Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia vybraných algoritmov a vlastnosti z teórie grafov

Diplomová práca

Pavol Korinek

V Bratislave 22. júla 2006

Vedúci: doc. RNDr. Andrej Ferko, PhD.

Abstrakt

Mladá, ale dnes už doširoka bohatá teória grafov, vzniká koncom 19. a ďalej sa rozvíja začiatkom 20. storočia. Počiatky tejto teórie formovali matematici, ktorý riešili rôzne problémy, pri ktorých sa dalo abstrahovať od povahy zadania. Algoritmy a dokázané vlastnosti niektorých grafov majú široké uplatnenie v informatike. Overiť vlastnosť na grafe, znamená dokázať, či graf má danú vlastnosť, alebo nemá. Každý graf má napríklad každý svoj vrchol spojený s každým druhým vrcholom hranou, alebo má spojené iba niektoré dvojice vrcholov. V tomto prípade hovoríme, že daný graf je kompletný, alebo nie je. Budeme uvažovať iba grafy z jednoduchými hranami. Vylučujeme multihrany[1] a slučky. Graf s jedným vrcholom je graf. Takýto graf nemôže mať ani jednu hranu. Graf s dvoma vrcholmi má dve konfigurácie: s hranou alebo bez hrany. Práve sme opísali prvé tri grafy z lexikografického usporiadania grafov. Takýmto postupom môžeme generovať grafy. Chceme ukazovať vlastnosti grafov. Grafy reprezentujeme ako body na ploche, alebo v priestore. Ofarbujeme ich podľa toho či overená vlastnosť je na preverovanom grafe splnená, alebo nie je. Máme možnosť nahliadnuť na grafickú reprezentáciu vybraného, overeného grafu. Grafy môžme generovať od nami zvolenej konfigurácie a počtu vrcholov. Graf ktorý sa vygeneruje, overí sa na ňom vlastnosť a potom sa zobrazí na plochu, alebo do priestoru ako farebný bod. Výsledný obrázok odhaľuje skupiny grafov, alebo naopak výnimky, ktoré danú vlastnosť nemajú.

[1] viacnásobné hrany

Obsah

Kapitola 1

Úvod

Ľudia sú ochotní uveriť tomu, čo si prajú.
Gaius Julius Caesar

1.1 Zadanie

Zadaním a cieľom tejto diplomovej práce je vizualizácia vlastností grafov, ktoré poznáme z predmetu "Teória grafov"[1]. Ideu prístupu k tomuto problému vytvoril Ernest Stibrányi[2], vo svojej diplomovej práci "Vizualizácia teórie grafov" z roku 1997. Predstavil myšlienku reprezentovať grafy ako vhodne rozmiestnené body v rovine. Grafy umiestnil v sústredných kružniciach. Na kružnici má grafy s rovnakým počtom vrcholov. Grafy sú na kružnici rovnomerne rozostúpené v poradí ako boli vygenerované. Každý graf vyfarbil farbou, podľa toho, či daný graf spĺňa danú vlastnosť, alebo nie. Predpokladáme, že pri takomto zobrazení bude možné odhaliť zhluky grafov, ktoré majú danú vlastnosť a predpovedať hypotézy či grafy od istej konfigurácie, alebo v istom rozmedzí konfigurácii spĺňajú danú vlastnosť. Autor predchádzajúcej práce zobrazoval grafy do roviny. Svojou aplikáciou[3] vygeneroval obrázky na ktorých zobrazoval vlastností grafov do počtu štyroch vrcholov. Náročnejšie výpočty pre väčší počet vrcholov neboli v roku 1997 technicky uskutočniteľné v prítomnom čase. Našou snahou je overiť vybrané vlastnosti z teórie grafov na čo najväčšom počte grafov a rozšíriť zobrazenie grafov aj do tretieho rozmeru.

1.2 Čestné prehlásenie

Cestne prehlasujem, že som túto prácu vypracoval sám. V práci bolo čerpané z odbornej literatúry a webových stránok. Za každým použitým článkom, alebo myšlienkou je uvedená referencia na použitú literatúru, alebo url[4]. Ak je v práci použitý nový pojem, tak je označený indexom a vysvetlený pod čiarou na strane kde bol uvedený, iba pri prvom výskyte v texte. Práca sa smie používať iba na študijné účely.

[1] predmet čerpajúci prevažne z matematiky logiky a kombinatoriky
[2] absolvent FMFI UK
[3] aplikácia - počítačový program, zvyčajne zameraný na jednu oblasť
[4] skratka pre Uniform Resource Locator, zjednodušene povedané adresa lokalizácie na sieti internet

1.3 Poďakovanie

Pekne ďakujem učiteľom doc. RNDr. Andrejovi Ferkovi, PhD., doc. RNDr. Milanovi Ftáčnikovi, PhD. a doc. RNDr. Rastovi Kráľovičovi, PhD. za odborné rady a usmernenia. Ďakujem učiteľskému kolektívu FMFI za odovzdané vedomosti, ktoré otvárajú každému absolventovi FMFI bránu do sveta vedy a odbornej informatickej gramotnosti.

1.4 Predhovor

Každý priaznivec teórie grafov sa už mnohokrát zamýšľal nad množstvom dôkazov, matematických viet, faktov a vlastností vyplývajúcich z abstraktných grafárskych[5] štruktúr, nad algoritmami, ktoré ako svoj stavový priestor používajú tiež nejakú štruktúru spĺňajúcu definíciu grafu. Niektoré sú pochopiteľné po jednom prečítaní, iné po dvoch, troch a hlbokým zamyslením sa. Na pochopenie tých zložitých zapísal nie jeden celý papier. Človek si už s ťažkosťami vie pamätať štruktúry, alebo stavové priestory hĺbky viac ako tri. Ak sa prípadne nejedná o homogénne abstraktné prostredie, tak ešte ťažšie. Preto väčšina z nás siahne po pere a papieri, čo je aj najbezpečnejší spôsob tvorenia. Vhodné náčrty, kľukne aj do vetvenia desať, si spokojne môžeme takto prejsť a zamyslieť sa nad špeciálnymi prípadmi. Tento spôsob už dnes prekonali počítače. V textovom editori sa predsa tak ľahko gumuje. Ale aj obrázky a texty, ktoré sa ľahko dajú prepísať či prekresliť nie sú vždy postačujúce ak navrhujeme nový zložitý algoritmus. V istej fáze návrhu je potrebné prejsť k realizácii niektorých logických celkov. Pri ladení algoritmov náročných na kontrolu veľkého množstva hodnôt je užitočné a prehľadné vhodné počítačové grafické znázornenie, ktoré sa prípadne mení podľa priebehu algoritmu. Bez zložitého overovania každej hodnoty je vtedy stále vidieť, či algoritmus beží správne. Ľahko je takto vidieť aj jedinečné nežiaduce výnimky. Ako príklad si môžeme predstaviť výpočet normál na Bezierovej ploche a ich vykresľovanie počas zmeny riadiacich vrcholov. Ak by sa aj jediná normála vypočítala zle, tak to ľahko uvidíme. V tejto práci zobrazujem grafy ako body v rovine, alebo priestore. Potom ich ofarbujem podľa toho či spĺňajú overovanú vlastnosť. Výsledný obrázok sa postupne generuje podľa stavu výpočtu. Prechádzajú sa postupne grafy od zvolenej konfigurácie[6] cez nasledujúce konfigurácie v lexikografickom usporiadaní. Stanovenou konfiguráciou ako konečnou, overovanie vlastností na jednotlivých grafoch končí. Ak si uvedomíme, že grafov s počtom vrcholov štyri je 64, ale grafov s piatimi vrcholmi je už 1024, ľahko nahliadneme, že počty grafov rastú z rastúcim počtom vrcholov exponenciálne. Aj dnešné rýchle počítače majú svoju métu overovania vlastností grafov do určitého počtu, ktorá sa dá dosiahnuť v prítomnom čase. Môj predchodca v roku 1997 sa dostal po úroveň štyroch vrcholov. Ja sa pomocou upravených zobrazovacích techník pokúsim dosiahnuť hranicu šiestich až ôsmich vrcholov. V nasledujúcej kapitole si pripomenieme pojmy z teórie grafov, následne v ďalších kapitolách zhrniem teoretické a praktické naštudované možnosti realizácie implementácie a potom ukážem samotnú implementáciu programu a vyhodnotenie výsledkov.

[5] grafy pozostávajúce z vrcholov a hrán

[6] konfigurácia grafu je počet vrcholov, počet hrán a rozmiestnenie hrán v grafe

Kapitola 2

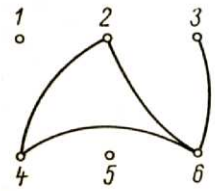
Teória grafov

Predstavivosť je dôležitejšia ako znalosti. Znalosti sú obmedzené, ale predstavivosť obklopuje celý svet.
Albert Einstein

2.1 Graf

Slovom graf budeme rozumieť diskrétnu štruktúru, ktorú je možné dobre znázorniť obrázkom v rovine, alebo priestore, pomocou vrcholov a hrán. Uvidíme, že sa takéto útvary - grafy - veľmi často vyskytujú v rôznych matematických úvahách, že dovoľujú prehľadne znázorniť mnohé vzťahy, ktoré sa zdajú na prvý pohľad neprehľadné a že sú užitočné aj v mnohých aplikáciach matematiky. Skôr ako pristúpime k presnej definícii, ukážeme si užitočnosť nového pojmu na príkladoch. [2]
Predstavme si, že je daná množina čísel $\{1, 2, 3, 4, 5, 6\}$ a že máme prehľadne znázorniť súdeliteľnosť[1] čísel z tejto množiny.

Túto úlohu môžeme vyriešiť napr. takto: Čísla $1, 2, 3, 4, 5, 6$ si znázorníme ako body v

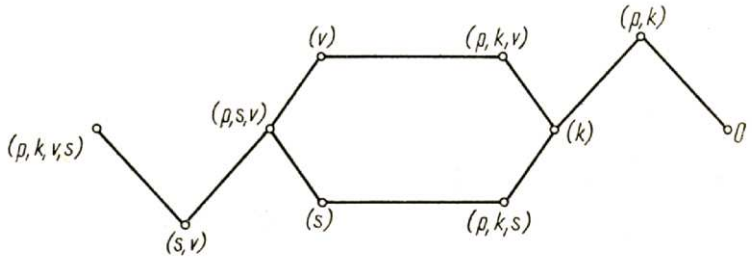


Obrázok 2.1: graf
[2]

rovine. (Obr. 2.1) Okrem Toho sú na obrázku 2.1 narysované oblúky medzi niektorými dvojicami týchto bodov. Oblúkom spájame práve tu dvojicu bodov, ktorá odpovedá dvom rôznym súdeliteľným číslam z danej množiny. V celku sme teda zostrojili útvar zložený zo štyroch

[1] čísla sú súdeliteľné ak jedno delí druhé bez zvyšku

oblúkov a šiestych bodov (ktoré sú na obrázku znázornené malými krúžkami). Z hotového obrázku je potom už na prvý pohľad zrejmé, ktoré dvojice uvažovaných čísel sú súdeliteľné a koľko je takých dvojíc. [2]
Uvažujme nasledovný príklad. Na ľavom brehu rieky stojí prievozník a má na svojej loďke previesť cez rieku kozu, vlka a seno. Loďka je malá a vojde sa do nej okrem prievozníka len jeden z uvedených troch pasažierov. Môže prievozník postupne dopraviť cez rieku kozu, vlka aj seno, ak nemôže ponechať osamote na brehu ani kozu s vlkom ani kozu so senom? [2]
Túto starú úlohu môžeme vyriešiť veľmi názorným spôsobom, ktorý opísal D. Kőnig[2].



Obrázok 2.2: graf
[2]

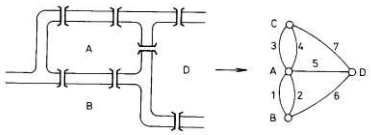
Najprv je na ľavom brehu štvorica prievozník - koza - vlk - seno, ktorú stručne označíme (p, k, v, s). Ďalej sú na ľavom brehu prípustné trojice (p, k, s), (p, k, v) a (p, s, v), dvojice (p, k) a (s, v) a samostatne tu môže ostať (v), (k), alebo (s). Konečný stav, pri ktorom prievozník, koza, vlk aj seno budú už na pravom brehu, označíme písmenom O. Popísali sme teda všetkých 10 možných prípadov, ktoré môžu nastať na ľavom brehu rieky. Každý z týchto prípadov môžeme znázorniť bodom v rovine, ako to ukazuje obrázok 2.2 [2]
Tu sme spojili niektoré body úsečkou, aby sme tým vyjadrili, že jedinou cestou lodičky je možné prejsť z jedného stavu do druhého. Náš obrázok podáva už veľmi prehľadným spôsobom odpoveď na otázku, ktorú sme si vyššie položili, prejdeme po úsečkách z bodu (p, k, v, s) do bodu O. Je vidieť, že ku splneniu úlohy musí prievozník cestovať cez rieku aspoň sedemkrát. [2]
Uviedol som dva príklady, v ktorých grafické znázornenie vyjasnilo riešenie. V prvom príklade sme uvažovali abstrakciu na všeobecný graf a v druhom príklade na orientovaný graf. [2]
Orientovaný graf môžeme zadefinovať tak, že každej hrane grafu priradíme určitý smer. Takéto útvary z orientovanými hranami sa uplatňujú v rôznych matematických úvahách. [2]

[2]Dénes Kőnig, (September 21, 1884 - Október 19, 1944), bol Maďarský matematik

2.2 História teórie grafov

Počiatky teórie grafov sú pomerne skromné. Na rozdiel od mnohých iných matematických disciplín, ktoré vznikli z významných (väčšinou fyzikálnych) problémov, pri vzniku teórie grafov často stáli úlohy z rekreačnej matematiky. [3]

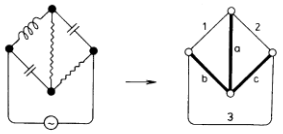
2.2.1 Problém mostov



Obrázok 2.3: problém mostov
[3]

Za otca teórie grafov sa pokladá vynikajúci matematik osemnásteho storočia L. Euler, ktorý v roku 1736 vyriešil nasledujúci problém (hlavolam). Cez mesto Koningsberg[3] tiekla rieka Pregil[4], ktorá tam vytvárala dva ostrovy. (Obr. 2.3) Tým rozdeľovala mesto na štyri časti, ktoré však boli pospájané siedmimi mostami. Úlohou bolo navrhnúť okružnú prechádzku po meste, pri ktorej sa prejde cez všetky mosty, ale po každom len raz. Euler (1736) vyriešil tento problém (ukázal, že takáto prechádzka neexistuje) a vytvoril aj teóriu riešiacu problémy takéhoto druhu. [3]

2.2.2 Elektrické siete



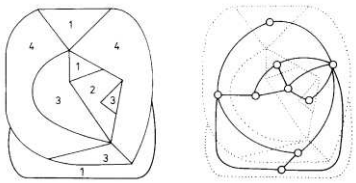
Obrázok 2.4: elektrické siete
[3]

Ďalší podnet, ktorý spomenieme, prišiel z fyziky. Čitateľ pravdepodobne pozná Kirchhoffove zákony pre elektrické siete. Tieto umožňujú napísať k danej sieti sústavu lineárnych rovníc. Takáto sústava bude však obsahovať zbytočné rovnice (lineárne závislé), keďže jeden z týchto zákonov nedáva žiadne obmedzenie na to, cez ktoré okruhy máme písať rovnice a preto ich píšeme cez všetky. Kirkhoff (1847) dokázal, že stačí uvažovať len nezávislé okruhy

[3]dnešný Kaliningrad
[4]dnešná Pregoľa

a uviedol ako ich treba hľadať. Na tento účel priradil elektrickej sieti grafické vyjadrenie ignorujúce fyzikálnu povahu prvkov (Obr. 2.4) a ukázal, že stačí v ňom nájsť nejakú kostru (hrubo vyznačená) a potom k tej kostre vždy pridať jednu zo všetkých hrán a uvažovať vzniknutý okruh. Takto sa získajú lineárne nezávislé okruhy a pre ne napísané rovnice budú tiež lineárne nezávislé. [3]

2.2.3 Farbenie máp



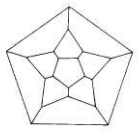
Obrázok 2.5: farbenie máp
[3]

Geografické (politické) mapy sa často kvôli prehľadnosti vyfarbovali. Aký je najmenší počet farieb, ktorý vždy stačí na zafarbenie mapy v rovine, resp. na guľovej ploche ? Prv než možno na otázku odpovedať, úlohu treba spresniť. Predovšetkým pôjde o také mapy, kde každý štát predstavuje súvislú oblasť. Od farbenia budeme žiadať, aby každé dva rôzne štáty mali rôznu farbu, ak majú spoločnú hranicu na nejakom úseku nenulovej dĺžky (ak majú iba konečný počet bodov spoločných, tak môžu mať tú istú farbu). Tento problém vznikol okolo roku 1850, odkedy sa mnohí pokúšali dokázať tzv. hypotézu o štyroch farbách, v ktorej sa hovorí, že štyri farby nám vždy stačia. (Obr. 2.5) ukazuje príklad, kedy sú štyri farby potrebné. [3]
K danej mape možno zostrojiť inú grafickú reprezentáciu tak, že v každom štáte zvolíme jeden bod a dva rôzne body spojíme čiarou práve vtedy, ak príslušné štáty majú spoločný úsek hranice. Tak je to urobené na obrázku 2.5. V tejto reprezentácii sa stačí zaoberať farbením bodov, pritom dva body spojené čiarou musia mať rôzne farby. [3]
Tento problém je veľmi slávny, stimuloval veľkú časť teórie grafov a podarilo sa ho vyriešiť len nedávno (1976). [3]

2.2.4 Hamiltonova hra

Írsky matematik R. W. Hamilton[5] v súvislosti s objavom nekomutatívnych algebier vynašiel v r. 1859 hru na pravidelnom dvanásťstene. (Obr. 2.6) Úlohou bolo cestovať z vrcholu do vrcholu po hranách tohto polyhedra[6] za predpísaných podmienok. Takto vzniklo niekoľko problémov. Jeden z nich bol problém nájdenia okružnej cesty, obsahujúcej každý vrchol práve raz. (Čitateľovi odporúčame jednu takú okružnú cestu nájsť.) Z tejto úlohy sa zrodil

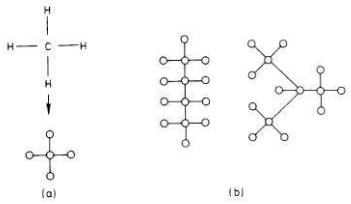
[5]Sir William Rowan Hamilton (August 4, 1805 - September 2, 1865) bol Írsky matematik, fyzik a astronóm
[6]geometrický objekt pozostávajúci z rovných plôch a rovných hrán



Obrázok 2.6: Hamiltonova hra
[3]

jeden smer výskumu v teórii grafov (problematika Hamiltonovských grafov). Hoci vidno analógiu s problémom mostov (Euler), predsa sú takéto úlohy omnoho ťažšie. [3]

2.2.5 Chemické grafy



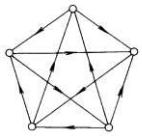
Obrázok 2.7: chemické grafy
[3]

Chemické zlúčeniny sa často znázorňujú graficky. K danému "sumárnemu vzorcu" priraďujeme tzv. "štruktúrny vzorec". Tieto štruktúrovane vzorce môžeme často zjednodušiť tak ako na obrázku 2.7, kde je CH_4 (metán). Na obrázku 2.4 (b) sú dva rôzne štruktúrne vzorce pre C_4H_{10}. Koľko existuje štruktúrnych vzorcov k danému sumárnemu vzorcu, je vo všeobecnosti ťažká otázka a pre uhľovodíky C_nH_{2n+2} sa ňou zaoberal už Cayley[7] v r. 1874. Hoci dosiahol len čiastočný úspech, grafické zobrazovanie sa ukázalo dobrou pomôckou. Práve na tomto základe Sylvester[8] (1878) používa termín graf v dnešnom zmysle teórie grafov. [3]

2.2.6 Turnaje

Uvažujme taký športový turnaj, kde každý hral s každým a vždy bol víťaz aj porazený. Graficky možno výsledok turnaja zobraziť napr. tak ako na obrázku 2.8) pre 5 účastníkov turnaja. (Šípka smeruje od víťaza k porazenému.) Pomocou takýchto grafických reprezentácii možno potom zaviesť rôzne zaujímavosti. Jedna z nich, ktorú ukázal Rédei, je takáto: Po skončení takéhoto turnaja vždy možno n účastníkov usporiadať do postupnosti $a_1, a_2, ..., a_n$

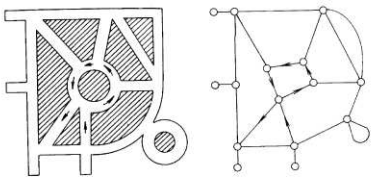
[7]Arthur Cayley (August 16, 1821 - Január 26, 1895) bol Britský matematik
[8]James Joseph Sylvester (September 3, 1814 Londýn, Marec 15, 1897 Oxford) bol Anglick7 matematik



Obrázok 2.8: turnaje
[3]

tak, že a_1 zvíťazil nad a_2, a_2 zvíťazil nad a_3 atď., až a_n-1 nad a_n. [3]

2.2.7 Cestná sieť



Obrázok 2.9: cestná sieť
[3]

Podobnými obrázkami možno zobrazovať časti cestnej siete. (Obr. 2.9) Tu vyznačíme križovatky, rázcestia, konce slepých uličiek a iné dôležité miesta, ktoré potom spájame čiarami reprezentujúcimi komunikácie, pritom šípka na čiare označuje jednosmernú komunikáciu. Takéto zobrazovanie je veľmi jednoduché a možno predpokladať, že vzniklo už v počiatkoch mapovania. Pre viaceré úlohy nám stačí uvažovať len grafické vyjadrenie. Neskoršie sa stretneme s viacerými praktickými úlohami z cestnej siete (nájsť najkratšiu cestu a pod.). [3]

2.2.8 Od hlavolamov ku teórii

Definitívny vznik teórie grafov sa viaže na rok 1936, kedy maďarský matematik D. Konig publikoval prvú monografiu z teórie grafov. [3]
Odvtedy sa táto disciplína začína prudko rozvíjať jednak z praktických podnetov a jednak z vnútorných potrieb teórie. [3]

2.3 Výučba teórie grafov na fakulte

Dnešná teória grafov je pomerne rozsiahla vedná disciplína, ktorá sa uplatňuje hlavne v informatike, ale aj v iných vedných oboroch. Výučbu teórie grafov na FMFI UK absolvuje každý študent informatiky, alebo matematiky v dĺžke aspoň jedného semestra. Za toto obdobie

sa naučí základné pojmy o grafoch, pomocou ktorých každý študent vie dokazovať základné vlastnosti grafov. Zhrnutím pojmov z teórie grafov si predurčujem množinu vlastností, z ktorých vybrané zahrniem do testovacieho procesu[9]. Celú preberanú látku z teórie grafov možno rozdeliť do nasledujúcich okruhov:

2.3.1 Úvod do teórie grafov

V tejto stati sa preberajú základne pojmy a vlastnosti grafov.
Uvediem vždy kombináciu pojem - štandardné označenie, pripadne synonymá oddelené bodkočiarkou.

1. Vrchol $v; v_i$
2. Hrana $e; e_i; v_i, v_j$
3. Graf $G; G'; G = (V, E) | V = (v_1, v_2, ..., v_n), E = (e_1, e_2, ..., e_n); G = (V, E), E \subseteq \frac{V}{2}; G = (V, E, I), I : E \longrightarrow \frac{V}{2} \cup \frac{V}{1}$
4. Pod - graf $G' \subseteq G \Longleftrightarrow V' \subseteq V \wedge E' \subseteq E$
5. Indukovaný pod - graf $G' \subseteq G \Longleftrightarrow V' \subseteq V \wedge E' \subseteq E \wedge e = v_i, v_j \wedge v_i = v'_i \in V' \wedge v_j = v'_j \in V' \Longrightarrow e' = v'_i, v'_j \in E'$
6. Komplement grafu $G' = k(G) \Longleftrightarrow V' = V \wedge v_i, v_j \in E \Longrightarrow v_i, v_j \notin E'$
7. Hranový pod - graf $G' = L(G) \Longleftrightarrow V' = V \wedge E' \subseteq E$
8. Stupeň vrcholu $deg(v)$
9. Kubický graf $\forall v \in V, deg_G(v) = 3$
10. Kružnica $\forall v \in V, deg_G(v) = 2$
11. Sled $u = v_1, e_1, v2, e_2, ..., v_{n-1}, e_{n-1}, v_n = v$
12. Ťah je ako sled bez opakujúcich sa hrán
13. Cesta je ako sled bez opakujúcich sa vrcholov
14. Uzavretý sled začína aj konci v tom istom vrchole
15. Kružnica je uzavretý sled
16. Dĺžka najkratšej cesty medzi dvoma vrcholmi $dist(u, v)$, ak neexistuje, tak $dist(u, v) = \infty$
17. Dĺžka najdlhšej cesty v grafe $diam(G)$
18. Priemer (Diameter) je dĺžka najdlhšej cesty v grafe

[9]mysli sa tým špecifická hodnotiaca funkcia založená na poznatkoch z teórie grafov

19. Excentricita vrcholu je najdlhšia vzdialenosť k inému ľubovoľnému vrcholu
20. Polomer (Rádius) je minimálna excentricita zo všetkých vrcholov
21. Minimálny stupeň v grafe $\delta(G)$
22. Maximálny stupeň v grafe $\Delta(G)$
23. Súvislý graf G je súvislý $\iff \forall v \in V, u \in V, dist(u, v) \neq \infty$
24. Komponent grafu je každý súvislý a maximálny pod - graf
25. Kompletný graf o n vrcholov K_n je $G|\forall v_i, v_j \in V, \exists e \in E, e == v_i, v_j$

2.3.2 Súvislosť

Samotný názov naznačuje, že sa jedná o to, či je daná štruktúra v jednom celku teda súvislá. Kladie sa dôraz na to, či sa súvislosť neporuší, ak sa vzťahy v grafe narušia.
Budem už len uvádzať základne pojmy, s ktorými sa študent môže oboznámiť na prednáške: "Teória grafov", alebo môže nahliadnuť do literatúry.

1. Artikulácia grafu G
2. Most grafu G
3. Nezávislé cesty
4. H cesta $H \subseteq V(G)$
5. X oddeľujúca A od B $X \subseteq V \cup E, A, B \subseteq V$
6. k - súvislosť
7. hranová k - súvislosť
8. max. k, G je súvislý $\kappa(G)$
9. max. k, G je hranovo súvislý $\lambda(G)$
10. Stupeň súvislosti vrcholu
11. 2 - súvislé grafy
12. Blok grafu
13. 3 - súvislé grafy
14. Kontrakcia
15. Eulerov polyhedralný vzorec

2.3.3 Stromy

V tomto okruhu sa preberá základná podskupina grafov stromy. To sú grafy, ktoré neobsahujú kružnicu. Preberajú sa základné prehľadávacie algoritmy[10] na týchto štruktúrach pod ktoré sa podpísal pán Tarry[11] .
Základné pojmy tejto state sú:

1. Strom
2. Acyklický graf
3. Koreň
4. Zakorenený strom
5. Zakorenený strom normálny
6. Kostra grafu
7. Normálna kostra grafu
8. Porovnateľnosť vrcholov vzhľadom na normálnu kostru
9. Prehľadávanie labyrintu
10. Prehľadávanie do hĺbky
11. DFS kostra
12. Modifikácia labyrintového prehľadávania

2.3.4 Eulerovské grafy

Tu sa študent dozvie ako vyriešiť samostatne problém Konigových mostov, oboznámi sa s podmienkami pre kreslenie útvarov jedným ťahom. A naučí sa aké sú nutné, alebo postačujúce podmienky aby bol graf Eulerovský[12]
Základné pojmy tejto state sú:

1. Eulerovský graf
2. Kvázi Eulerovský graf
3. Hľadanie Eulerovského ťahu

2.3.5 Priestory cyklov v grafe

Tu sa študent naučí vidieť analógiu medzi vektorovými priestormi z algebry a grafmi. Dozvie sa že aj graf môže mať bázu, ktorá ho jednoznačne generuje.

[10] takzvané labyrintové algoritmy, ktoré prehľadávajú labyrint tak, aby sa dostali na každé miesto

[11] Gaston Tarry (September 27, 1843 - Jún 21, 1913) bol Francúzky matematik

[12] graf, ktorý sa dá kresliť jedným ťahom pomenovaný po matematikovi a fyzikovi menom Leonhard Euler (Apríl 15, 1707 - September 18 [O.S. September 7] 1783)

2.3.6 Bipartitné grafy

Predstavujú grafy na rozdelenej množine vrcholov na dve skupiny. Znázorňujú sa vzťahy medzi nimi.

2.3.7 Párenia

Na bipartitných grafoch, všeobecných grafoch. Dokazujú sa tu tvrdenia o existencii úplného párovania. Rieši sa problém tanečnej školy.
Základné pojmy tejto state sú:

1. Párenie na grafe
2. Párenia v bipartitných grafoch
3. Striedavá cesta
4. Zväčšujúca s striedavá cesta
5. Problém tanečnej školy
6. 1 - faktor
7. d - faktor
8. Problém úplného párenia
9. Párenie vo všeobecných jednoduchých grafoch
10. Faktorovo kritický graf

2.3.8 Planárne grafy

Graf je abstraktný zápis čohosi čo ma pôvod v reálnom svete. Aby bol graf prehľadný, to je vtedy ak sa daný graf dá zobraziť tak, aby sa jeho hrany križovali čo najmenej. Ak sa žiadne dve hrany pretnúť nemusia ide o planárny graf.
Základné pojmy tejto state sú:

1. Planárnosť grafov
2. Rovnoľahlosť grafov
3. Cyklomatické číslo
4. Subdivízia hrany
5. Subdivízia grafu
6. Homomorfizmus
7. Topologický minor

8. Minor
9. Minor planárneho grafu
10. Plocha

2.3.9 Farbenia

Táto problematika ma korene pri vytváraní politických máp, kde dva susedné štáty sú vyplnené rôznymi farbami. Presnejšie zadanie bolo už v texte spomenuté. Rôzne iné logické problémy sa dajú previesť na problém farbenia grafov.
Základné pojmy tejto state sú:

1. Vrcholové farbenie
2. K - farbenie grafu
3. Chromatické číslo
4. Hranové farbenie
5. Hranové chromatické číslo
6. Mapa

2.3.10 Toky

V tejto oblasti sa rieši problém maximálnej priepustnosti danej siete ako abstrakcie napr. vodovodného potrubia, alebo dopravného systému, alebo počítačovej siete.
Základné pojmy tejto state sú:

1. Minimový rez
2. Maximálny tok

2.3.11 Hamiltonovské grafy

V tejto oblasti sa preslávili mená ako Dirac, Chvátal, Turán. Ide o špeciálne typy grafov a overovanie či spĺňajú Hamiltonovskú podmienku.

Kapitola 3

Vizualizácia informácii

Všetko čo je dnes dokázané, bolo v minulosti iba predstavou.
William Blake

3.1 Fotografia ako vizualizácia pravdy

Žijeme, o tom nie je pochýb, v dobe technických obrazov. Fotografie, obrázky z filmov, videa a digitálnych médií bojujú o našu pozornosť. Pokúšajú sa nás zvádzať, manipulovať s nami, erotizovať nás. Okrem toho nás aj informujú. Reč je o záplave fotografii, čo znie síce hrozivo, ale v podstate to poukazuje na fonologicky problém: Čo si s tými všetkými snímkami počať? Ako s nimi zaobchádzať ? Ako si vybrať ? A čo na druhú stranu - šancu získať prístup do kolektívnej fotografickej pamäti ? Čo ešte dokážeme vnímať ? [1]

3.2 Vizualizácia a štandardy

Informácie nás ovplyvňujú. Každý človek sa zaujíma o svet okolo seba. Všetci ľudia sa zaujímajú o svet okolo nás. Prijímame informácie o tom koľko je hodín, aké bude zajtra počasie, čo koľko stojí, alebo čo nového sa udialo v politike a vôbec vo svete. Prvým objavom, ktorý široko ovplyvnil šírenie informácii bola kníhtlač o ktorú ako ju poznáme v dnešnej dobe sa zaslúžil pán Guttenberg. Počas technickej revolúcii sa objavom rádia a neskôr televízie informovanosť ľudí značne urýchlila. Veľkou konkurenciou v obore masmédii je internet. Ten prichádza v 20. storočí a prudko ovplyvňuje rozvoj. Za posledné obdobie sa na internete vytvorilo a poskytuje toľko informácií ako za celé dejiny ľudstva. To poukazuje na to, že sa bude o to viac dbať na efektivitu, správnosť, v umení o dodržiavanie uznávaných techník a postupov. V žiadnom prípade takéto prísne podmienky neobmedzujú experiment, len určujú kadiaľ sa má či už veda, alebo umenie uberať. V praktickom svete sa kladie dôraz hlavne na jednoduchosť. Používateľ žiada, aby sa mu pracovalo ľahko a spracovávalo informácie intuitívne. Každý kto príde do styku s novou vecou, postupom, alebo programom sa nerád stretáva z ťažkopádnosťou. Dobrý návod, alebo program musí byť prehľadný. Zvládnutie nového postupu musí mať správny návod. Čínske príslovie hovorí: *Je lepšie raz vidieť ako*

tisíc krát počuť. Dnešný výrobcovia majú veľmi blízko ku zákazníkovi. Rôznymi spôsobmi sa snažia pochopiť potreby zákazníka. Dobrý návod obsahuje okrem textu aj názorné obrázky. Ak má byť návod, alebo program interaktívny, je veľmi vhodná vizualizácia.

3.3 Statická a dynamická vizualizácia

Zobraziť sa dá takmer úplne všetko. V prípade statickej vizualizácie ide vlastne o obrázky. Ak si chceme osviežiť pamäť, ako to vyzeralo na výlete pred časom, pozrieme si fotografie. Fotografia pomôže napríklad aj pri identifikácii. Názornejšia je dynamická vizualizácia, ktorá má povahu filmu, videá, alebo animácie. Fotografia a film sú prostriedky aj na zobrazovanie reality takej, akú ju môžeme vidieť vlastnými očami. V tom prípade ukazujú pravdu. Často krát je potrebné vizualizovať nie až tak reálne, alebo viditeľné veci, alebo dáta. Tie ale majú často svoj pôvod v reálnom svete.

3.4 Typy vizualizačných dát

Vizualizujú sa tieto typy dát:

1. nominálna skupina
    - biologické klasifikácie
    - populačné charakteristiky
2. kvantitatívna skupina
    - skalárne dáta
    - vektorové dáta
    - tensorové dáta
    - multihodnotové dáta

[4]

Poznáme mnoho vizualizačných techník a postupov. Jedným z nich sú technické úpravy obrazu, ktoré môžu sprehľadniť, alebo zvýrazniť pozorované. Tieto techniky vyplynuli z potrieb zobrazovať informácie z týchto odvetví:

- astrofyzika
- biológia
- chémia
- inžinierstvo
- geovýskum

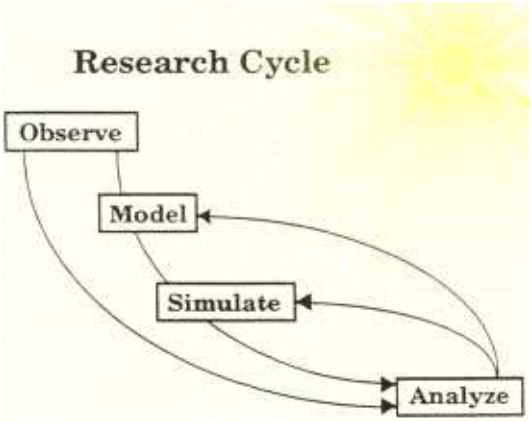
- medicína
- fyzika
- sociálny výskum
- matematika
- informatika

[4]

Informatika má vo vizualizácii zvláštne postavenie. Techniky najmä z tohto odvetvia prispievajú pre rozvoj vizualizácie. Špeciálne počítačová grafika. Ak zobrazujeme metódami počítačovej grafiky vlastné metódy a dáta tohto odvetvia, ide o druh reflexie, ktorá slúži na zdokonalenie vizualizačných techník. Dá sa využiť aj na zrýchlené učenie použitých postupov.

3.5 Cyklus výskumu

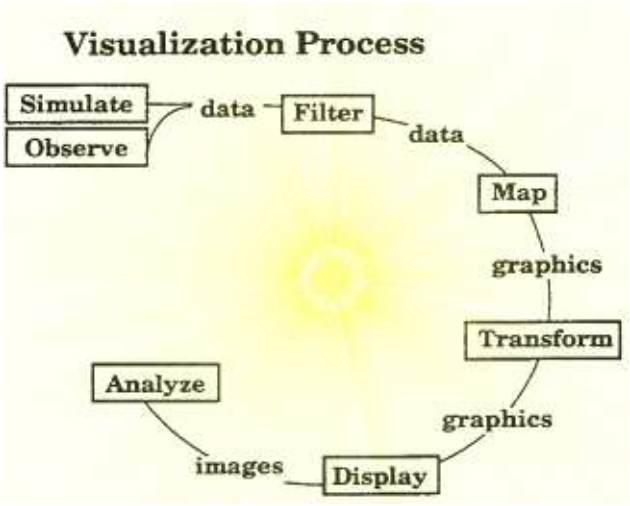
Počiatok výskumu je o pozorovaní, zapisovaní, zhrňovaní informácií o subjekte. To vyústi vo formulovanie a návrh modelu, prostredníctvom ktorého môžeme simulovať testovacie scenáre. Po každej simulácii nasleduje analýza priebehu a získaných výsledkov. Výsledky analýz môžeme ďalej štatisticky spracovať. Analýza je založená na porovnaní získaných, skutočných výsledkov a očakávaných výsledkov, predpokladaných už pri počiatočnom spracovaní informácii o riešenom probléme. Simulácia a analýza spätne implikuje prípadné zmeny modelu. (Obr. 3.1) Ten ohraničuje aj vytvára obor simulácie.



Obrázok 3.1: cyklus výskumu
[4]

3.6 Vizualizačný proces

Simuláciou a počiatočným zhromažďovaním informácií produkujeme dáta. Nie všetky chceme vždy zahrnúť ako vstupné dáta do výskumného cyklu. Všeobecné dáta je vhodné filtrovať a tým vytvárať špeciálne skupiny dát, od ktorých očakávame špecifický výsledok. Môžme tiež iba obmedziť množinu vstupných dát kvôli zmenšeniu oboru skúmania a sprehľadneniu výsledkov. Získané dáta mapujeme vybranou mapovacou funkciou na grafické dáta. Tato funkcia môže byť stanovená, alebo môže byť povolený výber z viacerých mapovacích funkcií. Získané grafické dáta môžeme vhodne transformovať na správne zobraziteľné dáta. Tie následne zobrazíme ako výsledný obrázok, ktorý je podkladom pre analýzu. Zvolené mapovanie, transformovanie a zobrazenie určuje konkrétnu vizualizáciu. To ako si to zvolíme, silne vplýva na výsledný obrázok. Pri rôznych vizualizáciach je možné vidieť, odhaliť, alebo zvýrazniť rôzne aspekty skúmaných dát.



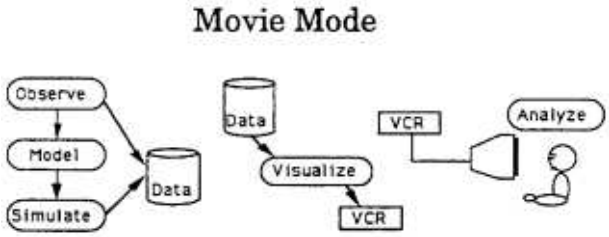
Obrázok 3.2: vizualizačný proces
[4]

3.7 Vizualizačné scenáre

Všeobecne pri vizualizácii potrebujeme vedieť čo ideme vizualizovať, aké dáta, aký typ dát a hlavne, čo tým chceme dosiahnuť. Rozlišujeme rôzne ložiská dát. Môžme mať dáta už zozbierané, alebo môžeme mať zdroj z neustálym prísunom nových dát. Je možné dáta zobrazovať priebežne ako do databázy prichádzajú, alebo si môžeme zvoliť pevne stanovenú časť zdrojových dát a pracovať iba na tejto množine. Ak zobrazujeme dynamické dáta, nemusíme sa vždy uspokojiť iba zo zobrazovaním zmien, ktoré možno nie sme ani schopní registrovať. Môžu nás zaujímať iba špecifické časti vizualizácie, alebo nás môže počas vizualizácie zaujať iba konkrétna skupina zo zobrazovaných dát. Cieľ vizualizácie určuje akú techniku je pre zobrazovanie dát najvhodnejšie použiť.

3.7.1 Filmový mód

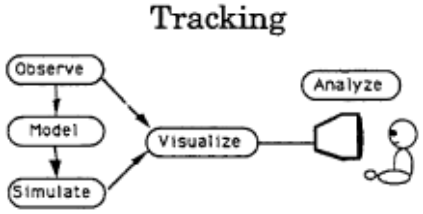
Filmovým módom rozumieme postup, ktorý sa skladá z troch častí. Tvorba vstypných dát pre vizualizáciu, v ktorej je zahrnuté zhromaždenie dát, vytvorenie modelu a simulácia. Druhou časťou je vizualizácia a tvorba filmu, alebo animácie. Záverečnou fázou je sledovanie vytvoreného videa. (Obr. 3.3) Jeho výhodou je, že pri tvorbe grafických dát nemusí byť prítomný pozorovateľ. Ten si už iba pozrie výsledok ako video. Nevýhodou takéhoto spracovania a vizualizácie dát je, že ak nastala chyba pri nastavení vizualizácie, tak je celé video chybné. Nie je možné interaktívne meniť spôsob vizualizácie. Nanajvýš by bolo ešte možné meniť výsledok filmárskymi technikami.



Obrázok 3.3: filmový mód
[4]

3.7.2 Sledovanie

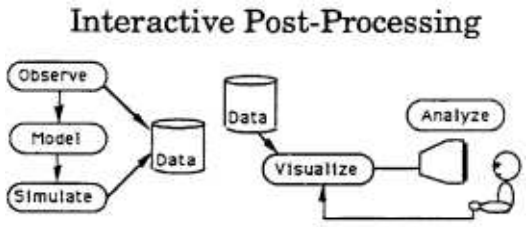
Na rozdiel od filmového módu je sledovanie založené na priamom zobrazovaní spracovávaných dát. Môže sa prirovnať k zobrazovaniu funkcií závislých od reálne nameraných dát, ktoré hneď zobrazíme. Pozorovateľ nemá ani v tomto prípade možnosť interaktívne ovplyvňovať vizualizáciu. Zhromažďovanie dát, alebo zhromaždené dáta, ich transformácia do modelu a simulácia sa priamo zobrazuje na monitor, ktorý sleduje pozorovateľ. (Obr. 3.4)



Obrázok 3.4: sledovanie
[4]

3.7.3 Interaktívne po - spracovanie

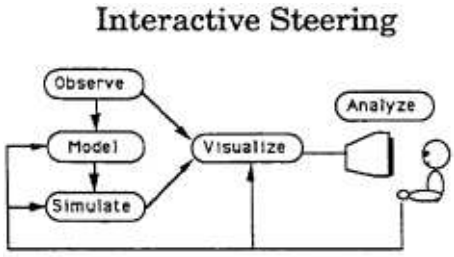
Tento druh vizualizačného scenára je podobný filmovému módu. Majú rovnakú prvú časť spracovania dát. Rozdiel je v tom, že v druhej časti sa vizualizácia nezaznamenáva na video. Zobrazuje sa na monitor pozorovateľa a ten má možnosť priamo, interaktívne ovplyvňovať samotnú vizualizáciu. (Obr. 3.5)



Obrázok 3.5: interaktívne po - spracovanie
[4]

3.7.4 Interaktívne riadenie

Pri takomto druhu scenára sú vstupné dáta interpretované modelom, simulované a vizualizované priamo pozorovateľovi na monitor. Ten môže interaktívne ovplyvňovať model, simuláciu aj vizualizáciu. (Obr. 3.6)



Obrázok 3.6: interaktívne riadenie
[4]

3.8 Vizualizácia grafu ako diagramu s dôrazom na zrozumiteľnosť a jasnosť

3.8.1 Konvencie pri kreslení grafov

- využívanie polyline
- planárnosť
- kolmé kreslenie
- kombinácie plavárne a kolmé
- reprezentácia s dôrazom na vizuálnosť

[6]

3.8.2 Hierarchia

- šípky tým istým smerom
- minimálne kríženie

[6]

3.8.3 Dôraz na rozlíšenie - parameter zrozumiteľnosti

- minimálny dištanc
  1. medzi vrcholmi
  2. medzi vrcholmi a neincedenčnými hranami

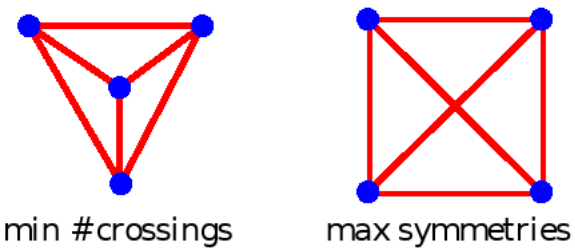
[6]

3.8.4 Uhlová zrozumiteľnosť

- Horná hranica
  $d = \Delta(G)$
  $\alpha \leq \frac{2\pi}{d}$

[6]

3.8.5 Estetické kritéria



Obrázok 3.7: výlučnosť kritérií
[6]

- niektoré vykreslenia sú lepšie ako iné
- čitateľnosť
- minimalizácia
  1. kríženia hrán
  2. oblastí

3. záhybov

- maximalizácia
  1. najmenšieho uhla
  2. symetrii
- nedá sa všetko
  1. minimálne kríženie
  2. maximálny počet symetrií

[6]

3.9 Zložitosť kreslenia grafu ako diagramu s dôrazom na zrozumiteľnosť a jasnosť

3.9.1 Planarita

- Testovanie planarity je možné v lineárnom čase
- Problém vzostupnej planarity je NP - ťažký[1]
- Problém minimálne kríženia hrán v grafe je tiež NP - ťažký

[6]

3.9.2 Planárne a kolmé kreslenie

- Vo všeobecnosti NP - ťažké
- Pre pevné vloženie dostávame polynomiálny čas

[6]

3.9.3 Stromy

- planárnosť
- priamočiarosť
- kreslenie stromu po úrovniach $\Omega(n^2)$ oblasť
- jednoduché kreslenie stromu vhodné pre binárne stromy $n-1$ šírky
- rekurzívne kreslenie stromu po úrovniach je neoptimálne

[1] definície NP problémov môže čitateľ nahliadnuť v [3]

- hľadanie optimálnej šírky je NP - ťažký problém
- Oblastné efektívne kreslenie stromu má $O(nlogn)$ oblastí, $O(n)$ šírku, and O(log n) hĺbku (Shiloach '76) [**?**]
- Open problem: determine the area requirement of planar upward straight-line drawing of trees[2] - Otvorený problém
- Open problem: can $O(n^{(\frac{1}{2})})$ size be achieved for nonupward planar straight line drawings of binary trees ? - Otvorený problém

[6]

3.9.4 Zložitosť planárneho kreslenia

- Testovanie planarity a konštruovanie planárneho vloženia môže byť urobené v lineárnom čase.
  1. prehľadávaním do šírky (Hopcroft Tarjan '74, Lempel Even Cederbaum '67)
  2. st - numbering and PQ - trees (de Fraysseix Rosenstiehl '82)
- Open problem: devise a simple and efficient planarity testing algorithm - Otvorený problém

[6]

3.9.5 Pružinový algoritmus

- hrany nahradíme pružinami
- ak bola dvojica vrcholov ďaleko o seba, tak ich pružina pritiahne k sebe
- ak bola dvojica vrcholov blízko pri sebe, tak ich pružina od seba odpudí

[6]

3.9.6 Iné prístupy

- Deklaratívny prístup
- Grafové gramatiky
- Neurónové siete
- Genetické algoritmy

Podrobnejšie a obšírnejšie informácie nájde čitateľ v [6], [7], [8]

[2]zložité definície otvorených problémov som kvôli presnosti nechal v Anglickom jazyku

3.10 Projekty na internete

- ILOG Vizualization
  Vytvára komplexne Java[3][16], .NET[4] a C++[5] grafické knižnice pre priemysel. Sú vhodné pre vývoj dátovo orientované, bohaté GUI[6] aplikácie, ktoré vyžadujú vysoký výkon, prenositeľnosť a intuitívnosť pri operáciach, ako napríklad pripájanie k databáze a synchronizácia.
  http://www.ilog.com/products/visualization/

- Drawing graphs
  http://math.ucsd.edu/ fan/graphdraw/
  Stránka je časťou Fun Chung Graham's web site[7]. Umožňuje stiahnuť zdrojové kódy algoritmov na grafoch. Poskytuje stiahnuteľné zdrojové kódy dema, ktoré si je možné ako on-line[8] aplikáciu vyskúšať.

- uDraw(Graph) - Silný nástroj pre vizualizáciu grafov
  Automatizuje zobrazovanie grafov. Vytvára mapovania, diagramy, hierarchie štruktúrovanej vizualizácie použitím automatických úprav, rýchlejšie ako každý iný bežný program. uDraw(Graph) API[9] môžete použiť aj vo svojich vlastných aplikáciách.
  http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html

- Drawing Graphs with VGJ
  VGJ, Vizualizácia Grafov pomocou Javy, je nástroj na vykresľovanie grafov a grafových schém. Sú dva spôsoby ako je možné graf importovať. Textovým popisom GML[10], alebo grafickým editorom na modelovanie grafov. Užívateľ si potom môže zvoliť algoritmus na zobrazenie grafu do organizovanej a estetickej podoby.
  http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html
  Aplikáciu je možné aj vyskúšať priamo na webovej stránke.
  http://www.eng.auburn.edu/department/cse/research/graph_drawing/vgj.html

- Tom Sawyer Software
  Významný dodávateľ vysoko rýchlostných grafických vizualizácii, schém a analytických systémov, ktoré Vám pri rozhodovaní lepšie umožňujú vidieť a interpretovať komplexné informácie.
  www.tomsawyer.com/home/index.php

[3] objektový programovací jazyk
[4] široká kolekcia produktov a technológii od firmy Microsoft
[5] objektový programovací jazyk založený na jazyku C a rozšírený o objektovo orientované programovanie
[6] Graphic User Interface - Grafické užívateľské prostredie
[7] kolekcia webových stránok
[8] on-line - za behu; on-line aplikácia zvyčajne spustiteľná v internetovom prehliadači
[9] Application Programming Interface - Rozhranie pre programovanie aplikácií
[10] Graph Modelling Language - Jazyk na modelovanie grafov

3.11 Projekt na FMFI UK

Práca Ernesta Stibrányiho prináša zaujímavú myšlienku zobrazovať grafy ako body na ploche. Zadefinoval zobrazenie, ktoré každému grafu priradí jeden bod na ploche. Grafy s rovnakým počtom vrcholov sú v jednej grupe. Grafy z jednej grupy pravidelne rozmiestnil na kružnicu. Každej grupe priradil jednu kružnicu s iným polomerom. Takto zobrazil lexikograficky vygenerované grafy do sústredných kružníc. Dané referenčné body potom vyfarbil podľa toho či daný graf spĺňa overovanú vlastnosť. Na vytvorenej dvojfarebnej množine bodov potom zostrojil triangluláciu. Každú plochu potom vyfarbil podľa prevládajúcej farbe vo vrcholoch. Takáto konštrukcia umožnila generovať obrázky zobrazujúce vlastností grafov. Ernesta Stibrányi z dôvodov vysokej výpočtovej zložitosti generoval grafy do počtu vrcholov štyri. Pri tomto počte vznikali prehľadné obrázky. Je potrebné zamyslieť sa nad tým, že pri grupách s vysokým počtom vrcholov rastie počet grafov exponenciálne. Preto je potrebné zvoliť veľmi veľkú kružnicu, tá by ale už bola ťažko zobraziteľná, alebo pozmeniť mapovaciu, alebo transformačnú funkciu pri vizualizácii. V tejto práci sa prikloním k druhej alternatíve.

3.12 Iné grafárske projekty vo svete

1. Graph Drawing Server - Brown University, USA
   - *loki.cs.brown.edu:8081/graphserver*
   - Roberto Tamassia (rt@cs.brown.edu)
2. GD Toolkit - University of Rome III
   - *www.dia.uniroma3.it/people/gdb/wp12/GDT.html*
   - Giuseppe Di Battista (dibattista@iasi.rm.cnr.it)
3. Graphlet - University of Passau, Germany
   - *www.fmi.uni-passau.de/Graphlet/*
   - Michael Himsolt (himsolt@fmi.uni-passau.de
4. GraphViz - AT & T Research
   - *www.research.att.com-sw-tools-graphviz*
   - Stephen North (north@research.att.com)

Kapitola 4

Aplikácia Grapher

Čo môžeš urobiť dnes, neodkladaj na zajtra.
príslovie

Cieľom tejto práce, ako je uvedené v úvode je vizualizácia vlastností grafov. Táto téma je nosnou stavebnou konštrukciou pri návrhu a implementácii aplikácie. V pod nadpise tejto diplomovej práce je zahrnutá aj vizualizácia grafárskych algoritmov ako pomôcka pri výučbe teórie grafov a algoritmov na grafoch. Pri zovšeobecňovaní témy som navrhol aplikáciu, ktorá dokáže vizualizovať vlastnosti grafov a vytvárať vizualizácie ľubovoľných algoritmov na grafoch. Sekundárnou ambíciou je aj mať možnosť sledovať algoritmus ako výpočet v jeho jednotlivých krokoch. Z teórie programov vieme, že program a jeho výpočet môžeme reprezentovať grafom, predikátmi a funkciami. Aby takto definovaný cieľ bolo možné realizovať, vedel som od začiatku, že budem musieť použiť interpretovaný jazyk, ktorý má zároveň podporu pre 3D akceleráciu[1]. Ako hlavný programovací jazyk pre užívateľské prostredie a vizualizáciu som zvolil Javu. Pomocou časti aplikácie, ktorá je naprogramovaná v Jave som schopný pracovať z grafmi, vizualizáciou a transformáciami. Výpočtovú časť som programoval v jazyku Python[2]. Dôvodom pre to bolo, že Python je jazyk v ktorom môžeme značne šetriť čo do množstva písania kódu a preto je vhodný aj na písanie pokusných algoritmov. Je to interpretovaný jazyk. V programovacom jazyku Java je spravená knižnica pomocou ktorej je možné vyvolávať z Javy kód písaný v jazyku Python. Takto je možné vykonávať kód Pythonu po atomických operáciách, alebo po vyšších abstrakciách čo môže napríklad znamenať cyklus. Tým môžeme presne vidieť v ktorej časti výpočtu sa nachádzame a ktorú časť výpočtu vizualizujeme. V nasledujúcich statiach sa pozrieme na jednotlivé aplikačné a logické celky.

[1] procesor počítača je odľahčený od zložitý grafických výpočtov, ktoré sú vykonávané špeciálnym procesorom priamo na grafickej karte

[2] vysoko úrovňový programovací jazyk

4.1 Programovacie jazyky a triedy

4.1.1 Aplikácia všeobecne

Pri hľadaní prostriedkov vhodných pre implementáciu aplikácie v obore vizualizácie, som zadefinoval nutné požiadavky, ktoré technológie musia spĺňať: možnosť ľahkej prepojiteľnosti na platformu, dobrá zásoba komponentov pre dynamický rozvoj aplikácie podľa potreby, dobre spolupracujúce komponenty jednotlivých tried, pohodlná manipulácia s objektmi, jednoduché používanie dynamických štruktúr a prepojenosť na iné programovacie jazyky. Tieto podmienky spĺňa programovací jazyk Java. Túto voľbu som zvolil po preštudovaní zdrojov: [15], [16], [17], [18], [19].

4.1.2 Grafický interakčný prostriedok

Moja aplikácia tvorí vizualizáciu a preto potrebujem technológiu pomocou ktorej je: pohodlná implementácia užívateľského prostredia, pohodlný spôsob vytvárania 3D scény, naviazanie na 3D akceleráciu a dobrá spolupráca s Java komponentmi. Vyhovujúca technológia je Java3D. Hlavné použité zdroje sú: [20], [21].

4.1.3 Paralelný výpočtový systém PCGS

Chcem overovať vlastnosti grafov čo do najvyššieho počtu vrcholov. Mohutnosti množín grafov rastú exponenciálne s rastúcim počtom vrcholov. Preto túto aplikáciu navrhujem ako paralelný program blízky systému PCGS. Technológia pre takúto implementáciu musí spĺňať tieto požiadavky: pohodlná implementácia, pohodlný spôsob programovania gramatiky a prepojenie na programovací jazyk Java. Vyhovujúce technológie sú Jython[3] a Java. Hlavné použité zdroje sú: [12], [23].

4.1.4 Generátor grafov

Grafy generujem v lexikografickom poradí. Potrebujem technológiu, ktorou implementujem takýto generátor a ktorá bude mať prepojenie na programovací jazyk Java. Vyhovujúce technológie sú: Jython alebo Huskell[4]. Hlavne použité zdroje sú: [12], [23].

4.1.5 Graf

Na implementáciu triedy Graf som vychádzal s týchto požiadaviek: pohodlná implementácia, pohodlný spôsob programovania, naviazanie na Java, možnosť zaznamenania štruktúry na disk spolu s naprogramovanými funkciami. Vyhovujúce technológie pre programovanie funkcii sú: Jython, Python, Huskell. Vyhovujúce formáty pre ukladanie grafu na disk sú: GML, XGGML, XML. Hlavne použité zdroje sú: [10], [11], [13], [22].

[3] implementácia jazyku Python pre použitie s programovacím jazykom Java

[4] funkcionálny programovací jazyk

4.1.6 Vstupy a výstupy

Graf sa dá vnútorne reprezentovať incidenčnou maticou[5]. To však nie je vždy postačujúce. Chcem aby sa dala uložiť aj grafická informácia o grafe. Je vhodné použiť štandardný formát súboru, ktorý je aj dostatočne všeobecný. Týmto podmienkam vyhovuje XML.

4.2 Projektové rozhodnutia

Základné kamene aplikácie tvoria programovacie jazyky: Java (j2se 1.5.0), Java3D. Gramatiku systému PCGS tvorí programovací jazyk Jython. Pre ukladanie grafu, jeho štruktúry a grafickej ako aj algoritmickej informácie používam jazyk XML.

4.3 Užívateľské prostredie

4.3.1 Pracovná plocha

Pracovná plocha nazvaná Desktop[6] v aplikácii Grapher je navrhnutá tak, aby sa užívateľovi pracovalo pohodlne a prehľadne. Je to docielené tým, že aplikácia podporuje editovanie a zobrazovanie vo viacerých oknách. Hlavnou triedou tohto balíka je RootFrame, ktorá je zdedená z triedy javax.swing.JFrame. Táto trieda vzniká hneď po spustení aplikácie vo funkcii main().

```
/**
 * User: palo
 * Date: 4.12.2005
 * Time: 22:26:28
 */
import desktop.RootFrame;

public class Grapher {

    public static void main(String[] args) {
        //volám konštruktor triedy RootFrame
        RootFrame frame = new RootFrame("RootFrame");
        //nastavím RootFrame viditeľný
        frame.setVisible(true);
    }
}
```

V konštruktore triedy RootFrame sa volá funkcia init() v ktorej sa vytvorí Layout[7] a tri základné interné okná pre vstupnú aj výstupnú konzolu a okno pre zobrazovanie grafov.

[5]binárna štvorcová matica o veľkosti počtu vrcholov majúca jednotku v i - tom a j - tom stĺpci vtedy a len vtedy ak i - ty a j - ty vrchol je spojený hranou

[6]Pracovná plocha, všetky komponenty aplikácie sa nachádzajú na pracovnej ploche

[7]Grafická úprava, v tomto význame celkový vzhľad a rozmiestnenie komponentov

```
    private void init() {
        //vytvorenie layout
        west = new WestRootFrameLayoutBuilder(this);
        //budovanie layout
        west.build(this);
        //nastavenie hraníc pre RootFrame
        setBounds(west.rectangle);
        //nastavenie Look and Feel atribútu
        setDefaultLookAndFeelDecorated(true);
        //nastavenie aby aplikácia skončila,
        //keď zavriem RootFrame okno
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                System.exit(0);
            }
        });
        //vytvorenie výstupnej konzoly
        createOutputInternalFrame(this);
        //vytvorenie vstupnej konzoly
        createInputInternalFrame(this);
        //vytvorenie okna pre zobrazovanie a editovanie grafov
        createDataStructViewInternalFrame(this);
}
```

Ukážeme si najprv čo sa deje pri vytváraní layout. Keďže trieda WestRootFrameLayoutBuilder vo svojom konštruktore volá iba konštruktor svojho predka RootFrameLayoutBuilder, ukážeme si iba ten.

```
    public RootFrameLayoutBuilder(RootFrame rootFrame) {
        int inset = Const.UNI_BOUND;
        //zistím si dimenziu obrazovky
        java.awt.Dimension screenSize = java.awt.Toolkit.getDefaultToolkit()
                        .getScreenSize();
        //vytvorím si vlastnú dimenziu zmenšenú o konštantu
        dimension = new java.awt.Dimension(screenSize.width - inset*2,
                        screenSize.height - inset*2);
        //nastavím virtuálnu dimenziu pre frame
        rectangle = new java.awt.Rectangle(dimension);
        insets = new java.awt.Insets(0, 0, dimension.height, dimension.width);
        //vytvorím desktopFactory
        DesktopFactory desktopFactory = new DesktopFactory();
        //vytvorím desktop
        desktop = desktopFactory.create(dimension, rootFrame);
        //umiestnim desktop do scrollpane
        scrollpane = new javax.swing.JScrollPane(desktop);
    }
```

Pri budovaní layout sa trieda scrollpane, ktorá obsahuje desktop pridá do RootFrame. Pozrieme sa čo sa deje pri vytváraní desktop pomocou DesktopFactory, ale najprv čo sa deje v konštruktore triedy Desktop.

```
public Desktop(RootFrame rootFrame) {
    super();
    //nastavím rootFrame
    this.rootFrame = rootFrame;
    //inicializujem pop-up menu
    popUpMenu = new JPopupMenu();
    //nastavím light weight na false, aby pop-up menu prekrývalo
    //komponenty, ktoré niesu light weight
    popUpMenu.setLightWeightPopupEnabled( false );
    init();
}

private void init() {
    //nastavím default desktop manager
    desktopManager = new javax.swing.DefaultDesktopManager();
    setDesktopManager(desktopManager);
    putClientProperty("JDesktopPane.dragMode", "outline");
    //nastavím parametre chovania desktop
    setVisible(true);
    setBackground(java.awt.Color.LIGHT_GRAY);
    setDragMode(javax.swing.JDesktopPane.OUTLINE_DRAG_MODE);
    setDoubleBuffered(true);
    //vytvorím pop-up menu
    createPopUpMenu(rootFrame);
    //nastavím zobrazovanie sa pop-up menu
    createMouseListener(rootFrame);
    this.addMouseListener(mouseListener);
}
```

Pri vytváraní Desktop pomocou DesktopFactory sa nastavia už len rozmery pracovnej plochy.

```
public Desktop create (java.awt.Dimension size, RootFrame rootFrame) {
    Desktop desktop = new Desktop(rootFrame);
    desktop.setPreferredSize(new java.awt.Dimension(size));
    desktop.rectangle = new java.awt.Rectangle(size);
    desktop.insets = new java.awt.Insets(0, 0, size.height, size.width);
    desktop.setAutoscrolls(true);
    return desktop;
}
```

Ukážeme si ďalej ako vznikajú vstupné a výstupne konzoly a ako vzniká okno na zobrazovanie a editovanie grafov.

4.3.2 Vstupná konzola

Po vytvorení inštancie z InternalFrameFactory je možné použiť metódu create s parametrami rootFrame, Const.TEST_IN, null, pre vytvorenie vstupnej konzoly.

```
    void createInputInternalFrame(RootFrame rootFrame) {
        //vytvorím internal frame factory
        InternalFrameFactory internalFrameFactory = new InternalFrameFactory();
        //vytvorím vstupnú konzolu
        javax.swing.JInternalFrame internalFrame = internalFrameFactory
                        .create(rootFrame, Const.TEST_IN, null);
        rootFrame.west.desktop.add(internalFrame, Const.BACK);
    }
```

Pozrime sa na časť kódu, ktorá odpovedá týmto parametrom, ale najprv si pozrime triedu InInternalFrame a jej konštruktor.

```
public class InInternalFrame extends javax.swing.JInternalFrame {
    //referencia na kontajner v tomto frame
    java.awt.Container container;
    //referencia na layout tohto frame
    WestInLayoutBuilder westInLayoutBuilder;
    //referencia na hlavný frame
    public RootFrame rootFrame;
    //referencia na text area komponent
    public javax.swing.JTextArea output;

    //konštruktor
    public InInternalFrame(RootFrame rootFrame, String title, boolean resizable,
                boolean closable, boolean maximizable, boolean minimizable) {
        super(title, resizable, closable, maximizable, minimizable);
        //funkcia init
        init(rootFrame);
    }
    //funkcia init
    private void init(RootFrame rootFrame) {
        //nastavím referencie
        this.rootFrame = rootFrame;
        this.output = rootFrame.west.output;
        container = this.getContentPane();
        //vytvorím layout pre InInternalFrame spolu s
        //ovládacími prvkami pre tento frame
        westInLayoutBuilder = new WestInLayoutBuilder(this);
        westInLayoutBuilder.build(container);
        //nastavím parametre pre frame
        setLocation(0, 0);
        setSize(500, 200);
```

```
        setBackground(java.awt.Color.YELLOW);
        setName(title);
        setTitle(title);
        setVisible(true);
        setDoubleBuffered(true);
        setAutoscrolls(true);
        setEnabled(true);
        setFocusable(true);
    }
```

Ďalej trieda InInternalFrame obsahuje dve funkcie openPyCode() a savePyCode(), ktoré umožňujú načítať a uložiť kód v jazyku Python pomocou dialógových okien. Ich význam sa dozvieme neskôr. Teraz sa môžeme pozrieť na vznik InInternalFrame v InternalFrameFactory.

```
.
.
.
} else if (type.equals(Const.TEST_IN)) {
    //vytvorím internal frame
    frame = new InInternalFrame(rootFrame, Const.TEST_IN, true, true, true, true);
    InInternalFrame inInternalFrame = (InInternalFrame) frame;
    //nastavím referenciu input pre hlavný frame
    rootFrame.west.input = inInternalFrame.westInLayoutBuilder.textArea;
    //vytvorím poslucháča pre odchytávanie presunu interného frame myškou
    frame.addComponentListener(new java.awt.event.ComponentListener() {
        .
        .
        .
        public void componentMoved(java.awt.event.ComponentEvent e) {

            java.awt.Component originator = e.getComponent();

            int originatorX = originator.getX();
            int originatorY = originator.getY();
            //okno nemôžeme dať v ľavo, alebo hore mimo pracovnú plochu
            if (originatorX < rootFrameInsets.left) {
                originator.setLocation(rootFrameInsets.left, originatorY);
            }
            if (originatorY < rootFrameInsets.top) {
                originator.setLocation(originatorX, rootFrameInsets.top);
            }
            //okno mame povolené umiestniť vpravo alebo dole mimo pracovnej
            //plochy a tým zväčšiť pracovnú plochu
            int count = desktop.getComponentCount();
```

```
if (count > 0) {
    java.awt.Component component = desktop.getComponent(0);
    int minX = component.getX();
    int minY = component.getY();
    int maxX = minX + component.getWidth();
    int maxY = minY + component.getHeight();

    for (int i = 0; i < count; i++) {

        component = desktop.getComponent(i);
        int componentX = component.getX();
        int componentY = component.getY();
        int componentMaxX = componentX + component.getWidth();
        int componentMaxY = componentY + component.getHeight();

        if (componentX < minX) {
            minX = componentX;
        }
        if (componentMaxX > maxX) {
            maxX = componentMaxX;
        }
        if (componentY < minY) {
            minY = componentY;
        }
        if (componentMaxY > maxY) {
            maxY = componentMaxY;
        }
    }

    if (maxX >= rootFrameInsets.right) {

        desktopInsets.right = maxX;
    } else {
        desktopInsets.right = rootFrameInsets.right;
    }

    if (maxY >= rootFrameInsets.bottom) {
        desktopInsets.bottom = maxY;
    } else {
        desktopInsets.bottom = rootFrameInsets.bottom;
    }
    //nastavím nové hranice a veľkosť virtuálnej pracovnej plochy
    desktopRect.setBounds(desktopInsets.left, desktopInsets.top,
            desktopInsets.right - desktopInsets.left,
            desktopInsets.bottom - desktopInsets.top);
```

```
                desktop.setPreferredSize(new java.awt.Dimension(desktopRect.width,
                                desktopRect.height));
                desktop.setSize(new java.awt.Dimension(desktopRect.width,
                      desktopRect.height));
            }
        }
    });
    .
    .
    .
```

4.3.3 Výstupná konzola

Analogicky ako som vytvoril vstupnú konzolu vytváram aj výstupnú konzolu. Funkcionalita je ale iná. Výstupná konzola slúži na kontrolu a videnie časti kódu v jazyku Python, ktorý sa vykonáva vo vlákne.

```
void createOutputInternalFrame(RootFrame rootFrame) {
    InternalFrameFactory internalFrameFactory = new InternalFrameFactory();
    //vytvorím výstupnú konzolu za pomoci InternalFrameFactory
    javax.swing.JInternalFrame internalFrame = internalFrameFactory.
            create(rootFrame, Const.TEST_OUT, null);
    rootFrame.west.desktop.add(internalFrame, Const.BACK);
}
```

4.3.4 Okno pre operácie z grafmi

Správanie sa interných okien ako okien je rovnaké, iná je ich funkcionalita. Pozrieme sa podrobnejšie na vznik DataStructViewInternalFrame, ktorý ma funkcie pre načítanie a uloženie grafu z disku a na disk vo formáte XML. DataStructViewInternalFrame má ďalej možnosť editovať a vizualizovať graf, pomocou malých programov v jazyku Python overovať funkcie na grafoch a v neposlednom rade aj vizualizovať dané vlastnosti a algoritmy na grafoch. Najprv sa pozrieme na DataStructViewInternalFrame z pohľadu užívateľského prostredia a potom v nasledujúcej stati o Prehliadači sa pozrieme na operácie na grafoch a tiež na samotnú vizualizáciu. InternalFrameFactory vytvára aj DataStructViewInternalFrame.

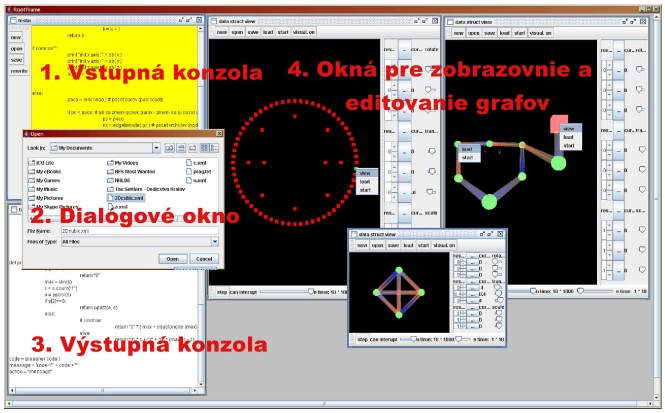
```
void createDataStructViewInternalFrame(RootFrame rootFrame) {
    InternalFrameFactory internalFrameFactory = new InternalFrameFactory();
    //vytvorím okno na zobrazovanie a editovanie grafov
    javax.swing.JInternalFrame internalFrame = internalFrameFactory.
            create(rootFrame, Const.DATA_STRUCT_VIEW, null);
    rootFrame.west.desktop.add(internalFrame, Const.FRONT);
}
```

Po vytvorení inštancie z InternalFrameFactory je možné použiť metódu create s parametrami rootFrame, Const.DATA_STRUCT_VIEW, null, pre vytvorenie DataStructViewInternalFrame. Pozrime sa najprv na konštruktor DataStructViewIntenalFrame().

```
public DataStructViewIntenalFrame(RootFrame rootFrame, String title,
            boolean resizable, boolean closable, boolean maximizable,
            boolean minimizable, Graph graph) {
    super(title, resizable, closable, maximizable, minimizable);
    //nastavím referenciu na hlavný frame
    this.rootFrame = rootFrame;
    //nastavím referenciu na výstupnú konzolu
    this.output = rootFrame.west.output;
    //nastavím referenciu na zobrazovaný graf reprezentovaný triedou Graph
    this.graph = graph;
    //zavolám inicializáciu
    init();
}

private void init() {
    //nastavím vlastnosti okna
    container = this.getContentPane();
    setLocation( rootFrame.west.desktop.insets.right - 450, 0 );
    setSize( 400, 300 );
    setBackground( java.awt.Color.GREEN );
    setVisible(true);
    setFocusable(true);
    createGraphAndLayoutAndViewer();
}

private void createGraphAndLayoutAndViewer() {
    //ak bol parameter graph null, tak vytvorím nový primitívny graph
    //bez vrcholov a hrán
    if (graph == null) {
        graph = new Graph("graph 0", 0,rootFrame, null);
    }
    //obnovím grafický objekt triedy Graph
    graph.updateScene();
    //vytvorím layout pre okno obsahujúci tlačídlá, toolbary a panely
    west = new WestDataStructViewLayoutBuilder( rootFrame, this);
    if (container != null) {
        container.removeAll();
    }
    west.build( container );
    //vytvorím poslucháčov na odchytávanie podnetov z tlačidiel a myši
    west.createButtonsListeners(graph);
    west.createToolBarButtonsListers(graph);
    west.createToolTimeBarButtonsListers(graph);
    //vytvorím plochu na zobrazovanie grafov
    viewer = new Viewer(rootFrame);
    //nastavím poslucháčov
```

```
        viewer.viewerCanvas3D.updateFirstListeners(rootFrame, graph);
        //pridám graf na zobrazenie
        viewer.connect(graph);
        //pridám zobrazovanie grafov do okna
        west.viewPanel.add("Center", viewer.viewerCanvas3D);
    }
```

Po úvodnej inicializácii môžeme zobraziť aplikáciu so vstupnou konzolou, výstupnou konzolou a s oknom pre zobrazovanie grafov. (Obr. 4.1)



Obrázok 4.1: pracovná plocha

Vstupná konzola označená číslom 1 (Obr. 4.1), má pozadie žltej farby. Slúži na načítavanie a editovanie malých programov v jazyku Python. Okno obsahuje menu na ľavej strane [19]. Tvoria ho tlačidlá: new, ktoré slúži na vymazanie doterajšieho kódu, open a save, ktoré slúžia na uloženie a načítanie kódu pomocou dialógového okna, označené číslom 2 (Obr. 4.1), a tlačidlo "initialize", umožňujúce inicializáciu obsahu vstupnej konzoly z výstupnej konzoly [18]. Výstupná konzola označená číslom 3 (Obr. 4.1), má pozadie bielej farby. Výstupná konzola slúži na zobrazovanie Python kódu. Nemá žiadnu inú funkcionalitu. Okná pre editovanie a zobrazovanie grafov, sú označené číslom 4 (Obr. 4.1). Základnou funkcionalitou takéhoto okna je zobrazovanie grafu ako štruktúry. Pridanie vrcholu je umožnené prostredným tlačidla myši. Označenie vrcholu je možné pomocou myši a tlačidla ctrl na klávesnici. Okno má implementovaný 3D selektor, ktorý umožňuje uchopenie a manipuláciu z vrcholom kdekoľvek v priestore. Po uchopení vrcholu myšou je možné pohybovať vrcholom v rovine, ktorá je umiestnená v bode vrcholu a je kolmá na vektor daný polohou kamery a vrcholom. Vyznačený vrchol môžeme posúvať po priamke danej spomínaným vektorom pomocou kolieska na myške. Ak označíme vrcholov viac, tak manipulujeme spolu so všetkými. Operácie editovania sa vykonávajú pomocou pop-up menu na zobrazovacej ploche. To umožňuje pridať hranu medzi dva označené vrcholy, vymazať ľubovoľný označený prvok grafu, označovať osobitne hrany, alebo vrcholy grafu a tiež robiť inverzie pri označovaní. Plocha na zobrazovanie grafov rozlišuje tri druhy pop-up menu. Prvé slúži na editovanie štruktúr grafu. Zobrazí sa po kliknutí pravým tlačidlom myši vtedy, ak kurzor myši smeruje mimo ľubovoľného prvku grafu. V prípade že myš ukazuje na vrchol, ktorý je reprezentovaný zelenou guľou, zobrazí sa nám druhý typ pop-up menu s funkciami na načítanie kódu so vstupnej konzoly do vrcholu grafu a funkciou štart, umožňujúcej poslanie inicializačnej správy po hrane svojím

susedným vrcholom. Tretie pop-up menu sa zobrazí ak kurzor myši ukazuje na graf, ktorý je v priestore reprezentovaný červenou kockou. Toto pop-up menu má funkcie ako pop-up menu pre vrchol a možnosť zobraziť štruktúru grafu v novom okne. Zobrazujeme graf, ktorý má okrem obyčajných vrcholov ešte špeciálne vrcholy, ktoré môžu mať štruktúru, teda sú grafmi. Túto vlastnosť využijeme neskôr pri overovaní vlastností grafov. Okno pre vizualizáciu obsahuje vpravo umiestnený ovládací panel slúžiaci na ovládanie transformácii. Skladá sa z troch štandardných častí pre rotáciu, posun a škálovanie. Každá časť ma ovládacie prvky pre transformáciu v smere v jednej z troch súradnicových osí. Hrubé nastavenie je možné pomocou Java komponentu slider. Jemné nastavenie je možné pomocou Java komponentu spinner. Po hrubej zmene nastavenia transformácie sa je možné, cez tlačidlo reset, vrátiť transformáciu do pôvodného nastavenia definovaného komponentom spinner. V hornej časti má okno pre vizualizáciu hlavné menu, ktoré má funkcie pre vytvorenie primitívneho grafu, načítanie grafu zo súboru, uloženie grafu na disk, hromadné načítanie kódu v jazyku Python do označených vrcholov v grafe z vstupnej konzoly, hromadné zaslanie inicializačnej správy z označených vrcholov a tlačidlo na zapnutie, alebo vypnutie vizualizácie putujúcich správ cez hrany. V dolnej časti okna sa nachádza panel pre nastavenie doby spánku vlákna, ak spracoval všetky obdŕžané správy a doby cyklu, v ktorom sa správa vizualizuje na hrane. Správnym nastavením tohto panelu sa dá docieliť výraznejšia a efektnejšia vizualizácia.

4.4 Prehliadač

Najzaujímavejšou časťou je vytváranie zobrazovania grafov. Pozrime sa na triedu Prehliadač nazvanej Viewer a jej konštruktor Viewer(rootFrame). Scéna sa v nej vytvára stromom, ktorý sa skladá z uzlov grúp a transformácii, ktorý ma v listoch jednoduché objekty nazývanom BSP strom.

```
public class Viewer {

   //premenné pre 3D kameru
   //poloha kamery
   private Point3d cameraLocation;
   //smer kamery
   private Point3d cameraLookAt;
   //os kamery
   private Vector3d cameraUpVector;
   //matica natočenia kamery
   private Transform3D viewTransform;
   //premenné pre vesmír
   private SimpleUniverse simpleUniverse;
   //koreň stromu scény
   private BranchGroup rootBranchGroup = new BranchGroup();
   //rotačná matica scény
   private TransformGroup view = new TransformGroup();
   //uzol
   private BranchGroup branchGroup = new BranchGroup();
```

```
//canvas = plocha na zobrazenie zdedená od Canvas3D
//pochádzajúca z Java3D
public ViewerCanvas3D viewerCanvas3D;
//referencie na hlavné okno a výstupnú konzolu
public RootFrame rootFrame;
public javax.swing.JTextArea output;

//konštruktor
public Viewer(RootFrame rootFrame) {
    this.rootFrame = rootFrame;
    output = rootFrame.west.output;
    //nastavenie atribútov pre uzol aby bolo možné modelovať graf počas
    //zobrazovania scény
    branchGroup.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);
    branchGroup.setCapability(BranchGroup.ALLOW_CHILDREN_WRITE);
    branchGroup.setCapability(BranchGroup.ALLOW_DETACH);
    //inicializácia kamery
    cameraLocation = new Point3d(0, 0, 0);
    cameraLookAt = new Point3d(0, 0, -1);
    cameraUpVector = new Vector3d(0, 1, 0);
    //inicializácia matice pohľadu
    viewTransform = new Transform3D();
    viewTransform.lookAt(cameraLocation, cameraLookAt, cameraUpVector);
    //vyváranie stromu
    view.setTransform(viewTransform);
    view.addChild(branchGroup);
    rootBranchGroup.addChild(view);
    GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
    //vytvorenie canvasu spolu z odchytávaním podnetov z myšky, a pop-up menu
    CanvasFactory canvasFactory = new CanvasFactory();
    viewerCanvas3D = canvasFactory.create(rootFrame, config);
    //canvas pridám do dataStructViewFrame až po vytvorení Viewer
    simpleUniverse = new SimpleUniverse(viewerCanvas3D);
    simpleUniverse.addBranchGraph(rootBranchGroup);
    //nastavenie vlastností zobrazovania
    ViewingPlatform viewingPlatform = simpleUniverse.getViewingPlatform();
    viewingPlatform.setNominalViewingTransform();
}
.
.
.
```

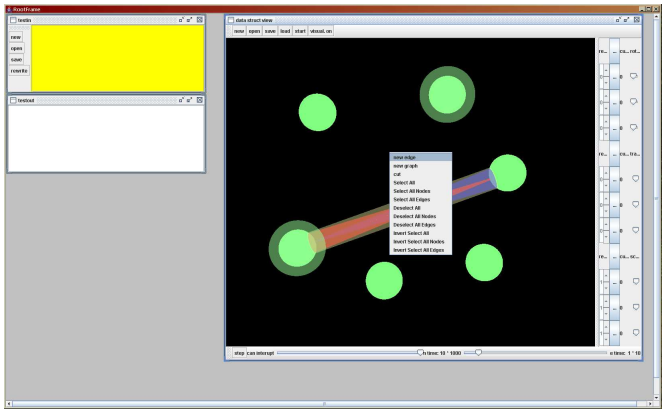
V konštruktore Viewer sa vytvára ViewerCanvas3D pomocou CanvasFactory a metódy create. Pozrime sa bližšie ako sa vytvára a inicializuje canvas[8], ktorý je potomkom triedy Canvas3D pochádzajúcej z knižníc 3DJava [20][21].

[8]plocha pre vykresľovanie grafickej scény

```
public class ViewerCanvas3D extends Canvas3D {

    //referencie pre poslucháčov udalostí
    //od okna, klávesnice a myšky
    java.awt.event.MouseListener mouseListener;
    java.awt.event.MouseMotionListener mouseMotionListener;
    java.awt.event.MouseWheelListener mouseWheelListener;
    java.awt.event.KeyListener keyListener;
    java.awt.event.ActionListener actionListener;
    //referencie pre tri druhy pop-up menu
    final JPopupMenu popUpMenuEdit;
    final JPopupMenu popUpMenuNewViewer;
    final JPopupMenu popUpMenuNode;
    //aktuálne súradnica myšky
    int mouseX;
    int mouseY;
    //referencie pre označený graf alebo vrchol
    Graph selectedGraph;
    Node selectedNode;

    //konštruktor
    public ViewerCanvas3D( GraphicsConfiguration config ) {
        super( config );
        //inicializácia pop-up menu pre editovanie
        popUpMenuEdit = new JPopupMenu();
        popUpMenuEdit.setLightWeightPopupEnabled(  false  );
        //inicializácia pop-up menu pre graf
        popUpMenuNewViewer = new JPopupMenu();
        popUpMenuNewViewer.setLightWeightPopupEnabled(  false  );
        //inicializácia pop-up menu pre vrchol
        popUpMenuNode = new JPopupMenu();
        popUpMenuNode.setLightWeightPopupEnabled(  false  );
    }
```

Kód jednotlivých poslucháčov ukazuje ako funguje 3D lokálizátor, editovanie grafov a zobrazovanie pop-up menu. (Obr. 4.2)

```
    .
    .
    .
    //vytvorím poslucháča pre myšku
    private void createMouseListener( final RootFrame rootFrame,
                final Graph graph ) {
        //vytvorím si referenciu na výstupnú konzolu
        final javax.swing.JTextArea output = rootFrame.west.output;
```



Obrázok 4.2: pop-up menu

```
//referenciu na koreň BSP stromu grafu
final PrimitivesModel primitivesModel = graph.primitivesModel;
//referenciu pre mapovanie stlačených kláves
final boolean[] keyPressed = rootFrame.kayPressed;
//vytvorím si potrebné objekty pre výpočty v 3D
final Point2d pixelMouseLocation = new Point2d();
final Point3d mouseLocation = new Point3d();
final Transform3D localToVworld = new Transform3D();
final Transform3D vWorldToImagePlate = new Transform3D();
final Point3d nodeLocation = new Point3d();
final Point2d pixelNodeLocation = new Point2d();
//transformácie z a do svetových súradníc pre 3D lokator
final Transform3D imagePlateTovWorld = new Transform3D();
final Transform3D inverseLocalToVworld = new Transform3D();
//pomocné objekty pre lokalizáciu hrany
final Point3d edgeLocation = new Point3d();
final Point2d pixelEdgeLocation = new Point2d();

mouseListener = new java.awt.event.MouseListener() {
    //poslucháč pre stlačenie myšky
    public void mousePressed( java.awt.event.MouseEvent e ) {
        //zistím ktoré tlačidlo myšky bolo stlačené
        int button = e.getButton();
        int nc;
        //opýtam si súradnice
        mouseX = e.getX();
        mouseY = e.getY();
        //na inicializujem pomocné boolové premenné pomocou
        //ktorých rozhodnem aké pop-up menu sa má zobraziť
        boolean NewViewerPoPUp = false;
        boolean NodePoPUp = false;
        //nastavím pixelMouseLocation objekt
```

```
pixelMouseLocation.set( ( double ) mouseX , ( double ) mouseY );
//opýtam si transformáciu zobrazovaného grafu
primitivesModel.getLocalToVworld( localToVworld );
//opýtam si transformáciu zo svetových súradníc
getVworldToImagePlate( vWorldToImagePlate );
//vymažem výstupnú konzolu
output.selectAll();
output.cut();
//zisťujem na ktorý vrchol 3D selektor ukazuje
for ( int i = 0; i < graph.getNodesCount(); i++ ) {
    //opýtam si polohu vrcholu
    nodeLocation.set( graph.getNodeLocation( i ) );
    //stransformujem ju do svetových súradníc
    localToVworld.transform( nodeLocation );
    //stransformujem ju do tzv. imagePlate súradníc
    vWorldToImagePlate.transform( nodeLocation );
    //získam 2D súradnice na priemetni
    getPixelLocationFromImagePlate( nodeLocation,
                        pixelNodeLocation );
    //ak transformované súradnice vrcholu sa prekrývajú
    //zo súradnicami myšky a
    if ( pixelNodeLocation.distance( pixelMouseLocation )
                        < Const.CLICK_ALLOWANCE ) {
        //ak bolo stlačené ľavé tlačidlo myši, tak
        if ( button == java.awt.event.MouseEvent.BUTTON1 ) {
            //označ vrchol a zobraz jeho Python kód
            //na výstupnú konzolu
            graph.clickNode( i );
            graph.showNodesCode( i );
            //ak bol stlačený kláves kontrol, tak
            if ( keyPressed[java.awt.event.KeyEvent.VK_CONTROL] ) {
                //označ vrchol a
                graph.invertSelectNode( i );
                //vypíš počet označených vrcholov
                output.append(" selected nodes count: "
                        + graph.selectedNodesCount + Const.newline);
            }
        //ak bolo stlačené pravé tlačidlo,
        //tak sa bude zobrazovať pop-up menu
        } else if ( button == java.awt.event.MouseEvent.BUTTON3 ) {
            //ak to bol graf, tak pre graf
            selectedNode = graph.isThisGraph( i );
            selectedGraph = ( Graph ) selectedNode;
            if ( null == selectedGraph ) {
                NewViewerPoPUp = false;
                selectedNode = graph.getNode( i );
```

```
                        //ak to bol vrchol, tak pre vrchol
                        popUpMenuNode.show( e.getComponent(),
                                        mouseX, mouseY );
                        NodePoPUp = true;
                    } else {
                        popUpMenuNewViewer.show( e.getComponent(),
                                        mouseX, mouseY );
                        NewViewerPoPUp = true;
                    }
                }

            }
        }
        .
        .
        .
```

Analogicky funguje detekcia polohy a selekcia hrán grafu v 3D. Ukážem ako pracuje poslucháč pre pohyb myšky potom, čo nastala detekcia polohy vrcholu grafu. Aby som mohol vrcholom pohybovať v rovine danej vrcholom a vektorom, ktorý je daný polohou kamery a vrcholom, budem potrebovať funkciu na výpočet priesečníka tejto roviny a vektora daným polohou kamery a súradnicami myšky.

```
Point3d intersectionRayAndPlane( Vector3d planeNormal, Point3d planePoint,
        Vector3d rayVector, Point3d rayPoint ) {

    Vector3d vector = new Vector3d();
    vector.sub( rayPoint, planePoint );

    Vector3d newRayVector = new Vector3d( rayVector );
    double t = - planeNormal.dot( vector ) / planeNormal.dot( newRayVector );
    newRayVector.scale( t );
    Point3d point = new Point3d( rayPoint );
    point.add( newRayVector );
    return point;
}
```

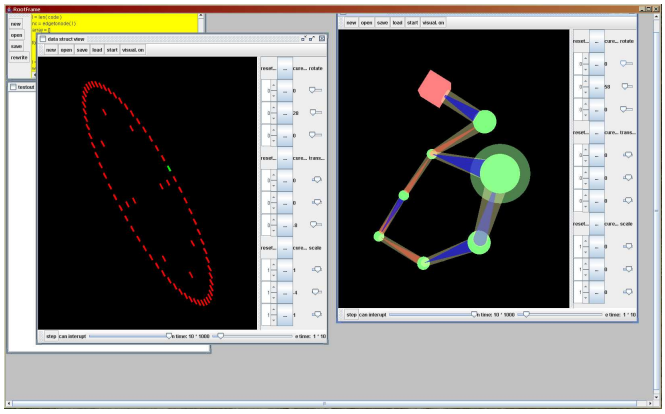
Teraz môžem napísať poslucháča pre pohyb myšky.

```
private void createMouseMotionListener( RootFrame rootFrame,
        final Graph graph ) {
    //referencia na koreň BSP stromu grafu
    final PrimitivesModel primitivesModel = graph.primitivesModel;
    //referencia na výstupnú konzolu
```

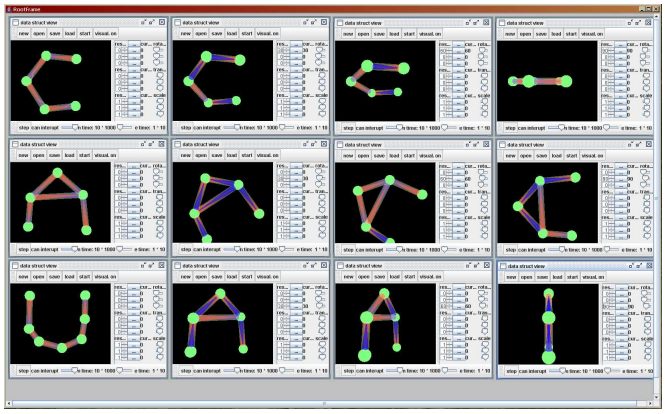
```
final javax.swing.JTextArea output = rootFrame.west.output;
//pomocné objekty
final Point2d pixelMouseLocation = new Point2d();
final Point3d mouseLocation = new Point3d();
final Point3d rightBottomCorner = new Point3d();
final Point3d rightTopCorner = new Point3d();
//pomocné transformácie a objekty
final Transform3D imagePlateTovWorld = new Transform3D();
final Vector3d toBottomVector = new Vector3d();
final Vector3d toTopVector = new Vector3d();
final Vector3d normalVector = new Vector3d();
final Point3d eyeLocation = new Point3d();
final Vector3d rayVector = new Vector3d();
//transformácie pre transformáciu zo svetových do
//lokálnych súradníc
final Transform3D localToVworld = new Transform3D();
final Transform3D inverseLocalToVworld = new Transform3D();

mouseMotionListener = new java.awt.event.MouseMotionListener() {
    public void mouseDragged( java.awt.event.MouseEvent e ) {
        if ( graph.getClicked() ) {
            //určí vektor kolmý na priemetňu pre pohyb
            //v smere danom vektorom
            pixelMouseLocation.set( ( double ) e.getX(),
                    ( double ) e.getY() );
            getPixelLocationInImagePlate( pixelMouseLocation,
                    mouseLocation );
            getPixelLocationInImagePlate( getWidth(), 0,
                    rightBottomCorner );
            getPixelLocationInImagePlate( getWidth(), getHeight(),
                    rightTopCorner );
            getCenterEyeInImagePlate( eyeLocation );
            //transformuj polohu kamery a priemetňu do svetových súradníc
            getImagePlateToVworld( imagePlateTovWorld );
            imagePlateTovWorld.transform( mouseLocation );
            imagePlateTovWorld.transform( rightBottomCorner );
            imagePlateTovWorld.transform( rightTopCorner );
            imagePlateTovWorld.transform( eyeLocation );
            //vytvor pomocné vektory
            toBottomVector.sub( rightBottomCorner, mouseLocation );
            toTopVector.sub( rightTopCorner, mouseLocation );
            //pomocou vektorového súčinu vytvor normálový vektor priemetne
            normalVector.cross( toBottomVector, toTopVector );
            normalVector.normalize();
            //vytvor vektor daný myškou a kamerou
            rayVector.sub( mouseLocation, eyeLocation );
```

```
                    rayVector.normalize();
                    //získaj transformáciu lokálnych súradníc grafu na
                    //svetové súradnice
                    primitivesModel.getLocalToVworld( localToVworld );
                    //vytvor inverznú transformáciu
                    inverseLocalToVworld.invert( localToVworld );
                    //posun myšou chytený vrchol na nové súradnice
                    Node node;
                    Point3d newPoint;
                    for ( int i = 0; i < graph.getNodesCount(); i++ ) {
                        node = graph.nodes.get( i );
                        if ( node.getClicked() ) {
                            localToVworld.transform( node.getLocation() );
                            newPoint = intersectionRayAndPlane( normalVector,
                                        node.getLocation(), rayVector, eyeLocation );
                            inverseLocalToVworld.transform( newPoint );
                            node.moveToAndUpdateScene( newPoint );
                        }
                    }
                }
            }
        };
    }
```

Trieda ViewerCanvas3D pozostáva ešte z createMouseWheelListener, pomocou ktorého môžeme označené vrcholy približovať alebo oddiaľovať od kamery v kolmom smere na priemetňu a definíciu všetkých troch druhov pop-up menu. Jednotlivé riešenia môže čitateľ nahliadnuť v priložených zdrojových kódoch. Viewer je nástroj na modelovanie, vizualizáciu a editáciu grafov (Obr. 4.3 a 4.4).



Obrázok 4.3: editovanie grafu



Obrázok 4.4: transformácie

4.5 Primitíva

Trieda Primitíva implementuje triedu PrimitivesFactory, ktorá slúži na definíciu grafických primitív v 3DJava. Poskytuje tak množinu 3D objektov, ktoré sú zobrazované pomocou balíka Viewer. Základnými prvkami sú vrcholy a hrany. Trieda definuje aj množstvo pomocných 3D telies potrebných pre vizualizáciu. Prvý si ukážeme vrchol .

```
public BranchGroup createNode(ColoringAttributes colorAttr) {
    BranchGroup bg = new BranchGroup();
    Sphere sphere = new Sphere(0.1f, Primitive.GENERATE_NORMALS, 120);
    sphere.setAppearance(createAppearanceNode(colorAttr));
    bg.addChild(sphere);
    return bg;
}
```

Vrcholom je guľa, ktorej nastavím veľkosť a farbu. Potom guľu pridám do BranchGroup, ktorú vrátim ako výsledok funkcie. Pozrime sa ako je spravené označenie vrcholu.

```
public BranchGroup createClickNode() {
    BranchGroup bg = new BranchGroup();
    Sphere sphere = new Sphere(0.50f, Primitive.GENERATE_NORMALS, 120);
    sphere.setAppearance(createAppearanceClickNode());
    bg.addChild(sphere);
    return bg;
}
```

Označený vrchol sa vyrobí tak, že na jeho súradniciach sa vykreslí väčšia, priesvitná guľa, ktorá ma inú farbu. Po odznačení vrcholu sa táto guľa zo scény odstráni. Analogicky sú zobrazované hrany ako valce, pričom požívam transformáciu kolmého valca do polohy vektora, daného dvoma vrcholmi pomocou kvaternionov[9]. Správa putujúca v hrane je ihlan

[9] nekomutativne rozšírenie komplexných čísel

umiestnený v priesvitnej hrane, transformovaný podobne ako hrana a navyše umiestnený medzi vrcholmi v pomere cesty z vrcholu A do vrcholu B.

```
public BranchGroup createLeftMessage() {
    BranchGroup bg = new BranchGroup();
    Cone cone  = new Cone(0.055f, 0.50f, Primitive.GENERATE_NORMALS, 120, 120,
            createAppearanceLeftMessage());
    bg.addChild(cone);
    return bg;
}
```

4.6 Balík Graf

Balík graf obsahuje triedy Cell, Node, Edge, Graph, Message a Parser. Ukážeme si význam jednotlivých tried, princípy vizualizácie paralelných algoritmov a overovanie vlastností na grafoch prostredníctvom aplikácie Grapher.

4.6.1 Trieda Cell

Trieda Cell je abstraktnou triedou reprezentujúca vrch abstrakcie dedičnej hierarchie, ktorá pozostáva z postupnosti Cell, Node, Edge, Graph. Je samostatným vláknom. Každá z týchto tried bola pri modelovaní rozdelená na tri základné podtriedy: Thread, Base a Graphic. V podtriedach Thread sú definované funkcie vrcholu ako vlákna, teda schopnosť samostatne spracovať správu, hrany tiež ako samostatného vlákna, teda schopnosť vizualizovať správy nezávisle, grafu ako vlákna, teda schopnosť vykonávať algoritmy reprezentované týmto grafom. V podtriedach Graphic sú definované funkcie pre vizualizáciu jednotlivých prvkov a v podtriedach Base sú definované funkcie pre manipuláciu z grafom, vrcholmi a hranami ako ich poznáme z teórie grafov.

```
public abstract class GraphicCell extends ThreadCell {

    //koreňová grupa objektu Cell a jeho transformačná matica
    public BranchGroup branchGroup;
    public TransformGroup transformGroup;
    //poloha
    protected Point3d location;
    //premenné nesúce informáciu čo sa bude zobrazovať
    private boolean showNumber;
    private boolean isActual;
    protected boolean drawed;
    protected boolean drawedClick;
    protected boolean drawedSelect;
    protected boolean drawedNote;
    private boolean exist;
    private boolean clicked;
```

```
private boolean selected;
private boolean noted;
//referencie na grafické primitíva pre Cell, označenie a zobrazenie čísla
BranchGroup Cell;
BranchGroup click;
BranchGroup select;
BranchGroup note;

public GraphicCell(String name, int number, RootFrame rootFrame, Graph owner) {
    super(name, number, rootFrame, owner);
    location = new Point3d();
    basicInit();
}

protected void basicInit() {
    init();
}

public GraphicCell(Point3d point3d, String name, int number,
        RootFrame rootFrame, Graph owner) {
    super(name, number, rootFrame, owner);
    location = new Point3d(point3d);
    locateInit();
}

protected void locateInit() {
    init();
    tinit();
}
//základná inicializácia semaforov pre vizualizáciu
protected void init() {
    drawed = false;
    drawedClick = false;
    drawedSelect = false;
    drawedNote = false;

    exist = true;
    clicked = false;
    selected = false;
    noted = false;
    //inicializácia grafických objektov 3DJava
    //povolenie zmeny scény modelovaním a transformáciami
    //počas vykresľovania a modelovanie BSP stromu
    branchGroup = new BranchGroup();
    branchGroup.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);
    branchGroup.setCapability(BranchGroup.ALLOW_CHILDREN_WRITE);
```

```
        branchGroup.setCapability(BranchGroup.ALLOW_DETACH);
        transformGroup = new TransformGroup();
        transformGroup.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);
        transformGroup.setCapability(BranchGroup.ALLOW_CHILDREN_WRITE);
        transformGroup.setCapability(BranchGroup.ALLOW_DETACH);
        transformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        transformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
        branchGroup.addChild(transformGroup);
    }

    protected void tinit() {
        //inicializácia lokalizácie
        Vector3d vector = new Vector3d(location);
        Transform3D tr = new Transform3D();
        tr.set(1, vector);
        transformGroup.setTransform(tr);
    }
    .
    .
    .
```

Trieda ďalej obsahuje metódy na vykreslenie objektu, označenie objektu či vykreslenie označených prvkov. Všetky operácie sú tvorené pridávaním a odoberaním objektov do a zo scény.

```
    .
    .
    .
    protected void draw(BranchGroup bg) {
        transformGroup.addChild(Cell);
        connect(bg);
        drawed = true;
    }

    protected void drawClick() {
        transformGroup.addChild(click);
        drawedClick = true;
    }
    .
    .
    .
protected void erase(BranchGroup bg) {
        disConnect(bg);
        drawed = false;
    }
```

```
    protected void eraseClick() {
        transformGroup.removeChild(click);
        drawedClick = false;
    }
    .
    .
    .
```

4.6.2 Trieda Node

Táto trieda predstavuje prvok vrchol grafu. Dedí funkcie od triedy Cell. Pozrime sa na jej definíciu a tiež na funkcionalitu spracovania správ. Najprv podtrieda Base.

```
public class BaseNode extends Cell {
    //počet prichádzajúcich a odchádzajúcich hrán
    public int leaveEdgesCount = 0;
    public int comeEdgesCount = 0;
    //referencie na susedné vrcholy a hrany
    protected java.util.List neighboursNodes;
    protected java.util.List neighboursEdges;
    //konštruktor
    public BaseNode(String name, int number, RootFrame rootFrame, Graph owner) {
        super(name, number, rootFrame, owner);
        neighboursNodes = new java.util.ArrayList();
        neighboursEdges = new java.util.ArrayList();
    }
    //konštruktor
    public BaseNode(Point3d point3d, String name, int number,
            RootFrame rootFrame, Graph owner) {
        super(point3d, name, number, rootFrame, owner);
        neighboursNodes = new java.util.ArrayList();
        neighboursEdges = new java.util.ArrayList();
    }
    //funkcia pre overenie čí ide o graf, alebo vrchol
    public boolean isThisGraph() {
        return false;
    }
    //metódy pre prístup k privátnym premenným
    public int getNeighboursNodesCount() {
        return neighboursNodes.size();
    }

    public int getNeighboursEdgesCount() {
        return neighboursEdges.size();
    }
```

```
    //metódy na pridávanie referencii o susedných hranách a vrcholoch
    public void addIncidentNode(Cell node) {
        neighboursNodes.add(node);
    }

    public void delIncidentNode(Cell node) {
        neighboursNodes.remove(node);
    }

    public void addIncidentEdge(Cell edge) {
        neighboursEdges.add(edge);
    }

    public void delIncidentEdge(Cell edge) {
        neighboursEdges.remove(edge);
    }
    .
    .
    .
```

Podtrieda ThreadNode dedí od podtriedy BaseNode.

```
public class ThreadNode extends BaseNode {
    //počítadlo prichádzajúcich správ
    public int messagesCounter = 0;
    //semafory pre kontrolu stavu vlákna
    public boolean bodyRun = false;
    public boolean bodyFinal = false;
    public boolean writeOutput = false;
    //vrchol má uložený kód, ktorý vlákno vykonáva
    public javax.swing.JTextArea pyCode = new javax.swing.JTextArea( "" );
    //referencia na Python interpreter
    PythonInterpreter interp = new PythonInterpreter();
    //doba spánku medzi kontrolou prichádzajúcej správy
    public int sleepTime;
    //semafor signalizujúci či môže byť vlákno zobudené
    //zo spánku ak prišla správa
    public boolean canDoInterupt = true;
    //konštruktor
    public ThreadNode( String name, int number, RootFrame rootFrame, Graph owner ) {
        super( name, number, rootFrame, owner );
        pyCode.setWrapStyleWord( true );
        sleepTime = Const.NODE_INIT_SLEEP_TIME;
    }
    //konštruktor
    public ThreadNode( Point3d point3d, String name, int number,
```

```
        RootFrame rootFrame, Graph owner ) {
    super( point3d, name, number, rootFrame, owner );
    pyCode.setWrapStyleWord( true );
    sleepTime = Const.NODE_INIT_SLEEP_TIME;
}
.
.
.
```

Podtrieda Thread definuje funkcie pre prácu so správami. Metóda sendMessage má parameter číslo hrany do ktorej správu pošle a parameter správu. Potom čo správu pošle po hrane, oznámi susednému vrcholu, aby prichádzajúcu správu spracoval. Správu pošle iba po odchádzajúcich hranách.

```
public void sendMessage( int i, Message m ) {
    Edge e;
    Node left;
    Node right;

    e = getIncidentEdge( i );
    left = e.getIncidentNodeLeft();
    right = e.getIncidentNodeRight();

    if ( this == left ) {
        if ( !e.leftDirExist ) return;
        e.addLeftMessage( m );
        right.messagesCounter ++;
        right.receiveMessage();
    } else {
        if ( !e.rightDirExist ) return;
        e.addRightMessage( m );
        left.messagesCounter ++;
        left.receiveMessage();
    }
}
```

Metóda receiveMessage spracuje prichádzajúcu správu.

```
public void receiveMessage() {
    //ak vlákno ešte nie je naštartované a nemá ešte algoritmus končiť
    if ( !bodyRun && !bodyFinal) {
        //tak naštartuje, správa sa spracuje v metóde run()
        bodyRun = true;
        start();
    }
```

```
        //ak je povolené prerušenie spánku, tak preruš
        if ( canDoInterupt ) interrupt();
    }

    public void sendMessage( Message m ) {
        for ( int i = 0; i < getNeighboursNodesCount(); i++ ) {
            sendMessage( i, m );
        }
    }
```

Metóda sendMessage bez parametrov generuje správu na základe vykonaného Python kódu vo vrchole.

```
    public void sendMessage() {
        //vykonaj kód
        runCode(interp);
        PyObject outmess;
        //vytiahni obsah premennej správa z interpretera
        outmess = interp.get( "message" );
        //ak správa nie je null, tak ju pošli všetkým susedom
        if ( outmess != null ) {
            for ( int i = 0; i < getNeighboursNodesCount(); i++ ) {
                sendMessage( i, new Message( outmess.toString() ) );
            }
        }
    }
    .
    .
    .
}
```

Podtrieda ThreadNode obsahuje ešte metódy na nahratie kódu do vrcholu a vykonanie nahratého kódu. Čitateľ môže tieto metódy nahliadnuť v priložených zdrojových kódoch. Podtrieda GraphicNode dedí od podtriedy ThreadNode. Implementuje metódy na vizualizáciu vrcholu.

```
public class GraphicNode extends ThreadNode {
    //objekty pre zobrazovanie čísla vrcholu
    protected Text3D noteText;
    protected ColoringAttributes colorAttr;
    //konštruktor
    public GraphicNode( String name, int number, RootFrame rootFrame,
            Graph owner ) {
        super( name, number, rootFrame, owner );
    }
```

```
    //konštruktor
    public GraphicNode( Point3d point3d, String name, int number,
            RootFrame rootFrame, Graph owner ) {
        super( point3d, name, number, rootFrame, owner );
    }
    .
    .
    .
    //inicializačné funkcie
    //Node preberá metódy na vykresľovanie od Cell
    //v inicializačných funkciách je nastavené aký objekt sa bude vykresľovať
    .
    .
    .
    //funkcia pre pohyb vrcholu v scéne
    public void moveToAndUpdateScene( Point3d point3d ) {
        //nastavenie lokalizácie
        setLocation( point3d );
        //nastavenie transformačnej matice
        tinit();
        int neighbouirsEdgesCount = getNeighboursEdgesCount();
        //prispôsobenie susedných hrán
        Edge e;
        for ( int i = 0; i < neighbouirsEdgesCount; i++ ) {
            e = getIncidentEdge( i );
            e.tinit();
        }
    }
```

Trieda Node, ktorá dedí od GraphicNode obsahuje konštruktor, metódy na generovanie nového suseda a rozsiahlu metódu run(), ktorá vykonáva kód v jazyku Python ako samostatné vlákno spracúva prichádzajúce správy. Opíšeme si iba najdôležitejšie časti.

```
public class Node extends GraphicNode {
    .
    .
    .
    public void run() {
        while ( bodyRun ) {
                .
                .
                .
                //pokiaľ sú správy na spracovanie
                while ( messagesCounter > 0) {
```

```
for ( int i = 0; i < getNeighboursEdgesCount(); i ++ ) {
    //získam referenciu na susednú hranu a vrchol
    e1 = getIncidentEdge( i );
    nl1 = e1.getIncidentNodeLeft();
    //získam správu
    if ( this == nl1 ) {
        recm = e1.getRightMessage();
    } else {
        recm = e1.getLeftMessage();
    }

    if ( recm == null ) {
        //ak mám povolený výpis, tak vypíšem na konzolu,
        //že z tejto hrany správa neprišla
        if ( writeOutput ) output.append( this.getName()
                + " empty queue in: " + e1.getName()
                + Const.newline );
    } else {
        //ak mám povolený výpis, tak vypíšem na konzolu,
        //že z tejto hrany správa prišla
        if ( writeOutput ) output.append( this.getName()
                + " received message: " + recm.getMessage()
                + "from: " + e1.getName() +Const.newline );
        //znížim počítadlo správ o jedna
        messagesCounter --;
        //odovzdám správu, na inicializujem hodnoty
        //premenných na základe správy
        runCode( interp, recm.getMessage() );
        //vykonám kód tohto vlákna
        runCode( interp );
        //opýtam si premennú action, v ktorej sa nachádza
        //typ udalosti, ktorá nasleduje po spracovaní kódu
        //v jazyku Python
        PyObject paction = interp.get( "action" );
        String jaction = "";
        try {
            jaction = paction.toString();
        } catch (NullPointerException e) {
            System.out.print( e.toString()
                    + " null pointer - action");
        }
        if ( jaction == "" ) {
            //ak je action prázdny, tak nerobím nič
            //System.out.print( "no action required"
            //+ Const.newline );
        } else {
```

```
                            //ak obsahuje pod slovo message
                            if ( Pattern.matches(".*message.*", jaction) ) {

                                PyObject pmessage = interp.get( "message" );
                                //System.out.print( pmessage + Const.newline);
                                String jmessage = "";
                                try {
                                    jmessage = pmessage.toString();
                                } catch (NullPointerException e) {
                                    System.out.print( e.toString()
                                    + " null pointer - message");
                                }
                                //tak vytiahnem z interpretra premennú
                                //message a rozpošlem ju ďalej
                                .
                                .
                                .
        try {
            //po spracovaní všetkých správ vlákno zaspí
            Thread.sleep( sleepTime );
        } catch ( InterruptedException e2 ) {
            if ( writeOutput ) output.append( this.getName()
                    + " interupted exception in run" + Const.newline );
        }
    }
```

Funkcia run() spracuje správy spôsobom, že ak dostane správu, ktorá nie je nič iné ako krátky program v jazyku Python, tak tu správu vykoná na interpreteri. Tým na inicializuje vstupné premenné. Potom vykoná hlavný kód vo vrchole, ktorý vyprodukuje novú premennú action. Podľa obsahu premennej action vlákno vie, čo ma robiť ďalej. Jednou z možností je že kód vygeneroval novú správu a tu rozpošle ďalej. Takýmto spôsobom vykonávam paralelný algoritmus, nakoľko každý vrchol je samostatné vlákno. Môže v hlavnom kóde vykonať jednoduchý príkaz, ale aj malý či zložitejší program. Úroveň abstrakcie rozdelenia príkazom do jednotlivých vrcholov, alebo vykonávanie programov v jednotlivých vrcholoch si určíme na základe toho, aké typy správ chceme vizualizovať. Správa môže niesť informáciu od najjednoduchšej, napríklad hodnotu premennej i pre for cyklus, až po hodnotu premennej, ktorá má význam z pohľadu vyššej abstrakcie. Najvyššia úroveň môže byť aj taká, že sa celý algoritmus vykoná v jednom vrchole, čo by ale znamenalo, že by som nemal čo vizualizovať a algoritmus by sa stal sekvenčným. Z tohto pohľadu sa smerom dole zvyšuje paralelizmus a šírka vizualizácie. Opačným smerom sa atomické operácie skladajú do malých programov a tie sa vykonávajú v jednotlivých vláknach. Vizualizujú sa informácie potrebné pre prenos dát medzi vláknami. Pre takýto nástroj je najvhodnejšie zvoliť vizualizáciu niekde uprostred úrovne abstrakcie. Rozdeliť algoritmus na jednotlivé logické celky a vizualizovať prenos dát medzi nimi. Takéto správy reprezentujú skutočné objekty s významom.

4.6.3 Trieda Edge

Trieda Edge v jej Podtriedach Base, Thread a Graphic obsahuje metódy pre pridávanie hrany do grafu a previazanie so susednými vrcholmi. Hrana je buď obojsmerná, alebo môže mať daný smer v akom môžu putovať správy. Podtrieda GraphicEdge obsahuje metódy na inicializáciu grafických primitív pre samotné hrany ako priesvitné valce, grafické primitíva pre správy ako ihlany putujúce v hranách. Z podtriedy graf si ukážeme transformáciu valca do polohy, aby presne spájal dva vrcholy ako gule. O túto transformáciu sa stará funkcia init()

```
protected void tinit() {
    //opýtam si polohu susedných vrcholov
    left =  getLeftLocation();
    right = getRightLocation();
    //nastavím sa do polohy medzi vrcholy s pomerom podľa parametra
    location.interpolate( left, right, Const.BASE_EDGE_LOCATION_PARAMETER );
    //vytvorím pomocný vektor pre posun
    Vector3d translation = new Vector3d( location );
    //získam škálu
    double scale = left.distance( right );
    Vector3d line = new Vector3d();
    //vytvorím smerový vektor
    line.sub( right, left );
    line.normalize();
    //vytvorím vektor v smere osi y
    //v tomto smere sa zatiaľ nachádza valec
    Vector3d up = new Vector3d( 0.0d, 1.0d, 0.0d );
    up.normalize();
    //zistím odklon valca od smerového vektora
    double alfa = - line.angle( up ) / 2.0d ;
    //vytvorím kolmý vektor na smerový a up vektor
    Vector3d ax = new Vector3d();
    ax.cross( line, up );
    ax.normalize();
    //vytvorím rotačný quaternion
    double sinAlfa = java.lang.Math.sin(  alfa  );
    quat4d = new Quat4d(  sinAlfa * ax.x, sinAlfa * ax.y, sinAlfa
                    * ax.z, java.lang.Math.cos( alfa ) );
    quat4d.normalize();
    //vytvorím vektor pre škálovanie
    //set size
    Vector3d scaleVector3D = new Vector3d( 1.0d, scale, 1.0d );
    //vytvorím maticu pre škálovanie
    transformScale3D.setScale( scaleVector3D );

    Transform3D transform3D = new Transform3D();
    //vytvorím maticu transformácie pre valec
```

```
        transform3D.set( quat4d, translation, 1.0d );
        //upravím škálovaním
        transform3D.mul( transformScale3D );
        //nastavím transformáciu pre valec resp. hranu
        transformGroup.setTransform( transform3D );
        //nastavím transformáciu posunu pre správu
        Transform3D noteTr3D = new Transform3D();
        noteTr3D.setTranslation(translation);
        noteTG.setTransform(noteTr3D);
        //nastavím pozíciu správ
        setLeftMessagePossition( leftMessagePossition );
        setRightMessagePossition( rightMessagePossition );
    }
```

O vykresľovanie správ putujúcich po hranách sa stará metóda run(). Vykresľovanie sa deje nezávisle v každej hrane ako vlákne. Podrobnejšie princípy vykresľovania môže čitateľ nahliadnuť v priložených zdrojových kódoch.

4.6.4 Trieda Graf

Trieda Graf implementuje základné metódy pre prácu z grafom. Medzi tie patrí pridanie vrcholu, odobratie vrcholu, pridanie hrany, odobratie hrany a metódy na označovanie prvkov grafu. Tieto základné metódy sú súčasťou triedy BaseGraph. Jej potomkom je trieda ThreadGraph, ktorá implementuje metódy na naštartovanie algoritmu na grafe, nahratie Python kódu do označených vrcholov, metódy pre štart a stop vizualizácie posielania správ po hranách grafu a metódy pre nastavenie možnosti prerušení a dĺžky spánku vlákien vo vrcholoch a hranách grafu, čím sa dá ovplyvniť efektivita vizualizácie putovania správ a rýchlosť priebehu distribuovaného algoritmu až na úroveň krokovania. Od triedy ThreadGraph dedí jej vlastnosti trieda GraphicGraph, ktorá implementuje metódy pre vykresľovanie grafu. Realizované sú pomocou metód na vykresľovanie vrcholov a hrán. Užitočné sú metódy pre selekciu a inverziu selekcie na vrcholoch a hranách spolu z metódou cut(), ktorá vymaže označené prvky. Hlavná trieda Graph dedí metódy a vlastnosti všetkých predchádzajúcich tried. Dedí aj od triedy Node, čo je vrchol grafu a preťažuje metódu run(), ktorá umožňuje grafu prijímať a spracovávať správy ako vrchol. Tento vrchol ako graf v jednom, môže na podnet špeciálnej správy generovať svoju vlastnú vnútornú štruktúru vrcholov ako ďalších pod - grafov. Správa obsahuje informáciu o tom aký graf sa má skonštruovať a na aké súradnice sa má umiestiť. Tieto informácie sa vygenerovali generátorom grafov a mapovacou funkciou. O tých si povieme v nasledujúcej kapitole.

Kapitola 5

Vizualizácia vlastností grafov

Jedna naozaj cenná vec je intuícia.
Albert Einstein

5.1 Generátor grafov

Generátor grafov je tvorený vrcholom grafu, ktorého kód v jazyku Python dokáže generovať binárne vektory reprezentujúce grafy v lexikografickom poradí. Generovanie je možné začať od ľubovoľnej konfigurácie. Vektorom je trojuholníková matica susedností nad diagonálou. Podrobnejší popis môže čitateľ nahliadnuť v [5]
Popíšeme si generátor grafov v jazyku Python. Funkcia pregene ak dostane na vstup vektor reprezentujúci graf, tak na výstup dá vektor reprezentujúci nasledujúci graf v lexikografickom usporiadaní.

```
\\funkcia ppazz posúva v reťazci s najľavejšiu jednotku za ktorou nasleduje
\\nula a jedno miesto doprava parameter x vypočítava funkcia pposi(s)
def ppazz(x, s):
        if s=="":
                return ""
        elif x[1]>0:
                if x[0]>0:
                        return "1" + ppazz([x[0]-1, x[1]-1], s[1:])
                else:
                        return "0" + ppazz([x[0], x[1]-1], s[1:])
        else:
                return "1" + s[1:]
\\funkcia pplus spočítava dvojrozmerne vektory x a y, pričom prenáša v
\\chvoste informačný bit o zastavení rekurzie pre funkciu pposi
def pplus(x, y):
        return [x[0] + y[0], x[1] + y[1], y[2]]
\\funkcia pposi vyrátava výskyt prvej jednotky na posun pre funkciu ppazz
def pposi(s):
        if s[1:] == "":
```

```
                return [0, 1, 1]
        elif s[0:1]=="0":
                return pplus([0, 1, 0], pposi(s[1:]))
        elif s[0:2]=="10":
                return [0, 1, 0]
        elif s[0:2]=="11":
                return pplus([1, 1, 0], pposi(s[1:]))
\\funkcia vyráta počet dvojíc o koľko sa má vektor predlžiť po pridaní
\\nového vrcholu
def edgetonode(x):
        if x == 0:
                return 0
        if x == 1:
                return 2
        z=0
        k=1
        while z<x:
                z = z + k
                k = k + 1
        return k
\\funkcia generuje nasledujúci graf v lexikografickom usporiadaní
def pregene(s):
        if s == "":
                return "0"
        \\zistim dĺžku vektora
        max = len(s)
        \\zistim počet jednotiek vo vektore
        c = s.count("1")
        \\vypočítam parametre výskytu jednotky pre posuv
        a = pposi(s)
        \\ak v chvoste a[2] som nemal poslednú konfiguráciu pre daný počet
        \\vrcholov a hrán, tak generujem ďalšiu konfiguráciu
        if a[2]==0:
                return ppazz(a, s)
        \\ak som v poslednej konfigurácii
        else:
                \\pre tento počet vrcholov
                if c==max:
                        \\generujem bezhranný graf so zvýšeným počtom vrcholov
                        return "0" * ( max + edgetonode (max) )
                \\pre tento počet hrán
                else:
                        \\pridám hranu
                        return "1" * c + "1" + "0" * (max - c - 1)
\\zapíšem Python kód do premennej message v ktorom priraďujem
\\premennej code konfiguráciu
```

```
message = 'code ="' + code + '"'
\\vygenerujem konfiguráciu, ktorú pošlem pri ďalšom zavolaní
code = pregene( code )
\\premennú action nastavím, tak aby sa správa poslala po
\\odchadzajucej hrane z vrcholu
action = "message"
```

Za generátor grafov musí byť umiestnený stopér, ktorý zastaví generovanie grafov po vygenerovaní požadovanej konfigurácie. Ukážeme si jednoduchý stopér, ktorý zastaví pre posielanie správ ak vygenerovaný graf má viac ako štyri vrcholy. Vektor pre grafy čo do mohutnosti vrcholov štyri sú dĺžky šesť.

```
\\ak je počet párov viac ako šesť
if len( code ) > 6:
        \\žiadna akcia nenastane
        action = ""
else:
        \\inak pre posielaj správy ďalej
        message = 'code = "' + code + '"'
        action = "message"
```

5.2 Mapovanie

Mapovač je vrchol grafu, ktorý dokáže prijať správu s informáciou o konfigurácii skúmaného grafu a odoslať túto konfiguráciu spolu z informáciou, kam sa takýto graf ako vrchol alebo bod, bude v rovine, alebo v priestore zobrazovať. Ako príklad si ukážeme umiestňovač do roviny, presnejšie jeho časť kódu v jazyku Python. Grafy zobrazuje do sústredných kružníc. Najprv si ukážeme kód inicializačného vrchola, ktorý posiela inicializačnú správu pre umiestňovač.

```
message = 'code = ""; x = "0.0"; y = "0.0"; z = "-5.0";
        pc = 0; ec = 0; nc = 0; angle = 0'
action="message"
```

Umiestňovač má po spracovaní message inicializované pre menné: x, y, z, pc, ec, nc a angle.

```
\\zistim počet párov vrcholov v grafe
paco = len( code )
\\ak sa zmení počet párov - zmení sa aj počet vrcholov
if pc < paco:
        \\určím počet párov
```

```
        pc = paco
\\určím počet vrcholov
nc = edgetonode( pc )
\\určim krok uhla
alfa = 2 * pi / pow( 2, pc )
\\vyrátam súradnice
x = str( float( nc ) * cos( angle ) )
y = str( float( nc ) * sin( angle ) )
\\vytvorím správu obohatenú o mapovanie grafu do roviny
message = 'code ="' + code + '";
        x = "' + x + '"; y = "' + y + '"; z = "' + z + '"'
\\posuniem uhol pre nasledujúci graf
angle = angle + alfa
\\nastavím akciu aby sa poslala správa
action = "message"
```

5.3 Overovanie vlastností na grafe

Overovač vlastností na grafe je vrchol, ktorý dokáže po prijatí správy z konfiguráciou grafu overiť, či tento graf spĺňa skúmanú vlastnosť, alebo nie. Generuje správu, ku ktorej je k danej konfigurácii priradená farba, ktorá odpovedá tomu, či graf skúmanú vlastnosť má alebo nie. Ako príklad si ukážeme kód overovača pre kubické grafy. Táto funkcia overuje či každý vrchol v grafe má troch susedov. Medzi výsledky pre jednotlivé vrcholy ukladá do pola array. Výpočet prebieha priamo nad vektorom, ktorý reprezentuje vygenerovaný graf.

```
 prefix = 1;
postfix = 1;
level = 0;
\\zisti počet dvojíc a vrcholov
l = len( code )
nc = edgetonode( l )
array = []
\\nainicializuj pole array
for i in range( nc ):
    array.append( 0 )

i = 0
while i < l:
        print code[i]
        if code[ i ] == "1":
                array[ level ] = array[ level ] + 1
                array[ postfix ] = array[ postfix ] + 1
        postfix = postfix + 1;
        if postfix == nc:
```

```
                prefix = prefix + 1
                postfix = prefix
                level = level + 1
        i = i + 1
\\nainicializuj farbu na zelenú
r = "0.0"
g = "1.0"
b = "0.0"
\\ak niektorý vrchol nemá troch susedov
for i in range( nc ):
        if array[ i ] != 3:
                \\nastav farbu na červenú
                r = "1.0"
                g = "0.0"
                b = "0.0"
                break
\\pribal informáciu o farbe do premennej message
message = 'code ="' + code + '"; x = "' + x + '"; y = "' + y + '";
        z = "' + z + '"; r = "' + r + '"; g = "' + g + '"; b = "' + b + '"'
\\nastav action aby bola vrchol odoslal vygenerovanú správu
action = "message"
```

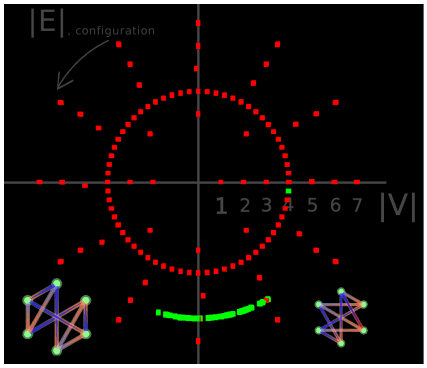
Vrchol grafu, ktorý je zároveň tiež grafom, vie prijímať správy a jeho kód pribaľuje k správe akciu generuj. Po prijatí takejto správy, vytvorí nový graf podľa obdŕžanej konfigurácie, umiesti ho do priestoru podľa obdŕžaných súradníc a priradí mu farbu podľa obdŕžaných premenných r, g, b. Štruktúra tohto grafu je nástroj na vizualizáciu vlastností grafov. Editor umožňuje rekurzívne prezerať štruktúru grafu ako vrcholu v inom grafe. Takto môžeme vidieť aj štruktúru grafu, ktorý si vyberieme na zobrazenie podľa toho, či už danú vlastnosť spĺňa, alebo nie.

5.4 Výsledky

Do výsledkov práce zaradujem výber obrázkov, ktoré sa dajú pomocou aplikácie Grapher generovať do priestoru, alebo roviny. Použité boli dva druhy mapovacích vrcholov a niekoľko druhov overovacích vrcholov. Výpočty sú realizované na počítači s hyper-thread-ovým procesorom s frekvenciou 2.8 GHz a pamäťou 512 MB.

5.4.1 Rovinná vizualizácia

Na obrázku (Obr. 5.1) je 2D vizualizácia vlastností grafov. Overovanou vlastnosťou je kubickosť. Grafy sú zobrazované ako body v rovine, umiestnené vzhľadom na počet vrcholov grafu a konfiguráciu. Tú tvorí počet hrán v grafe a spôsob ich rozmiestnenia. Môžme si všimnúť vzniknuté zhluky grafov, ktoré sú kubické. Generovanie obrázku trvalo približne dva dni. Počiatočná konfigurácia je graf s jedným vrcholom. Posledná konfigurácia je kompletný graf o siedmych vrcholoch.

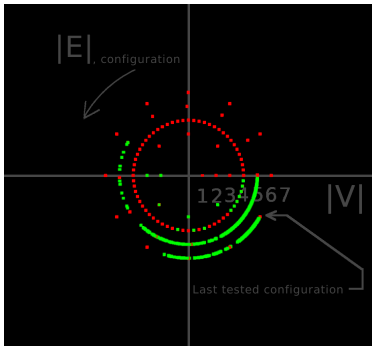


Obrázok 5.1: kubické grafy

Obrázok (Obr. 5.2) ukazuje preverenie Hamiltonovskej vlastností na základe vety z teórie grafov od Diraca. Na obrázku si znova môžeme všimnúť zhluky grafov, ktoré túto vlastnosť spĺňajú. Vizualizácia je generovaná na grafoch s počtom vrcholov od jedna až po šesť. Overovanie bolo pozastavené v konfigurácii tvorenej šiestimi vrcholmi a umiestnenej v poslednom oktante z dôvodu vyčerpania systémových prostriedkov. Uvediem vetu z teórie grafov, pomocou ktorej je zostrojená nutná Hamiltonova podmienka. Tá je využitá pri tejto vizualizácii.

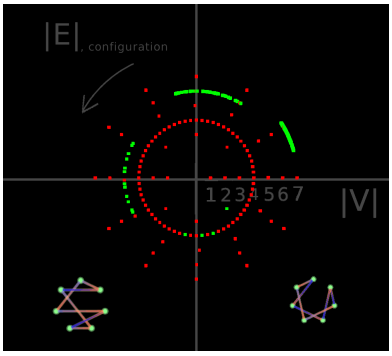
Veta Dirac. *Graf je Hamiltonovský ak pre minimálny stupeň grafu G s počtom vrcholov n platí $\delta(G) > n/2$.*

Dôkaz tohto tvrdenia môže čitateľ nahliadnuť v Literatúre o teórii grafov.



Obrázok 5.2: Hamiltonovské grafy

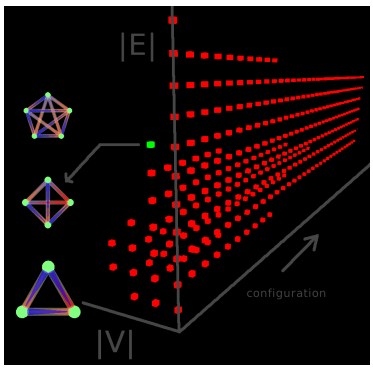
Na poslednom obrázku v obore 2D vizualizácie je preverenie grafov, ktoré sú kružnicami. Znova si môžeme všimnúť vzniknuté zhluky grafov, ktoré danú vlastnosť spĺňajú. Generovanie obrázku trvalo približne dva dni. Počiatočná konfigurácia je primitívny graf. Konečnou konfiguráciou je kompletný graf o siedmich vrcholoch.



Obrázok 5.3: pracovná plocha

5.4.2 Priestorová vizualizácia

Na obrázku (Obr. 5.4) je 3D vizualizácia kubických grafov do počtu vrcholov 5. Takýto obrázok je možné vygenerovať do niekoľkých minút. Vhodnými transformáciami je možne priblížiť a následne zobraziť konkrétne mnou vybrané grafy. Na obrázku je jediný kubický graf a to kompletný graf o štyroch vrcholoch.



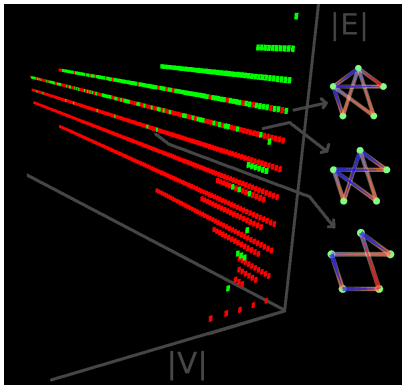
Obrázok 5.4: kubické grafy

Obrázok (Obr. 5.5) znázorňuje Hamiltonovské grafy do počtu vrcholov päť. Je možné vidieť že od istých počtov vrcholov a hrán je už väčšina grafov Hamiltonovských.

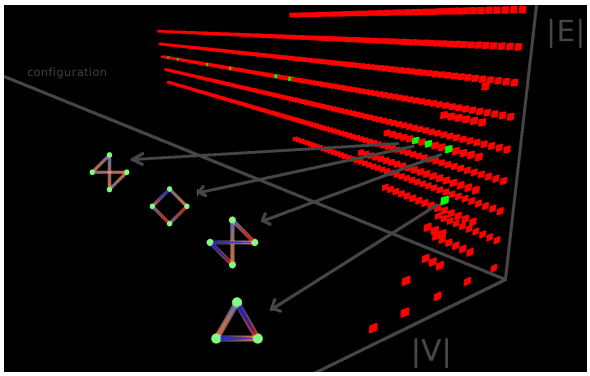
Na poslednom obrázku (Obr. 5.6) je vizualizácia grafov, ktoré sú kružnice. Vyobrazené sú všetky štruktúry grafov, ktoré sú kružnice o počte vrcholov tri a štyri.

5.5 Záver

Cieľom tejto práce bolo navrhnúť a implementovať aplikáciu, pomocou ktorej je možné vizualizovať vlastností grafov a algoritmov na grafoch. Výsledné vizualizácie svedčia o tom, že cieľ práce bol splnený. Aplikácia je poňatá dosť všeobecne, aby bolo možné sledovať priebeh algoritmu, ktorý je naprogramovaný v jazyku Python. Primárnou témou tejto práce je vizualizácia vlastností grafov. Digitálna príloha obsahuje zdrojové kódy aplikácie a aj vstupné a inicializačné súbory pre aplikáciu. Vstupné súbory vo formáte XML predstavujú graf, ktorý reprezentuje algoritmus overovania vlastností grafov. Tento algoritmus



Obrázok 5.5: Hamiltonovské grafy



Obrázok 5.6: kružnice

prebieha distribuovane ak použijeme multi-thread-ové, alebo viacprocesorové počítače. Prebiehajúci algoritmus je možné vizualizovať zobrazovaním správ, ktoré putujú medzi jednotlivými vláknami. Kód jazyka Python, ktorý sa vykonáva v jednotlivých vláknach môžeme vidieť na výstupnej konzole. Vygenerované výsledky vlastností grafov je možné sledovať v samostatnom okne. Aplikácia umožňuje vytvárať nové overujúce vlákna a takto overovať nové vlastnosti. Pri troche trpezlivosti sa pri jednoduchých overovacích funkciách môžeme za dobu dvoch dní dopracovať ku grafom až o siedmich vrcholov. Aplikácia umožňuje interaktívne sledovanie a aj transformovanie priebežného výsledku, čo ale vyžaduje uchovávať medzivýsledky v operačnej pamäti počítača. Týmto sa aplikácia stáva obmedzená, vzhľadom na systémové prostriedky. Rezervy aplikácie sú v totálnom pozastavení a znovu spustení algoritmu s prípadným sledovaním konkrétnej správy, ktorá ak sa dostane do vrcholu grafu a začne sa spracovávať, tak by bol umožnený pri spomalenom behu výpis riadkov kódu a obsahu premenných na konzolu. Z takto navrhnutej aplikácie, by sa takáto funkcionalita s trochou trpezlivosti dala dosiahnuť. Pre potenciálnych pokračovateľov tejto témy, ktorí by sa radi dostali za hranicu siedmych vrcholov by som odporúčal nasledovný postup: Vygenerovať si databázu vektorov, ktoré reprezentujú grafy do tabuľky, alebo tabuliek podľa počtu vrcholov v grafe. Druhým stĺpcom by bolo poradové číslo grafu. Ten by bol aj primárnym kľúčom pre túto tabuľku. S použitím mapovacích funkcií by sa vytvárali ďalej nové tabuľky, ktoré by obsahovali identifikačné číslo (id) grafu, ako kľúč a súradnice, kam sa daný graf má vykresliť. V ďalšom nezávislom procese, by sa pomocou množiny overovacích funkcii pre graf, pomo-

cou ktorej môžem overiť danú vlastnosť, vytvárali analogicky nové tabuľky, reprezentujúce vlastnosti na grafoch pozostávajúcich z id grafu a výsledku overenia. Vizualizácia by potom spočívala vo vybraní množiny grafov, pre ktorú už mám všetky potrebné informácie o vlastnostiach a umiestneniach a vo vykreslení grafov na základe zvolených tabuliek lokalizácie a vlastnosti. Pri skúmaní zaujímavých zhlukov, alebo skupín grafov, je možné spraviť zúženie oboru grafov pred testovaním, pri ktorom môžeme náhodne hľadať graf s danou vlastnosťou a potom vyšetriť množinu grafov v jeho okolí. Otvorená ostáva aj problematika zavedenia iných generátorov, ktoré môžu grafy s rovnakým počtom vrcholov a hrán generovať v iných poradiach, alebo použitie napríklad generátora pre neizomorfné grafy.

Literatúra

[1] Hans, M. K. *Slávne Fotografie*. Slovart 2003. ISBN 382282576X.

[2] Sedláček, J. *Kombinatorika v teórii a praxi - Úvod do teórie grafov*. pp. 24 - 26. Bratislava.

[3] Plesnik, J. *Grafové Algoritmy*. pp. 12 - 17. Bratislava: VEDA Vydavateľstvo slovenskej akadémie vied 1983.

[4] Hearn, D. - Baker, P., EUROGRAPHIC '91. *Scientific Visualisation*. pp. 2 - 6. Viedeň 1991.

[5] Stibrányi, E., Diplomová práca. *Vizualizácia teórie grafov*. Bratislava 1997.

[6] Cruz, I. F. - Tamassia R. *Graph Drawing Tutorial*. 2006. [5. 5. 2007]. http://www.cs.brown.edu/ rt/papers/gd-tutorial/gd-constraints.pdf

[7] Battista, D. G. - Tamassia, R. - Eades P. - Tolis, L. G. *Algorithms for drawing graphs an Annotated Bibliography*. 1994. [5. 5. 2007]. http://www.cs.brown.edu/ rt/gd-biblio.html

[8] Purchase, H.C. *Evaluating Graph Drawing Aesthetics*. pp 145-178. New York: Idea Group Publishing. 2004.

[9] Caldwell, C. K. *Graph Theory Tutorials*. 1995. [5. 5. 2007]. http://www.utm.edu/departments/math/graph/

[10] Tamassia, R. *Graph Drawing*. 2004. [5. 5. 2007]. http://www.ics.uci.edu/ eppstein/gina/gdraw.html

[11] Open GIS Consortium, *Geography Markup Language* 2004. [5. 5. 2007]. http://xml.coverpages.org/ni2004-03-26-a.html

[12] Hugunin, J. *What is Jython*. 2000. [5. 5. 2007]. http://www.jython.org/docs/whatis.html

[13] Bray, T. - Paoli, J. - Sperberg-McQueen, C. M. *Extensible Markup Language*. 1998. [5. 5. 2007]. http://www.w3.org/TR/1998/REC-xml-19980210.html

[14] Punin, J. - Krishnamoorthy, M. *eXtensible Graph Markup and Modeling Language*. 2001. [5. 5. 2007].
http://www.cs.rpi.edu/ puninj/XGMML/

[15] SUN. *JavaTM 2 Platform Standard Edition 5.0 API Specification*. 2004. [5. 5. 2007].
http://java.sun.com/j2se/1.5.0/docs/api/overview-summary.html

[16] SUN. *Java Technology Forums*, 2006. [5. 5. 2007].
http://forum.java.sun.com/index.jspa

[17] SUN. *How to Use Layered Panes*, 2006 [5. 5. 2007].
http://java.sun.com/docs/books/tutorial/uiswing/components/layeredpane.html

[18] University of British Columbia. *How to Use Menus*, 2006. [5. 5. 2007].
http://www.iam.ubc.ca/guides/javatut99/uiswing/components/menu.html

[19] SUN. *A Visual Index to the Swing Components*, 2006 [5. 5. 2007].
http://java.sun.com/docs/books/tutorial/uiswing/components/components.html

[20] SUN. *Java 3d*, 2006. [5. 5. 2007].
http://download.java.net/media/java3d/javadoc/1.4.0-latest/index-all.html
http://www.math.nyu.edu/phd_students/meyersr/links/java3d/index-all.html

[21] Kuželka, O. *Java a 3D grafika*. 2003. [5. 5. 2007].
http://interval.cz/clanky/java-a-3d-grafika-uvod/

[22] dom4j. *Parsing XML*. 2005. [5. 5. 2007].
http://www.dom4j.org/guide.html

[23] Chermside, M. *How do I invoke Python code from Java*. 2002. [5. 5. 2007].
http://www.faqts.com/knowledge_base/view.phtml/aid/19444/fid/1102