VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE ALGORITMŮ PRO PLÁNOVÁNÍ CESTY

PATH PLANNING ALGORITHMS VISUALISATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **JAKUB RUSNÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAROSLAV ROZMAN, Ph.D.**
SUPERVISOR

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Rusnák Jakub, Bc.**
Obor: Inteligentní systémy
Téma: **Vizualizace algoritmů pro plánování cesty**
Path Planning Algorithms Visualisation
Kategorie: Umělá inteligence

Pokyny:

1. Nastudujte algoritmy pro plánování cesty používané v robotice (roadmapy, cell decomposition, pravděpodobnostní, atd.).
2. Navrhněte webový applet, který umožní vizualizaci funkce algoritmů. Počítejte i s přidáváním algoritmů v budoucnu. Applet musí umět vytvořit překážky a zadat start a cíl. Dále musí být možné algoritmus krokovat a případně přeskakovat jednotlivé fáze. Dále musí applet obsahovat pseudokód, kde se bude zobrazovat právě vykonávaná část algoritmu.
3. Navržený applet včetně vybraných algoritmů (např. pravděpodobnostní) implementujte. Vytvořte i stránku s teoretickým popisem algoritmu.
4. Proveďte otestování a vytvořte manuál pro snadné přidávání dalších algoritmů.

Literatura:

- Principles of Robot Motion, Howie Choset et al., 2005, ISN 0-262-03327-5

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rozman Jaroslav, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce se věnuje vytvoření knihovny pro vizualizaci algortimů. Knihovna ulehčuje tvorbu uživatelského rozhraní pro aplikace s algoritmami. Její funkce je demonstrovaná na nekolika algoritmech pro plánovaní cesty. Výsledné aplikace jsou prezentovány na webové stránce.

Abstract

Thesis describes library for algorithm vizualization. It helps to create user interface for application with algorithms. It's usage is demonstrated on some pathfinding algorithms. These applications are presented on web page.

Kľúčové slová

java, plánování cesty, road mapa, pravdepodobnostní algoritmy, grafové algoritmy, vizualizace

Keywords

java, path planning, roadmap, probabilistic algorithms, graph algorithms, vizualization

Citácia

RUSNÁK, Jakub. *Vizualizace algoritmů pro plánování cesty.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rozman Jaroslav.

Vizualizace algoritmů pro plánování cesty

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Jaroslava Rozmana, Ph.D.

. . . . . .
Jakub Rusnák
15. mája 2017

Obsah

1 Úvod **3**

2 Algoritmy pre plánovanie cesty **5**
2.1 Generovanie a spracovanie konfigurácií 6
2.2 Pravdepodobnostné algoritmy 9
2.2.1 Pravdepodobnostná road mapa 11
2.2.2 Expanzívne stromy 13
2.2.3 Rapídne sa rozširujúce stromy 14
2.3 Kombinatorické plánovanie pohybu 16
2.3.1 Vertikálna bunková dekompozícia 16
2.3.2 Voronoiove diagramy 16
2.3.3 Grafy viditeľnosti 18
2.4 Prehľadávanie grafu 18

3 Návrh knižnice **23**
3.1 Hlavný panel a zobrazenia 25
3.2 Globálne informácie 28
3.3 Dátové štruktúry a ich vizualizácia 30

4 Použitie **32**
4.1 Práca s hlavným panelom a zobrazeniami 32
4.2 Geometrické tvary a dátové štruktúry 41
4.3 Práca s globálnymi objektami 42
4.4 Vytvorenie algoritmu 43
4.5 Ovládanie užívateľského rozhrania 44

5 Aplikácie s algoritmami pre hľadanie cesty **47**
5.1 Implementácia algoritmov 48
5.2 Webová stránka 50

6 Záver **51**

Literatúra **52**

Prílohy **54**

A Obsah CD **55**

B Vytvorenie projektu s knižnicou pre vizualizáciu algoritmov **56**
B.1 Vytvorenie projektu v IntelliJ 57
B.2 Vytvorenie projektu v Eclipse 60

C Spustenie webovej stránky **63**

D UML diagramy **65**

E Ukážky aplikácií **68**

Kapitola 1

Úvod

Počas štúdia programovania sa stretávame s množstvom algoritmov, ktoré je potrebné pochopiť. Tieto algoritmy sú prezentované niekoľkými spôsobmy:

1. Textový popis: Algoritmus je prezentovaný pseudokódom. Ten zvyčajne predstavuje úplný a presný postup na vykonanie výpočtu. Je sprevádzaný textovým popisom pre podrobnejšie vysvetlenie výpočtu a premenných, ale aj pre jednoduchšie pochopenie algoritmu. Textový popis je síce najpresnejší pre definíciu algoritmu, ale zároveň náročný pre vizualizáciu jeho činnosti. Tento spôsob je najčastejšie používaný pre popis a prezentáciu nových algoritmov a často vyžaduje vizualizáciu výpočtu pre jeho lepšie pochopenie.

2. Vizualizácia a interakcia: Jedná sa o program, ktorý prezentuje činnosť algoritmu na zobrazovacej ploche. Býva doplnený o pseudokód a textovú informáciu o práve vykonávanom kroku, ale vizualizácia je jeho dominantná funkcia. Užívateľ vie meniť parametre algoritmu, štruktúry, s ktorými pracuje a aj jeho konfiguračný priestor. Krokovaním výpočtu si užívateľ ujasní operácie vykonávané v rôznych fázach algoritmu.

3. Programovanie: Najlepším spôsobom porozumenia nejakého algoritmu je jeho implementácia. Programátor musí najprv algoritmu porozumieť, aby ho bol schopný správne naprogramovať. Tento spôsob vyžaduje najviac zdrojov (čas, znalosti ...).

Každá forma má svoje výhody aj nevýhody. Najlepší spôsob naučenia sa algoritmu je naprogramovať si ho. To však zaberá veľa času, preto je užitočné použiť externú knižnicu pre uľahčenie jeho tvorby. Predtým je ale nevyhnutné algoritmu porozumieť. Vizualizácia je pre človeka najlepší spôsob porozumenia algoritmu. Preto je výhodné spustiť si jeho demonštráciu a na rôznych vstupoch pozorovať, ako sa transformujú na výstupy. Nevýhodou je to, že takéto programy existujú len pre tie najpoužívanejšie algoritmy. Je ťažké nájsť uspokojivú vizualizáciu špecializovaných algoritmov. Často sú vo forme statických obrázkov, ktoré neumožňujú meniť ich konfiguráciu, doplnené textovým popisom.

Existujú webové stránky, ktoré sa vizualizáciou algoritmov zaoberajú. Príkladmy sú [10] a [8]. Tieto stránky obsahujú vizualizácie najmä pre operácie nad zoznamami a grafmi. Zobrazujú algoritmy priamo na stránke a používajú na to JavaScript.

Interakcia s algoritmom je veľmi dobrý spôsob ako mu porozumieť. Zadávaním vstupných hodnôt a zmenou parametrov algoritmu si užívateľ overí, že mu rozumie. Preto je dôležité, aby program nemal chyby a vysvetľoval funkciu algoritmu čo najlepšie. Dôležitým

prvkom je aj spoločné rozhranie. Ak sa užívateľ naučí pracovať s jedným algoritmom, bude pre neho jednoduchšie ovládať iný, ak budú mať spoločné prvky a štruktúru.

Najpopulárnejším spôsobom pre vytvorenie vizualizácie nejakého algoritmu je Java. Je to najmä preto, že je platformovo nezávislá a má veľa knižníc pre dátové štruktúry a operácie s nimi, čo uľahčuje tvorbu nového programu. Java poskytuje technológie aj pre zobrazenie programov na webe. Sú to aplety, ktoré sa dajú vložiť priamo do stránky a Java Web Start aplikácie, ktoré spúšťajú vzdialené aplikácie na užívateľovom počítači cez jnlp [16] protokol.

Problém je prezentovať takýto program na webe. HTML5 a JavaScript sú dominantné technológie pre webovú vizualizáciu. Java aplety prehliadače prestali podporovať a Web Start potrebuje JVM (Java Virtual Machine) na užívateľovom počítači. To je problém najmä pre mobilné zariadenia, ktoré sú čoraz viac rozšírené. Pre prezentáciu algoritmov na webe je výhodnejšie použiť JavaScript. Na druhú stranu je programovanie v ňom náročné. Integrácia programu do HTML stránky vyžaduje skúseného programátora. Programovanie v Jave je jednoduchšie, ale prezentovať ju na webe tak, aby bolo možné spustiť program na rôznych zariadeniach je problém.

Z dôvodov, ktoré som popísal vyššie som sa rozhodol urobiť vlastnú knižnicu pre vizualizáciu algoritmov. Bude určená pre výukové účely. Programovací jazyk bude java, pretože je programovanie v nej jednoduchšie a rýchlejšie. Bude používať Swing používateľské rozhranie. Po konzultácii s vedúcim som sa rozhodol použiť namiesto apletov technológiu Java Web Start. Všetky najpoužívanejšie prehliadače prestali aplety podporovať a preto sa nehodia pre prezentovanie na webe. Aplikácie sa budú dať spustiť na počítači s nainštalovanou Javou.

Knižnica zjednoduší tvorbu Swing rozhrania pre rôzne algoritmy. Mala by byť:

- Flexibilná - aby podporovala, čo možno najširšie spektrum algoritmov.
- Jednoduchá - aby s ňou zvládol pracovať aj menej skúsený programátor.
- Užitočná - aby programovanie uľahčovala.

V tejto práci budem rozlišovať dva typy aktérov:

- Užívateľ - bude si spúšťať algoritmy pre študijné účely. Bude pracovať iba s užívateľským rozhraním a detaily implementácie ho nezaujímajú.
- Programátor - bude pracovať s knižnicou na vytvorenie novej vizualizácie nejakého algoritmu.

Kapitola 3 podrobne popisuje návrh a vlastnosti celej knižnice a kapitola 4 ukazuje, ako ju použiť pre tvorbu vlastného algoritmu. V prílohe B je návod na vytvorenie nového projektu s knižnicou, ktorú som navrhol.

Témou tejto práce sú najmä algoritmy pre plánovanie cesty. V kapitole 2 popíšem princípy a najdôležitejšie algoritmy. Predstavuje úvod do plánovacích algoritmov a mnohé z nich použijem pre demonštráciu vytvorenej knižnice. V kapitole 5 sú popísané algoritmy, ktoré som vytvoril. V prílohách sú návrhové diagramy (D) a snímky obrazovky (E) pre vytvorené aplikácie.

Kapitola 2

Algoritmy pre plánovanie cesty

V robotike sa stretávame s úlohou plánovania cesty pre robota. Robot je situovaný v nejakom prostredí. V priestore sú umiestnené prekážky a jeho úlohou je nájsť cestu z počiatočnej konfigurácie do cieľovej. Zvyčajne chceme, aby bol robot autonómny, to znamená, aby vedel nájsť cestu sám.

Autonómny robot vie sám vykonávať akcie v prostredí. Potrebuje na to sústavu senzorov, ktoré prijímajú podnety z prostredia. Ak chce ovplyvňovať svet, v ktorom pracuje, musí byť vybavený efektormy. Je zvyčajne vybavený podvozkom s kolieskami. Existujú rôzne typy podvozkov, ktoré mu umožňujú pohybovať sa v priestore. Niektorí roboti sú statický a pohybujú len svojím ramenom, napríklad vo výrobných linkých. Táto kapitola predstaví algoritmy pre hľadanie cesty v známom priestore. Budeme predpokladať, že robot už má mapu prostredia z prekážkami a pozná aj svoju a cieľovú konfiguráciu.

Hľadanie cesty [22] je pojem z oblasti umelej inteligencie. Plánovanie vyžaduje mapu prostredia a robota, ktorý si je vedomý svojej pozície vzhľadom k mape a vie sa vyhýbať prekážkam. Je to proces, ktorý rozloží pohybovú úlohu na diskrétne pohyby tak, aby vyhovovali obmedzeniam a boli optimálne. Výsledkom bude cesta, ktorá dostane robota zo začiatočnej konfigurácie do koncovej a súčasne zabráni kolíziám s prekážkami. Algoritmy najčastejšie hľadajú optimálnu (najkratšiu) cestu. Algoritmy hodnotíme podľa:

- úplnosť - ak cesta existuje, tak ju nájde
- optimálnosť - ak nejaké riešenie existuje, potom optimálna metóda musí nájsť to najlepšie
- časová zložitosť
- priestorová zložitosť

Postupy pre hľadanie cesty majú aplikácie v robotických úlohách, inteligencii v počítačových hrách, simuláciách biologických molekúl, automatizácii v priemysle, prieskume miest s ťažkým alebo nebezpečným prístupom. Ja sa zameriam na plánovanie cesty pre robota v priestore s prekážkami.

V kapitole 2.1 popíšem vzorkovanie konfiguračného priestoru pre vytvorenie grafu. Ďalšie kapitoly sa budú venovať konkrétnym algoritmom. Kapitola 2.2 obsahuje pravdepodobnostné algoritmy pre tvorbu road máp. V kapitole 2.3 sú deterministické algortmy. Keď tieto metódy zostroja graf a pripoja k nemu počiatočný a koncový uzol, bude treba vyhľadať v ňom cestu. Algoritmami pre hľadanie cesty v grafe sa zaoberá kapitola 2.4.

2.1 Generovanie a spracovanie konfigurácií

Máme mapu prostredia a počiatočnú a koncovú konfiguráciu. Pre hľadanie cesty potrebujeme vytvoriť graf, ktorý bude reprezentovať cestu po mape. Uzly a hrany grafu nesmú ležať v prekážkach, aby nimi mohol robot prechádzať. Keď vytvoríme graf, pripojíme k nemu štart a cieľ a nájdeme v ňom cestu algoritmom pre hľadanie cesty v grafe. Prvým krokom je preto vytvorenie grafu.

Konfiguračný priestor je množina všetkých možných súradníc, ktoré môže robot nadobúdať. Ak máme mapu, bude to každá jej súradnica (aj v prekážkach). Voľným konfiguračným priestorom potom nazývame taký priestor, v ktorom sa môže robot voľne pohybovať, teda konfigurácia (súradnice) nesmie ležať v prekážke.

Vzorkovanie je dôležité, pretože na ňom závisí pokrytie voľného priestoru a tým ovplyvní aj graf a hľadanie cesty. Ak nebude graf súvislý, môže sa stať, že medzi štartom a cieľom neexistuje cesta v grafe, aj keby vo voľnom priestore existovala. Ak vygenerujeme málo vzorkov, nemusí sa nám podariť vytvoriť súvislý graf (ak to prostredie nevylučuje). Generovanie množstva vzorkov zase spomaľuje výpočet.

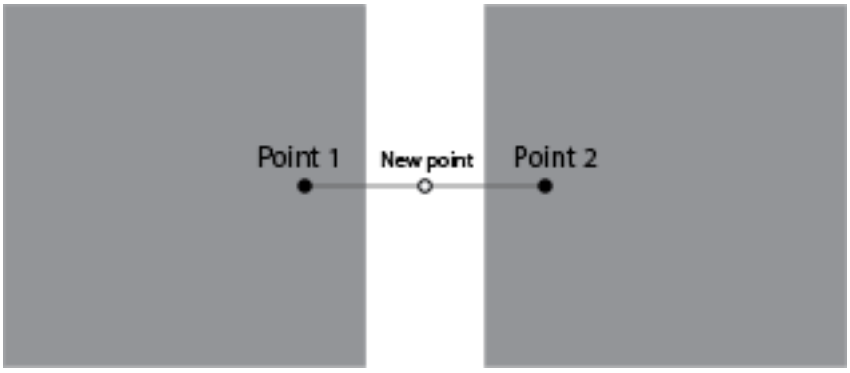
Náhodné generovanie s rovnomerným rozložením pravdepodobnosti môže všeobecne dobre fungovať, ale je nevhodné pre niektoré špeciálne situácie. Ak hľadáme cestu v priestore, ktorý obsahuje úzke priechody medzi prekážkami, je lepšie použiť modifikovaný algoritmus.

Filtrovací algoritmus RBB (Randomized Bridge Builder) sa snaží vzorkami zachytiť priestor v okolí úzkych priechodov. Algoritmus je vo výpise 2.1 a na obrázku 2.1. Využíva most, čo je úsečka medzi dvoma prekážkami, ktorej konce sú vo vnútri prekážok a stred leží vo voľnom konfiguračnom priestore. Ak sú oba vygenerované body v prekážke a stred nie je, potom sa stred pridá do grafu. Ak je však aspoň jeden z vygenerovaných bodov vo voľnom konfiguračnom priestore, žiadny bod sa do grafu nepridá. Tento algoritmus dobre funguje pre priestor s veľa prekážkami a úzkymi priestormy.

Kód 2.1 Randomized Bridge Builder.

Input:
C: konfiguračný priestor
Output:
Graph $G = (V, E)$

```
repeat
    náhodne vyber x z konfiguračného priestoru C s rovnomerným rozložením pravdepodobnosti
    if x nie je voľná konfigurácia then
        náhodne vyber x' z okolia x pomocou pravdepodobnostnej funkcie λ_x
        if x' nie je voľná konfigurácia then
            p ← bod v strede úsečky x,x'
            if p je voľná konfigurácia then
                pridaj p do grafu ako nový uzol
            end if
        end if
    end if
until mám dostatočný počet bodov
```



Obr. 2.1: Randomized Bridge Builder.

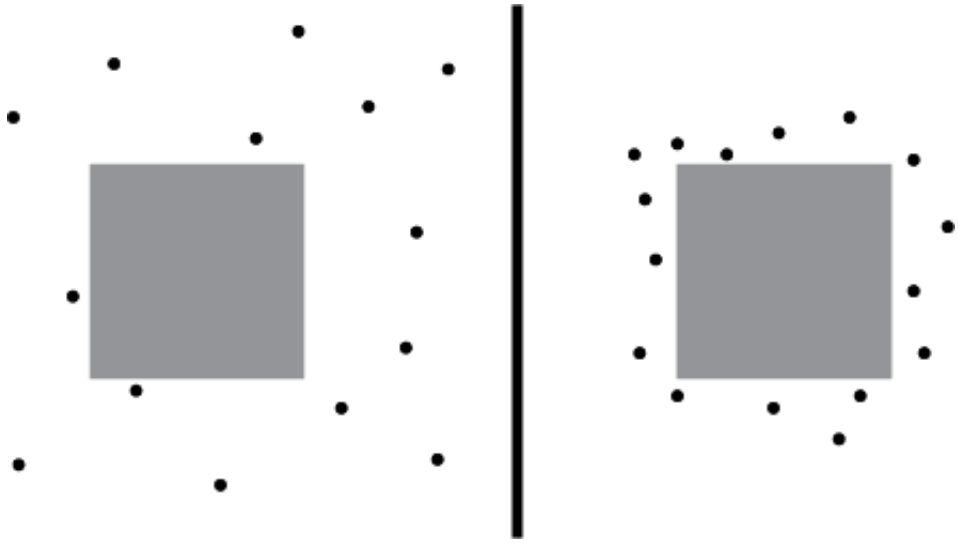
Ďalším spôsobom je vzorkovanie priestoru len v okolí prekážok. Algoritmus je vo výpise 2.2 a na obrázku 2.2. Najprv vygeneruje náhodný bod q_1 s rovnomerným rozložením pravdepodobnosti. Vo vzdialenosti *step* od q_1 je v náhodnom smere vygenerovaný bod q_2. AK sú oba body vnútri prekážky alebo naopak vo voľnom konfiguračnom priestore, algoritmus do grafu nepridá ani jeden. Iba v prípade, že sa jeden z bodov náchádza vnútri prekážky a druhý nie, je ten z voľného priestoru pridaný do grafu.

Problémom nemusia byť len úzke priechody ale aj množstvo prekážok. V takom prípade sa používa OBPRM (Obstacle PRM), ktorý sa snaží do grafu pridávať body v okolí prekážok. Pre každú náhodne vygenerovanú konfiguráciu, ktorá je v prekážke, nájde v náhodnom smere voľnú konfiguráciu. Potom delením intevalu medzi nimi nájde voľnú konfiguráciu, čo najbližšie k prekážke a pridá ju do grafu.

Kód 2.2 Vzorkovanie v okolí prekážok.

Input:
C: konfiguračný priestor
step: veľkosť kroku
Output:
Graph $G = (V, E)$
1: **repeat**
2: náhodne vygeneruj bod q_1
3: vygeneruj q_2 v náhodnom smere od q_1 a vzdialenosti *step* od neho
4: **if** q_1 nie je v prekážke $\wedge$ q_2 je v prekážke **then**
5: pridaj q_1 do grafu
6: **else if** q_1 je v prekážke $\wedge$ q_2 nie je v prekážke **then**
7: pridaj q_2 do grafu
8: **end if**
9: **until** mám dostatočný počet bodov

Funkcia spájania dvoch konfigurácií spája v grafe dve konfigurácie, pričom cesta nesmie kolidovať s prekážkami. Najjednoduchšie je spojiť ich úsečkou. Táto metóda nie je príliš výkonná, pretože nenájde cestu, ak sa medzi konfiguráciami nachádza prekážka. V takom prípade môžeme použiť niektorý jednodotazový algoritmus ako EST (kapitola 2.2.2) alebo RRT (kapitola 2.2.3). Spájanie priamkou je síce jednoduché ale veľmi výkonné. Za použitie jednodotazových algoritmov platíme náročnosťou výpočtu, čo niekedy môže vadiť.



Obr. 2.2: Vzorkovanie konfiguračného priestoru: vľavo) rovnomerné, vpravo) v okolí prekážok.

Závisí to od konkrétnej situácie. PRM (kapitola 2.2.1) používa spájanie priamkou, SRT zase jednodotazové algoritmy.

U spojovacích funkcií sledujeme, či sú symetrické a deterministické. Symetrický algoritmus zaručuje, že pre hľadanie cesty z bodu A do bodu B nájde rovnakú cestu ako pre hľadanie z B do A. Deterministický algoritmus nájde pri každom svojom volaní pre dva body A, B rovnakú cestu.

Spojovacia funkcia musí skontrolovať, že všetky konfigurácie medzi dvoma bodmy neležia v prekážkach. Môže použiť inkrementálne alebo rekurzívne delenie. Obe funkcie spojujúce konfigurácie úsečkou najprv danú úsečku medzi bodmy q' a q'' diskretizujú na postupnosť hodnôt $q_1, q_2 \dots q_n$. Líšia sa iba v spôsobe tejto diskretizácie, ako je to znázornené na obrázku 2.3.

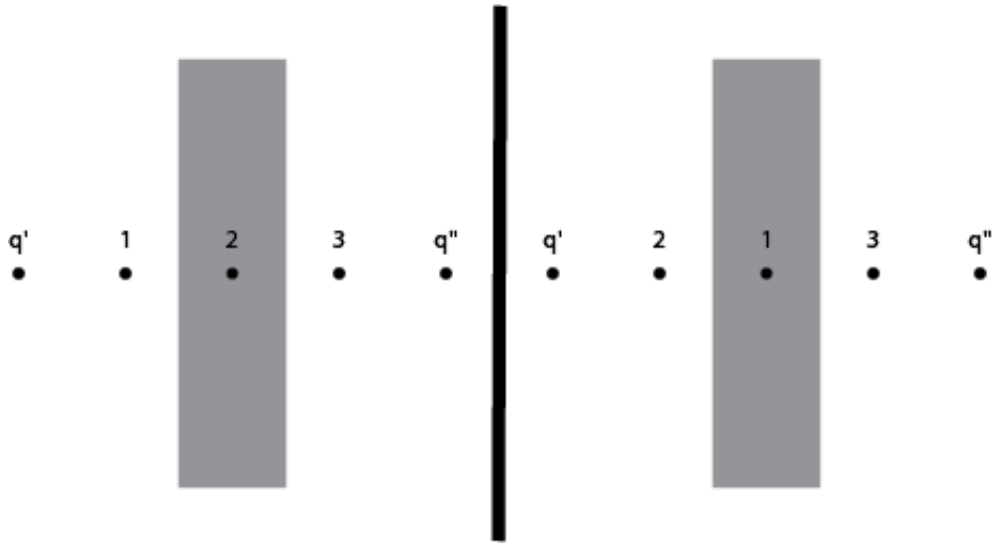
Inkrementálna verzia postupne testuje všetky možné konfigurácie medzi počiatočnou a koncovou konfiguráciou s určitou dĺžkou kroku.

Rerukzívne delenie rozdelí úsečku na polovicu a skontroluje stred. Ak je stred voľný, tak sa rekurzívne zavolá na obe polovice úsečky. Toto riešenie býva obecne efektívnejšie, pretože skôr odhalí kolízie. Je to z toho dôvodu, že dlhšie cesty majú väčšiu pravdepodobnosť, že budú kolidovať s prekážkou, zatiaľ čo inkrementálny prístup postupuje po krátkych úsekoch.

U oboch metód je dôležitá dĺžka kroku. Mala by byť čo najmenšia, aby správne zistila bod v prekážke. Menší krok síce zvýši výpočetnú náročnosť algoritmu, ale ak bude krok väčší ako priemer prekážky, metóda môže spojiť dve konfigurácie aj keď bude cesta v prekážke.

Keď ktorýmkoľvek pravdepodobnostným algoritmom nájdeme cestu medzi počiatočným a cieľovým bodom, môžeme túto cestu ďalej upravovať, aby vyhovovala určitým požiadavkom.

Najčastejším požiadavkom je nájdenú cestu skrátiť. Body cesty totiž často môžu byť priamo spojené, ale behom konštrukcie grafu spojené neboli z dôvodu, že sú príliš ďaleko na to, aby boli vzájomne zahrnuté do svojich najbližších bodov. Kratšiu cestu získame v prípade, že sa nám podarí spojiť dva nesusedné body cesty.



Obr. 2.3: Spájanie konfigurácií: vľavo) inkrementálnou, vpravo) rekurzívnou metódou.

Existuje viac variánt, ako vyberať dva nesusedné body cesty - napríklad vybrať dva náhodné body cesty alebo hladný prístup. Hladný prístup funguje tak, že sa snažíme z počiatočného bodu postupne priamo (úsečkou, ak teda nepracujeme s inou spojovacou funkciou) spojiť cieľový bod a ak neuspejeme, tak predposledný atď. Ak nemožno počiatočný bod spojiť so žiadnym iným, pokračujeme rovnakým spôsobom s následíkom počiatočného bodu.

Ďalším požiadavkom je vyhladiť nájdenú cestu. K tomu je vhodné použiť body nájdenej cesty ako riadiace body pre spline a potom testovať body na takto vzniknutej krivke, či nekolidujú s prekážkami.

Skracovanie aj vyhladzovanie spomaľuje celý pravdepodobnostný algoritmus, takže je vhodnejšie tieto optimalizácie vykonávať až po nájdení cesty a nie v priebehu vytvárania grafu.

2.2 Pravdepodobnostné algoritmy

Vyvinúť plánovanie, ktoré môže byť aplikované robotmy s veľa stupňami voľnosti je náročné. Existuje len málo efektívnych plánovačov, ktoré to dokážu. Medzi najefektívnejšie metódy pre tento typ problému patria pravdepodobnostné plánovače.

Prostredia, v ktorom robot pracuje nazývame konfiguračný priestor. Je to množina všetkých možných stavov, ktorých môže robot nadobúdať. Definujeme ju ako:

$$R(x_1, \ldots, x_n) = \{(x_1, \ldots, x_n) | \text{where } x_i \text{ satisfies some metric, for } i \in \{1, \ldots, n\}\}$$

Metrikou môže byť napríklad, že sa konfigurácia musí nachádzať v stanovenom priestore. V tomto priestore sa môžu nachádzať aj prekážky. Preto definujeme voľný konfiguračný priestor, ktorý reprezentuje pozície, v ktorých sa robot môže nachádzať bez toho, aby kolidoval s prekážkami. Štart a cieľ sa musia nachádzať v tomto priestore, rovnako ako cesta, ktorú algoritmus nájde.

Pravdepodobnostné algoritmy náhodne vzorkujú konfiguračný priestor a z bodov vytvoria graf, v ktorom hľadajú cestu. U vzoriek sa overí, či patria do voľného konfiguračného

priestoru (neležia v prekážke). Pre hľadanie cesty sa používajú algoritmy ako Dijkstrov algoritmus [20] alebo A* [17].

Pravdepodobnostné algoritmy sa rozdeľujú na:

- Jednodotazové - pre každú cestu medzi dvoma bodmy vytvoria nový graf, ktorý ich spája. Príkladmy sú Rapidly Exploring Random Trees (kapitola 2.2.3) alebo Expansive Space Trees (kapitola 2.2.2).
- Viacdotazové - vytvárajú graf v konfiguračnom priestore. Pri hľadaní cesty medzi dvoma bodmy, tieto body pridajú do grafu a hľadajú medzi nimi cestu. Graf sa zostaví raz na začiatku výpočtu a znova sa použije pre každé dva body, ktoré spájame. Patrí tu algoritmus Probabilistic Roadmap (kapitola 2.2.1).
- Kombinované - používajú oba prístupy, môžu byť použité pre opätovné hľadanie a interne môžu používať jednodotazový algoritmus. Takýmto algoritmom je Sampling Based Roadmap of Trees.

Voľba algoritmu záleží na situácii. Jednodotazové sú rýchlejšie pre nájdenie niekoľko málo ciest. Ale pretože pre každé hľadanie vytvárajú nový graf, nehodia sa pre situácie, kde ich frekventovane používame pre nájdenie cesty. V takých prípadoch je výhodnejšie investovať čas do inicializácie grafu reprezentujúceho voľný konfiguračný priestor a pri požiadavkách na získanie cesty využiť existujúci graf.

Pre plánovanie cesty potrebujeme reprezentovať prostredie počítačom. Rozlišujeme medzi dvoma prístupmy: diskrétnym a spojitým.

V diskrétnej aproximácii [2] je mapa rozdelená do blokov rovnakej (mriežka) alebo rozdielnej (topologická mapa) veľkosti. Diskrétne mapy vieme ľahko reprezentovať grafmy. Každá bunka je reprezentovaná bodom a spojená s ďalšími bunkami hranou, po ktorej sa môže robot pohybovať. Takéto grafy môžeme uložiť ako maticu susednosti.

Spojitá mapa [2] vyžaduje definovanie vnútra prekážok (napríklad polynómom) a cesty ako sekvencie bodov.

Fyzické rozmery robota komplikujú výpočet cesty, preto sa často robot redukuje na hmotný bod a všetky prekážky sa zväčšia o polovicu najdlhšej časti robota.

Návrh aplikácie

Aplikáciu pre vizualizáciu pravdepodobnostných algoritmov bude tvoriť hlavný panel. V ňom bude zobrazovacia plocha, panel nástrojov, pseudoalgoritmus a nastavenie parametrov.

Mapu pre tento algoritmus budem reprezentovať ako spojitý priestor s prekážkami. Prekážky budú polygóny a štart s cieľom body.

Užívateľké rozhranie musí obsahovať:

- Tlačítka na ovládanie algoritmu, vytváranie prekážok a pozícií. Budú sa nachádzať v paneli nástrojov.
- Komponenty pre vstupné hodnoty (parametre) algoritmu. Užívateľ bude môcť zadať počet uzlov v grafe, počet susedov.
- Zobrazenie mapy, kde sa bude ilustrovať funkcia algoritmu.
- Zobrazenie pseudokódu pre lepšie pochopenie algoritmu.

Aplikácia musí podporovať krokovanie, zmenu parametrov a mapy. Tieto funkcie sú dôležité pre lepšie pochopenie fungovania algoritmu.

2.2.1 Pravdepodobnostná road mapa

Pravdepodobnostná road mapa (PRM) [13] dokáže nájsť bezkolíznu trasu v prostredí so statickými prekážkami. Je to viacdotazový algoritmus pracujúci so spojitým konfiguračným priestorom. Metóda má dve fáze:

- Učiaca fáza - Vytvorí sa graf s uzlami, ktoré sú v bezkolíznej konfigurácii (neležia v prekážkach) a bezkolízne hrany predstavujúce cesty medzi nimi. Postup:
  - Vyber si náhodnú konfiguráciu, otestuj kolíziu a opakuj tento krok, dokým nie je konfigurácia bezkolízna.
  - Použitím lokálneho plánovača sa pokús spojiť nájdenú konfiguráciu do road mapy.
- Vyhľadávacia fáza - Hľadá cestu medzi počiatočnou a koncovou konfiguráciou. Výsledkom je sekvencia uzlov, ktoré tieto konfigurácie spojujú.

Učiaca fáza

Vytvára graf graf $G = (V, E)$, kde V je množina uzlov a E je množina hrán. Vstupom algoritmu je konfiguračný priestor, počet uzlov v grafe n a počet susedných uzlov k. Algoritmus zostaví graf tak, že najprv náhodne generuje konfigurácie z konfiguračného priestoru. Ak táto konfigurácia patrí voľnému konfiguračnému priestoru (pozícia nie je v prekážke), vloží uzol do grafu. Keď vygeneruje požadovaný počet uzlov, algoritmus sa ich pokúsi spojiť. Vyberie k najbližších susedov pre každý uzol. Ak je cesta medzi uzlom a jeho susedom voľná (neleží v prekážke), vloží hranu do množiny hrán.

Výstupom bude graf, ktorý reprezentuje konfiguračný priestor. Nezaručuje, že graf je spojitý. Táto fáza sa vykoná iba raz pre daný konfiguračný priestor.

Kód 2.3 Pseudokód algoritmu probabilistic road map pre zostavenie grafu.

Input:
C: konfiguračný priestor
n: počet uzlov grafu
k: počet najbližších susedov
Output:
$Graph\ G = (V, E)$

```
V ← ∅
E ← ∅
while |V| < n do
    repeat
        q ← náhodná konfigurácia z C
        V ← V ∪ {q}
    until q je voľná konfigurácia
end while
for all q ∈ V do
    N_q ← k najbližší sused podľa vzdialenostnej funkcie
    for all q' ∈ N_q do
        if (q, q') ∉ E ∨ Δ(q, q') ≠ NIL then
            E ← E ∪ {(q, q')}
        end if
    end for
end for
```

Vyhľadávacia fáza

Máme dané dva stavy, počiatočný (q_{init}) a koncový (q_{goal}), ktoré sa nesmú nachádzať v prekážkach. Budeme hľadať cestu v grafe. Môžeme použiť algoritmy ako Dijkstrov algoritmus alebo A*. Vstupom algoritmu budú body, medzi ktorými chceme nájsť cestu, graf, ktorý sme vytvorili v učiacej fáze pre voľný konfiguračný priestor, a počet najbližších susedov k.

Najprv pripojíme počiatočný a koncový stav ku grafu tak, že postupne skúšame, či niektorý z ich k susedov možno spojiť hranou, ktorá neleží v prekážke. Ak sa nám to podarí, nájdeme cestu medzi nimi niektorým algoritmom pre hľadanie cesty v grafe.

Výstupom bude cesta medzi začiatočným a koncovým bodom, ak existuje.

Kód 2.4 Pseudokód algoritmu PRM pre hľadanie cesty.

Input:
q_{init}: počiatočná konfigurácia
q_{goal}: koncová konfigurácia
k: počet najbližších susedov
$Graph\ G = (V, E)$: graf z učiacej fáze
Output:
cesta z q_{init} do q_{goal}

```
N_{q_init} ← k najbližší sused uzla q_init z V
N_{q_goal} ← k najbližší sused uzla q_goal z V
V ← V ∪ {q_init} ∪ {q_goal}
q' ← najbližší sused uzla q_init z N_{q_init}
repeat
    if Δ(q_init, q') ≠ NIL then
        E ← E ∪ (q_init, q')
    else
        q' ← ďalší sused uzla q_init z N_{q_init}
    end if
until q_init bol úspešne pripojený alebo N_{q_init} je prázdny
q' ← najzližší sused uzla q_goal z N_{q_goal}
repeat
    if Δ(q_goal, q') ≠ NIL then
        E ← E ∪ (q_goal, q')
    else
        q' ← ďalší sused uzla q_goal z N_{q_goal}
    end if
until q_goal bol úspešne pripojený alebo N_{q_goal} je prázdny
P ← najkratšia cesta z q_init do q_goal v G
if P nie je prázdny then
    return P
else
    return neúspech
end if
```

2.2.2 Expanzívne stromy

EST (Expansive-Spaces Trees) [1] je jednodotazový algoritmus pre riešenie kinodynamických úloh. Bol vyvinutý tak, aby rýchlo našiel cestu medzi štartom a cieľom.

Konštrukcia stromu

Nech je T_{init} strom začínajúci v štarte q_{init}. Plánovač najprv vyberie konfiguráciu q zo stromu a potom náhodnú konfiguráciu q_{rand} z okolia q. Konfigurácia q je vybraná náhodne s pravdepodobnosťou $\pi_T(q)$. Lokálny plánovač Δ sa pokúsi spojiť q s q_{rand}. Ak sa mu to podarí, q_{rand} sa pripojí do stromu, spolu s hranou (q, q_{rand}). Proces sa opakuje dokým sa do stromu nepridá stanovené množstvo uzlov. Pseudokód algoritmu je vo výpise 2.5.

Každá nová konfigurácia je spojená zo stromom, takže je graf spojitý. Funkcia výberu uzla na expanziu zaručí, že sa vyberú tie uzly, ktoré majú slabo obsadené oko-

lie a nedôjde tak k prevzorkovaniu nejakej oblasti. Touto funkciou môže byť napríklad $\pi = 1/\#neighbours$.

Kód 2.5 Pseudokód algoritmu EST.

Input:
q_0: konfigurácia koreňa stromu
n: počet pokusov pre expanziu stromu
Output:
strom $T = (V, E)$ s koreňom na q_0 a má $\leq n$ konfigurácií

```
1: V ← q0
2: E ← ∅
3: for i = 1 do n do
4:     q ← náhodne vybraná konfigurácia z T s pravdepodobnosťou π_T(q)
5:     extendEST(T, q)
6: end for
7: return T
```

Kód 2.6 Pseudokód algoritmu pre expanziu EST.

Input:
$T = (V, E)$: strom EST
q: konfigurácia, z ktorej sa bude expandovať T
Output:
nová konfigurácia q_{new} v susedstve q, alebo `NIL` v prípade neúspechu

```
1: q_new ← náhodne vybraná bezkolízna konfigurácia z okolia q
2: if Δ(q, q_new) then
3:     V ← V ∪ {q_new}
4:     E ← E ∪ {(q, q_new)}
5:     return q_new
6: end if
7: return NIL
```

Aplikácia, demonštrujúca tento algoritmus bude identická ako pre algoritmus PRM z predchádzajúcej kapitoly.

2.2.3 Rapídne sa rozširujúce stromy

Jedná sa o jednodotazový algoritmus [1], ktorý efektívne pokrýva priestor medzi štartom a cieľom.

Konštrukcia stromu

Majme stromy T_{init} a T_{goal}. Každý strom je inkrementálne expandovaný. V každej iterácii je náhodne vybraná konfigurácia q_{rand}. Nájde sa najbližšia konfigurácia q_{new} v strome k tomuto bodu a pokúsi sa ho k nemu pripojiť. Postupuje po krokoch `step_size` v priamej čiare od q_{new} do q_{rand}. Ak nenarazí na prekážku, tak sa do grafu pridá q_{rand}. Pseudokód je vo výpise 2.7.

Kód 2.7 Pseudokód algoritmu pre konštrukciu stromu.

Input:
q_0: konfigurácia koreňa stromu
n: maximálne množstvo pokusov o expanziu
Output:
strom $T = (V, E)$ s koreňom q_0 a maximálne n uzlami

```
1: V ← {q_0}
2: E ← ∅
3: for i = 1 do n do
4:     q_rand ← náhodná voľná konfigurácia
5:     extendRRT(T, q_rand)
6: end for
7: return T
```

Kód 2.8 Pseudokód algoritmu pre expanziu stromu.

Input:
$T = (V, E)$: RRT strom
q: konfigurácia pre expanziu
Output:
nová konfigurácia q_{new} alebo `NIL` v prípade neúspechu

```
1: q_near ← najbližší sused q zo stromu T
2: q_new ← bod na hrane q_near, q vo vzdialenosti step_size od q_near
3: if q_new nekoliduje s prekážkou then
4:     V ← V ∪ {q_new}
5:     E ← E ∪ {(q_near, q_new)}
6:     return q_new
7: end if
8: return NIL
```

Kód 2.9 Pseudokód algoritmu pre spojenie RRT.

Input:
$T = (V, E)$: RRT strom
q_{rand}: konfigurácia pre expanziu
Output:
connected ak q môže byť pripojený; *failure* inak

```
1: repeat
2:     q_new ← extendRRT(T, q)
3: until (q_new = q or q_new = NIL)
4: if q_new = q then
5:     return connected
6: else
7:     return failure
8: end if
```

2.3 Kombinatorické plánovanie pohybu

Kombinatorický prístup [14] hľadá cesty bez potreby aproximácie. Tieto algoritmy sú úplné, to znamená, že ak riešenie existuje, tak ho vždy nájde.

Kombinatorické plánovanie je obzvlášť vhodné pre 2D plánovanie. Pre toto prostredie produjujú oveľa lepšie výsledky ako pravdepodobnostné algoritmy z kapitoly 2.2.

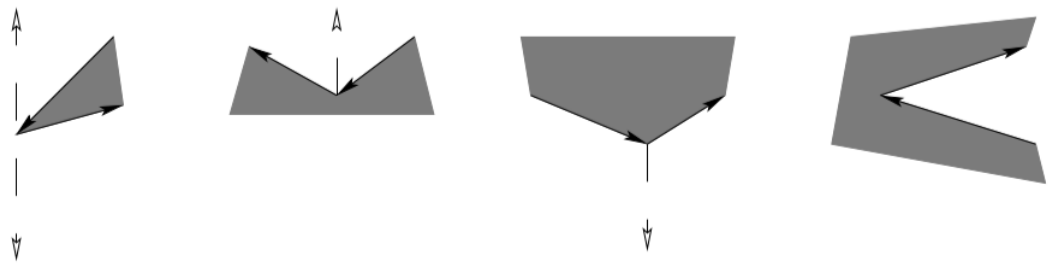
Takmer všetky vytvárajú road mapu, v ktorej sa hľadá cesta. Tá sa používa, keď chceme hľadanie cesty opakovať. Nemusíme tak pre každú zmenu počiatočného bodu konštruovať nový graf. Musí pre neho platiť, že pre akúkoľvek voľnú konfiguráciu dokážeme nájsť cestu do cieľa. Niektoré metódy najprv rozdelia priestor na bunky a potom skonštruujú graf, iné ho vytvoria priamo bez dekompozície.

2.3.1 Vertikálna bunková dekompozícia

Táto metóda rozdelí voľný konfiguračný priestor na bunky. Rozloženie by malo spĺnať tieto vlastnosti:

1. Výpočet cesty z jedného bodu do druhého vrámci bunky by malo byť triviálne, napríklad spojenie úsečkou.
2. Susednosť buniek musí byť ľahko zistiteľná.
3. Pre body (štart, cieľ) musí byť efektívne zistiť, v ktorej bunke sa nachádzajú.

Majme množinu P, ktorá reprezentuje vrcholy prekážky. Pre každý $p \in P$ vytvoríme lúč dohora a dole cez voľnú konfiguráciu, dokým nenarazí do prekážky. Môžu nastať 4 prípady znázornené na obrázku 2.4.

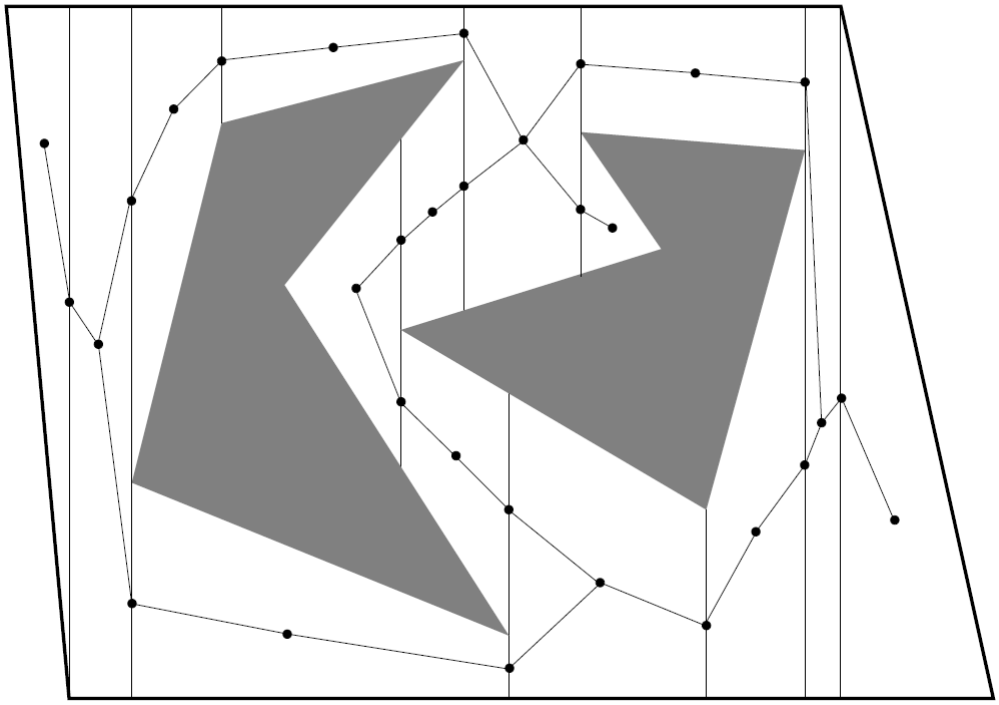


Obr. 2.4: Sú 4 prípady: 1) rozšírenie hore a dole, 2) iba hore, 3) iba dole, 4) nedá sa rozšíriť. Zdroj [14].

Keď máme priestor rozdelený na bunky, môžeme skonštruovať road mapu. V každej bunke vytvoríme bod v strede (ale môže byť aj iný) a pridáme ho do grafu. Pre všetky hrany, ktoré dve bunky zdieľajú pridáme nový bod v strede tejto hrany. Potom spojíme bod v strede bunky so všetkými bodmy na jej hraniciach. Tým vytvoríme road mapu, ako ukazuje obrázok 2.5.

2.3.2 Voronoiove diagramy

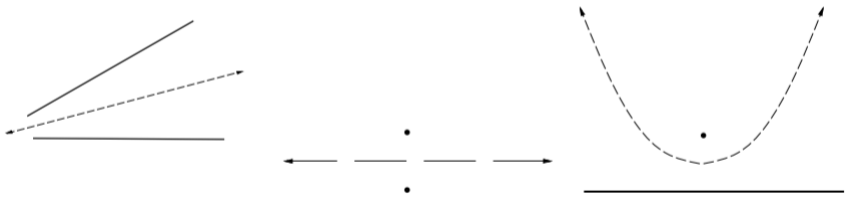
Tento prístup [14] sa snaží nájsť cestu, ktorá je čo najďalej od prekážok. Pre robotov, ktorý sa po ceste pohybujú, to minimalizuje pravdepodobnosť kolízie s prekážkou.



Obr. 2.5: Príklad road mapy pre bunkovú dekompozíciu. Zdroj [14].

Každý bod na hrane road mapy je v rovnakej vzdialenosti od dvoch bodov z hraníc prekážok. Každý vrchol road mapy je daný priesečníkom dvoch alebo viac hrán road mapy a je teda v rovnakej vzdialenosti od troch alebo viac bodov na hranici prekážok.

Pre vytvorenie mapy majme množinu všetkých hrán a vrcholov prekážok. Kandidáti na cesty sa generujú pre každé páry z tejto množiny. Pre každú hranu-hranu sa generuje priamka, pre vrchol-vrchol tiež priamka a pre vrchol-hranu sa generuje parabola. Tento postup ilustruje obrázok 2.6. Cesty, ktoré patria road mape sú potom tie, ktoré majú prienik s krivkami.

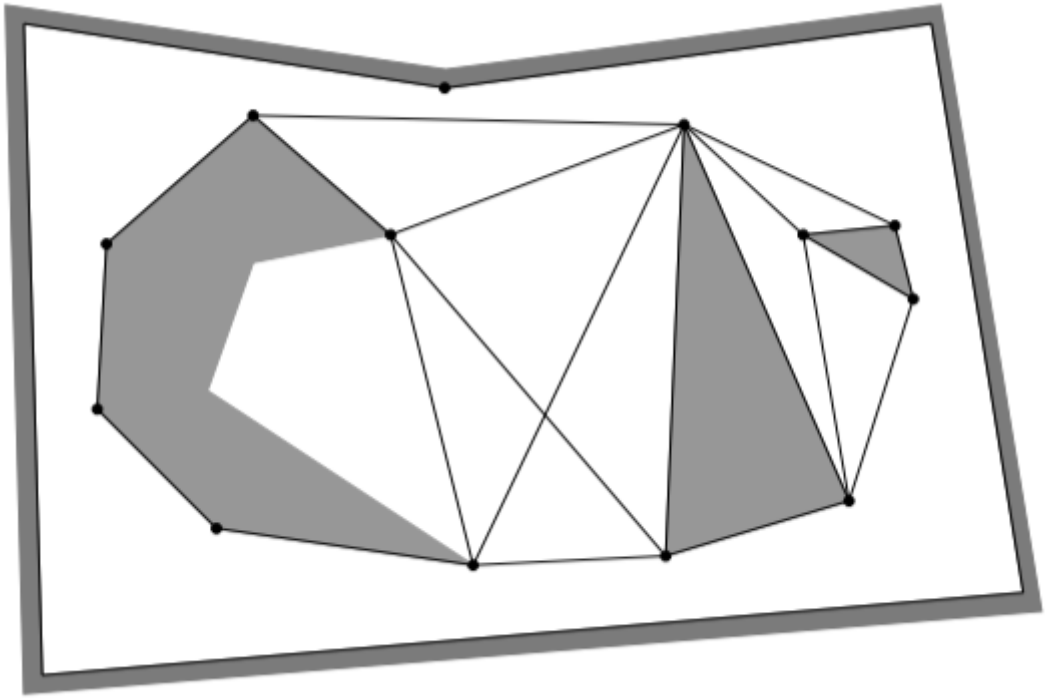


Obr. 2.6: Cesta medzi: 1) dvoma hranami, 2) dvoma vrcholmy, 3) vrcholom a hranou. Zdroj [14].

2.3.3 Grafy viditeľnosti

Ak chceme nájsť najkratšiu cestu môžeme použiť grafy viditeľnosti. Vrcholy grafu (road mapy) sú vrcholy prekážok. Nevýhodou je, že robot pri svojom pohybe prijde do tesného kontaktu s prekážkou.

Road mapu skonštruujeme tak, že do nej pridáme reflexné vrcholy. Reflexný vrchol je taký vrchol prekážky, ktorý má uhol vzhľadom ku voľnému konfiguračnému priestoru väčší ako π. To znamená, že všetky vrcholy konvexného polygónu sú reflexné. Do road mapy pridáme hrany, ktoré sú hranami prekážky a koncové body sú reflexné. Tiež pridáme hrany medzi pármy reflexných vrcholov, ktoré neležia v prekážkach. Príklad výslednej road mapy je na obrázku 2.7.



Obr. 2.7: Graf viditeľnosti. Zdroj [14].

2.4 Prehľadávanie grafu

V predchádzajúcich kapitolách som popísal ako sa zostaví graf pre prostredie s prekážkami. Aby sa bol robot schopný dostať do cieľa, musí vedieť nájsť cestu v tomto grafe. Pre túto úlohu sa používajú grafové algoritmy, ktoré dokážu nájsť cesty z jedného uzla do druhého.

Algoritmy sa hodnotia podľa štyroch kritérií:

- Úplnosť - metóda nájde riešenie ak existuje
- Optimálnosť - metóda nájde najlepšie riešenie
- Časová náročnosť

- Pamäťová náročnosť

Metódy sa delia na dve základné skupiny:

- Neinformované - nemajú spôsob ako ohodnotiť aktuálny stav a nevedia ani nič o cieľovom stave.
- Informované - majú informácie o cieľovom stave a vedia aktuálny stav ohodnotiť.

Grafy v robotických úlohách sú obvykle neorientované, to znamená, že robot sa po hrane grafu môže pohybovať v oboch smeroch. Keďže nás zvyčajne zaujíma najkratšia cesta je vhodné hrany ohodnotiť, napríklad číslom udávajúcim jej dĺžku. Algoritmy potom môžu túto hodnotu použiť k meraniu dĺžky nájdenej cesty.

Pre každý uzol si musíme pamätať, ako sme sa do neho dostali. Je to nutné pre spätné zostavenie cesty. Takže keď prejdeme po hrane z uzla q_1 do uzla q_2, tak do q_2 nastavíme indikátor `came_from` na q_1. Keď sa dostaneme do cieľa, stačí nám sledovať ukazateľe `came_from` až do štartovacieho uzla.

V ďalších kapitolách popíšem niektoré grafové algoritmy, ktoré pracujú nad neorientovanými ohodnotenými grafmy.

Prehľadávanie do šírky

Prehľadávanie do šírky (angl. Breadth First Search - BFS) rovnomerne skúma stavový priestor. Začne v štarte a postupne expanduje ulzy, ktoré si ukladá do fronty. Expanzia znamená, že vezme uzol a vygeneruje všetkých jeho susedov. Tých uloží na koniec fronty. Na začiatku fronty budú uzly, ktoré boli vygenerované skôr a sú teda bližšie začiatku. Ak by sme mali graf reprezentujúci mriežkovú mapu, potom by sa prehľadávanie podobalo záplavovému vyplňovaniu buniek. Nepoužíva informáciu o polohe cieľa a preto patrí medzi neinformované metódy.

Algoritmus používa dve štruktúry, frontu a množinu. Fronta **OPEN** obsahuje uzly určené pre expanziu. Keď sa generujú susedia pre vybraný uzol grafu, vkladajú sa práve na koniec tejto fronty. Nový uzol vybratý pre expanziu sa berie zo začiatku fronty. Po vybratí za uloží do množiny **CLOSED**, aby sme si poznačili, že sme tento uzol už navštívili. Tým sa zrýchli algoritmus, lebo nebude musieť prehľadávať uzly, v ktorých už bol. Algoritmus BFS je vo výpise 2.10.

Kód 2.10 Pseudokód algoritmu BFS.

Input:
$G = (V, E)$: graf s množinou uzlov V a množinou hrán E
q_{start}: štartovací uzol
q_{goal}: cieľový uzol
Output:
Vracia nájdenú cestu alebo `null`.

```
OPEN = nová fronta
CLOSED = nová množina
nastav q_start.came_from na null
vlož q_start do OPEN
while OPEN nie je prázdna do
    current = začiatok fronty OPEN
    vlož current do CLOSED
    if current je q_goal then
        return q_goal
    end if
    for neighbour: všetci susedia current do
        if neighbour nie je v OPEN ani v CLOSED then
            nastav neighbour.came_from na current
            pridaj neighbour do OPEN
        end if
    end for
end while
return null
```

Dijkstrov algoritmus

Algoritmus BFS (2.4) prehľadával priestor rovnomerne. Nezaujímal sa pritom o dĺžku cesty. Dijkstrov algoritmus pracuje podobne ako BFS a tiež patrí medzi neinformované metódy. Používa ohodnotenie hrán, ktoré reprezentuje dĺžku cesty.

Používa prioritnú frontu, kde sa ukladajú uzly pre preskúmanie. Priorita pre uzol je jeho cena (dĺžka cesty) prejdená od začiatočného uzlu. Je to suma všetkých hrán, po ktorých sme prešli. V množine `CLOSED` sa ukladajú uzly, po ktorých sme už prešli. Ale ak sa dostaneme do uzlu, ktorý je v `CLSOED`, ale má lepšie ohodnotenie (kratšia cesta), tak ho z `CLOSED` odstránime, pretože nová cesta, ktorou sme sa do neho dostali je kratšia. Pseudokód algoritmu je vo výpise 2.11.

Kód 2.11 Pseudokód Dijkstrovho algoritmu.

Input:
$G = (V, E)$: graf s množinou uzlov V a množinou hrán E
q_{start}: štartovací uzol
q_{goal}: cieľový uzol
Output:
Vracia nájdenú cestu alebo `null`.

```
OPEN = nová prioritná fronta
CLOSED = nová množina
nastav q_start.came_from na null
nastav q_start.cost na 0
vlož q_start do OPEN s prioritou q_start.cost
while OPEN nie je prázdna do
    current = začiatok fronty OPEN
    vlož current do CLOSED
    if current je q_goal then
        return q_goal
    end if
    for neighbour: všetci susedia current do
        new_cost = current.cost + cena prechodu z current do neighbour
        if neighbour nie je v OPEN ani v CLOSED alebo new_cost < neighbour.cost then
            nastav neighbour.cost = new_cost
            nastav neighbour.came_from na current
            pridaj neighbour do OPEN s prioritou neighbour.cost
        end if
    end for
end while
return null
```

A*

Najpopulárnejší algoritmus je A*, pretože je veľmi flexibilný a výkonný. Používa informáciu o prejdenej ceste podobne ako Dijkstrov algoritmus z kapitoly 2.4 a dopĺňa ho heuristikou spodného odhadu cesty do cieľa. Preto patrí medzi informované metódy. Funguje dobre pokým heuristika neprecení skutočnú cestu k cieľu.

Používa prioritnú frontu `OPEN`, v ktorej prioritu udáva funkcia $f(n) = g(n) + h(n)$, kde $g(n)$ je prejdená cesta od začiatku a $h(n)$ je spodný odhad cesty k cieľu. Do množiny `CLOSED` si ukladá preskúmané vrcholy. Vrchol z nej môže vybrať, ak so neho dostane a má menšiu hodnotu $g(n)$. Postup je vo výpise 2.12

Kód 2.12 Pseudokód A* algoritmu.

Input:

$G = (V, E)$: graf s množinou uzlov V a množinou hrán E
q_{start}: štartovací uzol
q_{goal}: cieľový uzol

Output:

Vracia nájdenú cestu alebo `null`.

```
OPEN = nová prioritná fronta
CLOSED = nová množina
nastav q_start.came_from na null
nastav q_start.cost na 0
vlož q_start do OPEN s prioritou q_start.cost
while OPEN nie je prázdna do
    current = začiatok fronty OPEN
    vlož current do CLOSED
    if current je q_goal then
        return q_goal
    end if
    for neighbour: všetci susedia current do
        new_cost = current.cost + cena prechodu z current do neighbour
        if neighbour je v OPEN a má väčšiu cenu ako new_cost then
            odstráň neighbour z OPEN
        end if
        if neighbour je v CLOSED a má väčšiu cenu ako new_cost then
            odstráň neighbour z CLOSED
        end if
        if neighbour nie je v OPEN ani v CLOSED then
            nastav neighbour.cost = new_cost
            nastav neighbour.came_from na current
            pridaj neighbour do OPEN s prioritou neighbour.cost + heuristika(neighbour,q_goal)
        end if
    end for
end while
return null
```

Kapitola 3

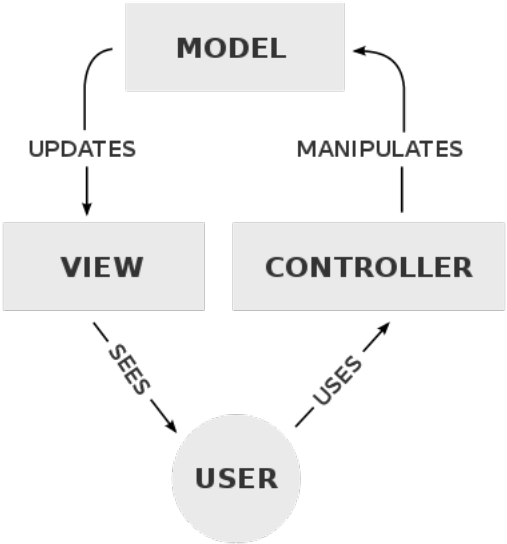
Návrh knižnice

Cieľom tejto kapitoly je navrhnúť knižnicu pre vizualizáciu algoritmov. Bude určená pre výukové účely. Mala by poskytovať prostriedky pre uľahčenie tvorby užívateľského rozhrania pre algoritmy. Musí byť dostatočne flexibilná, aby podporovala vizualizáciu širokého sprektra typov algoritmov.

Aplikácie s užívateľským rozhraním často implementujú návrhový vzor MVC. Táto knižnica bude tento vzor používať.

MVC (Model-View-Controller) [21] je návrhový vzor, ktorý sa používa pri implementácii aplikácií s užívateľskými rozhraniami. Skladá sa z troch častí, ktoré sú vzájomne prepojené a ilustruje ich obrázok 3.1.

- Model obsahuje dáta, logiku a pravidlá aplikácie.
- Zobrazenie je prezentácia modelu, zvyčajne pre človeka.
- Kontrolér spracováva vstup a konvertuje ho na príkazy pre model alebo zobrazenie.



Obr. 3.1: Návrhový vzor MVC.

Aplikácie s algoritmami majú podobnú štruktúru. Model tvorí algoritmus s dátovými štruktúrami a logikou. Tento model sa prezentuje pre užívateľa formou užívateľského rozhrania. Rozhranie tvoria prvky pre ovládanie algortimu, nastaviteľné parametre, zobrazenie stavu výpočtu alebo výstup algoritmu. Rôzne algoritmy sa od seba líšia dátami a výpočtami nad nimi, ale užívateľské rozhranie pre ich manipuláciu a zobrazenie je veľmi podobné.

Tento návrhový vzor použijem a navrhnem knižnicu pre uľahčenie tvorby prezentovacej vrstvy. Bude obsahovať prvky pre užívateľské rozhranie, vstupy a ovládanie aplikácie a algoritmu. Model bude obsahovať užitočné štruktúry, ktoré algoritmy používajú.

Keď sa pozrieme na ukážky programov algoritmov, ktoré sa dajú nájsť na webe uvidíme, že majú spoločné črty. Každý má zobrazovaciu plochu, kde sa graficky prezentuje výpočet. Vizualizácia vedúca k lepšiemu pochopeniu algoritmu je primárnou funkciou takýchto aplikácií. Zobrazovacia plocha je dominantnou, ale nie jedinou funkciou.

Mnohým algoritmom sa dajú meniť parametre. Užívateľ môže pozorovať zmeny v chovaní v závislosti na hodnote koeficientov, ktoré zadá. Často sa dajú meniť aj štruktúry, ktoré algoritmus používa. Napríklad zoznamy, kde si užívateľ môže zadať počet elementov, ich hodnoty. Zmena statového priestoru je typická pre algoritmy pre hľadanie cesty, kde máme mapu a algoritmus nájde cestu zo začiatku do cieľa a vyhýba sa pritom prekážkam. Nastavovanie parametrov algoritmu vedie k lepšiemu pochopeniu výpočtu. Nielenže je to zaujímavejšie, než sledovanie dopredu nastaveného výpočtu alebo animácie, ale pozorovanie zmien, ktoré parametre ovplyvňujú zvýši porozumenie algoritmu.

Všetky algoritmy sú v počítači reprezentované postupnosťou inštrukcií. Inštrukcie sú prezentované pseudokódom (niekedy kódom pre konkrétny programovací jazyk). Programy pre vizualizáciu algoritmov majú preto elementy pre zobrazenie pseudokódu. Užívateľovi to napovie ako algoritmus pracuje na úrovni počítača a zlepší chápanie princípu jeho fungovania.

Niekedy sa počas výpočtu vypisujú doplňujúce informácie. Môžu to byť napríklad hodnoty premenných, slovný popis vykonávanej inštrukcie, chyby. Táto funkcia nie je vždy prítomná (alebo dominantná), pretože čítať text, keď je k dispozícii zaujímavá vizualizácia je nelákavé. Avšak podrobný výpis toho, čo sa deje s premennými počas algoritmu je veľmi užitočný, ak sa užívateľ snaží o hlbšie pochopenie algoritmu. Preto by ho programy mali obsahovať a aspoň voliteľne prezentovať pre užívateľa.

Algoritmus je postupnosť inštrukcií. Zmyslom jeho vizualizácie je vysvetliť užívateľovi ako funguje. Mnohé programy preto ponúkajú možnosť krokovania výpočtu. Užívateľ sa môže zdržať pri nejakej inštrukcii, aby si ujasnil, čo robí, aby si prečítal sprievodný text. Môže sa vrátiť niekoľko krokov dozadu, alebo preskočiť inštrukcie, ktoré ho nezaujímajú. Voľnosť vracať sa a ovplyvňovať rýchlosť prezentovaného algoritmu tiež pomáha k jeho pochopeniu.

Knižnica, ktorú navrhnem musí mať všetky tieto prvky: zobrazovaciu plochu, pseudokód, textový panel. Pevné rozloženie by bolo príliš obmedzujúce. Ak by sa programátor rozhodol nejakú jeho časť neimplementovať, bolo by to miesto nevyužité. Neposkytovalo by to ani priestor pre budúce rozšírenia, ako napríklad pridanie ďalšieho prvku. Rozhranie preto musí byť flexibilné. Prvky nebudú mať pevné rozloženie a bude sa dať meniť ich veľkosť a počet.

Pri návrhu som použil jazyk UML. Diagramy tried pre dôležité časti aplikácie sú v prílohe D.

3.1 Hlavný panel a zobrazenia

Celá vizualizácia algoritmu bude v jednom hlavnom paneli (`JPanel`). Obsahuje prvky popísané v predchádzajúcej kapitole, ktoré poskytujú informácie o prezentovanom algoritme (zobrazovacia plocha, pseudokód, ...). Tieto prvky budem odteraz nazývať zobrazenia.

Zobrazenie je logický element aplikácie. Každé má jednu funkciu, napríklad zobrazenie pre vizualizáciu algoritmu, pseudokód a iné. Jeden algoritmus má niekoľko zobrazení, ktoré ilustrujú vykonávanie výpočtu pre užívateľa.

Hlavný panel spravuje zobrazenia, ich rozloženie a aktivitu. Všetky zobrazenia nemusia byť vždy aktívne (zobrazené v hlavnom paneli). Dajú sa vymieňať a schovávať, podľa toho čo si užívateľ chce práve zobraziť. Rozloženie nie je pevné. Hrany medzi zobrazeniami sa dajú posúvať, dva zobrazenia sa dajú spojiť, jedno zobrazenie sa dá rozdeliť na dve. Tieto akcie ilustruje obrázok 3.2 a príklad hlavného panela zo zobrazeniami je na obrázkoch v prílohe E. Na obrázku vpravo je ukázaný posun hrany. Keď sa myš presunie do blízkosti hrany, zmení ukazateľ a užívateľ môže potiahnuť myšou a zmeniť veľkosť zobrazenia. Keď klikne pravým tlačidlom v blízkosti hrany, ukáže sa mu kontextové menu, ako je na obrázku v strede v 3.2. Prvá položka je rozdelenie zobrazenia, ktoré ho rozdelí na dve časti horizontálne alebo vertikálne podľa toho, na ktorú hranu užívateľ klikne. Druhá položka je spojenie dvoch zobrazení. Tie sa dajú spojiť iba keď majú spoločnú hranu. Pravé tlačidlo myši vyvolá kontextové menu kdekoľvek v zobrazení. Bude obsahovať všetky zobrazenia, ktoré sú pre algoritmus definované a ak užívateľ zvolí jedno z nich, vymení sa za zobrazenie, ktoré menu vyvolalo. Ilustruje to obrázok vľavo z 3.2, kde vidno blokované zobrazenia, ktoré sú práve aktívne a teda sa nedajú vymeniť.

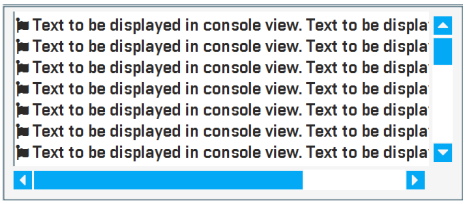
Obr. 3.2: Manipulácia zobrazení.

Hlavný panel spravuje všetky zobrazenia a vie s nimi manipulovať, čím podporuje široké spektrum rozložení. Používa `GridBagLayout`, ktorý rozdelí panel na mriežku a vloží komponenty do jej buniek.

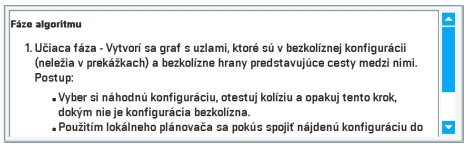
Zobrazenie je panel, ktorý obsahuje jeden aspekt algoritmu. Zvyčajne má jednu funkciu. Príkladom môže byť zobrazenie vizualizácie funkcie algoritmu, pseudokód, textový výstup, ovládací panel, panel s parametrami.

Zobrazenia sa vkladajú do hlavného panela. Každé má svoju pozíciu ľavého horného rohu panela (x, y - tá bunka), dĺžky a výšky (počet buniek). Tieto hodnoty udávajú jeho pozíciu v mriežke tvoriacej bunky rozloženia hlavného panela. Takže zobrazenia sa vkladajú do hlavného panela a každé zobrazenie má informácie o svojej pozícii v tomto rozložení.

Zmena veľkosti zobrazenia sa vykonáva zmenením počtu buniek, ktoré zaberá. Túto zmenu vykonáva hlavný panel cez metódy, ktoré poskytuje. Zobrazenie by nemalo me-



Obr. 3.3: `ConsoleView` - výpis výstupných textových informácií.



Obr. 3.4: `TextView` - panel s textovými informáciami.

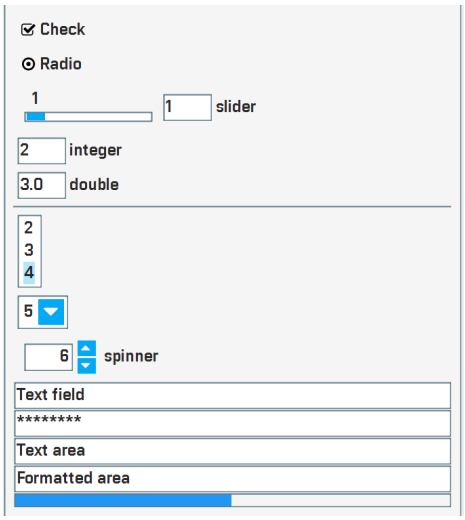
niť svoje vlastné obmedzenia, pretože zmena v nich zvyčajne vyžaduje zmenu v susedom zobrazení. Napríklad zväčšenie šírky jedného zobrazenia je sprevádzané zmenšením šírky susedného zobrazenia. Tieto zmeny sa vykonávajú v metódach hlavného panela a sú podrobne popísané v kapitole 4.

Hlavným cieľom tejto knižnice je uľahčiť tvorbu vizualizácie algoritmu. Dosahuje toho tým, že poskytuje znovupoužiteľné zobrazenia, ktoré majú definované chovanie. Programátor si prispôsobí len obsah a potom môže využívať ich funkce.

Ďalej stručne predstavím mnou vytvorené zobrazenia. Ich konkrétne použitie bude podrobne popísané v kapitole 4.

- `CodeView` - panel s pseudokódom. Textovo zapísaný algoritmus. Zvýrazňuje kľúčové slová a riadok s inštrukciou, ktorá sa práve vykonáva. Obrázok 3.5.
- `ConsoleView` - konzolový výpis textovej informácie. Slúži na podrobný výpis vykonávaného kroku pre užívateľa. Obrázok 3.3.
- `DrawView` - slúži na zobrazovanie vizualizácie. Obrázok 3.9.
- `MapView` - obsahuje mapu (konfiguračný priestor) pre algoritmy. Toto zobrazenie je špecializované a väčšina algoritmov ho nepoužije. Mapa pozostáva zo štartu, cieľa a prekážok. Zvláda aj ich pridávanie a upravovanie. Obrázok 3.10.
- `ParameterView` - vkladajú sa do neho vstupné komponenty. Je určený pre nastavenie parametrov algoritmu, simulácie alebo celej aplikácie. Obrázok 3.6.
- `StatusBarView` - panel s progres barom a textovým poľom pre výpis stavu výpočtu a dôležitých informácií pre užívateľa. Obrázok 3.8.
- `TextView` - je to textové pole pre zobrazenie podrobných informácií. Môže to byť popis celej aplikácie, algoritmu, návod na ovládanie alebo popis vstupných hodnôt a parametrov algoritmu. Text sa dá formátovať v HTML. Obrázok 3.4.
- `ToolbarView` - obsahuje tlačítka na ovládanie aplikácie a simulácie algoritmu. Obrázok 3.7.
- `GridMapView` - je mriežková mapa. Obsahuje štvorcové bunky, ktorým sa dá meniť typ. Obrázok 3.11.
- `GridMapSetView` - zobrazuje množiny buniek. Používa sa pre prehľadávacie algoritmy, ktoré si ukladajú bunky pre expanziu a bunky, ktoré už boli navštívené. Obrázok 3.12.

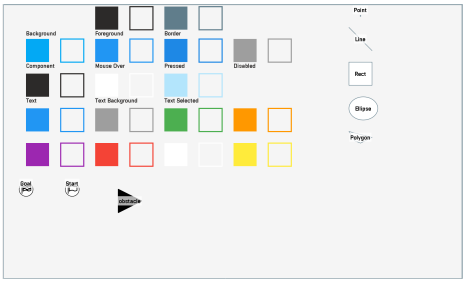
Obr. 3.5: `CodeView` - panel s pseudokódom.



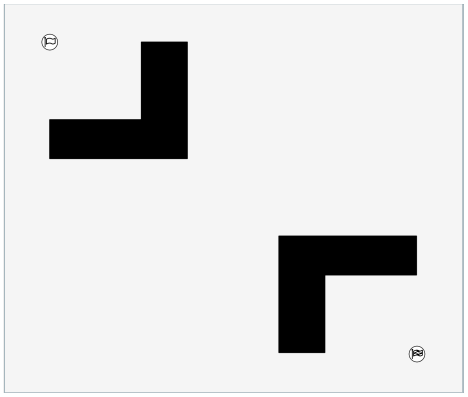
Obr. 3.6: `ParameterView` - panel so vstupnými komponentami.

Obr. 3.7: `ToolbarView` - tlačítka na ovládanie aplikácie.

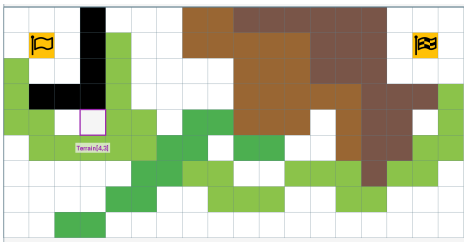
Obr. 3.8: `StatusBarView` - stav aplikácie a výpočtu.



Obr. 3.9: `DrawView` - vykresľovanie vizualizácie.



Obr. 3.10: `MapView` - mapa so štartom, cieľom a prekážkami.



Obr. 3.11: `GridMapView` - mriežková mapa s bunkami rôznych typov.

Obr. 3.12: `GridMapSetView` - zobrazenie množín buniek pre mriežkovú mapu.

Jedným z argumentov pre vytvorenie tejto knižnice bolo, aby algoritmy v nej mali podobné črty. Napríklad ovládanie a grafické elementy budú podobné a ak sa užívateľ naučí pracovať s jedným algoritmom, práca s iným, podobným, mu zaberie menej času a úsilia. To platí aj pre rozloženie zobrazení. Mnohé algoritmy budú mať podobné rozhranie. Aby sa rozloženie nemuselo vždy programovať ručne, poskytuje knižnica rámce (`Frame`). Každý rámec spravuje zoznam zobrazení pre algoritmus a podľa stavu aplikácie (popísané v kapitole 3.2) automaticky mení ich rozloženie v hlavnom paneli. Programátorovi stačí iba vytvoriť rámec a v konštruktore mu predať všetky zobrazenia, ktoré potrebuje. Flexibilita aplikácie tým neutrpí, pretože bude stále možné zobraznia rozširovať aj vymieňať, avšak pri zmene stavu sa zobrazenie vráti do pôvodného rozloženia. Je možné pridať aj nové rámce, ktoré budú implementovať rôzne rozloženia.

Pre zjednodušenie tvorby užívateľského rozhrania som vytvoril základné komponenty. V balíčku `view.uiComponents.buttons` sú tlačítka. Väčšina z nich ovláda zmenu stavu aplikácie, napríklad prechod do editačného alebo simulačného stavu. Sú tam aj tlačítka pre pridanie štartu, cieľa a prekážky do `MapView` a `GridMapView`.

V balíčku `view.uiComponents.input` sú vstupné komponenty, ako napríklad zaškrtávací box, vstupné polia. Každý obsahuje popis a majú aj jednu dôležitú funkciu, ktorá súvisí so sekciou 3.2. V konštruktore každej vstupnej komponenty je `String key`. Tento kľúč slúži pre globálny prístup k jej hodnote. To znamená, že vždy, keď užívateľ zadá novú hodnotu, uloží sa do globálnych vlastností `Properties` a bude prístupná z celej aplikácie. Môže to byť užitočné napríklad pre zadanie parametrov algoritmu. Vstupné komponenty tiež validujú vstup, takže do poľa pre čísla sa nedá vložiť hodnota iného typu.

3.2 Globálne informácie

Návrhový vzor Singleton poskytuje jedinú instanciu svojej triedy. Použil som ho pre triedu s globálnymi vlastnosťami `Properties`. Je to v skutočnosti výčtový typ (`Enum`), ktorý zaisťuje jedinečnosť instancie. Informácie, ktoré sú v ňom uložené spadajú do troch kategórií:

- Grafika - definuje grafické elementy ako farby, písmo. Používa sa pre vykresľovanie užívateľského rozhrania.
- Globálne objekty - hodnoty, ktoré by mali byť prístupné z celého programu. Môžu to byť napríklad parametre algoritmu.
- Stav aplikácie.

Java Swing aplikáciám sa dá nastaviť Look and Feel (LaF) [19], ktorý mení ich vzhľad. Pre komponenty mojej knižnice som vytvoril vlastný LaF. Rozširuje `Synth`, ktorý tvorí zá-

klad pre vlastný štýl. Na definíciu LaF som použil XML súbor spolu s instanciou `SynthPainter` pre vykresľovanie komponent. Oba používajú grafické informácie z globálnych vlastností `Properties`.

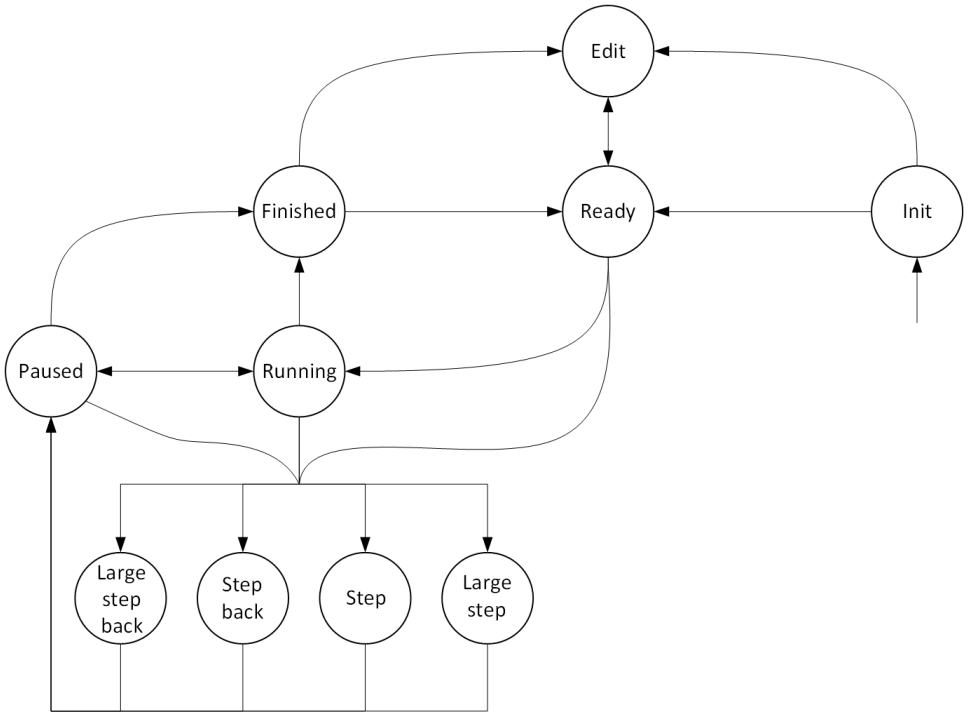
Všetky algoritmy majú svoje stavy spoločné. Môže to byť nastavovanie parametrov a konfiguračného priestoru, simulácia výpočtu, krokovanie, zobrazenie výsledkov. Knižnica má preto stavovú informáciu aplikácie. Je uložená v instancii triedy `Properties` a je preto prístupná z celého programu (cez getter a setter). Význam stavov prezentuje nasledujúci zoznam:

- `Init` - inicializačný stav. Aplikácia do neho prejde hneď po vytvorení hlavného panela.
- `Edit` - upravovacia fáza. Upravujú sa v nej parametre, premenné, konfiguračný priestor algoritmu.
- `Ready` - algoritmus je pripravený pre simuláciu.
- `Running` - beží simulácia algoritmu.
- `Paused` - simulácia je pozastavená.
- `Finished` - simulácia skončila a zobrazí sa výsledok výpočtu.
- `Large step back` - krokovanie výpočtu. Veľký krok dozadu. Veľký krok preskočí niekoľko inštrukcií, na významné miesto vo výpočte.
- `Step back` - malý krok o jednu inštrukciu späť.
- `Step` - krok o jednu inštrukciu dopredu.
- `Large step` - veľký krok dopredu na významné miesto vo výpočte.

Prechod medzi stavmy nie je vynucovaný. Je na programátorovi, či ich bude využívať a rešpektovať, ale niektoré triedy v knižnici ich používajú. Ich význam je popísaný vyššie v zozname a prechody medzi nimi zobrazuje stavový diagram na obrázku 3.13. Z obrázka by nemuselo byť úplne jasné krokovanie. Znamená to, že krokovanie môže byť spustené z akéhokoľvek stavu simulácie (pripravená, beží, pozastavená) a po vykonaní kroku sa výpočet pozastaví.

Použitie stavu a globálnych premenných je podrobnejšie popísané v kapitole 4.3.

Knižnica sa zaoberá vizualizáciou činnosti algoritmu. Algoritmus zvyčajne beží v samostatnom vlákne aby neblokoval užívateľské rozhranie. Jedným z požiadavkov bolo krokovanie algoritmu. Aby bolo možné krokovať dopredu aj dozadu, budeme potrebovať krok, ktorý je schopný inštrukciu vykonať ale aj vrátiť. Abstraktné triedy pre beh algoritmu sú `AlgClass` a pre krok `UndoableCommand`. Ako názov naznačuje, `UndoableCommand` je príkaz, ktorý sa dá vrátiť. To bude základná jednotka krokovania simulácie. `AlgClass` obsahuje zoznam takýchto príkazov. V metóde `simulate(List<UndoableCommand>)` sa do parametra uložia príkazy a simulácia sa ovláda cez zmenu stavu aplikácie. Popis, ako ich použiť pre vytvorenie vlastného algoritmu je v kapitole 4.4.



Obr. 3.13: Stavy aplikácie.

3.3 Dátové štruktúry a ich vizualizácia

Algoritmy manipulujú s dátovými štruktúrami. Vizualizácia algoritmu predstavuje grafickú prezentáciu takýchto štruktúr, ich zmeny a transformácie.

V kapitole 3.1 boli popísané zobrazenia a jedno z nich je vykresľovacie `DrawView`. Práve tu sa na zobrazovaciu plochu vizualizuje činnosť algoritmu. Používa na to grafický objekt `java.awt.Graphics`. Je teda výhodné, aby boli štruktúry, s ktorými algoritmus počíta, schopné vykresľovania.

Knižnica obsahuje základné geometrické útvary. Sú nimi bod, priamka, štvoruholník, elipsa a polygón. Pre ich implementáciu som použil JTS Topology Suite [6].

JTS je knižnica, ktorá vznikla ako externý projekt GeoTools [5]. Obsahuje numericky stabilné geometrické dátové štruktúry a operácie s nimi. Dedia od abstraktnej triedy `Geometry`, ktorá poskytuje spoločné rozhranie a operácie. Využitie JTS má tú výhodu, že zložité geometrické operácie sú v nej implementované a dobre otestované. Ale na rozdiel od tvarov v `java.awt.Shape` sa nedajú vykresľovať s `java.awt.Graphics2D`.

Preto grafické tvary, ktoré som vytvoril implementujú rozhranie `PaintableShape`. Má definované metódy pre vykreslenie tvaru, ktoré každý tvar implementuje. Dá sa nastaviť farba hrany a výplne objektu a hrúbka čiary.

Príkladom použitia tvarov sú objekty mapy. V balíčku `model.map` sú štart, cieľ a prekážka. Sú instanciami triedy `Geometry` a rozhrania `PaintableShape`, takže sa s nimi dajú robiť geometrické operácie a vykresľovať na zobrazovaciu plochu. Tým, že väčšinu funkcií dedia od svojich predkov je ich implementácia jednoduchá. S nimi sa dá vytvoriť mapa v zobrazení `MapView`. Mapa môže obsahovať jednu štartovaciu a jednu cieľovú pozíciu. Okrem

nich sa do mapy dajú pridávať prekážky. Keď je aplikácia v upravovacej fáze (`State.EDIT`), tak sa dajú myšou upravovať pozície štartu a cieľa a aj tvar prekážok.

Ďalším príkladom použitia tvarov je graf. Grafy sú v algoritmoch často používané. Pre graf som využil knižnicu JGraphT [15]. Sú v nej implementované rôzne typy grafov a poskytuje aj užitočné operácie s nimi.

S použitím tejto knižnice som implementoval jednoduchý graf, ktorý sa dá vykresliť na zobrazovaciu plochu. Dosiahol som to tým, že graf implementuje rozhranie `PaitableShape` a jeho hrany a uzly sú geometrické tvary, ktoré som popísal vyššie. Vykresľovanie grafu je teda zredukované na iteráciu cez jeho hrany a vrcholy a ich vykreslenie je už definované v ich nadtriedach.

S týmito jednoduchými geometrickými útvarmy sa dajú vytvoriť grafické dátové štruktúry, ktoré sa dajú zobrazovať na zobrazovacej ploche pri vizualizácii algoritmu. Knižnice, ktoré som použil zjednodušujú prácu pri manipulácii s tvarmy, pretože obsahujú základné operácie a algoritmy.

Kapitola 4

Použitie

Táto kapitola popisuje použitie knižnice pre vizualizáciu algoritmov. Vysvetľuje ako použiť existujúce elementy a ako vytvoriť nové. Zmyslom knižnice je uľahčiť tvorbu vizualizácie algoritmov a preto musia byť elementy ľahko použiteľné a prispôsobiteľné. Väčšinou to znamená, že trieda poskytuje určitú funkcionalitu a programátor iba definuje jej obsah.

Budú tu popísané verejné metódy, ktoré triedy poskytujú spolu s kúskami kódu pre ilustráciu. Najprv bude predstavený hlavný panel, ktorý tvorí základ celej aplikácie. Potom budú podrobne popísané zobrazenia, ktoré tvoria obsah aplikácie.

V kapitole 4.2 je popis vytvorenia vlastných dátových štruktúr. Záver kapitoly sa venuje vytvoreniu algoritmu a použitiu krokovania k jeho simulácii.

4.1 Práca s hlavným panelom a zobrazeniami

Hlavný panel `MainPanel` je abstraktná trieda. Keď bude chcieť programátor vytvoriť vlastný algoritmus, vytvorí potomka hlavného panela. Je dôležité tento panel inicializovať zavolaním konštruktoru predka. Je to z dôvodu, že sa musí nastaviť layout a registrovať sa u `ViewMouseListener`, ktorý bude umožňovať ovládanie rozloženia zobrazení myšou. Odkaz na instanciu hlavného panela sa uloží aj do globálnych premenných a je prístupný z celej aplikácie. Nastaví sa tiež vzhľad (look and feel) celej aplikácie. Príklad je vo výpise 4.1.

Je vhodné v konštruktore vytvoriť aj zobrazenia, ktoré bude používať a nastaviť rozloženie. Na rozloženie zobrazení sa používajú implementácie rozhrania `Frame`. Tento rámec sa stará o polohy zobrazení v závislosti na stave aplikácie a nastavuje ich automaticky. Ukážka použitia je vo výpise 4.1. Ak žiadny rámec nevyhovuje, dá sa jednoducho vytvoriť nový alebo použiť priamo metódy hlavného panela pre rozloženie obsahu. Priklady použitia sa dajú nájsť v zdrojových textoch v balíčku `view.frames`.

Kód 4.1: Vytvorenie vlastného hlavného panela.

```
public class MyMainPanel extends MainPanel{
    public MyMainPanel(){
        super(); // important for initialization

        MyMap map = new MyMap();
        MyToolbar toolbar = new MyToolbar();
        MyParameters params = new MyParameters();
        MyCode code = new MyCode();
        MyConsole console = new MyConsole();
```

```
        new Frame1(this, toolbar, map, params, code, console);
    }
}
```

Najdôležitejším aspektom knižnice sú zobrazenia. Ich cieľom je podstatne zjednodušiť vizualizáciu. Ďalej ukážem spôsoby použitia existujúcich zobrazení a ako sa dá vytvoriť nové.

Pseudokód

Pre zobrazenie pseudokódu slúži `CodeView`. Obsahuje textový panel, v ktorom je zobrazený kód. Panel je prístupný cez getter, ale pre manipuláciu s týmto zobrazením nie je potrebný.

Pseudokód sa skladá zo zoznamu riadkov s inštrukciami, vstupom a výstupom algoritmu. Má aj zoznam kľúčových slov, ktoré budú v pseudokóde zvýraznené. Zvýrazniť sa dá aj riadok s práve vykonávanou inštrukciou, čo je výhodné najmä pri krokovaní algoritmu.

Základné operácie so zobrazením sú:

- Pridanie vstupu - väčšina algoritmov má vstupné hodnoty. V tomto zobrazení sa dá nastaviť reťazec so vstupom metódou `addInput(String)`.
- Pridanie výstupu - podobne ako pri vstupe. Nastavuje sa metódou `addOutput(String)`.
- Pridanie riadku kódu - vloží jeden riadok pseudokódu do zobrazenia. Riadky sa zobrazujú v poradí, v akom boli vložené. Pridávajú sa metódou `addLine(String)`. Táto metóda je pre pridávanie preferovaná, pretože sama nastaví číslo riadka, ale dá sa použiť aj `addLine(Integer, String)`, kde `Integer` predstavuje číslo riadka.
- Vyčistenie obsahu - pre zmazanie všetkých riadkov slúži `clean()`.
- Zvýraznenie riadku - pri krokovaní je vhodné zvýrazniť práve vykonávaný riadok algoritmu. Zobrazenie uchováva číslo tohoto riadku a je prístupné cez `getHighlightLine()` a `setHighlightLine(int)`.
- Kľúčové slová - pre zvýraznenie kľúčových slov sa nastaví zoznam týchto reťazcov metódou `setKeywords(List<String>)`. Zobrazenie má tri preddefinované sady nastaviteľné metódami `setCKeywords()`, `setJavaKeywords()` a `setPseudocodeKeywords()`.

Po vytvorení vlastnej podtriedy zobrazenia s pseudokódom sa nastavia riadky kódu a celé zobrazenie je pripravené k použitiu. Je dôležité nezabudnúť zavolať konštruktor predka, aby sa celé zobrazenie inicializovalo. Príklad je vo výpise 4.2.

Kód 4.2: Vytvorenie vlastného zobrazenia pseudokódu.

```
public class MyCodeView extends CodeView{
    public static final String NAME = "Pseudocode";

    public MyCodeView(){
        super(NAME, null); // important, null for icon
        addLine("If(I.studied.hard)");
        addLine("  return A;");
        setHighlightLine(1);
    }
}
```

Textový výpis do konzole

Toto zobrazenie vypisuje textové informácie vo forme konzoly. Každý pridaný riadok sa pripojí na koniec textu. Slúži pre detailný popis vykonávaného kroku algoritmu a je užitočná pri krokovaní. Text je vložený do textového panela, ktorý je prístupný cez `getTextPane()`. Textu sa dá meniť veľkosť, ale táto funkcia je podrobnejšie popísaná v sekcii o ovládaní užívateľského rozhrania 4.5.

Základné operácie so zobrazením sú:

- Pridanie riadka - `printLine(String)` pripojí nový riadok na koniec textu v konzole.
- Zmazanie obsahu - `clearConsole()` zmaže všetok obsah.

Konzola nepotrebuje žiadne špeciálne informácie a preto si pri vytvorení jej podtriedy treba dať pozor iba na zavolanie konštruktoru nadtriedy.

Kód 4.3: Vytvorenie vlastnej konzoly.

```
public class MyConsoleView extends ConsoleView{
    public static final String NAME = "Console";

    public MyConsoleView(){
        super(NAME, null); // important, null for icon
    }
}
```

Kreslenie

Toto je zobrazenie pre vykresľovanie vizualizácie. Obsahuje zoznam tvarov `PaintableShape`. Tie sa vykresľujú v metóde `paintComponent(Graphics)`. Trieda spracováva udalosti myši a nimi sa dajú meniť tvary. To je popísané v sekcii 4.5.

Základné operácie so zobrazením sú:

- Pridanie tvaru - sa robí metódou `addShape(PaintableShape)`.
- Odstránenie tvarov - `clearShapes()` zmaže všetky tvary.
- Získanie tvarov - `getShapes()` sprístupňuje všetky tvary zobrazenia.

Ak si programátor vytvorí vlastné zobrazenie, opäť musí zavolať konštruktor nadtriedy. Ak chce vykresľovať vlastné prvky spolu s tvarmy, ktoré sú uložené v nadtriede je treba zavolať jej metódu pre kreslenie ako je ukázané vo výpise 4.4.

Kód 4.4: Vytvorenie vlastného zobrazenia pre kreslenie.

```
public class MyDrawView extends DrawView{
    public static final String NAME = "Draw";

    public MyDrawView(){
        super(NAME, null); // important, null for icon
    }

    @Override
    protected void paintComponent(Graphics g){
        super.paintComponent(g);//DrawView's shapes will be painted

        // My painting...
    }
}
```

Mapa

Toto je špecializované zobrazenie pre kreslenie mapy. Mapa pozostáva zo štartu, cieľa a množiny prekážok. Spracuje udalosti myši, ktorými sa dajú tieto elementy meniť priamo v užívateľskom rozhraní.

Základné operácie so zobrazením sú:

- Pridávanie a odstránenie objektov mapy - pridávanie cez `addObject(PaintableShape)`. Tieto objekty, by ale mali byť podtriedy objektov z balíčka `model.map`, aby fungovali s ostatnými metódami. Objekt sa odstráni metódou `removeObject(PaintableShape)`.
- Vyčistenie mapy - `clearMap()` zmaže celú mapu.
- Získanie elementov - štart sa získa metódou `getStart()`, cieľ `getGoal()` a zoznam prekážok pomocou `getObstacles()`. Všetky objekty mapy sú prístupné metódou `getMapObjects()`.
- Dotaz na priesečník s prekážkou - trieda má metódu `isInObstacle(PaintableShape)`, ktorá zistí, či nejaký objekt koliduje s prekážkou.

Mape sa dá nastaviť zvýraznený tvar. Slúži najme počas krokovania algoritmu, kedy je výhodné zobraziť tvar pre užívateľa, s ktorým sa práve manipuluje. Zvýraznený tvar sa nakreslí nad elementy mapy a použije sa iná farba. Dá sa mu nastaviť aj popisok a použitie je ukázané vo výpise 4.5.

Vytvorenie vlastnej podtriedy zobrazuje výpis 4.5. Podobne ako `DrawView` vykresľuje elementy mapy metódou `paintComponent(Graphics)`, takže treba zavolať nadtriedy pre zobrazenie elementov mapy.

Kód 4.5: Vytvorenie vlastnej mapy.

```
public class MyMapView extends MapView{
    public static final String NAME = "Map";

    public MyMapView(){
        super(NAME, null); // important, null for icon
    }

    @Override
    protected void paintComponent(Graphics g){
        super.paintComponent(g); // painting of map elements

        // My painting ...
    }
}
...
setHighlightedShape(map.getStart());
setHighlightedShapeLabel("Starting position");
map.repaint();
```

Nastavenie vstupných hodnôt

Obsahuje vstupné komponenty pre vkladanie hodnôt parametrov algoritmu alebo prispôsobenie užívateľského rozhrania. V balíčku `view.uiComponents.input` sú niektoré vstupné komponenty, ktoré boli stručne popísané v sekcii 3.1.

Základné operácie so zobrazením sú:

- Pridanie vstupnej komponenty - `add(Component)` pridá komponentu do panela.
- Vyčistenie panela - `removeAll()` odstráni všetky komponenty.

Vytvorenie zobrazenia a pridanie vstupných komponent ilustruje výpise 4.6.

Kód 4.6: Vytvorenie zobrazenia pre parametre.

```
public class MyParameterView extends ParameterView{
    public static final String NAME = "Parameters";

    public MyParameterView(){
        super(NAME, null); // important, null for icon
        // Component from view.uiComponents.input
        add(new IntegerField("Key", 3));

        JPanel panel = new JPanel();
        panel.add(new JTextField());
        panel.add(new JLabel("Text field"));
        add(panel);
    }
}
```

Statusové informácie

Toto zobrazenie slúži pre stavové informácie. Obsahuje progres bar a textové pole. Tieto komponenty sú prístupné cez svoje gettery.

Základné operácie so zobrazením sú:

- Nastavenie progresu - `setProgress(int)` zvýrazní percento vykonávanej úlohy.
- Status text - `setStatus(String)` vypíšet užívateľovi krátku správu.

Vytvorenie vlastného statusového zobrazenia je ukázané na výpise 4.7.

Kód 4.7: Vytvorenie status baru.

```
public class MyStatusBarView extends StatusBarView{
    public static final String NAME = "Status";

    public MyStatusBarView(){
        super(NAME, null); // important, null for icon
    }
}
```

Textový popis

Toto zobrazenie slúži pre textovú informáciu. Obsahuje textový panel prístupný cez getter, v ktorom sa text zobrazí. Dá sa formátovať v jazyku HTML. Slúži najmä na statické informácie o algoritme. Je to napríklad jeho funkcia, vysvetlenie parametrov, ovládanie aplikácie.

Základné operácie so zobrazením sú:

- Nastavenie textu - metódou `setText(String)` sa do panela nastaví text. Text môže obsahovať HTML značky, ktoré sa sformátujú vo výstupnom paneli.

Ak chceme vytvoriť zobrazenie s informáciami o algoritme, zdedíme textové zobrazenie a nastavíme mu informatívny text.

Kód 4.8: Vytvorenie textového zobrazenia.

```
public class MyTextView extends TextView{
    public static final String NAME = "Text";

    public MyTextView(){
        super(NAME, null); // important, null for icon
        setText("<h4>Algorithm</h4><br><p>Description.");
    }
}
```

Panel nástrojov

Zobrazenie, ktoré obsahuje tlačítka pre ovládanie aplikácie. Obsah panelu sa mení podľa stavu aplikácie. Má v sebe mapu, ktorá mapuje `State` na zoznam komponentov, ktoré sa majú v paneli zobraziť.

Základné operácie so zobrazením sú:

- Pridanie obsahu panela - obsah panela pre jeden stav aplikácie sa pridá metódou `addComponentsForState(State, Component...)`. Obsah sa musí pridať pre všetky stavy.

Vytvorenie panela nástrojov ilustruje výpis 4.9.

Kód 4.9: Vytvorenie panela nástrojov.

```
public class MyToolbarView extends ToolbarView{
        public static final String NAME = "Text";

    public MyToolbarView(){
        super(NAME, icon); // important, null for icon
        JButton edit = new EditButton();
        JButton sim = new SimulationButton();

        addComponentsForState(State.INIT, edit, sim);
        addComponentsForState(State.EDIT, edit, sim);
        addComponentsForState(State.READY, edit, sim);
        addComponentsForState(State.RUNNING, edit, sim);
        addComponentsForState(State.PAUSED, edit, sim);
        addComponentsForState(State.FINISHED, edit, sim);
        addComponentsForState(State.STEP_BACK, edit, sim);
        addComponentsForState(State.STEP, edit, sim);
        addComponentsForState(State.LARGE_STEP, edit, sim);
        addComponentsForState(State.LARGE_STEP_BACK, edit, sim);
    }
}
```

Mriežková mapa

Toto zobrazenie obsahuje mriežku štvorcových buniek. Každá bunka má svoj typ, definovaný výčtom `GridElement`. Jedným z typov je aj štart a cieľ. Zobrazenie sa používa pre demonštráciu činnosti hľadania cesty v bunkovej mape, kde bunky môžu mať rôzne vlastnosti.

Veľkosť buniek sa dá meniť troma spôsobmy.

1. Priama zmena volaním metódy `setCellSize(int)`, ktorá vymaže všetky bunky a vytvorí nové s novou veľkosťou tak, aby zaberali celý obsah zobrazenia.

2. Zobrazenie obsahuje kľúč do globálnych vlastností `GridMapView.CELL_SIZE_KEY`. Hodnota vlastnosti by mala byť celé číslo. Keď sa zmení, tak bude mapa na zmenu upozornená a automaticky nastaví novú veľkosť bunky. Dá sa to dobre použiť s komponentou v užívateľskom rozhraní, napríklad spinner.

3. Mapa implementuje poslúchač udalostí myši. Keď užívateľ podrží tlačidlo `ctrl` alebo `shift` a posunie koliekom myši, tak sa zmení veľkosť buniek. Toto funguje len vtedy, keď je aplikácia v editovacej fáze.

Mapa poskytuje užitočnú funkciu pre vrátenie susedov bunky. `get4Neighbours(Cell)` vráti štyroch susedov, s ktorými bunka zdieľa hranu. Pre získanie bunky podľa súradníc slúžia metódy `getCellAtGridPosition(int row, int column)` a `getCellAtPosition(double`

`x, double y)`. Prvej sa zadajú súradnice mriežky a druhá používa súradnice zobrazenia(napríklad pozícia myši).

Pri vizualizácii je užitočné zvýrazniť tvar, s ktorým sa manipuluje. Preto dokáže mapa zvýrazniť tvar, ktorý sa nastaví. Metódou `setHighlightedShape(PaintableShape)` a setHighlightedShapeLabel(String) sa nastaví tvar a jeho popis, ktoré sa vykreslia spolu s elementami mapy. Použitie spomínaných funkcií ilustruje výpis 4.10.

Kód 4.10: Vytvorenie mriežkovej mapy.

```
public class MyGridMap extends GridMapView{
   public static final String NAME = "Map";

   public MyGridMap(){
         // null for icon, cell factory creates new cells
      super(NAME, null, new CellFactory());
   }
}
...
setHighlightedShape(map.getCellAtGridPosition(0,0));
setHighlightedShapeLabel("Cell at position 0,0");
map.repaint();
```

Vytvorenie vlastného rámca

Zobrazenia sa vkladajú do hlavného panela. Rozloženie sa dá manuálne naprogramovať, ale pre väčšie pohodlie sú v balíčku `view.frames` definované rámce. V ich konštruktore sa pridajú zobrazenia a hlavný panel a rámec potom sám nastaví zobrazenia a ich pozície do hlavného panela v závislosti na stave aplikácie. Rámce implementujú rozhranie `Frame`, ktoré má jedinú metódu `positionViews()`. V nej sa rozložia zobrazenia v hlavnom paneli podľa toho, aký je stav aplikácie.

Ak žiadny vytvorený rámec nevyhovuje požiadavkám implementovaného algoritmu, môže si programátor vytvoriť vlastný. Nový rámec bude implementovať rozhranie `Frame` a ak má fungovať automaticky (reagovať na zmenu stavu aplikácie), tak aj rozhranie `Observer`. Ukážka implementácie je vo výpise 4.11.

Keď sa vytvorí nový hlavný panel, obsahuje jediné prázdne zobrazenie, ktoré zaberá celý jeho obsah. Aj zavolanie metódy `clearDisplayedViews()` má rovnaký účinok. Výmena jedného zobrazenia za druhé sa robí cez `MainPanel.changeView(oldView, newView)`. To znamená, že bol hlavný panel práve vytvorený, alebo jeho zobrazené zobrazenia vymazané, `mp.changeView(mp.getDisplayedViews().get(0), view);` vymení prázdne zobrazenie za zobrazenie v druhom parametri.

Zobrazenie sa dá rozdeliť na dva metódou `splitView(displayedView, newView, edge)`. V prvom parametri je zobrazenie, ktoré musí byť v hlavnom paneli. Druhý parameter má naopak zobrazenie, ktoré nesmie byť práve zobrazované v paneli. Tretí parameter udáva hranu, vzhľadom ku ktorej sa má zobrazenie rozdeliť. Ak je to napríklad severná hrana, `displayedView` sa rozdelí horizontálne na polovice. Vyššia polovica bude obsahovať `newView` a spodná `displayedView`. To znamená, že nové zobrazenia bude bližšie k hrane, ktorá bola zvolená.

Hlavný panel ukladá zobrazenia do mriežky. Dimenziu tejto mriežky má uloženú a prístupnú vo verejnej statickej vlastnosti `MAX_CELLS` a má hodnotu 100. To znamená, že ak je

zobrazené len jedno zobrazenie, zaberá 100 buniek horizontálne a 100 vertikálne. Ak toto zobrazenie rozdelíme horizontálne, tak budú oba zaberať 50 buniek horizontálne. Metóda `resizeView(view, -49, edge)` mení počet buniek zobrazenia. V prvom parametri je zobrazenie, ktorému chceme zmeniť veľkosť. Druhý parameter udáva počet buniek. Záporná hodnota bunky uberá, kladná pridáva. Ak uberieme jednému zobrazeniu x buniek, druhé zobrazenie, ktoré s ním susedí x buniek získa. V treťom parametri je hrana, ktorá určí z ktorej strany sa zmena aplikuje.

Kód 4.11: Vytvorenie rámca.

```
public class MyFrame implements Frame, Observer{
  public MyFrame(MainPanel mp, View toolbar, View draw,
                 View parameters, View code, View console){
    // Saving parameters to fields

    Properties.INSTANCE.addObserver(this);
  }

  @Override
  public void positionViews(){
    if(mp != null){
      mp.clearDisplayedViews();
      switch(State){
        case INIT:
        case EDIT:
          mp.changeView(mp.getDisplayedViews().get(0), draw);
          mp.splitView(draw, toolbar, Direction.NORTH);
          mp.resizeView(toolbar, -49, Direction.SOUTH);
          mp.splitView(draw, parameters, Direction.EAST);
          mp.resizeView(parameters, -15, Direction.WEST);
          break;
        case READY:
        case RUNNING:
        case PAUSED:
        case STEP:
        case LARGE_STEP:
        case STEP_BACK:
        case LARGE_STEP_BACK:
        case FINISHED:
          mp.changeView(mp.getDisplayedViews().get(0), draw);
          mp.splitView(draw, toolbar, Direction.NORTH);
          mp.resizeView(toolbar, -49, Direction.SOUTH);
          mp.splitView(draw, code, Direction.EAST);
          mp.resizeView(code, -15, Direction.WEST);
          mp.splitView(code, console, Direction.SOUTH);
          mp.resizeView(console, -25, Direction.NORTH);
          break;
      }
    }
```

```
  }

    @Override
    public void update(Observable obs, Object obj){
        if(obj instanceof State){
            positionViews();
        }
    }
}
```

4.2 Geometrické tvary a dátové štruktúry

Knižnica obsahuje triedy pre základné geometrické tvary. Postykujú geometrické operácie ako priesečník, obsah. Sú podrobne popísané v knižnici [6]. Okrem toho sú schopné sa vykresliť s použitím objektu `Graphics`. Všetky tvary implementujú rozhranie `PaintableShape`, ktoré definuje metódy pre vykresľovanie `paint(...)`. Dá sa nastaviť farba pre výplň a obvod tvaru, ale aj hrúbka čiary. Preddefinované hodnoty sú prístupné v rozhraní.

Knižnica pracuje s instanciami rozhranie `PaintableShape`. Pri vytvorení vlastnej dátovej štruktúry je preto výhodné, implementovať toto rozhranie. Ak bude novo vytvorený tvar potrebovať špeciálne vykresľovanie, je potrebné prepísať metódu `void paint(Graphics g, float strokeWidth, Color line, Color fill)`. Príkladom vlastného vykresľovania je trieda `Start` v balíčku `model.map`. Reprezentuje štartovací bod v mape. V metóde `paint(...)` nakreslí elipsu a na ňu obrázok so štartovacou ikonou (ikony sú popísané v sekcii 4.3). Príklad je vo výpise 4.12.

Kód 4.12: Vykreslenie štartu.

```
private static final int ICON = 50;
@Override
public void paint(Graphics g, float stroke, Color line, Color fill){
  int size = ICON + 10;
  Ellipse background = new Ellipse(getX(), getY(), size, size);
  background.paint(g, strokeWidth, Properties.BORDER, WHITE);
  Icon start =
    Properties.INSTANCE.getFontAwesomeIcon(ICON, "\uf11d", BLACK);
  start.paintIcon(null, g, getX() - ICON / 2, getY() - ICON / 2);
}
```

Rozhranie má ešte dve dôležité metódy. Každý tvar je instancia triedy `Geometry` z knižnice JTS [6]. Táto trieda poskytuje základné geometrické operácie. Keďže moja knižnica pracuje s tvarmy ako instanciami rozhrania `PaintableShape`, musí toto rozhranie definovať metódu pre získanie `Geometry`, aby sa mohly jej operácie využiť. Pre základné tvary stačí vrátiť `return this;`, ale pre zložitejšie dátové štruktúry, ktoré pozostávajú z kolekcie rôznych tvarov, to bude záležať na ich funkcii. Napríklad trieda `SimpleGraph` v balíčku `model.graph` vracia `null`, pretože nepotrebuje geometrické operácie s jej prvkami.

Ďalšou metódou je `resize(double x, double y)`, ktorá slúži na zmenu tvaru vzhľadom k súradniciam v parametroch. Každý tvar jej funkciu implementuje inak. Napríklad bod zmení svoje súradnice na (x,y). Polygon nájde svoj najbližší bod k týmto súradniciam a zmení jeho pozíciu. Ak je nový tvar odvodený od základných, tak stačí ponechať

implementáciu tejto metódy na predka. Ak si programátor neželá, aby sa dalo s objektom manipulovať, alebo to nie je vhodné, potom ponechá túto metódu prázdnu. Trieda `SimpleGraph` nepotrebuje, aby sa menila pozícia jej uzlov a preto ju má prázdnu.

Dôležitou dátovou štruktúrou je graf. Použil som knižnicu JGraphT [15], ktorá obsahuje rôzne grafy spolu s ich operáciami. Jednoduchý neorientovaný graf `SimpleGraph` implementuje rozhranie `PainableShape` a vie sa vykresliť na zobrazovaciu plochu. Jeho vrcholy musia byť podtriedy `SimpleVertex` a hrany `SimpleEdge`. Vytvorenie grafu vyžaduje, aby v konštruktore dostal továreň, ktorá vytvára hrany. Jednoduchá hrana `SimpleEdge` má verejnú statickú továreň pre jej vytvorenie. Jednoduchý graf je generický a vytvorenie instancie ilustruje výpis 4.13.

Kód 4.13: Vytvorenie grafu

```
SimpleGraph<SimpleVertex, SimpleEdge> graph =
  new SimpleGraph<>(new SimpleEdge.SimpleEdgeFactory<>());
SimpleVertex n1 = new SimpleVertex(100, 100);
SimpleVertex n2 = new SimpleVertex(300, 300);
graph.addVertex(n1);
graph.addVertex(n2);
graph.addEdge(n1, n2);
graph.paint(getGraphics());
```

4.3 Práca s globálnymi objektami

Globálne premenné sú uložené v triede `Properties`. Je to jedináčik implementovaný ako `enum`. Pristupuje sa k nemu cez `Properties.INSTANCE`.

Premenné sú uložené v mape, kde reťazec je kľúč a objekt je hodnota. Pristupuje sa k nim cez getter a setter ako ukazuje výpis 4.14. Podobne sa získa stav aplikácie.

Používa aj návrhový vzor pozorovateľ a triedy sa u nej môžu zaregistrovať a byť upozornení na zmenu stavu alebo premennej. Pridanie pozorovateľa znázorňuje výpis 4.14.

Slová v aplikácii sa zvyčajne poskytujú v externom súbore, čo uľahčuje lokalizáciu. Preto aj moja aplikácia používa reťazce zo súboru `Strings_en.properties`. Dá sa k nim pristupovať cez metódu `Properties.INSTANCE.getLocaleString(String)`, kde reťazec v parametri udáva kľúč do súboru s lokalizovaným textom.

Aplikácia používa ikony na oživenie užívateľského rozhrania. V poslednej dobe sa najmä vo webovom svete často používa Font Awesome [9]. Ikony sú vložené do fontu, ktorý aplikácia používa (`vaflevut_regular.ttf`). Pre sprístupnenie ikon slúži metóda `getFontAwesomeIcon(int, String, Color)`, ktorej sa zadá veľkosť ikony, znak reprezentujúci ikonu (na stránke Font Awesome [9] sa dá zistiť aký znak patrí k akej ikone) a jej farba.

Kód 4.14: Práca s globálnymi premennými.

```
// Changing to edit state
Properties.INSTANCE.setState(State.EDIT);
// Getting current application state
State current = Properties.INSTANCE.getState();

// Setting global property key
Properties.INSTANCE.setProperty("key", new Integer(1));
```

```
// Retrieving global property key
Integer i = (Integer) Properties.INSTANCE.getProperty("key");

// Adding observer for listening for changes
// Class must implement java.util.Observer
Properties.INSTANCE.addObserver(this);

// Getting locale string
Properties.INSTANCE.getLocaleString("key");

// Getting icon
Properties.INSTANCE.getFontAwesomeIcon(ICON_SIZE,
                        "\uf11d", Properties.BLACK);
```

4.4 Vytvorenie algoritmu

Knižnica uľahčuje tvorbu vizualizácie algoritmov. Tieto algoritmy by mali bežať v samostatnom vlákne, aby neblokovali užívateľské rozhranie. Mali by podporovať krokovanie pre jednoduchšie pochopenie ich činnosti. Užívateľ by mal byť schopný prechádzať kroky svojím tempom dopredu aj dozadu.

V balíčku `model.algorithm` je abstraktná trieda pre algoritmus `AlgClass`. Poskytuje podporu pre vyššie zmienené funkcie. Ovláda simuláciu na základe stavu aplikácie. Implementuje rozhranie `Runnable` pre beh v samostatnom vlákne.

Stav aplikácie je popísany v kapitole 3.2. Trieda pre algoritmus má metódy, ktoré ho využívajú. Napríklad metóda `waitOnSmallStep()` pri stave `State.STEP` zmení stav na `State.PAUSED`. Tým pozastaví výpočet a čaká, kým sa znova nezmení stav. Vracia `true` ak pozastavila výpočet. To znamená, že ak počas simulácie užívateľ stlačí tlačidlo pre malý krok, aplikácia zmení svoj stav na `State.STEP`. Keď algoritmus narazí na túto metódu, zmení stav aplikácie na `State.PAUSED` a čaká na zmenu stavu. Tým sa vykoná jeden krok algoritmu a užívateľ môže pozorovať zmeny a urobiť ďalší krok. Podobne fungujú aj ostatné metódy začínajúce na `wait...`. Metóda `waiting()` je alternatíva, ktorá združuje volania všetkých `wait...` metód. Používa sa pre implementáciu krokovania, pretože čaká na každom kroku aj počas výpočtu.

Spomaliť výpočet sa dá aj iným spôsobom ako len krokovaním. `AlgClass` má premennú `delay` prístupnú cez getter a setter, kde sa dá nastaviť doba čakania v milisekundách medzi krokmy. Ak je algoritmus spustený, tak po každom príkaze počká zadanú časovú jednotku. Čakanie sa dá nastaviť aj cez globálne premenné. Trieda má kľúč `AlgClass.DELAY_KEY`, ktorý používa pre získanie hodnoty z `Properties`. Ak je hodnota s týmto kľúčom celé číslo a zmení sa (napríklad ju nastaví nejaká vstupná komponenta užívateľského rozhrania), tak sa automaticky nastaví oneskorenie medzi krokmy algoritmu.

Krokovanie je implementované pomocou príkazov. Jeden príkaz predstavuje jeden krok. Tento krok musí byť možné vykonať dopredu aj dozadu. Používa sa abstraktná trieda `UndoableCommand` v rovnakom balíčku ako trieda pre algoritmus. Má dve dôležité metódy `redo()` a `undo()`, z ktorých prvá vykoná príkaz a druhá ho vráti. V konštruktore sa dá pridať pole objektov, ktoré sa potom môžu použiť vnútri tejto triedy a sú prístupné metódou `getObject(i)`, ktorá vráti i-ty objekt zadaný v konštruktore. Niekedy je vhodné preskočiť niekoľko krokov na ďalší významný príkaz alebo sekciu v algoritme. Aby užívateľ nemusel

krokovať všetky malé kroky, môže vykonať veľký skok. `UndoableCommand` má preto metódu `waitOnLargeStep()`, ktorá vracia `true`, ak má čakať na tomto príkaze v prípade veľkého skoku. Príklad použitia príkazu je vo výpise 4.15. Ukazuje príkaz pre pridanie štartu do grafu a inverznú operáciu odstránenie štartu. Prepísanie metódy `waitOnLargeStep()` zaručí, že sa na tomto príkaze počká v prípade veľkého skoku.

`AlgClass` má zoznam príkazov, ktoré tvoria simuláciu. Príkaz sa pridáva metódou `addCommand(UndoableCommand)`. Simulácia týchto príkazov sa spustí metódou `simulate`, ktorej sa v konštruktore predá zoznam príkazov, ktoré má odsimulovať. Príkazy uložené v triede sú prístupné cez getter. Simulácia sa potom ovláda zmenou stavu aplikácie.

Zhrniem to, čo bolo popísané vyššie. Príkazy algoritmu sú reprezentované triedou `UndoableCommand`, ktorá dokáže príkaz vykonať aj vrátiť. Celý algoritmus potom predstavuje zoznam týchto príkazov. Po zavolaní metódy `AlgClass.simulate(getCommands())` sa spustí simulácia príkazov. Tá sa ovláda cez stav aplikácie. To znamená, že keď užívateľ klikne na malý krok, vykoná sa jeden príkaz a výpočet sa pozastaví, kým nezmení znova stav (napríklad opäť na malý krok). Krokom späť sa vykoná inverzný príkaz (undo) a `AlgClass` sa posunie v zozname príkazov o jeden späť.

Kód 4.15: Vytvorenie príkazu.

```
addCommand(new UndoableCommand(){
    @Override
    public void redo(){
      graph.addVertex(start);
    }

    @Override
    public void undo(){
      graph.removeVertex(start);
    }

    @Override
    public boolean waitOnLargeStep(){
      return true;
    }
});
```

Je možné simuláciu ovládať pomocou zmeny stavu aplikácie (popísané v kapitole 3.2). Keď sa aplikácia prepne do stavu pripravenosti simulácie (`State.READY`), tak sa vytvorí nové vlákno s algoritmom. Po spustení simulácie (`State.RUNNING`, `State._STEP`) sa spustí vlákno a vykonávajú sa kroky. Vlákno sa preruší, ak výpočet skončí prirodzene, alebo ho preruší užívateľ (`State.FINISHED`, `State.EDIT`). Tento postup využívajú mnohé algoritmy a nie je potreba ho implementovať pre každý samostatne. Knižnica obsahuje hlavný panel `AlgClassMainPanel`, ktorému sa v konštruktore predá trieda s algoritmom a sám bude vytvárať nové vlákna s výpočtom podľa zmeny stavu aplikácie.

4.5 Ovládanie užívateľského rozhrania

V tejto kapitole popíšem ovládanie užívateľského rozhrania pre užívateľa, ktorý si spustí algoritmus.

Užívateľské rozhranie je prispôsobivé. Keď sa ukazateľ myši priblíži k hrane dvoch zobrazení, zmení sa mu ikona. Pri potiahnutí myši sa posunie hranica medzi zobrazeniami. Kliknutím pravým tlačidlom myši na hranu sa zobrazí kontextové menu. Má dve možnosti pre spojenie a rozdelenie zobrazení. Ak majú dva zobrazenia spoločnú hranu, dajú sa spojiť do jedného. Jedno zobrazenie sa dá rozdeliť do dvoch. Rozdelí sa podľa toho, na akej hrane sa zobrazí kontextové menu. Pri kliknutí pravým tlačidlom kdekoľvek vnútri zobrazenia sa zobrazí kontextové menu, ktoré obsahuje všetky zobrazenia, ktoré má algoritmus implementované. Zvolením jedného, ktoré práve nie je aktívne sa dá nahradiť za toto zobrazenie. Akcie ilustrujú obrázky 4.1.

Obr. 4.1: Manipulácia zobrazení.

V textových zobrazeniach sa dá meniť veľkosť textu. Sú dve možnosti, `[ctrl/shift]+[koliesko myši]` alebo `[ctrl]+[+]`, `[crtl]+[-]`. Aj v mape `GridMapView` sa dá meniť veľkosť buniek použitím kolieska myši.

Na ovládanie aplikácie sa používajú tlačítka s ikonami:

- Prepne aplikáciu do editovacieho režimu.
- Prepne aplikáciu do simulačného režimu.
- Spustí simuláciu. Ak je simulácia pozastavená, tak bude pokračovať.
- Pozastaví simuláciu.
- Zastaví simuláciu.
- Reštartuje simuláciu.
- « Veľký krok späť.
- < Malý krok späť.
- > Malý krok.
- » Veľký krok.

Vo vykresľovacích zobrazeniach (`DrawView`) a `MapView` sa dajú v editovacej fáze meniť vlastnosti tvarov, ktoré sú v nich zobrazené. Ak sa užívateľ priblíži myšou k nejakému tvaru, zvýrazní sa. Toto zvýraznenie naznačuje, že sa s ním dá manipulovať. Napríklad kliknutím

a potiahnutím myši sa dajú meniť pozície bodov, alebo veľkosť štvoruholníka alebo elipsy. Mapa má navyše akcie aj na pridávanie tvarov. Pri zvolení tlačidla na paneli nástrojov v editovacom režime sa dá pridať štart a cieľ a aj prekážky kliknutím myši na zobrazovaciu plochu.

Kapitola 5

Aplikácie s algoritmami pre hľadanie cesty

V tejto kapitole popíšem, ako som implementoval algoritmy popísané v úvode práce pomocou mnou vytvorenej knižnice. Algoritmy majú niektoré črty spoločné. Každý používa mapu a má rovnaké rozloženie zobrazení. Aplikácia má dve hlavné fáze. Editovaciu, v ktorej sa vytvára mapa a menia parametre algoritmu a simulačnú fázu, kde sa vizualizuje hľadanie cesty v mape.

Mapa sa vytvára v editovacej fáze. V paneli nástrojov sa zobrazia tlačítka na pridanie elementov:

- Po zvolení tlačítka a kliknutí na mapu v editovacej fáze sa pridá štartovací bod.
- Po zvolení tlačítka a kliknutí na mapu v editovacej fáze sa pridá cieľový bod.
- Po zvolení tlačítka a kliknutí na mapu v editovacej fáze sa pridá jeden bod prekážky. Pre pridanie celej prekážky je nutné polygón uzatvoriť (pridaním posledného bodu do tesnej blízkosti prvého bodu).
- Po zvolení tlačítka sa dajú odstraňovať objekty mapy. Pri priblížení k objektu sa zvýrazní bod, ktorý je blízko kurzora a po kliknutí sa celý objekt, ktorého je bod súčasťou odstráni.

Nastavovanie buniek v mriežkovej mape sa deje pomocou:

- Po zvolení tlačítka a kliknutí na mapu v editovacej fáze sa nastaví štartovací bod.
- Po zvolení tlačítka a kliknutí na mapu v editovacej fáze sa nastaví cieľový bod.
- Nastavenie bunky s najlepšou cenou prechodu.
- Nastavenie bunky s lepšou cenou prechodu.
- Nastavenie bunky s normálnou cenou prechodu.
- Nastavenie bunky s horšou cenou prechodu.

- ⩔ Nastavenie bunky s najhoršou cenou prechodu.
- ✖ Nastavenie prekážky. Na túto bunku sa nedá prejsť.

V paneli nástrojov sú tlačítka na pridávanie elementov mapy, v pravo je zobrazenie s parametrami algoritmu. V strede je mapa, ktorá obsahuje štart vľavo a cieľ vpravo. Je tam vidno aj dokončenú prekážku, ktorá je celá čierna. Modré úsečky sú časťou nedokončenej prekážky. Pri kliknutí do mapy sa k nej pridá ďalší bod. Ak sa klikne v blízkosti prvého bodu (znázornené ako modrá elipsa), tak sa prekážka uzatvorí a pridá do mapy. Obrázky sú v prílohe E

V simulačnej fáze sa vizualizuje algoritmus. Na paneli nástrojov sú tlačítka na ovládanie simulácie spolu s nastavením oneskorenia medzi krokmy. V pravej časti je pseudoalgoritmus s konzolou, kde sa zobrazujú informácie o práve vykonanom kroku. V strede sa vizualizujú algoritmy so zvýraznením manipulovaného objektu. Mriežková mapa má namiesto konzoly zobrazenie s množinami, ktoré algoritmy používajú. Typicky sa jedná o frontu pre zoznam buniek, ktoré sa budú expandovať a množinu buniek, ktoré už boli preskúmané.

V ďalšej kapitole popíšem konkrétne algoritmy a aplikácie, ktoré ich vizualizujú. Snímky obrazovky sa nachádzajú v prílohe E a ukazujú ako vyzerá užívateľské rozhranie.

5.1 Implementácia algoritmov

Algortimus PRM bol popísaný v kapitole 2.2.1. Implementoval som ho s pomocou piatich zobrazení. Zobrazenia pre konzolu, pseudokód, parametre a panel nástrojov som nemusel nijako funčne modifikovať, len som im nastavil obsah pre tento algoritmus v ich konštruktore. Mapa má tiež väčšinu svojej funkcionality a vykresľovania z nadtriedy. Tým, že knižnica poskytuje základné zobrazenia mi podstatne uľahčilo prácu na vizualizácii.

Hlavný panel pre algoritmus PRM, vytvorí v konštruktore všetky zobrazenia a použije rámec `Frame1`, ktorý ich automaticky rozloží podľa stavu aplikácie. Okrem zobrazení potrebuje aj vlákno s algoritmom. Pre jeho ovládanie som použil triedu `AlgClassMainPanel`, ktorá automaticky (podľa stavu aplikácie) vytvára a ovláda vlákno, v ktorom beží. Okrem toho ešte implementuje metódy z nadtriedy, ktoré vracajú reťazce s informáciami o algoritme a autorovi.

Najzložitejšia časť je algoritmus. Dedí od triedy `AlgClass` popísanej v 4.4. Využíva simulačné možnosti z nadtriedy, najmä pridanie príkazu a jeho simulácia. Algoritmus PRM je implementovaný podľa popisu z kapitoly 2.2.1. Funguje tak, že sa celý najprv vykoná a až potom sa odsimuluje. To znamená, že počas výpočtu pridáva príkazy pre simuláciu, ako je ukázané vo výpise 5.1. Metóda `changeUI()` nemanipuluje so štruktúrami algoritmu, ale len aktualizuje užívateľské rozhranie. Nastaví číslo riadka v zobrazení pseudokódu, vypíše informácie do konzoly a nastaví tvar, ktorý sa má v mape zvýrazniť spolu s jeho popiskom. Na konci algoritmu sa pridané príkazy odsimulujú metódou `simulate(getCommands())`, kde `getCommands()` vráti zoznam všetkých príkazov, ktoré sa pridali počas výpočtu algoritmu.

Tento spôsob simulácie má nasledujúce vlastnosti. Celý výpočet prebehne ešte predtým, ako sa spustí simulácia pre užívateľa. To znamená, že ak by nastal problém počas výpočtu, nespustí sa ani časť, ktorá bola v poriadku. Na začiatku algoritmu sa náhodne vygenerujú body, ktoré reprezentujú konfiguračný priestor. Do simulácie sa pridá len príkaz pridania (odstránenia) tohoto bodu do grafu, takže pri opätovnom vykonaní kroku sa nevygeneruje iný náhodný bod, ale pridá sa ten istý. Celá simulácia je deterministická a nezáleží na tom, ako v nej bude užívateľ krokovať. Ukážka aplikácie je na obrázku E.4 v prílohách.

Kód 5.1: Vytvorenie príkazu.

```
// Start of algorithm ...
addCommand(new UndoableCommand(){
  @Override
  public void redo(){
    graph.addVertex(node);
    changeUI(4,
      "Adding point to graph. "+graph.vertexSet().size()+" of "+n,
      node, node.toString());
  }

  @Override
  public void undo(){
    graph.removeVertex(node);
    changeUI(4, "Removing point from graph. " + node, null, null);
  }
});
// Rest of algorithm ...

// Simulation
simulate(getCommands());
```

Implementoval som ďalšie pravdepodobnostné algoritmy EST a RRT z kapitol 2.2.2 a 2.2.3. Sú veľmi podobné PRM, líšia sa len algoritmom konštruovania grafu. V editovacej fáze sa okrem mapy nastavia parametre algoritmu. Spustenie simulácie a krokovanie funguje rovnako ako pri algoritme PRM. Ukážky sú na obrázkoch E.5 a E.6.

Aplikácia s grafom viditeľnosti vizualizuje algoritmus z kapitoly 2.3.3. Princíp implementácie je podobný ako pri pravdepodobnostných algoritmoch. Na rozdiel od nich ale negeneruje body grafu náhodne. Používa body prekážok a je úplný, to znamená, že nájde cestu vždy keď existuje. Nemusia sa mu nastavovať žiadne parametre, takže v editovacej fáze sa vytvorí iba mapa. Ukážka simulácie je na obrázku E.7.

Pre grafové algoritmy som zvolil mriežkovú mapu. Bunka predstavuje vrchol a hrana medzi dvoma bunkami prechod. V editovacej fáze sa dajú nastavovať rôzne typy buniek. Líšia sa od seba cenami prechodov.

Algoritmy, ktoré hľadajú cestu v tejto mape sú BFS (kapitola 2.4), Dijkstrov algoritmus (kapitola 2.4) a A* (kapitola 2.4). Pre všetky som vytvoril podtiedu bunky `Cell` a pridal som do nej informáciu o tom, z akej bunky sme sa do nej dostali. Používa sa pri zostavení nájdenej cesty. Dijkstrov algoritmus a A* používajú pre hľadanie cesty informácie o prejdenej ceste zo štartu. Preto majú aj informáciu o cene bunky, ktorá udáva dĺžku trasy.

Pre hlavný panel som zvolil triedu `AlgClassMainPanel`, ktorej sa v konštruktore predá instancia algoritmu a sama ovláda tvorbu vlákien, v ktorých sa algoritmus simuluje. Samotná simulácia na rovnakom princípe ako v ostatných algoritmoch a bola popísaná na začiatku tejto kapitoly.

5.2 Webová stránka

Aplikácie s algoritmami, ktoré som vytvoril sú prezentované na webovej stránke. Túto stránku som vytvoril, aby boli prístupné verejnosti. Stránka obsahuje rôzne algoritmy. Každý má teoretické informácie, popis ovládania a súbory na stiahnutie.

Dá sa stiahnuť spustiteľná jar aplikácia, alebo sa dá algoritmus spustiť z webu cez protokol JNLP. Spustenie aplikácie je popísané na webovej stránke. Užívateľ musí mať nainštalovanú Javu a ak chce spustiť aplikácie cez web, bude musieť pridať danú doménu do zoznamu výnimiek v Java Security Panel vo svojom systéme.

Táto stránka by mala slúžiť pre združenie vizualizačných algoritmov a v budúcnosti by sa mohly pridať nové algoritmy.

Stránka je prístupná na adrese http://www.stud.fit.vutbr.cz/~xrusna03/index.php. Popis ako stránku spustiť lokálne je v prílohe C.

Kapitola 6

Záver

V práci som navrhol knižnicu pre vizualizáciu algoritmov. Jej cieľom je uľahčiť tvorbu užívateľského rozhrania pre prezentáciu algoritmov. Vizualizácia je dobrý spôsob prezentácie nového algoritmu. Ak je prispôsobivá (zmena parametrov, konfiguračného priestoru...) a vizuálne pútavá, zaujme nielen svojím obsahom. Je to výhodou pre prezentovanie a učenie sa nových algoritmov pre väčšinu študentov.

Študenti môžu pomocou tejto knižnice rýchlo implementovať algoritmus spolu s užívateľským rozhraním. Snažil som sa aby bola ľahko použiteľná a prispôsobivá, takže by mala poskytovať prostriedky pre tvorbu rôznych druhov algoritmov. Obsahuje mnoho preddefinovaných prvkov, ktoré sa môžu s výhodou použiť a rozširovať.

Táto práca bola zameraná hlavne na algoritmy pre hľadanie cesty. Použitie knižnice som demonštroval na niekoľkých pravdepodobnostných a grafových algoritmoch. Využitie existujúcich elementov užívateľského rozhrania mi podstatne uľahčilo prácu a umožnilo vytvoriť aplikácie, ktoré zdieľajú rovnaké elementy a ovládanie.

Vytvorené aplikácie sú prístupné na webovej stránke, ktorú som tiež vytvoril v rámci tohoto projektu. Stránka združuje rôzne algoritmy a poskytuje ich užívateľom, ktorý sa chcú naučiť ako fungujú.

Sila tejto knižnice je najme v znovupoužiteľných prvkoch užívateľského rozhrania. Dala by sa rozšíriť o ďalšie užitočné zobrazenia, aj viac špecializované.

Vytvoril som základné všeobecné geometrické tvary a dátové štruktúry. Bolo by ale dobré, doplniť aj ďalšie, viac špecializované, alebo doplniť viac funkcionality tým existujúcim. Tvarom by sa dala pridať reakcia na udalosti myši, čím by sa zlepšila ich použiteľnosť. Vylepšil by som aj dizajnovú stránku aplikácie. Použil by som pre užívateľské rozhranie JavaFX technológiu, ktorá je modernejšia a rozšíriteľnejšia ako Swing.

Knižnica by sa dala použiť napríklad v predmetoch, kde majú študenti implementovať nejaký algoritmus spolu s užívateľským rozhraním. Ušetrila by čas a prácu s tvorbou užívateľského rozhrania a najlepšie projekty by sa dali zverejniť pre ďalších študenov a pomôcť im pri štúdiu týchto algoritmov.

Literatúra

[1] Choset, H. *Principles of Robot Motion: Theory, Algorithms, and Implementation.* Prentice Hall of India, 2005. ISBN 9780262033275.

[2] Correll, N. : Introduction to Robotics. [Online; navštíveno 1.1.2017]. Dostupné z: <http://correll.cs.colorado.edu/?p=965>

[3] Foundation, E. : Eclipse. [Online; navštíveno 1.1.2017]. Dostupné z: <https://eclipse.org/>

[4] Foundation, T. A. S. : Apache Maven. [Online; navštíveno 1.1.2017]. Dostupné z: <https://maven.apache.org/>

[5] Foundation, T. O. S. G. : GeoTools. [Online; navštíveno 1.1.2017]. Dostupné z: <http://geotools.org/>

[6] Foundation, T. O. S. G. : JTS Topology Suite. [Online; navštíveno 1.1.2017]. Dostupné z: <http://docs.geotools.org/latest/userguide/library/jts/index.html>

[7] Friends, A. : XAMPP. [Online; navštíveno 1.1.2017]. Dostupné z: <https://www.apachefriends.org/index.html>

[8] Galles, D. : Visualizing Algorithms. [Online; navštíveno 1.1.2017]. Dostupné z: <https://www.cs.usfca.edu/~galles/visualization/>

[9] Gandy, D. : Font Awesome. [Online; navštíveno 1.1.2017]. Dostupné z: <http://fontawesome.io/>

[10] Halim, S. : VisuAlgo. [Online; navštíveno 1.1.2017]. Dostupné z: <https://visualgo.net/en>

[11] JetBrains : IntelliJ IDEA. [Online; navštíveno 1.1.2017]. Dostupné z: <https://www.jetbrains.com/idea/>

[12] Kvasnica, M. : *Vizualizace hledání cesty pro robota.* bakalářská práce, Brno, 2008.

[13] Lab, K. : Probabilistic Roadmap Method (PRM). [Online; navštíveno 1.1.2017]. Dostupné z: <http://www.kavrakilab.org/robotics/prm.html>

[14] LaValle, S. M. *Planning Algorithms.* Cambridge, U.K.: Cambridge University Press, 2006.

[15] Naveh, B.; Contributors : JGraphT. [Online; navštíveno 1.1.2017]. Dostupné z: <http://jgrapht.org/>

[16] Oracle : Java Network Launch Protocol. [Online; navštíveno 1.1.2017]. Dostupné z: <https://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnlp.html>

[17] Patel, A. : Introduction to A*. [Online; navštíveno 1.1.2017]. Dostupné z: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

[18] Thrun, S.; Burgard, W.; Fox, D. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN 0262201623.

[19] Tutorials, T. J. : How to Set the Look and Feel. [Online; navštíveno 1.1.2017]. Dostupné z: <https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html>

[20] Wikipedia : Dijkstra's algorithm. [Online; navštíveno 1.1.2017]. Dostupné z: <https://en.wikipedia.org/wiki/Dijkstra's_algorithm>

[21] Wikipedia : Model–view–controller. [Online; navštíveno 1.1.2017]. Dostupné z: <https://en.wikipedia.org/wiki/Model\T1\textendashview\T1\textendashcontroller>

[22] Wikipedia : Motion planning. [Online; navštíveno 1.1.2017]. Dostupné z: <https://en.wikipedia.org/wiki/Motion_planning>

Prílohy

Príloha A

Obsah CD

- `text prace` - obsahuje pdf s textom práce a latex zdrojové súbory
- `vizlib` - obsahuje knižnicu navrhnutú v tejto práci. V podriečinku `src` sú zdrojové súbory, v `doc` sú UML návrhové diagramy pre program Visio od Microsoftu, v zložke `target` je skompilovaná java knižnica `vizlib` spolu s dokumentáciou a zdrojovými súbormy.
- `cz` - obsahuje knižnicu `vizlib`. Priečinok je určený pre skopírovanie do lokálneho repozitára `Mavenu` (`C:/Users/user/.m2`).
- `Template` - šablóna pre vytvorenie nového projektu s použitím knižnice `vizlib`.
- `astar`, `BFS`, `dijkstra`, `EST`, `PRM`, `RRT`, `VisibilityGraph` - algoritmy a ich aplikácie, ktoré som v rámci tejto práce vytvoril.
- `www` - webová stránka s algoritmami. Potrebuje php server.

Príloha B

Vytvorenie projektu s knižnicou pre vizualizáciu algoritmov

Pred vytvorením projektu popíšem štruktúru a nástroje, ktoré projekt používa.

Pripravil som šablónu pre nový projekt (je na CD v priečinku `Template`). Projekt používa `Maven` [4], takže si stiahne potrebné knižnice z centrálneho repozitára. Knižnica, ktorú som vytvoril, nie je v tomto repozitári a nutné ju pridať do lokálneho repozitára (popísané v postupe).

Šablóna projektu vytvára okrem spustiteľnej aplikácie aj webovu aplikáciu Java Web Start. V priečinku `Template/src/main/jnlp` je súbor `template.vm`, ktorý slúži ako šablóna pre vytvorenie súboru na spustenie aplikácie z webu. O vytvorenie potrebných súborov sa stará `Maven`. Príkazom `package` vytvorí v zložke `Template/target/jnlp` jar súbory s aplikáciou a `launch.jnlp`.

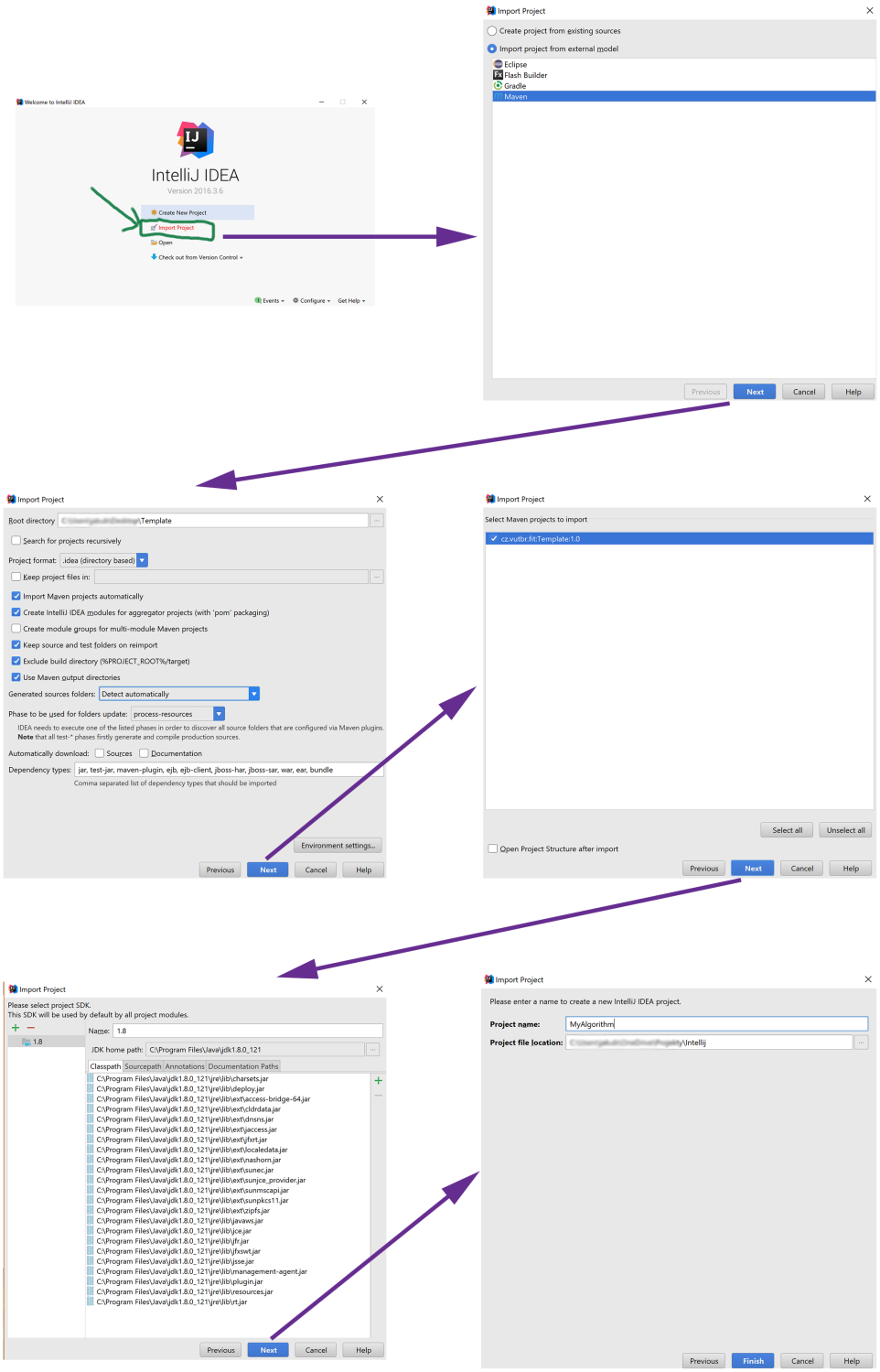
Spustenie Web Start aplikácie sa robí tak, že si užívateľ stiahne zo stránky súbor `launch.jnlp`. Tento súbor spustí JVM, ktorá načíta vzdialene aplikáciu. Po otvorení `launch.jnlp` v textovom editore má element `jnlp` atribút `codebase="http://home.page"`, ktorý predstavuje cestu k spustiteľnej aplikácii na webe. Hodnota, ktorá sa vloží do tohoto atribútu sa nastavuje v `pom.xml` v elemente `<project.homepage>http://home.page</project.homepage>` (doménové meno servera) a `<project.codebase>${project.homepage}</project.codebase>` (relatívna cesta k súboru v rámci servera. Aplikácie, ktoré som vytvoril v rámci tejto práce obsahujú príklady použitia.

Java z bezpečnostných dôvodov nedovoľuje spúšťať nepodpísané aplikácie. Preto som vytvoril súbor pre podpisovanie aplikácií `Template/dipKeystore`. V `pom.xml` som použil `maven-jarsigner-plugin`, ktorý tento súbor používa a podpisuje vytvorenú aplikáciu.

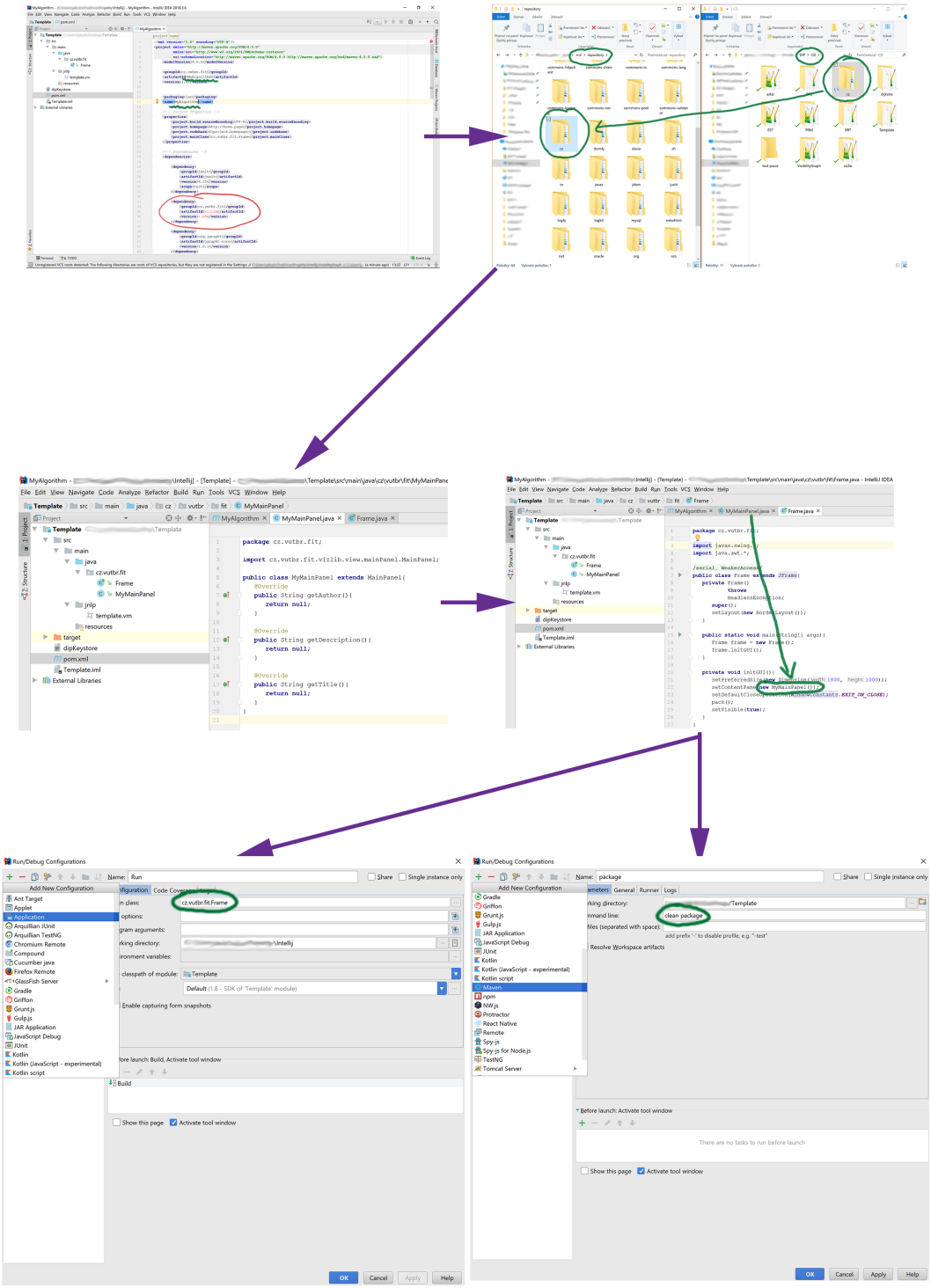
Vytvorenie Web Start Aplikácie je plne automatické a ak užívateľ nepotrebuje nič prispôsobiť, tak nemusí nič nastavovať.

B.1 Vytvorenie projektu v IntelliJ

1. Nakopírujte šablónu `Template` z CD do pracovného adresára IntelliJ.
2. Pri spustení IntelliJ vyberte voľbu `Import Project` (obrázok B.1 prvá obrazovka).
3. Zvoľte `Import project from external model` a položku `Maven` (obrázok B.1 druhá obrazovka).
4. Vyberte `Template` ako `Root directory` (obrázok B.1 obrazovka 3).
5. Importujte nájdený projekt (obrázok B.1 obrazovka 4).
6. Zvoľte požadovanú SDK (obrázok B.1 obrazovka 5).
7. Pomenujte projekt (obrázok B.1 obrazovka 6).
8. Po vytvorení projektu sa zobrazí chyba (obrázok B.2 obrazovka 1), že systém nevedel nájsť knižnicu pre vizualizáciu algoritmov `vizlib` (pretože nie je v žiadnom repozitári). Vlastné knižnice sa musia nainštalovať do lokálneho repozitára `Maven`u, ktorý je zvyčajne v `C:/Users/user/.m2` priečinku. Inštalácia prebieha príkazom `mvn install` (zdroj https://maven.apache.org/guides/mini/guide-3rd-party-jars-local.html). Alternatívou je nakopírovanie potrebných súborov ako ukazuje obrázok B.2 obrazovka 2. Priečinok `cz`, ktorý obsahuje `vizlib` knižnicu sa nachádza na CD.
9. Vytvorte hlavný panel algoritmu, ktorý dedí od hlavného panela z knižnice. Najjednoduchší hlavný panel je na obrázku B.2 obrazovka 3 `MyMainPanel`.
10. `Template` obsahuje triedu pre rámec `JFrame`, do ktorého treba vložiť vytvorený hlavný panel ako ukazuje obrazovka 4 obrázku B.2.
11. Nastavenie spustenia projektu ako java aplikácia je znázornené na obrázku B.2 obrazovka 5. Po zvolení konfigurácie `Application` sa nastaví hlavná trieda na rámec `Frame`, ktorý obsahuje vytvorený hlavný panel.
12. `Maven` konfigurácia je zobrazená na obrázku B.2 obrazovka 6. Po zvolení konfigurácie `Maven` sa nastavia argumenty príkazovej riadky na `clean package`. Tieto príkazy najprv odstránia všetky preložené súbory a potom zabalia celú aplikáciu do jar archívov, vygenerujú javadoc a zdrojové súbory. Výsledná zabalená aplikácia sa nachádza v adresári projektu v priečinku `target`.



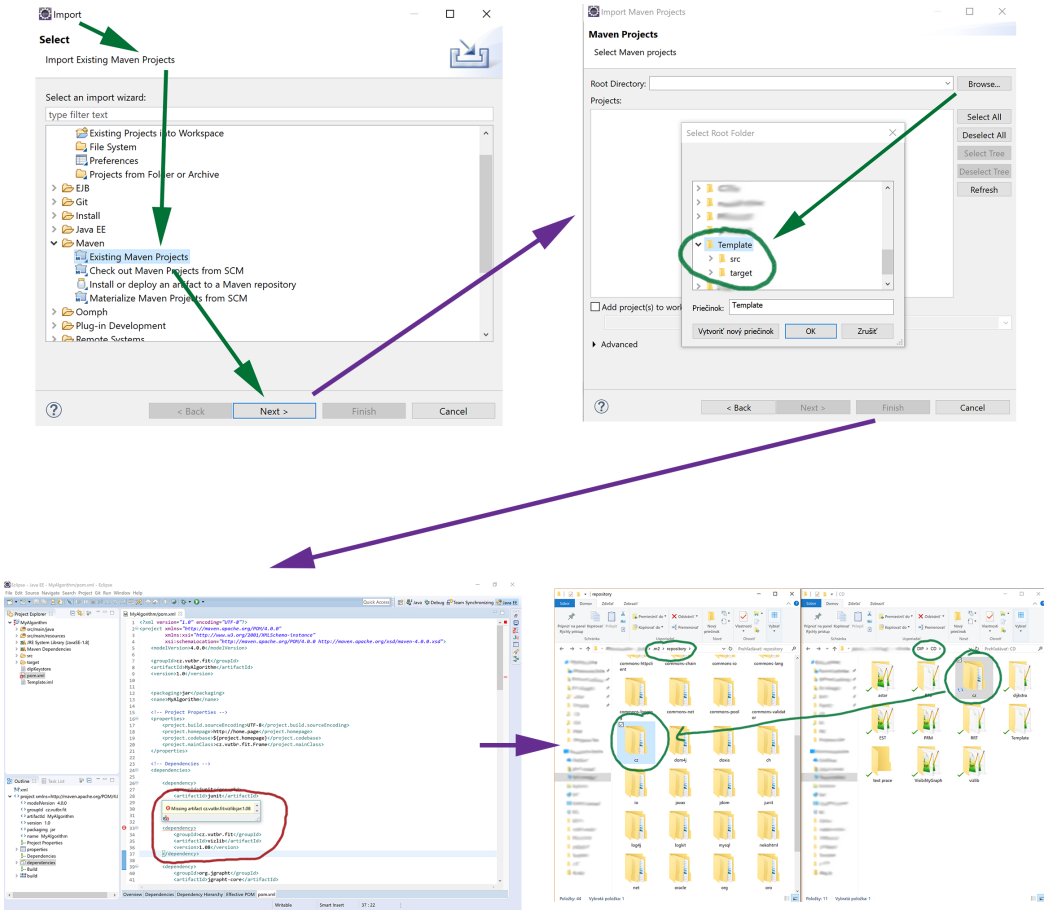
Obr. B.1: Vytvorenie nového projektu v IntelliJ.



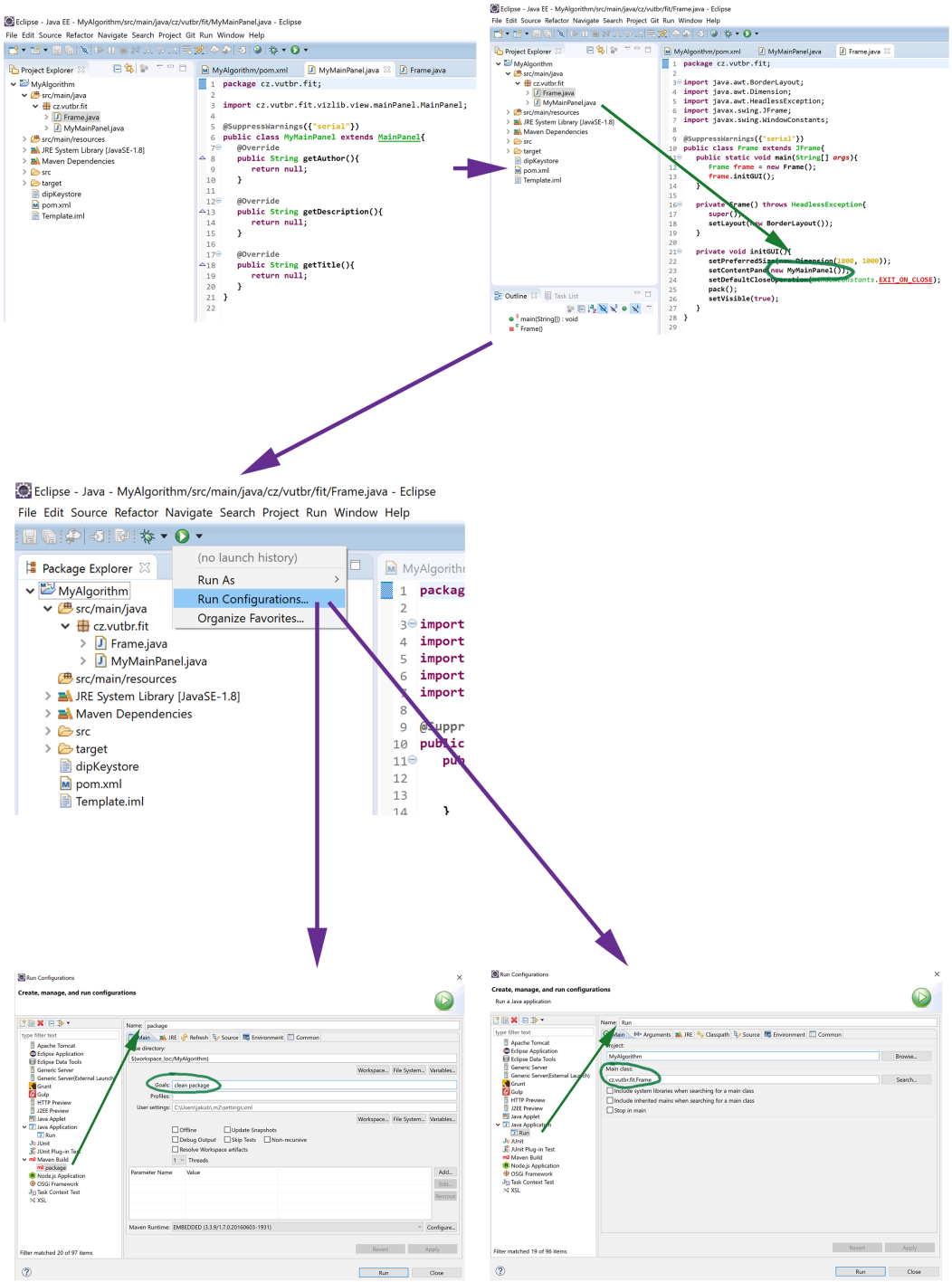
Obr. B.2: Vytvorenie spustiteľnej aplikácie v IntelliJ.

B.2 Vytvorenie projektu v Eclipse

1. Nakopírujte šablónu `Template` z CD do pracovného adresára Eclipse.
2. Po spustení Eclipse vyberte v menu voľbu `Import Project`.
3. Zvoľte `Maven` a položku `Existing Maven Projects` (obrázok B.3 obrazovka 1).
4. Vyberte `Template` ako `Root directory` (obrázok B.3 obrazovka 2).
5. Po vytvorení projektu sa zobrazí chyba (obrázok B.3 obrazovka 3), že systém nevedel nájsť knižnicu pre vizualizáciu algoritmov `vizlib` (pretože nie je v žiadnom repozitári). Vlastné knižnice sa musia nainštalovať do lokálneho repozitára `Maven`u, ktorý je zvyčajne v `C:/Users/user/.m2` priečinku. Inštalácia prebieha príkazom `mvn install` (zdroj https://maven.apache.org/guides/mini/guide-3rd-party-jars-local.html). Alternatívou je nakopírovanie potrebných súborov ako ukazuje obrázok B.3 obrazovka 4. Priečinok `cz`, ktorý obsahuje `vizlib` knižnicu sa nachádza na CD.
6. Vytvorte hlavný panel algoritmu, ktorý dedí od hlavného panela z knižnice. Najjednoduchší hlavný panel je na obrázku B.4 obrazovka 1 `MyMainPanel`.
7. `Template` obsahuje triedu pre rámec `JFrame`, do ktorého treba vložiť vytvorený hlavný panel ako ukazuje obrazovka 2 obrázku B.4.
8. Nastavenie spustenia projektu ako java aplikácia je znázornené na obrázku B.4 obrazovka 5. Po zvolení konfigurácie `Java Application` sa nastaví hlavná trieda na rámec `Frame`, ktorý obsahuje vytvorený hlavný panel.
9. `Maven` konfigurácia je zobrazená na obrázku B.4 obrazovka 4. Po zvolení konfigurácie `Maven Build` sa nastavia cieľe na `clean package`. Tieto príkazy najprv odstránia všetky preložené súbory a potom zabalia celú aplikáciu do jar archívov, vygenerujú javadoc a zdrojové súbory. Výsledná zabalená aplikácia sa nachádza v adresári projektu v priečinku `target`.



Obr. B.3: Vytvorenie nového projektu v Eclipse.



Obr. B.4: Vytvorenie spustiteľnej aplikácie v Eclipse.

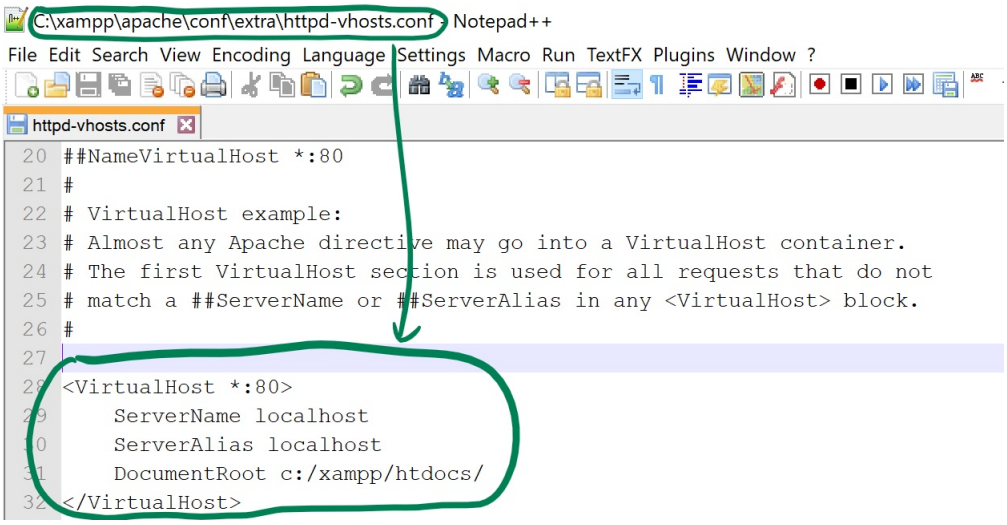
Príloha C

Spustenie webovej stránky

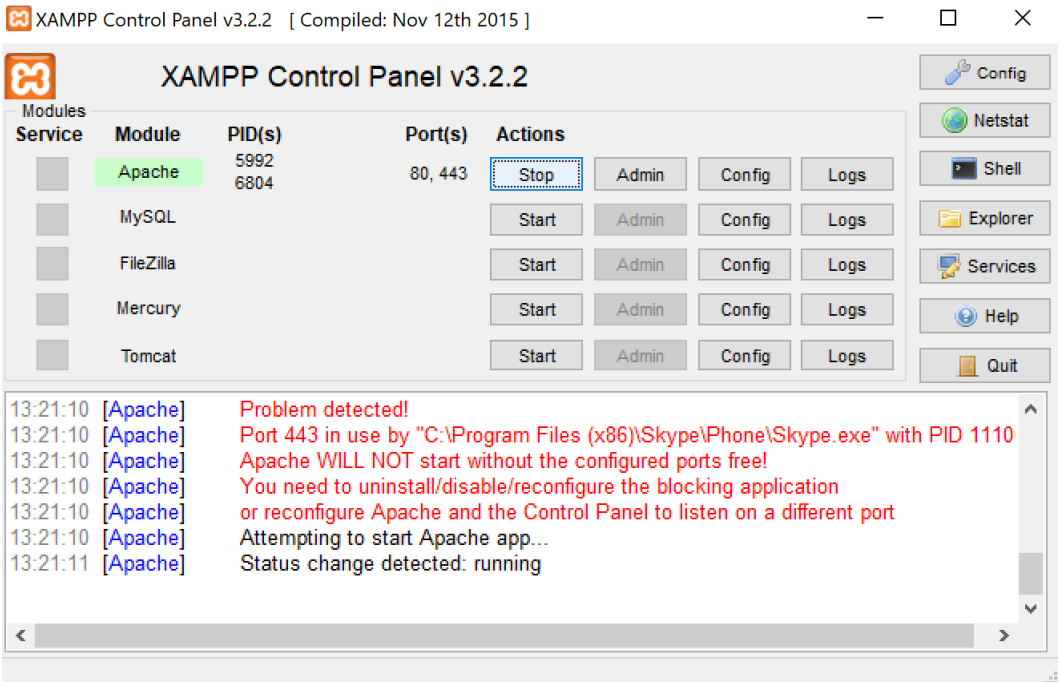
V priečinku `www` na CD je webová stránka s algoritmami, ktoré som vytvoril. Pre spustenie potrebuje webový server. Jednou z možností je použiť systém `XAMPP` [7]. Po jeho nainštalovaní sa vloží do `xampp/htdocs` priečinok `www` z CD.

Je možné, že bude treba nakonfigurovať XAMPP lokálny server. Konfigurácia je na obrázku C.1.

Spustenie servera je zobrazené na obrázku C.2. Stránka je prístupná na adrese `http://localhost/www/`.



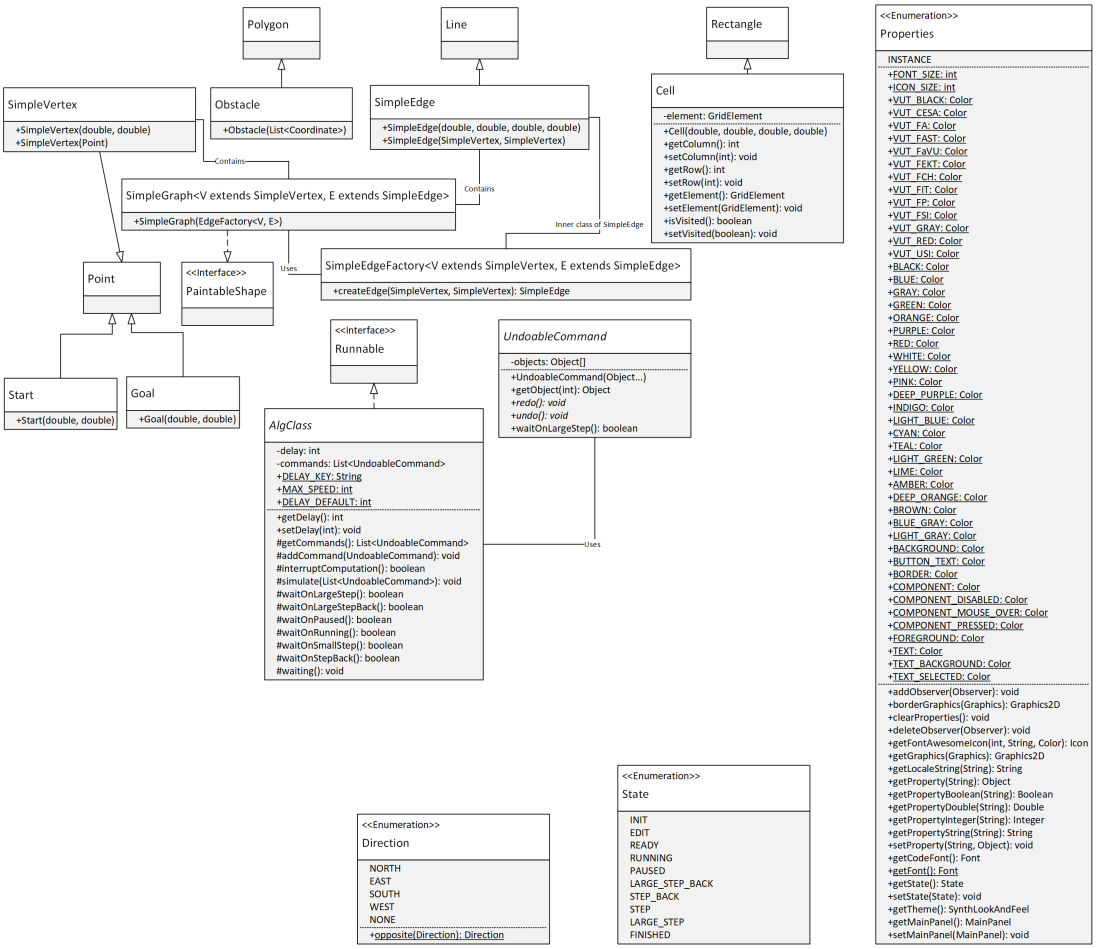
Obr. C.1: Konfigurácia XAMPP lokálneho servera.



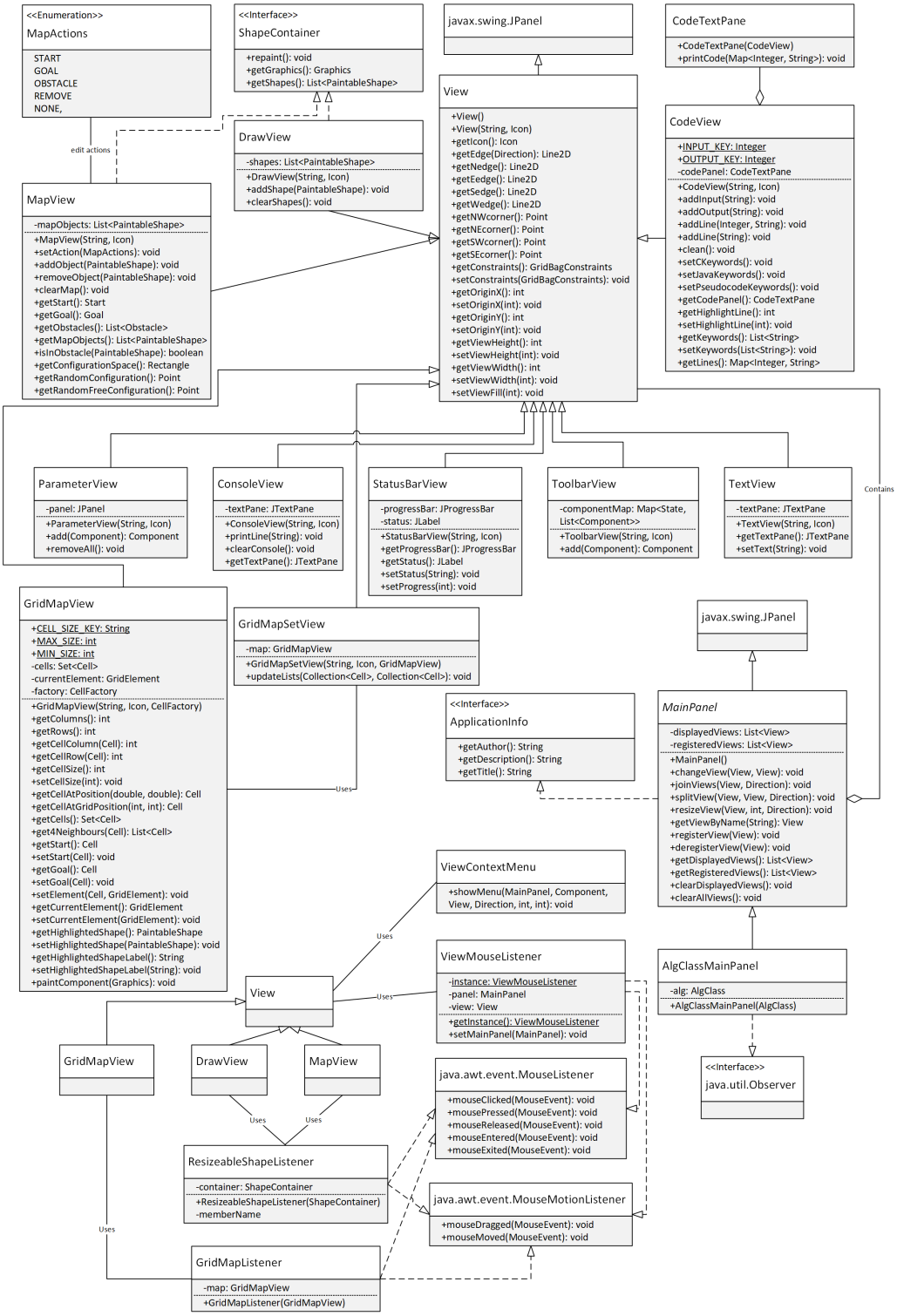
Obr. C.2: Spustenie XAMPP lokálneho servera.

Príloha D

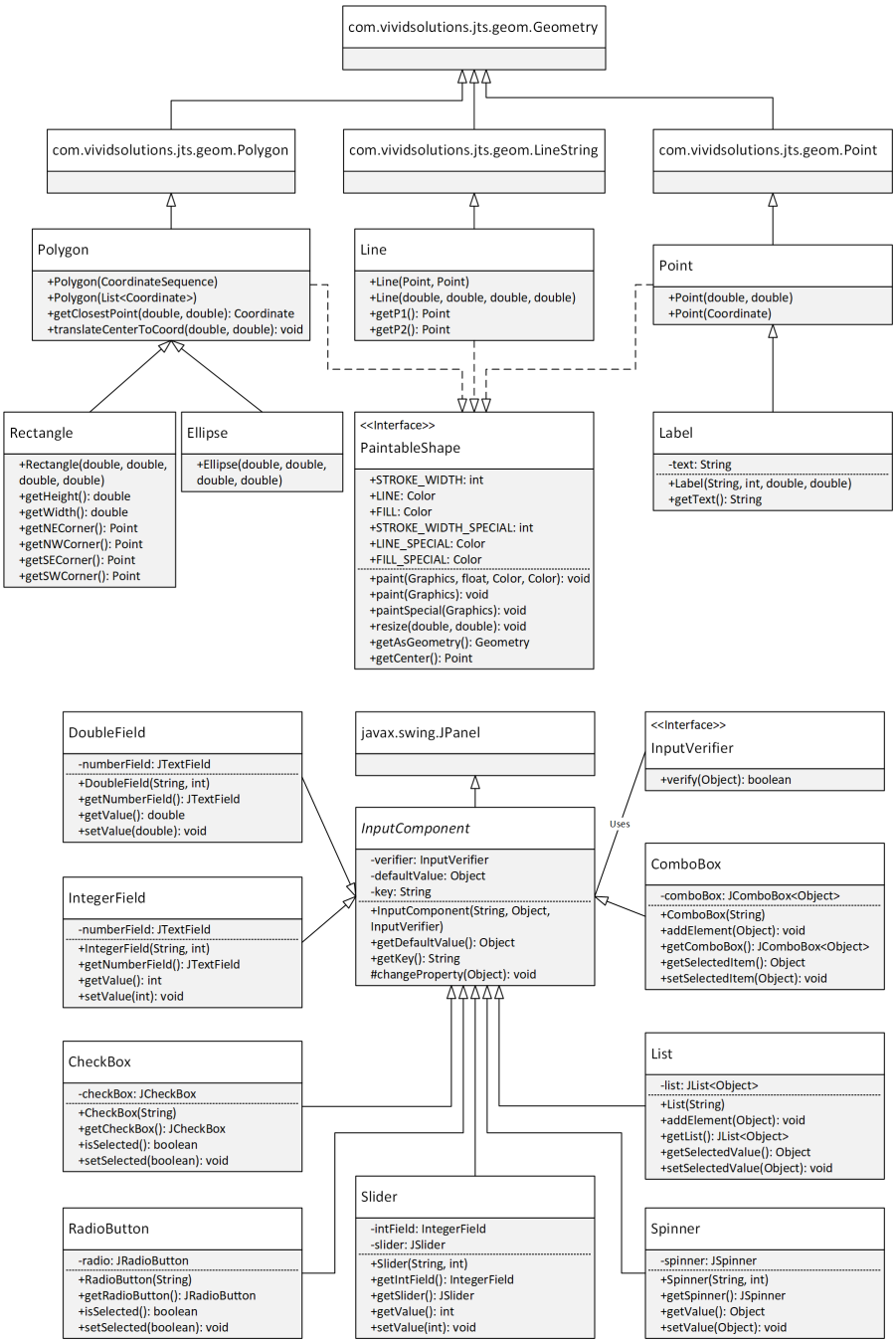
UML diagramy



Obr. D.1: Dátové štruktúry, algoritmus a globálne hodnoty.



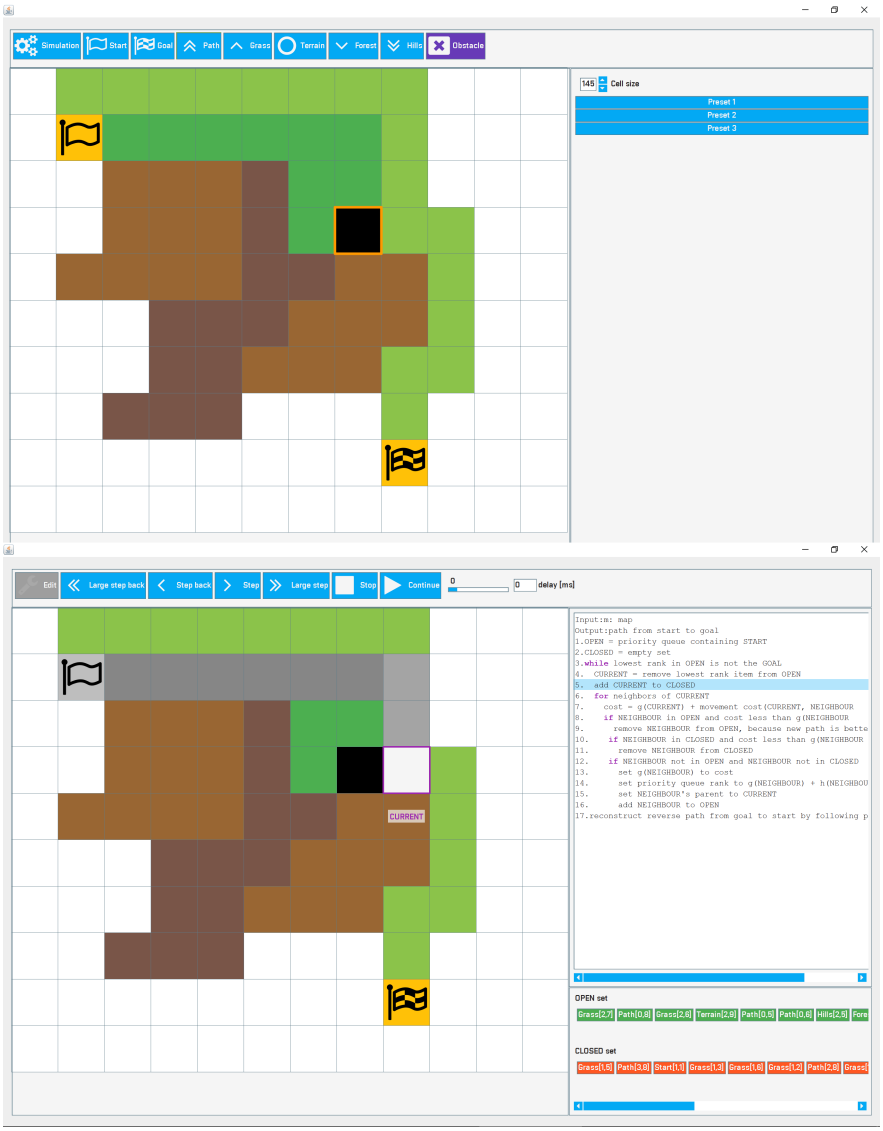
Obr. D.2: Zobrazenia a hlavný panel.



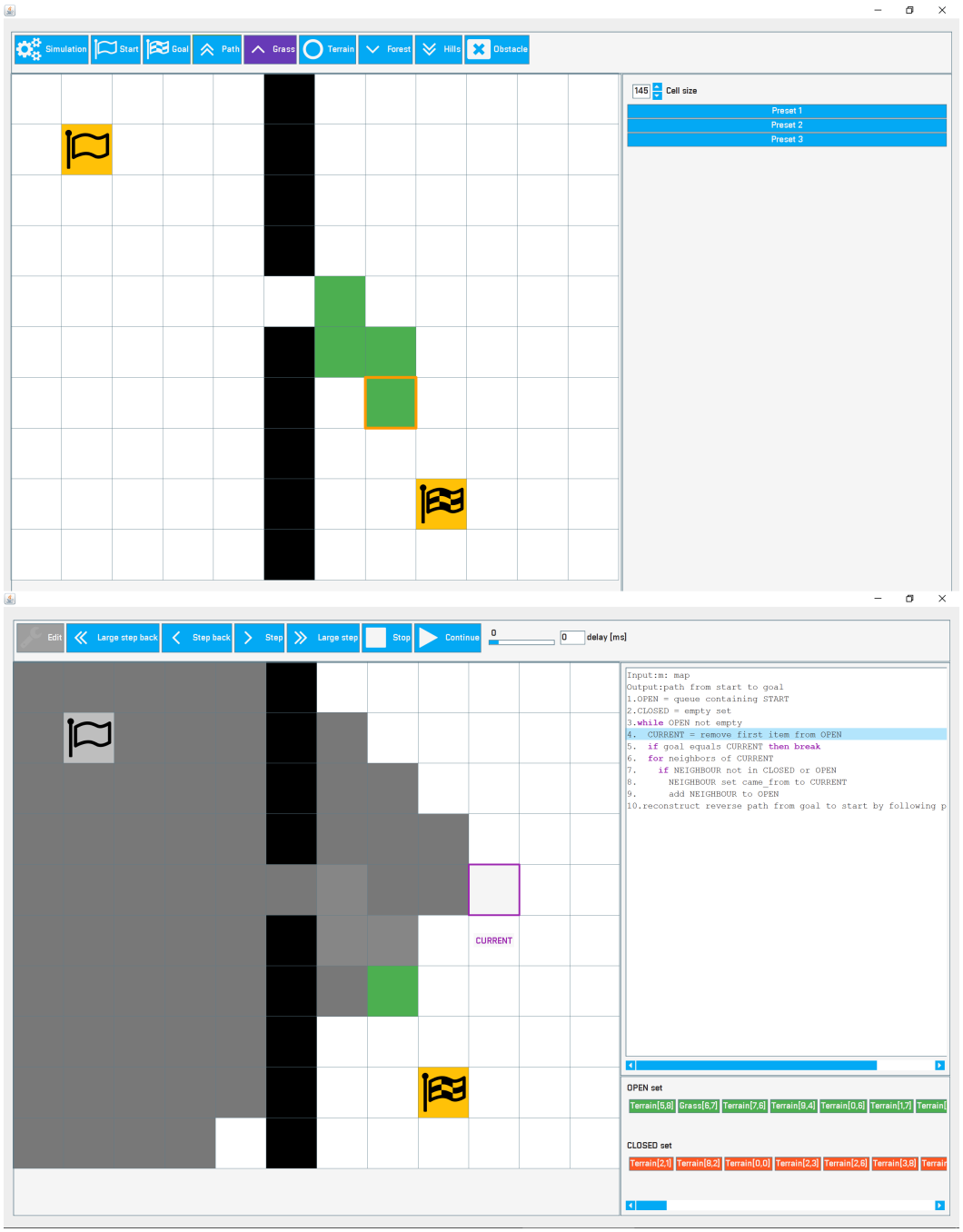
Obr. D.3: Geometrické tvary a vstupné komponenty.

Príloha E

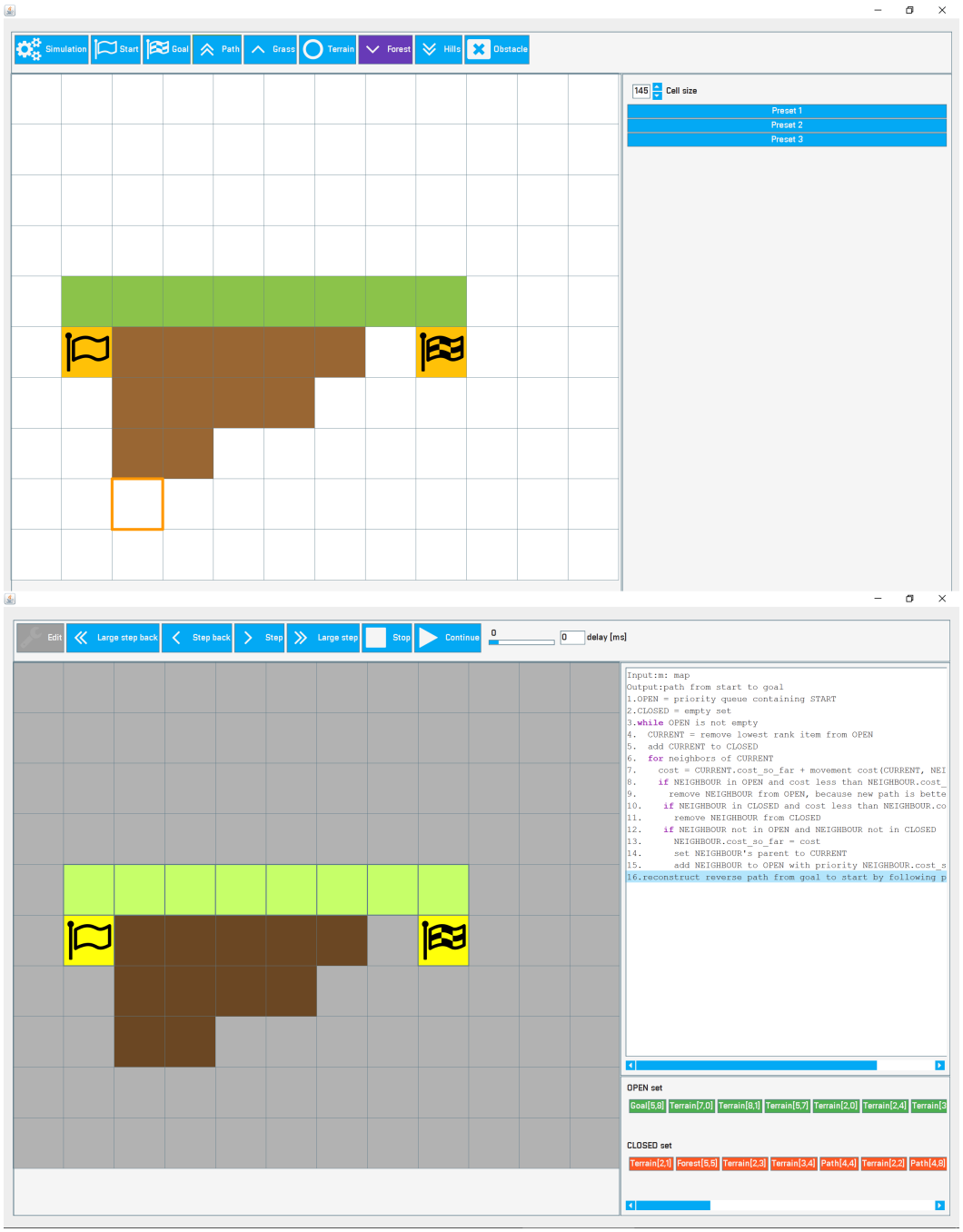
Ukážky aplikácií



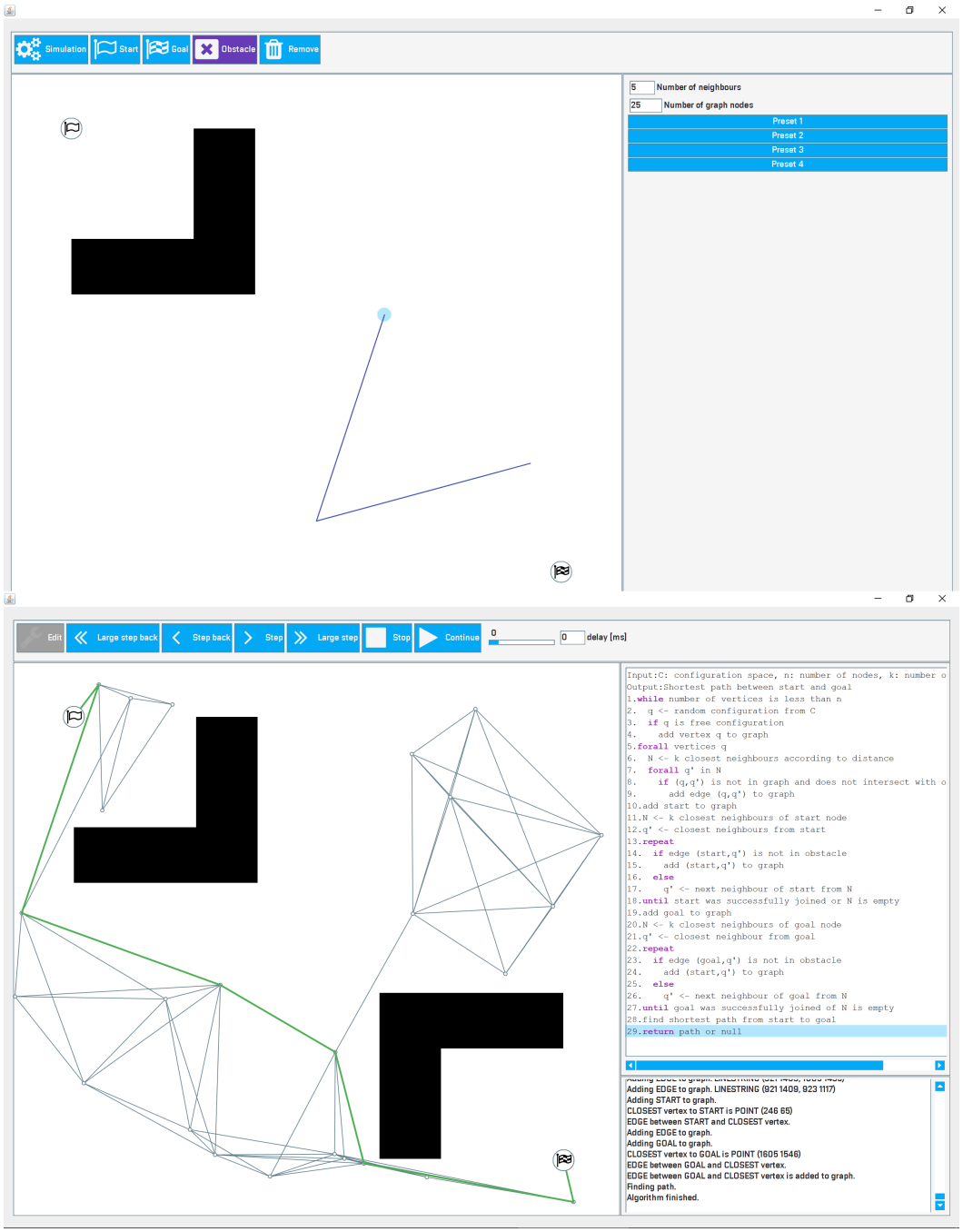
Obr. E.1: Applikácia algoritmu A*. Hore editovacia fáza, dole simulačná fáza.



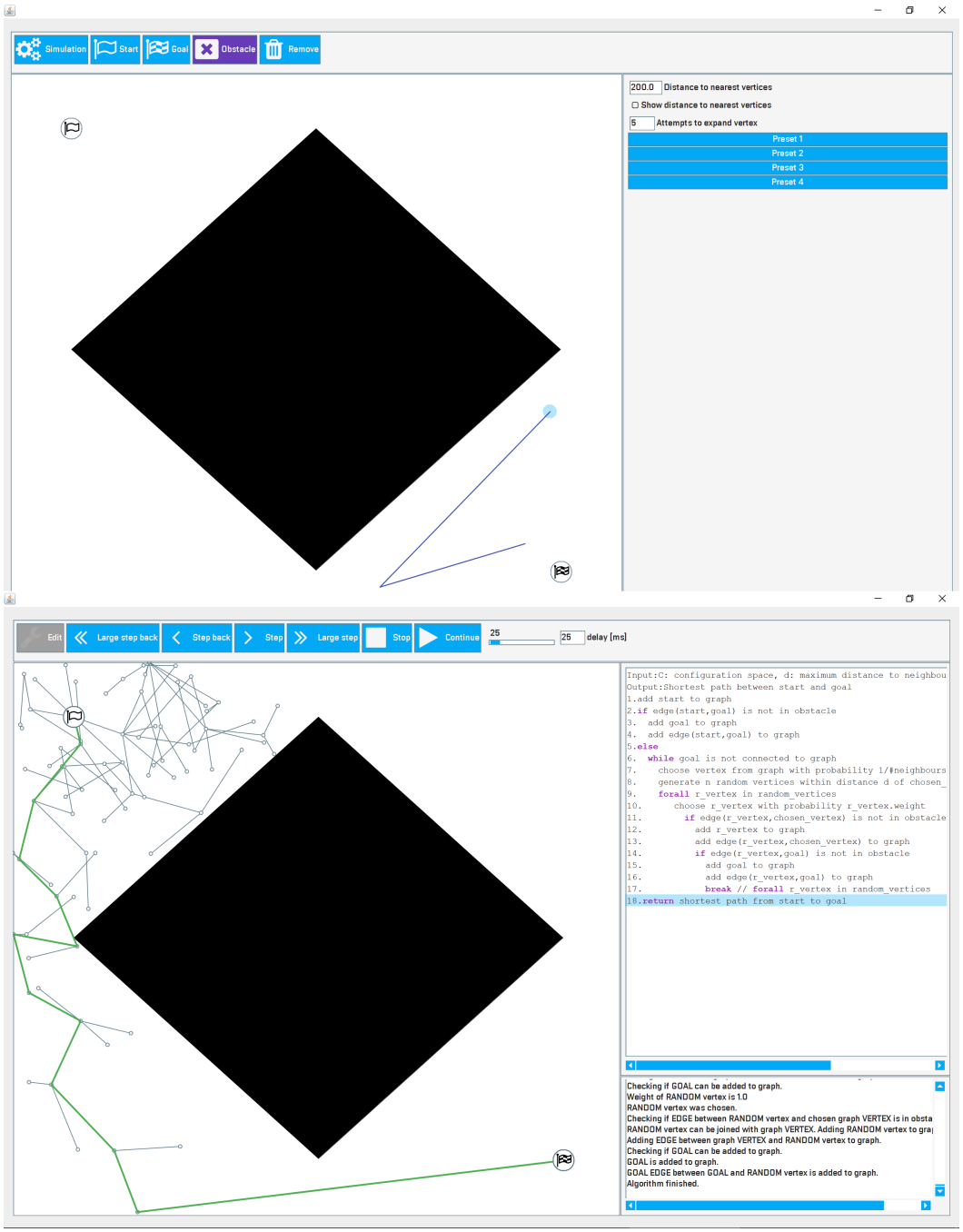
Obr. E.2: Applikácia algoritmu BFS. Hore editovacia fáza, dole simulačná fáza.



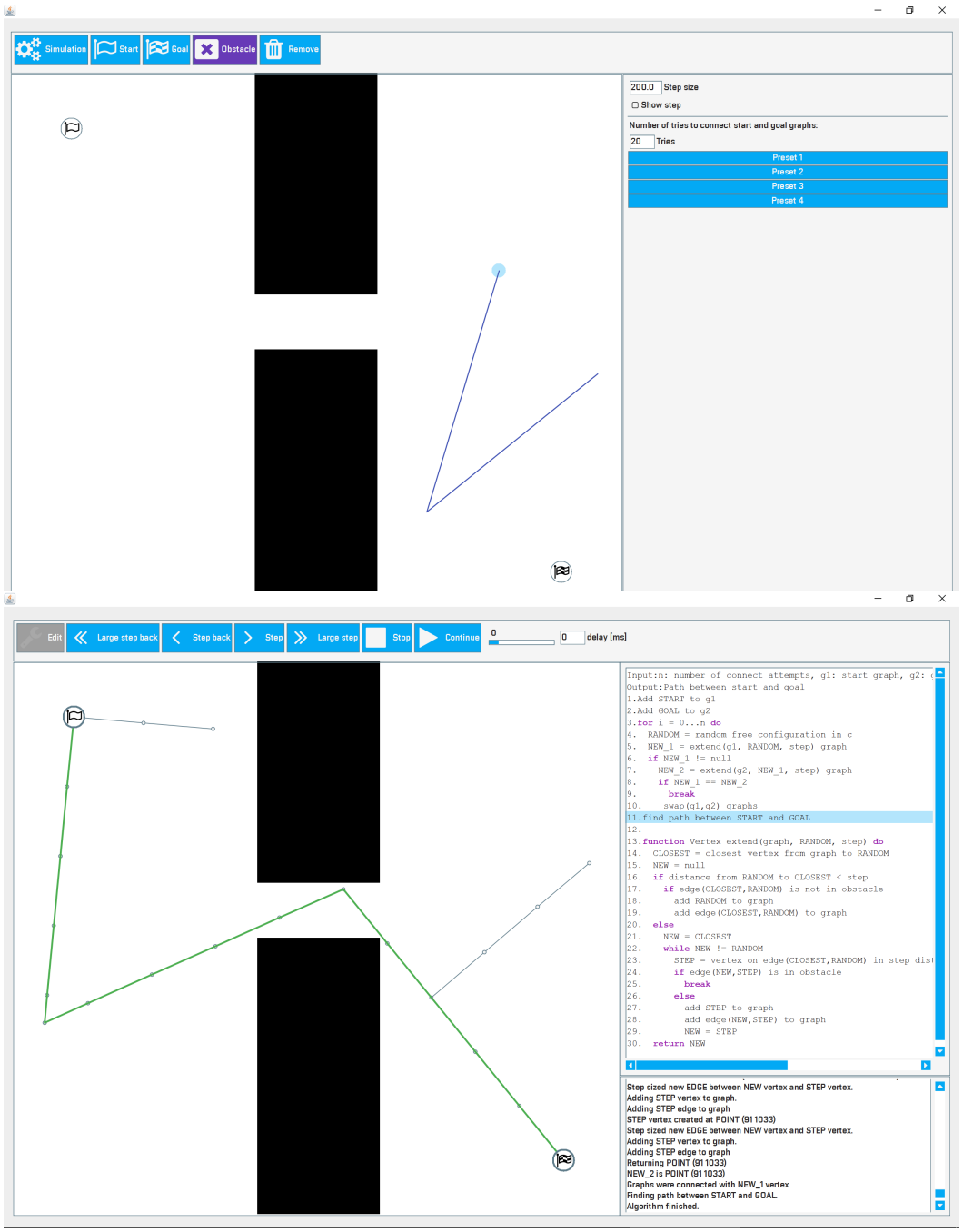
Obr. E.3: Applikácia Dijkstrovho algoritmu. Hore editovacia fáza, dole simulačná fáza.



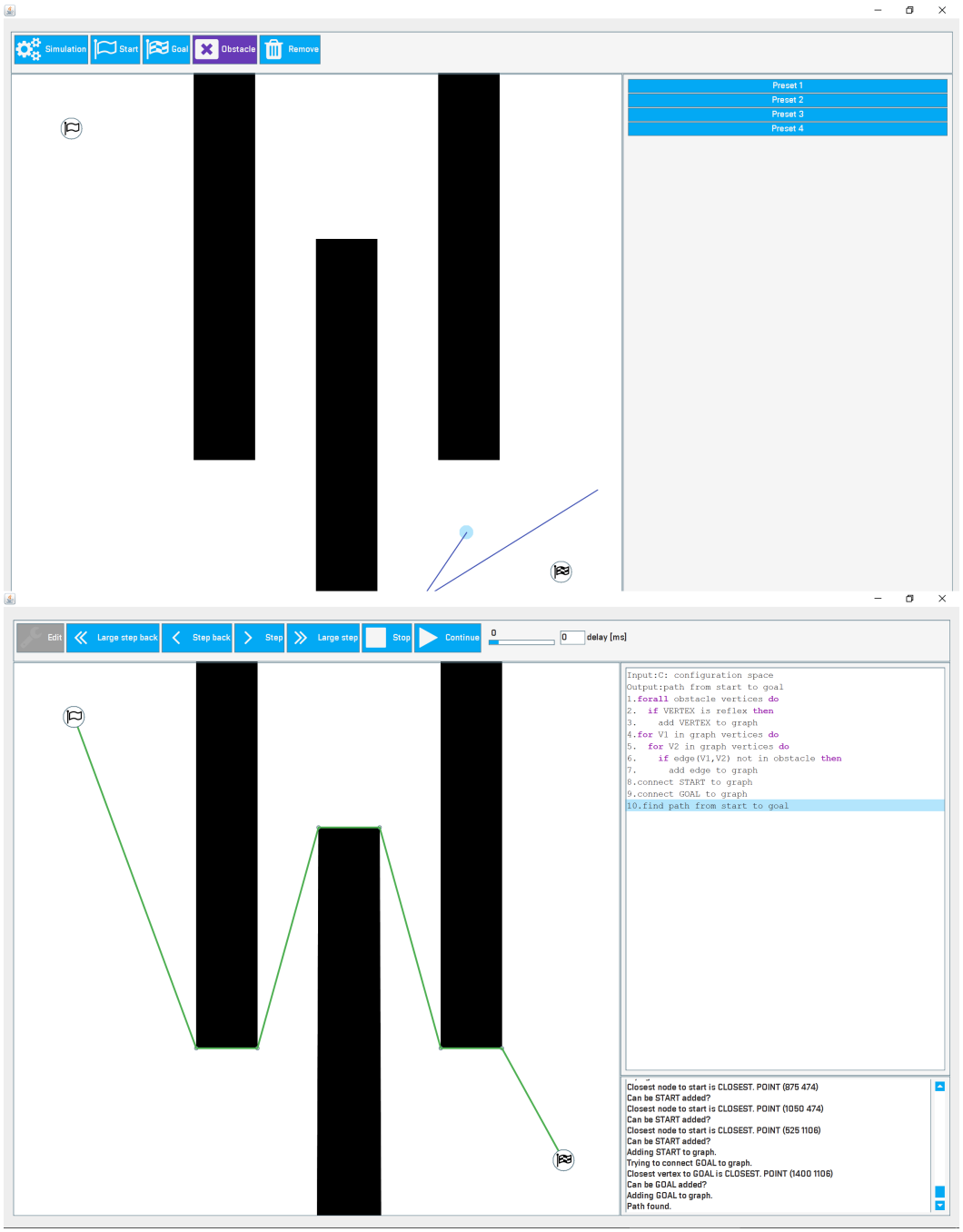
Obr. E.4: Applikácia algoritmu PRM. Hore editovacia fáza, dole simulačná fáza.



Obr. E.5: Applikácia algoritmu EST. Hore editovacia fáza, dole simulačná fáza.



Obr. E.6: Applikácia algoritmu RRT. Hore editovacia fáza, dole simulačná fáza.



Obr. E.7: Applikácia algoritmu grafu viditeľnosti. Hore editovacia fáza, dole simulačná fáza.