Univerzita Komenského, Bratislava
Fakulta matematiky, fyziky a informatiky

Vizuálna podpora dokazovania správnosti programov pomocou Floydovej metódy

Bakalárska práca

2014

Martin Filek

Univerzita Komenského, Bratislava
Fakulta matematiky, fyziky a informatiky

Vizuálna podpora dokazovania správnosti programov pomocou Floydovej metódy

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Katedra: Katedra informatiky
Vedúci: RNDr. Jana Katreniaková PhD.

Bratislava, 2014

Martin Filek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Martin Filek
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	slovenský

Názov: Vizuálna podpora dokazovania správnosti programov pomocou Floydovej metódy

Cieľ: Floydova metóda dokazovania správnosti redukuje dôkaz čiastočnej správnosti programu na dokazovanie predikátových formúl. Cieľom bakalárskej práce je vytvoriť program na vizuálnu podporu dokazovania správnosti programov pomocou Floydovej metódy - tvorbu a kontrolu invariantov a postup tvorby predikátových formúl vychádzajúc z používateľom navrhnutých invariantov.

Kľúčové slová: čiastočná správnosť programov, Floydova metóda, vizualizácia

Vedúci:	RNDr. Jana Katreniaková, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	28.10.2013

Dátum schválenia: 28.10.2013 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Ďakujem vedúcej bakalárskej práce RNDr. Jane Katreniakovej PhD. za ponúknutie vhodnej témy, trpezlivosť pri konzultáciách a nápomocné rady.

Abstrakt

Hlavným cieľom bakalárskej práce bolo vytvoriť aplikáciu na vizualizáciu Floydovej metódy, ktorá sa používa na dokazovanie čiastočnej správnosti programov. Práca poskytuje teoretický základ potrebný pre pochopenie jednotlivých krokov metódy. Obsahuje tiež popis vizualizačných techník a implementačné detaily. Aplikácia má pomáhať študentom pri štúdiu teoretickej informatiky v oblasti teórie programovania.

Kľúčové slová: Floyova metóda, správnosť programov, vizualizácia

Abstract

The main objective of the bachelor's thesis was to create an application for visualising of Floyd's Method, which is used for proving programs' partial correctness. The paper provides a theoretical foundation needed to understand the method's individual steps. It also contains a description of the visualising techniques and implementation details. The application is supposed to help fellow students in their studies of theoretical computer science in the field of programming theory.

Keywords: Floyd's Method, programs' correctness, visualisation

Obsah

Úvod

Dokazovanie správnosti programov je veľmi atraktívna téma teoretickej informatiky. Keby sa po napísaní ľubovoľného programu dalo automaticky overiť, či sme počas vývoja neurobili žiadnu chybu, mali by programátori citeľne zľahčenú prácu. Dnes v praxi používame testovanie. Napísaním testov však vieme iba odhaliť možné chyby, nedokážeme zistiť, či tam už žiadna ďalšia nie je.

Aj keď sú metódy na dokazovanie správnosti programov v praxi nepoužiteľné, je užitočné si ich vyskúšať na jednoduchých príkladoch. Preco? V práci sa budeme zaoberať hlavne Floydovou metódou, ktorá je z najznámejších metód aplikovateľná na najširšie spektrum programov. Ostatné spomenieme len okrajovo.

Informatika ako veda narastá obrovskou rýchlosťou. Informatici musia zápasiť stále s väčším objemom informácií, ktorý sa každý deň zväčšuje. Rozhodol som sa, že spríjemním mladším študentom štúdium teoretickej informatiky tým, že im uľahčím pochopenie Floydovej metódy. Preto hlavným cieľom tejto práce je vytvorenie aplikáciu, ktorá bude vizualizovať kroky Foydovej metódy.

Prácu chceme napísať tak, aby spolu s aplikáciou tvorili dostatočný základ pre čo najľahšie a najrýchlejšie pochopenie Floydovej metódy. Rozdelená bude na štyri kapitoly. Prvá bude mať za úlohu podať študentovi komplexné teoretické vedomosti o tejto metóde, ako aj o správnosti programov všeobecne. Druhá kapitola bude popisovať aké vizualizačné metódy sme použili na konkrétne kroky, ako aj ako sme aplikáciu navrhli z dizajnérskeho hľadiska. V tretej kapitole sa budeme venovať implementačným detailom. Uvedieme v nej zaujímavosti, s ktorými sme sa stretli počas vývoja aplikácie. Takisto si povieme aj o problémoch a konkrétnych riešeniach, ktoré sme použili. V poslednej kapitole budeme si ukážeme fungovanie aplikácie na konkrétnom príklade.

Pevne verím, že vytvorená aplikácia pomôže množstvu študentom lepšie pochopiť Floydovu metódu a tým aj spríjemní štúdium teoretickej informatiky.

Kapitola 1

Teoretický návod

1.1 Programovací jazyk

V tejto časti si ukážeme programovací jazyk, ktorý budeme používať v príkladoch v ďalších častiach, ako aj v samotnej praktickej časti.

Tento jazyk je veľmi jednoduchý, obsahuje len základné príkazy. Avšak ich kombináciou dokážeme nahradiť väčšinu zložitejších príkazov. Napríklad vhodným použitím podmienky, priradenia a príkazu skoku dokážeme nahradiť for cyklus, ktorý je bežnou súčasťou komplikovanejších jazykov.

Jednoduchosť jazyka nám umožní prehľadnejšie vizualizovať napísaný program.

Ukážka programu $P1$ z [1]

$begin\{[y1, y2] := [x, a]\}$

1. $y_1 := [g(y_1, y_2)]$;
2. $if p_1(y_1)\ then\ goto\ 5$;
3. $[y_1, y2] := [f_1(y_1), f_2(y_2)]$;
4. $goto1$;
5. $[y_2] := [g(y_2, y_1)]$;
6. $if p_2(y_2)\ then\ goto\ end$;
7. $[y_1, y_2] := [g_1(y_1), g_2(y_2)]$;
8. $goto\ 1$;

$end\{[z] := [g_1(y_1)]\}$

1.1.1 Premenné

V programoch budeme používať vstupné premenné tvaru $x, x_1, \cdots, x_{n_1}$, pracovné premenné tvaru $y, y_1, \cdots, y_{n_2}$ a výstupné premenné tvaru $z, z_1, \cdots, z_{n_3}$, pre $n_1, n_2, n_3 \geq 0$.

1.1.2 Príkazy

Kvôli jednoduchosti program podporuje iba 5 typov príkazov:

1. Príkaz priradenia - píšeme ho v tvare premenná := premenná, konštanta alebo výraz zložený z konštánt, premenných a aritmetických operátorov

2. Príkaz podmienky - používame ho v tvare $if(v)\ p$, kde v je výraz zložený z premenných, konštánt a operátorov porovnania, aritmetických operátorov a logických spojok. p je príkaz priradenia alebo skoku

3. Príkaz skoku - zapisujeme ho v tvare *goto* x, kde x je číslo riadku progra mu na ktorý chceme skočiť, teda nasledujúci riadok, ktorý sa vykoná. Alebo x môže byť vyjadrený slovom *end*, kedy sa vykoná posledný riadok programu a ukončí sa.

4. Príkaz *begin* - začiatok vykonávania programu

5. Príkaz *end* - koniec vykonávania programu

1.1.3 Začiatok a koniec programu

Začiatok programu zapisujeme v tvare $begin\{p_1\}$, kde p_1 je priradenie vstupných premenných tvaru $x, x_1, \cdots, x_{n_1}$ do pracovných premenných tvaru $y, y_1, \cdots, y_{n_2}$.

Koniec programu budeme písať v tvare $end\{p_2\}$, kde p_2 je priradenie pracovných premenných tvaru $y, y_1, \cdots, y_{n_2}$ do výstupných premenných tvaru $z, z_1, \cdots, z_{n_3}$.

1.1.4 Program reprezentovaný ako graf

O programe môžeme uvažovať aj ako o orientovanom grafe. Príkazy programu v ňom tvoria množinu vrcholov a platí, že hrana vedie z vrcholu V_1 do vrcholu V_2, ak existuje taká postupnosť príkazov behu programu, že príkaz, ktorý reprezentuje V_2 bude vykonaný hneď po príkaze, ktorý reprezentuje vrchol V_1.

Túto skutočnosť využijeme pri rozdeľovaní programu pomocou deliacich bodov. Umožní nám to vizualizovať program ako vývojový diagram a súčasne môžeme využiť aj algoritmy, ktoré fungujú na orientovaných grafoch.

1.2 Dokazovanie správnosti programov

V tejto časti si povieme o správnosti programov a o Floydovej metóde. Pri písaní tejto časti sme vychádzali hlavne z [2] a [3].

Program budeme charakterizovať pomocou vstupno-výstupných podmienok. Vstupná podmienka nám vymedzuje množinu vstupov, ktoré má daný program vedieť spracovať. Výstupná podmienka určuje, aké vlastnosti má mať výstup programu, vzhľadom na vstupné hodnoty po vykonaní programu.

1.2.1 Vymedzenie pojmu správny program

Označenia

$\overline{x}$ - vektor vstupných premenných, ktorý nadobúda hodnoty podľa vstupnej podmienky. Počas výpočtu sa tieto hodnoty nemenia

$\overline{y}$ - vektor pomocných premenných obsahuje premenné, do ktorých si môžeme uložiť medzivýsledky

$\overline{z}$ - vektor výstupných premenných, v ktorom sú na konci behu programu uložené výsledky

$P(\overline{x})$ - predikát, ktorý vyjadruje vstupnú podmienku

$Q(\overline{x}, \overline{z})$ - predikát, ktorý vyjadruje výstupnú podmienku

Definícia. Program označíme ako "čiastočne správny", ak pre každý vektor $\overline{x}$, ktorý spĺňa predikát $P(\overline{x})$ platí, že v prípade zastavenia programu bude pre vektor $\overline{z}$ splnený aj predikát $Q(\overline{x}, \overline{z})$.

Definícia. Program označíme ako "správny", ak je čiastočne správny a pre každý $\overline{x}$ z $P(\overline{x})$ v konečnom čase zastaví výpočet.

Definícia. Zastavením výpočtu programu sa rozumie stav, kedy program ukončí výpočet príkazom end.

Ďalej si ukážeme metódy, pomocou ktorých môžeme zistiť či je program správny alebo čiastočne správny.

1.2.2 Metódy dokazovania správnosti a čiastočnej správnosti programov

Najznámejšie metódy:

Floydova metóda - pretransformujeme dokazovanie čiastočného zastavenia programu na dokazovanie formúl špecifického jazyka. Touto metódou vieme dokázať široké spektrum programov. Programy môžu byť tvorené aj príkazmi skoku, čo nasledujúca metóda neumožňuje.

Hoerova metóda - metóda využíva špecializovaný logický systém, pomocou ktorého dokazujeme invariantné formuly. Metóda nám umožňuje dokazovanie štrukturovaných programov.

Metóda intermitentov - využívame špecializované formuly - intermitenty, ktoré zaručujú, že sa aspoň raz dostaneme na tie miesta v programe, kde boli umiestnené.

Pomocou prvej a druhej metódy vieme dokázať len čiastočné správnosť, neskôr vznikli ich rozšírenia, pomocou ktorých vieme zistiť, či program zastavil a tým vieme dokázať správnosť v plnom rozsahu. Pomocou metódy intermitentov vieme dokázať úplnú správnosť programu.

1.3 Floydova metóda

Floydova metóda sa skladá z 4 hlavných krokov:

1. Rozdelíme program deliacimi bodmi na konečné cesty.
2. Stanovíme invariant pre každý deliaci bod.
3. Pre každú cestu odvodíme verifikačnú podmienku v špecifickom jazyku, my budeme používať predikátový počet prvého rádu.
4. Každú verifikačnú podmienku overíme, teda dokážeme platnosť prediká tovej formuly.

V ďalších častiach si opíšeme postupne každý krok a ukážeme si konkrétny príklad.

1.3.1 Rozdelenie programu deliacimi bodmi

Pripomíname, že v 1.1.4 sme ukázali, ako sa dá program reprezentovať orientovaným grafom. V tejto časti túto skutočnosť využijeme.

Definícia. Cyklus je súvislý konečný podgraf programu v ktorom platí, že z každého vrcholu vychádza a do každého vrcholu vchádza práve jedna hrana.

Definícia. Cesta je súvislý konečný podgraf grafu, ktorý neobsahuje cyklus.

V každom programe sa vyskytujú minimálne dva deliace body a to začiatok a koniec programu, označme ich A,B. Ak postupnosť príkazov medzi bodmi A a B obsahuje

cyklus, potrebujeme ju rozdeliť ďalším bodom. Ideálne miesto na umiestnenie vnútorného deliaceho bodu je na mieste kde sa z cyklu vychádza. Napríklad v programe $P1$ je to riadok 5, pretože v prípade záporného vyhodnotenia podmienky opustíme cyklus.

Po rozdelení cyklu deliacim bodom, označme si ho C, skontrolujeme vzniknuté cesty medzi bodmi A a C a bodmi C a B na prítomnosť ďalších cyklov. Takýmto spôsobom postupujeme až pokým nám vzniknú konečné cesty, pre ktoré platí, že ani jedna z nich neobsahuje cyklus.

Na konci tohoto kroku musí platiť, že deliace body nám rozdelili program na konečný počet konečných ciest.

1.3.2 Sformulovanie invariantov

Pre začiatočný a koncový bod nám ako invariant slúži vstupná a výstupná podmienka, preto nám stačí sformulovať invarianty pre ostatné (ak sú nejaké) deliace body.

Definícia. Invariant I_A je podmienka viazaná k bodu programu A, vyjadrená ako formula špecifického jazyka, ktorá platí pri každom prechode bodom A.

Existujú rôzne možnosti zadefinovania invariantov, napríklad:

vyjadrenie vzťahu medzi vstupnými a pracovnými premennými $I_A(\overline{x}, \overline{y})$

vyjadrenie vzťahu medzi pracovnými premennými a výstupnými premennými $I_A(\overline{y}, \overline{z})$

Skonštruovanie invariantu je nerozhodnuteľný problém, ale je známych niekoľko heuristík, ktoré môžu dopomôcť k jeho vytvoreniu. Niektoré z nich môžete nájsť v [2]. Podrobnejšie sa tomuto kroku venovať nebudeme, pretože tento krok bude musieť pri vizualizácii urobiť študent manuálne.

1.3.3 Odvodenie verifikačných podmienok

Verifikačné podmienky konštruujeme pomocou spätnej substitúcie.

Označenia

$R_\alpha(\overline{x}, \overline{y})$ popisuje podmienku, ktorá musí platiť po prejdení cesty α.

$r_\alpha(\overline{x}, \overline{y})$ popisuje stav vektorov $\overline{x}, \overline{y}$ po prechode cestou α.

Budeme ním ukazovať zmenu vektora $\overline{y}$ po prejdení cesty α, vektor $\overline{x}$ sa nám nezmení, pretože obsahuje iba vstupné premenné.

Verifikačnú podmienku pre cestu α skonštruujeme pomocou nasledujúcich princípov, ktoré nám ukazujú zmeny stavov $R_\alpha(\overline{x}, \overline{y})$ a $r_\alpha(\overline{x}, \overline{y})$.

Ak za vykonaním príkazu je podmienka a stav premenných v tvare $R_\alpha(\overline{x}, \overline{y})$ a $r_\alpha(\overline{x}, \overline{y})$, potom pred vykonaním príkazu boli v tvare:

Príkaz skoku:

$$R_\alpha(\overline{x}, \overline{y}),\ r_\alpha(\overline{x}, \overline{y}) \text{ (bez zmeny)}$$

Príkaz priradenia p tvaru $\overline{y} = p(\overline{x}, \overline{y})$:

$$R_\alpha(\overline{x}, p(\overline{x}, \overline{y})),\ r_\alpha(\overline{x}, p(\overline{x}, \overline{y}))$$

True vetva podmienky i tvaru $if(i(\overline{x}, \overline{y}))$:

$$R_\alpha(\overline{x}, \overline{y}) \wedge i(\overline{x}, \overline{y}),\ r_\alpha(\overline{x}, \overline{y})$$

Verifikačná podmienka pre cestu α medzi deliacimi bodmi A a B je potom $I_A\ \&\ R_\alpha(\overline{x}, \overline{y}) \Rightarrow I_B(r_\alpha(\overline{x}, \overline{y}))$.

1.3.4 Dokázanie verifikačných podmienok

Posledný krok Floydovej metódy je overiť všetky skonštruované verifikačné podmienky, v našom prípade formuly predikátovej logiky.

Veta. Ak platia všetky verifikačné podmienky skonštruované korektne na programe P krokmi 1-3, potom program P je čiastočne správny.

Dôkaz. Presahuje cieľ tejto práce.

1.3.5 Príklad

Program počíta $\sqrt{x}$, bol prevzatý a upravený z [2].

$begin\{x \geq 0\}$

1. $y_1 := 0;$
2. $y_2 := 0;$
3. $y_3 := 1;$
4. $y_2 := y_2 + y_3;$
5. $if(y_2 > x)\ goto\ end;$
6. $y_1 := y_1 + 1;$
7. $y_3 := y_3 + 2;$
8. $goto\ 4;$

$end\{z > x^{(x/2)}\}$

1. krok: Rozdelenie programu na deliace body.
Program obsahuje jeden cyklus, deliaci bod je najvýhodnejšie uložiť na riadok 5, pretože to je miesto odkiaľ sa z cyklu vychádza. Označme si vzniknuté cesty α cestu zo začiatku programu do riadku 5, β cestu z riadku 5 naspäť na riadok 5 a γ cestu z riadka 5 na koniec programu.

2. krok: Konštrukcia invariantov.
Na sformulovanie invariantov I_B a I_E použijeme vstupnú a výstupnú podmienku a pre deliaci bod v riadku 5 použijeme nejakú zo známych heuristík, niektoré z nich môžete nájsť napríklad v [2].

Skonštruované invarianty:

$I_B : x \geq 0$

$I_5 : (y_1^2 \leq x \wedge y_2 = (y_1 + 1)^2 \wedge y_3 = 2y_1 + 1$

$I_E : z^2 \leq x < (z + 1)^2$

3. krok: Konštrukcia verifikačných podmienok.

Aplikovaním postupu z časti 3.3 dostávame tieto podmienky:

cesta α:

$$\forall x[I_B(x) \wedge true \Rightarrow I_2(x, 0, 1, 1)]$$

cesta β:

$$\forall x, y_1, y_2, y_3[I_2(x, y_1, y_2, y_3) \wedge y_2 \leq x \Rightarrow I_2(x, y_1 + 1, y_2 + y_3 + 2, y_3 + 2)]$$

cesta γ:

$$\forall x, y_1, y_2, y_3[I_2(x, y_1, y_2, y_3) \wedge y_2 > x \Rightarrow I_E(x, y_1)]$$

4. krok: Dokázanie verifikačných podmienok.

Prenechávame na usilovného čitateľa.

Kapitola 2

Vizualizácia

Cieľom predchádzajúcich častí bolo dať čitateľovi teoretické vedomosti ohľadom správnosti programov a ich dokazovania. Špeciálne dokazovania čiastočnej správnosti programu pomocou Floydovej metódy. Ukázali sme si taktiež aj zápis konkrétneho programovacieho jazyka, ktorý je svojou jednoduchosťou vhodný na vizualizáciu, a príkazmi, ktorými je vybavený, postačuje na tvorbu jednoduchých programov.

Hlavným cieľom tejto bakalárskej práce je však vytvoriť aplikáciu, ktorá dopomôže k lepšiemu pochopeniu tejto metódy tým, že bude do čo najväčšej miery vizualizovať kroky Floydovej metódy.

Ako sme sa dozvedeli z predchádzajúcej časti, metóda obsahuje kroky, ktoré je možné algoritmicky vyriešiť, ale aj tie, ktoré sú nerozhodnuteľnými problémami.

Snažili sme sa dosiahnuť, aby časti, ktoré sa dajú implementovať, slúžili iba ako kontrola pre študenta. Necháme teda priestor študentovi na to, aby si každý krok vyskúšal najprv sám a až potom ho skontrolujeme.

Medzi najdôležitejšie mechanizmy, ktoré sme implementovali, patria kontrola cyklov a generovanie verifikačných podmienok.

Ak študent urobil chybu pri výbere deliacich bodov, pokúsime sa mu čo najviac dopomôcť pochopiť, kde chybu urobil, tým, že mu ukážeme, do ktorého cyklu ešte nepridal deliaci bod.

Pri generovaní verifikačných podmienok mu necháme priestor na to, aby si ich skúsil odvodiť sám a potom si môže po krokoch kontrolovať správnosť ním vytvoreného

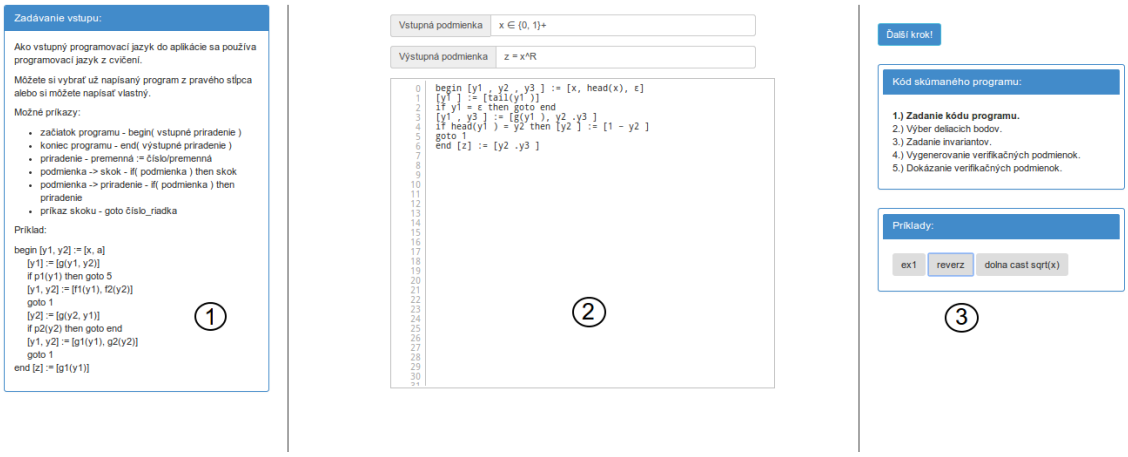
riešenia podľa nášho vygenerovaného.

Pri častiach, ktoré nevieme implementovať, sme sa snažili vytvoriť čo najlepšie prostredie pre úspešné zvládnutie kroku. Študentovi sme presne vymedzili, čo má urobiť, akým spôsobom má správne použiť naše prostredie. Taktiež má študent k dispozícii tie teoretické informácie, ktoré mu dopomôžu prísť na to, ako má správne vykonať aktuálny krok.

V tejto kapitole si ukážeme, akým spôsobom sme vizualizovali konkrétne časti Floydovej metódy. Keďže sme sa stretli s viacerými problémami, ukážeme si ich a povieme si, aké riešenie sme volili.

2.1 Rozloženie prvkov na stránke

Grafické okno aplikácie sme rozdelili na tri hlavné časti:



Obr. 2.1: Rozloženie prvkov na stránke

2.1.1 Plocha pre sprievodné texty

Ľavá časť pracovného prostredia, na obrázku číslo 2.1 je označená číslom 1. Táto plocha stránky je určená na zobrazovanie sprievodných textov, v ktorých študent nájde stručne vysvetlený krok, v ktorom sa práve nachádza. Obsahom bude hlavne teória aktuálneho kroku, ale aj inštrukcie, ako pracovať s užívateľským prostredím aplikácie. V kroku pre zadávanie vstupu študentovi zobrazíme zoznam príkazov a ukážkový program.

2.1.2 Vizualizačná plocha

Stredná časť pracovného prostredia, na obrázku číslo 2.1 je označená číslom 2. Je to hlavná a najväčšia časť, v ktorej sa zobrazuje vizualizácia. Pre každý krok má špecifickú funkciu:

- Zadanie vstupu - zobrazenie editora s očíslovanými riadkami, do ktorého študent vpisuje vstupný program. Študent tu taktiež dostane možnosť kliknutím na niektoré z tlačidiel, ktoré reprezentujú ukážkové programy, predvyplniť editor kódom programu, ktorý si vyberie.
- Výber deliacich bodov - vykreslenie programu ako vývojového diagramu, do ktorého sa klikaním na vrcholy dajú pridávať deliace body.
- Konštrukcia invariantov - zobrazenie textových políčok k deliacim bodom, do ktorých študent vpisuje invarianty.
- Konštrukcia verifikačných podmienok - vykreslenie ciest a následné zobrazenie postupu generovania verifikačných podmienok.
- Dokazovanie verifikačných podmienok - vypísanie verifikačných podmienok a zobrazenie stavu dokazovania.

2.1.3 Plocha zobrazujúca stav vizualizácie

Pravá časť pracovného prostredia, na obrázku číslo 2.1 je označená číslom 3. Vrchná podčasť tejto časti slúži na zobrazenie stavu vizualizácie, teda v ktorom kroku sa študent aktuálne nachádza. Spodná podčasť zobrazuje kód programu, ktorý študent zadal do editora.

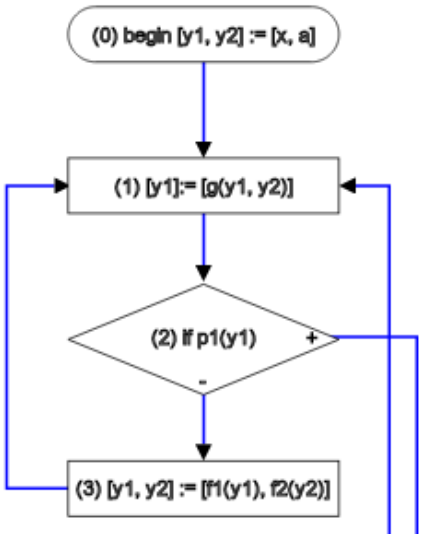
2.2 Zobrazenie programu vývojovým diagramom

V predchádzajúcich častiach sme hovorili o reprezentácii programu grafom s orientovanými hranami. Rozhodli sme sa zvoliť dizajn grafu na spôsob vývojového diagramu.

Vývojové diagramy sa väčšinou používajú na znázorňovanie algoritmov, ale jednoduchosť programovacieho jazyka, ktorý používame ako vstup do programu, nám umožňuje využiť tento druh diagramu s dostatočne jednoduchým a prehľadným výsledkom.

2.2.1 Príkazy

Pre lepšiu podobnosť grafu a programu vo forme textu sme sa rozhodli do útvaru, ktorý príkaz reprezentuje, zapísať celý príkaz. Každý typ príkazu má odlišný tvar útvaru. Snažili sme sa vybrať najčastejšie používané označovanie pre tento typ vývojového diagramu [6].



Obr. 2.2: Spôsob vykresľovania príkazov

Na obrázku číslo 2.2 je ukázané zobrazovanie príkazov podľa typu. Ďalej popíšeme príkazy podľa obrázka číslo 2.2 a čísel im prislúchajúcim:

0. Príkaz začiatku a konca programu je zobrazený v obdĺžniku s okrúhlymi kratšími hranami, v ktorom sa nachádza aj počiatočné priradenie vstupných premenných do pracovných.

1. Príkaz priradenia v obdĺžniku.

2. Príkaz podmienky v štvorci natočenom o 45 stupňov, podľa dĺžky príkazu roztiahnutý do bokov. Pravdivá vetva tvorí šípku označenú symbolom plus, nepravdivá

je označená symbolom mínus.

3. Pri tomto čísle si môžeme všimnúť znovu príkaz priradenia, avšak z neho smeruje šípka na príkaz označený číslom 1. Spomínaná šípka nám teda reprezentuje príkaz skoku, pre ktorý sme sa rozhodni nevytvoriť samostatný typ útvaru, ale zobrazujeme ho len týmto spôsobom.

2.2.2 Postupnosť vykresľovania príkazov

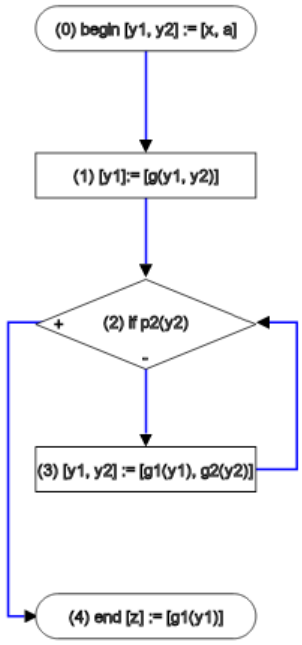
Počas parsovania vstupu a následnej implementácie vykresľovania vývojového diagramu sme objavili dva prístupy z hľadiska postupnosti príkazov, ktorými sa dá logicky zobraziť program.

Napríklad pre vstup:

$begin\{[y1, y2] := [x, a]\}$

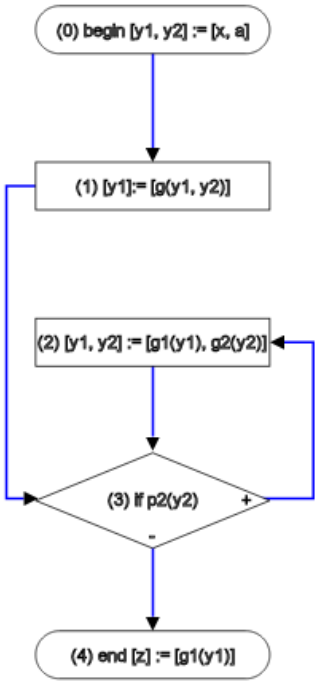
1. $y_1 := [g(y_1, y_2)]$;
2. $goto$ 4;
3. $[y_1, y_2] := [g_1(y_1), g_2(y_2)]$;
4. $if p_2(y_2)\ then\ goto$ 3;

$end\{[z] := [g_1(y_1)]\}$



Obr. 2.3: Prvý spôsob vykresľovania poradia príkazov

Obrázok číslo 2.3 ukazuje prvý prístup. Príkazy sú zoradené podľa toho, v akom poradí sú vykonávané. V niektorých prípadoch tento prístup prináša menší počet bočných hrán a tým aj prehľadnejšie vykreslenie. Tento prípad sa zdá byť prirodzenejší, má však jednu veľkú nevýhodu. Vo väčšine prípadov nie je zhodné poradie príkazov v programe vo forme vývojového diagramu a programu v textovej forme.



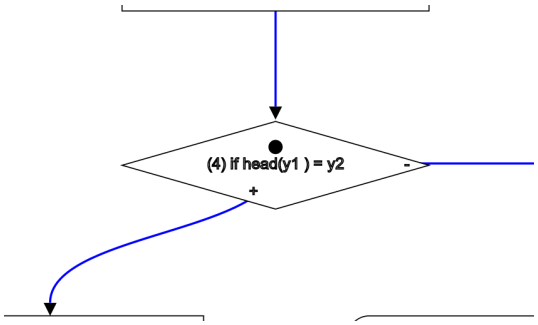
Obr. 2.4: Druhý spôsob vykresľovania poradia príkazov

Obrázok číslo 2.4 ukazuje druhý prístup. Príkazy sú zoradené podľa toho, v ktorom riadku sa nachádzajú v textovej forme programu zo vstupu, a iba šípky ukazujú skutočné poradie vykonávania programu. Jedná sa o priamočiarejšie prepojenie programu a vizualizácie, čo je z pedagogického pohľadu vhodnejšie. Po nakreslení a porovnaní niekoľkých príkladov sme usúdili, že tento prístup je lepší, čo nám potvrdil aj fakt, že sme našli oveľa viac príkladov používania tohoto prístupu v praxi.

2.2.3 Označovanie deliacich bodov

Rozhodli sme sa, že pre jednoduchosť bude mať študent možnosť pridávať deliaci bod iba pred príkaz. Teoreticky je možné deliaci bod umiestniť aj dovnútra príkazu, ale pre potreby vyučovania je tento spôsob postačujúci.

Po kliknutí študenta na vrchol sa nad ním vytvorí guľôčka s označením deliaceho bodu, ako je znázornené na obrázku číslo 2.5.



Obr. 2.5: Deliaci bod pred príkazom

2.3 Zobrazenie textových políčok na zadávanie verifikačných podmienok

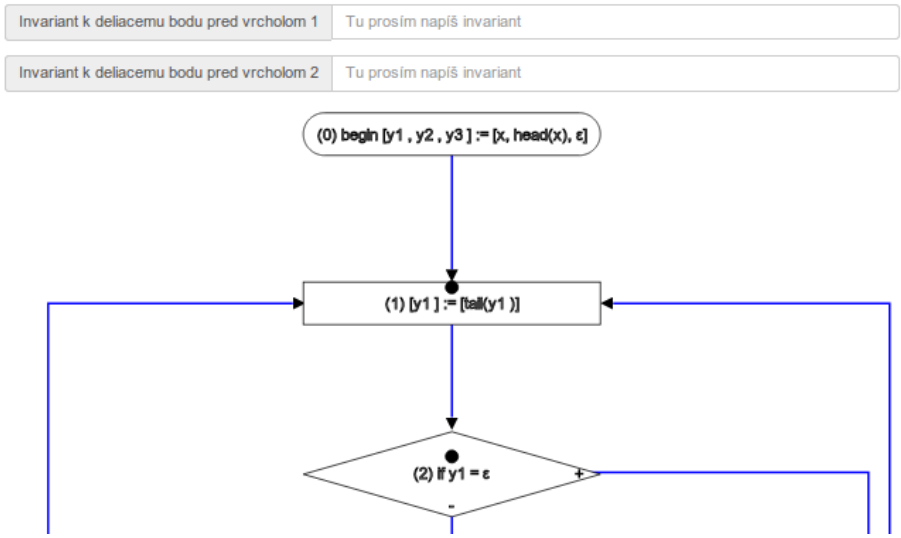
Potom ako študent označí deliace body a prejde na ďalší krok vizualizácie, mu potrebujeme dať možnosť, aby pre označené deliace body napísal invarianty. Teda podmienky, ktoré sú pri každom prechode daným bodom platné.

Vytvorenie invariantu je pre študenta väčšinou jedna z najťažších úloh. Okrem stručného návodu zobrazeného na ploche pre sprievodné texty, môže študentovi veľmi pomôcť strom reprezentujúci program, ktorý bol zobrazený v predchádzajúcom kroku. Preto sme sa rozhodli ho nechať v čitateľnej veľkosti. Pri väčšine programov by to nutne znamenalo, že ostane zobrazený na ploche pre vizualizáciu, pretože na ploche pre zobrazenie stavu vizualizácie je pri nízkom rozlíšení monitora príliš málo miesta.

Potrebujeme teda na vizualizačnú plochu, okrem spomínaného stromu, zobraziť aj taký počet textových políčok, koľko študent označil deliacich bodov. Tento program je vytváraný na výučbu a rozumná dĺžka programov, ktoré sa zvyknú predvádzať na cvičeniach alebo uvádzať v cvičebniciach je maximálne do dvadsať riadkov kódu. Z tohoto dôvodu nám stačí rozmýšľať v priemere o jednom až troch textových políčok, ktoré potrebujeme pridať.

Pre študenta by pravdepodobne bolo prirodzené, ak by sa textové políčko zobrazilo hneď vedľa študentom označeného deliaceho bodu. Avšak dĺžka invariantu sa bežne môže pohybovať aj v desiatkach znakov, čo by nás nútilo výrazne zúžiť alebo zmenšiť strom reprezentujúci program. Preto sme sa rozhodli zvoliť riešenie, ktoré je zobrazené

na obrázku číslo 2.6. Strom sa posunie nižšie o potrebný počet pixelov, podľa toho, koľko textových políčok potrebujeme. Keďže sú deliace body a políčka zmysluplne označené a v poradí, v akom sú v strome umiestnené deliace body, nemal by mať študent problém sa zorientovať po prechode na tento krok a bude mať dostatočný priestor na zapísanie aj zložitejších invariantov.



Obr. 2.6: Zobrazenie textových políčok

2.4 Zobrazenie ciest

Ďalším krokom je vygenerovanie verifikačných podmienok. Vytvorili sme algoritmus, ktorý verifikačné podmienky generuje za študenta. Keďže ale prvoradou úlohou programu je naučiť študenta ich vytvárať samotného, vymýšľali sme spôsoby, ako to dosiahnuť čo najlepšie.

Ak chceme zobraziť nie len konečnú verifikačnú podmienku, ale aj postup ako bola odvodená, potrebujeme uskutočniť nasledujúce úlohy:

- Každú cestu viditeľne zobraziť a pomenovať, vyznačiť na nej vrcholy
- Pre každý stav odvodzovania verifikačnej podmienky nájsť miesto pre vypísanie aktuálneho stavu R a r, ktoré reprezentujú stav podmienky respektíve stav vektorov premenných po prejdení cesty od konca po vrchol, pre ktorý podmienku zobrazujeme

- Znázorniť smer odvádzania, napríklad šípkou

2.4.1 Zobrazenie ciest použitím vývojového diagramu

Jedným z rozumne vyzerajúcich riešení je využiť už viac krát použitý strom reprezentujúci program a na ňom postupne vizualizovať generovanie podmienok. Toto riešenie však prináša viacero problémov.

Keďže dve cesty môžu mať spoločnú časť cesty, je potrebné ich medzi sebou odlišovať. Taktiež, ak by sme chceli mať naraz zobrazené stavy R a r pre viaceré cesty, len ťažko by sa nám hľadalo vhodné miesto na vizualizačnej ploche. Aj z poslednou úlohou by sa nám ťažko vysporiadavalo, pretože pridanie ďalších šípok do grafu by ho zneprehľadnilo.

Hlavne kvôli spomenutím dôvodom sme sa rozhodli hľadať iné riešenie.

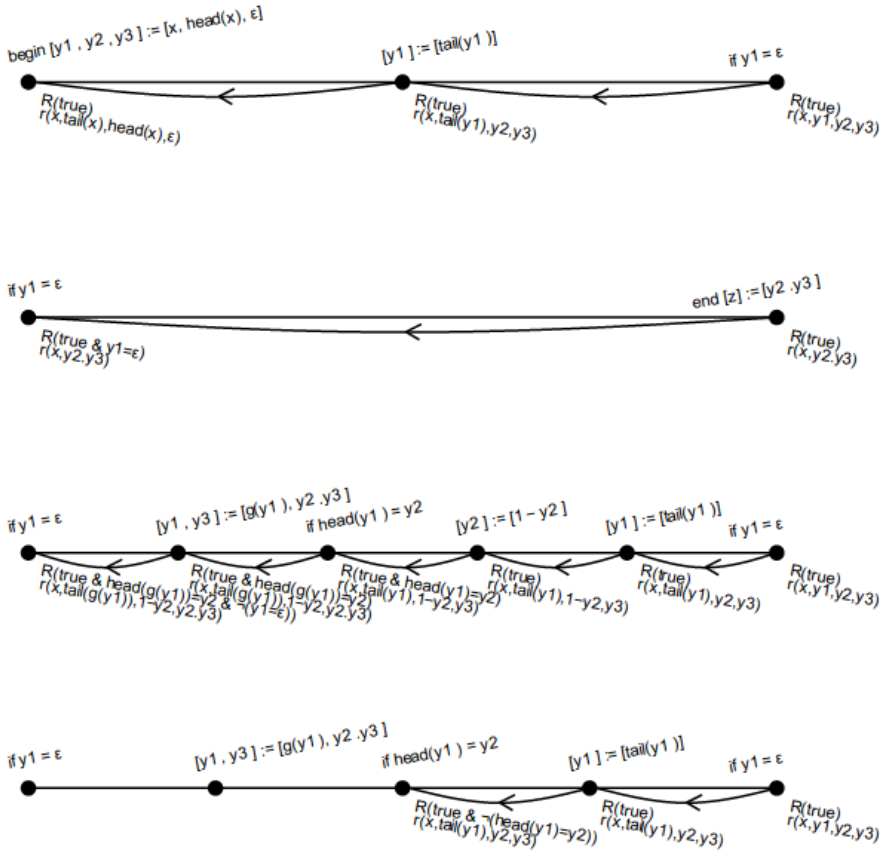
2.4.2 Zobrazenie ciest lineárne

Ako výhodnejší spôsob na zobrazenie procesu generovania verifikačných podmienok sa ukázalo byť zobrazenie každej cesty osobitne.

Cestu zobrazujeme vo forme horizontálnej čiary a vrcholy, ktoré sa na ceste nachádzajú reprezentujú rovnomerne rozmiestnené guličky. Nad každou guličkou je napísaný text príkazu, ako je to znázornené na obrázku číslo 2.7.

Poradie vrcholov určuje prirodzený smer cesty. Šípky ktoré vedú oblúkom popod túto čiaru nám ukazujú smer generovania.

Pod každou guľôčkou sú zobrazené zápisy R a r. Všetky texty sú mierne naklonené, aby sa texty neprekrývali.



Obr. 2.7: Zobrazenie spôsobu vykreslenia ciest

2.5 Zobrazenie verifikačných podmienok na dokázanie

Posledným krokom Floydovej metódy je dokázanie verifikačných podmienok. Študentovi vypíšeme verifikačné podmienky v plnej forme. To znamená, že dosadíme R a r do vzorca a nahradíme symboly reprezentujúce invarianty reálnymi invariantami, ktoré zadal študent v predchádzajúcich krokoch.

Vedľa každej verifikačnej podmienky umiestnime checkbox, ako je to znázornené na obrázku číslo 2.8. Po dokázaní verifikačnej podmienky si študent zaškrtne checkbox prislúchajúci verifikačnej podmienke. Po zaškrtnutí všetkých checkboxov študentovi zagratulujeme k úspešnému dokázaniu programu. Týmto je vizualizácia pre daný program ukončená.

Verifikačná podmienka	Dokázané?
(x ∈ {0, 1}+ & true)=>(ε .head(x) .tail(x) = x)	☑
(y3 .y2 .y1 = x & true & y1=ε)=>(z = x^R)	☑
(y3 .y2 .y1 = x & true & head(g(y1))=y2 & ¬(y1=ε))=>(1−y2.y3 .1−y2 .tail(g(y1)) = x)	☑
(y3 .y2 .y1 = x & true & ¬(head(g(y1))=y2) & ¬(y1=ε))=>(y2.y3 .y2 .tail(g(y1)) = x)	☐

Obr. 2.8: Zobrazenie verifikačných podmienok

Kapitola 3

Implementácia

Hlavným z cieľov bakalárskej práce je vytvoriť plne funkčnú a ľahko používateľnú aplikáciu na výučbu Floydovej metódy. Z tohto dôvodu bola pri výbere technológií jedným z hlavných kritérií dostupnosť aplikácie. Túto aplikáciu sme sa rozhodli vytvoriť na webovej platforme, aby nebola potrebná žiadna inštalácia a študent mohol využiť pohodlie svojho prehliadača.

3.1 Použité technológie

Aplikácia je single-page, teda používateľ prejde všetkými časťami vizualizácie bez toho, aby sa celá stránka musela po prvom načítaní prekresľovať.

Programovací jazyk aplikácie je JavaScript, preto bude aplikácia funkčná iba za predpokladu, že klientský počítač ho bude plnohodnotne podporovať. Pri všetkých bežne používaných internetových prehliadačoch to nie je problém, ak nie je zakázaný používateľom. Na zrýchlenie práce som využil knižnicu jQuery.

Ako pri takmer každej vizualizácii, tak aj pri našej sme sa čo najviac snažili využívať obrazy. Preto sme potrebovali vybrať nástroj, pomocou ktorého by sme 2D grafiku v prehliadači vykresľovali.

3.2 Výber nástroja na vykresľovanie grafických častí

Prioritou pri vyberaní nástroja bola podpora prehliadačov, aby nebolo potrebné pre používanie našej aplikácie doinštalovávať podporný program, ako je Flash alebo iné. Zároveň sme potrebovali nástroj, pomocou ktorého dokážeme implementovať všetky časti vizualizácie. Hlavné požiadavky kladené na grafický nástroj boli:

1. Možnosť vykresliť graf reprezentujúci program, teda vykreslenie geometrických útvarov a čiar.

2. Schopnosť vyvolať udalosť v prípade, že užívateľ bude chcieť kliknutím myšou na príkaz, ktorý reprezentuje geometrický útvar, pridať deliaci bod.

3. Okamžité prekresľovanie časti obrázka. Napríklad po kliknutí na príkaz potrebujeme upovedomiť užívateľa, že sa mu podarilo pridať deliaci bod a po opätovnom kliknutí ho odobrať.

Najčastejšie používanými nástrojmi, ktoré spĺňajú tieto požiadavky, sú Canvas a SVG. S oboma týmito nástrojmi by bolo možné vizualizáciu implementovať [7]. Keďže sú ale tieto nástroje veľmi odlišné, výber výhodnejšieho prostriedku nám môže značne uľahčiť prácu.

3.2.1 Canvas

Súčasťou špecifikácie HTML5 je aj $< canvas >$ element, ktorý slúži na vykresľovanie grafiky založenej na princípe manipulácie s pixelmi. Práca s canvasom je veľmi jednoduchá. Môžeme využívať naprogramované funkcie na vykresľovanie geometrických útvarov.

Zvyšné dve požiadavky sú tiež realizovateľné. Po kliknutí na plochu canvasu dostávame údaj o presných súradniciach dotknutého bodu. Avšak ak chceme zistiť, či je spomínaný bod súčasťou niektorého z vrcholov, potrebujeme si pre každý vrchol držať množinu bodov, ktoré pokrýva. Potom už vieme realizovať označovanie deliacich bodov, ako aj detektovať, o aký vrchol ide.

3.2.2 SVG

SVG (Scalable Vector Graphics) je značkový jazyk z rodiny XML jazykov, ktorý nám umožňuje vykresliť 2D grafiku a animácie. Keďže pracuje na báze vektorov, obrázok vytvorený v jazyku svg je možné zväčšovať a zmenšovať bez straty ostrosti. Takisto poskytuje jednoduché funkcie na vykreslenie geometrických útvarov.

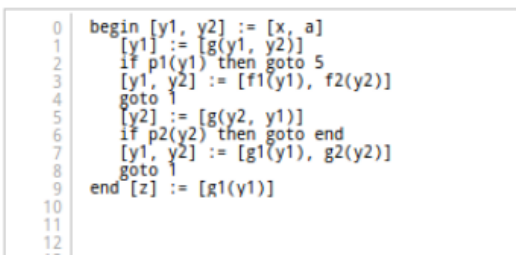
Každý samostatný útvar je reprezentovaný osobitným elementom. Do svg elementov vieme pridávať názvy tried alebo upravovať niektoré vlastnosti pomocou jazyka CSS. Tak ako pri html elementoch, aj pri svg elemente vieme pomocou jazyka JavaScript odchytávať udalosti vyvolané užívateľom. Keďže si do elementu vieme zapísať aj ľubovoľné pomocné informácie, dokážeme veľmi jednoducho zistiť, na ktorý vrchol užívateľ klikol.

SVG nám poskytuje omnoho elegantnejšie a jednoduchšie prostriedky, pomocou ktorých môžeme reagovať na užívateľovo správanie. Tento fakt bol pre nás kľúčový a preto sme uprednostnili SVG pred canvasom.

V ďalších častiach popisujeme problémy, s ktorými sme sa stretli počas implementácie aplikácie, a riešenia, ktoré sme použili. Každá podkapitola predstavuje jeden krok vizualizácie.

3.3 Krok 1: Zadávanie vstupu

V prvom kroku zadáva študent do editora kód programu, ktorý chce vizualizovať. Namiesto bežného html elementu textarea využívame jeho rozšírenie o číselnú lištu. Tá je dôležitá hlavne pre príkazy skoku, za ktorými nasleduje číslo riadku, ktorým má vykonávanie programu pokračovať ďalej.



Obr. 3.1: Editor s vyznačenými riadkami

Po ukončení zadávania vstupu študent stlačí potvrdzujúce tlačidlo. Následne overujeme, či je zadaný kód v správnom formáte, a či študent správne používa príkazy programovacieho jazyka. V prípade nekorektného vstupu program vypíše číslo riadku, v ktorom objavil chybu.

Vstup sa ďalej odošle do parsera a získajú sa z neho nasledujúce údaje:

1. Objekt v tvare JSON, ktorý sa použije na vytvorenie vrcholov grafu pomocou knižnice D3.

2. Pole objektov, ktoré reprezentujú riadky vstupu. Potrebujeme ich pre správne vytvorenie čiar po príkazoch skoku a pre detekciu cyklov.

3.4 Krok 2: Výber deliacich bodov

V tomto kroku má študent za úlohu rozbiť cykly programu, ak nejaké sú, deliacimi bodmi. Program reprezentujeme ako graf, kde príkazy programu tvoria vrcholy grafu a postupnosť ich možného vykonávania definujú orientované čiary. Pri vizualizovaní tohto kroku sa nám teda vynára viacero problémov:

1. Rozmiestniť vrcholy na ploche pre vizualizáciu tak, aby bol graf čo najčitateľnejší, a ak to nie je nutné, aby sa neprekrýval žiaden z vrcholov.

2. Zabrániť pretínaniu hrán grafu alebo ho aspoň minimalizovať.

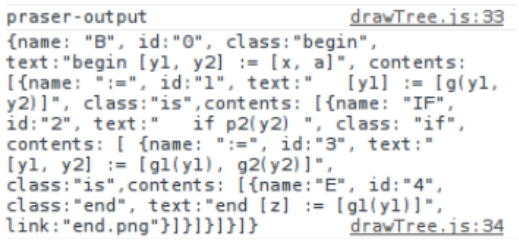
3. Zmysluplne označiť deliaci bod.

4. Skontrolovať, či študent rozbil všetky cykly v grafe.

3.4.1 Vypočítanie pozícií vrcholov

Základným problémom pri vykreslení vývojového diagramu je výpočet pozície vrcholov, aby bol graf čo najprehľadnejší a dojem z neho čo najestetickejší. Ak je v pravdivej vetve príkazu podmienky príkaz priradenia, strom sa nám rozvetví na dva podstromy a výpočet sa stane zložitejším.

Knižnica D3 je jednou z najznámejších knižníc na vizualizáciu dát jazyka Javascript. Jedným z jej layoutov je aj *tree layout*, ktorý dokáže vypočítať pozície vrcholov z dát vo formáte JSON, ktorý je zobrazený na obrázku číslo 3.2.



Obr. 3.2: Potrebné dáta vo formáte JSON

Dôležité dáta pre každý vrchol sú:

1. *id* - unikátny identifikátor, ktorý umožňuje presne určiť vrchol. Hlavné využitie je pri udalostiach vyvolaných užívateľom.

2. *class* - identifikátor triedy. Pre každý druh príkazu je unikátny. Používa sa pri štýlovaní vrcholov.

3. *text* - kompletný text príkazu, ktorý sa užívateľovi zobrazí ako text vo vrchole.

4. *contents* - zoznam detí v rovnakej štruktúre.

Pre samotný výpočet pozícií vrcholov je najdôležitejšia práve štruktúra JSON objektu, a teda ktorý vrchol má aké a koľko detí. Medzi deti vrchola sú v našom prípade zaradené iba vrcholy, do ktorých sa nedostaneme pomocou príkazu *goto*, pretože tento príkaz reprezentujeme iba bočnou šípkou.

Ďalšími údajmi, ktoré knižnica potrebuje, sú veľkosť plochy, ktorú môže použiť na vykreslenie grafu, a veľkosť vrchola.

Rozmery vypočítame z veľkosti vykresľovacej plochy prehliadača. Výšku okna získame jednoducho pomocou príkazu $\$(window).width()$ a odpočítaním odsadenia. Vypočítať šírku okna je trocha komplikovanejšie. Graf má obvykle niekoľko bočných hrán, ktoré dostaneme príkazmi *goto*. Keďže nechceme, aby sa hrany prekrývali, každú odsadíme od predchádzajúcej o stanovenú dĺžku. Pri parsovaní je preto dôležité zapamätať

si najväčší počet hrán na jednej strane. Poslednú vec, s ktorou musíme rátať, je odpočítanie šírky posuvnej lišty. Príkaz vracajúci šírku okna počíta s tým, že je zobrazená vždy a my chceme dosiahnuť, aby užívateľ videl celý graf naraz.

3.4.2 Minimalizovanie pretínania hrán

Dôležitou súčasťou vizualizácie je aj celková prehľadnosť. Konkrétne pri našom grafe ju môžu narušovať hrany, ktoré vychádzajú a vchádzajú do rovnakých vrcholov alebo sa pretínajú navzájom.

Vrcholy nášho grafu ležia na jednej priamke, okrem prípadu, ak za príkazom podmienky nasleduje priradenie. V tom prípade sa nám jeden vrchol vysunie mimo tejto priamky. Tento fakt nám umožňuje vybrať, či ľubovoľnú hranu nakreslíme napravo od grafu alebo naľavo. Týmto spôsobom vieme aspoň čiastočne sprehľadniť vzniknutí graf.

Pri hľadaní algoritmu na implementáciu tohto delenia hrán sme sa dostali k zaujímavému zdroju [4].

Autori sa v ňom venovali úrovňovému vykresľovaniu grafov, konkrétne jeho častí, kde sa vrcholy na jednotlivých úrovniach usporiadavajú tak, aby sa minimalizoval počet krížení hrán. Vo všeobecnosti je tento problém NP-úplný [5]. Autori ukázali, že to platí aj za predpokladu, že na každej úrovni je iba jeden normálny vrchol a niekoľko vrcholov, ktoré označujú hrany prechádzajúce cez túto úroveň. Znamená to, že rozhodnúť, ktoré hrany majú byť vpravo respektíve vľavo od vrchola a v akom poradí, je ťažký problém. Autori navrhujú aj heuristiku ako problém riešiť, avšak pri veľkosti grafu, čo používame, sme sa rozhodli implementovať jednoduchší algoritmus.

Použité riešenie najskôr zabezpečí, aby pre hranu vybral stranu, na ktorej bude mať minimálny počet hrán, ktoré končia v rovnakom vrchole, v ktorom daná hrana začína, respektíve končí. Ak sú z tohto hľadiska strany rovnocenné, algoritmus vyberie tú stranu, ktorá ma menší počet už pridelených hrán.

3.4.3 Označenie deliaceho bodu

Ako sme už spomenuli v popise SVG jazyka, JavaScript nám umožňuje zachytiť udalosť kliknutia na vrchol grafu. Jediné, čo sme potrebovali urobiť, bolo vytvoriť funkciu, ktorá čaká na užívateľovo kliknutie, a do elementu, ktorý reprezentuje daný vrchol, pridá informáciu identifikujúcu vrchol ako deliaci bod.

3.4.4 Kontrola cyklov

Po označení deliacich bodov študentom je potrebné skontrolovať, či sa mu podarilo prerušiť všetky cykly. Tento kontrolný mechanizmus pracuje na základe rekurzívnej funkcie. Pomocou prehľadávania do hĺbky prejdeme všetkými možnými cestami po grafe. V prípade, že narazíme na cyklus, overíme si, či sa na ceste nachádza deliaci bod. Ak nie, vypíšeme študentovi upozornenie s číslami vrcholov, ktoré tvoria cyklus. Ak graf cyklus neobsahuje, chceme mať všetky cesty uložené v poli, aby sme mohli podľa nich generovať verifikačné podmienky.

3.5 Krok 3: Zadávanie invariantov

Tento krok je pre nás veľmi jednoduchý na implementáciu. Z dôvodov, ktoré sme popísali v kapitole 2, potrebujeme pridať na hornú časť vizualizačnej plochy textboxy s názvami deliacich bodov.

Vďaka schopnosti jazyka JavaScript manipulovať s HTML DOM, čiže so štruktúrou stránky zloženej z elementov, dokážeme elementy napríklad skrývať, pridávať, mazať alebo meniť ich atribúty. Táto cenná vlastnosť nám postačí na vykonanie aktuálneho kroku. Nad SVG element, v ktorom sa nachádza vygenerovaný graf, umiestnime prázdny div element. Tento element bude pri ostatných krokoch skrytý. V tomto kroku preiterujeme cez všetky vrcholy, ktoré boli v predchádzajúcom kroku označené ako deliaci bod, a pre každý jeden pridáme do div elementu jedno políčko na zápis invariantu s označením daného bodu.

3.6 Krok 4: Odvodenie verifikačných podmienok

V predošlom kroku sa nám podarilo nájsť všetky cesty v grafe, na ktorých koncoch sú vždy invarianty. Našou úlohou je pre každú cestu vyskladať zápis R, teda podmienku, ktorá je vždy platná po prejdení danou cestou. Takisto si potrebujeme zapamätať aj zmenu vektora $\overline{y}$, ktoré označujeme r.

Ako je spomenuté v kapitole 2, študentovi chceme zobraziť aj postup, akým boli verifikačné podmienky odvodené, nie len ich konečný tvar. To dosiahneme jednoducho tak, že si každý stav generovania uložíme do poľa.

Budovanie R a r je implementované postupnou spätnou iteráciou cez vrcholy, z ktorých je zložená cesta. Pre každý vrchol zistíme, aký príkaz reprezentuje, a podľa podmienok, ktoré sme uviedli v kapitole 1, upravíme zápisy R a r. Následne si ich odložíme do poľa a prejdeme na ďalšiu iteráciu.

3.7 Krok 5: Dokázanie verifikačných podmienok

V poslednom kroku vizualizácie je našou úlohou vypísať študentovi verifikačné podmienky. Keďže sme v predchádzajúcom kroku vygenerovali pre každú cestu R a r, stačí nám ich dosadiť do vzorca.

Vytvoríme tabuľku, kde jeden riadok tvorí verifikačná podmienka a checkbox, ktorý študent zaškrtne, ak sa mu ju podarilo dokázať. Pri každom označení podmienky za dokázanú vykonáme kontrolu, či sa študentovi podarilo dokázať všetky podmienky. Kontrolu realizujeme preiterovaním cez všetky checkboxy a zistením si hodnoty každého z nich.

Kapitola 4

Demonštrácia aplikácie na konkrétnom príklade

V tejto kapitole si ukážeme ako sa aplikácia používa na konkrétnom príklade. Cieľom je popísať fungovanie aplikácie z pohľadu študenta. Uvedieme tu aj niekoľko rád, ktoré môžu dopomôcť k pohodlnejšiemu používaniu aplikácie.

4.1 Zadanie kódu programu

Obr. 4.1: Prvý krok vizualizácie

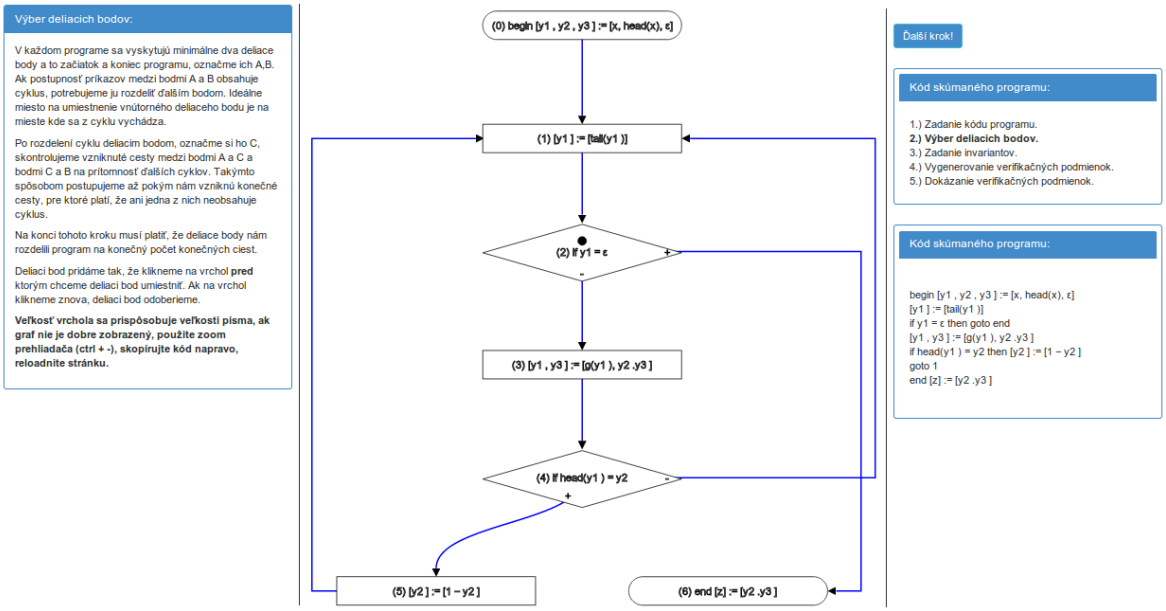
Na obrázku číslo 4.1 môžete vidieť úvodnú obrazovku aplikácie. Máme viacero možností, ako prejsť týmto krokom. Najjednoduchšie a najpohodlnejšie je vybrať si jeden z ukážkových príkladov. Na tento účel slúžia tlačítka v hornej časti vizualizačnej plochy. Po kliknutí na jeden z nich sa predvyplní textové pole pre kód programu a takisto aj textové polia vstupnej a výstupnej podmienky.

Ďalšou možnosťou je napísať si vlastný program. Príkazy, ktoré môžeme používať máme vypísané na ploche pre sprievodné texty.

V našom príklade si ukážeme príklad, ktorý počíta reverz. Kód programu spolu s vstupnou a výstupnou podmienkou môžete vidieť na obrázku číslo 4.1.

Po potvrdení tohto kroku prebehne kontrola syntaxe a na prípadné chyby nás program upozorní.

4.2 Výber deliacich bodov



Obr. 4.2: Druhý krok vizualizácie

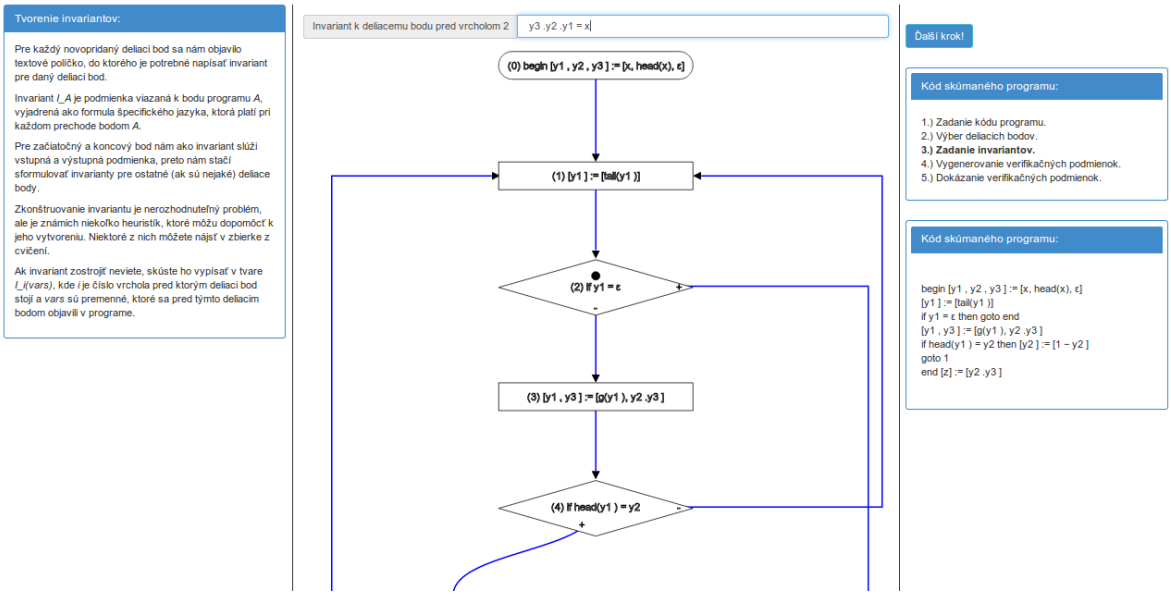
Obrázok číslo 4.2 nám ukazuje druhý krok vizualizácie. Na vizualizačnej ploche môžeme vidieť program vo forme vývojového diagramu.

V tomto kroku je našou úlohou kliknutím na niektoré vrcholy vybrať deliace body programu tak, aby sa v každom cykle nachádzal aspoň jeden z nich. Pri pohľade na graf je ľahké všimnúť si, že program obsahuje cykly práve dva. Konkrétne $1 \to 2 \to 3 \to 4 \to 5 \to 1$ a $1 \to 2 \to 3 \to 4 \to 1$. Deliaci bod umiestnime pred vrchol 2, tým pádom bude zasahovať do oboch cyklov a v ďalšom kroku nám bude stačiť zostrojiť iba jeden invariant.

Po potvrdení tohto kroku prebehne kontrola cyklov. Program nás upozorní a vypíše nám poradie vrcholov cyklu, ak nejaký neobsahuje deliaci bod.

Ak by sme mali problém s tým, že sa nám vrcholy prekrývajú kvôli dĺžke programu a počtu vrcholov, je potrebné si pred týmto krokom pomocou lupy v prehliadači upraviť veľkosť objektov. Návod ako to dosiahnuť je napísaný na ploche pre sprievodné texty.

4.3 Zadanie invariantu



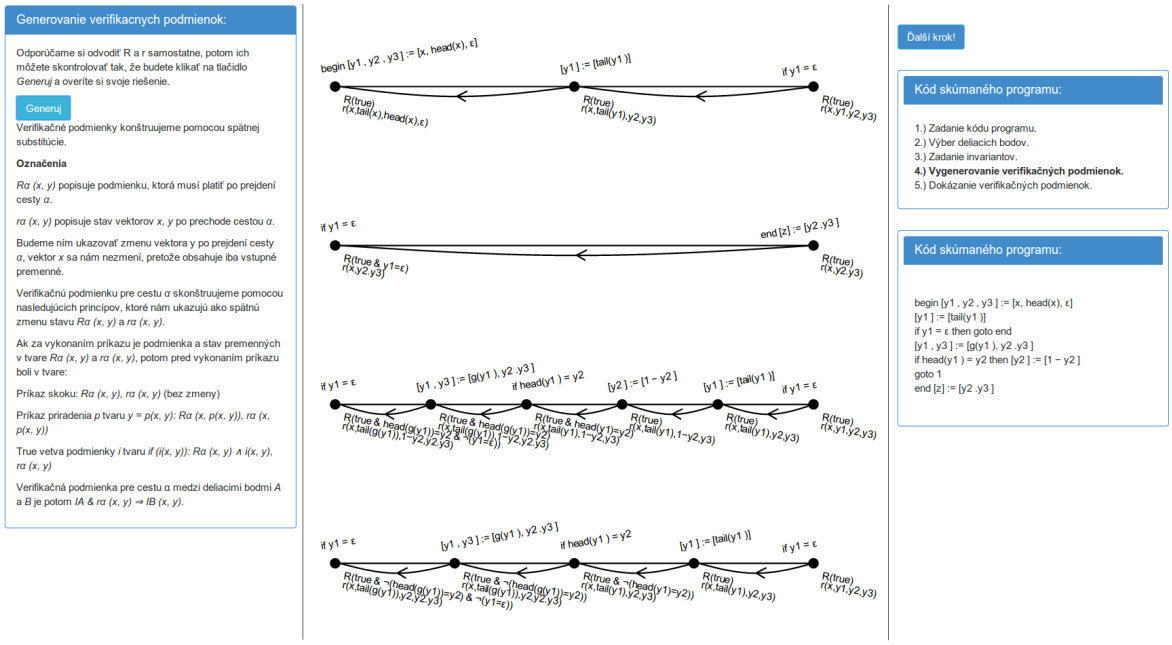
Obr. 4.3: Tretí krok vizualizácie

Oproti predchádzajúcemu stavu nám na vizualizačnej ploche pribudlo textové políčko s označením deliaceho bodu, ktorý sme vybrali v predchádzajúcom kroku. Môžeme ho vidieť na obrázku číslo 4.3.

Pri tomto kroku si študent musí poradiť sám a s pomocou jednej z heuristík odvodiť invariant.

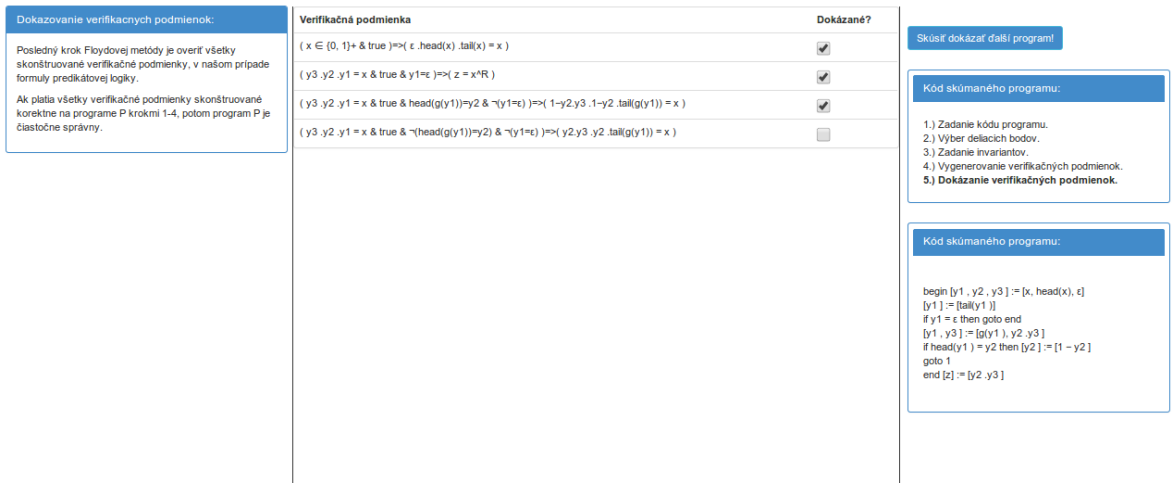
Pre tento príklad je invariant v tvare $y_3^R.y_2.y_1 = x$.

4.4 Generovanie verifikačných podmienok



Obr. 4.4: Štvrtý krok vizualizácie

Predposledným krokom vizualizácie je odvodenie verifikačných podmienok. Na začiatku sa nám zobrazia cesty s označeniami vrcholov. Postupným klikaním na tlačidlo sa zobrazujú postupne aktualizované zápisy R a r. Pre náš príklad ich môžete vidieť vygenerované všetky na obrázku číslo 4.4.



Obr. 4.5: Piaty krok vizualizácie

Na obrázku číslo 4.5 sú zobrazené verifikačné podmienky, ktoré vznikli dosadením

R, r a invariantov do vzorca.

Záver

Bakalárska práca spĺňa ciele stanovené v úvode a spolu s vytvorenou aplikáciou by mali výrazne zjednodušiť a urýchliť pochopenie Floydovej metódy.

Pre študenta môže slúžiť ako plnohodnotný teoretický návod k Floydovej metóde, pomocou ktorej je možné dokázať čiastočnú správnosť programu.

Práca ďalej popisuje vývoj aplikácie. Zaoberáme sa v nej spôsobmi, ktoré sme si vyberali pre vizualizáciu konkrétnych krokov. Taktiež popisuje krok po kroku implementáciu aplikácie. Počas programovania sme narazili na množstvo zaujímavých problémov a použili sme viacero zaujímavých technológií, do tejto práce sme sa snažili vybrať najzaujímavejšie z oboch. Do bakalárskej práce sme pridali aj kapitolu, ktorá má za cieľ ukázať prácu s vytvorenou aplikáciou na konkrétnom príklade. Študentovi, ktorému ide iba o pochopenie Floydovej metódy, odporúčame prečítať si prvú a poslednú kapitolu.

Dúfame, že v budúcnosti sa podarí vytvoriť viac použiteľných aplikácií na edukačné účely. Myslíme si, že aj v teoretickej informatike je stále ešte veľa nespracovaných zaujímavých tém.

Dodatok A

Zdrojové kódy

Súčasťou práce je priložené CD, na ktorom sú uložené zdrojové kódy aplikácie.

Literatúra

[1] Jombík, O.: *Matematická teória programovania. Zbierka riešených úloh*, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 2007

[2] Prívara, I.: *Základy matematickej teórie programov*, Inštitút informatiky a štatistiky,Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 2003

[3] Orendáč, M. - Floydova metóda, dostupné na internete (11.5.2014): s.ics.upjs.sk/morendac/Floydova%20Metoda.ppt

[4] Masuda, S., Nakajima, K. Kashiwabara, T.,Fujisawa, T.,:*Crossing Minimization in Linear Embeddings of Graphs*, IEEE Transactions On Computers, 1990

[5] Garey, M. , Johnson, D.: *Crossing Number is NP-Complete*, SIAM Journal on Algebraic Discrete Methods, Vol. 4, No. 3 : pp. 312-316, 1983

[6] Blauch, J.,A., Johnson, D., P.: *Structured Design Using Flowcharts*, Padnos School of Engineering, 2001

[7] dev.opera.com - Svg or Canvas - Choosing Between the Two, dostupné na internete (11.5.2014):http://dev.opera.com/articles/svg-or-canvas-choose/