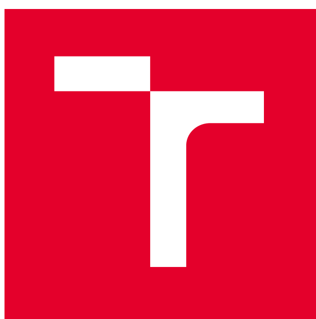


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

MULTIKRITERIÁLNÍ METODY PLÁNOVANÍ CESTY

MULTICRITERIA METHODS OF PATH PLANNING

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — **Bc. Milan Šabata**

AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Šoustek, Ph.D.**

SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav: Ústav automatizace a informatiky
Student: **Bc. Milan Šabata**
Studijní program: Aplikovaná informatika a řízení
Studijní obor: bez specializace
Vedoucí práce: **Ing. Petr Šoustek, Ph.D.**
Akademický rok: 2024/25

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Multikriteriální metody plánovaní cesty

Stručná charakteristika problematiky úkolu:

Úkolem systému pro plánování cesty je nejčastěji najít cestu z počáteční do cílové pozice, která je optimální dle některého zvoleného kritéria, jako je například délka cesty, doba cestování, průchodnost terénem nebo spotřeba energie. Pro použití plánovače cest v reálném prostředí je však vhodné zvažovat kombinaci několika kritérií tak, aby byla zvolena co nejvýhodnější trasa. Existuje řada vícekriteriálních metod, které se úspěšně využívají např. v robotice, videohrách ale i sportu.

Cíle diplomové práce:

Analyzovat přístupy k vícekriteriálnímu plánování cesty.
Analyzovat problematiku plánování cesty v orientačním běhu.
Implementovat vybrané metody plánování cesty v kontextu orientačního běhu.
Provést a vyhodnotit ověřovací a srovnávací experimenty na reálných mapových podkladech.

Seznam doporučené literatury:

McCRORY, Stephen, Bhavyansh MISHRA, Jaehoon AN, Robert GRIFFIN, Jerry PRATT a Hakki Erhan SEVIL. Humanoid Path Planning over Rough Terrain using Traversability Assessment. Online. 2022. Dostupné z: https://doi.org/10.48550/arXiv.2203.00602. [cit. 2023-10-19].

FERNANDEZ, Juan A., J. GONZALEZ, L. MANDOW a J. L. PÉREZ-DE-LA-CRUZ. Mobile robot path planning: a multicriteria approach. Online. Engineering Applications of Artificial Intelligence. 1999, vol. 12, iss. 4, s. 543–554. ISSN 09521976. Dostupné z: https://doi.org/10.1016/S0952-1976(99)00018-4. [cit. 2023-10-19].

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2024/25

V Brně, dne

L. S.

Ing. Pavel Heriban, Ph.D. ředitel ústavu	doc. Ing. Jiří Hlinka, Ph.D. děkan fakulty

ABSTRAKT

Orientační běh klade vysoké nároky na efektivní navigaci, kde nalezení časově výhodnější trasy není triviální úlohou a je silně ovlivněno faktory jako sklon terénu, typ povrchu a vegetace. Standardní navigační nástroje často selhávají v predikci reálné časové náročnosti a identifikaci nejrychlejší varianty postupu. Tato diplomová práce se zabývá návrhem, implementací a vyhodnocením vícekriteriálního systému pro plánování časově optimální trasy v orientačním běhu. Hlavním cílem je vytvořit systém schopný zpracovat relevantní mapová data a na jejich základě, s využitím vhodných algoritmů, identifikovat trasu minimalizující předpokládaný čas průběhu závodníka.

Systém zpracovává vektorová mapová data ve formátu OMAP a rastrová data digitálního modelu terénu. Tato data jsou převedena do grafové reprezentace, kde váhy hran reflektují časovou náročnost průchodu terénem, zahrnující i modelování vlivu sklonu pomocí Toblerovy funkce. Byly implementovány a porovnány různé algoritmy pro hledání nejkratší cesty, včetně Dijkstrova algoritmu, A*, a any-angle algoritmů Theta* a Lazy Theta* pro CPU. Pro akceleraci výpočtů na rozsáhlých mapách byly vyvinuty paralelní varianty pro GPU s využitím CUDA, konkrétně Delta-Stepping a vlastní heuristicky akcelerovaný algoritmus HADS.

Součástí práce je i desktopová aplikace umožňující načtení map, konfiguraci algoritmů a vizualizaci výsledků. Implementované algoritmy byly testovány a porovnány na reálných mapových podkladech orientačního běhu z hlediska výpočetní rychlosti a kvality nalezených tras, včetně konfrontace s postupy zkušených závodníků.

ABSTRACT

Orienteering places high demands on effective navigation, where finding a time-optimal route is a non-trivial task, strongly influenced by factors such as terrain slope, surface type, and vegetation. Standard navigation tools often fail to predict real-time demands and identify the fastest route option. This diploma thesis deals with the design, implementation, and evaluation of a multi-criteria system for planning time-optimal routes in orienteering. The main goal is to create a system capable of processing relevant map data and, based on this data using suitable algorithms, identifying the route that minimizes the competitor's predicted travel time.

The system processes vector map data in OMAP format and raster data from a Digital Terrain Model. This data is converted into a graph representation, where edge weights reflect the time cost of traversing the terrain, including modeling the influence of slope using Tobler's hiking function. Various shortest path algorithms were implemented and compared, including Dijkstra's algorithm, A*, and any-angle algorithms Theta* and Lazy Theta* for CPU. To accelerate computations on large maps, parallel variants for GPU

using CUDA were developed, specifically Delta-Stepping and a custom heuristically accelerated algorithm HADS.

The work also includes a desktop application enabling map loading, algorithm configuration, and results visualization. The implemented algorithms were tested and compared on real orienteering map data in terms of computational speed and the quality of the found routes, including comparison with the routes chosen by experienced competitors.

KLÍČOVÁ SLOVA

orientační běh, plánování trasy, hledání cesty, A* algoritmus, Theta* algoritmus, Lazy Theta* algoritmus, Delta-Stepping algoritmus, HADS algoritmus, GPU, CUDA, paralelní algoritmy, optimalizace trasy, OMAP, digitální model terénu, Toblerova funkce, vícekriteriální optimalizace.

KEYWORDS

orienteering, route planning, pathfinding, A* algorithm, Theta* algorithm, Lazy Theta* algorithm, Delta-Stepping algorithm, HADS algorithm, GPU, CUDA, parallel algorithms, route optimization, OMAP, Digital Terrain Model (DTM), Tobler's hiking function, multi-criteria optimization.

ÚSTAV AUTOMATIZACE
A INFORMATIKY

2025

BIBLIOGRAFICKÁ CITACE

ŠABATA, Milan. *Multikriteriální metody plánovaní cesty*. Brno, 2025. Dostupné také z: `https://www.vut.cz/studenti/zav-prace/detail/162428`. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky, Vedoucí práce: Ing. Petr Šoustek, Ph.D.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato diplomová práce je mým původním dílem, vypracoval jsem ji samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury.

Jako autor uvedené práce dále prohlašuji, že v souvislosti s vytvořením této práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků.

V Brně dne 21. 5. 2025 . . . .

Milan Šabata

PODĚKOVÁNÍ

Rád bych na tomto místě vyjádřil upřímné poděkování všem, kteří přispěli ke vzniku této diplomové práce.

Především děkuji mému vedoucímu práce, Ing. Petru Šoustkovi, Ph.D., za jeho odborné vedení, cenné rady, trpělivost a čas, který mi věnoval během celého procesu tvorby práce, od počátečních konzultací až po finální úpravy.

Dále bych chtěl poděkovat všem kolegům, přátelům a členům akademické obce, kteří si práci přečetli v různých fázích jejího vzniku a poskytli mi konstruktivní zpětnou vazbu, připomínky a podněty k zamyšlení, které významně přispěly ke zlepšení její kvality.

Zvláštní poděkování patří také zkušeným kartografům, panu Janu Potštejnskému a panu Tomáši Leštinskému, za jejich neocenitelnou pomoc s mapovými podklady, poskytnuté konzultace ohledně tvorby map pro orientační běh a za objasnění praktického významu jednotlivých mapových značek v reálném terénu. Jejich expertní vhled byl pro správnou interpretaci dat a modelování klíčový.

V neposlední řadě děkuji své přítelkyni, že to se mnou vydržela, i když jsem ji v posledních měsících kvůli této práci zanedbával.

OBSAH

1 Úvod **11**
2 Přehled metod hledání optimální trasy **13**
2.1 Metody hledání optimální trasy 13
2.1.1 Základní algoritmy prohledávání grafu 14
2.1.2 Pokročilé algoritmy prohledávání grafu 19
2.1.3 Paralelní algoritmy pro problém nejkratší cesty z jednoho zdroje (SSSP) 28
3 Orientační běh: Charakteristika a principy **33**
3.1 Disciplíny orientačního běhu 34
3.2 Mapa a její význam v orientačním běhu 34
3.2.1 Příklad fragmentu mapy 36
3.2.2 OpenOrienteering Mapper a formát `.omap` 37
4 Paralelní výpočty a akcelerace na GPU **41**
4.1 Základy paralelního počítání 41
4.2 Architektura GPU a GPGPU 41
4.3 Platforma NVIDIA CUDA 42
5 Návrh a implementace řešení **45**
5.1 Zpracování vstupních dat 45
5.1.1 Mapová data (Formát XML/`.omap`) 45
5.1.2 Výšková data (Digitální model terénu) 46
5.1.3 Vztah mezi mapovými a výškovými daty 48
5.2 Tvorba grafové reprezentace terénu 48
5.2.1 Diskretizace mapových prvků a rasterizace 48
5.2.2 Zpracování plošných objektů (Scanline Fill vs. Flood Fill) 49
5.2.3 Výpočet vah hran (Traversal Cost) 52
5.2.4 Reprezentace neprůchodných oblastí 53
5.3 Implementace algoritmů pro hledání trasy 54
5.3.1 Implementace CPU algoritmů 55
5.3.2 Implementace paralelních algoritmů s využitím CUDA 63
6 Výsledná aplikace: OMAP Pathfinding Processor **70**
6.1 Architektura a technologie 70
6.2 Uživatelské rozhraní 72
6.2.1 Vstupní soubory a hlavní akce 72
6.2.2 Oblast pro vizualizaci 72
6.2.3 Panel nastavení 73
6.2.4 Menu, nástrojová lišta a stavový řádek 74
6.3 Pracovní postup (Workflow) 74
7 Experimenty a výsledky **76**

7.1 Experimentální prostředí . . . 76
7.2 Testovací data a metodika . . . 76
7.2.1 Testovací mapy a scénáře . . . 76
7.2.2 Metodika měření výkonnosti a kvality . . . 79
7.3 Výsledky syntetických benchmarků . . . 80
7.3.1 Výsledky pro mapu „Les Včelný“ . . . 80
7.3.2 Výsledky pro mapu „Oceán“ . . . 80
7.3.3 Výsledky pro mapu „Orlí vrch“ . . . 81
7.4 Diskuse výsledků syntetických testů . . . 81
7.5 Vizuální analýza a pozorování (Mapa „Oceán“) . . . 83
7.6 Vizuální analýza a pozorování (Mapa „Orlí vrch“) . . . 87
7.7 Souhrnná diskuse a interpretace výsledků . . . 90
8 Závěr . . . 92
Seznam použité literatury . . . 94
Seznam obrázků . . . 97
Seznam příloh . . . 98
A První příloha CMake . . . 99
A.1 Struktura adresářů projektu . . . 99
A.2 Proces sestavení pomocí CMake . . . 100
A.2.1 Klíčové prvky `CMakeLists.txt` . . . 100
A.2.2 Postup sestavení . . . 102
B Druhá příloha Visual Studio . . . 103
B.1 Struktura adresářů a klíčových souborů projektu . . . 103
B.2 Proces sestavení ve Visual Studiu . . . 104
B.2.1 Předpoklady pro sestavení . . . 104
B.2.2 Konfigurace projektu ve Visual Studiu . . . 104
B.2.3 Postup sestavení . . . 105

1 Úvod

Orientační běh představuje sportovní disciplínu, která klade vysoké nároky nejen na fyzickou kondici, ale především na schopnost rychlé a efektivní navigace v neznámém terénu. Nalezení optimální trasy mezi kontrolními body zde není triviálním úkolem minimalizace vzdušné vzdálenosti. Zásadní vliv na rychlost postupu závodníka mají faktory jako převýšení, charakter povrchu, hustota a typ vegetace, vodní toky či další terénní překážky. Problém algoritmického modelování a hledání optimálních cest v orientačním běhu je předmětem zájmu již delší dobu, přičemž první heuristické přístupy se objevily již v 80. letech 20. století [1]. Standardní navigační nástroje, které často pracují pouze s metrikou délky nebo zjednodušenými modely terénu, však často selhávají v predikci reálné časové náročnosti zvolené trasy a neumožňují identifikovat skutečně nejrychlejší variantu postupu. Tento nedostatek, spolu s potřebou zohlednit komplexní faktory ovlivňující pohyb v terénu, jak zdůrazňují i novější práce zabývající se analýzou nejmenších nákladů (Least-Cost Path) na mapách pro orientační běh [2], představuje výzvu pro vývoj pokročilejších metod plánování tras.

Tato diplomová práce se zabývá právě tímto vícekriteriálním problémem hledání časově optimální trasy v orientačním běhu. Hlavním cílem práce je navrhnout, implementovat a vyhodnotit systém, který dokáže zpracovat relevantní mapová data a na jejich základě, s využitím vhodných algoritmů, identifikovat trasu minimalizující předpokládaný čas průběhu závodníka. Systém musí zohledňovat klíčové faktory ovlivňující rychlost pohybu, jako jsou sklon terénu, typ povrchu a průchodnost vegetace, což jsou aspekty zdůrazňované i v současných přístupech k modelování nákladů pohybu [2].

Pro dosažení tohoto cíle práce nejprve řeší problematiku získávání a předzpracování dat. Vstupní data jsou čerpána z mapových souborů ve formátu OMAP pomocí vlastní aplikace Mapper, což umožňuje získat podrobná vektorová data o terénních prvcích. Tato data jsou doplněna o rastrové informace o elevaci terénu. Důraz je kladen na efektivní zpracování i rozsáhlých mapových podkladů s tisíci objekty, včetně řešení výpočetní a paměťové náročnosti.

Následně se práce věnuje reprezentaci mapových dat pro algoritmické zpracování. Vektorová a rastrová data jsou převáděna do grafové struktury, kde vrcholy a hrany reprezentují terén a jejich váhy odrážejí časovou náročnost průchodu daným úsekem (zahrnující délku, převýšení a odpor prostředí). Pro efektivní převod plošných objektů mapy do grafu jsou implementovány a porovnány metody jako flood fill a scan line fill.

Jádrem práce je návrh a implementace algoritmů pro hledání optimální trasy v připraveném grafovém modelu (principy algoritmů viz kapitola. Experimentálně bylo implementováno a porovnáno několik přístupů, které byly specificky adaptovány pro práci s reálnými náklady průchodu terénem (v implementaci využívající vzorkování terénu a modifikovaný nákladový model, např. inspirovaný Toblerovou funkcí, označeno jako 'Tobler-

Sampled'). Mezi zkoumané algoritmy patří jak standardní grafové metody jako Dijkstrův algoritmus a heuristický algoritmus A*, tak i any-angle algoritmy – konkrétně Theta* a jeho optimalizovaná varianta Lazy Theta*. Tyto algoritmy umožňují hledání tras, které nejsou striktně vázány na hrany grafu a mohou tak lépe aproximovat plynulý pohyb v terénu.

Klíčovou výzvou při práci s rozsáhlými mapovými podklady je výpočetní náročnost. Proto byla značná pozornost věnována akceleraci výpočtů pomocí paralelního zpracování na grafické kartě (GPU) s využitím technologie CUDA (teoretický základ viz podkapitola. Za tímto účelem byly vyvinuty a testovány specifické paralelní implementace algoritmů, včetně A* pro GPU, algoritmu Delta-Stepping (který je pro paralelní zpracování obzvláště vhodný) a vlastního algoritmu HADS pro GPU. V závěrečné části práce probíhá vyhodnocení navrženého řešení. Implementované algoritmy jsou testovány na reálných mapách a tratích orientačního běhu. Porovnává se jak jejich výpočetní rychlost a efektivita akcelerace, tak kvalita nalezených tras. Výsledky algoritmů jsou konfrontovány s trasami zvolenými zkušenými orientačními běžci v reálných závodech, což umožňuje posoudit praktickou relevanci a přesnost navrženého systému.

Struktura práce je následující: Po tomto úvodu následuje teoretická část. Ta nejprve v kapitole 2 představuje přehled metod hledání optimální trasy, zahrnující základní, pokročilé i paralelní SSSP algoritmy. Následně se kapitola 3 věnuje specifikům orientačního běhu, jeho disciplínám, mapám a významu mapových značek. Tuto část uzavírá kapitola 4 popisující paralelní výpočty a platformu CUDA. Praktická část práce začíná kapitolou 5, která detailně rozebírá návrh a implementaci řešení, počínaje zpracováním vstupních dat , přes tvorbu grafové reprezentace terénu až po implementaci CPU a GPU algoritmů pro hledání trasy. Vyvinutá desktopová aplikace *OMAP Pathfinding Processor* je představena v kapitole 6, včetně její architektury, uživatelského rozhraní a pracovního postupu. Experimentální ověření a analýza výsledků jsou obsahem kapitoly 7, která popisuje testovací prostředí, metodiku, prezentuje syntetické benchmarky a vizuální porovnání s reálnými daty, a završuje ji souhrnná diskuse. Práce je zakončena kapitolou 8, shrnující dosažené výsledky a navrhující budoucí směřování.

2 Přehled metod hledání optimální trasy

Nalezení skutečně optimální trasy v orientačním běhu představuje komplexní problém, který leží na pomezí několika vědních a technických disciplín. Pro efektivní návrh a implementaci řešení, kterému se věnuje tato práce, je nezbytné nejprve porozumět existujícím teoretickým konceptům, používaným metodám a současným trendům v relevantních oblastech. Tato kapitola proto poskytuje ucelený přehled současného stavu poznání problematiky modelování terénu, výpočtu nákladů pohybu a algoritmů pro hledání nejkratší či časově nejrychlejší cesty.

Stěžejní částí této kapitoly bude přehled základních i pokročilých algoritmů pro hledání nejkratší cesty v grafech, které jsou relevantní pro naši úlohu. Budou popsány principy algoritmů jako Breadth-First Search, Dijkstrův algoritmus, A* a jeho Any-Angle varianty (Theta*, Lazy Theta*). Zvláštní pozornost bude věnována také algoritmům navrženým pro paralelní zpracování, jako je Delta-Stepping, a technologiím umožňujícím akceleraci těchto výpočtů, konkrétně platformě NVIDIA CUDA.

Cílem této rešerše je poskytnout pevný teoretický základ pro vlastní návrh a implementaci systému popsaného v následujících kapitolách, zasadit naši práci do širšího kontextu a identifikovat oblasti, kde existuje prostor pro další výzkum a vylepšení. Porozumění současnému stavu poznání je klíčové pro kritické zhodnocení dosažených výsledků a jejich přínosu.

2.1 Metody hledání optimální trasy

Nalezení optimální trasy mezi dvěma body v mapě představuje klasický výpočetní problém známý jako problém nejkratší cesty (Shortest Path Problem – SPP). V kontextu této práce je cílem najít nejen geometricky nejkratší trasu, ale časově nejvýhodnější trasu mezi startem a cílem, případně mezi dvěma kontrolními body na trati orientačního běhu. Tento úkol je typicky modelován pomocí grafu $G = (V, E)$, kde V je množina vrcholů a E je množina hran spojujících tyto vrcholy.

Vrcholy grafu mohou reprezentovat diskrétní body v terénu (např. středy buněk v mřížce, křižovatky cest, charakteristické body terénu), zatímco hrany reprezentují možné přímé přesuny mezi těmito body. Každé hraně $(u, v) \in E$ je přiřazena váha $w(u, v)$, která v mém případě odpovídá odhadovanému času potřebnému k překonání úseku mezi body u a v. Tato váha není konstantní, ale závisí na více faktorech odvozených z mapových dat [3]:

- Délka úseku mezi u a v.
- Převýšení (sklon terénu) mezi u a v.
- Typ povrchu a vegetace v okolí úseku (ovlivňující průchodnost).

Cílem je tedy nalézt cestu (sekvenci vrcholů a hran) z počátečního vrcholu $s \in V$ do cílového vrcholu $t \in V$ takovou, aby součet vah hran na této cestě byl minimální.

Samotné algoritmy pro hledání cesty musí pracovat s korektně vytvořeným grafem, který realisticky odráží možnosti pohybu v terénu. Návrh vhodné reprezentace mapy jako grafu (viz následující podkapitola o diskrétním prostoru) je proto klíčový, aby se zamezilo nelogickým nebo nemožným přesunům (např. průchod nepřekonatelnou překážkou) [4].

Existuje celá řada algoritmů pro řešení problému nejkratší cesty, které se liší svými vlastnostmi, výpočetní složitostí a vhodností pro různé typy grafů (např. neorientované/orientované, nevážené/vážené, s/bez záporných vah). V následujících podkapitolách představíme základní algoritmy relevantní pro tuto práci [5].

2.1.1 Základní algoritmy prohledávání grafu

Prohledávání do šířky (Breadth-First Search – BFS)
Algoritmus prohledávání do šířky (BFS) je jedním ze základních algoritmů pro systematické prohledávání grafu. Patří mezi tzv. neinformované vyhledávací algoritmy, což znamená, že při výběru dalšího vrcholu k prozkoumání nevyužívá žádné dodatečné informace o vzdálenosti k cíli (heuristiku).

BFS začíná v určeném startovním vrcholu s a postupně prozkoumává graf po úrovních. Nejprve navštíví všechny vrcholy přímo dosažitelné ze startu (úroveň 1), poté všechny dosud nenavštívené vrcholy dosažitelné z vrcholů úrovně 1 (úroveň 2), a tak dále. K udržení pořadí vrcholů k návštěvě využívá datovou strukturu fronta (Queue), fungující na principu FIFO (First-In, First-Out) [6, 7].

Princip fungování:

1. Inicializace: Vytvoří se prázdná fronta Q a množina (nebo pole) `visited` pro označení již navštívených vrcholů. Startovní vrchol s se vloží do fronty Q a označí se jako navštívený. Často se také inicializují pole `distance` (vzdálenost od s) a `parent` (předchůdce na cestě od s). Pro s je $distance(s) = 0$ a $parent(s) = \text{null}$.
2. Hlavní smyčka: Dokud fronta Q není prázdná:
   (a) Vyjme se vrchol u z čela fronty Q.
   (b) Pro každého souseda v vrcholu u:
      i. Pokud v nebyl dosud navštíven (`visited[v]` je false):
         - Označí se v jako navštívený (`visited[v] = true`).
         - Nastaví se $distance(v) = distance(u) + 1$.
         - Nastaví se $parent(v) = u$.
         - Vloží se v na konec fronty Q.

Algoritmus skončí, když je fronta prázdná (prozkoumána celá dosažitelná část grafu) nebo když je nalezen cílový vrchol t (pokud hledáme cestu pouze k němu) [8].

Pseudokód: Následující pseudokód popisuje algoritmus BFS pro nalezení cesty z vrcholu s do vrcholu t [7].

Algoritmus 1 Breadth-First Search (BFS)

```
 1: function BFS(G, s, t)
 2:     Vytvoř frontu Q
 3:     Vytvoř pole visited o velikosti |V|, inicializuj na false
 4:     Vytvoř pole parent o velikosti |V|, inicializuj na null
 5:     Q.enqueue(s)                                  ▷ Vlož startovní vrchol do fronty
 6:     visited[s] ← true
 7:     while Q není prázdná do
 8:         u ← Q.dequeue()                           ▷ Vyjmi vrchol z čela fronty
 9:         if u = t then
10:             return parent ▷ Cíl nalezen, vrať pole předchůdců pro rekonstrukci cesty
11:         end if
12:         for all vrchol v takový, že (u, v) ∈ E (soused u) do
13:             if visited[v] = false then
14:                 visited[v] ← true
15:                 parent[v] ← u
16:                 Q.enqueue(v)                      ▷ Vlož nenavštíveného souseda do fronty
17:             end if
18:         end for
19:     end while
20:     return null                                   ▷ Cíl nebyl nalezen (není dosažitelný)
21: end function
```

Význam a omezení v kontextu hledání nejrychlejší trasy: Pro rekonstrukci cesty po nalezení cíle t stačí postupovat zpětně pomocí pole `parent` od t k s. Přestože BFS nachází cestu s minimálním počtem hran, což neodpovídá mému cíli nalézt časově nejrychlejší cestu ve váženém grafu (kde různé hrany mají různou časovou náročnost), je důležité tento algoritmus pochopit. Slouží jako základní koncept prohledávání grafu, ze kterého vycházejí nebo se s ním porovnávají pokročilejší metody. Ukazuje princip systematického prozkoumávání stavového prostoru (v mém případě grafu reprezentujícího terén). Pro naši úlohu optimalizace ve váženém grafu budou klíčové algoritmy jako Dijkstrův algoritmus a další, kterým se budeme věnovat v následujících kapitolách [8].

Dijkstrův algoritmus

Dijkstrův algoritmus je jedním z nejznámějších a nejzákladnějších algoritmů pro nalezení nejkratších cest v ohodnoceném (váženém) grafu. Na rozdíl od BFS, který je optimální pouze pro nevážené grafy (kde minimalizuje počet hran), Dijkstrův algoritmus pracuje s grafy, kde každá hrana (u, v) má přiřazenou nezápornou váhu $w(u, v) \geq 0$, a hledá cestu

s minimálním součtem těchto vah. Jak bylo zmíněno, graf může reprezentovat různorodé problémy – od dopravních sítí a map přes počítačové sítě až po modelování procesů.

Algoritmus navrhl nizozemský informatik Edsger W. Dijkstra v roce 1956 a publikoval jej v roce 1959. Ve své základní formě najde nejkratší cesty z jednoho určeného startovního vrcholu s ke všem ostatním dosažitelným vrcholům v grafu. Algoritmus lze snadno modifikovat tak, aby se zastavil, jakmile je nalezena nejkratší cesta ke specifickému cílovému vrcholu t [9, 6].

Princip fungování a základní myšlenka: Dijkstrův algoritmus pracuje na greedy (hladovém) principu. Postupně buduje množinu vrcholů S, pro které již byla nalezená definitivní nejkratší cesta od startu s. V každém kroku algoritmus vybere z vrcholů, které ještě nejsou v S, ten vrchol u, který má aktuálně *nejmenší odhadovanou vzdálenost* od startu s. Tento vrchol u je přidán do množiny S a jeho vzdálenost je prohlášena za finální. Následně algoritmus aktualizuje (tzv. relaxuje) odhady vzdáleností pro všechny sousedy vrcholu u, kteří ještě nejsou v S.

Pro efektivní výběr vrcholu s nejmenší aktuální vzdáleností se typicky používá datová struktura prioritní fronta (Priority Queue), která uchovává vrcholy nezařazené v S a umožňuje rychlé nalezení a vyjmutí vrcholu s minimální prioritou (vzdáleností) [9].

Matematický popis a proces relaxace: Algoritmus udržuje pro každý vrchol $v \in V$ následující informace:

- `dist[v]`: Odhad délky (váhy) nejkratší cesty z s do v. Na začátku je $\texttt{dist}[\texttt{s}] = \texttt{0}$ a $\texttt{dist}[\texttt{v}] = \infty$ pro všechna $v \neq s$.
- `parent[v]`: Předchůdce vrcholu v na aktuálně nejlepší nalezené cestě z s do v. Na začátku `parent[v] = null` pro všechna v.

Prioritní fronta Q na začátku obsahuje všechny vrcholy V, přičemž priorita vrcholu v je dána hodnotou `dist[v]` [8].

Hlavní cyklus algoritmu probíhá následovně:

1. Dokud prioritní fronta Q není prázdná:
   (a) Vyjmi z Q vrchol u s nejmenší hodnotou `dist[u]`. Tento vrchol u je nyní považován za definitivně zpracovaný (jeho `dist[u]` je finální délka nejkratší cesty z s do u).
   (b) Pro každého souseda v vrcholu u (tj. pro každou hranu $(u, v) \in E$):
      i. Provede se operace relaxace hrany (u, v): Pokud platí, že cesta do v přes u je kratší než dosud známá nejlepší cesta do v, tedy pokud:

$$\texttt{dist[u]} + w(u, v) < \texttt{dist[v]},$$

pak aktualizujeme informace pro v:

- $\texttt{dist[v]} \leftarrow \texttt{dist[u]} + w(u, v)$
- $\texttt{parent[v]} \leftarrow u$

- Aktualizujeme prioritu vrcholu v v prioritní frontě Q na novou, nižší hodnotu `dist[v]`.

Korektnost a podmínka nezáporných vah: Klíčem ke korektnosti Dijkstrova algoritmu je fakt, že když je vrchol u vybrán z prioritní fronty jako vrchol s nejmenší aktuální vzdáleností `dist[u]`, tato vzdálenost je již skutečně délkou nejkratší cesty z s do u. Toto platí díky podmínce nezáporných vah hran ($w(u, v) \geq 0$).

Intuitivní zdůvodnění: Předpokládejme, že by existovala kratší cesta do u, než je aktuální `dist[u]` v momentě jeho výběru z Q. Tato kratší cesta by musela procházet přes nějaký vrchol x, který ještě nebyl vybrán z Q (jinak by `dist[u]` bylo již dříve relaxováno na nižší hodnotu). Protože všechny váhy hran jsou nezáporné, muselo by platit, že `dist[x]` $\leq$ délka kratší cesty do u. Zároveň by `dist[x]` $<$ `dist[u]` (protože cesta přes x je kratší). To je ale ve sporu s tím, že jsme z Q vybrali právě u jako vrchol s minimální vzdáleností. Proto `dist[u]` musí být v okamžiku výběru optimální.

Pokud by graf obsahoval hrany se zápornou vahou, Dijkstrův algoritmus by nemusel nalézt správnou nejkratší cestu (pro takové případy se používají jiné algoritmy, např. Bellman-Fordův).

Výpočetní složitost: Složitost Dijkstrova algoritmu závisí na implementaci prioritní fronty:

- Použití binární haldy (standardní implementace): Časová složitost je $\mathcal{O}((E+V)\log V)$ nebo častěji uváděná jako $\mathcal{O}(E\log V)$, pokud je graf souvislý ($E \geq V - 1$).
- Použití Fibonacciho haldy: Teoreticky lepší časová složitost $\mathcal{O}(E + V \ \log V)$.
- Bez prioritní fronty (prohledáváním pole vzdáleností v každém kroku, vhodné pro husté grafy): Časová složitost $\mathcal{O}(V^2)$.

Prostorová složitost je $\mathcal{O}(V)$ pro uložení vzdáleností, předchůdců a struktur prioritní fronty.

Pseudokód: Následující pseudokód popisuje Dijkstrův algoritmus pro nalezení nejkratších cest z vrcholu s [8].

Algoritmus 2 Dijkstrův algoritmus

```
 1: function DIJKSTRA(G = (V, E, w), s)
 2:     Vytvoř pole dist o velikosti |V|, inicializuj na ∞
 3:     Vytvoř pole parent o velikosti |V|, inicializuj na null
 4:     dist[s] ← 0
 5:     Vytvoř prioritní frontu Q
 6:     for all vrchol v ∈ V do
 7:         Q.insert(v, dist[v])            ▷ Vlož všechny vrcholy do Q s prioritou dist[v]
 8:     end for
 9:     while Q není prázdná do
10:         u ← Q.extractMin()              ▷ Vyjmi vrchol s nejmenší prioritou (vzdáleností)
11:         if dist[u] = ∞ then
12:             ▷ Pokud je min. vzdálenost nekonečno, ostatní vrcholy jsou nedosažitelné
13:             break
14:         end if
15:         for all vrchol v takový, že (u, v) ∈ E (soused u) do
16:                                                          ▷ Relaxace hrany (u, v)
17:             if dist[u] + w(u, v) < dist[v] then
18:                 dist[v] ← dist[u] + w(u, v)
19:                 parent[v] ← u
20:                 Q.decreaseKey(v, dist[v])               ▷ Aktualizuj prioritu v Q
21:             end if
22:         end for
23:     end while
24:     return dist, parent                 ▷ Vrať pole vzdáleností a předchůdců
25: end function
```

Dijkstrův algoritmus je fundamentálním nástrojem pro řešení problému nejkratší cesty ve vážených grafech s nezápornými hranami. Jeho principy, zejména proces relaxace a využití prioritní fronty pro efektivní výběr dalšího vrcholu k prozkoumání, tvoří základ, na kterém staví nebo se s ním porovnávají mnohé další optimalizační techniky a algoritmy používané v této oblasti. Představuje klíčový koncept pro pochopení metod hledání optimální trasy v modelech, jako je ten náš. Jak uvidíme v dalších kapitolách, existují jeho rozšíření, například algoritmus A*, která přidáním heuristických informací umožňují efektivnější prohledávání rozsáhlých grafů směrem k definovanému cíli. Těmto pokročilejším metodám a jejich aplikaci na problém orientačního běhu se budeme věnovat podrobněji [10, 9].

2.1.2 Pokročilé algoritmy prohledávání grafu

Algoritmus A*

Algoritmus A* je dalším významným algoritmem pro hledání nejkratší cesty v ohodnoceném grafu. Je považován za rozšíření Dijkstrova algoritmu a patří mezi informované vyhledávací algoritmy. Jeho klíčovou vlastností je, že kromě již známé ceny cesty od startu využívá i heuristickou funkci k odhadu ceny zbývající cesty do cíle. Tímto způsobem se snaží „chytřeji“ prohledávat graf a zaměřit se na směry, které se jeví jako nejslibnější pro dosažení cíle, což často vede k výrazně rychlejšímu nalezení optimální cesty než u Dijkstrova algoritmu, zejména v rozsáhlých prostorech.

Algoritmus byl původně popsán Peterem Hartem, Nilsem Nilssonem a Bertramem Raphaelem v roce 1968 a našel široké uplatnění v oblastech jako je umělá inteligence, robotika (plánování pohybu), počítačové hry (hledání cesty postav) a právě i v problémech hledání optimální trasy v geografických informačních systémech (GIS) [11, 10].

Základní myšlenka a heuristická funkce: Hlavní myšlenkou A* je prioritizovat prozkoumávání těch vrcholů, které se zdají být na nejlepší cestě k cíli. K tomu využívá vyhodnocovací funkci f(n) pro každý vrchol n.

$$f(n) = g(n) + h(n)$$

kde:

- $g(n)$ je skutečná cena (váha) nejlepší dosud nalezené cesty ze startovního vrcholu s do vrcholu n. Tato hodnota je stejná jako `dist[n]` v Dijkstrově algoritmu a je postupně zpřesňována.
- $h(n)$ je heuristický odhad ceny (váhy) nejkratší cesty z vrcholu n do cílového vrcholu t. Funkce $h(n)$ je heuristika – informovaný odhad, který nemusí být přesný, ale měl by co nejlépe aproximovat skutečnou zbývající cenu. Kvalita heuristiky $h(n)$ zásadně ovlivňuje výkon algoritmu A*.
- $f(n)$ tedy představuje odhad celkové ceny cesty z s do t procházející přes vrchol n. Algoritmus v každém kroku vybírá k expanzi vrchol s nejnižší hodnotou $f(n)$. [11, 12]

Definice a příklady heuristik: Heuristická funkce $h(n)$ představuje „informovaný odhad“ ceny zbývající cesty z aktuálního vrcholu n do cíle t. Kvalitní heuristika využívá znalosti specifické pro daný problém k odhadnutí této ceny. Cílem je, aby heuristika co nejlépe aproximovala skutečnou nejkratší cestu $h^*(n)$, ale zároveň byla výpočetně nenáročná, protože se vyhodnocuje pro mnoho vrcholů během prohledávání.

V kontextu hledání cesty v geometrickém prostoru (jako je mapa) se často používají heuristiky založené na vzdálenosti:

- **Euklidovská vzdálenost (přímá vzdušná vzdálenost):** Nejběžnější heuristika pro mapy. Vypočítá se jako přímá vzdálenost mezi souřadnicemi vrcholu $n = (x_n, y_n)$

a cíle $t = (x_t, y_t)$:

$$h(n) = \sqrt{(x_n - x_t)^2 + (y_n - y_t)^2}$$

Tato heuristika je obvykle admisibilní, pokud cena cesty (např. čas) neklesá s ujetou vzdáleností. V mém případě, kdy hledáme časově nejrychlejší cestu, může být tato vzdálenost převedena na odhad času například vydělením odhadovanou maximální možnou rychlostí pohybu v daném terénu (nebo průměrnou rychlostí na nejrychlejším povrchu).

- **Manhattanská vzdálenost (taxikářská metrika):** Vhodná pro mřížkové grafy, kde je povolen pouze pohyb ve směru os (horizontálně a vertikálně). Vypočítá se jako součet absolutních rozdílů souřadnic:

$$h(n) = |x_n - x_t| + |y_n - y_t|$$

Je admisibilní, pokud cena pohybu je konstantní a pohyb je omezen na 4 směry.

- **Šachovnicová (Chebyshevova) vzdálenost:** Používá se v mřížkách, kde je povolen i diagonální pohyb za stejnou nebo mírně vyšší cenu než pohyb ve směru os. Vypočítá se jako maximum z absolutních rozdílů souřadnic:

$$h(n) = \max(|x_n - x_t|, |y_n - y_t|)$$

(Přesnější varianty mohou kombinovat diagonální a osové kroky) [13].

Volba vhodné heuristiky je klíčová. Příliš jednoduchá heuristika (např. $h(n) = 0$) snižuje A* na Dijkstrův algoritmus a ztrácí výhodu informovaného prohledávání. Příliš složitá nebo neadmisibilní heuristika může vést k rychlejšímu nalezení (neoptimálního) řešení nebo naopak výpočty zpomalit. Pro náš problém hledání časově nejrychlejší trasy v orientačním běhu je přímá vzdálenost upravená podle maximální možné rychlosti rozumným výchozím bodem pro heuristiku $h(n)$ [12, 11].

Vlastnosti heuristické funkce: Aby algoritmus A* zaručeně našel optimální (nejkratší) cestu, musí heuristická funkce $h(n)$ splňovat určitou podmínku, nejčastěji se vyžaduje admisibilita (přípustnost):

- **Admisibilita:** Heuristika $h(n)$ je admisibilní, pokud pro každý vrchol n platí, že $h(n)$ nikdy nepředcení nebo nenadcení skutečnou cenu nejkratší cesty z n do cíle t. Formálně, $h(n) \leq h^*(n)$, kde $h^*(n)$ je skutečná cena optimální cesty z n do t. Jinými slovy, heuristika je „optimistická“. Příkladem typické admisibilní heuristiky pro geografické problémy je přímá vzdušná vzdálenost (Euklidovská vzdálenost) mezi n a t, pokud cena cesty neklesá s délkou (což platí pro časovou náročnost v mém případě).

Někdy se také pracuje se silnější podmínkou konzistence (monotónnosti):

- **Konzistence:** Heuristika $h(n)$ je konzistentní, pokud pro každou hranu (u, v) s váhou $w(u, v)$ platí trojúhelníková nerovnost: $h(u) \leq w(u, v) + h(v)$. Konzistentní heuristika je vždy i admisibilní. Pokud je heuristika konzistentní, A* prozkoumává vrcholy v pořadí neklesajících hodnot $f(n)$ a zaručuje, že když je vrchol n vybrán k expanzi, byla již nalezena optimální cesta do n (tj. $g(n)$ je finální). To zjednodušuje implementaci, protože není nutné znovu zpracovávat vrcholy, které již byly uzavřeny.

Pokud heuristika není admisibilní, A* může najít cestu rychleji, ale není zaručeno, že bude optimální. Pokud $h(n) = 0$ pro všechny n, pak $f(n) = g(n)$ a algoritmus A* se chová přesně jako Dijkstrův algoritmus [4, 12].

Optimálnost, úplnost a efektivita:

- **Úplnost:** A* je úplný, což znamená, že pokud cesta mezi s a t existuje, algoritmus ji najde (za předpokladu konečného grafu a kladných vah hran).
- **Optimálnost:** A* je optimální (tj. garantuje nalezení nejkratší cesty), pokud je heuristická funkce $h(n)$ admisibilní.
- **Efektivita:** A* je optimálně efektivní pro danou heuristiku, což znamená, že žádný jiný algoritmus používající stejnou heuristiku a zaručující optimalitu nenavštíví méně vrcholů než A* (až na možné rovnosti $f(n)$). V praxi je A* často mnohem rychlejší než Dijkstrův algoritmus, protože heuristika jej „navádí“ správným směrem a prozkoumává podstatně menší část grafu. V nejhorším případě (např. velmi špatná heuristika nebo $h(n) = 0$) se může jeho výkon blížit Dijkstrovu algoritmu.

Výpočetní složitost A* je ve špatném případě stejná jako u Dijkstrova algoritmu, tj. $\mathcal{O}(E \log V)$ nebo $\mathcal{O}(E + V \log V)$ s vhodnou prioritní frontou. V praxi však bývá výrazně lepší, pokud je heuristika kvalitní. Prostorová složitost je dána především velikostí Open a Closed set, v nejhorším případě $\mathcal{O}(V)$ [10].

Pseudokód: Následující pseudokód detailně popisuje algoritmus A* [14, 11, 12].

Algoritmus 3 Algoritmus A* - Reconstruct-Path

```
1: function ReconstructPath(parent, current)
2:     totalPath ← {current}
3:     while parent[current] is not null do
4:         current ← parent[current]
5:         totalPath.prepend(current)          ▷ Přidej na začátek seznamu
6:     end while
7:     return totalPath
8: end function
```

Algoritmus 4 Algoritmus A*

```
 1: function AStar(G = (V, E, w), s, t, h)                 ▷ h je heuristická funkce
 2:     Vytvoř prioritní frontu openSet (min-priorita podle f-skóre)
 3:     Vytvoř množinu closedSet
 4:     Vytvoř pole parent o velikosti |V|, inicializuj na null
 5:     Vytvoř pole gScore o velikosti |V|, inicializuj na ∞
 6:     Vytvoř pole fScore o velikosti |V|, inicializuj na ∞
 7:     gScore[s] ← 0
 8:     fScore[s] ← h(s)                        ▷ Odhad ceny ze startu do cíle přes start
 9:     openSet.add(s, fScore[s])
10:     while openSet není prázdná do
11:         current ← openSet.extractMin()            ▷ Vyjmi vrchol s nejnižším fScore
12:         if current = t then
13:             return ReconstructPath(parent, t)          ▷ Cíl nalezen, vrať cestu
14:         end if
15:         closedSet.add(current)
16:         for all vrchol neighbor takový, že (current, neighbor) ∈ E do
17:             if neighbor ∈ closedSet then
18:                 continue                       ▷ Tento soused již byl plně zpracován
19:             end if
20:             tentative_gScore ← gScore[current] + w(current, neighbor)
21:             if tentative_gScore < gScore[neighbor] then
22:                                            ▷ Našli jsme lepší cestu k tomuto sousedovi
23:                 parent[neighbor] ← current
24:                 gScore[neighbor] ← tentative_gScore
25:                 fScore[neighbor] ← gScore[neighbor] + h(neighbor)
26:                 if neighbor ∉ openSet then
27:                     openSet.add(neighbor, fScore[neighbor])
28:                 else
29:                     openSet.updateKey(neighbor, fScore[neighbor])
30:                                                             ▷ Nebo decreaseKey
31:                 end if
32:             end if
33:         end for
34:     end while
35:     return null                          ▷ Cíl nebyl nalezen (cesta neexistuje)
36: end function
```

Mechanismus algoritmu: A* používá dvě hlavní datové struktury:

- **Open Set (Otevřená množina):** Obvykle implementována jako prioritní fronta. Obsahuje vrcholy, které byly objeveny, ale ještě nebyly plně prozkoumány (tj. jejich sousedé nebyli expandováni). Vrcholy jsou v ní uspořádány podle hodnoty $f(n)$.
- **Closed Set (Uzavřená množina):** Obvykle implementována jako množina (např. hash set) nebo pole příznaků. Obsahuje vrcholy, které již byly plně prozkoumány.

Algoritmus udržuje pro každý relevantní vrchol informace $g(n)$ a $parent(n)$ (předchůdce na nejlepší cestě).
Postup algoritmu:

1. Inicializace: Vytvoří se prázdná Open Set a Closed Set. Do Open Set se vloží startovní vrchol s. Nastaví se $g(s) = 0$ a $f(s) = h(s)$ (nebo $g(s) + h(s)$). Pro ostatní vrcholy $g(n) = \infty$.
2. Hlavní smyčka: Dokud Open Set není prázdná:
   (a) Vyjme se z Open Set vrchol n s nejnižší hodnotou $f(n)$.
   (b) Pokud je n cílový vrchol t, algoritmus končí. Cesta se rekonstruuje zpětně pomocí pole `parent` od t k s.
   (c) Přesune se vrchol n z Open Set do Closed Set.
   (d) Pro každého souseda m vrcholu n:
      i. Pokud je m v Closed Set, přeskočí se (pokud je heuristika konzistentní; u admisibilní, ale nekonzistentní heuristiky může být nutné i uzavřené vrcholy znovu otevřít, pokud se najde lepší cesta).
      ii. Vypočítá se *předběžná* hodnota $g_{tentative}(m) = g(n) + w(n, m)$, kde $w(n, m)$ je váha hrany z n do m.
      iii. Pokud m není v Open Set NEBO pokud $g_{tentative}(m) < g(m)$ (tj. našli jsme lepší cestu do m):
         - Nastaví se `parent[m]` $= n$.
         - Nastaví se $g(m) = g_{tentative}(m)$.
         - Vypočítá se $f(m) = g(m) + h(m)$.
         - Pokud m není v Open Set, přidá se do Open Set s prioritou $f(m)$. Pokud již v Open Set je, aktualizuje se jeho priorita (operace ‘decreaseKey�‘ nebo re-insert).
3. Pokud Open Set je prázdná a cíl nebyl nalezen, cesta mezi s a t neexistuje [10].

Algoritmus A* je díky své efektivitě a garantované optimalitě (při splnění podmínek na heuristiku) jedním z nejpoužívanějších algoritmů pro hledání nejkratší cesty a představuje klíčovou metodu pro řešení problému nastoleného v této práci. Jeho úspěch v praxi však silně závisí na volbě vhodné a výpočetně nenáročné heuristické funkce [14].

Algoritmus Theta* a Any-Angle Pathfinding

Zatímco A* efektivně hledá optimální cestu ve smyslu součtu vah hran v daném grafu, jeho výsledná cesta je přirozeně omezena strukturou tohoto grafu. V případě, že graf reprezentuje diskretizovaný prostor, například mřížku, nalezená cesta se skládá pouze z úseků mezi sousedními vrcholy (buňkami), což může vést k nepřirozeně „zubatým“ nebo zbytečně dlouhým trasám ve srovnání s plynulým pohybem v reálném prostoru. Problém hledání trasy v orientačním běhu, kde se běžec může pohybovat relativně volně terénem, vybízí k hledání metod, které toto omezení překonávají.

Algoritmus Theta* představuje právě takové vylepšení, patřící do rodiny tzv. Any-Angle Pathfinding algoritmů. Tyto algoritmy, ač stále pracují na grafové reprezentaci, se snaží nalézt kratší a přirozeněji vypadající cesty, které nejsou striktně vázány na existující hrany grafu. Theta* staví na základech A* (sdílí jeho mechanismus s Open/Closed Set a funkcí $f(n) = g(n) + h(n)$), ale modifikuje klíčovou část – proces aktualizace vrcholů – tak, aby umožnil vytvářet „zkratky“ mezi nesousedními vrcholy, pokud mezi nimi existuje přímá viditelnost (Line-of-Sight, LoS) [15].

Princip Line-of-Sight (LoS): Zatímco A* při zpracování vrcholu n zvažuje pro jeho souseda m pouze cestu vedoucí přímo z n (s cenou $g(n) + w(n, m)$), Theta* zvažuje i alternativní cestu: cestu vedoucí přímo z předchůdce vrcholu n, tedy z $\texttt{parent}(n)$, do m. Pokud existuje přímá „čára pohledu“ (Line-of-Sight) mezi $\texttt{parent}(n)$ a m – což znamená, že přímý úsek mezi těmito dvěma body neprochází žádnou překážkou (např. neprůchodnou buňkou mřížky) – pak může být vrchol m napojen přímo na $\texttt{parent}(n)$, čímž se potenciálně vytvoří kratší cesta než přes n [15].

Modifikace procesu aktualizace vrcholu (Update Vertex): Hlavní změna oproti A* se odehrává při zpracování souseda m aktuálního vrcholu n, který byl právě vyjmut z Open Set:

1. **Zvaž cestu přes parent(n):** Nejprve se algoritmus podívá na předchůdce aktuálního vrcholu n, označme ho $p = \texttt{parent}(n)$. Vypočítá se potenciální cena cesty do m přes p: $g_{via_p}(m) = g(p) + \text{cost}(p, m)$, kde $\text{cost}(p, m)$ je cena přímého přesunu z p do m (obvykle Euklidovská vzdálenost nebo jiná metrika zohledňující terén podél přímky).
2. **Zkontroluj Line-of-Sight (LoS):** Provede se test, zda existuje přímá viditelnost mezi p a m. Implementace LoS testu závisí na reprezentaci prostředí (např. v mřížce se kontroluje, zda úsečka (p, m) neprotíná žádnou buňku označenou jako překážka).
3. **Aktualizuj, pokud je cesta přes parent(n) lepší A existuje LoS:** Pokud platí, že $g_{via_p}(m) < g(m)$ (cesta přes p je kratší než dosud nejlepší známá cesta do m) a zároveň existuje LoS mezi p a m, pak se aktualizují informace pro m:
   - $\texttt{parent[m]} \leftarrow p$
   - $g(m) \leftarrow g_{via_p}(m)$
   - Vypočítá se $f(m) = g(m) + h(m)$ a m se přidá/aktualizuje v Open Set.

4. **Jinak zvaž standardní cestu přes n (jako v A*):** Pokud podmínky v bodě 3 nejsou splněny (cesta přes p není lepší, nebo neexistuje LoS), algoritmus zváží standardní cestu přes n. Vypočítá $g_{via_n}(m) = g(n)+w(n,m)$ (cena cesty přes hranu (n,m)). Pokud $g_{via_n}(m) < g(m)$, aktualizují se informace pro m standardním A* způsobem:
   - `parent[m]` $\leftarrow n$
   - $g(m) \leftarrow g_{via_n}(m)$
   - Vypočítá se $f(m) = g(m) + h(m)$ a m se přidá/aktualizuje v Open Set.

Tato modifikace umožňuje Theta* generovat cesty, které mohou obsahovat dlouhé přímé úseky, na rozdíl od A*, jehož cesty jsou typicky „zubaté“ v mřížkových reprezentacích [15].
Výkon a optimalita: Theta* může najít výrazně kratší cesty než A* v prostředích, kde jsou možné přímé přesuny. Nicméně, výpočet LoS může být časově náročný, zejména ve složitých prostředích nebo při vysokém rozlišení mřížky. Celková výpočetní náročnost Theta* je proto často vyšší než u A*. Theta* s admisibilní heuristikou najde optimální cestu v tzv. *visibility graphu* (grafu, kde hrany spojují vzájemně viditelné vrcholy), ale tato cesta nemusí být nutně *globálně* nejkratší možná cesta v kontinuálním prostoru, i když k ní bývá blízko [16].

Lazy Theta*
Lazy Theta* je optimalizovaná varianta algoritmu Theta*, která se snaží snížit výpočetní náklady spojené s častými testy Line-of-Sight. Jak název napovídá, její přístup je „líný“ v tom smyslu, že odkládá provedení LoS testu až do okamžiku, kdy je vrchol skutečně vybrán z Open Set k expanzi.
Princip odloženého LoS testu: V standardním Theta*, když je vrchol n expandován, LoS test se provádí pro *každého* jeho souseda m (mezi `parent`(n) a m). V Lazy Theta* se tento test neprovádí ihned. Když je soused m objeven nebo je pro něj nalezena potenciálně lepší cesta přes n, je do Open Set přidán/aktualizován s předpokladem, že jeho předchůdcem je n (jako v A*), bez provedení LoS testu.

Teprve když je vrchol m vyjmut z Open Set jako vrchol s nejnižším f-skóre, provede se LoS test mezi jeho aktuálním (potenciálním) předchůdcem $p =$ `parent`(m) a předchůdcem tohoto předchůdce $p' =$ `parent`(p).

- Pokud existuje LoS mezi p' a m, pak se předchůdce m aktualizuje na p' (`parent`$[m] \leftarrow p'$) a jeho g-hodnota se přepočítá na $g(m) = g(p') + \text{cost}(p', m)$. Toto je klíčový krok Lazy Theta* – finální přiřazení rodiče a g-hodnoty se děje až při extrakci z Open Set.
- Pokud LoS mezi p' a m neexistuje, zůstává předchůdce m nastaven na p (`parent`$[m] \leftarrow p$) a jeho g-hodnota $g(m)$ je ta, se kterou byl do Open Set vložen (tj. $g(p)+w(p,m)$).

Poté algoritmus pokračuje standardní expanzí sousedů vrcholu m.

Pseudokód (Zjednodušený náznak rozdílů):

Zatímco kompletní pseudokód by byl podobný A*/Theta*, zde jsou klíčové rozdíly v logice [16]:

Algoritmus 5 Theta* (rozdíl v expanzi sousedů):

// za caption funguje aj * pro necislovani

1: **function** ExpandNeighbor($n, m, parent, gScore, openSet, h$)
2: $p \leftarrow parent[n]$
3: **if** $p \neq$ `null` **and** LineOfSight(p, m) **then**
4: $g_{via_p} \leftarrow gScore[p] + \text{cost}(p, m)$
5: **if** $g_{via_p} < gScore[m]$ **then**
6: $parent[m] \leftarrow p$; $gScore[m] \leftarrow g_{via_p}$
7: UpdateOpenSet($m, gScore[m] + h(m)$, openSet)
8: **return** ▷ Aktualizováno přes p, konec pro tohoto souseda
9: **end if**
10: **end if** ▷ Jinak (nebo pokud cesta přes p nebyla lepší), zvaž cestu přes n
11: $g_{via_n} \leftarrow gScore[n] + w(n, m)$
12: **if** $g_{via_n} < gScore[m]$ **then**
13: $parent[m] \leftarrow n$; $gScore[m] \leftarrow g_{via_n}$
14: UpdateOpenSet($m, gScore[m] + h(m)$, openSet)
15: **end if**
16: **end function**

Algoritmus 6 Lazy Theta* (rozdíl při extrakci vrcholu a při expanzi sousedů):

```
1: function MainLoop(openSet, closedSet, parent, gScore, h, t)
2:     while openSet není prázdná do
3:         current ← openSet.extractMin()                    ▷ Lazy update kroku
4:         p ← parent[current]
5:         if p ≠ null then
6:             p′ ← parent[p]
7:             if p′ ≠ null and LineOfSight(p′, current) then
8:                 gScore[current] ← gScore[p′] + cost(p′, current)
9:                                                        ▷ Aktualizuj g a parent
10:                parent[current] ← p′
11:            end if
12:        end if
13:        if current = t then return ReconstructPath(...)
14:        end if
15:        closedSet.add(current)
16:        for all soused m vrcholu current do
17:            if m ∈ closedSet then continue
18:            end if
19:                                       ▷ Standardní A* update (bez LoS zde)
20:            g_tentative ← gScore[current] + w(current, m)
21:            if g_tentative < gScore[m] then
22:                parent[m] ← current; gScore[m] ← g_tentative
23:                UpdateOpenSet(m, gScore[m] + h(m), openSet)
24:            end if
25:        end for
26:    end whilereturn null
27: end function
```

Výhody a nevýhody Lazy Theta*: Hlavní výhodou Lazy Theta* je potenciální snížení počtu drahých LoS testů. Test se provádí pouze pro vrcholy, které jsou skutečně vybrány jako součást (potenciálně) nejlepší cesty, nikoli pro všechny generované sousedy. To může vést ke zrychlení v prostředích, kde jsou LoS testy výpočetně náročné.

Nevýhodou může být, že některé vrcholy mohou být vloženy do Open Set s neoptimální (vyšší) g-hodnotou, která je korigována až při jejich extrakci. To může teoreticky vést k mírnému zvýšení počtu expandovaných vrcholů ve srovnání se standardním Theta*, ale v praxi je úspora na LoS testech často významnější. Nalezené cesty jsou typicky stejné nebo velmi podobné jako u standardního Theta* [16].

Porovnání Theta* a Lazy Theta*: Oba algoritmy patří do rodiny Any-Angle a snaží se najít kratší cesty než A* na mřížkách. Theta* provádí LoS testy dříve (při generování sousedů), což může vést k mírně méně expanzím, ale za cenu více LoS testů. Lazy Theta* odkládá LoS testy (až na extrakci vrcholu), čímž snižuje jejich počet, ale může mírně zvýšit počet expanzí kvůli potenciálně méně přesným g-hodnotám v Open Set před finální aktualizací. Volba mezi nimi závisí na relativní ceně LoS testu vůči ceně expanze vrcholu a operací s prioritní frontou. V kontextu hledání cesty pro orientační běh, kde terén může být komplexní a LoS testy mohou být náročné, Lazy Theta* představuje atraktivní variantu [16].

2.1.3 Paralelní algoritmy pro problém nejkratší cesty z jednoho zdroje (SSSP)

Algoritmy jako Dijkstrova algoritmu nebo A* jsou vysoce efektivní pro nalezení nejkratší cesty, avšak jejich povaha je inherentně sekvenční. Zpracování vrcholů je řízeno prioritní frontou, kde v každém kroku typicky vybíráme a zpracováváme jediný vrchol s nejnižší prioritou (vzdáleností nebo f-skóre). Při práci s velmi rozsáhlými grafy, jaké mohou vznikat při detailním modelování terénu pro orientační běh (miliony nebo i miliardy vrcholů a hran), se sekvenční přístup stává výpočetně limitujícím faktorem. Doba výpočtu může být neúnosně dlouhá, což motivuje vývoj a nasazení paralelních algoritmů pro problém nejkratší cesty z jednoho zdroje (Single-Source Shortest Path – SSSP).

Cílem paralelních SSSP algoritmů je rozdělit výpočetní zátěž mezi více výpočetních jednotek (např. jádra CPU nebo masivně paralelní architektury GPU), a tím dosáhnout výrazného zrychlení celého procesu. Jedním z významných algoritmů navržených pro paralelní zpracování SSSP je algoritmus Delta-Stepping.

Algoritmus Delta-Stepping

Algoritmus Delta-Stepping, navržený Ulrichem Meyerem a Peterem Sandersem v článku: Delta-stepping: a parallelizable shortest path algorithm, je efektivní paralelní algoritmus pro SSSP problém v grafech s nezápornými vahami hran. Na rozdíl od Dijkstrova algoritmu, který je *label-setting* (vzdálenost vrcholu je finální, když je vyjmut z prioritní fronty), Delta-Stepping patří mezi *label-correcting* algoritmy. To znamená, že odhad vzdálenosti (`dist`) pro daný vrchol může být během výpočtu vícekrát aktualizován (opravován) na nižší hodnotu, než se dosáhne finální optimální vzdálenosti. Tato flexibilita umožňuje efektivnější paralelizaci [17].

Základní princip a parametr Delta (Δ): Klíčovou myšlenkou algoritmu je rozdělení vrcholů do tzv. šuplíků (buckets) na základě jejich aktuální odhadované vzdálenosti od startu s. Každý šuplík B_i obsahuje vrcholy v, pro které platí $i \cdot \Delta \leq \texttt{dist}[v] < (i+1) \cdot \Delta$, kde $\Delta > 0$ je uživatelem zvolený parametr (šířka šuplíku).

Algoritmus postupuje ve fázích, kde v každé fázi zpracovává vrcholy z aktuálně nejnižšího neprázdného šuplíku B_i. Místo striktního zpracování vždy jen jednoho vrcholu s absolutně nejnižší vzdáleností (jako Dijkstrův algoritmus) umožňuje Delta-Stepping pa-

ralelně zpracovávat všechny vrcholy v rámci jednoho šuplíku, jejichž vzdálenosti jsou si „dostatečně blízké“ (v rozmezí Δ) [17]. Dalším důležitým konceptem je rozdělení hran na **lehké (light)** a **těžké (heavy)**:

- Hrana (u, v) je **lehká**, pokud její váha $w(u, v) \leq \Delta$.
- Hrana (u, v) je **těžká**, pokud její váha $w(u, v) > \Delta$.

Toto rozdělení je zásadní pro řízení relaxací a zachování určité míry správnosti i při paralelním zpracování. [17]

Paralelizmus v Delta-Stepping: Paralelizmus se v algoritmu uplatňuje na několika úrovních:

1. **Relaxace lehkých hran:** Všechny vrcholy u patřící do aktuálně zpracovávaného šuplíku B_i mohou být zpracovány **paralelně**. Pro každý takový vrchol u jsou paralelně relaxovány všechny jeho *lehké* odchozí hrany (u, v). Relaxace znamená výpočet nové potenciální vzdálenosti pro souseda v a případné naplánování jeho přesunu do správného šuplíku. Tyto požadavky na relaxaci (dvojice $\langle v, \text{nová_dist} \rangle$) jsou shromážděny.
2. **Zpracování požadavků na relaxaci:** Po dokončení paralelní relaxace lehkých hran jsou shromážděné požadavky paralelně aplikovány. Pro každý požadavek $\langle v, \text{nová_dist} \rangle$, pokud je nová_dist $<$ `dist`$[v]$, je `dist`$[v]$ aktualizováno a vrchol v je přesunut do odpovídajícího nového (nebo stejného) šuplíku B_j, kde $j = \lfloor \text{nová_dist}/\Delta \rfloor$. Přesuny do šuplíků vyžadují synchronizaci nebo atomické operace pro zajištění korektnosti datových struktur.
3. **Opakování pro lehké hrany v rámci šuplíku:** Kroky 1 a 2 se opakují, dokud v aktuálním šuplíku B_i existují vrcholy, jejichž relaxace lehkých hran může vést k dalším změnám v tomto nebo nižších šuplík.
4. **Relaxace těžkých hran:** Jakmile jsou všechny relaxace lehkých hran pro šuplík B_i dokončeny (tj. šuplík je „stabilizován“ vůči lehkým hranám), jsou paralelně relaxovány všechny *těžké* hrany vycházející z vrcholů, které byly v průběhu zpracování B_i z něj odstraněny. Výsledné požadavky na relaxaci jsou opět shromážděny a paralelně aplikovány, čímž se naplní vyšší šuplíky B_j $(j > i)$.
5. **Přechod na další šuplík:** Po úplném zpracování šuplíku B_i (včetně relaxace těžkých hran) algoritmus přechází k dalšímu nejnižšímu neprázdnému šuplíku B_{i+k} $(k \geq 1)$.

Synchronizace je typicky potřeba mezi jednotlivými fázemi (např. mezi fází relaxace lehkých hran a fází aplikace požadavků, a mezi zpracováním různých šuplíků) [17].

Matematická podstata a korektnost: Algoritmus využívá parametr Δ k nalezení kompromisu mezi striktní sekvenčností Dijkstrova algoritmu a masivní (ale potenciálně neefektivní) paralelizací Bellman-Fordova typu. Tím, že zpracovává vrcholy v rozmezí Δ společně a odděluje relaxaci lehkých a těžkých hran, zajišťuje, že když algoritmus dokončí

zpracování šuplíku B_i, všechny vrcholy v s finální optimální vzdáleností $\texttt{dist}^*[v] \leq i \cdot \Delta$ již tuto vzdálenost mají korektně vypočtenou. Podmínka nezáporných vah hran je stále nutná. Korektnost label-correcting přístupu je zajištěna tím, že i když může být vrchol relaxován „předčasně“ (než je dosažena jeho finální vzdálenost), algoritmus se k němu může později vrátit a jeho vzdálenost opravit, pokud je nalezena kratší cesta. Volba parametru Δ je klíčová pro výkon – příliš malé Δ omezuje paralelizmus, příliš velké Δ může vést k nadbytečné práci (opakované relaxace). Optimální Δ často závisí na charakteristikách grafu (rozložení vah hran) [17].

Pseudokód s náznakem paralelizmu: Celkový algoritmus Delta-Stepping lze rozdělit do několika klíčových fází: inicializace, hlavní smyčka procházející šuplíky, vnitřní smyčka zpracovávající aktuální šuplík (relaxace lehkých hran a aplikace), a fáze relaxace těžkých hran po dokončení zpracování šuplíku.

1. Inicializace: Nejprve se inicializují vzdálenosti, vytvoří se struktura šuplíků a startovní vrchol se vloží do prvního šuplíku.

Algoritmus 7 Delta-Stepping: Inicializace

1: **function** DeltaStepping_Init(V, s)
2: Inicializuj `dist[v]` na ∞ pro $v \in V$
3: `dist[s]` $\leftarrow 0$
4: Vytvoř pole šuplíků B (např. pole seznamů nebo množin)
5: $B[0]$.add(s)
6: **return** `dist`, B
7: **end function**

2. Hlavní smyčka a zpracování šuplíku: Algoritmus iteruje přes indexy šuplíků. Pro každý neprázdný šuplík B_i se inicializují pomocné struktury a spustí se vnitřní smyčka pro zpracování lehkých hran a následně fáze zpracování těžkých hran [17].

Algoritmus 8 Delta-Stepping: Hlavní smyčka

1: // Pokračování z inicializace...
2: $i \leftarrow 0$ ▷ Index aktuálního šuplíku
3: **while** existuje neprázdný šuplík B_k s $k \geq i$ **do**
4: Najdi nejmenší i takové, že B_i není prázdný
5: $S \leftarrow \emptyset$ ▷ Množina vrcholů zpracovaných v této fázi šuplíku i
6: ProcessBucketLightEdges($B, i, S,$ `dist`$, \Delta, G$) ▷ Viz Algoritmus 9
7: ProcessHeavyEdges($S,$ `dist`$, \Delta, G, B$) ▷ Viz Algoritmus 10
8: **end while**
9: **return** `dist`

3. Zpracování lehkých hran (Vnitřní smyčka): Tato část se opakuje, dokud aktuální šuplík B_i není prázdný (stabilizovaný vůči lehkým hranám). V každé iteraci se paralelně relaxují lehké hrany vycházející z vrcholů v B_i, shromáždí se požadavky a ty se následně paralelně aplikují, případně se vrcholy přesouvají mezi šuplíky [17].

Algoritmus 9 Delta-Stepping: Zpracování lehkých hran pro šuplík B_i

```
1:  function ProcessBucketLightEdges(B, i, S, dist, Δ, G)
2:      while B[i] není prázdný do
3:          Req_light ← ∅
4:          parallel for každý vrchol u ∈ B[i] do          ▷ Paralelní relaxace lehkých hran
5:          for all hrana (u, v) taková, že w(u, v) ≤ Δ do
6:              Req_light.add(⟨v, dist[u] + w(u, v)⟩)
7:          end for
8:          end parallel for
9:          S ← S ∪ B[i]                                    ▷ Přidej zpracované vrcholy do S
10:         B[i] ← ∅                                        ▷ Vyprázdni aktuální iteraci šuplíku
11:         parallel for každý požadavek ⟨v, d⟩ ∈ Req_light do    ▷ Paralelní aplikace
12:         if d < dist[v] then
13:             j_old ← ⌊dist[v]/Δ⌋
14:             j_new ← ⌊d/Δ⌋
15:             dist[v] ← d                                 ▷ Potenciálně atomická operace
16:             if j_new < i then
17:                 B[i].add(v)                             ▷ Vrať do aktuálního pro okamžité zpracování
18:             else
19:                 if j_old ≠ j_new and v ∈ B[j_old] then
20:                     Odstraň v z B[j_old]                ▷ Synchronizace/zámek?
21:                 end if
22:                 if v ∉ B[j_new] and j_new ≥ i then      ▷ Přidej pokud tam není
23:                     B[j_new].add(v)                     ▷ Synchronizace/zámek?
24:                 end if
25:             end if
26:         end if
27:         end parallel for
28:     end while
29: end function
```

4. Zpracování těžkých hran: Po stabilizaci šuplíku B_i vůči lehkým hranám se relaxují těžké hrany vycházející ze všech vrcholů u, které byly v průběhu zpracování B_i z něj odstraněny (uloženy v S). Požadavky se opět shromáždí a paralelně aplikují, čímž se naplní vyšší šuplíky [17].

Algoritmus 10 Delta-Stepping: Zpracování těžkých hran

```
 1: function ProcessHeavyEdges(S, dist, Δ, G, B)
 2:     Req_heavy ← ∅
 3:     parallel for každý vrchol u ∈ S do          ▷ Paralelní relaxace těžkých hran
 4:     for all hrana (u, v) taková, že w(u, v) > Δ do
 5:         Req_heavy.add(⟨v, dist[u] + w(u, v)⟩)
 6:     end for
 7:     end parallel for
 8:     parallel for každý požadavek ⟨v, d⟩ ∈ Req_heavy do
 9:                                                  ▷ Paralelní aplikace požadavků
10:     if d < dist[v] then
11:         j_old ← ⌊dist[v]/Δ⌋
12:         j_new ← ⌊d/Δ⌋
13:         dist[v] ← d                              ▷ Potenciálně atomická operace
14:         if j_old ≠ j_new and v ∈ B[j_old] then
15:             Odstraň v z B[j_old]                 ▷ Synchronizace/zámek?
16:         end if
17:         if v ∉ B[j_new] then                     ▷ Přidej jen pokud tam není
18:             B[j_new].add(v)                      ▷ Synchronizace/zámek?
19:         end if
20:     end if
21:     end parallel for
22: end function
```

Algoritmus Delta-Stepping představuje silný nástroj pro akceleraci výpočtu nejkratších cest na moderních paralelních architekturách. Jeho efektivita závisí na správné volbě parametru Δ a na charakteristikách vstupního grafu, ale pro mnoho typů grafů, včetně těch reprezentujících mapy nebo silniční sítě, nabízí významné zrychlení oproti sekvenčním metodám. Implementace na GPU (např. pomocí CUDA) může dále využít masivní paralelizmus pro zpracování jednotlivých vrcholů a hran.

3 Orientační běh: Charakteristika a principy

Orientační běh (OB) je sportovní odvětví kombinující fyzický běh s navigačními schopnostmi. Jeho kořeny sahají na přelom 19. a 20. století, přičemž první oficiální závody se konaly ve Skandinávii (např. Norsko v roce 1897). Základním principem tohoto sportu je absolvování předem určené tratě vyznačené na speciální mapě pomocí kontrolních stanovišť (kontrol) v co nejkratším čase [18]. Historicky se snahy o algoritmické řešení problémů spojených s orientačním během objevují již několik desetiletí, jak dokládá například práce Tsiligiridise [1] zabývající se heuristickými metodami.

Klíčovým prvkem orientačního běhu je samostatná volba postupu mezi jednotlivými kontrolami. Závodník obdrží mapu se zakreslenou tratí až těsně před startem nebo v okamžiku startu. Následně musí během rychlého pohybu terénem interpretovat mapové informace, analyzovat terén před sebou a neustále činit rozhodnutí o nejvhodnější trase k další kontrole. Kromě fyzické náročnosti klade orientační běh extrémní nároky na mentální výkon závodníka. Ten musí nepřetržitě analyzovat mapu a terén, činit rychlá rozhodnutí o volbě postupu a zároveň udržovat vysoké tempo běhu. Sebemenší zaváhání či navigační chyba může vést k významné časové ztrátě, což činí rozhodování pod tlakem nedílnou součástí sportu [19].

Právě tato nutnost volby postupu činí z orientačního běhu komplexní výzvu. Na rozdíl od běžných navigačních úloh, kde se často hledá optimální trasa v rámci definované sítě cest (např. silniční síť), orientační běžec se pohybuje ve volném, často členitém terénu. Optimální postup nemusí nutně kopírovat existující cesty či pěšiny, ale může vést napříč lesem, přes kopce či bažiny, přičemž klíčovým faktorem je odhad reálné průchodnosti a časové náročnosti různých variant [19]. Tento aspekt zdůrazňují i moderní přístupy k analýze tras v OB, které se snaží modelovat náklady na pohyb na základě detailních mapových dat [2].

Identifikovat univerzálně „optimální“ nebo „nejlepší“ trasu je proto obtížné. Ideální postup může záviset nejen na objektivních charakteristikách terénu, ale i na individuálních schopnostech a preferencích závodníka. Zatímco někteří preferují fyzicky náročnější, ale kratší varianty vyžadující přesnou navigaci, jiní mohou volit delší, ale běžecky snazší nebo navigačně jistější postupy po cestách. Cílem závodníka je však vždy minimalizovat výsledný čas, což vyžaduje nalezení trasy, která je pro něj osobně nejrychlejší. Cílem modelování optimální trasy, jakému se věnuje tato práce, je proto identifikovat teoreticky nejrychlejší variantu na základě objektivních terénních charakteristik a kvantifikovatelných faktorů ovlivňujících rychlost pohybu. Existují i jiné formulace optimalizačních problémů v OB, jako je například Orienteering Problem (OP), kde je cílem maximalizovat počet získaných bodů v daném časovém limitu, a pro které byly navrženy specifické algoritmické přístupy, například založené na Self-Organizing Maps [20].

3.1 Disciplíny orientačního běhu

Orientační běh se postupem času diverzifikoval do několika hlavních disciplín, které se liší délkou tratí, charakterem terénu, a tedy i nároky kladenými na závodníka:

- **Klasická trať (Long):** Původní formát závodu, často označovaný jako „královská disciplína". Vyznačuje se dlouhou tratí (čas vítěze v elitní mužské kategorii se obvykle pohybuje kolem 90-100 minut) a klade značné nároky na fyzickou vytrvalost. Zásadní roli hraje strategická volba postupu na dlouhých úsecích mezi kontrolami, často s důrazem na obíhání náročných pasáží nebo využití cest.
- **Krátká trať (Middle):** Tato disciplína je charakteristická kratší délkou (čas vítěze kolem 30-35 minut) a technicky náročnějším terénem s hustší sítí kontrol. Klade větší důraz na přesnou a jemnou práci s mapou, rychlé čtení terénních detailů a častější změny směru. Fyzická náročnost spočívá spíše v intenzitě a schopnosti udržet vysoké tempo v obtížném terénu.
- **Sprint:** Nejkratší a nejrychlejší disciplína (čas vítěze okolo 12–15 minut), která se často odehrává v městském prostředí (parky, sídliště, historická centra). Vyžaduje maximální běžeckou rychlost a především schopnost extrémně rychlého rozhodování a čtení mapy ve vysokém tempu, často s mnoha umělými překážkami a volbami trasy na krátkých úsecích.

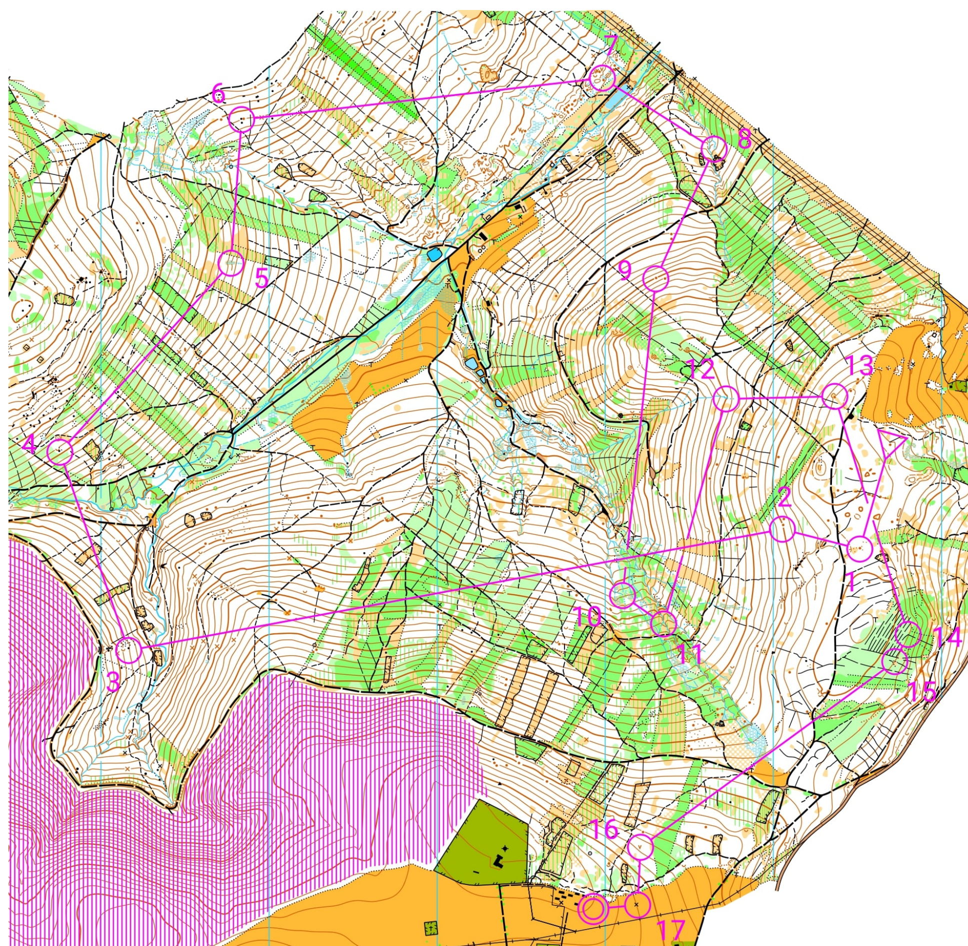
Kromě těchto tří hlavních individuálních disciplín existují i další formáty, jako jsou štafety nebo specifické typy závodů (např. noční OB, lyžařský OB), jejichž pravidla a specifika jsou definována národními i mezinárodními svazy [21]. Pro účely této práce jsou však relevantní především charakteristiky individuálních disciplín a výzvy spojené s volbou postupu v různě náročném terénu. Tsiligirides [1] ve své práci také rozlišuje mezi standardním OB (Orienteering Event, OE) a Score OB (SOE), kde závodníci nemusí navštívit všechny kontroly a snaží se maximalizovat skóre.

3.2 Mapa a její význam v orientačním běhu

Mapa je pro orientačního běžce naprosto esenciálním nástrojem, který umožňuje nejen orientaci v neznámém terénu, ale především informované rozhodování o volbě optimálního postupu. Mapy pro orientační běh jsou specifické svou vysokou podrobností a zaměřením na prvky relevantní pro pohyb běžce terénem, jak ilustruje příklad na Obrázku 1.

Používaná měřítka se liší podle disciplíny: pro lesní disciplíny (klasická, krátká trať) jsou typická měřítka 1:15 000, 1:10 000 nebo 1:7 500, zatímco pro sprint v městském či parkovém prostředí se používají podrobnější měřítka jako 1:5 000 nebo 1:4 000. Tato měřítka umožňují detailní zobrazení terénních tvarů, vegetace, vodních prvků i umělých objektů na relativně velké ploše.

Vzhled a obsah map pro orientační běh jsou standardizovány mezinárodními mapovými klíči, které zajišťují srozumitelnost map po celém světě. Pro lesní disciplíny je určující klíč ISOM 2017-2 (International Specification for Orienteering Maps), zatímco pro sprint platí ISSprOM 2019-2 (International Specification for Sprint Orienteering Maps). Tyto specifikace definují používané symboly, jejich barvy, rozměry a přesný význam [22, 23]. Právě na interpretaci těchto symbolů a jejich převodu na náklady pohybu se zaměřují i moderní přístupy k analýze tras, jako je metoda Least-Cost Path [2].



Obr. 1: Ukázka mapy ze závodu MČR

Barvy a jejich význam

Hnědá – Terénní tvary (Landforms): Základem jsou vrstevnice (Contours, ISOM 101), které spojují místa se stejnou nadmořskou výškou (standardně s ekvidistancí 5 m nebo 2.5 m) a umožňují číst sklon a tvar terénu. Každá pátá vrstevnice je zvýrazněna (Index contour, 102). Patří sem i symboly pro kupy (Small knoll, 109), prohlubně (Small depression, 111; Pit, 112), terénní hrany (Earth bank, 104), či erozní rýhy (Erosion gully, 107).

Černá – Skalní útvary a umělé objekty (Rock and Man-made features): *Skály a kameny:* Nepřelezitelné srázy (Impassable cliff, 201), přelezitelné skalky (Cliff, 202), balvany (Boulder, 204; Large boulder, 205), kamenitá pole (Boulder field, 208) či holá skála (Bare rock, 214).
Umělé objekty: Silnice a cesty různé šířky – široké silnice (502), vozové cesty (504), pěšiny (505, 506), budovy (521), ruiny (523), zdi (513), ploty (516), nepřekonatelné překážky (515, 518), posedy (527), kamenné mužíky (Cairn, 526) [22].
Modrá – Vodní prvky (Water and Marsh): Nepřekonatelné vodní plochy (301), broditelné řeky (302), překonatelné vodní toky (304, 305), malé vodoteče (306), mokřady – neprůchodné (307) a průchodné (308), studánky (312), vodní jámy (303) [22].
Zelená – Vegetace (Vegetation): Udává průběžnost terénu. *Bílá* značí dobře průběžný les (Forest, 405).
Světle zelená: Vegetace zpomalující běh na 60–80 % (406/407).
Středně zelená: Chůze, 20–60 % (408/409).
Tmavě zelená: Neprostupná, pod 20 % (410).
Dále např. významné stromy (417), keře (418), rozhraní vegetace (416) [22].
Žlutá – Otevřené prostranství (Open land): Louky a pole (401), hůře průběžný otevřený terén (403), roztroušené stromy (402/404) [22].
Fialová (Purpurová) – Zákres tratě (Course symbols): Start (701), kontrola (703), číslo kontroly (704), spojnice (705), cíl (706), povinné úseky (707), zakázané oblasti (709), zakázané cesty (711). [22]

Interpretace mapy

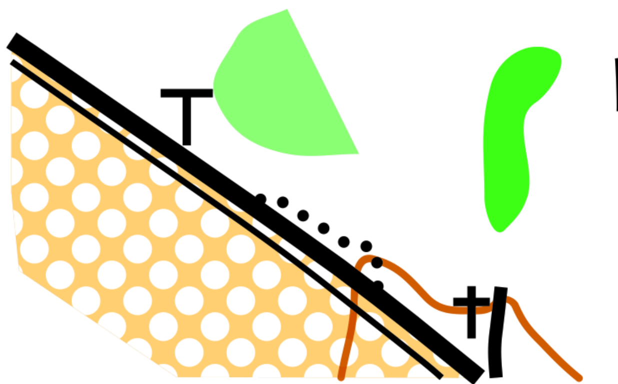
Pro efektivní navigaci musí závodník nejen znát význam jednotlivých symbolů, ale především umět interpretovat jejich kombinace (např. hustý zelený les ve strmém svahu vyznačeném hustými hnědými vrstevnicemi) a na základě těchto informací rychle odhadovat časovou náročnost různých variant postupu [22].

3.2.1 Příklad fragmentu mapy

Pro lepší ilustraci toho, jak se kombinují různé mapové symboly a jak reprezentují reálný terén, se podívejme na následující zjednodušený výřez mapy (viz Obrázek 2):

V tomto malém fragmentu vidíme několik typických prvků:

1. **Černá linie (silnice/cesta):** Dominantní černá linie, pravděpodobně znázorňující cestu (Road, ISOM 503) nebo širší silnici (Wide road, ISOM 502), která rozděluje mapový výřez. Taková komunikace obvykle nabízí rychlý postup.
2. **Žlutá plocha s bílými tečkami (otevřená plocha):** Na jedné straně cesty se nachází žlutá plocha s bílými tečkami, která reprezentuje otevřený terén s roztroušenými stromy (Open land with scattered trees, ISOM 402) nebo případně sad (Orchard, ISOM 413). Obecně se jedná o dobře průběžnou plochu (např. louka, pole).



Obr. 2: Příklad výřezu mapy pro orientační běh

3. **Bílá plocha (běžný les):** Druhá strana cesty je převážně bílá, což standardně značí běžný, dobře průběžný les (Forest, ISOM 405), kde se předpokládá dobrá viditelnost a možnost rychlého běhu.
4. **Zelené plochy (hustší vegetace):** Uvnitř bílého lesa vidíme dva „ostrůvky“ zelené barvy. Světlejší zelená značí oblast se zpomaleným během (Vegetation: slow running, ISOM 406/407), zatímco tmavší/jasnější zelená indikuje oblast, kde je postup výrazně pomalejší, na úrovni chůze nebo boje (Vegetation: walk/fight, ISOM 408-410). Tyto plochy představují pro běžce překážku nebo nutnost zvolit obíhání.
5. **Hnědá linie (vrstevnice):** V dolní části výřezu prochází hnědá vlnitá linie – vrstevnice (Contour, ISOM 101), která ukazuje tvar terénu a změnu nadmořské výšky.
6. **Černé bodové symboly (umělé objekty):** Vidíme zde dva černé symboly. Jeden připomínající písmeno 'T' může představovat posed nebo malou věž (Small tower, ISOM 525). Druhý symbol ve tvaru kříže ('+') se nachází na silnější hnědé čárkované linii (terénní hraně/valu) a pravděpodobně značí přechodové místo (Crossing point, ISOM 519), jako je branka, schůdky nebo průlez přes tuto překážku.
7. **Černá tečkovaná linie:** Podél cesty vede černá tečkovaná linie. Může se jednat o hranici vegetace (Distinct vegetation boundary, ISOM 416), méně zřetelnou pěšinu (Less distinct small footpath, ISOM 507) nebo například rozpadlý plot (Ruined fence, ISOM 517). Její přesný význam by závisel na kontextu okolní mapy.

Tento jednoduchý příklad ukazuje, jak mapa kombinuje informace o průchodnosti (barvy ploch), terénním reliéfu (hnědé linie), komunikacích (černé linie) a specifických objektech (černé symboly), které musí běžec interpretovat pro nalezení nejrychlejší cesty.

3.2.2 OpenOrienteering Mapper a formát `.omap`

Pro tvorbu a úpravu map pro orientační běh existuje vedle komerčních nástrojů také bezplatná open-source alternativa **OpenOrienteering Mapper (OOMapper)**. Jedná se

o multiplatformní software (dostupný pro Windows, macOS, Linux i Android), který poskytuje komplexní nástroje pro kreslení map podle aktuálních mezinárodních standardů, včetně ISOM 2017-2 a ISSprOM 2019-2, stejně jako nástroje pro návrh tratí [24].

Pro účely této práce jsou klíčové následující vlastnosti OOMapperu:

- **Nativní formát `.omap`:** Data jsou ukládána ve vlastním formátu `.omap`, který je založen na jazyce XML. To umožňuje relativně snadný programový přístup k mapovým datům a jejich struktuře, což je zásadní pro automatizované zpracování v rámci této práce. Formát je navržen pro rychlé zpracování a minimalizaci velikosti souboru. Existuje i „upovídanější“ varianta `.xmap`, vhodná pro systémy správy verzí jako je například Git.
- **Podpora `.ocd`:** OOMapper umožňuje import map a symbolových sad z rozšířeného proprietárního formátu `.ocd` (verze 6 až 2018) a export do starších verzí `.ocd` (verze 8 až 12). Tato kompatibilita usnadňuje práci s existujícími mapovými podklady.
- **Georeferencování:** Program podporuje georeferencování map, tedy propojení mapových souřadnic s reálnými geografickými souřadnicemi (např. pomocí systémů jako Krovak JTSK nebo WGS84). Tato funkce je nezbytná pro kombinaci mapových dat s externími daty, jako jsou digitální modely terénu (DMT) pro výpočet elevace a sklonu [24].

Struktura souboru `.omap`

Jak bylo zmíněno, formát `.omap` je založen na XML. Následující zkrácený příklad ukazuje základní strukturu a klíčové části souboru relevantní pro tuto práci:

Výpis 1: Ukázka struktury souboru .omap

```
<map xmlns="http://openorienteering.org/apps/mapper/xml/v2" version="9">
  <notes/>
  <georeferencing
  scale="10000" auxiliary_scale_factor="1.000097"
  declination="3.81" grivation="12.9">
    <projected_crs id="EPSG">
      <spec language="PROJ.4">+proj=krovak +lat_0=49.5 [...]
      +units=m +no_defs</spec>
      <parameter>5514</parameter>
      <ref_point x="-854000" y="-998000"/>
    </projected_crs>
    <geographic_crs id="Geographic coordinates">
      <spec language="PROJ.4">+proj=latlong +datum=WGS84</spec>
      <ref_point_deg lat="50.3406696" lon="12.78741839"/>
    </geographic_crs>
  </georeferencing>
```

```
  <colors count="42">
    <!-- Definice barev pro tisk a~zobrazení -->
    ...
  </colors>
  <barrier version="6" required="0.6.0">
    <symbols count="183" id="OCD">
      <!-- Definice všech použitých mapových symbolů (z~ISOM atd.) -->
      <!-- Každý symbol má své ID, barvu, typ (bod, linie, plocha)... -->
      ...
    </symbols>
    <parts count="1" current="0">
      <part name="Výchozí␣část">
        <objects count="3817">
          <!-- Zde jsou uloženy všechny objekty na mapě -->
          <!-- Příklad plošného objektu (les): -->
          <object symbol="405" type="area">
            <coords>...</coords> <!-- Souřadnice vrcholů polygonu -->
          </object>
          <!-- Příklad liniového objektu (cesta): -->
          <object symbol="505" type="line">
            <coords>...</coords> <!-- Souřadnice bodů linie -->
          </object>
          <!-- Příklad bodového objektu (posed): -->
          <object symbol="527" type="point">
            <coords>...</coords> <!-- Souřadnice bodu -->
          </object>
          ...
        </objects>
      </part>
    </parts>
  </barrier>
</map>
```

Pro účely zpracování dat v této diplomové práci jsou nejdůležitější následující části struktury `.omap`:

- **`<georeferencing>`**: Obsahuje informace nezbytné pro převedení mapových souřadnic na reálné geografické souřadnice a naopak. Zahrnuje použité mapové měřítko (`scale`), definici souřadnicového systému (zde Krovak pomocí specifikace PROJ.4)

a referenční body pro transformaci. Tyto údaje jsou klíčové pro správné umístění mapy vůči datům o elevaci.

- `<symbols>`: Tato podkapitola definuje všechny mapové značky použité v mapě, často načtené ze standardní symbolové sady (např. odpovídající ISOM 2017-2, zde identifikované jako `id=„OCD"`). Každý symbol zde má přiřazen unikátní identifikátor (ID), typ (bod, linie, plocha) a další vlastnosti. Tento seznam slouží jako legenda pro interpretaci objektů v mapě.
- `<objects>`: Toto je nejdůležitější část pro extrakci dat. Obsahuje soupis všech konkrétních objektů zakreslených v mapě. Každý objekt (`<object>`) má přiřazené ID symbolu (`symbol=„..."`), které odkazuje na definici v sekci `<symbols>`, a geometrická data (`<coords>...</coords>`) definující jeho polohu a tvar v mapových souřadnicích (jako bod, linie/lomená čára, nebo polygon/plocha). Právě z těchto objektů získáváme geometrické a sémantické informace (co daný objekt představuje – les, cestu, bažinu atd.) potřebné pro vytvoření grafové reprezentace terénu.

Sekce `<colors>` definuje barvy používané v mapě, což je důležité pro vizuální reprezentaci a tisk, ale pro analýzu průchodnosti terénu v této práci nejsou přímo využívány (informace o typu terénu je nesena ID symbolu objektu). Díky strukturované povaze XML lze tyto informace efektivně parsovat a dále zpracovávat [25] [26].

4 Paralelní výpočty a akcelerace na GPU

Jak bylo naznačeno v předchozí kapitole při popisu algoritmu Delta-Stepping, řešení problému nejkratší cesty na rozsáhlých grafech může být výpočetně velmi náročné. Tradiční sekvenční algoritmy, které provádějí operace jednu po druhé na jednom procesorovém jádře (CPU), mohou pro detailní mapy s miliony vrcholů a hran vyžadovat neakceptovatelně dlouhou dobu výpočtu [27]. Efektivním přístupem ke zrychlení takových úloh je využití *paralelního počítání*, kde je problém rozdělen na menší nezávislé (nebo částečně závislé) části, které jsou zpracovávány současně více výpočetními jednotkami. V posledních desetiletích se jako vysoce výkonná platforma pro masivně paralelní výpočty etablovaly *grafické procesory (GPU).*

4.1 Základy paralelního počítání

Paralelní počítání je výpočetní paradigma, kde jsou instrukce programu prováděny souběžně (paralelně), na rozdíl od sekvenčního provádění. Cílem je zkrátit celkovou dobu výpočtu rozdělením práce. Základními stavebními kameny jsou:

- **Úloha (Task):** Logicky oddělená část výpočtu, která může běžet paralelně s jinými.
- **Proces (Process):** Instance běžícího programu s vlastním adresním prostorem. Paralelizmus mezi procesy je hrubozrnný.
- **Vlákno (Thread):** Lehčí výpočetní jednotka v rámci procesu, která sdílí jeho adresní prostor. Umožňuje jemnozrnnější paralelizmus. Moderní CPU mají několik jader (cores), která mohou současně provádět instrukce více vláken.

Návrh paralelních algoritmů přináší specifické výzvy, jako je *synchronizace* mezi vlákny přistupujícími ke sdíleným datům, *rozdělení zátěže (load balancing)* mezi výpočetní jednotky a minimalizace *komunikační režie.*

4.2 Architektura GPU a GPGPU

Grafické procesory (GPU) byly původně navrženy pro akceleraci grafických operací, jako je renderování 3D scén. Jejich architektura se zásadně liší od CPU: zatímco CPU obsahují několik málo (např. 4-32) vysoce výkonných jader optimalizovaných pro složité sekvenční úlohy a větvení kódu, moderní GPU disponují *stovkami až tisíci jednodušších výpočetních jader*. Tato jádra jsou optimalizována pro provádění stejné operace (nebo velmi podobných operací) na velkém množství dat současně. Tento model se označuje jako *SIMD (Single Instruction, Multiple Data)* nebo přesněji *SIMT (Single Instruction, Multiple Threads)* v případě NVIDIA GPU [27].

Díky této masivně paralelní architektuře se GPU ukázaly jako velmi vhodné i pro negrafické výpočty, které vykazují vysoký stupeň *datového paralelizmu* – tedy mohou být formulovány jako aplikace stejné operace na velké množství datových prvků. Tento přístup se označuje jako *GPGPU (General-Purpose computing on Graphics Processing Units)*. Problémy jako jsou maticové operace, simulace fyzikálních jevů, zpracování signálů a obrazu, strojové učení, a právě i některé grafové algoritmy (včetně SSSP) mohou na GPU dosáhnout řádově vyššího výkonu než na CPU [28].

4.3 Platforma NVIDIA CUDA

CUDA (Compute Unified Device Architecture) je proprietární platforma a programovací model vyvinutý společností NVIDIA, který umožňuje vývojářům využívat výpočetní výkon kompatibilních GPU NVIDIA pro GPGPU. CUDA poskytuje rozšíření pro běžné programovací jazyky (primárně C/C++) a sadu knihoven a nástrojů pro vývoj paralelních aplikací.

Programovací model (Kernels, Threads, Blocks, Grids)
Základním stavebním kamenem CUDA aplikace je *kernel*. Kernel je funkce napsaná programátorem (typicky v C++ s CUDA rozšířeními), která je určena ke spuštění na GPU. Při spuštění kernelu z hostitelského kódu (běžícího na CPU) se specifikuje, kolikrát a v jaké konfiguraci má být tento kernel paralelně vykonán [27]. Výkonný kód kernelu je prováděn masivním počtem *vláken (threads)*. Tato vlákna jsou organizována do hierarchické struktury:

- **Blok (Block):** Skupina vláken (typicky 1D, 2D nebo 3D uspořádání, např. 256 nebo 512 vláken), která mohou vzájemně spolupracovat pomocí rychlé sdílené paměti (shared memory) a synchronizačních bariér `__syncthreads()`. Vlákna v rámci jednoho bloku jsou typicky naplánována ke spuštění na jednom Streaming Multiprocessoru (SM) na GPU.
- **Grid (Mřížka):** Skupina všech bloků (opět 1D, 2D nebo 3D uspořádání), které společně vykonávají daný kernel. Bloky v gridu jsou na sobě obecně nezávislé a nemohou se přímo synchronizovat ani přímo sdílet data přes sdílenou paměť (pouze přes pomalejší globální paměť).

Každé vlákno má unikátní identifikátor v rámci svého bloku (`threadIdx`) a každý blok má unikátní identifikátor v rámci gridu (`blockIdx`). Tyto identifikátory programátor využívá k rozlišení, kterou část dat má které vlákno zpracovat. Tento model umožňuje škálovat výpočet na různě výkonná GPU s různým počtem SM a jader.

Paměťový model

Efektivní využití různých typů pamětí dostupných na GPU je naprosto klíčové pro dosažení vysokého výkonu v CUDA. GPU disponuje hierarchií pamětí s různými vlastnostmi (rychlost, velikost, rozsah platnosti) [27] :

- **Registry (Registers):** Nejrychlejší paměť, privátní pro každé vlákno. Slouží pro lokální proměnné v kernelu. Jsou omezeným zdrojem.
- **Lokální paměť (Local Memory):** Privátní pro každé vlákno, ale fyzicky umístěná v pomalé globální paměti (off-chip DRAM). Používá se, když se lokální data nevejdou do registrů (např. velká lokální pole). Přístup je pomalý.
- **Sdílená paměť (Shared Memory):** Rychlá on-chip paměť (řádově rychlejší než globální), sdílená všemi vlákny *v rámci jednoho bloku.* Umožňuje efektivní komunikaci a spolupráci vláken v bloku. Má omezenou velikost (např. desítky KB na blok/SM). Programátor ji musí explicitně alokovat a spravovat. Je klíčová pro mnoho optimalizačních technik.
- **Globální paměť (Global Memory):** Největší dostupná paměť (GB), ale zároveň nejpomalejší (off-chip DRAM). Je dostupná všem vláknům ve všech blocích (celému gridu) a také CPU (pro přenos dat). Přístupy do globální paměti by měly být minimalizovány a optimalizovány (tzv. coalesced access – souvislý přístup vláken ve warpu k souvislým datům).
- **Konstantní paměť (Constant Memory):** Malá (desítky KB), read-only paměť cacheovaná na GPU. Efektivní, pokud všechna vlákna ve „warpu“ (skupina 32 vláken provádějících instrukce synchronně) čtou stejnou adresu. Vhodná pro uložení konstantních parametrů algoritmu.
- **Texturová paměť (Texture Memory):** Read-only paměť cacheovaná na GPU, optimalizovaná pro prostorovou lokalitu přístupů (typické pro textury v grafice). Může být výhodná i pro některé GPGPU patterny.

Správné rozvržení dat a výpočtů s ohledem na paměťovou hierarchii (minimalizace přístupů do globální paměti, maximální využití registrů a sdílené paměti) je zásadní pro výkon CUDA aplikací [27].

Základní principy vývoje v CUDA

Typický vývojový cyklus CUDA aplikace zahrnuje:

1. **Alokace paměti na GPU:** Vyhrazení prostoru v globální paměti GPU pro vstupní a výstupní data.
2. **Přenos dat z CPU na GPU:** Kopírování vstupních dat z hlavní paměti počítače (host memory) do globální paměti GPU (device memory). Tento přenos přes sběrnici PCIe je relativně pomalý a měl by být minimalizován.

3. **Spuštění kernelu na GPU:** CPU spustí kernel na GPU se specifikovanou konfigurací gridu a bloků. CPU mezitím může pokračovat v jiné práci nebo čekat na dokončení kernelu (asynchronní vs. synchronní spuštění).
4. **Výpočet na GPU:** Vlákna na GPU paralelně provádějí kód kernelu, čtou data z pamětí GPU, provádějí výpočty a zapisují výsledky zpět do pamětí GPU.
5. **Přenos dat z GPU na CPU:** Kopírování výsledků z globální paměti GPU zpět do hlavní paměti CPU.
6. **Uvolnění paměti na GPU.**

CUDA poskytuje API pro všechny tyto kroky. Kód je často kombinací standardního C/C++ (běžícího na CPU - host code) a speciálních funkcí a syntaxe pro kód běžící na GPU (device code - kernels).

Tato platforma a její principy byly využity v této práci pro implementaci a akceleraci paralelních algoritmů hledání nejkratší cesty, jak bude popsáno v následujících kapitolách věnovaných praktické implementaci.

5 Návrh a implementace řešení

Tato kapitola se věnuje praktické realizaci systému pro hledání časově optimální trasy v orientačním běhu. Navazuje na teoretické koncepty popsané v předchozích kapitolách – specifika map pro OB, algoritmy hledání cesty a principy paralelního zpracování na GPU. Budou zde popsány jednotlivé kroky návrhu a implementace, od zpracování vstupních dat, přes tvorbu grafové reprezentace terénu až po implementaci a optimalizaci samotných algoritmů pro hledání cesty na CPU i GPU.

5.1 Zpracování vstupních dat

Základem pro jakoukoliv analýzu a hledání optimální trasy jsou kvalitní vstupní data. Můj systém pracuje se dvěma primárními zdroji informací: digitální mapou pro orientační běh a digitálním modelem terénu (DMT) poskytujícím výšková data.

5.1.1 Mapová data (Formát XML/`.omap`)

Jako zdroj detailních informací o terénních prvcích, vegetaci, komunikacích a překážkách slouží digitální mapa pro orientační běh. V rámci tohoto projektu byla data získávána ze souborů ve formátu XML, který strukturou odpovídá nativnímu formátu `.omap` používanému softwarem OpenOrienteering Mapper (viz podkapitola 3.2.2) nebo exportům z jiných mapových programů (např. OCAD). Tento formát je založen na XML, což usnadňuje jeho strojové zpracování. Klíčové informace extrahované z XML souboru pro potřeby mé aplikace zahrnují:

- **Definice symbolů (`<symbol>`):** Každý symbol používaný v mapě má svou definici, která obsahuje unikátní identifikátor (`id`), kód podle standardu ISOM 2017-2 (`code`, např. „505.1“ pro pěšinu, „408“ pro hustý les) a typ symbolu (`type`, 0=bod, 1=plocha, 2=linie). Tyto definice jsou zásadní pro správnou interpretaci významu mapových objektů.
- **Mapové objekty (`<object>`):** Jednotlivé prvky zakreslené v mapě (stromy, kameny, cesty, plochy lesa, budovy atd.). Každý objekt odkazuje na ID svého symbolu, specifikuje svůj typ (bod, linie, plocha) a obsahuje geometrická data.
- **Geometrická data (`<coords>`):** Souřadnice definující tvar a polohu objektu. Jsou uloženy jako textový řetězec, kde jednotlivé body (vrcholy) jsou odděleny (typicky mezerou nebo středníkem v některých exportech). Každý bod je definován souřadnicemi X a Y v mapovém souřadnicovém systému. Některé body mohou navíc nést speciální příznaky (`flag`), které ovlivňují interpretaci geometrie:
  - `HolePoint (16)`: Označuje začátek sekvence vrcholů definující díru (nevyplněnou oblast) uvnitř plošného objektu.

– `GapPoint (4) / DashPoint (32)`: Označují, že linie nebo segment hranice končící v tomto bodě se nemá kreslit (přerušovaná čára, plot s průchody). Toto je důležité pro správné kreslení hranic, ale i pro logiku tvorby grafu (např. neuzavření cesty).

Načítání a parsování těchto XML dat je prvním krokem zpracování a je realizováno v rámci třídy `MapProcessor` s využitím externí knihovny `tinyxml2` pro efektivní práci s XML strukturou. Výsledkem této fáze je interní reprezentace mapových objektů a jejich geometrie, připravená pro další kroky normalizace a rasterizace.

5.1.2 Výšková data (Digitální model terénu)

Druhým klíčovým vstupem jsou data o nadmořské výšce terénu, typicky ve formě digitálního modelu terénu (DMT). Tato data jsou nezbytná pro výpočet sklonu terénu, který zásadně ovlivňuje rychlost pohybu běžce a je klíčovým prvkem pro výpočet časově nejrychlejší trasy pomocí Toblerovy funkce.

Výšková data pro náš systém vstupují jako samostatný datový zdroj, nezávislý na mapovém souboru. Předpokládá se, že tato data jsou dostupná ve formě **pravidelné rastrové mřížky**, kde každá buňka obsahuje hodnotu nadmořské výšky. Data mohou pocházet z různých zdrojů, například z veřejně dostupných DMT (jako DMR 5G ČÚZK) nebo z jiných GIS systémů. Pro naši aplikaci jsou relevantní následující parametry těchto dat:

- **Hodnoty nadmořské výšky:** Vektor obsahující výškové hodnoty pro všechny buňky mřížky, typicky uložené po řádcích (row-major order).
- **Rozměry mřížky:** Šířka (`elevation_width`) a výška (`elevation_height`) mřížky v počtu buněk.
- **Rozlišení mřížky (`elev_cell_resolution`):** Reálná velikost hrany jedné buňky výškové mřížky v metrech (např. 5m pro DMR 5G). Předpokládáme čtvercové buňky.
- **Souřadnice počátku mřížky (`elevation_origin_x`, `elevation_origin_y`):** Reálné geografické souřadnice (např. v S-JTSK) levého horního (nebo jiného definovaného) rohu výškové mřížky.

Je důležité zdůraznit, že výšková mřížka může (a často bude) mít jiné rozlišení a jiný počátek než logická mřížka vytvořená z mapových dat. Způsob získání těchto dat může být různý, jak popisuje následující podpodkapitola.

Konkrétní implementace získávání výškových dat přes API

Ačkoliv systém umožňuje načítání výškových dat z různých zdrojů (např. lokálních souborů DMT), jednou z implementovaných komponent pro získávání těchto dat je modul využívající externí webové API a skriptovací jazyk Python. Tento přístup poskytuje fle-

xibilitu a umožňuje získat relativně aktuální výšková data pro libovolnou oblast světa bez nutnosti manuálního stahování a přípravy datových sad.

Celý proces je zapouzdřen do skriptu v jazyce Python (`elevation_logic.py`), který je volán z hlavní C++ aplikace. Tento skript komunikuje s veřejně dostupným API (v implementaci použito rozhraní Open Elevation API) a využívá knihovnu `pyproj` pro přesné geodetické transformace mezi souřadnicovými systémy. Proces získání výškové mřížky probíhá v následujících krocích:

1. **Vstupní parametry:** Skript přijímá jako vstup geografické souřadnice (Lat/Lon) a interní mapové souřadnice (získané z georeferencování mapy nebo určené uživatelem) známého kotevního bodu (anchor point), dále rozsah mapových souřadnic (minimální a maximální X, Y) oblasti zájmu, měřítko mapy (`map_scale`) a požadované rozlišení výsledné výškové mřížky v metrech (`desired_resolution_meters`).
2. **Výpočet projektovaných hranic:** Pomocí knihovny `pyproj` a definovaných souřadnicových systémů (WGS84 pro Lat/Lon - EPSG:4326 a S-JTSK pro projektované souřadnice - EPSG:5514) skript nejprve převede kotevní bod na projektované souřadnice. Následně, s využitím měřítka mapy a mapových souřadnic hranic (a s aplikací inverze osy Y pro správnou orientaci), vypočítá přesné hranice oblasti zájmu v projektovaných souřadnicích (např. S-JTSK).
3. **Definice cílové mřížky:** Na základě vypočtených projektovaných hranic a požadovaného rozlišení je určena velikost výsledné výškové mřížky (počet sloupců a řádků). Pro zajištění pokrytí i okrajových částí a pro potřeby interpolace je tato mřížka mírně rozšířena (padding) – typicky o jednu buňku na každé straně. Zároveň je vypočtena souřadnice počátku této výsledné mřížky v projektovaném systému.
4. **Generování dotazovacích bodů:** Pro každou buňku výsledné (rozšířené) mřížky je vypočtena souřadnice jejího středu v projektovaném systému.
5. **Transformace do Lat/Lon:** Všechny vypočtené projektované souřadnice středů buněk jsou hromadně transformovány zpět do geografických souřadnic (Lat/Lon, WGS84), které jsou vyžadovány externím API.
6. **Dotazování API (v dávkách):** Vzhledem k limitům API na počet bodů v jednom dotazu jsou geografické souřadnice odesílány na server API v dávkách (batches) (např. po 100 bodech). Skript postupně odesílá požadavky a shromažďuje vrácené hodnoty nadmořské výšky. Zahrnuje základní ošetření chyb, jako jsou timeouty nebo chyby při parsování odpovědi API (v případě chyby je pro daný bod typicky vrácena hodnota NaN - Not a Number).
7. **Výsledek:** Skript vrací zpět do C++ aplikace kompletní sadu získaných výškových hodnot v jednorozměrném poli spolu s metadaty výsledné mřížky: její šířku, výšku, rozlišení v metrech a souřadnice jejího počátku v projektovaném systému (např. S-JTSK).

Tato komponenta tak poskytuje jeden ze způsobů, jak získat potřebná výšková data. Jak bylo zmíněno, návrh systému umožňuje snadné přidání alternativních zdrojů dat (např. přímé čtení GeoTIFF souborů) implementací odpovídajícího rozhraní pro získání výškové mřížky a jejích metadat.

5.1.3 Vztah mezi mapovými a výškovými daty

Pro správnou kombinaci informací z mapy (typ terénu, překážky) a DMT (sklon) je nezbytné prostorově zarovnat oba datové zdroje. Toho se dosahuje pomocí znalosti jejich reálných geografických souřadnic. Třída `MapProcessor` při zpracování XML mapy (pokud obsahuje georeferencování) nebo na základě souřadnic objektů vypočítá reálné souřadnice počátku (`logical_origin_x`, `logical_origin_y`) a rozlišení (`logical_cell_resolution`) své výsledné logické mřížky. Rozdíl mezi počátkem logické mřížky a počátkem výškové mřížky (`origin_offset_x`, `origin_offset_y`) je pak klíčovým parametrem pro modul `ElevationSampler`, který implementuje správné vzorkování výškových dat na pozicích odpovídajících buňkám logické mřížky během pathfindingu. Tento proces bude detailněji popsán v podkapitole 5.2 a 5.3.1.

5.2 Tvorba grafové reprezentace terénu

Po úspěšném načtení a základním zpracování vstupních mapových a výškových dat je dalším klíčovým krokem jejich transformace do struktury, se kterou mohou efektivně pracovat algoritmy pro hledání nejkratší cesty. Vzhledem k povaze mapových dat a problému hledání cesty v terénu byla zvolena diskrétní reprezentace pomocí pravidelné mřížky (gridu). Tento přístup je standardní v mnoha aplikacích prostorové analýzy a pathfindingu [29, 30]. Tato mřížka, označovaná jako `logical_grid` (výstup třídy `MapProcessor`), slouží přímo jako základ pro grafovou reprezentaci $G = (V, E)$, kde:

- **Vrcholy (Nodes)** V**:** Každá buňka `(x, y)` v logické mřížce představuje jeden vrchol grafu. Celkový počet vrcholů je $V =$ šířka $\times$ výška mřížky.
- **Hrany (Edges)** E**:** Hrany v grafu jsou implicitní a reprezentují možnost přechodu mezi sousedními buňkami mřížky. Typicky se uvažuje 8-směrné sousedství (horizontální, vertikální a diagonální přechody). Hrana tedy existuje mezi buňkou `(x, y)` a jejími 8 sousedy, pokud jsou tito sousedé v mezích mřížky a nejsou označeni jako neprůchodní.

5.2.1 Diskretizace mapových prvků a rasterizace

Převod spojitých nebo vektorových mapových dat (body, linie, plochy) na diskrétní mřížku vrcholů se nazývá rasterizace. Tento proces je implementován v rámci třídy `MapProcessor`. Jak bylo popsáno v kontextu implementace, používá se dvoufázový přístup:

1. **Pass 1 (Parallel Boundaries):** Nejprve jsou paralelně rasterizovány pouze hranice všech objektů, aby se na příslušné buňky mřížky zapsala informace o základní ceně terénu daného objektu a nastavil příznak `FLAG_BOUNDARY`. Neprůchodné objekty (s cenou ≤ 0) zde mají prioritu.
2. **Pass 2 (Rasterization & Rules):** Následně se (paralelně per feature) zpracovávají jednotlivé objekty:
   - **Liniové a bodové objekty:** Jsou rasterizovány pouze jejich hranice/body pomocí Bresenhamova algoritmu, s respektováním příznaků `GapPoint/DashPoint` a bez uzavírání linií.
   - **Plošné objekty:** Je rasterizována jejich hranice (včetně hranic děr) a následně je vyplněn jejich vnitřek.

Výsledkem rasterizace je mřížka `logical_grid`, kde každá buňka `(x, y)` obsahuje strukturu `GridCellData` s informacemi odvozenými z mapových objektů, které tuto buňku pokrývají:

- `value`: Základní multiplikátor ceny průchodu terénem (např. 1.0 pro les, 0.8 pro cestu, 1.67 pro hustník, ≤ 0 pro neprůchodné prvky). Tato hodnota *nezahrnuje* vliv sklonu.
- `flags`: Příznaky jako `FLAG_IMPASSABLE`, `FLAG_ROAD_PATH` atd., které mohou dále ovlivnit logiku pathfindingu.

5.2.2 Zpracování plošných objektů (Scanline Fill vs. Flood Fill)

Korektní zpracování plošných objektů (lesy, louky, vodní plochy, neprůchodné oblasti), včetně těch s dírami, je pro správné určení ceny terénu zásadní.

Zvažovaná metoda: Flood Fill

Během vývoje byla zvažována a experimentálně ověřována metoda Flood Fill (vyplňování záplavou). Tento přístup typicky začíná z jednoho bodu (*seed point*) uvnitř plochy a rekurzivně nebo pomocí fronty/zásobníku navštěvuje a označuje všechny dosažitelné sousední buňky, které nepatří k hranici objektu a nebyly ještě navštíveny.

Algoritmus 11 Flood Fill (Konceptuální pseudokód s frontou)

```
1: function FloodFill(startX, startY, grid, boundarySet)
2:     if není VMezích(startX, startY) or then
        grid[startX, startY] je Hranice or
        grid[startX, startY] je Navštívena
3:         return                              ▷ Neplatný start nebo již zpracováno
4:     end if
5:     Vytvoř frontu Q
6:     Q.enqueue(startX, startY)
7:     Označ grid[startX, startY] jako Navštívena
8:     Přidej startX, startY do VyplněnéOblasti
9:     while Q není prázdná do
10:        P ← Q.dequeue()
11:        for all soused N bodu P do
12:            if je VMezích(N.x, N.y) and then
               N není v boundarySet and
               grid[N.x, N.y] není Navštívena
13:                Označ grid[N.x, N.y] jako Navštívena
14:                Přidej N do VyplněnéOblasti
15:                Q.enqueue(N)
16:            end if
17:        end for
18:    end while
19: end function
```

V mém kontextu se však metoda Flood Fill ukázala jako problematická. Mezi hlavní obtíže patřila nespolehlivost nalezení vhodného startovního bodu uvnitř komplexních tvarů s dírami (algoritmus může snadno začít uvnitř díry nebo mimo objekt) a potenciální neefektivita v případech, kdy bylo nutné testovat mnoho kandidátních bodů nebo kdy rekurze (při rekurzivní implementaci) vedla k přetečení zásobníku u velkých ploch.

Použitá metoda: Scanline Polygon Filling

Z důvodů robustnosti a předvídatelnosti byla pro finální implementaci v rámci `MinimalRasterizer` zvolena metoda Scanline Polygon Filling. Tento algoritmus pracuje po řádcích (scanlines) mřížky. Využívá dvě hlavní datové struktury:

- **Edge Table (ET):** Seznam hran polygonu (a jeho děr), seřazený podle minimální Y souřadnice hrany. Pro každou hranu uchovává její maximální Y souřadnici, X souřadnici v místě minimálního Y a převrácenou hodnotu směrnice (1/m).

- **Active Edge Table (AET):** Seznam hran, které protínají aktuálně zpracovávaný řádek (scanline), dynamicky aktualizovaný a udržovaný seřazený podle X souřadnice.

Algoritmus postupuje následovně:

1. Pro aktuální řádek y:
   (a) Odstraní z AET hrany, pro které y je větší nebo rovno jejich y_{max}.
   (b) Přidá do AET hrany z ET, pro které je y rovno jejich y_{min}.
   (c) Seřadí hrany v AET podle aktuální X souřadnice (x_{curr}).
   (d) Prochází seřazenou AET po dvojicích hran. Pro každou dvojici ($hrana_i, hrana_{i+1}$) vyplní všechny buňky na řádku y mezi X souřadnicemi $\lceil x_{curr,i} \rceil$ a $\lfloor x_{curr,i+1} \rfloor$.
   (e) Aktualizuje x_{curr} pro všechny hrany v AET přičtením převrácené hodnoty směrnice ($x_{curr} += 1/m$).
2. Přejde na další řádek $y + 1$ a opakuje kroky a-e.

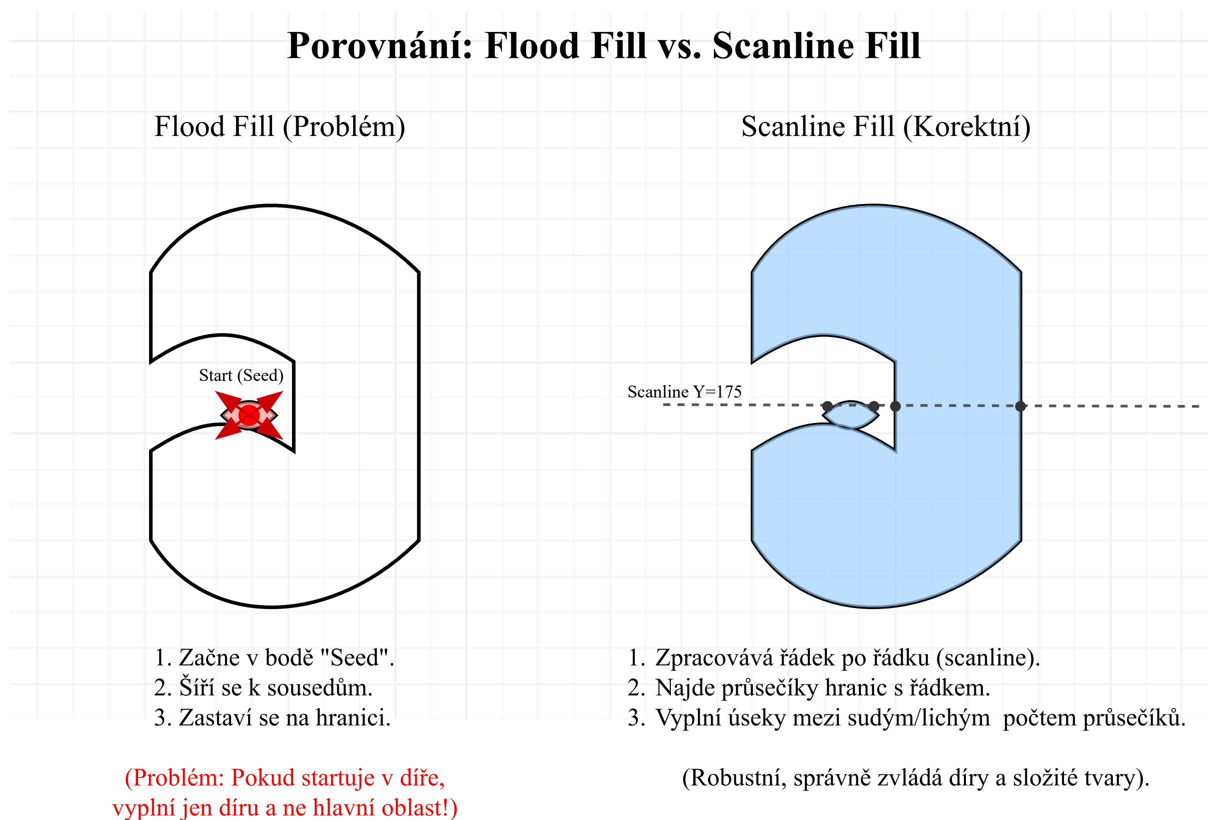
Algoritmus 12 Scanline Polygon Fill (Konceptuální pseudokód)

```
1: function ScanlineFill(polygonVertices, holeVerticesLists)
2:     EdgeTable ← VytvořEdgeTable(polygonVertices, holeVerticesLists)
3:     ActiveEdgeTable ← PrázdnýSeznam
4:     FilledPoints ← PrázdnáMnožina
5:     for y od minY do maxY polygonu do
6:                                             ▷ 1. Odstranit staré hrany z AET
7:         Odstraň z ActiveEdgeTable hrany, kde hrana.ymax ≤ y
8:                                             ▷ 2. Přidat nové hrany z ET do AET
9:         if EdgeTable obsahuje klíč y then
10:            Přidej hrany z EdgeTable[y] do ActiveEdgeTable
11:        end if                              ▷ 3. Seřadit AET podle x
12:        Seřaď ActiveEdgeTable podle hrana.current_x   ▷ 4. Vyplnit mezi páry hran
13:        for i od 0 do ActiveEdgeTable.size() − 2 krok 2 do
14:            x_start ← ActiveEdgeTable[i].current_x
15:            x_end ← ActiveEdgeTable[i + 1].current_x
16:            for x od ⌈x_start⌉ do ⌊x_end⌋ do
17:                FilledPoints.add(x, y)
18:            end for
19:        end for
20:                                             ▷ 5. Aktualizovat x souřadnice v AET
21:        for all hrana H v ActiveEdgeTable do
22:            H.current_x ← H.current_x + H.inv_slope
23:        end for
24:    end for
25:    return FilledPoints
26: end function
```

Tento přístup spolehlivě zvládá komplexní polygony i díry a jeho výkon je méně závislý na tvaru a velikosti plochy ve srovnání s Flood Fill.

Vizuální porovnání metod

Rozdíl ve výsledku a robustnosti mezi metodou Flood Fill a Scanline Fill může být značný, zejména u objektů složitějších tvarů nebo s nevhodně umístěným startovním bodem pro Flood Fill. Obrázek 3 ilustruje typické problémy Flood Fill (např. nevyplnění části oblasti, vyplnění díry) ve srovnání s korektním výsledkem poskytnutým metodou Scanline Fill.



Obr. 3: Ilustrace rozdílu mezi výsledkem metody Flood Fill (vlevo, potenciálně chybný) a Scanline Fill (vpravo, korektní) pro vyplnění plošného objektu.

5.2.3 Výpočet vah hran (Traversal Cost)

Jak bylo naznačeno, graf je tvořen buňkami mřížky a implicitními hranami mezi sousedy. Klíčovým prvkem pro algoritmy hledání cesty je definice váhy (ceny) $w(u, v)$ přechodu z buňky (vrcholu) $u = (x, y)$ do sousední buňky $v = (nx, ny)$. V mém systému tato cena reprezentuje *odhadovaný čas* potřebný k překonání tohoto úseku a není pevně uložena v grafu, ale je počítána dynamicky („on-the-fly") během běhu pathfinding algoritmů (konkrétně A* a jeho variant, viz podkapitola 5.3.1). Výpočet ceny $w(u, v)$ zahrnuje tři hlavní komponenty:

1. **Základní geometrická vzdálenost:** Délka přechodu mezi středy buněk u a v. Pro horizontální a vertikální sousedy je rovna `log_cell_resolution`, pro diagonální sousedy je rovna `log_cell_resolution` $\times \sqrt{2}$. Tato složka je v implementaci reprezentována faktorem `costs[dir]` (1.0 nebo ≈ 1.414).
2. **Základní cena terénu:** Multiplikátor odvozený z typu terénu v cílové buňce v, uložený jako `logical_grid.at(nx, ny).value`. Hodnota > 1 znamená pomalejší terén, < 1 rychlejší (cesty), ≤ 0 neprůchodný. Tato hodnota pochází přímo z interpretace ISOM symbolů mapy.
3. **Penalizace za sklon (Toblerova funkce [31]):** Faktor zohledňující vliv sklonu terénu na rychlost pohybu. Vypočítává se následovně:
   - Získají se nadmořské výšky středu buňky u a v pomocí `ElevationSampler` (který provádí bilineární interpolaci z dat DMT).
   - Vypočítá se výškový rozdíl Δh.
   - Vypočítá se reálná vzdálenost mezi středy buněk Δd (rovna `log_cell_resolution` nebo `log_cell_resolution` $\times \sqrt{2}$).
   - Vypočítá se sklon $S = \Delta h / \Delta d$.
   - Aplikuje se modifikovaná Toblerova „hiking function" pro výpočet faktoru rychlosti: `SlopeFactor` $= e^{-3.5 \cdot |S+0.05|}$.
   - Výsledná časová penalizace za sklon je `time_penalty` $= 1/$`SlopeFactor` (pokud `SlopeFactor` $> \epsilon$, jinak je nekonečná – neprůchodný svah).

Celková cena (čas) přechodu $w(u, v)$ je pak kombinací těchto faktorů:

$$w(u, v) = (\text{Geometrická vzdálenost}) \times (\text{Základní cena terénu}_v) \times (\text{Penalizace za sklon}_{u \to v})$$

$$w(u, v) = (\texttt{costs[dir]}) \times (\texttt{logical_grid.at(nx, ny).value}) \times (\texttt{time_penalty})$$

Tento dynamický výpočet umožňuje přesněji modelovat časovou náročnost pohybu v závislosti na lokálních podmínkách terénu a sklonu [31].

5.2.4 Reprezentace neprůchodných oblastí

Oblasti, kterými běžec nemůže projít (budovy, nepřekonatelné ploty, vodní plochy, velmi hustá vegetace, zakázané oblasti), musí být v grafové reprezentaci adekvátně zachyceny. V mém systému jsou tyto oblasti identifikovány během zpracování mapy a v logické mřížce `logical_grid` jsou označeny buď:

- Základní cenou terénu `value` $\leq 0.0f$.
- Nebo nastavením příznaku `FLAG_IMPASSABLE`.

Algoritmy hledání cesty (např. A*) pak při prohledávání sousedů kontrolují tyto podmínky a buňky označené jako neprůchodné jednoduše ignorují – tj. nevytvářejí hrany vedoucí do těchto buněk a nezařazují je do Open Set. Tím je zajištěno, že nalezená cesta neprochází

zakázanými nebo fyzicky nepřekonatelnými oblastmi. Příznak `FLAG_BOUNDARY` se pro účely průchodnosti ignoruje, slouží primárně pro vizualizaci nebo jiné analýzy.

Tato mřížková struktura s dynamicky počítanými vahami hran, odvozenými z kombinace mapových a výškových dat, tvoří vstupní graf pro všechny implementované pathfinding algoritmy popsané v následující podkapitole.

5.3 Implementace algoritmů pro hledání trasy

Tato podkapitola popisuje konkrétní softwarovou implementaci algoritmů pro hledání nejkratší (časově nejrychlejší) cesty, které byly teoreticky představeny v kapitole 2.1. Implementace byla provedena v jazyce C++ a zahrnuje jak standardní sekvenční varianty běžící na CPU, tak paralelní varianty akcelerované pomocí platformy CUDA na GPU (viz podkapitola 5.3.2). Všechny algoritmy pracují nad grafovou reprezentací terénu ve formě logické mřížky, jak bylo popsáno v podkapitole 5.2, a využívají dynamický výpočet cen hran (viz podkapitola 5.2.3) zohledňující základní cenu terénu a penalizaci za sklon odvozenou z Toblerovy funkce.

Základní algoritmy (BFS a Dijkstrův algoritmu): Pro účely testování, ladění a ověření správnosti vytvořené grafové reprezentace a výpočtu cen byly implementovány i základní grafové algoritmy BFS (funkce `findBFSPath_Tobler_Sampled`) a Dijkstrův algoritmus (`findDijkstraPath_Tobler_Sampled`).

- **BFS:** Jak bylo vysvětleno v teorii (podkapitola 2.1), BFS prohledává graf do šířky a garantuje nalezení cesty s minimálním počtem hran. Jeho implementace je jednoduchá, využívá frontu `std::queue` a pole `visited` a `parents`. V kontextu váženého grafu (kde hrany mají časové náklady) nenachází časově optimální cestu, ale sloužil jako první krok k ověření dosažitelnosti mezi body a správnosti základní logiky pohybu po mřížce a detekce překážek. Jeho implementace v `findBFSPath_Tobler_Sampled` (jak je vidět v poskytnutém kódu) správně ignoruje parametry týkající se výšky a rozlišení, protože pro samotnou logiku BFS nejsou potřeba.
- **Dijkstrův algoritmus:** Teoreticky popsaný v kapitole 2.1, Dijkstrův algoritmus nachází nejkratší cestu ve váženém grafu s nezápornými vahami. Moje implementace `findDijkstraPath_Tobler_Sampled` je velmi podobná implementaci A* (viz níže), používá prioritní frontu, pole `g_scores` a `parents`, a dynamicky počítá cenu přechodu včetně Toblerovy funkce. Hlavní rozdíl oproti A* spočívá v tom, že nepoužívá heuristiku. Priorita fronty je určena pouze hodnotou `g_scores[uzel]`, tedy dosud nalezenou nejlepší cenou cesty od startu. Dijkstrův algoritmus tak prohledává graf systematicky „všemi směry“ od startu a garantuje nalezení optimální cesty, ale může být výrazně pomalejší než A* ve velkých prostorech, protože neupřednostňuje směr k cíli. Sloužil jako důležitý referenční bod pro ověření optimality cest nalezených pomocí A* a jeho variant.

Přestože jsou tyto algoritmy funkční, pro hlavní cíl práce – efektivní hledání časově optimální trasy – nejsou ideální, a proto se dále soustředíme na implementaci A* a jeho Any-Angle variant.

5.3.1 Implementace CPU algoritmů

Hlavní úsilí na CPU bylo věnováno algoritmům, které kombinují přesnost Dijkstrova algoritmu s rychlostí informovaného prohledávání pomocí heuristiky, a metodám, které se snaží generovat přirozenější trasy. Konkrétně se jedná o A*, Theta* a Lazy Theta*.

Implementace A* s Toblerovou funkcí (`findAStarPath_Tobler_Sampled`)
Tato funkce (`AStarToblerSampled.cpp`) představuje základní implementaci A* algoritmu (teorie v podkapitole 2.1.2), přizpůsobenou specifikům naší úlohy – hledání časově optimální cesty v terénu s proměnlivou náročností a sklonem.
Datové struktury a inicializace: Implementace využívá standardní přístup pro A*. Klíčové jsou vektory pro ukládání stavu pro každý vrchol (buňku) mřížky:

- `g_scores`: Ukládá dosud nejlepší nalezenou cenu (čas) cesty od startovního vrcholu. Inicializováno na ∞.
- `f_scores`: Ukládá odhadovanou celkovou cenu cesty přes daný uzel ($f = g + h$). Inicializováno na ∞.
- `parents`: Ukládá index předchůdce na nejlepší cestě pro zpětnou rekonstrukci. Inicializováno na -1.
- `closed`: Příznak (bool), zda byl uzel již finálně zpracován. Inicializováno na `false`.

Deklarace těchto vektorů jako `static thread_local` je technický detail umožňující bezpečné použití funkce ve více vláknech současně (např. pro paralelní testování nebo hledání více cest), aniž by si vlákna přepisovala data. V kontextu jednoho volání se chovají jako statické proměnné alokované při prvním volání ve vlákně.

Výpis 2: Inicializace datových struktur pro A*

```cpp
// --- A* Data Structures ---
// static: persists across function calls within the same thread
// thread_local: each thread gets its own copy
static thread_local std::vector<float> g_scores;
static thread_local std::vector<float> f_scores;
static thread_local std::vector<bool> closed;
static thread_local std::vector<int> parents;
try {
    // Resize and initialize vectors for the current grid size
    g_scores.assign(log_size, std::numeric_limits<float>::max());
    f_scores.assign(log_size, std::numeric_limits<float>::max());
    closed.assign(log_size, false);
    parents.assign(log_size, -1);
```

```
}
catch (const std::bad_alloc&) { /* Handle error */ return resultPath; }

// --- Priority Queue ---
// Custom comparator lambda function for the priority queue
auto cmp = [&](int left_idx, int right_idx) {
    // Primary comparison: lower f_score is higher priority
    if (std::fabs(f_scores[left_idx] - f_scores[right_idx]) > EPSILON) {
        return f_scores[left_idx] > f_scores[right_idx];
        // Note: > for min-heap behavior
    }
    // Secondary comparison (tie-breaker):
    //      lower g_score is higher priority
    // This helps find the optimal path if multiple paths
    // have the same estimated
    // total cost
    return g_scores[left_idx] > g_scores[right_idx];
    };
// Declare the priority queue using the custom comparator
std::priority_queue<int, std::vector<int>, decltype(cmp)> openQueue(cmp);

// --- Initialization ---
g_scores[startIdx] = 0.0f; // Cost from start to start is 0
// Calculate initial f_score using the heuristic estimate to the end node
f_scores[startIdx] = calculate_heuristic(start.x, start.y, end.x, end.y,
heuristic_type);
openQueue.push(startIdx); // Add the start node to the open set
```

Prioritní fronta `openQueue` (implementující *open set*) je klíčová pro efektivitu A*. Použití `std::priority_queue` s vlastním komparátorem `cmp` zajišťuje, že operace `top()` vždy vrátí index vrcholu s nejnižším f-skóre. Komparátor navíc implementuje *tie-breaking* – při rovnosti f-skóre dává přednost vrcholu s nižším g-skóre. To není nezbytně nutné pro nalezení optimální cesty (pokud je heuristika konzistentní), ale může mírně ovlivnit prohledávaný prostor a je to běžná praxe.

Dynamický výpočet ceny přechodu („On-the-fly Cost Calculation“): Nejvýraznějším rysem této implementace je dynamický výpočet ceny přechodu $w(u, v)$ mezi sousedními buňkami $u = (x, y)$ a $v = (nx, ny)$, jak bylo teoreticky popsáno v podkapitole 5.2.3 a ukázáno ve fragmentu kódu níže (viz 3). Tento přístup má několik výhod:

- **Přesnost:** Umožňuje zohlednit lokální podmínky (konkrétní sklon mezi u a v, typ terénu v v) velmi přesně.

- **Flexibilita:** Nevyžaduje předpočítání a ukládání vah všech možných hran do paměti, což by bylo pro velkou mřížku s komplexními vahami neefektivní nebo nemožné. Graf je definován implicitně mřížkou a pravidly pro výpočet ceny.
- **Integrace různých datových zdrojů:** Objekt `ElevationSampler` elegantně zapouzdřuje logiku pro získávání výškových dat z mřížky, která může mít jiné rozlišení a počátek než logická mřížka pathfindingu. Provádí bilineární interpolaci pro získání výšky v přesném (reálném) středu buňky logické mřížky, což zvyšuje přesnost výpočtu sklonu.

Výpis 3: Výpočet ceny přechodu v A*

```
// --- Explore Neighbors ---
for (int dir = 0; dir < NUM_DIRECTIONS; ++dir) {
    const int nx = x + dx[dir];
    const int ny = y + dy[dir];
    // ... (kontrola mezí mřížky a~'closed' setu) ...
    const int neighborIdx = toIndex(nx, ny, log_width);
    const GridCellData& neighborCell = logical_grid.at(nx, ny);

    // Obstacle Check
    if(neighborCell.value<= 0||neighborCell.hasFlag(GridFlags::FLAG_IMPASSABLE))
    { continue; }

    // --- Cost Calculation ---
    // A. Slope
    float world_x_neigh = (static_cast<float>(nx) + 0.5f) * log_cell_resolution;
    float world_y_neigh = (static_cast<float>(ny) + 0.5f) * log_cell_resolution;
    float neighbor_elevation = elevation_sampler.getElevationAt(world_x_neigh,
        world_y_neigh);
    float delta_h = neighbor_elevation - current_elevation;
    // current_elevation získáno dříve
    float delta_dist_world = (dir < 4) ? log_cell_resolution : log_diag_dist;
    float S~= (delta_dist_world > EPSILON) ? (delta_h / delta_dist_world):0.0f;

    // B. Tobler's Factor
    float SlopeFactor = expf(-3.5f * fabsf(S + 0.05f));

    // C. Final Cost
    float time_penalty =
    (SlopeFactor > EPSILON) ? (1.0f / SlopeFactor) : infinite_penalty;
    if (time_penalty >= infinite_penalty) { continue; } // Neprůchodný svah

    float base_terrain_cost = neighborCell.value;       // Základní cena z~mapy
```

```
    float base_geometric_cost = costs[dir];             // 1.0 nebo sqrt(2)
    float final_move_cost = base_geometric_cost * base_terrain_cost * time_penalty;

    // --- Update Neighbor ---
    float tentative_g = current_g + final_move_cost;

    if (tentative_g < g_scores[neighborIdx]) {
        parents[neighborIdx] = currentIdx;
        g_scores[neighborIdx] = tentative_g;
        f_scores[neighborIdx] = tentative_g +
        + calculate_heuristic(nx, ny, end.x, end.y, heuristic_type);
        openQueue.push(neighborIdx);
    }
} // End neighbor loop
```

Výpočet zahrnuje získání výšek, výpočet sklonu S, aplikace Toblerovy funkce ($e^{-3.5|S+0.05|}$) pro získání rychlostního faktoru, jeho převrácení na časovou penalizaci a následné vynásobení základní cenou terénu z logické mřížky a geometrickým faktorem. Následuje standardní krok relaxace A*.

Heuristiky (`calculate_heuristic`): Volba heuristiky zásadně ovlivňuje výkon A*. Funkce `calculate_heuristic` (definovaná v `PathfindingUtils.hpp`) poskytuje několik možností, přepínaných parametrem `heuristic_type`:

- **HEURISTIC_EUCLIDEAN (0):** Přímá vzdušná vzdálenost. Je admisibilní (nikdy nepředcení skutečnou cenu), ale nemusí být nejefektivnější, protože ignoruje omezení pohybu po mřížce a náklady terénu/sklonu.
- **HEURISTIC_DIAGONAL (1):** Diagonální (octile) vzdálenost, vhodná pro mřížky s 8-směrným pohybem. Odhaduje minimální počet kroků. Je také admisibilní, ale stále ignoruje náklady terénu/sklonu.
- **HEURISTIC_MANHATTAN (2):** Manhattanská vzdálenost. Pro 8-směrný pohyb není vhodná ani admisibilní.
- **HEURISTIC_MIN_COST (3):** Škálovaná Diagonální vzdálenost. Bere diagonální vzdálenost (minimální počet kroků) a násobí ji odhadem minimálního možného kombinovaného nákladového faktoru (terén $\times$ sklonová penalizace), zde natvrdo 0.8. Cílem je, aby heuristika zůstala admisibilní (nepodcenila časovou náročnost) i na velmi rychlých úsecích (cesta z kopce). Jedná se o nejpřesnější a teoreticky nejefektivnější admisibilní heuristiku z nabízených možností pro náš problém.

Pro náš problém je nejvhodnější Diagonální nebo (lépe) Škálovaná Diagonální heuristika (`HEURISTIC_MIN_COST`).

Ostatní aspekty: Implementace korektně se stará o překážky (kontrola `value <= 0.0f` nebo `FLAG_IMPASSABLE`) a obsahuje robustní rekonstrukci cesty ze startu do cíle pomocí pole `parents`.

Implementace Theta* (`findThetaStarPath_Tobler_Sampled`)
Algoritmus Theta* (teorie v podkapitole 2.1.2) byl implementován s cílem generovat vizuálně přirozenější a potenciálně kratší trasy než standardní A* na mřížce. Motivace pro tento Any-Angle přístup vychází přímo z povahy orientačního běhu – závodník se v terénu nemusí striktně držet os mřížky nebo předdefinovaných cest, ale může si zvolit přímý směr přes otevřenou plochu nebo řidší les, pokud mu to terén a viditelnost dovolí. Theta* se snaží tuto možnost modelovat tím, že umožňuje „zkratky“ mezi vrcholy, které nejsou přímými sousedy v mřížce.
Princip Line-of-Sight (LoS) a výpočty segmentů: Jádrem Theta* je schopnost ověřit přímou viditelnost mezi dvěma vrcholy a spočítat cenu (čas) pohybu po této přímé úsečce. V naší implementaci jsou pro tyto účely klíčové dvě pomocné funkce (definované v rámci `ThetaStarToblerSampled.cpp`, uvnitř anonymního namespace):

- **`hasLineOfSight(x0, y0, x1, y1, ...)`:** Tato funkce používá Bresenhamův algoritmus (viz implementace v `getLineSegmentCells`) k identifikaci všech buněk mřížky, které protíná úsečka mezi body $(x0, y0)$ a $(x1, y1)$. Následně pro každou tuto buňku (s výjimkou startovní buňky $(x0, y0)$) zkontroluje, zda není neprůchodná (tedy zda `logical_grid.at(cx, cy).value > EPSILON` a zároveň nemá nastaven příznak `FLAG_IMPASSABLE`). Pokud je nalezena jakákoliv neprůchodná buňka na úsečce, funkce vrací `false`, jinak vrací `true`. Je to klíčový, ale potenciálně náročný krok.
- **`calculateSegmentCost(x_start, y_start, x_end, y_end, ...)`:** Pokud `hasLineOfSight` potvrdí viditelnost, tato funkce spočítá časovou náročnost pohybu po přímé úsečce mezi (x_{start}, y_{start}) a (x_{end}, y_{end}). Opět interně využívá `getLineSegmentCells` k získání sekvence buněk podél přímky. Následně iteruje přes jednotlivé „mikro-kroky“ mezi středy po sobě jdoucích buněk na této úsečce. Pro každý mikro-krok vypočítá jeho délku (`delta_dist_world`), získá výšky pomocí `ElevationSampler`, vypočítá sklon S, Toblerův faktor a časovou penalizaci (`time_penalty`). Nakonec vynásobí délku mikro-kroku časovou penalizací a základní cenou terénu *cílové* buňky daného mikro-kroku (`logical_grid.at(current_gp.x, current_gp.y).value`). Součet těchto nákladů pro všechny mikro-kroky tvoří celkovou cenu segmentu. Tento výpočet je ještě náročnější než LoS test, protože zahrnuje opakované volání `ElevationSampler` a výpočty sklonu.

Kvalita a výkon Theta* silně závisí na efektivitě těchto dvou funkcí.

Modifikovaná relaxace sousedů: Největší změna oproti A* nastává v hlavní smyčce při zpracování sousedů `neighborIdx = (nx, ny)` vrcholu `currentIdx = (x, y)`. Místo jednoduché relaxace hrany $(current, neighbor)$ se provádí komplexnější logika:

Výpis 4: Optimalizovaná logika zpracování souseda v Theta*

```
// Předpoklady: definované x, y (current), nx, ny (soused), currentIdx, atd.
// const float current_g = g_scores[currentIdx]; // Již známo
// const int parent_of_currentIdx = parents[currentIdx]; // Již známo

float tentative_g;
int chosen_parentIdx;

// 1. VÝCHOZÍ STAV: Standardní A* krok (aktuální uzel -> soused)
chosen_parentIdx = currentIdx;
float cost_current_to_neighbor = calculateSegmentCost(x, y, nx, ny, logical_grid,
                                    elevation_sampler, infinite_penalty);

// Pokud je soused nedosažitelný ani standardním A* krokem, přeskočíme ho.
if (cost_current_to_neighbor >= infinite_penalty) {
    continue; // Přeskoč na dalšího souseda
}
tentative_g = current_g + cost_current_to_neighbor;



// 2. THETA* LOGIKA: Pokus o~přímé spojení (rodič aktuálního -> soused)
if (parent_of_currentIdx != -1) { // Aktuální uzel má rodiče (není start)
    int grandparent_x, grandparent_y;
    toCoords(parent_of_currentIdx, log_width, grandparent_x, grandparent_y);

    // Je viditelnost od prarodiče (rodiče aktuálního) k~sousedovi?
    if (hasLineOfSight(grandparent_x, grandparent_y, nx, ny,
                       logical_grid, log_width, log_height))
    {
        float cost_grandparent_to_neighbor = calculateSegmentCost(
            grandparent_x, grandparent_y, nx, ny,
            logical_grid, elevation_sampler, infinite_penalty);

        // Pokud je cesta od prarodiče k~sousedovi průchodná
        if (cost_grandparent_to_neighbor < infinite_penalty) {
            float g_via_grandparent = g_scores[parent_of_currentIdx]
                                    + cost_grandparent_to_neighbor;
```

```
            // A~pokud je tato cesta kratší než dosud nejlepší
            if (g_via_grandparent < tentative_g) {
                tentative_g = g_via_grandparent;
                chosen_parentIdx = parent_of_currentIdx;
            }
        }
    }
    // Else: není viditelnost nebo cesta od prarodiče neprůchodná
    // -> zůstává standardní A* krok (již nastaveno).
}
// Else: aktuální uzel je start -> použije se standardní A* krok (již nastaveno).


// 3. AKTUALIZACE SOUSEDA, pokud byla nalezena lepší cesta
if (tentative_g < g_scores[neighborIdx]) {
    parents[neighborIdx] = chosen_parentIdx;
    g_scores[neighborIdx] = tentative_g;
    f_scores[neighborIdx] = tentative_g +
        calculate_theta_heuristic(nx, ny, end.x, end.y, elevation_sampler /*,...*/);
    openQueue.push(neighborIdx);
}
```

Implementace tedy pečlivě zvažuje obě možnosti (přímou cestu od rodiče vs. standardní krok) a volí tu výhodnější a proveditelnou.

Rekonstrukce cesty a význam pro OB: Protože pole `parents` nyní obsahuje odkazy, které mohou přeskakovat sousední buňky (při použití LoS zkratky), přímým sledováním rodičů získáme pouze sekvenci bodů – bodů, kde se mění směr nebo kde byla LoS zkratka využita. Pro získání finální cesty po jednotlivých buňkách mřížky, která reprezentuje skutečný pohyb, je nutné provést interpolaci mezi těmito body. Po získání reverzní sekvence bodů a jejím otočení implementace iteruje přes dvojice ($waypoint_i, waypoint_{i+1}$) a pro každou dvojici volá funkci `getLineSegmentCells`, která vrátí všechny buňky mřížky ležící na úsečce mezi nimi. Tyto buňky (kromě první, která byla přidána v předchozím kroku) se postupně přidají do výsledného vektoru cesty. Tento Any-Angle přístup má velký potenciál pro modelování tras v OB, protože výsledné cesty (zejména na otevřených prostranstvích) lépe kopírují plynulý pohyb běžce a mohou být geometricky kratší a časově rychlejší než striktně mřížkové cesty z A*. Výsledná trasa je vizuálně hladší. Cenou za to je však vyšší výpočetní náročnost kvůli opakovaným LoS testům a výpočtům ceny celých segmentů (`calculateSegmentCost`), které mohou zahrnovat mnoho mikro-kroků a volání `ElevationSampler`.

Implementace Lazy Theta* (`findLazyThetaStarPath_Tobler_Sampled`)

Lazy Theta* (teorie v podkapitole 2.1.2) je optimalizací Theta* s cílem snížit počet výpočetně náročných volání funkcí `hasLineOfSight` a `calculateSegmentCost`. Princip spočívá v odložení

kontroly LoS a potenciální „přepojení“ na prarodiče až na moment, kdy je uzel vybírán z prioritní fronty (*open set*).

Lazy Update Krok: Hlavní rozdíl oproti Theta* je začlenění „Lazy Update“ kroku na začátek hlavní smyčky A*, ihned po vyjmutí vrcholu `currentIdx` z `openQueue`. Tento krok kontroluje, zda by cesta k `currentIdx` nebyla kratší, pokud by vedla přímo od jeho prarodiče (`grandParentIdx`) namísto od rodiče (`parentIdx`).

Výpis 5: Lazy Update krok v Lazy Theta*

```
while (!openQueue.empty()) {
    const int currentIdx = openQueue.top();
    openQueue.pop();

    if (closed[currentIdx]) { continue; }
    // Uzel již mohl být zpracován dříve s~lepší cestou

    // --- Lazy Update Step ---
    int parentIdx = parents[currentIdx];
    if (parentIdx != -1) { // Current není start
        int grandParentIdx = parents[parentIdx];
        if (grandParentIdx != -1) { // Rodič current není start
            int x_curr, y_curr, x_gp, y_gp; // Souřadnice
            toCoords(currentIdx, log_width, x_curr, y_curr);
            toCoords(grandParentIdx, log_width, x_gp, y_gp);

            // 1. Zkontroluj LOS od prarodiče k~aktuálnímu vrcholu
            if (hasLineOfSight(x_gp, y_gp, x_curr, y_curr, logical_grid, ...)){
                // 2. Spočítej cenu přímého segmentu od prarodiče
                float segment_cost=calculateSegmentCost(x_gp,y_gp,x_curr,y_curr);

                if (segment_cost < infinite_penalty) { // Cesta je průchodná
                    // 3. Vypočítej potenciální g-skóre přes prarodiče
                    float g_via_grandparent = g_scores[grandParentIdx] +
                    + segment_cost;
                    // 4. Pokud je cesta přes prarodiče kratší NEŽ AKTUÁLNÍ
                    // g-skóre
                    // (které mohlo být nastaveno cestou přes rodiče parentIdx)
                    if (g_via_grandparent < g_scores[currentIdx]) {
                        // AKTUALIZUJ cestu pro currentIdx!
                        g_scores[currentIdx] = g_via_grandparent; // Lepší cena
                        parents[currentIdx] = grandParentIdx;
                        // Změň rodiče na prarodiče
```

```
                    // Důležité: Přepočítej i~f-skóre pro konzistenci
                    // (i~když už je venku z~PQ)
                    f_scores[currentIdx] =
                    = g_scores[currentIdx] + calculate_theta_heuristic(...);
                }
                //else: Cesta přes rodiče byla lepší nebo stejná, nic neměň
            }
            // else: Přímá cesta od prarodiče není průchodná, nic neměň
        }
        // else: Není LOS od prarodiče, nic neměň
    }
    // else: Rodičem je start, není prarodič, nic neměň
}
// else: Current je start, nemá rodiče, nic neměň
// --- Konec Lazy Update Stepu ---
closed[currentIdx] = true; // Označ jako uzavřený AŽ PO lazy update
if (currentIdx == endIdx) { break; } // Cíl nalezen
// ... (Zpracování sousedů jako ve standardním A*) ...
```

Tento krok se provádí pouze jednou pro každý uzel, když je finálně zpracováván.
Zpracování sousedů (jako A*): Po provedení Lazy Update kroku (který mohl změnit `g_scores[currentIdx]` a `parents[currentIdx]`), následuje zpracování sousedů vrcholu `currentIdx`. Tato část je nyní výrazně jednodušší než u Theta* – probíhá přesně jako ve standardním A* algoritmu (viz listing 3). Pro každého souseda `neighborIdx` se spočítá pouze cena jednoho kroku `step_cost = calculateSegmentCost(current, neighbor, ...)` a provede se standardní relaxace bez dalších LoS testů.
Výhody a nevýhody: Odložením LoS testu Lazy Theta* výrazně snižuje počet volání výpočetně náročných funkcí `hasLineOfSight` a `calculateSegmentCost`. Očekává se proto, že bude rychlejší než standardní Theta*, zejména na velkých mapách nebo v případech, kde jsou LoS testy komplexní. Drobnou teoretickou nevýhodou může být, že některé vrcholy mohou vstoupit do prioritní fronty s mírně nadhodnoceným g-skóre (protože se ještě nezkusila zkratka přes prarodiče), což by mohlo vést k prozkoumání o něco více vrcholů než u Theta*. V praxi však bývá úspora na LoS testech dominantní. Kvalita nalezené cesty by měla být srovnatelná s Theta*. Rekonstrukce cesty je opět stejná jako u Theta* (přes body a interpolaci).

Všechny tyto CPU implementace byly důkladně testovány a slouží jako základ pro další optimalizace a jako referenční metody pro porovnání s paralelními GPU variantami, které jsou popsány v následující podkapitole.

5.3.2 Implementace paralelních algoritmů s využitím CUDA

Zatímco CPU implementace poskytují funkční řešení, jejich výkon může být nedostatečný pro zpracování velmi detailních map nebo pro scénáře vyžadující rychlou odezvu. Limitem CPU je nejen nižší počet výpočetních jader, ale často i propustnost paměti a inherentně sekvenční

povaha algoritmů jako Dijkstrův algoritmus nebo A*, které v každém kroku závisí na nalezení jednoho globálního minima v prioritní frontě. Pro dosažení výrazného zrychlení byla využita platforma NVIDIA CUDA (viz teoretický základ v podkapitole 4) k implementaci paralelních verzí algoritmů hledání cesty, které dokáží využít masivně paralelní architekturu moderních grafických procesorů (GPU).

GPU nabízí řádově vyšší výpočetní výkon (díky stovkám/tisícům jader) a výrazně vyšší paměťovou propustnost než CPU, což je předurčuje pro úlohy s vysokým stupněm datového paralelizmu. Princip GPGPU spočívá v přenesení relevantních dat (graf, stavové informace) do paměti GPU a spuštění CUDA kernelů – funkcí vykonávaných paralelně mnoha vlákny GPU. Každé vlákno typicky zpracovává malou část úlohy (např. jeden uzel grafu nebo jeho sousedy). Klíčovou výzvou je efektivní návrh kernelů, minimalizace přenosů dat mezi CPU a GPU (které jsou relativně pomalé přes PCIe sběrnici), efektivní využití hierarchie GPU pamětí a správná synchronizace mezi vlákny a kernely.

V rámci této práce byly na GPU implementovány následující algoritmy: Delta-Stepping, A* a varianta HADS (Heuristic-Accelerated Delta-Stepping).

Implementace Delta-Stepping na GPU (`findPathGPU_DeltaStepping`)

Algoritmus Delta-Stepping (teorie v podkapitole 2.1.3) je svou strukturou přirozeně vhodný pro paralelizaci na GPU, a to z několika důvodů:

- **Uvolnění striktního pořadí:** Na rozdíl od Dijkstrova algoritmu/A*, které musí v každém kroku najít a zpracovat globálně nejlepší uzel, Delta-Stepping zpracovává *všechny* vrcholy v rámci jednoho šuplíku (intervalu vzdáleností $[i\Delta, (i+1)\Delta)$) víceméně současně. To umožňuje masivní paralelizmus – každému vrcholu v aktivním šuplíku může být přiřazeno jedno nebo více GPU vláken pro relaxaci jeho hran.
- **Lokalita zpracování:** Relaxace hran vycházejících z vrcholů v aktuálním šuplíku B_i typicky ovlivňuje jen omezený počet dalších šuplíků (B_i a vyšší), což umožňuje rozdělit práci do relativně nezávislých fází.

Parametr Δ zde hraje klíčovou roli v kompromisu: menší Δ se blíží Dijkstrově algoritmu, větší Δ umožňuje více paralelizmu díky tomu, možnost zpracování vrcholů najednou, ale za cenu potenciálně více opakovaných relaxací (label-correcting povaha).

Datové struktury na GPU a komunikace CPU-GPU: Implementace (viz `DeltaSteppingGPU.cpp`) vyžaduje pečlivou správu dat mezi CPU a GPU:

- **Přenos vstupů:** Na začátku jsou na GPU přenesena kompletní data logické mřížky (`d_logical_values`, `d_logical_flags`) a výšková data (`d_elevation_values`) spolu s jejich metadaty (struktury `LogicalGridInfoGPU_fwd`, `ElevationGridInfoGPU_fwd`). Tento přenos je relativně objemný, ale provádí se jen jednou.
- **Stavové informace SSSP na GPU:** Pole pro vzdálenosti (`d_distance`), rodiče (`d_parents`) a příslušnost ke šuplíkům (`d_bucket`, `d_nextBucket`) jsou alokována a inicializována (startovní uzel) v globální paměti GPU. Většina výpočtů pak probíhá přímo na těchto datech na GPU.

- **Pomocné proměnné:** Jednoelementová pole na GPU pro řízení a synchronizaci: `d_changed` (příznak, zda došlo ke změně v relaxační fázi), `d_nextNonEmptyBucket` (pro nalezení dalšího šuplíku), `d_isBucketEmptyFlag`.

Alokace paměti na GPU se provádí pomocí `cudaMalloc` a přenos dat z hostitele pomocí `cudaMemcpy` (nebo `cudaMemcpyAsync` se synchronizací streamu), jak je vidět v implementaci funkce `findPathGPU_DeltaStepping`.

Ilustrace paralelizmu v Delta-Stepping (Konceptuální pseudokód): Abych lépe ilustroval, kde se uplatňuje masivní paralelizmus GPU, představme si zjednodušený pseudokód zaměřený na zpracování jednoho šuplíku B_i. Tento pseudokód zdůrazňuje operace, které mohou probíhat souběžně na mnoha vláknech GPU.

Algoritmus 13 Paralelní zpracování šuplíku v Delta-Stepping

1: **repeat** ▷ Fáze relaxace lehkých hran
2: Resetuj `d_changed` $= 0$
3: **parallel** pro každý uzel u na GPU:
4: **if** $d_bucket[u] = currentBucket$ **a** $d_distance[u] < \infty$ **then**
5: $dist_u = d_distance[u]$; $elev_u =$ getElevationAtDevice(...)
6: **for** všechny sousedy v vrcholu u **do**
7: $cost = calculateToblerEdgeCost(..., elev_u)$
8: **if** $cost \leq \Delta$ **then** ▷ Lehká hrana
9: $new_dist = dist_u + cost$
10: $old_dist =$ **atomicMinFloat**$(\&d_distance[v], new_dist)$
11: **if** $new_dist < old_dist$ **then**
12: Aktualizuj `d_nextBucket[v]`; `d_parents[v] = u`
13: `atomicExch(&d_changed, 1)`
14: **end if**
15: **end if**
16: **end for**
17: **end if**
18: **Synchronizuj**
19: **parallel** pro každý uzel k na GPU:
20: **if** $d_nextBucket[k] \neq -1$ **then**
21: $d_bucket[k] = d_nextBucket[k]$; $d_nextBucket[k] = -1$
22: **end if**
23: **Synchronizuj**
24: Zkopíruj `d_changed` na CPU (`h_changed_flag`)
25: **until** `h_changed_flag` $= 0$
26: **parallel** pro každý uzel u (z B_i) na GPU:// Fáze relaxace těžkých hran
27: **if** u patřil do S **then**
28: **for** všechny sousedy v vrcholu u **do**
29: Spočítej $cost(u, v)$
30: **if** $cost > \Delta$ **then** ▷ Těžká hrana
31: $new_dist = dist_u + cost$

32: old_dist = **atomicMinFloat**($\&d_distance[v], new_dist$)
33: **if** $new_dist < old_dist$ **then**
34: Aktualizuj `d_nextBucket[v]`; `d_parents[v] = u`
35: **end if**
36: **end if**
37: **end for**
38: **end if**
39: **Synchronizuj**
40: **parallel** pro každý uzel k na GPU:
41: **if** $d_nextBucket[k] \neq -1$ **then**
42: $d_bucket[k] = d_nextBucket[k]$; $d_nextBucket[k] = -1$
43: **end if**

Tento pseudokód ilustruje klíčovou výhodu Delta-Stepping pro GPU:

- **Masivní paralelizmus relaxací:** Kroky označené jako `parallel for each` mohou být vykonány tisíci vláken GPU současně. Každé vlákno nezávisle zpracovává jeden nebo více vrcholů u z aktuálního šuplíku a relaxuje jeho hrany. Výpočetně náročná funkce `calculateToblerEdgeCost` se tak provádí paralelně pro mnoho hran najednou.
- **Atomické operace pro konzistenci:** Konflikty při zápisu do sdílených polí (hlavně `d_distance`) jsou řešeny pomocí atomických operací (zde `atomicMinFloat`), které zaručují správnost i při souběžném přístupu.
- **Hrubozrnná synchronizace mezi fázemi:** Synchronizace je potřeba hlavně mezi jednotlivými fázemi (např. po dokončení všech relaxací před aplikací změn šuplíků, nebo mezi fází lehkých a těžkých hran). Algoritmus nevyžaduje jemnozrnnou synchronizaci nebo složitou správu globální fronty jako A*.

Díky této struktuře dokáže Delta-Stepping efektivně využít výpočetní sílu a vysokou paměťovou propustnost GPU pro zpracování velkých grafů, což vede k výraznému zrychlení oproti sekvenčním CPU algoritmům.

Popis CUDA kernelů: Hlavní výpočetní práce je provedena několika CUDA kernely, které jsou volány z hostitelského kódu v rámci řídicí smyčky:

- **`relaxLightEdgesKernel / relaxHeavyEdgesKernel`:** Tyto dva kernely provádějí samotnou relaxaci hran, jak je naznačeno v pseudokódu výše. Jsou spuštěny s konfigurací gridu a bloků pokrývající všechny vrcholy mřížky (*grid-stride loop*). Každé vlákno GPU zodpovídá za podmnožinu vrcholů. Klíčové operace zahrnují kontrolu příslušnosti ke šuplíku, paralelní výpočet `calculateToblerEdgeCost` (včetně volání `getElevationAtDevice`), rozlišení lehkých/těžkých hran a atomickou aktualizaci `d_distance` pomocí `atomicMinFloat`. Při zlepšení cesty se plánuje přesun do `d_nextBucket` a aktualizuje se rodič a příznak `d_changed`.
- **`updateBucketsKernel`:** Tento jednodušší kernel iteruje přes všechny vrcholy a paralelně aplikuje naplánované přesuny z `d_nextBucket` do `d_bucket`.

- **`isBucketEmptyKernel / findNextBucketKernel`:** Slouží k řízení hlavní smyčky na CPU. Paralelně (s využitím atomických operací nebo paralelní redukce se sdílenou pamětí) zjišťují stav šuplíků a umožňují hostiteli efektivně přejít na další relevantní krok.

Hostitelská řídicí smyčka: Funkce `findPathGPU_DeltaStepping` na CPU spravuje celý proces. Její hlavní smyčka (viz ukázka v `lst:deltastepping_host_loop`) řídí střídání fází relaxace lehkých a těžkých hran, volá příslušné kernely, zajišťuje potřebnou synchronizaci (`cudaStreamSynchronize` nebo `cudaDeviceSynchronize`) a kontroluje příznaky (`h_changed_flag`) pro řízení vnitřní smyčky relaxace lehkých hran. Komunikace mezi CPU a GPU je během hlavní smyčky minimalizována na přenos malých řídicích informací.

Výpis 6: Hlavní smyčka Delta-Stepping na CPU (zjednodušeno)

```
while (currentBucket < MAX_BUCKET_SAFETY) {
    // Najít další neprázdný šuplík (pomocí isBucketEmptyKernel
    // a findNextBucketKernel)
    // ... (kód pro nalezení/aktualizaci currentBucket) ...
    if (/* žádný další šuplík */) break;

    // Fáze lehkých hran (opakuje se, dokud jsou změny)
    do {
        h_changed_flag = 0; // Reset flag
        CUDA_CHECK(cudaMemcpyAsync(d_changed, &h_changed_flag, ...));
        // Reset na GPU
        CUDA_CHECK(launch_relaxLightEdges(...)); // Spustit kernel
        CUDA_CHECK(launch_updateBuckets(...));  // Aplikovat změny
        CUDA_CHECK(cudaStreamSynchronize(stream)); // Počkat na dokončení
        CUDA_CHECK(cudaMemcpy(&h_changed_flag, d_changed, ...));
        // Zkontrolovat flag
    } while (h_changed_flag);

    // Fáze těžkých hran (spustí se jednou)
    CUDA_CHECK(launch_relaxHeavyEdges(...));
    CUDA_CHECK(launch_updateBuckets(...));
    CUDA_CHECK(cudaStreamSynchronize(stream));

    // Přechod na další šuplík (aktualizováno na začátku další iterace)
    // Poznámka: Původní kód zde měl currentBucket++, ale korektnější je
    // hledat další neprázdný šuplík až na začátku další iterace while.
}
// Zkopírovat výsledky (parents) zpět a~rekonstruovat cestu...
```

Výkonnostní aspekty: Efektivita Delta-Stepping na GPU závisí na správné volbě parametru Δ a `threshold_arg`, výkonu pamětí, režii atomických operací a náročnosti výpočtu ceny hrany.

Díky své struktuře je však obecně považován za jeden z nejefektivnějších paralelních SSSP algoritmů pro mnoho typů grafů.

Implementace A* a HADS na GPU

Kromě Delta-Stepping, který řeší obecný SSSP problém, byly implementovány i algoritmy využívající heuristiku pro rychlejší nalezení cesty ke konkrétnímu cíli.

A* na GPU (`findPathGPU_AStar`): Přímá a efektivní implementace A* na GPU naráží na zásadní problém: paralelní správa globální prioritní fronty (open setu). Operace jako vložení, nalezení minima a extrakce minima jsou v sekvenčním A* klíčové, ale jejich efektivní paralelizace pro tisíce vláken je velmi obtížná. Existují různé přístupy (paralelní heapy, distribuované fronty), ale bývají komplexní.

Implementace (`AStarGPU.cpp`, `AStarKernels.cu`) proto volí zjednodušený přístup, který obchází explicitní paralelní prioritní frontu:

- **Reprezentace stavu:** Používá pole struktur `PathNodeGPU` obsahující g, f, rodiče a stav (`visited`: 0=unvisited, 1=open, 2=closed). Množina „open“ je tedy implicitní (všechny vrcholy se stavem 1).
- **Iterativní výběr a expanze jednoho vrcholu:** Hlavní smyčka na CPU v každé iteraci:
  1. **Najde globálně nejlepší otevřený uzel:** Pomocí dvoufázové paralelní redukce na GPU (kernely `findLocalBestAStarNodes` a část `findGlobalAndExpandAStar`) se najde index vrcholu s minimálním f-skóre mezi všemi vrcholy se stavem 1.
  2. **Expanduje sousedy tohoto vrcholu:** Kernel `findGlobalAndExpandAStar` (jeho druhá část) pak paralelně (typicky jen 8 vlákny) expanduje sousedy nalezeného nejlepšího vrcholu, provede výpočet ceny kroku (`calculateToblerEdgeCost`) a heuristiky (`calculate_heuristic_gpu`), a aktualizuje informace sousedů (g, f, rodič, stav=1) pomocí `atomicMinFloat` pro g-skóre.
- **Ukončení:** Smyčka končí, pokud je nalezen cíl (nastaven příznak `d_path_found_flag`) nebo pokud nejsou nalezeny žádné další otevřené vrcholy (`d_no_open_nodes_flag`).

Tento přístup sice využívá GPU pro redukci a expanzi sousedů, ale expanduje pouze jeden uzel za iteraci, což je jeho hlavní limitace. Míra využití GPU může být nízká, pokud je expanze jednoho vrcholu rychlá a většina času se stráví čekáním na dokončení kernelů a komunikací s CPU pro řízení iterací. Výhodou je relativní jednoduchost implementace oproti plně paralelní prioritní frontě.

HADS na GPU (`findPathGPU_HADS`): HADS (Heuristic-Accelerated Delta-Stepping) vznikl jako pokus zkombinovat výhody obou přístupů: zachovat masivní paralelizmus Delta-Stepping při zpracování šuplíků a zároveň využít heuristiku pro zaměření prohledávání směrem k cíli, jak to dělá A*. Cílem je redukovat celkový počet zpracovaných vrcholů a šuplíků oproti čistému Delta-Stepping, a tím zrychlit nalezení cesty ke konkrétnímu cíli, při zachování vyššího stupně paralelizmu než u implementovaného A* na GPU.

- **Princip:** Základní struktura a řídicí smyčka na hostiteli zůstává stejná jako u algoritmu Delta-Stepping.
- **Heuristické prořezávání (Pruning):** Klíčový rozdíl je v relaxačních kernelech

(`relaxLightEdgesHADS_Kernel`, `relaxHeavyEdgesHADS_Kernel`). Předtím, než vlákno relaxuje hranu (u, v), porovná heuristickou hodnotu aktuálního vrcholu h_u s heuristickou hodnotou souseda h_v. Pokud $h_v > h_u \times$ PRUNE_FACTOR, kde PRUNE_FACTOR ≥ 1.0 je ladicí parametr, relaxace se neuskuteční. Tím se efektivně „prořezávají“ větve prohledávání, které vedou výrazně „špatným“ směrem od cíle.

- **Výpočet/Použití Heuristiky:** Heuristika se počítá na GPU pomocí `calculate_heuristic_gpu`. Implementace navíc umožňuje volitelný předvýpočet heuristiky pro vrcholy v určitém radiu kolem cíle (`precompute_local_heuristic_kernel`) a uložení do pole `d_heuristic`. Relaxační kernely pak primárně používají tuto předpočítanou hodnotu (pokud je platná), čímž šetří výpočetní čas v hlavní smyčce. Pokud předpočítaná hodnota není dostupná, dopočítá se heuristika za běhu.

Výpis 7: Heuristické prořezávání v HADS kernelu (koncept)

```
// Uvnitř relaxLightEdgesHADS_Kernel / relaxHeavyEdgesHADS_Kernel:
for (int dir = 0; dir < 8; ++dir) {
    // ... (výpočet nx, ny, neighborIdx) ...
    float h_u = calculate_heuristic_gpu(x, y, goal_x, goal_y, W);
    // Heuristika aktuálního
u~float h_v = d_heuristic[neighborIdx];
    // Zkus předpočítanou
    if (h_v < 0.0f) { // Pokud nebyla (-1.0f značí \uv{nepočítáno})
        h_v = calculate_heuristic_gpu(nx, ny, goal_x, goal_y, W);
        // Dopočítej
    }

    // *** HEURISTICKÉ PROŘEZÁVÁNÍ ***
    if (h_v > h_u * PRUNE_FACTOR) {
    // Pokud soused směřuje příliš od cíle
        continue; // Přeskoč tento směr, neprováděj relaxaci
    }

    // ... (pokračuj výpočtem váhy hrany a~relaxací jako v~Delta-Stepping)
}
```

Očekává se, že HADS bude podobně rychlý jako čistý Delta-Stepping pro hledání cesty ke konkrétnímu cíli, protože prozkoumá méně grafu. Zároveň by měl být schopný lépe využít paralelizmus GPU než implementovaný A* díky zpracování celých šuplíků najednou. Úspěch HADS však silně závisí na správném naladění parametrů Δ, `threshold_arg` a `PRUNE_FACTOR`. Příliš agresivní prořezávání (`PRUNE_FACTOR` blízké 1.0) může vést k nenalezení optimální cesty, zatímco příliš volné prořezávání snižuje jeho výhodu oproti Delta-Stepping.

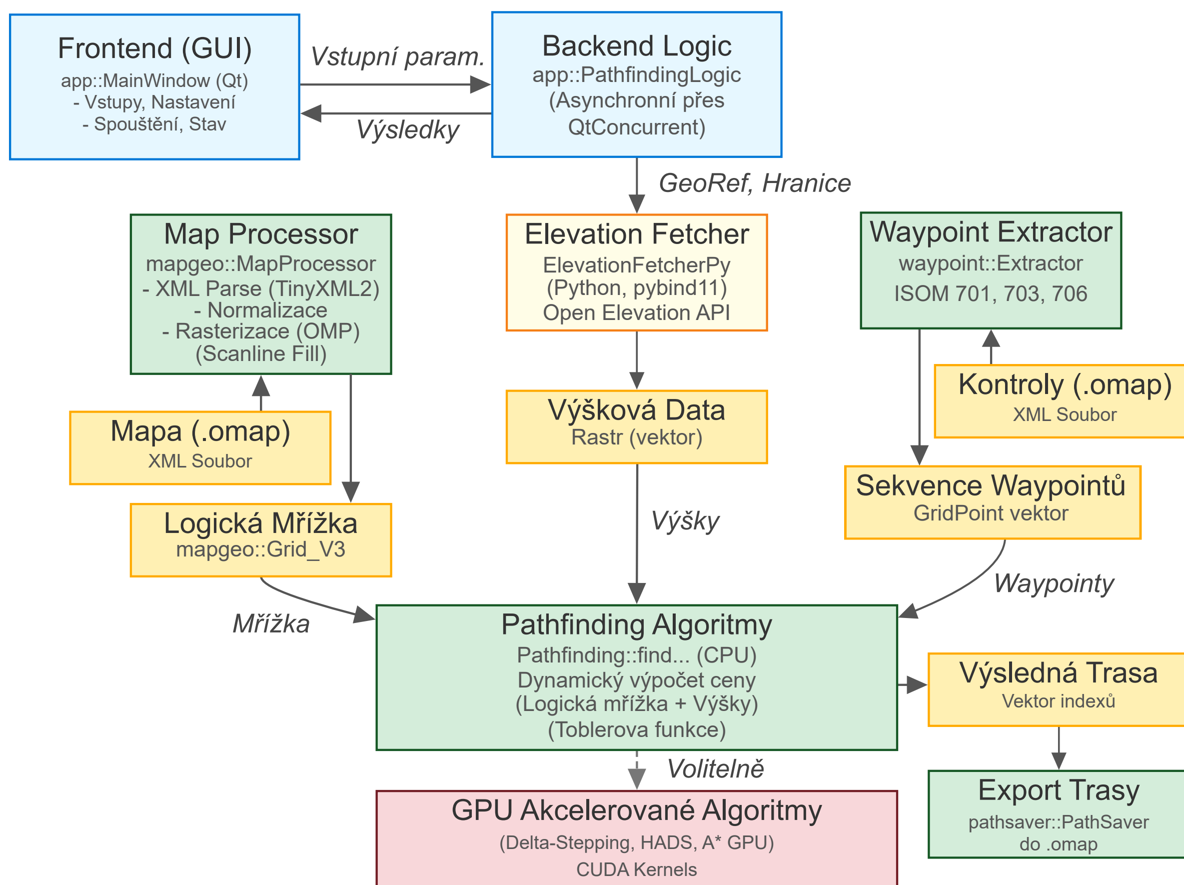
Výběr nejvhodnějšího GPU algoritmu a jejich parametrů je předmětem experimentálního vyhodnocení v další části práce.

6 Výsledná aplikace: OMAP Pathfinding Processor

Pro praktické využití implementovaných algoritmů a snadné experimentování byla vyvinuta desktopová aplikace s grafickým uživatelským rozhraním (GUI). Aplikace, pojmenovaná *OMAP Pathfinding Processor*, integruje všechny fáze zpracování – od načtení mapových a výškových dat (jak bylo popsáno v podkapitole 5.1), přes jejich konverzi na grafovou reprezentaci (podkapitola 5.2) až po spuštění vybraného pathfinding algoritmu (implementace viz podkapitola 5.3) a export výsledné trasy. Byla postavena s využitím C++ a multiplatformního frameworku Qt, což zajišťuje její funkčnost napříč různými operačními systémy. Tato kapitola popisuje architekturu, uživatelské rozhraní a pracovní postup této aplikace.

6.1 Architektura a technologie

Aplikace je napsána v jazyce C++ s využitím frameworku Qt (verze 6.3) pro tvorbu grafického uživatelského rozhraní a správu aplikační logiky (signály a sloty, event loop, správa oken a widgetů). Tato volba umožňuje relativně snadný vývoj GUI a zajišťuje přenositelnost aplikace.



Obr. 4: Schéma aplikace a jejího vnitřního propojení

Logika aplikace je rozdělena do dvou hlavních částí viz. 4:

- **Frontend (GUI):** Implementováno pomocí standardních Qt widgetů organizovaných ve třídě `app::MainWindow`. Zodpovídá za interakci s uživatelem, sběr vstupních parametrů prostřednictvím formulářů a ovládacích prvků, zobrazování stavu a výsledků (aktuálně textově ve stavovém řádku, s plánovanou grafickou vizualizací) a spouštění výpočetních úloh. Pro perzistenci nastavení využívá třídu `QSettings`. Vzhled aplikace lze přepínat mezi světlým a tmavým tématem pomocí dynamicky aplikovaných Qt Stylesheets. Oblast pro vizualizaci v hlavním okně je aktuálně placeholder; detailní vizualizace je realizována externě (viz popis níže).
- **Backend (Logika):** Zapouzdřuje komplexní výpočetní logiku v rámci třídy `app::PathfindingLogic`. Tato třída implementuje:
  - **Zpracování mapy:** Využívá třídu `mapgeo::MapProcessor` k parsování XML/ `.omap` souborů (s pomocí knihovny `TinyXML2`, viz popis formátu v podkapitole 3.2.2) a generování logické mřížky (`mapgeo::Grid_V3`) pomocí popsaného dvoufázového rasterizačního procesu (paralelní hranice v Pass 1 s OpenMP, Scanline Fill (podkapitola 5.2) a aplikace pravidel v Pass 2). Zahrnuje normalizaci souřadnic a výpočet rozlišení mřížky.
  - **Získání výškových dat:** Volá externí Python skripty prostřednictvím třídy `ElevationFetcherPy` (využívající knihovnu `pybind11` pro propojení C++ a Pythonu). Jeden skript (`elevation_logic.py`) zajišťuje stažení a zpracování DMT dat pro danou oblast mapy (na základě georeferencování a hranic mapy získaných pomocí `GeoRefScanner`) a vrací rastr výšek. Další skript je použit pro převod případných referenčních bodů z Lat/Lon na projektované souřadnice (např. S-JTSK).
  - **Extrakce bodů:** Používá `waypoint::WaypointExtractor` k nalezení startu (symbol 701), cíle (706) a případných kontrol (symbol 703) v souboru s kontrolami a převádí jejich souřadnice na indexy v logické mřížce. Reference na standardní mapové značky viz kapitola 3.2.
  - **Hledání cesty:** Spouští vybraný pathfinding algoritmus (implementovaný jako funkce v namespace `Pathfinding`, viz implementace v kapitole 5.3.1 a 5.3.2) postupně pro každý úsek mezi body (Start $\rightarrow$ K1 $\rightarrow$ K2 $\rightarrow \cdots \rightarrow$ Cíl).
  - **Export trasy:** Využívá `pathsaver::PathSaver` k uložení výsledné kompletní trasy do nového `.omap` souboru.

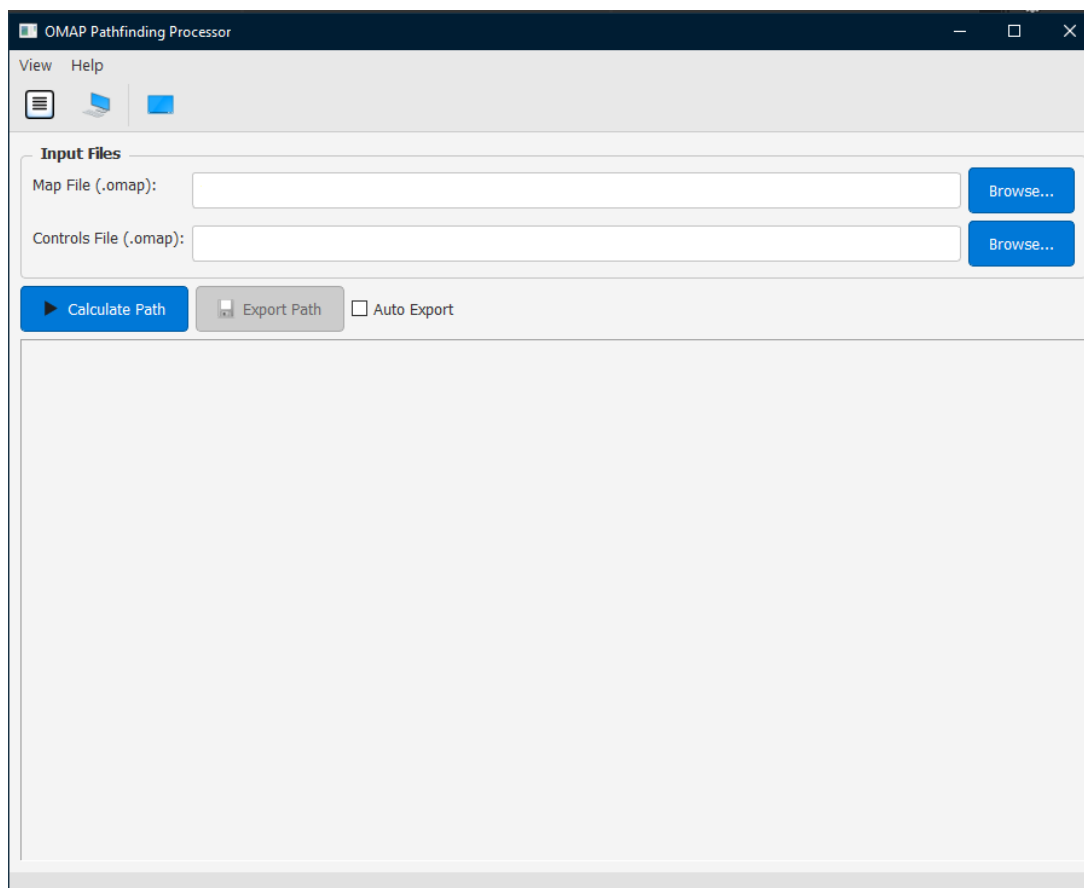
Pro zajištění responzivity GUI a využití dostupného hardwaru aplikace implementuje:

- **Asynchronní zpracování:** Celý proces v pozadí (`PathfindingLogic::processAndFindPath`) je spouštěn asynchronně pomocí `QtConcurrent::run`, aby neblokoval GUI. `QFutureWatcher` sleduje dokončení a signalizuje výsledek.
- **CPU Paralelizmus (OpenMP):** První fáze rasterizace v `MapProcessor` je paralelizována pomocí direktiv OpenMP. Počet vláken je nastavitelný v GUI.

- **NVIDIA CUDA:** Jak bylo detailně popsáno v teoretické části (kapitola 4) a v popisu implementace (podkapitola 5.3.2), platforma CUDA je využita pro akceleraci paralelních pathfinding algoritmů (Delta-Stepping, A*, HADS) na kompatibilních GPU.

6.2 Uživatelské rozhraní

Hlavní okno aplikace (viz Obrázek 5) je navrženo pro přehledné zadávání vstupů a konfiguraci algoritmů.



Obr. 5: Hlavní okno po spuštění aplikace OMAP Pathfinding Processor.

6.2.1 Vstupní soubory a hlavní akce

Sekce **„Input Files“** umožňuje výběr mapového a kontrolního `.omap` souboru pomocí tlačítek „Browse...“. Tlačítko **„Calculate Path“** spouští hlavní výpočetní proces. Tlačítko **„Export Path“** (aktivní po úspěšném výpočtu) a volba **„Auto Export“** řídí uložení výsledné trasy.

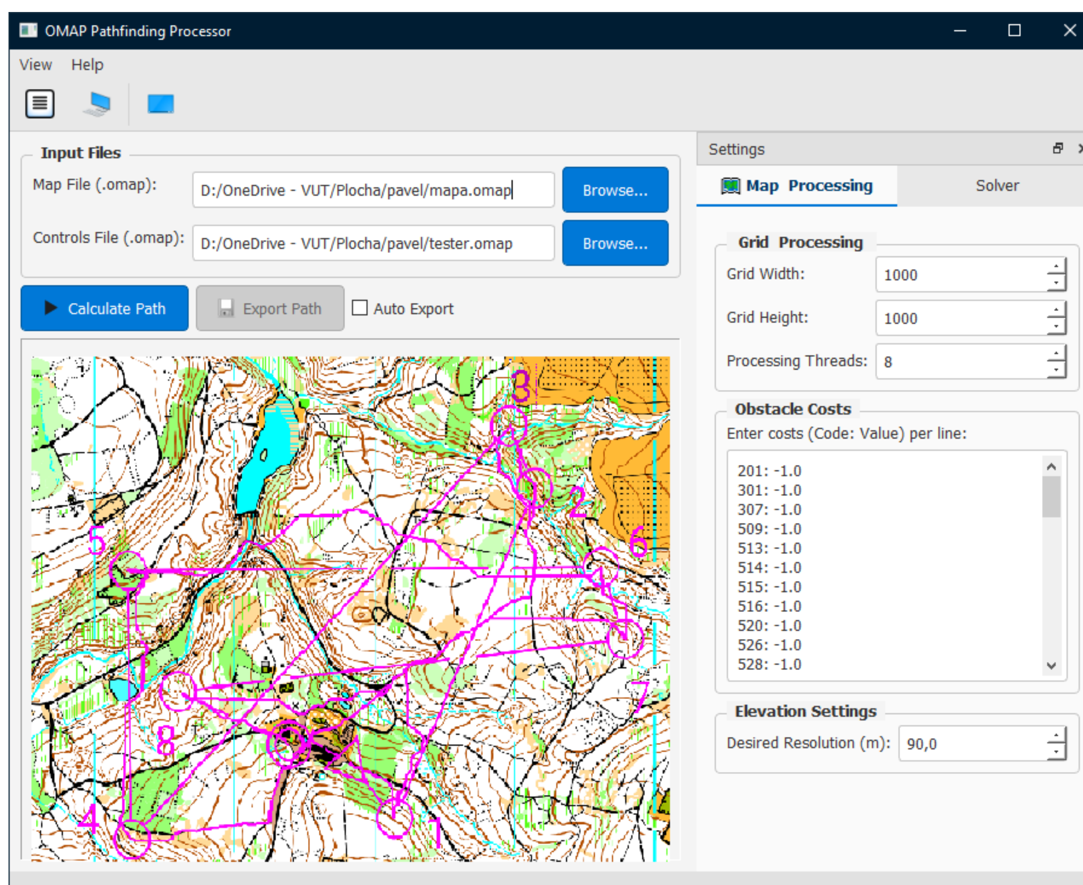
6.2.2 Oblast pro vizualizaci

Oblast „Map Area“ je určena pro zobrazení mapy a tras přímo v aplikaci. Aktuální vizualizace výsledků probíhá externě pomocí Python skriptu, který načte data a vygeneruje zjednodušený pohled na mapu a nalezenou trasu (podobně jako na Obrázku 8).

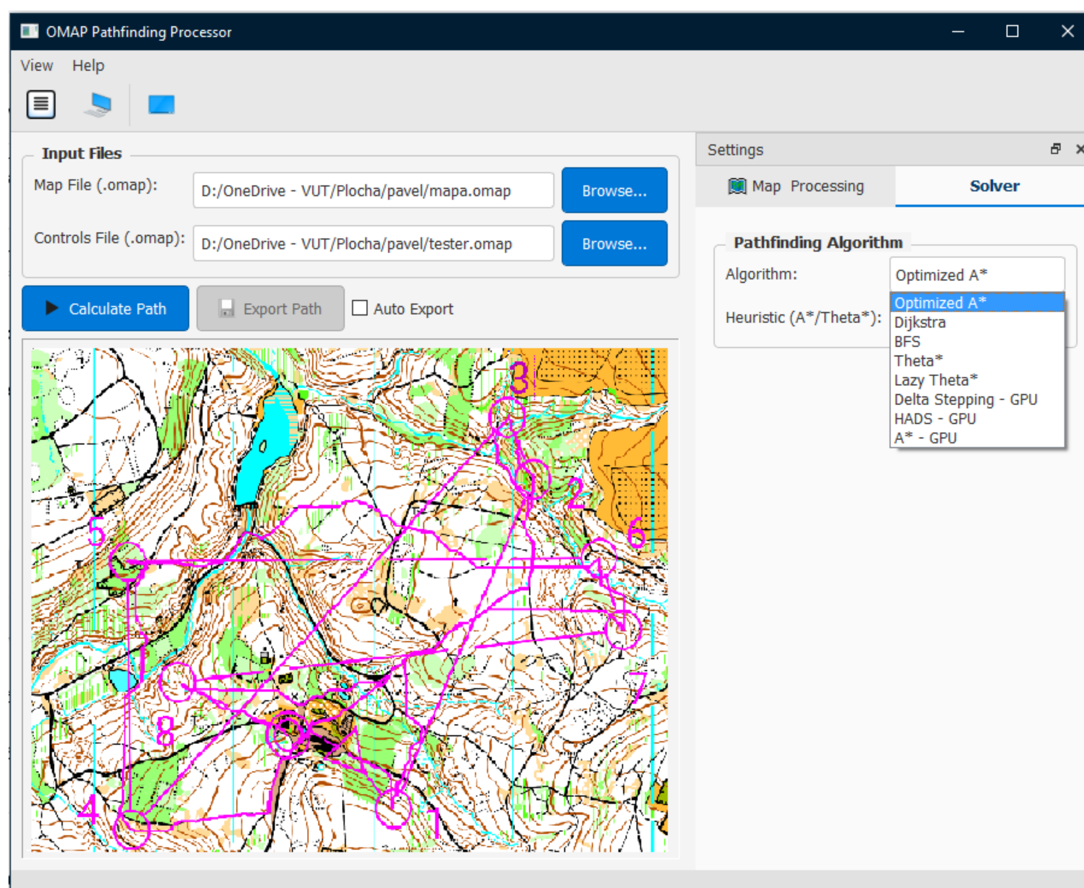
6.2.3 Panel nastavení

Dokovatelný panel **„Settings“** (Obrázky 6, 7) obsahuje dvě záložky:

- **Map & Processing:** Nastavení rozměrů mřížky, počtu OpenMP vláken, definice cen překážek pro ISOM symboly (viz teoretický popis značek v podkapitole 3.2) a požadované rozlišení výškových dat. Obsahuje také skupinu **GPU Parameters** pro ladění parametrů paralelních algoritmů (např. Δ pro Delta-Stepping popsaný v podkapitole 2.1.3).
- **Solver:** Výběr pathfinding algoritmu (A*, Dijkstrova algoritmu, BFS, Theta*, Lazy Theta*, Delta Stepping - GPU, HADS - GPU, A* - GPU – viz kapitola 2.1 a podkapitola 5.3) a výběr heuristiky (pro A*, Theta*, Lazy Theta* – viz popis heuristik v podkapitole 2.1.2).



Obr. 6: Panel nastavení - záložka „Map & Processing“.



Obr. 7: Panel nastavení - výběr algoritmu.

6.2.4 Menu, nástrojová lišta a stavový řádek

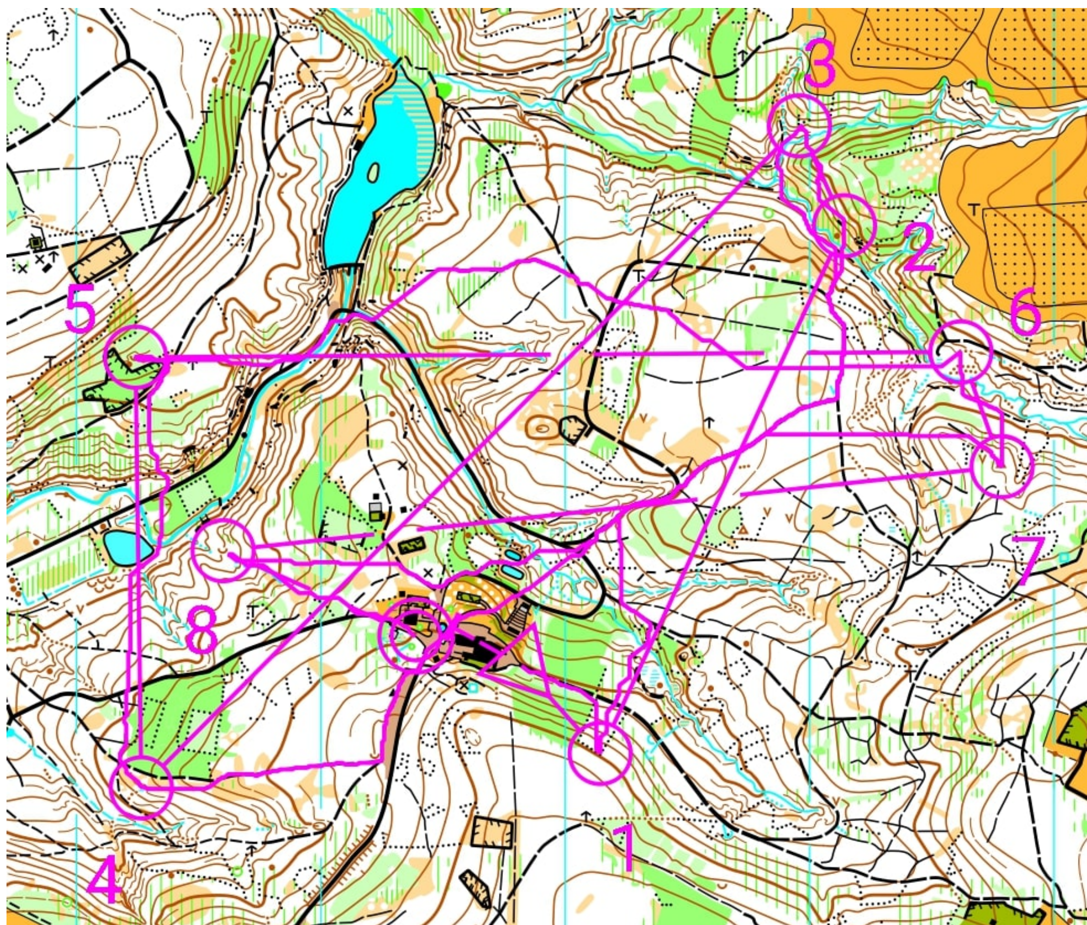
Standardní ovládací prvky pro přístup k nastavením, nápovědě a přepínání vzhledu. Stavový řádek informuje o průběhu a výsledcích výpočtu.

6.3 Pracovní postup (Workflow)

Typické použití aplikace pro nalezení optimální trasy probíhá v následujících krocích:

1. **Načtení souborů:** Uživatel pomocí tlačítek „Browse...“ vybere mapový `.omap` soubor (formát viz 3.2.2) a soubor s kontrolami.
2. **Konfigurace zpracování mapy:** V panelu nastavení (záložka „Map & Processing“) uživatel zkontroluje nebo upraví rozměry cílové mřížky, počet vláken, definice cen překážek (odvozené od ISOM symbolů viz 3.2) a případně požadované rozlišení výškových dat.
3. **Konfigurace hledání cesty:** V záložce „Solver“ uživatel vybere požadovaný pathfinding algoritmus a jeho parametry (heuristiku, Δ, atd., jak bylo popsáno v sekcích 5.3.1 a 5.3.2).
4. **Spuštění výpočtu:** Uživatel stiskne tlačítko „Calculate Path“. Aplikace na pozadí (asynchronně) provede kroky popsané v podkapitole 6.1 (zpracování mapy, získání výšek, extrakce bodů, spuštění algoritmu pro segmenty trasy).

5. **Export výsledků:** Po úspěšném dokončení je uživatel (pokud není aktivní „Auto Export") vyzván k uložení výsledku. Aplikace vygeneruje nový `.omap` soubor, který je kopií původního kontrolního souboru, do kterého je přidán nový objekt (typu linie, s výrazným symbolem) reprezentující nalezenou optimální trasu (viz příklad na Obrázku 8).



Obr. 8: Příklad mapy ve formátu `.omap` s vykreslenou trasou exportovanou z aplikace (fialová čára).

Aplikace *OMAP Pathfinding Processor* tak představuje komplexní nástroj pro analýzu tras v orientačním běhu, umožňující porovnání různých algoritmů a jejich parametrů na reálných datech. Výsledky experimentů získané pomocí této aplikace jsou prezentovány v následující kapitole (7).

7 Experimenty a výsledky

Tato kapitola prezentuje výsledky experimentálního ověření implementovaných algoritmů. Cílem bylo zhodnotit jejich výkonnost a charakteristiky nalezených tras na syntetických datech reprezentujících různé typy terénu typické pro orientační běh.

7.1 Experimentální prostředí

Všechny experimenty a měření výkonnosti byly provedeny na následující hardwarové a softwarové konfiguraci.

Hardwarová konfigurace

Testovací sestava byla tvořena osobním počítačem s následujícími specifikacemi:

- **Procesor (CPU):** Intel Core i7-9700K @ 3.60GHz (8 jader / 8 vláken).
- **Grafický procesor (GPU):** NVIDIA GeForce GTX 1080 Ti (11 GB GDDR5X).
- **Operační paměť (RAM):** 32 GB DDR4.
- **Úložiště:** SSD NVMe Samsung 960 EVO.

Softwarová konfigurace

Pro vývoj a testování aplikace byly použity následující softwarové nástroje a knihovny:

- **Operační systém:** Microsoft Windows 10 Pro N (verze 10.0.19045).
- **Vývojové prostředí a kompilátor:** Microsoft Visual Studio 2022, kompilátor MSVC.
- **Qt Framework:** Verze 6.3.
- **NVIDIA CUDA Toolkit:** Verze 12.5.
- **Ovladač grafické karty NVIDIA:** Verze 572.47.
- **Knihovna TinyXML2:** Pro parsování XML.
- **OpenMP:** Podpora zajištěna kompilátorem MSVC.
- **Python a klíčové knihovny:** Python 3.9, `pybind11`, `requests`, `pyproj`.

7.2 Testovací data a metodika

Pro objektivní posouzení výkonnosti a vlastností implementovaných algoritmů bylo nutné definovat konzistentní experimentální prostředí, sadu reprezentativních testovacích dat a jasnou metodiku pro sběr a vyhodnocení výsledků. Tato sekce detailně popisuje tyto aspekty. Cílem bylo provést syntetické porovnání algoritmů na mapách s odlišnými charakteristikami, aby bylo možné lépe pochopit jejich chování v různých podmínkách a identifikovat jejich silné a slabé stránky.

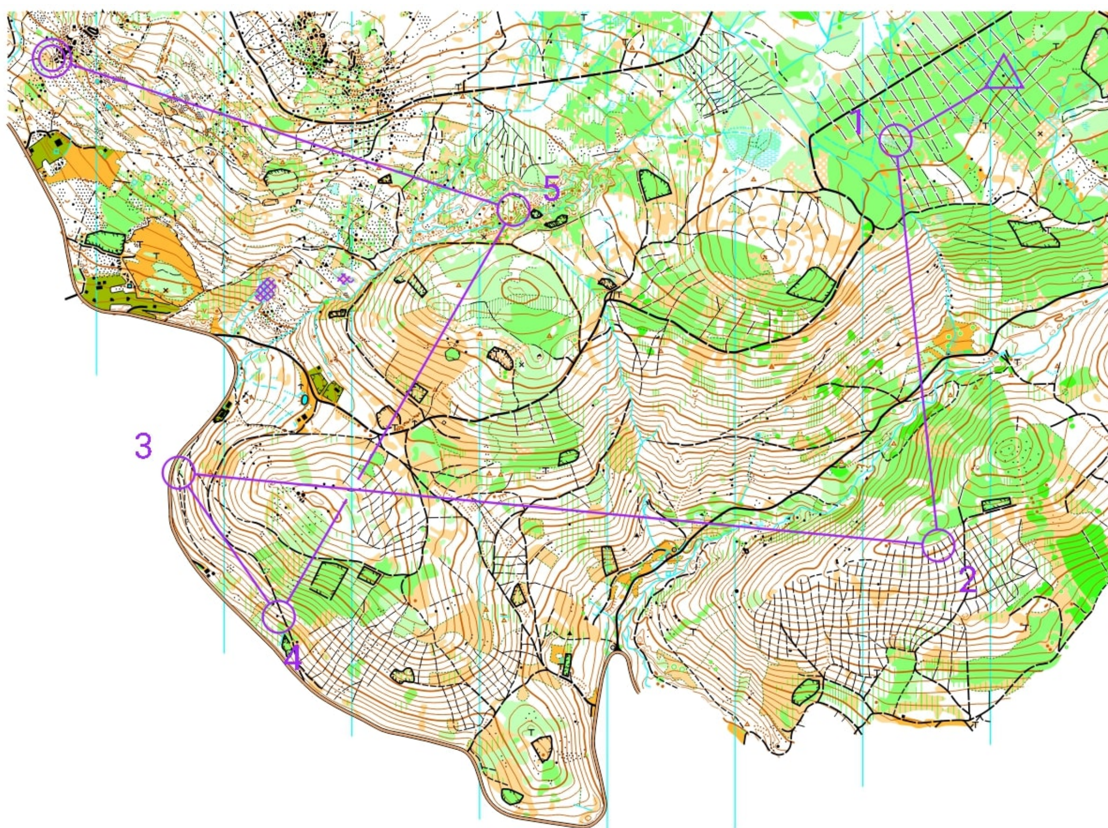
7.2.1 Testovací mapy a scénáře

Pro syntetické porovnání algoritmů byly použity tři reálné mapy pro orientační běh, reprezentující různé charakteristiky terénu a výpočetní náročnost.

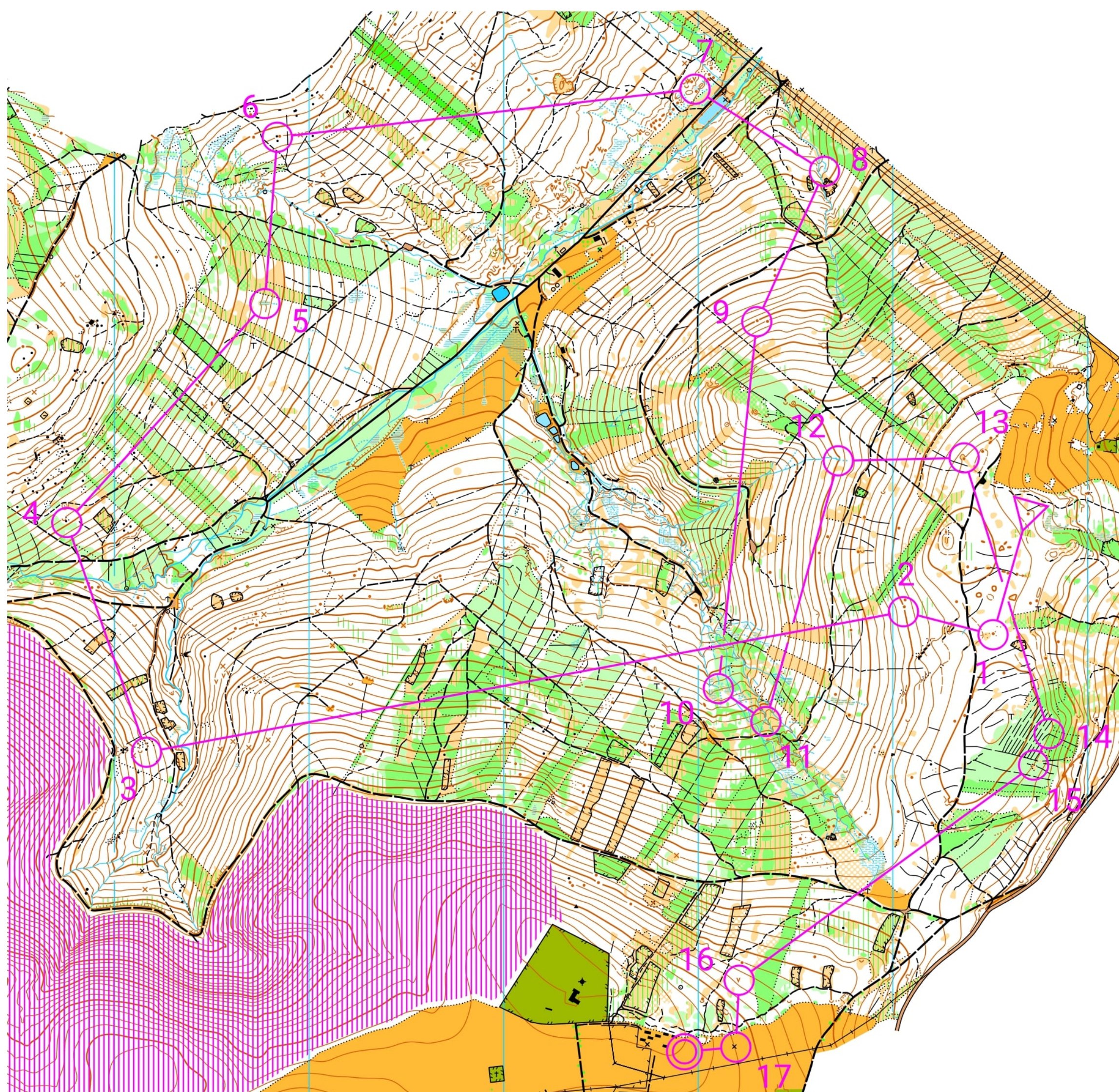
1. **Mapa „Les Včelný“:**
    - **Lokalita a původ:** Reálná mapa lesa Včelný (Rychnov nad Kněžnou, východní Čechy).
    - **Charakteristika:** Členitý lesní terén (viz Obrázek 9) s hustými vrstevnicemi, variabilní vegetací a sítí komunikací.
    - **Zpracování:** Logická mřížka 1000x1000 buněk, rozlišení cca 2.27 m/buňka.
    - **Komplexita dat:** 4500 mapových objektů, velikost souboru 2 MB.
    - **Testovaný scénář:** Nalezení cesty mezi 10 body (S+8K+C) z `tester.omap`, délka trasy 6 km.
2. **Mapa „Oceán“:**
    - **Lokalita a původ:** Jižní část mapy „Oceán“ (závody ČP/ŽA u Nejdku, západní Čechy).
    - **Charakteristika:** Kopcovitý terén (viz Obrázek 10) se střední hustotou cest, oplocenkami, hustníky a kamenitými pasážemi.
    - **Zpracování:** Logická mřížka 1000x1000 buněk, rozlišení cca 2.27 m/buňka.
    - **Komplexita dat:** 7500 mapových objektů (pro celou mapu), velikost souboru odpovídající.
    - **Testovaný scénář:** Nalezení cesty mezi 7 body (S+5K+C) z `tester.omap`, délka trasy 6.3 km.
3. **Mapa „Orlí vrch“:**
    - **Lokalita a původ:** Reálná mapa ze závodu Mistrovství ČR na klasické trati 2013.
    - **Charakteristika:** Terén horského údolí (viz Obrázek 11) s relativně hustou sítí cest v JZ části. Kombinuje náročná stoupání s traverzy a během po hřebeni. Mapa obsahuje značné množství různých typů porostů.
    - **Zpracování:** Logická mřížka 3000x3000 buněk, rozlišení cca 1.58 m/buňka.
    - **Komplexita dat:** 15200 mapových objektů, velikost souboru 8.3 MB.
    - **Testovaný scénář:** Nalezení cesty pro trať s 19 body (Start + 17 kontrol + Cíl), odpovídající reálnému závodu o délce 10.4 km (modelovaná délka 12-13 km).

Pro všechny mapy byla použita výšková data získaná z API (`api.open-elevation.com`, 90 m rozlišení). Definice cen překážek pro ISOM kódy odpovídala standardní konfiguraci v aplikaci.

Obr. 9: Náhled testovací mapy „Les Včelný“.



Obr. 10: Náhled testovací mapy „Oceán“.



Obr. 11: Náhled testovací mapy „Orlí vrch“.

7.2.2 Metodika měření výkonnosti a kvality

- **Měření času:** Průměr a směrodatná odchylka z 5 běhů kompletního hledání cesty.
- **Metriky kvality:** Průměrný počet buněk, geometrická délka trasy (m), úspěšnost (počet úspěšných běhů / 5).
- **Parametry algoritmů:**
  - CPU A*/Lazy Theta*: Heuristika „Min Cost“. CPU Theta* nebyl pro mapu „Orlí vrch“ testován kvůli vysoké výpočetní náročnosti.
  - GPU Delta-Stepping a HADS: Použity parametry z automatické ladící fáze. Pro mapy Včelný/Oceán ($\Delta \approx 113.5/227.1$, $T = 10.0$, HADS $R = 100, P = 1.0, W = 0.95$). Pro Orlí vrch ($\Delta \approx 78.9$, $T = 2.0$, HADS $R = 200, P = 1.0, W = 0.95$).
  - GPU A*: Testován s agresivní heuristickou váhou $W = 1.0$ (pro Orlí vrch) a $W = 0.95$ (pro Včelný, kde byl také testován s konzervativnější heuristikou).

7.3 Výsledky syntetických benchmarků

Následující tabulky a analýza shrnují výsledky porovnání implementovaných algoritmů na testovacích mapách.

7.3.1 Výsledky pro mapu „Les Včelný“

Tabulka 1 reprezentuje mapu s rozlišením 1000x1000 a 10 body reprezentující kontroly.

Tab. 1: Výsledky porovnání algoritmů na mapě „Les Včelný“

Algoritmus	**Avg Time (ms)**	**Std Dev (ms)**	**Avg Length (m)**	**Avg Cells**
CPU A*	207.82	1.09	6051.77	2315
CPU Theta*	3638.12	275.22	6040.16	2319
CPU Lazy Theta*	1400.61	52.92	6040.16	2319
GPU Delta-Stepping	1216.75	27.26	6051.77	2315
GPU HADS	872.30	34.77	5861.32	2268
GPU A* (Konzerv.)†	86695.06†	4331.91	6051.77	2315
GPU A* (Agresiv.)*	472.64	38.91	5378.26	2076

†GPU A* s konzervativní heuristikou. *GPU A* s agresivní heuristikou.

7.3.2 Výsledky pro mapu „Oceán“

Tabulka 2 reprezentuje mapu s rozlišením 1000x1000 a 7 body reprezentující kontroly.

Tab. 2: Výsledky porovnání algoritmů na mapě „Oceán“

Algoritmus	**Avg Time (ms)**	**Std Dev (ms)**	**Avg Length (m)**	**Avg Cells**
CPU A*	383.07	1.63	6382.75	2425
CPU Theta*	4735.44	104.66	6764.64	2579
CPU Lazy Theta*	2040.43	12.83	6764.64	2579
GPU Delta-Stepping	920.11	77.71	6382.75	2425
GPU HADS	801.51	10.24	6365.67	2415
GPU A* (Konzerv.)	164202.16	5835.53	6385.96	2426

7.3.3 Výsledky pro mapu „Orlí vrch“

Tabulka 3 reprezentuje mapu s rozlišením 3000x3000 a 19 body reprezentující kontroly.

Tab. 3: Výsledky porovnání algoritmů na mapě „Orlí vrch“

Algoritmus	Avg Time (ms)	Std Dev (ms)	Avg Length (m)	Avg Cells
CPU A*	1316.70	106.66	12113.64	6686
CPU Lazy Theta*	8060.48	73.35	13103.14	6801
GPU Delta-Stepping	7566.27	65.85	12113.64	6686
GPU HADS	6881.68	27.69	11937.41	6634
GPU A* (Agresiv.)	5712.94	57.93	11364.06	6318

Poznámka: CPU Theta* nebyl pro tuto mapu testován kvůli vysoké výpočetní náročnosti.

7.4 Diskuse výsledků syntetických testů

Výsledky benchmarků na všech třech mapách potvrzují některé trendy a odhalují zajímavé aspekty chování implementovaných algoritmů.

Výkon CPU algoritmů a škálovatelnost: CPU A* se konzistentně ukazuje jako nejrychlejší sekvenční algoritmus na všech testovaných mapách. Jeho čas roste s velikostí problému (1000x1000 vs 3000x3000 a delší trasa), ale stále si udržuje prvenství mezi CPU metodami. Any-Angle algoritmy Lazy Theta* a (kde byl testován) Theta* jsou výrazně pomalejší. Na největší mapě „Orlí vrch“ byl CPU Theta* dokonce vynechán kvůli očekávané extrémní době běhu. Lazy Theta* je i na této velké mapě 6x pomalejší než CPU A*. Ukazuje se, že výpočetní režie LoS testů a zejména `calculateSegmentCost` (který iteruje přes mnoho buněk a opakovaně volá `ElevationSampler`) je pro tyto algoritmy na CPU velmi vysoká a špatně škáluje s délkou LoS segmentů a komplexitou terénu pod nimi.

Výkon GPU algoritmů a přínos paralelizace:

- **Delta-Stepping a HADS:** Na menších mapách (Včelný, Oceán) byly tyto GPU algoritmy rychlejší než CPU Any-Angle, ale pomalejší než CPU A*. Na velké mapě „Orlí vrch“ se však situace mění: GPU HADS (6.9 s) a GPU Delta-Stepping (7.6 s) jsou již rychlejší než CPU Lazy Theta* (8.1 s) a výrazně rychlejší, než by byl CPU Theta*. Stále však nedosahují rychlosti CPU A* (1.3 s). To naznačuje, že ačkoliv režie GPU je stále přítomna, s rostoucí velikostí problému (9x více buněk u Orlího vrchu) se začíná více projevovat přínos masivního paralelizmu při zpracování velkého počtu vrcholů v šuplících. HADS je konzistentně rychlejší než Delta-Stepping, což potvrzuje efektivitu heuristického prořezávání i na větších grafech.
- **Potenciál pro škálování GPU:** Výsledky naznačují, že pro ještě větší grafy nebo na výkonnějším GPU by Delta-Stepping a HADS mohly CPU A* překonat. Jejich schopnost zpracovávat mnoho vrcholů souběžně je klíčová pro škálovatelnost, zatímco CPU A* je limitován svým sekvenčním single-core charakterem.

- **GPU A*:** Na menších mapách s konzervativní heuristikou byl nepoužitelně pomalý. Na mapě „Orlí vrch“, kde byl testován s vysoce agresivní heuristikou ($W = 1.0$), dosáhl překvapivě dobrého času (5.7 s), rychlejšího než Delta-Stepping i HADS a dokonce i než CPU Lazy Theta*. To je však vykoupeno výrazně kratší nalezenou cestou (11.36km vs 12.11km u A*/Delta), což silně naznačuje, že tato agresivní heuristika vedla algoritmus k suboptimálnímu řešení. Tento výsledek demonstruje, jak zásadní je vliv heuristiky na chování A* – agresivní heuristika může zrychlit nalezení nějaké cesty, ale za cenu ztráty garance optimality. Pro srovnání optimality by bylo nutné použít stejnou (admisibilní) heuristiku.

Kvalita nalezených tras: CPU A* a GPU Delta-Stepping konzistentně nacházejí trasy s velmi podobnou (pravděpodobně optimální dle modelu) geometrickou délkou. CPU Any-Angle varianty na menších mapách našly mírně delší trasy. GPU HADS konzistentně nachází geometricky nejkratší trasy, což je zajímavé a může to být dáno kombinací prořezávání a odlišného průzkumu grafu. Rozdíl mezi geometrickou délkou a „časovou optimálností“ je klíčový – kratší trasa nemusí být rychlejší, pokud vede náročným terénem nebo velkým stoupáním.

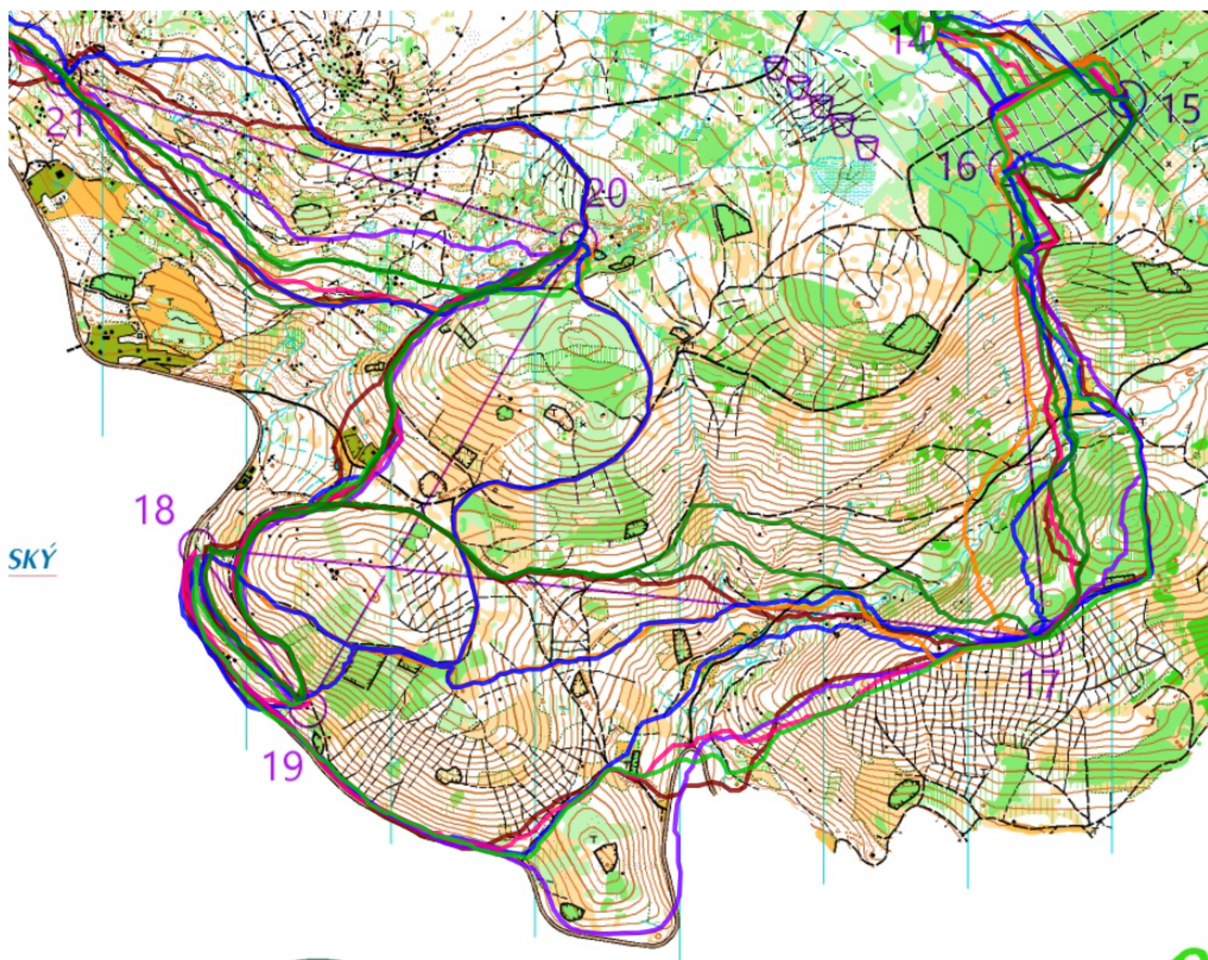
Závěrečné shrnutí syntetických testů: Pro úlohy o velikosti map „Včelný“ a „Oceán“ (1000x1000, 6km trasa) zůstává optimalizovaný CPU A* nejrychlejší metodou garantující optimální výsledek dle modelu. Na výrazně větší mapě „Orlí vrch“ (3000x3000, 12km trasa) se již začíná projevovat potenciál GPU HADS a Delta-Stepping, které překonávají pomalejší CPU Any-Angle varianty, ačkoliv CPU A* je stále rychlejší. GPU A* s agresivní heuristikou může být rychlý, ale za cenu optimality. Volba algoritmu tedy závisí na požadované rychlosti, velikosti problému, dostupném hardwaru a akceptovatelné míře suboptimality (v případě agresivních heuristik nebo prořezávání v HADS). Lazy Theta* je jasně preferovanou Any-Angle alternativou na CPU.

7.5 Vizuální analýza a pozorování (Mapa „Oceán“)

Pro detailní posouzení, jak se trasy generované jednotlivými klíčovými algoritmy shodují či liší od reálných postupů elitních běžců, byla provedena vizuální analýza. Testovaný úsek na mapě „Oceán“ odpovídal v reálném závodě úseku mezi kontrolami 15 (Start) a 21 (Cíl), s pěti mezikontrolami (K1-K5 odpovídající závodním K16-K21).

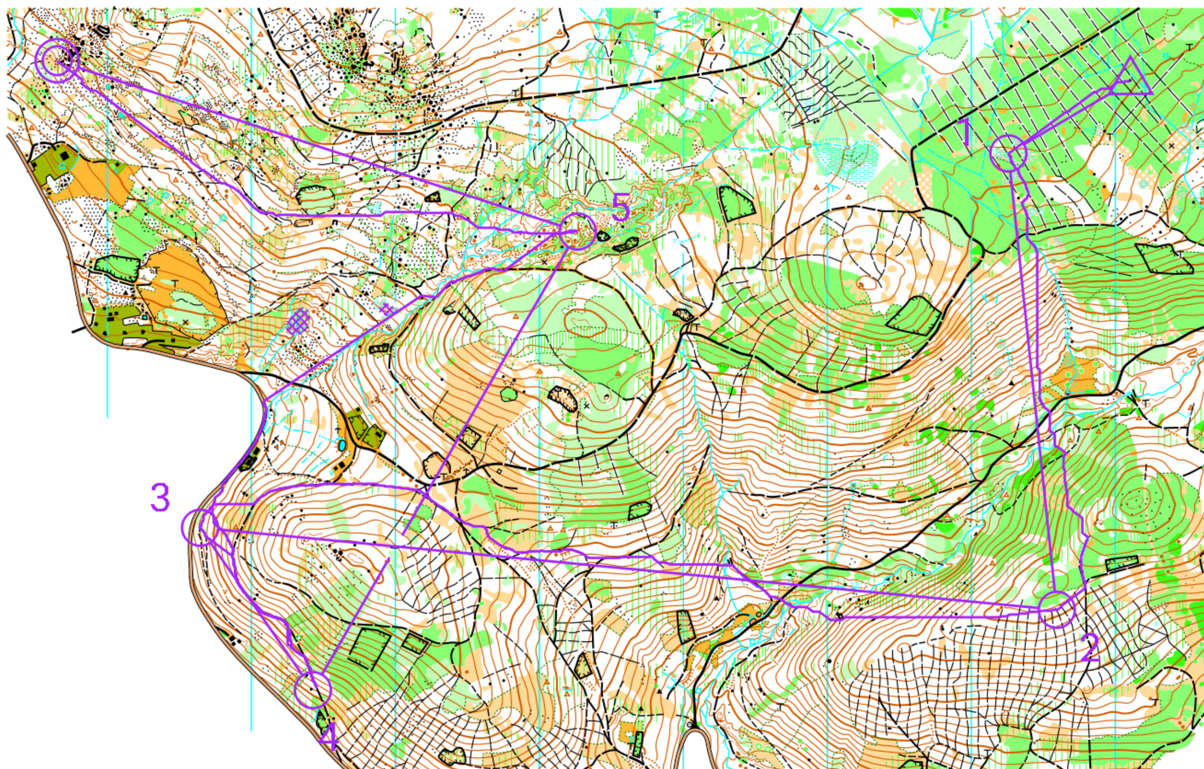
Reálné GPS trasy závodníků: Obrázek 12 nejprve ukazuje překryv přibližně 15 GPS záznamů elitních běžců (kategorie H21E) z daného závodu. Je patrná typická variabilita ve volbě postupů, nicméně lze identifikovat hlavní „koridory“ a preferované varianty, které slouží jako reference pro porovnání s modelovanými trasami. Běžci se obecně snaží využívat cesty a zřetelné terénní prvky, ale někteří volí i přímější, technicky náročnější varianty.

Následně porovnáme trasy generované jednotlivými algoritmy s těmito reálnými postupy.



Obr. 12: Překryv reálných GPS záznamů elitních běžců na mapě „Oceán“ pro úsek kontrol 16-21.

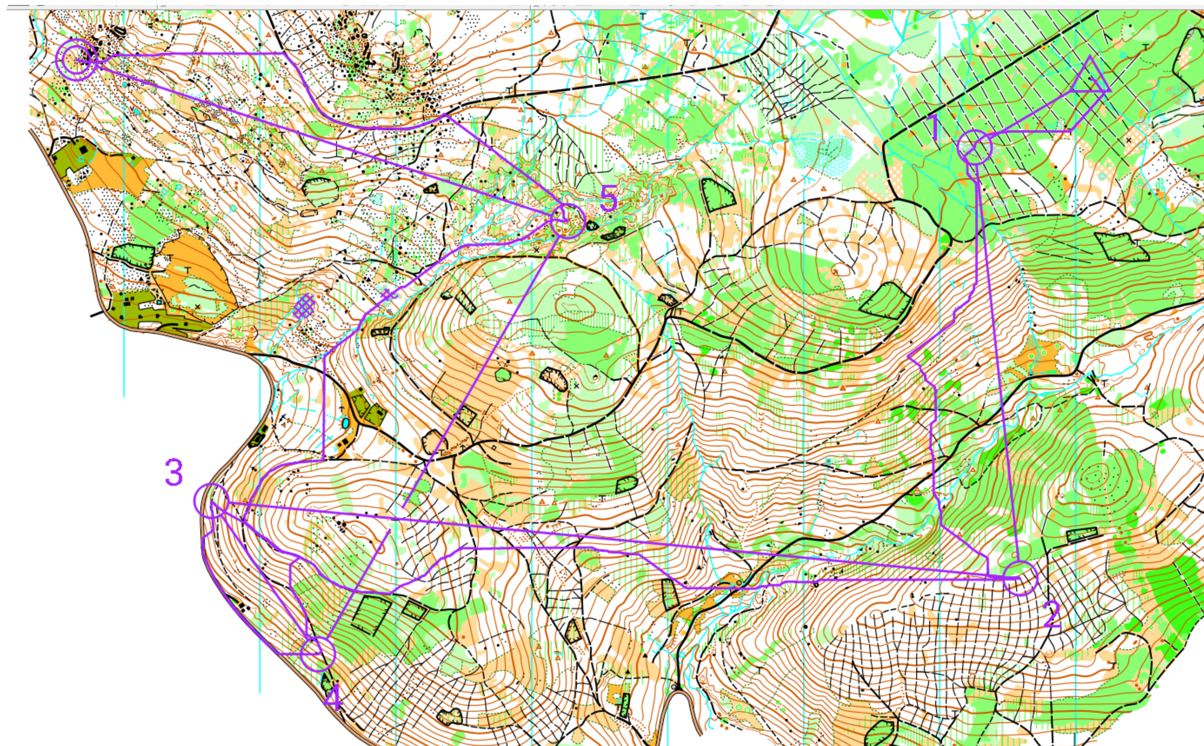
CPU A* (a GPU Delta-Stepping): Trasa nalezená algoritmem CPU A* (fialová čára na Obrázku 13), která je téměř identická s výsledkem GPU Delta-Stepping, vykazuje tendenci k volbě časově nejefektivnějších cest dle modelu, což často znamená využití existujících komunikací.



Obr. 13: Porovnání trasy CPU A* na mapě „Oceán“ (úsek kontrol Start(15) → Cíl(21)).

V úsecích Start(15) → K1(16), K1(16) → K2(17) a K4(19) → K5(20) se trasa A* velmi dobře shoduje s hlavními proudy reálných běžců, využívá cesty a optimálně obíhá překážky. Výraznější odlišnost je patrná v postupu K2(17) → K3(18). Zatímco mnoho elitních běžců volilo severnější variantu s mírnějším stoupáním po cestě, A* navrhl jižnější variantu. Ta sice zpočátku využívá cestu v klesání, ale následně zahrnuje agresivnější přímé stoupání ke kontrole K3(18). Jak bylo diskutováno v podkapitole 5.2.3 o výpočtu ceny hran, náš model s Toblerovou funkcí může méně penalizovat klesání. A* tedy mohl vyhodnotit úsporu času na klesání a kratší vzdálenost stoupání jako celkově výhodnější, zatímco reální běžci mohli preferovat plynulejší stoupání nebo minimalizaci absolutně nastoupaných metrů.

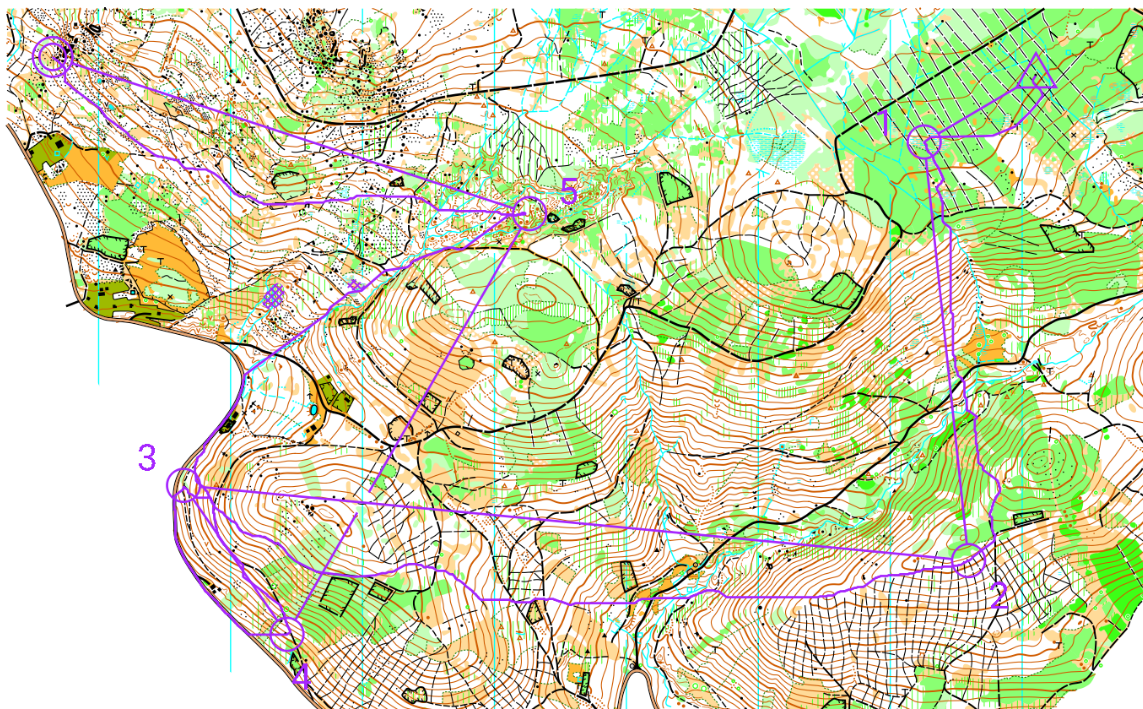
CPU Lazy Theta*: Algoritmus CPU Lazy Theta* (fialová čára na Obrázku 14) jako zástupce Any-Angle přístupu hledá přímější spojení.



Obr. 14: Porovnání trasy CPU Lazy Theta* na mapě „Oceán“ (úsek kontrol Start(15) → Cíl(21)).

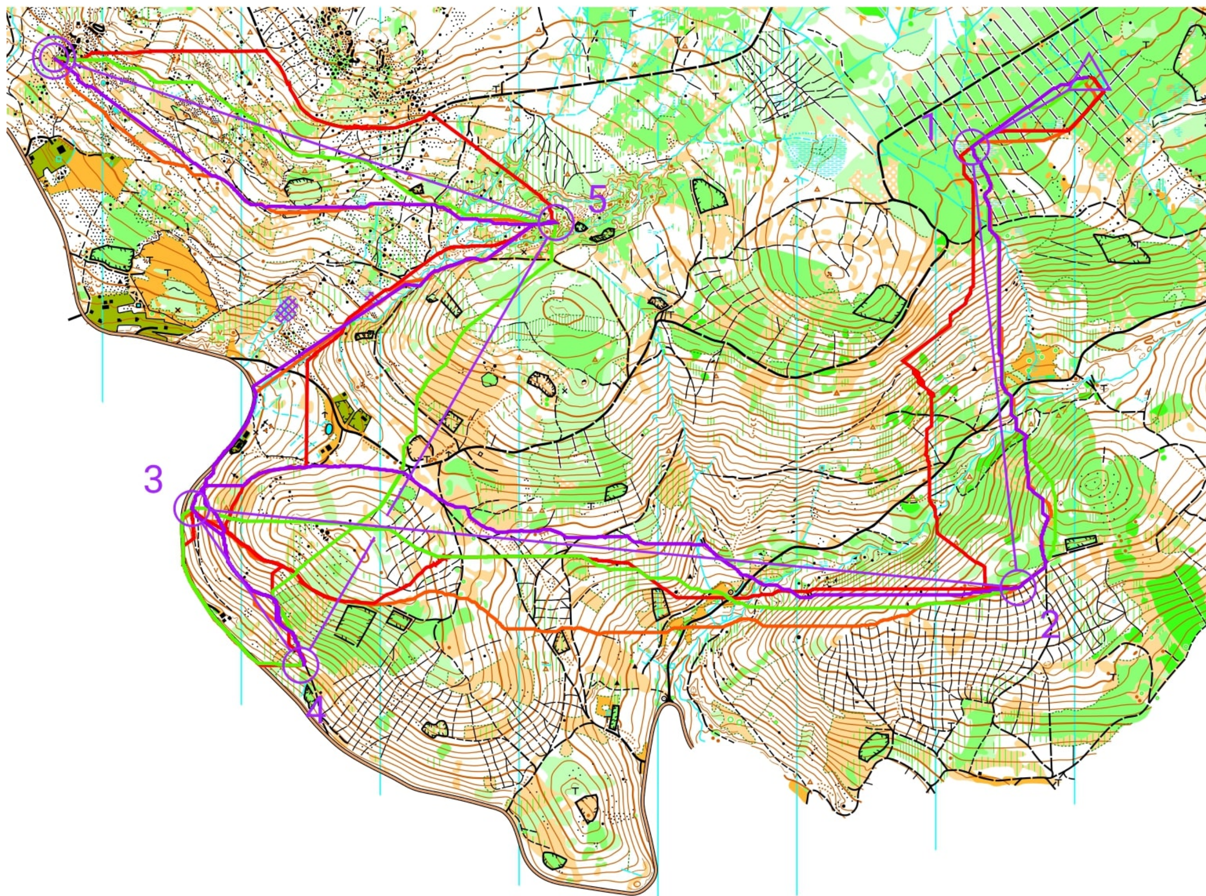
Nejvýraznější odchylka od A* a části reálných běžců je viditelná v úseku Start(15) → K1(16). Lazy Theta* zde identifikoval možnost dlouhého přímého běhu přes les (viz princip Line-of-Sight popsaný v podkapitole 2.1.2), následně se napojil na cestu a přiblížil se k hlavnímu koridoru. Toto chování, kdy algoritmus preferuje delší přímé úseky, i když opouští zjevné cesty, může odpovídat strategii některých rychlých běžců v přehledném terénu. V dalších úsecích, např. K1(16) → K2(17) nebo K3(18) → K4(19), se trasa Lazy Theta* více podobá A* a reálným běžcům, což naznačuje, že přímé „zkratky“ nebyly modelem vyhodnoceny jako rychlejší, pravděpodobně kvůli hustší vegetaci, sklonu nebo nutnosti překonávat drobné překážky podél LoS.

GPU HADS: Trasa generovaná algoritmem GPU HADS (fialová čára na Obrázku 15) často volí kompromisní nebo unikátní postupy díky kombinaci Delta-Stepping a heuristického prořezávání (viz podkapitola 5.3.2).



Obr. 15: Porovnání trasy GPU HADS na mapě „Oceán“ (úsek kontrol Start(15) → Cíl(21)).

HADS demonstruje tendenci využívat dlouhá klesání (např. v první části postupu K2(17) → K3(18), kde se odklání od A*), aby maximalizoval rychlost dle Toblerovy funkce. V úseku K1(16) → K2(17) volí podobný postup jako A* a většina běžců. Nejoriginálnější postup ukazuje HADS v úseku K3(18) → K4(19). Zatímco A* a Lazy Theta* se drží více cest na severní straně, HADS volí přímější jižní směr přes členitější terén s menšími cestami a průseky. Tento postup se shoduje s volbou několika reálných elitních běžců, kteří se nebáli riskovat technicky náročnější pasáž. To ukazuje, že HADS dokáže identifikovat „agresivnější“ varianty, které mohou být pro určité typy běžců (nebo při specifickém nastavení parametrů `PRUNE_FACTOR` a `Heuristic Weight`) relevantní.

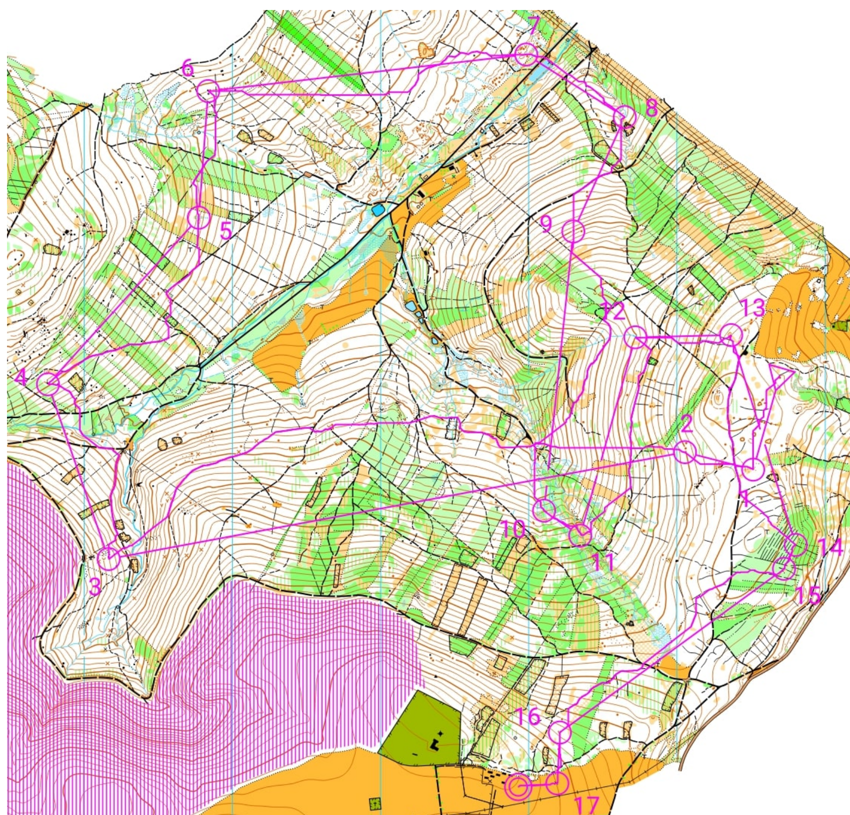


Obr. 16: Porovnání všech algoritmů CPU A* (fialová), GPU A* (světle zelená),Lazy THETA*(červená) a HADS(oranžová) na mapě „Oceán“

7.6 Vizuální analýza a pozorování (Mapa „Orlí vrch“)

Pro ilustraci chování modelu na mapě s odlišným charakterem a větší rozlohou byl proveden také vizuální rozbor trasy na mapě „Orlí vrch“. Vzhledem k rozsáhlosti trasy (19 waypointů, 10.4 km reálné délky) se zde zaměříme na porovnání trasy generované algoritmem CPU A* (který v syntetických testech konzistentně nacházel optimální cestu dle modelu) s reprezentativním GPS záznamem jednoho z elitních běžců.

(a) Náhled mapy s reálným běžcem závodu MČR 2013 „Orlí vrch“.



(b) Náhled mapy „Orlí vrch“ s A* jeho řešení.

Obr. 17: Srovnání náhledu mapy MČR 2013 „Orlí vrch“ s reálným běžcem (nahoře) a řešením pomocí algoritmu A* (dole).

Jak je patrné z Obrázku 17b, trasa generovaná algoritmem A* vykazuje na většině úseků mezi kontrolami vysokou míru shody s postupem reálného závodníka. Oba postupy efektivně využívají existující cesty a pěšiny, volí logické obíhání výrazných terénních překážek a snaží se minimalizovat zbytečné stoupání v kopcovitém terénu charakteristickém pro tuto mapu.
Zde lze rozpoznat několik zajímavých úseků:

- **Většina úseků:** Na mnoha postupech, například mezi kontrolami 5-6, 10-11 nebo 16-17, se modelovaná trasa A* a reálný postup téměř dokonale překrývají. To naznačuje, že v těchto pasážích nákladový model (kombinace ceny terénu a Toblerovy funkce pro sklon) dobře aproximoval faktory ovlivňující rozhodování běžce a A* nalezl skutečně optimální variantu.
- **Odlišná volba na postupu 2-3:** Zde je vidět jedna z mála výraznějších odchylek. Zatímco algoritmus A* volil pravděpodobně cestu s optimálním profilem dle modelu, reálný závodník zvolil variantu s výraznějším klesáním a následným stoupáním, které se z pohledu modelu mohlo jevit jako časově méně výhodné (ztráta výšky, kterou bylo nutné znovu nabrat). Toto může být způsobeno buď mírnou navigační chybou závodníka, nebo preferencí jiného typu terénu/průchodnosti, kterou model plně nezachytil.
- **Drobná nejistota před kontrolou 14:** V oblasti před kontrolou 14 je na GPS záznamu patrné mírné zaváhání nebo krátká ztráta směru reálného běžce, zatímco model A* vede přímočařeji k cíli. Tento typ malých odchylek je v orientačním běhu běžný a model, který má dokonalé informace o mapě a své poloze, jej samozřejmě nevykazuje.

Celkově lze konstatovat, že i na této rozsáhlé a náročné mapě model A* generuje trasy, které jsou velmi podobné a srovnatelné s postupy zkušených orientačních běžců. Odchylky jsou spíše bodové a mohou být přičítány buď specifickým rozhodnutím závodníka v daný okamžik, drobným navigačním nepřesnostem, nebo limitacím modelu v přesném zachycení všech nuancí reálného terénu a fyziologie běhu. Skutečnost, že A* dokáže nalézt takto relevantní postupy, potvrzuje robustnost použitého nákladového modelu a vhodnost A* algoritmu pro tento typ úlohy.

Závěrečné shrnutí porovnání s realitou: Modelované trasy vykazují značnou shodu s reálnými postupy elitních běžců, zejména v pasážích, kde je volba optimálního postupu relativně jednoznačná. Rozdíly se objevují v technicky náročnějších úsecích nebo tam, kde modelové předpoklady (např. o vlivu sklonu na rychlost, přesnost mapových dat o podrostu) nemusí plně odpovídat komplexnímu rozhodování běžce. Každý z testovaných algoritmů nabízí mírně odlišný „pohled“ na optimální trasu: A* jako modelově nekompromisní optimalizátor, Lazy Theta* jako hledač přímějších spojení a HADS jako flexibilní metoda schopná nalézt jak konzervativní, tak agresivnější varianty. Výběr „nejlepšího“ algoritmu pro predikci reálného chování je obtížný a závisí na mnoha faktorech, včetně stylu konkrétního běžce a přesnosti vstupních dat.

7.7 Souhrnná diskuse a interpretace výsledků

Experimentální vyhodnocení na syntetických datech i porovnání s reálnými GPS záznamy elitních běžců přineslo řadu důležitých poznatků o výkonnosti a charakteristikách implementovaných algoritmů a celého navrženého systému pro hledání optimální trasy v orientačním běhu.

Výkonnostní aspekty a role hardwaru: Klíčovým zjištěním napříč testovanými scénáři je skutečnost, že pro danou velikost problému (mřížky až 3000x3000 buněk) a použitou hardwarovou konfiguraci zůstává optimalizovaný sekvenční CPU A* nejrychlejší metodou pro nalezení cesty, která je dle modelu optimální. Přestože GPU algoritmy jako Delta-Stepping a HADS demonstrují schopnost paralelizovat výpočet a na největší mapě („Orlí vrch“) již překonávají pomalejší CPU Any-Angle varianty, nedosahují zatím rychlosti CPU A*. Tento výsledek poukazuje na významnou režii spojenou s GPU výpočty (přenosy dat, latence spouštění kernelů, synchronizace), která u problémů této velikosti ještě může převážit nad přínosy masivního paralelizmu. Lze však důvodně předpokládat, že pro výrazně rozsáhlejší grafy nebo na výkonnějších GPU architekturách by se škálovatelnost Delta-Stepping a HADS projevila dominantněji. Implementace A* na GPU s expanzí jednoho uzlu za iteraci se ukázala jako nevhodná a nekonkurenceschopná.

Charakteristiky nalezených tras a relevance algoritmů: Porovnání kvality tras odhalilo zajímavé rozdíly. Zatímco CPU A* a GPU Delta-Stepping (s vhodnými parametry) konzistentně nacházely trasy s velmi podobnou, pravděpodobně časově optimální délkou dle modelu, Any-Angle algoritmy (Theta*, Lazy Theta*) a GPU HADS často generovaly geometricky odlišné, někdy i kratší, trasy.

- **Any-Angle (Lazy Theta*):** Prokázal schopnost nalézt přímější spojení přes otevřenější terén, což lépe odpovídá reálnému pohybu běžce v určitých situacích. Vyšší výpočetní náročnost na CPU je však značná. Jeho přínos byl patrnější tam, kde model vyhodnotil přímé pasáže jako výhodné; v členitějším terénu se jeho volby více blížily A*.
- **GPU HADS:** Tento algoritmus se ukázal jako nejrychlejší z GPU metod a díky heuristickému prořezávání často nacházel geometricky nejkratší trasy. Jak ukázalo porovnání s reálnými GPS daty na mapě „Oceán“, jeho „agresivnější“ volby (např. využití strmých klesání, přímější průseky) mohou odpovídat stylu některých elitních běžců. Flexibilita jeho parametrů umožňuje ladit míru této „agresivity“, ale zároveň vyvolává otázku garance nalezení skutečně časově optimální trasy dle modelu, pokud je prořezávání příliš silné.

Porovnání s reálnými GPS trasami na mapě „Oceán“ potvrdilo, že všechny hlavní algoritmy (CPU A*, Lazy Theta*, GPU HADS) dokáží generovat smysluplné a relevantní trasy. Rozdíly mezi modelovanými a reálnými postupy lze přičíst jak limitacím modelu (nezahrnutí všech faktorů jako únava, mikro-navigace, aktuální stav podrostu, riziko), tak variabilitě v rozhodování samotných běžců. Žádný model pravděpodobně nikdy dokonale nezkopíruje lidské rozhodování, ale cílem je poskytnout co nejlepší aproximaci a nástroj pro analýzu.

Limitace modelu a implementace: Hlavní identifikovanou limitací současného řešení je rozlišení a přesnost vstupních výškových dat. Veřejně dostupná DMT s rozlišením 90 m nemusí zachytit jemné terénní detaily, které jsou pro orientačního běžce klíčové. To může ovlivnit přesnost výpočtu sklonu a následně i realističnost Toblerovy funkce a celého nákladového modelu. Modulární návrh systému však umožňuje budoucí integraci přesnějších dat. Dalším aspektem je již zmíněná skutečnost, že pro testované scénáře se plně neprojevil očekávaný drtivý výkonnostní přínos GPU akcelerace oproti vysoce optimalizovanému CPU A*. To zdůrazňuje, že paralelizace není samospásná a její efektivita silně závisí na povaze problému, velikosti dat a konkrétní implementaci.

Celkové zhodnocení a přechod k závěru: Vyvinutý systém a provedené experimenty demonstrují komplexnost problému hledání optimální trasy v orientačním běhu. Ukázalo se, že různé algoritmické přístupy nabízejí odlišné kompromisy mezi rychlostí výpočtu, geometrickou charakteristikou trasy a potenciální věrností reálnému chování. Zatímco CPU A* zůstává silným a rychlým nástrojem pro danou velikost problému, paralelní GPU algoritmy, zejména HADS, naznačují směr pro budoucí škálování na ještě komplexnější úlohy. Porovnání s reálnými daty potvrdilo, že modelované trasy jsou relevantní, ale zároveň odhalilo prostor pro další vylepšování nákladového modelu a zohlednění dalších faktorů. Detailnější shrnutí přínosů práce, jejích limitací a návrhů na budoucí práci bude uvedeno v následující kapitole.

8 Závěr

Cílem této diplomové práce bylo navrhnout, implementovat, vyhodnotit systém pro hledání časově optimální trasy v orientačním běhu. Důraz byl kladen na zohlednění reálných faktorů ovlivňujících rychlost postupu, jako je charakter terénu, jeho sklon, rovněž na prozkoumání efektivity různých algoritmických přístupů, včetně jejich akcelerace pomocí výpočtů na grafickém procesoru (GPU).

V rámci práce byla úspěšně vyvinuta desktopová aplikace *OMAP Pathfinding Processor*. Ta integruje celý proces od načtení, zpracování mapových dat ve formátu `.omap`, externích výškových dat, přes tvorbu diskrétní mřížkové reprezentace terénu až po samotné hledání cesty. Byly implementovány, porovnány jak standardní CPU algoritmy (A*, Theta*, Lazy Theta*), tak jejich paralelní GPU varianty (Delta-Stepping, experimentální A* na GPU). Vlastním přínosem bylo navržení a implementace metody HADS (Heuristic-Accelerated Delta-Stepping), která kombinuje robustní paralelizmus algoritmu Delta-Stepping s heuristickým prořezáváním inspirovaným algoritmem A*, s cílem efektivněji zaměřit prohledávání na GPU. Nákladový model pro výpočet cen hran grafu dynamicky kombinuje základní cenu terénu odvozenou z ISOM symbolů mapy, penalizaci za sklon dle Toblerovy funkce.

Experimentální vyhodnocení na několika reálných mapách pro orientační běh, včetně map s různou charakteristikou, komplexitou („Les Včelný“, „Oceán“, „Orlí vrch“), ukázalo následující klíčová zjištění. Pro testované velikosti problémů (mřížky 1000x1000 až 3000x3000 buněk) se jako nejrychlejší metoda garantující optimální výsledek dle modelu ukázal optimalizovaný sekvenční CPU A*. Any-Angle algoritmy (Theta*, Lazy Theta*) na CPU generovaly přirozenější trasy, ovšem za cenu výrazně vyšší výpočetní náročnosti, přičemž Lazy Theta* byl z nich efektivnější. Paralelní GPU algoritmy založené na Delta-Stepping (čistý Delta-Stepping, HADS) prokázaly potenciál pro škálování; na největší testované mapě „Orlí vrch“ již překonaly CPU Lazy Theta*, ačkoliv stále nedosáhly rychlosti CPU A*. HADS konzistentně nabízel nejlepší výkon z GPU metod, generoval zajímavé, často geometricky nejkratší trasy, což naznačuje přínos heuristického prořezávání. Implementace A* na GPU s expanzí pouze jednoho uzlu za iteraci se ukázala jako neefektivní. Porovnání s reálnými GPS záznamy elitních běžců na mapě „Oceán“ ukázalo, že všechny hlavní implementované algoritmy (CPU A*, Lazy Theta*, GPU HADS) jsou schopny nalézt trasy vykazující značnou podobnost s postupy špičkových závodníků, i když se liší v preferenci některých typů průchodu terénem. Každý algoritmus tak může reprezentovat mírně odlišný „styl“ běhu nebo strategii.

Hlavním přínosem této práce je návrh, realizace komplexní aplikace pro analýzu, hledání optimálních postupů v orientačním běhu, která integruje moderní pathfinding algoritmy, technologie. Práce poskytuje srovnání výkonnosti, charakteristik různých přístupů (CPU vs. GPU, standardní vs. Any-Angle vs. Delta-like) na reálných datech. Vyvinutý software může sloužit jako platforma pro další výzkum, experimentování v oblasti modelování optimálních tras. Taktéž je praktický pro organizátory závodů, aby našli pravděpodobné nejrychlejší postupy, jejich délky.

Identifikovanou limitací současného řešení je především nízké rozlišení vstupních výškových dat (typicky 90 m z veřejných API), které nemusí dostatečně přesně zachycovat jemné

terénní nerovnosti klíčové pro volbu postupu v OB. Existují dvě hlavní cesty pro zlepšení: buď implementace metod pro extrakci detailnějších výškových informací přímo z vrstevnic mapy pro orientační běh (což je náročný úkol sám o sobě), nebo získání, integrace přesnějších externích DMT (např. lidarová data). Díky modulárnímu návrhu aplikace, zejména třídy `ElevationSampler`, by však integrace přesnějších výškových dat měla být relativně přímočará; aplikace je na takové rozšíření připravena. Další limitací je, že pro testované velikosti problémů se plně neprojevil očekávané masivní zrychlení GPU algoritmů oproti optimalizovanému CPU A*, což naznačuje, že režie GPU je stále významná.
Pro budoucí rozvoj se nabízí několik zajímavých směrů:

- **Rasterizace mapy na GPU:** Přesun fáze zpracování mapy na GPU by mohl výrazně zrychlit celkový čas potřebný pro přípravu dat, zejména pro velmi velké, komplexní mapy.
- **Využití pro trénink AI:** Základ aplikace, generované trasy by mohly sloužit jako vstup pro systémy umělé inteligence. Natrénováním modelu na GPS datech reálných běžců, jejich postupech by AI mohla naučit „styl“ konkrétního běžce, následně předvídat jeho volby nebo navrhovat trasy přizpůsobené jeho schopnostem.
- **Automatická kalibrace nákladového modelu:** Pomocí velkého množství GPS dat, odpovídajících map by bylo možné se pokusit automaticky (např. strojovým učením) aproximovat „reálné“ časové náklady jednotlivých mapových symbolů, sklonů, tím vylepšit přesnost nákladové funkce.
- **Implementace Fuzzy A*:** Varianta A* algoritmu pracující s fuzzy logikou by mohla lépe modelovat neurčitost, vágnost informací v mapě nebo v odhadu schopností běžce. Základní koncepty pro tuto variantu jsou připraveny; integrace by vyžadovala úpravy v reprezentaci cen, heuristiky.
- **Testování na rozsáhlejších datech, výkonnějším GPU:** Pro ověření škálovatelnosti GPU algoritmů by bylo přínosné provést testy na výrazně větších mapách, modernějších GPU architekturách.

Celkově lze konstatovat, že cíle práce byly naplněny. Byl vytvořen funkční systém schopný hledat časově optimální trasy v komplexním terénu s využitím různých pokročilých algoritmů; byla provedena jejich evaluace. Práce demonstruje potenciál, výzvy spojené s aplikací moderních algoritmických a výpočetních technik na specifický problém orientačního běhu.

Seznam použité literatury

[1] Tsiligiridis, T. Heuristic Methods Applied to Orienteering. *Journal of the Operational Research Society*, Září 1984, sv. 35, s. 797–809.

[2] Albert, G. a Sárközy, Z. Route planning on orienteering maps with least-cost path analysis. *Proceedings of the ICA*, 2021, sv. 4, s. 4. Dostupné z: `https://ica-proc.copernicus.org/articles/4/4/2021/`.

[3] Rosenkrantz, D. J.; Stearns, R. E. a Lewis, P. M. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing [online]. Posl. aktualizace*, Sep 1977, sv. 6, č. 3, s. 563–581. Dostupné z: `https://doi.org/10.1137/0206041`.

[4] Lerner, J.; Wagner, D. a Zweig, K. A. *Algorithmics of large and Complex Networks: Design, analysis, and simulation.* Springer, 2009.

[5] Birx, A.; Disser, Y.; Hopp, A. V. a Karousatou, C. An improved lower bound for competitive graph exploration. *Theoretical Computer Science [online]*, 2021, sv. 868, s. 65–86. ISSN 0304-3975. Dostupné z: `https://doi.org/10.1016/j.tcs.2021.04.003`.

[6] *5.7.1 Dijkstra Algorithm.* Dostupné z: `http://students.ceid.upatras.gr/~papagel/project/kef5_7_1.htm`.

[7] Wenderlich, R. *Introduction to a* pathfinding.* Září 2011. Dostupné z: `https://www.kodeco.com/3016-introduction-to-a-pathfinding`.

[8] Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 1974, sv. 5, č. 2, s. 115–135. ISSN 0004-3702. Dostupné z: `https://www.sciencedirect.com/science/article/pii/0004370274900265`.

[9] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, Dec 1959, sv. 1, č. 1, s. 269–271.

[10] *Artificial Intelligence: A Modern Approach, 4th US ed.* Dostupné z: `https://aima.cs.berkeley.edu/`.

[11] Hart, P. E.; Nilsson, N. J. a Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 1968, sv. 4, č. 2, s. 100–107.

[12] Delling, D.; Sanders, P.; Schultes, D. a Wagner, D. *Engineering route planning algorithms.* Leden 2009. 117–139 s. Dostupné z: `https://doi.org/10.1007/978-3-642-02094-0_7`.

[13] van der Heijden, F.; Duin, R.; de Ridder, D. a Tax, D. *Classification, parameter estimation and state estimation: an engineering approach using matlab.* United States: Wiley, září 2004. ISBN 978-0-470-09013-8.

[14] *Implementation notes*. 2025. Dostupné z: https://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html.

[15] Uras, T. a Koenig, S. An empirical comparison of Any-Angle Path-Planning algorithms. *Proceedings of the International Symposium on Combinatorial Search*, Září 2021, sv. 6, č. 1, s. 206–210. Dostupné z: https://doi.org/10.1609/socs.v6i1.18382.

[16] Daniel, K.; Nash, A.; Koenig, S. a Felner, A. Theta*: Any-Angle path planning on grids. *Journal of Artificial Intelligence Research*, Říjen 2010, sv. 39, s. 533–579. Dostupné z: https://arxiv.org/abs/1401.3843.

[17] Meyer, U. a Sanders, P. -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms [online]*, 2003, sv. 49, č. 1, s. 114–152. ISSN 0196-6774. Dostupné z: https://doi.org/10.1016/S0196-6774(03)00076-2.

[18] Burda, M. *Historie Orienťáku? To Je Povinná Zátěž na záda a Místo čipu vědomostní otázky...* online. November 2020. Dostupné z: https://www.svetbehu.cz/historie-orientaku-to-je-povinna-zatez-a-misto-cipu-vedomostni-otazky/. [cit. 2025-04-29].

[19] Dostupné z: https://www.theworldgames.org/sports/Orienteering-34.

[20] Faigl, J.; Pěnička, R. a Best, G. Self-organizing map-based solution for the Orienteering problem with neighborhoods. In: *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2016, s. 001315–001321.

[21] *Pravidla OB 2022* online. Dostupné z: https://www.orientacnibeh.cz/upload/dokumenty/sekce-ob/pravidla-ob-2022.pdf. [cit. 2025-04-29].

[22] *ISOM Specifications* online. Dostupné z: https://omapwiki.orienteering.sport/specifications/isom/. [cit. 2025-04-29].

[23] *ISSProm Specifications* online. Dostupné z: https://omapwiki.orienteering.sport/specifications/issprom/. [cit. 2025-04-29].

[24] *Mapper* online. Dostupné z: https://www.openorienteering.org/apps/mapper/. [cit. 2025-04-29].

[25] *OpenOrienteering Mapper User Manual* online. Dostupné z: https://www.openorienteering.org/mapper-manual/pages/. [cit. 2025-04-29].

[26] *Frequently Asked Questions (FAQ) | OpenOrienteering Mapper User Manual* online. Dostupné z: https://www.openorienteering.org/mapper-manual/pages/faq.html. [cit. 2025-04-29].

[27] *1. Introduction - CUDA C++ Programming Guide* online. February 2025. Dostupné z: https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [cit. 2025-04-29].

[28] *CUDA Device Architecture* online. Dostupné z: `https://modal.com/gpu-glossary/device-hardware/cuda-device-architecture`. [cit. 2025-04-29].

[29] MITCHELL, J. S. B. An Algorithmic Approach to Some Problems in Terrain Navigation. *Artif. Intell.*, 1988, sv. 37, 1-3, s. 171–201. Dostupné z: `https://doi.org/10.1016/0004-3702(88)90054-9`.

[30] ETTEN, J. van. R Package gdistance: Distances and Routes on Geographical Grids. *Journal of Statistical Software*, 2017, sv. 76, č. 13, s. 1–21. Dostupné z: `https://www.jstatsoft.org/index.php/jss/article/view/v076i13`.

[31] TOBLER, W. Three presentations on geographical analysis and modeling: Non-isotropic geographic modeling; speculations on the geometry of geography; and Global Spatial Analysis (93-1). *EScholarship, University of California [online]*, Jun 2015. Dostupné z: `https://escholarship.org/uc/item/05r820mz`.

Seznam obrázků

1 Ukázka mapy ze závodu MČR 35
2 Příklad výřezu mapy pro orientační běh 37
3 Ilustrace rozdílu mezi výsledkem metody Flood Fill a Scanline Fill 52
4 Schéma aplikace a jejího vnitřního propojení 70
5 Hlavní okno po spuštění aplikace OMAP Pathfinding Processor. 72
6 Panel nastavení - záložka „Map & Processing“. 73
7 Panel nastavení - výběr algoritmu. 74
8 Příklad mapy ve formátu `.omap` s vykreslenou trasou 75
9 Náhled testovací mapy „Les Včelný“. 78
10 Náhled testovací mapy „Oceán“. 78
11 Náhled testovací mapy „Orlí vrch“. 79
12 Překryv reálných GPS záznamů elitních běžců na mapě „Oceán“ 83
13 Porovnání trasy CPU A* na mapě „Oceán“ 84
14 Porovnání trasy CPU Lazy Theta* na mapě „Oceán“ 85
15 Porovnání trasy GPU HADS na mapě „Oceán“ 86
16 Porovnání všech algoritmů na mapě „Oceán“ 87
17 Náhledy mapy Orlí vrch s reálným běžcem a A* řešením 88

Seznam příloh

A **První příloha CMake** ... **99**
B **Druhá příloha Visual Studio** ... **103**

A První příloha CMake

Tato příloha poskytuje přehled organizace zdrojových kódů projektu a popisuje kroky potřebné pro jeho sestavení pomocí systému CMake.

A.1 Struktura adresářů projektu

Hlavní adresář projektu obsahuje konfigurační soubory CMake a následující klíčové podadresáře:

- `/.vs/` - Adresář generovaný prostředím Visual Studio, obsahující dočasné soubory projektu a cache (není součástí verzování).
- `/cuda/` - Obsahuje kód specifický pro CUDA:
  - `/cuda/include/` - Hlavičkové soubory (`.cuh`) pro CUDA kernely a device funkce. Obsahuje například `PathfindingUtilsGPU.cuh`.
  - `/cuda/src/` - Zdrojové soubory CUDA kernelů (`.cu`), například `AStarKernels.cu`, `DeltaSteppingKernels.cu`, `HADSKernels.cu`.
- `/include/` - Hlavičkové soubory (`.hpp`, `.h`) pro C++ části projektu, logicky členěné do podadresářů:
  - `/algoritms/` - Hlavičkové soubory pro všechny implementované pathfinding algoritmy (CPU i GPU wrappery).
  - `/debug/` - Pomocné nástroje pro ladění.
  - `/IO/` - Knihovny a třídy pro vstup/výstup (např. `tinyxml2.h`, `PathSaver.hpp`).
  - `/logic/` - Hlavičkové soubory pro hlavní aplikační logiku (`PathfindingLogic.hpp`) a rozhraní (`BackendInterface.hpp`).
  - `/map/` - Komponenty pro zpracování mapy, elevace a geometrie (`MapProcessor.hpp`, `ElevationFetcherPy.hpp`, `WaypointExtractor.hpp` atd.).
- `/out/` - Výstupní adresář generovaný CMake, typicky obsahující podadresáře pro různé konfigurace sestavení (např. `/debug/`, `/release/`). Zde se nachází výsledný spustitelný soubor a další artefakty sestavení.
- `/python/` - Obsahuje Python skripty využívané aplikací, například `elevation_logic.py` pro získávání výškových dat.
- `/src/` - Zdrojové soubory C++ (`.cpp`), členěné podobně jako adresář `/include/`:
  - `main.cpp` - Hlavní vstupní bod aplikace Qt.
  - `/algoritms/` - Implementace CPU pathfinding algoritmů a C++ wrapperů pro GPU algoritmy.
  - `/debug/` - Implementace ladících nástrojů.
  - `/gui/` - Implementace grafického uživatelského rozhraní (`main_window.cpp`).
  - `/IO/` - Implementace tříd pro vstup/výstup.
  - `/logic/` - Implementace hlavní aplikační logiky.
  - `/map/` - Implementace komponent pro zpracování mapy a geometrie.

V kořenovém adresáři projektu se nachází hlavní soubor `CMakeLists.txt`, který definuje celý proces sestavení, a soubory `CMakePresets.json` a `CMakeUserPresets.json` pro usnadnění konfigurace v různých vývojových prostředích.

A.2 Proces sestavení pomocí CMake

Aplikace využívá systém CMake pro generování nativních souborů sestavení (např. projekty pro Visual Studio, Makefile) a kompilaci. Soubor `CMakeLists.txt` orchestruje celý proces.

A.2.1 Klíčové prvky `CMakeLists.txt`

- **Minimální verze CMake a definice projektu:**

Výpis 8: Definice projektu

```
cmake_minimum_required(VERSION 3.14)
project(MinimalQtProject2 VERSION 1.0 LANGUAGES CXX)
```

- **Nastavení C++ standardu a Qt:** Je vyžadován C++17 a jsou povoleny automatické nástroje Qt (MOC, RCC, UIC). Následně se vyhledává požadovaná verze Qt5 (komponenta Widgets).

Výpis 9: Nastavení C++ a Qt

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)
find_package(Qt5 COMPONENTS Widgets REQUIRED)
```

- **Integrace Pythonu a Pybind11:** CMake skript dynamicky vyhledá instalaci Pythonu (verze 3.9 nebo vyšší) a knihovnu Pybind11 pomocí introspekce Pythonu. Tato integrace je nezbytná pro propojení C++ s Python skripty pro získávání výškových dat.

Výpis 10: Integrace Pythonu a Pybind11

```
find_package(Python3 3.9 COMPONENTS Interpreter Development REQUIRED)
execute_process(
    COMMAND "${Python3_EXECUTABLE}" -c "import␣pybind11;
␣␣␣␣print(pybind11.get_include())"
    OUTPUT_VARIABLE PYBIND11_INCLUDE_DIR ...
)
target_include_directories(${PROJECT_NAME} PRIVATE ...
${Python3_INCLUDE_DIRS} ${PYBIND11_INCLUDE_DIR})
target_link_libraries(${PROJECT_NAME} PRIVATE ...
${Python3_LIBRARIES})
```

- **Podpora CUDA (volitelná):** Sestavení s podporou CUDA je řízeno volbou `USE_CUDA`. Pokud je povolena, CMake aktivuje jazyk CUDA, nastaví standard C++ pro CUDA (CUDA C++14) a vyhledá CUDA Toolkit. Zdrojové soubory `.cu` a `.cuh` jsou přidány k projektu a jsou nastaveny příslušné příznaky kompilátoru a cílové architektury GPU (Pascal, Volta, Turing).

Výpis 11: Podpora CUDA

```
option(USE_CUDA "Enable␣CUDA␣acceleration" OFF)
if(USE_CUDA)
    enable_language(CUDA)
    set(CMAKE_CUDA_STANDARD 14)
    find_package(CUDA REQUIRED)
    add_compile_definitions(USE_CUDA=1)
    set_target_properties(${PROJECT_NAME}
    PROPERTIES CUDA_ARCHITECTURES "52;60;70;75")
    target_compile_options(${PROJECT_NAME}
    PRIVATE $<$<COMPILE_LANGUAGE:CUDA>:--use_fast_math>)
endif()
```

- **Sběr zdrojových souborů:** Soubory `.cpp`, `.hpp`, `.h`, `.cu` a `.cuh` jsou rekurzivně sbírány z příslušných adresářů (`src/`, `include/`, `cuda/`). Pokud není povoleno CUDA, GPU specifické implementační soubory (`*GPU.cpp`, `*.cu`) jsou ze seznamu zdrojových souborů C++ odstraněny.
- **Definice spustitelného souboru a include cest:** Je definován hlavní spustitelný soubor projektu a jsou nastaveny cesty k hlavičkovým souborům.

Výpis 12: Definice spustitelného souboru

```
add_executable(${PROJECT_NAME} ${CPP_SOURCES}
${HPP_HEADERS} ${CUDA_SOURCES} ...)
target_include_directories(${PROJECT_NAME} PRIVATE
${CMAKE_CURRENT_SOURCE_DIR}/include
${CMAKE_CURRENT_SOURCE_DIR}/src ...)
```

- **Podpora OpenMP (volitelná):** Pro paralelizaci na CPU je využíváno OpenMP. CMake se pokusí nalézt OpenMP a pokud je detekováno, přidá potřebné příznaky kompilátoru a linkeru.
- **SIMD Optimalizace (AVX2, volitelná):** Projekt umožňuje volitelně zapnout optimalizace pomocí instrukční sady AVX2, pokud ji kompilátor a cílová platforma podporují.
- **Správa Python závislostí:** Je vytvořen soubor `requirements.txt` se seznamem Python knihoven (`requests`, `pyproj`) a je definován vlastní cíl `install_python_deps` pro jejich snadnou instalaci pomocí `pip`.

A.2.2 Postup sestavení

Pro sestavení aplikace je doporučeno použít CMake generátor pro preferované vývojové prostředí (např. Visual Studio) nebo pro makefiles (např. Ninja, NMake).

1. **Předpoklady:**
   - Kompilátor C++ s podporou C++17 (např. MSVC z Visual Studia, GCC, Clang).
   - Nainstalované Qt5 (nebo Qt6 dle konfigurace) vývojové knihovny a nástroje.
   - Nainstalovaný Python (verze 3.9 nebo vyšší) a správce balíčků `pip`.
   - Nainstalovaná knihovna Pybind11 pro Python (`pip install pybind11`).
   - Pokud má být povolena podpora CUDA: Nainstalovaný NVIDIA CUDA Toolkit (verze odpovídající nastavení v CMake) a kompatibilní ovladače GPU.

2. **Konfigurace CMake:** Z kořenového adresáře projektu vytvořte adresář pro sestavení (např. `build`) a spusťte v něm CMake pro konfiguraci projektu. Příklad pro Visual Studio:

   ```
   mkdir build
   cd build
   cmake .. -G "Visual Studio 17 2022" -DUSE_CUDA=ON -DUSE_OPENMP=ON -DUSE_AVX2=ON
   ```

   Přepínače `-DUSE_CUDA`, `-DUSE_OPENMP`, `-DUSE_AVX2` lze nastavit na `ON` nebo `OFF` podle potřeby. Pro jiné generátory (např. Ninja) upravte parametr `-G`.
3. **Instalace Python závislostí (pokud je třeba):** Po úspěšné konfiguraci CMake lze spustit cíl pro instalaci Python závislostí:

   ```
   cmake --build . --target install_python_deps
   ```

4. **Kompilace projektu:** Spusťte kompilaci hlavního projektu:

   ```
   cmake --build . --config Release
   ```

   (Nahraďte `Release` za `Debug` pro ladicí sestavení).
5. **Spuštění aplikace:** Po úspěšné kompilaci se spustitelný soubor `MinimalQtProject2.exe` (nebo odpovídající název na jiných platformách) nachází ve výstupním adresáři (např. `build/Release/`). Ujistěte se, že adresář s Python skripty (`python/`) je dostupný v cestě, odkud se aplikace spouští, nebo že je cesta k nim správně nakonfigurována (např. pomocí `PYTHON_MODULE_PATH` v CMake nebo kopírováním skriptů k exe).

Tento postup by měl umožnit úspěšné sestavení a spuštění aplikace na kompatibilním systému.

B Druhá příloha Visual Studio

Tato příloha poskytuje přehled organizace zdrojových kódů projektu a popisuje kroky a konfigurace potřebné pro jeho sestavení ve vývojovém prostředí Microsoft Visual Studio.

B.1 Struktura adresářů a klíčových souborů projektu

Projekt je organizován v rámci řešení Visual Studia (`complete build.sln`). Klíčové soubory a adresáře zahrnují:

- Kořenový adresář (D:\):
  - `complete build.sln`: Soubor řešení Visual Studia.
  - `visualize_grid_plotly.py`: Python skript pro vizualizaci (pravděpodobně používaný samostatně pro analýzu).
  - `.editorconfig`: Konfigurační soubor pro editory kódu.
- Adresář projektu `/complete build/`:
  - `complete build.vcxproj`: Hlavní soubor projektu MSVC obsahující nastavení kompilace a linkeru.
  - `complete build.vcxproj.filters`: Definuje strukturu souborů v Průzkumníku řešení Visual Studia.
  - Zdrojové soubory (`.cpp`, `.hpp`, `.h`, `.cu`, `.cuh`): Všechny C++, CUDA a hlavičkové soubory jsou umístěny přímo v tomto adresáři. Organizace kódu je primárně logická (dle názvů souborů) spíše než fyzická (v podadresářích).
    - **Mapové moduly:** Např. `MapProcessor.cpp/hpp`, `ElevationFetcherPy.cpp/hpp`, `GeoRefScanner.cpp/hpp`, `WaypointExtractor.cpp/hpp`.
    - **Pathfinding algoritmy (CPU):** Např. `AStarToblerSampled.cpp/hpp`, `ThetaStarToblerSampled.cpp/hpp`,`LazyThetaStarToblerSampled.cpp/hpp`.
    - **Pathfinding algoritmy (GPU wrappery a kernely):** Např. `AStarGPU.cpp/hpp` + `AStarKernels.cu`, `DeltaSteppingGPU.cpp/hpp` + `DeltaSteppingKernels.cu`, `HADS_GPU.cpp/hpp` + `HADSKernels.cu`. Hlavičkový soubor `PathfindingUtilsGPU.cuh` definuje device funkce a struktury.
    - **Vstup/Výstup:** `PathSaver.cpp/hpp`, `tinyxml2.cpp/h`.
    - **GUI (předpoklad, pokud by bylo součástí tohoto projektu):** Pokud by GUI bylo součástí tohoto projektu, byly by zde soubory jako `main_window.cpp/hpp` a hlavní `main.cpp` (nebo `complete build.cpp` jako vstupní bod).
    - **Externí knihovny:** `httplib.h`, `json.hpp`.
  - `elevation_logic.py`: Python skript pro získávání výškových dat, volaný z C++.
  - `/x64/`: Adresář obsahující výstupní soubory sestavení (pro Debug a Release konfigurace), včetně `.exe`, `.pdb`, `.obj` souborů a logů kompilace.
  - `/__pycache__/`: Cache pro Python skripty.

- Adresář `/x64/`: (Mimo adresář projektu, ale relevantní pro výstup) Obsahuje finální spustitelné soubory pro Debug a Release konfigurace.

Výpis také ukazuje adresář `/.vs/`, který obsahuje dočasné soubory Visual Studia, a adresáře `/out/build/debug(release)/`, které by odpovídaly výstupu CMake, pokud by byl použit. V tomto případě je však primární struktura dána řešením Visual Studia.

B.2 Proces sestavení ve Visual Studiu

Aplikace se sestavuje jako standardní projekt C++/CUDA ve vývojovém prostředí Microsoft Visual Studio.

B.2.1 Předpoklady pro sestavení

- **Microsoft Visual Studio:** Verze 2022 (nebo kompatibilní, která podporuje dodaný `.sln/.vcxproj`). Je nutné mít nainstalované komponenty pro vývoj v C++ a ideálně i podporu pro CMake, pokud by se projekt v budoucnu migroval.
- **NVIDIA CUDA Toolkit:** Verze 12.5 (nebo verze specifikovaná v nastavení projektu Visual Studia). Musí být správně integrován s Visual Studiem (typicky se děje automaticky při instalaci CUDA Toolkitu).
- **Ovladače NVIDIA GPU:** Aktuální ovladače podporující verzi CUDA Toolkitu.
- **Python:** Nainstalovaná verze Pythonu 3.9 (nebo kompatibilní) a přidaná do systémové cesty (PATH).
- **Python knihovny:** Nezbytné Python knihovny, zejména `pybind11`, `requests` a `pyproj`. Tyto je třeba nainstalovat do použitého Python prostředí, např. pomocí `pip`:

```
pip install pybind11 requests pyproj
```

- **Knihovna TinyXML2:** Zdrojové soubory (`tinyxml2.cpp`, `tinyxml2.h`) jsou přímo součástí projektu.
- **(Volitelně) Qt Framework:** Pokud by GUI bylo součástí tohoto projektu (což z výpisu souborů přímo v 'complete build' adresáři není zřejmé, ale bylo zmíněno dříve), byly by potřeba i vývojové knihovny Qt. Pokud je 'complete build.cpp' konzolová aplikace, Qt není nutné.

B.2.2 Konfigurace projektu ve Visual Studiu

Soubor `complete build.vcxproj` obsahuje všechna potřebná nastavení pro kompilaci C++ a CUDA kódů. Klíčové aspekty konfigurace (kontrolovatelné ve vlastnostech projektu ve VS) zahrnují:

- **C++ Standard:** Nastaven na C++17 nebo novější.
- **CUDA C++:** Nastavení pro NVIDIA CUDA Compiler (NVCC), včetně cílových architektur GPU (např. compute_52,sm_52;compute_60,sm_60;...).

- **Include adresáře:** Cesty k hlavičkovým souborům projektu, CUDA Toolkitu, Pythonu a Pybind11.
- **Knihovní adresáře a knihovny:** Cesty k CUDA knihovnám a Python knihovnám, a jejich linkování.
- **Podpora OpenMP:** Pokud je využívána, je povolena v nastavení kompilátoru (např. `/openmp` pro MSVC).
- **Předprocesorové definice:** Například definice jako `USE_CUDA` (pokud je použita pro podmíněnou kompilaci).

B.2.3 Postup sestavení

1. Otevřete soubor řešení `complete build.sln` v Microsoft Visual Studiu.
2. Ujistěte se, že jsou splněny všechny předpoklady, zejména správná instalace a konfigurace CUDA Toolkitu a Pythonu (včetně Pybind11 a dalších knihoven).
3. Vyberte požadovanou konfiguraci sestavení (např. `Debug` nebo `Release`) a cílovou platformu (např. `x64`).
4. Spusťte sestavení projektu (typicky klávesou F7 nebo přes menu Build -> Build Solution).
5. Visual Studio provede kompilaci C++ souborů pomocí MSVC a `.cu` souborů pomocí NVCC, následně provede linkování a vytvoří spustitelný soubor (např. `x64/Release/complete build.exe`).
6. Pro spuštění aplikace se ujistěte, že Python skript `elevation_logic.py` je ve stejném adresáři jako spustitelný soubor, nebo že je cesta k němu správně nastavena v C++ kódu, který jej volá.

V případě problémů se sestavením je nutné zkontrolovat cesty k SDK a knihovnám v nastavení projektu ve Visual Studiu a logy kompilátoru pro identifikaci konkrétních chyb.