Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra matematiky

Bakalářská práce

Navigace jedinců v rámci davů

Plzeň, 2012 Jakub Szkandera

Poděkování

Tímto bych rád poděkoval vedoucí této bakalářské práce, Prof. Dr. Ing. Ivaně Kolingerové, za možnost podílet se na části vědeckého projektu na katedře informatiky, za cenné rady a připomínky, ale hlavně za trpělivost a podporu. Stejně důležité poděkování patří i Ing. Tomáši Vomáčkovi za důležité připomínky a návrhy k implementační části této práce. V neposlední řadě pak děkuji mé rodině, která mi umožnila studovat na vysoké škole, a za její podporu.

Prohlášení

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

V Plzni dne …… ……

Navigation of individuals in a crowd

Crowd simulation in urban models is one of hot problems in today computer graphics. A subproblem of crowd simulation is path planning of individuals. In this work we apply some well-known algorithms as A* search algorithm Floyd-Warshall algorithm and well-known data structures, such as Navigation graph and Cell and Portal graph. We present results of experiments and comparison of some of these approaches.

Obsah

1 Úvod

Hledání cest je jednou ze základních oblastí teorie grafů, zároveň patří k nejpotřebnějším využívané. Můžeme se s ním setkat např. při navigaci robota, kdy je nutné dynamicky měnit jeho trasu v závislosti na okolí, anebo při navigaci chodců, při simulaci davů v modelech měst.

Cílem této práce je vyzkoušet některé již známé algoritmy pro hledání cest ve statickém a předem známém prostředí, které by se daly použít pro hledání trasy jednotlivců v závislosti na vztahu chodce ke čtvrti (tzn. oblíbenost čtvrti) a na vztahu chodce k jinému chodci (tzn. oblíbenost jiné nebo stejné sociální skupiny) při simulaci davů v modelech měst.

Práce je součástí vědeckého projektu MŠMT LH11006 Interaktivní geometrické modely pro simulaci přírodních jevů a davů. V rámci tohoto společného projektu s Purdue University v USA je vyvíjen model pohybu chodců po simulovaném městském prostředí.

Navržené alternativní algoritmy k A* algoritmu byly vyvinuty ve spolupráci s Ing. Tomášem Vomáčkou, doktorandem na KIV.

2 Virtuální model města

2.1 Historie programu

Program EcoSim [BM*11] byl vyvinut v rámci výzkumné skupiny počítačové grafiky na již zmíněné Purdue University (West Lafayette, Indiana, USA) a vedoucím této skupiny je Doc. Bedřich Beneš. Tento výzkumný tým se zabývá urbanistickými simulacemi, mezi něž patří simulace růstu rostlin ve virtuálním městě, animace davů a simulace rozvoje městské struktury v závislosti na okolních podmínkách, jako je např. dopravní dostupnost, struktura blízkého terénu, blízkost vodních ploch, atp.

V původní verzi EcoSim (verze vyvinutá v Purdue University) slouží pro simulaci růstu rostlin ve virtuální městské zástavbě. Pro tento účel využívá výpočtu na GPU. Jmenovitě Compute Unified Device Architecture (zkráceně CUDA [Nvi12]), která byla vyvinuta a stále je vyvíjena společností NVIDIA. EcoSim využívá technologie, jako jsou např. programovací jazyk C/C++ a OpenGL [Khr12] pro renderování grafiky a GLUT (OpenGL Utility Toolkit) pro uživatelské rozhraní (GUI, Graphical User Interface).

V současné době program EcoSim a jeho nejnovější verze slouží také k simulaci pohybu chodců ve virtuálním městě. Cílem společného projektu se ZČU je vytvořit virtuální model města společně s chodci tak, aby jeho výsledky co nejpřesněji odpovídaly reálnému chování lidí pohybujících se po městě s ohledem na jejich oblíbenost městských čtvrtí a jejich vztahy mezi sociálními skupinami a zároveň aby byl tento model použitelný v reálných sociologických aplikacích. Důležité je, že tento projekt se zabývá každým jedincem, což je jedním ze dvou hlavních modelů (kapitola 2.3).

EcoSim navíc, na rozdíl od své předchozí verze, používá knihovnu CGAL [CGA12], která se využívá ve velkém množství výpočtů, jež jsou prováděny v programu.

2.2 Cíl práce

Momentální stav programu umožňuje např. načíst geometrickou strukturu města, jednotlivé chodce současně s jejich požadavky, najít cestu z počátečního uzlu do koncového a to vše zobrazit. V současné době ing. T. Vomáčka pracuje na části řešení založené na umělé inteligenci a v budoucnu bude zapotřebí např. vytvořit i animace chodců a řešit jejich vzájemné kolize.

Cílem této práce je zaměřit se na malou část z tohoto vědeckého projektu, přesněji na vysokoúrovňové vyhledávání cest každého chodce. Záměrem vyšší úrovně je naplánovat optimální cestu chodce z bodu x do bodu y. Pro tento cíl je zapotřebí vybrat a implementovat algoritmy pro hledání nejkratší cesty, modifikovat je a porovnat je. Je nutné upozornit, že při tomto vyhledávání není uvažována kolize s jiným chodcem, protože pouze naplánujeme cestu. Nemůžeme proto říci, zda daný chodec potká někoho z jiných chodců a kde. Kolize a vztahy mezi sociálními skupinami jsou problémem nízkoúrovňového vyhledávání. Z tohoto důvodu nejsou v této práci uvažovány vztahy mezi chodci, které jsou uvedeny v zásadách pro vypracování.

2.3 Rozdělení modelů

Navigace chodců městem má dva dominantní úhly pohledu na problematiku. První model pohlíží na dav jako na celek (vyšší úroveň), kdežto druhý model se zabývá každým chodcem zvlášť (nižší úroveň) a oba modely mají svá pro i proti.

Dav

Větší část projektů reprezentuje pohyb davu jakožto pohyb kontinua [TCP06]. Na dav se pohlíží jako na celistvý útvar a daný problém se řeší např. pomocí Navier-Stokesových rovnic [Hug03], které slouží pro výpočet toku kapalin.

Výhoda této metody spočívá v tom, že není nutné se zajímat o každého jedince zvlášť, čímž se snižuje časová složitost programu. Tímto způsobem lze například modelovat pohyb velkého davu lidí, kteří mají stejný cíl, nebo tok davu úzkými průchody a podobně.

Ve výhodě této metody tkví zároveň její nevýhoda, a to, že každý chodec má stejný cíl a zanedbávají se jeho osobní požadavky. Tudíž touto metodou nelze simulovat chování jedinců a jejich rozhodování podle prioritních potřeb.

Jednotlivec

Druhým pohledem na problematiku modelování chování chodců je odstranění nedostatků v předchozím přístupu. Tato metoda se nezaměřuje na dav jako na celek, ale na jeho dílčí části, jedince [TCP06]. Tímto přístupem je možné modelovat potřeby jednotlivce tak, že už se nebude muset chovat jako ovce ve stádu.

Cílem toho přístupu je dát každému chodci volnost v rozhodování o jeho potřebách. S tím se ovšem váže vyšší časová a paměťová náročnost.

3 Definice důležitých pojmů

Tato kapitola slouží k nadefinování důležitých pojmů, které se budou nadále vyskytovat v textu. Definice 3.1-3.8 jsou převzaty z [ČKR04], definice 3.9, 3.11 a 3.12 jsou převzaty z [Fuk06] a definice 3.10 je z [Záb05].

Vzhledem k tomu, že celá práce pojednává o hledání cesty z bodu x do bodu y, která je předmětem teorie grafů, je nutné uvést, co vůbec pojem graf znamená.

Definice 3.1 (Graf) Graf G je dvojice $G = (V, E)$, kde V je konečná množina *vrcholů* (*uzlů*) a E je konečná množina hran, $E \subset \binom{V}{2}$, přičemž

$$\binom{V}{2} = \{\{x, y\} : x, y \in V \text{ a } x \neq y\}$$

je množina všech dvouprvkových množin (*neuspořádaných dvojic*) prvků množiny V. Vrcholy $x, y \in V$ jsou *sousední*, pokud $\{x, y\} \in E$.

Důležitým zjištěním definice 3.1 je, že definovaný graf je neorientovaný. Jeho hrany nemají směr, protože jsou definovány jako neuspořádaná dvojice vrcholů, a navíc nepovolujeme násobnost hran (více než jedna hrana mezi vrcholy x a y). Tímto zjištěním jsme dostali důležitou reprezentaci grafu, proto je potřebné si nadefinovat i druhý pohled, a to graf orientovaný.

Definice 3.2 (Orientovaný graf) *Orientovaný graf* je dvojice $G = (V, E)$, kde V je množina vrcholů a $E \subset V \times V$ je množina hran.

Z definice je možné vyčíst, že hrany jsou nyní podmnožinou kartézského součinu $V \times V$ a tedy uspořádané dvojice vrcholů. Ani v tomto případě nepovolujeme násobnost hran, ale může se zde objevit „protichůdnost“ hran mezi dvěma vrcholy. Dalším důležitým pojmem pro tento text je souvislost grafu.

Definice 3.3 (Souvislý graf) Graf G je *souvislý*, pokud pro každé dva vrcholy x, y existuje v grafu G cesta z x do y. V opačném případě je graf G *nesouvislý*.

Z teorie grafů si definujeme další dva základní pojmy, které jsou pro tuto práci klíčové. Jelikož se tato práce zabývá hledáním cest chodců, zajímá nás, jakou vzdálenost chodec ujde. Z tohoto důvodu je nutné přidat dodatečnou informaci o „délce“ jednotlivých hran, čímž získáme tzv. ohodnocený graf a s ním související váhu cesty.

Definice 3.4 (Ohodnocený graf) *Ohodnocený orientovaný graf* (G, w) je orientovaný graf G spolu s reálnou funkcí w: $E(G) \to (0, \infty)$. Je-li e hrana grafu G, číslo $w(e)$ se nazývá její *ohodnocení* nebo *váha*.

Než budeme moci zavést pojem váhy cesty, potřebujeme si nejdříve zavést samotný pojem cesta a s ním velmi úzce související pojem sled. S těmito pojmy budeme mít připravené zázemí pro definování váhy cesty.

Definice 3.5 (Sled) *Sled* (z vrcholu u do vrcholu v) v grafu G je libovolná posloupnost $(u = v_0, v_1, \dots, v_k = v)$, kde v_i jsou vrcholy grafu G a pro každé $i = 1, \dots, k$ je $v_{i-1}v_i$ hranou grafu G. Číslo $k + 1$ je *délka* tohoto sledu. Říkáme, že sled *prochází* vrcholy $v_0, \dots, v_k$ nebo že na něm tyto vrcholy *leží*.

Definice 3.6 (Cesta) *Cesta* z u do v v grafu G je sled vrcholů $(u = v_0, v_1, \dots, v_k = v)$, ve kterém se každý vrchol v_i objevuje pouze jednou.

Definice 3.7 (Váha podgrafu) Nechť M je množina hran ohodnoceného orientovaného grafu (G, w). 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 3.8 (Minimální cesta) Jsou-li u, v vrcholy grafu G, pak *minimální cesta* z u do v je každá cesta, jejíž váha je minimální (tj. žádná cesta z u do v nemá menší váhu). *Vážená vzdálenost* $d^w(u, v)$ vrcholů u, v je váha minimální cesty z u do v.

Z definice 3.8 lze vyčíst, že minimální cesta nemusí být zdaleka nejkratší, z hlediska počtu hran. V případě, že hrany jsou ohodnoceny vzdálenostmi mezi jednotlivými uzly, tak výsledná spočtená minimální cesta je zároveň cestou nejkratší. Kdyby např. byly hrany ohodnoceny časem, který je potřebný k přechodu z vrcholu u do vrcholu v, pak se jedná pouze o minimální cestu a nikoliv i o nejkratší cestu.

Navigace chodců je řešena ve městě, nad kterým je spočtena tzv. Constrained Delaunay triangulation (dále jen CDT), česky nazývána jako Delaunayova triangulace s omezením. Vzhledem k tomu je nutné si nadefinovat další pojmy, a to triangulace a Delaunayova triangulace (zkráceně DT), kterou potřebujeme před nadefinováním pojmu CDT.

Definice 3.9 (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 E je maximální.

Definice 3.10 (DT) *Delaunayova triangulace* $\mathrm{DT}(V)$ definovaná na množině bodů $V \in R^2$ je množina trojúhelníků takových, že

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

Nyní už je nutné zavést si ještě pojem viditelnosti, který je použitý v definici CDT, a následně můžeme definovat i samotný pojem CDT.

Definice 3.11 (Viditelnost) Dva vrcholy x a y v grafu $G = (V, E)$ jsou navzájem *viditelné*, pokud přímka mezi x a y neprotíná žádnou hranu z E.

Definice 3.12 (CDT) Nechť $G = (V, E)$ je planární graf takový, že $E \neq 0$. Triangulace $\Delta = (V, E \cup E')$ je *CDT* grafu G, pokud všechny hrany $xy \in E'$ jsou takové, že body x, y jsou navzájem viditelné v grafu G, a hrana xy splňuje podmínku prázdné opsané kružnice vzhledem k vrcholům viditelným z x a y v grafu G.

4 Algoritmy hledání cest

Naprostá většina grafových algoritmů se zabývá problémem nejkratší cesty [Wik12]. Hledání cesty mezi dvěma vrcholy (nebo uzly) v grafu takové, že součet ohodnocení hran mezi těmito vrcholy je minimální. Například hledání nejrychlejší cesty z jednoho místa do druhého při jízdě autem. V tomto případě vrcholy grafu představují místa (města) a hrany, které jsou váženy podle času potřebného k dosažení další lokality, reprezentují silnice (cesty) mezi jednotlivými místy.

Problémy týkající se nejkratší cesty se velmi často dělí podle toho, mezi jakými vrcholy je nutné nalézt nejkratší cestu, na následující čtyři skupiny:

- *The single-pair shortest path problem* – hledání nejkratší vzdálenosti mezi dvěma vrcholy grafu. Mezi řešení tohoto problému patří například Dijkstrův algoritmus (kapitola 4.1) a taktéž A* vyhledávací algoritmus (kapitola 4.2).
- *The single-source shortest path problem* – hledání nejkratší možné cesty v grafu G mezi vrcholem x a všemi ostatními vrcholy.
- *The single-destination shortest path problem* – hledání nejkratší možné cesty v orientovaném grafu ze všech možných vrcholů do vrcholu x.
- *The all-pairs shortest path problem* – hledání nejkratší cesty mezi všemi možnými dvojicemi vrcholů x_1 a x_2.

Mezi základní grafové algoritmy patří prohledávání do hloubky (Depth-first search, DFS, Cha03]) a prohledávání do šířky (Breath-first search, BFS [Cha03]). Zabývat se však budeme podrobněji pouze prohledáváním do šířky, na jehož myšlence stojí dále uvedené algoritmy.

Breath-first search

Breath-first search (BFS) je strategie, spadající do *single-pair* algoritmů, pro průchod přes uzly v grafu. Díky své jednoduchosti je hojně využíván v mnoha jiných algoritmech, jako jsou např. Dijkstrův algoritmus (kapitola 4.2) nebo Prim-Jarníkův, sloužící k nalezení minimální kostry. V této práci není BFS užíván přímo pro hledání nejkratší cesty, ale velmi pomáhá při řešení následujících problémů a hlavně jejich následných modifikací.

Nechť máme graf $G = (V, E)$ a počáteční vrchol $s \in V$. BFS [Cha03] určuje vrcholy dosažitelné z počátečního uzlu následovně: Nejprve najde všechny uzly, které jsou dosažitelné z počátečního uzlu s přes jednu hranu. Dále najde každý uzel, který je dosažitelný z uzlu s přes dvě hrany. Obecně všechny vrcholy ve vzdálenosti k od počátečního uzlu s jsou navštíveny dříve než uzly se vzdáleností $k + 1$.

Algoritmus nejprve obarví každý vrchol bílou barvou, což znamená, že daný uzel dosud nebyl navštíven. Když poprvé dosáhne vrcholu m, obarví ho na černo, pokud byly navštíveny všechny jeho sousední vrcholy, v opačném případě je obarven šedivě. Tento postup se opakuje a končí v případě, kdy je nalezen hledaný koncový uzel (*single-pair shortest path problem*) nebo pakliže všechny uzly byly již zpracovány (*single-source shortest path problem*), tzn. každý vrchol má černou barvu.

BFS má asymptotickou časovou složitost $O(|V| + |E|)$ a jeho použitím vzniká BFS strom.

4.1 Dijkstrův algoritmus

Dijkstrův algoritmus [LaV06, JNC11], objevený roku 1959 panem E. W. Dijkstrou, patří mezi grafové vyhledávací algoritmy řešící problematiku *single-source* a také *single-pair shortest path* v nezáporně ohodnocených grafech. Tento algoritmus je v podstatě zobecněním breath-first search algoritmu, při kterém algoritmus nepostupuje podle počtu hran, ale podle vzdálenosti od zdrojového uzlu. Představme si, že vrcholy V ohodnoceného grafu G představují města a jeho hrany E reprezentují vzdálenosti mezi každými dvěma městy, jež jsou propojeny silnicemi. Pak Dijkstrův algoritmus nalezne minimální (resp. nejkratší) cestu z města u do města v grafu G.

Nechť $G = (V, E)$ je graf s nezáporně ohodnocenými hranami a vrchol $s \in V$ je počátečním uzlem. Prvním krokem Dijkstrova algoritmu je nastavení hodnoty dosažitelné vzdálenosti všem uzlům. Počátečnímu uzlu je přiřazena nulová vzdálenost a všem zbylým je přiřazena nekonečná vzdálenost, což znamená, že všechny zbylé uzly jsou nedosažitelné z počátečního vrcholu s. Navíc všechny uzly označíme jako nenavštívené a počáteční uzel vložíme do prioritní fronty. Vyjmeme z fronty uzel s nejmenší prioritou (tzn. nejmenší vzdáleností), označíme ho jako navštívený a všem jeho nenavštíveným sousedním uzlům aktualizujeme nastavenou vzdálenost. Pokud nově spočtená vzdálenost je menší než aktuální hodnota vzdálenosti v daném uzlu a uzel je nenavštívený, vložíme ho do prioritní fronty. Pakliže se tento uzel již v prioritní frontě nachází, pouze aktualizujeme jeho vzdálenost a znovu ho do fronty nevkládáme. Dále dochází k opakovanému vyjímání uzlů z prioritní fronty a nastavování jejich vzdálenosti až do doby, kdy najdeme hledaný cílový uzel (*single-pair*) nebo dokud není fronta prázdná (*single-source*), pak algoritmus končí.

Nejjednodušším implementačním řešením je možné dosáhnout asymptotické složitosti $O(V^2)$. Avšak při vhodně zvolené reprezentaci prioritní fronty lze asymptotickou složitost vylepšit až na $O(|E| + |V|log|V|)$.

4.2 A* vyhledávací algoritmus

Rozšířením Dijkstrova algoritmu dostaneme A* algoritmus, prvně popsaný P. Hartem, N. Nilssonem a B. Raphaelem [NR68] v roce 1968, pro vyhledávání optimálních cest v nezáporně ohodnocených grafech, který se oproti Dijkstrovu algoritmu snaží zredukovat celkový počet prozkoumaných uzlů pomocí tzv. heuristického odhadu váhy cesty k dosažení cíle z daného místa [LaV06]. Optimální cestou rozumíme cestu minimální, tzn. nejkratší, nejrychlejší nebo nejoblíbenější cestu v závislosti na reprezentaci ohodnocených hran grafu G.

Nechť $G = (V, E)$ je graf s nezáporně ohodnocenými hranami, vrchol $s \in V$ je počátečním uzlem a vrchol $e \in V$ je koncovým uzlem. V prvé řadě spočteme heuristickou funkci $f(x)$ pro počáteční uzel s, který následně vložíme do prioritní fronty (Algoritmus 4.1, *open_list*). V dalším kroku vyjmeme z prioritní fronty uzel s nejmenší hodnotou heuristické funkce $f(x)$ a uložíme ho do sběrného kontejneru (Algoritmus 4.1, *closed_list*), což znamená, že byl uzel zpracován a již se nebude nijak měnit. Následně pro každý sousední uzel zpracovávaného vrcholu spočteme heuristickou funkci $f(x)$, ale pouze v případě, že se nenachází v již uzavřených uzlech. Výpočet dále zjišťuje, zda sousední uzel není v prioritní frontě. Pokud ano a nově spočtená heuristická funkce je menší než heuristická funkce uzlu v prioritní frontě, původní heuristickou funkci přepíšeme nově spočtenou funkcí $f(x)$ a

aktualizujeme cestu grafem. V opačném případě, tzn. prvek se nenachází ve frontě, přiřadíme sousednímu uzlu spočtenou heuristickou funkci $f(x)$, vložíme ho do prioritní fronty a zároveň ho přidáme do cesty grafem G. Výpočet dále pokračuje ve smyčce, kdy se opět vybere prvek z prioritní fronty (Algoritmus 4.1, *open_list*) s nejmenší hodnotou heuristické funkce $f(x)$ a zpracují se sousední uzly. Algoritmus končí v případě, že jsme zrekonstruovali cestu chodce po nalezení hledaného cílového uzlu e. Každý uzel obsahuje informaci o sousedním uzlu, přes který k němu vede nejkratší cesta (*pedestrian.path* v Algoritmu 4.1). S touto informací můžeme zpětně (z uzlu e do uzlu s) zrekonstruovat nejkratší cestu.

Algoritmus 4.1 A* search algoritmus

Vstup: *pedestrian* obsahuje počáteční i koncový uzel a zároveň nalezenou nejkratší cestu
Výstup: *pedestrian.path* nejkratší nalezená cesta chodce *pedestrian*

```
// Počáteční inicializace pro počáteční uzel
closed_list = empty
open_list.add(pedestrian.start)
Compute heuristic function f of pedestrian.start
// Dokud existují nezpracované uzly, hledáme nejkratší cestu
while open_list is not empty
   current = node with lowest priority of heuristic function f
   // Pokud jsme nalezli cílový uzel, výpočet končí
   if current is the same as pedestrian.destination then
      break
   remove current from open_list and add it to closed_list
   // Pro všechny sousední uzly zpracovávaného uzlu upravíme heuristickou funkci
   for all neighbor of current do
      if neighbor is not in closed_list then
         // Pro nezpracovaný sousední uzel spočteme heuristickou funkci
         Compute heuristic function temporary_f of neighbor
         if neighbor is not in open_list then
            // Pokud soused není ve frontě, nastavíme heuristickou funkci a vložíme do fronty
            Set heuristic function f to temporary_f and add neighbor to open_list
            Add neighbor to pedestrian.path
         else
            if heuristic function temporary_f < heuristic function f of neighbor then
               // Pokud sousední uzel je ve frontě a nově spočtená heuristická funkce je
               // menší než nastavená, aktualizujeme hodnotu funkce na nižší
               Change heuristic function f of neighbor to temporary_f
               Change pedestrian.path
   end for
end while
Reconstruct the path of pedestrian
```

Z popisu průběhu algoritmu je zřejmé, že principielně je Dijkstrův algoritmus totožný s A* algoritmem, protože oba používají prioritní frontu a identickým způsobem vyhledávají sousední uzly. Nejmarkantnější změnou a také v podstatě jedinou oproti Dijkstrovu algoritmu je zavedení heuristické funkce $f_{(x)}$ v A* a to ve tvaru

$$f(x) = g(x) + h(x), \tag{4.1}$$

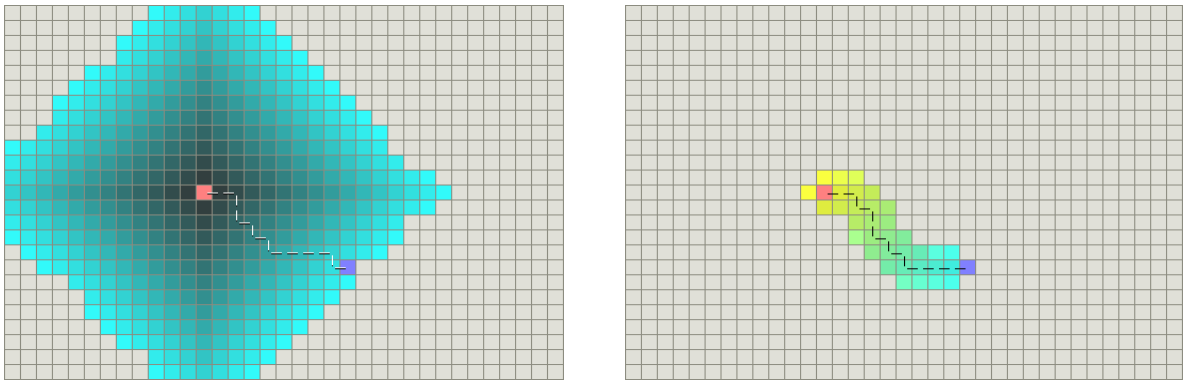
kde $g(x)$ představuje vzdálenost mezi počátečním a současným uzlem, $h(x)$ heuristický prvek, odhadující délku zbylé trasy od současného do koncového uzlu. K výpočtu heuristického prvku lze využít Manhattanskou, diagonální nebo Eukleidovskou metriku [Pat12], kde obzvláště první dvě metriky se používají pro pohyb po mřížce.

Díky heuristickému prvku je A* algoritmus výpočetně rychlejší, protože oproti Dijkstrovu algoritmu navštíví menší počet uzlů (obr.4.1 převzatý z [Pat12], blok Introduction to A*). V případě, že je heuristický prvek $\forall x: h(x) = 0$, dostáváme Dijkstrův algoritmus pro výpočet nejkratší vzdálenosti.

Časová složitost záleží na použité heuristice. V nejhorším případě je exponenciální, ale může být i polynomiální v případě, že prohledávaný prostor je stromem a zároveň splňuje následující podmínku

$$|h(x) - h^*(x)| = O(\log h^*(x)),$$

kde $h^*_{(x)}$ je optimální heuristika, tj. přesná vzdálenost z uzlu s do koncového uzlu.



Obrázek 4.1: Množství navštívených buněk Dijkstrovým (nalevo) a A* (napravo) algoritmem (Růžová barva značí počáteční uzel, tmavě modrá uzel koncový a zbylé barvy odpovídají hloubce prohledávání)

4.3 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus (zkráceně FW [JNC11]) je algoritmus pro hledání nejkratší cesty v ohodnoceném orientovaném (resp. neorientovaném) grafu a patří mezi algoritmy pro tzv. *all-pairs shortest path problem*. FW povoluje záporné ohodnocení hran a zároveň detekuje negativně ohodnocený cyklus, pokud nějaký existuje. Za předpokladu, že máme graf $G = (V, E)$ a neexistuje v něm žádný negativně ohodnocený cyklus, FW algoritmus najde nejkratší cestu mezi každým párem vrcholů.

FW algoritmus je uveden jako algoritmus 4.2. Nejprve inicializuje distanční matici d a matici předchůdců v. Matice d je nastavena s nulami na diagonále, mezi vrcholy spojené hranou je uloženo ohodnocení hrany a všude jinde (tzn., kde hrana nevede) je nastaveno nekonečno. Podobným způsobem je inicializována matice předchůdců v. Rozdíl spočívá v tom, že v místě, kde má distanční matice nekonečnou vzdálenost, bude mít matice v hodnotu nula, a tam, kde matici d přiřazujeme vzdálenost mezi uzly, přiřadíme matici v číslo vrcholu.

Pro takto připravené matice FW algoritmus spustí hlavní výpočet nejkratší cesty mezi všemi vrcholy. Ten se dělí na jednodušší kroky používající jistý druh rekurzivní procedury. Tato myšlenka spočívá v tom, že dočasně označíme vrcholy grafu G jako $V = \{1,2,\dots,n\}$ a zavoláme $d(i,j)$, což je nejkratší vzdálenost z vrcholu i do vrcholu j, která jenom používá vrcholy od 1 do k. K tomu je zapotřebí rekurzivní výraz

$$d(i,j) = min\{d(i,j), d(i,k) + d(k,j)\}, \forall k \in V.$$

Algoritmus 4.2 Floyd-Warshallův algoritmus

```
Výstup: d distanční matice grafu
Výstup: v matice předchůdců
Vstup: graph obsahující konečnou množinu hran a uzlů
// Inicializace distanční matice a matice předchůdců
Initialize d, v to zeros
for i = 0 to graph.nodes - 1 do
   for j= 0 to graph.nodes - 1 do
      if exists edge between node i and j in graph then
         d[i][j] = graph.edge[i][j]
         v[i][j] = i
      else
         d[i][j] = infinity
   end for
end for
// Výpočet nejkratší cesty mezi všemi vrcholy grafu a modifikace matice předchůdců
for k = 0 to graph.nodes - 1 do
   for i = 0 to graph.nodes - 1 do
      for j = 0 to graph.nodes - 1 do
         // Pokud existuje kratší cesta, tak přepiš distanční matici a matici předchůdců
         if d[i][j] > (d[i][k] + d[k][j]) then
            d[i][j] = d[i][k] + d[k][j]
            v[i][j] = v[k][j]
      end for
   end for
end for
```

V pseudokódu 4.2 lze vidět, že Floyd-Warshallův algoritmus pracuje se složitostí $O(n^3)$.

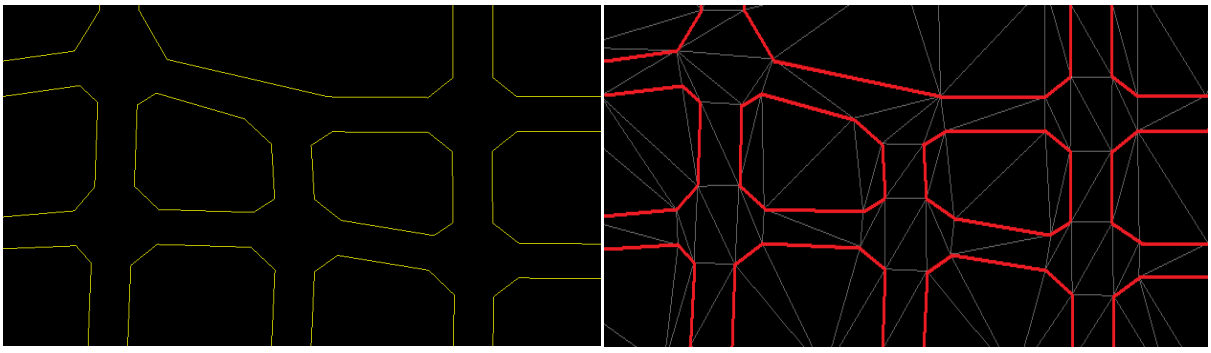
5 Navržená řešení a jejich implementace

Pátá kapitola slouží k popisu řešení s užitím metod ze čtvrté kapitoly a k jejich implementačním zajímavostem v programovacím jazyce C++. Než se ale zaměříme na samotné řešení algoritmů, je důležité nejdříve uvést, že všechna načítání souborů a výpočty uvedené na této straně byly implementovány Ing. Tomášem Vomáčkou nebo Doc. Bedřichem Benešem a jeho výzkumnou skupinou a já jsem se na nich nepodílel. Nejprve si uvedeme datové soubory, které jsou nutné pro správné spuštění programu.

Prvním důležitým datovým souborem je soubor s příponou *.geom*, který obsahuje strukturu virtuálního města. Tím se rozumí svým způsobem jistý druh územního plánu, kde musí být zřetelně zanesené silnice, cesty a bloky budov či budovy samotné. Mezi další potřebná data patří soubory s příponou *.xml*, v nichž jsou uložené informace o chodcích. V prvním z těchto souborů jsou uloženy vztahy mezi chodcem ze sociální skupiny m a všemi čtvrtěmi, které se ve virtuálním městě nacházejí. V druhém z těchto souborů je uložen vztah chodce ze sociální skupiny m k chodci ze sociální skupiny n. Předposledním důležitým souborem je obrázek s příponou *.png*, který znázorňuje jednotlivé rozdělení čtvrtí ve městech. V neposlední řadě je klíčový i soubor s příponou *.txt*, obsahuje údaje o chodcích (tzn. planární souřadnice počátečního a koncového bodu jeho polohy ve městě a také informaci o tom, ke které sociální skupině náleží).

Při inicializování samotného programu se nejprve načte struktura města, jejíž zpracování patří k nejsložitějším výpočtům, které program řeší. V načtené struktuře bloků a silnic (Obr. 5.1, nalevo) nejprve program určí důležité vrcholy a spočte jejich planární souřadnice. Důležitými vrcholy rozumíme takové vrcholy, které se nacházejí na rozích bloků budov (resp. křižovatek), tzn. vrcholy, v nichž se protínají úsečky reprezentující hranici mezi budovou a silnicí. Tyto nalezené vrcholy pak slouží jako opěrné body, pro něž je pomocí knihovny CGAL spočtena CDT (Obr 5.1 napravo) a každému spočtenému trojúhelníku je přiřazena informace ze souboru s rozdělením čtvrtí o tom, jakou čtvrť reprezentují.

Dále se načtou soubory se vztahy, obsahující oblíbenost mezi sociálními skupinami a oblíbenost čtvrti chodcem. Tyto vztahy jsou uloženy do dvou dvourozměrných polí typu *float*. Na závěr se načtou do datové struktury *vector* (datový kontejner knihovny STL) instance reprezentující jednotlivé chodce a naším úkolem je navrhnout algoritmy pro hledání nejkratší cest ve virtuálním městě pro tyto chodce a modifikovat je tak, aby počítaly i s oblíbeností čtvrti.



Obrázek 5.1: Nalevo struktura města, napravo město se spočtenou CDT

5.1 A* search algorithm

Návod, jak řešit A* nám dává kapitola 4.3, proto se zde budeme zabývat převážně implementačními zajímavostmi.

Jak na začátku páté kapitoly bylo zmíněno, vycházíme z virtuálního města pokrytého CDT. Nemáme tedy žádný graf, na kterém by se dal použít A* algoritmus, a proto si ho musíme vytvořit. Nejjednodušším způsobem je představa, kdy každý trojúhelník reprezentuje uzel grafu a hrany jsou reprezentovány jako vzdálenosti jejich středů, který budeme v případě A* algoritmu používat.

A* algoritmus je implementován ve třídě *streetLayer.cpp* v metodě *computeTrajectory(CPedestrian &ped, bool const shortestPath).* Parametr *ped* je objekt třídy *pedestrian.cpp* a nese sebou informace o počátečním a koncovém uzlu a také informaci o příslušnosti k sociální skupině. Druhým parametrem je *shortestPath* typu *bool*, který využívá až volaná metoda *float getBasicAreaRating().* Ta slouží ke spočtení vzdálenosti mezi dvěma trojúhelníky. Vstupními parametry jsou dva trojúhelníky pro výpočet ohodnocení hrany, dále parametr *ped*, kvůli příslušnosti k sociální skupině a v neposlední řadě již zmíněný parametr *shortestPath*. Ten nám udává, jestli hledáme nejkratší cestu či minimální v jiném smyslu (kapitola 5.1.1 a 5.1.2) a podle toho spočte ohodnocení hran.

Důležité je také, že prioritní frontu reprezentujeme haldou z knihovny STL (knihovna C++) a že výslednou trasu ukládáme do objektu *ped* v podobě sběrného kontejneru *vector* knihovny STL.

Našim cílem je, aby A* algoritmus nehledal pouze nejkratší cestu, ale bral v potaz i oblíbenost dané čtvrti chodcem. Z tohoto důvodu je potřebné rozšířit metodu *getBasicAreaRating()* tak, aby nehledala pouze vzdálenost mezi středy dvou trojúhelníků, ale aby k této vzdálenosti připočetla funkce vyjadřující oblíbenost, což řeší následující dvě podkapitoly.

5.1.1 Váhová funkce

Navržené řešení

Nejkratší cestu je možné upravit za použití vzdáleností mezi jednotlivými místy (ohodnocení hran) a funkcí reprezentující oblíbenosti čtvrti, která nabývá hodnot z intervalu

$$I = \langle -1, 1\rangle, \text{kde} \begin{cases} -1 & \text{nejméně oblíbená čtvrť} \\ 1 & \text{nejoblíbenější čtvrť} \end{cases}.$$

Za použití výše zmíněných parametrů vytvoříme váhovou funkci vyjadřující oblíbenost

$$p(a(x), w) = a(x) \cdot w,$$

kde $x \in I, a_{(x)}$ je polynom a w je ohodnocení hran (zpravidla vzdálenost mezi uzly), Tuto funkci následně budeme používat místo ohodnocení hran.

Navíc pro náš problém předpokládáme, že graf je souvislý, neorientovaný a má nezáporně ohodnocené hrany. V našem případě jsou první dva předpoklady splněny. Problém však nastává u nezápornosti ohodnocení hran. Je zřejmé, že bychom mohli dostat výsledek váhové funkce v záporném tvaru, a proto ještě musíme náš interval transformovat tak, aby byl

podintervalem nezáporných čísel. Nejlépe se jeví transformace intervalu I na interval I_c. Od jedničky odečteme libovolné číslo intervalu I, čímž dostaneme transformovaný interval

$$I_c = \langle 0, 2\rangle, \text{kde} \begin{cases} 0 & \text{nejoblíbenější čtvrť} \\ 2 & \text{nejméně oblíbená čtvrť} \end{cases}. \tag{5.1}$$

Tato transformace intervalu, kdy nejoblíbenější čtvrť nabývá nejmenší hodnoty, je velmi důležitá pro ohodnocení hran. Požadujeme totiž, aby nejmenší ohodnocení hran měly nejoblíbenější čtvrtě, čehož dosáhneme s intervalem I_c. Navíc při aplikaci nového intervalu na náš problém nevznikne singularita v podobě záporně ohodnocené hrany. Tímto dojde k drobné změně ve váhové funkci

$$p\big(a_{(x)}, w\big) = a_{(x)} \cdot w, \tag{5.2}$$

kde $x \in I_c$.

Implementace

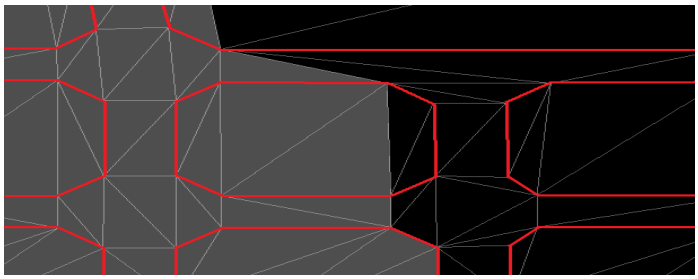
V našem případě nahradíme polynom $a_{(x)}$ polynomem prvního stupně, jehož pomocí dosáhneme pravděpodobně nejlepšího vztahu mezi vzdáleností a oblíbeností cesty, takže váhová funkce (5.2) bude následně ve tvaru

$$p(x, w) = x \cdot w,$$

kde $x \in I_c$, a bude zastupovat prvek $g_{(x)}$ v heuristické funkci (4.1).

Váhovou funkci lze samozřejmě ovlivňovat stupněm polynomu $a_{(x)}$, kde čím vyšší bude tento polynom, tím více se bude výsledná trasa chodce blížit k trase funkce oblíbenosti (viz 5.1.2). V opačném případě, kdybychom stupeň snižovali, tak se výsledná trasa bude stále více přimykat k nejkratší možné cestě.

Již uvedená metoda *getBasicAreaRating()* nepočítá pouze váhovou funkci, ale zároveň řeší i přechody z jedné čtvrti do druhé. Každý trojúhelník z CDT má pevně stanovenou náležitost ke čtvrti, což je při vykreslování řešeno odlišnými barvami povrchu trojúhelníků (obr.5.2). Jak již bylo zmíněno v kapitole 5.1, počítají se v této metodě vzdálenosti mezi středy trojúhelníků. Tuto informaci využijeme i pro případ přechodu z jedné čtvrti do jiné. Pomocí vektorového součinu určíme obsah obou trojúhelníků. Následně spočteme jejich poměr a na závěr rozdělíme délku mezi středy v poměru obou obsahů. Každou část vynásobíme příslušnou funkční hodnotou oblíbenosti k dané čtvrti a obě hodnoty sečteme, čímž dostaneme ohodnocení hrany dvou trojúhelníků při přechodu mezi dvěma čtvrtěmi.



Obrázek 5.2: Hranice mezi čtvrtěmi

5.1.2 Funkce oblíbenosti

Návrh řešení

Nejkratší cesta počítala pouze se vzdáleností mezi jednotlivými vrcholy grafu G a vůbec nepočítala s oblíbeností čtvrtí, což je jeden možný extrém, který je vyřešen váhovou funkcí v kapitole 5.1.1. Avšak existuje i druhý možný extrém, kterému budeme říkat funkce oblíbenosti. Zde oproti nejkratší cestě vzdálenost nehraje žádnou roli a záleží pouze na oblíbenosti čtvrtí.

Stejně jako u váhové funkce i zde předpokládáme, že graf je souvislý, neorientovaný a má nezáporně ohodnocené hrany. Z třetího předpokladu vyplývá, že budeme muset opět transformovat původní interval I na interval

$$I_c = \langle 0, 2 \rangle, \text{kde} \begin{cases} 0 & \text{nejoblíbenější čtvrť} \\ 2 & \text{nejméně oblíbená čtvrť} \end{cases}.$$

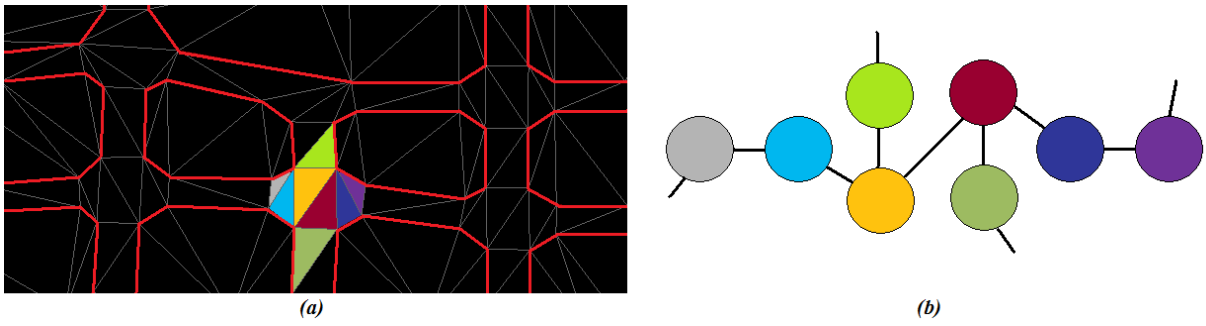
Váha hran bude nyní závislá pouze na oblíbenosti čtvrti a její hodnota bude z intervalu I_c.

Implementace

Tuto modifikaci nejkratší cesty lze opět řešit pomocí metody *getBasicAreaRating()*, kde použijeme naprosto totožnou taktiku jako u váhové funkce. Případ přechodu z jedné čtvrti do druhé bude řešen stejně a použijeme i tutéž váhovou funkci (5.2). Jediným rozdílem bude, že ohodnocení hran bude mít váhu 1, čímž nám zůstane pouze funkce oblíbenosti.

5.2 Floyd-Warshall

Floyd-Warshallův algoritmus je popsán v kapitole 4.4, a proto se zaměříme hlavně na jeho implementaci a návrh lepšího řešení. V případě FW zvolíme podobně jako u A* algoritmu nejlehčí možný pohled na reprezentaci grafu. Každý trojúhelník bude představovat vrchol grafu a hrany mezi vrcholy budou ohodnoceny vzdálenostmi mezi středy trojúhelníků (Obr. 5.3).



Obrázek 5.3: (a) Volba každého trojúhelníku jako vrchol grafu, (b) Reprezentace grafu

Floyd-Warhallův algoritmus je implementován ve třídě *graphAlgorithms.cpp* jako všechny zbylé metody, které budou zmíněny do konce páté kapitoly. Přesněji v metodě *computeFloydWarshall()*, jehož vstupním parametrem jsou všechny trojúhelníky ležící na silnicích. Ty nalezneme pomocí metody *computeStreetFaces()*, jejíž vstupním parametrem je objekt chodce, který obsahuje informaci o trojúhelníku ležícím na silnici. Od tohoto

trojúhelníku hledáme všechny zbylé trojúhelníky pomocí BFS algoritmu a uloží je do *vectoru*. Jakmile známe všechny trojúhelníky ležící na silnicích, metoda *computeFloydWarshall()* inicializuje distanční matici typu *float* a matici předchůdců typu *int*, pro něž je následně spuštěn samotný FW algoritmus. Hledání výsledné cesty z vrcholu u do vrcholu v pak řeší rekurzivní metoda *path()*.

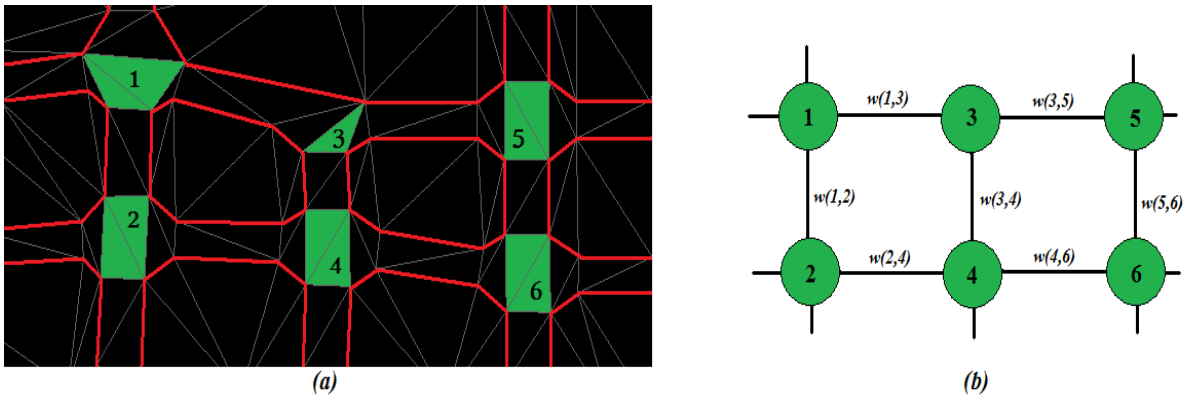
Časová složitost Floyd-Warshallova algoritmu je kubická a v případě naší reprezentace grafu bude výpočet trvat velmi dlouho. Virtuální město může být reprezentováno statisíci trojúhelníky, z nichž každý je považován za uzel, což je možné řešit tímto způsobem pro naše malé virtuální město, ale je to nevhodný přístup pro reálnou aplikaci na nějakém velkém městě jako je např. Praha. Z toho důvodu chceme snížit počet vrcholů reprezentující graf, abychom dosáhli urychlení výpočtu, k čemuž slouží myšlenka navigačního grafu (Navigation graph, [YM*08]) nebo Buněčného a portálového (Cell and Portal graph, CPG, [PA*08]).

5.2.1 Navigační graf

Návrh řešení

V grafové teorii se nejčastěji používá příklad, kdy vrcholy grafu reprezentují města a ohodnocené hrany mezi nimi zastupují vzdálenosti silnic, které vedou mezi městy. A tento přístup se pokusíme aplikovat nikoliv na města a silnice, ale na křižovatky a ulice mezi nimi. V datových strukturách se tomuto přístupu říká navigační graf (zkráceně NG [YM*08]). Díky NG můžeme zredukovat přebytečné uzly a urychlit tak výpočet. Pojmem přebytečné uzly rozumíme takové uzly, které leží na silnici mezi křižovatkami.

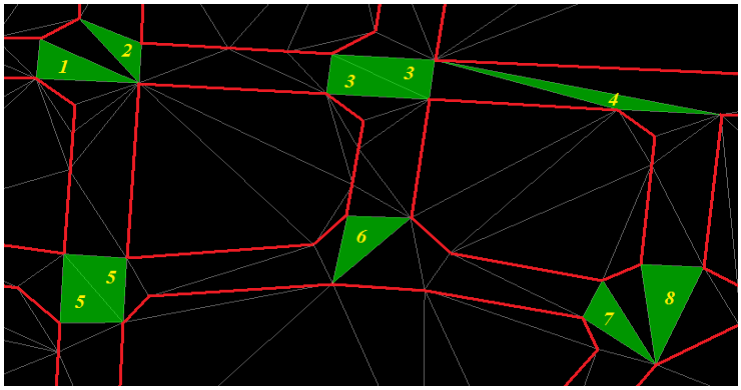
Nejprve je potřebné najít trojúhelníky, které leží na křižovatce a přiřadit jim informaci o tom, na které křižovatce leží (viz obr. 5.4 (a)). Dalším problémem je zjistit mezi kterými křižovatkami vede cesta. Při znalosti těchto dvou věcí lze sestrojit navigační graf (obr. 2.4 (b)), jehož hrany jsou v tomto případě vzdáleností mezi křižovatkou x a y.



Obrázek 5.4: (a) Nalezené křižovatky, (b) Navigační graf

Implementace

Než budeme moci vytvořit reprezentaci NG, musíme předzpracovat všechny trojúhelníky ležící na křižovatce, které jsou vstupním parametrem metody *computeCrossRoads ()*. V první části program určí ze vstupního parametru všechny trojúhelníky reprezentující křižovatky a přiřadí jim číslo křižovatky. Jsou to všechny trojúhelníky, které nemají ani jednu hranu s omezením. Hrana s omezením je taková hrana trojúhelníku, která tvoří hranici mezi blokem budov a silnicí. Je zároveň zaručeno, že minimálně jeden takový trojúhelník leží na každé křižovatce a zároveň, že leží pouze na křižovatce. Může se stát, že na křižovatku reprezentuje více trojúhelníků, ale stejný index dáme pouze těm, které spolu sousedí (Obr 5.5). Tento způsob zaručuje stabilitu výpočtu, protože mohou existovat i jednotrojúhelníkové cesty a my nemůžeme jednoznačně říci, zda leží na křižovatce nebo mimo ni.



Obrázek 5.5: Reprezentace křižovatek i s jejich indexy

Dalším krokem této metody je určení silnic mezi křižovatkami. Tuto část řeší metoda pomocí modifikovaného algoritmu BFS, který najde mezi křižovatkami silnice, přiřadí jim identifikační číslo silnice a uloží je do dvourozměrného pole. Do jednorozměrného pole si ukládáme délku silnice, tzn. vzdálenost od jedné křižovatky do druhé.

Tímto máme všechny důležité údaje předzpracované a můžeme spustit metodu *navigationGraph(),* která nejprve inicializuje distanční matici typu *float* a matici předchůdců typu *int*. Rozdílem oproti FW je, že vrcholy grafu nyní reprezentují křižovatky a ohodnocení nesou vzdálenost mezi křižovatkou u a křižovatkou v. Na závěr pak metoda zahájí samotný výpočet FW, nyní však pro mnohem menší počet vrcholů.

Pro vyhledání nalezené cesty grafem z trojúhelníku x do trojúhelníku y pak slouží rekurzivní metoda *pathNG(),* která nalezne minimální cestu z křižovatky u do křižovatky v. Avšak trojúhelníky x a y nemusí ležet na křižovatce, proto před spuštěním metody *pathNG()* spustíme metodu *graph.findCrossFace()*, která najde nejbližší křižovatku k trojúhelníkům x a y. Tato volba vede k minimalizaci chyby, protože potřebujeme odhadnout jakou křižovatkou vede minimální cesta.

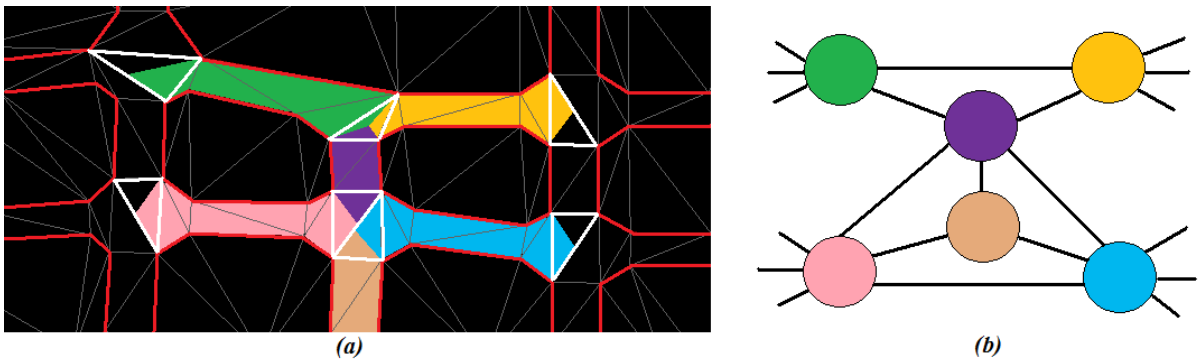
5.2.2 Buněčný a portálový graf

Návrh řešení

Přístup buněčného a portálového grafu (zkráceně CPG [PA*08]) je velmi podobný přístupu navigačního grafu. Jediným rozdílem je, že se v případě CPG vrcholy grafu stanou celé silnice a hranami budou křižovatky, které pouze ponesou informaci, že z ulice x se můžeme dostat do y a za jaké náklady.

V tomto přístupu je nutné najít nejdříve najít jednotlivé křižovatky jako u navigačního grafu a z nich následně určit jednotlivé cesty vedoucí mezi nimi (obr. 2.4 (a)). Nalezené cesty tvoří reprezentaci vrcholů grafu (obr. 2.4) a křižovatky tvoří hrany grafu. Zde je nutné podotknout, že trojúhelníky reprezentující křižovatku mohou náležet dvěma až třem ulicím. Z toho důvodu mají některé trojúhelníky na obr 2.4 (a) více barev (hrany bíle zvýrazněny). Na závěr vytvoříme CPG (obr. 2.4 (b)) reprezentaci grafu, jehož hrany jsou také ohodnocené. Pouze nejsou ohodnocení kvůli přehlednosti uvedeny v obrázku 2.4 (b).

Na obrázku 2.4 (a) jsou vidět nalezené ulice. Každá ulice tvoří vrchol grafu podle svého identifikačního čísla. Ze zelené ulice (Obr. 2.4 (a)) se můžeme přes křižovatku dostat do fialové ulice (Obr. 2.4 (a)), proto tato křižovatka bude tvořit hranu mezi vrcholy reprezentující zelenou a fialovou ulici (Obr. 2.4 (b)).



Obrázek 2.4: (a) Nalezené ulice, (b) CPG

Implementace

Totožně jako u navigačního grafu spustíme nejdříve metodu *computeCrossRoads ()*, která nám předzpracuje data (kapitola 5.1.2).

Následně můžeme metodou *cellAndPortalGraph()* inicializovat distanční matici typu *float* a matici předchůdců typu *int*, kde vrcholy nyní reprezentují celé silnice. Zajímavé je ohodnocení hran, které je trochu nezvyklé. Při přechodu z ulice u do ulice v má hrana mezi nimi ohodnocení délky silnice ze které vycházíme, tedy délku ulice u.

Hledání cesty grafem je pak prováděno rekurzivní metodou *pathCPG()* a výsledná cesta je uložena do datového kontejneru *vector*, který je následně uložen do objektu chodce.

6 Experimenty a výsledky

V této kapitole jsou umístěny obrázky nalezených minimálních cest v programu EcoSim, případné komentáře k obrázkům a na závěr časová náročnost jednotlivých metod.

Zdrojové kódy programu EcoSim jsou implementovány ve vývojovém prostředí Microsoft Visual Studio 2008 Professional za pomoci programovacího jazyku C/C++. Pro správný chod programu byla použita knihovna CGAL 3.9 a technologie CUDA 4.0.

Implementace a testování dat byla prováděna na notebooku Asus ze série M50VC, který disponuje sestavou:

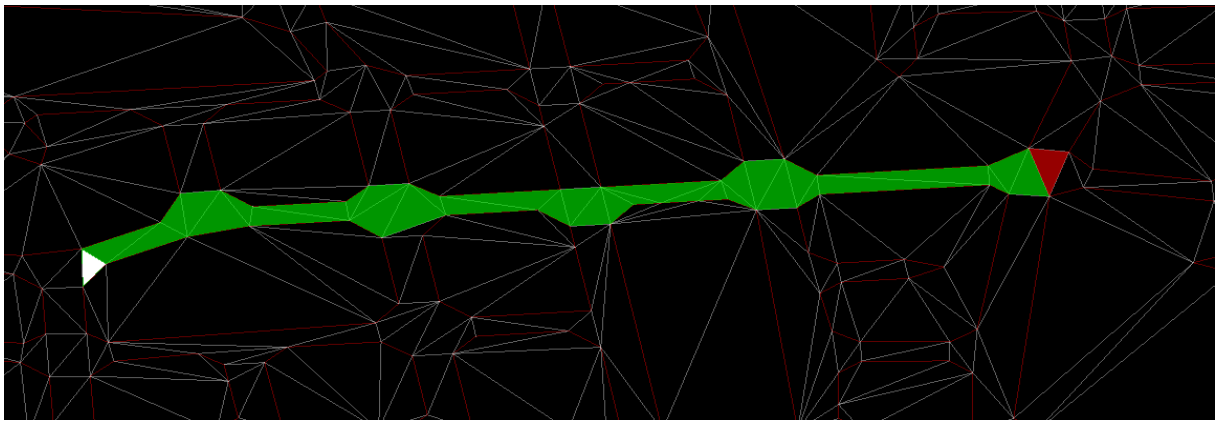
Procesor:	Intel Core2 Duo P7350 o 2.00 GHz
Operační paměť:	4,00 GB
Grafická karta:	nVIDIA GeForce 9300M GS 512MB
Operační systém:	Windows Vista Home Premium 32 bit

V následujících obrázcích jsou nalezené cesty barevně odlišeny zaprvé kvůli vyšší přehlednosti a zároveň kvůli příslušnosti chodce k sociální skupině. Žlutá barva představuje trasu chodce z první, zelená z druhé, modrá ze třetí, tyrkysová ze čtvrté a fialová z páté sociální skupiny. Navíc jsou na obrázku vyznačeny červené a bílé trojúhelníky. Červený trojúhelník je počátečním uzlem a bílý trojúhelník koncovým.

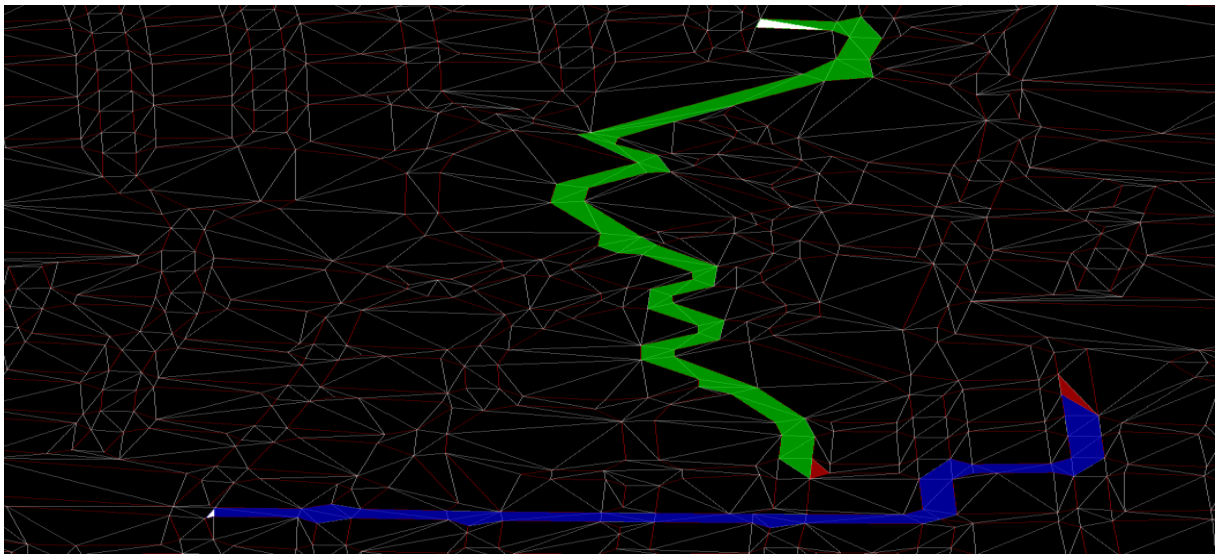
6.1 Nejkratší cesta

Jestliže existuje právě jedna nejkratší cesta, pak by všechny testované metody (A*, FW, CPG a NG) měly nalézt stejnou nejkratší cestu. V některých případech tomu tak opravdu je (Obr. 6.1), ale existují případy, kdy se nalezené cesty liší (Obr 6.2-6.4).

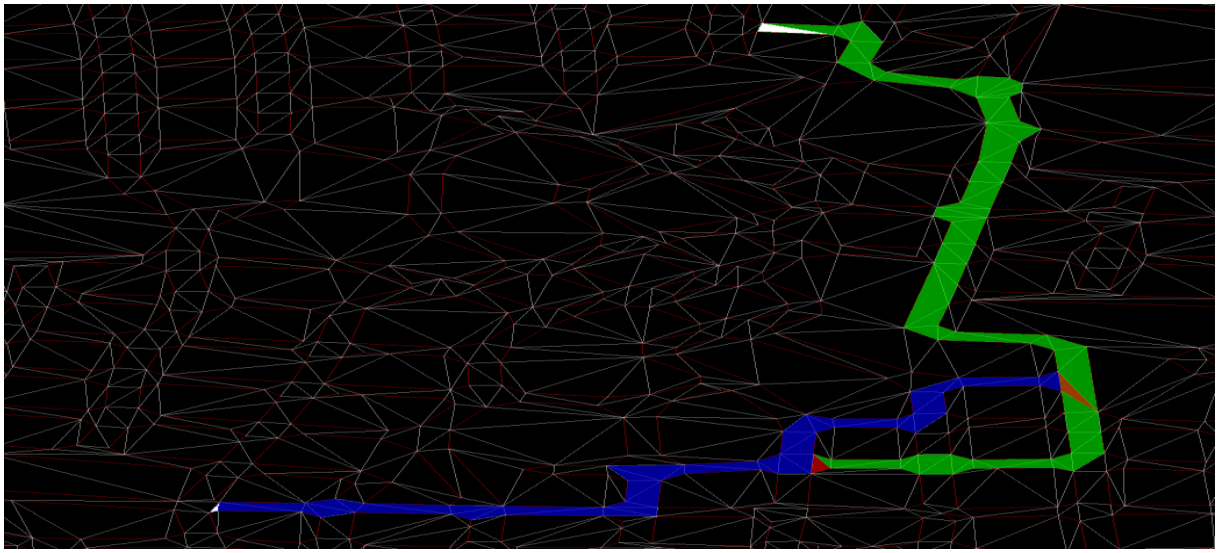
Na obrázcích 6.2-6.4 jsou vykresleny dvě různé trasy dvou různých chodců z důvodů úspory místa. Je vidět, že algoritmy A* a FW naleznou v každém případě právě nejkratší cestu (Obr. 6.2), avšak metody CPG a NG ji nalézt nemusí (Obr 6.3 a 6.4). Navíc nemůžeme přesně rozhodnout, která z těchto metod dává lepší výsledky. Je možné, že CPG (Obr. 6.3, modrá) najde lepší cestu chodci a než NG (Obr. 6.4, modrá), ale také se může stát, že NG (Obr. 6.4, zelená) nalezne kratší cestu chodci b než CPG (Obr. 6.3, zelená).



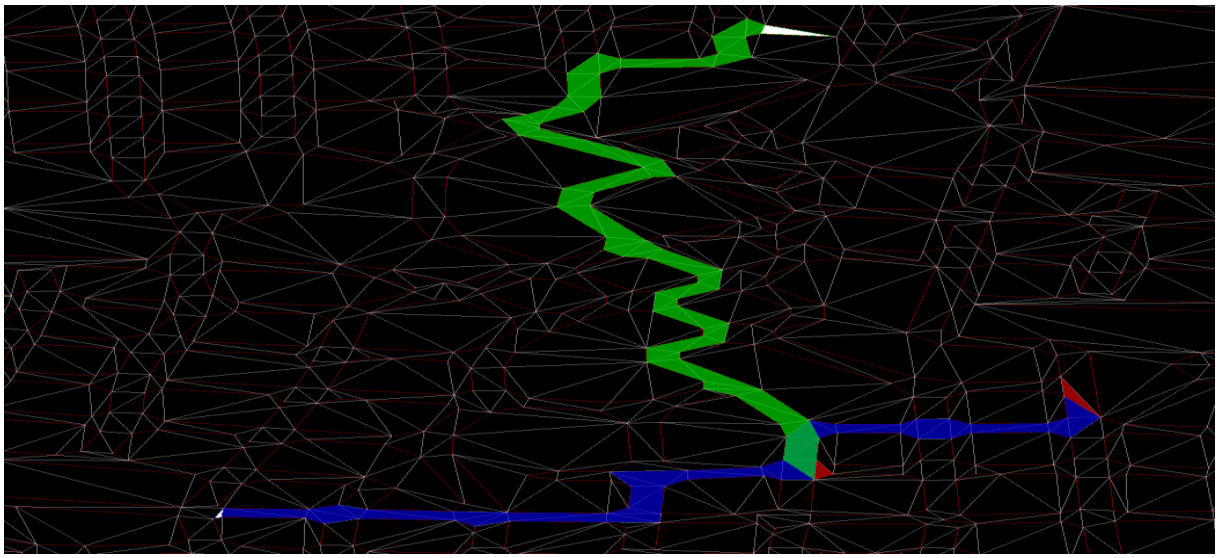
Obrázek 6.1: Nejkratší cesta



Obrázek 6.2: Nejkratší cesta nalezená A* a FW algoritmem



Obrázek 6.3: Nejkratší cesty nalezené přístupem CPG



Obrázek 6.4: Nejkratší cesty nalezené přístupem NG

CPG v některých případech nenajde nejkratší cestu kvůli své definici (kapitola 5.2.2). Při přechodu z první ulice do druhé započítáváme celou délku první ulice do ušlé vzdálenosti. Tím se dopustíme v některých případech chyby, protože potřebujeme přejít jeden trojúhelník, abychom se dostali na další ulici, ale započteme všech např. deset, které reprezentují ulici. Druhé chyby se můžeme dopustit při nalezení cílové ulice, protože algoritmus již nezpracovává vzdálenost této ulice. Kvůli tomu není důležité, z jaké strany do ulice vstoupíme, algoritmus se pouze snaží minimalizovat vzdálenosti mezi první a poslední ulicí, v nichž může být zanesena chyba, kvůli reprezentaci CPG.

V případě NG je chyba způsobena metodou *graph.findCrossFace()* zmíněnou v kapitole 5.2.1, v níž se snažíme k danému počátečnímu a koncovému uzlu nalézt nejbližší trojúhelník reprezentující křižovatku, které představují vrcholy grafu. Může se ovšem stát, že nejkratší cesta nevede přes tyto nejbližší křižovatky, nýbrž přes ty vzdálenější, čímž se dopustíme chyby a algoritmus následně nemusí nalézt nejkratší cestu. Avšak tento problém lze vyřešit tím, že nebudeme používat pouze nejbližší křižovatky, ale také ty vzdálenější a vzájemně je kombinovat. Tím nalezneme čtyři různé cesty, z nichž vybereme tu nejkratší.

6.2 Váhová funkce

Při váhové funkci už potřebujeme znát i hodnoty oblíbeností různých čtvrtí chodci a zde uvedené výsledky byly testovány na matici oblíbeností

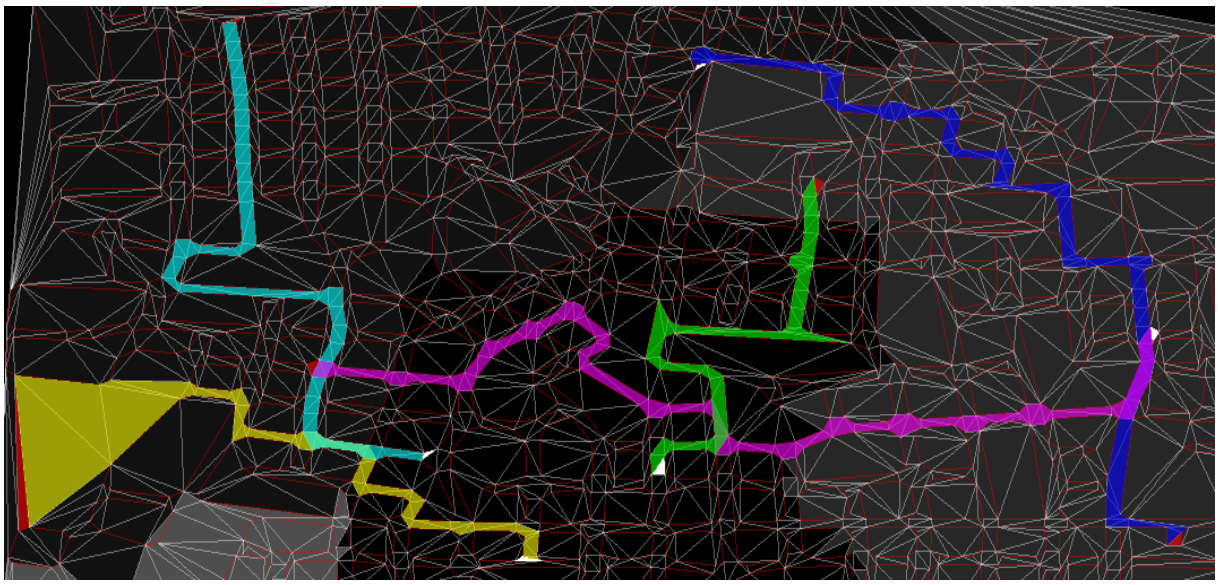
$$M = \begin{pmatrix} 1 & 0.3 & -0.4 & 0.48 & 0.7 \\ 0.5 & 1 & 0.4 & 0.1 & 0.5 \\ -0.2 & 0.37 & 0.99 & 0.3 & -0.5 \\ 0.2 & -0.52 & 0.41 & 0.87 & -1 \\ 1 & 0.2 & -0.4 & 0 & 0.5 \end{pmatrix}, \tag{3}$$

kde sloupce reprezentují čtvrti a řádky chodce. To znamená, že na pozici 1,2 je hodnota oblíbenosti chodce z první sociální skupiny ke čtvrti druhé sociální skupiny. Připomeňme si, že maximální kladná hodnota je 1 a reprezentuje jeho nejoblíbenější oblast, kterou chce pokud možno navštívit. Nejneoblíbenější čtvrť pak reprezentuje hodnota -1 a hodnoty oblíbenosti se pohybují na intervalu $\langle -1,1 \rangle$. Navíc z matice M lze vyčíst, že naše virtuální město obsahuje pět čtvrtí (Obr. 6.5) a pět sociálních skupin. V obecném případě nemusí být počet sociálních skupin stejný jako počet čtvrtí.

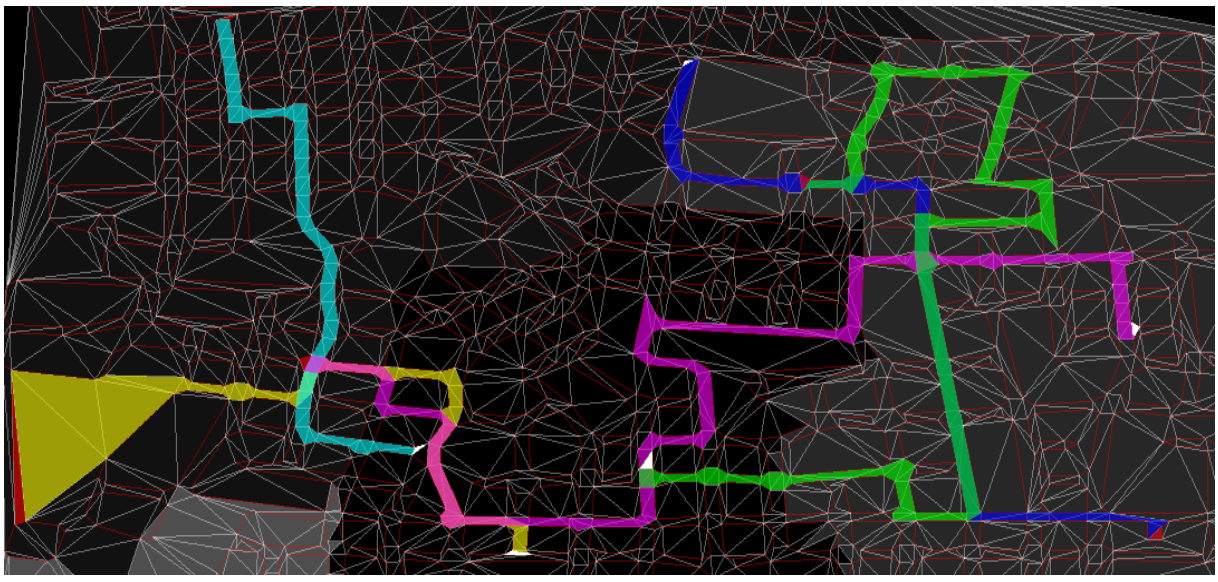
U matice M je uvedeno jakou barvou jsou vykresleny čtvrtě a chodci pro lepší orientaci v následujících obrázcích.

Obrázek 6.5: Rozdělení virtuálního města podle čtvrtí

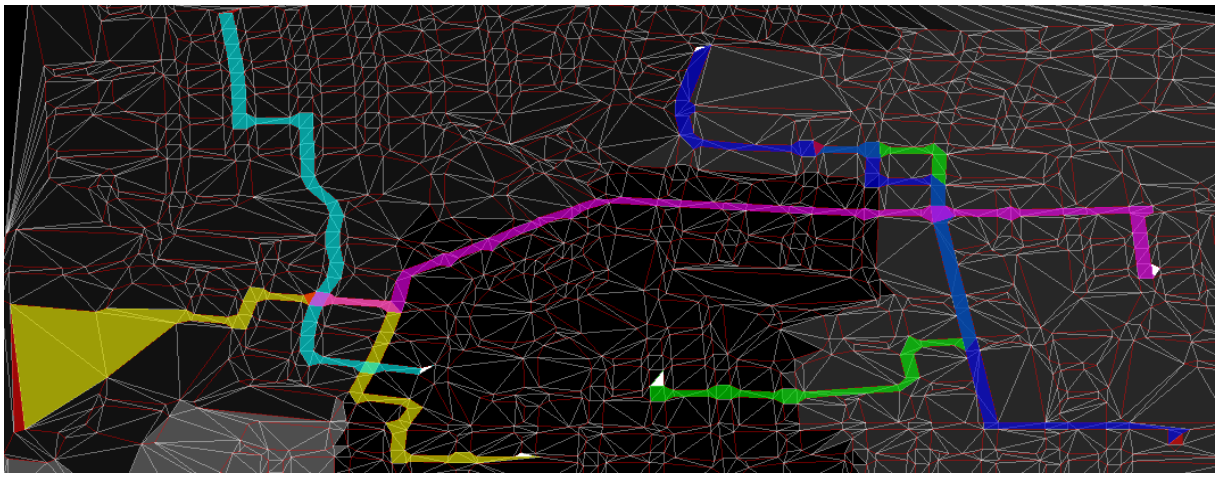
Nejprve se zaměříme na globální pohled hledání trasy a porovnáme výsledky různých metod. Každá metoda určí odlišnou cestu, což je patrné už na první pohled (Obr. 6.6 - 6.9). Jedinou výjimkou jsou metody CPG a NG, jejichž nalezené cesty jsou totožné nebo je mezi nimi malý rozdíl. Na první pohled vypadá lépe výsledek přístupu CPG a NG (Obr. 6.8 a 6.9) než výsledky A* (Obr. 6.6) nebo Floyd-Warhallova (Obr.6.7) algoritmu. Avšak po důkladnějším zkoumání zjistíme, že ačkoliv výsledné cesty A* a FW (Obr. 6.6, 6.7) vypadají opticky hůře, jsou naprosto v pořádku. To je zapříčiněné tím, že při reprezentaci ohodnocení hran v nejoblíbenější čtvrti nabývá hrana nulové hodnoty. Proto se může stát, že v nejoblíbenější čtvrti mohou vznikat různě dlouhé cesty, jejichž součet ohodnocení hran v reprezentačním grafu stále dává nulu. Touto reprezentací můžeme dosáhnout cesty, která vypadá, jako kdyby se chodec procházel po své nejoblíbenější čtvrti (Obr. 6.7 zelená a fialová trasa). Kdybychom nechtěli připustit tuto trasu, tak pouze upravíme interval (5.1) z nezáporného na kladný.



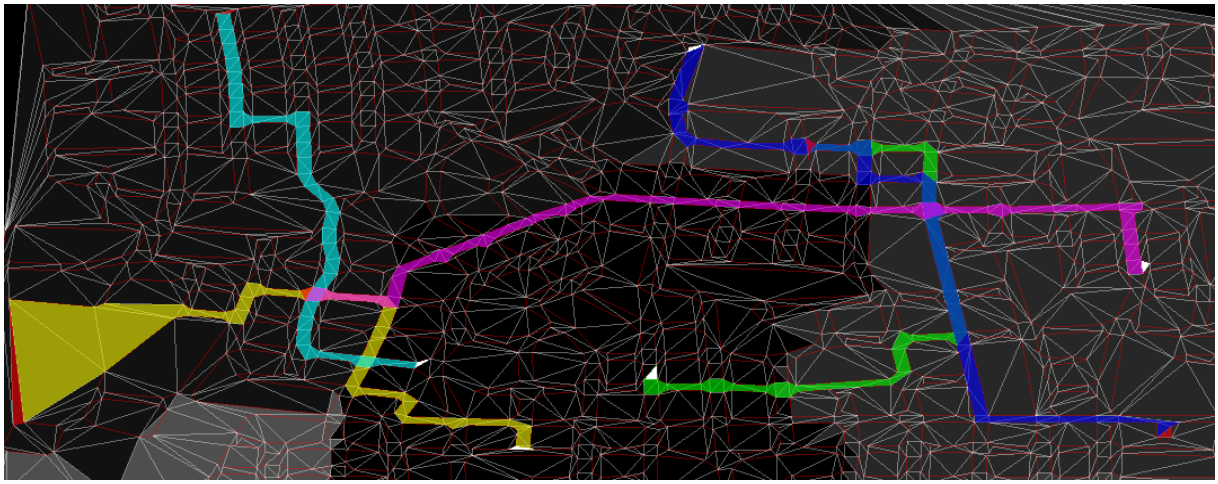
Obrázek 6.6: Nalezené trasy pro 5 různých chodců z různých sociálních skupin pomocí A*



Obrázek 6.7: Nalezené trasy pro 5 různých chodců z různých sociálních skupin pomocí FW

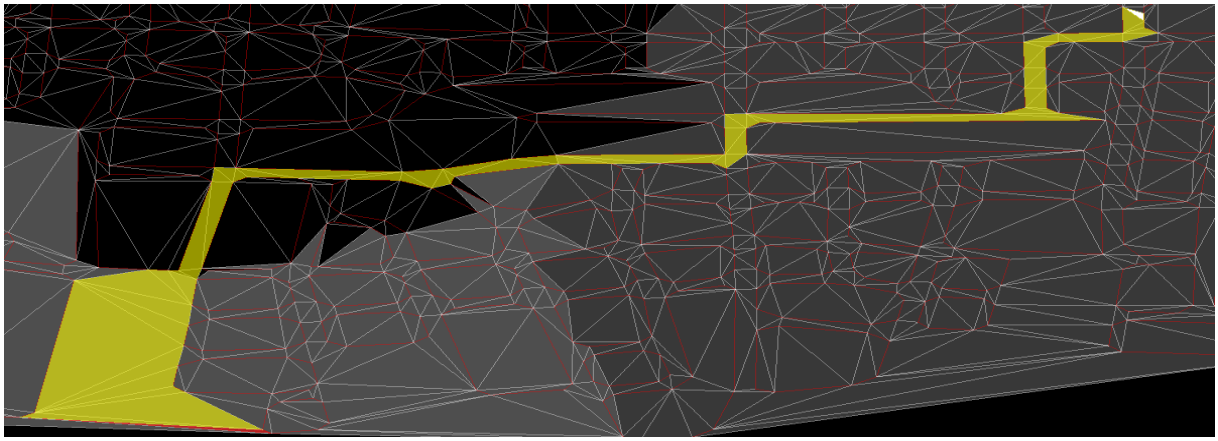


Obrázek 6.8: Nalezené trasy pro 5 různých chodců z různých sociálních skupin přístupem NG

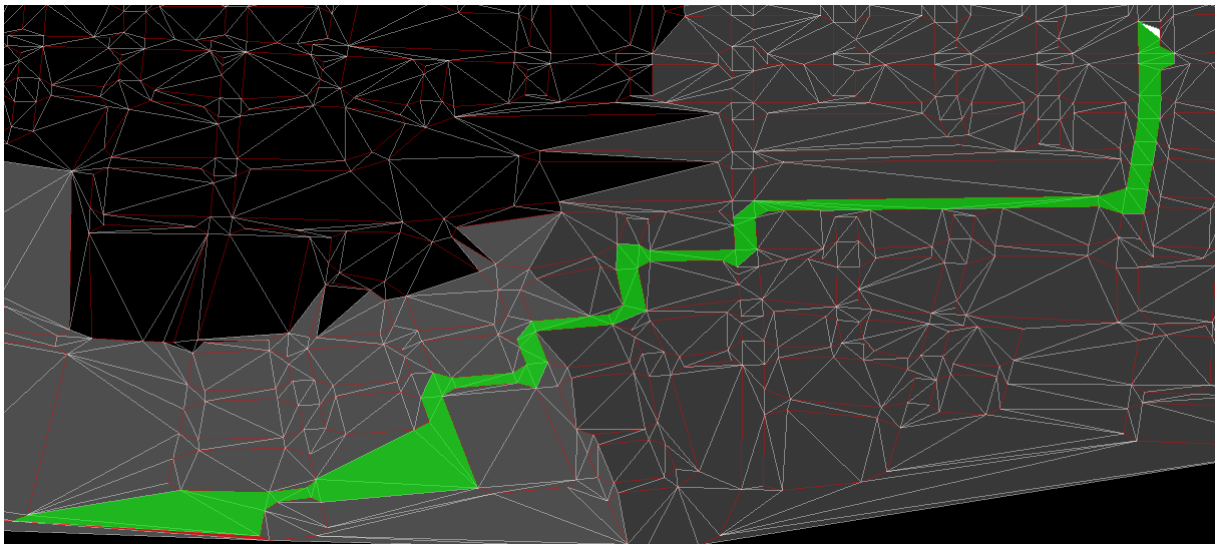


Obrázek 6.9: Nalezené trasy pro 5 různých chodců z různých sociálních skupin přístupem CPG

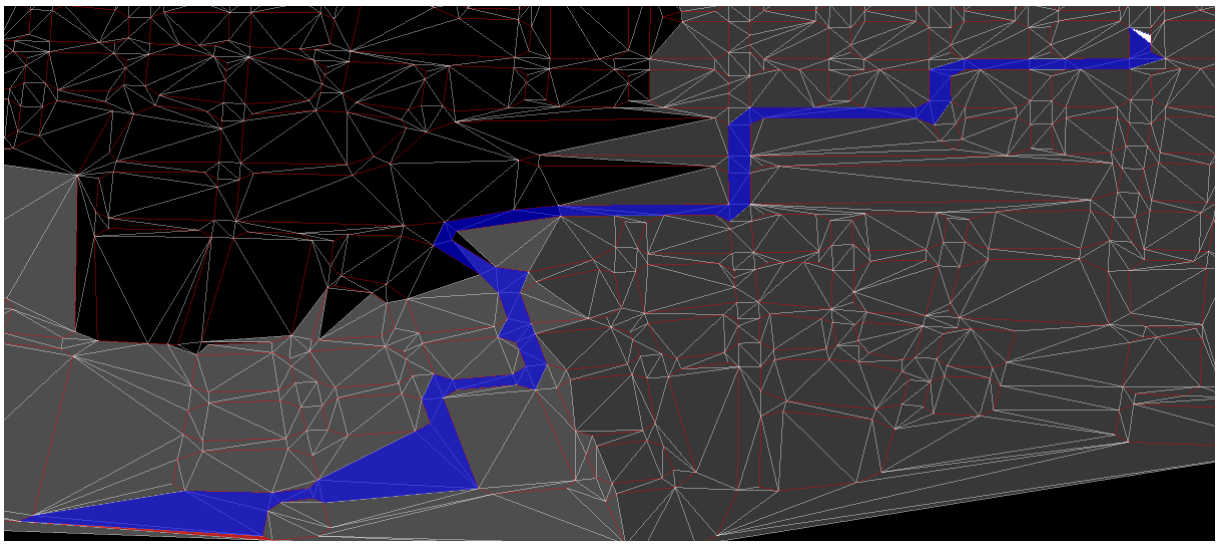
Dále se zaměříme na hledání pouze jedné cesty a budeme pozorovat, jak se bude nalezená cesta měnit v závislosti na různých chodcích, jejich vztazích k různým čtvrtím a také rozdílných metodách. Na uvedených obrázcích 6.10 – 6.13 je vidět, že A* hledá cesty pokud možno přes oblíbenější čtvrtě, ale stále se snaží hledat cestu tak, aby co nejrychleji dosáhl koncového uzlu.



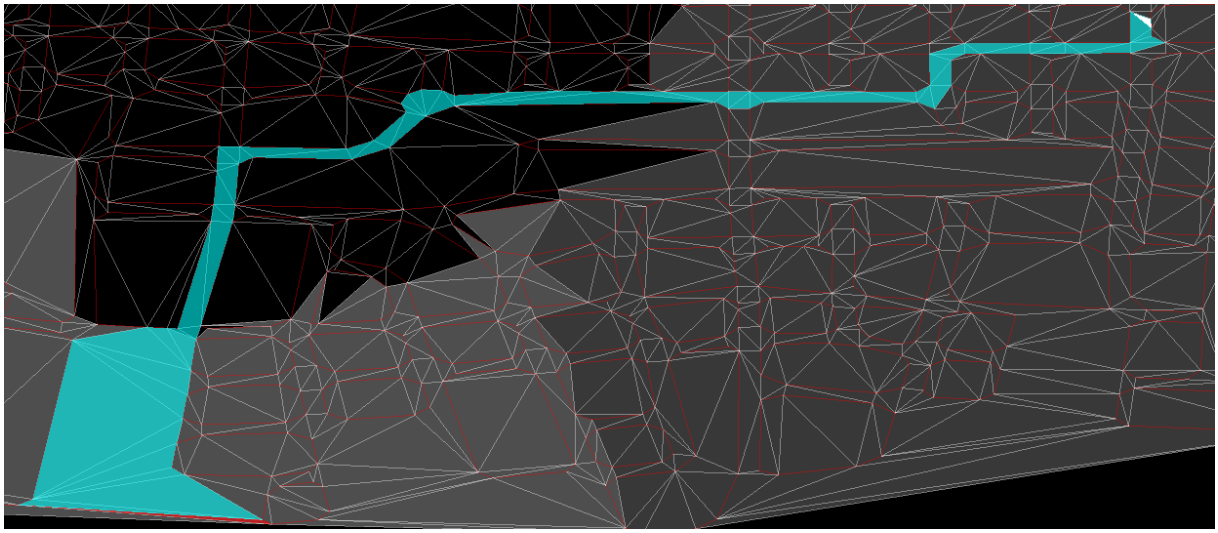
Obrázek 6.10: Nalezená cesta chodce první a páté sociální skupiny A* algoritmem



Obrázek 6.11: Nalezená cesta chodce druhé sociální skupiny A* algoritmem

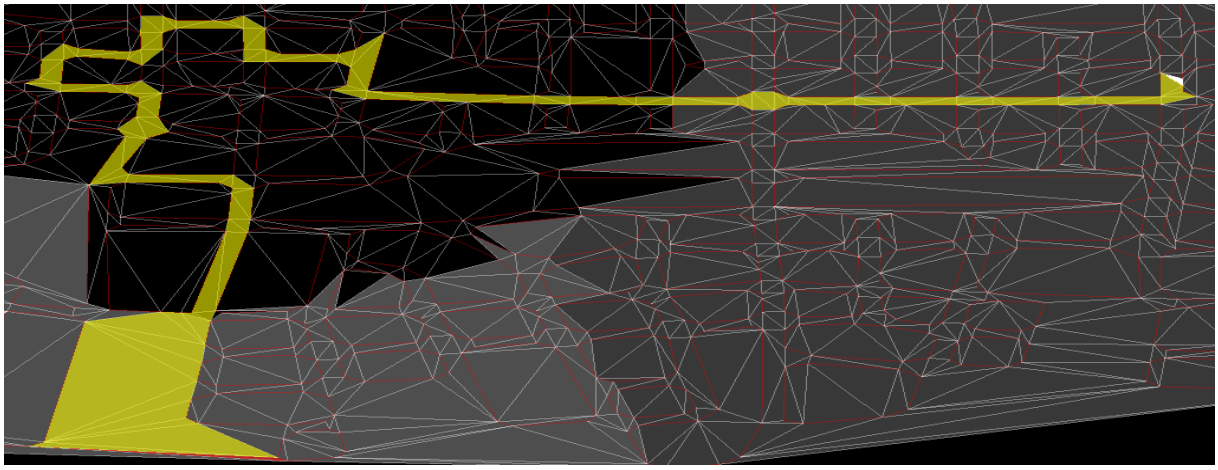


Obrázek 6.12: Nalezená cesta chodce třetí sociální skupiny A* algoritmem

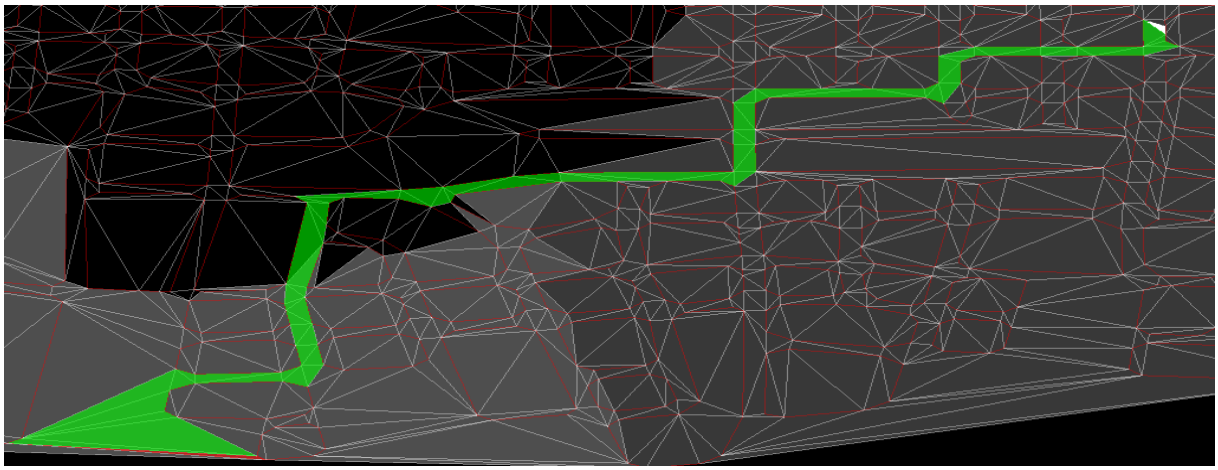


Obrázek 6.13: Nalezená cesta chodce čtvrté sociální skupiny A* algoritmem

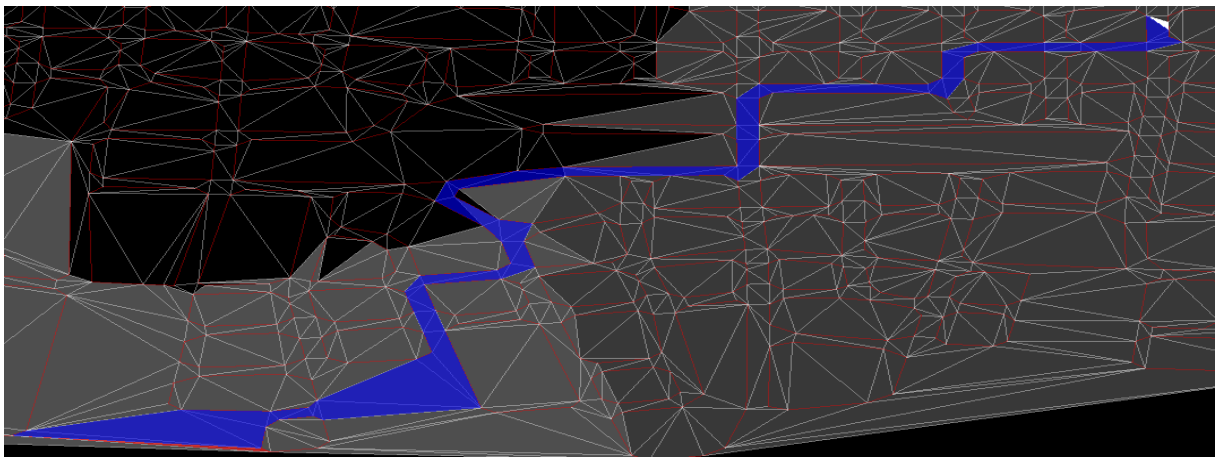
Výsledné cesty Floyd-Warshallova algoritmu jsou na obrázcích 6.14-6.17. FW algoritmus nachází stejné cesty jako A* algoritmus (Obr 6.16 a 6.17), ale protože nepřidává k vyhledávání heuristický prvek (4.1), tak mohou pro náš interval (5.1) vznikat i cesty odlišné od těch, které nalezne A* algoritmus (Obr. 6.14 a 6.15). Může se dokonce stát, že výsledná cesta se na první pohled neblíží přímo ke koncovému uzlu, ale chodec se „prochází“ po své nejoblíbenější čtvrti (Obr. 6.14).



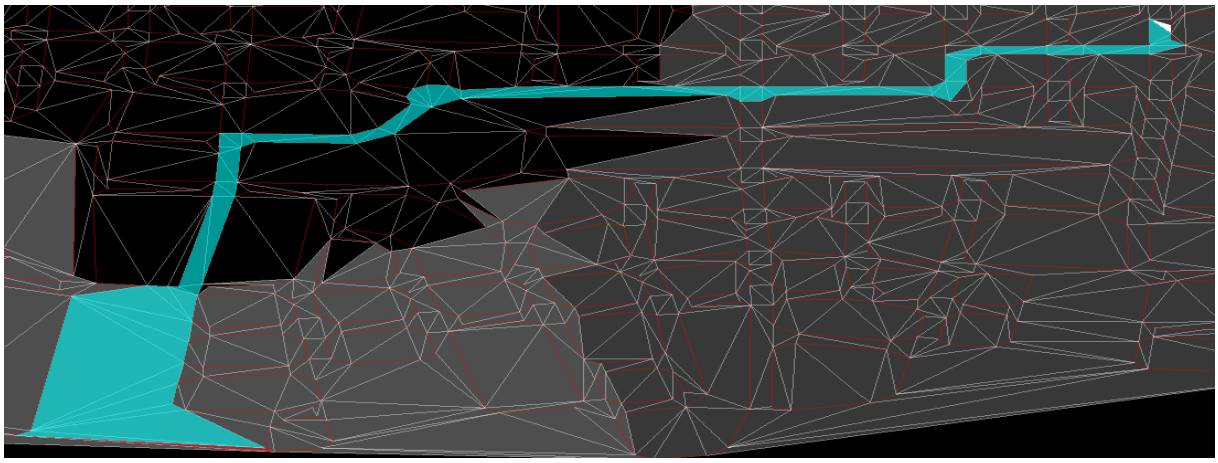
Obrázek 6.14: Nalezená cesta chodce první a páté sociální skupiny FW algoritmem



Obrázek 6.15: Nalezená cesta chodce druhé sociální skupiny FW algoritmem

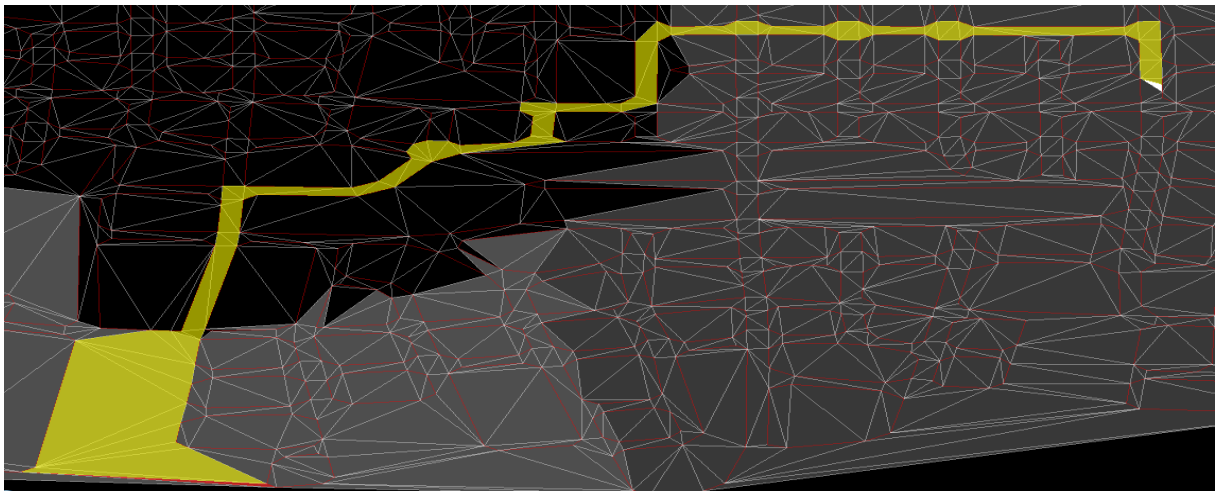


Obrázek 6.16: Nalezená cesta chodce třetí sociální skupiny FW algoritmem

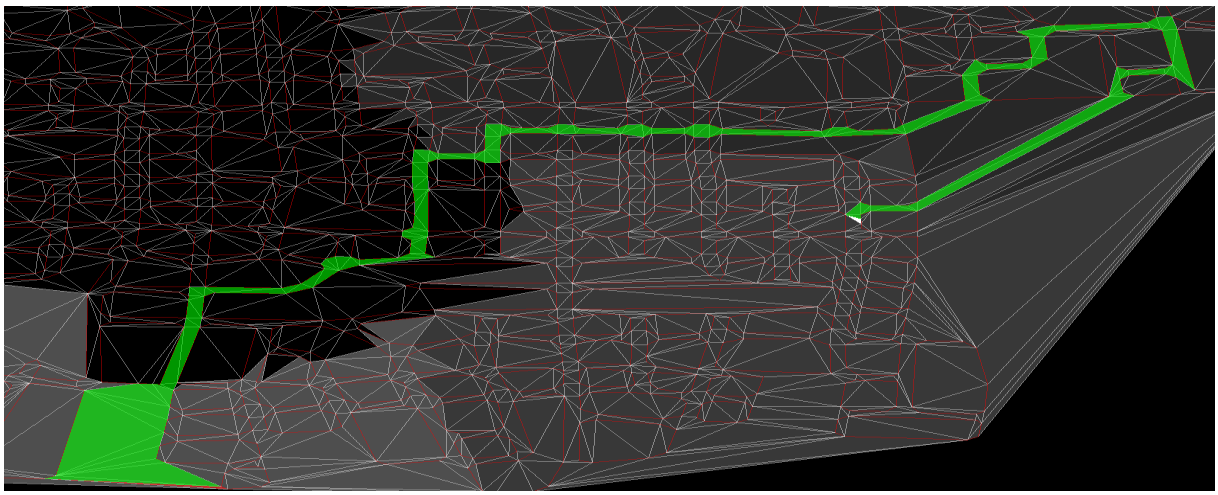


Obrázek 6.17: Nalezená cesta chodce čtvrté sociální skupiny FW algoritmem

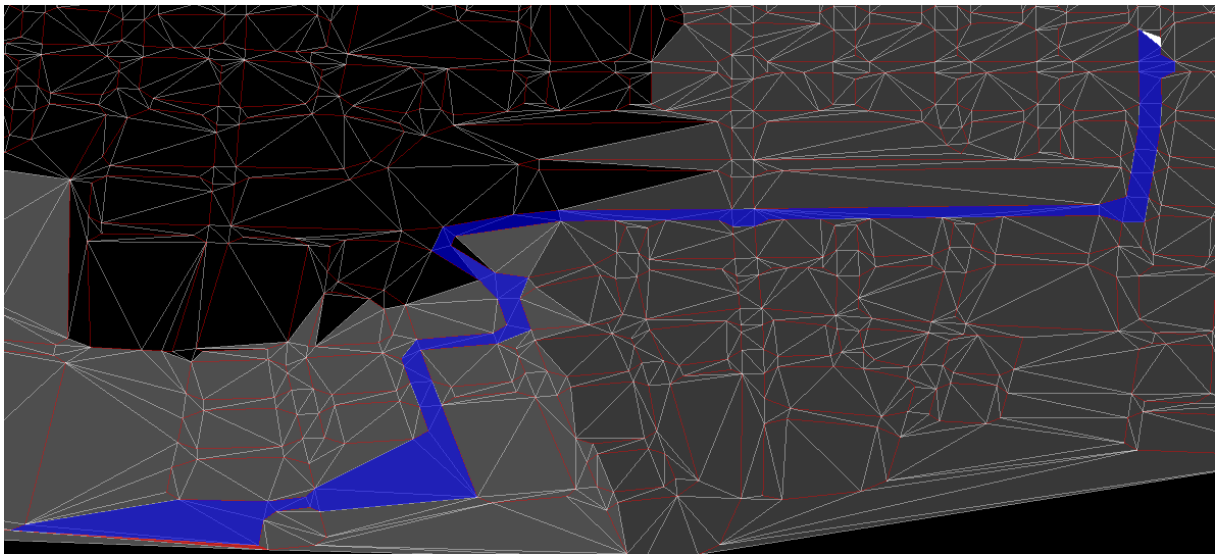
Podobně jako FW algoritmus funguje i přístup CPG (Obr. 6.19-6.22) a NG (Obr. 6.18-6.21). Oba tyto přístupy až na výjimky (Obr. 6.18 a Obr. 6.22) nacházejí totožné cesty. Opět je vidět, že i přístup CPG a NG se snaží hledat cestu přes nejoblíbenější čtvrť, pokud nemusíme jít přes velmi neoblíbenou, a výsledná cesta opět vypadá jako by se chodec „procházel“ (Obr. 6.19), což je následek užitého intervalu (5.1).



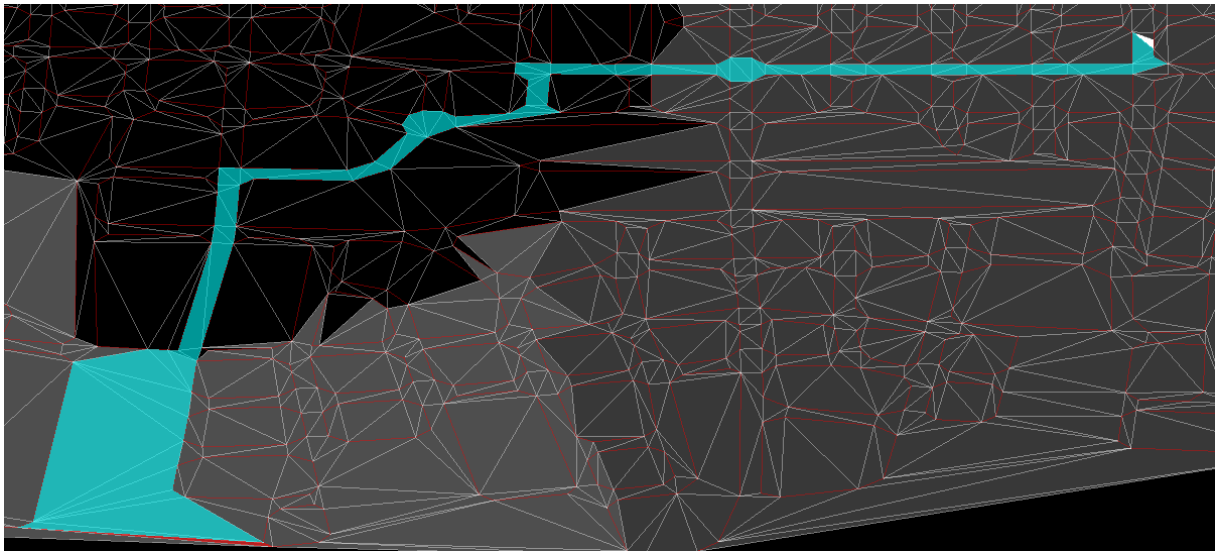
Obrázek 6.18: Nalezená cesta chodce první a páté sociální skupiny přístupem NG



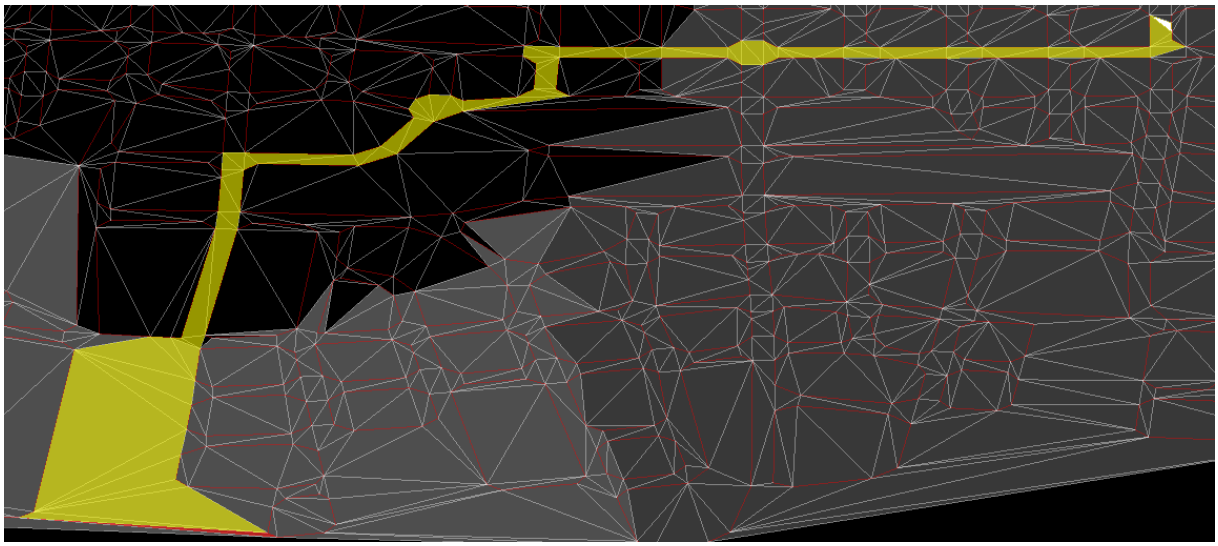
Obrázek 6.19: Nalezená cesta chodce druhé sociální skupiny přístupem NG a CPG



Obrázek 6.20: Nalezená cesta chodce třetí sociální skupiny přístupem NG a CPG



Obrázek 6.21: Nalezená cesta chodce čtvrté sociální skupiny přístupem NG a CPG



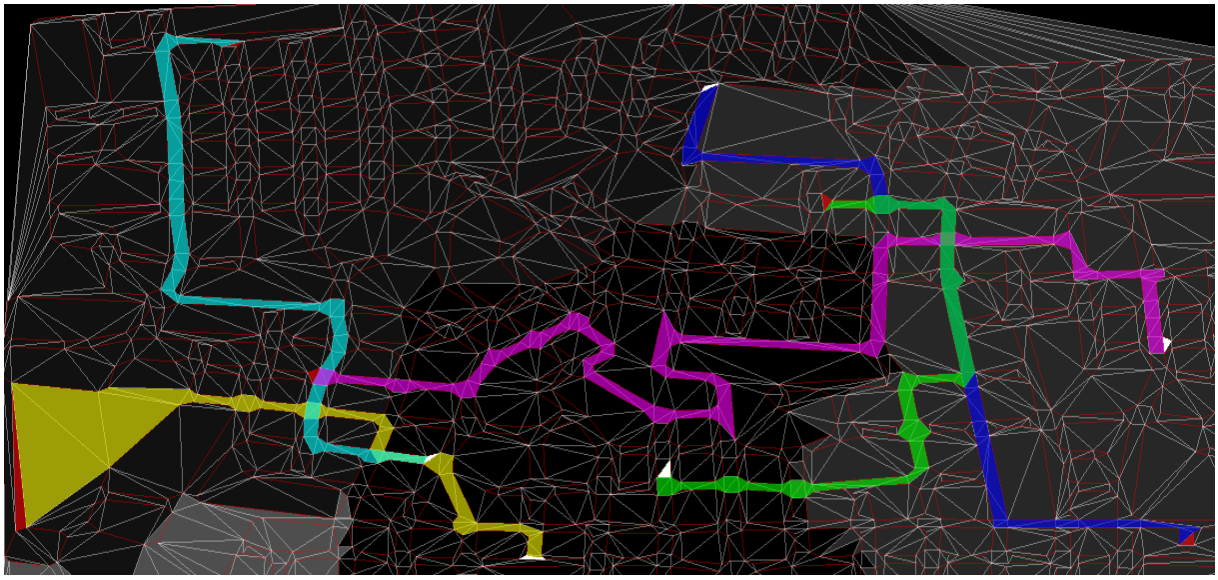
Obrázek 6.22: Nalezená cesta chodce první a páté sociální skupiny CPG přístupem

6.3 Funkce oblíbenosti

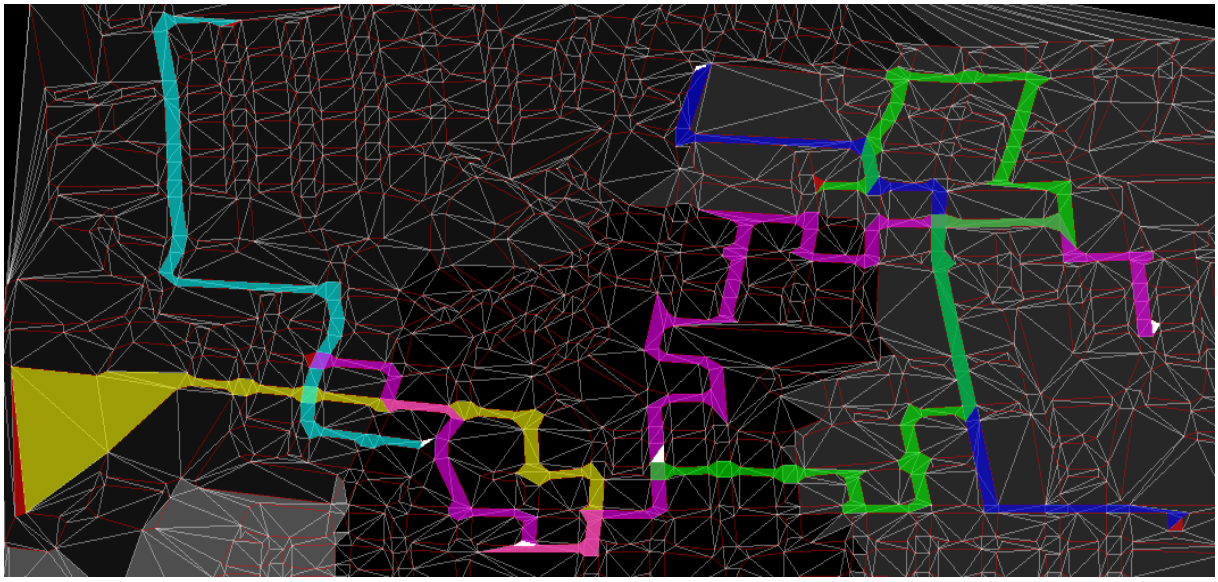
Podobně jako v kapitole 6.2 budeme potřebovat matici oblíbeností, z toho důvodu pro následující experimenty použijeme opět matici M (3).

Funkce oblíbenosti je velmi podobná váhové funkci s tím rozdílem, že vzdálenost zde nehraje už žádnou roli. Algoritmy hledají minimální cestu ve smyslu nejoblíbenější cesty chodce, při které nezáleží na metrice.

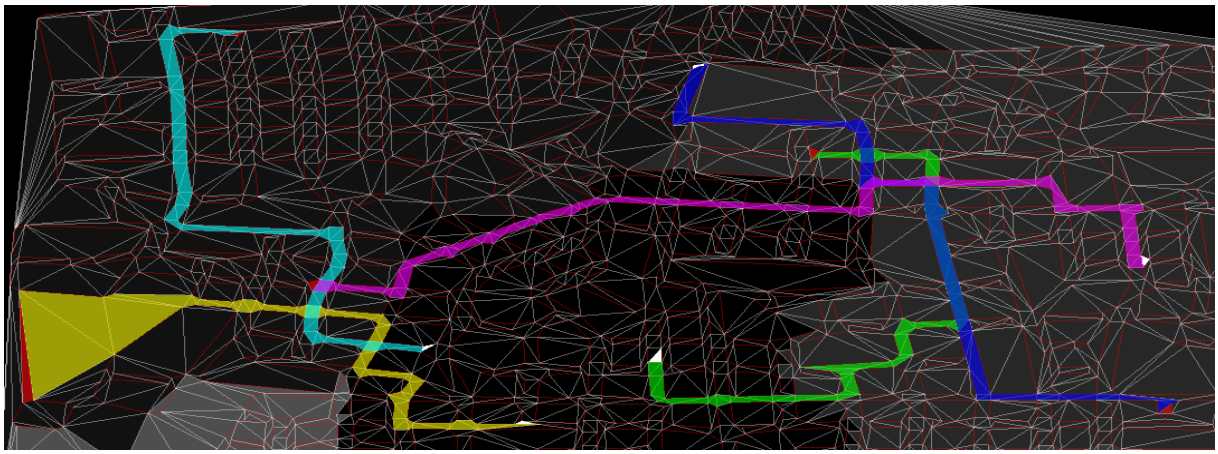
Podobně jako u váhové funkce i zde jsou si přístupy CPG a NG (Obr. 6.25 a 6.26) v nalezených cestách velmi podobné. Ačkoliv opět vypadají opticky velmi dobře, nevznikají u nich často situace, kdy výsledná cesta vypadá, jako kdyby chodec nepospíchal a pouze se procházel po virtuálním městě. Takové cesty velmi často dosahuje FW algoritmus (Obr. 6.24) a v omezeném množství i A* (Obr. 6.23).



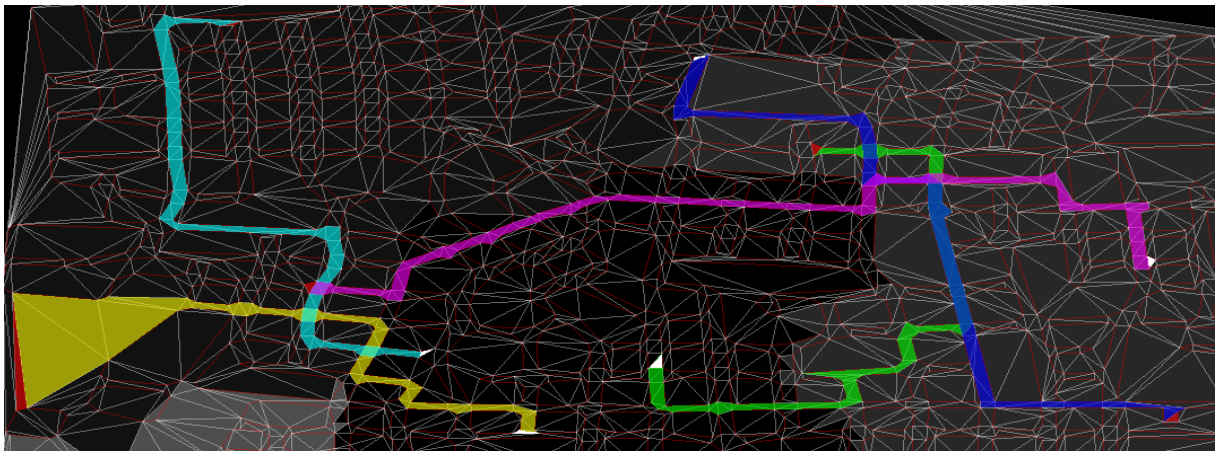
Obrázek 6.23: Nalezené trasy pro 5 různých chodců z různých sociálních skupin přes A*



Obrázek 6.24: Nalezené trasy pro 5 různých chodců z různých sociálních skupin přes FW



Obrázek 6.25: Nalezené trasy pro 5 různých chodců z různých sociálních skupin přístupem NG



Obrázek 6.26: Nalezené trasy pro 5 různých chodců z různých sociálních skupin přístupem CPG

6.4 Časová a paměťová složitost

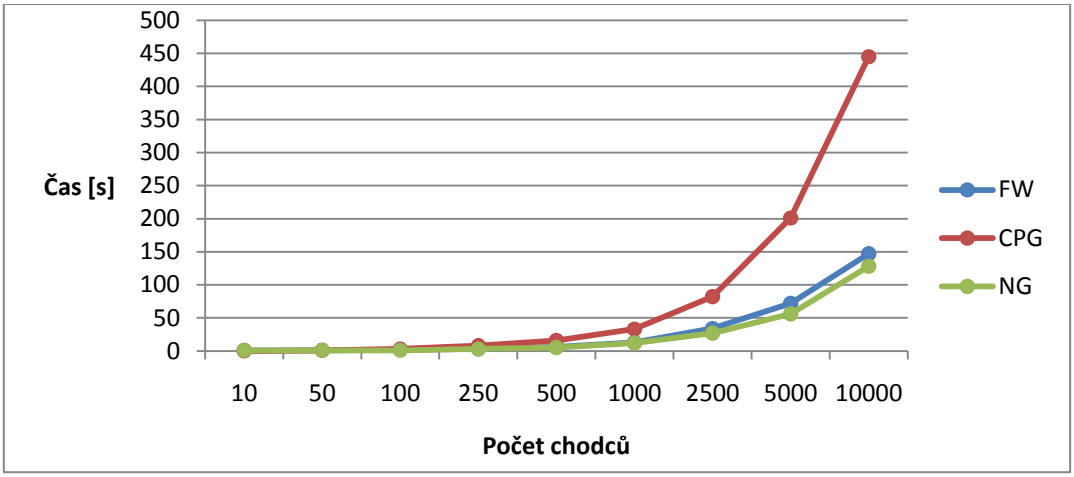
Důležitým aspektem je také časová a paměťová náročnost výpočtu programu. Pro naše virtuální testovací město o 3548 trojúhelnících reprezentujících ulice, jsme změřili časovou složitost řešených metod. Čas výpočtu každé z implementovaných metod byl změřen desetkrát a výsledné časy byly zprůměrovány.

Prvním aspektem měření bylo předzpracování metod. A* algoritmus nepotřebuje žádné předzpracování, proto je v tomto ohledu nejméně časově náročný (Tabulka 6.1). Následují ho pak navigační graf, buněčný a portálový graf a na závěr Floyd-Warshallův algoritmus, kdy časová náročnost předzpracování odpovídá vyjmenovanému pořadí.

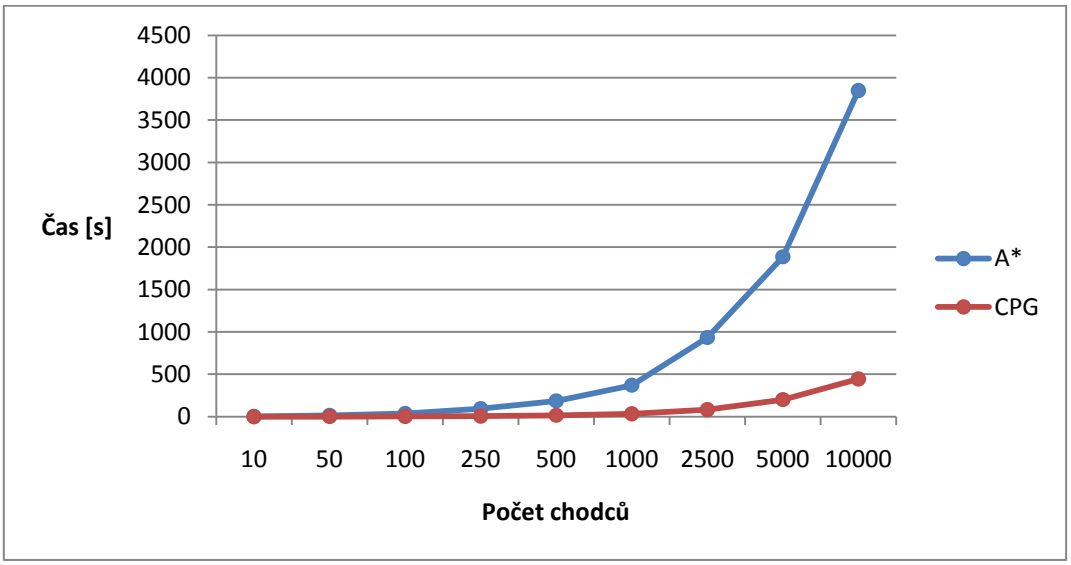
Metoda	Čas [s]
A*	0
FW	254
CPG	6
NG	5

Tabulka 6.1: Časy předzpracování jednotlivých řešení

Dále změříme čas výpočtu trasy chodců. Měření bylo provedeno na 250 různých trasách, které se následně opakovaly. V tomto případě podává nejlepší časové výsledky princip NG. Podobně jako NG je na tom Floyd-Warshallův algoritmus, který dále následuje princip CPG a až následně A* algoritmus. Je zřejmé, že všechny měřené metody mají polynomiální průběh (Obr. 6.27 a 6.28). Uvedené grafy (Obr. 6.27 a 6.28) obsahují čas výpočtu tras chodců, kde není zahrnutý čas potřebný pro předzpracování.



Obrázek 6.27: Časová náročnost výpočtu trasy chodců



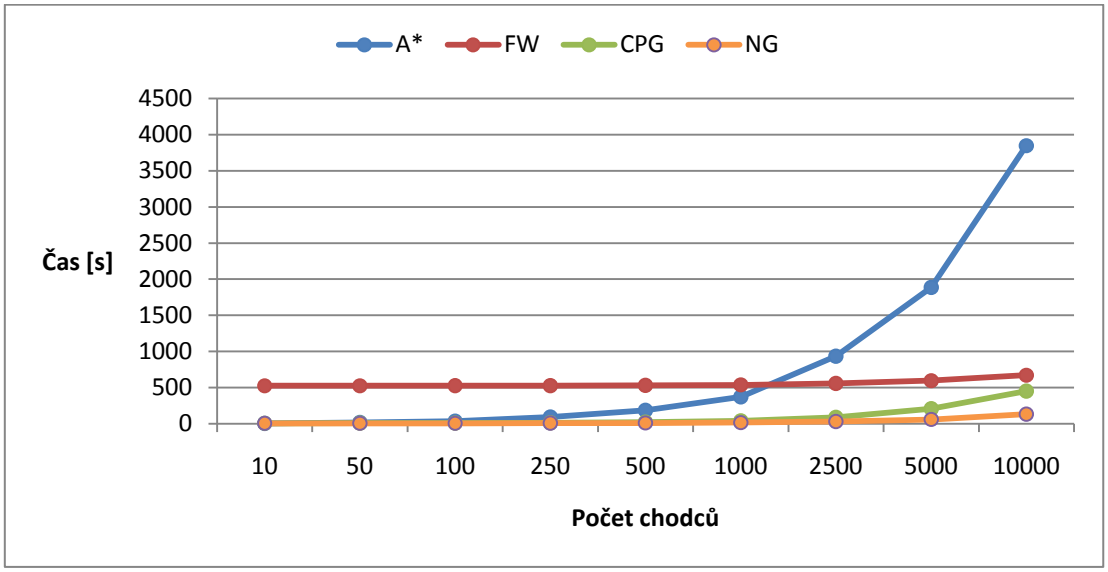
Obrázek 6.28: Časová náročnost výpočtu trasy chodců

Nejhůře vychází výpočet A* algoritmu. To je zapříčiněno tím, že A* prohledává více uzlů a hlavně že musí pro každé dva trojúhelníky reprezentující vrchol počítat vzdálenost mezi jejich středy. Počítání této vzdálenosti je při velkém množství trojúhelníků časově náročné, což se projevuje i na časovém průběhu hledání trasy chodce.

Velmi podobně jsou na tom FW algoritmus a přístup NG. FW algoritmus používá velmi jednoduchou rekurzivní metodu pro vyhledání cesty, ale pro velký počet uzlů, kdežto NG používá složitější rekurzivní metodu, ale pro mnohem menší počet uzlů a ve výsledku si je doba výpočtu trasy obou metod velmi podobná.

Ačkoliv je přístup CPG velmi podobný NG, tak hledání výsledné cesty je přibližně čtyřikrát delší než u NG, protože CPG používá složitou rekurzivní metodu, která výpočet zpomaluje.

Když nyní skloubíme čas na předzpracování a čas samotného výpočtu do jednoho grafu (Obr. 6.29), tak nejrychlejší metodou je navigační graf. Do jistého momentu je pak rychlejší CPG než FW, ale při 15000-20000 chodců bude rychlejší FW algoritmus (pro naše testovací město). I přesto že FW je znevýhodněn dobou předzpracování, tak ve výsledku dopadl lépe než A* algoritmus. Ten dopadl nejhůře, protože výpočet tras pro 10000 chodců trval více než hodinu (Obrázek 6.29).



Obrázek 6.29: Časová náročnost testovaných metod

Na závěr zbývá už jen paměťová náročnost metod. A* algoritmus pro svůj výpočet nepotřebuje žádnou paměť navíc, kde by byla uložená data pro správný chod algoritmu. Ovšem paměťová náročnost zbylých metod je minimálně $2N^2$(tedy $O(N^2)$), a to v případě Floyd-Warshallova algoritmu, protože potřebujeme dvě matice o rozměrech N. V případě přístupu CPG a NG je tato náročnost $O(N^2 + N)$, kde podobně jako u FW potřebujeme paměť na distanční matici a matici předchůdců, ale navíc potřebujeme mít uložené i silnice, k čemuž slouží pole silnic velikosti N.

Paměťová náročnost při hledání nejkratší cesty jsme si již řekli, ale v případě váhové funkce nebo funkce oblíbenosti je paměťová složitost větší. Tato změna se netýká pouze A* algoritmu. V případě, že máme pět různých sociálních skupin, tzn. 5 čtvrtí, tak pro každou sociální skupinu existuje jiná váhová funkce či funkce oblíbenosti. Z toho důvodu bude paměťová náročnost FW algoritmu v obecném případě $O(mN^2)$ a paměťová náročnost CPG a NG pak $O(mN^2 + mN)$, kde m je počet sociálních skupin ve městě.

S tím také vzroste čas předzpracování FW algoritmu a CPG a NG přístupu. V našem případě čas předzpracování (Tabulka 6.1) každé metody vzroste m-krát. Při velkém počtu čtvrtí m se např. FW algoritmus stane nejhorší metodou jak z hlediska paměti, tak z hlediska časové náročnosti.

7 Závěr

Navigace jedinců ve virtuálním městě v závislosti na oblíbenosti jednotlivých oblastí města je zajímavý problém a dozajista by se pro jeho řešení našlo uplatnění. Ani ne tak proto, že navigace chodců s podle oblíbenosti městských čtvrtí nebyla veřejně publikována, ale pro svůj skrytý potenciál. Umím si představit, že tento model by mohli využívat např. v hypermarketech, aby zvýšili svůj zisk. Rozdíl by byl pouze v tom, že by se jednalo o model obchodu, nikoliv města, a městské oblasti by byly nahrazeny úseky se zeleninou, elektrem, pečivem, atd. Své uplatnění by také mohl nalézt např. v oblasti cestovního ruchu, kde by se simuloval pohyb turistů. Následně by podle simulace mohlo být zkoumáno, proč turisté nenavštěvují např. katedrálu a jakým způsobem zvýšit její oblíbenost, atd.

Jaká je tedy nejlepší metoda z testovaných řešení? To je velice diskutabilní otázka v případě váhové funkce nebo funkce oblíbenosti, protože ani jedna z metod nemá převažující klady nad ostatními metodami. Vždy se najde problém, kdy jinak horší metoda spočítá lepší výsledek než zbylé metody. Velmi záleží na tom, kolik chodců chceme ve městě mít, kolik paměti jsme ochotni umožnit pro daný problém, atd.

V případě hledání nejkratší cesty by do sto chodců bylo vhodné použít A* algoritmus, pro větší počet chodců princip navigačního grafu. To i přesto, že pravděpodobně nenajde nejkratší cestu při prvním hledání, proto bychom se pokusili najít cestu čtyřikrát (kapitola 6.1) a vybrali tu nejkratší cestu. Tím se sice čtyřnásobně prodlouží výpočet, čímž se přiblížíme k době výpočtu CPG, ale na rozdíl od CPG navigační graf vždy nalezne nejkratší cestu.

8 Literatura

[BM*11] B. Beneš, M. A. Massih, P. Jarvis, D. G. Aliaga, C. A. Vanagas, **Urban Ecosystem Design**
Purdue University, 2011

[Cha03] S. K. Chang, **Data Structures and Algorithms**
World Scientific Publishing Co Pte Ltd, 2003

[ČKR04] *R. Čada, T. Kaiser a Z. Ryjáček,* **Diskrétní matematika**
Skripta, Západočeská univerzita v Plzni, 2004

[Fuk06] *M. Fuksa*, **Delaunayova triangulace s omezením (CDT) v $\mathbf{E^2}$ a $\mathbf{E^3}$**
Diplomová práce, Západočeská univerzita v Plzni, 2006

[Hug03] R. L. Hughes, **The flow of human crowds**
Department of Civil and Environmental Engineering, The University of Melbourne, 2003

[JNC11] D. Joyner, M. Van Nguyen, N. Cohen, **Algorithmic Graph Theory**
Version 0.6, 6 January 2011

[LaV06] *S. M. LaValle*, **Planning algorithms**
Cambridge university press, 2006

[NR68] N. J. Nilsson, B. Raphael, **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**
Transactions on Systems Science and Cybernetics SSC4, p. 100–107, 1968

[PA*08] N. Pelechano, J. Allbeck, N. Badler, B. Barsky, **Virtual Crowds: Methods, Simulation, and Control**
Morgan & Claypool Publishers, 14 November 2008

[TCP06] A.Treuille, S. Cooper, Z. Popović, **Continuum Crowds**
Proceeding SIGGRAPH '06 ACM SIGGRAPH Papers, p. 1160-1168, 2006

[YM*08] B. Yersin, J. Maïm, F. Morini, D. Thalmann, **Real-time crowd motion planning: Scalable Avoidance and Group Behavior**
Springer-Verlag, pages 859-870, 7 August 2008

[Záb05] J. Zábranský, **Triangulace povrchu a úlohy na nich**
Diplomová práce, Západočeská univerzita v Plzni, 2005

8.1 Elektronické zdroje

[CGA12] CGAL, **Computational Geometry Algorithms Library**,
http://www.cgal.org/

[Khr12] **OpenGL - The Industry's Foundation for High Performance Graphics**
©2012 Khronos Group, http://www.khronos.org/opengl, May 2012

[Nvi12] **CUDA**, © 2012 NVIDIA Corporation,
http://www.nvidia.com/object/cuda_home_new.html, May 2012

[Pat12] A. Patel, amitp@cs.stanford.edu, **Introduction to A***
http://theory.stanford.edu/~amitp/GameProgramming/index.html

[Wik12] Wikipedia, **Shortest path problem**
http://en.wikipedia.org/wiki/Shortest_path_problem, Last modified on 21 January 2012 at 21:23.