

Univerzita Karlova  
Přírodovědecká fakulta



Grafové algoritmy

Petr Havel, Jan Bartušek

1. ročník NGKDPZ

Praha

9. 1. 2026

## Úloha 3 : Nejkratší cesta grafem

Implementujte Dijkstru algoritmus pro nalezení nejkratší cesty mezi dvěma zadanými uzly grafu.

Vstupní data budou představována silniční sítí doplněnou vybranými sídly (alespoň 100 uzelů). S využitím přiloženého skriptu konvertujte podkladová data do grafové reprezentace.

Otestujte různé varianty volby ohodnocení  $w$  hran grafu tak, aby nalezená cesta měla:

- nejkratší Eukleidovskou vzdálenost,
- nejmenší transportní čas (2 varianty, bez/se zohledněním klikačnosti komunikací).

Pro druhou variantu optimální cesty navrhněte také vhodnou metriku, která zohledňuje rozdílnou dobu jízdy na různých typech komunikací dle jejich návrhové rychlosti  $v$  a klikačnosti  $\kappa$ , příkladem může být

$$t(u, v) = \kappa \frac{l(u, v)}{v(u, v)}.$$

Pro výpočet  $\kappa$  využijte poměr délky polylinie  $l$  představující diskrétní křivku a vzdálenosti  $s$  koncových bodů polylinie

$$\kappa = \frac{l(u, v)}{s(u, v)}.$$

Každou z variant otestujte pro dvě dvě různé cesty tvořené alespoň 20 uzly. Výsledky umístěte do tabulky, vlastní cesty vizualizujte. Dosažené výsledky porovnejte s vybraným navigačním SW (alespoň 3).

Krok	Hodnocení
Dijkstra algoritmus.	20b
Návrh jiného ohodnocení hran zohledňující křivolkost silniční sítě.	+5b
Zohlednění vlivu DMT.	+15b
Nalezení nejkratších cest mezi všemi dvojicemi uzelů.	+15b
Nalezení minimální kostry Borůvka/Kruskal.	+15b
Nalezení minimální kostry Jarník/Prime.	+15b
Využití heuristiky Weighted Union	+5b
Využití heuristiky Path Compression	+5b
<b>Max celkem:</b>	<b>95b</b>

Čas zpracování: 3-4 týdny.

V rámci úkolu byly řešeny následující bonusové úlohy:

- Nalezení minimální kostry grafu pomocí Borůvkova/Kruskalova algoritmu (+15 bodů)
- Využití heuristiky Weighted Union (+5 bodů)
- Využití heuristiky Path Compression (+5 bodů)

Celkem 25 bodů.

## 1. Úvod a zadání

Cílem úlohy bylo implementovat algoritmy pro analýzu grafů a jejich aplikaci na silniční síť. Hlavním zadáním bylo nalezení nejkratší cesty mezi uzly grafu pomocí Dijkstrova algoritmu při různých variantách ohodnocení hran. Dále bylo cílem porovnat výsledné trasy s běžně dostupnými navigačními systémy a ověřit jejich realističnost. Jako bonusová úloha bylo řešeno nalezení minimální kostry grafu.

Řešení práce je postaveno na postupném zpracování dat pomocí několika Python skriptů, které na sebe navazují. V první fázi jsou stažena a připravena vstupní data představující reálnou silniční síť. Následně jsou tato data převedena do grafové reprezentace, nad kterou jsou aplikovány jednotlivé algoritmy. Výsledné nejkratší cesty jsou exportovány do formátu vhodného pro vizualizaci v prostředí QGIS, kde je možné je prostorově analyzovat a porovnat s trasami navrženými externími navigačními službami.

## 2. Teoretická část

Pro řešení úlohy byly využity základní grafové algoritmy pracující nad uliční sítí, která byla reprezentována jako neorientovaný ohodnocený graf. V našem případě vrcholy představují křižovatky a hrany jednotlivé silniční úseky. K nalezení nejkratší cesty byl zvolen Dijkstrův algoritmus, který Mareš a Valla (2017) popisují zjednodušeně tak, že: „Pro každý původní vrchol si pořídíme „budík“. Jakmile k vrcholu zamíří vlna, nastavíme jeho budík na čas, kdy do něj vlna má dorazit (pokud míří po více hranách najednou, zajímá nás, kdy dorazí poprvé).“ V rámci skriptu dijk.py bylo tohoto postupu docíleno pomocí prioritní fronty, která umožňuje efektivní výběr uzlu s minimálním odhadem vzdálenosti a následnou relaxaci hran.

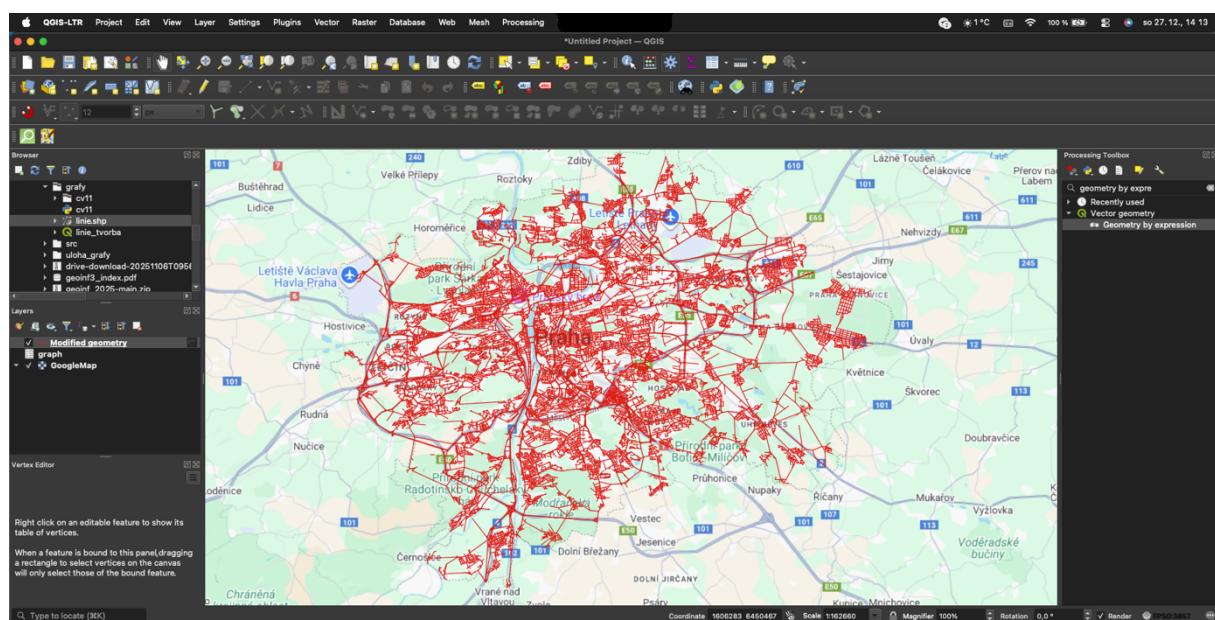
V bonusové části práce bylo řešeno nalezení minimální kostry grafu pomocí algoritmu, který v roce 1956 popsal Joseph Kruskal. „Tento algoritmus je založen na „hladovém“ přístupu: zkouší přidávat hrany od nejlehčí po nejtěžší a zahazuje ty, které by vytvořily cyklus“ (Mareš a Valla 2017). Pro efektivní správu komponent a detekci cyklů byla ve skriptu kruskal.py využita datová struktura Union-Find, která „reprezentuje komponenty souvislosti grafu a umí na nich provádět následující operace: Find(u,v) zjistí, zda vrcholy u a v leží v téže komponentě, Union(u,v) přidá hranu uv, čili dvě komponenty spojí do jedné“ (Mareš a Valla 2017).

### 3. Implementace a data

#### download\_dat.py

Skript download\_dat.py slouží k zajištění a předzpracování vstupních dat.

Na začátku skriptu je využita knihovna OSMnx, která umožňuje přímý přístup k datům od OpenStreetMap. Jako zájmové území je zvolena oblast Praha, Czechia. Pomocí funkce `graph_from_place` je stažena silniční síť určená pro automobilovou dopravu (`network_type="drive"`). Parametr `simplify=True` zajišťuje topologické zjednodušení sítě (odstranění zbytečných lomových bodů).



Po stažení grafu jsou do jeho hran dopočteny další důležité atributy. Funkce `add_edge_speeds` přiřazuje jednotlivým úsekům odhadovanou návrhovou rychlosť na základě typu komunikace. Následně funkce `add_edge_travel_times` vypočítává orientační čas průjezdu hranou, který vychází z délky úseku a jeho rychlostního omezení. Tím vzniká základ pro optimalizaci tras nejen podle délky, ale i podle času jízdy.

Dalším krokem je extrakce souřadnic jednotlivých uzelů grafu. Pro každý uzel jsou uloženy jeho souřadnice x (zeměpisná délka) a y (zeměpisná šířka) do slovníků. Tyto souřadnice jsou později použity při exportu grafu do textového formátu a také při vizualizaci výsledných tras v prostředí QGIS.

```
xs = {n: float(G.nodes[n]["x"]) for n in G.nodes}
ys = {n: float(G.nodes[n]["y"]) for n in G.nodes}

def s_uv(u, v):
    return ox.distance.great_circle(ys[u], xs[u], ys[v], xs[v])
```

Součástí skriptu je také definice funkce pro výpočet přímé (eukleidovské) vzdálenosti mezi dvěma uzly grafu. S využitím metody great\_circle poradila AI. Tato vzdálenost představuje délku spojnice mezi koncovými body hrany a slouží k výpočtu klikatosti komunikace. Klikatost je vypočítána jako poměr skutečné délky silničního úseku a přímé vzdálenosti mezi jeho konci. Tento parametr umožňuje zohlednit fakt, že některé komunikace jsou výrazně zakřivené a jejich projetí může být časově náročnější, než by odpovídalo samotné délce.

Na základě výpočtů jsou vytvořeny tři různé varianty ohodnocení hran grafu. První varianta odpovídá čisté geometrické délce hran, druhá variantě minimalizující celkový čas jízdy a třetí varianta kombinuje čas jízdy s vlivem klikatosti komunikace. Pro každou dvojici uzlů je vybrána nejlepší (nejvýhodnější) hrana podle dané metriky, aby v grafu nevznikaly duplicitní paralelní hrany se stejnými koncovými uzly.

Výsledkem skriptu je export tří textových souborů: graph.txt, graph\_time.txt a graph\_time\_krivost.txt. Každý soubor obsahuje seznam hran ve tvaru souřadnic počátečního a koncového uzlu a jejich váhy. Tyto soubory představují vstupní data pro další skripty, ve kterých je graf převáděn do interní struktury a nad ním jsou aplikovány algoritmy pro hledání nejkratších cest a minimálních koster.

## prevod\_na\_graph.py

Skript prevod\_na\_graph.py zajišťuje převod předzpracovaných dat silniční sítě do interní grafové struktury vhodné pro aplikaci grafových algoritmů. Navazuje na skript download\_dat.py, ze kterého přebírá textový popis hran silniční sítě ve formě souřadnic jejich koncových bodů a odpovídajících vah.

Vstupní data jsou načítána funkcí loadEdges, která z textového souboru postupně čte jednotlivé hrany a ukládá jejich počáteční a koncové body spolu s vahami. Následně je z těchto hran vytvořen seznam všech unikátních uzlů grafu. Funkce build\_nodes shromažďuje všechny koncové body hran a odstraňuje duplicitu, čímž vzniká množina uzlů silniční sítě.

Protože grafové algoritmy pracují s celočíselnými identifikátory uzlů, jsou geografické souřadnice v dalším kroku převedeny na číselná ID. Funkce pointsToIDs vytváří jednoznačné mapování mezi souřadnicemi uzlů a jejich identifikátory, což umožňuje efektivní práci s grafovou strukturou.

Vlastní graf je sestaven ve funkci edgesToGraph pomocí adjacency listu. Graf je uvažován jako neorientovaný, a proto je každá hrana uložena v obou směrech se stejnou vahou. Výsledkem skriptu je interní reprezentace neorientovaného ohodnoceného grafu, která je následně využita při výpočtu nejkratších cest.

```

3  def loadEdges(file_name):
4      PS, PE, W = [], [], []
5      with open(file_name, encoding="utf-8") as f:
6          for line in f:
7              x1, y1, x2, y2, w = line.split()
8              PS.append((float(x1), float(y1)))
9              PE.append((float(x2), float(y2)))
10             W.append(float(w))
11     return PS, PE, W
12
13 def build_nodes(PS, PE):
14     PSE = list(set(PS + PE))
15     PSE.sort()
16     return PSE
17 def pointsToIDs(PSE):
18     return {pt: i for i, pt in enumerate(PSE)}
19
20 def edgesToGraph(D, PS, PE, W):
21     G = defaultdict(dict)
22
23     for i in range(len(PS)):
24         a = D[PS[i]]
25         b = D[PE[i]]
26         w = W[i]
27         if b not in G[a] or w < G[a][b]:
28             G[a][b] = w
29             G[b][a] = w
30
31     return G

```

## dijk.py

Skript dijk.py obsahuje implementaci Dijkstrova algoritmu a slouží k nalezení nejkratší cesty mezi zadáným startem a cílem pro různé varianty ohodnocení hran. Ve skriptu jsou pro demonstraci uvedeny dvě možné dvojice startovních a cílových bodů, přičemž vždy je aktivní pouze jedna z nich. První, (zakomentovaná) dvojice odpovídá trase z malostranské hospody U Hrocha na Kubánské náměstí, zatímco druhá dvojice odpovídá trase z metra Háje na Košířské náměstí v blízkosti Kina Kavalírka.

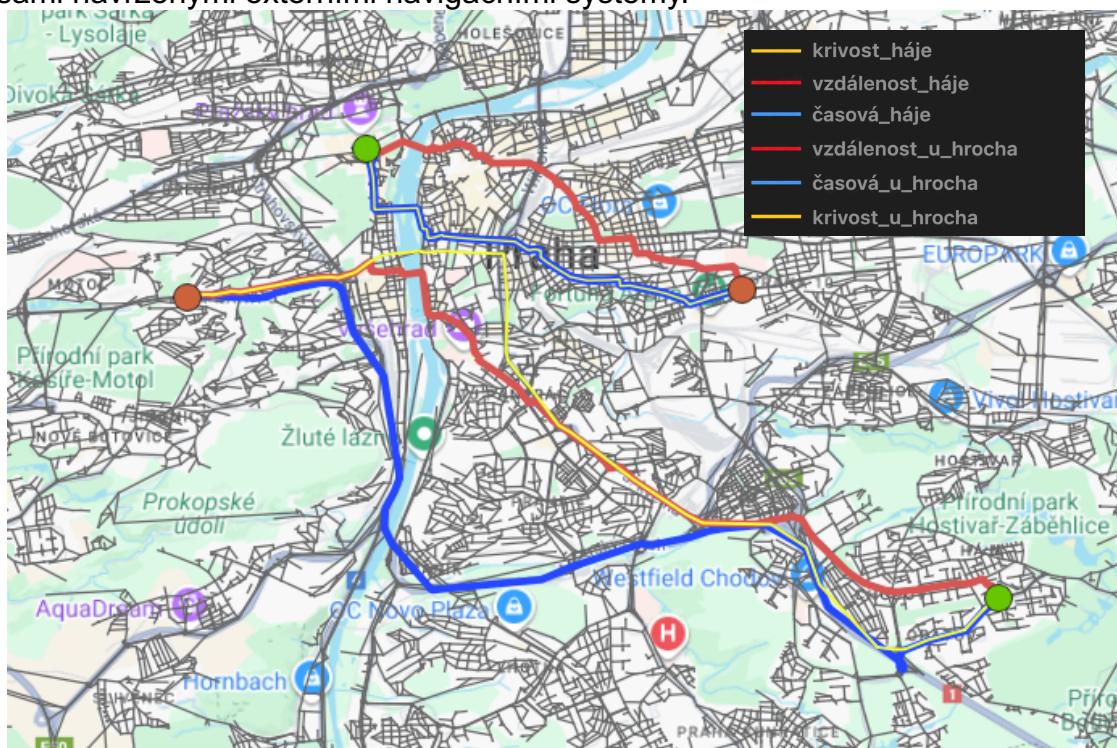
Na začátku skriptu jsou importovány pomocné funkce z modulu prevod\_na\_graph.py, které zajišťují načtení hran, vytvoření seznamu uzlů, mapování uzlů na celočíselné identifikátory a sestavení listu neorientovaného ohodnoceného grafu. Pro efektivní

implementaci Dijkstrova algoritmu je použita prioritní fronta, která umožňuje vždy vybírat uzel s aktuálně nejmenší známou vzdáleností od startu. Protože zadané startovní a cílové body neodpovídají vždy přesně uzlům silniční sítě, je definována pomocná funkce pro nalezení nejbližšího uzlu grafu k daným souřadnicím.

Samotný Dijkstrův algoritmus funguje pomocí relaxace hran. Pro každý uzel je udržována aktuálně nejlepší známá vzdálenost od startu a informace o jeho předchůdci na nejkratší nalezené cestě. Algoritmus postupně rozšiřuje množinu zpracovaných uzlů, dokud není nalezena optimální cesta do cíle nebo dokud nejsou vyčerpány všechny dosažitelné uzly. Díky použití prioritní fronty je výpočet efektivní i pro graf s větším počtem uzlů. Po ukončení algoritmu je nalezená nejkratší cesta rekonstruována zpětným procházením pole předchůdců. Výsledkem je konkrétní posloupnost uzlů, která reprezentuje optimální trasu mezi startem a cílem pro danou variantu ohodnocení hran.

Pro každou variantu ohodnocení hran je výsledek uložen do samostatného CSV souboru. Konkrétně jsou generovány soubory cesta\_vzdalenost.csv, cesta\_cas.csv a cesta\_krivost\_cas.csv, které odpovídají optimalizaci podle geometrické délky, času jízdy a času jízdy se zohledněním klikačnosti komunikací. Každý CSV soubor obsahuje řádky odpovídající jednotlivým uzlům nalezené trasy a je strukturován pomocí atributů order, id, x a y. Atribut order určuje pořadí uzlu v trase a umožňuje zachovat správnou topologii při následném zpracování.

Takto vytvořené CSV soubory jsou přímo připraveny pro vizualizaci v prostředí QGIS. Po jejich načtení jako bodových vrstev lze jednotlivé body seřadit podle atributu order a převést je na líniovou vrstvu, která reprezentuje výslednou trasu. Tento postup umožňuje přehledné porovnání jednotlivých variant nejkratších cest mezi sebou i s trasami navrženými externími navigačními systémy.



## 4. Výsledky

Výsledky výpočtu nejkratších cest jsou shrnutы в tabulkách, které zobrazují počet uzelů výsledné trasy, celkovou délku trasy v kilometrech a odhadovaný čas jízdy v minutách pro jednotlivé varianty.

V případě trasy Háje – Kavalírka jsou patrné výraznější rozdíly mezi jednotlivými variantami. Varianta optimalizovaná na vzdálenost vede přes nejkratší možnou trasu z geometrického hlediska, avšak obsahuje nejvyšší počet uzelů a zároveň vykazuje nejdelší čas jízdy. Naopak varianta optimalizovaná na čas volí delší trasu z hlediska vzdálenosti, ale s menším počtem uzelů a kratší celkovou dobou jízdy. Varianta zohledňující klikačnost komunikací představuje kompromis mezi oběma přístupy – délka trasy i čas jízdy se nachází mezi hodnotami dosaženými u čistě vzdálenostní a časové optimalizace. Tyto rozdíly odpovídají očekávání a potvrzují, že volba váhy má vliv na výslednou trasu.

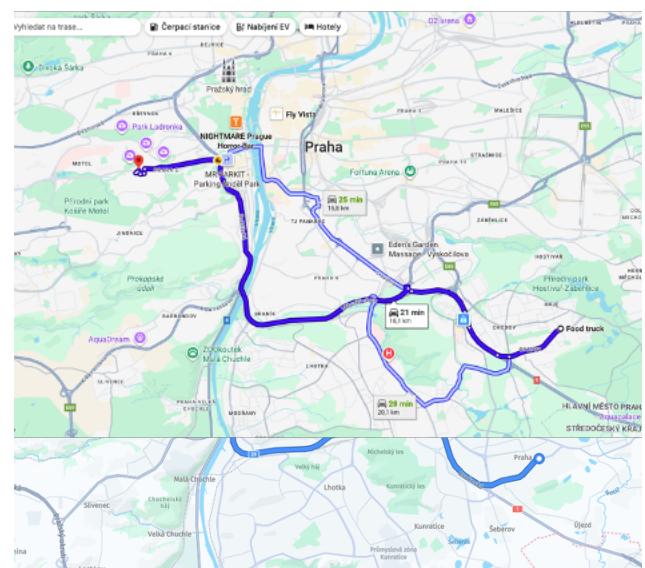
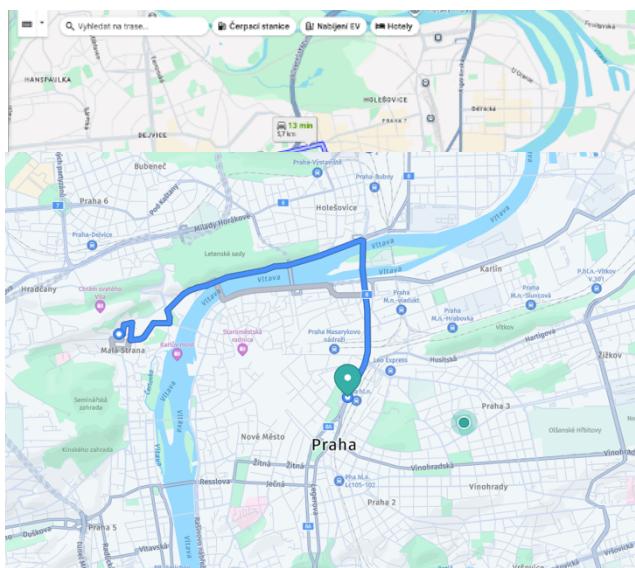
Metro Háje – Kavalírka

varianta	uzlu	vzdalenost_km	cas_min
<b>vzdalenost</b>	110	14.5769	18.15
<b>cas</b>	54	17.457	15.18
<b>krivost</b>	80	15.3485	15.52

U druhé trasy, vedené mezi lokalitami U Hrocha – Kubánské náměstí, jsou rozdíly mezi jednotlivými variantami stále pozorovatelné. Varianta optimalizovaná na vzdálenost vede po mírně kratší trase. Varianty optimalizované na čas a na čas se zohledněním klikačnosti komunikací vedou po totožné trase. V tomto případě tedy faktor klikačnosti neovlivnil výběr výsledné trasy oproti čistě časové optimalizaci, což by mohlo souviset s kratší vzdáleností celé trasy.

U Hrocha – Kubánské náměstí

varianta	uzlu	vzdalenost_km	cas_min
<b>vzdalenost</b>	91	6.9139	10.03
<b>cas</b>	90	7.0546	8.9
<b>krivost</b>	90	7.0546	8.9



Pro porovnání s navigačními systémy byly využity tři různé navigace: Google Maps, Mapy.com a HERE WeGo. Pro každou z testovaných tras byly v těchto systémech vyhledány doporučené trasy a jejich odhadované časy jízdy. Z porovnání vyplývá, že navržené trasy i odhadované časy jízdy se ve většině případů velmi dobře shodují s výsledky získanými pomocí vlastního řešení. Navigační systémy volí velmi podobné průběhy tras a rozdíly v délce či času jízdy jsou pouze v řádu jednotek procent.

Rozdíly mezi trasami lze pozorovat v hustě zastavěných částech města, kde navigační systémy někdy volí alternativní průjezd po hlavních komunikacích. Tyto odchylky jsou však očekávatelné a vyplývají především z rozdílných datových podkladů a zohledňovaných omezení.

## 5. Řešení bonusové části

### kruskal.py

Skript kruskal.py slouží k nalezení minimální kostry grafu pomocí Kruskal / Borůvkova algoritmu. Řeší bonusovou část úlohy zaměřenou na analýzu struktury grafu. Zatímco v předchozích částech práce byly řešeny nejkratší cesty mezi konkrétními dvojicemi uzlů, cílem tohoto skriptu je nalézt takovou množinu hran, která propojí všechny uzly grafu s minimálním možným součtem vah a zároveň neobsahuje žádné cykly.

Princip algoritmu spočívá v postupném spojování komponent grafu pomocí nejvhodnějších hran. Hrany jsou zpracovávány podle jejich váhy a do výsledné struktury jsou přidávány pouze v případě, že propojují dvě dosud oddělené komponenty. Tím je zajištěno, že výsledná struktura tvoří minimální kostru grafu.

Pro efektivní správu komponent grafu a detekci cyklů je ve skriptu použita datová struktura Union-Find. Každý uzel je na začátku považován za samostatnou komponentu a v průběhu algoritmu dochází k jejich sjednocování. V samotném scriptu jsou využity obě optimalizační heuristiky uvedené v zadání. Heuristika Weighted Union zajišťuje, že menší strom je při sjednocování vždy připojen k většímu, čímž se omezuje růst hloubky struktury a heuristika Path Compression je použita při vyhledávání reprezentanta komponenty a vede k přímému přesměrování uzlů na kořen, což výrazně zrychluje opakování operace.

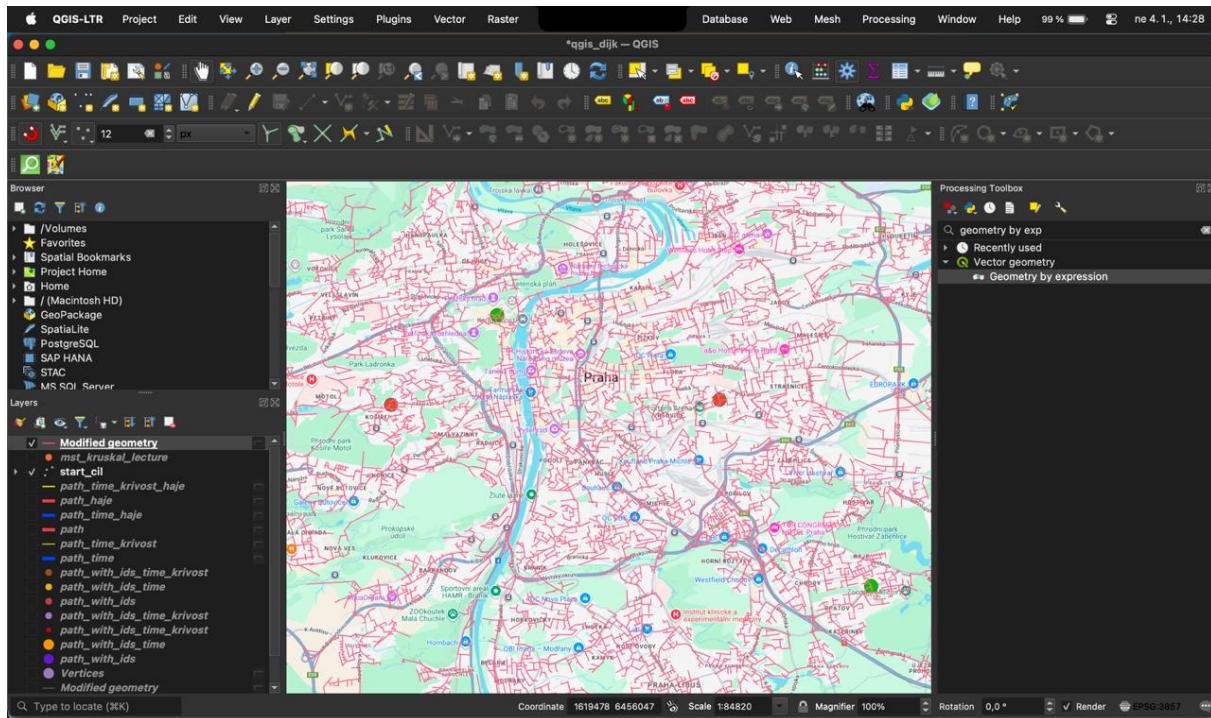
Výsledkem běhu skriptu je nejen výpočet minimální kostry, ale také její export do CSV souboru kostra.csv. Tento soubor obsahuje seznam hran tvořících minimální kostru

```

8  def find(u, p):
9      while p[u] != u:
10         p[u] = p[p[u]]
11         u = p[u]
12     return u
13
14 def union(u, v, p, r):
15     root_u = find(u, p)
16     root_v = find(v, p)
17
18     if root_u != root_v:
19         if r[root_u] > r[root_v]:
20             p[root_v] = root_u
21         elif r[root_v] > r[root_u]:
22             p[root_u] = root_v
23         else:
24             p[root_u] = root_v
25             r[root_v] = r[root_v] + 1
26
27 # Kruskal
28 def mstk(V, E):
29     T = []
30     wt = 0
31     p = [inf] * (len(V) + 1)
32     r = [inf] * (len(V) + 1)
33     for v in V:
34         make_set(v, p, r)
35     ES = sorted(E, key=lambda x: x[2])
36     for e in ES:
37         u = e[0]
38         v = e[1]
39         w = e[2]
40         koren_u = find(u, p)
41         koren_v = find(v, p)
42
43         if koren_u != koren_v:
44             union(u, v, p, r)
45             T.append([u, v, w])
46             wt = wt + w
47
48     return wt, T

```

grafu včetně souřadnic jejich koncových bodů. Díky tomu je výstup připraven pro vizualizaci v prostředí QGIS. Po načtení CSV souboru jako bodové vrstvy lze využít nástroj „Geometry by expression“ díky kterému vytvoříme z bodové liniovou vrstvu kostry grafu.



## 6. Závěr

Tato semestrální práce se věnovala implementaci grafových algoritmů na reálné silniční síti Prahy. Hlavním přínosem bylo ověření vlivu různých metrik ohodnocení hran na výslednou trasu v rámci Dijkstrova algoritmu. Ukázalo se, že zatímco optimalizace na vzdálenost minimalizuje délku, vede k časově nejnáročnějším výsledkům. Naopak časová optimalizace a započtení metriky klikatosti komunikací poskytly realističtější trasy s kratší dobou jízdy, které se v řádu jednotek procent shodovaly s komerčními navigacemi Google Maps, Mapy.cz a HERE WeGo. V rámci bonusového zadání byl úspěšně aplikován Kruskalův algoritmus pro nalezení minimální kostry grafu, přičemž efektivitu výpočtu zvýšilo využití heuristik Weighted Union a Path Compression v datové struktuře Union-Find. Celý proces od stažení dat přes analýzu v Pythonu až po vizualizaci v prostředí QGIS prokázal funkčnost navrženého řešení pro komplexní geoinformatické úlohy.

Při vypracování práce byl jako podpůrný nástroj využit model LLM Gemini. AI asistrovala především u skriptu `download_dat.py` při transformaci dat z knihovny OSMnx do formátu kompatibilního s grafovými algoritmy, při návrhu výpočtu klikatosti a také s logikou exportu tří různých souborů zároveň. Dále byla využita k efektivnímu zápisu třídy UnionFind v rámci skriptu `kruskal.py` a k finální stylistické úpravě textu práce.

## Seznam použité literatury a zdrojů

1. MAREŠ, Martin a Tomáš VALLA. *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, 2017. ISBN 978-80-88168-22-5.
2. BOEING, Geoff. *OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks*. Computers, Environment and Urban Systems, 2017.
3. Tomáš BAYER, Poskytnuté přednáškové prezentace:  
Úvod do grafových algoritmů. Orientovaný, neorientovaný graf. BFS. DFS.  
Nejkratší cesta grafem. Minimální kostra. Relaxace hrany. Dijkstra. Kostra grafu.  
Borůvkův / Kruskalův algoritmus. Union-Find.