

3. Nejkratší cesta grafem

Matěj Klimeš, Tomáš Zbíral
ZS 2024/25, číslo skupiny: 2, datum zpracování: 30.11.2024

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 vybraném 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 $P = \{p_i\}_{i=1}^n$ 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.

1.1 Řešené bonusové úlohy

Nad rámec zadání byly řešeny tyto bonusové úlohy:

- Řešení úlohy pro grafy se záporným ohodnocením
- Nalezení nejkratších cest mezi všemi dvojicemi uzlů
- Nalezení minimální kostry Kruskal
- Nalezení minimalní kostry Prim
- Využití heuristiky Weighted Union
- Využití heuristiky Path Compression

2 Teoretický úvod

Graf je matematická struktura, která popisuje vztahy mezi objekty. Skládá se z množiny uzlů (U) a hran (H), které spojují dvojice uzlů. Grafy se využívají v různých oblastech, například při plánování tras, analýze sítí nebo navigaci. Grafy lze dělit dle orientace hran na:

- **Neorientované grafy:** Hrany nemají směr, spojení mezi uzly je obousměrné.
- **Orientované grafy:** Hrany mají směr, což znamená, že spojení mezi uzly je jednosměrné.
- **Částečně orientované grafy:** Některé hrany jsou orientované, zatímco jiné nejsou.

Grafy lze dále dělit podle toho, zda mají jejich hrany přiřazeny váhy:

- **Ohodnocené grafy:** Hrany mají přiřazeny váhy, které vyjadřují vzdálenost, čas nebo jinou charakteristiku.
- **Neohodnocené grafy:** Hrany nemají přiřazeny váhy, všechny jsou považovány za rovnocenné.

Jednou z nejznámějších úloh je hledání **nejkratší cesty** mezi dvěma uzly grafu, což je problém, který lze efektivně řešit pomocí **Dijkstra algoritmu**. Další důležitou úlohou je nalezení **minimální kostry grafu**, což lze řešit pomocí algoritmů jako je **Kruskalův algoritmus** nebo **Primův algoritmus**. Pro nalezení **nejkratších cest** mezi všemi dvojicemi uzlů v grafu je využíván **Floyd-Warshallův algoritmus**.

2.1 Dijkstra algoritmus

Dijkstra algoritmus je greedy algoritmus, který se používá k hledání nejkratší cesty mezi dvěma uzly grafu. Graf může být orientovaný i neorientovaný, ale hrany musí mít kladné váhy. Algoritmus postupuje tak, že vždy zpracuje uzel s nejmenší známou vzdáleností (vahou) od počátečního uzlu a následně aktualizuje vzdálenosti (vahy) ke všem jeho sousedům. Průběh algoritmu je následující:

- Nejprve se všem uzlům nastaví ohodnocení na ∞ (nekonečno), kromě počátečního uzlu, jehož váha je 0. (Inicializace)
- Využívá se prioritní fronta (priority queue), která vždy vybírá uzel s nejmenším ohodnocením k prozkoumání.
- Pro každého souseda aktuálního uzlu se vypočítá potenciální nové ohodnocení (součet ohodnocení aktuálního uzlu a hrany).
- Pokud je nové ohodnocení menší než dříve uložené, dojde k jeho aktualizaci a soused je přidán do prioritní fronty. (Relaxace)

Matematicky lze relaxaci vyjádřit následovně pro hranu $u \rightarrow v$ s váhou $w(u, v)$:

$$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w(u, v))$$

Kde:

- $\text{dist}[v]$ je aktuální odhad vzdálenosti do uzlu v ,
- $\text{dist}[u]$ je aktuální odhad vzdálenosti do uzlu u ,
- $w(u, v)$ je váha hrany mezi uzly u a v .

- Algoritmus končí, když jsou všechny uzly zpracovány nebo pokud je dosažen cílový uzel.

Výstupem algoritmu je:

1. Seznam uzlů tvořících nejkratší cestu z počátečního uzlu do cílového uzlu (seznam předchůdců).
2. Celkové ohodnocení této cesty.

Bellman-Fordův algoritmus

Bellman-Fordův algoritmus slouží k hledání nejkratší cesty v grafu, který může obsahovat hrany se zápornými vahami. Algoritmus pracuje na principu relaxace hran, kdy se postupně zlepšují odhadované vzdálenosti k uzlům. Průběh algoritmu je následující:

- Váha počátečního uzlu je nastavena na 0, zatímco ohodnocení všech ostatních uzlů je nekonečno (∞).
- Algoritmus opakuje $V - 1$ iterací (kde V je počet uzlů). V každé iteraci se pro každou hranu zkouší, zda lze vzdálenost (ohodnocení) ke koncovému uzlu zkrátit (zlepšit). Pokud ano, aktualizuje vzdálenost a předchůdce.
- Po $V - 1$ iteracích se provede další kontrola hran. Pokud lze vzdálenosti stále zlepšit, graf obsahuje záporný váhový cyklus.
- Nejkratší cesta k cílovému uzlu je sestavena zpětně pomocí seznamu předchůdců.

Výstupem algoritmu je:

1. Seznam uzlů tvořících nejkratší cestu z počátečního uzlu do cílového uzlu (seznam předchůdců).
2. Celkové ohodnocení této cesty.

2.2 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus je metoda pro nalezení nejkratších cest mezi všemi dvojicemi uzlů v grafu. Funguje na principu dynamického programování a zvládá i grafy se zápornými vahami hran, pokud neobsahují záporné cykly. Průběh algoritmu je následující:

- Pro každý uzel u a v je vzdálenost $\text{dist}[u][v]$ nastavena na váhu hrany $u \rightarrow v$, pokud existuje, nebo na ∞ , pokud hrana neexistuje. Pro uzly samotné platí $\text{dist}[u][u] = 0$.
- Algoritmus postupně zvažuje každý uzel k jako prostřední bod cesty. Pro všechny dvojice uzlů i a j se ohodnocení aktualizuje podle vztahu:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]).$$

Výstupem Floyd-Warshallova algoritmu je matice vzdáleností dist , která obsahuje nejkratší vzdálenosti mezi všemi dvojicemi uzlů v grafu.

2.3 Kruskalův algoritmus

Kruskalův algoritmus je metoda pro nalezení minimální kostry grafu (minimal spanning tree). Postupuje tak, že vybírá hrany podle jejich vah a přidává je do výsledné kostry, pokud tím nevytváří cyklus. Průběh algoritmu je následující:

- Načtou se všechny hrany grafu a seřadí se podle jejich vah vzestupně.
- Pro každý uzel se vytvoří disjunktní množina pomocí datové struktury Union-Find.
- Postupně se vybírají hrany ze seřazeného seznamu.
- Pokud vybraná hrana propojuje dvě oddělené komponenty (tj. nevytváří cyklus), přidá se do minimální kostry a komponenty se sloučí.
- Proces končí, když minimální kostra obsahuje $V - 1$ hran, kde V je počet uzlů v grafu.

Výstupem algoritmu je seznam hran tvořících minimální kostru grafu.

2.4 Jarníkův (Primův) algoritmus

Jarníkův (Primův) algoritmus je algoritmus pro nalezení minimální kostry neorientovaného grafu, který začíná u náhodného uzlu a postupně přidává do kostry nejlevnější hrany, které spojují již vybraný uzel s novým uzlem. Průběh algoritmu je následující:

- Algoritmus začíná u libovolného uzlu grafu (např. první uzel z grafu).
- Vytvoříme prázdnou množinu navštívených uzelů a prázdný seznam hran minimální kostry (MST).
- Používáme prioritní frontu (pq) pro výběr nejlevnější hrany.
- Z fronty vybereme hranu s nejnižší váhou.
- Pokud cílový uzel ještě nebyl navštíven, přidáme tuto hranu do minimální kostry.
- Přidáme všechny sousední uzly do fronty s ohodnocením váhy příslušné hrany.
- Proces pokračuje, dokud všechny uzly nejsou navštíveny.

Výstupem algoritmu je seznam hran tvořících minimální kostru grafu.

2.5 Heuristika Weighted Union a Path Compression

Weighted Union: Tato heuristika minimalizuje výšku stromu při spojování dvou množin. Při sjednocování dvou množin porovnává počet uzelů (velikost množiny) nebo hloubku stromu. Menší množina je vždy připojena jako podstrom větší množiny, což snižuje pravděpodobnost vzniku hlubokých stromů.

Path Compression: Tato heuristika urychluje operaci hledání zástupce množiny (*find*). Při procházení stromu jsou všechny uzly na cestě k zástupci přímo připojeny k němu. To zajistí, že všechny následné operace *find* budou mít téměř konstantní časovou složitost.

3 Pracovní postup

Pro řešení úlohy byla použita data z databáze ArcČR500 verze 3.3. Konkrétně byly využity vrstvy Silnice_2015 a Obce_body pro oblast okresu Mladá Boleslav.

- Nejprve byla vstupní data exportována pomocí sw ArcGIS Pro do formátu GeoPackage.
- Pro převod dat z GeoPackage do grafové reprezentace byl vytvořen skript geopackage_to_graph.py. Ve skriptu jsou implementovány následující operace:
 - Načtení dat o silnicích a sídlech z GeoPackage souborů pomocí knihovny geopandas.
 - Vytvoření mapování rychlostí silnic na základě jejich tříd.
 - Sísla jsou přiřazena k nejbližšímu bodu na některé linii silnice pomocí funkce `find_nearest_point_on_line`.
 - Sísla jsou přidána do grafu jako vrcholy (uzly) s názvy.
 - Přidání krajních bodů linií silnic jako uzel grafu.
 - Uzlům jsou přiřazeny pozice (souřadnice) a označení, zda jde o sídlo.
 - Pro každou linii silnice je vypočítána:
 - * Délka linie.
 - * Křivost (poměr délky linie k přímé vzdálenosti mezi začátkem a koncem).
Pokud máme křivku, která spojuje bod A a bod B , křivost κ se dá vyjádřit následovně:

$$\kappa = \frac{L}{d}$$

kde:

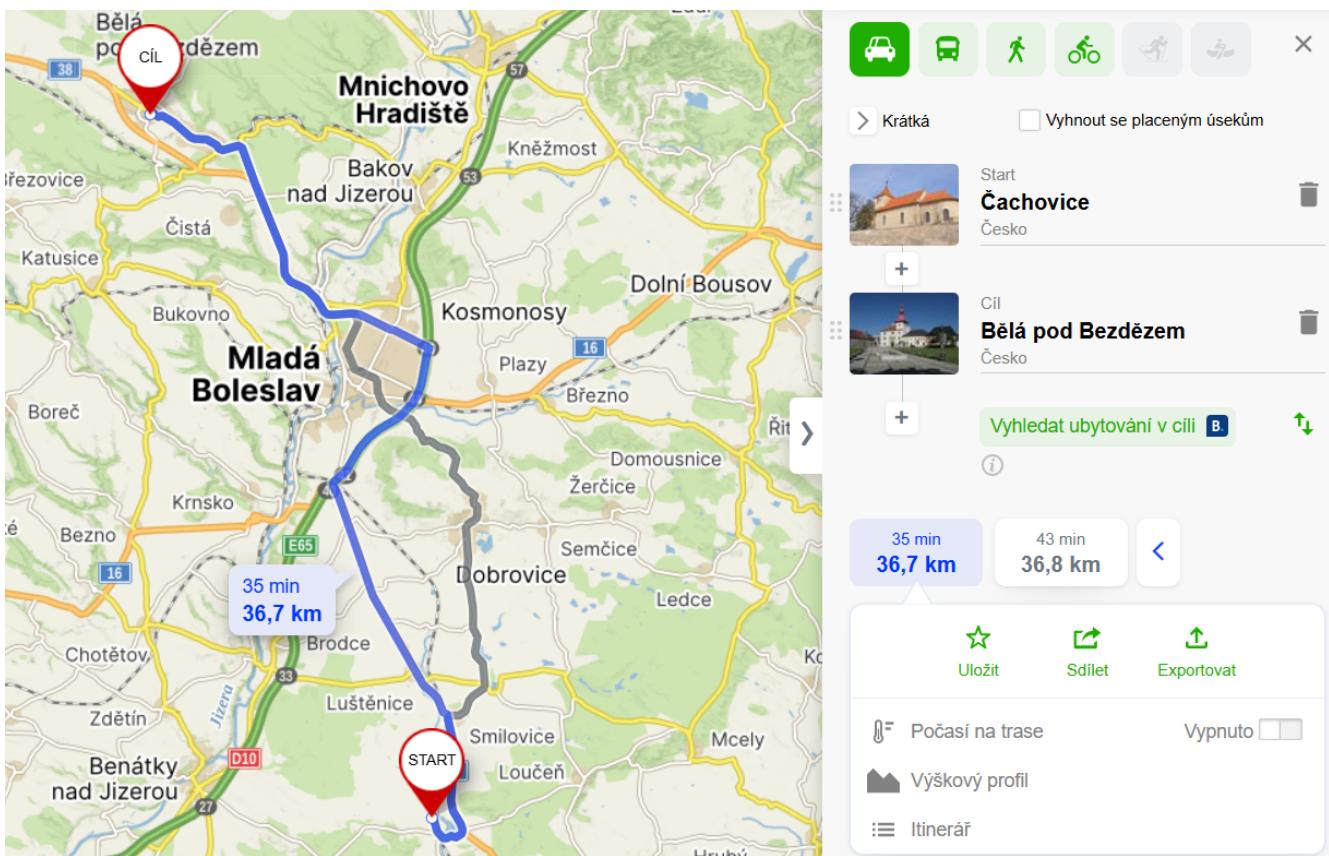
- L je délka křivky (linie),
- d je přímá vzdálenost mezi body A a B (tj. délka přímky spojující tyto body).
- * Rychlosť (na základě třídy silnice).
- * časová náročnost přejetí úseku:

$$\text{čas. nár.} = \frac{\text{délka} \times \text{křivost}}{\text{rychlosť}}$$

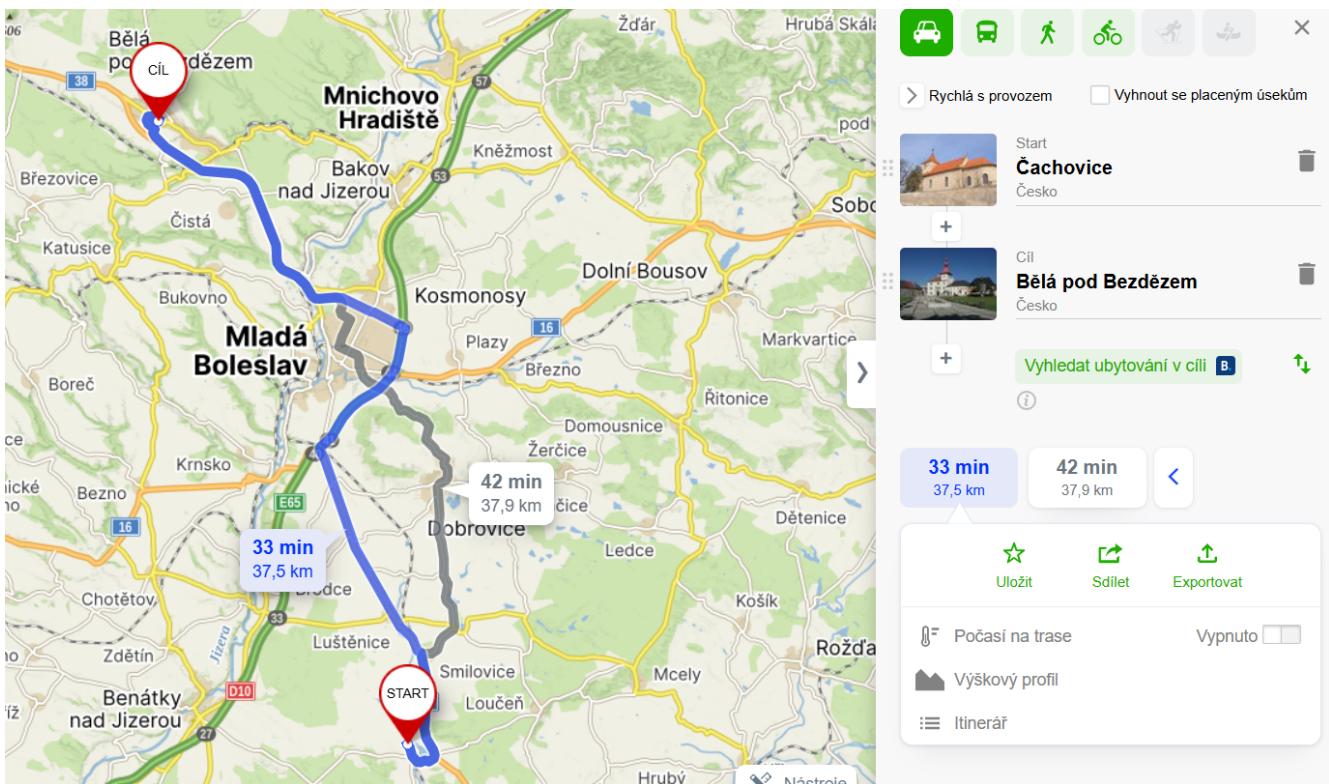
- Hrany jsou přidány do grafu s těmito atributy. Tyto atributy budou dále sloužit k ohodnocení hran.
- Graf je uložen do souboru `graph.pkl` ve formátu pickle pro pozdější použití.
- Dále jsou na vytvořeném grafu počítány algoritmy viz výše.
- Výsledky jsou na závěr vypsány a vizualizovány.

4 Výsledky

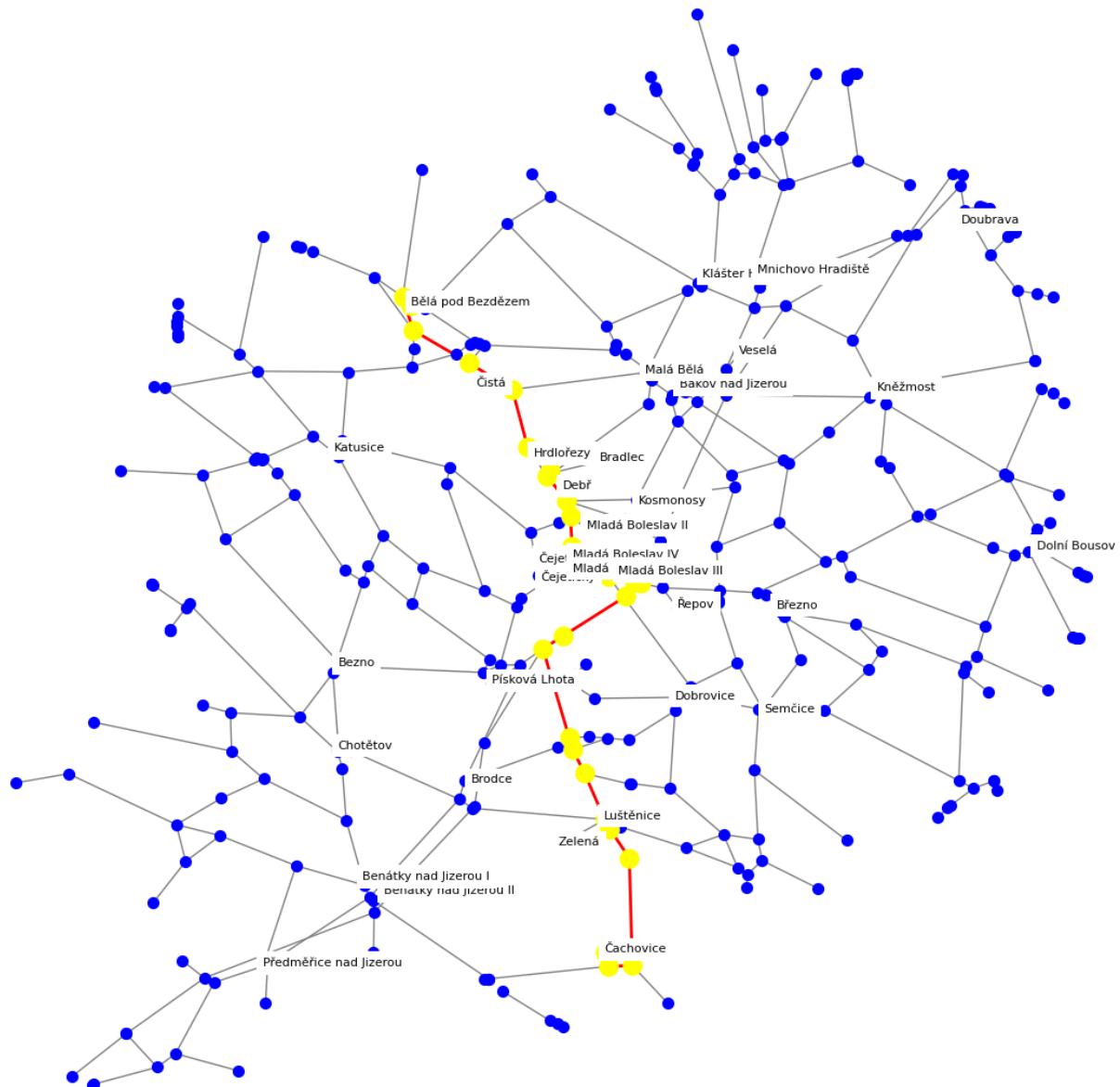
4.1 Čachovice–Bělá pod Bezdězem



Obrázek 1: Nejkratší cesta dle Mapy.cz



Obrázek 2: Nejrychlejší cesta dle Mapy.cz

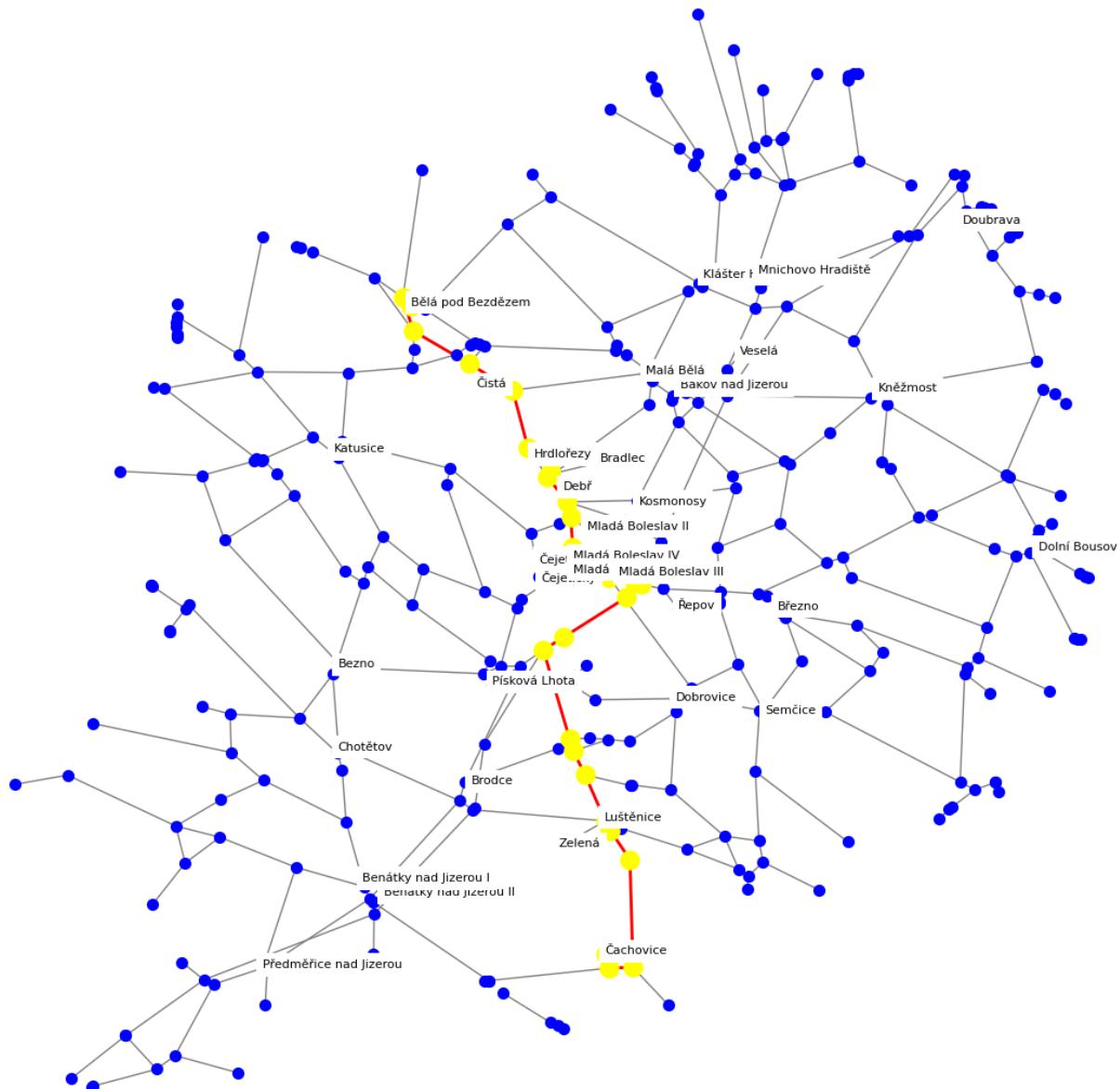


Obrázek 3: Vypočtená nejkratší cesta

Nejkratší cesta mezi 'Čachovice' a 'Bělá pod Bezdězem':

'Čachovice' → 'Node 15' → 'Node 334' → 'Node 338' → 'Node 218' → 'Node 39'
 → 'Node 228' → 'Node 244' → 'Node 242' → 'Node 147' → 'Node 359'
 → 'Node 360' → 'Node 150' → 'Node 149' → 'Node 67' → 'Node 44'
 → 'Node 19' → 'Node 7' → 'Node 145' → 'Node 144' → 'Node 21'
 → 'Node 211' → 'Node 49' → 'Node 214' → 'Node 5' → 'Bělá pod Bezdězem'

Délka cesty: 36.0 km



Obrázek 4: Vypočtená nejrychlejší cesta s uvážením klikatosti

Nejrychlejší cesta mezi 'Čachovice' a 'Bělá pod Bezdězem' s uvážením klikatosti:

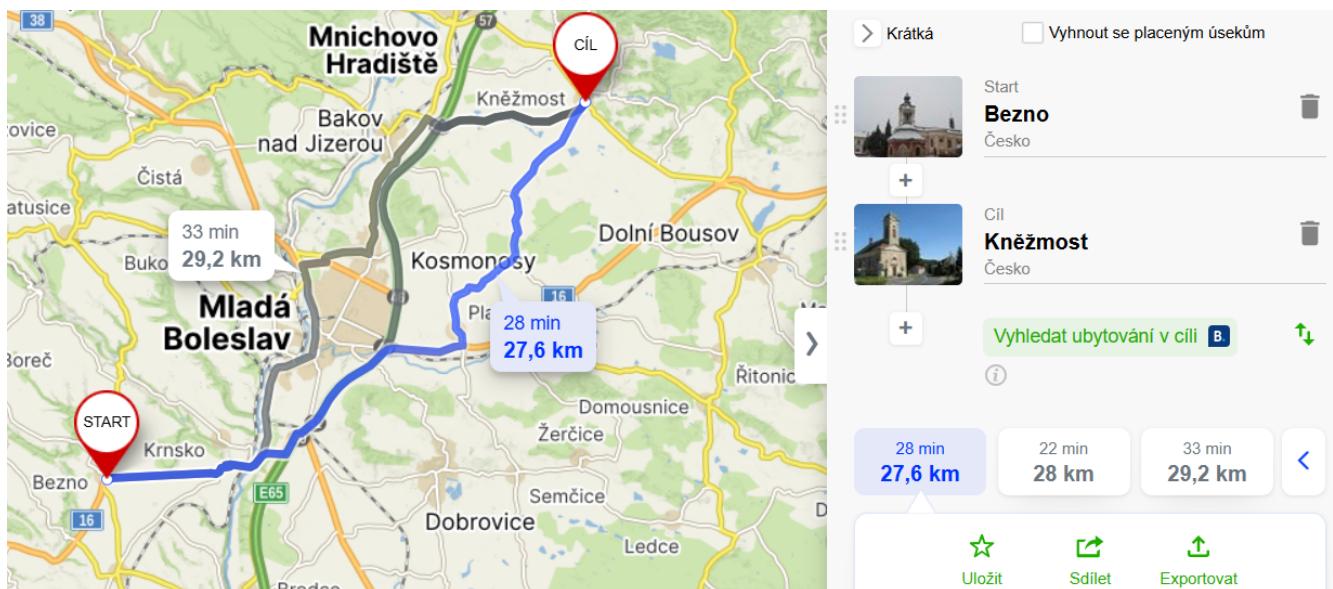
'Čachovice' → 'Node 15' → 'Node 334' → 'Node 338' → 'Node 218' → 'Node 39'
 → 'Node 228' → 'Node 244' → 'Node 242' → 'Node 147' → 'Node 359'
 → 'Node 360' → 'Node 150' → 'Node 149' → 'Node 67' → 'Node 44'
 → 'Node 19' → 'Node 7' → 'Node 145' → 'Node 144' → 'Node 21'
 → 'Node 211' → 'Node 49' → 'Node 214' → 'Node 5' → 'Bělá pod Bezdězem'

Transportní čas: 40 minut

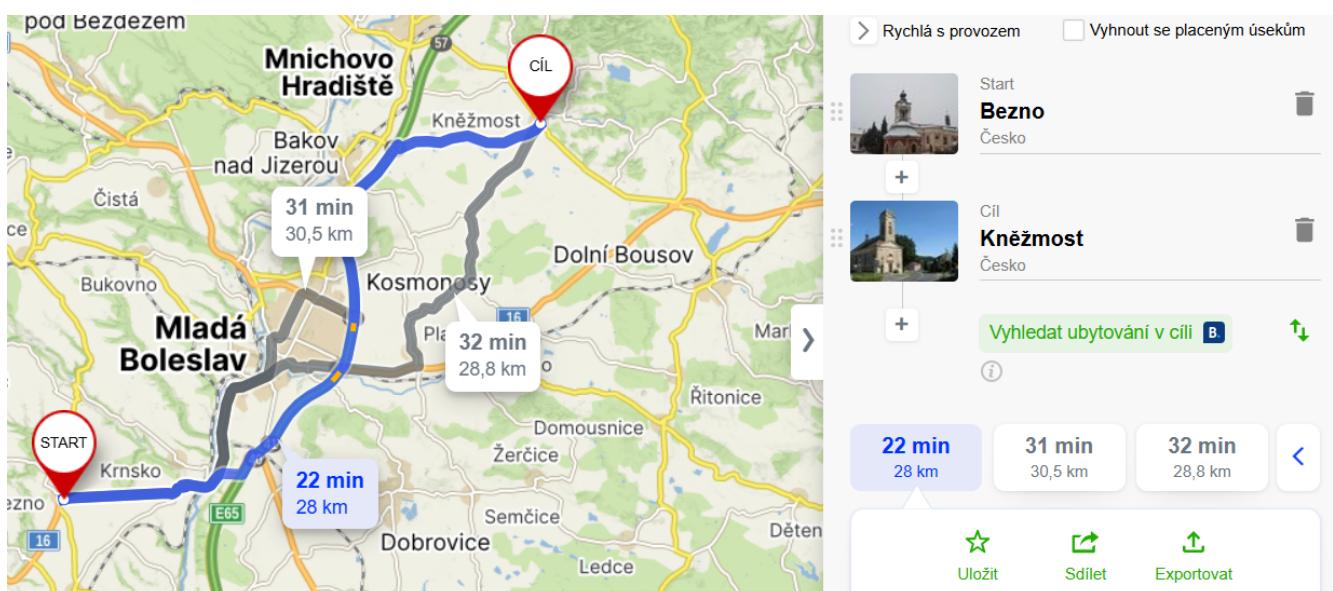
Zdroj	Nejkratší cesta	Nejrychlejší cesta
Vypočteno	36.0 km	40 min
Mapy.cz	36.7 km	33 min

Tabulka 1: Tabulka porovnání vypočtených hodnot se serverem Mapy.cz

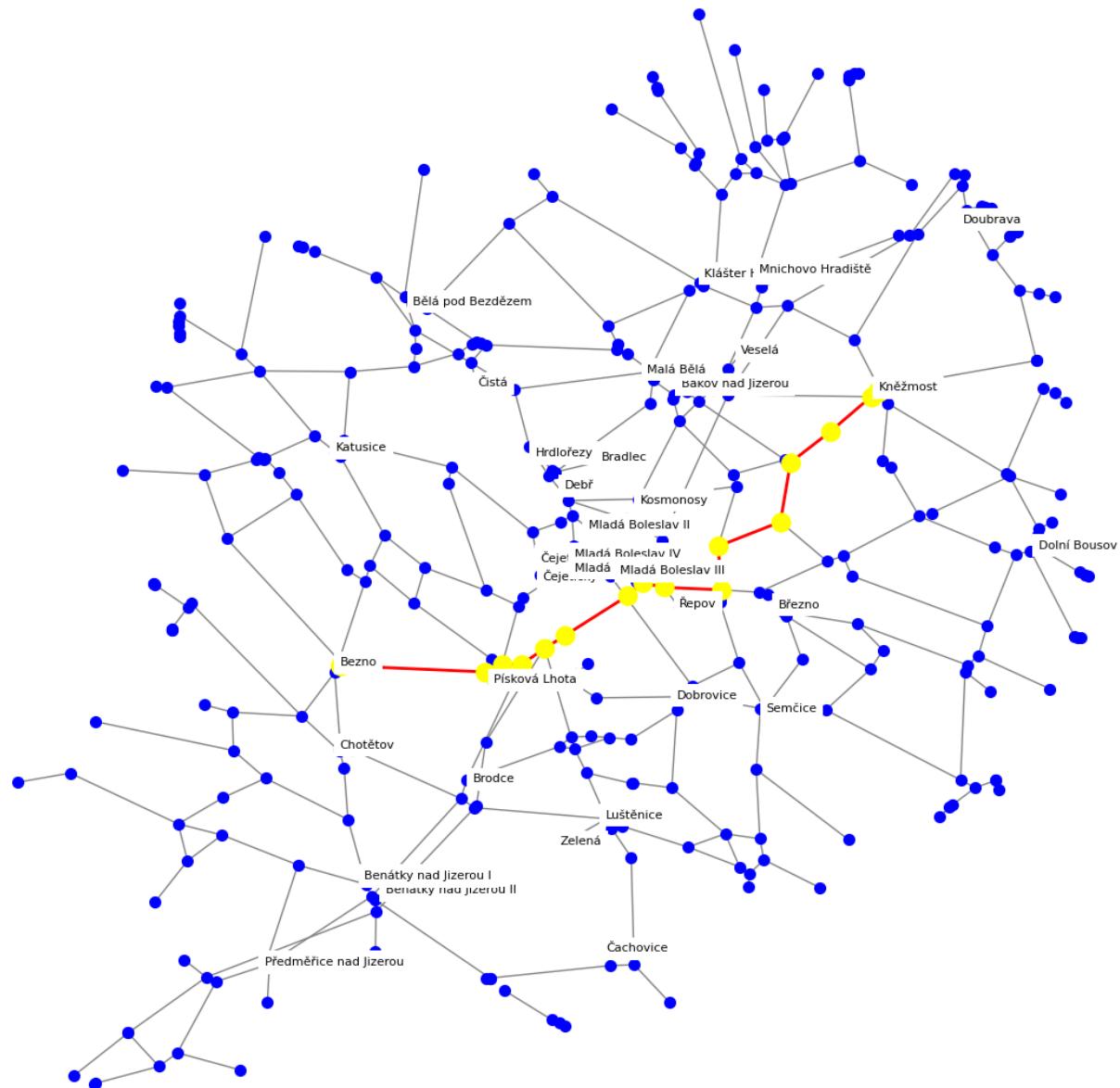
4.2 Bezno–Kněžmost



Obrázek 5: Nejkratší cesta dle Mapy.cz



Obrázek 6: Nejrychlejší cesta dle Mapy.cz

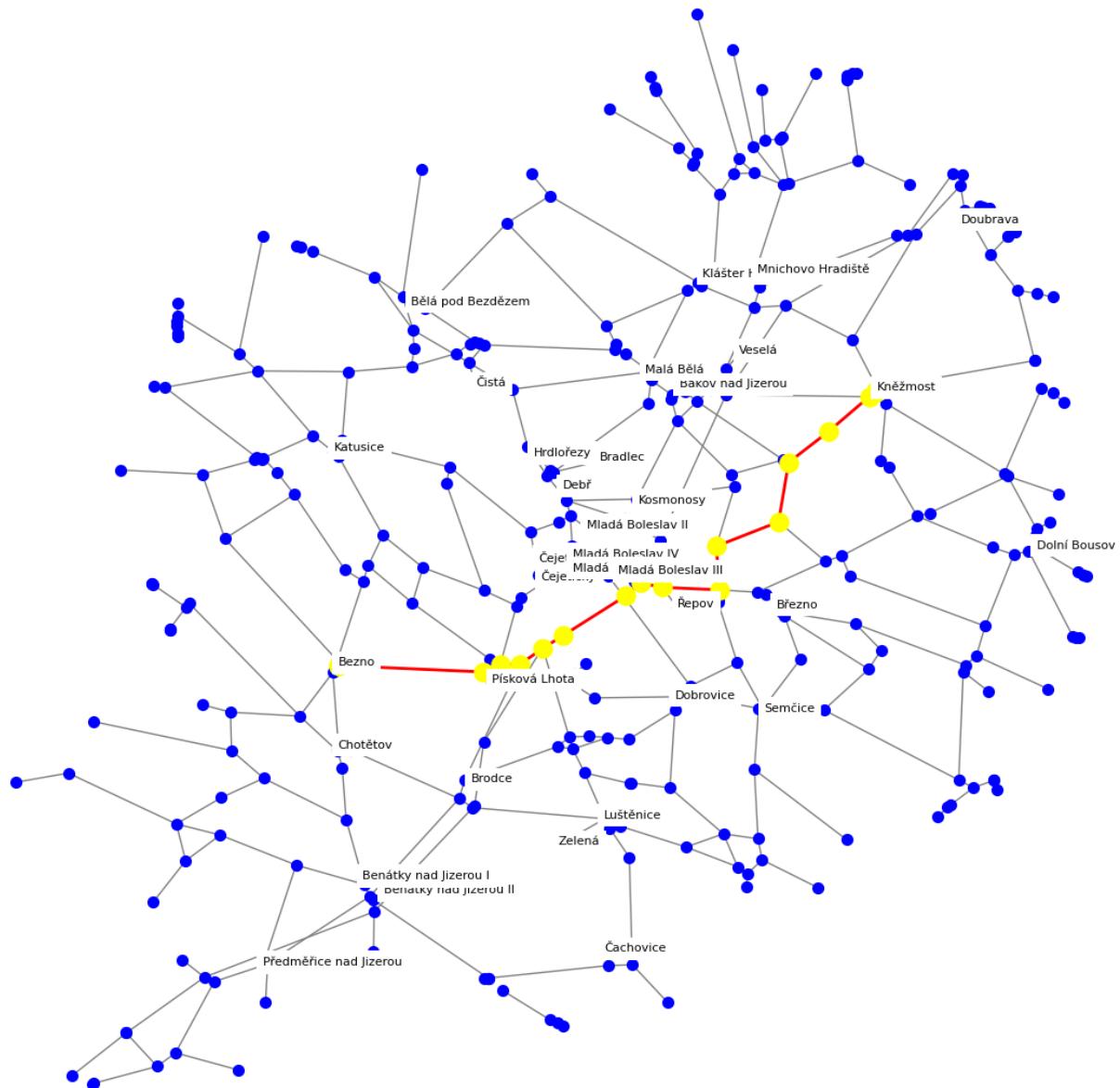


Obrázek 7: Vypočtená nejkratší cesta

Nejkratší cesta mezi 'Bezno' a 'Kněžmost':

'Bezno' → 'Node 29' → 'Node 146' → 'Node 148' → 'Node 11' → 'Node 147'
→ 'Node 359' → 'Node 360' → 'Node 150' → 'Node 35' → 'Node 155'
→ 'Node 258' → 'Node 257' → 'Node 259' → 'Node 260' → 'Node 247'
→ 'Node 57' → 'Kněžmost'

Délka: 26.1 km



Obrázek 8: Vypočtená nejrychlejší cesta s uvážením klikatosti

Nejrychlejší cesta mezi 'Bezno' a 'Kněžmost' s uvážením klikatosti:

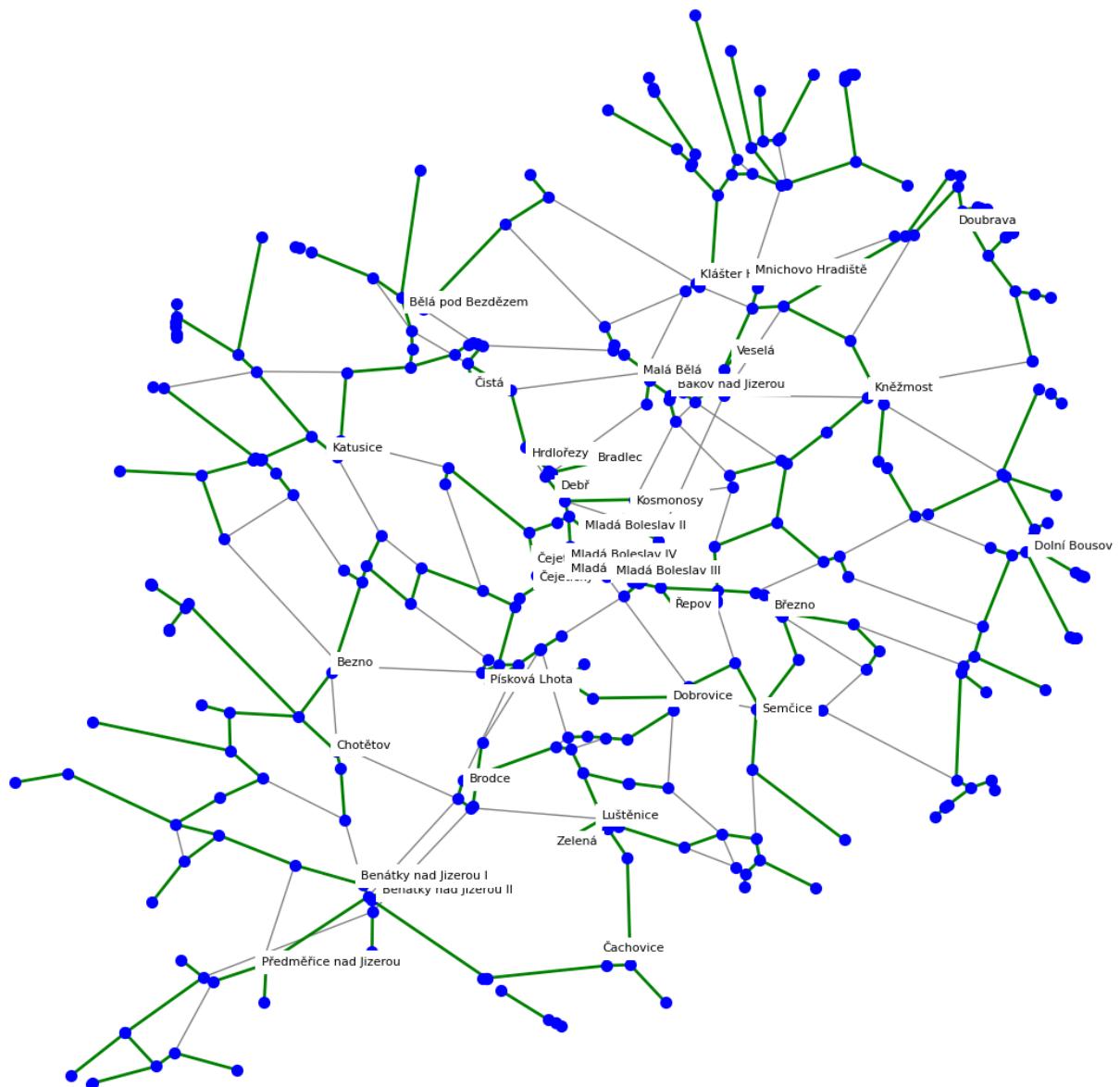
'Bezno' → 'Node 29' → 'Node 146' → 'Node 148' → 'Node 11' → 'Node 147'
 → 'Node 359' → 'Node 360' → 'Node 150' → 'Node 35' → 'Node 155'
 → 'Node 258' → 'Node 257' → 'Node 259' → 'Node 260' → 'Node 247'
 → 'Node 57' → 'Kněžmost'

Transportní čas: 18 minut

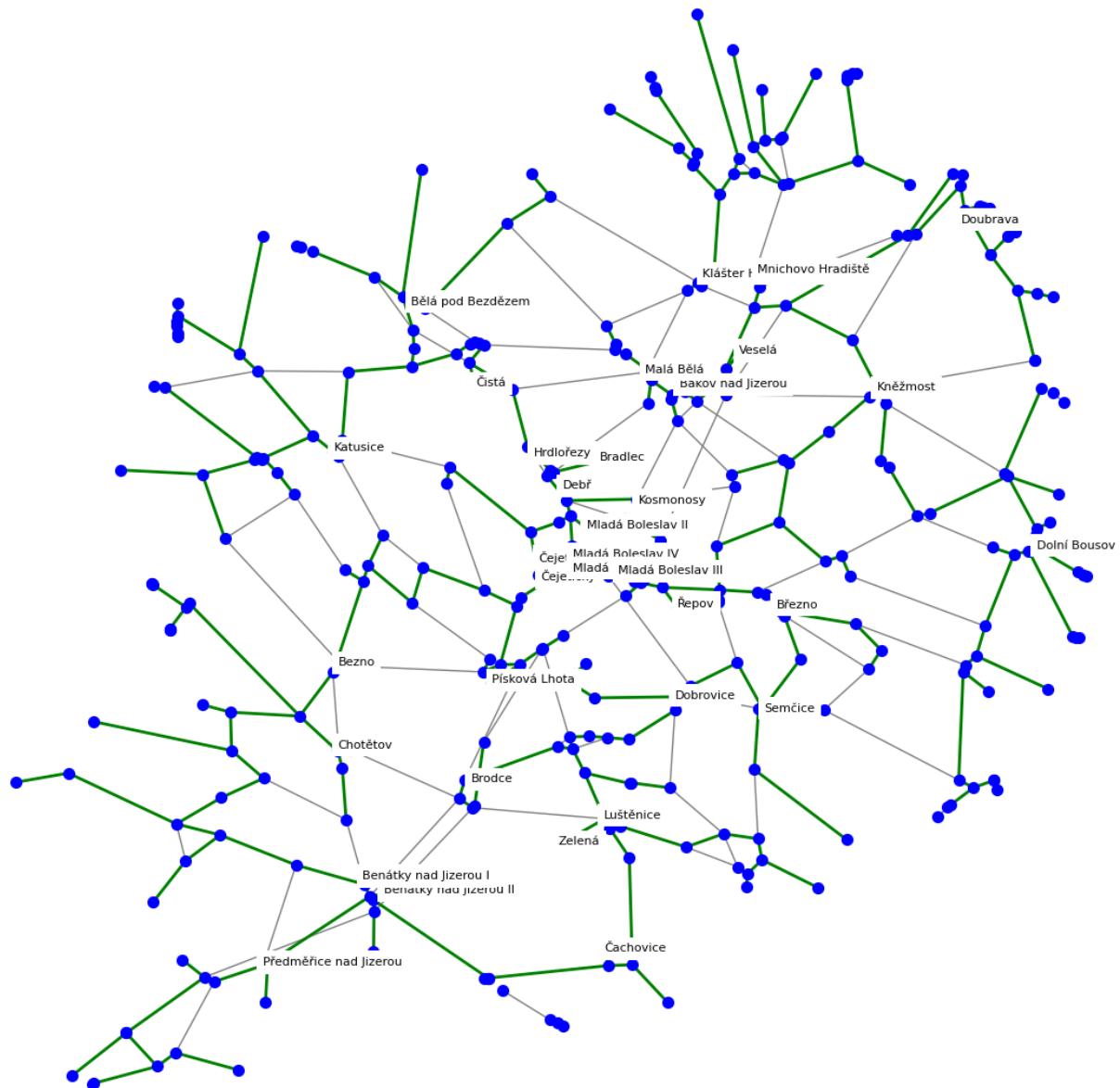
Zdroj	Nejkratší cesta	Nejrychlejší cesta
Vypočteno	26.1 km	18 min
Mapy.cz	27.6 km	22 min

Tabulka 2: Tabulka porovnání vypočtených hodnot se serverem Mapy.cz -

4.3 Minimální kostra



Obrázek 9: Minimální kostra dle Kruskalova algoritmu



Obrázek 10: Minimální kostra dle Primova algoritmu

5 Závěr

- V rámci úlohy byly implementovány a aplikovány ztěžejnější grafové algoritmy na reálnou silniční síť (okres Mladá Boleslav).
- Silniční síť byla převedena z formátu GeoPackage do grafové reprezentace a následně byly prováděny grafové algoritmy.
- Pro nalezení nejkratší cesty mezi uživatelem zvolenými uzly byl implementován Dijkstrův algoritmus, který pracuje s grafy s kladným ohodnocením hran. Pro grafy se záporným ohodnocením hran byl použit Bellman-Fordův algoritmus.
- Dále byly nalezeny nejkratší cesty mezi všemi dvojicemi uzelů v grafu pomocí Floyd-Warshallova algoritmu.

- Pro výpočet minimální kostry grafu byly implementovány algoritmy Primův a Kruskalův.
- Také byly implementovány heuristiky Weighted Union a Path Compression, které optimalizují některé výpočty.
- Dosažené výsledky jsou srovnatelné s výsledky serveru Mapy.cz. Jak je patrné z tabulky 1, na trase Čachovice–Bělá pod Bezdězem se délka cesty liší o 0.7 km a čas o 7 minut. Na trase Bezno–Kněžmost se pak délka trasy liší o 1.5 km a čas o 4 minuty, jak lze pozorovat v tabulce 2.
- Vypočtená minimální kostra má délku 454 km.
- Návrhy na vylepšení: Vytvoření GUI, sofistikovanější způsob ohodnocení hran, lepší způsob vykreslení grafu.

6 Přílohy

Příloha 1 - pseudokódy

Příloha 2 - python skripty

V Praze dne 30.11.2024

Matěj Klimeš, Tomáš Zbíral

Algorithm 1 Pseudokód pro skript geopackage_to_graph.py

- 1: **Načtení dat:**
 - 2: Načíst soubor silnic silnice_gp.gpkg pomocí geopandas
 - 3: Načíst soubor sídel sidla_gp.gpkg pomocí geopandas
 - 4: Vypsat sloupce v souborech pro ověření struktury dat
 - 5: **Rychlostní mapa:**
 - 6: Definovat rychlostní mapu pro různé třídy silnic
 - 7: rychlostni_mapa = {1: 130, 2: 110, 3: 90, 4: 70, 5: 50, 6: 30}
 - 8: **Funkce pro nalezení nejbližšího bodu na silnici:**
 - 9: Funkce find_nearest_point_on_line(point, lines):
 - 10: Pro každý segment silnice v lines:
 - 11: Vypočítat vzdálenost mezi bodem sídla a bodem na silnici
 - 12: Vybrat nejbližší bod
 - 13: **Funkce pro přiřazení vrcholu k sídlu nebo rozdelení silnice:**
 - 14: Funkce split_line_or_assign_vertex(line, point, vertices, settlements_info):
 - 15: Pokud je silnice MultiLineString:
 - 16: Vybrat nejbližší podlinii k bodu sídla
 - 17: Pokud je silnice LineString:
 - 18: Rozdělit silnici na dvě části podle nejbližšího bodu
 - 19: Pokud je bod sídla u konce silnice, přiřadit ho k existujícímu vrcholu
 - 20: **Funkce pro výpočet křivosti silnice:**
 - 21: Funkce calculate_crookedness(line):
 - 22: Vypočítat délku silnice a eukleidovskou vzdálenost mezi počátečním a koncovým bodem
 - 23: Vypočítat křivost jako poměr délky a eukleidovské vzdálenosti
 - 24: **Zpracování grafu:**
 - 25: Vytvořit prázdný graf graph = nx.Graph()
 - 26: Inicializovat mapy point_to_vertex a vertex_to_name
 - 27: **Přidání sídel do grafu:**
 - 28: Pro každý bod sídla:
 - 29: Najít nejbližší bod na silnici pomocí find_nearest_point_on_line
 - 30: Pokud byl bod nalezen, přidat ho do grafu a přiřadit jméno
 - 31: **Přidání silnic do grafu:**
 - 32: Pro každou silnici:
 - 33: Získat počáteční a koncový bod silnice
 - 34: Vypočítat délku, křivost a náklady silnice
 - 35: Přidat hranu do grafu s odpovídajícími atributy
 - 36: **Uložení grafu:**
 - 37: Uložit graf do souboru graph.pkl pomocí pickle
-

Algorithm 2 Algoritmus Dijkstra

```
0: funkce dijkstra(graph, start, end)
    start ← uzel počátečního bodu
    end ← uzel cílového bodu
0: start ← resolve_node(start)
0: end ← resolve_node(end)
0: dist ← slovník, kde každý uzel má hodnotu  $\infty$  (kromě start, který má hodnotu 0)
0: dist[start] ← 0
0: prev ← slovník, kde každý uzel má hodnotu None
0: pq ← prioritní fronta (fronta s prioritou podle vzdálenosti) ← {(0, start)}
0: while pq není prázdná do
0:     (current_dist, current_node) ← heapq.heappop(pq) {odeber první prvek z fronty}
0:     if current_node == end then
0:         break {Pokud jsme dosáhli cílového uzlu, ukončujeme}
0:     end if
0:     if current_dist > dist[current_node] then
0:         continue {Pokud je vzdálenost větší, než je uložená, přeskočíme tento uzel}
0:     end if
0:     for každý soused (neighbor, weight) v sousedních uzlech current_node do
0:         distance ← current_dist + weight
0:         if distance < dist[neighbor] then
0:             dist[neighbor] ← distance
0:             prev[neighbor] ← current_node
0:             heapq.heappush(pq, (distance, neighbor))
0:         end if
0:     end for
0: end while
0: path ← prázdný seznam
0: current ← end
0: while current is not None do
0:     path.insert(0, current)
0:     current ← prev[current]
0: end while
0: return path, dist[end] =0
```

Algorithm 3 Kruskalův algoritmus

```
0: funkce kruskal(graph)
    graph je graf, který obsahuje uzly a hrany s váhami
0: edges ← seznam prázdných hran
0: for každá hrana (u, v, data) v grafu do {Pro každou hranu v grafu}
0:   weight ← data.get('weight', 1) {Získání váhy hrany, nebo 1 pokud není definována}
0:   edges.append((weight, u, v)) {Přidání hrany do seznamu}
0: end for
0: edges.sort() {Seřazení hran podle váhy}
0: uf ← UnionFind(graph.nodes) {Vytvoření objektu pro disjunktní množiny}
0: mst ← prázdný seznam {Seznam pro ukládání hran minimální kostry}
0: for každá hrana (weight, u, v) v edges do {Pro každou hranu v seřazeném seznamu}
0:   if uf.find(u) ≠ uf.find(v) then {Pokud uzly nejsou ve stejně množině}
0:     uf.union(u, v) {Spoj uzly u a v do stejně množiny}
0:     mst.append((u, v, weight)) {Přidej tuto hranu do minimální kostry}
0:   end if
0: end for
0: return mst {Vrátí minimální kostru jako seznam hran} =0
```

Algorithm 4 Primův algoritmus

```
0: funkce prim(graph)
    start_node ← první uzel v grafu
    visited ← prázdná množina
    mst ← prázdný seznam
    pq ← seznam obsahující (0, start_node, None)
0: while pq není prázdný do
0:   Odebíráme prvek (weight, current, previous) z pq
0:   if current je v visited then
0:     pokračuj
0:   end if
0:   Přidáme current do visited
0:   if previous není None then
0:     Přidej (previous, current, weight) do mst
0:   end if
0:   for každý neighbor uzel current do
0:     edge_weight ← váha hrany mezi current a neighbor (výchozí váha 1)
0:     if neighbor není v visited then
0:       Přidej (edge_weight, neighbor, current) do pq
0:     end if
0:   end for
0: end while
0: return mst =0
```

Algorithm 5 Algoritmus Floyd-Warshall

```
0: funkce floyd_warshall(graph)
   nodes ← seznam všech uzlů v grafu
   dist ← slovník, kde pro každý uzel  $u$  vytvoříme slovník  $v$  s počátečními hodnotami  $\infty$ 
0: for každý uzel  $u$  v nodes do
0:   dist[u][u] ← 0
0: end for
0: for každou hranu  $(u, v)$  do v grafu
0:   weight ← váha hrany (pokud není, nastavíme na 1)
0:   dist[u][v] ← weight
0: end for
0: for každý uzel  $k$  v nodes do
0:   for každý uzel  $i$  v nodes do
0:     for každý uzel  $j$  v nodes do
0:       dist[i][j] ← min(dist[i][j], dist[i][k] + dist[k][j])
0:     end for
0:   end for
0: end for
0: return dist = 0
```

Algorithm 6 Algoritmus Bellman-Ford

```
0: funkce bellman_ford(graph, start, end)
   start ← uzel počátečního bodu
   end ← uzel cílového bodu
0: start ← resolve_node(start)
0: end ← resolve_node(end)
0: dist ← slovník, kde každý uzel má hodnotu  $\infty$  (kromě start, který má hodnotu 0)
0: dist[start] ← 0
0: prev ← slovník, kde každý uzel má hodnotu None
0: for každě iterace od 1 do (počet uzlů - 1) do
0:   for každou hranu  $(u, v)$  v grafu do
0:     weight ← váha hrany
0:     if dist[u] + weight < dist[v] then
0:       dist[v] ← dist[u] + weight
0:       prev[v] ← u
0:     end if
0:   end for
0: end for
0: for každou hranu  $(u, v)$  v grafu do
0:   weight ← váha hrany
0:   if dist[u] + weight < dist[v] then
0:     raise ValueError("Graph contains a negative weight cycle")
0:   end if
0: end for
0: path ← prázdný seznam
0: current ← end
0: while current is not None do
0:   path.insert(0, current)
0:   current ← prev[current]
0: end while
0: return path, dist[end] = 0
```

Algorithm 7 Třída UnionFind

```
0: class UnionFind
    Tato třída implementuje disjunktní množiny s path compression a union by rank.
0: funkce constructor(nodes)
    nodes je seznam uzlů grafu.
0: parent  $\leftarrow$  slovník, kde každý uzel ukazuje sám na sebe (parent[node] = node)
0: rank  $\leftarrow$  slovník, kde každý uzel má počáteční rank 0 (rank[node] = 0)
0: funkce find(node)
0: if parent[node]  $\neq$  node then {Pokud uzel není kořen}
0:     parent[node]  $\leftarrow$  find(parent[node]) {Proveď path compression, rekurzivně najdi kořen}
0: end if return parent[node] {Vrať kořen uzlu}
0: funkce union(u, v)
0: root_u  $\leftarrow$  find(u) {Najdi kořen uzlu u}
0: root_v  $\leftarrow$  find(v) {Najdi kořen uzlu v}
0: if root_u  $\neq$  root_v then {Pokud kořeny nejsou stejné}
0:     if rank[root_u] > rank[root_v] then {Pokud rank kořenu u je větší}
0:         parent[root_v]  $\leftarrow$  root_u {Připoj v pod u}
0:     else if rank[root_u] < rank[root_v] then {Pokud rank kořenu v je větší}
0:         parent[root_u]  $\leftarrow$  root_v {Připoj u pod v}
0:     else {Pokud mají stejné ranky}
0:         parent[root_v]  $\leftarrow$  root_u {Připoj v pod u}
0:         rank[root_u]  $\leftarrow$  rank[root_u] + 1 {Zvyš rank kořenu u}
0:     end if
0: end if
```

Algorithm 8 Převod grafu na seznam sousednosti

```
0: funkce graph_to_adj_list(graph, use_length_as_weight)
    graph  $\leftarrow$  graf, který má hrany s váhami (nebo délkami)
    use_length_as_weight  $\leftarrow$  boolean, který určuje, zda použít "length" jako váhu hrany (True/False)
0: adj_list  $\leftarrow$  prázdný slovník (seznam sousednosti)
0: for každá hrana (u, v, data) v grafu do
0:     if use_length_as_weight then
0:         weight  $\leftarrow$  data['length']
0:     else
0:         weight  $\leftarrow$  data.get('weight', 1) {Pokud není váha definována, použijeme 1 jako default}
0:     end if
0:     if u není v adj_list then
0:         adj_list[u]  $\leftarrow$  prázdný seznam
0:     end if
0:     if v není v adj_list then
0:         adj_list[v]  $\leftarrow$  prázdný seznam
0:     end if
0:     adj_list[u].append((v, weight))
0:     adj_list[v].append((u, weight)) {Protože graf je neorientovaný}
0: end for
0: return adj_list = 0
```

Algorithm 9 Řešení uzlu na index vrcholu

```
0: funkce resolve_node(node)
    node je buď jméno uzlu (string) nebo index vrcholu (integer)
0: if node je typu str then {Pokud je uzel jméno sídliště}
    name_to_vertex ← slovník vytvořený převrácením vertex_to_name
    (mapování z jména na vertex)
0:   if node je klíč v name_to_vertex then {Pokud existuje odpovídající index vrcholu}
0:     return name_to_vertex[node] {Vrať index vrcholu}
0:   else
0:     raise ValueError("Settlement 'node' not found in the graph.")
0:   end if
0: else{Pokud je uzel již integer (index vrcholu)}
0:   return node {Vrať přímo index}
0: end if=0
```

Algorithm 10 Vizualizace grafu

```
0: procedure VISUALIZEGRAPH(graph, highlight, path, mst, floyd_paths)
0:   pos ← Pozice uzlů z atributů grafu nebo generování rozložení
0:   labels ← Labely uzlů z atributů grafu
0:   Nakreslit graf s pos a základním stylováním
0:   if path není prázdný then
0:     Zvýraznit hrany cesty červeně
0:   end if
0:   if mst není prázdný then
0:     Zvýraznit hrany MST zeleně
0:   end if
0:   if floyd_paths není prázdný then
0:     Zvýraznit hrany z Floyd-Warshall algoritmu fialově
0:   end if
0:   Přidat popisky a titulky
0:   Zobrazit vizualizaci grafu
0: end procedure=0
```
