Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia grafu distribuovaného výpočtu pre nástroj Paranoia

Bakalárska práca

2017
Martin Pecen

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia grafu distribuovaného výpočtu pre nástroj Paranoia

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Jana Katreniaková, PhD

Bratislava, 2017
Martin Pecen

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Vizualizácia grafu distribuovaného výpočtu pre nástroj Paranoia
Visualization of distributed execution graph for Paranoia tool

Cieľ: Kontrola korektnosti distribuovaných programov je náročná úloha, pretože je nemožné skontrolovať všetky alternatívy, ako môžu jednotlivé podúlohy na rôznych vláknach bežať paralelne. Nástroj Paranoia, na základe spustenia aplikácie na jednom vlákne, získa zoznam operácií vykonaných aplikáciou a vzťahy typu "X musí bežať pred Y". Tieto je možné chápať ako orientovaný graf, na základe ktorého Paranoia hľadá nekorektné možnosti výpočtu, kedy nie je zaručené správne poradie vykonania operácií, ktoré také poradie vyžadujú. Cieľom bakalárskej práce je vytvoriť grafické prostredie, ktoré zobrazí graf výpočtu a umožní navigáciu v ňom vrátane možnosti jednoducho identifikovať a prechádzať chýbajúce hrany nájdené nástrojom Paranoia.

Kľúčové slová: vizualizácia, graf, distribuovaný výpočet, OCR

Vedúci: RNDr. Jana Katreniaková, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 25.10.2016

Dátum schválenia: 25.10.2016 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakuje svojej školiteľke bakalárske práce,
RNDr. Jane Katreniakovej, PhD za cenné rady a pripomienky pri písaní tejto práce.

Abstrakt

Kontrola korektnosti distribuovaných programov je náročná úloha, pretože je nemožné skontrolovať všetky alternatívy, ako môžu jednotlivé podúlohy aplikácie bežať na rôznych vláknach paralelne. Nástroj Paranoia získa zoznam operácií vykonaných aplikáciou a vzťahy typu "X musí bežať pred Y". Tieto operácie je možné chápať ako orientovaný graf. Na základe tohto grafu Paranoia hľadá nekorektné možnosti výpočtu. V bakalárskej práci sme navrhli postup, ako takýto graf vykresliť, pomocou hierarchického kreslenia. Použili sme Sugiyamovho frameworku, ktorý spadá do tejto oblasti. Rozobrali sme kroky tohto frameworku a na základe nášho postupu sme vytvorili grafické prostredie, ktoré zobrazí tento graf získaný z Paranoie. Grafické prostredie taktiež poskytuje používateľovi zjednodušenie navigácie v grafe v podobe posúvania a minimalizovania vrcholov.

Kľúčové slová: vizualizácia, graf, distribuovaný výpočet, OCR, hierarchické kreslenie, Sugiyamov framework

Abstract

Checking the correctness of distributed programs is a challenging task, because it is impossible to check all the alternatives of how individual application sub-tasks can run on different fibers in parallel. Tool Paranoia will get a list of operations Executed by the application and relationships like "X must run before Y". These operations can be seen as an oriented graph. Based on this graph, Paranoia looks for incorrect calculation options. In the Bachelor's thesis we have proposed a procedure to plot such a graph, with the help of hierarchical drawing. We used Sugiyama's framework, which falls within this area. We've broken down the steps of the framework and based on it we have created a graphical environment that will display the graph acquired from Paranoia. The graphical environment also provides the user with simplified navigation in the graph in the form of moving and minimizing the vertices.

Keywords: visualization, graph, distributed computing, OCR, hierarchical drawing, Sugiyama's Framework

Obsah

Zoznam obrázkov

Úvod

Open Community Runtime je runtime, ktorý sa používa pri veľmi výkonných počítačoch. Programy, ktoré sú spúšťané v tomto prostredí sa vyznačujú tým, že majú veľké množstvo komponentov a preto zvyknú byť komplikované. Takéto programy sa reprezentujú pomocou orientovaných grafov. Je úplne prirodzené, že vznikajú nástroje, ktoré sa snažia hľadať chyby v takýchto grafoch, keď že sú rozsiahle a komplikované, čo vedie k chybám. Jedným z nich je nástroj Paranoia. Tento nástroj po ukončení validácie grafu chce zobraziť samotný graf, ktorý validoval, aby si používateľ mohol pozrieť chyby, prípadne ľahšie nájsť ďalšie, ktoré Paranoia nezachytila. Bohužiaľ zatiaľ neexistuje žiadny vizualizátor respektíve editor pre Open Community Runtime-mové programy. Kvôli čomu, musí Paranoia používať všeobecný nástroj na kreslenie grafov. Preto sme sa rozhodli jeden vytvoriť.

Našim cieľom tejto práce je vytvoriť vizualizátor grafov pre nástroj Paranoia. Oblasť vizualizácie grafov je pomerne dobre prebádaná. Existuje mnoho rôznych prístupov ako vykresľovať grafy, ktoré sa snažia optimalizovať rozmanité kritéria hovoriace o kvalite vykresleného grafu. Paranoia potrebuje vykresľovať orientované grafy. V tejto oblasti vykresľovania grafov, už existuje relatívne dosť prístupov ako vykresľovať orientované grafy. Z pomedzi všetkých sme sa rozhodli použiť *Sugiyamavov Framework*. Tento framework spadá do kategórie *Hierarchického vykresľovania*, ktoré sa veľmi často využíva práve na kreslenie orientovaných grafov kvôli jeho vlastnosti grafického zvýrazňovania orientácie hrán. Náš najväčší problem, s ktorým sa stretneme pri *Sugiyamavom Frameworku* je rozhodnutie sa, čo budeme chápať za vrcholy v algoritmoch frameworku. Ukážeme, že v podstate máme len jednu možnosť. Následky tohto rozhodnutia ale spôsobia, že algoritmy frameworku si budeme musieť prispôsobiť.

V kapitole 2 si predstavíme iné využitie grafov ako je to matematické respektíve informatické. V kapitole 2 si taktiež vymenujeme základné spôsoby vykresľovania a pomenujeme základné kritéria, na ktoré sa prihliada pri kreslení grafov.

V kapitole 3 si predstavíme Open Community Runtime, povieme o myšlienke skrývajúcej sa za celým projektom. Vysvetlíme si, ako Open Community Runtime funguje.

Vymenujeme jednotlivé objekty a vysvetlíme, ako sa z týchto objektov vytvorí graf, ktorý kontroluje nástroj Paranoia a my ho chceme vizualizovať.

V kapitole 4 si povieme niečo viac o hierarchickom kreslení a na aké grafy sa používa. Následne si v tejto kapitole predstavíme *Sugiyamavov Framework*. Povieme si o jednotlivých jeho krokoch a algoritmoch, ktoré ho implementujú. Ukážeme problémy, ktoré sa vyskytli v súvislosti s implementáciou týchto algoritmov a predstavíme naše heuristiky, najdôležitejší krok frameworku.

V kapitole 5 si povieme o podrobnostiach implementácie našeho vizualizátora. Aký vstup dostávame z Paranoie. Ukážeme si ako vyzerajú naše objekty predstavujúce graf. Vymenujeme možnosti, ktoré umožňuje náš nástroj používateľovi, aby mu zjednodušil analýzu grafu a predstavíme naše ambície do budúcnosti, ako by sme chceli vylepšiť náš nástroj.

Kapitola 1

Potrebné definície z teórie grafov

V tejto kapitole si zadefinujeme základné pojmy z teórie grafov, čo je graf, podgraf, orientovaný graf, cyklický graf, kostra grafu a ďalšie pojmy, ktoré budeme používať v texte.

Definícia 1 *Nech E je systém dvojprvkových podmnožín konečnej množiny V. Usporiadanú dvojicu $G = (V, E)$ nazývame* ***graf****. Prvky množiny V nazývame vrcholy a prvky množiny E nazývame hrany grafu G.*

Definícia 2 *Hovoríme, že vrchol v je* ***incidentný*** *s hranou e, ak je jedným z koncových vrcholov hrany e. Taktiež hovoríme, že hrana spája vrcholy, ktoré sú s ňou* ***incidentné****.*

Definícia 3 ***Podgraf*** *je graf, ktorý vznikne z pôvodného grafu vymazaním niektorých jeho vrcholov a všetkých hrán incidentných k týmto vrcholom, poprípade ešte vymazaním ďalších jeho hrán.*

Definícia 4 *Nech je E systém usporiadaných dvojíc konečnej množiny V. Usporiadanú dvojicu G = (V, E) nazývame* ***orientovaný graf****.*

Definícia 5 *Ak graf obsahuje uzavretú postupnosť prepojených vrcholov, tak sa nazýva* ***cyklický graf****.*

Inými slovami, graf je cyklický, ak existuje cesta s aspoň jednou hranou z vrcholu do toho istého vrcholu.

Definícia 6 ***Strom*** *je neprázdny súvislý graf (z každého vrcholu sa vieme dostať do všetkých ostatných vrcholov), pre ktoré platí, že z každého vrchola vedie práve jedna cesta do každého ďalšieho vrchola.*

Definícia 7 ***Kostra grafu*** *je ľubovolný podgraf grafu G na množine všetkých jeho vrcholov, pre ktorý platí, že medzi každými dvoma vrcholmi existuje práve jedna cesta.*

Pre algoritmy, ktoré budeme prezentovať neskôr, sú potrebné aj tieto ďalšie pojmy ako je sink, source, tranzitívna hrana a ďalšie.

Definícia 8 *Nech G = (V, E) je orientovaný graf a v ∈ V, tak ak pre žiadne u ∈ V neexistuje hrana (v ,u), tak potom v sa volá* ***sink***

Definícia 9 *Nech G = (V, E) je orientovaný graf a v ∈ V, tak ak pre žiadne u ∈ V neexistuje hrana (u ,v), tak potom v sa volá* ***source***

Definícia 10 *Nech $G = (V, E)$ je orientovaný graf a hrana $(u, v) \in E$, tak takáto hrana sa nazýva* ***tranzitívna hrana****, ak existujú vrcholy $u, q_1, q_2, ..., q_{n-1}, q_n \in V$ pre ktoré platí $(u, q_1), (q_1, q_2), ..., (q_{n-1}, q_n), (q_n, v) \in E$.*

Inými slovami, je to vrchol, do ktorého vedú dve rôzne cesty, jedna je priama a druhá nepriama.

Definícia 11 ***Tranzitívny uzáver*** *orientovaného grafu $G = (V, E)$ je nadmnožina hrán E_c grafu G, taká že ak $\forall u \in V\ \exists v, q \in V,\ \ (u, v) \in E \wedge (v, q) \in E \Rightarrow (u, q) \in E_c$.*

Definícia 12 ***Tranzitívna redukcia*** *orientovaného grafu G je najmenšia možná množina hrán grafu G, v ktorej majú vrcholy rovnakú dostupnosť ako v jednom z tranzitívnych uzáverov grafu G.*

Kapitola 2

Grafy a ich využitie

Graf je matematický konštrukt. Skladá sa z hrán a vrcholov, kde každá hrana spája práve dva vrcholy. Grafy sú často využívané informatikmi na uchopenie problému a jeho následne riešenie, prípadne jeho zjednodušenie pomocou nástrojov z teórie grafov. Grafy sa taktiež využívajú na vizualizáciu štruktúr a vzťahov medzi nimi. Práve vizualizácia štruktúr je rozhodujúcou zložkou nástrojov pre veľmi veľa aplikácií vo vede a výskume. Ukážkovým príkladom je softvérové inžinierstvo. Žiadny projekt sa nezaobíde bez dobre vymyslenej špecifikácie. V dnešnej dobe sa každá špecifikácia skladá z veľa rôznych grafov a diagramov ako napríklad *use-case* diagramov, *class* diagramov, *aktivity* diagramov a podobne. Ďalším príkladom sú databázové systémy a ich *relačno-entitné* diagramy. Príkladov je naozaj veľa a z rôznych oblastí, nie len v IT. Preto je veľmi prirodzené, že sa veľa informatikov začalo venovať vykresľovaniu grafov a diagramov. Vznikli celé odvetvia prístupov ku kresleniu grafov. Vzniklo veľa kritérií na porovnávanie kvality zobrazených grafov, aby diagramy boli ľahko pochopiteľné. Preto by mali byť zrozumiteľné a dobre čitateľné. Ďalšie príklady: pokrytie telefónnych sietí, *PERT* diagram pre projektový manažment, distribuované výpočty, paralelné architektúry, internetové surfovanie, stavové diagramy, elektronické schémy a mnoho ďalších aplikácií.

2.1 Spôsoby vykresľovania

Bežná konvencia pri kreslení grafov definuje hrany ako úsečky medzi dvomi vrcholmi respektíve krivkami, v niektorých prípadoch definuje aj orientáciu reprezentovanú šípkami. Vrcholy reprezentujeme geometrickými útvarmi ako napríklad kruhy, štvorce alebo zložitejšie útvary obsahujúce aj texty. Keďže grafy využívame aj na grafickú reprezentáciu, tak vzniklo veľa spôsobov ako ich kresliť. Každá metóda kreslenia sa viaže k tomu načo bude slúžiť výsledný obraz. Najčastejšie metódy vykresľovania grafov podľa [2] sú

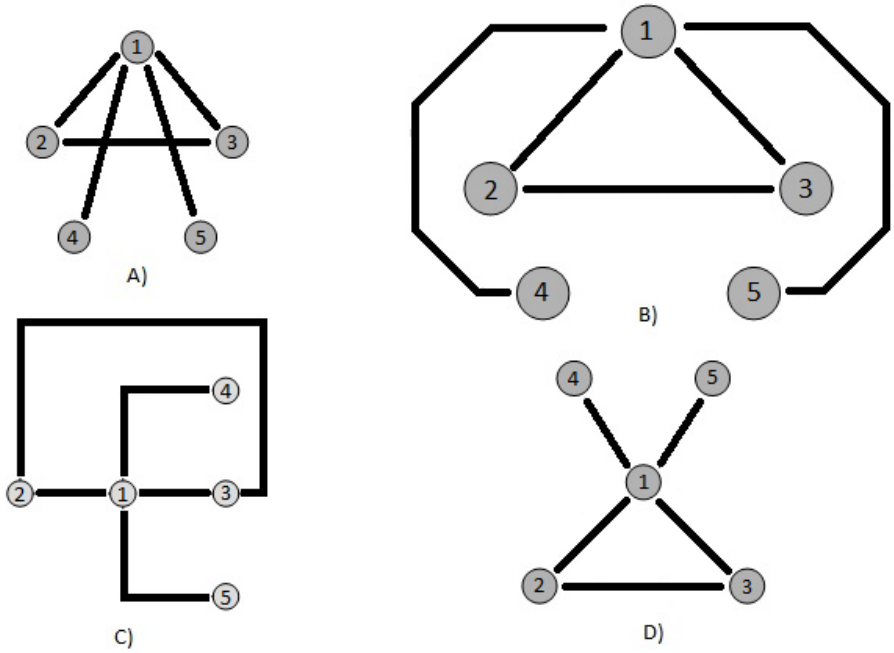
- **Rovné čiary** - Hrany sú úsečky
- **Lomené čiary** - Hrany sú viac krát lomené čiary
- **Ortogonálne čiary** - Hrany pozostávajú z vodorovných a zvislých čiar
- **Planárne kreslenie** - Je to také rozloženie vrcholov a hrán, aby sa žiadne dve hrany nekrížili (nie každý graf sa dá takto nakresliť)
- **Kreslenie kompletných grafov** - zaoberá sa kreslením kompletných grafov (kompletný graf je graf, v ktorom má každý vrchol hranu s každým iným vrcholom)
- **Hierarchické(Vrstevné) kreslenie** - Vytvorené pre kreslenie orientovaných acyklických grafov. Vrcholy sa rozdelia do skupín, vrcholy tej istej skupiny ležia na rovnakej priamke rovnobežnej s osou x
- **Dominantné kreslenie** - Je také rozloženie vrcholov, aby ak existuje cesta z vrcholu u do vrcholu v tak vtedy a len vtedy má vrchol v obe súradnice vyššie ako vrchol u
- **Stromové rozloženie** - Graf je v zakorenenej stromovej formácii. Každý vrchol (okrem koreňa) má rodiča a deti. Rodič sa nachádza nad vrcholom a deti sú pod vrcholom. Používa sa pri vykresľovaní stromovitých štruktúr.
- **Radiálne kreslenie** - Vrcholy sa rozumne rozmiestnia do kružnice tak aby došlo k čo najmenšiemu kríženiu hrán

Na obrázku 2.1 môžeme vidieť rôzne metódy vykresľovania grafov.

Kreslenie lomených hrán ponúka výraznú flexibilitu, keďže čiary nemusia ísť priamo ale môžu sa na príklad vzájomne vyhýbať. Kreslenie hrán pomocou lomených čiar môže graf aj zneprehľadniť, ak to preženieme s počtom zlomov jednotlivých hrán. Okrem toho, systém lomených čiar je podstatne zložitejší ako system rovných čiar. Ak používame body ako reprezentáciu vrcholov, tak potom ortogonálne čiary sú použiteľné iba ak vrcholy majú najviac stupeň štyri.

2.2 Dôležité kritéria

Najdôležitejším kritériom vôbec je typ grafu. Predsa nechceme všeobecný graf vykresľovať pomocou algoritmov na kreslenie orientovaných grafov alebo planárny graf pomocou kompletných grafov.



Obr. 2.1: na obrázku môžeme vidieť rôzne metódy vykresľovania: A) Priame čiary, B) Lomené čiary, C) Ortogonálne čiary, D) Planárne kreslenie

Je dôležité, aby grafy mali čo najjednoduchší tvar, aby boli čo najlepšie čitateľné. Ak chceme, aby výsledný graf bol čitateľnejší a použiteľný, musíme dodržiavať určité kritéria. Medzi najdôležitejšie patria kritéria hovoriace o minimálnych vzdialenostiach objektov grafu, ktorý kreslíme.

- **Minimálna** vzdialenosť medzi dvomi vrcholmi
- **Minimálny** uhol medzi susediacimi hranami
- **Minimálna** vzdialenosť medzi vrcholom a neincidentnou hranou

Okrem vyššie spomenutých kritérií sa prihliada aj na estetické kritéria, ktoré sa dajú len ťažko zmerať, ako je napríklad symetria. Pre kreslenie grafov bolo definovaných veľa rôznych estetických vlastností s cieľom nájsť objektívne ohodnotenie použiteľnosti rôznych metód kreslenia. Toto ohodnotenie nám napomáha vybrať správnu metódu vykresľovania. Pretože rôzne metódy vykresľovania priamo ovplyvňujú ako bude graf vyzerať. Avšak ak chceme dbať na veľa týchto kritérií, tak takýto graf bude príliš ťažko nakresliteľný. Obyčajne ak chceme optimalizovať jedno kritérium, tak sa nám ostatné zhoršia. Preto sa zvyčajne vyberá jedno alebo zopár kompatibilných kľúčových kritérií, ktoré sa budú dodržiavať pre daný problém na úkor ostatných. Medzi časté požiadavky patria:

- Počet kríženia čiar

- Minimalizovanie lomenia čiar
- Pomer dĺžky čiar - pomer najkratšej čiary k najdlhšej čiare
- Symetria
- Rovnomerné rozmiestňovanie vrcholov
- Požiadavky na veľkosť - minimalizovanie rozmerov grafu alebo aká veľká je potrebná mriežka, do ktorej sa vložia vrcholy
- Smerovanie čiar - v niektorý prípadoch chceme zvýrazniť orientáciu hrán, tým že čiary smerujú rovnakým smerom.

Viac informácií o kritériách a ich kvalite môžeme nájsť v prezentácií [14] alebo na stránke [2].

Kapitola 3

OCR (The Open Community Runtime)

V tejto kapitole si predstavíme Open Community Runtime, povieme si o jeho myšlienke a objektoch, ktoré ju implementujú.

Runtime je prostredie, v ktorom beží program. Toto prostredie ponúka programátorovi systémové knižnice a prostriedky, na ktoré runtime dohliada.

The Open Community Runtime alebo OCR je runtime určený pre veľmi výkonné počítače ako napríklad Exascale počítače (sú to počítače, ktoré dokážu vykonať aspoň 10^{18} plávajúcich operácii za sekundu - FLOPS). Takéto počítače sa vyznačujú veľmi veľkým množstvom komponentov. Preto programové modely pre tento typ počítačov musia spĺňať viacero mimoriadnych požiadaviek ako napríklad:

- Schopnosť vyjadriť rádovo milión závislostí.
- Schopnosť výpočtu napredovať k užitočným výsledkom i napriek zlyhaniu niektorého systémového komponentu.
- Schopnosť výpočtu dynamicky sa prispôsobovať rôznemu výkonu a spotrebe energie.

Základnou myšlienkou skrývajúcou sa za OCR je reprezentácia výpočtu orientovaným acyklickým grafom úloh na blokoch dát. Vykonávanie týchto úloh je riadené pomocou udalostí. Samotný mechanizmus runtimu spočíva v myšlienke, že každá úloha je inicializovaná a zaradená medzi úlohy pripravené na spustenie, ak všetky požiadavky, od ktorých daná úloha závisí sú splnené. Je zaručené, že sa určite raz dostane na rad a vykoná sa. Po ukončení behu úlohy sú upozornené udalosťou všetky ďalšie úlohy, závislé na skončení tejto úlohy.

3.1 OCR objekty

OCR objekt je entita spravovaná pomocou OCR. Každý objekt má svoje jedinečné identifikačné číslo ID, ktorým sa dá jednoznačne identifikovať. OCR objekty zohrávajú veľkú úlohu, pretože každý OCR program je definovaný pomocou nich. OCR má tri základné typy objektov:

- **Úloha spustená udalosťou (EDT)** - Jednotka práce v OCR programe.
- **Data blok (DB)** - Súvislý kus pamäte prístupný všetkým OCR objektom, ktoré máju naň smerník.
- **Udalosť** - Objekt určujúci závislosti medzi dvomi objektami. Jeho úlohou je spúšťanie EDT.

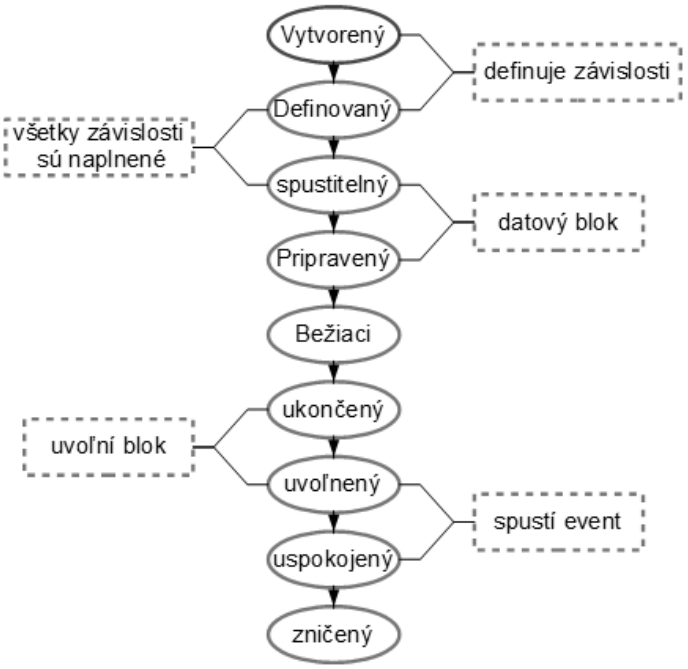
3.1.1 Úloha spustená udalosťou - EDT

Úloha spustená udalosťou (EDT) je objekt slúžiaci ako kontajner pre časť kódu, ktorú chceme vykonať.
EDT si môžme predstaviť ako ľubovolný objekt majúci jednu neblokujúcu metódu, ktorá čaká na svoje zavolanie. EDT počas svojho života prechádza niekoľkými stavmi. Prvým stavom je stav "vytvorený". V tomto stave sa objekt vytvorí, je mu pridelené ID a procedúra, ktorá sa má vykonať. V ďalšom stave sa definujú a nalinkujú závislosti hovoriace o prerekvizitách na beh procedúry. Dostáva sa do stavu, v ktorom čaká na splnenie všetkých skôr definovaných podmienok. Keď sa splnia, tak si vypýta DB, odpovedajúci operačnej pamäti, v ktorej pobeží. Pokračuje v stave "spustiteľnýä čaká na spustenie procedúry. Po skončení behu vráti návratovú hodnotu a uvoľní DB. V poslednom stave sa spustí udalosť oznamujúca ukončenie procedúry kvôli prípadným EDT čakajúcim na koniec tejto procedúry. Objekt potom uvoľní všetky svoje zdroje a zanikne. Na obrázku 3.1 môžeme vidieť diagram stavov EDT.

Práca vykonaná v EDT je definovaná procedúrou. Táto procedúra okrem iného obsahuje informáciu o počte závislostí ako sú: potrebné bloky dát a zoznam udalostí, ktoré musia nastať pred tým ako sa spustí procedúra. Návratová hodnota EDT je ID objektu alebo NULL.

3.1.2 Data blok

Data blok (DB) je OCR objekt, ktorého úlohou je uchovávať premenné, smerníky a iné dáta. Je to jediný spôsob ako uchovávať dáta mimo EDT a zároveň to je jediný spôsob ako zdielať dáta medzi jednotlivými EDT. EDT môže získať DB jedine tak, že požiada o jeho vytvorenie alebo formou prerekvizity (napr. ako parameter). K DB



Obr. 3.1: prechody stavov EDT

môžu pristupovať aj viaceré objekty súčasne, jedná sa teda o zdielanú pamäť, ktorú si programátor musí ustrážiť.

3.1.3 Udalosť

Udalosť je riadiaci OCR objekt, ktorého hlavnou úlohou je koordinovať a synchronizovať aktivity ostatných objektov. Čiastočne určuje poradie, v ktorom sa majú EDT vykonávať.
Objekt udalosť má list referencií na objekty čakajúce na signál a spúšťajúce pravidlá, ktoré hovoria kedy sa má udalosť aktivovať a začať signalizovať pripojeným objektom, že už nastala situácia, ktorú mala udalosť ohlásiť.
Udalosť je taktiež stavový objekt, jeho stavový diagram môžeme vidieť na obrázku 3.2.

Aktuálny problém vrámci OCR súvisí so synchronizáciou udalostí. Udalosti sa spúšťajú, ak sa naplní podmienka na ich spustenie a práve pri tom nastávajú chyby. Synchronizácia je riešená pomocou semaforov. Pribúdajúce závislosti pričítajú do semaforu jednotku. Každá splnená závislosť naopak odčíta z neho jednotku. Keď sa semafor dostane na nulu, tak udalosť sa spustí. Problém nastáva, keď sa semafor dostane na nulu skôr než sa odčítajú všetky závislosti. Preto je dôležité správne určiť poradie, v ktorom sa majú jednotlivé závislosti riešiť. Doterajšie riešenie kontroly tohto problému je tzv. prehľadávanie s návratom. Čo je ale veľmi pomalé riešenie.

Obr. 3.2: prechody stavov udalosti

3.2 Tvorba grafu reprezentujúceho OCR výpočet

OCR program je definovaný ako orientovaný graf, kde stavy EDT a udalostí sú vrcholmi grafu a prepojenia medzi nimi predstavujú hrany grafu. Tento graf je podľa špecifikácie [13] acyklický. Nemôže sa stať, aby EDT A čakal na skončenie EDT B a ten čakal na skončenie objektov, ktoré čakajú na A. Takéto uviaznutie nemôže nastať, bolo by to porušenie špecifikácie a teda považované za logickú chybu programu.

Takáto reprezentácia programu nám dovoľuje rozbiť výpočet na malé časti, ktoré sa môžu vykonávať paralelne, čo veľmi pomôže pri optimalizovaní rýchlosti behu programu alebo spotreby energie. Taktiež sa runtime ľahko spamätá z prípadného výpadku systémového komponentu. Stačí, aby zopakoval bežiaci EDT na inom komponente a rozložil prácu pokazeného komponentu na ostatné.

Ďalšie informacie o Open Community Runtime možno nájsť v špecifikácií od Tima Mattson a Romaina Cledat [13] alebo na wiki stránkach [1].

3.3 Paranoia

V tejto kapitole si povieme niečo o nástroji Paranoia. Ako môžme vidieť v kapitole 3, OCR program je významný tým, že obsahuje veľké množstvo závislostí medzi objektami. Preto je pochopiteľné, že vznikol softvér, ktorý sa snaží eliminovať prípadné chyby ako sú napríklad chyby z nepozornosti alebo základné logické chyby (cyklické čakanie a podobné).

Paranoia je užitočný program pre programátorov začínajúcich pracovať s OCR, ale

prax ukázala, že sa zíde aj zbehlým programátorom. Nástroj predstavuje niečo podobné, ako keď nám programovacie prostredie červeno podčiarkne nekonečný cyklus v zdrojovom kóde alebo pokus o zavolanie metódy na objekte, ktorý ešte nemusel byť inicializovaný. Ide o program, ktorý na vstupe dostane OCR program reprezentovaný grafom. Program analyzuje či graf je validný. Ak nie, tak graficky znázorní chyby, ktorých sa programátor dopustil. Napríklad, či po ukončení EDT funkcie nezabudol objekt zničiť alebo objekt mal byť nechtiac spustený pred jeho vytvorením, respektíve po jeho zničení a mnoho ďalších iných prípadov.

Po dokončení analýzy program vyhodnotí graf. Či je validný alebo nie. Následne analyzovaný graf nakreslí. Nástroj Paranoia na vykresľovanie grafu aktuálne používa program DOT.

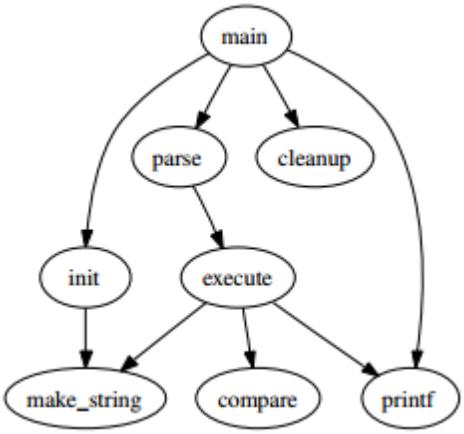
3.3.1 DOT

DOT je softvér na kreslenie orientovaných grafov. Komunikuje prostredníctvom príkazového riadku. Je tiež vizualizačná služba pre web alebo je použiteľný s kompatibilným grafickým rozhraním. Vstupom pre DOT je textový súbor, výstupom je súbor vo formáte GIF, PNG, SVG, teda obrázok. Jednoduchý príklad môžeme vidieť na obrázku 3.3.

DOT používa rovnaký prístup kreslenia grafov ako väčšina hrierarchicky vykreslovacích programov založených na prácach od Sugiyama [3] a Warfield [15]. Myšlienka algoritmu rozmiestňovania vrcholov spočíva v hierarchickom usporadúvaní. Algoritmus na kreslenie potrebuje, aby bol graf acyklický. Prvým krokom algoritmu je potreba rozbiť všetky cykly v grafe tím, že sa otočia hrany spôsobujúce cykly. V druhom kroku sa každému vrcholu pridelí úroveň, ktorá určuje Y-novú súradnicu daného vrchola. V treťom kroku sa určuje poradie vrcholov v rámci každej úrovne tak, aby sa minimalizoval počet prekrížených hrán. Posledným krokom je nastaviť X-ovú súradnicu tak, aby boli hrany čo najkratšie. Tento prístup ku kresleniu orientovaných grafov je veľmi rýchly a účinný.

Program DOT je silný nástroj. Dáva možnosť veľkého množstva nastavení, akými sú definovanie podgrafov, štruktúrovaných vrcholov a mnoho ďalších. Určite je vhodný pre kreslenie UML diagramov. Ale pre program Paranoia sa DOT nehodí. Drvivá väčšina OCR grafov nie je planárna (nedá sa nakresliť bez prekríženia hrán), čo znamená, že čitateľnosť takýchto grafov je rádovo nižšia. Niektoré grafy sú až tak zložité, že neexistuje dostatočne rýchly algoritmus na ich primerane pekné vykreslenie. Preto nie je veľmi rozumné používať program, ktorého výsledok je už nemenný obrázok. Je uži-

```
1:  digraph G {
2:      main -> parse -> execute;
3:      main -> init;
4:      main -> cleanup;
5:      execute -> make_string;
6:      execute -> printf
7:      init -> make_string;
8:      main -> printf;
9:      execute -> compare;
10: }
```



Obr. 3.3: vstup a výstup programu dot, obrazky sú z príručky [7]

točnejšie, aby používateľ vedel editovať graf aj po jeho vykreslení. Bolo by napríklad lepšie, keby bol používateľ schopný odsunúť v danú chvíľu všetky nepotrebné časti grafu alebo ich minimalizovať, ak je graf príliš nečitateľný.

DOT je program na kreslenie všeobecných orientovaných grafov. Robí to dobre, ale Paranoia potrebuje program, ktorý rozmiestňuje jednotlivé vrcholy podľa ich logického zaradenia, čo DOT nedokáže. On sa len snaží o nekríženie a skracovanie hrán. Preto nie je dobrou voľbou pre nástroj Paranoia aj napriek naozaj dobre odladenému algoritmu na kreslenie.

Pre tých, ktorí hľadajú program na kreslenie grafov odporučame DOT ako užitočnú aplikáciu. Syntax jeho vstupu je jednoduchá a zrozumiteľná. V priručke *Drawing graphs with dot* [7] nájdete všetko potrebné na jeho používanie.

Našim cieľom bude naprogramovať nástroj, ktorý by dokázal nahradiť DOT. V prvom rade náš nástroj bude program špeciálne prispôsobený pre Paranoju. Teda z Paranoje pôjde výstup pre náš program. Náš program zo vstupu vytvorí graf, ktorý potom nakreslí. Následne si používateľ bude môcť graficky upraviť a teda zjednodušiť graf podľa potreby. Do budúcnosti by náš nástroj mohol byť schopný používateľovi dovoliť v grafickom rozhraní opravovať prípadné chyby grafov.

Kapitola 4

Hierarchické kreslenie grafov

Pre kreslenie OCR grafov sme sa rozhodli použiť Hierarchické kreslenie grafov [12]. Kvôli tomu, že bol vytvorený na vykresľovanie orientovaných grafov a kvôli jeho zvýrazňovaniu následností v grafe. Veľmi pekne je na týchto grafoch vidieť, že ak sa chceme dostať do vrcholu v tak musíme prejsť cez konkrétne vrcholy. Čo bolo hlavným dôvodom pri vyberaní kandidáta pre naše vykresľovanie. Konkrétne sme si vybrali Sugiyamavov Framework. Tento framework je veľmi populárny medzi programami na vykresľovanie orientovaných grafov. Počas implementácie nám tento framework spôsoboval problémy lebo bohužiaľ tento framework nepočíta s podgrafmi, ktoré predstavujú naše objekty ako EDT alebo udalosti. Preto sme museli jednotlivé kroky alternovať a modifikovať. V tejto kapitole si povieme o Sugiyamovom Frameworku, predstavíme si jednotlivé kroky frameworku, algoritmy, ktoré ich riešia a ukážeme naše heuristiky.

Vo veľa prípadoch orientovaný graf reprezentuje hierarchiu. Preto chceme, aby bol graf nakreslený tak, aby ju bolo vidieť. Predstavme si hierarchiu ako orientovaný acyklický graf, ktorého vrcholy sú rozdelené do disjunktných vrstiev. Príkladmi hierarchií alebo veľmi blízko k nim majú *PERT* diagramy pre projektový manažment, *aktivity* diagramy alebo *use-case* diagramy zo softvérového inžinierstva. Veľmi podobné k hierarchickým grafom sú radiálne kreslenie a kružnicové kreslenie. V radiálnom kreslení sa vrcholy rozmiestňujú na sústredné kružnice. Tieto kružnice predstavujú niečo podobné ako vrstvy v hierarchickom kreslení. Všetky tri vyššie spomenuté metódy majú spoločne, že vzťahy medzi vrcholmi chceme graficky zvýrazniť. Potrebujeme ukázať, že objekty sú vo sťahu nielen so susednými vrcholmi ale aj s ostatnými. V našom prípade tým vzťahom je následnosť vrcholov, teda chceme ukázať, ktoré objekty OCR programu sa musia spustiť skorej ako ostatné. Dosiahneme to tým, že cesty v grafe smerujú jedným smerom. Zvyčajne ide o smer z vrchu dole a z ľava do prava ale smery sa môžu alternovať podľa typu grafu, ktorý sa má kresliť.

Di Battista v štúdii [4] sa zaoberal algoritmami na kreslenie orientovaných grafov. v tejto štúdií porovnával dve veľké skupiny algoritmov. Na jednej strane to boli hierarchické algoritmy a na druhej algoritmy založené na mriežkach. Záverom štúdie bolo, že hierarchické kreslenie je lepšie ako to mriežkové v predchádzaní kríženia sa hrán a celkovej čitateľnosti grafov.

Čo sa týka grafov, ktoré sú len čiastočne hierarchické, tak je možné použiť túto metódu kreslenia. Samozrejme nefunguje to vždy. Keď si zoberieme náhodný graf, tak nás výsledok nepoteší. Lebo nedokážeme rozumne rozdeliť vrcholy do vrstiev. Čo nám spôsobí, že graf bude vyzerať chaoticky.

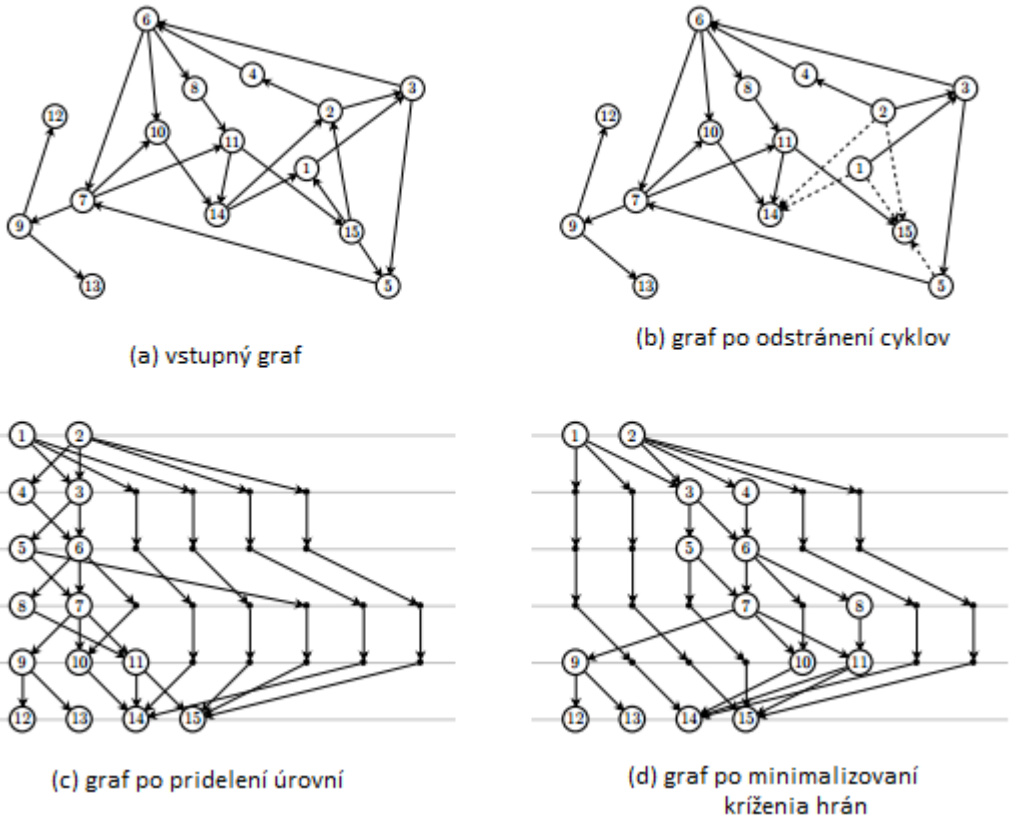
V ďalšej časti si povieme o konkrétnom prístupe k hierarchickému kresleniu. Sugiyama vytvoril framework s hierarchickým prístupom na kreslenie orientovaných grafov. Jeho prístup je veľmi úspešný. Svoju popularitu má vďaka efektivite a rýchlosti algoritmov. Dokonca je použiteľný aj na neorientované grafy. Stačí každej hrane pridať smer. Treba to však spraviť rozumne, musíme si dať pozor nech graf zbytočne nezacyklíme.

4.1 Sugiyamavov Framework

Sugiyamavov Framework kreslí orientované grafy (môžu obsahovať aj cykly). Motiváciou Sugiyamavho frameworku je optimalizovať niekoľko estetických vlastností grafu, ktoré robia graf čitateľnejší. Jednotlivé kroky Sugiyamovho frameworku priamo riešia tieto estetické myšlienky:

- Hrany by mali smerovať jedným smerom
- Krátke hrany sú čitateľnejšie
- Rovnomerné rozloženie vrcholov zabráni vzniku zhlukov vrcholov
- Hrany, ktoré sa nelámu sú lepšie čitateľné

Na obrázku 4.1 je ukázané ako jednotlivé kroky framevorku riešia požadovanú estetickosť grafu. Poprvé sa vyrieši uniformné smerovanie hrán pomocou rušenia cyklov v grafe a to otáčaním hrán spôsobujúcich cykly. Výsledkom je orientovaný graf už bez cyklov obrázok (b), ďalej vyrobíme vrstvy obrázok (c). Taktiež sa dlhé hrany nahradia sériou krátkych hrán s pomocnými vrcholmi. Ďalej spermutujeme vrcholy na každej vrstve tak, aby sme znížili počet krížení hrán ako na obrázku (d).



Obr. 4.1: Na obrázku môžeme vidieť jednotlivé kroky Sugiyamavovho frameworku, obrázok je zo [12]

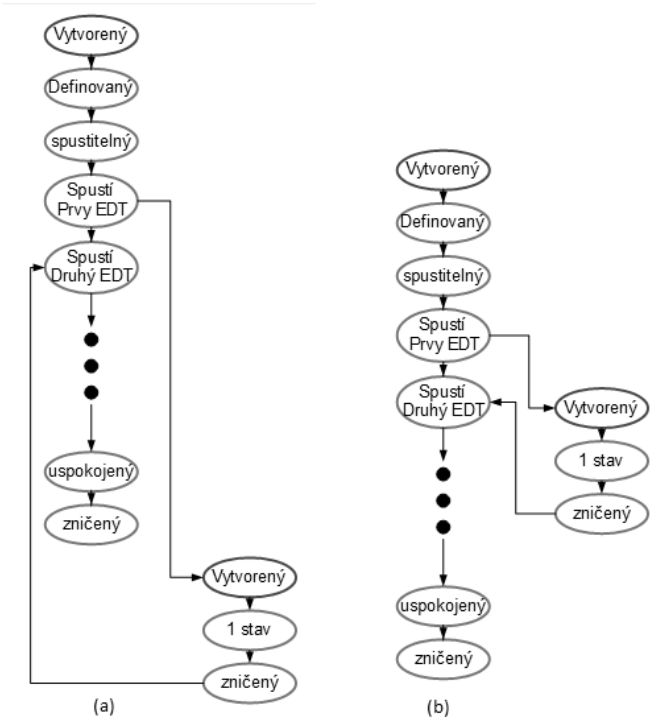
Ako môžeme vidieť Sugiyamavov Framework kreslí orientované grafy pomocou vrstiev vrcholov. Čo je presne to, čo chceme dosiahnuť. Chceme zvýrazniť následnosť jednotlivých EDT a udalostí, ktoré ich spúšťajú. Preto sme sa rozhodli použiť tento framework na vykresľovanie našich grafov predstavujúcich OCR program. V ďalšej časti sa pozrieme konkrétnejšie na jednotlivé kroky, ukážeme si jednotlivé algoritmy a porovnáme ich z ich alternatívami ale ešte predtým si zadefinujeme už veľa krát spomínanú hierarchiu.

Definícia 13 ***Úrovňový graf*** $G = (V, E, \varphi)$ *je acyklický orientovaný graf s funkciou* $\varphi :\rightarrow \{1, 2, ..., n\}, k \geq 1$ *, množina* V *je množina vrcholov, ktorá je zložená* $V = V_1 \cup V_2 \cup ... \cup V_k$ *a* $v \in V_k, \varphi(v) = k$ *pre všetky* k *a zároveň* $V_j \cap V_i = \emptyset$ *pre* $i \neq j$

Definícia 14 ***Hierarchia*** *je úrovňový graf, kde každý vrchol* $v \in V_j$ *kde* $j > 1$ *má hranu (w,v) kde* $\varphi(w) = j - 1$

Skôr ako začneme samotnú implementáciu Sugiyamovho frameworku treba si rozmyslieť, čo budú predstavovať vrcholy vo frameworku. Principiálne máme na výber z dvoch možností. Buď budú vrcholmi frameworku samotné vrcholy (stavy) programu alebo vrcholmi budú objekty programu a to EDT a Udalosť. Prvá možnosť znie jednoducho, ale po rýchlom rozmyslení si uvedomíme, že nám prináša riziko, pretože nám záleží na vzájomných polohách medzi konkrétnymi vrcholmi. Nechceme aby jednotlivé stavy objektov boli príliš roztrúsené po obrázku. Druhá možnosť: Ak zvolíme celé objekty za vrcholy a hrana medzi vrcholmi A, B bude existovať len vtedy ak existuje hrana v programe zo stavu objektu A do stavu objektu B. Táto možnosť vyzerá na prvý pohľad rozumnejšie. Jednotlivé stavy objektov budú blízko seba. Všetko by sa malo dať pekne rozmiestniť a je možné predchádzať kríženiu hrán presne podľa frameworku. Lenže potom si uvedomíme, že jednotlivé vrcholy by mali rôznu veľkosť, takmer všetky objekty sú spustené iným objektom a potom mu aj hlásia svoje skončenie. Takéto krátke cykly by boli problematické. Nie preto, že sú cyklami lebo framework myslí aj na lámanie cyklov v grafe, ale kvôli estetickosti. Predstavme si, že máme veľmi veľké EDT, ktoré volá ďalšie malé EDT, ktoré sa len vytvorí, niečo málo spraví a hneď sa vráti naspať. Toto malé EDT bude nakreslené pod veľkým EDT a budeme mať dve dlhé čiary. Najprv jednu dlhú vedúcu z dlhého EDT do kratšieho EDT a potom jednu vedúcu naspäť do veľkého EDT. Na obrázku 4.2 (a) je znázornený tento problem. Tieto dlhé čiary spôsobujú zneprehľadňovanie grafu a taktiež spôsobujú kríženie hrám, keďže vedú pozdĺž celým podgrafom. My by sme chceli aby malé EDT vystúpilo vyššie a tým skrátilo tieto dlhé čiary, ako môžeme vidieť na obrázku 4.2 (b). Preto sme sa rozhodli, že naša implementácia bude niečo medzi prvou a druhou možnosťou. Za vrcholy grafu si zvolíme objekty, tieto objekty budeme simulovať pomocou podgrafov. Ale na druhú stranu, ak sa chceme vyhnúť problému dlhých čiar, tak zaplatíme tým, že vrcholy

(naše objekty) budú prechádzať cez viacero vrstiev. Presnejšie budú prechádzať cez toľko vrstiev, koľko majú stavov.



Obr. 4.2: Na obrázku (a) je znázornený problém dlhých čiar, na obrázku (b) je naše riešenie

4.2 Rušenie cyklov

Sugiyamavov framework je celý postavený na prideľovaní vrcholom úrovne. Preto graf, ktorý chceme vizualizovať nesmie obsahovať orientované cykly. Keby ich obsahoval, tak nemôžme použiť smer hrán na určenie priority, ktorý vrchol má byť nakreslený vyššie. Preto je prirodzené, že prvým krokom frameworku bude vysporiadanie sa z grafmi, ktoré obsahujú cykly. Zvykne sa predpokladať, že graf neobsahuje orientované cykly dĺžky dva. Graf obsahujúci cyklus dĺžky dva je taký graf, ktorý obsahuje hrany (u, v) a (v, u). Keby ich obsahoval, tak sa jedna z nich na čas odstráni a po pridelení všetkým vrcholom úrovne sa hrana naspäť doplní. Ostatné cykly sa zrušia tak, že sa hranám, ktoré ich spôsobujú zmení smer.

Definícia 15 ***Feedback set (FS)*** *je množina hrán v grafe G, ktorým ak zmeníme smer, tak graf G bude acyklický.*

Hľadanie FS nieje ťažké. Ľubovolné očíslovanie vrcholov rozdeľuje hrany na dve množiny hrán. Prvá bude obsahovať všetky hrany $e = (u, v)$, kde ohodnotenie u je menej

ako ohodnotenie v. Druhá množina bude obsahovať všetky hrany $w = (u, v)$, kde ohodnotenie u je väčšie ako ohodnotenie v. Obe tieto množiny sú FS. Nanešťastie obe množiny môžu byť veľké a preto používanie tejto metódy môže byť nevýhodné. Bolo by rozumnejšie hľadať FS s najmenšou kardinalitou.

Definícia 16 *Nech $G = (V, E)$ je graf. Potom množina hrán $F_e \subset E$ taká, že graf $G_0 = (V, E \backslash F_e)$ neobsahuje cykly, se nazýva* ***Feedback arc set (FAS)****.*

Bohužiaľ, hľadanie najmenšej možnej FS je ťažké. Ľahko môžeme vidieť, že každý FS je FAS. Naopak to neplatí, ak máme vrcholy a, b, c a hrany $\{(a, b), (b, c), (a, c)\}$ tak ľahko môžeme vidieť, že FAS je ľubovolná podmnožina hrán grafu ale FS nemôže byť trojprvková podmnožina. I napriek tomu, že nie každý FAS je FS, tak FAS minimálnej kardinality je FS. Preto problém hľadania minimálneho FS je tak isto ťažký ako problém hľadania FAS o ktorom je známe že je NP-ťažký [8].

Ak sa rozhodneme pri implementácií použiť vrcholy OCR programu ako vrcholy frameworku, tak celý krok rušenia cyklov môžme preskočiť lebo graf predstavujúci OCR program je acyklický a teda neobsahuje cykly. Avšak ak sa rozhodneme pri implementácií použiť objekty ako vrcholy frameworku i napriek problému dlhých čiar, obrázok 4.2, tak musíme riešiť rušenie cyklov lebo objekty očividne tvoria cykly.

Keďže hľadanie minimálneho FS je NP-ťažký problém, musíme ho aproximovať. Existuje viacero heuristík na hľadanie FS. Pre framework je použiteľný ľubovolný algoritmus. Prvou možnosťou môže byť náš príklad spomenutý vyššie. Pomocou konkrétnych očíslovaní vrcholov namiesto ľubovolného očíslovania vieme aspoň trochu optimalizovať veľkosť FS množín. Veľmi častým očíslovaním je očíslovanie pomocou prehľadávania do hĺbky.

4.2.1 Prehľadávanie do hĺbky

Patrí medzi najjednoduchšie algoritmy na rušenie cyklov, tento algoritmus je upravené prehľadávanie do hĺbky. Naše upravenie algoritmu spočíva v zapamätaní si cesty, ktorou práve ideme. Čiže algoritmus vo svojej rekurzívnej podobe by vyzeral nasledovne. Na začiatku si pripravíme množinu na pamätanie si už navštívených vrcholov a množinu na pamätanie si cesty. Postupne spúšťame rekurzívnu funkciu na prehľadávanie na všetkých ešte nenavštívených vrcholoch. Rekurzívna funkcia bude vykonávať nasledovné. Skontroluje či sme daný vrchol už nenavštívili, ak áno skončí. Ďalej pre každého suseda skontroluje či daný susedný vrchol už je v množine cesta, ak áno tak našiel hranu spôsobujúcu cyklus. Ak nie, tak sa zavolá rekurzívna funkcia so susedným vrcholom ako parametrom. Tento algoritmus je jednoduchý a priamočiary ale negarantuje žiadnu

Obr. 4.3: **Upravené prehľadávanie do hĺbky**

```
máme pripravené globálne množiny cesta, navstivene, zle − hrany
while E ∩ navstivene ≠ ∅
    vyber v ∈ E\navstivene
    DFS(v)
end while

def DFS(v):
    if v ∈ navstivene
        return
    end if
    cesta ← cesta ∪ {v}
    for u susediace s v
        if u ∈ cesta
            zlhrany ← zle − hrany ∪ {(v, u)}
        else
            DFS(u)
        end if
    end if
    cesta ← cesta − {v}
```

aproximáciu.

4.2.2 Berger-Shor algoritmus

Bol prvým algoritmom, ktorý v polynomickom čase aproximoval minimálny FAS problém z pomerom lepším ako dva v najhoršom prípade.

Majme graf $G = (V, E)$ a nech $E_a \subseteq E$ je množina hrán taka že graf $G_a = (V, E_a)$ je acyklický. $\delta(v)$ budeme označovať množinu všetkých vchádzajúcich aj vychádzajúcich hrán vrchola v. $\delta^-(v)$ budeme označovať množinu všetkých vchádzajúcich hrán do vrchola v a množinu $\delta^+(v)$ budeme označovať ako množinu všetkých vychádzajúcich hrán z vrchola v. Na začiatku algoritmu si inicializujeme prázdnu množinu E_A a začneme po jednom kontrolovať všetky vrcholy grafu G.
Pre každý vrchol $v \in V$ ak $|\delta^+(v)| \geq |\delta^-(v)|$ tak $E_A \leftarrow E_A \cup \delta^+(v)$ inak $E_A \leftarrow$

$E_A \cup \delta^-(v)$.

Časová zložitosť tohoto algoritmu je $O(|V| + |E|)$ Berger a Shor dokázali, že $G = (V, E_A)$ je orientovaný acyklický graf a preto $F = E/E_A$ je FS. Poľahky vidieť, že F je nanajvýš polovicou všetkých hrán, $|F| \leq \frac{|E|}{2}$.

4.2.3 Rušenie cyklov pažravou metódou

Eades a spol [5] našli iný spôsob vyrábania FS. Predstavili algoritmus na rušenie cyklov v orientovanom grafe. Tento algoritmus sa dnes volá Greedy Cycle Removal. Ak si uvedomíme že incidentné hrany k sinkom ani sourocom nemôžu byť súčasťou žiadneho cyklu, tak vieme vylepšiť výsledok Berger-Shor algoritmu. Vieme vytvoriť algoritmus, ktorý vždy nájde FS. Algoritmus má lineárnu časovú zložitosť a aproximáciu aspoň tak dobrú ako Berger-Shor algoritmus. Podobne ako Berger-Shor algoritmus po jednom kontroluje vrcholy. Ale vyrába FS množinu iným spôsobom. Podstata pažravého algoritmu spočíva v usporiadaní vrcholov a následnom vybraní hrán do FS. Vybrané hrany sú tie, ktoré vedú proti usporiadaniu, teda vedú z neskorších vrcholov do skorších. Eades a spol ukázali, že kardinalita FS nájdená algoritmom je nanajvýš $\frac{|E|}{2} - \frac{|V|}{6}$ [5] a pre grafy so stupňom najviac tri je kardinalita FS najviac $\frac{2}{3}|E|$ [5].

Pažravá metóda okrem toho, že nám dáva garanciu $\frac{|E|}{2} - \frac{|V|}{6}$ je pre naše ďalšie heuristiky v podkapitole *Prideľovanie úrovní* 4.3 výhodnejšia oproti prehľadávaniu do hĺbky, Pretože naša heuristika je založená na pravidle: prideľuj úroveň podľa prvého vrcholu. Hrany incidentné k tomuto vrcholu zvyčajne nebývajú vybraté pažravou metódou.

4.3 Prideľovanie úrovní

Keď už máme graf zbavený cyklov môžeme pristúpiť k ďalšiemu kroku Sugiyamovho frameworku. V tomto kroku sa vytvoria disjunktné množiny vrcholov. Tento krok sa využíva vo všetkých hierarchických vykresleniach. Či už vykresľujete podľa Sugiyamu alebo si vyberiete radiálne kreslenie, prípadne sa rozhodnete pre niečo iné. V každom z nich nájdete krok, v ktorom sa prideľujú úrovne. Pretože to, v akej množine sa daný vrchol nachádza bude určovať jeho pozíciu na obrázku. Po pridelení úrovní sa ešte zvyknú pridať do grafu pomocné vrcholy, ktoré napomáhajú estetickým vlastnostiam grafu ako je skracovanie hrán, prekrývanie sa hrán, prechod hrán cez vrcholy a ďalšie.

Ak nemáme žiadne nároky na prideľovanie úrovní, tak nájdenie rozdelenia do úrovní nie je až tak ťažké. Stačí si nájsť kostru grafu a na nej použiť prehľadávanie do šírky alebo hĺbky a pri každom vnorení vrcholu pridať úroveň rodiča zväčšenú o jeden. Ale

Obr. 4.4: **Greedy Cycle Removal algoritmus**

$S \leftarrow \emptyset$
while G je prázdny **do**
    while G obsahuje sink **do**
        vyber sink v a odstráň ho z G
        hrany prichádzajúce do v pridaj do S
    end while
    while G obsahuje source **do**
        vyber source u a odstráň ho z G
        hrany vychádzajúce z u pridaj do S
    end while
    if G nieje prázdny **do**
        vyber taky vrchol w že $|\delta^+(w)| - |\delta^-(w)|$ je maximum
        odstráň w z G
        pridaj w do S_l
    end while

veľmi často sú požiadavky ďalšie na prideľovanie úrovní. Tieto požiadavky sú ťažko dodržateľné, ak ich chceme dodržiavať všetky naraz. Preto sa vyberie jedna alebo zopár kompatibilných požiadaviek, na ktoré sa dbá na úkor ostatných. Asi najčastejším príkladom je minimalizovanie počtu pomocných vrcholov, ktoré by sme museli pridať do grafu. Aby sme si neprotirečili, je dobré ak použijeme pomocné vrcholy ale len na miestach kde ich treba. Tento počet je rozumné držať čo najnižšie, lebo pomocné vrcholy majú tendenciu pribúdať vo veľkom počte. Mať veľa vrcholov v jednotlivých vrstvách nie je rozumné, lebo to výrazne spomaľuje ďalšie kroky frameworku. Okrem toho vďaka tejto požiadavke sa nám môže podariť sprehľadniť výsledný graf. Vyplýva to z toho, že ak máme menej pomocných vrcholov, tak náš graf bude mať kratšie hrany a hrany sa nebudú toľko lámať. Samozrejme existuje viacej estetických dôvodov prečo je dobré minimalizovať počet pomocných vrcholov. Exitujú algoritmy, ktoré riešia problém rozdelenia do vrstiev s minimom pomocných vrcholov v polynomickom čase. Jeho riešenie sa dá simulovať pomocou lineárneho celočíselného programovania. Toto riešenie predstavi Healy a spol [9]. Okrem toho sa tento problém dá prerobiť na Minimum-cost flow problem. Minimum-cost flow problem je optimalizačný problém, v ktorom sa hľadá „najlacnejší“spôsob prepravy toku z vrchola v do vrchola u cez orientovaný graf.

Ďalšou vlastnosťou, ktorá sa dá ovplyvniť vďaka konkrétnemu pristupovaniu k prideľovaniu úrovní je výška a šírka grafu. Vieme zredukovať šírku grafu úplne priamočiarym postupom obmedzovania počtu vrcholov v jednotlivých úrovniach. Na úkor čitateľnosti dokážeme graf zredukovať na ľubovolnú šírku. Na druhú stranu ak nepotrebujeme obmedzovať veľkosť grafu, tak výška a šírka grafu je užitočná pri určovaní mierky, v ktorej budeme chcieť vykresľovať. Pri definovaní šírky grafu sa často ľudia rozchádzajú, na jednej strane sa šírka grafu predstavuje maximálny počet vrcholov v jednej vrstve a na druhej strane maximálny počet vrcholov a pomocných vrcholov v jednej vrstve. My preferujeme druhú možnosť lebo je presnejšia. Ak chceme graf z minimálnou výškou, tak môžeme použiť algoritmus na hľadanie najdlhšej cesty v grafe uvedený v texte nižšie, ktorý to zvládne v lineárnom čase. Tak isto minimalizovanie počtu pomocných vrcholov dokážeme vyriešiť pomocou algoritmu prevyšujúceho vrcholy. Ak potrebujeme graf z ohraničenou šírkou, tak máme problém. Síce ak táto šírka je počítaná bez pomocných vrcholov, tak stačí prideliť každý vrchol do svojej vlastnej vrstvy. Ale ak musíme počitať aj pomocnými vrcholmi tak tento problém je NP-ťažký [12]. To isté platí aj pre graf, ktorý má mať obmedzenú výšku aj šírku [12] a nepomôže nám ani nepočítanie pomocných vrcholov.

V ďalšej časti textu sa pozrieme bližšie na náš algoritmus ale aj algoritmy, ktoré riešia požiadavky uvedené vyššie v texte.

4.3.1 Naše heuristiky na prideľovanie vrstiev

Ako sme už spomenuli vyššie, najjednoduchším spôsobom ako vytvoriť rozdelenie vrcholov grafu do vrstiev je nájsť kostru grafu a na ňu použiť prehľadávanie do hĺbky alebo do šírky s tým, že sa pri každom vnorení pridelí vrcholu úroveň podla typu hrany. Ak sme išli proti smeru hrany, tak sa pridelí o jedno menšia úroveň a ak sme išli v smere hrany, tak sa pridelí o jedno vyššia úroveň. Bohužiaľ tento spôsob nám nezaručuje nič z toho, čo sme si povedali v predchádzajúcej časti.

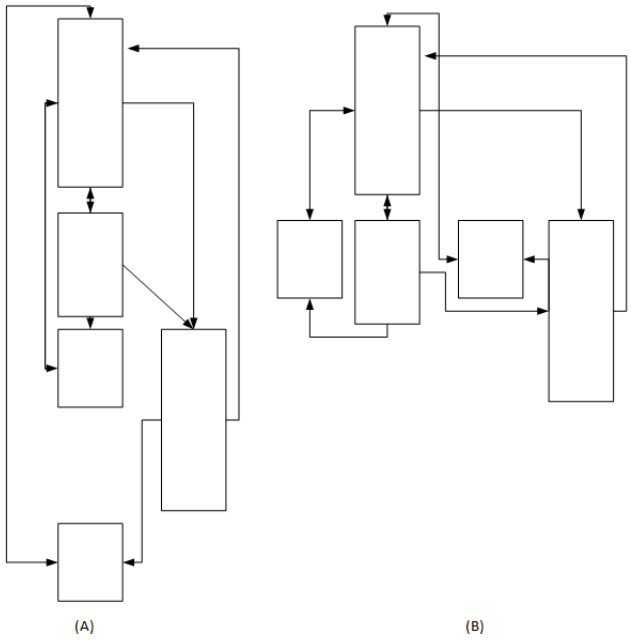
Náš graf, ktorý chceme vizualizovať je atypický lebo obsahuje podgrafy, ktorých vrcholy chceme vykresliť blízko seba. Prvý nápad, ktorý sme mali bol jednoduchý. Zabudnime, že máme podgrafy a priamo rozdeľme jednotlivé vrcholy do úrovní. Toto pomocné rozdelenie nám povie kam budú patriť podgrafy. Riešili sme to pomocou upraveného prehľadávania do šírky s tým, že sme sa vnárali do vrcholov, v ktorých sme ešte neboli alebo do tých, ktoré mali menšiu alebo rovnakú úroveň ako vrchol, v ktorom sa práve nachádzame. Tento druhý prípad, prečo sa vnoriť je dôležitý kvôli tranzitívnym hranám. Aby bolo zaručené, že vrchol incidentý k tranzitívnej hrane dostane najhoršiu možnú úroveň. Po prideľovaní úrovní vrcholom sme sa naspäť vrátili k

podgrafom. Každý podgraf dostal minimálnu úroveň s pomedzi všetkých jeho vrcholov. Táto heuristika vyzerala sľubne. Po naimplementovaní sa ale začali ukazovať problémy. Najzjavnejším problémom bola výška grafu. V grafe vznikali medzery. Čím väčší bol graf, tým sa tvorili väčšie medzery. Na obrázku 4.5 môžeme vidieť graf po pridelení vrstiev. Je na ňom znázornený problém z výškou grafu.

Obr. 4.5: Na obrázku môžeme vidieť graf po pridelení vrstiev prvou heuristikou.

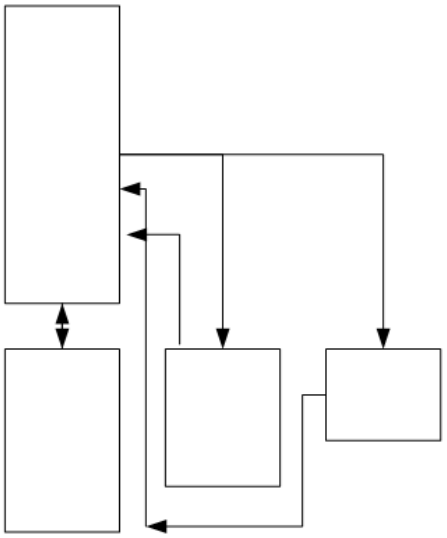
Druhá heuristika, s ktorou sme prišli, bola založená na považovaní podgrafov za vrcholy. Zanedbali sme veľkosť jednotlivých podgrafov a predstavili si ich za rovnako veľké. Ďalej sme chceli použiť jeden z klasických algoritmov na prideľovanie úrovní. Ale nový graf už mal aj cykly. Preto sme použili prehľadávanie do hĺbky na ich rozbitie. Neskôr sme tento algoritmus nahradili podstatne efektívnejšou alternatívou a tou bolo *Greedy Cycle Removal*, ktorý sme predstavili vyššie v texte. Môžeme potvrdiť

markantný rozdiel. Ako môžeme vidieť na obrázku 4.6. Prehľadávanie do hĺbky nám rozdeľovalo cykly podstatne menej intuitívne a spôsoboval veľké zmeny pri vyberaní úrovní. Okrem toho Prehľadávanie do hĺbky negarantuje žiadnu aproximáciu. Čo sa tiež podpísalo na výslednom vykreslení. Keď sme už mali vyriešené cykly, tak sme na grafe mohli spustiť algoritmus na prehľadávanie do šírky, ktorý prideľoval úrovne podobne ako v prvej heuristike. Neskôr sme ho nahradili nami upraveným algoritmom hľadajúcim najdlhšie cesty v grafe, ktorý predstavíme v texte neskôr. Po dokončení algoritmu na priraďovanie úrovní sme jednotlivé vrstvy prispôsobili podgrafom tak, aby sa do nich zmestili (každá úroveň mala veľkosť najväčšieho podgrafu, ktorý obsahovala).



Obr. 4.6: Na obrázku môžeme vidieť výrazné zlepšenie výsledku Greedy Cycle Removal oproti Prehľadávaniu do hĺbky.

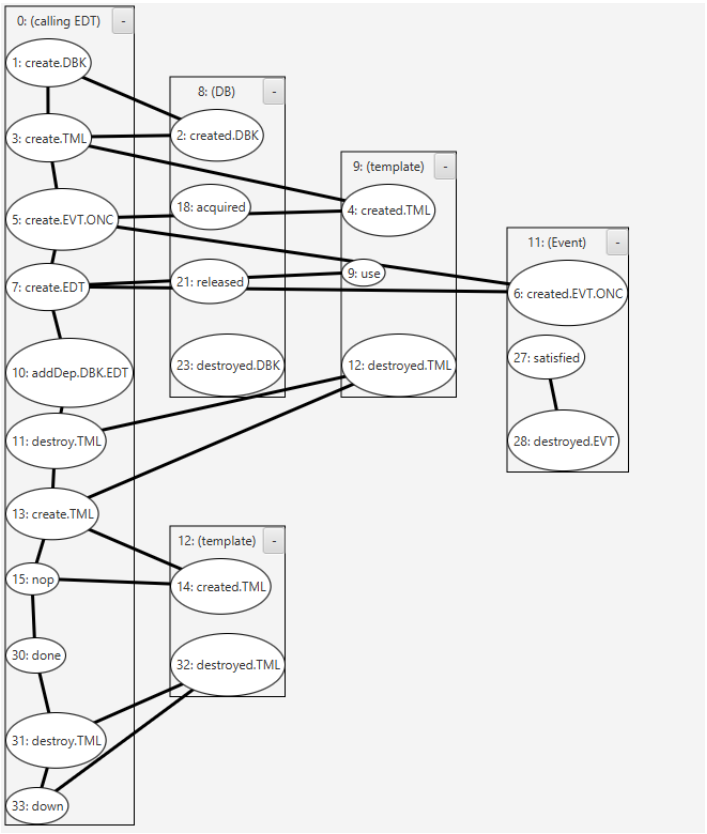
Táto heuristika nám prišla na prvý pohľad rozumnejšia a flexibilnejšia, keďže môžeme použiť algoritmus na priraďovanie úrovní podľa našej potreby (minimalizovať výšku, obmedziť šírku alebo minimalizovať počet pomocných vrcholov). Ale po naimplementovaní a spustení vykresľovania môžeme pozorovať veľké nedostatky. Vrcholom sa síce dobre pridelia úrovne, ale ako môžeme vidieť na obrázku 4.7 a ešte krajšie na obrázku 4.2 hrany, ktoré vedú z vrcholu do vrcholu a naspäť sú problém. Hrany sú zbytočne dlhé. Bohužiaľ takýchto prípadov je v grafoch predstavujúcich OCR program veľa. Tento problém by spôsoboval veľký zmätok a tým by drasticky redukoval čitateľnosť grafu. Tým že medzi jednotlivými podgrafmy je veľa hrán vo veľa prípadoch sú dokonca viacnásobné hrany, tak táto heuristika je nepoužiteľná. Preto táto heuristika dopadla horšie ako predošlá heuristika.



Obr. 4.7: Na obrázku môžeme vidieť náčrt grafu po pridelení vrstiev druhou heuristikou.

Našou treťou a finálnou heuristikou, ktorú sme naimplementovali je návrat k prvej heuristike. Ale vylepšíme ju o vybublinkovanie jednotlivých podgrafov. Takže prvým krokom je priradenie každému vrcholu úroveň pomocou nášho upraveného Algoritmu na hľadanie najdlhších ciest 4.10 (predstavený nižšie v texte). Jednotlivé podgrafy si usporiadame podľa úrovní prvých vrcholov v podgrafoch (nižšia úroveň je skôr). Prechádzame cez podgrafy v zostupnom poradí a priraďujeme im úroveň. Úroveň podgrafu je najnižší rodič prvého vrchola v podgrafe zväčšená o jeden. Okrem prideľovania úrovní podgrafom prepisujeme úrovne aj všetkým vrcholom v podgrafe. Vrcholom je priradená úroveň: Podgraf plus pozícia v ňom začínajúc od nuly. Tento krok prepisovania vrcholov nám zabezpečí vyplávanie podgrafov čo najvyššie ako sa dá. Dynamické prepisovanie úrovní vrcholov si môžeme dovoliť. Za túto vlastnosť vďačíme algoritmu na hľadanie najdlhších ciest. lebo podgrafy, ktoré ešte nemajú pridelenú úroveň sa teoreticky môžu pýtať na úroveň len podgrafov, ktorým už bola pridelená úroveň, keďže prvý vrchol v podgrafe musí mať všetkých rodičov vo vyšších úrovniach. Keby nemal, tak by rodič, ktorý má vyššiu úroveň musel byť neskôr spracovaný ako syn, čo je v rozpore s algoritmom na hľadanie najdlhších ciest.

Táto heuristika spája to dobré z oboch predchádzajúcich heuristík. Z prvej si zobrala priraďovanie úrovní čo najvyššie ako sa dá a z druhej zas možnosť použiť klasický algoritmus na prideľovanie úrovní. Táto heuristika sa snaží vyhýbať problému dlhých čiar a ako môžeme vidieť na Obrázku 4.8 maximalizuje zvýrazňovanie hierarchickosti v grafe.



Obr. 4.8: Na obrázku môžeme vidieť graf po vykreslení pomocou tretej heuristiky.

4.4 Populárne algoritmy na prideľovanie úrovní

Veľká časť heuristík zaoberajúcich sa minimalizovaním výšky pri konkrétnej maximálnej šírke sú postavené na riešení problému viac procesorového plánovača so závislými úlohami. Ide o problém, v ktorom máme množinu úloh $\{u_1, u_2, ...u_n\}$. Každej úlohe je pridelený čas za koľko sa vykoná. Ďalej máme m rôznych procesorov, na ktorých sa majú úlohy vykonávať. Okrem toho sú uvedené závislosti, ktoré hovoria, aké úlohy musí byť splnené predtým ako je možne spustiť danú úlohu. Našim cieľom je nájsť také poradie úloh, aby doba vykonávania úloh bola čo najkratšia. Tento problém je NP-ťažký. Tento problém je ekvivalentný nájdeniu úrovní pre vrcholy s maximálnou šírkou m pre každú úroveň a minimálnou výškou grafu. Ukážeme si dva veľmi často používané algoritmy *Algoritmus hľadajúci najdlhšie cesty* a *Coffman-Grahamou algoritmus* [10].

4.4.1 Algoritmus hľadajúci najdlhšie cesty

Tento algoritmus rieši problém plánovača pre neobmedzený počet procesorov. Rieši sa pomocou hľadania najdlhších ciest v grafe. Respektíve prideľuje sa úroveň vrcholu podľa dĺžky najdlhšej cesty do daného vrcholu Na začiatku si algoritmus zvolí veľké

Obr. 4.9: **Algoritmus hľadajúci najdlhšie cesty**

```
uroven ← 1
A ← ∅
B ← ∅
while A ≠ V
    vyber si vrchol u ∉ A a všetci jeho rodičia patria do B
    if našli sme u
        priraď u aktuálnu úroveň
        A ← A ∪ {u}
    else
        uroven ← uroven + 1
        B ← B ∪ A
    end if
end while
```

číslo predstavujúce dĺžku najdlhšej cesty v grafe. Toto číslo môže byť ľubovolné veľké a budeme ho volať α. Potom každému sinku sa pridelí táto úroveň α. Ďalej sa vyberá vrchol, ktorého všetci rodičia už majú pridelenú úroveň. Tomuto vrcholu je pridelená úroveň najnižšieho rodiča mínus jeden. Tento proces sa opakuje pokiaľ sa všetkým vrcholom nepridelí nejaká úroveň.
Algoritmus: Na začiatku sa inicializujú dve množiny A, B a premenná v ktorej budeme mať aktuálnu úroveň. Na začiatku to bude úroveň jeden. Množina A bude predstavovať všetky už priradené vrcholy a množina B bude predstavovať vrcholy, ktorým sme pridelili nižšiu úroveň ako je aktuálna. Potom si budeme v cykle vyberať vrcholy s pomedzi ešte nevybraných, ktorých rodičia už boli vybraný a všetci rodičia majú úroveň vyššiu ako je aktuálna. Tento cyklus sa bude opakovať pokiaľ sa množina A nebude rovnať množine všetkých vrcholov V.

Môžeme si všimnúť, že tento algoritmus má lineárnu časovú zložitosť, keďže každý vrchol je spracovaný práve raz. Jeho výhodou je jednoduchosť a rýchlosť. Tento Algoritmus nám nájde pridelenie s najkratšou výškou, ale na druhú stranu sa vôbec nezaujíma o minimalizovanie pomocných vrcholov.

Pre našu implementáciu sme si zvolili vyššie opísaný prístup priraďovania vrstiev. Keďže naše vrcholy sú podgrafy s rôznou veľkosťou, tak sme potrebovali algoritmus, ktorý by dokázal pracovať aj s vrcholmi vo vnútri podgrafov, aby objekty mohli vybublať najvyššie ako je možne. Algoritmus sme si museli prispôsobiť 4.10. Na hľadanie ďalších nepridelených vrcholov sme sa inšpirovali prehľadávaním do šírky. Okrem toho takéto rozdeľovanie vrcholov veľmi pekne graficky zvýrazní hierarchickosť grafu, kto-

rého hlavnou úlohou je vizualizácia programu a teda graficky ukázať, ktoré EDT má prednosť pred ostatnými.

Ako prvý krok inicializujeme úrovne vrcholov na mínus nekonečno. Ďalej sa inicializuje prázdna fronta, do ktorej sa budú pridávať vrcholy na spracovanie a pridá sa do nej vrchol reprezentujúci objekt, ktorý spúšťa celý program. Ako príklad si môžeme predstaviť funkciu *main*, ktorá sa zavolá, keď spustíme klasický exe-súbor. Tomuto vrcholu sa pridelí úroveň nula. Potom sa v cykle vyberie vrchol p z fronty a všetkým vrcholom susedným vrcholom s p, ktoré majú menšiu úroveň ako p je pridelená nová úroveň a ak všetci rodičia p sú vybavený, tak p je vložené na koniec fronty. Tento cyklus sa opakuje pokiaľ nie je fronta prázdna.

Obr. 4.10: **Upravený algoritmus hľadajúci najdlhšie cesty**

$fronta \leftarrow [main]$
$main \leftarrow 0$
$vybaven \leftarrow \{main\}$
while fronta nieje prazdna
    $p \leftarrow fronta$
    for q from $p.synovia$
        if $p > q$
            $index \leftarrow$ index prvku z ktorého vedie hrana
            $q \leftarrow p.uroven + 1$
            if všetci rodičia q sú v množine $vybaven$
                $fronta \leftarrow q$
                $vybaven \leftarrow vybaven \cup \{q\}$
            end if
        end if
    end for
end while

Tento Algoritmus je jednoduchý a má tiež výbornú zložitosť. Okrem toho je priamo v ňom zabudované vyhýbanie sa dlhým čiaram.

4.4.2 Coffman–Grahamov algoritmus

Ďalšou možnosťou ako riešiť problém plánovača je Coffman–Grahamou algoritmus. Na rozdiel od algoritmu hľadajúceho najdlhšie cesty, tento algoritmus rieši problém plánovača pre najviac $n < \infty$ procesorov. Keďže tento problém je NP-ťazký, tak ho aproximuje. Algoritmus nám garantuje rozdelenie do vrstiev so šírkou najviac n a výškou najviac $2 - \frac{2}{n}$ krát väčšou ako je optimálna výška. Algoritmus je komplexnejší a s časovou zložitosťou $O(|V^2|)$ [10] je pomalší ako predchádzajúce dva algoritmy. Má aj špeciálne požiadavky. Graf $G = (V, E)$, na ktorý chceme použiť Coffman–Grahamov algoritmus nesmie obsahovať žiadne tranzitívne hrany, alebo ak ich obsahuje tak sa musí vypočítať tranzitívna redukcia grafu a na novom grafe $G_{red}(V, E_r)$ sa môže vykonať algoritmus.

Algoritmus spočíva z viacerých krokov. V prvom kroku je vrcholom pridelené číslo index, podľa ktorého sa vrcholy budú prideľovať v ďalšom kroku (index predstavuje prioritu, čím menšie číslo, tým skôr sa mu pridelí úroveň). Indexy sa prideľujú nasledovne: Všetkým vrcholom nastav index na nekonečno. Ďalej vyberaj po jednom vrchol s indexom nekonečno, ktorého rodičia majú najmenšie indexy (kontroluje sa to lexikograficky, teda ak sa najväčší rovnajú, tak sa porovnávajú druhý najväčší a tak ďalej). V druhom kroku sa vrcholom prideľujú úrovne. Úrovne sa prideľujú tak, aby v každej vrstve bolo čo najviac vrcholov, ale počet neprekročil maximálne povolené množstvo pre jednu úroveň.

Ak nemáme žiadne veľké nároky na graf, tak asi najjednoduchšia rozumná možnosť ako riešiť priraďovanie úrovní je použitie algoritmu na hľadanie najdlhších ciest v grafe. Ak by náš graf mal mať limitovanú šírku, tak by bolo rozumnejšie použiť Coffman-Grahamov algoritmus. Bohužiaľ oba algoritmy majú zlé výsledky, čo sa týka počtu pomocných vrcholov. Existuje jednoduchá heuristika od Tarassovova a spol. [11], ktorá sa dá aplikovať na už hotové rozdelenie vrcholov do úrovní aby zredukovala tento počet pomocných vrcholov. My sme sa ju rozhodli nezahrnúť do našej práce, namiesto nej sme použili princíp vybublinkovania (v tretej heuristike) vrcholov čo najvyššie ako sa dá. Ak by sme nemali problém s dlhými čiarami respektíve naše vrcholy by mali rovnakú dĺžku, tak by sme použili algoritmus od dvojice Healy a Nikolov [9], ktorý je priamo navrhnutý na redukovanie počtu pomocných vrcholov. Považujeme za rozumné znižovať tento počet, lebo potom výsledný graf je jednoduchší. Graf má kratšie čiary, ktoré sa menej lámu.

4.5 Pridávanie pomocných vrcholov

V Sugiyamovom Framework sa po rozdelení vrcholov do úrovní pridajú pomocné vrcholy. Pomocné vrcholy sa pridávajú do grafu, ak susedné vrcholy nie sú v susedných úrovniach, teda nahradíme jednu hranu reťazou viacerých hrán, tak aby v každej úrovni bol jeden vrchol. Pomocné vrcholy sú užitočné lebo vďaka nim je ľahko možné sa vyhnúť kríženiu hrany s vrcholom. Ďalej nám tiež uľahčuje vyhýbanie sa kríženiu hrán.

Keďže naše heuristiky používajú algoritmy, ktoré generujú veľké množstvo medzier medzi vrcholmi, čo by spôsobovalo pridávanie nezanedbateľného počtu vrcholov, tak sme sa rozhodli neimplementovať pomocné vrcholy, lebo by sme znásobili počty vrcholov.

4.6 Zoraďovanie vrcholov

Tento krok frameworku je taktiež veľmi dôležitý, lebo určovanie poradia vrcholov v jednotlivých vrstvách vie veľmi zredukovať počet krížení hrán. Bohužiaľ pre nás, sa nedá použiť jednoduchá heuristika na znižovanie kríženia hrán ako napríklad k už zoradeným vrcholov vyššej vrstvy nájdeme permutáciu vrcholov z nižšej vrstvy takú, ktorá má najmenší počet prekrížení hrán medzi týmito vrstvami. Keďže všetky naše vrcholy sú podgrafy a prechádzajú cez viacero vrstiev, preto skúšanie permutovania takýchto vrcholov by bolo náročne.

Rozhodli sme sa použiť heuristiku na zoraďovanie vrcholov, ktorá sa nezaoberá krížením hrán, keďže nemôžme ľubovolne permutovať podgrafy na každej úrovni samostatne. Naša heuristika sa ale na druhú stranu snaží maximálne možne graficky zvýrazniť tvorbu jednotlivých objektov v OCR programe. Jediné pravidlo heuristiky je: podgraf musí byť nakreslený čo najbližšie na pravo od všetkých rodičov jeho prvého vrchola. Implementovali sme ju pomocou dvojrozmerného poľa, do ktorého sme si zaznačovali pozície, už umiestnených podgrafov. Pričom, ak nemôžeme situovať podgraf hneď vedľa, tak sa ho pokúsime umiestniť na prvú voľnú pozíciu napravo od rodičov. Keď nájdeme vhodnú pozíciu, tak si ju zaznačíme do poľa a upozorníme všetkých susedné podgrafy, že tento podgraf má novú pozíciu. Výhoda tejto heuristiky je jej jednoduchosť implementácie a už spomínané grafické zvýrazňovanie.

Kapitola 5

Implementácia

V tejto kapitole si povieme o implementácii samotného nástroja vizualizujúceho program.

Rozhodli sme sa na implementáciu použiť jazyk Java. Pretože Java skrýva veľa nízko úrovňových aspektov programovania a je multiplatformová (netreba prekompilovať zdrojové kódy, keď chceme preniesť program z jednej platformy na druhú). Na tvorbu grafické rozhrania sme sa rozhodli použiť knižnicu *JavaFx*. V tejto knižnici sa ľahko vyrábajú okienkové aplikácie a dá sa vďaka nej jednoducho kresliť.

5.1 Vstup

Ako prvé si povieme ako náš program dostáva vstup. Naša aplikácia získava údaje z Paranoie v podobe súborov. V týchto súboroch je popísaní graf. Prvý súbor *trace_ ops.dat* obsahuje stavy objektov OCR programu. Tento súbor je v binárnom formáte. Súbor pozostáva len zo zoznamu vrcholov grafu. Čiže si musíme vypočítať koľko vrcholov obsahuje z dĺžky súboru. Každý vrchol je reprezentovaný 64 bajtami:

1. Osem bajtov predstavuje jedinečný identifikátor. Tento identifikátor je jedinečný pre celý súbor.
2. Šestnásť bajtov predstavuje názov, respektíve druh stavu. Je zakódovaný v ascii, končiaci nulou.
3. Osem bajtov tvorí identifikátor objektu. Toto číslo hovorí, ktoré stavy patria, ku ktorým objektom.
4. 4 x Osem bajtov predstavujúcich doplňujúce argumenty. Pre našu prácu nie sú dôležité.

Druhý súbor *trace_ edges.dat* hovorí o hranách grafu. Súbor je taktiež v binárnom formáte. Obsahuje len zoznam hrán medzi vrcholmi zo *trace_ ops.dat* súboru. Jedna hrana je popísaná v šestnástich bajtoch:

1. Osem bajtov predstavuje jedinečný identifikátor vrcholu, z ktorého vychádza hrana.
2. Osem bajtov predstavuje jedinečný identifikátor vrcholu, do ktorého vedie hrana.

5.2 Grafové štruktúry

Naša aplikácia počas čítania vstupných dát (najprv sa načítajú vrcholy potom hrany) hneď vytvára dátové štruktúry reprezentujúce graf a priamo do nich zapisuje vstupné hodnoty. Principiálne máme tri typy objektov (štruktúr). Objekty graf, podgraf a vrchol.

1. **Graf** je hlavný objekt. Je iba jeden. Obsahuje referencie na podgrafy a vrcholy. Jeho hlavnými metódami sú pridaj vrchol a pridaj hranu do grafu. Jeho úlohou je správne poskladanie grafu. To znamená, že vytvára podgrafy a vytvára a vkladá vrcholy do správnych podgrafov. Ďalej má tento objekt referenciu na hlavnú kresliacu plochu, na ktorej bude zobrazený celý graf.

2. **Podgraf** je objekt predstavujúci inštanciu OCR objektu v grafe. Jeho úlohou je vykresľovanie konkrétnych OCR objektov. Obsahuje metódu pridaj vrchol. Táto metóda pridá vrchol do podgrafu. Podgraf dostáva v konštruktore referenciu na hlavnú kresliacu plochu. Objekt si vytvorí svoju plochu a tú pridá do hlavnej plochy, na túto svoju plochu si podgraf bude kresliť vrcholy (stavy), umiestňuje ich pod seba podľa typu vrcholov. Podgraf vieme posúvať na hlavnej ploche pomocou metód nastav pozíciu.

3. **Vrchol** je objekt predstavujúci inštanciu stavu OCR objektu. Tento objekt dostáva referencie na svojich susedou a rodičov pomocou metód pridaj suseda a pridaj rodiča. Objekt v konštruktore dostáva referenciu na kresliacu plochu, do ktorej sa pridá útvar podľa typu vrchola (plnohodnotný vrchol je znázornený elipsou s textom a pomocný vrchol je iba malí prázdny štvorček). Objekt vieme posúvať na ploche pomocou metód nastavujúcich x-ovú a y-ovú súradnicu. Každý vrchol zabezpečuje vykresľovanie hrán, ktoré z neho vychádzajú.

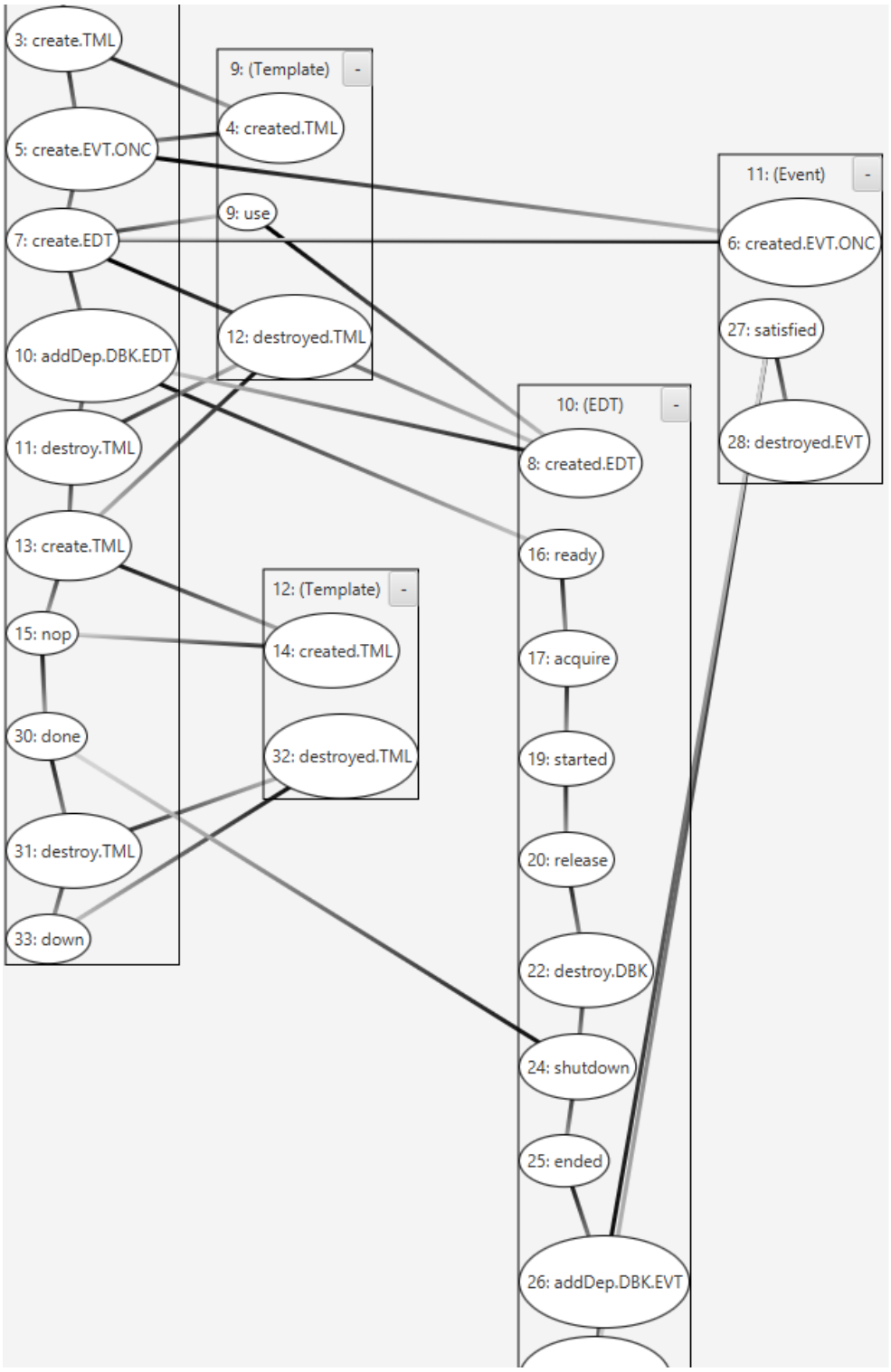
Po načítaní vstupu do dátových štruktúr nasleduje rozmiestňovanie podgrafov na konkrétne pozície. Na tento úkon použijeme jednu z troch našich heuristík na prideľovanie úrovní, spomínanú vyššie v texte. Po rozmiestnení sa graf vykreslí. Na vykresľovanie nám poslúži *JavaFx*, ktorá za nás kreslí a posúva geometrické útvary. Jediné čo potrebujeme spraviť je, držať si referencie objektov predstavujúce časti grafov a následne ich vložiť na správne plochy a nastaviť im ich pozície. O všetko ostatné sa postará *JavaFx*.

5.3 Funkcionalita

Naša aplikácia sa oproti DOTu líši tým, že nekreslí statické obrázky ale umožňuje používateľovi modifikovať vykreslenie grafu v grafickom rozhraní. Aplikácia dovoľuje používateľom posúvať podgrafy podľa ich vôle. Keďže sme neimplementovali pomocné vrcholy v našej heuristike, kvôli ich potenciálne veľkému množstvu, rozhodli sme sa dať používateľovi možnosť pridať pomocné vrcholy manuálne podľa vlastných požiadaviek. Pomocný vrchol sa pridá do grafu rozdelením hrany, kliknutím pravého tlačítka na myši na ľubovolnej hrane.

Keďže grafy OCR programu sú veľmi komplikované, tak sme sa rozhodli pridať používateľom možnosť minimalizovať podgrafy do jedného vrcholu a zneviditeľniť všetky menej potrebné vchádzajúce a vychádzajúce hrany vrcholov podgrafu, aby používateľ mohol jednoduchšie skontrolovať graf. Na obrázku 5.1 vidíme graf vykreslený našim prostredím a na obrázkoch 5.2, 5.3, 5.4 môžeme vidieť ako naše prostredie zjednodušuje navigáciu v grafe.

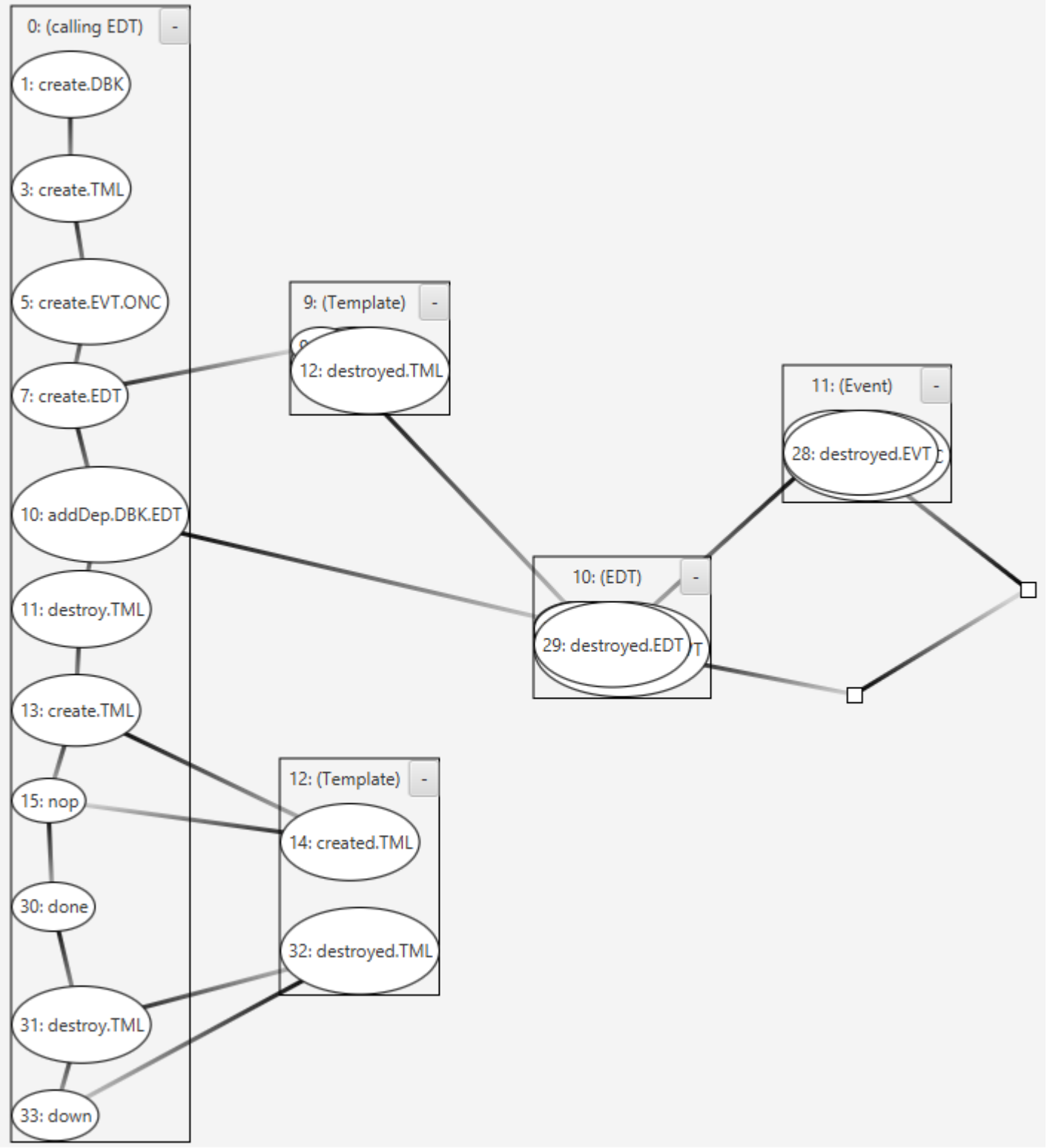
Do budúcnosti by sme chceli, aby náš nástroj neslúžil len na kontrolovanie chýb respektíve ich hľadanie, ale aj na korekciu grafov. To znamená, že by používateľ mal možnosť v grafickom rozhraní opravovať nekorektné grafy a náš nástroj by za nich generoval nový už opravený graf predstavujúci OCR program.



Obr. 5.1: Na obrázku môžeme vidieť graf vykreslený našim prostredím. Na obrázkoch 5.2, 5.3, 5.4 môžeme vidieť ako naše prostredie zjednodušuje navigáciu v grafe

Obr. 5.2: Na obrázku môžeme vidieť výsledok po minimalizovaní objektu EDT

Obr. 5.3: Na obrázku môžeme vidieť výsledok po pridaní pomocných vrcholov



Obr. 5.4: Na obrázku môžeme vidieť výsledok po minimalizovaní ostatných vrcholov

Záver

Problematika kreslenia grafov OCR programov, ktorej sme sa v práci venovali je pomerne veľká a doteraz neexistuje žiadny nástroj, ktorý by vizualizoval grafy OCR programov, ale pokiaľ si nájdeme dobrú metódu, tak je zvládnuteľná a vieme sa dopracovať k pomerne dobrým výsledkom.

Predstavili sme implementáciu *Sugiyamovho Frameworku*, ktorú sme si museli mierne poupraviť, kvôli problému s výberom vrcholov. Navrhli a implementovali sme heuristiku na prideľovanie úrovní podgrafom, vďaka ktorej sme boli reálne schopní použiť *Sugiyamov Framework*, keďže framework nepočíta s podgrafmi. Táto heuristika sa pokúša minimalizovať výšku grafu a tým graficky podporuje hierarchickosť vykresleného grafu. Má vynikajúcu časovú zložitosť. Na druhú stranu, nijakým spôsobom sme sa nezaoberali počtom pomocných vrcholov, ktoré by sme museli pridávať do grafu. Preto sme sa rozhodli neintegrovať pomocné vrcholy do nášho rozmiestňovacieho algoritmu ale nechali sme na používateľa, nech si pridá pomocné vrcholy kam potrebuje. Ďalší problém spôsobený podgrafmy je usporiadanie vrcholov v jednotlivých úrovniach, keďže podgrafy v našej implementácií prechádzajú naraz cez viac úrovní a preto sme nemohli pomocou permutovania vrcholov minimalizovať počet krížení hrán. Namiesto toho sme si zvolili jednoduchú heuristiku, ktorá sa sústredí na zvýraznenie hierarchickosti grafu, čo veľmi dobre vplývalo na konečné vykreslenie.

Narozdiel od DOTu, ktorý doteraz používala Paranoia. Náš vizualizátor prijíma rovnaký vstup ako Paranoia, čo uľahčí prácu pre Paranoiu, keďže nemusí generovať špeciálny vstup pre DOT. Ďalej náš vyzualizátor nevytvára statické obrázky ale dáva možnosť používateľovi lepšie analyzovať graf tým, že dovoľujeme používateľovi posúvať a zjednodušovať si graf.

Literatúra

[1] Open community runtime - extreme scale software stack, 2014. Dostupné na `https://xstackwiki.modelado.org/Open_Community_Runtime`.

[2] Graph drawing, 2017. Dostupné na `https://en.wikipedia.org/wiki/Graph_drawing`.

[3] K. Sugiyama a S. Tagawa a M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, Feb 1981.

[4] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tamassia, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. Drawing directed acyclic graphs: An experimental study. *International Journal of Computational Geometry & Applications*, 2000.

[5] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 1993.

[6] Michael Fröhlich and Mattias Werner. Demonstration of the interactive graph visualization system da vinci. In *International Symposium on Graph Drawing*. Springer, 1994.

[7] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot, 2006.

[8] Michael R Garey and David S Johnson. *Computers and intractability*. wh freeman New York, 2002.

[9] Patrick Healy and Nikola S Nikolov. A branch-and-cut approach to the directed acyclic graph layering problem. In *International Symposium on Graph Drawing*, pages 98–109. Springer, 2002.

[10] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[11] Nikola S Nikolov and Alexandre Tarassov. Graph layering by promotion of nodes. *Discrete Applied Mathematics*, 154(5):848–860, 2006.

[12] Nikola S. Nikolov a spol. Patrick Healy. *Handbook of Graph Drawing and Visualization*. CRC Press, 2013.

[13] Romain Cledat Tim Mattson. Ocr the open community runtime interface, 2016.

[14] Marc van Kreveld Utrecht University. Quality ratios in graph drawing, 2015. Prezentácia.

[15] J. N. Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, July 1977.

Elektronická príloha

Bakalárska práca taktiež obsahuje zdrojové kódy nášho grafického prostredia v elektronickej forme, pribalené na konci práce.