tieto implementácie následne porovnať, zoznámiť sa s frameworkom *Frama-C*, ktorý slúži pre verifikáciu programov v programovacom jazyku C. Po tejto teoretickej príprave implementovať jednu z uvedených implementácií dátovej štruktúry Union-Find a následne využiť znalosti o verifikačnom frameworku pre verifikáciu našich implementácií.

Kapitola 1

Union-Find

V tejto kapitole si popíšeme aké operácie dátová štruktúra Union-Find podporuje, aké sú možnosti jej využitia, vysvetlíme si ako sme v rámci tejto práce reprezentovali štruktúru, ako operácie tejto štruktúry fungujú, aké sú možnosti ich optimalizácie a aké sú ich výhody a nevýhody.

Pre vysvetlenie dátovej štruktúry Union-Find a jej internej reprezentácie sa potrebujeme oboznámiť s nasledujúcim názvoslovím:

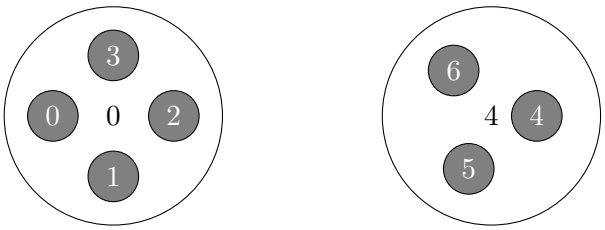
- *nezávislé (disjunktné) množiny* – dve množiny nazveme nezávislými (disjunktnými) práve vtedy, keď ich prienik je prázdna množina
- *(orientovaný) graf* G – je usporiadaná dvojica $G = (V, E)$, kde:
  - $V = V(G)$ je konečná neprázdna množina vrcholov grafu G
  - $E = E(G)$ je množina hrán a platí, že: $\forall e \in E(G), \exists u, v \in V(G) : e = \{u, v\}$ (respektíve $e = (u, v)$) [1]
- *veľkosť grafu* G – je počet vrcholov grafu G (veľkosť množiny $V(G)$)
- *podgraf (orientovaného) grafu* G – je taký (orientovaný) graf H, že:
  - $V(H) \subseteq V(G)$
  - $E(H) \subseteq E(G)$ a zároveň $\forall e \in E(H), \exists u, v \in V(H) : e = \{u, v\}$ (respektíve $e = (u, v)$) [1]
- *indukovaný (orientovaný) podgraf* – je ľubovoľný (orientovaný) graf H, ktorý je podgrafom grafu G a platí preň $E(H) = E(G) \cap \binom{V(H)}{2}$ [2]
- *(orientovaný) sled* – je striedavá postupnosť vrcholov a hrán grafu G $(v_0, e_1, v_1, ..., e_n, v_n)$, kde pre každú hranu e_i platí, že $e_i = \{v_{i-1}, v_i\}$ (respektíve $e_i = (v_{i-1}, v_i)$) [1]
- *(orientovaná) cesta* – je (orientovaný) sled, v ktorom sa neopakujú vrcholy [1]

- *dĺžka cesty* – je počet hrán, ktorými je cesta tvorená
- *kružnica (cyklus)* – je graf, pre ktorý platí:
  - $V = \{0,\ ...,\ n-1\}$
  - $E = \{(i,\ (i+1)\ mod\ (n)) \mid 0 \leq i < n\}$ [2]
- *slučka* – je kružnica o jednom vrchole
- *súvislý graf* – je graf, v ktorom medzi každou dvojicou vrcholov existuje cesta [1]
- *komponenta súvislosti* – je súvislý indukovaný podgraf, ktorý je v inklúzií maximálny[3] [2]
- *strom* – je súvislý graf, ktorý neobsahuje kružnicu ako podgraf (s možnou výnimkou slučky) [1]
- *les* – je graf, ktorý neobsahuje kružnicu ako podgraf [2]
- *zakorenený strom* – je strom, ktorého jeden vrchol je označený ako kořeň a ten obsahuje slučku [2]
- *rodič vrcholu v* – je taký vrchol, do ktorého vedie hrana z vrcholu v
- *hĺbka stromu* – je počet vrcholov najdlhšej cesty z koreňa stromu
- *rád stromu* – je dĺžka najdlhšej cesty z koreňa stromu
- *kostra* – je podgraf, ktorý obsahuje všetky vrcholy pôvodného grafu a zároveň je to strom bez slučky [2]
- *váha* – je funkcia $w : E(G) \to R$, ktorá každej hrane grafu G priradí číslené ohodnotenie [2]
- *najľahšia hrana grafu G* – je taká hrana grafu G, ktorej váha je najmenšia zo jeho všetkých hrán grafu G
- *váha grafu G* – je súčet váh hrán grafu G [2]
- *minimálna kostra* – je kostra, ktorej váha je najnižšia možná [2]

[3] pridaním ľubovoľného ďalšieho vrcholu by podgraf už nebol súvislý

1.1 Čo to je Union-Find?

Union-Find je dátová štruktúra zložená z kolekcie nezávislých (disjunktných) množin, kde každá množina má priradený jednoznačný identifikátor. Často je možné sa s touto dátovou štruktúrou stretnúť aj pod názvom *Merge-Find* alebo *Disjoint Set*.



Ukážka 1.1: Vennove diagramy reprezentujúce nezávislé (disjunktné) množiny s identifikátormi 0 a 4

Union-Find je dátová štruktúra podporujúca nasledujúce tri operácie:

- pridanie prvku (`makeSet(x)`) – tento prvok `x` reprezentuje novú množinu obsahujúcu iba tento prvok
  - štruktúra **nedovoluje** vkladanie prvkov, ktoré sa už v niektorej z množín nachádzajú – to nám napovedá aj jedno z pomenovaní tejto dátovej štruktúry a to *Disjoint Set*, čo v preklade znamená *nezávislá (disjunktná) množina*
- zjednotenie množín (`union(x,y)`) – umožňuje tvorbu množiny obsahujucej všetky prvky množín, do ktorých patria prvky `x` a `y` – pôvodné dve množiny zanikajú a stáva sa z nich jedna nová množina
- identifikácia množiny, do ktorej prvok `x` patrí (`find(x)`) – táto operácia umožňuje získať identifikátor množiny, do ktorej patrí prvok `x`

1.2 Aké je využitie?

Operácie podporované dátovou štruktúrou Union-Find sú často požadované v rámci vybraných problémov. Príklady problémov, v ktorých možno využiť tieto operácie sú:

- hľadanie minimálnej kostry (*Kruskalov algoritmus* [2]) – to funguje následovne:
  1. začíname lesom, ktorý neobsahuje žiadne hrany (dátová štruktúra Union-Find obsahuje množiny, ktoré pozostáva z vrcholov tohto lesa)

2. z grafu odoberieme najľahšiu hranu a pre vrcholy, ktoré ju tvoria, zistíme, do ktorej množiny (respektíve stromu) patria (ak sú tieto množiny rôzne vykonáme ich zjednotenie – `union`) – tento krok opakujeme dokým nespracujeme všetky hrany [3]

- hľadanie komponent súvislosti – tento problém sa dá riešiť modifikáciou algoritmou *depth first search* alebo *breadth first search* [4], ďalším možným riešením tohto problému je použitie dátovej štruktúry Union-Find, tá funguje nasledovne:

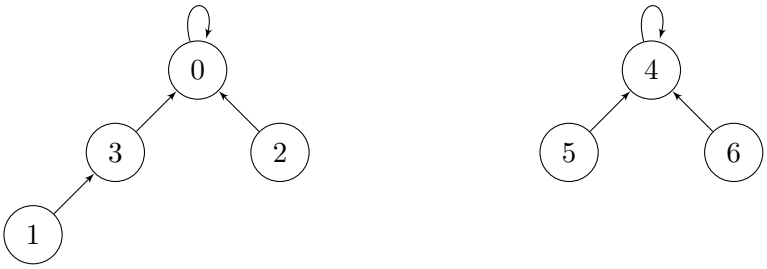
  1. vytvoríme štruktúru Union-Find obsahujúcu všetky vrcholy grafu
  2. pre každú dvojicu vrcholov tvoriacich hranu vykonáme operáciu `union` – po spracovaní platí, že všetky vrcholy jednej komponenty súvislosti sa nachádzajú v práve jednej množine dátovej štruktúry [5]

- detekcia kružnice v grafe – cyklus v grafe možno detekovať modifikovanou verziou algoritmu *depth first search* [6], alternatívnou možnosťou je použitie dátovej štruktúry Union-Find, tá funguje nasledovne:

  1. vytvoríme štruktúru Union-Find obsahujúcu všetky vrcholy grafu
  2. spracujeme postupne každú hranu grafu – zistíme, do ktorej množiny patria vrcholy tvoriace spracovávanú hranu (`find`), tu môžu nastať dve situácie:
     - ak vrcholy patria do rôznych množin – vykonáme zjednotenie týchto množín (`union`)
     - ak vrcholy patria do rovnakých množin, znamená to, že sme detekovali cyklus – spracované hrany obsahujú cestu medzi týmito vrcholmi a tá spolu s práve spracovávanou hranou tvorí cyklus [7]

1.3 Aké sú možnosti implementácie?

Táto dátová štruktúra reprezentuje nezávislé (disjunktné) množiny ako kolekciu orientovaných zakorenených stromov, kde každý strom predstavuje jednu množinu [2, 8]. Stromy reprezentujúce množinu obsahujú orientované hrany tvoriace cestu do koreňa stromu, ktorého hodnota tvorí jednoznačný identifikátor množiny (hodnotu v koreni možno označiť za identifikátor množiny vďaka tomu, že Union-Find neumožňuje vkladanie duplicitných prvkov). Vďaka tejto vlastnosti nám pre získanie identifikátoru množiny stačí nájsť koreň stromu množiny obsahujucej požadovaný prvok.



Ukážka 1.2: Kolekcia zakorenených stromov reprezentujúcich množiny vennovho diagramu z ukážky 1.1

Orientované hrany zároveň predstavujú vzťah zjednotenia dvoch množín a to, ako tieto hrany budú konštruované záleží na implementácií operácie zjednotenia (`union`) a v niektorých prípadoch aj na implementácií operácie nájdenia identifikátoru množiny (`find`), ktorá v niektorých optimalizovaných verziách manipuluje s hranami stromu množiny. Porovnaním jednotlivých implementácií sa v minulosti zaoberali Tarjan a Leeuwen [9], ktorý v roku 1984 publikovali článok, v ktorom prišli s novým spôsobom implementácie operácie `find` pomocou pólenia cesty a pomocou delenia cesty, o ktorých si povieme bližšie v podkapitolách 1.3.5 a 1.3.6. Pre implementácie path compression a implementácie optimalizujúce operáciu `union` sa autorov nepodarilo dohľadať.

1.3.1 Základná implementácia

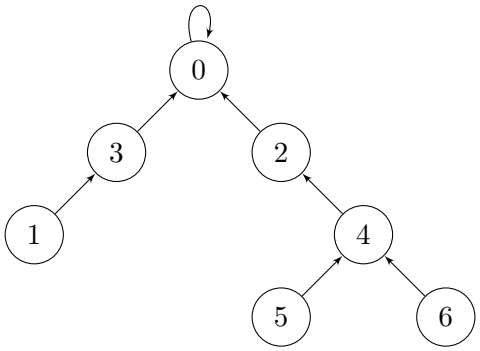
Ako možno vyvodiť z názvu, táto implementácia nevyužíva žiadne spôsoby pre urýchlenie jednotlivých operácií – jedná sa o veľmi jednoduché riešenie. Operácia `find` je vykonané pomocou prechodu cesty od prvku ku koreňu, ktorý reprezentuje identifikátor množiny – ten obsahuje slučku, preto nám stačí nájsť vrchol splňujúci so slučkou.

Rovnako ako operácia `find` aj operácia `union` je naimplementovaná veľmi jednoducho. Operáciu `union` si popíšeme na prípade zjednotenia dvoch prvkoch z rôznych množín:

1. pomocou funkcie `find` nájdeme identifikátory množín, do ktorých patria prvky určené pre zjednotenie

2. skontrolujeme, či sa jedná o rôzne množiny (identifikátory množín získané v predošlom kroku sú rôzne)

3. do vrcholu obsahujúceho prvý prvok určený pre zjednotenie vytvoríme hranu z koreňa množiny, do ktorej patrí prvok druhý



Ukážka 1.3: Výsledok volania operácie `union(2,6)` na množiny reprezentované v ukážke 1.2 pri využití základnej implementácií

Ako si možno všimnúť v ukážke 1.3, táto implementácia nerieši hĺbku stromu a teda operácia `find` môže trvať zbytočne dlho (až $O(n)$, kde n je počet prvkov množiny), preto sa v ďalších podkapitolách pozrieme ako možno operácie optimalizovať, aby sme dosiahli lepších výsledkov z hľadiska rýchlosti operácií, ktoré sú závislé na hĺbke stromuŁ.

1.3.2 Zjednotenie podľa rádu

Táto implementácia optimalizuje hĺbku stromu pomocou pravidla určujúceho akým spôsobom sa vytvorí hrana medzi stromami množín pri vykonaní zjednotenia. Operácia `find` je ponechana bez zmien. Pre túto implementáciu je potrebné udržovať informáciu o ráde stromu každej množiny.

Pravidlo pre zjednotenie hovorí o tom, že pri prepojení stromov množín obsahujúcich prvky určené pre zjednotenie vytvoríme hranu z koreňa stromu nižšieho rádu, do koreňa stromu vyššieho rádu (v prípade, že sa jedná o stromy rovnakého rádu, na orientácií tejto hrany nezáleží).

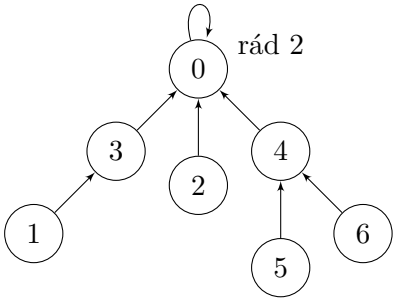
Tým spôsobíme to, že nový strom je maximálne rádu o jedna väčšieho než maximum z pôvodných dvoch stromov. Zatiaľ čo v základnej implementácií by vznikal strom, ktorého hĺbka môže byť až súčtom hĺbok oboch stromov. Vďaka tejto skutočnosti bude vo veľa prípadoch hĺbka stromu menšia ako pri použití základnej implementácie a teda aj pri operácií `find` sa vykoná menší počet operácií (cesta do koreňa bude kratšia).

Operácia `union` dvoch prvkov z rôznych množín sa vykoná nasledovne:[4]

1. pomocou funkcie `find` nájdeme identifikátory množín oboch prvkov
2. skontrolujeme, či sa jedná o rôzne množiny (identifikátory množín získané v predošlom kroku sú rôzne)

[4] prvé dva kroky sú identické ako v prípade základnej implementácie

3. vytvoríme orientovanú hranu z koreňa stromu nižšieho rádu, do koreňa stromu vyššieho rádu (ak rády boli identické na orientácií hrany nezáleží)
4. ak boli stromy identického rádu upravíme rád novovzniknutého stromu

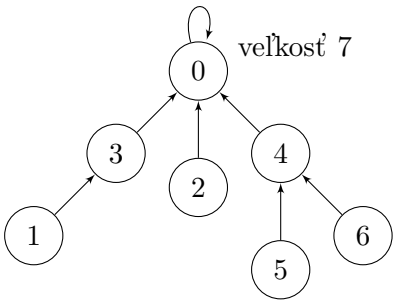


Ukážka 1.4: Výsledok volania operácie `union(2,6)` na množiny reprezentované v ukážke 1.2 pri využití zjednotenia podľa rádu

1.3.3 Zjednotenie podľa veľkosti

Táto implementácia rovnako ako zjednotenie podľa rádu upravuje operáciu `union` a operáciu `find` ponecháva bez zmien. Pre túto implementáciu je potrebné udržovať informáciu o počet vrcholov stromu každej množiny.

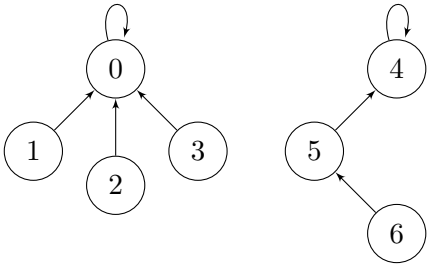
Modifikacia zjednotenia spočíva vo vzniku pravidla, ktoré určuje ako vznikne hrana medzi dvoma stromami množín po zjednotení. Jediný rozdiel od zjednotenia na základe rádu je v tom, že tentokrát je pre nás dôležitá informácia o veľkosti stromu. Táto modifikácia spôsobuje to, že nový strom bude mať vo väčšine prípadov menšiu hĺbku ako po zjednotení pomocou základnej implementácie a teda aj pri operácií `find` sa vykoná menší počet operácií (cesta do koreňa bude kratšia).



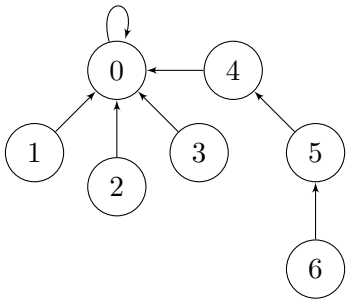
Ukážka 1.5: Výsledok volania operácie `union(2,6)` na množiny reprezentované v ukážke 1.2 pri využití zjednotenia podľa veľkosti

Táto implementácia môže vytvárať stromy väčšej hĺbky a to aj pri malom počte prvkov (vrcholov), čo môže viesť aj k zhoršeniu rýchlosti operácie `find`

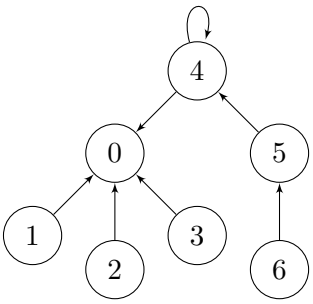
v porovnaní so zjednotením podľa rádu a to napríklad v prípade existencie stromu obsahujúceho tri vrcholy, ale rádom 2 a stromu obsahujúceho štyri vrcholoch, ale rádom 1.



Ukážka 1.6: Príklad stromov reprezentujúcich množiny, ktoré pri zjednotení podľa rádu budú mať menšiu hĺbku ako pri zjednotení podľa veľkosti



Ukážka 1.7: Výsledok volania operácie `union(0,4)` na množiny reprezentovanej v ukážke 1.6 pri využití zjednotenia podľa veľkosti



Ukážka 1.8: Výsledok volania operácie `union(0,4)` na množiny reprezentovanej v ukážke 1.6 pri využití zjednotenia podľa rádu

Ako možno vidieť na ukážkach 1.7 a 1.8 výsledné stromy sa líšia v hĺbkach, ktorá ovplyvňuje časovú zložitosť operácie `find`.

1.3.4 Kompresia cesty

Na rozdiel od predošlých optimalizácií táto implementáciu mení operáciu `find`, nie `union`. Funkcia `union` prebieha rovnako ako v prípade základnej implementácie. [9]

Modifikácia operácie `find` spočíva v tom, že po nájdení identifikátoru množiny sú všetky hrany vrcholov nachádzajúcich sa na ceste z vrcholu obsahujúceho prvok, ktorého množinu sme chceli identifikovať, nahradené hranou do koreňa množiny.

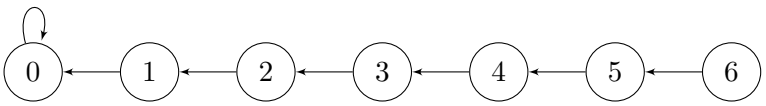
Tým redukujeme dĺžky ciest v rámci stromu, prípadne to môže viesť až k redukcií hĺbky stromu (za predpokladu, že sa pokúšame nájsť identifikátor na základe prvku, ktorý sa nachádza na jedinej najdlhšej ceste v strome, viď ukážka 1.10).

Operácia `find` teda pozostáva z nasledujúcich dvoch krokov:

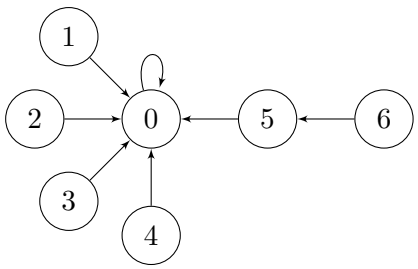
1. nájdeme koreň stromu, ktorý obsahujúceho špecifikovaný prvok

2. prejdeme cestu od vrcholu obsahujúceho prvok do koreňa znovu a pre každý vrchol na ceste vykonáme nahradu pôvodnej hrany za hranu vedúcu do koreňa

Pre názornosť operácie `find` v nasledujúcich implementáciach použijeme množinu reprezentovanú nasledovne:



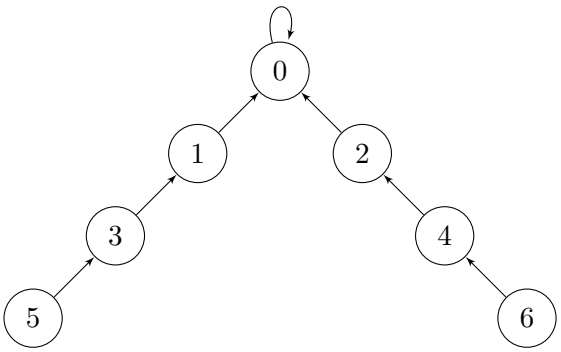
Ukážka 1.9: Množina reprezentovaná ako cesta



Ukážka 1.10: Výsledok volania operácie `find(5)` na množine reprezentovanej v ukážke 1.9 pri využití kompresie cesty

1.3.5 Delenie cesty

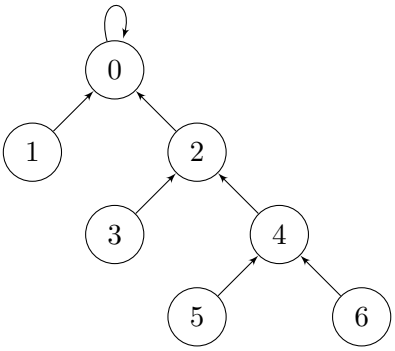
Delenie cesty rovnako ako kompresia cesty modifikuje operáciu `find`, no dĺžku cesty z vrcholu do koreňa zmenšuje, neminimalizuje.[5] Delenie cesty po nájdení identifikátoru znovu prejde celú cestu a každému vrcholu zmení cieľ jeho hrany do rodiča jeho pôvodného rodiča (viď ukážka 1.11).



Ukážka 1.11: Výsledok volania operácie `find(6)` na množine reprezentovanej v ukážke 1.9 pri využití delenia cesty

1.3.6 Pólenie cesty

Táto implementácia je veľmi podobná implementácií delenia cesty. Rozdiel je ten, že pri pólení ciest vykonávame zmenu hrany iba na každom druhom vrchole na ceste z prvku, na ktorý sme volali operáciu `find` a koreňom jeho množiny (viď ukážka 1.12). V prípade delenia cesty však vykonávame zmenu hrany vrcholu v každom vrchole na ceste do koreňa.



Ukážka 1.12: Výsledok volania operácie `find(6)` na množine reprezentovanej v ukážke 1.9 pri využití pólenia cesty

[5] minimalizácia dĺžky cesty môže nastať v prípade viacnásobného využitia funkcie, alebo v prípade veľmi krátkej cesty

1.4 Porovnanie implementácií

Základná implementácia dátovej štruktúry Union-Find má výhodu vo svojej jednoduchosti, no jej zásadnou nevýhodou je jej rýchlosť, ktorá bude v porovnaní s optimalizovanými verziami pri väčšom počte volaní operácií dátovej štruktúry znateľná.

Implementácie zjednotenie podľa rádu a zjednotenie podľa veľkosti optimalizujú rýchlosť (v porovnaní so základnou implementáciou často znižujú hĺbku stromov množín pri operácií `union`), no cenou za túto vyššiu rýchlosť sú väčšie nároky na pamäť.

Implementácie kompresia cesty, delenie cesty a pólenie cesty optimalizujú rýchlosť (znižujú hĺbku stromov množín pri každom volaní operácie `find` na prvok, ktorý nie je koreňom stromu množiny, alebo s koreňom priamo spojený hranou), no cenou za túto vyššiu rýchlosť je väčší počet operácií v rámci operácie `find`.

Pri výbere akým spôsobom budeme implementovať túto dátovú štruktúru je potrebné vedieť, či budem vykonávať veľké množstvo operácií a teda či budeme potrebovať optimalizovať. Ak je odpoveď záporná, je základná implementácia najjednoduchšou voľbou. Ak predpokladáme, že bude pre nás rýchlosť podstatná, je potrebné si položiť otázku, či si môžem dovoliť využitie pamäti naviac, alebo budem radšej vykonávať väčšie množstvo operácií v rámci operácie `find`. Medzi rýchlosťami optimalizovaných verzií nebude príliš veľký rozdiel, teda následný výber už bude zcela na nás.

Klady a zápory nájdeme zhrnuté v tabuľke 1.13.

Názov implementácie	Klady	Zápory
Základná implementácia	jednoduchosť	rýchlosť
Zjednotenie podľa rádu	rýchlosť	pamäťová náročnosť
Zjednotenie podľa veľkosti	rýchlosť	pamäťová náročnosť
Kompresia cesty	rýchlosť	počet operácií v rámci `union`
Delenie cesty	rýchlosť	počet operácií v rámci `union`
Pólenie cesty	rýchlosť	počet operácií v rámci `union`

Ukážka 1.13: Porovnanie kladov a záporov jednotlivých implementácií

Kapitola 2

Frama-C

V tejto kapitole sa budeme venovať frameworku *Frama-C*, ktorý slúži pre verifikáciu zdrojových kódov napísaných v programovacom jazyku C. Vysvetlíme si ako tento framework funguje, popíšeme si, čo je jeho základným stavebným kameňom, rozoberieme si význam a využitie vybraných kľúčových slov jazyka *ACSL*[6] (jazyk používaný v rámci tohto frameworku) a na záver sa pozrieme na pluginy,[7] ktoré sme pri práci s týmto frameworkom použili.

2.1 Čo to je Frama-C?

Frama-C je open-source[8] framework slúžiaci pre verifikáciu zdrojových kódov. Jeho použitím sa užívateľ snaží odhaliť neočakávané chovania a dokázať správnu funkčnosť zdrojového kódu alebo implementácie. Framework umožňuje užívateľovi definovať požiadavky na chovanie časti kódu a vďaka nim ukázať, že jeho implementácia tieto požiadavky splňuje. Pri pečlivej tvorbe požiadavkov možno povedať, že požadovaná časť kódu funguje správne a neobsahuje žiadne chyby [10].

Framework tiež podporuje pluginy, ktorých použitie môže overovanie spríjemniť, prípadne uľahčiť. My sme sa rozhodli v rámci našej práce použiť niekoľko z nich, konkrétne *WP*, *EVA* a *RTE*. Tieto pluginy si bližšie predstavíme v podkapitole 2.4.

2.2 Kontrakt

Framework pre špecifikáciu požiadavkov používa takzvaný *kontrakt* – jedná sa o základný stavebný kameň tohto frameworku.

[6] ANSI/ISO C Specification Language

[7] rozšírenia pridávajúce funkcionality

[8] zdrojové kódy sú verejne prístupné a ktokoľvek ich môže modifikovať pri dodržaní licenčných práv

Kontrakt pozostáva z požiadavkov, ktoré by mali platiť pre jednotlivé funkcie, cykly či vetvenia a nachádzajú sa v anotačných komentároch. Tieto požiadavky sú vyjadrené pomocou jazyka *ACSL*, ktorý obsahuje vstavané funkcie a predikáty, datové typy premenných a takzvaný ghost code a ghost premenné [11]. Kontrakt sa bežne skladá zo vstupných a výstupných požiadavkov, pre ktoré sa používajú kľúčové slová *requires* a *ensures*. Tieto kľúčové slová a niekoľko ďalších si bližšie vysvetlíme v nasledujúcej časti tejto kapitoly.

```
1 //@ requires <some_condition>;
2 void print ( int number );
```

Ukážka 2.1: Jednoriadkový kontrakt

```
1 /*@
2   @ requires <some_condition>;
3   @
4   @ ensures <some_condition>;
5 */
6 void append ( char ** destination, char * source );
```

Ukážka 2.2: Viacriadkový kontrakt

Požiadavky, ktoré nájdeme v rámci kontraktu môžeme rozdeliť do troch kategórií a to na tie, ktoré musia byť splnené pred, počas a po vykonaní kódu. Nakoľko takéto premýšľanie nad kontraktom je veľmi prirodzené budú aj naše kontrakty tvorené najprv požiadavkami, ktoré musia byť splnené pred vykonaním daného kusu kódu, následne požiadavkami súvisejúcimi so samotným behom a až na záver požiadavky, ktoré musia byť splnené po vykonaní bloku kódu.

2.3 Kľúčové slová jazyka ACSL

Anotačný jazyk *ACSL* slúži pre špecifikáciu požiadavkov na funkcie, cykly a vetvenia, jeho kľúčové slová môžu slúžiť pre špecifikáciu vstupných alebo výstupných podmienok, logických výrazov a funkcií označujúcich kód prístupný len v rámci kontraktov[9] alebo aj identifikátory stavu premenných (napríklad stav parametru pred a po vykonaní funkcie).

Kľúčové slová jazyka *ACSL* delíme na termy, predikáty a zvyšné kľúčové slová. Termy a predikáty sú špecifické tým, že na rozdiel od zvyšku sa ich názvy začínajú spätným lomítkom (napríklad *true* a *false*) [11].

[9] takzvaný ghost code

2.3.1 Logické konštrukcie

Kontrakt pozostáva z požiadavkov, ktoré sú formulované ako logické výrazy. Tie môže byť vhodné reťaziť a tvoriť tak komplexnejšie požiadavky. To možno docieliť logickými spojkami, ktorými sú negácia (`!`) konjunkcia (`&&`), disjunkcia (`||`), implikácia (`==>`) alebo ekvivalencie (`<==>`).

Mimo používania logických spojok možno v rámci logických výrazov využívať zástupné logické výrazy reprezentujúce pravdu (`\true`) a nepravdu (`\false`).

2.3.2 Klauzula requires

Klauzula *requires* sa využíva v rámci kontraktu funkcie a špecifikuje vstupné požiadavky, napríklad aké musia byť hodnoty parametrov a aká musí byť štruktúra ich dát. Pre pokračovanie v analýze požiadavkov na funkciu je potrebné, aby klauzuly *requires* boli splnené (ak špecifikované vstupné požiadavky nie sú splnené, nemôžeme garantovať ako bude vyzerať výsledok). V prípade väčšieho množstva požiadavkov je možné ich reťazenie pomocou operátoru konjunkcie alebo možno viackrát použiť túto klauzulu.

Použitie klauzuly *requires* si ukážeme na funkcií `logarithm`, ktorá vráti hodnotu dekadického logaritmu pre hodnotu nachádzajúcu sa v argumente. Nakoľko je logaritmus definovaný iba pre kladné hodnoty, požadujeme kladný argument aj v rámci našej funkcie (viď kontrakt v ukážke 2.3).

```
1 //@ requires number > 0;
2 int logarithm ( int number );
```

Ukážka 2.3: Príklad klauzule *requires*

Ak kontrakt neobsahuje túto klauzulu predpokladá sa, že nemáme žiadne vstupné požiadavky [11]. To si ukážeme na funkcií `printOne`, ktorá vypíše na štandardný výstup číslo jeden (viď ukážka 2.4, ktorá ukazuje alternatívny zápis kontraktu v situácií, kde nemáme žiadne vstupné požiadavky).

```
1 //@ requires \true;
2 void printOne ( void );
```

Ukážka 2.4: Ekvivalent kontraktu bez klauzule *requires*

2.3.3 Klauzula ensures

Klauzula *ensures* v rámci kontraktu funkcie dopĺňa klauzulu *requires*. Táto klauzula špecifikuje výstupné podmienky funkcie, napríklad aká musí byť návratová hodnota alebo čo musí platiť pre vstupné parametry, **ale až na konci funkcie** – napríklad v prípade zmeny hodnôt v poli, ktoré funkcia

obdržala ako argument. Pre špecifikáciu viacerých výstupných podmienok možno využiť operátor konjunkcie alebo viackrát použiť túto klauzulu.

Použitie klauzuly *ensures* si ukážeme na funkcii `square`, ktorá zmení hodnotu celočíselného argumentu na hodnotu jeho druhej mocniny. Keďže druhá mocnina ľubovolného celého čísla je číslo nezáporné, budeme požadovať nezápornú hodnotu argumentu na konci funkcie – to dosiahneme pomocou klauzule *ensures* (viď kontrakt v ukážke 2.5).

```
1 //@ ensures ( * number ) >= 0;
2 void square ( int * number );
```

Ukážka 2.5: Klauzula *ensures*

Podobne ako v prípade klauzuly *requires*, tak aj pri vynechaní klauzuly *ensures* sa predpokladá, že na funkciu nie sú kladené žiadne výstupné podmienky. Teda je táto možnosť ekvivalentná nasledujúcemu zápisu:

```
1 //@ ensures \true;
2 void printOne ( void );
```

Ukážka 2.6: Ekvivalent kontraktu bez klauzule *ensures*

Táto klauzula zároveň obsahuje jednu limitáciu a to konkrétne v prípade použitia kľúčového slova `goto`. Pri použití tohto kľúčového slova nie je klauzula *ensures* overovaná. [11]

2.3.4 Klauzula assigns

Táto klauzula sa využíva v rámci kontraktu funkcie a špecifikuje, ktoré premenné sa môžu (**nemusia**) v rámci funkcie meniť. V prípade viacerých premenných je možné využiť klauzulu viackrát alebo ich vypísať ako zoznam (v takom prípade musia byť premenné oddelené čiarkami). Premenné, ktoré sa v rámci tejto klauzule špecifikujú musia byť argumentom funkcie alebo globálne premenné.[10]

Použitie tejto klauzuly si ukážeme na funkcií `swap`, ktorá bude vykonávať výmenu obsahu dvoch ukazatelov obsahujúcich hodnoty typu `int`. Nakoľko funkcia zmení obsah tejto pamäti vstupujúcej do funkcie je vhodné tieto zmeny označiť a to práve pomocou klauzule *assings* (viď ukážka 2.7).

[10] nemôže sa jednať o lokálne premenné funkcie nakoľko tieto premenné pri volaní funkcie ešte neexistujú

```
1 //@ assigns * number1, * number2;
2 void swap ( int * number1, int * number2 );
```

Ukážka 2.7: Klauzula *assings*

Nakoľko niektoré funkcie nemusia obsah premenných meniť vôbec je možné túto klauzulu využiť s kľúčovým slovom *nothing*. Takúto situáciu si ukážeme na funkcií `printOne`, ktorá nemodifikuje obsah žiadnej premennej.

```
1 //@ assigns \nothing;
2 void printOne ( void );
```

Ukážka 2.8: Kontrakt popisujúci situáciu, kedy nemodifikujeme obsah žiadnej premennej

Pokiaľ klauzulu *assigns* úplne vynecháme neznamená to, že funkcia nemodifikuje obsah žiadnej premennej. Znamená to, že bude kontrola modifikácie premenných vynechaná.

2.3.5 Klauzula allocates

Jedná sa o klauzulu využívanú v rámci kontraktu funkcie. Pomocou tejto klauzuly možno špecifikovať množinu ukazatelov, ktoré môžu byť v rámci funkcie alokované (môžu, **nemusia**). Ostatné ukazatele by nemali zmeniť svoj alokačný stav. Alokačný stav môže byť napríklad *statický*, *register*, *dynamický* alebo *nealokovaný* [11].

Využitie tejto klauzule si ukážeme na funkcií `allocatePointer`, ktorej argumentom bude ukazatel na hodnotu typu `int`. V rámci tejto funkcie bude tento ukazatel dynamicky alokovaný.

```
1 //@ allocates number;
2 void allocatePointer ( int * number );
```

Ukážka 2.9: Klauzula *allocates*

To, že funkcia nebude žiadny ukazatel alokovať je možné špecifikovať pomocou kľúčového slova *nothing* (viď ukážka 2.10).

```
1 //@ allocates \nothing;
2 void printOne ( void );
```

Ukážka 2.10: Kontrakt popisujúci situáciu, kedy nealokujeme žiadnu pamäť

V prípade, že táto klauzula nie je použitá predpokladá sa, že v rámci funkcie môže prebehnúť zmena alokačného stavu pre ľubovoľný ukazatel (bude alokovaný).

2.3.6 Klauzula frees

Klauzula *frees* sa využíva v rámci kontraktu funkcie. Jedná sa o klauzulu doplňujúcu klauzulu *allocates* – je to v podstate jej opak. V rámci tejto klauzuly možno špecifikovať množinu ukazatelov, ktorých pamäťové bunky sa behom funkcie môžu uvoľniť.

Túto klauzulu si možno ukázať priamo na štandardnej funkcií jazyka C a to `free`, ktorá uvoľní pamäť, na ktorú ukazuje ukazatel v argumente funkcie.

```
1 //@ frees * pointer;
2 void free ( void * pointer );
```

Ukážka 2.11: Klauzula *frees*

Ak funkcia žiadnu pamäť neuvoľňuje je možné túto klauzulu použiť spoločne s kľúčovým slovom *nothing* (viď ukážka 2.12).

```
1 //@ frees \nothing;
2 void printOne ( void );
```

Ukážka 2.12: Kontrakt popisujúci situáciu, kedy neuvoľnuje žiadnu premennú

V prípade, že táto klauzula nie je použitá predpokladá sa, že v rámci funkcie môže byť uvoľnené ľubovoľné pamäťové bunky.

2.3.7 Kľúčové slovo predicate

Predicate[11] je vo vedeckej disciplíne logiky považovaný za logický výrazy, o ktorom možno rozhodnúť či je pravdivý, alebo nepravdivý [12]. Nakoľko jazyk *ACSL* používa pre popis vstupných a výstupných podmienok logické výrazy, ktoré sa môžu opakovať – môžu byť použité ako vstupné aj výstupné podmienky alebo vo viacerých funkciách – je vhodné znižovať redundanciu, čo možno docieliť práve tvorbou predikátov.

Toto kľúčové slovo nám teda umožňuje špecifikovať pomenované logické výraz obsahujúce rôzne premenné[12], ktoré možno následne využívať v rámci kontraktov a tak zprehľadniť kód a znížiť redundanciu komplikovaných výrazov v rámci kontraktov.

[11] v preklade *predikát*
[12] argumenty

Toto kľúčové slovo si ukážeme na funkcií, ktorá zmení hodnotu argumentu na absolutnú hodnotu jeho druhej odmocniny. Naviac keďže je v obore reálnych čísel (a teda aj v obore celých čísel) definovaná len pre nezáporné hodnoty budeme požadovať, aby bola hodnota argumentu pred aj po vykonaní funkcie nezáporná. To docielime použitím kľúčových slov *requires* a *ensures*, no pre zníženie redundancie si zadefinujeme predikát *is_nonnegative*, ktorý bude predstavovať logický výraz určujúci či jeho argument ukazuje na nezápornú hodnotu (viď ukážka 2.13).

```
1  /*@ predicate is_nonnegative ( integer * number ) = (
2    @      ( * number ) >= 0
3    @ );
4  */
5
6  /*@
7    @ requires is_nonnegative ( number );
8    @
9    @ ensures is_nonnegative ( number );
10 */
11 void squareRoot ( int * number );
```

Ukážka 2.13: Klauzula *predicate*

2.3.8 Kľúčové slovo logic

Kľúčové slovo *logic* umožňuje v rámci jazyka *ACSL* špecifikovať funkcie, ktoré možno využiť v rámci predikátov a logických výrazov. Tieto funkcie nemôžu využivať premenné ani cykly.[13] Tento problém je často možné vyriešiť použitím rekurzie.

Príkladom logickej funkcie môže byť funkcia, ktorá získa maximum z poľa. Implementáciu tejto logickej funkcie možno vidieť v ukážke 2.14.

```
1  /*@ logic integer max_in_array ( integer * numbers,
2    @                             integer length ) = (
3    @      ( length == 1 )
4    @      ?
5    @      numbers [ 0 ]
6    @      :
7    @      max ( numbers [ 0 ],
8    @            max_in_array ( numbers + 1, length - 1 )
9    @          )
10   @ );
11 */
```

Ukážka 2.14: Klauzula *logic*

[13] gramatika jazyka *ACSL* to v tele klauzule *logic* nepovoľuje

2.3.9 Klauzula loop variant

Klauzula *loop variant* umožňuje vykonať kontrolu správnosti cyklu pomocou kontraktu – umožňuje predísť zacykleniu a to pomocou špecifikácie matematického výrazu, pre ktorý platia nasledujúce pravidlá:

- hodnota výrazu musí byť celočíselná
- na začiatku každej iterácie musí byť hodnota výrazu menšia ako v iterácií predošlej
- až do ukončenia cyklu nesmie byť hodnota výrazu záporná

Na základe všetkých týchto podmienok možno špecifikovať výraz, ktorý zaručí, že instancia problému sa zmenšuje a teda, že cyklus zaručene skončí. Avšak použitie tejto klauzule nemusí byť vždy jednoduché, dokonca v niektorých situáciách môže byť tvorba vhodného výrazu nereálna (napríklad pri načítaní vstupu až do doby kedy na vstupe dostaneme `EOF`[14]).

Klauzulu *loop variant* si ukážeme na funkcií `printArray`, ktorá vypíše všetky čísla nachádzajúce sa v poli `array` s dĺžkou `n`:

```
1 void printArray ( int * array, int n ) {
2     //@ loop variant n - i;
3     for ( int i = 0; i < n; i ++ ) {
4         printf ( "%d ", array [ i ] );
5     }
6 }
```

Ukážka 2.15: Klauzula *loop variant*

2.3.10 Klauzula loop invariant

Klauzula *loop invariant* umožňuje špecifikovať hodnoty, ktoré môže nadobudnúť iteračná premenné v rámci cyklu. Tým možno docieliť napríklad to, že nepristupujeme mimo pole. Pre výraz obsiahnutý v tejto klauzule zároveň platí, že pri štandardnom ukončení cyklu musí byť podmienka *loop invariantu* splnená aj s hodnotou po ukončení poslednej iterácie (príklad viď ukážka 2.16).

```
1 void printArray ( int * array, int n ) {
2     //@ loop invariant 0 <= i <= n;
3     for ( int i = 0; i < n; i ++ ) {
4         printf ( "%d ", array [ i ] );
5     }
6 }
```

Ukážka 2.16: Klauzula *loop invariant*

[14] End-of-file – koniec súboru

Špeciálne pre túto klauzulu platí, že nemusí byť splnená v prípade, že je cyklus ukončený pomocou kľúčových slov `break`, `return`, alebo `goto`.[11]

2.3.11 Klauzula loop assigns

Klauzula *loop assigns* je alternatívou klauzuly *assigns* špecificky určenej pre cykly. Rovnako ako *assigns* určuje premenné, ktoré môžu meniť svoju hodnotu v rámci funkcie, *loop assigns* umožňuje špecifikovať premenné, ktoré môžu meniť svoju hodnotu v rámci iterácií cyklu.

Príklad použitia tejto klauzule si ukážeme na funkcií `clearArray`, ktorá vynuluje všetky hodnoty nachádzajúce sa v celočíselnom poli `array` s dĺžkou `n`.

```
1 void clearArray ( int * array, int n ) {
2     //@ loop assigns array[0..(n-1)];
3     for ( int i = 0; i < n; i ++ ) {
4         array [ i ] = 0;
5     }
6 }
```

Ukážka 2.17: Klauzula *loop assigns*

V rámci ukážky 2.17 sme použili konštrukciu `array[0..(n-1)]`, ktorá špecifikuje všetky pamäťové bunky, v rámci ktorých sú uložené dáta poľa.

2.3.12 Klauzula behavior

Táto klauzula umožňuje tvorbu kontraktu pre funkcie, ktorých chovanie sa líši napríklad na základe charakteru vstupných dát. Chovania funkcie môžu byť v rámci kontraktu pomenované pridaním názvu za klauzulu *behavior*, názov chovania však nie je povinný.

Príklad použitia klauzuly *behavior* si ukážeme na funkcií `append`, ktorá pridáva prvok na koniec dynamicky alokovaného poľa. Chovanie tejto funkcie sa bude líšiť z hľadiska alokovaných a uvoľnených prostriedkov podľa toho či budeme potrebovať pole zväčšiť, alebo nie.

V prípade použitia klauzule *behavior* môže nastať situácia, že existuje podmienka, ktorá platí pre všetky možné chovania funkcie, tieto požiadavky nie je potreba špecifikovať opakovane, možno ich špecifikovať pred prvým použitím klauzule (viď riadok číslo 7 ukážky 2.18).

V rámci kontraktu v ukážke by bolo možné v oboch prípadoch špecifikovať aj napríklad premenné, do ktorých sa zapisuje (*assigns*), to sme však pre zvýšenie čitateľnosti kontraktu vynechali nakoľko to nie je podstatou tejto ukážky.

2.3.13 Klauzula complete

V niektorých prípadoch môže byť vhodné vytvoriť kompletný popis chovania funkcie. Toto možno označiť v rámci jazyka *ACSL* kľúčovým slovom *complete* následovaným názvami všetkých chovaní, ktoré tvoria tento kompletný popis, v prípade, že sa jedná o všetky chovania možno zoznam názvov chovaní nahradiť slovom *behaviors*.

2.3.14 Klauzula disjoint

Mimo označenie, že sa jedná o kompletný popis chovaní môže byť vhodné označiť, že niektoré z chovaní sú vzájomne disjunktné.[15] To možno docieliť použití kľúčového slova *disjoint*, po ktorom následujú názvy vzájomne disjunktných chovaní, alebo slovo *behaviors* ak sú všetky chovania v kontrakte vzájomne disjunktné.

Táto klauzula sa nachádza aj v ukážke 2.18, nakoľko vždy môže nastať iba jedna zo situácií – máme dosť miesta a teda prvok vložíme (*just_append*), alebo musíme pole zväčšiť a až následne vložiť nový prvok (*reallocate*).

```
1  typedef struct CArray {
2      int * array;
3      int capacity, size;
4  } CArray;
5
6  /*@
7    @ requires toAppend -> capacity > 0 && toAppend -> size >= 0;
8    @ behavior reallocate:
9    @      allocates toAppend -> array;
10   @      frees toAppend -> array;
11   @
12   @ behavior just_append:
13   @      allocates \nothing;
14   @      frees \nothing;
15   @
16   @ complete behaviors;
17   @ disjoint behaviors;
18 */
19 void append ( CArray * toAppend, int number );
```

Ukážka 2.18: Klauzula *behavior*

2.3.15 Klauzula assumes

Klauzula *assumes* je veľmi podobná klauzule *requires* – obe tieto klauzule špecifikujú vstupné podmienky funkcie. Rozdiel medzi nimi je ten, že *assumes* je klauzula špecifická pri delení chovaní (*behavior*) do jednotlivých možností.

[15] priechod funkcie nemôže spĺňať viac ako jedno z týchto chovaní

Pomocou tejto klauzule sa definujú požiadavky, ktoré musia byť splnené, aby sa aplikovalo konkrétne chovanie (*behavior*).

Použitie tejto klauzule si ukážeme rozšírením kontraktu funkcie `append` z ukážky 2.18.

```
1  typedef struct CArray {
2      int * array;
3      int capacity, size;
4  } CArray;
5
6  /*@
7    @ requires toAppend -> capacity > 0 && toAppend -> size >= 0;
8    @ behavior reallocate:
9    @      assumes toAppend -> size == toAppend -> capacity;
10   @
11   @      allocates toAppend -> array;
12   @      frees toAppend -> array;
13   @
14   @ behavior just_append:
15   @      assumes toAppend -> size < toAppend -> capacity;
16   @
17   @      allocates \nothing;
18   @      frees \nothing;
19   @
20   @ complete behaviors;
21   @ disjoint behaviors;
22 */
23 void append ( CArray * toAppend, int number );
```

Ukážka 2.19: Klauzula *assumes*

2.3.16 Kľúčové slovo ghost

Kľúčové slovo *ghost* sa používa pre definíciu takzvaného ghost kódu. Všeobecne je syntax ghost kódu veľmi podobná ako syntax jazyka C. Ghost kód má naviac prístup k bežným premenným, ale nesmie ich modifikovať, smie modifikovať iba ghost premenné [11].

2.3.17 Kľúčové slovo \result

Jedná sa o kľúčové slovo pomocou ktorého sa možno odkazovať na návratovú hodnotu funkcie. Príklad si ukážeme na funkcií `square`, ktorá vracia hodnotu druhej mocniny jej argumentu.

```
1 //@ ensures \result == number * number;
2 int square ( int number );
```

Ukážka 2.20: Kľúčové slovo *\result*

2.3.18 Kľúčové slovo \null

Kľúčové slovo *null* je vstavaný výraz reprezentujúci `NULL` – makro jazyka C, ktoré predstavuje ukazatel, ktorý neukazuje na žiadny objekt.

2.3.19 Kľúčové slovo \nothing

Toto kľúčové slovo predstavuje v rámci jazyka *ACSL* prázdnu množinu. Jeho použitie je možné napríklad pri funkciách, ktoré žiadnu pamäť nealokujú, neuvoľňujú, alebo nemenia obsah argumentov ani globálnych premenných.

Príklad použitia si ukážeme na funkcií `square` a to rozšírením kontraktu z ukážky 2.20.

```
1  /*@
2    @ allocates \nothing;
3    @
4    @ assigns \nothing;
5    @
6    @ fress \nothing;
7    @
8    @ ensures \result == number * number;
9  */
10 int square ( int number );
```

Ukážka 2.21: Kľúčové slovo *nothing*

2.3.20 Predikát \valid

Tento predikát je možno použiť pri práci s ukazatelmi[16]. Predikát *valid* je vyhodnotený ako pravda práve vtedy, keď jeho parametrom je ukazatel na adresu, ktorú je bezpečné čítať aj do nej zapisovať.

2.3.21 Predikát \freeable

Tento predikát je vhodný pri práci s dynamicky alokovanou pamäťou. Predikát *freeable* je totižto vyhodnotený ako pravda iba pokiaľ jeho parameter je ukazatel, ktorý možno bezpečne uvoľniť.

2.3.22 Kľúčové slová \forall a \exists

Kľúčové slová *forall* a *exists* reprezentujú matematické kvantifikátory ∀ a ∃. Pomocou nich je možné napríklad detekovať, či sa prvok nachádza v poli, alebo či sú všetky prvky poľa v požadovanom rozsahu.

Príklad si ukážeme na funkcií `detect`, ktorá vracia hodnotu 1, ak sa v poli `array` dĺžky `n` nachádza číslo `number`, inak vracia hodnotu 0. Tvorbu tohto

[16] iného typu ako `void *`

kontraktu si ukážeme pomocou kľúčového slova *forall* aj kľúčového slova *exists*.

```
1 //@ ensures ( \forall integer i; 0 <= i < n ==> array [ i ] !=
      number ) ==> \result == 0;
2 int detect ( int * array, int n, int number );
```

Ukážka 2.22: Kľúčové slovo *forall*

```
1 //@ ensures ( \exists integer i; 0 <= i < n ==> array [ i ] ==
      number ) ==> \result == 1;
2 int detect ( int * array, int n, int number );
```

Ukážka 2.23: Kľúčové slovo *exists*

2.3.23 Labely

Labely sú kľúčové slová, ktoré sa používajú pre označenie stavu programu pred, prípadne po vykonaní kusu kódu. S labelami sa možno stretnúť aj u niektorých predikátov napríklad *freeable*.

Jazyk *ACSL* má 6 vstavaných labelov a to:

- *Pre*/*Old* – odkazuje na stav programu pred vykonaním bloku kódu
- *Post* – odkazuje na stav programu po vykonaní bloku kódu
- *LoopEntry* – odkazuje na stav programu pred prvým vstupom do bloku cyklu
- *LoopCurrent* – odkazuje na stav programu na začiatku aktuálnej iterácie cyklu
- *Here* – odkazuje na aktuálny stav programu (ten záleží od toho v akej klauzule label použijeme) [13]
  - v prípade použitia v klauzuli `requires` a `assumes` sa tento label chová ako `Pre`
  - v prípade použitia v klauzuli `ensures` sa tento label chová ako `Post`

2.3.24 Kľúčové slovo \at

Hodnota premenných sa môže počas behu funkcie meniť. Avšak v niektorých prípadoch je potreba vedieť hodnotu premennej pred začiatkom funkcie aj na jej konci. To možno docieliť pomocou kľúčového slova *at*. Toto kľúčové slovo má dva argumenty premennú a label. Vďaka tomuto kľúčovému slovu možno napríklad porovnať obsah poľa pred a po vykonaní funkcie.

2.3.25 Kľúčové slovo \old

Jedná sa o kľúčové slovo, ktoré sa používa pre prístup k hodnote, ktorú premenná mala pred vykonaním funkcie (takzvaný pre-state). Jedná sa o syntanktický cukor pre výraz *\at(x,Pre)*.

2.4 Pluginy

Frama-C umožňuje rozširovať fukcionalitu tohto frameworku pomocou tvorby rozšírení (takzvaných pluginov). V rámci tejto práce sme sa rozhodli využiť niekoľko z nich a tie si popíšeme v nasledujúcich podkapitolách.

2.4.1 WP

Plugin *WP* umožňuje dokázovať, že kontrakty budú splnené pre všetky možné priechody kódu. Pri použití tohto pluginu sa využívajú externé dokazovače (*proof assistants*). Jedná sa o všeobecnú náhradu použitia unit testov. [14]

2.4.2 RTE

RTE je považované menší plugin, ktorý je súčasťou pluginu *WP*. Tento plugin sa používa pre generovanie assertov pre odchytávanie potencionálnych chýb za behu programu (*runtime errors*) – tie sa generujú hlavne pri vetveniach. [15]

2.4.3 EVA

EVA (celým názvom *Evolved Value Analysis*) je plugin, ktorý podobne ako *WP* hľadá chyby na základe tvorby všetkých možných priechodov funkciou – pomocou neho možno odvodiť, že kód neobsahuje žiadne chyby behu programu (*runtime errors*) [16]. Pomocou tohto pluginu tiež možno zistiť aké hodnoty môžu jednotlivé premenné nadobúdať.

Kapitola 3

Implementácia a overenie dátovej štruktúry Union-Find

V tejto kapitole rozoberieme našu implementáciu dátovej štruktúry Union-Find v programovacom jazyku C. Pozrieme sa z akých členských premenných sa skladajú štruktúry udržujúce dáta pre jednotlivé implementácie, popíšeme si implementáciu funkcie `makeSet` a základnú implementáciu funkcií realizujúcich operácie `union` a `find` a tiež si povieme, ako sa líšia implementácie týchto funkcií v optimalizovaných implementáciach rozobraných v kapitole 1.

Následne spojíme naše implementácie so znalostmi nadobudnutými v kapitole 2, konkrétne sa budeme zaoberať spôsobom verifikácie funkcií pri jednotlivých implementáciach. Pre verifikáciu sme využili niekoľko predikátov a logickú funkciu, tie si popíšeme v samostatných podkapitolách a na záver zhodnotíme ako verifikácia našich implementácií dopadla.

Naše implementácie tiež obsahujú funkciu `freeSet` a pomocné funkcie `contains` a `swap`, ktorých implementácia a verifikácia je pomerne nezaujímavá, preto ich v tejto kapitole vynecháme, no aj tak ich možno nájsť v prílohe tejto práce.

3.1 Implementácia

V rámci našej implementácie dátovej štruktúry sme riešili problém s odkazovaním na prvky množiny pri volaní funkcií `find` a `union`. Užívateľský najpríjemnejšou možnosťou by bolo využívať priamo hodnotu prvku obsiahnutého v množine – tu by však bolo problematické to, že by tým časová zložitosť oboch funkcií závisela na spôsobe vyhľadania prvku v rámci dátovej štruktúry, čo by mohlo spôsobiť jej degradáciu.

Tento problém sme sa preto rozhodli vyriešiť tým, že tieto funkcie pracujú s indexom, na ktorom je uložený prvok v rámci dátovej štruktúry – tento index užívateľ obdrží ako návratovú hodnotu funkcie `makeSet` pri vložení prvku.

Vďaka tomu nezávisí časová zložitosť od vyhľadávania prvku v štruktúre. Rozhranie ale spolieha na užívateľa, že bude vedieť na akom indexe sa nachádzajú požadované prvky.

V tejto podkapitoly sa stretneme s niekoľkými ukážkami funkcií, pre ich plný kontext v implementácií môže byť vhodné nahliadnuť do zdrojových kódov danej implementácie, ktoré možno nájsť samostatne, ale aj s kontraktmi v prílohe tejto práce.

3.1.1 Udržiavanie dát

Základom implementácie je udržiavanie dát potrebných pre prácu s touto dátovú štruktúru – pre zjednodušenie práce s ňou sme vytvorili štuktúru udržujúcu všetky potrebné dáta. Pre základnú implementáciu a implementácie využívajúce optimalizácie kompresia cesty, delenie cesty a pólenie cesty štruktúra vyzerá rovnako ako v ukážke 3.1.

```
1 typedef struct TUnionFind {
2     int *   parents;
3     int *   elements;
4     int     capacity;
5     int     size;
6 } UnionFind;
```

Ukážka 3.1: Základná štruktúra `UnionFind`

V prípade implementácií zjednotenie podľa rádu a zjednotenie podľa veľkosti však potrebujeme naviac pole rádov, respektíve veľkostí stromov reprezentujúcich jednotlivé množiny. V týchto implementáciach preto štruktúra naviac obsahuje členskú premennú `int * ranks`, respektíve `int * sizes`.

Štruktúra teda bude obsahovať vybrané položky z nasledujúceho výčtu:

- `parents` – pole udržujúce orientované hrany medzi prvkami množiny, špeciálne je potreba označiť koreň stromu obsahujúci identifikátor každej množiny a to je možné vykonať nasledujúcimi spôsobmi:
  1. definícia takzvanej zarážky – hodnoty, ktorá by označovala, že prvok je koreň stromu
  2. tvorba slučky v koreni – tento spôsob implementácie sme použili my
- `elements` – pole hodnôt tvoriacich nezávislé (disjunktné) množiny
- `capacity` – veľkosť dynamicky alokovaných polí tejto štruktúry
- `size` – aktuálny počet prvkov v dynamicky alokovaných poliach tejto štruktúry

- `ranks` – pole reprezentujúce rád stromov (pod)množiny (táto členská premenná sa nachádza iba v implementácií zjednotenie podľa rádu)
- `sizes` – pole reprezentujúce počet vrcholov stromov (pod)množiny (táto členská premenná sa nachádza iba v implementácií zjednotenie podľa veľkosti)

3.1.2 Základná implementácia funkcie makeSet

Funkcia `makeSet` pozostáva z niekoľkých možných situácií:

- pridávame množinu do neexistujúcej štruktúry Union-Find
  1. skontrolujeme, že štruktúra neexistuje (ukazateľ na ňu je `NULL`)
  2. alokujeme ukazateľ na štruktúru `UnionFind`
  3. inicializujeme veľkosť (`size`) a kapacitu (`capacity`) polí (počiatočná kapacita je konštanta definovaná prostredníctvom makra)
  4. alokujeme polia `parents` a `elements`
  5. pridáme nový prvok do poľa `elements` a vytvoríme slučku v rámci poľa `parents` pre detekciu koreňa stromu
  6. vrátime index, na ktorý bol prvok uložený (v tomto prípade hodnotu 0)
- pridávame množinu pozostávajúcu z hodnoty, ktorá sa nenachádza v žiadnej z už existujúcich množín
  1. skontrolujeme, že štruktúra existuje
  2. skontrolujeme či sa prvok v štruktúre ešte nenachádza
  3. vykonáme kontrolu kapacity polí (ak zistíme, že nemáme kapacitu pre nový prvok vykonáme realokáciu[17] zdvojnásobenie kapacít všetkých polí)
  4. pridáme nový prvok do poľa `elements` a vytvoríme slučku v rámci poľa `parents` pre detekciu koreňa stromu
  5. vrátime index, na ktorý bol prvok uložený
- pridávame množinu pozostávajúcu z hodnoty, ktorá sa už nachádza v niektorej z množín štruktúry
  1. skontrolujeme, že štruktúra existuje
  2. skontrolujeme či sa prvok v štruktúre ešte nenachádza
  3. na štandardný chybový výstup (`stderr`) vypíšeme informáciu o tom, že sme sa pokúsili vložiť prvok, ktorý už v niektorej z množín je

[17] zväčšenie kapacity

4. vrátime hodnotu -1, ktorá reprezentuje chybu pri vkladaní

Ukážku tejto funkcie neuvádzame z dôvodu jej rozsiahlosti, implementáciu tejto funkcie možno nájsť v prílohe tejto práce.

3.1.3 Rozdiely vo funkcií makeSet naprieč implementáciami

V prípade implementácií kompresia cesty, delenie cesty a pólenie cesty je táto funkcia identická ako jej základná implementácia. V implementáciach zjednotenie podľa rádu a zjednotenie podľa veľkosti funkcia obsahuje nasledujúce zmeny:

- alokujeme pole naviac (v implementácií zjednotenie podľa rádu sa jedná o pole `ranks` a v implementácií zjednotenie podľa veľkosti o pole `sizes`)
- pri vkladaní prvku vkládame hodnoty aj do týchto nových polí (v implementácií zjednotenie podľa rádu sa jedná o hodnotu 0, čo znamená, že strom množiny je rádu 0 a v implementácií zjednotenie podľa veľkosti je táto hodnota 1 nakoľko nová množina pozostáva z jednoho prvku)

3.1.4 Základná implementácia funkcie find

Ako už vieme operácia `find` má za úlohu získať identifikátor množiny, do ktorej patrí prvok, pre jej realizáciu sme vytvorili funkciu `find`. Identifikátorom množiny je v rámci našej implementácie index prvku, ktorý je koreňom stromu množiny – o koreni vieme, že obsahuje slučku, ktorá sa nachádza v závislostiach v poli `parents`. Koreň stromu preto možno detekovať tak, že nájdeme vrchol, do ktorého sa možno dostať pomocou orientovaných hrán v poli `parents` a zároveň bude obsahovať slučku (za rodiča bude mať seba samého).

```
1  int find ( int elementIndex, UnionFind * set ) {
2      if ( elementIndex >= 0 && elementIndex < set -> size ) {
3          int id = set -> parents [ elementIndex ];
4          while ( id != set -> parents [ id ] ) {
5              id = set -> parents [ id ];
6          }
7          return id;
8      }
9      else {
10         fprintf ( stderr, "Invalid element index!\n" );
11         return -1;
12     }
13 }
```

Ukážka 3.2: Základná implementácia funkcie `find`

V rámci samotnej funkcie musíme riešiť dve situácie a to:

- index prvku odkazuje mimo pole – túto situáciu detekujeme kontrolou, že index prvku nepatrí do intervalu $\langle 0, n)$, kde n predstavuje veľkosť poľa a chovanie funkcie je teda nasledujúce:

  1. skontrolujeme, že prvok nepatrí do požadovaného intervalu
  2. na štandardný chybový výstup (`stderr`) vypíšeme hlášku, že sa jedná o nesprávny index
  3. vrátime hodnotu `-1` indikujúcu, že nastala chyba (index siaha mimo pole)

- index prvku je vhodný – to vieme zaručiť ak index prvku patrí do intervalu $\langle 0, n)$, kde n je veľkosť poľa a chovanie funkcie je teda nasledujúce:

  1. skontrolujeme, že prvok patrí do požadovaného intervalu
  2. inicializujeme premennú `id` na hodnotu rodiča prvku na indexe `elementIndex`
  3. hodnotu identifikátoru meníme na rodiča prvku na indexe `id` až do doby než narazíme na slučku (rodič má rovnaký index ako je aktuálna hodnota premennej `id`)
  4. vrátime hodnotu uloženú v premennej `id`, ktorá obsahuje identifikátor množiny obsahujúcej prvok na indexe `elementIndex`

3.1.5 Rozdiely vo funkcií find naprieč implementáciami

V implementáciach zjednotenie podľa rádu a zjednotenie podľa veľkosti je implementácia funkcie `find` identická s tou základnou. Pre ostatné implementácie nastávajú zmeny v rámci stromu množiny obsahujucej prvok na indexe `elementIndex`.

Implementácie kompresia cesty, pólenie cesty a delenie cesty menia dĺžku cesty z vrcholu na indexe do koreňa.

Implementácia kompresia cesty pri vykonaní operácie `find` skráti cestu v rámci stromu množiny tak, že všetky prvky na ceste z prvku do koreňa pripojí priamo pod koreň. Toto docielime pridaním cyklu vykonávajúceho kompresiu po zistení identifikátoru koreňa.

Tento cyklus znova prejde celú cestu z prvku až do koreňa a pre každý prvok nastaví jeho rodiča na identifikátor koreňa (`id`). Tento cyklus sa nachádza na riadkoch 9 až 13 v ukážke 3.3.

```
int find ( int elementIndex, UnionFind * set ) {
    if ( elementIndex >= 0 && elementIndex < set -> size ) {
        int id = set -> parents [ elementIndex ];

        while ( id != set -> parents [ id ] ) {
            id = set -> parents [ id ];
        }

        while ( set -> parents [ elementIndex ] != id ) {
            int tmp = set -> parents [ elementIndex ];
            set -> parents [ elementIndex ] = id;
            elementIndex = tmp;
        }
        return id;
    }
    else {
        fprintf ( stderr, "Invalid element index!\n" );
        return -1;
    }
}
```

Ukážka 3.3: Funkcia `find` pri implementácií kompresie cesty

Implementácia delenie cesty upravuje funkciu `find` tak, že pri priechode stromu cestou do koreňa každému prvku zmení hodnotu ich rodiča na rodiča ich rodiča tieto zmeny nastávajú už pri prvotnom priechode cestou z vrcholu do koreňa.

```
int find ( int elementIndex, UnionFind * set ) {
    if ( elementIndex >= 0 && elementIndex < set -> size ) {
        int id = elementIndex;

        while ( id != set -> parents [ id ] ) {
            int tmp = id;
            id = set -> parents [ id ];
            set -> parents [ tmp ] = set -> parents [ id ];
        }
        return id;
    }
    else {
        fprintf ( stderr, "Invalid element index!\n" );
        return -1;
    }
}
```

Ukážka 3.4: Funkcia `find` pri implementácií delenia cesty

Poslednou implementáciou meniacou funkciu `find` je implementácia pólenie cesty, ktorá vykonáva to isté ako implementácia delenie cesty len pre každý druhý prvok viď ukážka 3.5.

```
int find ( int elementIndex, UnionFind * set ) {
    if ( elementIndex >= 0 && elementIndex < set -> size ) {
        int id = elementIndex;

        while ( ( id ) != set -> parents [ id ] ) {
            set -> parents [ id ] =
             set -> parents [ set -> parents [ id ] ];
            ( id ) = set -> parents [ id ];
        }
        return id
    }
    else {
        fprintf ( stderr, "Invalid element index!\n" );
        return -1;
    }
}
```

Ukážka 3.5: Funkcia `find` pri implementácií pólenia cesty

3.1.6 Základná implementácia funkcie union

Funkcia `union` predstavuje implementáciu operácie `union`, ktorá vykoná zjednotenie množín obsahujúcich prvky nachádzajúce sa na indexoch `elementIndex1` a `elementIndex2`. Táto funkcia vytvára z dvoch stromov dvoch množín jeden strom pomocou náhrady slučky jednoho stromu za hranu vedúcu zo stromu jednej množiny do množiny druhej.

Všeobecne funkcia `union` rieši tri rôzne situácie:

- aspoň jeden z indexov prvkov určených pre zjednotenie je nevhodný – to nastáva v moment keď aspoň jeden z indexov prvkov pre zjednotenie nepatrí do intervalu $\langle 0, n)$, kde n je veľkosť poľa a v tento moment sa funkcia chová nasledovne:
  1. vykonáme kontrolu, či aspoň jeden z indexov je nevhodný – to vykonáme pomocou kontroly, či pre niektorý z prvkov funkcie `find` vráti hodnotu `-1`
  2. skontrolujeme, ktorý z indexov nie je vhodný – pre každý nevhodný index vypíšeme chybovú hlášku na štandardný chybový výstup (`stderr`)
  3. vrátíme hodnotu `false`, ktorá signalizuje, že nastala chyba pri zjednotení
- normálne zjednotenie – chceme zjednotiť dva prvky, ktoré patria do rôznych množín, to vykonáme nasledovne:
  1. zistíme identifikátor množiny obsahujucej prvok na prvom indexe (to isté vykonáme aj pre prvok na druhom indexe)

2. skontrolujeme, že sa nejedená o rovnaký identifikátor
3. nahradíme slučku v koreni druhej množine hranou z koreňa druhej množiny do vrcholu s prvkom na indexe `elementIndex1`
4. vrátime hodnotu `true` označujúcu, že operácia prebehla bez chyby

- prvky na indexoch `elementIndex1` a `elementIndex2` patria do rovnakej množiny – táto situácia je ošetrená pomocou porovnania identifikátorov množín
  - táto situácia prebieha rovnako ako normálne zjednotenie, jediný rozdiel je ten, že nevytvárame hranu medzi stromami množín, lebo prvky už do rovnakej množiny patria (vytvorením hrany by sme mohli vytvoriť cyklus)
  - návratovú hodnotu ponechávame `true` nakoľko podľa očakávaní prvky na týchto indexoch budú po vykonaní funkcie patriť do rovnakej množiny

```
1  bool unionSet ( int elementIndex1, int elementIndex2,
2                  UnionFind * set ) {
3      int firstParent = find ( elementIndex1, set );
4      int secondParent = find ( elementIndex2, set );
5
6      if ( firstParent == -1 || secondParent == -1 ) {
7          if ( firstParent == -1 ) {
8              fprintf ( stderr,
9               "Invalid index for first element!\n" );
10         }
11         if ( secondParent == -1 ) {
12             fprintf ( stderr,
13              "Invalid index for second element!\n" );
14         }
15         return false;
16     }
17
18     if ( firstParent == secondParent ) {
19         return true;
20     }
21     set -> parents [ secondParent ] = elementIndex1;
22     return true;
23 }
```

Ukážka 3.6: Základná implementácia funkcie `unionSet`

3.1.7 Rozdiely vo funkcií union naprieč implementáciami

Funkcia `union` je pre implementácie kompresia cesty, pólenie cesty a delenie cesty identická ako pre základnú implementáciu.

Rozdiel je v implementáciach zjednotenie podľa rádu a zjednotenie podľa veľkosti, no tieto zmeny sú minimálne. V týchto implementáciach máme naviac polia udržujúce rád, respektive veľkosť stromu každej množiny, ktoré sa používajú práve v tejto funkcií pre určenie orientácie hrany naprieč stromami množín určených pre zjednotenie.

V prípade implementácie zjednotenie podľa rádu vytvárame hranu z koreňa stromu nižšieho rádu do koreňa stromu rádu vyššieho (ak je rád identický na orientácií hrany nezáleží). Kvôli tomu môže byť potreba zmena poradia operandov, realizáciu tejto zmeny možno vidieť na riadkoch 21 až 24 ukážky 3.7. Zároveň ak sú rády stromov identické je potreba o jedna zvýšiť rád stromu reprezentujúceho zjednotené množiny realizáciu tejto zmeny možno vidieť na riadkoch 27 až 30 ukážky 3.7.

```
1  bool unionSet ( int elementIndex1, int elementIndex2,
2                  UnionFind * set ) {
3      int firstParent = find ( elementIndex1, set );
4      int secondParent = find ( elementIndex2, set );
5      if ( firstParent == -1 || secondParent == -1 ) {
6          if ( firstParent == -1 ) {
7              fprintf ( stderr,
8               "Invalid index for first element!\n" );
9          }
10         if ( secondParent == -1 ) {
11             fprintf ( stderr,
12              "Invalid index for second element!\n" );
13         }
14         return false;
15     }
16
17     if ( firstParent == secondParent ) {
18         return true;
19     }
20
21     if ( set -> ranks [ firstParent ] >
22            set -> ranks [ secondParent ] ) {
23         swap ( & firstParent, & secondParent );
24     }
25
26     set -> parents [ firstParent ] = secondParent;
27     if ( set -> ranks [ firstParent ] ==
28            set -> ranks [ secondParent ] ) {
29         set -> ranks [ secondParent ] ++;
30     }
31     return true;
32 }
```

Ukážka 3.7: Funkcia `unionSet` pri implementácií zjednotenie podľa rádu

V prípade implementácie zjednotenie podľa veľkosti je funkcia `union` obdobná ako v implementácií zjednotenie podľa rádu. Vytvárame hranu z koreňa

stromu obsahujúceho menší počet vrcholov do koreňa stromu s väčším počtom vrcholov (ak je počet vrcholov identický na orientácií hrany nezáleží). Kvôli tomu môže byť potreba zmena poradia operandov, jej realizáciu možno vidieť na riadkoch 21 až 24 ukážky 3.8. Zároveň pri zjednotení vznikne strom, ktorého počet vrcholov je súčtom počtu vrcholov jednotlivých stromov realizáciu tejto zmeny možno vidieť na riadkoch 27 a 28 ukážky 3.8.

```
1  bool unionSet ( int elementIndex1, int elementIndex2,
2                  UnionFind * set ) {
3      int firstParent = find ( elementIndex1, set );
4      int secondParent = find ( elementIndex2, set );
5      if ( firstParent == -1 || secondParent == -1 ) {
6          if ( firstParent == -1 ) {
7              fprintf ( stderr,
8               "Invalid index for first element!\n" );
9          }
10         if ( secondParent == -1 ) {
11             fprintf ( stderr,
12              "Invalid index for second element!\n" );
13         }
14         return false;
15     }
16
17     if ( firstParent == secondParent ) {
18             return true;
19     }
20
21     if ( set -> sizes [ firstParent ] >
22           set -> sizes [ secondParent ] ) {
23         swap ( & firstParent, & secondParent );
24     }
25
26     set -> parents [ firstParent ] = secondParent;
27     set -> sizes [ secondParent ] +=
28      set -> sizes [ firstParent ];
29     return true;
30 }
```

Ukážka 3.8: Funkcia `unionSet` pri implementácií zjednotenie podľa veľkosti

3.2 Verifikácia

V tejto kapitoly si zhrnieme aké požiadavky máme na jednotlivé funkcie a v prípade tých komplikovanejších si ukážeme ako boli vyjadrené pomocou predikátov a logických funkcií v rámci jazyka *ACSL*.

V kontraktoch funkcií `makeSet`, `find` a `unionSet` požadujeme pred aj po vykonaní funkcie splnenie požiadavkov `\freeable_set`, `\valid_parts`, `\is_acyclic`, prípadne `\valid_sizes` pri implementácií zjednotenie podľa veľkosti, alebo `\valid_ranks` pri implementácií zjednotenie podľa rádu. Tieto

požiadavky vo forme predikátov si vysvetlíme v samostatných podkapitolách a potom sa pozrieme aké požiadavky sme mali pre jednotlivé funkcie, no popíšeme si ich len slovným popisom, nie ukážkami kontraktov z dôvodu ich rozsahu. Celé kontrakty funkcií možno nájsť v prílohe tejto práce.

3.2.1 Predikát \freeable_set

Predikát `\freeable_set` predstavuje logický výraz, ktorý kontroluje, že nemôže nastať situácia kedy by ukazatel na štruktúru `UnionFind` bol správny, ale pritom niektoré z polí, ktoré obsahuje štruktúra ako členskú premennú nebolo alokované.

3.2.2 Predikát \valid_parts

Predikát `\valid_parts` predstavuje logický výraz, ktorý kontroluje, že nemôže nastať situácia kedy by ukazatel na štruktúru `UnionFind` bol správny, ale niektorá z členských premenných by neobsahovala vhodnú hodnotu alebo by sme nemohli do niektorého z polí zapísať.

3.2.3 Logická funkcie `find`

Logická funkcia `find` funguje podobne ako funkcia `find` pre našu dátovú štruktúru (ďalej len funkcia `find`) – získava identifikátor množiny obsahujucej prvok na zadanom indexe. Avšak logická funkcia `find` a funkcia `find` majú dva zásadné rozdiely a to, že logická funkcia je implementovaná rekurzívne a detekuje, ak objaví cyklus dĺžky aspoň 2 (v ten moment vracia hodnotu -1). Dôvodom implementácie pomocou rekurzie je ten, že logické funkcie nepodporujú premenné.

3.2.4 Predikát \is_acyclic

Predikát `\is_acyclic` predstavuje logický výraz, ktorý kontroluje, že pre štruktúru, na ktorú ukazatel ukazuje je vhodná a neobsahuje cyklus s výnimkou slučky – to sme docieli detekciou cyklu pomocou logickej funkcie `find`.

3.2.5 Predikát \valid_ranks

Predikát `\valid_ranks` predstavuje logický výraz, ktorý kontroluje, že pre pole `ranks`, ktoré je členskou premennou štruktúry `UnionFind` pri implementácií zjednotenia podľa rádu platí usporiadanie rádov stromov – teda, že strom reprezentujúci množinu má nižší rád ako strom rodiča, jedinou výnimkou je koreň, ktorého rodič je on sám teda rád stromu rodiča má rovnaký rád ako strom pôvodnej množiny.

3.2.6 Predikát \valid_sizes

Predikát `\valid_sizes` je podobný predikátu `\valid_ranks`, no je relevantný len v implementácií zjednotenie podľa veľkosti, nie zjednotenie podľa rádu. Jediný rozdiel medzi týmito predikátmi je ten, že predikát `\valid_sizes` kontroluje hodnoty nachádzajúce sa v poli `sizes`, nie `ranks`. Pre túto implementáciu platí, že počet vrcholov stromu reprezentujúceho množinu je nižší ako počet vrcholov stromu rodiča (ktorý je nadmnožinou). Jedinou výnimkou je keď narazíme na slučku – strom rodiča je v ten moment identický ako strom pôvodný a preto je ich veľkosť tiež zhodná.

3.2.7 Predikát \correctly_unioned

Predikát `\correctly_unioned` predstavuje logický výraz testujúci, že nová množina obsahuje iba prvky množín, do ktorých patrili prvky na indexoch `element1` a `element2` a zároveň, že ostatné prvky patria do rovnakej množiny ako pred vykonaním zjednotenia.

3.2.8 Overenie funkcie makeSet

Požiadavky na túto funkciu sa líšia na základe rôznych situácií, napríklad v niektorých implementáciach je vykonaná alokácia, v niektorých nie, konkrétne:

- dátová štruktúra ešte neexistuje
  - ukazateľ na dátovú štruktúru musí byť `NULL`
  - dynamicky sa alokuje dátová štruktúra a jej polia `parents` a `elements`, prípadne `sizes`, respektíve `ranks`
  - zapisuje sa do ukazatela `set`
  - žiadna pamäť sa neuvoľňuje
  - návratová hodnota je 0 (vkladamé na index 0), v rámci poľa `parents` bude posledný prvok novovložený prvok, bola vytvorená slučka, prípadne pre prvok boli správne inicializované hodnoty v rámci poľa `sizes`, respektive `ranks`
- pridávame prvok bez potreby zväčšenia kapacít polí
  - dátová štruktúra musí existovať (byť alokovaná), mať dostatočnú kapacitu a nesmie obsahovať vkladaný prvok
  - žiadna pamäť sa nealokuje, ani neuvoľňuje
  - zapisuje sa na posledný neobsadený index všetkých polí, ktoré sú súčasťou štruktúry a do členskej premennej `size`

  - návratová hodnota je index novovloženého prvku, v rámci poľa `parents` bude posledný prvok novovložený prvok, bola vytvorená slučka, prípadne pre prvok boli správne inicializované hodnoty v rámci poľa `sizes`, respektíve `ranks`

- pridávame prvok v dôsledku čoho musíme zväčšiť polia
  - dátová štruktúra musí existovať (byť alokovaná), nemať dostatočnú kapacitu a nesmie obsahovať vkladaný prvok
  - všetky dynamicky alokované polia sa v rámci zväčšenia alokujú a pravdepodobne aj uvoľnia
  - zapisuje sa na všetky indexy polí, ktoré sú súčasťou štruktúry a do členských premenných `size` a `capacity`
  - návratová hodnota je index novovloženého prvku, v rámci poľa `parents` bude posledný prvok novovložený prvok, bola vytvorená slučka, prípadne pre prvok boli správne inicializované hodnoty v rámci poľa `sizes`, respektíve `ranks`

- pokus o pridanie prvku, ktorý sa už v dátovej štruktúre nachádza
  - dátová štruktúra musí existovať (byť alokovaná) a musí už obsahovať požadovaný prvok
  - žiadna pamäť sa nealokuje, ani neuvoľňuje
  - žiadna časť dátovej štruktúry sa nemení
  - návratová hodnota je -1 a indikuje, že sa nepodarilo vkladanie prvku

3.2.9 Overenie funkcie find

Funkcia `find` pozostáva z dvoch situácií a to keď sa jedná o vhodný index, alebo nie, požiadavky v rámci týchto situácií sú nasledujúce:

- index prvku je validný
  - index patrí do rozsahu $\langle 0, n)$, kde n je hodnota členskej premennej `size`
  - nebola vykonaná žiadna alokácia, ani uvoľnenie pamäti
  - zmeny nastávajú v hodnote, na ktorú ukazuje ukazatel
  - žiadna časť dátovej štruktúry sa nemení
  - návratová hodnota je `true` a indikuje, že nájdenie identifikátoru množiny prebehlo úspešne, identifikátor je koreňom nejakého stromu (vrchol obsahuje slučku), identifikátor je hodnota patriaca do rozsahu $\langle 0, n)$

- index prvku nie je validný
  - index nepatrí do rozsahu $\langle 0, n)$, kde n je hodnota členskej premennej `size`
  - žiadne premenné nie su v tejto situácií alokované, ani uvoľnené
  - žiadna časť dátovej štruktúry sa nemení
  - návratovou hodnotou je `false` a indikuje, že nájdenie identifikátoru nebolo úspešné

3.2.10 Overenie funkcie union

Vo funkcií `union` riešime dve situácie rovnako ako v prípade funkcie `find` a to, keď máme vhodné indexy a keď aspoň jeden z indexov nie je vhodný.

Požiadavky sú v týchto situáciách nasledovné:

- oba indexy sú vhodné
  - oba indexy patria do intervalu $\langle 0, n)$, kde n je hodnota členskej premennej `size`
  - žiadne prostriedky nie sú alokované, ani uvoľnené
  - v rámci dátovej štruktúry sa zmení nejaká z hodnôt polí `parents`, prípadne `ranks`, respektive `sizes`
  - návratová hodnota je `true`, ktorá indikuje, že prvky na indexe patria do rovnakej množiny a skontolujeme, že nové množiny vyzerajú podľa očakávaní pomocou predikátu `correctly_unioned`
- aspoň jeden z indexov nie je vhodný
  - aspoň jeden z indexov nepatrí do intervalu $\langle 0, n)$, kde n je hodnota členskej premennej `size`
  - žiadna pamäť nebola alokovaná, ani uvoľnená
  - žiadna časť dátovej štrúktury sa nemení
  - návratová hodnota je `false` a indikuje, že pri zjednotení nastala chyba

3.2.11 Výsledky verifikácie

Výsledky verifikácie našich implementácií nedopadli úplne optimálne. Dôvodom nie zcela uspokojivých výsledkov bola obmedzenosť a nedostatočná implementácia niektorých funkcionalít frameworku *Frama-C*.

Najväčším problémom v rámci frameworku sú logické funkcie, u ktorých je problém ten, že nie je v nich možné tvoriť cykly a ani premenné. Kvôli týmto obmedzeniam je nutné v prípadne zložitejších funkcií využiť rekurziu, čo by

vo všeobecnosti nebol taký problém, no v prípadne tohto frameworku rekurzia tvorí docela veľký problém nakoľko tu rekurzia nie je stabilná – výsledky verifikácie predikátov využívajúcich logické funkcie obsahujúce rekurziu nie sú spoľahlivé (framework sa pokúsi výraz overiť, ale bez výsledku, alebo nastanú chyby v predpokladoch) [13].

Ďalším problémom frameworku sú kľúčové slová `allocates`, `assigns` a `frees`, ktorých funkčnosť je diskutabilná.

Klauzula `allocates` nie je vôbec overovaná – po vykonaní overenia dostaneme výsledok „Never tried: no status is available for this property", čo znamená, že výraz nikdy nebol otestovaný a neexistuje preň žiadny výsledok (tento stav sa označuje modrou kružnicou).

V prípade klauzulí `assigns` a `frees` je výsledok verifikácie hláška „Unknown: a verification has been attempted, but without conclusion", čo znamená, že overenie prebehlo, ale z nejaké dôvodu bez výsledku (označuje sa žltým kruhom), prípadne „Valid under hypothesis: verified (but has dependencies with Unknown status)", čo znamená, že v rámci overovania nastala nejaká chyba, ktorá môže byť rôzneho charakteru (označuje sa zeleno-oranžovým kruhom), v niektorých prípadoch tieto problémy spôsobuje rekurzia.

Ďalšou zaujímavou problematikou tohto frameworku v prípade kľúčových slov `allocates` a `frees` je, že by sa malo jednať o pamäťové bunky, ktoré budú v rámci funkcie alokované, respektíve uvoľnené a budú iného typu ako `void *`, čo ale v rámci jazyka C nie je možné nakoľko jazyk C používa v rámci funkcií interne *syscally*, ktoré alokujú práve `void *`. Pre túto nejasnosť sme použili tieto kľúčové slovo v prípade ľubovoľnej dynamickej alokácie, keď boli dáta typu `void *` alokované a následne interpretované iným spôsobom (zmenou dátového typu pomocou pretypovania).

```
/*@ requires \freeable_set(set);
    requires \valid_parts(set);
    requires \is_acyclic(set);

    behavior valid:
      assumes 0 ≤ elementIndex < set->size;
      ensures 0 ≤ *\old(setID) < \old(set)->size;
      ensures *\old(setID) ≡ find(\old(set), \old(elementIndex), 0);
      ensures *(\old(set)->parents + *\old(setID)) ≡ *\old(setID);
      ensures (\result ≠ 0) ≡ \true;
      ensures \freeable_set(\old(set));
      ensures \valid_parts(\old(set));
      ensures \is_acyclic(\old(set));
      assigns *setID;
      allocates \nothing;
  */
_Bool find(int elementIndex, UnionFind *set, int *setID)
```

Ukážka 3.9: Časť výsledkov verifikácie základnej implementácie funkcie `find`

Kapitola 4

Analýza a benchmark

V tejto kapitole si popíšeme analýzu naších implementácií pomocou *symbolic execution*, v krátkosti si popíšeme čo je to *symbolic execution engine Klee*, ktorý vykonáva práve *symbolic execution* na programoch napísaných v jazyku C a tiež si zhrnieme výsledky tejto analýzy.

Mimo toho si v tejto kapitole zhrnieme vykonaný benchmark – aké nástroje sme použili, akým spôsobom sme tvorili testy a ako dopadli jednotlivé implementácie vo vzájomnom porovnaní rýchlosti.

4.1 Analýza pomocou symbolic execution

Symbolic execution analyzuje program tak, že určí aké hodnoty musia premenné nadobúdať, aby boli vykonané všetky časti kódu, do ktorého premenné zasahujú [17]. Analýza pozostáva z tvorby takzvaných symbolických premenných a následného spustenia nástroja vykonávajúceho túto analýzu. Tento nástroj nepovažuje tieto symbolické premenné za bežné premenné majúce nejakú hodnotu, ale vždy keď narazí na podmienku, ktorú by mohla ovplyvniť jedna zo symbolických premenných vykoná paralelizáciu a nastaví v jednotlivých vetvách hodnoty premenných tak, aby spôsobil vykonanie všetkých možných priechodov daným vetvením.

Problémom takejto analýzy je, že vykonáva všetky prechody funkciami, to však môže byť časovo náročné v prípade veľkého množstva vetvení a tiež v prípade zložitých podmienok. No ak užívateľ vytvorí test rozumného rozsahu je možné pomocou tejto analýzy odhaliť chyby poškodzujúce pamäť (memory corruption) [18].

4.1.1 Symbolic execution engine – Klee

Symbolic execution predstavuje pojem zastrešujúci spôsob analýzy uvedený v predošlej kapitole. Pre konkrétne vykonanie tejto analýzy sa však používa takzvaný *symbolic execution engine*, ktorý musí byť implementovaný pre

konkrétny programovací jazyk. V prípade jazyka C je možné použiť napríklad *symbolic execution engine Klee*.

Klee nie je *symbolic execution engine* určený výlučne pre jazyk C, ale je určený pre *LLVM bytecode*, ktorý možno zo zdrojových kódov napísaných v jazyku C vygenerovať pomocou Clang kompilátoru, ktorý je vyvíjany ako súčasť projektu *LLVM*.

Pre prácu s Klee je možné využiť Docker kontajner priamo od vývojárov *Klee*.[18] V tomto kontajneri je pripravené *Klee* a kompilátor Clang, ktoré potrebujeme pre prácu. Pre základnú prácu s týmto *symbolic execution engine* v jazyku C je potreba vedieť vytvoriť symbolické premenné, pre tieto potreby možno použiť funkciu `klee_make_symbolic(...)` z knižnice `klee/klee.h`. Jedná sa o jedinú funkciu, ktorú sme v rámci našej práce použili. Po vytvorení testu pre *symbolic execution engine* je potreba kód preložiť pomocou kompilátoru Clang do *LLVM byte code*, na ktorom možno následne spustiť *Klee*.

Klee spustením vygeneruje priečinok `klee-out-N`, kde `N` je poradové číslo spustenia. V tomto priečinku možno nájsť všetky testy, ktoré *Klee* vygenerovalo. V týchto súboroch možno nájsť aké hodnoty mali symbolické premenné v rámci testu (tieto hodnoty možno zobraziť pomocou nástroja `ktest-tool` na súbor testu). Ak test spôsobí chybu, ktorú *Klee* detekuje bude naviac vygenerovaný súbor `testN.TYPE.err`, kde `N` je číslo testu a `TYPE` identifikuje typ detekovanej chyby. [19]

V rámci testovania našej implementácie dátovej štruktúry Union-Find sme sa rozhodli pre každú implementáciu vytvoriť štyri testy. Tri, kde každý volá práve jednu z funkcií na základe symbolickej premennej a jeden, ktorý volá jeden raz funkciu `makeSet` a `unionSet` a dvakrát funkciu `find`. Všetky tieto testy sme postupne spustili pre jednotlivé implementácie, výsledky boli pozitívne – pre žiadnu z implementácií nebola nájdená žiadna chyba.

Zaujímavá situácia, ktorá behom použitia tohto engine nastala je, že test, ktorý sme pôvodne vytvorili pre volanie všetkých troch funkcií, obsahoval príliš veľa symbolických premenných a volaní funkcií. Z dôvodu, že naše funkcie obsahujú zložené podmienky počet prechodov našim kódom narástol na príliš veľké množstvo, čo znamenalo veľké množstvo testov a teda aj dlhá doba trvania analýzy. V našom prípade sme nechali *symbolic execution engine* bežať sedem dní a následne jeho beh ukončili. Po hlbšom zamyslení nad testom a funkciami sme zistili, že kvôli zložitejším podmienkam a väčšiemu množstvu vetvení by analýza v dohľadnej dobe pravdepodobne nedobehla. Z tohto dôvodu sme sa rozhodli testy zmenšiť.

4.2 Benchmark

Benchmark je proces spúšťania počítačového programu, časti programu alebo nejakých operácií s cieľom zistenia relatívnej výkonnosti v porovnaní s iným

[18] https://klee.github.io/docker/

subjektom testu [20]. Jeho cieľom je väčšinou iba zistenie výkonnosti a nehľadí na správnu funkčnosť.

Nakoľko sme potrebovali vykonať benchmark niekoľkých implementácií na rovnakých testovacích sadách využili sme modularity programovacieho jazyku C a za pomoci direktív preprocesoru vytvorili súbor, ktorý na základe argumentov kompilátoru vloží implementáciu, ktorú chceme testovať.

Keďže najpodstatnejšou informáciou v prípade testovania rýchlosti programov je čas, vyžadujeme vysokú presnosť. Z tohto dôvodu sme využili nástroj `perf stat`[19], ktorý poskytuje detailne štatistiky o behu programu.

4.2.1 Tvorba testov

V rámci benchmarku je veľmi podstatná testovacia sada, na ktorej je benchmark vykonaný. Pre potrebu tvorby testovacej sady je možné využiť niekoľko prístupov:

- ručné písanie testov – tento prístup by bol v pre tvorbu testovacích sád, ktoré by bežali aspoň desiatky sekúnd veľmi pracné, no aj napriek tomu sme pár testov vytvorili aj týmto spôsobom (konkrétne sa jednalo o test potencionálne problematických hodnôt a o test na malej instancií)

- generovanie testov za behu programu pomocou generátoru na princípe fuzzeru[20] – tento prístup by bol problematický z toho dôvodu, že čas behu testovacej sady by bol závislí na operáciách vykonaným pre generovanie testov

- externé generovanie testov – jedná sa o cestu, ktorou sme sa vydali my v rámci tejto práce, vytvorili sme si dva generátory testov (generátor na štýl fuzzeru a generátor, ktorý vygeneruje k volaní funkcie `union` a l volaní funkcie `find`)[21]

Pre generovanie sme teda vytvorili dva skripty napísané v jazyku Python, ktoré do súborov generujú testovacie sady, ktoré je možné následne vložiť do zdrojových kódov určených pre testovanie. Tieto skripty rovnako ako predpripravené zdrojové kódy pre benchmark možno nájsť v prílohe tejto práce.

4.2.2 Výsledky benchmarku

Pre benchmark sme vygenerovali testovacie sady pomocou našich skriptov. Jedna testovacia sada bola vygenerovaná pomocou skriptu `generator.py`,

[19] https://www.man7.org/linux/man-pages/man1/perf-stat.1.html

[20] fuzzer je program náhodne generujúci dáta pre testovaný program

[21] hlavný rozdiel medzi naším fuzzerom a generátorom je ten, že generátor generuje najprv volania funkcie `union` až potom funkcie `find` zatiaľ, čo fuzzer vykonáva volania týchto funkcií v náhodnom poradí

tá pozostáva z množiny o velikosti stotisíc prvkov a na tejto množine bola tristotisíckrát funkcia `unionSet` a dvestotisíckrát funkcia `find`. Druhá testovacia sada bola generovaná pomocou skriptu `fuzzer.py` a pozostáva z množiny o štyristotisíc prvkoch a približne dvestodvadsaťpäť tisíc volaní funkcie `unionSet` a približne sedemdesiatpäť tisíc volaní funkcie `find`.

Každú z našich implementácií sme spustili na test obsahujúci obe generované sady a merali ich dobu behu. Od výsledkov sme očakávali priepastný rozdiel medzi implementáciou základnou a všetkými optimalizovanými implementáciami – to sa potvrdilo.

Ďaľej sme očakávali rýchlostné rozdiely naprieč optimalizovanými implementáciami pohybujúce sa v nízkych jednotkách sekúnd – to sa však nepotvrdilo, rozdiely medzi optimalizovanými verziami boli veľmi malé, pohybovali sa v nízkych desatinách sekúnd.

Benchmark bol spustený na clusteri DGX, ktorý patrí katedre teoretickej informatiky našej fakulty, na tomto stroji bolo pred každým spustením testu vykonanie vyprázdnenie pamäte cache pre zaistenie rovnakých podmienok testu a pre minimalizáciu zásahu externých faktorov do výsledkov benchmarku. Výsledky tohto testu možno vidieť v tabuľke 4.1.

Názov implementácie	Doba behu testu [s]
Pólenie cesty	211,124699350
Zjednotenie podľa veľkosti	211,184645635
Zjednotenie podľa rádu	211,218076311
Delenie cesty	211,257025787
Kompresia cesty	211,283052981
Základná implementácia	305,016617468

Ukážka 4.1: Výsledky benchmarku zoradené vzostupne podľa doby behu

Záver

Cieľom tejto práce bolo naštudovať možností implementácie dátovej štruktúry Union-Find a oboznámiť sa s frameworkom Frama-C, ktorý slúži pre overenie kódov napísaných v jazyku C a následne prepojiť tieto znalosti pre tvorbu jednej z implementácií a jej následnú verifikáciu.

V rámci kapitoli 1 sme sa oboznámili s dátovou štuktúrou Union-Find a možnosťami jej implementácie pre dosiahnutie optimalizácie vybraných operácií. Na záver kapitoly sme sa tiež pozreli na vzájomné porovnanie týchto implementácií. V kapitole 2 sme si vysvetlili základné prostriedky pre verifikáciu kódu pomocou frameworku Frama-C. V kapitole 3 sme si povedali akým spôsobom sme dátovú štruktúru Union-Find a jej operácie naimplementovali a tiež ako sme tieto implementácie verifikovali. Na záver tejto kapitoly sme zhrnuli problémy, na ktoré sme behom práce s frameworkom narazili a limitovali naše možnosti verifikácie, tie zahŕňali nedostatočnú schopnosť kontroly dynamickej alokácie a uvoľňovania pamäte, problémy s rekurzivne definovanými požiadavkami slúžiacimi pre verifikáciu a podobne.

Všetky ciele práce boli splnené a zadanie sme si následne rozšírili o implementáciu a verifikáciu implementácií využívajúcich optimalizáciu operácií dátovej štruktúry, ďalej sme otestovali implementácie pomocou *symbolic execution engine* a vykonanil benchmark našich implementácií. Tieto rozšírenia možno nájsť v kapitolách 3 a 4

Práca by aj napriek tomu mohla byť do budúcnosti rozšírená napríklad vykonaním verifikácie pomocou novej verzie frameworku, ktorá v momentálnej dobe nie je dostupná alebo využitím iných nástrojov alebo frameworkou slúžiacich pre rovnaké účely. V prípade úspešnosti následnej verifikácie tohto kódu by tiež bolo možné na tejto práci stavať a pokúsiť sa o verifikáciu algoritmov využívajúcich túto dátovú štruktúru, napríklad Kruskalova algoritmu.

Literatúra

[1] Hliněný P.: *Základy Teorie Grafů pro (nejen) informatiky*. FI MUNI, první vydání, marec 2010, [cit. 2023-3-07]. Dostupné z: `https://is.muni.cz/do/rect/el/estud/fi/js10/grafy/Grafy-text10.pdf`

[2] Mareš M., Valla T.: *Průvodce labyrintem algoritmů*. CZ.NIC, z. s. p. o., druhé vydání, november 2022, ISBN 978-80-88168-66-9, [cit. 2023-3-07]. Dostupné z: `http://pruvodce.ucw.cz/static/pruvodce.pdf`

[3] Šubelj L.: Algorithms for spanning trees of unweighted networks. [online], máj 2022, doi:10.48550/arXiv.2205.06628, [cit. 2023-03-23]. Dostupné z: `https://arxiv.org/pdf/2205.06628.pdf`

[4] F., D.: Union–Find Algorithm for cycle detection in a graph. [online], máj 2022, [cit. 2023-04-21]. Dostupné z: `https://www.baeldung.com/cs/graph-connected-components`

[5] R., E.: Disjoint Set Union Data Structure. [online], február 2023, [cit. 2023-04-21]. Dostupné z: `https://www.baeldung.com/cs/disjoint-set-union-data-structure`

[6] Baeldung: Detecting Cycles in a Directed Graph. [online], [cit. 2023-04-21]. Dostupné z: `https://www.techiedelight.com/union-find-algorithm-cycle-detection-graph`

[7] TechieDelight: Union–Find Algorithm for cycle detection in a graph. [online], marec 2023, [cit. 2023-04-21]. Dostupné z: `https://www.baeldung.com/cs/detecting-cycles-in-directed-graph`

[8] Fischer M. J., Galler B.: An improved equivalence algorithm. *Communications of the ACM*, máj 1964, doi:10.1145/364099.364331, [cit. 2022-12-28]. Dostupné z: `https://dl.acm.org/doi/pdf/10.1145/364099.364331`

[9] Tarjan R. E., Leeuwen J. v.: Worst-case Analysis of Set Union Algorithms. *Journal of the Assoaatton for Computmg Machinery*, apríl 1984, doi: 10.1145/62.2160, [cit. 2022-12-28]. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/62.2160

[10] Correnson L. a kol.: *Frama-C User Manual*. 26 vydání, [cit. 2023-01-15]. Dostupné z: https://frama-c.com/download/frama-c-user-manual.pdf

[11] CEA LIST and INRIA: *ACSL: ANSI/ISO C Specification Language*. První vydání, [cit. 2023-01-15]. Dostupné z: https://frama-c.com/download/acsl.pdf

[12] Švejdar V.: LOGIKA: neúplnost, složitost a nutnost. [online], marec 2002, [cit. 2023-04-18]. Dostupné z: https://www1.cuni.cz/~svejdar/book/LogikaSve2002.pdf

[13] Blanchard A.: *Introduction to C program proof with Frama-C and its WP plugin*. Zeste de Savoir, júl 2020, [cit. 2023-02-25]. Dostupné z: https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

[14] LIST, C.: WP. [online], [cit. 2023-02-25]. Dostupné z: https://frama-c.com/fc-plugins/wp.html

[15] CEA LIST: RTE. [online], [cit. 2023-02-25]. Dostupné z: https://frama-c.com/fc-plugins/rte.html

[16] CEA LIST: Eva, an Evolved Value Analysis. [online], [cit. 2023-02-25]. Dostupné z: https://frama-c.com/fc-plugins/eva.html

[17] Anand S., Godefroid P., Tillmann N.: Demand-Driven Compositional Symbolic Execution. [online], 2008, doi:10.1007/978-3-540-78800-3_28, [cit. 2023-03-29]. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-540-78800-3_28

[18] Baradaran S. a kol.: A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes. *IET Research Journals*, december 2022, ISSN 1751-8644, doi:10.48550/arXiv.2210.04258, [cit. 2023-03-23]. Dostupné z: https://arxiv.org/abs/2210.04258

[19] The KLEE Team: Testing a Simple Regular Expression Library. [cit. 2023-03-23]. Dostupné z: https://klee.github.io/tutorials/testing-regex/

[20] Fleming Philip J., Wallace, John J.: How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, marec 1986, doi:10.1145/5666, [cit. 2023-03-23]. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/5666.5673

Dodatok **A**

Obsah priloženého média

- `README.md` ..... stručný popis obsahu média
- `src` ..... zdrojové kódy pre verifikáciu, benchmark a Klee
  - `testing` ..... zdrojové kódy pre benchmark a Klee
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF
- `VM` ..... export virtuálneho stroja s predpripravenou Frama-C a Klee