ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE FAKULTA STAVEBNÍ KATEDRA GEOMATIKY					
Název předmětu: **Geoinformatika**					
Úloha: U3	**Název úlohy:** Nejkratší cesta grafem				
Akademický rok: 2024/2025	**Semestr:** zimní	**Studijní skupina:** 102C	**Vypracoval:** Michal Kovář Filip Roučka Magdaléna Soukupová	**Datum:** 3. 12. 2024	**Klasifikace:**

1 Zadání

Implementujte Dijkstra algoritmus pro nalezení nejkratší cesty mezi dvěma uzly grafu. Vstupní data budou představována silniční sítí doplněnou vybranými sídly.
Otestujte různé varianty volby ohodnocení hran grafu tak, aby nalezená cesta měla:

- nejkratší Eukleidovskou vzdálenost,
- nejmenší transportní čas (2 varianty).

Ve vybranémm GIS konvertujte podkladová data do grafové reprezentace představované neorientovaným grafem. Pro druhou variantu optimální cesty navrhněte vhodnou metriku, která zohledňuje rozdílnou dobu jízdy na různých typech komunikací dle jejich návrhové rychlosti a klikatosti. Využijte např. poměr délky polylinie $\{p_i\}_{i=1}^{n}$ a vzdálenosti koncových bodů polylinie

$$f = \frac{l(P)}{\|p_1 - p_n\|_2}.$$

Každou z variant otestujte pro dvě různé cesty. Výsledky umístěte do tabulky, vlastní cesty vizualizujte. Dosažené výsledky porovnejte s vybraným navigačním SW.

2 Bonus

- *Rešení úlohy pro grafy se záporným ohodnocením.*
- *Nalezení nejkratších cest mezi všemi dvojicemi uzlů.*
- *Nalezení minimální kostry* **Kruskal**.
- *Nalezení minimální kostry* **Primel**.
- *Využití heuristiky* **Weighted Union**.
- *Využití heuristiky* **Path Compression**.

3 Popis problému

Graf je matematická datová struktura popisující vztahy mezi objekty. Skládá se z množiny uzlů (`U`) a množiny hran (`H`), které spojují dvojice uzlů. Graf slouží jako topologický model reality, kde je prostorová informace méně důležitá než vzájemné vztahy mezi jednotlivými prvky. Grafy nacházejí široké uplatnění v různých oblastech, jako je analýza dopravních sítí, řešení logistických problémů, optimalizace tras, plánování, navigace, zajištění propustnosti sítí a další [1][2].

3.1 Typy grafů podle orientace hran

Grafy lze rozdělit podle orientace hran na následující typy:

- **Neorientované grafy**: Hrany nemají směr. Jestliže existuje hrana mezi uzly `A` a `B`, vztah mezi těmito uzly je symetrický a nezáleží na jejich pořadí[1].
- **Orientované grafy**: Každá hrana má definovaný směr, což znamená, že spojuje dvojici uzlů v určitém pořadí. Pokud existuje například hrana z uzlu `A` do uzlu `B`, nevyplývá z toho existence hrany v opačném směru[1].
- **Částečně orientované grafy**: Tyto grafy kombinují vlastnosti orientovaných a neorientovaných grafů. Některé hrany mohou být orientované, zatímco jiné zůstávají neorientované[1].

3.2 Typy grafů podle ohodnocení hran

Grafy se dále rozlišují na základě ohodnocení hran, které vyjadřuje jejich „náročnost" na průchod:

- **Neohodnocené grafy**: Všechny hrany mají stejné ohodnocení (například jednotkové).
- **Ohodnocené grafy**: Hrany mají přiřazeny různé ohodnocení, které mohou reprezentovat například vzdálenost, časovou náročnost, energetickou spotřebu nebo jiné charakteristiky.

3.3 Typické problémy řešené na grafech

Grafy poskytují základ pro řešení mnoha úloh, mezi nejčastější patří:

1. **Hledání nejkratší cesty**: Cílem je nalézt cestu, která minimalizuje součet ohodnocení hran na této cestě [2].

2. **Hledání minimální kostry grafu**: Jedná se o podgraf obsahující všechny uzly původního grafu, který je souvislý, bez kružnic, a jehož součet ohodnocení hran je minimální [2].

4 Popis metod

4.1 Nejkratší cesta grafem

Jedná se o postup, který určí cestu s nejmenší sumou ohodnocení z uzlu `A` do `B`. Pro určení nejkratší cesty grafem byla vytvořena třída `ShortestPath`, která je inicializována grafem reprezentovaným slovníkem. Klíče tohoto slovníku odpovídají uzlům grafu a hodnoty jsou další slovníky, které popisují sousedy každého uzlu a ohodnocení hran mezi nimi. Sousedé jsou uloženi jako klíče, přičemž ohodnocení hran mezi uzly jsou přiřazeny jako hodnoty.
Metody třídy `ShortestPath` jsou popsány v příloze A. Metody třídy `ShortestPath` obsahuje následující metody:

- `shortest_cost_path`
  Je metoda, která slouží k určení nejkratší cesty mezi dvěma body. Použije se v případě kdy nevíme zda se v grafu nachází hrana se záporným ohodnocením. V případě že se v grafu nachází hrana se záporným ohodnocením, použije se metoda `bellman_ford`, v případě že graf neobsahuje zápornou hranu použije se metoda `dijkstra`.
  Vstup : graf G, startovní uzel A, koncový uzel B
  Výstup : seznam předchůdců p, celková ohodnocení cesty $d[B]$

- `dijkstra`
  Je metoda, která využívá k určení nejkratší cesty Dijkstrův algoritmus. Použije se v případě kdy víme že v grafu se nenachází hrana se záporným ohodnocením. Tento algoritmus využívá postupného zpřesňování odhadu nejkratší délky od výchozího uzlu s do cílového uzlu k. Hodnota $d[v]$ je aktuální odhad nejkratší vzdálenosti k uzlu v, hodnota $w[u][v]$ představuje ohodnocení hrany (u, v), $d[u]$ je aktuální odhad nejkratší vzdálenosti k uzlu u. Pokud $d[u] + w[u][v] < d[v]$, hodnota $d[u] + w[u][v]$ představuje nový odhad nejkratší vzdálenosti $d[v]$. Při relaxaci je v každém kroku vybírán takový vrchol u, který má nejmenší hodnotu $d[u]$[3].
  Vstup : graf G, startovní uzel A, koncový uzel B
  Výstup : seznam předchůdců p, celková ohodnocení cesty $d[B]$

- `bellman_ford`
  Je metoda, která využívá k určení nejkratší cesty Bellmanův–Fordův algoritmus. Použije se v případě kdy víme že v grafu se nachází hrana se záporným ohodnocením. Algoritmus pracuje iterativně, postupným zpřesňováním odhadu nejkratší vzdálenosti od výchozího uzlu s ke všem ostatním uzlům v grafu. Hodnota d[v] reprezentuje aktuální odhad nejkratší vzdálenosti k uzlu v, hodnota $w[u][v]$ pak váhu hrany (u, v), zatímco $d[u]$ je aktuální odhad nejkratší vzdálenosti k uzlu u. Pro každý uzel u a jeho souseda v platí pravidlo relaxace: pokud $d[u] + w[u][v] < d[v]$, hodnota $d[v]$ se aktualizuje na $d[v] = d[u] + w[u][v]$. Tento proces probíhá opakovaně pro všechny hrany grafu. Algoritmus vyžaduje $|V| - 1$ iterací, kde $|V|$ je počet uzlů v grafu, aby zajistil, že všechny nejkratší cesty budou nalezeny. V případě existence záporných cyklů v grafu dokáže Bellman-Ford algoritmus tyto cykly detekovat. Pokud po $|V| - 1$ iteracích stále existuje možnost relaxace některé hrany, graf obsahuje záporný cyklus.
  Vstup : graf G, startovní uzel A, koncový uzel B
  Výstup : seznam předchůdců p, celková ohodnocení cesty $d[B]$

4.2 Minimální kostra grafu:

Jedná se o postup, který určí minimální kostru pro zadaný graf. Minimální kostra grafu je podgraf, který obsahuje všechny uzly původního grafu a co nejmenší množství hran, tak aby byl stále souvislý a součet

vah těchto hran byl minimální. Pro určení minimální kostry grafu byla vytvořena třída `MST`. Třída `MST` je inicializována grafem, který je reprezentován slovníkem, kde klíče odpovídají uzlům grafu a hodnoty jsou další slovníky popisující sousedy každého uzlu a váhy hran mezi nimi. Sousedé jsou uloženi jako klíče, přičemž váhy hran mezi uzly jsou přiřazeny jako hodnoty.
Metody třídy `MST` jsou popsány v příloze B. Metody třídy `MST` obsahuje následující metody:

- `prim`
  Je metoda, která slouží k určení minimální kostry grafu za pomoci Jarníkova algoritmu. Kostra je reprezentována pomocí prioritní fronty Q setříděnou podle ohodnocení. Výpočet začíná v libovolném uzlu grafu. V prvním kroku jsou do seznamu uloženy všechny uzly a jsou jim nastaveny nekonečné vzdálenosti od podstromu. Dokud není Q prázdná opakovaně je vybírán neuzavřený uzel v incidující s uzlem u existující kostry takový, že jeho ohodnocení $w(u, v) = min$. Uzel v je přidán do kostry a hodnoty jeho předchůdce je uložena do p[3]
  Vstup : graf G
  Výstup : seznam hran $(\text{start}, \text{end}, \text{weight})$ T, celková ohodnocení wt

- `boruvka`
  Je metoda, která slouží k určení minimální kostry grafu za pomoci Borůvkova algoritmu. Pro zrychlení procesu je použita heuristika Weighted Union a heuristika Path Compresion. V prvním kroku každému uzlu x je vytvořena podmnožina $X = \{x\}$. Postupně jsou procházeny jednotlivé hrany h (koncové uzly u, v) a jsou sjednocovány jednotlivé podmnožiny U, V příslušející těmto uzlům. Množiny U, V jsou disjunktní. Pokud se oba uzly u, v nacházejí v jiných podmnožinách, výsledkem sjednocení je množina $U = U \bigcap V$, množina V zaniká [3].
  Vstup : graf G
  Výstup : seznam hran $(\text{start}, \text{end}, \text{weight})$ T, celková ohodnocení wt

- `plot_mst`
  Je metoda, která slouží k vykreslení minimální kostry. Algoritmus přetváří seznam hran stromu $(\text{start}, \text{end}, \text{weight})$ T a seznamu souřadnic C na sadu souřadnic vhodných pro vykreslení. Použitím přerušení čar (`None`) zajišťuje, že hrany jsou vykresleny odděleně, a poskytuje čistou vizualizaci struktury stromu s použitím rovnocenného měřítka a mřížky.
  Vstup : seznam hran $(\text{start}, \text{end}, \text{weight})$ T, souřadnice uzlů C, styl linie $line$

4.3 Vyhledání cesty v grafu:

Pro vyhledávání a práci s grafy byla vytvořena třída `GraphPathFinder`, která dědí vlastnosti tříd `ShortestPath` a `MST`. Třída `GraphPathFinder` je inicializována grafem, který je reprezentován slovníkem, kde klíče odpovídají uzlům grafu a hodnoty jsou další slovníky popisující sousedy každého uzlu a váhy hran mezi nimi. Sousedé jsou uloženi jako klíče, přičemž váhy hran mezi uzly jsou přiřazeny jako hodnoty.
Metody třídy `GraphPathFinder` jsou popsány v příloze C. Metody třídy `GraphPathFinder` obsahuje následující metody:

- `DFS`
  Je metoda, která slouží k prohledání grafu do hloubky. Z posledního uzlu u jsou postupně prohledávány všechny hrany na dosud nenavštívené uzly v. V případě více incidujících uzlů bude vybrán uzel v s nejnižším číslem. Z něj bude prohledávání probíhat stejným způsobem. Pokud žádná žádná taková hrana neexistuje označí algoritmus uzel u za uzavřený a vrací se na, ze kterého se do uzlu u dostal. V tomto uzlu opět prohledává hrany na dosud nenavštívené sousední uzly, po jejich prohledávání uzel uzavře a vrací se na uzel, ze kterého se do tohoto uzlu dostal [3].
  Vstup : graf G, startovní uzel A
  Výstup : seznam předchůdců p

- `BFS`
  Je metoda, která slouží k prohledání grafu do šířky. Všechny uzly jsou označeny jako nové, nemají žádného předchůdce a jsou přidány do do fronty. Všem uzlům s výjimkou počátečního uzlu s nastavíme vzdálenost od počátku inf, uzlu s hodnotou 0. Je vybrán první otevřený uzel u a všem jeho novým sousedům v je určena "vzdálenost"od počátku $d(v) = d(u)+1$, $p(v) = u$. Po prohledání všech nových uzlů v incidujících s uzlem u označen uzel u za uzavřený a je odebrán z fronty. Ve frontě jsou obsaženy uzly, které ještě nebyly označeny jako uzavřené. V dalším kroku je zvolen další

první otevřený uzel z fronty a pokračuje se stejným způsobem [3].
Vstup : graf G, startovní uzel A
Výstup : seznam předchůdců p

- `all_shortest_paths`
  Je metoda, která slouží k vyhledání nejkratší cesty mezi všemi uzly. Algoritmus implementuje výpočet všech nejkratších cest mezi všemi dvojicemi uzlů v grafu. Používá metody `shortest_cost_path` pro výpočet nákladů a `rec_path` pro rekonstrukci cest. Výsledky jsou uloženy do slovníku paths, kde každý záznam obsahuje cestu a její náklady.
  Vstup : graf G
  Výstup : slovník obsahující dvojici bodů a nejkratší cestu mezi nimi *paths*

- `plot_graph`
  Je metoda, která slouží k vykreslení celého grafu. Algoritmus vykresluje graf, kde uzly jsou zobrazeny jako body na základě jejich souřadnic a jsou označeny jejich identifikátory. Hrany mezi uzly jsou zobrazeny jako čáry mezi sousedními uzly. Na konci je přidána mřížka pro lepší přehlednost a popisky os pro orientaci.
  Vstup : graf G, souřadnice uzlů C, styl linie *line*, barva bodů uzlů points, velikost uzlů a point_size

- `plot_path`
  Je metoda, která slouží k vykreslení cesty. Algoritmus vykresluje cestu mezi uzly grafu na základě jejich souřadnic. Pro každý uzel na cestě získá jeho souřadnice z mapy C, přidá je do seznamů a následně vykreslí čáru spojující tyto uzly.
  Vstup : souřadnice uzlů C, cesta P a styl linie *linie*

- `plot_node_names`
  Je metoda, která slouží k vykreslení názvů uzlů. Algoritmus vykresluje uzly jako černé body a přidává k nim textové popisky. Pro lepší čitelnost textu na různých pozadích je kolem textu vytvořen efekt halo pomocí několika bílých textových vrstev s různými posuny. Nakonec je na vrcholu všech textových vrstev vykreslen samotný název uzlu v černé barvě.
  Vstup : souřadnice a názvy uzlů *node_names*

- `rec_red_nodes`
  Je metoda, která slouží k vykreslení specifických uzlů červeně. Algoritmus prochází seznam uzlů *nodes* a pro každý uzel, který je obsažen v mapě souřadnic C, vykreslí červený bod na pozici tohoto uzlu.
  Vstup : seznam uzlů *nodes*, seznam souřadnic C, velikost bodu *size*

- `rec_path`
  Je metoda, která slouží ke zpětnému vytvoření cesty mezi dvěma body. Algoritmus slouží k rekonstrukci cesty mezi dvěma uzly v grafu, kde jsou rodičovské informace uchovávány v poli P. Rekonstrukce začíná od cílového uzlu a postupně sleduje rodiče až k výchozímu uzlu. Pokud se algoritmus vrátí k výchozímu uzlu, vrátí cestu, jinak vypíše chybu, že cesta je neplatná.
  Vstup : startovní uzel A, koncový uzel B, seznam předchůdců p
  Výstup : cesta *path*

- metody třídy `ShortestPath`

- metody třídy `MST`

5 Postup

Úloha byla implementována v programovacím jazyce Python. Hlavní program byl vytvořen ve dvou verzích. První verze, `main.py`, umožňuje uživateli zadat vstupní data pro výpočet nejkratší cesty mezi dvěma obcemi prostřednictvím příkazové řádky. Druhá verze, `main_gui.py`, nabízí jednoduché grafické uživatelské rozhraní (GUI), které umožňuje interaktivní výběr obcí z rozbalovacích seznamů (comboboxů), mezi kterými má být nalezena nejkratší cesta.

5.1 Potřebné moduly

Pro tvorbu programu byly využity následující knihovny a moduly:

- `tkinter` – pro vytvoření grafického uživatelského rozhraní v druhé verzi programu.
- `matplotlib` – pro vizualizaci grafu a zobrazení výsledků výpočtu.
- `graph_reader` – vlastní modul pro načítání grafu a souvisejících dat (uzlů a obcí).
- `graph_path_finder` – vlastní modul, který implementuje zadané algoritmy.

5.2 Načtení dat

Nejprve bylo potřeba načíst data o grafu a souřadnicích uzlů. Tato data jsou uložena v textových souborech, které jsou následně zpracovány pomocí funkcí `read_graph` a `read_nodes_names`:

```
file = './U3/data/graph_unweighted.txt'
municipalities_file = './U3/data/municipalities_nearest_nodes.txt'
G, C = read_graph(file)
municipalities = read_nodes_names(municipalities_file)
```

Funkce `read_graph` načte graf a souřadnice uzlů, přičemž data jsou uložena do proměnných `G` (graf) a `C` (souřadnice uzlů). Funkce `read_nodes_names` načte slovník názvů obcí a jejich souřadnic.

5.3 Přiřazení uzlů k obcím

Pro uživatelsky přívětivé vyhledávání cesty bylo nutné přiřadit každé obci odpovídající uzel grafu. Tento krok je realizován pomocí slovníku `municipality_to_node`, který mapuje název obce na uzel:

```
municipality_to_node = {}
for municipality, (x, y) in municipalities.items():
    for node, coords in C.items():
        if coords == [x, y]:
            municipality_to_node[municipality] = node
```

Tento kód porovnává souřadnice obcí s odpovídajícími souřadnicemi uzlů a přiřazuje obcím jejich příslušné uzly v grafu.

5.4 Získání vstupu od uživatele

V obou verzích programu je uživatel požádán o zadání počátečního a koncového uzlu pro nalezení nejkratší cesty. Ve verzi s příkazovou řádkou je tento vstup získán pomocí funkce `input`, zatímco ve verzi s grafickým uživatelským rozhraním (GUI) je tento vstup získán prostřednictvím rozbalovacích nabídek, kde uživatel vybere obce z předem definovaného seznamu. Pokud jsou obě hodnoty validní, odpovídající uzly jsou získány z mapy `municipality_to_node`.

Příklad kódu pro verzi s příkazovou řádkou:

```
# Verze pro příkazovou řádku
start_node_name = input("Enter the starting node name: ")
end_node_name = input("Enter the ending node name: ")
```

Příklad kódu pro verzi s GUI:

```
# Verze pro GUI
start_combobox = ttk.Combobox(root, values=sorted(list(municipality_to_node.keys())))
end_combobox = ttk.Combobox(root, values=sorted(list(municipality_to_node.keys())))
start_combobox.grid(row=0, column=1, padx=10, pady=10)
end_combobox.grid(row=1, column=1, padx=10, pady=10)
```

5.5 Výpočty a algoritmy

Pro analýzu grafu a hledání cest mezi uzly byla v programu využita třída `GraphPathFinder`, která implementuje různé algoritmy pro prohledávání grafu a hledání cest. Mezi tyto algoritmy patří:

- **BFS (Breadth-First Search) a DFS (Depth-First Search):** Tyto algoritmy slouží k prohledání grafu. BFS používá prohledávání grafu do šířky, zatímco DFS se používá pro prohledávání grafu do hloubky.
- **Dijkstra a Bellman-Ford:** Tyto algoritmy jsou určeny k nalezení nejkratší cesty mezi dvěma uzly v grafu, přičemž Dijkstra je efektivní pro grafy s kladnými váhami, zatímco Bellman-Ford zvládá i grafy s negativními váhami.
- **Prim a Borůvka:** Tyto algoritmy slouží k nalezení minimální kosterního stromu grafu, což je podmnožina hran grafu, která spojuje všechny uzly s minimální váhou.

Pro každý z těchto algoritmů byl napsán kód, který provádí výpočty a zobrazuje výsledky uživateli. Příklad použití algoritmů je následující:

```
# Vytvoření instance pro práci s grafem
SP = GraphPathFinder(G)

# BFS a DFS pro počáteční uzel
bfs_path = SP.BFS(start_node)
dfs_path = SP.DFS(start_node)

# Hledání všech nejkratších cest
all_shortest_paths = SP.all_shortest_paths()

# Hledání nejkratší cesty mezi počátečním a koncovým uzlem (samo vybere vhodný algoritmus)
shortest_path, min_cost = SP.shortest_cost_path(start_node, end_node)
specific_path = SP.rec_path(start_node, end_node, shortest_path)

# Dijkstra's algorithm
dijkstra_path, dijkstra_weight = SP.dijkstra(start_node, end_node)

# Bellman-Ford algorithm
bellman_ford_path, bellman_ford_weight = SP.bellman_ford(start_node, end_node)

# Minimum spanning tree using Prim's algorithm
mst_prim, mst_weight_prim = SP.prim()

# Minimum spanning tree using Borůvka's algorithm
mst_boruvka, mst_weight_boruvka = SP.boruvka()
```

5.6 Vizualizace grafu

Pro vizualizaci grafu a výsledků výpočtů byla využita knihovna `matplotlib`. Před samotným vykreslením výsledků byly v programu připraveny metody pro vykreslování, které obsahují bloky kódu, jež by se v kódu často opakovaly. Tyto metody jsou součástí třídy `GraphPathFinder` a umožňují efektivní zobrazení různých aspektů grafu, jako jsou uzly, cesty mezi uzly nebo minimální kostery, a to pomocí metod `plot_graph`, `plot_red_nodes`, `plot_path`, `plot_mst` a `plot_node_names`. Výsledky vykreslení jsou zobrazeny ve třech oknech, kde každé okno obsahuje jiný výstup (například graf s nejkratší cestou nebo s minimální kostru podle Primova či Borůvkova algoritmu). Ukázka kódu pro vykreslení grafu a výsledků vypadá následovně:

```
# Vykreslení grafu
SP.plot_graph(C)

# Zvýraznění počátečního a koncového uzlu
SP.plot_red_nodes({start_node, end_node}, C)

# Vykreslení cesty, pokud existuje
if path:
    SP.plot_path(path, C)

# Zobrazení názvů uzlů
SP.plot_node_names(municipalities)

# Vykreslení minimální kostry grafu
SP.plot_mst(C, mst, line='r-')

# Nastavení názvu grafu
plt.title("Title")

# Zobrazení grafu
plt.show()
```

6 Vstupní data

Vstupní data pocházejí z geodatabáze ArcČR 500 Verze 3.3, konkrétně z vrstev `silnice_2015` a `obce_body`.[4]

6.1 Načtení dat

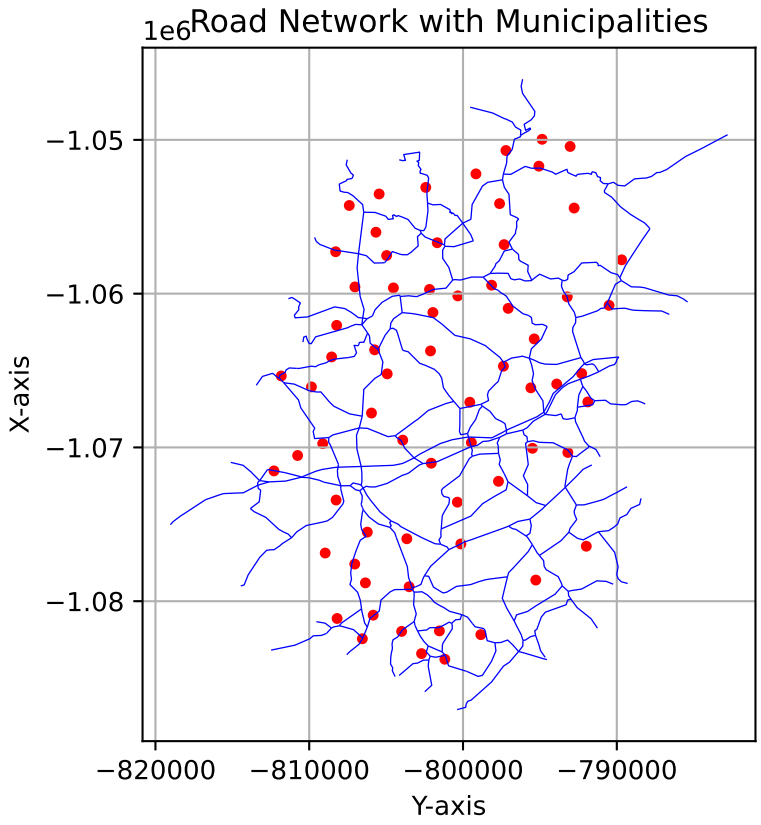
Tyto vrstvy byly importovány do softwaru QGIS, kde byl proveden výběr dat pro území okresu Rokycany a následný export dat do formátu GeoPackage.

- `roads.gpkg`
- `municipalities.gpkg`

Tyto soubory byly dále načteny do Python skriptu `convert_to_graph.py`, kde byla data následně zpracována za použití knihoven `GeoPandas` a `Shapely` pro práci s prostorovými daty, a `Matplotlib` pro vizualizaci.

6.2 Vizualizace dat

Pro ověření zda byla data načtena v pořádku byly silnice a obce vizualizovány pomocí knihovny `Matplotlib`. Silniční síť byla vykreslena modře a obce červeně. Zde je ukázka silnic a obcí pro okres Rokycany:



Obrázek 1: Silnice a obce v okrese Rokycany sloužící pro tvorbu grafu

6.3 Výpočet dalších atributů silnic

K jednotlivým silnicím byly přidány následující atributy:

- **Délka silnice**: Délka každé silnice byla spočítána jako součet délek všech segmentů v geometrii silnice. Tento výpočet byl proveden použitím funkce `length` z knihovny `GeoPandas`, která vrací celkovou délku geometrie silnice. Tento atribut je uložen v sloupci `length`.
- **Rychlostní limit**: Každé silnici byl přiřazen rychlostní limit na základě její třídy (sloupec `TRIDA`). Tento atribut byl mapován podle předdefinovaných pravidel, kde dálnicím (`TRIDA` = 1) byl přiřazen rychlostní limit 130 km/h, rychlostním silnicím (`TRIDA` = 2) 110 km/h, silnicím 3. třídy (`TRIDA` = 3) 90 km/h, 4. třídě 70 km/h, 5. třídě 50 km/h a neevidovaným silnicím (`TRIDA` = 6) 30 km/h. Tento způsob přiřazování rychlostí neodpovídá přesně realitě, ale vzhledem k tomu, že data neobsahují informace o návrhové rychlosti, byl tento způsob zvolen jako nejvhodnější. Rychlostní limit je uložen ve sloupci `design_speed`.
- **Zakřivení silnice**: Zakřivení silnice bylo vypočteno jako poměr celkové délky silnice, která je uložena ve sloupci `length`, a přímé vzdálenosti mezi jejím počátečním a koncovým bodem. Tento výpočet slouží k určení, jak moc je silnice klikatá.

  Vzorec pro výpočet zakřivení je následující:

$$\texttt{curvature} = \frac{\texttt{length}}{\sqrt{(x_n - x_1)^2 + (y_n - y_1)^2}}$$

  kde:

  - (x_1, y_1) jsou souřadnice prvního bodu geometrie silnice.
  - (x_n, y_n) jsou souřadnice posledního bodu geometrie silnice.

Hodnota zakřivení poskytuje informaci o klikatost silnice:

  - Pokud je zakřivení rovno 1, znamená to, že silnice je přímá (její celková délka odpovídá přímé vzdálenosti mezi počátečním a koncovým bodem).
  - Čím vyšší je hodnota zakřivení, tím více je silnice klikatá.

  Výpočet zakřivení byl implementován ve funkci `calculate_curvature`, která iteruje přes všechny geometrie silnic v tabulce a provádí výpočet na základě výše uvedeného vzorce. Tento atribut byl následně uložen do nového sloupce `curvature`.

- **Náklady na cestu v závislosti na délce**: Tyto náklady byly spočítány jako inverzní hodnota délky silnice. Předpokládáme, že čím delší je silnice, tím vyšší jsou náklady. Tento výpočet je realizován ve sloupci `length_weight`:

$$\texttt{length\_weight} = \frac{1}{\texttt{length}}$$

  Tento parametr slouží k zohlednění nákladů na cestu podle délky silnice, přičemž kratší silnice mají nižší náklady.

- **Náklady na cestu v závislosti na rychlostním limitu**: Tyto náklady byly spočítány jako inverzní hodnota rychlostního limitu silnice. Předpokládá se, že silnice s vyšším rychlostním limitem umožňují rychlejší dopravu, čímž snižují náklady na cestu. Tento výpočet je implementován ve sloupci `speed_weight`:

$$\texttt{speed\_weight} = \frac{1}{\texttt{design\_speed}}$$

  Pro ověření byly některé vypočtené hodnoty kontrolně vypsány. Ukázka dat je uvedena v tabulce:

Tabulka 1: Atributy silnic pro tvorbu grafů

geometry	curvature	length_weight	speed_weight
MULTILINESTRING ((-807558, -1083474), ...)	1.0126	0.000661	0.014286
MULTILINESTRING ((-806520, -1082398), ...)	1.0000	0.001283	0.014286
MULTILINESTRING ((-806220, -1081679), ...)	1.1412	0.000230	0.014286
MULTILINESTRING ((-803307, -1079227), ...)	1.0000	0.011225	0.014286
MULTILINESTRING ((-802511, -1078346), ...)	1.0070	0.000426	0.014286

6.4 Zpracování silniční sítě a uzlů

Pro každý `MultiLineString` byly extrahovány souřadnice počátečních a koncových bodů všech segmentů pomocí funkce `extract_nodes_from_multilinestring`, čímž byla vytvořena sada uzlů.

6.5 Přiřazení uzlů k obcím

Každé obci byla přiřazena nejbližší silniční uzel na základě minimální Eukleidovské vzdálenosti. Tento výpočet byl realizován funkcí `compute_nearest_node`, která iteruje přes všechny obce a určuje nejbližší uzel pro jejich souřadnice. Výsledky byly uloženy do textového souboru `municipalities_nearest_nodes.txt`, který obsahuje název obce a souřadnice jejího nejbližšího uzlu. Jako oddělovač je využívána čárka jelikož některé obce mají víceslovné názvy a využití mezery jako oddělovače by vedlo k chybám. Ukázka prvních tří řádků souboru:

```
Skořice,-799379.4678585269,-1081458.7507021464
Kařízek,-791431.2708886713,-1066960.8780070543
Drahoňův Újezd,-797727.8870032951,-1059772.5634250343
```

6.6 Export do souborů grafů

Silniční síť byla následně převedena do formátu neorientovaného grafu, přičemž byly vytvořeny čtyři varianty reprezentace grafu podle různých kritérií vážení hran:

- Graf s váhami na základě délky (`graph_length_weight.txt`),
- Graf s váhami na základě zakřivení (`graph_curvature_weight.txt`),
- Graf s váhami na základě rychlosti (`graph_speed_weight.txt`),
- Nevážený graf s konstantními náklady (`graph_unweighted.txt`).

Každý z těchto souborů obsahuje informace o jednotlivých hranách grafu, konkrétně souřadnice počátečního a koncového uzlu, a odpovídající váhu hrany. Ukázka dat z jednoho souboru je uvedena níže:

```
-807558.2415553108 -1083474.283921726 -806520.2105997838 -1082398.5459269881 1
-806520.2105997838 -1082398.5459269881 -806220.5434165187 -1081679.2349191494 1
-806220.5434165187 -1081679.2349191494 -803307.7924209237 -1079227.170260936 1
```

6.7 Načtení dat z textového souboru

Pro načtení těchto dat z textového souboru do hlavního programu slouží `graph_reader.py`, který umožňuje načíst data do programu z `*.txt` souborů. Konkrétně obsahuje dvě funkce pro načítání dat z textových soborů a to:

6.7.1 Funkce pro načítání grafu

Funkce `read_graph` načítá data z textového souboru, který obsahuje informace o hranách grafu ve formátu:

```
x1 y1 x2 y2 w
```

kde $(x1, y1)$ a $(x2, y2)$ jsou souřadnice počátečního a koncového uzlu hrany a w je váha hrany.

Tato funkce nejprve načte všechny hrany, poté vytvoří mapu bodů na identifikátory uzlů a nakonec vygeneruje samotný graf ve formě slovníku, kde každé ID uzlu ukazuje na jeho sousedy spolu s váhami hran.

6.7.2 Funkce pro načítání názvů uzlů

Funkce `read_nodes_names` načítá data z textového souboru, který obsahuje názvy obcí a jejich souřadnice ve formátu:

```
name, x, y
```

Tato funkce vytváří slovník, kde klíčem je název obce a hodnotou je dvojice souřadnic (x, y), které reprezentují polohu dané obce. Tento slovník je následně využíván pro přiřazení každé obci nejbližšího uzlu v grafu.

```
from graph_reader import read_graph, read_nodes_names  # Import graph loading functions

# Path to the graph file (with weights or unweighted)
file = './U3/data/graph_unweighted.txt'

# Path to the municipality file
municipalities_file = './U3/data/municipalities_nearest_nodes.txt'

# Load graph and coordinates
G, C = read_graph(file)

# Load municipalities
municipalities = read_nodes_names(municipalities_file)
```

Konkrétně:

- **G (graf)**: Slovník, jehož klíče představují identifikátory uzlů, přičemž každý uzel je reprezentován jako klíč ve formě čísla (ID uzlu). Hodnoty tohoto slovníku jsou další slovníky, které pro každý uzel mapují jeho sousedy (id sousedních uzlů) na váhy hran mezi nimi. Váha hrany může být například délka cesty mezi těmito uzly nebo jiný parametr (např. zakřivení nebo rychlost).
- **C (souřadnice)**: Slovník, který mapuje každé ID uzlu na seznam obsahující souřadnice tohoto uzlu ve formátu $[x, y]$, kde x a y jsou prostorové souřadnice uzlu v daném referenčním systému.

Příklad struktury grafu `G` a souřadnic `C` může vypadat takto:

```
# Příklad reprezentace grafu
{
    1: {2: 5.0, 3: 10.0}, # Uzel 1 je spojen s uzlem 2 váhou 5.0 a uzlem 3 váhou 10.0
    2: {1: 5.0, 3: 2.0},  # Uzel 2 je spojen s uzlem 1 váhou 5.0 a uzlem 3 váhou 2.0
    3: {1: 10.0, 2: 2.0}  # Uzel 3 je spojen s uzlem 1 váhou 10.0 a uzlem 2 váhou 2.0
}

# Příklad souřadnic (C) pro každý uzel
{
    1: [-799379.4678585269, -1081458.7507021464],# Uzel 1 a jeho souřadnice
    2: [-791431.2708886713, -1066960.8780070543],# Uzel 2 a jeho souřadnice
    3: [-797727.8870032951, -1059772.5634250343] # Uzel 3 a jeho souřadnice
}
```

Kromě toho je v programu načítán soubor `municipalities_nearest_nodes.txt`, který obsahuje názvy obcí a souřadnice jejich nejbližších uzlů. Tento soubor je načítán funkcí `read_nodes_names`. Výsledkem je slovník, který mapuje názvy obcí na identifikátory uzlů grafu, na kterých se dané obce nacházejí. Struktura tohoto slovníku vypadá takto:

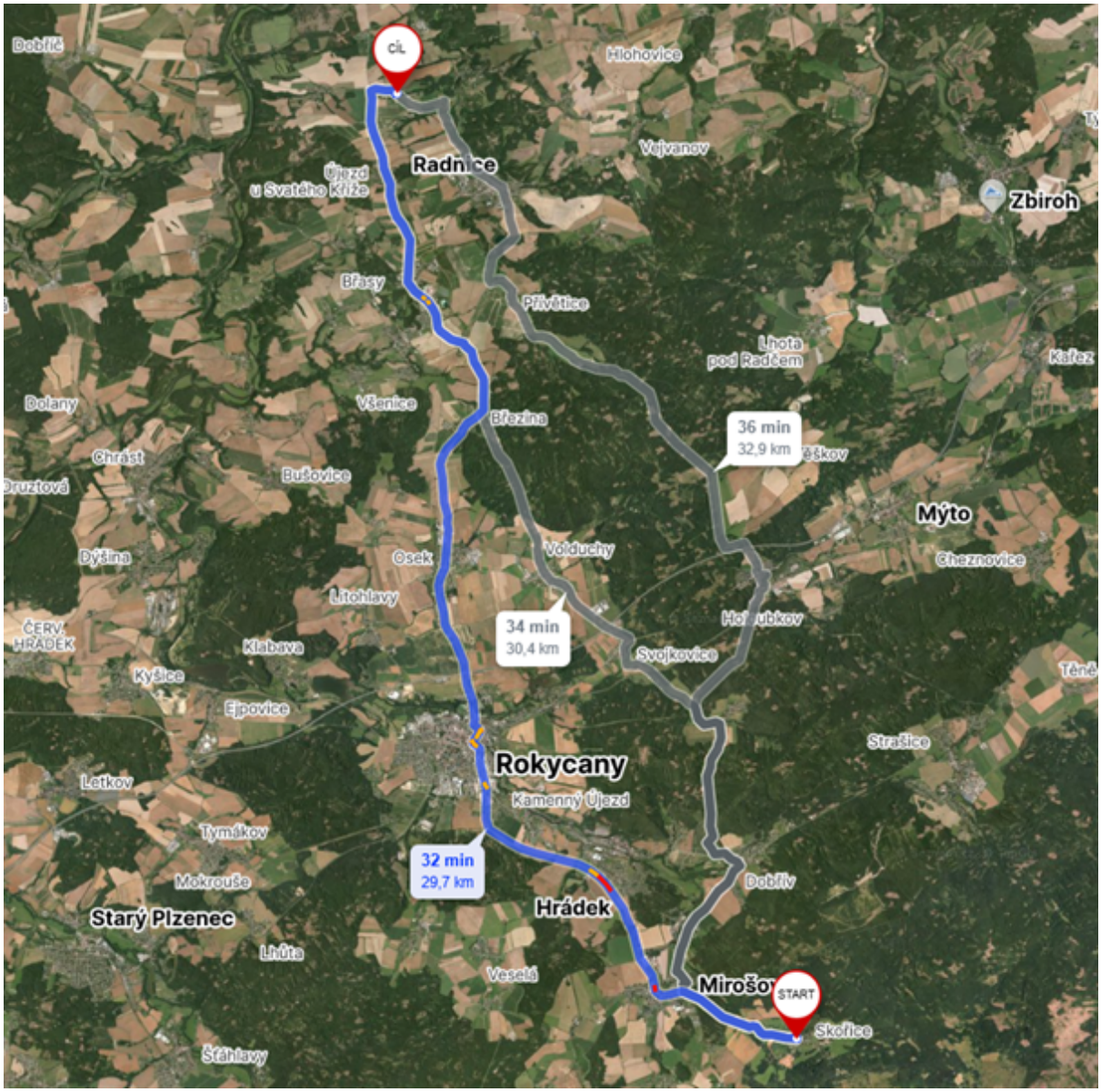
```
# Příklad slovníku názvů obcí a jejich nejbližších uzlů
{
    'Skořice': 1,  # Obec 'Skořice' je přiřazena k uzlu 1
    'Kařízek': 2,  # Obec 'Kařízek' je přiřazena k uzlu 2
    'Drahoňův Újezd': 3  # Obec 'Drahoňův Újezd' je přiřazena k uzlu 3
}
```

7 Výstupní data

7.1 Nejkratší cesta

7.1.1 Zvolená trasa pro testování: Skořice → Němčovice

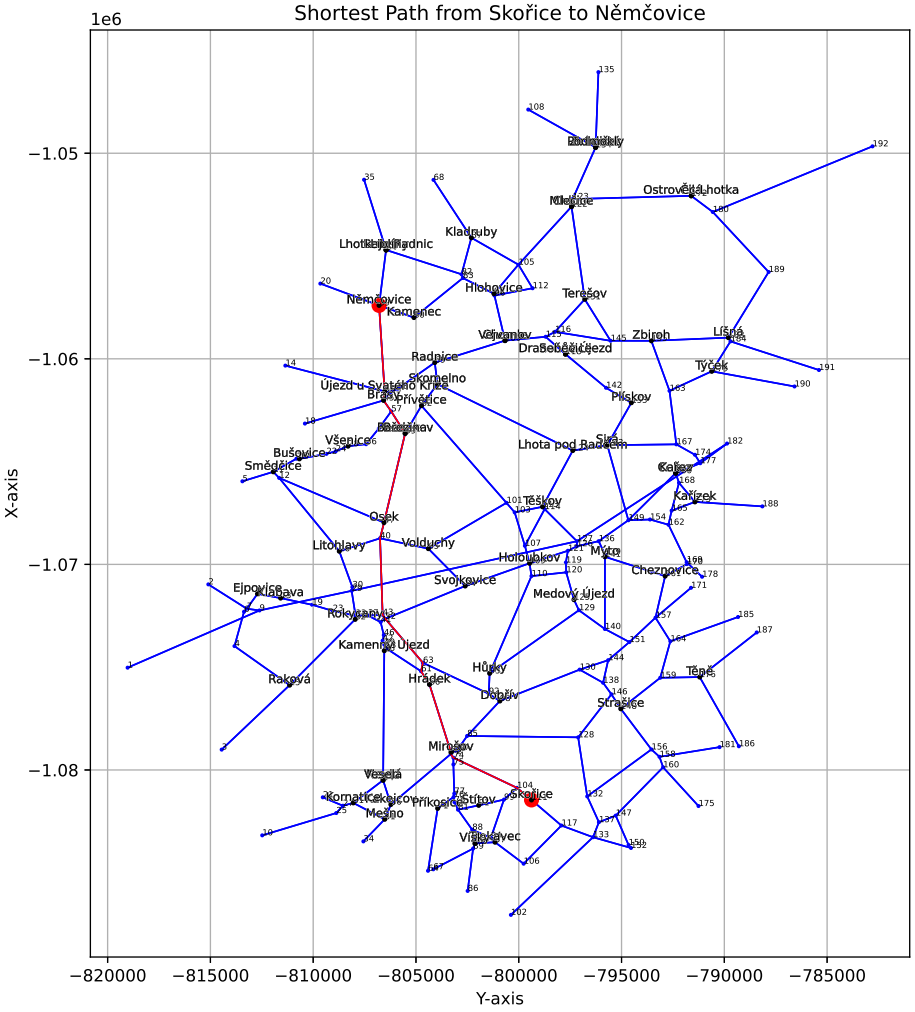
Nejkratší cesta podle Mapy.cz:



Obrázek 2: Nejkratší cesta podle Mapy.cz

Délka cesty: 29.7 km

Neohodnocený graf

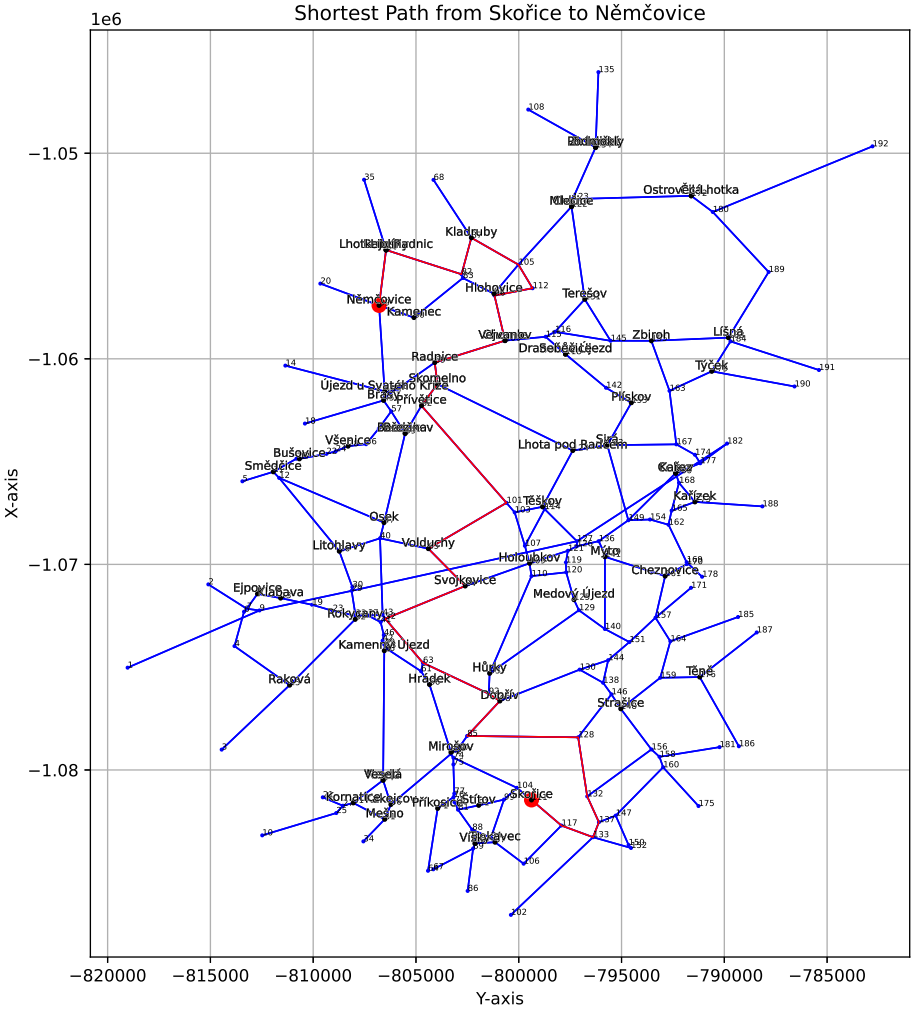


Obrázek 3: Nejkratší cesta v neohodnoceném grafu

Cesta z 111 do 39: [39, 50, 45, 57, 59, 47, 40, 43, 52, 63, 61, 66, 73, 72, 74, 104, 111] s váhou 16.0 (celkem prochází přes 16 hran)

Délka cesty: 29.3 km

Ohodnocený graf podle délky

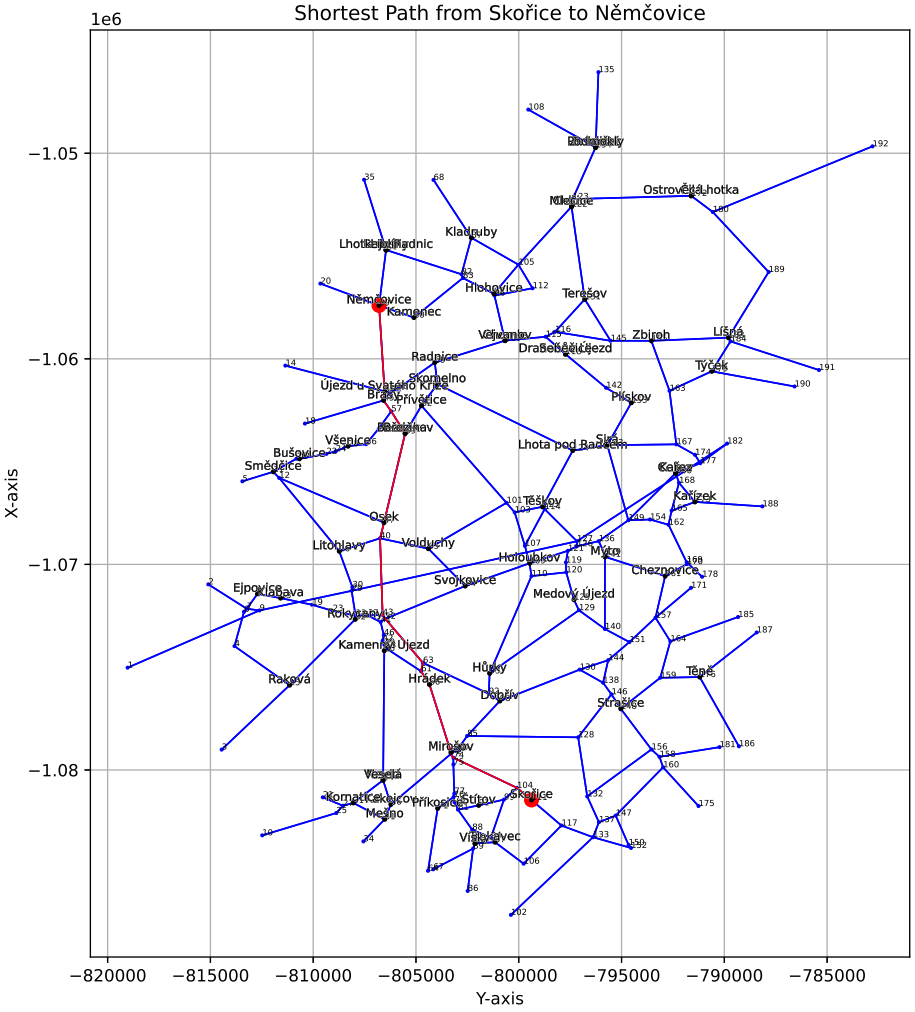


Obrázek 4: Nejkratší cesta v ohodnoceném grafu podle délky

Cesta z 111 do 39: [39, 54, 82, 87, 105, 112, 96, 100, 69, 70, 62, 101, 65, 84, 52, 63, 92, 98, 85, 128, 132, 137, 133, 117, 111] s váhou 0.011966

Délka cesty: 28.2 km

Ohodnocený graf podle návrhové rychlosti

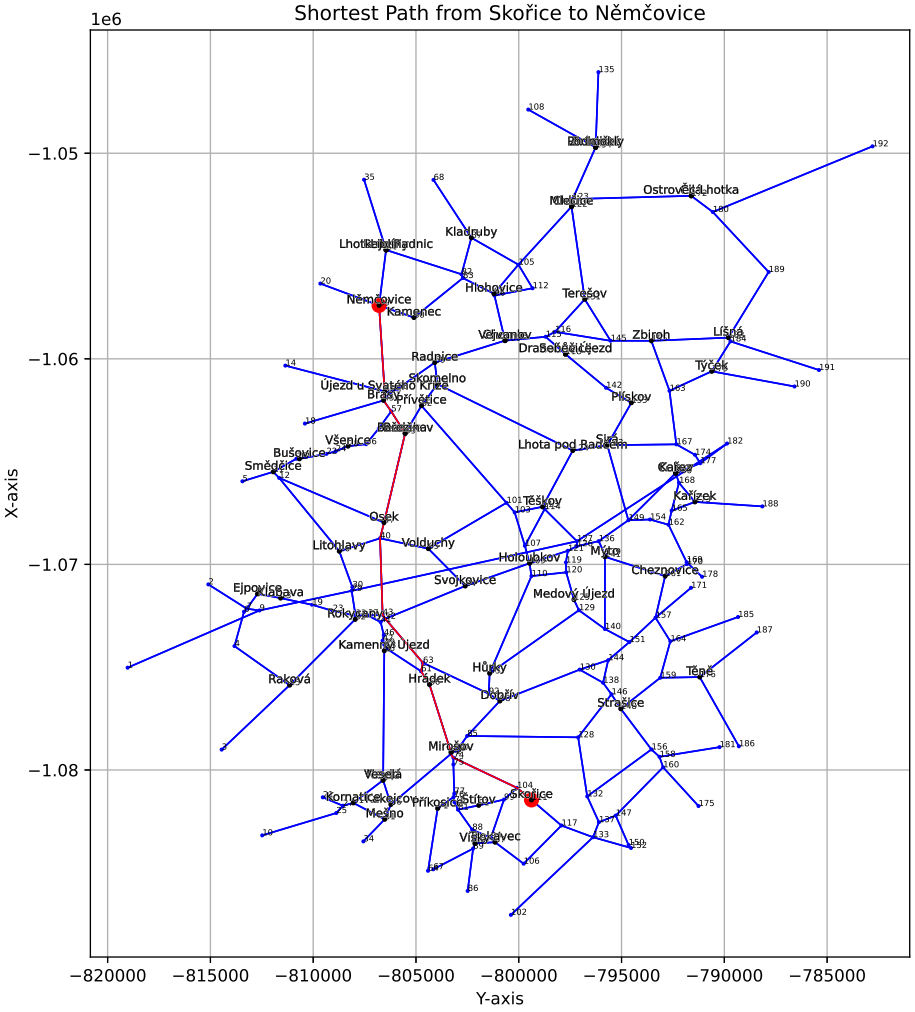


Obrázek 5: Nejkratší cesta v ohodnoceném grafu podle návrhové rychlosti

Cesta z 111 do 39: [39, 50, 45, 57, 59, 47, 40, 43, 52, 63, 61, 66, 73, 72, 74, 104, 111] s váhou 0.26857

Délka cesty: 29.3 km

Ohodnocený graf podle klikatosti



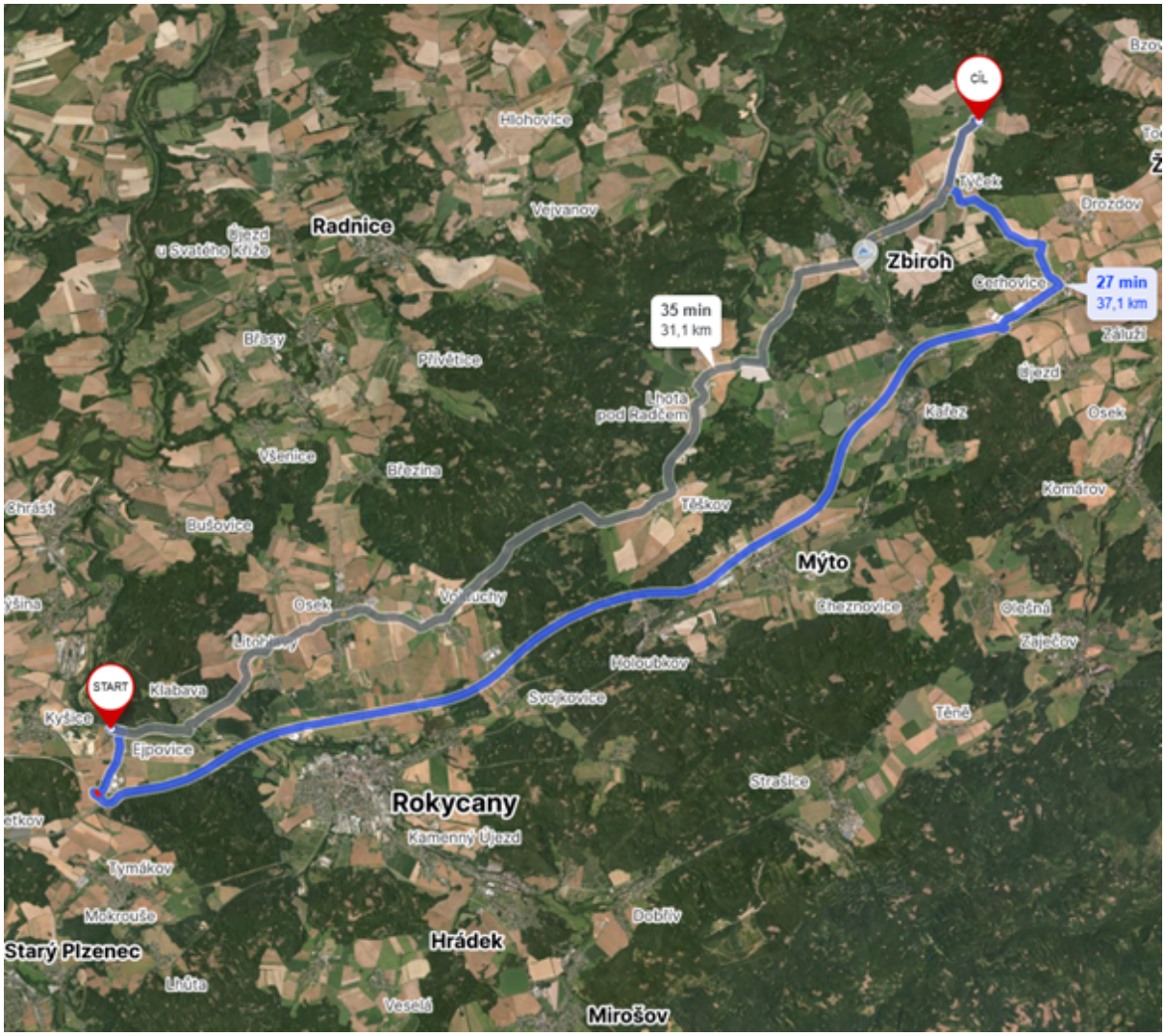
Obrázek 6: Nejkratší cesta v ohodnoceném grafu podle klikatosti

Cesta z 111 do 39: [39, 50, 45, 57, 59, 47, 40, 43, 52, 63, 61, 66, 73, 72, 74, 104, 111] s váhou 16.533

Délka cesty: 29.3 km

7.1.2 Zvolená trasa pro testování: Ejpovice → Líšná

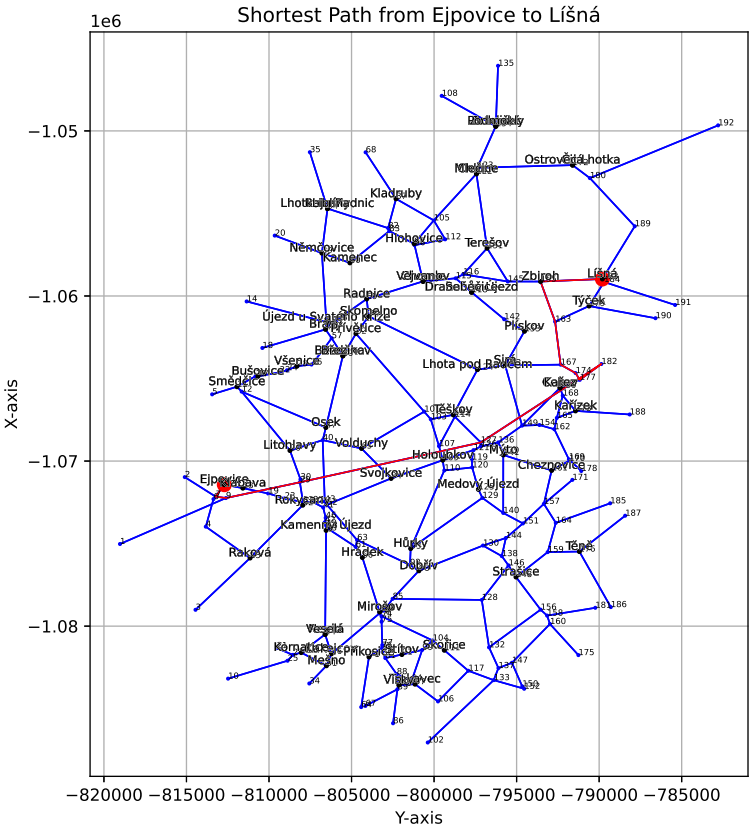
Nejkratší cesta podle Mapy.cz:



Obrázek 7: Nejkratší cesta podle Mapy.cz

Délka cesty: 37.1 km

Neohodnocený graf

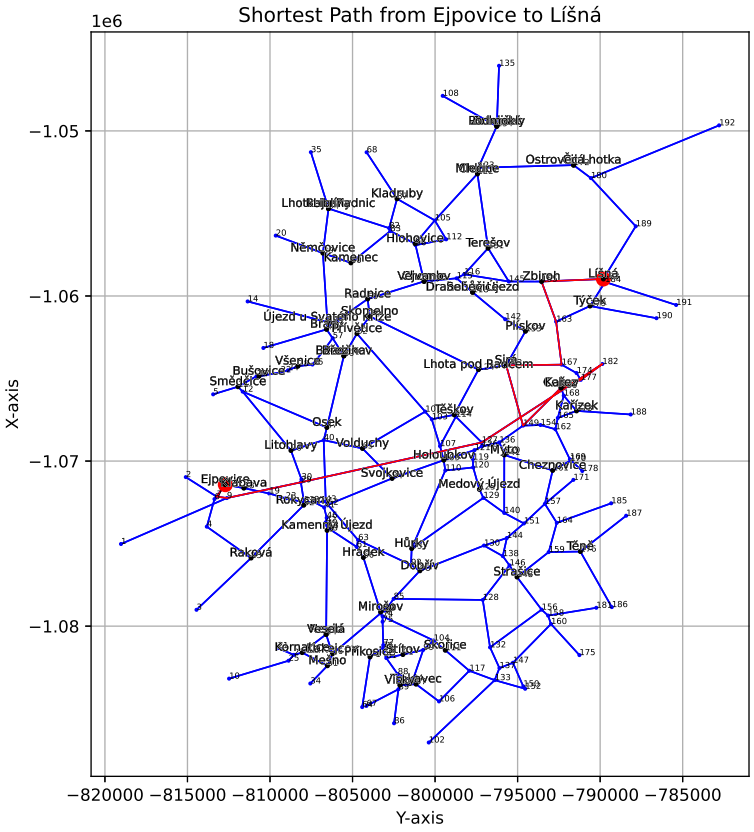


Obrázek 8: Nejkratší cesta v neohodnoceném grafu

Cesta z 8 do 183: [183, 155, 163, 167, 174, 177, 182, 127, 29, 9, 7, 8] s váhou 11.0 (celkem prochází přes 11 hran)

Délka cesty: 38,2 km

Ohodnocený graf podle délky

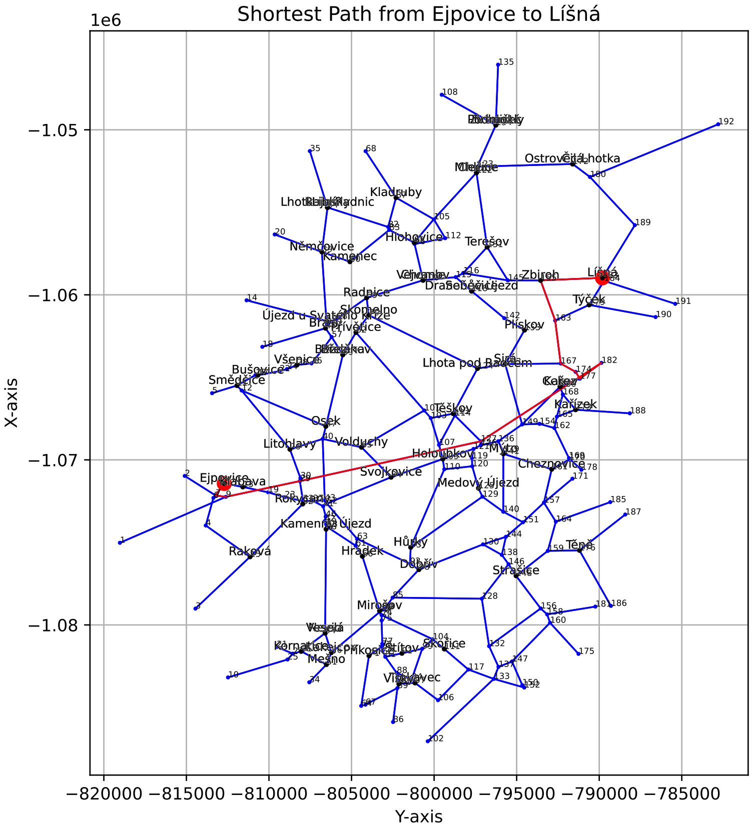


Obrázek 9: Nejkratší cesta v ohodnoceném grafu podle délky

Cesta z 8 do 183: [183, 155, 163, 167, 143, 149, 166, 177, 182, 127, 29, 9, 7, 8] s váhou 0.0062782

Délka cesty: 37,4 km

Ohodnocený graf podle návrhové rychlosti

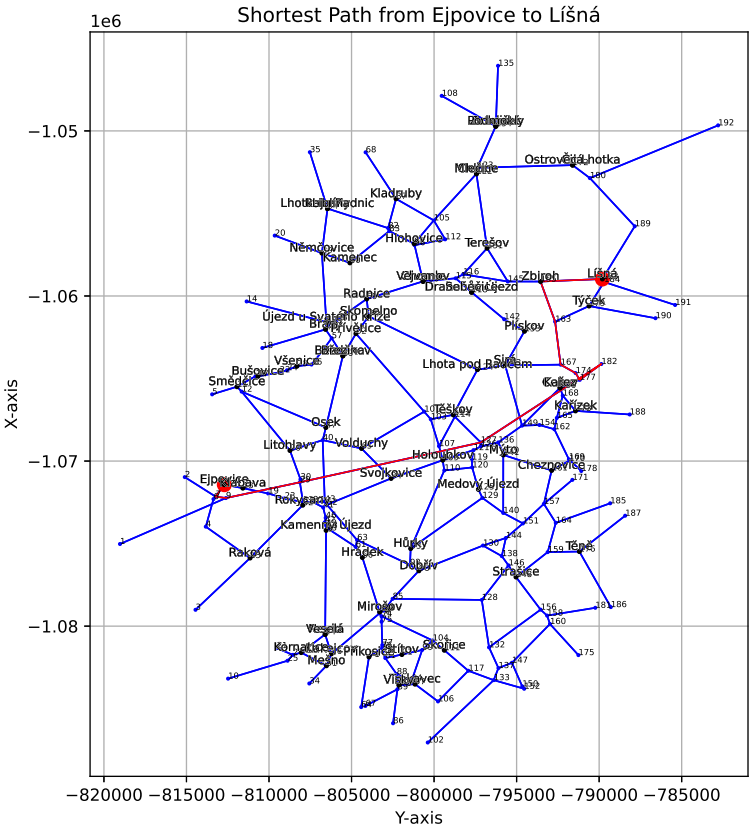


Obrázek 10: Nejkratší cesta v ohodnoceném grafu podle návrhové rychlosti

Cesta z 8 do 183: [183, 155, 163, 167, 174, 177, 182, 127, 29, 9, 7, 8] s váhou 0.13648

Délka cesty: 38,2 km

Ohodnocený graf podle klikatosti

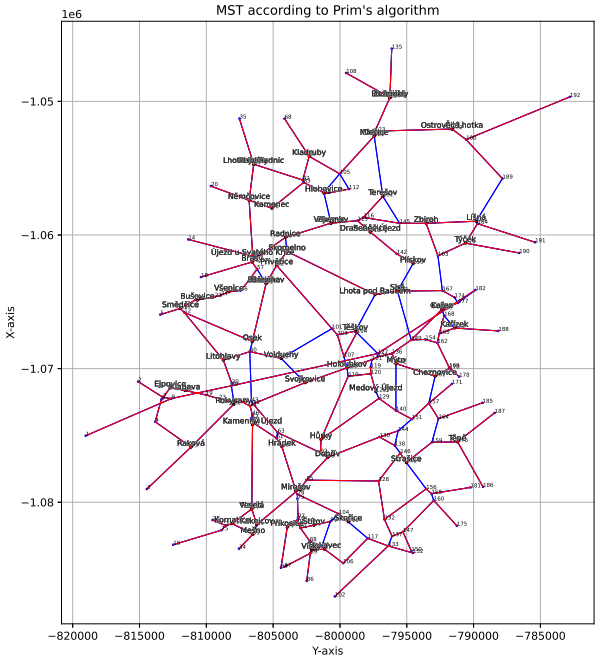


Obrázek 11: Nejkratší cesta v ohodnoceném grafu podle klikatosti

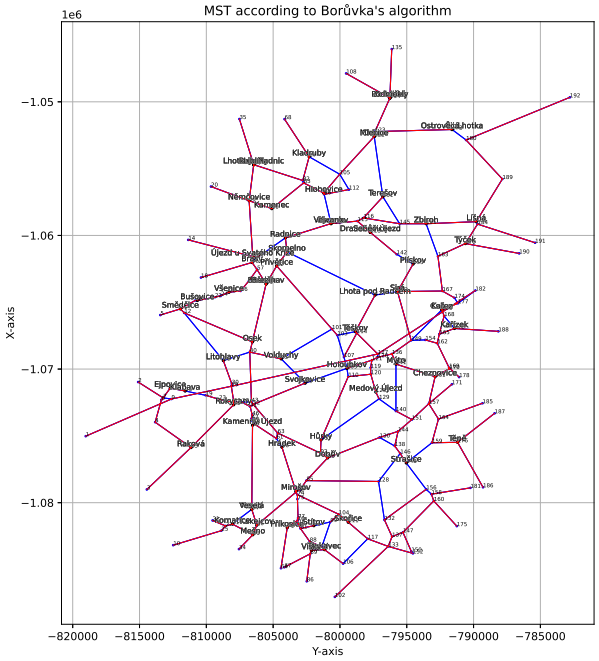
Cesta z 8 do 183: [183, 155, 163, 167, 174, 177, 182, 127, 29, 9, 7, 8] s váhou 11.629

Délka cesty: 38,2 km

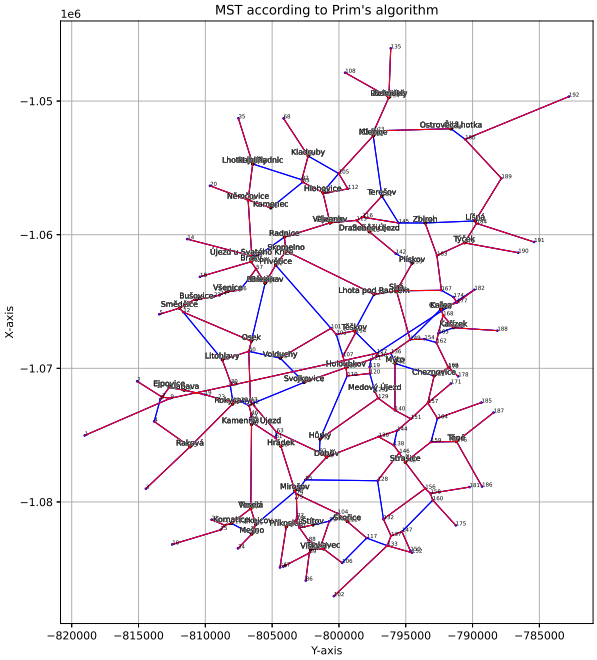
7.2 Minimální kostra



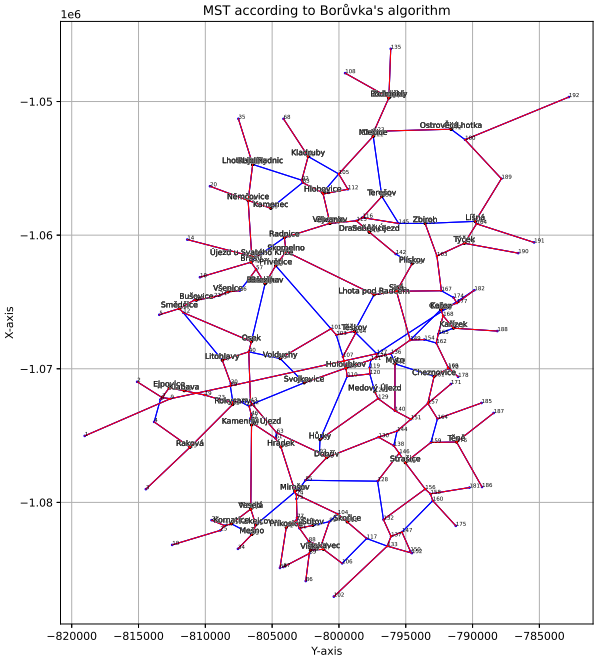
Obrázek 12: Minimální kostra neohodnoceného grafu podle Primova algoritmu



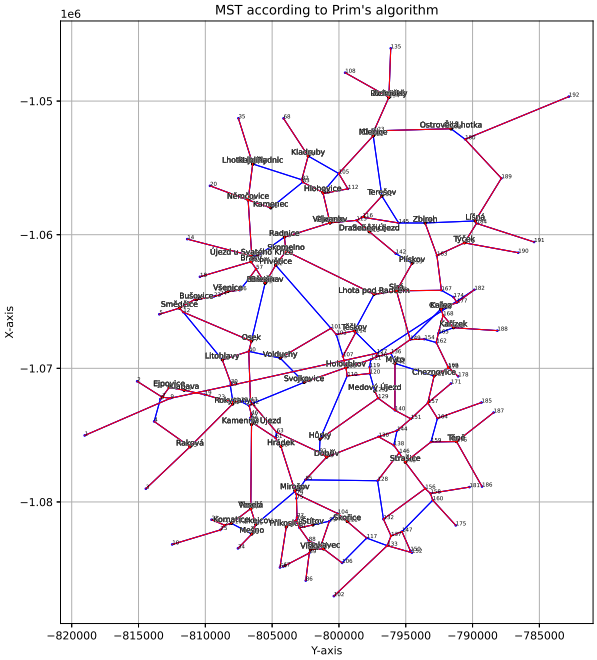
Obrázek 13: Minimální kostra neohodnoceného grafu podle Borůvkova algoritmu



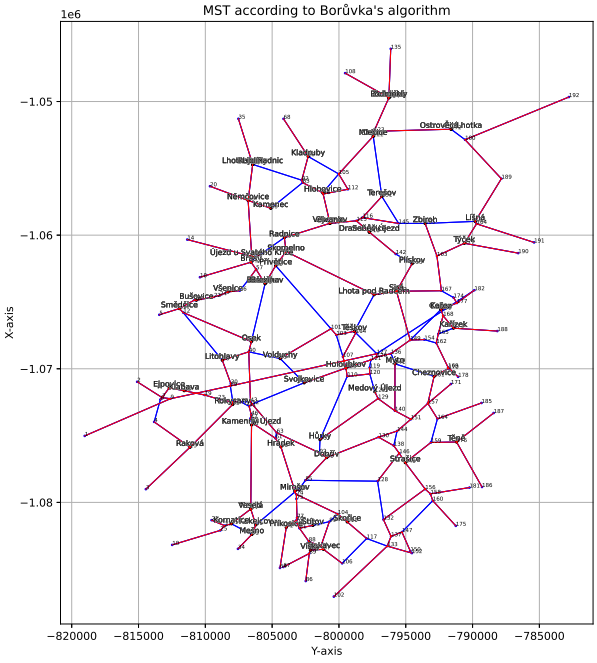
Obrázek 14: Minimální kostra v ohodnoceném grafu podle klikatosti podle Primova algoritmu



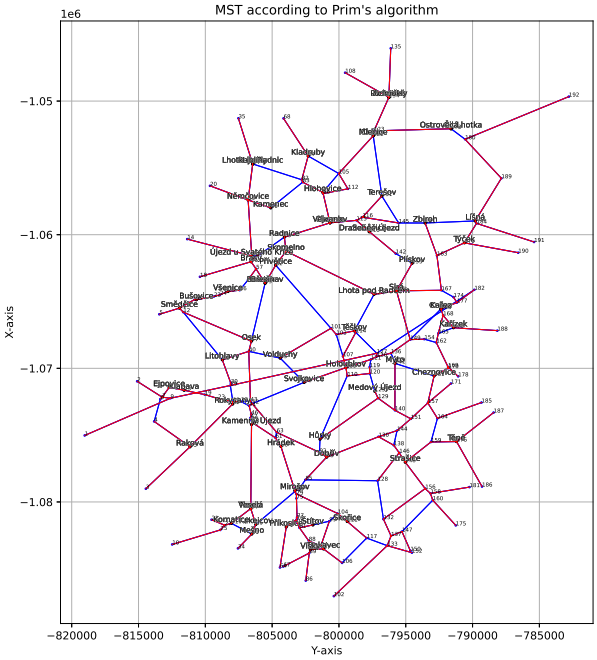
Obrázek 15: Minimální kostra v ohodnoceném grafu podle klikatosti podle Borůvkova algoritmu



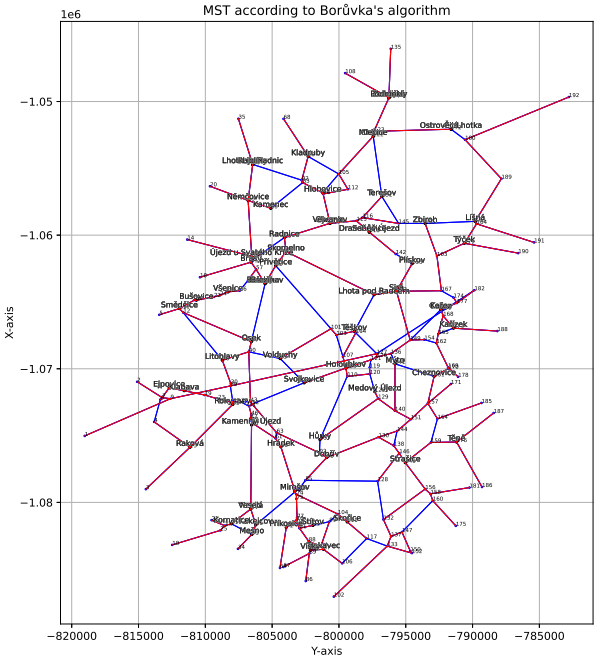
Obrázek 16: Minimální kostra v ohodnoceném grafu podle délky podle Primova algoritmu



Obrázek 17: Minimální kostra v ohodnoceném grafu podle délky podle Borůvkova algoritmu



Obrázek 18: Minimální kostra v ohodnoceném grafu podle návrhové rychlosti podle Primova algoritmu



Obrázek 19: Minimální kostra v ohodnoceném grafu podle návrhové rychlosti podle Borůvkova algoritmu

8 Závěr

V rámci této úlohy byly implementovány algoritmy pro řešení úloh spojených s grafy, konkrétně pro nalezení nejkratších cest a minimálních koster. Nejprve byla na základě geografických dat pro území okresu Rokycany vytvořena grafová reprezentace, která byla následně použita k testování různých variant algoritmů. Bylo provedeno prohledávání grafu do hloubky (DFS) a prohledávání grafu do šířky (BFS). Dále byla nalezena nejkratší cesta mezi uživatelem zvolenými obcemi pomocí Dijkstrova algoritmu, který je určen pro nalezení nejkratší cesty mezi dvěma uzly v grafu s kladně ohodnocenými hranami. Pro grafy obsahující záporné ohodnocení hran byl aplikován Bellman-Fordův algoritmus. Dalším krokem bylo vyhledání nejkratších cest mezi všemi dvojicemi uzlů v grafu. Pro nalezení minimálních koster grafu byly implementovány dva různé algoritmy: Primův a Borůvkův algoritmus. K optimalizaci běhu některých algoritmů byly použity heuristiky, jako jsou Weighted Union a Path Compression. Ve všech případech byly výsledky vypisovány do konzole a byly vytvořeny vizualizace, které zprostředkovaly přehled o nalezených cestách a minimálních kostrách (viz výsledky).

8.1 Další Možné Neřešené Problémy a Náměty na Vylepšení

- **Tvorba grafu:** Při tvorbě grafové reprezentace jsou obce reprezentovány bodovou vrstvou, která je přiřazena k nejbližším uzlům. Tento přístup však může vést k chybnému přiřazení, pokud se obec nachází v oblasti, kde není žádný uzel v bezprostřední blízkosti. V takovém případě může být obec přiřazena k uzlu, který leží na silnici, která jí vůbec neprochází. Možných řešení tohoto problému je několik. Jednou z variant by bylo využití polygonové vrstvy obcí pro generování grafu. Tento přístup by ale mohl vést k situaci, kdy by jedna obec byla reprezentována více uzly. Aktuální přístup přiřazuje každé obci právě jeden uzel, přičemž zůstávají volné uzly, které nepatří žádné obci a reprezentují „křižovatky mimo obce“. Dalším problémem je přítomnost krátkých silničních úseků, které jsou v grafech zohledněny jako hrany, přestože neodpovídají významným dopravním trasám. Řešením by mohlo být sloučení těchto krátkých úseků silnic dohromady s většími úseky silnic.

- **Rekonstrukce cesty:** Na základě uživatelova zadání vracet rovnou uzly, které tvoří nejkratší cestu a ne pouze seznam předchůdců.

- **Vykreslení grafu:** Vytvořit robustnější metody, které dokážou rychleji vykreslit graf. Bylo by také výhodné vytvořit třídu, která by se specializovala pouze na vykreslování grafů, čímž by ulehčila práci uživateli s vizualizacemi grafů.

- **Uzly spojené dvěma hranami:**Vytvořit metodu, která dokáže efektivně vykreslit situaci, kdy jsou dva uzly spojeny více než jednou hranou. Tato funkce by mohla být užitečná v případě, že by hrany reprezentovaly pouze přímou spojnici mezi dvěma body.

- **Spojení všech uzlů nejkratší cestou:** Pro zrychlení výpočtů by bylo efektivnější přejít na Floydův–Warshallův algoritmus, který vyhledává všechny nejkratší cesty v jednom průchodu. Aktuálně je problém řešen pomocí dvou vnořených cyklů `for`, které hledají nejkratší cestu pro každou kombinaci uzlů. Zároveň se momentálně u každé dvojice kontroluje, zda neexistují hrany se záporným ohodnocením, což by stačilo provést pouze jednou.

- **Výpočet celkové délky/času:** Při výpočtu nejkratší cesty pomocí různých kritérií je počítán součet vah nejkratší cesty, což ovšem neodpovídá celkové délce/času. Bylo by tedy vhodné vypisovat kromě váhy i čas. Momentálně byla výsledná délka nalezené cesty spočtena ručně v gis softwaru pomocí nalezených čísel uzlů.

- **Práce s přesnějšími daty:** Výsledky výpočtu nejkratší cesty se liší od cest nalezených pomocí Mapy.cz, pravděpodobně kvůli tomu, že Mapy.cz obsahují aktuálnější a podrobnější data. Data ArcČR 500 jsou značně generalizovaná. Při přiřazování obcí k nejbližším uzlům dochází k dalším nepřesnostem, které negativně ovlivňují data a mohou mít vliv na nalezené nejkratší cesty.

Odkazy

[1] Tomáš Bayer. *Nejkratší cesty grafem.* URL: https://github.com/k155cvut/ygei/blob/main/prednasky/geoinf8.pdf (cit. 25. 11. 2024).

[2] Tomáš Bayer. *Nejkratší cesty grafem.* URL: https://github.com/k155cvut/ygei/blob/main/prednasky/geoinf9.pdf (cit. 25. 11. 2024).

[3] Tomáš Bayer. *Algoritmy v digitální kartografii.* 2008. vyd. Praha: Karolinum, 2008. ISBN: 978-80-246-1499-1.

[4] ArcData. *ArcCR 500-3.3 Popis dat.* URL: https://download.arcdata.cz/data/ArcCR_500-3.3-Popis-dat.pdf (cit. 27. 11. 2024).

A Metody třídy ShortestPath

Metoda `shortest_cost_path`

```
1: Vstup: graf G, startovní uzel A, koncový uzel B
2: Výstup: seznam předchůdců p, celková ohodnocení cesty d[B]
3: if A = B then
4:    return p, d
5: end if
6: if Pokud graf má zápornou hranu then
7:    použij metodu bellman_ford(G,A,B)
8: else
9:    použij metodu dijkstra(G,A,B)
10: end if
11: return p, d
```

Metoda `dijkstra`

```
1: Vstup: graf G, startovní uzel A, koncový uzel B
2: Výstup: seznam předchůdců p, celkové ohodnocení cesty d[B]
3: if A = B then
4:    return p, d[B]
5: end if
6: Inicializace:
7:    p ← prázdný seznam předchůdců
8:    d ← seznam vzdáleností, každý prvek nastaven na ∞
9:    PQ ← prioritní fronta
10: do PQ přidán uzel A s ohodnocením 0
11: d[A] ← 0
12: while PQ není prázdná do
13:    Získání uzlu u ← uzel s nejmenším ohodnocením z PQ
14:    Odstranění u z PQ
15:    if d[u] < ohodnocení uzlu u then
16:       pokračuj
17:    end if
18:    if u = B then
19:       ukonči cyklus
20:    end if
21:    for každého souseda v uzlu u do
22:       w_uv ← ohodnocení hrany mezi u a v
23:       if d[u] + w_uv < d[v] then
24:          d[v] ← d[u] + w_uv
25:          p[v] ← u
26:          do PQ přidán v s ohodnocením d[v]
27:       end if
28:    end for
29: end while
30: return p, d[B]
```

Metoda `bellman_ford`

Vstup: graf G, startovní uzel A, koncový uzel B
Výstup: seznam předchůdců p, vzdálenost k cílovému uzlu $d[B]$
if $A = B$ **then**
  return $[-1]$, 0
end if
$n \leftarrow$ počet uzlů v grafu
$d \leftarrow [\infty] \times (n + 1)$
$p \leftarrow [-1] \times (n + 1)$
$d[A] \leftarrow 0$
for $i = 1$ **to** $n - 1$ **do**
  for každý uzel u v grafu G **do**
    for každý soused v a ohodnocení hrany mezi u a v **do**
      if $d[u] \neq \infty$ a $d[u] +$ ohodnocení $< d[v]$ **then**
        $d[v] \leftarrow d[u] +$ ohodnocení
        $p[v] \leftarrow u$
      end if
    end for
  end for
end for
Kontrola negativních cyklů:
for každý uzel u v grafu G **do**
  for každý soused v a ohodnocení hrany mezi u a v **do**
    if $d[u] \neq \infty$ a $d[u] +$ ohodnocení $< d[v]$ **then**
      raise `ValueError("Negative cycle detected in the graph")`
    end if
  end for
end for
return p, $d[B]$

B Metody třídy MST

Metoda `prim`

```
1: Vstup: graf G
2: Výstup: seznam hran (start, end, weight) T, celkové ohodnocení wt
3: n ← počet uzlů v grafu G
4: v ← [0] × (n + 1)
5: pq ← prioritní fronta
6: wt ← 0
7: T ← [ ]
8: Inicializace:
9: start_node ← libovolný uzel z G
10: for každého souseda neighbor a ohodnocení weight uzlu start_node do
11:    Přidej hranu (weight, start_node, neighbor) do pq
12: end for
13: v[start_node] ← 1
14: while pq není prázdná do
15:    (weight, from_node, to_node) ← pq.get()
16:    if v[to_node] = 0 then
17:       v[to_node] ← 1
18:       Přidej hranu (from_node, to_node, weight) do T
19:       wt ← wt + weight
20:       for každého souseda neighbor a ohodnocení edge_weight uzlu to_node do
21:          if v[neighbor] = 0 then
22:             Přidej hranu (edge_weight, to_node, neighbor) do pq
23:          end if
24:       end for
25:    end if
26: end while
27: return T, wt
```

Metoda `boruvka`

```
1: Vstup: graf G
2: Výstup: seznam hran (start, end, weight) T, celková ohodnocení wt
3: V, E ← seznam vrcholů a hran z grafu G
4: T ← [ ]
5: wt ← 0
6: n ← |V|
7: P ← [−1] × (n + 1)
8: r ← [0] × (n + 1)
9: Inicializace:
10: for každý vrchol v ∈ V do
11:    P[v] ← v
12: end for
13: ES ← hrany E setříděné podle ohodnocení (vzestupně)
14: for každou hranu (u, v, w) ∈ ES do
15:    if find(u, P) ≠ find(v, P) then
16:       (P, r) ← weighted_union(u, v, P, r)
17:       Přidej (u, v, w) do T
18:       wt ← wt + w
19:    end if
20: end for
21: return T, wt
```

Metoda `plot_mst`

Vstup: seznam hran (start, end, weight) T, souřadnice uzlů C, styl linie *line*
$x\_edges \leftarrow [\,]$
$y\_edges \leftarrow [\,]$
for (start, end, weight) v T **do**
    x_edges.extend($[C$[start][0], C[end][0], None$]$)
    y_edges.extend($[C$[start][1], C[end][1], None$]$)
end for
Vykreslení hran:
plt.plot(x_edges, y_edges, line, lw = 1)
Přidání mřížky a nastavení os:
plt.grid(True)
plt.axis(equal)

Metoda `_find`

Vstup: uzel u, pole rodičů P
Výstup: kořen množiny obsahující u
$v \leftarrow u$
Najdi kořen:
while $P[u] \neq u$ **do**
    $u \leftarrow P[u]$
end while
$root \leftarrow u$
Kompresní fáze:
while $v \neq root$ **do**
    $up \leftarrow P[v]$
    $P[v] \leftarrow root$
    $v \leftarrow up$
end while
return *root*

Metoda `_weighted_union`

Vstup: uzly u, v, pole rodičů P, pole hodností r
Výstup: aktualizovaná pole rodičů P a hodností r
$root_u \leftarrow$ **find**(u, P)
$root_v \leftarrow$ **find**(v, P)
if $root_u \neq root_v$ **then**
    if $r[root_u] > r[root_v]$ **then**
        $P[root_v] \leftarrow root_u$
    else if $r[root_v] > r[root_u]$ **then**
        $P[root_u] \leftarrow root_v$
    else
        $P[root_u] \leftarrow root_v$
        $r[root_v] \leftarrow r[root_v] + 1$
    end if
end if
return P, r

C Metody třídy GraphPathFinder

Metoda `DFS`

```
1: Vstup: graf G, startovní uzel A
2: Výstup: seznam předchůdců p
3: n ← počet vrcholů v G
4: p ← [−1] × (n + 1)
5: S ← [0] × (n + 1)
6: St ← [ ]
7: St.append(A)
8: while St není prázdný do
9:    A ← St.pop()
10:   S[A] ← 1
11:   for v v reverse(G[A]) do
12:      if S[v] = 0 then
13:         p[v] ← A
14:         St.append(v)
15:      end if
16:   end for
17:   S[A] ← 2
18: end while
19: return p
```

Metoda `BFS`

```
1: Vstup: graf G, startovní uzel A
2: Výstup: seznam předchůdců p
3: n ← počet vrcholů v G
4: p ← [−1] × (n + 1)
5: S ← [0] × (n + 1)
6: Q ← [ ]
7: S[A] ← 1
8: Q.append(A)
9: while Q není prázdná do
10:   start ← Q.pop(0)
11:   for v v G[A] do
12:      if S[v] = 0 then
13:         S[v] ← 1
14:         p[v] ← A
15:         Q.append(v)
16:      end if
17:   end for
18:   S[A] ← 2
19: end while
20: return p
```

Metoda `all_shortest_paths`

Vstup: graf G
Výstup: slovník obsahující dvojici bodů a nejkratší cestu mezi nimi *paths*
$paths \leftarrow \{\ \}$
$keys \leftarrow$ seznam všech uzlů v G
for $i \leftarrow 0$ **to** $\text{len}(keys) - 1$ **do**
  for $j \leftarrow i + 1$ **to** $\text{len}(keys) - 1$ **do**
    $(P, d_{\min}) \leftarrow \text{shortest\_cost\_path}(keys[i], keys[j])$
    $path \leftarrow \text{rec\_path}(keys[i], keys[j], P)$
    $paths[(keys[i], keys[j])] \leftarrow (path, d_{\min})$
  end for
end for
return *paths*

Metoda `plot_graph`

Vstup: graf G, souřadnice uzlů C, styl linie *line*, barva bodů uzlů points, velikost uzlů a point_size
for node, (x, y) **in** C **do**
  Vykresli bod na souřadnicích (x, y) s barvou points a velikostí point_size
  Přidej popisek uzlu node na pozici $(x + 5, y)$ s velikostí písma 5
end for
for node, neighbors **in** G **do**
  $x_{\text{start}}, y_{\text{start}} \leftarrow C[\text{node}]$
  for neighbor **in** neighbors **do**
    $x_{\text{end}}, y_{\text{end}} \leftarrow C[\text{neighbor}]$ {Souřadnice sousedního uzlu}
    Vykresli čáru mezi $(x_{\text{start}}, y_{\text{start}})$ a $(x_{\text{end}}, y_{\text{end}})$ se stylem line a tloušťkou 1
  end for
end for
Nastav popisky os: X-osa a Y-osa
Aktivuj mřížku

Metoda `plot_path`

Vstup: souřadnice uzlů C, cesta P a styl linie *linie*
$x_{\text{coords}} \leftarrow [\ ]$
$y_{\text{coords}} \leftarrow [\ ]$
for node **in** path **do**
  $(x, y) \leftarrow C[\text{node}]$
  Přidej x do x_{coords}
  Přidej y do y_{coords}
end for
Vykresli čáru mezi souřadnicemi v x_{coords} a y_{coords} se stylem line a tloušťkou 1

Metoda `plot_node_names`

Vstup: souřadnice a názvy uzlů *node_names*
for každý uzel node a jeho souřadnice (x, y) v *node_names* **do**
  Vykresli černý bod na souřadnicích (x, y) s velikostí bodu markersize
  for každý posun (dx, dy) v {(-10, 100), (10, 100), (0, 90), (0, 110), (-10, 90), (10, 110), (-10, 110), (10, 90)} **do**
    Vykresli bílý text na souřadnicích $(x + dx, y + dy)$ s velikostí písma fontsize a barvou bílé
  end for
  Vykresli černý text na souřadnicích $(x, y + 100)$ s velikostí písma fontsize a barvou černé
end for

Metoda `plot_red_nodes`

Vstup: seznam uzlů *nodes*, seznam souřadnic C, velikost bodu *size*
for každý uzel node v *nodes* **do**
  if node $\in C$ **then**
    Získat souřadnice (x, y) z $C[\text{node}]$
    Vykreslit červený bod na souřadnicích (x, y) s velikostí *size*
  end if
end for

Metoda `rec_path`

```
1: Vstup: startovní uzel A, koncový uzel B, seznam předchůdců p
2: Výstup: cesta path
3: path ← [ ]
4: while B ≠ A and B ≠ −1 do
5:     path.append(B)
6:     B ← p[B]
7: end while
8: path.append(B)
9: if B ≠ start then
10:    print("Incorrect path")
11: end if
12: return path
```