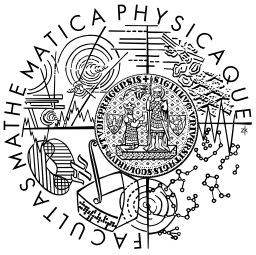


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Šimon Navrátil

Hledání v polygonální mapě

Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: RNDr. Ondřej Pangrác, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Název práce: Hledání v polygonální mapě

Autor: Šimon Navrátil

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: RNDr. Ondřej Pangrác, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Vyhledávání nejkratší cesty je dobře probádaná oblast pro dikrétní problémy. Ne všechny problémy lze ale přímo popsat grafem a v orientačním běhu může běžec zvolit cestu kudykoliv. I tak ale musí z mapy vybrat tu nejrychlejší. To komplikuje i fakt, že v jednotlivých oblastech mezi kontrolami se pohybuje různě rychle. Pro nalezení optimální cesty je tedy potřeba najít nějaké spojité řešení. V této práci je popsáno, jak dostat z mapového souboru polygonální reprezentaci mapy a jak v ní pak následně vyhledávat nejrychlejší cestu, pomocí dvou různých přístupů.

Klíčová slova: Nejkratší cesta Konvexní optimalizace Orientační běh

Title: Polygon map navigation

Author: Šimon Navrátil

Institute: Computer Science Institute of Charles University

Supervisor: RNDr. Ondřej Pangrác, Ph.D., Computer Science Institute of Charles University

Abstract: Finding the shortest path is a well-researched area for discrete problems. However, not all problems can be directly described by a graph, and in orienteering the runner can choose the path whichever way he wants, but he has to choose the fastest one just from the map. This is made more complicated by the different speed in different areas between the control points. In order to find the optimal path, a continuous solution has to be found. This work describes how to get a polygonal representation of a map from a map file and how to search the fastest path in it using two different approaches.

Keywords: Shortest path Convex optimization Orienteering

Obsah

Úvod **3**

1 Představení problému **4**
1.1 Mapa . . . 4
1.2 Model reality . . . 4
1.3 Řešení problému . . . 4
1.3.1 Hledání posloupnosti polygonů . . . 5
1.3.2 Hledání optimálních průsečíků . . . 5
1.3.3 Postup řešení . . . 5

2 Optimalizace **6**
2.0.1 Formulace úlohy . . . 7
2.0.2 Účelová funkce . . . 7
2.1 Řešení úlohy . . . 9

3 Geometrie **10**
3.1 Parsování objektů . . . 10
3.1.1 Beziérovy křivky . . . 10
3.1.2 Liniové objekty . . . 10
3.2 Oříznutí polygonů . . . 10
3.3 Sjednocení polygonů . . . 11
3.4 Konvexní dekompozice . . . 12
3.5 Vytvoření grafu . . . 12

4 Algoritmus **14**
4.1 Heuristický algoritmus . . . 14
4.1.1 A* . . . 14
4.2 Prohledávácí algoritmus . . . 15
4.2.1 Prořezávání . . . 15

5 Vývojová dokumentace **16**
5.1 Úvod . . . 16
5.2 app . . . 16
5.3 maplib . . . 16
5.4 mapanalyzer . . . 17
5.4.1 Booleovské operace na polygonech . . . 17
5.4.2 Dekompozice na konvexní polygony . . . 17
5.4.3 Optimalizační solvery . . . 18
5.4.4 Algoritmus . . . 19
5.4.5 Konfigurační soubor . . . 19
5.5 Spuštění . . . 19
5.5.1 Přímé spuštění . . . 19
5.5.2 Vytvoření spustiltelného souboru . . . 19
5.5.3 Spuštění jako modul pythonu . . . 20

6 Uživatelská dokumentace **21**

Závěr **23**

Seznam použité literatury **24**

Úvod

Vyhledávání v mapách je dlouho známý a dobře probádaný problém. Každý den používáme navigaci v autě nebo vyhledávání v jízdních řádech městské hromadné dopravy. Tyto programy většinou řeší nějaký diskrétní problém, který lze reprezentovat pomocí grafu a v něm efektivně vyhledávat nejkratší a potažmo nejrychlejší cestu cestu. (Kapitola 1 [6]) Podobný problém je třeba řešit v orientačním běhu. Běžec má mapu, ve které jsou vyznačené různými barvami oblasti, ve kterých se pohybuje různou rychlostí, a musí najít nejrychlejší trasu z jedné kotroly na druhou.

Pokud by se běžec pohyboval všude konstantní rychlostí, lze opět vytvořit graf ze všech krajních bodů všech oblastí, mezi kterými bude hrana pouze pokud na sebe v mapě uvidí - tzv. visibility graph (Sekce 4.4 [13]).

Běžec se ovšem všude pohybuje jinak rychle a jednotlivé oblasti jsou tedy ohodnocené, což by se dalo asi nejefektivněji řešit diskrétně, podrozdělením na určité malé části, ale cílem této práce je prozkoumat, jak by se k tomuto problému dalo přistoupit jako ke spojitému, a najít tak tu úplně nejrychlejší (optimální) cestu.

V první kapitole je blíže představen tento problém, ve druhé kapitole je pak rozebrána optimalizační část řešení. Ve třetí jsou popsány veškeré geometrické postupy a problémy, které vyvstaly při převádění mapového souboru na použitelnou reprezentaci v programu. Ve čtvrté a páté kapitole je vývojová a uživatelská dokumentace.

1. Představení problému

1.1 Mapa

V orientačním běhu dostane běžec papírovou mapu podle mapového klíče ISOM2017-2 [7]. V elektronické podobě se tyto mapy vytváří nejčastěji v programu OCAD nebo programu OpenOrienteering Mapper. Druhý jmenovaný je opensource a formát ve kterém mapy ukládá (.xmap nebo úspornější a častější .omap) je veřejně přístupný textový formát (narozdíl od proprietárního binárního formátu .ocd). V mapovém klíči jsou specifikované bodové, liniové a plošné značky. U liniových a plošných značek můžeme podle klíče předpokládat rychlost, kterou se skrz danou oblast běžec může pohybovat. Ta se udává relativně - v procentech plné rychlosti, protože každý se samozřejmě pohybuje jinak rychle. Například na cestě se může běžec pohybovat stoprocentní rychlostí, ale v hustém lese bude rychlost pouze 60%.

1.2 Model reality

Z takto zadané mapy můžeme vytvořit disjunktní pokrytí celé plochy jednotlivými oblastmi, v kterých podle mapové značky můžeme odhadnout rychlost. V tomto modelu reality vyhledáváme cestu ze startovního ($s = (s_x,s_y)$) do cílového bodu($e = (e_x,e_y)$). Některé oblasti při tom mohou být nepřekonatelné (tj. pohybujeme se přes ně nulovou rychlostí), což jsou v realitě například budovy nebo skály. V tomto modelu chceme vyhledat nejrychlejší a tedy optimální možnou cestu.

Definice 1 (Polygon)**.** *Polygonem zde rozumíme uzavřenou podmnožinu roviny ($\mathbb{R}^2$), zadané pouze seřazeným seznamem vrcholů.*

Definice 2 (Disjunktní pokrytí)**.** *Disjunktní pokrytí je množina polygonů taková, že jejich vnitřky jsou po dvou disjunktní (mohou se protínat pouze ve vrcholech a hranách) a jejich sjednocení pokrývá celou naší mapu (uzavřenou podmnožinu roviny).*

1.3 Řešení problému

Jelikož je cílem najít optimální řešení, nelze mapu podrozdělit na malé podoblasti (například pixely v určitém rozlišení) a tím problém diskretizovat a řešit jako hledání nejkratší cesty v ohodnoceném grafu, protože tím bychom našli pouze aproximaci, ale musíme k hledání přistoupit nějak spojitě. Pokud si všechny oblasti dokážeme převést na konvexní polygony, optimálním řešením bude lomená čára, jejíž body jsou na hranicích polygonů s různou rychlostí a jejíž jednotlivé segmenty tvoří úsečky (rovné čáry), protože rychlost v jednom polygonu je konstantní a díky konvexitě čas překonání polygonu závisí jen na vzdálenosti bodů, kde jsme do polygonu vstoupili, a kde jsme ho opustili.

Tímto jsme rozdělili problém vlastně na dva. První je najít nějakou posloupnost polygonů ve kterých se budeme pohybovat a druhý je najít body na společných hranách (tj. body kde bude lomená čára protínat jednotlivé hranice polygonů - dále „průsečíky“) těchto polygonů, skrz které když půjdeme, tak nám cesta bude trvat nejkratší dobu.

1.3.1 Hledání posloupnosti polygonů

Pokud máme celou plochu pokrytou disjunktními polygony, můžeme si vytvořit graf sousedních polygonů. Každý polygon bude tvořit vrchol grafu a do množiny hran (E) budou patřit všechny dvojice polygonů, které budou mít alespoň jeden společný bod, tedy společný vrchol nebo společnou hranu. Vrcholům s rychlostí 0 nepřiřadíme žádné sousední vrcholy (budou v grafu tvořit vlastní komponentu).

Definice 3 (Přípustná posloupnost polygonů)**.** *Přípustná posloupnost polygonů pak bude posloupnost* P_0,P_1,P_2,...,P_n*, kde* $s \in P_0$ *a* $e \in P_n$ *a zároveň* $(P_{i-1}, P_i) \in E$ $\forall i \in [1..n]$ *(v prvním polygonu je start, v posledním cíl a vedlejší polygony spolu sousedí v grafu).*

1.3.2 Hledání optimálních průsečíků

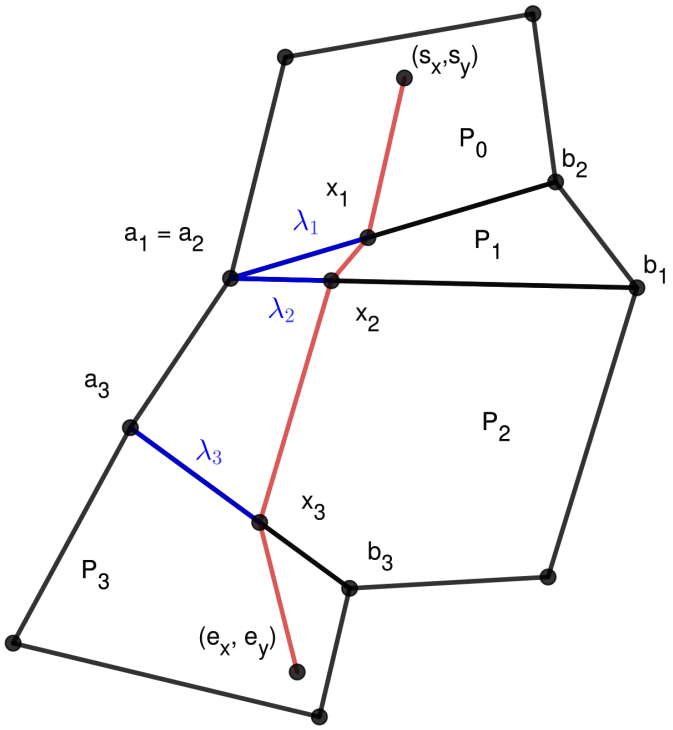
Jestliže společné hrany přípustné posloupnosti polygonů P_0,P_1,P_2,...,P_n jsou c_1,c_2,...,c_n(kde $c_i \in P_i$ a zároveň $c_i \in P_{i-1}$, tedy c_i je hranou mezi P_i a P_{i-1}), pak hledáme průsečíky x_1,x_2,...,x_n (pro které platí, že $\forall i \in [1..n]$: $x_i \in c_i$ - body na společných hranách sousedních polygonů posloupnosti), takové že cesta skrz s,x_1,x_2,...,x_n,e bude nejrychlejší cesta skrz danou posloupnost polygonů. Na to potřebujeme spočítat dobu kterou nám zabere se dostat skrz daný polygon, která je pro konvexní polygon závislá pouze na vzdálenosti $||x_{i-1},x_i||$, narozdíl od nekonvexního polygonu, pro který je výpočet mnohem komplikovanější, protože úsečka x_{i-1},x_i nemusí celá ležet v polygonu P_{i-1}.

1.3.3 Postup řešení

V mapě se obecně mohou nacházet různé druhy polygonů. Jejich hranici může tvořit jakákoliv křivka (ne nutně jen úsečky), mohou mít jakýkoliv tvar, mohou mít díry, hranici, která protíná sama sebe, nebo mohou být jakkoliv nekonvexní. Proto abychom byli schopni vůbec rozumně definovat vnitřek polygonu a počítat dobu po kterou bude trvat jeho překonání, chceme pracovat pouze s konvexními polygony, které jsou zadané pomocí posloupnousti vrcholů a jejichž hranici tvoří pouze úsečky mezi těmito vrcholy. Nevyhovující polygony ze vstupu je potřeba upravit.

Rychlosti pro jednotlivé polygony jsou určené jejich mapovou značkou a převod mezi mapovými značkami a rychlostmi je daný konfiguračním souborem.

2. Optimalizace



Obrázek 2.1: Příklad optimalizační úlohy

Pro každou přípustnou posloupnost polygonů můžeme najít optimální cestu skrz tyto polygony jako řešení následující optimalizační úlohy.

Předpoklady:

- Hledáme nejkratší cestu aniž bychom opustili dané polygony (optimalizujeme tedy cestu, která je celá uvnitř dané posloupnosti polygonů).
- Polygony jsou konvexní a zadané pomocí posloupnosti vrcholů, mezi kterými vede úsečka (ne jakákoliv jiná obecná křivka).
- Posloupnost polygonů na vstupu je přípustná a díky tomu, že jsou všechny polygony konvexní, tvoří průnik dvou sousedních polygonů posloupnosti právě jedna úsečka (může být i degenerovaná, tj. jeden bod).

2.0.1 Formulace úlohy

Polygony	P_0,P_1,$P_2 \dots P_n$
Společné hrany	$c_1 = (a_1,b_1)$,$c_2 = (a_2,b_2), \dots, c_n = (a_n,b_n)$
Startovní bod	s
Cílový bod	e
Rychlosti	v_0,v_1,$v_2, \dots, v_n$
Proměnné	$\bar{\lambda} = \lambda_1, \lambda_2, \dots, \lambda_n$; λ_i určuje průsečík x_i
Průsečíky	$x_i = a_i + \lambda_i * (b_i - a_i)$; $x_i \in c_i$
$\min \bar{\lambda}$	$\frac{\|\|(s,x_1)\|\|}{v_0} + \frac{\|\|(x_1,x_2)\|\|}{v_1} + \ldots + \frac{\|\|(x_n,e)\|\|}{v_n}$
za podmínek	$\lambda_i \in [0,1] \quad \forall i \in 0..n$

2.0.2 Účelová funkce

Účelová funkce je součet časů po které budeme překonávat jednotlivé polygony, takže na základě λ_i známe jednotlivé průsečíky a díky konvexitě jednotlivých polygonů můžeme spočítat vzdálenost, kterou je třeba překonat, a ze znalosti rychlosti v daném polygonu pak spočítáme čas průchodu polygonem jako $čas = \frac{vzdálenost}{rychlost}$.

Cesta:

$$\begin{aligned} L &= ||s;x_1|| \ + \ ||x_1;x_2|| \ + \quad \dots \quad + ||x_n;e|| = \\ &= \sqrt{(s-x_1)^2} \ + \sqrt{(x_1-x_2)^2} \ + \quad \dots \quad + \sqrt{(x_n-e)^2} = \\ &= \sqrt{(s_x - x_{1_x})^2 + (s_y - x_{1_y})^2} + \sqrt{(x_{1_x} - x_{2_x})^2 + (x_{1_y} - x_{2_y})^2} + \\ &\quad \dots \quad + \quad \sqrt{(x_{n_x} - e_x)^2 + (x_{n_y} - e_y)^2} \end{aligned}$$

Čas cesty:

$$\begin{aligned} T = &\frac{1}{v_0}\sqrt{(s_x - x_{1_x})^2 + (s_y - x_{1_y})^2} + \frac{1}{v_1}\sqrt{(x_{1_x} - x_{2_x})^2 + (x_{1_y} - x_{2_y})^2} + \\ &\dots \quad + \quad \frac{1}{v_n}\sqrt{(x_{n_x} - e_x)^2 + (x_{n_y} - e_y)^2} \end{aligned}$$

Definice 4 (Konvexní množina)**.** *Množina S, je konvexní právě tehdy když $\forall(x,y) \in S$ a $\theta \in [0,1]$ platí $\theta x + (1-\theta)y) \in S$ (Pro všechny x,y z definičního oboru úsečka $(f(x),f(y))$ leží nad grafem.)*

Definice 5 (Konvexní funkce)**.** *Funkce $f : \mathbb{R}^n \to \mathbb{R}$, je konvexní právě tehdy když definiční obor f je konvexní množina a zároveň $\forall(x,y) \in \boldsymbol{dom} f$ a $\theta \in [0,1]$ platí $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$ (Pro všechny x, y z definičního oboru úsečka $(f(x),f(y))$ leží nad grafem.)*

Definice 6 (Afinní funkce)**.** *Funkce $f : \mathbb{R}^n \to \mathbb{R}$, je afinní právě tehdy když $\forall x \in \textbf{\textit{dom}}\ f\ \exists$ lineární funkce g a $b \in \mathbb{R}$, že $f(x) = g(x) + b$*

Lemma 1. *Funkce $a : \mathbb{R}^n \to \mathbb{R}$ je afinní a funkce $k : \mathbb{R} \to \mathbb{R}$ je konvexní, pak funkce $f(x) = k(a(x))$ je konvexní.*

Důkaz. $\forall(x,y) \in \mathbf{dom} f$ a $\theta \in [0,1]$ platí

$$\begin{aligned} f(\theta x + (1-\theta)y) &= k(a(\theta x + (1-\theta)y)) \\ &= k(\theta a(x) + (1-\theta)a(y)) \\ &\leq \theta k(a(x)) + (1-\theta)k(a(y)) = \theta f(x) + (1-\theta)f(y) \end{aligned}$$

První rovnost je z definice f a druhá díky afinnitě a. Následující nerovnost je pak díky konvexitě k a poslední rovnost opět z definice f.

□

Lemma 2. *Euklidovská vzdálenost ($||\cdot||$) je konvexní funkce.*

Důkaz. $\forall(x,y) \in \mathbb{R}$ a $\theta \in [0,1]$ platí

$$\begin{aligned} ||\theta x + (1-\theta)y|| &\leq ||\theta x|| + ||(1-\theta)y|| \\ &= |\theta| \cdot ||x|| + |(1-\theta)| \cdot ||y|| = \theta||x|| + (1-\theta)||y|| \end{aligned}$$

První nerovnost plyne z trojúhelníkové nerovnosti, druhá z toho, že norma je homogenní a třetí z toho, že $\theta \in [0,1]$.

□

Lemma 3. *Součet konvexních funkcí je konvexní funkce.*

Důkaz. Funce f a g jsou konvexní a tedy platí:

$$\begin{aligned} f(\theta x + (1-\theta)y) &\leq \theta f(x) + (1-\theta)f(y) \\ g(\theta x + (1-\theta)y) &\leq \theta g(x) + (1-\theta)g(y) \end{aligned}$$

A tedy platí také:

$$\begin{aligned} &f(\theta x + (1-\theta)y) + g(\theta x + (1-\theta)y) \\ &\leq \theta f(x) + (1-\theta)f(y) + \theta g(x) + (1-\theta)g(y) \\ &= \theta(f(x) + g(x)) + (1-\theta)(f(y) + g(y)) \end{aligned}$$

□

Věta 4. *Účelová funkce (u) je konvexní.*

Důkaz. Definujme si funkce $g_0, \ldots, g_n : \mathbb{R}^n \to \mathbb{R}$ kde $g_i(\lambda) = a_i + \lambda_i(b_i - a_i)$ a $g_0 = s$ a $g_n = e$, které jsou všechny afinní. Účelová funkce u se pak dá zapsat jako:

$$u(\bar{\lambda}) = \sum_{1}^{n} ||(g_{i-1}(\lambda), g_i(\lambda))||.$$

Díky lemmatům pak plyne že u je konvexní.

□

2.1 Řešení úlohy

Úloha má tedy konvexní účelovou funkci a pouze lineární omezující podmínky. Ve skutečnosti je vlastně omezena na (n-1)rozměrnou jednotkovou krychli (tedy $[0,1]^{n-1}$). I když neexistuje přesný vzorec na výpočet optimálního řešení, existují podle Boyda a Vandenbergheho [1] pro takovou úlohu, podobně jako u lineárního programování, effektivní solvery.

Navíc pro přípustnou posloupnost polygonů existuje vždy nějaké řešení (například skrz středy společných hran sousedních polygonů) a díky konvexitě účelové funkce je každé lokální optimum i optimem globálním (též Boyd a Vandenberghe [1]).

3. Geometrie

V této kapitole bychom chtěli popsat, jak z mapového souboru získáme model, ve kterém potom lze vyhledávat.

3.1 Parsování objektů

Načtení jednotlivých objektů ze souboru je čistě technická záležitost, ale i u toho bylo potřeba vyřešit několik otázek. U každého objektu bylo potřeba získat hranici, barvu a případné hranice děr daného polygonu.

3.1.1 Beziérovy křivky

V souboru je hranice uložena jako posloupnost bodů, ovšem některé body je potřeba na základě příznaku interpretovat jako Beziérovu křivku. Beziérova křivka (v tomto případě konkrétně kubická, tj. 3. řádu) může být jednodušše aproximována lomenou čarou pomocí De Casteljauova algoritmu (viz. kapitola 3.4 Farin & Hansford[4]). Kubická křivka je definivána jako

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, 0 \leq t \leq 1$$

(viz. kapitola 3.3 Farin & Hansford [4]).

V tomto algoritmu nejde vlastně o nic jiného než o vyhodnocení křivky v určitých hodnotách parametru t. Čím více bodů vyhodnotíme, tím přesnější aproximaci dostaneme, zároveň ale roste i komplexita polygonu (počet jeho vrcholů), proto je nejjednodušší křivku nahrazovat pouze nějakým malým pevně daným počtem bodů.

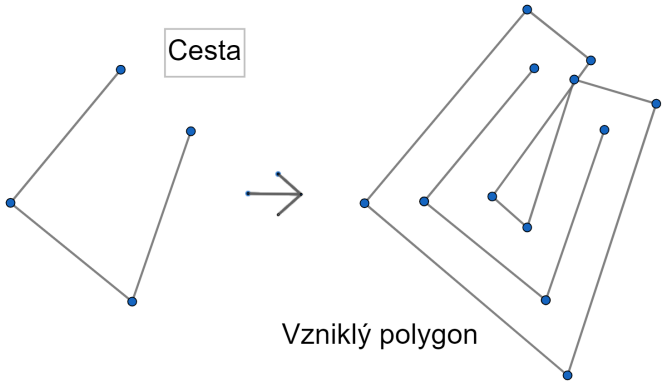
3.1.2 Liniové objekty

Další komplikací jsou liniové objekty. Hlavní smysl mají, když jsou nepřekonatelné (například sráz, zeď, plot atd.). Dále jsou liniovými objekty reprezentovány i cesty, na kterých je naopak pohyb rychlejší. Jak se vypořádat s těmito objekty se přímo nabízí v mapovém klíči, který definuje minimální rozměry nepřekonatelných a překonatelných značek (viz. ISOM2017-2 sekce 2.11 [7]), proto je možné nahradit liniové objekty úzkými polygony o minimálním rozměru aniž by se tak změnil smysl značek či zachycení reality.

Z algoritmického hlediska to není tak triviální úloha, jak by se mohlo na první pohled zdát, protože může docházet k různým topologickým změnám (například vznik děr nebo nových polygonů) „nafukovaných“ polygonů (viz obrázek 3.1), ale přesto existují algoritmy, které se s tímto problémem (v angličtině nazývaným polygon offsetting) dokáží více či méně robustně vypořádat (viz Cacciola (2003) [2]).

3.2 Oříznutí polygonů

Jelikož potřebujeme disjunktní pokrytí polygony, je potřeba vyřešit i drobné mezery mezi polygony a hlavně jejich překryv (obojí je většinou vzniklé nedo-



Obrázek 3.1: Přiklad problémů při „nafukování“ polygonů.

konalostí kresby mapařem). Z mapového souboru je možné přečíst barvy všech objektů a také priority jednotlivých barev. Díky tomu je možné seřadit objekty od nejvyšší priority k nejnižší a v tomto pořadí je procházet a ořezávat všechny ostatní objekty s nižší prioritou.

Oříznutí polygonu podle jiného polygonu je vlastně jedna z booleovských operací na polygonech, na které existují různé algoritmy, ale většina využívá nějakých speciálních případů. Čím obecnější polygon máme, tím komplikovanější tento problém je, ale existuje i algoritmus pro booleovské operace na obecných poygonech, tedy těch které nejsou konvexní, mohou mít díry a jejichž strany se mohou protínat (viz Vatti a Bala R. 1992 [12]).

3.3 Sjednocení polygonů

Tento krok je dobrý obecně ke zjednodušení. Jednak se tím odstraní ty hrany polygonů, které se samy protínají a také se tím mapa alespoň trochu zjednodušší, protože můžeme sjednotit všechny polygony ve kterých se pohybujeme stejnou rychlostí.

Z algoritmického hlediska je to velmi podobné ořezávání polygonů, protože jde pouze o jinou booleovskou operaci, což umožňuje využití stejného algoritmu a potažmo stejné knihovny.

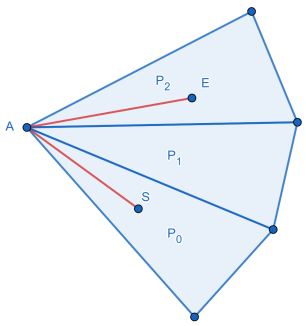
3.4 Konvexní dekompozice

Jak už bylo řečeno ve dřívějších kapitolách, potřebujeme, aby všechny polygony byly konvexní, a toho lze dosáhnout podrozdělením všech nekonvexních polygonů na menší konvexní části. Nejjednodušší cestou je triangulace, ale vznikne tím nejvíce polygonů. Naopak najít optimální řešení co do kardinality (počtu polygonů), je podle Fernandéze et al. [5] sice možné pomocí polynomiálního algoritmu, ale pouze pro speciální případ bez děr.

Pro polygony s děrami se jedná o NP-těžký problém. Existují však postupy (běžící v polynomiálním čase), které se i pro obecné polygony s děrami snaží najít co nejlepší řešení. Fungují tak, že vylepšují počáteční triangulaci, aby se optimálnímu řešení alespoň co nejvíce přiblížily (jeden z nich je například popsán ve výše zmíněném článku [5]).

3.5 Vytvoření grafu

Když už máme disjunktní pokrytí celé mapy polygony, chceme si nad nimi vytvořit graf, ve kterém uvidíme, který polygon se kterým sousedí. Za sousední polygony můžeme brát ty, které mají společnou hranu a je otázka jestli za společnou hranu chceme počítat i společný bod. Jako lepší se ukázalo nepřidávat do grafu hrany mezi polygony sousedícími pouze přes jeden bod, protože tím se graf stává složitějším a jejich vynecháním o žádné řešení nepřijdeme. To lze demostrovat následujícím obrázkem. Řešení jdoucí z P_0 do P_2 přes bod A je totiž ekvivalentní řešení jdoucímu z P_0 do P_1 a pak do P_2, kde ale oba přechody (mezi P_0 a P_1 a mezi P_1 a P_2) jdou také skrz bod A.



Obrázek 3.2: Přiklad ekvivalentních řešení.

Samotné vytváření grafu je provedeno triviálně v $O(n^2)$, kde n je počet polygonů, otestováním všech dvojic polygonů, jestli spolu sousedí. (Reálně se kontrolují pouze ty dvojice polygonů, u nichž se protínaly ohraničující obdélníky.) Zjištění, jestli spolu polygony sousedí je opět kvadratické v počtu jejich vrcholů, protože je potřeba u každé dvojice hran najít jejich úrůsečík. Kvůli tomu může na větších datech už samotné vytváření grafu trvat poměrně dlouho.

Nedokonalosti kresby byly odtraněny podložením celé mapy jedním polygonem získaným jako konvexní obal všech ostatních polygonů, čímž se vyplnily mezery mezi polygony přečtenými z mapy a mohli jsme tak získat disjunktní pokrytí.

4. Algoritmus

K řešení byly využity dva různé přístupy. Jeden algoritmus prohledává všechny možné cesty v grafu, a tedy posloupnosti polygonů (dále „prohledávácí algoritmus“). Druhý se snaží nějakou heuristikou chytře najít výhodnou posloupnost polygonů, a v ní pak najít optimální řešení (dále „heuristický algoritmus“). Nemáme ale záruku, že nelezené řešení je nejlepší mezi všemi.

4.1 Heuristický algoritmus

Pro nalezení co nejlepší posloupnosti polygonů z grafu, lze využít standardní algoritmy na hledání nejkratší cesty, například Dajkstrův algoritmus (viz. kap. 6 Mareš a Valla [8]) nebo jeho obdoba, která umí vzít v potaz nějakou heuristiku, A* (viz. kap. 4 Russel a Norvig [10]). Otázkou ale je, jak ohodnotit jednotlivé hrany v grafu, a jak dobré (špatné) řešení můžeme zaručit.

4.1.1 A*

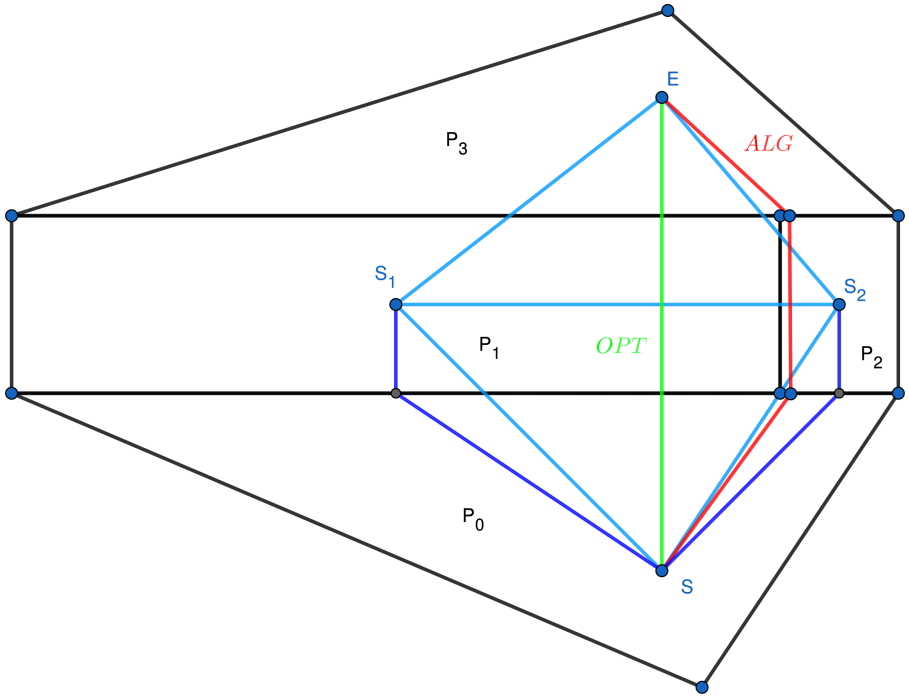
V algoritmu A*, byly ohodnoceny hrany grafu následující funkcí:

P_1,P_2 ... sousední polygony
v_1,v_2 ... jejich rychlosti
S_1,S_2 ... jejich středy (průměr všech vrcholů)
(a,b) ... jejich společná hrana
v_{max} ... maximální rychlost v mapě (většinou 100)
$w(P_1,P_2)$ ... ohodnocení hrany mezi P_1 a P_2
$h(P_i)$ ... honota heuristické funkce pro polygon P_i

$$w(P_1,P_2) = 1/v_1||(S_1, \frac{a+b}{2})|| + 1/v_2||(\frac{a+b}{2}, S_2)||$$
$$h(P_i) = 1/v_{max}||(S_i, e)||$$

Ohodnocení hrany mezi sousedními polygony je doba cesty mezi jejich středy přes střed jejich společné hrany. Heuristika pro A* je nejkratší možný čas, za který se dá teoreticky dostat do cílového bodu (vzdušnou čarou a maximální rychlostí).

To bohužel nezaručuje nic o nalezeném řešení, protože pro libovolné ϵ, o které chceme aby se lišilo řešení algoritmu (ALG) a optimální řešení (OPT), můžeme sestrojit mapu následujícím způsobem (viz obrázek 4.1). Stačí 4 polygony, ve kterých bude všude rychlost 1 a OPT tedy vede přímo mezi startovním a cílovým bodem, skrz polygon P_1. Polygon P_1 ovšem může být libovolně široký, tak aby cesta do jeho středu (modrá) byla delší než cesta do středu polygonu P_2. Algoritmus pak vždy vybere cestu skrz polygon P_2.



Obrázek 4.1: Přiklad špatného řešení

4.2 Prohledávácí algoritmus

Mějme graf s n vrcholy, počet cest z s do e je v obecném grafu exponenciální. Jelikož jsme v našem případě vynechali hrany, které spojují polygony sousedící pouze ve společném vrcholu, vyhledáváme vlastně v duální grafu ke grafu, jehož stěny tvoří polygony z mapy.

Dalším malým zlepšením je prořezávání na základě již získaných nejlepších cest (ať už doposud nalezených nebo získaných na začátku výše popsaným heuristickým algoritmem 4.1). Problémem tohoto řešení samozřejmě je jeho exponenciální časová složitost, kvůli které běží neúnosně dlouho, a i když se můžeme snažit nejprve prohledávat „nejnadějnější“ cesty, tak dokud neprohledáme všechny, nemůžeme mít záruku, že jsme našli tu nejlepší.

4.2.1 Prořezávání

Při znalosti nějakého řešení o hodnotě w^* můžeme z grafu vynechat všechny polygony podle následujícího kritéria:

$p_s \ldots$ nejbližší bod v polygonu k startovnímu bodu
$p_e \ldots$ nejbližší bod v polygonu k cílovému bodu

$$w^* \leq 1/v_{max}||(s, p_s)|| + 1/v_{max}||(p_e, e)||$$

Jde o proříznutí všech polygonů mimo elipsu s ohnisky v bodech s, e a součtem vzdáleností od nich $w^* * v_{max}$.

5. Vývojová dokumentace

5.1 Úvod

Program byl napsán v jazyce Python, který byl zvolen pro svou jednoduchost a množství open source knihoven. Již předem totiž bylo zřejmé, že bude potřeba využít mnoho již existujících algoritmů, a zároveň implementovat grafické rozhraní, diky kterému se výsledky výpočtů dají nějak rozumně interpretovat. Díky Pythonu by měl být také přenositelný mezi platformami, ale testován byl pouze na Windows 10.

K řešení závislostí mezi jednotlivými balíčky byl využit program Python Poetry. V dalších částech budou podrobněji představeny jednotlivé části programu, což jsou:

- **app** - Část kódu, která zajišťuje komunikaci s uživatelem.
- **maplib** - Část kódu, která parsuje soubor s mapou.
- **mapanalyzer** - Část kódu, kde probíhají veškeré výpočty.

5.2 app

Na uživatelské rozhraní je v pythonu standardní knihovnou tkinter, která se jevila jako dostačující. Navíc je multiplatformní. Funguje ale pouze v hlavním vlákně, takže bylo nutné proložit výpočty a interakci s uživatelem. K tomu slouží funkce *main*, která je implementována jako iterátor a postupně vykonává všechny potřebné operace a je volána z hlavní smyčky tkinteru vždy, když není potřeba interakce s uživatelem. Pokud tedy chceme vidět, co postupně program dělá, je celá struktura programu popsána právě v této funkci.

Modul *presenters* nabízí tři různá uživatelská rozhraní: *Viewer*, *TestViewer* a *DebugViewer*. Všechna využívají třídu *ZoomCanvas* což je wrapper okolo knihovní třídy *tkinter.Canvas*, kterou obohacuje o posouvání, přibližování a oddalování. Liší se tak, že první je určený pro produkci, druhý pro testování (kód neběží v hlavní smyčce tkinteru a díky tomu se dají debuggovat vyhozené vyjímky) a třetí je jen na jednorázové použití, například při debuggování.

5.3 maplib

Tento balíček slouží ke čtení a parsování mapového souboru. Je implementováno parsování souborů typu .omap a .xmap, které jsou na disku zapsány jako textové soubory ve formátu xml a na jejich parsování tedy posloužil balíček *xml.etree.ElementTree*, který je součástí standardní knihovny. Rozhraní tvoří funkce *load_map*, která vrací objekt *Map*. Pro implementaci dalších formátů stačí napsat další třídu, která dědí od *Map* (už funguje *OOMapperMap*) a zařadit ji do funkce *load_map*.

5.4 mapanalyzer

V této části je implementována veškerá logika, vysvětlená v předcházejících kapitolách. Hlavním modulem je *analyzer*, který poskytuje všechny důležité funkce pro procesování jednotlivých polygonů a také nevíce používanou třídu třídu *Area*, která reprezentuje polygon a vytváří se z objektu a symbolu určité oblasti (které získáme z knihovny maplib). Při parsování se nahrazují Beziérovy křivky polygony. A všechny funkce v tomto modulu pak berou na vstupu právě tyto třídy a zase je vrací (po nějaké úpravě).

5.4.1 Booleovské operace na polygonech

Pro ořezávání a sjednocování polygonů bylo nutné najít vhodnou a dostatečně robustní knihovnu. Vyzkoušeny byly Shapely, Pyclipper a také implementace algoritmu podle Martinéze et al. [9]. Žádný se ale nepodařilo odladit natolik, aby bez problému zprocesoval všechny polygony z mapy (v ručně kreslené mapě se zřejmě vyskytuje hodně nepřesností a nepravidelností), a vždy se našly nějaké hraniční případy, které nešlo zpracovat.

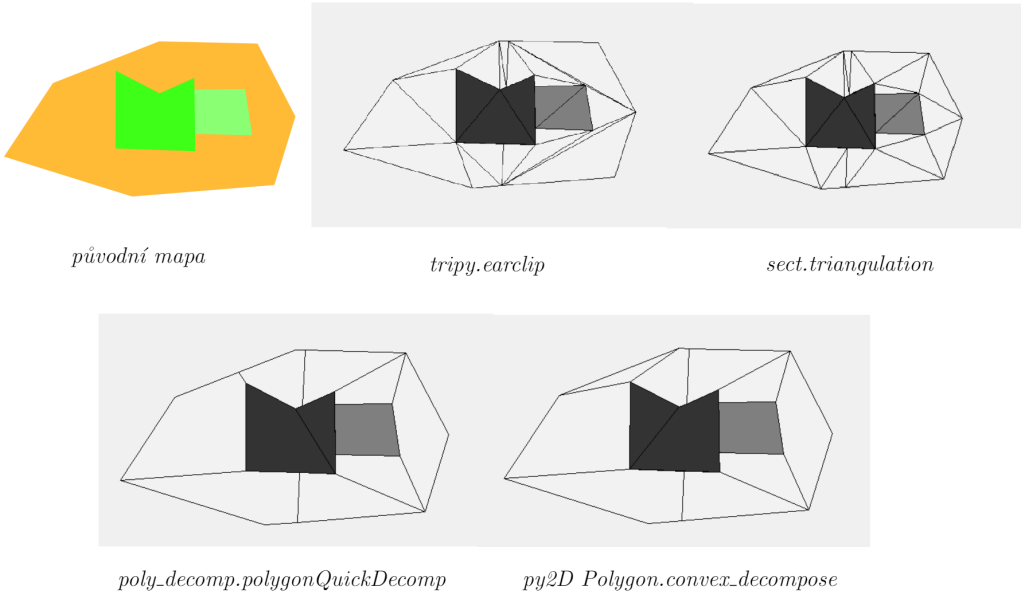
Nakonec se nejlépe osvědčila knihovna Pyclipper se zapnutou možností *StrictlySimple*, která se jeví funkčně a robustně. Jde o implementaci Vattiho algoritmu v c++ [12] a jeho wrapper pro python. Výhodou této knihovny mimo jiné je, že nenabízí pouze všechny 4 booleovské operace na obecných polygonech, ale i jejich offsetování. Všechna tato funkcionalita je volána z modulu *clipper*.

5.4.2 Dekompozice na konvexní polygony

Modul *geometry* nabízí jednak funkci *convex_hull* na vytvoření konvexního obalu předaných polygonů pomocí knihovny SciPy a také funkci *decompose*, která předané polygony podrozdělí na konvexní. Na implementaci této funkce jsme vyzkoušeli hodně různých knihoven a funkcí a hlavními kritérii byla hlavně robustnost a počet vytvořených polygonů, protože čím méně jich bylo tím menší nám z toho pak vnikne graf. Samozřejmě byla také potřeba podpora pro polygony s děrami a proto ji měli všechny vyzkoušené funkce.

- *tripy.earclip* Tato funkce provede triangulaci a hlavní nevýhodou tedy je velký počet vytvořených polygonů.
- *sect.triangulation constrained_delaunay_triangles* Tato funkce také najde pouze triangulaci a navíc není ani přesně určená k tomu, co my potřebujeme.
- *sect.decomposition polygon_trapezoidal* Tato funkce už by se jevila jako užitečnější, ale nevrací polygony, ale graf, z kterého bychom museli polygony zpětně rekonstruovat.
- *poly_decomp.polygonDecomp* Použitý algoritmus by měl podle dokumentace být optimální, ovšem tato funkce z nějakého důvodu skoro na všech vstupech padala.
- *poly_decomp.polygonQuickDecomp* Použitý algoritmus by měl podle dokumentace být suboptimální, ale zato běžet výrazně rychleji. Tato funkce fungovala dobře na menších vstupech, ale končila často vyjímkou.

- *py2D Polygon.convex_decompose* Nakonec byla použita tato knihovna, která implementuje algoritmus od Fernandéze et al. [5], protože se osvědčila jako nejrobustnější.



Obrázek 5.1: Porovnání convexních dekompozicí

5.4.3 Optimalizační solvery

Pro vyřešení úlohy kovexní optimalizace bylo potřeba najít vhodný solver.

SciPy

Hojně používanou knihovnou na matematické a vědecké výpočty v pythonu je SciPy postavená nad knihovnou numpy a obsahuje i užitečné balíčky pro optimalizaci (scipy.optimize), který obsahuje více různých metod pro hledání minima určité funkce. Pro náš problém se z této knihovny podle Varoquauxa [11] jevila nejužitečněji metoda L-BFGS-B (tj. Limited-memory Broyden-Fletcher-Goldfarb-Shanno with bounds) a to z několiku důvodů. Jednak využívá pro hledání minima gradient, který je v našem případě pouze numericky aproximovaný. Naše funkce by sice měla mít gradient a dokonce i Hessián vyjádřitelný analyticky, ale bylo by to tak náročné, jak pro bezchybnou implementaci, tak na výpočetní čas, že by se to s největší pravděpodobností nevyplatilo. A pak také dovoluje omezení problému na vícerozměrnou krychli, což je přesně to co potřebujeme. Mimo tuto metodu byly vyzkoušeny i všechny ostatní, které dovolovaly omezení pouze na nějakou doménu.

CVXPY

Dále existuje knihovna CVXPY, která se jevila jako ideální, protože poskytuje jednotné rozhraní k více různým solverům. K zadaní problému využívá speciální

syntaxe a mimo jiné díky ní využívá techniku Signed disciplined convex programming (DCP), což slouží k ověření, že funkce je skutečně konvexní, pomocí skládání funkcí u nichž již známe určité vlastnosti (znaménko, konkavitu/konvexitu, monotonnost etc.). Program tak vlastně za nás provede podobný důkaz jako jsme dělali v sekci o účelové funkci 2.0.2. (4. kapitola Diamond & Boyd [3])

Knihovna převádí problém do standardní kónické formy, ve které ji může řešit několikero dostupných solverů, mezi kterými jsou i tři open source. Jedná se o CVXOPT, ECOS a SCS, přičemž první dva využívají metody vnitřního bodu.(3. kapitola Diamond & Boyd [3]).

Řešení

Z těchto možností se po zběžném testování nejrychleji jevil ECOS solver volaný knihovnou CVXOP. Celé řešení optimalizačního problému je v modulu *CP-solver.py* a je zde implementována i funkce *solve_all*, která postupně volá všechny solvery (a metody) na ten stejný problém, a vypisuje informace o časové náročnosti jednotlivých volání.

5.4.4 Algoritmus

Kód řešící hledání cesty je v modulu *algorithm.py* a poskytuje 2 funkce: *find_good_path*, která implementuje heuristický algoritmus 4.1 a *find_best_path*, která implementuje prohledávací algoritmus 4.2. Jednotlivé algoritmy jsou implementovány ve třídách a pro prohledávací algoritmus existuje i možnost pro paralelní spuštění ve více vláknech.

5.4.5 Konfigurační soubor

Konfigurační soubor specifikuje mapování mezi mapovoými značkami a rychlostmi v oblastech s touto značkou. Formát je jednoduchý - komentáře jsou za znakem # a jinak se na každém řádku očekávají 2 údaje oddělené čárkou.

5.5 Spuštění

U zdrojových kódů je i soubor *README.md*, kde je více informací, jak lze program distribuovat/spustit, ale existují vzásadě následující možnosti.

5.5.1 Přímé spuštění

Je potřeba nainstalovat python a python poetry, pak stačí spustit *poetry install* ve složce *src* a *poetry run ___main___.py*

5.5.2 Vytvoření spustiltelného souboru

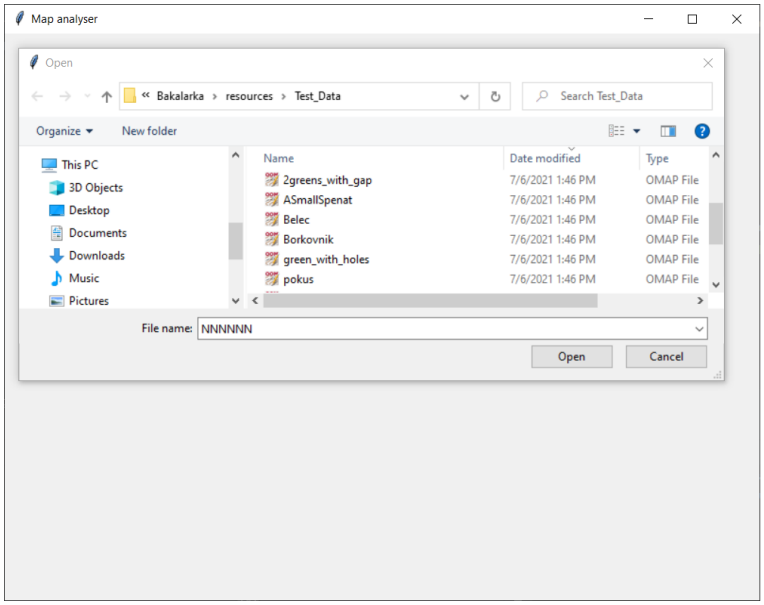
Pomocí programu *pyinstaller*, který je také součástí závislostí specifikovaných v poetry (díky čemuž není potřeba nic instalovat), lze příkazem *pyinstaller map_analyser.spec* vytvořit spustitelný soubor.

5.5.3 Spuštění jako modul pythonu

Pomocí poetry lze také vygenerovat balíček pro python, který je pak spustitelný příkazem *python -m název_vygenerovaného_souboru*, případně lze ještě tento balíček publikovat do veřejného repozitáře (například pypi), odkud jde nainstalovat jednoduše pomocí příkazu *pip install map_analyser*. Pomocí poetry jde celý tento proces zjednodušit do dvou příkazů *poetry build* a *poetry publish*.

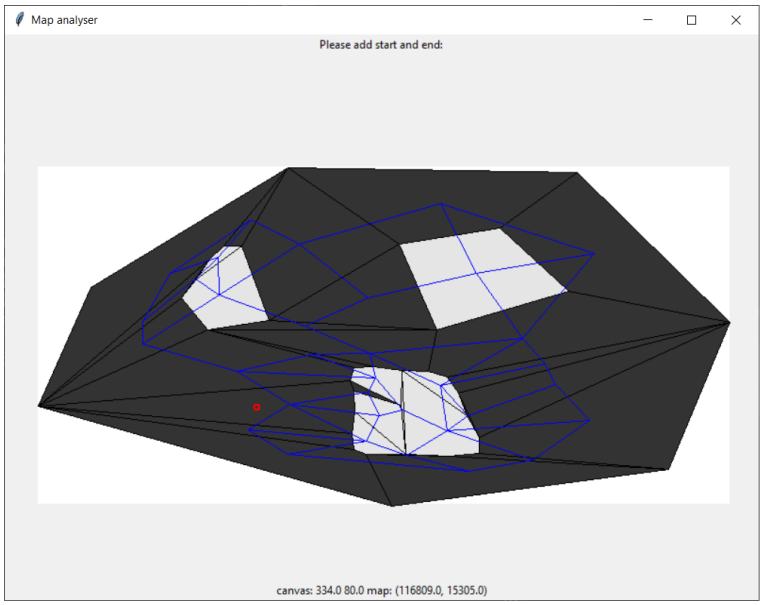
6. Uživatelská dokumentace

Program by měl být přenosný mezi platformami, ale testován byl pouze na Windows 10, x64. Po spuštění užívatel vybere soubor s mapou.



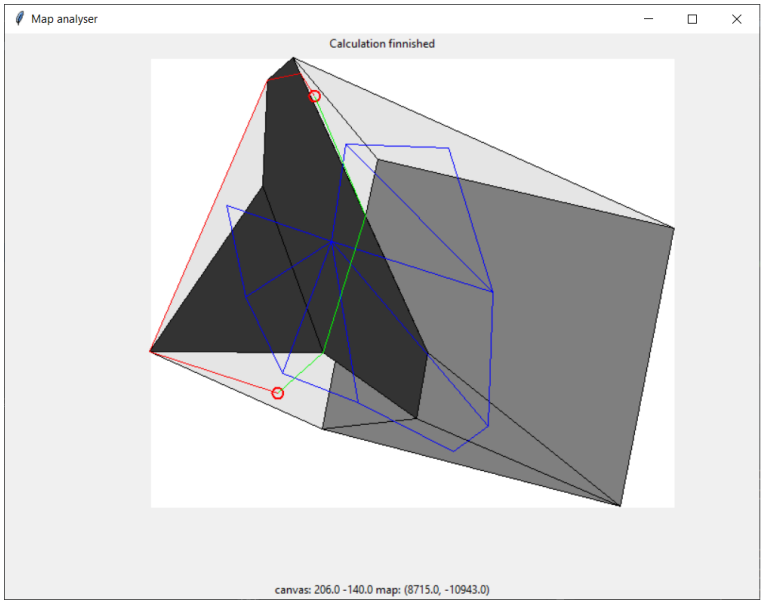
Obrázek 6.1: Výběr mapy

Dále se v horní liště ukazuje co právě v programu probíhá, pak je uživatel vyzván k vybrání startovního a cílového bodu a zobrazí se vytvořené disjunktní pokrytí i s hranamy vytvořeného grafu, kde jednotlivé polygony mají odstín černé, šedé nebo bílé podle rychlosti pohybu v nich.



Obrázek 6.2: Výběr startu a cíle

Po skončení výpočtů se zobrazí zelenou barvou výsledek heuristického algoritmu 4.1 a červenou barvou výsledek prohledávacího algoritmu 4.2.



Obrázek 6.3: Zobrazení výsledků výpočtů

Pokud program zrovna nepočítá, je možno mapou hýbat (uchopení kolečkem nebo pravým tlačítkem myši) nebo ji přibližovat/oddalovat (kolečkem myši).

Uživatel má ještě možnost měnit rychlosti v jednotlivých mapových značkách pomocí konfiguračního souboru (configISOM2017.csv).

Závěr

Převést mapový soubor na polygonální reprezentaci se už samo o sobě ukázalo jako komplikovaný problém. Je potřeba udělat hodně úprav a čištění, a přestože na všechny problémy existují efektivní algoritmy, mohou i tak na reálné mapě běžet poměrně dlouho (i desítky minut). Tímto přístupem také vznikne mnoho polygonů s hodně vrcholy, což komplikuje tvorbu grafu sousedících polygonů. K postavení grafu by se určitě dalo využít efektivnějších algoritmů se zametáním roviny, jako je popsáno u Mareše a Vally v Sekci 16.2 [8]. Ovšem vzhledem k tomu, že prohledávání všech cest běží neúnosně dlouho už na mnohem menších vstupech, postačil triviální přistup.

Obrázek 6.4: Převod mapy (vlevo) na polygonální reprezentaci (vpravo)

Z navržených algoritmů se heuristický algoritmus jeví jako rozumně použitelný, protože podle Mareše a Vally (Sekce 6.2 [8]) s binární haldou seběhne nejhůře v čase $O((n+m)\ log\ n)$. Nalezené řešení často není optimální a může být teoreticky jakkoliv nepřesné, v praxi se ale jeví jako použitelné. Dalo by se také zkusit najít lepší ohodnocení jednotlivých hran grafu (například se zohledněním směru mezi startem a cílem).

U prohledávacího algoritmu, který je exponenciální, se už na trochu větších vstupech (šlo zhruba o malé desítky, ale časová náročnost nezávisí jen na počtu polygonů, ale i na stupních jednotlivých vrcholů), nepodařilo problém zjednodušit natolik, aby doběhl za několik minut, a to ani za pomoci prořezávání. Z tohoto hlediska by se asi opravdu pro praktické použití hodilo mnohem více problém nějakým způsobem diskretizovat. Na malých vstupech ale dává prohledávání často lepší řešení než heuristický algoritmus.

Seznam použité literatury

[1] Boyd, S. a Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, The Edinburgh Building, Cambridge, CB2 8RU, UK. ISBN 978-0-521-83378-3.

[2] Cacciola, F. (2003). A Survey of Polygon Offseting Strategies. URL `http://fcacciola.50webs.com/Offseting%20Methods.htm`.

[3] Diamond, S. a Boyd, S. (2016). CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, **17**(83), 1–5.

[4] Farin, G. a Hansford (2000). *The essentials of CAGD*. CRC Press, Taylor & Francis Group, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742. ISBN 978-1-56881-123-9.

[5] Fernández, J., G.-Tóth, B., Cánovas, L. a Pelegrín, B. (2008). A practical algorithm for decomposing polygonal domains into convex polygons by diagonals. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, **16**, 367–387. doi: 10.1007/s11750-008-0055-2.

[6] Goodwin, S, D., Menon, S. a Price, R, G. (2006). Pathfinding in open terrain. volume 8.

[7] International orienteering federation (2019). ISOM 2017-2 (Adjusted version published January 2019). URL `https://orienteering.sport/wp-admin/admin-ajax.php?action=shareonedrive-download&id=663580750D0C0BCE!44930&dl=1&account_id=663580750d0c0bce&listtoken=a007eb3197c53fc5cdbb183a685617ee`.

[8] Mareš, M. a Valla, T. (2017). *PRŮVODCE LABYRINTEM ALGORITMŮ*. CZ.NIC, z.s.p.o., Milešovská 5, 130 00 Praha 3. ISBN 978-80-88168-19-5.

[9] Martinez, F., Rueda, Antonio, J. a Feito, Francisco, R. (2009). A new algorithm for computing boolean operations on polygons. *Computers & Geosciences*, **35**(6), 1177–1185.

[10] RUSSELL, S. J. a NORVIG, P. (2003). *Artificial Intelligence: A Modern Approach*. ISBN 0-13-790395-2.

[11] Varoquaux, G. (2020). Mathematical optimization: finding minima of functions. URL `https://scipy-lectures.org/advanced/mathematical_optimization/`.

[12] Vatti, Bala, R. (1992). A generic solution to polygon clipping. *Commun. ACM*, **35**(7), 56–63. ISSN 0001-0782. doi: 10.1145/129902.129906. URL `https://doi.org/10.1145/129902.129906`.

[13] Vinther, A. a Strand-Holm (2015). *Pathfinding in Two-dimensional Worlds: A Survey of Modern Pathfinding Algorithms, and a Description of a New Algorithm for Pathfinding in Dynamic Two-dimensional Polygonal Worlds*. PhD thesis, Aarhus Universitet, Datalogisk Institut.