Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra matematiky

Diplomová práce

Algoritmy hledání cest pro městské prostředí

Plzeň 2015 Jakub Szkandera

Zadání

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne

Jakub Szkandera

Poděkování

Chtěl bych velmi poděkovat Prof. Dr. Ing. Ivaně Kolingerové, vedoucí této práce, za její užitečné rady a připomínky, cenný čas, který mi při konzultacích věnovala, a hlavně trpělivost, jenž měla se mnou i s touto prací. Poděkování patří také panu Bc. Ondřeji Kaasovi a panu Ing. Pavlu Brandejskému za jejich knihovny, které pomohly k rozšíření této práce. V neposlední řadě bych chtěl poděkovat své rodině, zejména své babičce Věře Krčkové, za podporu v průběhu celého studia.

Abstrakt

Tato práce pojednává o globální a lokální navigaci chodců v dynamicky se měnících virtuálních modelech měst. Tento problém je stále velmi aktuální, takže existuje mnoho metod, jenž ho řeší.

Cílem práce je zvolit vhodnou existující metodu a navrhnout vlastní řešení, které tuto metodu v některém směru vylepší. Toto řešení je nutné implementovat a vhodně otestovat.

V úvodu je popsána problematika hledání minimálních cest. Za ní následuje popis některých existujících metod. Popis řešení obsahuje zvolenou existující metodu a modifikaci našeho problému, jenž jsou následně důkladně otestovány.

Abstract

This thesis deals with global and local navigation of pedestrians in dynamicaly changing virtual city models. The problem is still very popular, so there are plenty methods that solve it.

The goal of this thesis is to choose appropriate existing method and propose own solution that makes the method better in one direction. This solutin has to be implemented and correctly tested.

In the beginning the problematics of minimal path planning is descibed. It's followed by the description of some existing methods. Description of solutin contains chosen existing method and it's modification, which are thoroughly tested.

Tato práce byla podporována MŠMT v rámci projektu Kontakt LH11006 Interaktivní geometrické modely pro simulaci přírodních jevů a davů.

OBSAH

1 Úvod **1**

2 Existující metody **3**

Definice 4
A* 6
LPA* 7
D* 9
Originální D* 10
Focused D* 11
D* Lite 12
Field D* 16

3 Řešení **19**

Struktura města 19
Vizualizace 22
Globální vyhledávací algoritmy 25
A* algoritmus 26
D* Lite algorimus 26
Známé prostředí 28
Částečně známé prostředí 29
Lokální vyhledávací algoritmy 30
Metoda lokálního maxima 32
Metoda lokálního minima 33
Skupinky chodců 35
Shlukování 35
Společná cesta 36
Kolize chodců 37

4 Experimenty **39**

Hledání cest v budovách 39
Načtení dat 41
Předzpracování 43
Vyhledávací algoritmy 45
Globální přepočet cesty 45
Lokální přepočet cesty 52
Srovnání 54
Skupinky chodců 56

Shlukování 56
Společná cesta 57
Kolize chodců 62

5 Závěr **64**

Reference **66**

A Obrázky **68**

B Vstupní soubory **73**

C Uživatelská dokumentace **74**

1 ÚVOD

Hledání cest hraje velmi důležitou roli v mnoha oblastech výzkumu a aplikace, jako jsou počítačové hry a virtuální prostředí, robotika nebo molekulární biologie. Ve své nejobecnější formě je úkolem dostat pohybující se objekt z počáteční pozice do pozice koncové v předem definovaném prostředí. Jako příklad můžeme zmínit libovolnou strategickou hru, např. Heroes of Might and Magic [23]. Zvolený hrdina se začne přesouvat automaticky na cílovou pozici (pozice označená uživatelem) v závislosti na virtuálním prostředí, tzn. vyhýbá se překážkám (hory, řeky, budovy, atd.) a preferuje pohyb po vyznačených stezkách před ostatním prostředím (bažina, zasněžené pláně, atd.).

Ačkoliv by se mohlo zdát, že je hledání cest v dnešní době vytěžená a nepříliš složitá záležitost, opak je pravdou. Zaprvé se stále nacházejí oblasti hledání cest, které se zkoumají a vylepšují své metody, a zadruhé i ve velkých projektech je hledání cest často velkým problémem. Do první skupiny můžeme zařadit např. pohyb robota prozkoumávajícího prostředí, kdy postupně objevuje svoje okolí a snaží přepočítat cestu na základě aktualizovaných dat. Druhým příkladem zkoumaných problémů jsou krizové situace uvnitř budov, kdy se simuluje např. únik lidí z hořící budovy. Do druhé skupiny můžeme zařadit např. hru League of Legends [19]. Je to v současnosti jedna z nejhranějších her, jejíž hráčskou komunitu tvoří desítky milionů lidí, a i přesto dokáže velmi často negativně překvapit ve vyhledávání cest.

V předešlé práci jsme se věnovali základním algoritmům pro statická prostředí, které jsme implementovali do programu EcoSim. Program EcoSim byl vyvinut na Purdue University (West Lafayette, Indiana, USA) výzkumnou skupinou počítačové grafiky pod vedením Doc. Bedřicha Beneše. EcoSim původně sloužil k simulaci růstu rostlin ve virtuálním městě [1]. Ing. Tomáš Vomáčka tento program převzal a upravil ho pro simulování pohybu chodců. Řešitel této diplomové práce do modifikované verze programu EcoSim přispěl v předchozích letech vyhledávacími algoritmy pro statická prostředí.

Mezi cíle této práce patří výběr jedné metody z existujících algoritmů pro hledání cest v časově proměnném prostředí. V této části se zaměřujeme na vyhledávání cest pro jedince. Tento přístup je sám o sobě velmi paměťově náročný, nemluvě o metodách specializovaných na dynamické prostředí. Ty jsou buď časově, anebo paměťově náročné, a proto je vhodné navrhnout vlastní metodu či upravit metodu stávající, která by dávala v alespoň jednom směru lepší výsledky (přesnější výpočet, časová či paměťová úspora). Důležité je navržené a existující metody mezi sebou porovnat a rozhodnout, která metoda je za daných podmínek lepší.

Dalším cílem této práce je urychlení výpočtu v případě většího počtu chodců. Pokud bude mít sto chodců totožnou cestu (stejný počátek i cíl cesty), pak nám

pro urychlení programu stačí tuto cestu spočítat pouze jednou a nikoliv stokrát. V případě, že těchto sto chodců bude mít podobnou cestu, spočteme jednu cestu, kterou použijeme pro všechny ostatní chodce. Tzn. všechny počáteční pozice budou v kružnici s poloměrem ϵ, analogicky pro cílové pozice. Tím dostaneme rychlejší výpočet, ovšem za cenu jisté nepřesnosti. Všechny tyto metody je nutné velmi dobře otestovat a porovnat je mezi sebou.

Úvod do problematiky hledání nejkratší cesty, definování důležitých pojmů a představení existujících metod pro dynamicky se měnící prostředí se věnuje kapitola 2. V kapitole 3 je popsán výběr použitých knihoven a navržené řešení použitých algoritmů. Experimentům navrženého řešení je věnována kapitola 4.

2 EXISTUJÍCÍ METODY

Základní problém plánování cesty se zabývá nalezením minimální cesty ve statickém známém prostředí (tzn. geometrie prostředí je známá a v průběhu výpočtu se nemění). Tento problém byl již algoritmicky uspokojivě vyřešen, a proto se zaměřujeme na zajímavější problém, kdy se prostředí může dynamicky měnit (např. vznik nové překážky nebo nové a levnější cesty). Toto prostředí může být předem známé, tedy vždy, když nastane změna hranového ohodnocení v grafu G, metoda na každou změnu patřičně zareaguje. Dalším případem může být vyhledávání v neznámém prostředí, kdy metoda reaguje na změnu hranového ohodnocení grafu G až v případě, že danou část grafu již prozkoumala.

V současnosti existuje mnoho metod pro hledání nejkratší cesty, a proto vybereme pouze ty, které vyhovují našemu problému. Hledáme tedy takové metody, které jsou schopny hledat cestu mezi dvěma vrcholy v dynamicky se měnícím prostředí, které může být známé, částečně nebo kompletně neznámé.

Zaprvé potřebujeme pouze metody, které lze použít pro výpočet v dynamickém prostředí. Tedy metody uzpůsobené na změny hranového ohodnocení v grafu G. Tím nám odpadají například metody BFS, DFS, Dijkstrův a Floyd-Warshallův algoritmus. Ačkoliv A* algoritmus slouží také pro výpočet nejkratší cesty ve statickém prostředí, popis tohoto algoritmu je v podkapitole 2. Zaprvé z jeho algoritmu vychází první nově navržený algoritmus D* (podkapitola 2) a zadruhé je možné použít naivní přístup a při každé grafové změně znovu spustit A* algoritmus pro přepočet cesty.

Dalším podstatným požadavkem jsou algoritmy vhodné pro hledání tras chodců jako jednotlivců. Každý chodec má v řešeném problému svou vlastní trasu nezávisle na ostatních, čímž odpadají algoritmy určené speciálně pro davové modelování chodců.

Mohli bychom taktéž požadovat metody, které řeší kolize chodce s okolím, nebo metody řešící kolize chodců s ostatními chodci. V této práci se ale zabýváme pouze grafovým vyhledáváním cest, nicméně využíváme metodu pana Ing. Pavla Brandejského, která řeší kolizi mezi chodci.

Ze všech algoritmů, které tyto požadavky splňují, jsme vybrali pouze heuristické algoritmy pro dynamicky se měnící prostředí. Ačkoliv se jedná o algoritmy primárně vyvíjené pro navigaci robota, zvolili jsme tento typ algoritmů, protože nejvíce vyhovuje našemu řešenému problému. Nezabýváme se tedy dalšími známými algoritmy, jako jsou algoritmy hledání cest na bázi Voronoi diagramů [3] nebo pravděpodobnostního plánování [3] či dokonce hledání cest pomocí Laplaceovy rovnice [4].

V podkapitole 2 je popsán algoritmus A*, který slouží pro hledání cesty v předem známém a statickém prostředí. Další popsanou metodou, která již slouží pro výpo-

čet cesty ve známém a dynamickém prostředí, je Lifelong Planning A* (kapitola 2). Velká pozornost je věnována skupině D* algoritmů (podkapitola 2), které byly vyvinuty primárně pro navigaci robotů.

Definice

V této podkapitole se nachází veškeré definice použitých pojmů jak z diskrétní matematiky [5], tak z ostatních vědeckých odvětví. Rozhodli jsme se dát všechny definice do jedné kapitoly, abychom jimi nenarušovali konzistenci textu. Zároveň bude vždy při prvním použití některého výrazu uveden odkaz na jeho definici.

Definice 2.1 (Graf)**.** Graf je uspořádaná dvojice $G = (V, E)$, kde V je konečná neprázdná množina a $E \subset \binom{V}{2} \cup V^2$.

Graf $G = (V, E)$ nazveme neorientovaným, pokud množina $E \subset \binom{V}{2}$, nebo orientovaným, pokud množina $E \subset V^2$.

Definice 2.2 (Sled)**.** Sled (z vrcholu s do vrcholu u) v grafu $G = (V, E)$ je libovolná posloupnost $(s = u_0, u_1, ..., u_k = u)$, kde u_i jsou vrcholy grafu G a pro každé $i = 1, ..., k$ je $u_{i-1}u_i$ hranou grafu G. Číslo $k + 1$ je délka tohoto sledu. Říkáme, že sled prochází vrcholy $u_0, ..., u_k$ nebo že na něm tyto vrcholy leží.

Definice 2.3 (Cesta)**.** Cesta z vrcholu s do vrcholu u v grafu $G = (V, E)$ je sled (Definice 2.2) vrcholů $(s = u_0, u_1, ..., u_k = u)$, ve kterém se každý vrchol u_i vyskytuje pouze jednou.

Definice 2.4 (Ohodnocený graf)**.** Ohodnocený orientovaný graf (G, w) je orientovaný graf spolu s reálnou funkcí $w : E(G) \to (0, \infty)$. Je-li e hrana grafu G, číslo $w(e)$ se nazývá ohodnocení nebo váha.

Definice 2.5 (Váha podgrafu)**.** Nechť M je množina hran ohodnoceného orientovaného grafu (G, w) (Definice 2.4). Váha $w(M)$ množiny M je součet vah jednotlivých hran $e \in M$. Pro stručnost definujeme váhu $w(P)$ cesty P jako váhu její množiny hran.

Definice 2.6 (Minimální cesta)**.** Jsou-li s, u vrcholy grafu G, pak minimální cesta z vrcholu s do vrcholu u je každá cesta, jejíž váha (Definice 2.5) je minimální (tj. žádná cesta z s do u nemá menší váhu). Vážená vzdálenost $d^w(s, u)$ vrcholů s, u je váha minimální cesty z s do u.

Pozastavme se a uvědomme si, že minimální cesta nemusí být nutně i cestou nejkratší. Spočtená minimální cesta je nejkratší cestou v případě, že hrany jsou ohodnoceny vzdálenostmi mezi jednotlivými vrcholy grafu G. Jakékoliv jiné ohodnocení hran se považuje pouze za minimální cestu.

Definice 2.7 (Souvislý graf)**.** Graf G je souvislý, pokud pro každé dva vrcholy s, u existuje v grafu G cesta z s do u. V opačném případě je graf G nesouvislý.

Definice 2.8 (Kružnice)**.** Kružnicí (cyklem) v grafu G rozumíme posloupnost vrcholů a hran $(s_0, e_1, s_1, e_2, ..., e_k, s_k = s_0)$, kde vrcholy $s_0, ..., s_{k-1}$ jsou navzájem různé vrcholy grafu G a pro každé $i = 1, 2, ..., k$ je $e_i = s_{i-1}, s_i \in E(G)$.

Definice 2.9 (Strom)**.** Souvislý graf (Definice 2.7), který neobsahuje jako podgraf žádnou kružnici (Definice 2.8), se nazývá strom.

Tímto máme připravenou půdu pro výrazy z diskrétní matematiky, ale stále nám chybí nadefinovat pár pojmů používané hlavně v oblasti informatiky a počítačové grafiky [8]. Poslední definice je pak z oblasti funkcionální analýzy [6].

Definice 2.10 (Triangulace)**.** Nechť $G = (V, E)$ je graf, kde V je množina vrcholů a E je množina navzájem se neprotínajících hran určených vrcholy V. Graf G se nazývá triangulace, pokud množina E je maximální.

Definice 2.11 (DT)**.** Delaunayho triangulace $DT(V)$ definovaná na množině bodů $V \in \mathbb{R}^2$ je množina trojúhelníků takových, že

- bod $p \in \mathbb{R}^2$ je vrchol trojúhelníků v $DT(V) \Leftrightarrow p \in V$,
- průsečík dvou trojúhelníků v $DT(V)$ je prázdný nebo se jedná o společnou hranu či vrchol,
- kružnice opsaná každému trojúhelníku z $DT(V)$ neobsahuje žádný bod množiny V.

Definice 2.12 (Viditelnost)**.** Dva vrcholy s a u v grafu $G = (V, E)$ jsou navzájem viditelné, pokud přímka mezi s a u neprotíná žádnou hranu z E.

Definice 2.13 (CDT)**.** Nechť $G = (V, E)$ je planární graf takový, že $E \neq 0$. Triangulace $\Delta = (V, E \cup E')$ (Definice 2.10) je Delaunayho triangulace (Definice 2.11) s omezením (tzv. CDT) grafu G, pokud všechny hrany $(s, u) \in E'$ jsou takové, že body s a u jsou vzájemně viditelné (Definice 2.12) v grafu G, a hrana (s, u) splňuje podmínku prázdné opsané kružnice vzhledem k vrcholům viditelným z s a u v grafu G.

Definice 2.14 (Prioritní fronta)**.** Prioritní fronta je dynamická množina umožnující efektivní realizaci operací vkládání libovolných nových prvků a jejich vybírání v pořadí jejich velikosti pomocí následujících operací:

- *vlož_prvek*
- *najdi_maximum*, resp *najdi_minimum*
- *smaž_maximum*, resp. *smaž_minimum*

Definice 2.15 (Metrika a metrický prostor)**.** Nechť X je neprázdná množina. Pokud $d : X \times X \to \mathbb{R}_0^+$ je zobrazení splňující $\forall x, y, z \in X$:

(i) $d(x, y) = 0 \Leftrightarrow x = y$,

(ii) $d(x, y) \leq d(x, z) + d(z, y)$,

(iii) $d(x, y) = d(y, x)$,

pak d nazýváme metrikou a dvojici (X, d) nazýváme metrickým prostorem.

A*

Vyhledávací algoritmus A* (čte se jako „*A star*“) se velmi hojně využívá pro nalezení minimální cesty (Definice 2.6) mezi dvěma různými vrcholy grafu G (Definice 2.1), jehož hranové ohodnocení se s časem nemění. Metoda je známá pro svůj výkon a přesnost a poprvé ji popsali Peter Hart, Nils Nilsson and Bertram Raphael v roce 1968 [9]. A* algoritmus dosahuje z časového hlediska lepších výsledků než Dijkstrův algoritmus z roku 1959, neboť se jedná o jeho rozšíření.

Jak A* algoritmus (Alg. 1) prochází graf, zaměřuje se na cestu s nejnižším známým hranovým ohodnocením (Alg. 1, řádek 4), zatímco alternativní části této cesty, objevené v průběhu výpočtu, si uchovává seřazené v prioritní frontě (Definice 2.14). Pokud má procházená cesta (Definice 2.3) kdykoliv v průběhu výpočtu vyšší cenu než jiná alternativní cesta v prioritní frontě (Alg. 1, *open_list*), metoda cestu s vyšší cenou opustí a vrátí se k alternativní cestě s nejnižší cenou. Tento proces se opakuje, dokud není dosaženo cíle.

A* používá heuristickou funkci $f(x)$ pro určení pořadí, ve kterém se budou uzly grafu G navštěvovat (Alg. 1, řádek 10). Tato heuristika je součtem dvou funkcí [3]. První funkcí je cena cesty $g(x)$ z počátečního uzlu $s_{start} \in G$ do současně navštíveného uzlu $s_{current} \in G$. Druhou funkcí je tzv. heuristický odhad $h(x)$ z libovolného uzlu $s \in G$ do cílového uzlu $s_{goal} \in G$. Navíc musí být přípustným heuristickým odhadem, tedy nesmí přecenit vzdálenost mezi libovolným uzlem $s \in G$ a cílovým uzlem $s_{goal} \in G$.

Pakliže bude heuristika pro všechny vrcholy $s, u \in G$, mezi kterými existuje hrana $(s, u) \in G$, splňovat dodatečnou podmínku $h(s) \leq d(s, u) + h(u)$, pak funkci $h(x)$ nazveme konzistentní [9]. Funkce $d(s, u)$ je v našem případě metrikou (definice 2.15). V takovém případě může být A* algoritmus naprogramován efektivněji. Přesněji lze metodu upravit tak, že žádný vrchol grafu G nebude zpracován více než jednou.

Časová náročnost A* algoritmu je závislá na použité heuristice $h(x)$. V nejhorším případě je exponenciální, tedy počet navštívených uzlů roste exponenciálně v závislosti na velikosti nalezeného řešení.

Algoritmus 1 A* algoritmus

Require: node *start*, node *end*

1: *open_list* ← *start* ▷ Vlož počáteční uzel do prioritní fronty
2: *closed_list* ← ∅ ▷ Inicializuj zpracované vrcholy
3: **while** *open_list* is not empty **do**
4: *current* ← *open_list.top*() ▷ Zvol vrchol s nejnižším ohodnocením
5: **if** *current* = *end* **then** ▷ Pokud jsme v cíli, sestroj cestu
6: reconstruct *path* ▷ Sestav nejkratší cestu
7: add *current* to *closed_list* from *open_list* ▷ *current* byl zpracován
8: **for** each neighbor *neigh* of *current* **do**
9: **if** *neigh* is not in *closed_list* **then** ▷ Pro nezpracovaný sousední vrchol
10: *temp* ← f(*neigh*) ▷ Spočti heuristickou funkci $f(x)$
11: **if** *neigh* is not in *open_list* **then**
12: *neigh.f* ← *temp* ▷ Nastav prioritu souseda na *temp*
13: add *neigh* to *path* and *open_list* ▷ Vlož souseda do prior. fronty
14: **else**
15: **if** *temp* < *neigh.f* **then** ▷ Nová heur. funkce je nižší než minule
16: *neigh.f* ← *temp* ▷ Aktualizuj cenu heuristické funkce
17: change *path* ▷ Pozměň cestu
18: reconstruct *path* ▷ Sestav nejkratší cestu

Předpokládejme, že hledáme cestu mezi počátečním uzlem $s_{start} \in G$ a jedním cílovým uzlem $s_{goal} \in G$, kde prohledávaný graf G je stromem (Definice 2.9). V takovém případě lze dosáhnout polynomiální složitosti, když bude navíc heuristická funkce splňovat podmínku

$$|h(x) - h^*(x)| = O(\log h^*(x)), \tag{2.1}$$

kde $h^*(x)$ je optimální heuristika. Tedy heuristika, která vrací přesnou hodnotu vzdálenosti mezi libovolným uzlem $x \in G$ a cílovým uzlem $s_{goal} \in G$. Jinými slovy vztah (2.1) říká, že chyba heuristiky $h(x)$ neporoste rychleji než logaritmus „dokonalé heuristiky“ $h^*(x)$, která vrací přesnou a minimální možnou vzdálenost z vrcholu $x \in G$ do vrcholu $s_{goal} \in G$.

LPA*

LPA* algoritmus, celým názvem Lifelong Planning A* [12], je inkrementální vyhledávací metoda s heuristikou pro opakované nalezení cesty mezi dvěma zadanými vrcholy grafu G, jehož ohodnocení hran se mění v čase. Inkrementální vyhledávání

spočívá v přepočítávání tzv. počátečních vzdáleností (tj. vzdálenost mezi počátečním vrcholem $s_{start} \in G$ a libovolným vrcholem $s \in G$), které se změnily nebo nebyly doposud spočteny. Heuristické vyhledávání slouží pouze pro přepočet počátečních vzdáleností, které jsou důležité pro přepočet minimální cesty z počátečního vrcholu $s_{start} \in G$ do vrcholu koncového $s_{goal} \in G$. Díky inkrementální heuristice algoritmus LPA* přepočítává velmi malé procento počátečních vzdáleností.

Lifelong Planning A* lze použít na hledání cesty v konečném a známém grafu G, jehož hranové ohodnocení s časem roste či klesá. Algoritmus je v tomto případě schopen vždy nalézt nejkratší cestu mezi počátečním vrcholem $s_{start} \in G$ a koncovým vrcholem $s_{goal} \in G$.

LPA* používá pro výpočet počáteční vzdálenosti dvě proměnné v každém vrcholu $s \in G$. První je tzv. g-hodnota $g(s)$ a druhá je tzv. rhs-hodnota $rhs(s)$. Rhs-hodnota vrcholu $s \in G$ je založena na g-hodnotách uzlů, se kterými sousedí. Tato hodnota vždy splňuje následující podmínku:

$$rhs(s) = \begin{cases} 0, & \text{pro } s = s_{start} \\ min_{s' \in neigh(s)}(g(s') + c(s', s)), & \text{jinak,} \end{cases} \tag{2.2}$$

kde funkce $c(s', s)$ udává hranové ohodnocení hrany, která vede mezi vrcholem s' a vrcholem s. Vrchol $s \in G$ se nazývá lokálně konzistentní, pokud platí $g(s) = rhs(s)$. V opačném případě se nazývá lokálně nekonzistentní. Pokud jsou všechny uzly $s \in G$ lokálně konzistentní, pak jejich g-hodnoty jsou rovny počátečním vzdálenostem. Vzhledem k tomu, že pro nalezení nejkratší cesty není nutné, aby byly všechny vrcholy grafu G lokálně konzistentní, zavádí se heuristika $h(s, s_{goal})$. Ta pomáhá algoritmu LPA* se zaměřit pouze na relevantní g-hodnoty, které jsou důležité pro nalezení nejkratší cesty mezi počátečním uzlem $s_{start} \in G$ a koncovým uzlem $s_{goal} \in G$. Na heuristiku jsou kladeny totožné požadavky jako na heuristiku používanou v A* algoritmu (viz předchozí kapitola 2).

Prioritní fronta Q obsahuje pouze vrcholy grafu G, které jsou lokálně nekonzistentní, a metoda se je snaží opravit. Nekonzistentní vrcholy $s \in Q$ v prioritní frontě řadíme podle speciálního klíče $k(s)$. Tento klíč není nic jiného než dvousložkový vektor ve tvaru:

$$k(s) = \begin{pmatrix} k_1(s) \\ k_2(s) \end{pmatrix} = \begin{pmatrix} \min(g(s), rhs(s)) + h(s, s_{goal}) \\ \min(g(s), rhs(s)) \end{pmatrix}$$

Prvky prioritní fronty Q jsou řazeny lexikograficky. To znamená, že nejprve porovnáváme první složky vektorů $k(s), k(u)$ pro $s, u \in Q, s \neq u$. V případě, že jsou první složky totožné, porovnáváme druhé složky vektorů $k(s), k(u)$ pro $s, u \in Q, s \neq u$.

Samotná metoda pak začíná nejprve inicializací problému hledání minimální cesty. Nejprve se g-hodnoty všech uzlů $s \in G$ nastaví na nekonečno a jejich rhs-hodnoty se nastaví podle vztahu 2.2. Pouze počáteční vrchol $s_{start} \in G$ se touto inicializací stane lokálně nekonzistentním vrcholem, a proto je vložen do zatím prázdné prioritní fronty.

Po inicializaci metoda spustí svoje hlavní jádro programu pro výpočet nejkratší cesty z počátečního uzlu $s_{start} \in G$ do cílového uzlu $s_{goal} \in G$. Tato část algoritmu LPA* při prvním spuštění nalezne cestu totožnou s cestou, kterou by našel algoritmus A*. Dokonce navštíví totožné vrcholy grafu G jako A* a navíc ve stejném pořadí, díky předešlé inicializaci.

Lokálně nekonzistentní uzel $s \in G$ nazveme přeceněným, pakliže $g(s) > rhs(s)$. Pokud metoda nalezne přeceněný uzel $s \in G$, přiřadí jeho g-hodnotě $g(s)$ jeho rhs-hodnotu $rhs(s)$ a tím i jeho počáteční vzdálenost, čímž se zároveň z přeceněného vrcholu $s \in G$ stane vrchol konzistentní. Do konce hlavního výpočtového jádra programu se už g-hodnota vrcholu $s \in G$ nemění.

Lokálně nekonzistentní uzel $s \in G$ nazveme podceněným, jestliže $g(s) < rhs(s)$. Pokud metoda začne zpracovávat takový uzel s, nastaví jeho g-hodnotu na $g(s) = \infty$. Tím se z vrcholu s stane uzel lokálně konzistentní nebo přeceněný. V případě přeceněnosti vrcholu může tato změna negativně ovlivnit sousední vrcholy, proto je nutné nejprve tyto prvky upravit tak, aby se staly lokálně konzistentními, než je vložíme nebo vyjmeme z prioritní fronty Q. Tento proces probíhá tak dlouho, dokud koncový uzel $s_{goal} \in G$ není taktéž lokálně konzistentní, pak výpočet končí. Cesta z počátečního vrcholu $s_{start} \in G$ do koncového vrcholu $s_{goal} \in G$ neexistuje, jestliže $g(s_{goal}) = \infty$.

Na závěr si uvědomme, že LPA* metoda se do velké míry shoduje s A* algoritmem. Rozdílem je, že LPA* rozšiřuje A* algoritmus pro dynamické prostředí.

D*

V současné době se pojmem D*, čte se jako „D star“, označuje hned několik algoritmů. Ačkoliv se tyto algoritmy vnitřně liší, jejich hlavní myšlenka je stejná, a proto se řadí do jedné skupiny. Všechny algoritmy v této kapitole jsou schopny řešit hledání cesty v dynamickém a známém, částečně známém či neznámém prostředí.

Do skupiny D* algoritmů patří primárně metody, které byly vyvinuty v robotice pro vyhledávání cesty pohybujícímu se robotovi. První takovou metodou byl originální D* algoritmus, který vycházel ze statického A* algoritmu. Následně byl tento algoritmus ještě upraven, vylepšen a pojmenován jako D* Focused. Ve velkém množství článků je naprostá většina metod porovnávána s algoritmem D* Lite, aby

ukázaly, že jsou v něčem lepší. Příkladem jedné takové metody může být Field D*, který se snaží nalézt lepší cestu než ostatní zmínění zástupci.

Všechny tři metody (originální D*, Focused D* a D* Lite) jsou metody, které řeší totožný problém. Tyto metody byly předně vymyšleny pro nalezení optimální cesty v dynamickém a neznámém prostředí z počátečního vrcholu s_{start} do koncového vrcholu s_{goal}, ale jde je použít i pro statické prostředí nebo prostředí známé či částečně neznámé. Po prvním nalezení cesty se robot začne pohybovat směrem k cíli a v případě, že dojde ke změně prostředí, jsou tyto algoritmy schopny se přizpůsobit a nalézt novou optimální cestu. Tento proces pokračuje tak dlouho, dokud robot nedosáhne koncového uzlu s_{goal}.

Ačkoliv je možné tyto metody použít i pro statické případy, je lepší použít metody specializované pro statická prostředí. Ušetříme tím velké množství paměti.

Originální D*

Původní D* algoritmus [14] představil Anthony Stentz v roce 1994. Jeho název, který se čte jako D star, pochází z termínu „dynamický A*“. Tento termín byl použit, protože se jedná o zobecnění statického algoritmu A* pro prostředí dynamicky se měnící (změna ohodnocení hran, vložení nového vrcholu atd.) [3].

Podobně jako A* algoritmus i D* algoritmus využívá prioritní frontu Q, do které se vkládají uzly, které potřebují být zpracovány nebo u nich došlo k nějaké změně. Každý vrchol $s \in G$ má několik stavů $\tau(s)$, podle kterých se rozhoduje, jak se daný vrchol $s \in Q$ zpracuje. Prvním stavem je $\tau(s) = NEW$. Význam tohoto stavu je, že daný vrchol doposud nikdy nebyl v prioritní frontě Q. Druhý stav $\tau(s) = OPEN$ nás informuje o tom, že se vrchol s právě nachází v prioritní frontě. Třetím stavem označme $\tau(s) = CLOSED$, který má význam již zpracovaného uzlu s. Takový uzel byl v prioritní frontě, byl zpracován a následně z prioritní fronty vyřazen.

Prvky prioritní fronty se řadí podle klíče $k_{min}(s)$, pro který platí $k_{min}(s) = \min_{\forall c(s,u)} c(s_{start}, s)$, kde funkce $c(s_{start}, s)$ vrací cenu cesty mezi počátečním vrcholem s_{start} a vrcholem s. Všimněme si, že klíč $k_{min}(s)$ je minimální hodnota ze všech možných cest. Klíč $k_{min}(s)$ pro porovnání prvků prioritní fronty Q je tedy roven ceně nejkratší cesty z počátečního uzlu s_{start} do uzlu s. Cesta do vrcholu s, jejíž cena je stejná jako hodnota $k_{min}(s)$, je nejkratší cestou, a tedy i cestou optimální.

Funkce pro výpočet klíče $k_{min}(s)$ klasifikuje každý uzel v prioritní frontě Q do dvou typů. První typ označíme jako $RAISE$, pokud $k_{old,min}(s) < \min_{\forall c(s,u)} c(s_{start}, s)$. Označíme tak uzly, jejichž k-hodnota byla v předchozím iteraci menší než v současné iteraci. Druhý typ označíme stavem $LOWER$, kam patří všechny vrcholy s, pro které platí $k_{old,min}(s) \geq \min c(s_{start}, s)$. Tedy cena cesty v současné iteraci je menší než

v iteraci předchozí. Tyto dva stavy $RAISE$ a $LOWER$ slouží k šíření informace o změně ceny cesty do sousedních uzlů.

Zaměřme se nyní na samotný algoritmus D*. D* iterativně zvolí uzel $s \in Q$ z prioritní fronty Q a vyhodnotí ho. Zvolený uzel s s nejmenším klíčem $k_{min}(s)$ je z prioritní fronty vyřazen. Pokud je jeho stav $LOWER$, máme optimální cenu do vrcholu s, protože starý $k_{old,min}(s)$ je roven novému $k_{min}(s)$. Následně zkontrolujeme všechny sousedy vyjmutého vrcholu s z fronty Q, abychom zjistili, zda nemůže být jejich cena cesty snížena.

Pokud nastane případ, že vyjmutý uzel z prioritní fronty je ve stavu $RAISE$, je možné, že cesta není nejkratší. Je tedy nutné nejprve zkontrolovat sousedy, do nichž vede optimální cesta, a přizpůsobit se jim. Tím zamezíme šíření chyby do dalších sousedních uzlů, které doposud nebyly zpracovány nebo na zpracování čekají. Dále se postupuje jako v případě stavu $LOWER$.

Focused D*

Focused D* algoritmus [15], jak může název napovídat, je rozšířením originálního D* algoritmu a v roce 1995 ho představil stejný autor jako D* algoritmus, Anthony Stentz. Na rozdíl od původní verze využívá heuristiku, aby se algoritmus zaměřil na nejdůležitější vrcholy. Tím se sníží celkový počet zpracovaných vrcholů grafu G a dojde k urychlení výpočtu nejkratší cesty.

Analogicky k původnímu D* algoritmu využívá i Focused D* prioritní frontu Q, do které jsou vkládány vrcholy určené ke zpracování. Všechny vrcholy grafu mohou nabývat stavů NEW, $OPEN$, $CLOSED$. Navíc lze prvkům v prioritní frontě přiřadit dva další stavy $LOWER$ a $RAISE$. Všechny tyto stavy byly vysvětleny v předchozím algoritmu (kap. 2) a v tomto algoritmu mají naprosto stejný účel.

Základní struktura algoritmu je totožná s původním D* algoritmem, a proto se zaměříme pouze na místo, kde se tyto dvě metody liší. Rozdílem je klíč, podle kterého se řadí prvky v prioritní frontě. Dříve stačilo řadit prvky podle jejich minimální ceny, za kterou je možné se do nich dostat. Nyní k seřazení prvků používáme klíč $f(s_{current}, s)$, který je součtem dvou funkcí

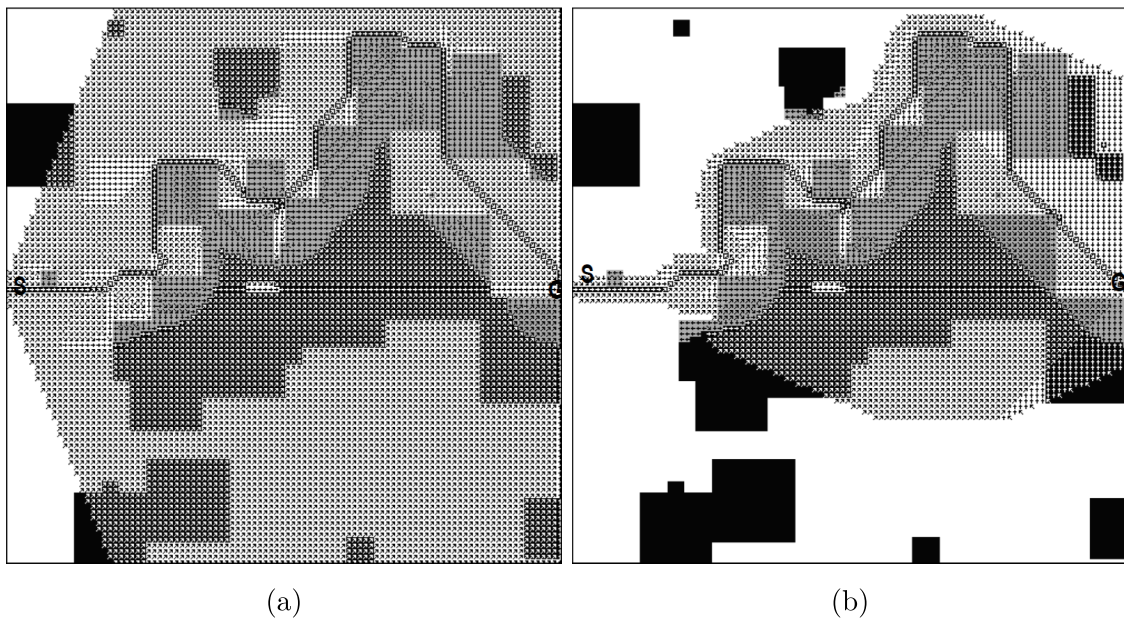
$$f(s_{current}, s) = c(s_{goal}, s) + h(s_{current}, s),$$

kde funkce $c(s_{goal}, s)$ je cena cesty z koncového vrcholu s_{goal} do vrcholu s a funkce $h(s_{current}, s)$ je heuristický odhad vzdálenosti mezi současnou pozicí robota $s_{current}$ a vrcholem s. Funkce $f(s_{current}, s)$ tedy není ničím jiným než odhadem vzdálenosti mezi současnou pozicí robota $s_{current}$ a cílovým uzlem s_{goal}.

Přidání heuristiky do D* algoritmu je hlavním rozdílem mezi původním D* algoritmem a Focused D* algoritmem. Novější verze D* algoritmu prochází mnohem menší počet uzlů, což je vidět na obrázku 2.1 (převzato z [15]).

Na obrázku 2.1 si můžeme všimnout bílých a černých oblastí. Bílá oblast reprezentuje prostupnou oblast, tedy oblast, kudy se může robot pohybovat. Naopak černá oblast reprezentuje neprostupné území, tedy např. skálu. Každá šipka pak reprezentuje jeden vrchol grafu G, který byl alespoň jednou vložen do prioritní fronty. Směr šipek nás informuje o tom, odkud jsme se do dané pozice dostali. Jedná se o problém nalezení cesty v neznámém prostředí mezi počátečním vrcholem s (levá strana obrázku 2.1 (a) i (b)) a koncovým vrcholem g (pravá strana obrázku 2.1 (a) i (b)). Z obrázku je pak patrné, že metoda Focused D* (Obr. 2.1 (b)) nalezne cestu rychleji, protože díky heuristické funkci musí navštívit mnohem menší počet vrcholů než původní D* algoritmus (Obr. 2.1 (a)).

Metoda Focused D* obsahuje ještě drobné rozdíly od původního D* algoritmu, které nejsou tolik podstatné a v případě hlubšího zájmu mohou být nalezeny v článku [15].



(a) (b)

Obr. 2.1: Cesta a počet zpracovaných uzlů (a) originálním D* algoritmem, (b) Focused D* algoritmem

D* Lite

V kapitole 2 byl popsán LPA* algoritmus, který opakovaně nachází nejkratší cestu mezi počátečním uzlem $s_{start} \in G$ a koncovým uzlem $s_{goal} \in G$ při každé grafové

změně (změna ohodnocení hran, odebrání vrcholu $s \in G$, atd.). Algoritmus D* Lite [11] vychází z metody LPA* podobně jako originální D* vychází z A* algoritmu. Rozdílem je, že D* Lite opakovaně nachází cestu ze současného vrcholu $s_{current} \in G$ do cílového vrcholu $s_{goal} \in G$ při každé změně grafu G, zatímco se robot pohybuje po nalezené cestě směrem k cíli $s_{goal} \in G$. Metoda nepotřebuje žádné předpoklady k rychlosti změn ohodnocení hran grafu G, jejich velikosti nebo jak daleko od současného uzlu $s_{current} \in G$ se změna objeví.

Pro problém navigace robota do cílové pozice v neznámém prostředí je možné použít LPA* algoritmus. Jak již bylo zmíněno, LPA* (kap. 2) vyhledává cestu mezi počátečním uzlem $s_{start} \in G$ a koncovým uzlem $s_{goal} \in G$. Jejich g-hodnoty jsou počítány jako vzdálenost od počátečního vrcholu s_{start}, nicméně pro D* Lite je potřeba změnit směr vyhledávání tak, aby se g-hodnoty vrcholů počítaly od koncového vrcholu $s_{goal} \in G$. V neorientovaném grafu stačí pouze prohodit počáteční uzel s koncovým. V orientovaném grafu je navíc nutné změnit směr orientovaných hran.

Na heuristiku jsou kladeny totožné nároky jako v případě A* a LPA* algoritmu. Požadujeme tedy nezápornost a zpětnou konzistenci heuristiky:

$$\begin{aligned} &h(s,u) \geq 0 \land h(s,u) = 0 \Leftrightarrow u = s, && \text{(nezápornost)} \\ &h(s_{start}, s) \leq d(s_{start}, s') + h(s', s), && \text{(zpětná konzistence)} \end{aligned} \tag{2.3}$$

kde funkce $d(s,u)$ je metrika, $h(s,u)$ je heuristika $\forall s, u \in G$ a s' je uzel sousedící s uzlem $s \in G$. Obecněji to znamená, že tím, jak se robot pohybuje směrem k cíli, se vrchol s_{start} mění, a proto musí heuristika splňovat vlastnosti (2.3) pro všechny $s_{start} \in G$.

Pokud je po prvním prohledání grafu G g-hodnota v počátečním uzlu $g(s_{start}) = \infty$, cesta z vrcholu s_{start} do vrcholu s_{goal} neexistuje. V opačném případě je možné sledovat nejkratší cestu z počátečního vrcholu s_{start} do cílového vrcholu s_{goal}. Cíle dosáhneme pohybem ze současného uzlu $s_{current}$ do sousedního uzlu s', který minimalizuje výraz $d(s_{current}, s') + g(s')$. Tento proces opakujeme tak dlouho, dokud nedojdeme do cíle s_{goal}.

K vyřešení problému dosažení cíle v neznámém prostředí je nutné upravit algoritmus LPA*, ačkoliv většina jeho funkcí a metod se nemění. Hlavní metoda musí být upravena, aby se vždy, kdy dojde k pohybu robota, správně přepočítaly veškeré prvky v prioritní frontě Q. Heuristika se s pohybem robota mění, protože je vždy počítána s ohledem na současnou pozici robota $s_{current}$, a proto je tato část velmi důležitá. Tato úprava změní pouze klíče všech uzlů $s \in Q$, ale nijak neovlivní lokální konzistenci uzlů a tím nezvětší (resp. nezmenší) velikost prioritní fronty Q.

Průběh metody je pak velmi podobný LPA* algoritmu. Nejprve se inicializuje problém hledání cesty. Všem vrcholům nejprve nastaví jejich g-hodnoty na nekonečno, tedy $g(s) := \infty \forall s \in G$, a rhs-hodnoty podle vztahu (2.2). Upozorněme,

že se jedná o zpětné vyhledávání, a proto se koncový uzel $s_{goal} \in G$ stane lokálně nekonzistentním uzlem (uzel s, pro který platí $g(s) \neq rhs(s)$) a je následně vložen do prázdné prioritní fronty Q.

D* Lite následně vypočte nejkratší cestu mezi současným uzlem $s_{start} \in G$ a koncovým uzlem $s_{goal} \in G$. Pokud robot není v cíli a změnila se jeho poloha, přenastavíme s_{start}, aby odpovídal současné pozici $s_{current}$ robota. V případě hledání cesty v neznámém statickém prostředí je algoritmus u konce. V opačném případě, kdy hledáme cestu v dynamickém prostředí, algoritmus ještě v průběhu pohybu robota zkoumá okolí, jestli nedošlo k nějaké změně, a tedy jestli je nutné cestu přepočítat. Když se objeví změna hranového ohodnocení grafu G, algoritmus upraví rhs-hodnoty ovlivněných uzlů. Tyto ovlivněné vrcholy jsou vloženy do prioritní fronty Q, pakliže se stanou lokálně nekonzistentními. Upravíme klíč všech uzlů v prioritní frontě, podle kterého se lokálně nekonzistentní vrcholy řadí, a přepočteme nejkratší cestu. Všechny kroky opakujeme, pokud se objeví další změna ohodnocení hran, dokud robot nedojde do cíle.

První verze D* Lite má nevýhody v neustálém přeskupování prvků v prioritní frontě Q, což může být časově náročné, jestli prioritní fronta Q často obsahuje velký počet uzlů. Druhá a sofistikovanější verze D* algoritmu využívá vyhledávací metodu odvozenou z originálního D* algoritmu, aby se vyhnula častému řazení prvků v prioritní frontě Q. Na rozdíl od první verze požadujeme po heuristice, aby byla nezáporná a nejen zpětně konzistentní, ale také dopředně konzistentní, tedy

$$\begin{aligned} &h(s,u) \geq 0 \wedge h(s,u) = 0 \Leftrightarrow u = s, && \text{(nezápornost)} \\ &h(s,s'') \leq d(s,s') + h(s',s''), && \text{(zpětnědopředná konzistence)} \end{aligned} \tag{2.4}$$

pro všechny vrcholy $s, s', s'' \in G$.

Heuristika $h(s,s')$ musí být také přípustným heuristickým odhadem, tedy heuristika $h(s,s')$ nesmí být větší než metrická vzdálenost $d(s,s') \forall s, s' \in G$, mezi kterými vede hrana $(s,s') \in G$. Heuristiky s těmito vlastnostmi splňují taktéž podmínky na heuristiku v prvním verzi algoritmu D* Lite.

Optimalizovaná verze D* Lite algoritmu (Alg. 2) řadí prvky prioritní fronty podle klíčů, které mají nižší hodnoty než v první verzi. Inicializace vrcholů probíhá stejně jako v první verzi D* Lite (Alg. 2, řádek 3). V případě pohybu robota z vrcholu s do vrcholu s' , kde zjistí, že došlo k změně hranového ohodnocení grafu G, přepočte první složku klíče $\mathbf{k}(s)$ (dvousložkový vektor). Přepočte se pouze první složka vektoru, která by se mohla zmenšit až o $h(s,s')$. Druhá složka není závislá na heuristice, proto zůstává beze změny.

Abychom se neustálému snižování klíče, podle kterého se vrcholy v prioritní frontě řadí, vyhnuli a zároveň tím omezili neustále řazení prvků v prioritní frontě,

použijeme proměnnou k_m. Kdykoliv jsou hodnoty klíčů $\mathbf{k}(s)$ prioritní fronty Q přepočítávány, musí k nim být přidána hodnota k_m, která se přidává do první složky

Algoritmus 2 Optimalizovaná verze D* Lite algoritmu

1: **procedure** CalcKey(Node s) ▷ Výpočet klíče prioritní fronty
2: **return** $[\min(s.g, s.rhs) + h(s_{start}, s) + k_m; \min(s.g, s.rhs)]$
3: **procedure** Initialize ▷ Inicializace problému
4: $Q \leftarrow \varnothing$, $k_m \leftarrow 0$
5: **for all** $s \in G$ **do** $s.rhs \leftarrow s.g \leftarrow \infty$ ▷ Nastav rhs a g-hodnoty
6: $s_{goal}.rhs \leftarrow 0$ ▷ Nejvyšší cena do cíle je 0
7: $Q.Insert(s_{goal}, CalcKey(s_{goal}))$ ▷ Vlož cíl do prioritní fronty
8: **procedure** UpdateVertex(Node s) ▷ Aktualizace vrcholu
9: **if** $s.g \neq s.rhs$ AND $s \in Q$ **then** $Q.Update(s, CalcKey(s))$
10: **else if** $s.g \neq s.rhs$ AND $s \notin Q$ **then** $Q.Insert(s, CalcKey(s))$
11: **else if** $s.g = s.rhs$ AND $s \in Q$ **then** $Q.Remove(s)$
12: **procedure** ComputeShortestPath
13: **while** $Q.TopKey() \dot{<} Key(s_{start})$ OR $s_{start}.rhs > s_{start}.g$ **do**
14: $s \leftarrow Q.Top()$ ▷ Zvol vrchol s nejmenším klíčem fronty
15: $k_{old} \leftarrow Q.TopKey()$ ▷ Ulož starý klíč vrcholu s
16: $k_{new} \leftarrow CalcKey(s)$ ▷ Spočti nový klíč vrcholu s
17: **if** $k_{old} \dot{<} k_{new}$ **then** $Q.Update(s, k_{new})$
18: **else if** $s.g > s.rhs$ **then** ▷ Cena cesty je vyšší než její max. hodnota
19: $s.g \leftarrow s.rhs$ ▷ Sniž cenu cesty
20: **for all** neighbours $neigh$ of vertex s **do** ▷ Pro každý sousední vrchol
21: **if** $neigh \neq s_{goal}$ **then** ▷ Sousední vrchol není cíl
22: $neigh.rhs \leftarrow \min(neigh.rhs, d(neigh, s) + s.g)$ ▷ Největší cena cesty rhs do vrcholu $neigh$ je minimum z rhs a ceny cesty ze souseda s
23: UpdateVertex($neigh$) ▷ Aktualizuj vrchol
24: **else**
25: $g_{old} \leftarrow s.g$, $g.s \leftarrow \infty$ ▷ Ulož starou cenu cesty, nová je ∞
26: **for all** neighbours $neigh$ of vertex s and vertex s **do** ▷ Pro vrchol s a jeho sousední vrcholy
27: **if** $neigh.rhs = d(neigh, s) + g_{old}$ **then**
28: **if** $neigh \neq s_{goal}$ **then** ▷ Sousední vrchol není cíl
29: $neigh.rhs \leftarrow \min_{s' \in Succ(neigh)}(d(neigh, s') + s'.g)$ ▷ Minimální cesta orientovaná do vrcholu $neigh$ ze všech jeho sousedních vrcholů
30: UpdateVertex($neigh, s$) ▷ Aktualizuj vrchol

vektoru $\mathbf{k}(s)$ (Alg. 2, řádek 2). Tím dosáhneme toho, že se pořadí lokálně nekonzistentních uzlů v prioritní frontě Q nemění. Hodnoty klíčů $\mathbf{k}(s)$ se stále přibližují hodnotám klíčů první verze D* Lite algoritmu právě přidáním proměnné k_m do první složky vektoru $\mathbf{k}(s)$.

V každém kroku výpočtu zvolíme vrchol s s nejmenším klíčem z prioritní fronty Q (Alg. 2, řádek 14) a na základě klíče z prioritní fronty rozhodneme, jak ho dál zpracovat. Pokud je jeho starý klíč $\mathbf{k}_{old}(s)$ menší než nově spočtený $\mathbf{k}_{new}(s)$, aktualizujeme klíč vrcholu s v prioritní frontě Q (Alg. 2, řádek 17). V odlišných případech probíhá výpočet pro vrchol s stejně jako v případě první verze algoritmu D* Lite. Provádí stejné operace a dokonce ve stejném pořadí jako první verze. Z toho je možné vyvodit, že optimalizovaná verze sdílí mnoho vlastností s původní verzí včetně její korektnosti.

Algoritmus může být také použit pro hledání cesty v neznámém prostředí, kde by graf G měl tvar sítě. Každý vrchol $s \in G$ by měl právě osm sousedních vrcholů. Cena hran $(s, s_{neigh}) \in G$ by byla zpočátku jednička, ale kdyby robot nalezl nepřístupné místo, pak by se cena hrany změnila na nekonečno.

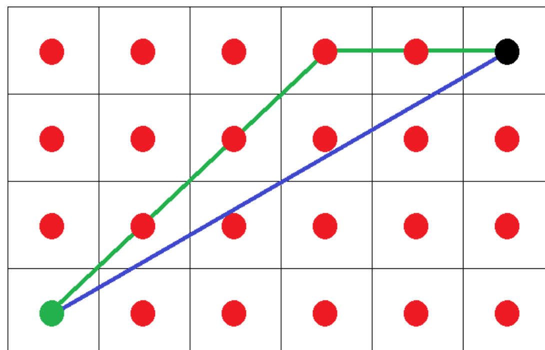
Field D*

Field D* algoritmus je metoda založená na interpolačním vyhledávání cesty a jejím interpolačním přepočtu. Tato metoda rozšiřuje D* a D* Lite algoritmy používáním lineární interpolace k efektivnímu výpočtu cest s nízkou cenou. Tyto cesty jsou vzhledem k použité lineární interpolaci optimální a v praxi velmi efektivní. V současné době se Field D* algoritmus hojně využívá v robotice pro rozsáhlé oblasti.

Problém, který řeší Field D* algoritmus, můžeme formulovat následovně. V rovinném prostředí rozděleném na uniformní síť čtvercových buněk T, kterým je přiřazena cena $c : T \to [0; +\infty)$ za průchod buňkou, a dva body $s_{start}, s_{goal} \in T$ najdi cestu sítí T z bodu s_{start} do bodu s_{goal} s minimální cenou.

Takový problém je důležité vhodně aproximovat. Velmi často se používá přístup, kdy se uniformní síť reprezentuje jako speciální neorientovaný planární graf G, jehož každý vrchol $s \in G$ má právě osm sousedních uzlů. Na tomto grafu se následně najde cesta grafem G. Obecně se přiřadí vrchol každému středu buňky a hranu dvěma uzlům přiřadíme právě tehdy, když spolu dvě buňky sousedí. Cena takové hrany odpovídá metrice mezi dvěma sousedními uzly.

Současné metody, které hledají nejkratší cestu pro takto interpretovaný problém, neposkytují přesná řešení, což je vidět na obrázku 2.2. Metody pro tento problém bez překážek poskytují zelenou nejkratší cestu z počátečního uzlu (zelený) do koncového uzlu (černý). To se snaží opravit metoda Field D*, jejíž snaha je nalézt kratší cestu (Obr. 2.2, modrá).



Obr. 2.2: Rozdíl v nalezené cestě a nejkratší cestě.

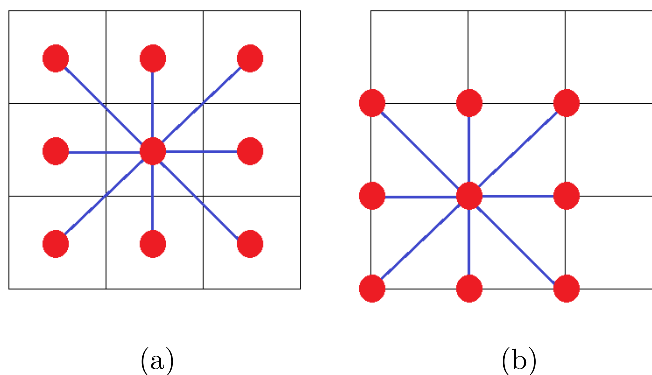
Field D* algoritmus je rozšířením již popsaných D* algoritmů. Využívá lineární interpolaci k výpočtu optimálních cest. Tato metoda nalézá přímější a levnější cesty než klasické buňkové vyhledávací algoritmy, aniž by utrpěla na výpočtu cesty v reálném čase.

Hlavním klíčem metody je upravení výpočtu uzlové ceny mezi libovolným vrcholem s a jeho sousedními vrcholy. Uzlovou cenou je myšlena nejlevnější cesta mezi daným vrcholem a cílovým vrcholem s_{goal}. V buňkovém grafu se pak taková vzdálenost počítá

$$g(s) = \min(c(s, s'), g(s')),$$

kde s' jsou vrcholy sousedící s vrcholem s, funkce $c(s, s')$ je cena hrany mezi vrcholem s a vrcholem s' a funkce $g(s')$ je uzlová cena sousedního vrcholu s'.

Další změnou oproti běžným buněčným algoritmům je reprezentace sítě jako grafu. Obvykle se střed buňky označí jako vrchol (Obr. 2.3 (a)). Přístup Field D* algoritmu je trochu odlišný. Vrchol grafu G nyní netvoří střed buňky, ale její roh (Obr. 2.3 (b).



(a) (b)

Obr. 2.3: (a) Běžná reprezentace buňkového grafu, (b) Reprezentace buňkového grafu v Field D* algoritmu

Algoritmus Field D* podle provedených testů poskytuje lepší výsledky [7] (ve smyslu nižší ceny výsledné cesty) než algoritmus D* Lite v případě buňkového prohledávání prostředí. Nalezená levnější cesta je vyvážena delší dobou výpočtu. Nicméně sami autoři metody [7] uvádějí, že metoda Field D* nemusí vždy nalézt levnější cestu než ostatní buněčné vyhledávací algoritmy, ačkoliv se jedná o velmi vzácný případ.

Poznamenejme, že nápad, se kterým přichází Field D*, lze jednoduchou úpravou přidat do všech buňkově zaměřených dynamických D* algoritmů. V současné době již existuje modifikovaná verze tohoto algoritmu, který tento nedostatek v podobě delšího časového výpočtu vylepšuje.

3 ŘEŠENÍ

Z prostudovaných existujících algoritmů pro hledání cest s časově proměnným ohodnocením a neúplným ohodnocením (viz kap. 2) byl vybrán algoritmus D* Lite. Důvodem této volby jsou převážně jeho výhody zmíněné v kapitole 2. Tedy jednodušší algoritmus a snaha ostatních novějších algoritmů být v něčem lepší než D* Lite. Nalezená cesta D* Lite algoritmem je vždy minimální možná, a proto budeme považovat cestu nalezenou D* Lite metodou za přesný výsledek. Díky tomu můžeme rozhodnout, jestli ostatní metody dávají také přesnou cestu či nikoliv.

Druhou podstatnou částí této práce je snaha ušetřit výpočetní čas pro hledání cest jednotlivcům. Bude tedy nutné rozhodnout, za jakých podmínek můžeme o dvou různých chodcích říci, že se mohou pohybovat po stejné cestě, i když se jejich počáteční pozice trochu liší. Tedy jestli je to vůbec možné a pokud je, tak o jaký čas urychlíme výpočet a jak velké chyby se při tom dopustíme.

Každá metoda hledání cest v této kapitole má trochu odlišné předpoklady, proto budou jednotlivé předpoklady vždy uvedeny na začátku dané podkapitoly.

Struktura města

Prvním problémem je zvolit vhodná data reprezentující město, která lze upravit a znázornit je jako neorientovaný graf G. Existují dva základní typy dat, kterými můžeme reprezentovat strukturu městské dopravní sítě. Prvním je tzv. silniční síť a druhým jsou bloková data, jejichž rozdíly jsou zachyceny v tabulce 3.1.

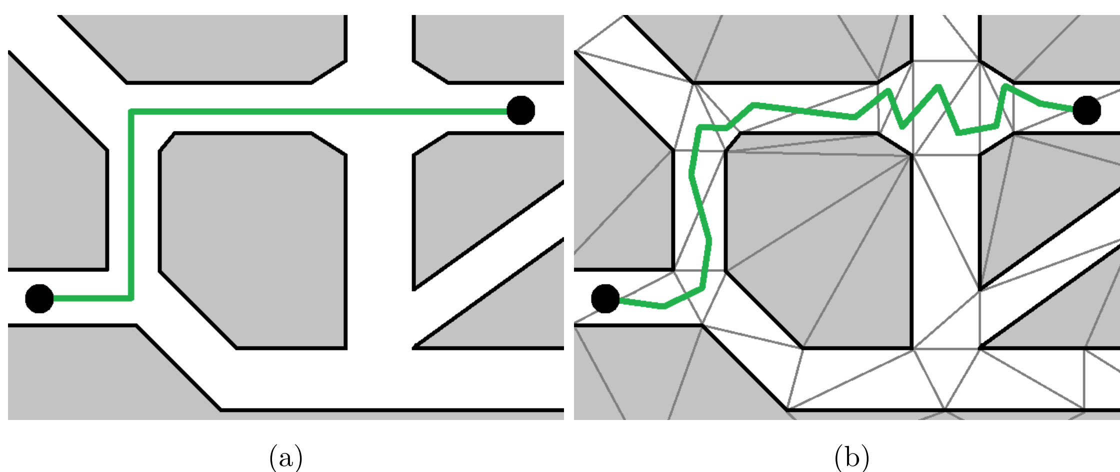
Silniční síť	Bloková data
podchody a mosty	parky a náměstí
rychlejší vytvoření grafu	pomalejší vytvoření grafu
reálná data na OSM	reálná data nejsou k dispozici
odtrhnutí se od EcoSim	pokračování v EcoSim
přímější cesta	„zig-zag“ cesta

Tab. 3.1: Rozdíly mezi dvěma typy vstupních dat

Podívejme se nejprve na data silniční sítě, kde křižovatky znázorníme jako vrcholy grafu G, a silnice, které je spojují, jako hrany grafu G. Každá silnice má svoji délku d. Tuto délku můžeme připsat ke každé hraně, čímž dostaneme ohodnocený graf G. Výhodou těchto dat je, že již nemusíme ručně přiřazovat vrcholy grafu křižovatkám a hrany grafu silnicím, protože jsou dostupná na serveru Open Street Map [18]. Jak již název napovídá, jedná se o databázi silnic, budov, řek a jiných

venkovních prostor, která je přístupná všem. Její největší klad je zároveň největším záporem, protože kdokoliv může data přidávat či měnit. Může se tedy stát, že např. někde není propojení dvou křižovatek silnic, i když ve skutečnosti propojené jsou. Dalším negativem mohou být chybějící základní parametry, jako je šířka silnic či jejich nadmořská výška.

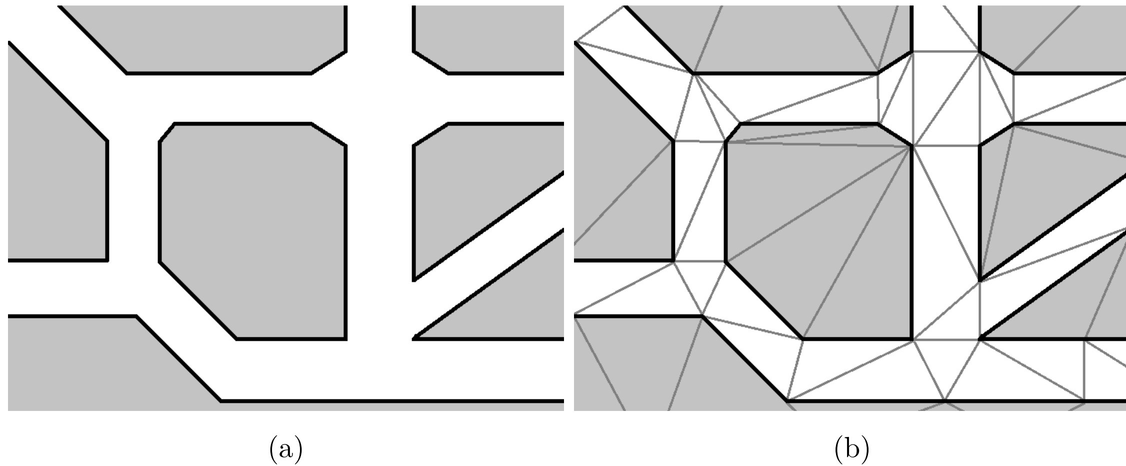
Ve 2D prostředí mohou data silniční sítě mohou reprezentovat taktéž podchody či mosty. Nevadí nám, že výsledný graf není planární. Na druhou stranu s nimi nelze dobře reprezentovat parky či náměstí, protože silniční síť není schopna zachovat jejich tvar či velikost. Výhodou tohoto typu dat je výsledná nalezená cesta. Na obrázku 3.1 (a) je vidět příklad výsledné cesty. Můžeme si všimnout, že cesta je přímá, a dokážeme si představit, že se taková cesta od skutečné nebude příliš lišit.



Obr. 3.1: (a) Výsledná cesta na datech silniční sítě, (b) Výsledná cesta na blokových datech

Druhým přístupem je použití tzv. blokových dat. Tato data, která jsou primárně pro počítačovou grafiku, od sebe oddělují prostupná místa od neprostupných. Načítáme tedy posloupnost bodů, které definují polygony. Tyto polygony označíme jako neprostupné, čímž dostaneme strukturu virtuálního města (Obr. 3.2 (a)). Následně je nutné pro všechny opěrné body (body tvořící polygon) spočítat CDT (Definice 2.13, Obr. 3.2 (b)). Na obrázku 3.2 jsou bílou barvou vyznačeny silnice a šedivě jsou vyznačeny neprostupné bloky budov. Nyní již můžeme intuitivně reprezentovat CDT grafem G. Každý trojúhelník znázorníme jako uzel grafu G a každé dva sousedící trojúhelníky jako hrany grafu G. Takto reprezentovaný graf je ještě možné upravit pomocí vhodné datové struktury. Tou může být buněčný a portálový graf [13][16] nebo navigační graf [13][16], čímž zmenšíme rozměr grafu a zároveň urychlíme výpočet následných vyhledávacích grafových metod. Problém těchto dat je, že reálná data nejsou nikde volně k dispozici. Existuje trochu bolestivý způsob, jak tato

data vytvořit, který je ale mnohem rychlejší než ruční nadefinování bloků budov. Lze použít data silniční sítě ze serveru Open Street Map a načíst je programem Esri CityEngine [17], který silniční síť převede na bloková data. Ta je nutné ještě v programu Esri CityEngine ručně upravit, protože se všechny silnice nemusí korektně převést na bloková data. Problém tohoto přístupu je vnesení chyby do blokových dat. Šířka silnic je nastavena podle vnitřních parametrů programu, protože data z Open Street Map neobsahují údaj o šířce cesty.



(a) (b)

Obr. 3.2: (a) Struktura města pomocí blokových dat, (b) Spočtená CDT na blokových datech

Dalším problémem těchto dat je nalezená cesta (Obr. 3.1 (b)). Je zřejmé, že tato cesta je delší než v případě silničních dat (Obr. 3.1 (a)). Navíc je zřejmé, že tato cesta má k reálnému pohybu chodce velmi daleko, pokud se nesnažíme simulovat pohyb opilého jedince. Tuto výslednou cestu na blokových datech je možné aproximovat, aby byla přímější a intuitivnější, což bude stát další výpočetní čas.

Bloková data nám neumožňují vytvářet podchody a mosty, protože jejich reprezentace nám umožňuje vytvářet pouze planární graf G. Na druhou stranu tento typ dat velmi dobře reprezentuje parky a náměstí (např. tvar a velikost). Tyto rozdíly uvažujeme ve 2D prostředí a ve vyšší dimenzi už nemusí nutně platit. Např. ve třetí dimenzi by již bylo možné do blokových dat zahrnout i mosty a podchody.

Na základě těchto rozdílů mezi jednotlivými typy dat jsme se rozhodli pro data silniční sítě. Touto volbou jsme museli upustit od programu EcoSim, protože tento program načítal bloková data městské struktury. Navíc máme k dispozici pouze jeden soubor s vstupními daty pro program EcoSim a nepodařilo se nám najít reálná bloková data, jenž by byly volně ke stažení. To je první hlavní důvod, proč jsme se rozhodli od programu EcoSim upustit a vytvořit program pro volně dostupná

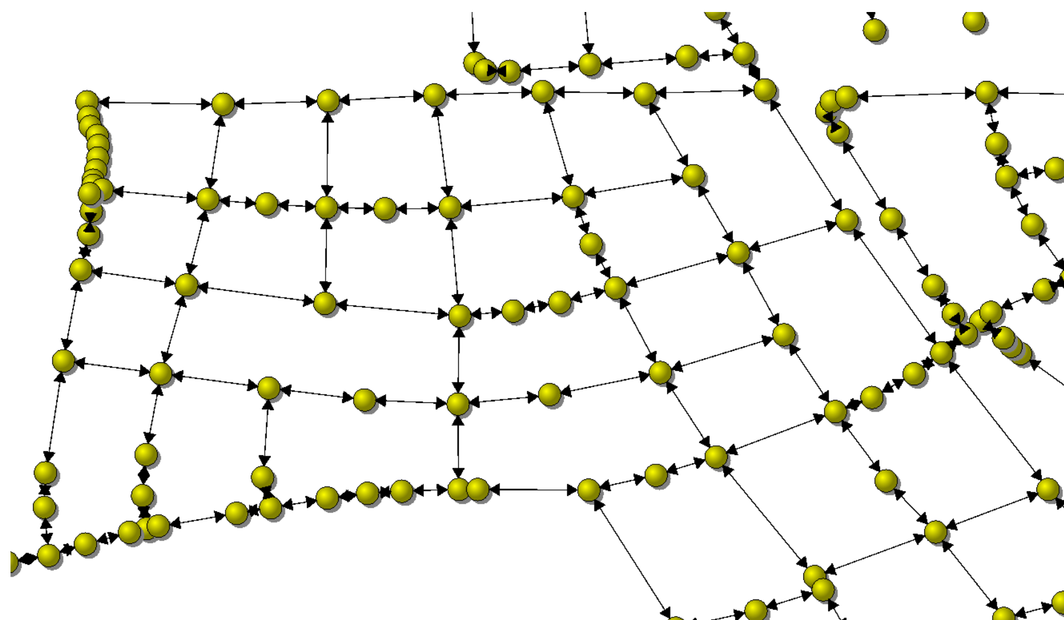
a reálná data silniční sítě. Dalším hlavním důvodem je jejich rychlejší vytvoření grafové reprezentace z tohoto typu dat.

Vizualizace

Bylo tedy nutné vytvořit nový program, který by data silniční sítě načítal a vhodně vizualizoval, aby bylo možné posoudit nejen numerické výsledky výpočtu, ale taktéž optickou změnu cest.

Vzhledem k tomu, že nebyly kladeny žádné specifické požadavky na programovací jazyk či použité knihovny, jsme se rozhodli ponechat stejný programovací jazyk, jakým je naprogramovaný EcoSim, tedy C/C++. Učinili jsme tak, aby bylo možné bez větších problémů přenést implementované metody do programu EcoSim, kdybychom se k němu rozhodli vrátit.

Abychom nemuseli vytvářet veškeré uživatelské prostředí a vizualizaci úplně sami, použili jsme k jejímu vytvoření knihovnu Qt [22] ve verzi 5.3.2. To je knihovna specializovaná právě na jednoduché vytvoření uživatelského prostředí pro programovací jazyk C++. Qt knihovna mimo jiné obsahuje také OpenGL knihovnu [21], která slouží pro tvorbu aplikací počítačové grafiky. Používá se při tvorbě virtuální reality, CAD programů či počítačových her.



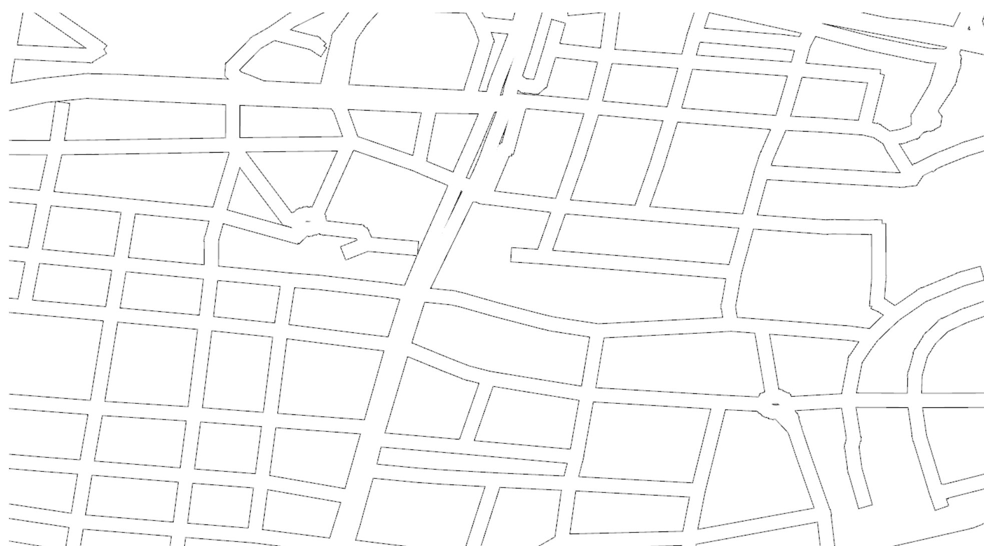
Obr. 3.3: Reprezentace grafu

První a nejjednodušší možností, jak vykreslit půdorys města, je vykreslit načtený neorientovaný graf G (Obr. 3.3). K tomuto způsobu reprezentace dat lze využít tutoriál, který obsahuje knihovna Qt, a jednoduše ho upravit pro vlastní potřeby.

Problém nastává v momentě, kdy chceme přidat do vykreslení i pohyb chodce. Vykreslit pohyb chodce je v tomto případě možné, ale opticky nevyhovující. I přesto je tento typ vizualizace použit, ale pouze pro porovnání rozložení vrcholů $s \in G$ a hran $e \in G$ neorientovaného grafu G.

Další možností, jak zobrazit půdorys města, je vytvořit vlastní jednoduché vykreslování grafu (Alg. 3). Toto zobrazení se bude od toho předchozího lišit v šířce hran, tedy silnic. Námi implementovaná metoda, která je velice jednoduchá, zobrazí půdorys města, který bude obsahovat silnice s nenulovou šířkou. Nejprve každé hraně $e \in G$ spočteme její normálu n, kterou následně normujeme. Každá hrana grafu navíc obsahuje reprezentaci dvou úseček $e.first, e.second$, jejichž počátek je v proměnných $e.first.start$ a $e.second.start$ a konec v $e.first.end$ a $e.second.end$. Nejprve tedy vykreslíme každé hraně černě vyplněný čtyřúhelník tvořený vrcholy $e.first.start, e.first.end, e.second.end$ a $e.second.start$. V dalším kroku pak vykreslíme bíle vyplněný čtyřúhelník, který tvoří tytéž vrcholy, jen je ve směru, resp. proti směru normály n užší než černý čtyřúhelník. Vrcholy jsou seřazeny tak, aby tvořily strany čtyřúhelníku. Tedy $e.first.start, e.first.end$ tvoří první stranu, $e.first.end$, $e.second.end$ druhou stranu, $e.second.end, e.second.start$ třetí stranu a $e.second.start$, $e.first.start$ čtvrtou stranu. Nezáleží na tom, jakým vrcholem začneme, ani na směru pohybu (pravotočivý či levotočivý), nutné je pouze, aby sousední vrcholy tvořily hranu čtyřúhelníku. V opačném případě dojde ke špatné vizualizaci.

Tímto způsobem je možné dosáhnout nenulové šířky silnic $2k$ (Obr. 3.4). Uvědomme si, že algoritmus 3 je možné upravit tak, že se normála každé hrany spočte vždy pouze jednou, čímž zvýšíme výpočetní rychlost.



Obr. 3.4: Vizualizace algoritmem 3

Algoritmus 3 Metoda pro vizualizaci silnic

Require: Undirected graph G

1: **for all** edges $e \in G$ **do**
2: compute normal n of edge e
3: $n \leftarrow \frac{n}{||n||}$
4: $e.first \leftarrow e + (k + \epsilon) \cdot n$, $e.second \leftarrow e - (k + \epsilon) \cdot n$
5: plot black quadrilateral $(e.first.start, e.first.end, e.second.end, e.second.start)$
6: $e.first \leftarrow e + k \cdot n$, $e.second \leftarrow e - k \cdot n$
7: plot white quadrilateral $(e.first.start, e.first.end, e.second.end, e.second.start)$

Tento způsob vykreslování není optimální, a proto jsme zkoušeli samotnou vizualizaci optimalizovat a urychlit. Implementovali jsme vlastní metodu, která se snaží o sofistikovanější vizualizaci. Princip tohoto algoritmu spočívá v tom, že nejprve pro každou hranu $e \in G$ neorientovaného grafu G spočteme její normálu n, kterou je nutné následně normovat (Alg. 4). Každá hrana grafu navíc obsahuje reprezentaci dvou úseček $e.first, e.second$. První úsečku definujeme jako posunutí hrany grafu ve směru normály o velikost k, druhou pak jako posunutí proti směru normály o tutéž velikost k. Tímto způsobem každou silnici o nulové šířce můžeme vykreslit jako silnici o šířce $2k$.

Algoritmus 4 Metoda pro výpočet vizualizace silnic

Require: Undirected graph G

1: **for all** edges $e \in G$ **do**
2: compute normal n of edge e
3: $n \leftarrow \frac{n}{||n||}$
4: $e.first \leftarrow e + k \cdot n$, $e.second \leftarrow e - k \cdot n$
5: Sort all edges $e \in G$
6: **for all** nodes $s \in G$ **do**
7: **for all** edges $e \in G$ coming from node s **do**
8: $e_1 \leftarrow e$, $e_2 \leftarrow$ right-handed neighbouring edge of edge e
9: $p \leftarrow$ intersection $(e_1.second, e_2.first)$
10: change $e_1.second$ length from p to end of $e_1.second$
11: change $e_2.first$ length from p to end of $e_2.first$
12: plot city structure

Po seřazení všech takto spočtených úseček $e.first, e.second$ je dalším krokem upravení jejich délek. Zvolíme vrchol $s \in G$ projdeme všechny hrany, které z něj vedou. Nejprve zvolíme hranu $e_1 \in G$ a její sousední hranu v pravotočivém smyslu $e_2 \in G$. Následně spočteme průsečík p dvou nejbližších úseček $e_1.second$ a $e_2.first$

sousedících hran $e_1, e_2 \in G$. Délku těchto úseček náležitě upravíme (zkrátíme či prodloužíme) s ohledem na jejich nalezený průsečík p. Tento postup opakujeme pro všechny vrcholy $s \in G$ neorientovaného grafu G.

Vizualizace městského půdorysu už je potom jednoduchá a velmi rychlá. Stačí vždy pouze vykreslit pro každou hranu $e \in G$ její dvě předem spočtené a přilehlé hranové úsečky $e.first$ a $e.second$. Problém této metody spočívá v poměrně složitém hledání chyby v seřazení hran (časově náročné). To je vidět na obrázku 3.5, kde jsou chybně vykreslené hrany (cesta vlevo dole a kruhový objezd na pravé straně obrázku). Z nedostatku času jsme od této metody upustili a používáme první metodu. Jsme si vědomi, že to není optimální způsob vykreslování. Nejedná se ale o hlavní cíl této práce, a proto se nám tento způsob vizualizace jeví prozatím postačující.

Obr. 3.5: Chybné vykreslení upravenou metodou

Globální vyhledávací algoritmy

Jednou ze dvou hlavních částí této diplomové práce jsou vyhledávací algoritmy. Ze statických algoritmů jsme jako zástupce zvolili A* algoritmus, abychom mohli porovnat metodu pro dynamicky se měnící prostředí s metodou pro statické prostředí. Zároveň je tato volba učiněna proto, abychom zjistili, do jaké míry je možné opakovaně používat statický algoritmus v dynamickém prostředí.

Druhou důležitou metodou je algoritmus pro dynamicky se měnící prostředí. Rozhodli jsme se pro existující algoritmus D* Lite, který ze všech D* algoritmů nachází nejlepší cestu.

A* algoritmus

Návod, jak implementovat heuristický algoritmus A* pro výpočet nejkratší cesty ve statickém prostředí, dává kapitola 2. Vzhledem k tomu, že naše implementovaná verze se shoduje s pseudokódem, nebudeme tedy popisovat algoritmus samotný, ale upozorníme pouze na několik implementačních zajímavostí.

Samotná implementace A* algoritmu se nachází v souboru *graph.cpp* a je možné ho spustit příkazem $computeTrajectory(CPedestrian\ \&ped)$. Je tedy zřejmé, že potřebujeme jediný parametr *ped*, což není nic jiného než objekt třídy *pedestrian.cpp*. Ten v sobě obsahuje informace o počáteční a koncové pozici chodce, tedy počáteční uzel s_{start} a cílový uzel s_{goal} neorientovaného grafu G. Nalezenou nejkratší cestu pak ukládáme do sběrného kontejneru *vector* z knihovny STL (základní knihovna C++), který je již připraven v objektu *ped*.

Podstatné je také zmínit, že prioritní frontu reprezentujeme haldou taktéž z knihovny STL. Halda je stromová datová struktura splňující podmínky prioritní fronty.

D* Lite algorimus

První důležitou věcí v D* Lite algoritmu je vytvoření vlastní kopie neorientovaného grafu G každému chodci. Tato kopie je nutná podmínka pro vyhledávání trasy každému chodci. Důvodem je jedinečné ohodnocení grafových vrcholů g-hodnotami a rhs-hodnotami. Každá cesta ohodnotí graf rozdílně, proto je nutné vytvořit kopie grafu, aby si algoritmus nepřepisoval data uložená ve vrcholech grafu G.

Je zřejmé, že na rozdíl od A* algoritmu, kterému stačí pro jeden tisíc chodců hledajících nejkratší cestu jeden neorientovaný graf G, bude D* Lite mnohonásobně paměťově náročnější. V případě již zmíněných tisíci chodců budeme muset vytvořit tisíc kopií grafu G. Ačkoliv se může tento přístup zdát poněkud nesmyslný, předpokládáme, že při velkém počtu hranových změn bude D* Lite algoritmus mnohem výhodnější.

Na druhou stranu je možné, že už z předpokladů úlohy je vidět, že použití D* Lite algoritmu bude lepší než A* algoritmus. Příkladem může být, že každý chodec má specifické ohodnocení grafu. V takovém případě je nutné i pro A* algoritmus vytvořit každému chodci vlastní graf. Pokud navíc budeme v této úloze uvažovat dynamicky se měnící prostředí, je D* Lite jasným kandidátem na lepší algoritmus.

Algoritmus D* Lite byl popsán v kapitole 2, takže máme návod, jak ho implementovat. Nicméně zde byl D* Lite popsán pro orientovaný ohodnocený graf a my uvažujeme graf neorientovaný ohodnocený, proto je nutné algoritmus drobně obměnit. Pseudokód upravené metody pro neorientovaný graf je uveden v algoritmu 5. Nejdůležitějším rozdílem je úprava prohledávaných sousedních vrcholů z vrcholů,

do kterých vede orientovaná hrana, na prohledání všech sousedních vrcholů (Alg. 5, řádka 24 či 30).

Algoritmus 5 D* Lite pro neorientovaný graf G a chodce *ped*

Require: Pedestrian *ped*

1: **procedure** CalcKey(Node s, Pedestrian *ped*) ▷ Výpočet klíče prioritní fronty
2: **return** $[\min(s.g, s.rhs) + h(ped.start, s) + ped.k_m; \min(s.g, s.rhs)]$
3: **procedure** Initialize(Pedestrian *ped*) ▷ Inicializace grafu G pro chodce *ped*
4: $ped.Q \leftarrow \varnothing$, $ped.k_m \leftarrow 0$
5: **for all** $s \in ped.G$ **do** $s.rhs \leftarrow s.g \leftarrow \infty$ ▷ Nastav rhs a g-hodnoty
6: $ped.end.rhs \leftarrow 0$ ▷ Nejvyšší cena do cíle je 0
7: $ped.Q.Insert(ped.end, CalcKey(ped.end, ped))$ ▷ Vlož cíl do prioritní fronty
8: **procedure** UpdateVertex(Node s, Pedestrian *ped*) ▷ Aktualizace vrcholu
9: **if** $s.g \neq s.rhs$ AND $s \in ped.Q$ **then** $ped.Q.Update(s, CalcKey(s))$
10: **else if** $s.g \neq s.rhs$ AND $s \notin ped.Q$ **then** $ped.Q.Insert(s, CalcKey(s))$
11: **else if** $s.g = s.rhs$ AND $s \in ped.Q$ **then** $ped.Q.Remove(s)$
12: **procedure** ComputeShortestPath(Pedestrian *ped*)
13: **while** $ped.Q.TopKey() \dot{<} Key(ped.start, ped)$ OR $ped.start.rhs > ped.start.g$ **do**
14: $s \leftarrow ped.Q.Top()$ ▷ Zvol vrchol s nejmenším klíčem fronty
15: $k_{old} \leftarrow ped.Q.TopKey()$ ▷ Ulož starý klíč vrcholu s
16: $k_{new} \leftarrow CalcKey(s, ped)$ ▷ Spočti nový klíč vrcholu s
17: **if** $k_{old} \dot{<} k_{new}$ **then** $ped.Q.Update(s, k_{new})$
18: **else if** $s.g > s.rhs$ **then** ▷ Cena cesty je vyšší než její max. hodnota
19: $s.g \leftarrow s.rhs$ ▷ Sniž cenu cesty
20: **for all** neighbours *neigh* of vertex s **do** ▷ Pro každý sousední vrchol
21: **if** $neigh \neq ped.end$ **then** ▷ Sousední vrchol není cíl
22: $neigh.rhs \leftarrow \min(neigh.rhs, d(neigh, s) + s.g)$
23: UpdateVertex$(neigh, ped)$ ▷ Aktualizuj vrchol
24: **else**
25: $g_{old} \leftarrow s.g$, $g.s \leftarrow \infty$ ▷ Ulož starou cenu cesty, nová je ∞
26: **for all** neighbours *neigh* of vertex s and vertex s **do**
27: **if** $neigh.rhs = d(neigh, s) + g_{old}$ **then** ▷ Největší cena cesty je cena původní cesty
28: **if** $neigh \neq ped.end$ **then** ▷ Sousední vrchol není cíl
29: $neigh.rhs \leftarrow \min_{s' \in Neigh(neigh)}(d(neigh, s') + s'.g)$ ▷ Minimum ze všech cest do vrcholu *neigh* z jeho sousedních vrcholů
30: UpdateVertex$(neigh, ped)$ ▷ Aktualizuj vrchol

Podle algoritmu 5 je v třídě *graph.cpp* implementovaná optimalizovaná verze D* Lite algoritmu pro neorientovaný graf G. Je možné si všimnout, že jediným potřebným parametrem je objekt chodce *ped* definovaný ve třídě *pedestrian.cpp*. Jak jsme již poznamenali výše, každý chodec obsahuje vlastní kopii grafu G, tedy $ped.G$. Zároveň je taktéž nutné pro každou metodu vytvořit samostatnou prioritní frontu (Alg. 5, $ped.Q$). Důvodem je taktéž snaha se vyhnout přepisování dat v prioritní frontě jako v případě přepisování dat ve vrcholech grafu G.

Dále je taktéž důležité vysvětlit význam značení $\dot{<}$, které je použito např. na 17. řádku v algoritmu 5. Takto značíme porovnávání vektorů v lexikografickém smyslu. Přesněji řekneme, že vektor k_1 je menší než vektor k_2, pokud první složka vektoru k_1 je menší než první složka vektoru k_2, tedy $k_1.x < k_2.x$. Zároveň můžeme také říci, že vektor k_1 je menší než vektor k_2, pokud jsou si jejich první složky rovny a druhá složka prvního vektoru k_1 je menší než druhá složka vektoru k_2, tedy $k_1.x = k_2.x \wedge k_1.y < k_2.y$.

Známé prostředí

První nabízenou možností je řešit hledání nejkratší cesty ve známém dynamickém prostředí. Kdykoliv nastane změna v dynamicky se měnícím prostředí, reagujeme na to přepočítáním cesty, protože víme, že změna nastala, a víme, kde.

Dynamické prostředí reprezentujeme neorientovaným grafem G, jehož hrany mohou v čase t vznikat, zanikat či měnit svoje ohodnocení pro $t \in [0, +\infty)$. Vždy, když v čase t dojde k libovolné grafové změně, spustíme zvolený globální algoritmus, který nalezne novou nejkratší cestu na změněném grafu G'. Tento proces opakujeme pro každého chodce tak dlouho, dokud nedojde do požadované cílové pozice.

Tato část je řešena ve třídě *thread.cpp*, kde se nachází hlavní výpočetní cyklus programu. Pseudokód si nebudeme uvádět, protože se jedná o velice jednoduchý problém. Vždy, když dojde ke změně grafu G, program na to zareaguje a okamžitě přepočte cestu, která se následně může lišit od původní cesty.

První možností je použít při každé grafové změně A* algoritmus. Uvědomme si, že než chodec ujde třeba 100 metrů (z celkové vzdálenosti např. 100km), tak může v grafu dojít k změnám v řádech tisíců. Přepočítávat pokaždé celou cestu určitě není moudré, ale jedná se o naivní přístup. Druhým extrémním případem může být, že naopak v grafu nedojde k téměř žádným změnám. V takovém momentu je naopak použití A* algoritmu rozumné.

Druhou možností je použití D* algoritmu, který ale nelze volat pouze příkazem $ComputeShortestPath(Pedestrian\ ped)$, ale je nutné přidat pár řádek kódu k jeho správnému použití.

Algoritmus 6 Přepočet cesty D* Lite algoritmem ve změněném grafu G'

1: **if** there is a change in graph G **then**
2: $\quad ped.k_m = ped.k_m + h(s_{start}, ped.start)$
3: $\quad s_{start} \leftarrow ped.start$
4: $\quad$ **for all** changed edges $e = (s, u)$ **do** $d_{old} = d(s, u)$ in G
5: $\quad\quad$ **if** $d_{old} > d(s, u)$ in G' **then**
6: $\quad\quad\quad$ **if** $s \neq s_{goal}$ **then** $s.rhs \leftarrow \min(s.rhs, c(s, u) + u.g)$ in G'
7: $\quad\quad$ **else if** $s.rhs = d_{old} + u.g$ **then**
8: $\quad\quad\quad$ **if** $s \neq s_{goal}$ **then** $s.rhs \leftarrow \min_{s' \in Neigh(s)}(d(s, s') + s'.g)$
9: $\quad\quad$ UpdateVertex(s, ped)
10: $\quad$ ComputeShortestPath(ped)

Z algoritmu 6 je možné vyčíst, že D* Lite algoritmus nepočítá celou cestu znovu, protože již z prvního vyhledávání má vlastní kopii grafu specificky ohodnocenou. Algoritmus 6 reaguje na změnu pouze lokálně. Pokud by změna ovlivňovala nalezenou cestu, algoritmus přepočítá pouze část úseku ovlivněný touto změnou a nikoliv celou cestu jako A* algoritmus. Z toho je patrné, že pro velký počet grafových změn je D* Lite vhodnější volbou, protože neupravuje celou cestu, ale pouze nutný úsek cesty, který se stal lokálně nekonzistentní. Na druhou stranu není moc smysluplné používat D*Lite pouze pro jednu nebo dvě grafové změny. Bylo by lepší použít dvakrát statický algoritmus A* než zabírat velké množství paměti pro kopii grafu G.

Částečně známé prostředí

Další problém, který řešíme, je hledání nejkratší cesty v částečně známém dynamickém prostředí. Chodec si nejprve naplánuje nejkratší cestu v grafu G počátečního vrcholu s_{start} do koncového vrcholu s_{goal} a vydá se po ní. V průběhu jeho pohybu do cílového vrcholu s_{goal} probíhají změny v hranovém ohodnocení (hrany vznikají, zanikají či mění své ohodnocení) grafu G, na které nijak nereaguje, protože o nich neví. Na změnu grafu G reaguje až v případě, že se na jeho naplánované cestě vyskytla změna, která v původním plánování nebyla. Je tedy zřejmé, že pokud by se v průběhu jeho pohybu objevila nová hrana grafu G, která by jeho cestu k cílovému vrcholu s_{goal} urychlila, chodec by se po ní nevydal. Nemá totiž o takové změně tušení.

Tento problém je také vyřešen ve třídě *thread.cpp*. Pro výpočet A* algoritmem opakovaně voláme metodu pro výpočet nejkratší cesty jako v případě známého prostředí. Rozdílem je, že tuto metodu nevoláme při každé grafové změně, ale pouze v případě, že takovou změnu objevíme na naplánované cestě (např. chodec dojde

k zablokované ulici).

Algoritmus 7 Přepočet cesty D* Lite algoritmem ve změněném grafu G'

1: **if** pedestrian *ped* found changed edge $e = (s, u)$ on his path **then**
2: $ped.k_m = ped.k_m + h(s_{start}, ped.start)$
3: $s_{start} \leftarrow ped.start$
4: $d_{old} = d(s, u)$ in G
5: **if** $d_{old} > d(s, u)$ in G' **then**
6: **if** $s \neq s_{goal}$ **then** $s.rhs \leftarrow \min(s.rhs, c(s, u) + u.g)$ in G'
7: **else if** $s.rhs = d_{old} + u.g$ **then**
8: **if** $s \neq s_{goal}$ **then** $s.rhs \leftarrow \min_{s' \in Neigh(s)}(d(s, s') + s'.g)$
9: UpdateVertex(s, ped)
10: ComputeShortestPath(ped)

Pro případ výpočtu cesty D* Lite algoritmem trochu upravíme algoritmus 6 zmíněný pro známé prostředí na algoritmus 7. Z algoritmu 7 je zřejmé, že na grafové změny reagujeme pouze v případě, že je chodec *ped* nalezne na své předem naplánované cestě.

Lokální vyhledávací algoritmy

Představme si situaci, kdy si plánujeme výlet. Náš kamarád nám doporučí navštívit např. Karlovy Vary a zdejší památky, které bychom si měli projít. Naplánujeme si tedy trasu, po které se vydáme, a vyrazíme. Když jsme na místě a od první památky se přesouváme ke druhé, nastane problém. Část naší naplánované trasy je přehrazena a nelze tudy projít. Mapu jsme si zapomněli doma a zároveň se nechceme ptát nikoho z kolemjdoucích, protože by to neměl být až takový problém. Naší prioritou tedy není naplánovat novou nejkratší cestu městem, ale obejít překážku a dostat se zpět na naši původní trasu. A tu obejdeme odhadem, kudy by to mohlo být nejblíž, abychom si moc nezašli. A mohou nastat dvě možnosti. Buď jsme cestu odhadli správně a došli jsme na původní trasu, nebo jsme špatně odhadli směr a raději se už zeptáme, kudy se dostat zpět.

Existují i další příklady, které se nemusí nutně týkat turistů. Od návštěv neznámých měst až po procházení málo známou čtvrtí rodného města. A právě takový problém řeší lokální vyhledávací algoritmus 8. Než se pustíme do rozboru algoritmu, formulujme si problém, který řešíme.

Každý chodec má svoji cestu naplánovanou pomocí globálního algoritmu z počátečního vrcholu s_{start} do koncového vrcholu s_{goal}. Pokud na své trase nenarazí

na překážku, bez větších obtíží dorazí do cíle s_{goal}. V opačném případě je nutné objevenou překážku obejít. Zde nastává spuštění algoritmu 8.

Algoritmus 8 Lokální vyhledávací metoda

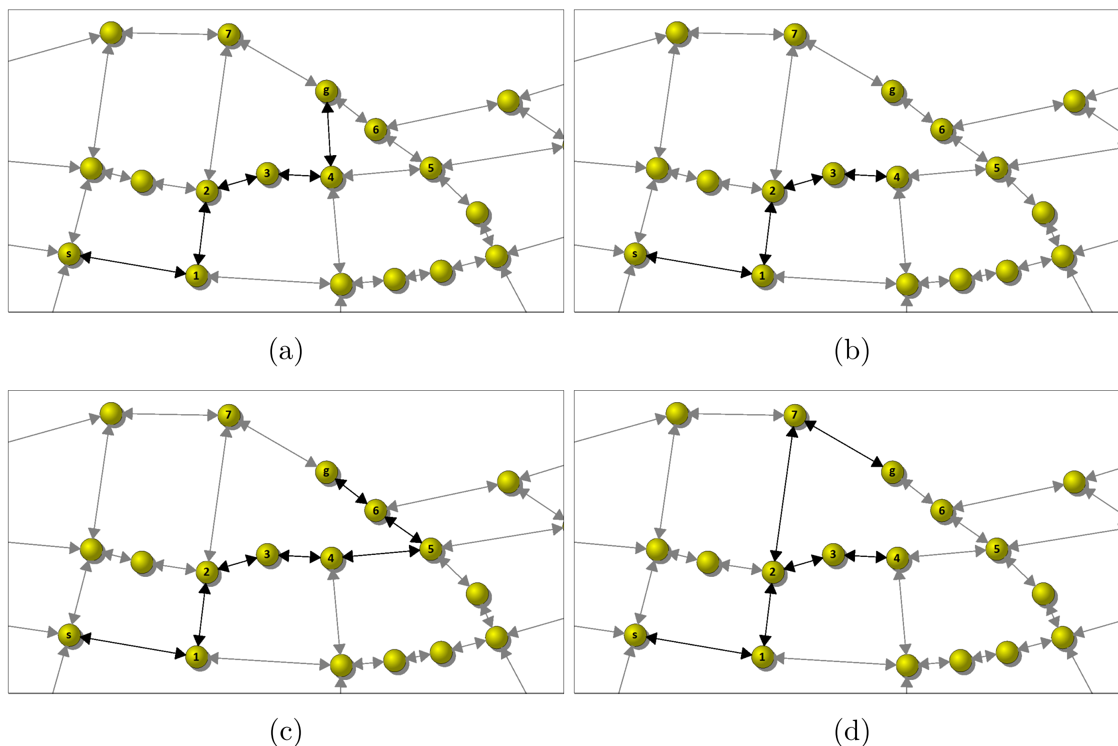
Require: Pedestrian *ped*, Undirected graph G

1: $pos \leftarrow ped.start$, $dist \leftarrow \infty$
2: **while** *ped.end* not found **do**
3:     **for all** neighbours of $pos \in G$ **do**
4:         **if** $h(neigh, ped.end) < dist$ **then** $dist \leftarrow h(neigh, ped.end)$
5:     Add *neigh* with minimal *dist* to *ped.path*
6:     $pos \leftarrow neigh$
7:     **if** $h(neigh, ped.end)$ is large **then** compute path with global method

Vstupní parametr *ped* je objekt třídy *pedestrian.cpp*, jak již jsme zmínili dříve. Tento parametr obsahuje současnou pozici chodce *ped.start* a hledaný vrchol *ped.end* grafu G, tedy vrchol grafu za vzniklou překážkou. Princip metody je, že pro všechny vrcholy *neigh* sousedící s vrcholem současné pozice chodce *ped.start* provedeme odhad vzdálenosti z pozice sousedního vrcholu *neigh* do hledaného místa *ped.end* a vybereme nejmenší možný. Chodec se přemístí do vrcholu *neigh* s nejmenším odhadem vzdálenosti do uzlu *ped.end* a výpočet odhadu začíná znovu.

Podstatné na tomto algoritmu jsou čtyři věci. Zaprvé je možné, že se chodec bude neustále vracet na svoji pozici, odkud přišel. Je tedy nutné tento problém odstranit. Zadruhé, jak již bylo zmíněno na začátku této podkapitoly, může se stát, že algoritmus vybere špatnou cestu, a proto je nutné i tento problém ohlídat. Řešení je v algoritmu 8 zaneseno na deváté řádce, kdy v případě příliš velké vzdálenosti od hledaného vrcholu *ped.end* raději spustíme globální metodu, která chodce nasměruje správným směrem anebo ho alespoň informuje o tom, že cesta do této pozice neexistuje. Zatřetí odhadem vzdálenosti máme na mysli heuristiku $h(s, u)$, která spočte Eukleidovskou metriku mezi vrcholy $s \in G$ a $u \in G$. Začtvrté si uvědomme, jaké vrcholy volíme pomocí heuristického odhadu a jaký to může mít následek, což si vysvětlíme s pomocí obrázku 3.6.

Na obrázku 3.6 (a) je vidět příklad nalezené cesty z vrcholu s do vrcholu g. Nalezenou cestu tvoří posloupnost vrcholů $(s, 1, 2, 3, 4, g)$. Černě vykreslené hrany indikují cestu chodce a šedivé hrany reprezentují existující průchod (silnici nebo chodník). Když chodec dojde do vrcholu č. 4, zjistí, že cesta mezi vrcholem č. 4 a vrcholem g není průchozí (Obr. 3.6 (b)). Je tedy nutné nalézt novou cestu a je zřejmé, že nejkratší cesta by po následném výpočtu měla vypadat jako na obrázku 3.6 (c), tedy celou cestu by tvořila posloupnost vrcholů $(s, 1, 2, 3, 4, 5, 6, g)$. Nicméně



Obr. 3.6: (a) Naplánovaná cesta chodce, (b) Nalezená překážka v grafu G, (c) Optimální přepočet cesty, (d) Cesta nalezená lokální metodou

lokální metoda nalezne jinou cestu (Obr. 3.6 (d)), kterou tvoří posloupnost vrcholů $(s, 1, 2, 3, 4, 3, 2, 7, g)$.

Eukleidovská vzdálenost mezi vrcholy č. 3 a g je menší než vzdálenost mezi vrcholy č. 5 a g, proto chodec zvolí cestu zobrazenou na Obr. 3.6 (d).

Jsme si plně vědomi toho, že tato metoda neposkytuje vždy optimální cestu. Nicméně to po ní ani nepožadujeme. Tato metoda byla sestrojena za účelem nejrychlejší opravy cesty. Daní za předpoklad rychlosti je ztráta korektnosti cesty v některých případech. Zároveň jsme se pokusili metodu upravit tak, abychom zamezili podobným problémům, jejichž příklad byl uveden výše a ilustrován na Obr. 3.6.

Metoda lokálního maxima

První upravenou verzí je metoda, která využívá maximální vzdálenosti mezi sousedními vrcholy. Nejprve projdeme všechny vrcholy $neigh$ sousedící s vrcholem $ped.start$, ve kterém se nachází chodec. Hledáme takový sousední vrchol $neigh$, jehož vzdálenost od vrcholu $ped.start$ je maximální. Tuto vzdálenost $d(neigh, ped.start)$ si uložíme do proměnné max.

Dalším krokem algoritmu je opět výpočet odhadů vzdálenosti mezi současnou po-

zicí chodce *ped.start* a jeho sousedními vrcholy *neigh*. Z těchto odhadů $h(neigh, ped.end)$ vybereme ten, který je nejmenší. To opakujeme tak dlouho, dokud nenalezneme pozici *ped.end* nebo pokud nejsme příliš daleko od této pozice. Je tedy zřejmé, že jádro metody zůstává stejné. Rozdíl je, že nepoužíváme přesnou pozici sousedních vrcholů *neigh*, ale jejich upravenou pozici vzhledem k maximální vzdálenosti *max* (Alg. 9, řádek 14).

Přesněji to znamená následující věc. Mohou nastat dva případy. Prvním je, že vzdálenost současné pozice *ped.start* a sousedního vrcholu *neigh* je maximální možná, tedy $d(ped.start, neigh) = max$. V tomto případě přecházíme přímo k odhadu vzdálenosti $h(neigh, ped.end)$. Druhou a zajímavější možností je případ, kdy vzdálenost $d(ped.start, neigh) < max$. V takovém případě vypočítáme novou pozici $neigh_{new}$, která leží ve směru vrcholu *neigh* a pro jejíž vzdálenost platí $d(ped.start, neigh_{new}) = max$. Z této nové pozice vypočteme odhad $h(neigh_{new}, ped.end)$.

Algoritmus 9 Vyhledávací metoda lokálního maxima

Require: Pedestrian *ped*

1: **procedure** MAX(node *pos*, graph G)
2: $max \leftarrow -\infty$
3: **for all** neighbours of $pos \in G$ **do**
4: **if** $h(neigh, pos) > max$ **then** $max \leftarrow h(neigh, pos)$
5: **return** *max*
6: **procedure** LOCALPATH(Pedestrian *ped*)
7: $pos \leftarrow ped.start$, $dist \leftarrow \infty$
8: $max \leftarrow \text{MAX}(pos, ped.graph)$
9: **while** *ped.end* not found **do**
10: **for all** neighbours of $pos \in G$ **do**
11: compute new *neigh* position with *max* respect
12: **if** $h(neigh, ped.end) < dist$ **then** $dist \leftarrow h(neigh, ped.end)$
13: Add *neigh* with minimal *dist* to *ped.path*
14: $pos \leftarrow neigh$
15: **if** $h(neigh, ped.end)$ is large **then** compute path with global method

Metoda lokálního minima

Myšlenka druhé verze je téměř totožná jako v případě metody lokálního maxima. Rozdíl tkví v tom, že nepoužíváme maximální vzdálenosti mezi sousedními vrcholy, ale vzdálenost minimální. Nejprve projdeme všechny vrcholy *neigh* sousedící s vrcholem *ped.start*, ve kterém se nachází chodec. Hledáme takový sousední

uzel *neigh*, jehož vzdálenost od vrcholu *ped.start* je minimální. Tuto vzdálenost $d(neigh, ped.start)$ si uložíme do proměnné *min*. To je prvním rozdílem mezi algoritmem lokálního maxima (Alg. 9) a algoritmem lokálního minima (Alg. 10).

Analogicky k metodě lokálního maxima používáme odhad vzdálenosti nikoliv z pozice sousedního vrcholu *neigh*, ale z upravené pozice $neigh_{new}$ ve směru vrcholu *neigh* (Alg. 10). Mohou nastat dva případy. Zaprvé vzdálenost $d(ped.start, neigh) = min$, metoda nic neupravuje a odhadne vzdálenost $h(neigh, ped.end)$ ze sousedního vrcholu *neigh*. Druhým případem je $d(ped.start, neigh) > min$. Zde je nutné vypočítat novou pozici $neigh_{new}$, ze které provedeme odhad. Tato pozice leží na hraně spojující vrcholy *ped.start* a *neigh* a její vzdálenost je $d(ped.start, neigh_{new}) = min$.

Algoritmus 10 Vyhledávací metoda lokálního minima

Require: Pedestrian *ped*

1: **procedure** MIN(node *pos*, graph G)
2: $min \leftarrow \infty$
3: **for all** neighbours of $pos \in G$ **do**
4: **if** $h(neigh, pos) > max$ **then** $min \leftarrow h(neigh, pos)$
5: **return** *min*
6: **procedure** LOCALPATH(Pedestrian *ped*)
7: $pos \leftarrow ped.start$, $dist \leftarrow \infty$
8: $min \leftarrow \text{MIN}(pos, ped.graph)$
9: **while** *ped.end* not found **do**
10: **for all** neighbours of $pos \in G$ **do**
11: compute new *neigh* position with *min* respect
12: **if** $h(neigh, ped.end) < dist$ **then** $dist \leftarrow h(neigh, ped.end)$
13: Add *neigh* with minimal *dist* to *ped.path*
14: $pos \leftarrow neigh$
15: **if** $h(neigh, ped.end)$ is large **then** compute path with global method

Na závěr poznamenejme jednu důležitou věc. Ačkoliv v metodách lokálního maxima a lokálního minima počítáme odhad $h(neigh_{new}, ped.end)$ z nově spočtené pozice $neigh_{new}$ (ve směru sousedního vrcholu *neigh*) do vrcholu *ped.end*, vždy necháme chodce dojít právě do sousedního vrcholu *neigh*. To děláme i přesto, že nově spočtená pozice $neigh_{new}$ se může nacházet před sousedním vrcholem *neigh* (lokální minimum), resp. za sousedním vrcholem *neigh* (lokální maximum).

Skupinky chodců

Dalším řešeným problémem jsou tzv. skupinky chodců. Představme si situaci, kdy skupinka kamarádů jde do kina nebo školní třída jde na exkurzi. V takovém případě není nutné, aby každý chodec z této skupinky měl vlastní cestu, protože vždy následují jedinou osobu anebo je jejich cesta naprosto stejná.

Motivací tohoto problému jsou již výše zmíněné skupinky chodců a hlavně úspora jak výpočtové paměti, tak výpočtového času. Naší snahou je pro n chodců putujících z počátečního vrcholu s_{start} do koncového vrcholu s_{goal} v grafu G spočítat tuto cestu pouze prvnímu chodci n_0 a přiřadit ji všem ostatním $(n-1)$ chodcům.

Představme si, že tento problém řešíme pro skupinku sta chodců $(n = 100)$. Když spočteme cestu pouze jednou a přiřadíme jí ostatním, hned ve výpočtu ušetříme rychlost výpočtu o 99 dalších spuštění A* algoritmu či D* Lite algoritmu a v případě D* Lite algoritmu ušetříme dokonce velké množství paměti. Není totiž nutné každému chodci vytvářet kopii grafu, nýbrž jednou ze skupinky sta chodců. Tím ušetříme paměť nutnou pro 99 kopií grafu a to už je velká paměťová úspora.

Tento přístup nechceme užít pouze pro skupinky chodců, jejichž počáteční uzly s_{start} a koncové uzly s_{goal} jsou totožné, ale snažíme se ho rozšířit. Rozšířením myslíme, že bereme v potaz skupinku chodců, jejichž počáteční vrchol s_{start} vrchol leží v kružnici o poloměru ϵ a jejichž cílový vrchol s_{goal} vrchol leží v jiné kružnici se stejným poloměrem ϵ.

Shlukování

Určovat skupinky by šlo hrubou silou, ale ve výsledném měření by pravděpodobně převážila situace, kdy by nemělo tento přístup skupinek smysl využívat. Tedy čas potřebný k roztřídění chodců hrubou silou by byl příliš velký. Pro tento problém nás napadla možnost využít problematiky shlukování bodů. Kvůli nedostatku času a poměrné složitosti problematiky shlukování jsme upustili od vlastní implementace a požádali pana Bc. Ondřeje Kaase, který tento problém řešil ve své bakalářské práci [10], o jeho knihovnu, která tuto problematiku řeší.

Knihovna pana Bc. Kaase je naprogramovaná v jazyce C# a náš program je naprogramovaný v jazyce C++. Existuje jednoduché propojení C# knihovny s programem psaným v C++, ale daná knihovna musí splňovat speciální požadavky. Tyto požadavky zde nebudeme vypisovat, pouze zmíníme, že je bohužel knihovna pro shlukování bodů nesplňuje. Druhou možností je propojit knihovny pomocí několika nastavování parametrů a implementováním propojovacího kódu, který je ale časově velice náročný. Rozhodli jsme se proto propojení knihovny shlukování s naším kódem provést trochu rychleji a ne vnitřně.

Vytvořili jsme program v jazyce C#, který využívá knihovnu shlukování bodů a vypočtené shlukování uloží do souboru (Alg. 11). Nejprve spustíme náš program a načteme strukturu města a data chodců. Jakmile se načtou data chodců, uloží se jejich pozice počátečního vrcholu a pozice koncového vrcholu do textového souboru *input.txt* (Alg. 11, řádky 6 a 7) a následně je spuštěn program psaný v jazyce C# (Alg. 11, řádek 8). Ten textový soubor načte, spočte shlukování bodů (Alg. 11, řádky 10-12), které následně uloží do nového textového souboru *output.xml* (Alg. 11, řádky 14-15), a ukončí se. Po ukončení programu v C# je možné načíst data ze souboru *output.xml* naším programem v C++, který vytvoří skupinky chodců. Pro ně můžeme následně spustit hlavní smyčku programu.

Algoritmus 11 Shlukování bodů reprezentující chodce

```
Require: Pedestrians peds
 1: procedure CLUSTERS(pedestrians peds, file txt)
 2:     txt ← peds.size                        ▷ Ulož počet chodců do textového souboru
 3:     txt ← clustreing_parameter             ▷ Ulož shlukovací parametr do souboru
 4:     for all pedestrians peds do
 5:         txt ← peds.id                      ▷ Ulož id chodce do souboru
 6:         txt ← peds.startPosition           ▷ Ulož pozici počátečního vrcholu do souboru
 7:         txt ← peds.endPosition             ▷ Ulož pozici koncového vrcholu do souboru
 8:     execute externProgram                  ▷ Spusť program pro shlukování
 9: procedure EXTERNPROGRAM                    ▷ Program pro shlukování
10:     load file txt                          ▷ Načti textový soubor s daty chodců
11:     create array peds of pedestrians       ▷ Ulož data chodců do pole
12:     compute clusters clus from peds        ▷ Spočti shlukování pro pozice chodců
13:     for all clusters c from clus do        ▷ Pro každý shluk
14:         save c.center to xml               ▷ Ulož chodce ve středu shluku do souboru .xml
15:         save c.others to xml               ▷ Ulož ostatní chodce do souboru .xml
```

Je nutné poznamenat, že tento způsob je funkční pouze pod operačním systémem Windows. Ke spuštění externího programu používáme knihovnu specifickou pouze pro tento operační systém, což zároveň vylučuje plnou funkčnost pod jinými operačními systémy. Zároveň jsme si vědomi, že to není korektní propojení knihovny s programem. V budoucnu bude tento oslí můstek odstraněn, ale pro naše současné testování je to postačující.

Společná cesta

Jakmile je výpočet shlukování hotov, můžeme načíst soubor *output.xml*, který obsahuje shluky chodců. Každý shluk v tomto souboru obsahuje nejprve střed shluku,

tedy chodce *ped* ve středu shluku, a následně všechny ostatní chodce patřící do této skupiny. Načtením tohoto souboru se přepne jádro našeho programu na výpočet pomocí skupinek chodců. Výpočet probíhá pro každý shluk bodů, tedy skupinku chodců (Alg. 12). Nejprve spočteme nejkratší cestu chodci *center*, který byl označen jako střed shluku. Pro ostatní chodce *ped* nejprve vypočteme cestu *path* z vrcholu *ped.start* do vrcholu *center.start*. Dále nakopírujeme na konec cesty *path* nejkratší cestu spočtenou pro chodce *center*. Na závěr pak na konec cesty *path* přidáme nejkratší cestu z vrcholu *center.end* do vrcholu *ped.end*. Tímto způsobem počítáme cestu pro všechny shluky.

Algoritmus 12 Výpočet cesty pro skupinky chodců

Require: Clusters *clust*
1: **for all** cluster *c* from clusters *clust* **do**
2: compute shortest path for *c.center*
3: **for all** others pedestrians *ped* from cluster *c* **do**
4: *ped.path* ← shortest path from *ped.start* to *c.center*
5: *ped.path*.add(*c.center.path*)
6: *ped.path*.add(shortest path from *c.end* to *ped.end*)

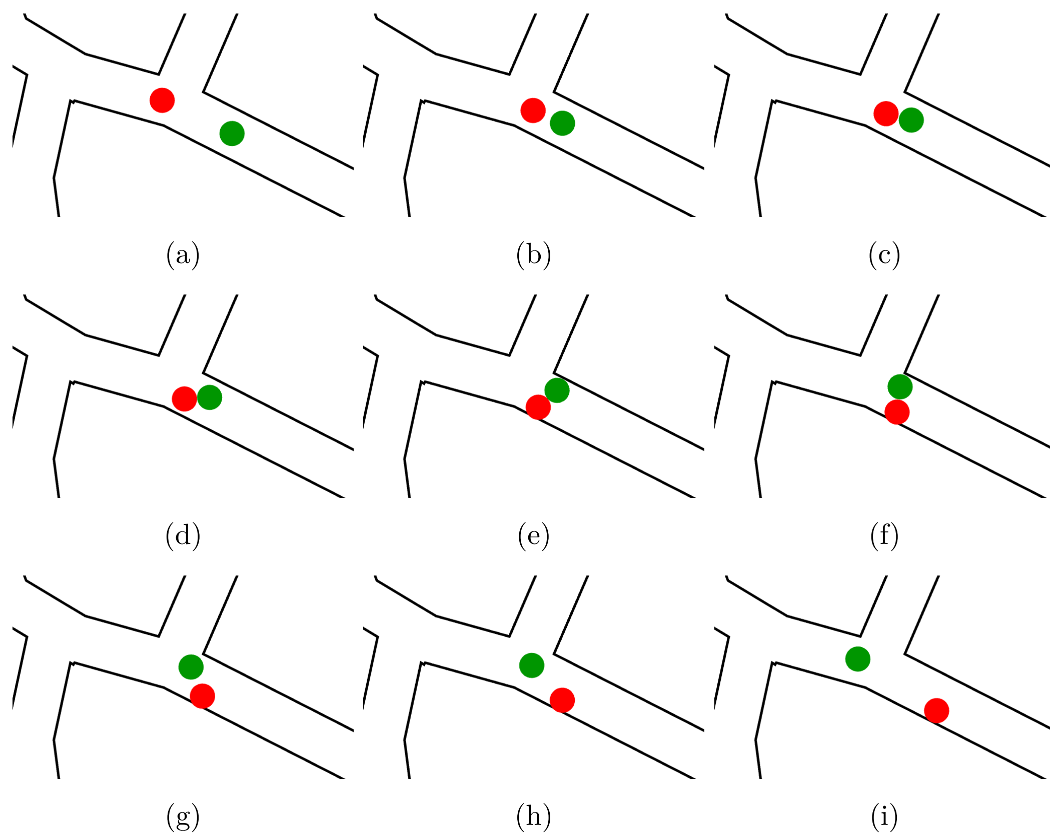
Kolize chodců

Mezi řešitelem této diplomové práce a diplomové práce [2] Ing. Pavla Brandejského proběhla v roce 2014 spolupráce. Kolega Brandejský řešil kolize chodců mezi sebou. Prošel existující metody, které danou problematiku řeší. Mezi ně patří lokální metody pro řešení kolizí mezi chodci, globální metody či metody, které se neorientují pouze na jedince. Z nich následně zvolil nejvhodnější metodu, která řeší kolize mezi chodci lokálně.

Jádrem spolupráce bylo z naší strany poskytnutí některých testovacích dat městské struktury, na kterých by mohl řešit lokální kolizi chodců. Na oplátku nám kolega Brandejský poskytl knihovnu na řešení lokálních kolizí chodců, kterou jsme mohli zakomponovat do naší práce, aby pohyb chodců vypadal o něco věrohodněji.

Na obrázku 3.7 je vyexportován příklad detekování kolize dvou chodců v našem programu. Červený chodec se pohybuje zleva doprava a zelený chodec se pohybuje směrem opačným. Časová posloupnost jejich pohybu je seřazena vzestupným abecedním označením, tedy Obr. 3.7 (a) je počáteční stav a Obr. 3.7 (i) je koncový stav.

Je možné si všimnout, že od obrázku 3.7 (e) dále červený chodec přesahuje oblast ohraničující průchozí cestu. To je zaprvé způsobeno tím, že neřešíme kolize chodců



Obr. 3.7: Pohyb dvou chodců a jejich vzájemná kolize

s okolními objekty. Zadruhé, použitá data neposkytují přesnou šířku silnic, chodníků a stezek a v neposlední řadě je průměr vizualizovaného chodce polovinou šířky silnice, což není realisticky korektní. Nicméně toto řešení velmi usnadňuje optické porovnávání pohybu a cest chodců.

4 EXPERIMENTY

Celý program byl vytvořen a testován na notebooku Lenovo IdeaPad Y510p, jehož sestava je následující:

Procesor	Intel® Core™ i5-4200M (3M Cache, 2.50 GHz)
Operační paměť	8GB DDR3 1600MHz
Grafická karta	nVidia Geforce GT755M DDR5 2GB
Operační systém	Windows 7 Professional 64 bit

Před konzultací výsledků programu hledání cest je důležité zmínit, jaké jsou vhodné prvky testování, mezi něž patří

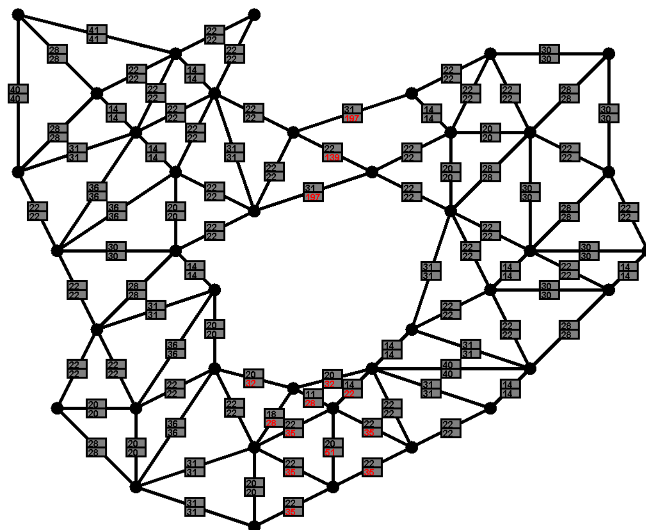
- *Doba předzpracování* - doba inicializace programu před jeho hlavní smyčkou. V našem případě se tedy jedná o načtení dat struktury města, vlastností chodců anebo připravení dat pro metodu vyhledávání cest.
- *Doba nalezení cesty* - doba, za kterou je program schopen najít cestu z počátečního bodu s_{start} do cílového bodu s_{goal}. Toto je jedna z nejdůležitějších charakteristik programu, protože vyhledávání nové cesty může v každé iteraci hlavní smyčky programu proběhnout několikrát.
- *Korektnost cesty* - rozdíl nalezené cesty od cesty optimální, tedy chyba způsobená použitím metody. Jedná se o důležitý faktor testování, který bude korelovat s výsledky doby nalezení cesty. Příkladem může být rychleji nalezená cesta, která ale nemusí být minimální možná.

Hledání cest v budovách

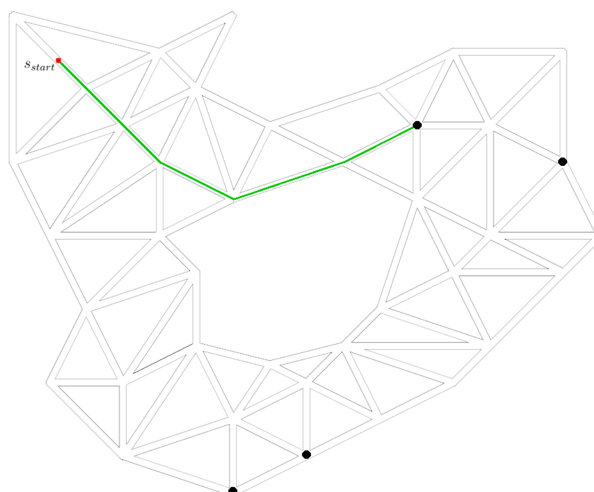
Než se pustíme do testování našeho programu a používaných metod, ukažme si možné využití našeho programu. Pan Patrik Kořínek se ve své bakalářské práci, která je vytvářena současně s touto diplomovou prací, zaměřuje na vytváření unikátních ohodnocení interních prostor každému chodci. Toto hranové ohodnocení neorientovaného grafu G je počítáno pomocí několika atributů chodce, např. zdraví, mobilita, rychlost chodce a další.

Pan Kořínek nám poskytl některá specifická ohodnocení grafu. Uveďme si zde alespoň jeden příklad, jak takové ohodnocení a průběh simulace může vypadat. Počáteční cesta chodce je spočtena na původním grafu, jehož ohodnocení je vidět na obrázku 4.1, který byl převzat od pana Kořínka. Každá ohodnocená hrana má dvě hodnoty. První hodnota je počáteční hodnota grafu a druhá hodnota indikuje změnu hranového ohodnocení v čase $t > 0$. Nalezená minimální cesta chodce v čase $t_0 = 0$ je na obrázku 4.2, kde je chodec reprezentován červeným bodem, možné východy z budovy jsou reprezentovány černými body a nalezená cesta je vykreslena zeleně.

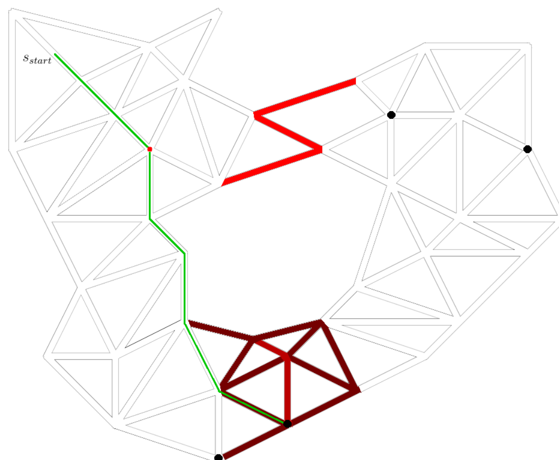
Chodec v čase $t_1 > t_0$ zjistí, že v budově začalo hořet, a proto změní svojí cestu (Obr. 4.3) a snaží se dojít k jinému východu, ke kterému není tolik nebezpečné dojít. Změna hranového ohodnocení grafu je vidět na obrázku 4.1, na obrázku 4.3 je změna ohodnocení znázorněna červenými odstíny. Čím je hranové ohodnocení vyšší, tím světlejší červená barva ho reprezentuje. V čase $t_2 > t_1$ chodec zjistí, že se zřítil strop a je nutné trasu opět změnit, protože již není možné dojít k původnímu východu (Obr. 4.4). Neexistující hrany jsou na obrázku 4.3 vyznačeny šedivou barvou.



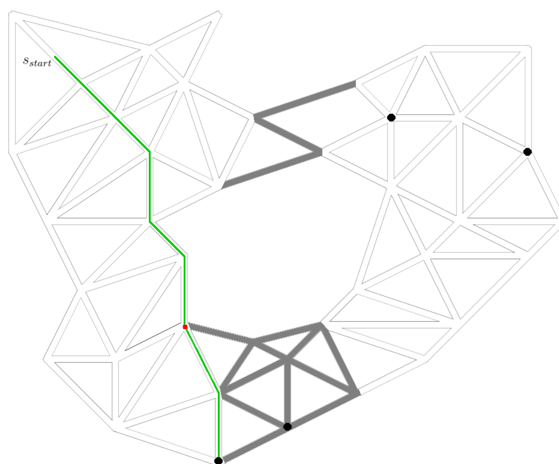
Obr. 4.1: Ohodnocený graf reprezentující interiér budovy.



Obr. 4.2: Počáteční cesta nalezená v interiéru budovy.



Obr. 4.3: První přeplánování cesty z důvodu ohně.



Obr. 4.4: Druhé přeplánování trasy z důvodu zříceného stropu.

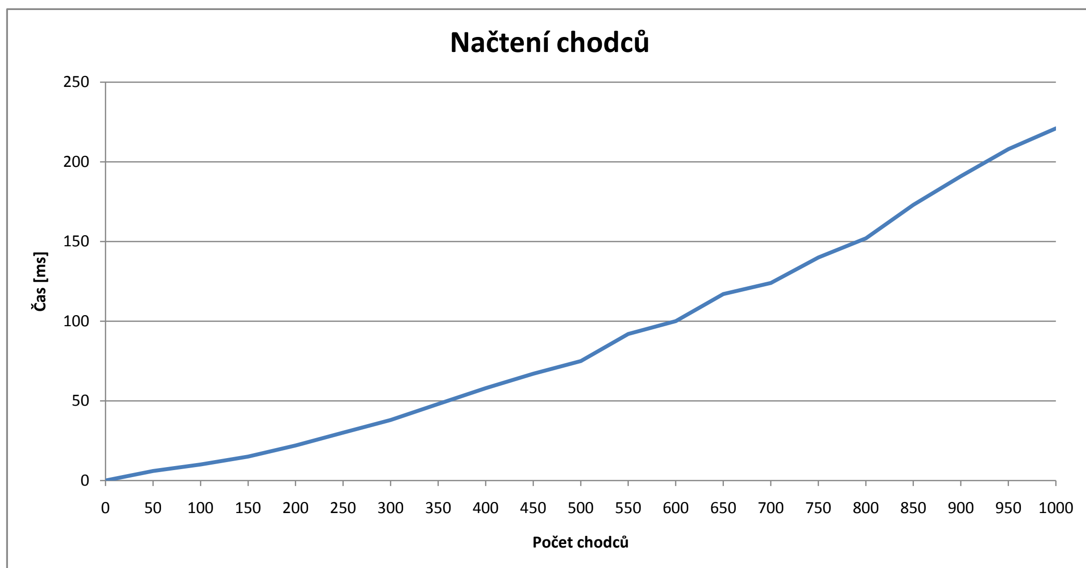
Načtení dat

Implementovaný program načítá právě dva soubory. Prvním souborem, který načítáme, je struktura virtuálního města. Z tohoto souboru načteme data, alokujeme paměť a vhodně data reprezentujeme (v našem případě je třeba vytvořit neorientovaný graf G). V tabulce 4.1 si můžeme všimnout, že s rostoucím počtem uzlů roste časová náročnost vytváření programové reprezentace dat, tedy neorientovaného grafu G, což je předpokládaný výsledek. Nicméně si také můžeme všimnout, že větší počet hran pro stejný počet uzlů nemusí nutně znamenat větší časovou náročnost. Přesný důvod této zvláštnosti nám není úplně znám, protože k reprezentaci neorientovaného grafu G využíváme veřejně dostupnou knihovnu boost [20]. Poslední sloupec tabulky 4.1 informuje o tom, zda načtená data obsahují kromě silnic taktéž cesty přístupné pouze chodcům (např. chodník, park, apod.).

Soubor města	Počet uzlů	Počet hran	Čas [ms]	Pěší zóny
Rokycany	1185	1312	575	ne
Rokycany	1185	1335	585	ano
PilsenC	1914	2190	949	ne
PilsenC	1914	2268	901	ano
PilsenCS	2359	2691	1350	ne
PilsenCS	2359	2800	1191	ano
PilsenCSE	3369	3797	2017	ne
PilsenCSE	3369	3944	2039	ano
PilsenCSEW	5437	5977	4312	ne
PilsenCSEW	5437	6193	4587	ano
Pilsen	6702	7315	5676	ne
Pilsen	6702	7571	5896	ano

Tab. 4.1: Načítání dat struktury města

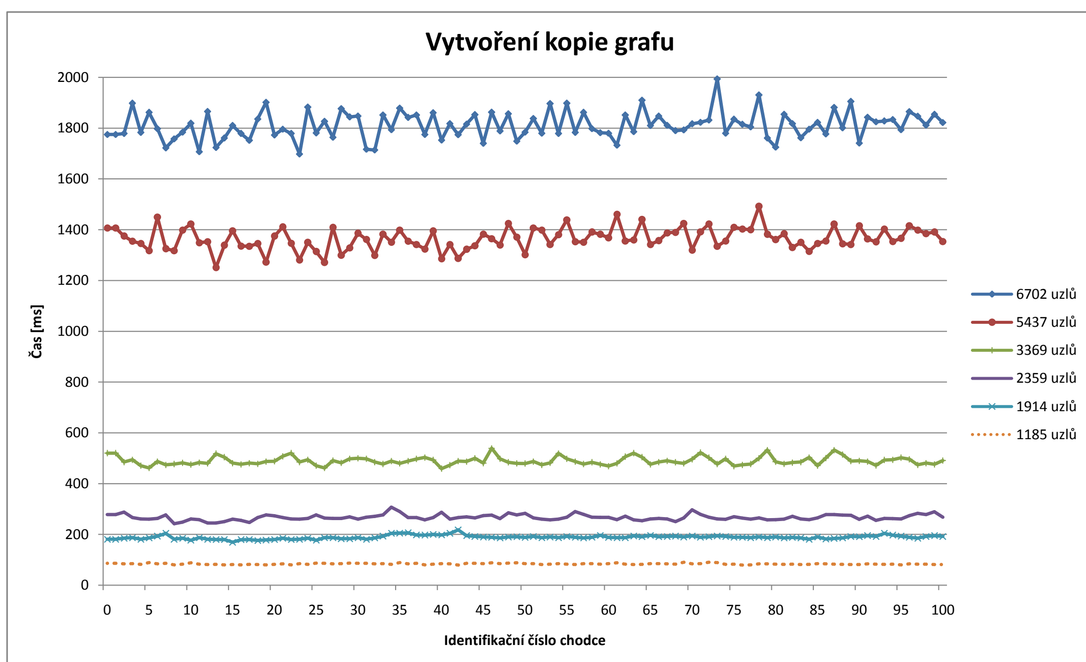
Druhým načítaným souborem je soubor s vlastnostmi chodců, tedy s jejich počáteční pozicí s_{start}, pozicí cílovou s_{goal}, rychlostí a také časem simulace. Ten slouží jako indikátor, kdy se začne chodec pohybovat k cíli. Na obrázku 4.5 je vidět časové zatížení programu načítáním dat chodců, např. načtení tisíce chodců netrvá ani čtvrt vteřiny.



Obr. 4.5: Závislost rychlosti načtení dat chodců na jejich rozměru.

Předzpracování

Ačkoliv předzpracování dat nebo také tzv. *preprocessing* není hlavním faktorem testování programu, stále se jedná o velice důležitou část experimentů k pokrytí největších časových intervalů výpočtu. Do této části spadají dva důležité problémy. Prvním je vytváření kopií neorientovaného grafu G. Předpokládáme, že se každý chodec pohybuje po stejně ohodnoceném neorientovaném grafu G. Z kapitoly 2 víme, že každý výpočet cesty D* Lite algoritmem vytváří unikátní ohodnocení vrcholů s grafu G. Musíme tedy vytvořit každému chodci vlastní kopii grafu G, ve které D* Lite najde nejkratší cestu. Časová náročnost tohoto kopírování je vidět v grafu na obrázku 4.6.

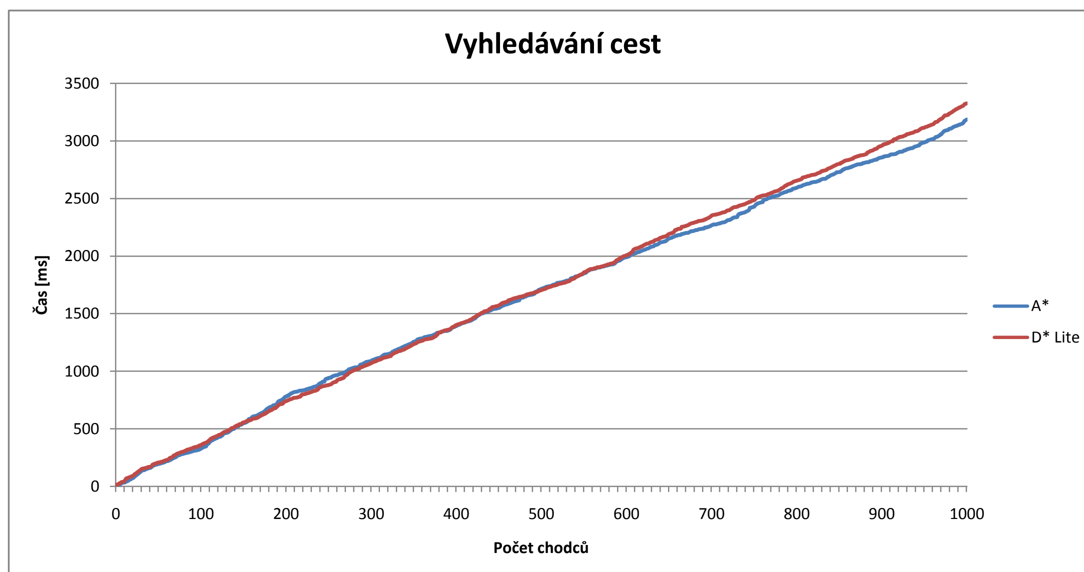


Obr. 4.6: Rychlost kopírování dat struktury neorientovaného grafu.

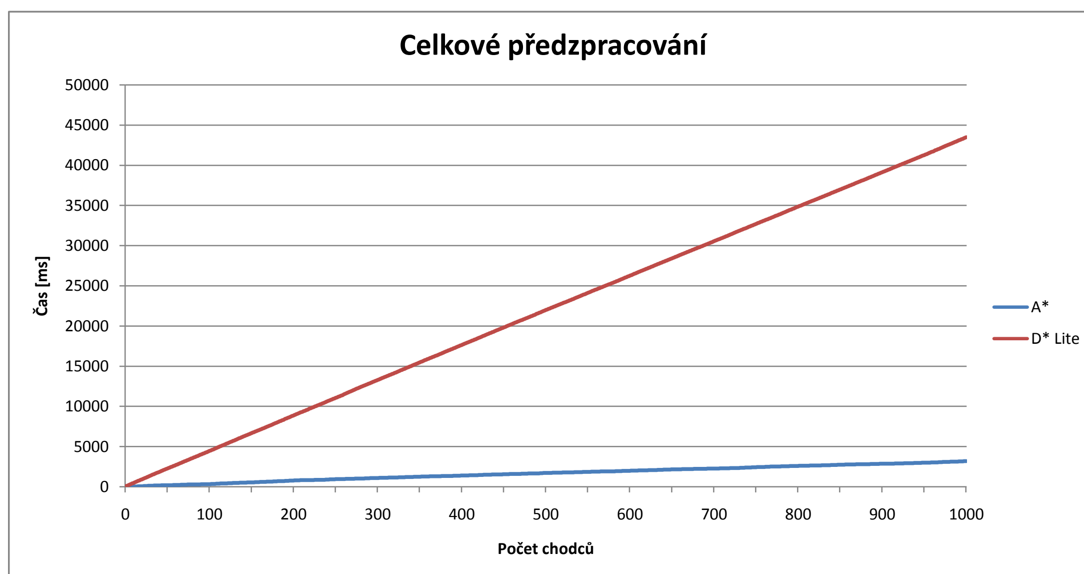
Uvědomme si, že toto předzpracování (vytváření kopií grafu G) není nutné provádět pro A* algoritmus. Důvodem je předpoklad, že každý chodec se pohybuje po stejně ohodnoceném grafu G, a samotný A* algoritmus, který nevytváří specifické ohodnocení grafu.

Pokud bychom předpokládali, že se každý chodec pohybuje po grafu G s unikátním ohodnocením, bylo by nutné vytvořit každému chodci vlastní reprezentaci grafu G. Tím by odpadla nutnost vytváření kopií grafu G a jejich časová složitost (Obr. 4.6) by nahradila časová složitost načítání dat a reprezentace struktury virtuálního města (Tab. 4.1). Zároveň by se tím smyla výhoda A* algoritmu oproti D* Lite algoritmu.

Druhým a posledním problémem předzpracování před simulací je výpočet počáteční cesty. Ta je počítána buď A* algoritmem, nebo D* Lite algoritmem. Rozdíl v rychlosti nalezení nejkratší cesty v grafu G těchto dvou metod je minimální (Obr. 4.7).



Obr. 4.7: Porovnání výpočtové rychlosti algoritmů A* a D* Lite pro ze souboru *Rokycany.xml*.



Obr. 4.8: Celé předzpracování pro 1000 chodců pro graf ze souboru *Rokycany.xml*.

Nyní již zbývá porovnat časy celého předzpracování. Připomeňme si, že předpokládáme pohyb každého chodce po totožném grafu G. Na obrázku 4.8 je vidět

časová náročnost celého předzpracování, tedy vytváření kopií včetně výpočtu nejkratší cesty. Toto porovnání bylo vytvořeno pro soubor *Rokycany.xml* (Tab. 4.1) a je zřejmé, že v případě rozměrnějších dat by výpočet předzpracování algoritmem D* Lite bylo časově náročnější.

Vyhledávací algoritmy

V momentě, kdy se program dostane do hlavní smyčky a chodec se začne pohybovat po spočtené cestě, přestáváme uvažovat statické prostředí a začínáme používat prostředí dynamické. Hrany neorientovaného grafu mohou zanikat, vznikat či pouze měnit svoje ohodnocení. Zánik či odstranění hrany z grafu můžeme chápat jako uzavření mostu kvůli renovaci nebo uzavření silnice z důvodu nebezpečí (např. dopravní nehoda či dokonce teroristický útok). Naopak vznik hrany může představovat otevření nově postaveného mostu, tunelu nebo jiné cesty a nebo odstranění následků dopravní nehody. Poslední možností je změna ohodnocení hrany např. v závislosti na výskytu počtu lidí. Pokud se v daném úseku tvoří davy, bude ohodnocení hrany vyšší než běžně a naopak, pokud zde nebude téměř žádný chodec, tak ohodnocení hrany snížíme.

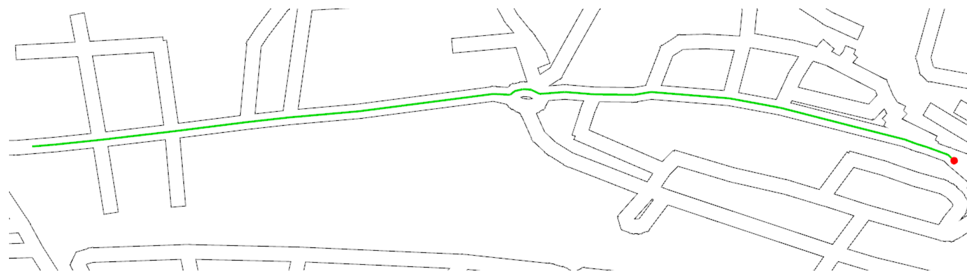
Na tyto grafové změny je nutné vhodně reagovat použitím globálních či lokálních metod pro opravu cesty, a proto se v této podkapitole budeme zabývat testováním globálních a lokálních metod. Zajímá nás hlavně doba výpočtu nejkratší cesty, její oprava v případě změny grafu G a korektnost výsledné cesty. Nejprve se budeme zajímat o rozdíly mezi globálními algoritmy, tedy A* algoritmem a D* algoritmem. Porovnáme jejich nalezené cesty, rychlost výpočtu a rychlost opravení cesty. Následně se zaměříme na porovnání lokálních metod, které budeme testovat podobně jako metody globální, mezi sebou a na závěr i porovnání lokálních metod s metodami globálními, kde nás bude hlavně zajímat ušetřený čas a chyba způsobená lokálními metodami.

Globální přepočet cesty

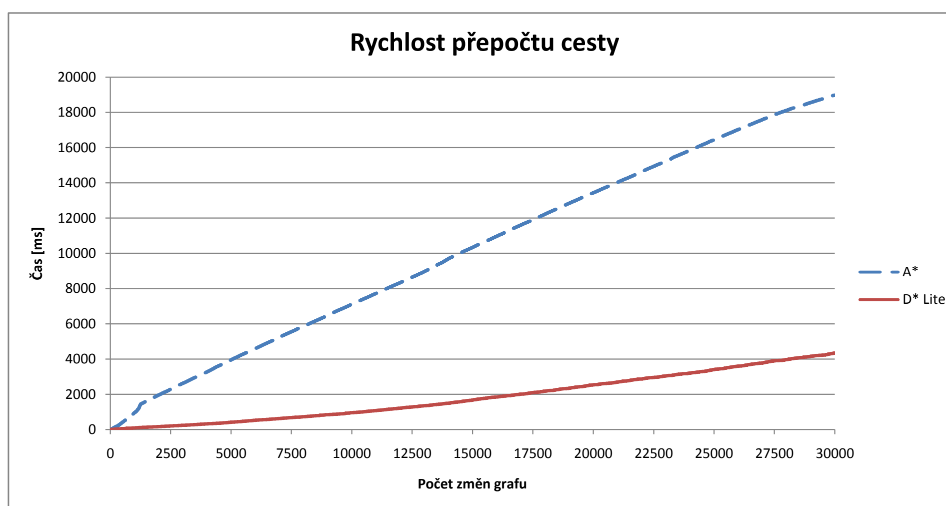
Uveďme si rovnou, že se výsledky globálních metod vždy shodují. Nalezená nejkratší cesta pomocí A* algoritmu je totožná s nejkratší cestou, kterou nalezne D* Lite algoritmus. Není proto nutné porovnávat rozdíl nalezených cest. Na obrázku 4.9 je příklad cesty nalezené globálními algoritmy. Červený bod představuje chodce na počáteční pozici s_{start} a zelená křivka představuje nalezenou nejkratší cestu, po níž se bude chodec pohybovat.

Dalším faktorem, který bychom měli posoudit, je přepočet cesty v případě změny ohodnocení grafu. Předpokládáme, že D* Lite bude přepočítávat cestu rychleji než

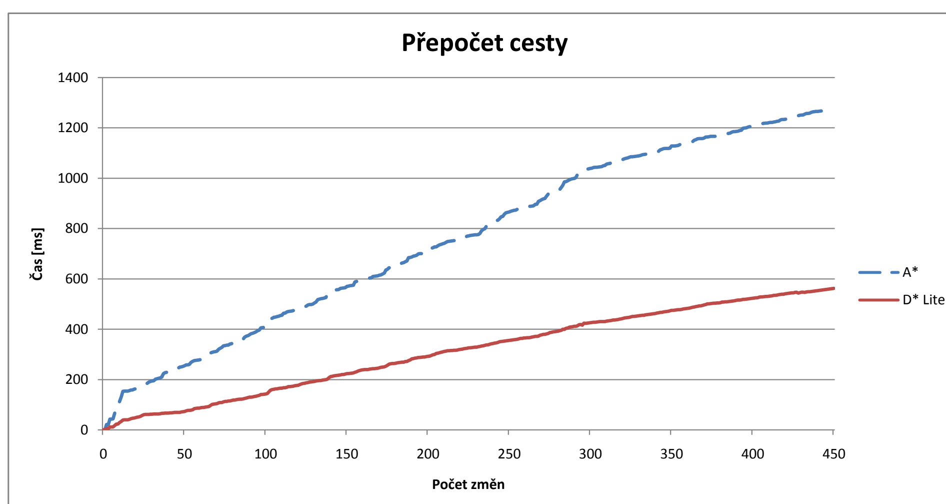
A*, protože k tomu byl primárně určen. Tento předpoklad nám potvrzuje i měření na obrázcích 4.10 a 4.11.



Obr. 4.9: Totožná cesta nalezená pomocí A* a D* Lite algoritmů.



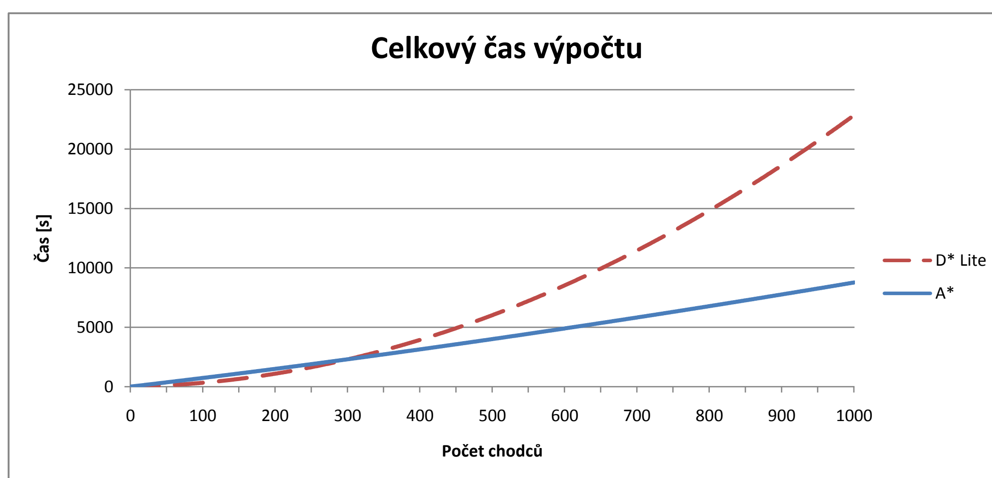
Obr. 4.10: Časová náročnost výpočtu změn pro jednoho chodce ve známém prostředí.



Obr. 4.11: Rychlost výpočtu změn pro tisíc chodců ve částečně známém prostředí.

V prvním zmíněném grafu (Obr. 4.10) je změřen čas přepočítání cesty jednoho chodce při neustálých změnách grafu G při známém prostředí. V druhém grafu (Obr. 4.11) je zanesena časová náročnost přepočtení změn pro 1000 chodců v prostředí částečně známém. V tomto případě přepočteme cestu pouze v případě, že na naplánované cestě narazíme na změnu v ohodnocení grafu G.

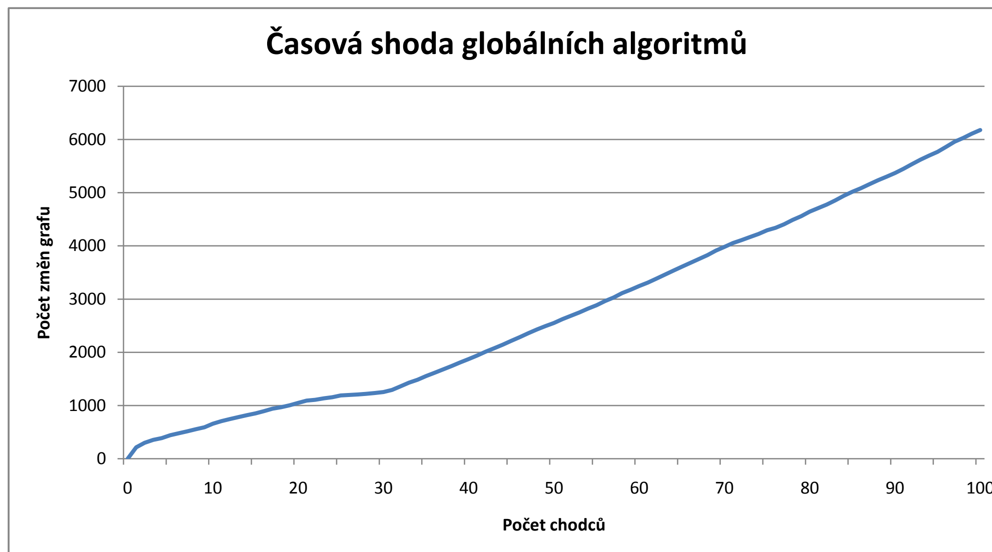
Představme si simulaci pohybu až tisíce chodců po nejmenším testovaném virtuálním městě (*Rokycany.xml*). Každý chodec navíc bude muset svoji cestu přepočítat vždy, když dojde v grafu G ke změně. Nastane až 20000 změn v grafu, tedy každý chodec bude nucen k 20000 přepočtům své cesty. V takovém případě je vidět, že D* Lite algoritmus je rychlejší pouze pro prvních 300 chodců, následně je opět předehnán A* algoritmem, který je pro tisíc chodců mnohem rychlejší (Obr. 4.12).



Obr. 4.12: Celkový čas výpočtu pro 1000 chodců a 20000 přepočtů cesty pro každého chodce.

Zajímá nás hlavně, kdy je lepší použít A* algoritmus oproti D* Lite algoritmu a naopak. V grafu (Obr. 4.13) znázorněno pro jaké parametry se celkový výpočet A* a D* Lite algoritmů shoduje v nejmenším testovaném grafu *Rokycany.xml*. Na vodorovné ose je počet chodců a na ose svislé je počet změn grafu G, tedy počet přepočítání cesty každého chodce. A* algoritmus je vhodný použít pro parametry tvořící oblast pod křivkou (Obr. 4.13), tedy např. pro 50 chodců, jejichž cesta se bude přepočítávat každému z nich stokrát. Naopak pro parametry definující oblast nad křivkou je vhodnější použít D* Lite algoritmus. Tedy např. pro 50 chodců, u nichž dojde k 5000 přepočtům cesty (celkem 250000 přepočtů v průběhu simulace).

Zbývá nám porovnat rychlost výpočtu celé cesty. Tato část experimentu je znázorněna na obrázku 4.7 v kapitole 4. Z něj je patrné, jak jsme již dříve zmínili, že oběma metodám trvá přibližně stejnou dobu najít nejkratší cestu. Nicméně je to závěr, který jsme očekávali, protože D* Lite ve svém jádru obsahuje myšlenku A*



Obr. 4.13: Shodná časová náročnost A* a D* Lite algoritmů pro graf *Rokycany.xml*.

algoritmu. Pokud budeme brát v potaz celkový čas, je nutné přidat i čas vytváření kopií neorientovaného grafu G v D* Lite algoritmu. V tomto případě je zřejmé, že výpočet D* Lite algoritmu trvá déle než výpočet A* algoritmu (4.8 v kapitole 4).

Než přejdeme k výsledkům lokálních metod, uveďme si ještě příklad rozdílu nalezené cesty ve známém prostředí a v prostředí částečně známém. Naplánujeme počáteční cestu chodci z počáteční pozice s_{goal} v čase $t_0 = 0$ (Obr. 4.14). Chodec (Obr. 4.14, červený bod) se vydá po své cestě a v čase $t_1 > t_0$ je informován o změně v neorientovaném grafu G, a proto ihned přizpůsobí svojí cestu této změně (Obr.(4.15)). Následně se v čase $t_2 > t_1$ dojde k jiné změně v grafu G a chodec opět přepočte svoji cestu, aby odpovídala cestě minimální (Obr.(4.16)). Až do konce simulace se žádná další změna neobjeví, a tak chodec dojde až do cílového vrcholu s_{goal} svojí cesty bez dalšího přepočítávání.

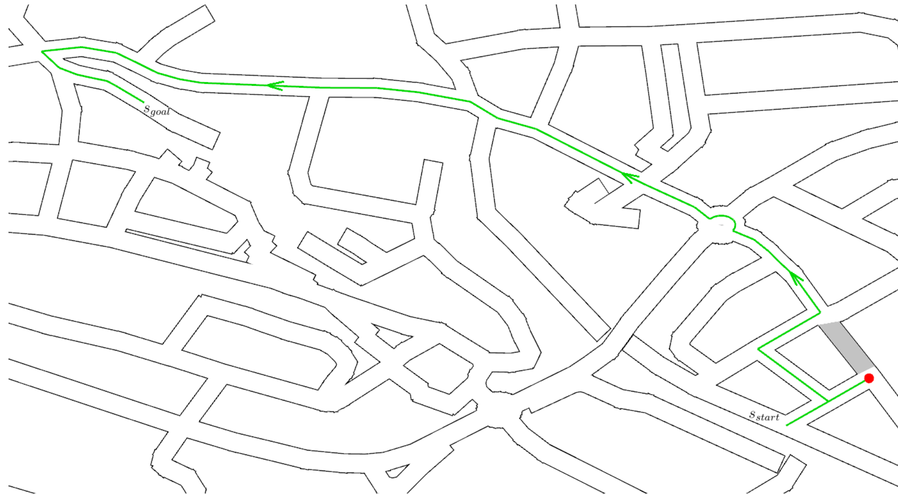
Oproti tomu vyhledání cesty témuž chodci v částečně známém grafu probíhá následovně. Stejně jako v případě známého prostředí nejprve spočteme počáteční cestu v čase $t_0 = 0$. Ta je totožná s počáteční cestou ve známém prostředí (Obr. 4.14). Chodec se vydá po své cestě a v čase $t_1 > t_0$ se změní ohodnocení grafu G. Chodec o tom neví, a proto pokračuje dál v cestě, dokud v čase $t_2 > t_1$ nenarazí na překážku. V tento moment přepočte cestu (Obr. 4.17) a následně pokračuje po aktualizované cestě. V čase $t_3 > t_2$ dojde opět ke změně grafu, o které chodec nemá tušení. Pokračuje tedy dále po naplánované cestě až do času $t_4 > t_3$, kdy objeví překážku na své cestě, a najde si proto novou cestu (Obr. 4.18). Následně pokračuje až do koncového uzlu s_{goal}, protože neobjeví žádné další překážky.

Z výše uvedeného příkladu je zřejmé, že cesta nalezená globálním algoritmem ve známém prostředí je kratší než cesta nalezená v částečně známém prostředí. To

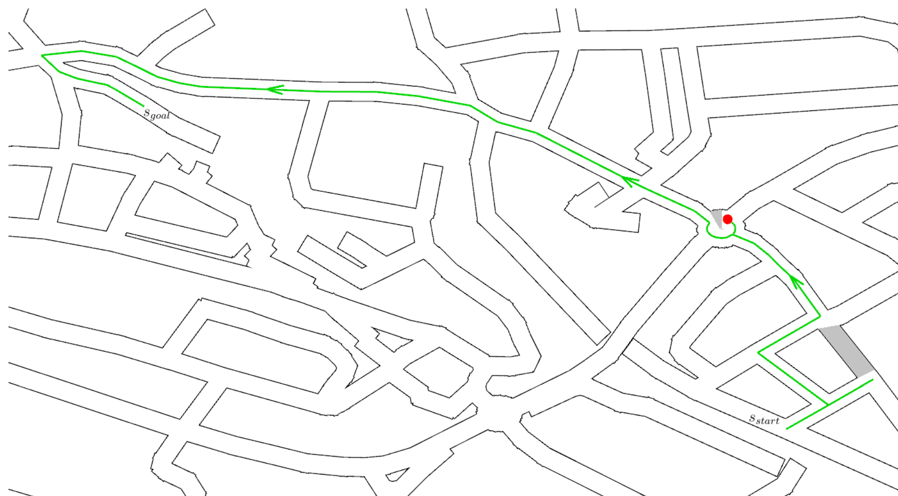
Obr. 4.14: Počáteční cesta spočtená globálním algoritmem.

Obr. 4.15: Přepočtení cesty v době výskytu první změny v grafu G.

Obr. 4.16: Přepočtení cesty v době výskytu další změny v grafu G.

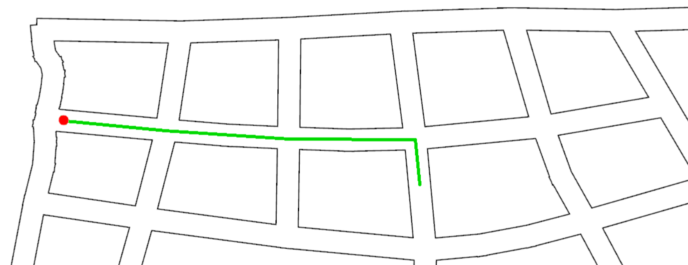


Obr. 4.17: Přepočtení cesty v době výskytu první změny v grafu G.

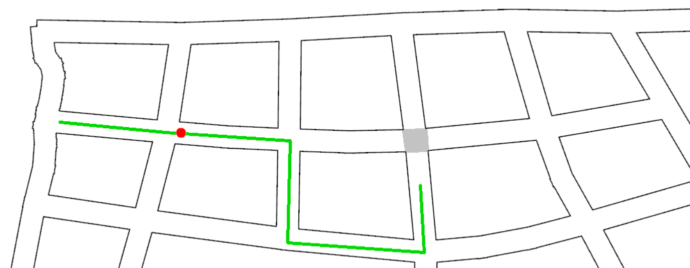


Obr. 4.18: Přepočtení cesty v době výskytu další změny v grafu G.

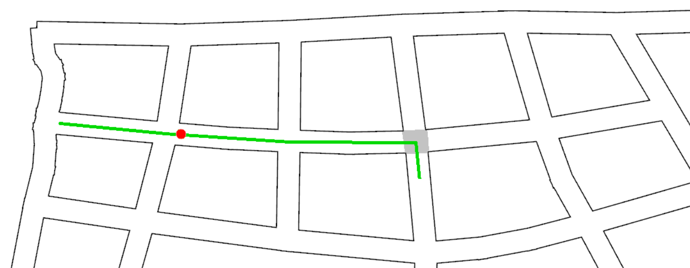
platí pro naprostou většinu nalezených cest, ale výjimečně se může stát, že cesta nalezená v částečně známém prostředí bude kratší než cesta nalezená ve známém prostředí. Tento případ je znázorněn na obrázcích 4.19 až 4.21.



Obr. 4.19: Počáteční cesta chodce.



Obr. 4.20: Přepočtení cesty v době výskytu změny ve známém prostředí.



Obr. 4.21: Pokračování v cestě v době výskytu změny v částečně známém prostředí.

Mohlo by se např. stát, že spočteme počáteční cestu v čase t_0, která je pro obě prostředí totožná (Obr. 4.19). Chodec se následně pohybuje po naplánované cestě a v čase $t_1 > t_0$ dojde na naplánované cestě k změně. Ve známém prostředí dojde k okamžitému přepočítání cesty (Obr. 4.20), kdežto v částečně známém prostředí chodec pokračuje dále, dokud nedojde k překážce (Obr. 4.21). V čase $t_2 > t_1$ (moment těsně před objevením překážky v částečně známém prostředí) dojde ke změně grafu G, kdy tato překážka zmizí a cesta se opět uvolní. Je tedy zřejmé, že cesta prošlá chodcem v částečně známém prostředí je kratší než v prostředí známém.

Shrňme si nyní poznatky o globálních metodách. Je zřejmé, že nalezení počáteční cesty A* algoritmem bude rychlejší než algoritmem D* Lite. Důvodem je, že A* algoritmus nepotřebuje vytvářet kopie neorientovaného grafu G pro každého chodce. Pokud musíme výslednou cestu přepočítat, protože se v grafu G objevila hranová změna, D* Lite tuto původní cestu aktualizuje rychleji. Když tyto dva poznatky spojíme, zjistíme, že existují takové kombinace parametrů (počet chodců a počet přepočtů cesty), pro něž je doba výpočtu obou metod totožná (Obr. 4.13). D* Lite algoritmus se vyplatí pouze pro enormní množství přepočtů cest, tedy pro 1000 chodců musí dojít ke změnám grafu G (*Rokycany.xml*) v řádu desítek milionů, aby byl rychlejší než A*. Pro větší grafy je počet potřebných změn mnohonásobně větší, aby byl D* Lite výhodnější.

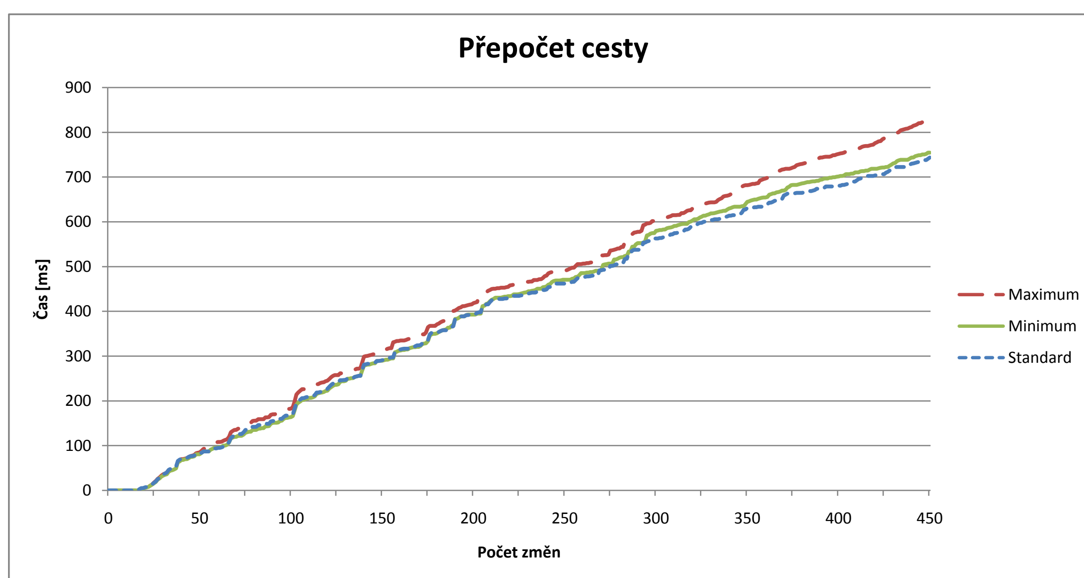
Pakliže přejdeme k úloze, kde pro všechny chodce neexistuje jednotné hranové ohodnocení grafu G, ale každý z nich uvažuje vlastní specifické hranové ohodnocení

grafu G, výhodnějším algoritmem je D* Lite. V takovém případě jediná výhoda A* algoritmu zmizí, tedy bude nutné vytvářet vlastní kopie grafu G každému chodci i pro A* algoritmus. Předzpracování obou algoritmů bude téměř stejně časově náročné, ale D* Lite algoritmus bude rychleji přepočítávat cestu při změně hranového ohodnocení grafu G.

Lokální přepočet cesty

Nyní se zaměřme na testování metod, které slouží pouze k rychlé opravě cesty v lokálním místě trasy. První metodou je standardní lokální metoda, druhou je lokální metoda s použitím maxima a třetí lokální metoda s použitím minima. Tyto metody nemá smysl používat pro známé prostředí, protože nebudeme přepočítávat cestu vždy, když se v neorientovaném grafu G vyskytne změna. Naopak je vhodné je používat pro případ částečně známého prostředí či dokonce pro neznámé prostředí, které ale v této práci neuvažujeme.

Prvním aspektem testování je porovnat rychlost výpočtu všech tří naprogramovaných lokálních metod mezi sebou. Tyto metody používáme pouze pro přepočet již nalezené cesty, proto nás zajímá pouze závislost času na počtu změn. Můžeme si všimnout, že standardní lokální metoda je nepatrně rychlejší než lokální metoda s využitím minima. Nejpomalejší metodou je pak lokální metoda s využitím maxima (Obr. 4.22).



Obr. 4.22: Závislost času na počtu přepočítání cest lokálními metodami v grafu *PilsenCS*.

Dále je také důležité si uvést, kolikrát bylo nutné použít globální vyhledávací metodu k opravení cesty. O použití globální metody rozhoduje podmínka přípustnosti,

která přejde ke globální metodě ve dvou případech. Prvním případem je moment, kdy chodec dojde na konec slepé silnice, a druhým pak případ, kdy se chodec příliš vzdálil od cílového vrcholu svého lokálního přepočtu. Tedy metrická vzdálenost $d(s_{current}, s_{end}) > \epsilon$. V následujících měřeních je podmínka přípustnosti nastavena na 50 metrů.

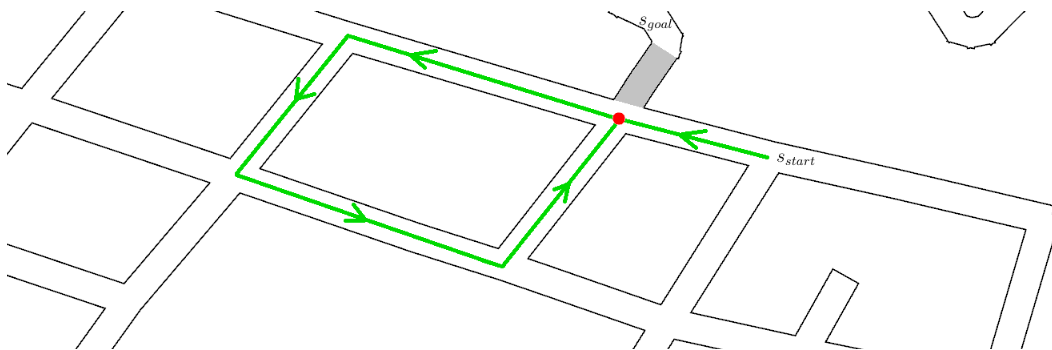
Pro tisíc chodců v částečně známém prostředí (*Rokycany.xml*) dojde k 500 přepočtům cesty. Při použití standardní lokální metody nebo metody s použitím minima dojde pouze k 470 přeplánováním cesty (Tab. 4.2). V případě lokální metody s použitím maxima dokonce pouze k 462 přepočtům. Důvodem je korekce cesty globální metodou. V momentě, kdy dojde k opravě globální metodou, se může stát, že globální metoda ve svém výpočtu opraví i pozici následujícího přepočtu.

Typ lokální metody	Celkový počet přepočtů	Počet globálních oprav
Standard	470	276
Maximum	462	354
Minimum	470	276

Tab. 4.2: Počet přepočtů a globálních oprav lokálních metod v grafu *PilsenCS*.

Můžeme si také všimnout, že ve standardní metodě a metodě s minimem dojde přibližně v 58% případů ke globální opravě. V metodě s maximem dokonce přibližně v 77% případů. Můžeme např. zdvojnásobit toleranci vyhledávání lokálních metod, tedy zdvojnásobíme přípustnou vzdálenost pro upravení cesty lokálními metodami. Nicméně testováním dostaneme stejné údaje jako jsou v tabulce 4.2 a nalezená cesta je horší než v předchozím případě. Při použití jiné městské struktury se počet oprav cest pomocí globálního algoritmu liší maximálně o 5%.

Pokud bychom odstranili podmínku přípustnosti a dovolili bychom lokálním metodám vyhledávat v celém grafu, v některých případech by nebylo nutné používat korekci globální metodou. Dokonce by lokální metoda s využitím minima byla v některých případech lepší než standardní lokální metoda. Na druhou stranu by se mohlo stát, jak testování ukázalo, že se některé cesty mohou zacyklit a chodci chodí neustále dokola (např. kolem bloku budov). Lokální metody slouží k nejrychlejší možné opravě cesty, a proto mají v paměti pouze uzel, ze kterého chodec přišel. Může se tedy např. stát, že chodec *ped* jde z vrcholu s_{start} do vrcholu s_{goal}, ale v čase $t > 0$ narazí na překážku (Obr. 4.23, šedivě). Ze své pozice $s_{current}$ (Obr. 4.23, červený bod) odhadne sousední vrchol s_{neigh}, z jakého je to nejblíže do vrcholu s_{goal}. Odhad se provádí heuristikou, která používá Eukleidovskou vzdálenost. Chodec *ped* se přemístí do vrcholu s_{neigh} a provádí odhad znovu. Může tedy vzniknout zacyklená cesta, příklad je vidět na obrázku 4.23.



Obr. 4.23: Zacyklení lokální metody bez podmínky přípustnosti.

Otestovali jsme tři lokální metody, které slouží pouze pro prostředí částečně známé, z nichž nejlépe obstála standardní lokální metoda. Poznamenejme, že tyto metody byly navrženy pro nejrychlejší možný přepočet, a proto nejsou příliš sofistikované a nalezené cesty je nutné často opravovat pomocí metod globálních.

Srovnání

Nyní se zaměříme na porovnání časové složitosti globálních a lokálních metod a zároveň na korektnost nalezené trasy. Tu budeme poměřovat pomocí relativní chyby podle vztahu

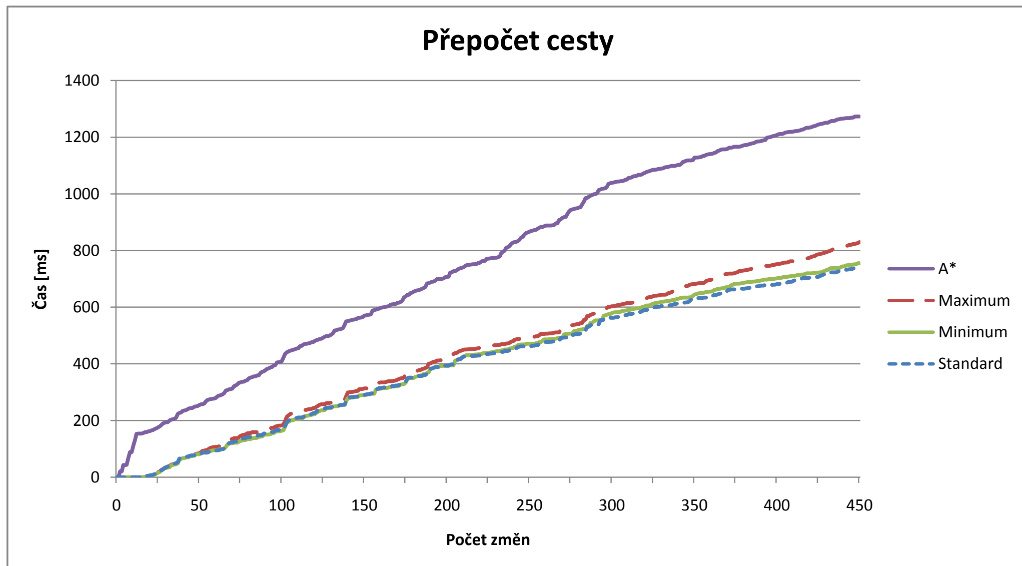
$$dx = \frac{local.path - global.path}{global.path}, \tag{4.1}$$

kde *local.path* je délka cesty nalezené lokální metodou a *global.path* je délka cesty nalezené metodou globální. Navíc *global.path* považujeme za přesnou délku cesty, protože globální metody nachází nejkratší možné cesty.

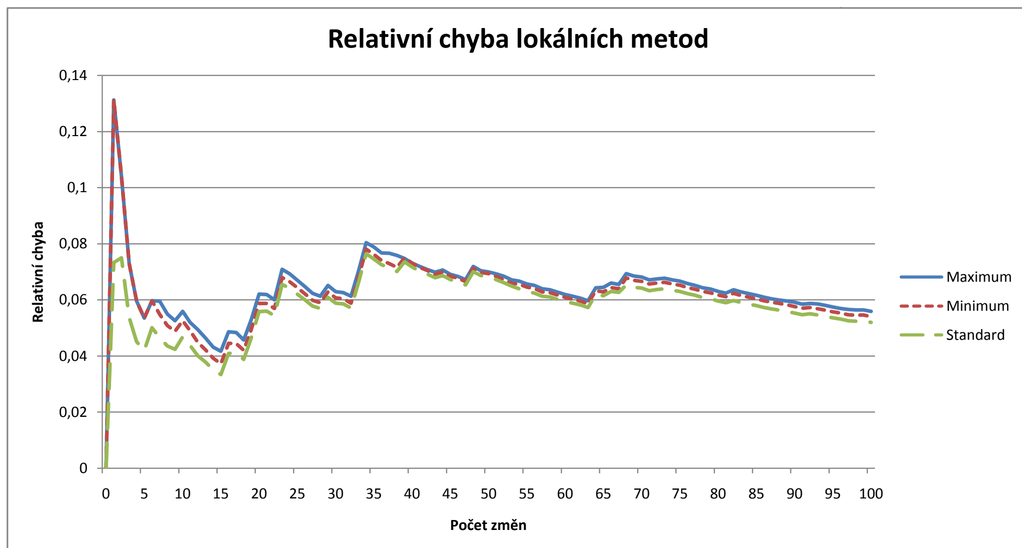
Porovnejme výsledky dosažené v přepočtení cesty pomocí globálních a lokálních metod. Přepočtení cesty globální metodou je časově náročnější než přepočet cesty metodou lokální (Obr. 4.24). Použití globální metody pro opravu cesty je přibližně dvojnásobně časově náročnější než použití metody lokální. To je výsledek, který jsme mohli očekávat, protože lepší lokální metody používají opravu cesty pomocí globální metody přibližně v 55% případů (viz kapitola 4).

Časová náročnost není jediný prvek, který chceme mezi těmito přístupy porovnat. Záleží také velmi na korektnosti výsledné cesty, tedy jak moc se liší nalezená cesta pomocí lokální metody od nejkratší nalezené cesty. Toto porovnání je zaneseno v grafu (Obr. 4.25), kde jsme měřili relativní chybu, které se dopustíme, na jedno přepočtení cesty. Je zde naneseno měření pouze pro 100 přepočtů, aby byl vývoj chyby rozpoznatelný v malém počtu změn. Při větším počtu změn se relativní chyba ustálí

na 5-6%. Můžeme si všimnout, že standardní metoda opět vychází nejlépe a nejhůře dopadla metoda s použitím maxima.



Obr. 4.24: Závislost času na počtu přepočítání cest lokálními metodami a A* algoritmem v grafu *PilsenCS*.



Obr. 4.25: Závislost relativní chyby na počtu změn cesty v grafu *PilsenCSE*.

Kdybychom opět zdvojnásobili přípustnou vzdálenost prohledávaných uzlů, relativní chyba by se ve všech případech zvětšila. Vývoj chyby by vypadal stejně jako v předchozím případě (Obr. 4.25). Rozdílem je maximální chyba, která by byla 18%, a ustálení velikosti chyby na 6-7%.

Ve známém prostředí je použití lokálních metod nevhodné, protože k tomu zaprvé nejsou sestrojené a zadruhé ve známém prostředí požadujeme obvykle přesnou cestu. Ovšem v případě částečně známého prostředí je použití lokálních vyhledávacích metod výhodné. Dosáhneme téměř dvojnásobně rychlejšího přepočtu cesty oproti metodě globální. Toho jsme dosáhli s průměrnou chybou nepřesahující 6%, tedy nalezené cesty lokálními metodami jsou průměrně o 1.06 krát delší než cesty nalezené metodami globálními.

Skupinky chodců

V této části se zaměříme hlavně na testování rychlosti přístupu vytváření skupinek chodců a velikosti relativní chyby, které se tímto přístupem dopustíme. Naší snahou je ušetřit paměť a zároveň urychlit výpočet tím, že chodců, kteří by se pohybovali po stejné cestě nebo po podobné cestě spočteme jednu společnou cestu. Zajímá nás, jestli je tento přístup možné použít pro všechny typy dat (náhodné rozdělení chodců) nebo jenom pro specifické (velký počet chodců ve skupině).

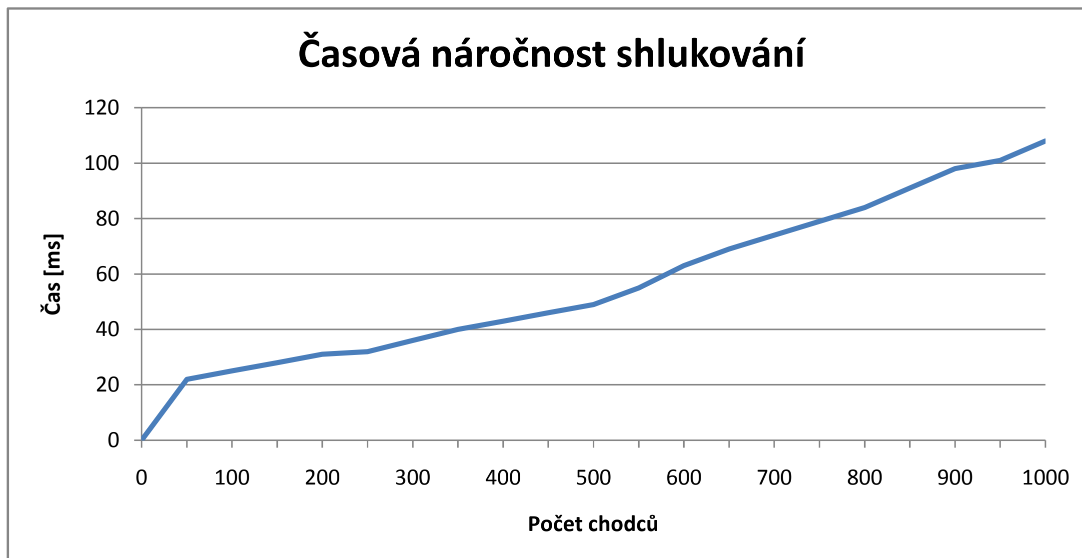
Nejprve provedeme testování shlukování bodů, kde nás zajímá rychlost výpočtu v našem programu. Dalším testováním, jemuž program podrobíme, je rychlost výpočtu počáteční cesty pro skupinky chodců. V neposlední řadě je nutné porovnat, jaké chyby se tímto přístupem dopustíme. Tedy porovnat samostatnou a jedinečnou počáteční cestu chodce s jeho cestou ve skupince nejbližších chodců.

Shlukování

Předpokládali jsme, že vzhledem k nešikovnému propojení knihovny s naším programem bude výpočet shlukování trvat poměrně dlouho, avšak testy ukázaly opak. Na grafu (Obr. 4.26) si můžeme všimnout, že výpočet shlukování pro tisíc chodců trvá přibližně desetinu vteřiny, což je velmi překvapivá informace.

Shlukování bodů začíná momentem, kdy vytvoříme textový soubor s daty chodců v našem programu. Následně ho načteme externím programem psaném v C# jazyce, který spočte shlukování a uloží ho do *xml* souboru. Tedy v grafu (Obr. 4.26) je zanesen čas, během něhož jsou všechny tyto procesy hotovy.

Díky tomuto měření jsme se rozhodli nechat výpočet shlukování proběhnout vždy při načtení dat chodců. Důvodem je již zmíněná velmi nízká časová složitost, která je v porovnání s ostatními výpočty zanedbatelná.



Obr. 4.26: Doba výpočtu shlukování pro tisíc chodců.

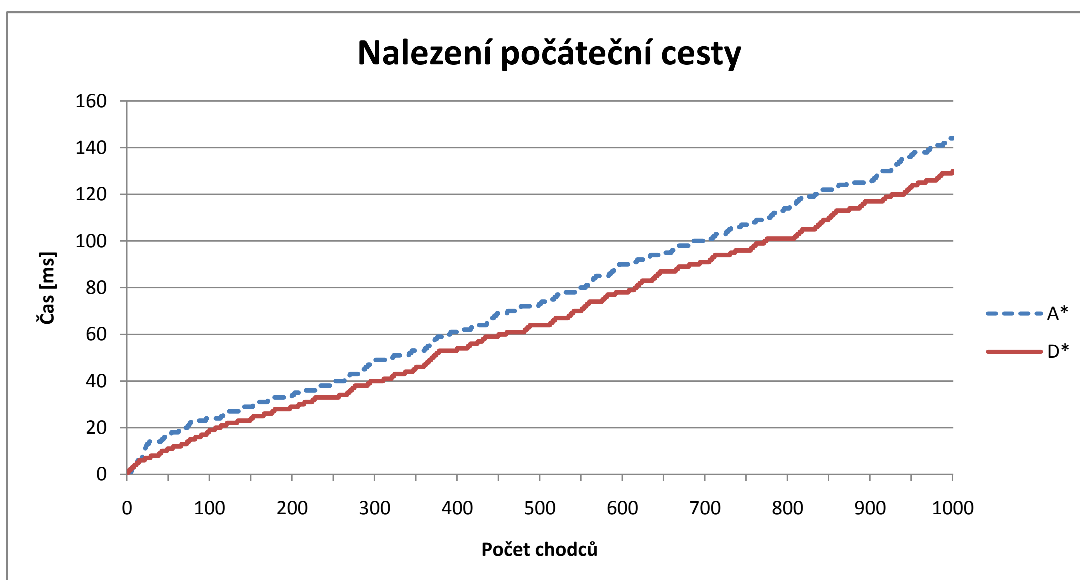
Společná cesta

V této kapitole si pro názornost uvedeme nejprve výsledky pomocí ručního shlukování, aby bylo jednodušší si představit chování chodců. Následně do celého výpočtu zapojíme i automatické shlukování, které je počítáno knihovnou pana Bc. Ondřeje Kaase.

Zřejmě mohou nastat dva extrémní případy. Prvním takovým případem je, že každý chodec *ped* bude mít rozdílnou počáteční pozici s_{start} a zároveň žádný chodec *neigh* v okolí chodce *ped* ($d(ped, neigh) \leq \epsilon$) nebude mít stejný směr ke svému koncovému vrcholu s_{goal}. Tedy žádný chodec *neigh* nacházející se ve vzdálenosti $d(ped, neigh) \leq \epsilon$ od chodce *ped* nemá koncový vrchol $neigh.s_{goal}$ v oblasti o poloměru $d(ped.s_{goal}, neigh.s_{goal}) \leq \epsilon$. V takovém případě je zřejmé, že každému chodci bude spočtena jeho vlastní cesta globálním algoritmem, protože shlukování nenajde žádné skupinky chodců vyhovující vstupním parametrům. Zřejmě bude výsledný čas výpočtu větší než v případě, kdy skupinky chodců neuvažujeme (Obr. 4.8), protože musíme do výsledného času započítat také čas výpočtu shlukování. Nicméně z předchozí podkapitoly 4 víme, že pro tisíc chodců bude výpočet trvat pouze o 108 ms déle než bez výpočtu shlukování. Ve výsledku bude čas předzpracování se shlukováním bodů lišit pouze nepatrně od předzpracování bez shlukování (Obr. 4.8).

Druhým extrémním případem je výpočet velkého množství chodců, jejichž počáteční pozice s_{start} jsou totožné. To samé platí taktéž pro jejich koncový vrchol s_{goal}. Je tedy zřejmé, že v takovém případě není nutné počítat každou cestu zvlášť, ale stačí ji spočítat pouze jednomu chodci a všem ostatním ji zkopírovat. Důvodem vytvoření kopií ostatním chodcům je fakt, že každý chodec může vyrazit v jiný

časový okamžik. Může se tedy stát, že výsledná cesta bude nakonec odlišná v závislosti na změnách v grafu G. V porovnání s prvním extrémním případem je tento extrémní případ výpočetně mnohonásobně rychlejší (Obr. 4.27). V grafu není zahrnuto vytváření kopie neorientovaného grafu G každému chodci. Zaprvé bylo uvedeno již v kapitole 4 a zadruhé čas kopírování neorientovaného grafu G zastínil časovou závislost A* algoritmu, která opticky splývala s vodorovnou osou grafu (Obr. 4.27).



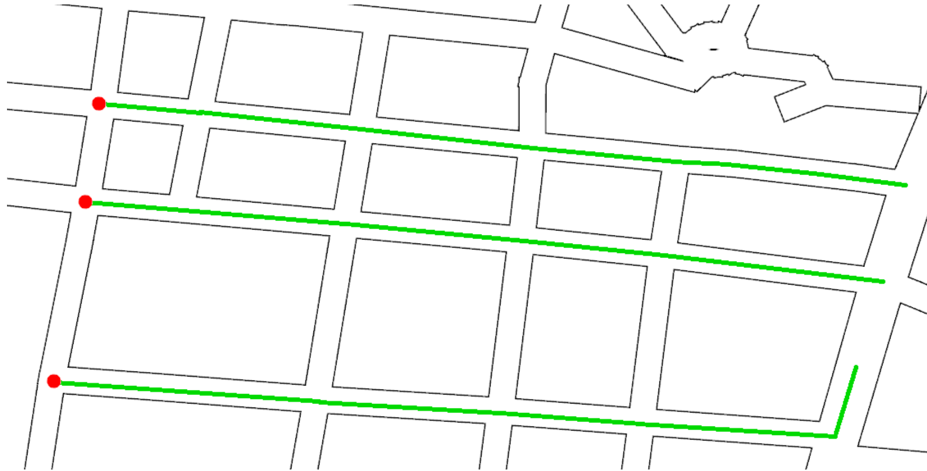
Obr. 4.27: Rychlost nakopírování jedné nalezené cesty skupině chodců.

Dva možné extrémní případy jsme si uvedli, a proto se nyní zaměříme na ostatní případy. Porovnejme nejprve, jak se vůbec cesty nalezené s použitím shlukování liší od cest, které shlukování neuvažují. Na obrázku 4.28 jsou vidět tři nalezené nejkratší cesty pro tři různé chodce, kde červený bod představuje chodce v počátečním vrcholu s_{goal} a zelená křivka nalezenou nejkratší cestu. Tento výsledek dostaneme, když nepoužijeme metodu shlukování.

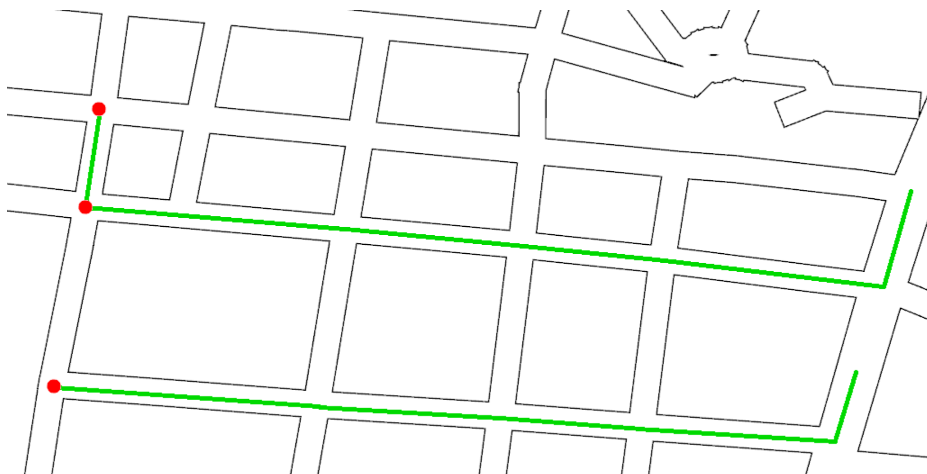
Pokud metodu shlukování bodů použijeme pro oblast o poloměru $\epsilon = 25$ metrů, dostaneme shluk dvou bodů (Obr. 4.29). Je vidět, že první chodec ped_{top} (v horní části obrázku) využije cestu, kterou sleduje druhý chodec ped_{mid} (prostřední). Je tedy nutné dopočítat pouze dva malé úseky cesty chodce ped_{top}. Prvním je cesta z pozice $ped_{top}.start$ do pozice $ped_{mid}.start$, druhým je poté cesta z pozice $ped_{mid}.end$ do pozice $ped_{top}.end$.

V případě, že rozšíříme okruh přípustných bodů shlukování na $\epsilon = 50$, připadne do shlukování i třetí chodec ped_{bot} (Obr. 4.30). Ten analogicky využije cestu chodce ped_{mid} jako v předchozím případě chodec ped_{top}.

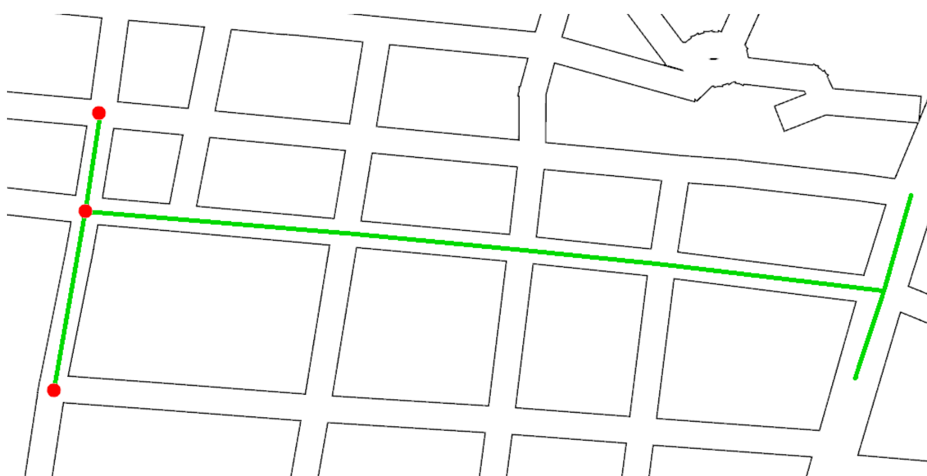
Tato metoda pro urychlení výpočtu vypočte shluky všech chodců nezávisle na čase, kdy se chodec vydá na svoji cestu. Představme si tedy několik chodců, z nichž část



Obr. 4.28: Nalezená nejkratší cesta třem chodcům.

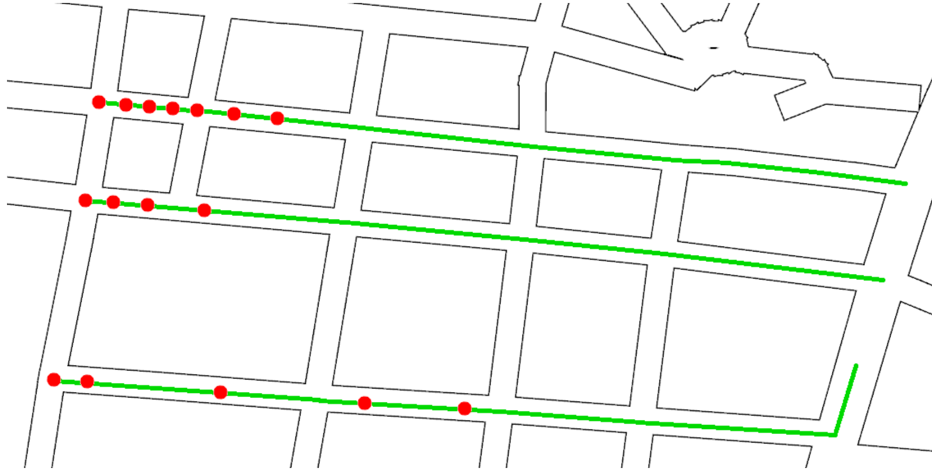


Obr. 4.29: Nalezená cesta s využitím shlukování bodů pro $\epsilon = 25$.

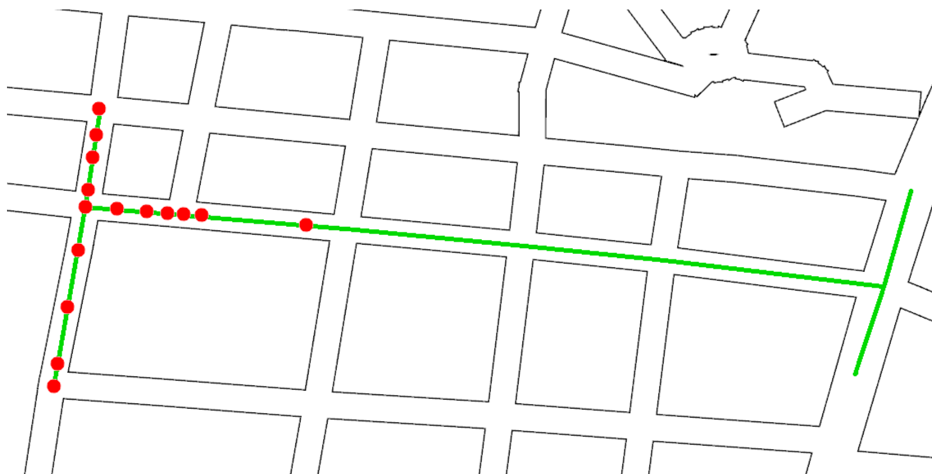


Obr. 4.30: Nalezená cesta s využitím shlukování bodů pro $\epsilon = 50$.

vychází z počáteční pozice chodce ped_{top}, někteří z pozice chodce ped_{mid} a ostatní z pozice chodce ped_{bot}. Každý chodec navíc vyrazí v jiném časovém okamžiku. Výsledek takové simulace v čase $t > 0$, kdy každému chodci spočteme vlastní cestu, je vidět na obrázku 4.31. Po použití shlukování bodů a využití již nalezené cesty dostaneme ve stejném časovém okamžiku simulace $t > 0$ výsledek na obrázku 4.32.



Obr. 4.31: Pohyb chodců po nalezené nejkratší cestě.



Obr. 4.32: Pohyb chodců po nalezené cestě s využitím shlukování bodů pro $\epsilon = 50$.

Nyní víme, jak se metoda chová, a proto si ukažme výsledky dosažené automatickým výpočtem shlukování. Tato knihovna používá svůj specifický parametr η. Pro data z Open Street Map je tento parametr $\eta \approx 3 \cdot \epsilon \cdot 10^{-5}$. Při testování tisíce náhodně rozmístěných chodců jsme zjistili, že použití parametru $\eta > 0.005$ ($\epsilon \approx 165$ metrů) pro shlukování je nevhodné (Tab. 4.3). To platí pouze pro data pocházející z databáze Open Street Map a pro jiná data se může lišit. Důvodem možné odlišnosti je reprezentace vzdálenosti vrcholů (cm, m, km, atd.) neorientovaného

grafu G. Dále si uveďme, že minimální relativní chyba všech skupin byla 0. Tuto chybu počítáme analogicky ke vztahu 4.1, jen nebudeme používat lokální metody, ale společnou cestu. Vztah pro výpočet chyby pak přejde ve vztah

$$dx = \frac{same.path - global.path}{global.path}, \tag{4.2}$$

kde *same.path* je délka nalezené cesty s využitím společné cesty a *global.path* je cesta nalezená globálním algoritmem.

V tabulce 4.3 je uvedena nejmenší relativní chyba, která je vyšší než 0, pokud taková existuje. Dále si můžeme všimnout, že s rostoucí velikostí výsledných shluků (shlukovací parametr) probíhá výpočet mnohem rychleji. Zároveň s rostoucí velikostí shluků se dopouštíme velkých chyb a je zřejmé, že pro parametr shlukování $\eta = 0.01$ ($\epsilon \approx 330$ metrů) existuje i cesta, která je dvakrát delší než cesta nejkratší.

Počet skupin	Shlukovací par.	Min. rel. chyba	Max. rel. chyba	Čas [ms]
1000	0	0	0	3739
998	0.005	0.057	0.057	3557
988	0.01	0.007	1	3242
972	0.015	0.006	1.706	3264
947	0.02	0.002	1.706	3173
905	0.025	0.002	44.511	3101
854	0.03	0.002	44.511	3079
787	0.035	0.002	44.511	3047
729	0.04	0.002	44.511	3008
671	0.045	0.003	44.511	2764
618	0.05	0.003	44.511	2692

Tab. 4.3: Shlukování pro tisíc náhodně rozmístěných chodců.

Nyní si ukažme situaci, kdy rozložení chodců není náhodné. Jedná se o speciálně vytvořená data, pro něž neplatí vztah $\eta \approx 3 \cdot \epsilon \cdot 10^{-5}$. Hledáme trasu 33 chodců, kteří jsou umístěni jeden poblíž druhého. Tedy všechny jejich počáteční pozice leží v kružnici o poloměru $\epsilon = 0.5$m a jejich cíle leží v odlišné kružnici o témže poloměru $\epsilon = 0.5$m. Testování ukazuje (Tab. 4.4), že v takovém případě pro parametr shlukování $\eta = 2$ ($\epsilon \approx 0.2$m) dosáhneme stále dobrého výsledku, protože výsledná cesta je větší maximálně o 8%. Ušetřený čas pro takto malý počet chodců je téměř zanedbatelný. Uvědomme si, že v předchozím testování (Tab. 4.3) jsme dosáhli velkých časových úspor za cenu neakceptovatelně velkých chyb.

Počet skupin	Shlukovací parametr	Min. rel. chyba	Max. rel. chyba
33	0	0	0
33	0.1	0	0
18	0.2	0.02	0.04
11	0.3	0.02	0.06
7	0.4	0.02	0.079
4	0.5	0.019	0.08
4	1	0.019	0.08
3	2	0.019	0.08
2	3	0.019	0.121
2	4	0.019	0.121
2	5	0.019	0.121
1	10	0.02	0.141

Tab. 4.4: Shlukování pro 33 specielně umístěných chodců.

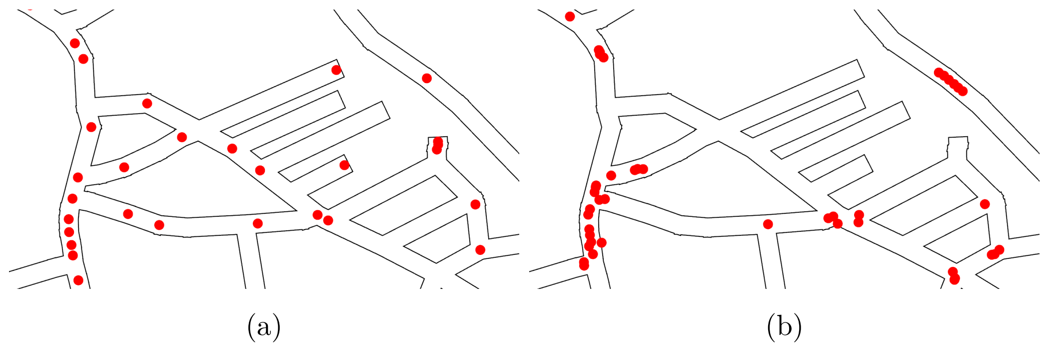
Výpočet shlukování je vhodné provést při každém načtení dat chodců, protože je zřejmé, že v případě velkého zastoupení chodců, které by šlo rozumně reprezentovat skupinkou, dosáhneme enormní časové úspory. Navíc tuto časovou úsporu bude doprovázet rozumná relativní chyba výsledné cesty. V případě, že nebudeme chtít počítat cesty pomocí společné cesty, nás výpočet shlukování nijak neomezuje. Jedná se o nepatrný časový okamžik, který výsledný čas simulace téměř neovlivní.

Kolize chodců

Jak již bylo zmíněno v kapitole 3, proběhla drobná spolupráce mezi touto diplomovou prací a diplomovou prací pana Ing. Pavla Bradejského [2] V této podkapitole si uvedeme pár zjištění o lokálním počítání kolizí mezi chodci.

Chtěli jsme ukázat, jak se liší časová náročnost výpočtu pohybu chodců naší metodou, která kolize chodců ignoruje, a metodou pana Ing. Brandejského. Toto bohužel není možné korektně změřit a porovnat, protože obě metody interpretují parametr rychlosti v_{ped} chodce odlišným způsobem. Optickým porovnáním a měřením času jsme zjistili, že pro totožný parametr rychlosti dojde k menšímu počtu volání metody pana Ing Brandejského pro malý počet chodců. Mohlo by se zdát, že je tento způsob rychlejší a lepší než naše metoda, ale výpočet metody kolizí používá parametr rychlosti v_{ped} k vyjádření větší vzdálenosti než naše metoda.

Pro větší množství chodců je výpočet kolizí chodců nevhodný. V případě ignorování kolizí mezi chodci dojde každý chodec do svého požadovaného koncového



Obr. 4.33: (a) Konečná pozice chodců při neuvažování kolizí chodců, (b) Zablokování chodců při uvažování kolizí mezi nimi.

vrcholu s_{goal} v grafu G (Obr. 4.33 (a)). Pokud kolize budeme uvažovat, často se stává, že dojde k nekonečnému cyklu metody pro lokální detekci kolizí chodců. Tento problém nastává v místě, kdy se potká více chodců na jednom místě. Představme si, že chodec ped_1 obchází chodce ped_2. Než ho ale stihne správně obejít, vstoupí mu do cesty chodec ped_3, který mu zablokuje cestu. Stane se tak, že se na jednom místě nahromadí více chodců, se kterými si lokální metoda není schopná poradit. Neustále přepočítává jejich pozice, jenž se nemění, a žádný z těchto chodců nikdy nedojde do cíle (Obr. 4.33 (b)).

Kolize chodců slouží jako rozšíření našeho programu a ukazuje jeden ze směrů, kterým bychom chtěli v budoucnu tuto práci rozšířit. Pro malé počty chodců nebo pro chodce, kteří nechodí příliš blízko u sebe, je metoda pana Ing. Brandejského spolehlivá a funkční. Problém nastává v případě velkého počtu chodců nebo jejich přílišné blízkosti. Je zde vysoká pravděpodobnost, že se v některém uzlu potká příliš mnoho chodců a metoda se zacyklí. Jedná se tedy o zajímavé rozšíření, které ale není 100% spolehlivé.

5 ZÁVĚR

V této práci jsme se zaměřili hlavně na dynamické heuristické algoritmy pro hledání cest v dynamickém neznámém prostředí, které byly detailně popsány, tedy D* algoritmy. Ačkoliv byly tyto algoritmy primárně vyvíjeny v robotice pro navigaci robota, jejich myšlenka je velmi podobná problematice vyhledávání trasy jedinci. Z těchto algoritmů jsme zvolili optimalizovanou verzi D* Lite algoritmu, který je v současné době jedním z nejlepších D* algoritmů.

Výsledky D* Lite algoritmu jsme se rozhodli porovnávat se statickým algoritmem A*. Potvrdili jsme náš předpoklad, že je A* algoritmus vhodnější volbou pro ohodnocený neorientovaný graf G, jehož hranové ohodnocení je stejné pro všechny chodce. Překvapivým zjištěním je, že D* Lite algoritmus v takovém případě mnohdy není lepší ani v případě, kdy se každému chodci musí přepočítat cesta, protože se změnilo hranové ohodnocení grafu G. D* Lite algoritmus je tedy výhodné používat, pokud má každý chodec unikátní grafovou reprezentaci virtuálního města. V takovém případě odpadá jediná výhoda A* algoritmu, kterou měl v předešlém problému, a je nepraktické ho používat.

Dále jsme porovnali lokální metody s metodami globálními a lokální metody mezi sebou. Očekávali jsme o něco rychlejší výpočet pomocí lokálních metod než nám ukázalo testování a naměřené hodnoty. Nicméně snížení času výpočtu téměř o polovinu je také dobrý výsledek. Zrychlení doprovází nalezená cesta, která je průměrně o 6% delší než nejkratší možná, což je přijatelná cena za téměř dvojnásobnou rychlost. Ze všech tří lokálních metod vychází standardní lokální metoda nejlépe a nejhůře pak lokální metoda s využitím maxima. Při odstranění podmínky přípustnosti vychází lokální metoda s využitím minima v některých případech lépe než metoda standardní. Problémem je velká pravděpodobnost zacyklení výpočtu bez této podmínky.

Důležité je také zmínit, že je možné spočítat pro 50-100 chodců 200 změn pomocí lokálních metod nebo 100 změn pomocí globálních metod bez okem znatelné trhané vizualizace chodců a jejich pohybu. Zároveň si poznamenejme, že pro tisíc chodců už vizualizace není úplně plynulá, a proto jsme netestovali metody pro větší počet chodců.

Poslední velkou částí této práce bylo zjistit, jestli lze nějakým způsobem využít i podobnost pozic chodců. Otestováním jsme zjistili, že přístup vytváření skupinek chodců má velké výhody ve všech směrech. Pro vhodná data dosáhneme velké časové a paměťové úspory za cenu nepřesné cesty. Tato nepřesnost záleží na voleném parametru shlukování a dosahuje malých hodnot při jeho rozumné volbě. Výpočet shlukování má smysl provést i v případě, že je rozložení chodců ve vstupních datech náhodné a shlukování nevytvoří žádné skupinky. Důvodem je, že čas výpočtu

shlukování je téměř zanedbatelný a ostatní výpočty nezatěžuje.

Závěrem bychom chtěli nastínit budoucí pokračování v této problematice. Mezi budoucí cíle patří opravení metody pro vizualizaci struktury města a celkové urychlení vizualizace pro testování většího množství dat. Pozornost si také zaslouží lokální metody, které bychom chtěli upravit tak, aby došlo ke globální opravě cesty ještě v menším procentu výpočtů. V neposlední řadě bychom se chtěli hlouběji zaměřit na skupinky chodců, kde bychom chtěli vytvořit vlastní speciální shlukování nebo minimálně vylepšit propojení knihovny shlukování s naším programem, protože testování ukázalo, že se jedná o velmi smysluplnou úpravu programu.

REFERENCE

[1] *Beneš P., Massih M. A., Jarvis P., Aliaga D. G.:* **Urban Ecosystem Design**, Symposium on Interactive 3D Graphics and Games, ACM, p. 167-174, 2011

[2] *Brandejský, P.:* **Lokální navigace chodců ve virtuálních modelech měst**, Diplomová práce, Západočeská univerzita v Plzni, 2014

[3] *Brož P., Zemek M., Kolingerová I., Szkandera J.:* **Dynamic Path Planning with Regular Triangulations**, nabídnuto k publikaci pro Machine Graphics and Vision, 2015

[4] *Connoly C. I., Burns J. B., Weiss R.:* **Path Planning Using Laplace's Equation**, IEEE International Conference on Robotics and Automation, p. 2102-2106, 1990

[5] *Čada R., Kaiser T., Ryjáček Z.:* **Diskrétní matematika**, Skripta, Západočeská univerzita v Plzni, 2004

[6] *Drábek P., Kufner A.:* **Úvod do funkcionální analýzy**, Skripta, Západočeská univerzita v Plzni, 1993

[7] *Ferguson D.; Stentz A.:* **Field D*: An Interpolation-based Path Planner and Replanner**, Robotics Research, p. 239-253, 2007

[8] *Fuksa M.:* **Delaunayova triangulace s omezením (CDT) v E^2 a E^3**, Diplomová práce, Západočeská univerzita v Plzni, 2006

[9] *Hart P. E., Nilsson N. J., Raphael B.:* **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**, IEEE Transactions on Systems Science and Cybernetics 4(2), p. 100-107, 1968

[10] *Kaas, O.:* **Využití shlukování pro GIS aplikace**, Bakalářská práce, Západočeská univerzita v Plzni, 2013

[11] *Koenig, S., Likhachev M.:* **D* Lite**, American Association for Artificial Intelligence, p. 476-483, 2002

[12] *Koenig S., Likhachev M., Furcy D.:* **Lifelong Planning A***, Artificial Intelligence, Volume 155, Issues 1–2, p. 93–146, 2004

[13] *Pelechano N., Allbeck J., Badler N.:* **Virtual Crowds: Methods, Simulation, and Control**, Synthesis Lectures on Computer Graphics and Animation 3(1), p. 1-176, 2008

[14] *Stentz A.:* **Optimal and Efficient Path Planning for Partially-Known Environments**, IEEE International Conference on Robotics and Automation, p. 3310-3317, 1994

[15] *Stentz A., Likhachev M.:* **The Focussed D* Algorithm for Real-Time Replanning**, IJCAI, p. 1652-1659, 1995

[16] *Szkandera, J.:* **Navigace jedinců v rámci davu**, Bakalářská práce, Západočeská univerzita v Plzni, 2012

[17] © *Esri:* **Esri CityEngine**, `http://www.esri.com/software/cityengine`, 2015

[18] © *Přispěvatelé OpenStreetMap:* **Open Street Map**, `http://www.openstreetmap.org`, 2015

[19] © *Riot Games:* **League of Legends**, `http://eune.leagueoflegends.com`, 2015

[20] *Siek J., Lee L-Q., Lumsdaine A.:* **The Boost Graph Library**, `http://www.boost.org/doc/libs/1_58_0/libs/graph/doc/`, 2001

[21] © *The Khronos Group:* **Computational Geometry Algorithms Library**, `https://www.opengl.org`, 2015

[22] © *The Qt Company:* **Qt Library**, `https://www.qt.io`, 2015

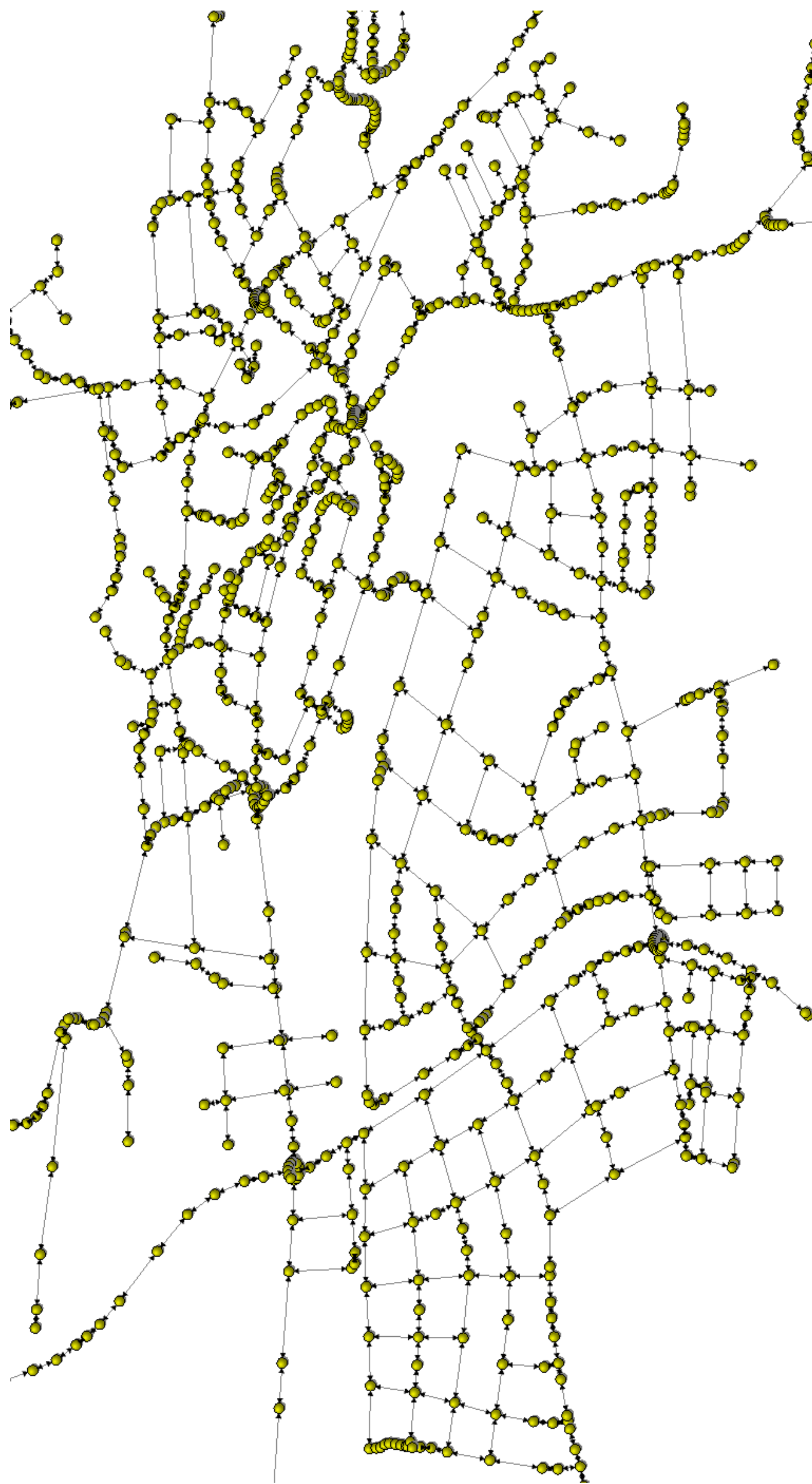
[23] © *Ubisoft:* **Heroes of might and Magic**, `http://en.wikipedia.org/wiki/Heroes_of_Might_and_Magic`, 2015

PŘÍLOHY

A OBRÁZKY

Následující obrázku ukazují vizualizaci virtuálního města z dat, která jsou k dispozici na Open Street Map [18]. Na prvním obrázku A.1 je vidět půdorys města Rokycany, na druhém obrázku A.2 je pak vidět jeho grafová reprezentace, tedy rozložení hran a uzlů neorientovaného grafu G ze souboru *Rokycany.xml*. Další obrázek ukazuje taktéž půdorys virtuálního města (Obr. A.3). Jedná se o část města Plzeň, přesněji o jeho centrum a přilehlé okolí. Na posledním obrázku A.4 je vidět část rozložení hran a vrcholů neorientovaného grafu G ze souboru *Pilsen.xml*

Obr. A.1: Silniční půdorys virtuálního města Rokycan.



Obr. A.2: Rozložení uzlů a hran grafu G reprezentujícího Rokycany.

Obr. A.3: Silniční půdorys části virtuálního města Plzně.

Obr. A.4: Rozložení grafu reprezentujícího Rokycany.

B VSTUPNÍ SOUBORY

```xml
<UrbanSim><!-- kořen .xml souboru -->
 <Bounds minLat="49.7196946" maxLat="49.7643966" minLon="13.5659492" maxLon="13.6270951" scale="80000.0"/>
 <Nodes><!-- Vrcholy grafu G -->
  <!-- y-ová pozice, x-ová pozice a id vrcholu -->
  <node lat="49.7517317" lon="13.5981654" id="0"/>
  <node lat="49.7509889" lon="13.5986711" id="1"/>
  <node lat="49.7487587" lon="13.5997461" id="2"/>
  <node lat="49.7428993" lon="13.5905527" id="3"/>
 </Nodes>
 <Edges><!-- Hrany grafu G -->
  <edge source="0" destination="1" weight="-1" id="1"/>
  <edge source="1" destination="3" weight="-1" id="2"/>
  <edge source="2" destination="3" weight="-1" id="3"/>
 </Edges>
 <EdgeChanges><!-- Změny v grafu G -->
  <!-- id změny, počáteční a koncový vrchol, simulační čas změny, váha a existence hrany -->
  <edge id="0" source="1" destination="0" time="0.1" weight="-1" exist="0"/>
  <edge id="1" source="2" destination="3" time="0.4" weight="-1" exist="0"/>
  <edge id="2" source="1" destination="0" time="1.1" weight="10" exist="1"/>
 </EdgeChanges>
</UrbanSim>
```

Obr. B.1: Příklad *.xml* souboru obsahující strukturu virtuálního města.

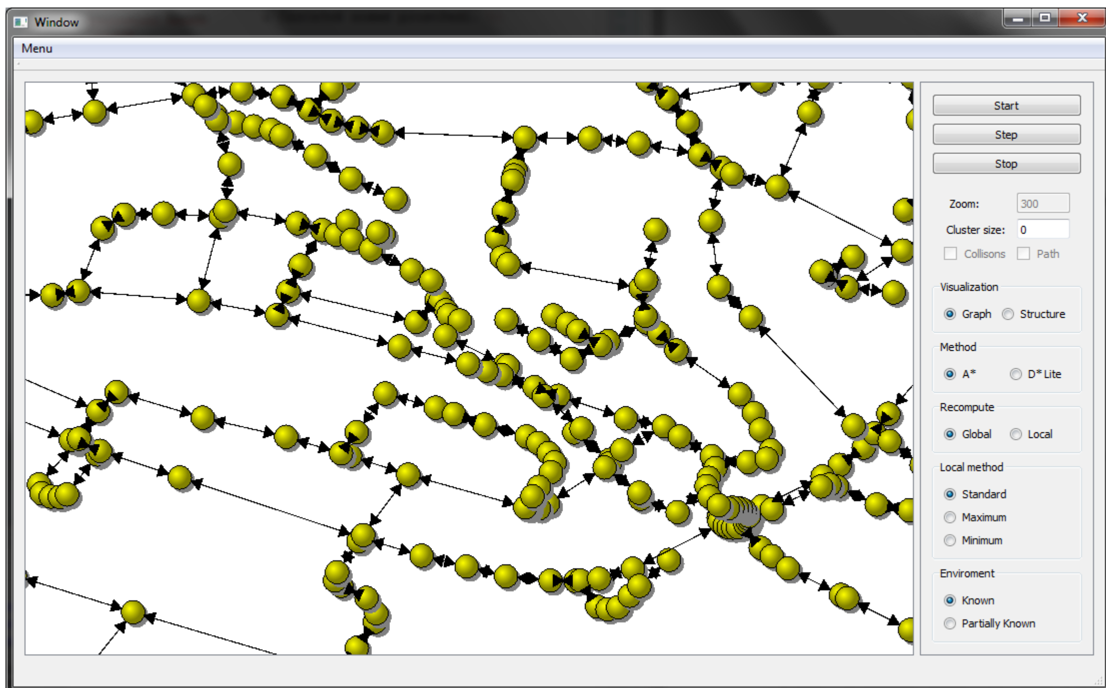
```xml
<UrbanSim><!-- kořen .xml souboru -->
 <Pedestrians><!-- list chodců -->
  <!-- počáteční a cílový vrchol cesty, čas začátku simulace -->
  <pedestrian start="0" destination="1000" time="0"/>
  <pedestrian start="11" destination="42" time="1"/>
  <pedestrian start="3" destination="71" time="14.3"/>
  <pedestrian start="32" destination="107" time="0"/>
 </Pedestrians>
</UrbanSim>
```

Obr. B.2: Příklad *.xml* souboru obsahující parametry každého chodce.

```xml
<UrbanSim><!-- kořen .xml souboru -->
  <Clusters count="2"> <!-- List shluků a jejich počet -->
    <Cluster id="0"><!-- První shluk -->
      <Center id="4" /><!-- Střed shluku -->
      <Pedestrian id="3" /><!-- Další prvek shluku -->
      <Pedestrian id="1" />
    </Cluster>
    <Cluster id="1"><!-- Druhý shluk -->
      <Center id="0" /><!-- Střed shluku -->
      <Pedestrian id="2" /><!-- Další prvek shluku -->
    </Cluster>
  </Clusters>
</UrbanSim>
```

Obr. B.3: Příklad *.xml* souboru obsahující vypočtené shlukování.

C UŽIVATELSKÁ DOKUMENTACE



Obr. C.1: Uživatelské rozhraní našeho programu.

Na obrázku C.1 je vidět uživatelské rozhraní naší aplikace. Toto rozhraní můžeme rozdělit na tři nejdůležitější části. První je tlačítko Menu v levém horním rohu, pod kterým se skrývá načítání dat. Druhou částí je plocha s vykresleným grafem G zabírající největší část aplikace, která slouží pro vizualizaci načtených dat a vykreslování simulace. Poslední částí je ovládání aplikace na pravé straně uživatelského rozhraní.

Menu → Load Structure	Načte existující soubor reprezentující virtuální město. Soubor musí mít příponu *xml*.
Menu → Load Pedestrians	Načte existující soubor chodců s jejich atributy. Soubor musí mít příponu *xml*.
Menu → Load Clusters	Načte soubor shluků spočtených externím programem. Soubor musí mít příponu *xml*

Tab. C.1: Funkce menu

Start	Spuštění simulace.
Step	Jeden krok simulace.
Stop	Zastavení simulace.
Zoom	Nastavení přiblížení scény. Potvrzení enterem.
Cluster size	Parametr shlukování. Potvrzení enterem.
Collisions	Kolize mezi chodci.
Path	Vykreslení nalezené cesty.
Graph	Vizualizace virtuálního města jako graf G.
Structure	Vizualizace půdorysu města.
A*	A* algoritmus.
D* Lite	D* Lite algoritmus.
Global	Přepočet cesty pomocí globálního algoritmu.
Local	Přepočet cesty pomocí lokálního algoritmu.
Standard	Standardní lokální metoda.
Maximum	Lokální metoda s použitím maxima.
Minimum	Lokální metoda s použitím minima.
Known	Známé prostředí.
Partially Known	Částečně známé prostředí.

Tab. C.2: Ovládání simulace

Kolečko myši	Přiblížení a oddálení scény.
Klávesa +	Přiblížení scény.
Klávesa -	Oddálení scény.
Klávesa -	Oddálení scény.
Klávesa šipky	Translace scény ve směru šipky.

Tab. C.3: Funkce vykreslovací oblasti pro volbu Graph

Kolečko myši	Přiblížení a oddálení scény.
Pravé tlačítko myši + pohyb myši	Translace vykreslené scény.
Klávesa P	Vytvoří screenshot vykreslené scény.

Tab. C.4: Funkce vykreslovací oblasti pro volbu Structure