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 C102	vypracoval Matyáš Pokorný Tereza Černohousová	datum 3.12.2024	klasifikace

Technická zpráva

1 Bonusové úlohy

Z bonusových úloh námi byly zpracovány:

- Řešení úlohy pro grafy se záporným ohodnocením.
- Nalezení cesty mezi všemi dvojicemi uzlů
- Nalezení minimální kostry Kruskal
- Nalezení minimální kostry Prime

2 Pracovní postup

2.1 Dijkstra algoritmus

Dijkstra algoritmus je metoda pro hledání nejkratší cesty v grafu. Jeho princip spočívá v postupném prozkoumávání grafu, přičemž se začíná od počátečního uzlu. Algoritmus udržuje množinu již navštívených uzlů a pro každý uzel počítá vzdálenost od počátečního uzlu. V každém kroku se vybere uzel s nejmenší vzdáleností, který ještě nebyl navštíven, a z jeho sousedních uzlů se aktualizují vzdálenosti. Tento proces pokračuje, dokud se všechny uzly nezpracují nebo dokud není nalezena cesta k cílovému uzlu.

Dijkstra algoritmus garantuje nalezení optimální cesty, přičemž jeho časová složitost závisí na použité struktuře pro uložení uzlů. V naší implementaci byla pro uložení uzlů použita prioritní fronta.

Náš algoritmus vytváří seznam předchůdců v optimální cestě pro daný počáteční uzel. Poté, je vytvořena cesta další zadaný bod - koncový.

Algoritmus pracuje s různým oceněním hran, v našem případě se jedná o:

- vzdálenost danou délkou polylinie,
- transportní čas daný délkou polylinie a rychlostí danou třídou silnice,
- transportní čas daný délkou polylinie a rychlostí podle třídy silnice, navíc s uvážením křivosti silnice.

Python skript *dijkstra_better.py* poté obsahuje další funkce, které provedou na vytvořeném grafu nalezení nejkratší cesty dijkstra algoritmem a cestu graficky zobrazí. Spolu s vrcholy, které cestu tvoří je zároveň vypočtena celková cena cesty.

2.2 Bellman-Fordův algoritmus

Pro ošetření grafů se záporným ohodnocením byl Dijkstrův algortimus upraven do podoby Bellman-Fordova algoritmu, který dokáže detekovat záporné cykly a v případě jejich existence výpočet zastavit. Tento algoritmus implemetnuje skript *modif_dijkstra.py* a je testován na příkladovém grafu, a nikoliv na reálné síti (ta neobsahuje záporné hrany).

2.3 Nalezení cest mezi všemi vrcholy

Pro nalezení cest mezi všemi dvojicemi vrcholů byl výpočet pro nalezení cest zabalen do dvou *for* cyklů, ketré pro každý počáteční a koncový bod vytvoří cestu a uloží do určené proměnné. Tento postup je implementován ve skriptu *dijkstra_all2all.py*. Postup je sice velmi jednoduchý, ale v jazyku Python trvá velmi velmi dlouho v reálné silniční síti.

2.4 Borůvkův algoritmus

Borůvkův (nebo Kruskalův) algoritmus slouží pro nalezení minimální kostry v souvislém a váženém grafu. Algoritmus pracuje iterativně postupným spojováním komponent grafu. V každé iteraci identifikuje pro každou komponentu hranu s nejmenší vahou, která ji spojuje s jinou komponentou. Tyto hrany jsou poté přidány do kostry, čímž se komponenty propojí. Proces pokračuje, dokud všechny komponenty nesplynou v jednu souvislou kostru. Algoritmus implementuje skript *boruvka.py*.

2.5 Jarníkův algoritmus

Jarníkův (Primův) algoritmus je algoritmus pro nalezení minimální kostry v souvislém a váženém grafu. Algoritmus začíná u libovolného vrcholu a postupně přidává hrany s nejmenší váhou, které spojují již navštívené vrcholy s dosud nenavštívenými. Tento proces pokračuje, dokud nejsou zahrnuty všechny vrcholy grafu. Klíčovou vlastností algoritmu je jeho hladový (chamtivý) přístup, kdy v každém kroku volí pouze nejlevnější hranu přidávající nový vrchol. Algoritmus implementuje skript *jarnik.py*, přičemž běžný Jarníkův algoritmus si neporadí s reálnou sítí, která obecně nemusí být souvislá, proto je prezentován skript *jarnik_nesouvisly.py*, který je ošetřen tak, aby přeskakoval malé nesouvislé komponenty grafu.

2.6 Implementace v reálně síti

Síť komunikací, ze kterých byl tvořen graf a počítána cesta, byl vzat z databáze ArcČR500 verze 3.3. Z těchto dat byla použita datová vrstva Silnice_2015, a to z okresů Cheb a Tachov.

Vrstva silnic byla exportovaná jako Geopackage, jakožto otevřený a standartizovaný moderní formát pro práci s prostorovými daty. K výběru dat na konkrétním území a k uložení do požadovaného formátu byl využit program QGIS.

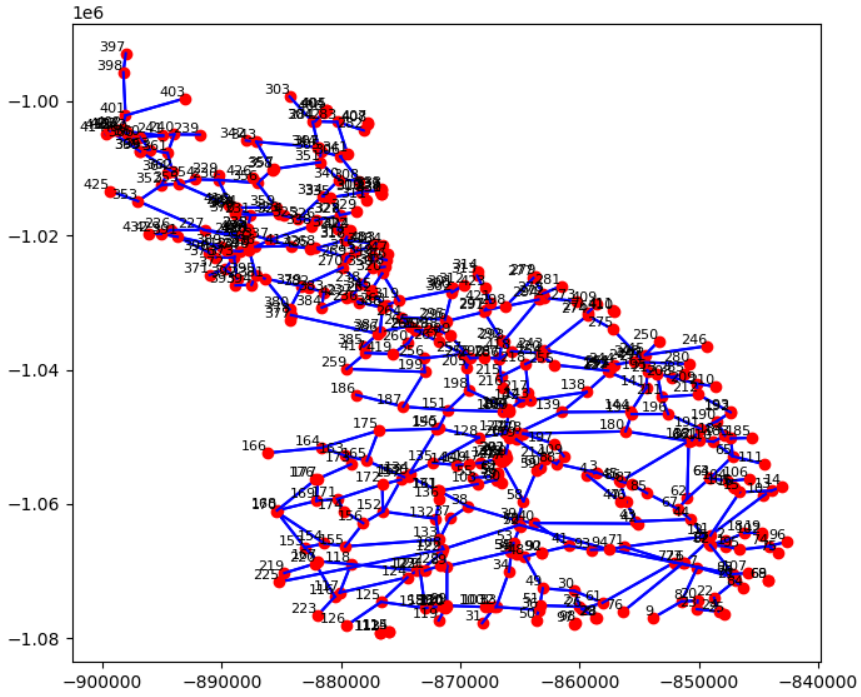
Z tohoto formátu je síť načítána funkcí *lines2graph_gpkg* a převádí ho na dva slovníky, jeden obsahující graf, který obsahuje ke každému vrcholu, který má celočíselný index (ten začíná od

1 a jde až do N = počet vrcholů) a k němu odpovídající indexy sousedních vrcholů a cena cesty (vzdálenost, čas,...) k nim.

Názvy obcí a souřadnice jejich definičních bodů byly získány obdobnou cestou jako silniční síť pomocí skriptu s funkcemi *obce_load.py*.

3 Výsledky

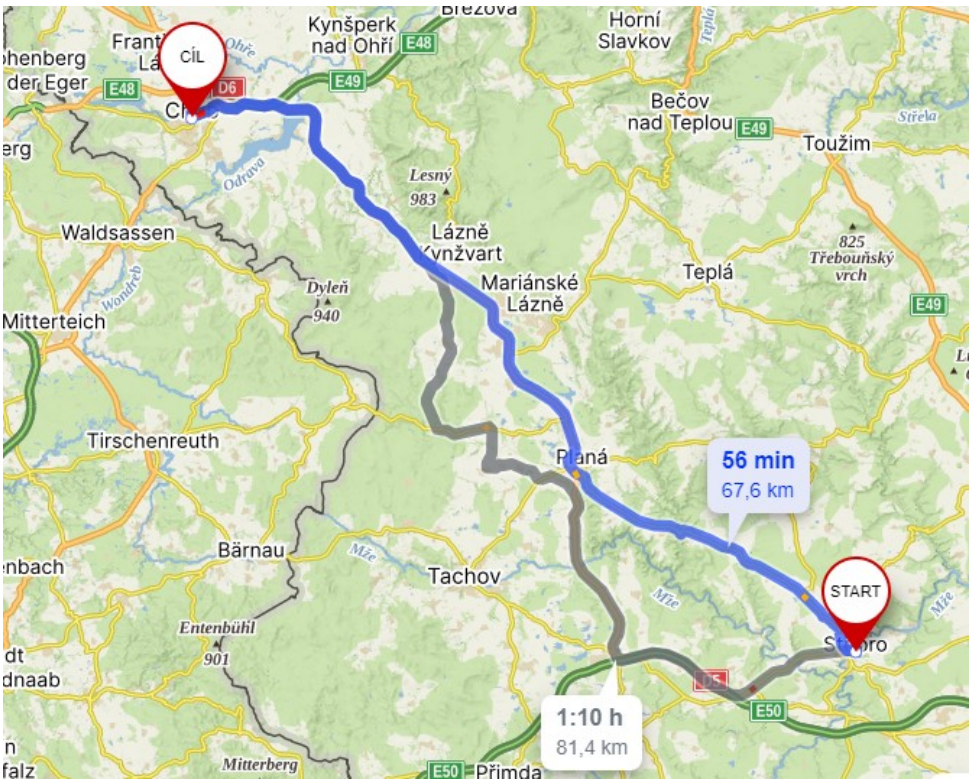
3.1 Vizualizace sítě



Obrázek 1: Vizualizace silniní sítě okresů Cheb a Tachov

3.2 Trasa zvolená pro testování

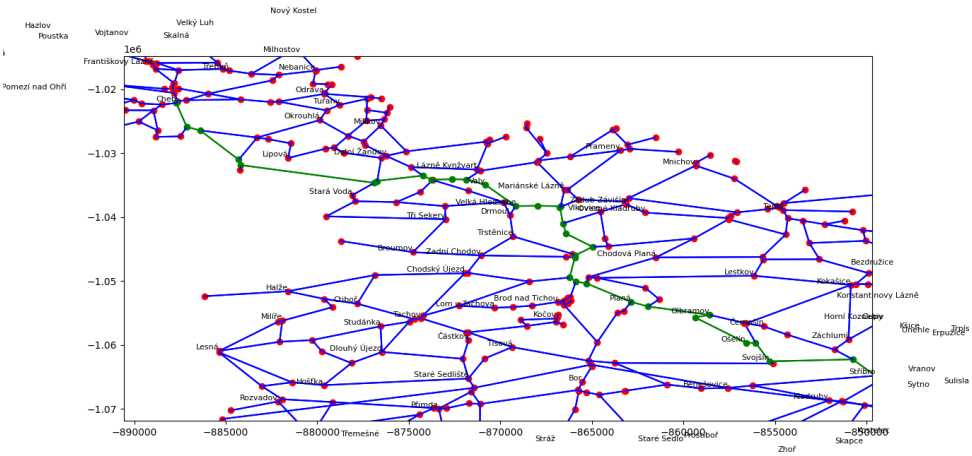
Stříbro ⇒ Cheb



Obrázek 2: Navrhované trasy ze služby Mapy.cz

3.3 Cesta pomocí nejkratší vzdálenosti

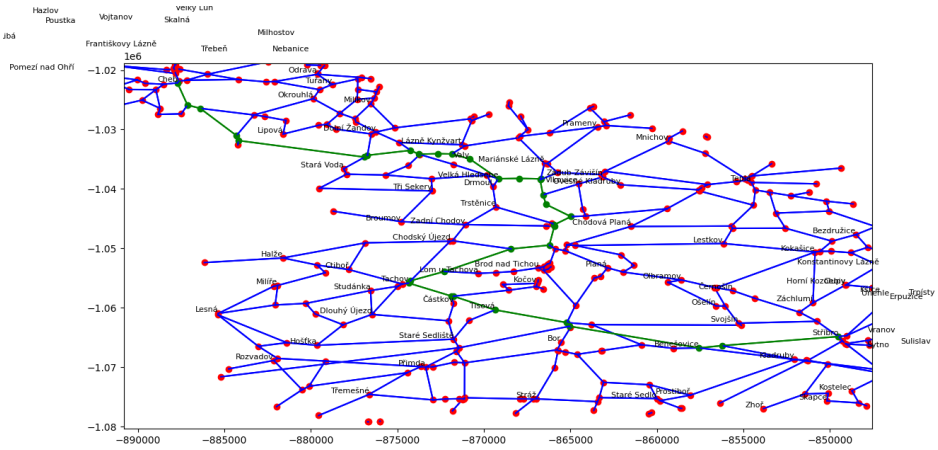
Vrcholy přes které vede cesta:
396, 395, 381, 380, 378, 386, 387, 265, 266, 299, 288, 289, 258, 290, 287, 286, 215, 216, 142, 207, 189, 127, 206, 179, 214, 88, 3, 4, 47, 46, 43, 44, 12
Vzdálenost ujetá po nejkratší cestě:
cca. 55.4 km



Obrázek 3: Nejkratší cesta

3.3.1 Cesta pomocí nejmenšího času

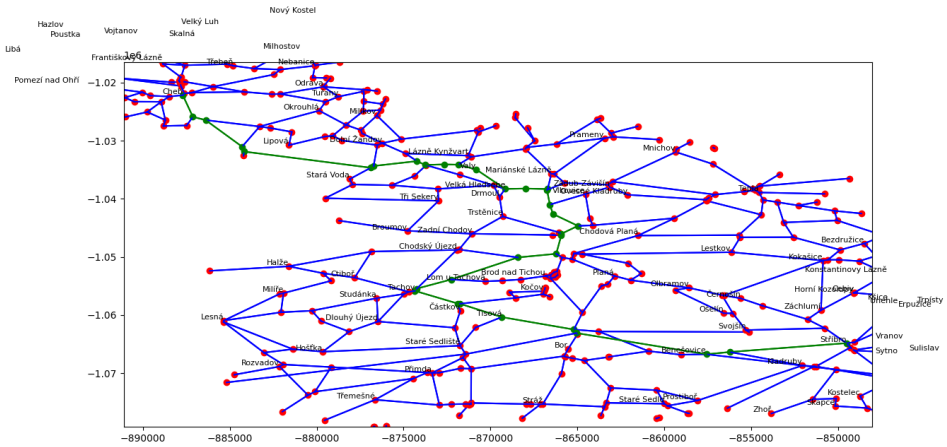
Vrcholy přes které vede cesta:
396, 395, 381, 380, 378, 386, 387, 265, 266, 299, 288, 289, 258, 290, 287, 286, 215, 216, 142, 207, 189, 127, 128, 135, 134, 130, 131, 161, 38, 39, 78, 94, 71, 12
Doba jízdy po nejrychlejší cestě:
cca. 49 minut



Obrázek 4: Nejrychlejší cesta

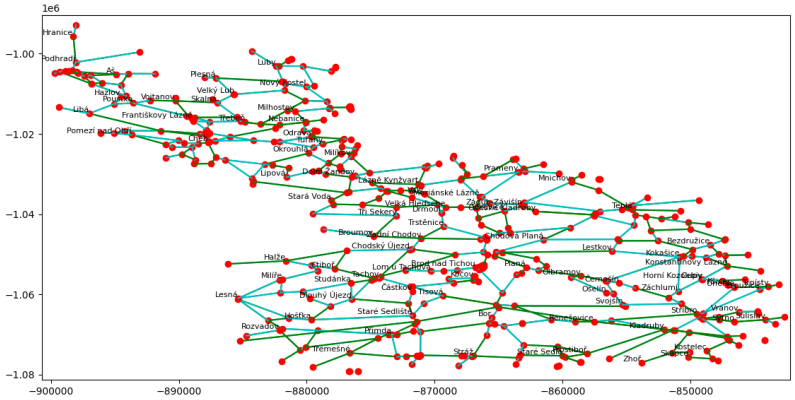
3.3.2 Cesta pomocí nejmenšího času s uvážením klikatosti

Vrcholy přes které vede cesta:
396, 395, 381, 380, 378, 386, 387, 265, 266, 299, 288, 289, 258, 290, 287, 286, 215, 216, 142, 207, 189, 127, 128, 135, 134, 130, 131, 161, 38, 39, 78, 94, 71, 12
Doba jízdy po nejrychlejší cestě s uvážením křivosti silnic:
cca. 52 minut



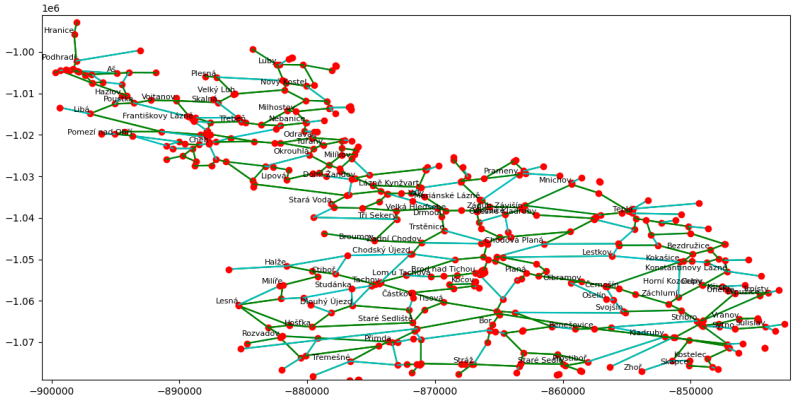
Obrázek 5: Nejrychlejší cesta s uváženou křivostí

3.4 Minimální kostra Borůvka



Obrázek 6: Minimální kostra (zeleně)

3.5 Minimální kostra Jarník



Obrázek 7: Minimální kostra (zeleně

3.6 Nejkratší cesta v grafu se záporným ohodnocením hran

Prezentuje skript *modif_dijkstra.py*.

3.7 Nejkratší cesta v grafu mezi všemi vrcholy

Prezentuje skript *dijkstra_all2all.py*. Z časových důvodů skript prezentuje testovací graf o 9 vrcholech.

4 Závěr

Výsledkem zpracování této úlohy jsou skripty a funkce, které z veřejných dat ve formátu GeoPackage vytvoří graf, na kterém jsou prováděny různé grafové úlohy.

První část úloh se věnovala nalezení nejkratších a nejrychlejších cest. K tomu byl použit Dijkstrův algoritmus a Bellman-Fordův algoritmus pro grafy se záporně ohodnocenými hranami. Fukce cesty vykreslují v rámci silniční sítě, s vrcholy, kterými jsou různá pojmenovaná sídla. Algoritmus byl testován na trase Stříbro-Cheb, a vzdálenost i časová náročnost vypočtená z grafu se blíží hodnotám, které nám předkládá obdobná služba serveru Mapy.cz.

Druhá část byla zaměřena na hledání minimálních koster. Ty nacházeli Borůvkův a Jarníkův algoritmus. Při použití na reálných datech však bylo potřeba občas udělat lehké úpravy, jelikož reálná data často produkují nesouvislé grafy. Minimální kostry byly vypočteny s ohodnoceným grafem podle euklidovské vzdálenosti. Minimální kostry byly obdobně jako v první části, vyobrazeny do silniční sítě s pojmenovanými sídly na stejném území.

V Praze dne: 3.12. 2024 **T. Černohousová**

M. Pokorný

Pseudokód pro funkci `lines2graph_gpkg`

Algorithm 1 lines2graph_gpkg

Require: Typ váhy `type_of_weight`: 'Euclidean distance', 'Transport time', nebo 'Transport time with deviality'

Require: Soubor `file`: název geopackage souboru (implicitně 'silnice.gpkg')

Ensure: Graf `G_m` a souřadnice uzlů `C_m`

Načti geopackage soubor pomocí knihovny `fiona`
Inicializuj prázdné slovníky `G`, `C`, a `W`
Definuj očekávané rychlosti podle tříd silnic (`road_classes`)
for každou čáru `line` v síti `net` **do**
    Získej souřadnice začátku a konce čáry z geometrie (`line.geometry`)
    Urči startovní a koncový bod: `start` a `end`
    if `start` není v `C` **then**
        Přidej `start` do `C`
    end if
    if `end` není v `C` **then**
        Přidej `end` do `C`
    end if
    if `start` není v `G` **then**
        Inicializuj `G[start]` jako prázdný seznam
    end if
    if `end` není v `G` **then**
        Inicializuj `G[end]` jako prázdný seznam
    end if
    Přidej `end` do seznamu sousedů `G[start]`
    Přidej `start` do seznamu sousedů `G[end]`
    if `type_of_weight` je 'Euclidean distance' **then**
        Vypočítej váhu `w` jako délku čáry (`line.properties['SHAPE_Length']`)
    else if `type_of_weight` je 'Transport time' **then**
        Vypočítej vzdálenost `d` a rychlost `v` podle třídy silnice
        Vypočítej čas `t = d / v`
        Nastav váhu `w = t`
    else if `type_of_weight` je 'Transport time with deviality' **then**
        Vypočítej čas `t` a poměr `curvature_ratio = d / Euclidean_distance`
        Nastav váhu `w = t * curvature_ratio`
    end if
    if `start` není v `W` **then**
        Přidej `w` do `W[start]`
    end if
    if `end` není v `W` **then**
        Přidej `w` do `W[end]`
    end if
end for

Pseudokód pro funkce `obcefromgpkg`, `obec2node_of_graph`, a `node2obec_of_graph`

Algorithm 2 `obcefromgpkg`

Require: Název souboru `file`, ve kterém je geopackage (implicitně 'obce.gpkg')

Ensure: Slovník `O`, který mapuje názvy obcí na jejich souřadnice

Načti geopackage soubor pomocí knihovny `fiona` a ulož jako `obce`
Inicializuj prázdný slovník `O`
for každá obec `obec` v `obce` **do**
    Získat souřadnice `coords` z `obec.geometry['coordinates']`
    Získat název obce `name` z `obec.properties['NAZ_OBEC']`
    Přidat dvojici klíč-hodnota `(name, coords)` do `O`
end for
return `O`

Algorithm 3 `obec2node_of_graph`

Require: Slovník `O` (obce a jejich souřadnice) a slovník `C` (uzly a jejich souřadnice)

Ensure: Slovník `towns_closest_to_points_from_C`, který mapuje uzly na názvy nejbližších obcí

Inicializuj prázdný slovník `towns_closest_to_points_from_C`
for každá obec `town`, její souřadnice `def_coord` v `O` **do**
    Inicializuj `closest_coord` na `None` a `min_distance` na nekonečno
    for každý uzel `id`, jeho souřadnice `coord` v `C` **do**
        Spočítej vzdálenost `distance` mezi `def_coord` a `coord`
        if `distance` $<$ `min_distance` **then**
            Aktualizuj `min_distance` na `distance`
            Aktualizuj `closest_coord` na `id`
        end if
    end for
    Přidej dvojici `(closest_coord, town)` do `towns_closest_to_points_from_C`
end for
return `towns_closest_to_points_from_C`

Algorithm 4 `node2obec_of_graph`

Require: Slovník `O` (obce a jejich souřadnice) a slovník `C` (uzly a jejich souřadnice)

Ensure: Slovník `towns_closest_to_points_from_C`, který mapuje názvy obcí na nejbližší uzly

Inicializuj prázdný slovník `towns_closest_to_points_from_C`
for každá obec `town`, její souřadnice `def_coord` v `O` **do**
    Inicializuj `closest_coord` na `None` a `min_distance` na nekonečno
    for každý uzel `id`, jeho souřadnice `coord` v `C` **do**
        Spočítej vzdálenost `distance` mezi `def_coord` a `coord`
        if `distance` $<$ `min_distance` **then**
            Aktualizuj `min_distance` na `distance`
            Aktualizuj `closest_coord` na `id`
        end if
    end for
    Přidej dvojici `(town, closest_coord)` do `towns_closest_to_points_from_C`
end for
return `towns_closest_to_points_from_C`

Pseudokód pro funkce `dijkstra`, `rPath`, a další

Algorithm 5 `rPath`

Require: Začátek cesty `u`, konec cesty `v`, seznam předchůdců `P`

Ensure: Seznam `path`, obsahující body na cestě mezi `u` a `v`

Inicializuj prázdný seznam `path`
while `v` $\neq$ `u` **and** `v` $\neq -1$ **do**
    Přidej `v` do `path`
    Aktualizuj `v` na `P[v]`
end while
Přidej `v` do `path`
if `v` $\neq$ `u` **then**
    Vypiš `'Incorrect path'`
end if
return `path`

Algorithm 6 `sum_of_weights`

Require: Cesta `path`, graf `G`

Ensure: Celková váha `w` cesty

Inicializuj váhu `w` na 0
for `i` od 0 do `len(path) - 2` **do**
    Přičti váhu `G[path[i]][path[i+1]]` k `w`
end for
return `w`

Algorithm 7 `dijkstra`

Require: Graf `G`, startovní bod `start`

Ensure: Seznam předchůdců `P`, vzdálenosti `d` od `start`

Inicializuj délku grafu `n` na `len(G) + 1`
Inicializuj vzdálenosti `d` na nekonečno (`inf`) a předchůdce `P` na `-1`
Inicializuj prioritní frontu `PQ`
Přidej do `PQ` dvojici `(0, start)`
Nastav `d[start]` na 0
while `PQ` není prázdná **do**
    Získej `du, u` z `PQ`
    for každého souseda `v` a jeho váhu `wuv` v `G[u]` **do**
        if `d[v]` > `d[u]` + `wuv` **then**
            Aktualizuj `d[v]` na `d[u]` + `wuv`
            Nastav `P[v]` na `u`
            Přidej `(d[v], v)` do `PQ`
        end if
    end for
end while
return `P, d`

Algorithm 8 Hlavní skript

Načti graf `G` a souřadnice `C` pomocí funkce `lines2graph_gpkg`
Načti obce `O` a jejich uzly `OC` pomocí funkcí `obcefromgpkg` a `obec2node_of_graph`
Spusť algoritmus Dijkstry na grafu `G` se startem v bodě `83`, ulož `P, D`
Získej cestu `path` pomocí `rPath(83, 268, P)`
Spočítej náklady `cost` cesty pomocí `sum_of_weights(path, G)`
Graficky zobraz graf a vypočtenou cestu

Pseudokód pro Borůvkův algoritmus a pomocné funkce

Algorithm 9 `dict2lists`

Require: Graf `G` v podobě slovníku

Ensure: Vrcholy `V` a hrany `E` ve formě seznamů

1: Inicializuj seznam vrcholů `V` jako klíče slovníku `G`
2: Inicializuj prázdný seznam hran `E`
3: **for** každý vrchol `v` v `V` **do**
4: **for** každého souseda `k` vrcholu `v` **do**
5: Přidej hranu `[v, k, G[v][k]]` do `E`
6: **end for**
7: **end for**
8: **return** `V, E`

Algorithm 10 `find`

Require: Vrchol `u`, pole předchůdců `P`

Ensure: Kořen komponenty obsahující `u`

1: **while** `P[u]` $\neq$ `u` **do**
2: Nastav `u = P[u]`
3: **end while**
4: **return** `u`

Algorithm 11 `union`

Require: Vrcholy `u`, `v`, pole předchůdců `P`

Ensure: Spojení komponent obsahujících `u` a `v`

1: Najdi kořeny komponent `u` a `v` pomocí `find`
2: **if** kořeny nejsou shodné **then**
3: Spoj komponenty: nastav `P[root_v] = root_u`
4: **end if**

Algorithm 12 `make_set`

Require: Vrchol `u`, pole předchůdců `P`

Ensure: Inicializace komponenty obsahující pouze `u`

1: Nastav `P[u] = u`

Algorithm 13 `boruvka`

Require: Graf `G` ve formě slovníku

Ensure: Minimální kostra `T` a její váha `wt`

Transformuj graf pomocí `dict2lists(G)` na vrcholy `V` a hrany `E`
Inicializuj prázdný seznam `T` a váhu `wt = 0`
Inicializuj pole předchůdců `P` velikosti `|V| + 1` s hodnotami `inf`
for každý vrchol `v` v `V` **do**
    Inicializuj množinu pro `v` pomocí `make_set(v, P)`
end for
Seřaď hrany `E` podle jejich vah
for každou hranu `[u, v, w]` v `E` **do**
    if `find(u, P)` $\neq$ `find(v, P)` **then**
        Spoj komponenty pomocí `union(u, v, P)`
        Přidej hranu `[u, v, w]` do `T`
        Aktualizuj váhu: `wt = wt + w`
    end if
end for
return `T, wt`

Algorithm 14 Hlavní skript

Načti graf `G` a souřadnice `C` pomocí `lines2graph_gpkg`
Načti obce `O` a jejich uzly `OC` pomocí funkcí `obcefromgpkg` a `obec2node_of_graph`
Spusť Borůvkův algoritmus na grafu `G`, výsledky ulož do `T, weight`
Vypiš minimální kostru `T` a celkovou váhu `weight`
Graficky zobraz graf a minimální kostru (zelená pro hrany v kostře, modrá pro ostatní)

Pseudokód pro Primův algoritmus (Jarník) a pomocné funkce

Algorithm 15 `jarnik`

Require: Graf `G` ve formě slovníku, počáteční vrchol `start`

Ensure: Minimální kostra `T` a její váha `wt`

Inicializuj prázdný seznam hran `T`, váhu `wt = 0`
Inicializuj množinu navštívených vrcholů `visited` = {}, nenavštívených vrcholů `unvisited` = $\{$`v` $\in G\}$ *Nastav počáteční vrchol* `current_node = start`
Inicializuj počitadlo komponent `count_of_components = 0`
while `|visited| < |G|` **do**
  if `current_node` není v `visited` **then**
    Přidej `current_node` do `visited`, odeber z `unvisited`
  end if
  Inicializuj `min_edge = None`
  for každý vrchol `u` v `visited` **do**
    for každého souseda `v` vrcholu `u` s váhou hrany `w` **do**
      if `v` není v `visited` a (`min_edge` je prázdná nebo `w < min_edge[2]`) **then**
        Nastav `min_edge = (u, v, w)`
      end if
    end for
  end for
  if `min_edge` **then**
    Přidej hranu `min_edge` do `T`, přičti její váhu k `wt`
    Přidej cílový vrchol `v` z `min_edge` do `visited`, odeber z `unvisited`
    Nastav `current_node = v`
  else
    Vypiš varování o nespojitosti grafu
    Zvýši `count_of_components` o 1
    if `unvisited` není prázdné **then**
      Vyber náhodný vrchol z `unvisited` jako `current_node`
    else
      Přerušení cyklu
    end if
  end if
end while
Vypiš počet nespojených komponent `count_of_components + 1`
return `T, wt`

Algorithm 16 Hlavní skript

Načti graf `G` a souřadnice `C` pomocí `lines2graph_gpkg`
Načti obce `O` a jejich uzly `OC` pomocí funkcí `obcefromgpkg` a `obec2node_of_graph`
Spusť Primův algoritmus na grafu `G`, výsledky ulož do `T, weight`
Vypiš minimální kostru `T` a celkovou váhu `weight`
Graficky zobraz graf a minimální kostru:
for každý vrchol `node` a sousedy `neighbors` v `G` **do**
    Získej souřadnice `(x, y)` a `(nx, ny)` pro každou hranu
    if hrana patří do `T` **then**
        Zobraz ji zeleně
    else
        Zobraz ji modře
    end if
    Označ vrchol jako červený bod
    Přidej textový popisek (pokud existuje) na základě `OC[node]`
end for
Zobraz graf pomocí `plt.show()`