EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 36146475399069956

VYUŽITIE PROGRAMOVÉHO BALÍKA PYTHON V EKONOMICKÝCH OPTIMALIZAČNÝCH ÚLOHÁCH

DIPLOMOVÁ PRÁCA

2025 **Dmytro Musevych**

EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

VYUŽITIE PROGRAMOVÉHO BALÍKA PYTHON V EKONOMICKÝCH OPTIMALIZAČNÝCH ÚLOHÁCH

DIPLOMOVÁ PRÁCA

Študijný program: Informačný manažment

Študijný odbor: Ekonómia a manažment

Školiace pracovisko: Katedra operačného výskumu

Vedúci záverečnej práce: doc. Ing. Karol Szomolányi, PhD

Bratislava 2025 **Dmytro Musevych**

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Dmytro Musevych
Študijný program: informačný manažment (Jednoodborové štúdium, inžiniersky II. st., denná forma)
Študijný odbor: ekonómia a manažment
Typ záverečnej práce: Inžinierska záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Využitie programového balíka Python v ekonomických optimalizačných úlohách

Anotácia: Práca prezentuje využitie programového balíka Python v ekonomických optimalizačných úlohách.

Vedúci: doc. Ing. Karol Szomolányi, PhD.
Katedra: KOVE FHI - Katedra operačného výskumu a ekonometrie

Dátum zadania: 13.02.2024

Dátum schválenia: 11.03.2024

prof. Mgr. Juraj Pekár, PhD.
vedúci katedry

ABSTRAKT

MUSEVYCH, Dmytro: *Využitie programového balíka Python v ekonomických optimalizačných úlohách*. – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra operačného výskumu a ekonometrie. – Vedúci záverečnej práce: doc. Ing. Karol SZOMOLÁNYI, PhD. – Bratislava: FHI EU, 2025, s. 91.

Cieľom záverečnej práce je preskúmať možnosti riešenia optimalizačných úloh v oblasti logistiky pomocou balíkov programovacieho jazyka Python, analyzovať najnovšie vedecké články a riešiť praktickú úlohu. Práca je rozdelená do troch kapitol, obsahuje 4 tabuľky, 26 obrázkov a 13 príloh. Prvá kapitola je rozdelená na viacero častí. Prvá časť je venovaná rozboru základných prípadov použitia jazyka Python v optimalizačných dopravných úlohách. Druhá časť prvej kapitoly je zameraná na analýzu vedeckých článkov, ktoré sa venujú výskumu, vývoju a implementácii jedinečných softvérových riešení integrovaných s jazykom Python na riešenie optimalizačných úloh. Druhá kapitola je taktiež rozdelená na viacero častí a je venovaná riešeniu praktickej dopravnej úlohy na základe všeobecne známych údajov o meste Bratislava a z otvorených zdrojov. V prvých troch častiach tejto kapitoly sa uskutočňuje formulácia úlohy, zber a popis údajov, ako aj metodológia ich zberu. Štvrtá časť druhej kapitoly sa zameriava na programovú implementáciu riešenia úlohy. Táto časť je rozdelená na ďalšie dve podčasti, v ktorých sa úloha skúma za dvoch rôznych podmienok. Posledná, piata časť druhej kapitoly, je venovaná analýze výsledkov oboch metód. Tretia, záverečná kapitola je venovaná sumarizácii výsledkov práce a definovaniu potenciálnych smerov ďalšieho rozvoja.

Kľúčové slová: Python, Route Optimization, Costs Optimization, NetworkX, TSP

ABSTRACT

MUSEVYCH, Dmytro: *The Use of the Python Programming Package in Economic Optimization Problems.* – University of Economics in Bratislava. Faculty of Economic Informatics; Department of Operations Research and Econometrics. – Supervisor: doc. Ing. Karol SZOMOLÁNYI, PhD. – Bratislava: FHI EU, 2025, pp. 91.

The aim of this thesis is to explore the possibilities of solving optimization problems in the field of logistics using Python programming packages, to analyze recent scientific articles, and to solve a practical task. The thesis is divided into three chapters and contains 4 tables, 26 figures, and 13 appendices. The first chapter is divided into several sections. The first section focuses on the analysis of basic use cases of the Python language in transportation optimization problems. The second section is dedicated to the analysis of scientific articles focused on research, development, and implementation of unique software solutions integrated with Python for solving optimization tasks. The second chapter is also divided into several sections and focuses on solving a practical transportation problem based on publicly available data about the city of Bratislava and open data sources. The first three sections of this chapter include the formulation of the task, data collection and description, and the methodology of data gathering. The fourth section of the second chapter focuses on the programmatic implementation of the solution. This section is further divided into two parts, where the task is examined under two different sets of conditions. The final, fifth section of the second chapter is dedicated to the analysis of the results of both methods. The third and final chapter summarizes the results of the thesis and outlines potential directions for further development.

Key words: Python, Route Optimization, Costs Optimization, NetworkX, TSP

Obsah

Úvod

Téma optimalizačných úloh bude vždy aktuálna v podmienkach modernej ekonomiky, kde rýchlosť a efektívnosť rozhodovania priamo ovplyvňujú konkurencieschopnosť podnikov. Oblasti logistiky a dopravy patria medzi kľúčové zložky každej ekonomiky, a preto neustály technologický pokrok tlačí na podniky, ktoré sú nútené hľadať nové optimalizačné nástroje, pričom im zároveň ponúka nové spôsoby riešenia problémov. Napriek existencii výkonných komerčných systémov na vykonávanie optimalizačných výpočtov však mnohé malé a stredné podniky nemajú k takýmto riešeniam prístup pre ich vysokú cenu alebo zložitosť integrácie.

Práve preto ma zaujala možnosť vytvorenia dostupného a adaptívneho modelu na plánovanie rozvozu, ktorý by zohľadňoval špecifiká reálnych podmienok, ako sú časové obmedzenia či priorita niektorých adries, pričom hlavným faktorom by bola aj zmena dopravnej situácie na cestách. Programovací jazyk Python vďaka svojej otvorenosti, flexibilite a širokej škále špecializovaných knižníc ponúka príležitosti na realizáciu takýchto modelov aj v nekomerčnom prostredí.

Výber tejto témy je zároveň podmienený mojím osobným záujmom o prepojenie informatiky, modelovania a analýzy softvérových riešení. Rozhodol som sa zvoliť prípad, ktorý simuluje reálne podmienky a využíva reálne údaje a fakty o doprave v Bratislave. Tento výskum je aktuálny, pretože model dopravnej úlohy v rámci tohto mesta nielenže odhalí problémy a špecifiká doručovania v reálnom živote, ale môže byť okamžite aplikovaný v praxi na prijímanie manažérskych rozhodnutí.

Cieľom tejto práce je taktiež preskúmať najnovšiu akademickú literatúru týkajúcu sa využitia jazyka Python v rôznych oblastiach ekonomických optimalizačných úloh, najmä pri riešení dopravných problémov, hľadaní najkratšej cesty, lineárnom programovaní a iných úlohách. Pri analýze vedeckých prác sa chcem zamerať na určenie úlohy Pythonu v komplexných systémoch, jeho integráciu a interakciu s inými softvérovými komponentmi a systémami, jeho miesto v architektúre, výhody a obmedzenia, ako aj na praktické príklady jeho použitia.

1. Teoretická časť

1.1. Praktické využitie jazyka Python v dopravnej oblasti

Balík Python sa v dopravnom sektore využíva rôznymi spôsobmi. Jedným z najčastejších použití je analýza údajov, ako aj modelovanie dopravných tokov a optimalizácia dopravných systémov. Python sa tiež používa na vývoj softvéru pre plánovanie trás, optimalizáciu logistiky a sledovanie dopravných prostriedkov (Tyagi, 2024). Práve v tomto bode vzniká prienik medzi potrebou optimalizácie a využitím Pythonu, čo vytvára široký priestor pre výskum v súvislosti s témou tejto vedeckej práce. Ďalej sú podrobnejšie rozpracované štyri oblasti v logistike s rôznymi prístupmi k využitiu jazyka Python a praktickými príkladmi: analýza sietí, optimalizácia trasy, rozvrhu a nákladov.

1.1.1. Analýza sietí

Analýza sietí je proces modelovania a analýzy dopravných sietí, ako sú cestné komunikácie, železnice alebo letecké trasy. Môže pomôcť porozumieť štruktúre, výkonnosti a dynamike siete, ako aj identifikovať potenciálne úzke miesta, preťaženie alebo neefektívnosť (Ren, 2022).

V jazyku Python existuje niekoľko knižníc, ktoré môžu pomôcť vykonávať sieťovú analýzu, napríklad NetworkX, PyGraphviz alebo igraph ((Harberg et al., 2008); (Jovanovic et al., 2024)). Tieto knižnice umožňujú vytvárať, upravovať a vizualizovať sieťové grafy, ako aj aplikovať rôzne algoritmy, ako je najkratšia cesta, sieťový tok alebo merania centrality.

Pozrime si jednoduchý praktický príklad riešenia úlohy z oblasti analýzy sietí s cieľom identifikovať preťažený uzol. Na zostavenie siete použijeme reálne vzdialenosti medzi mestami Slovenskej republiky.

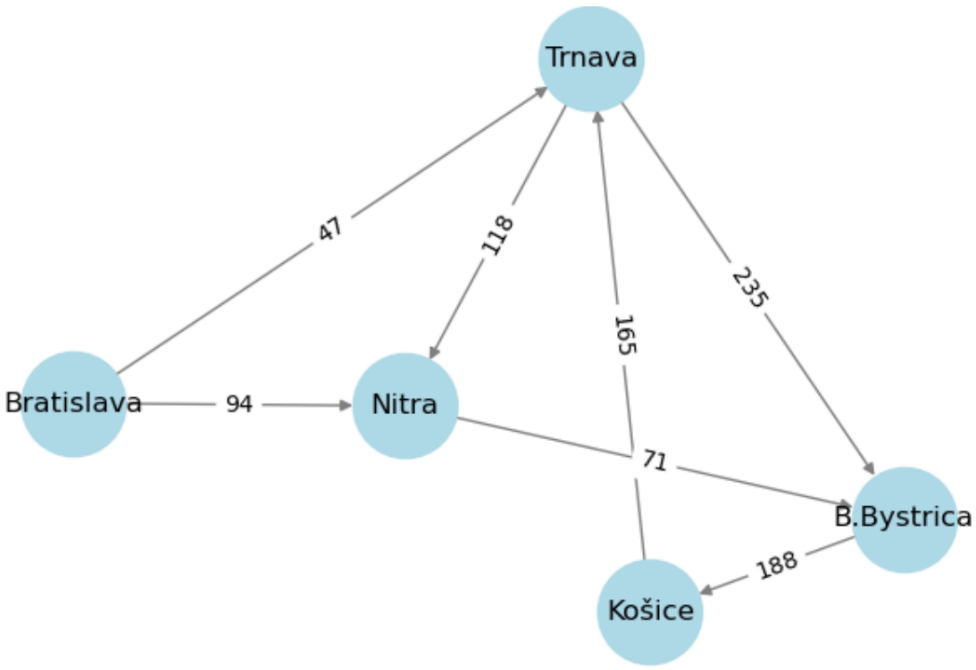
Rebro	Vzdialenosť
Bratislava – Trnava	47 km
Bratislava – Nitra	94 km
Trnava – Nitra	118 km
Trnava – Banská Bystrica	235 km
Nitra – Banská Bystrica	71 km

Banská-Bystrica – Košice	188 km
Košice – Trnava	165 km

Tabuľka 1. Vzdialenosti medzi mestami SR

Najviac zaťažený uzol v tomto prípade bude ten, ku ktorému vedie najväčší počet najkratších ciest, pretože práve tam sa budú dopravné prostriedky najrýchlejšie zhromažďovať. Tým pádom bude zaťaženie operácií spracovania nákladu v ňom väčšie ako v iných uzloch. V závislosti od stupňa zaťaženia sa prijímajú rozhodnutia o rozšírení kapacít uzla alebo o pridaniu ďalšieho uzla na odľahčenie existujúceho.

V kóde Prílohy č.1 pre riešenie úlohy používajú sa knižnice *networkx* (NetworkX, 2025) na analýzu siete a *matplotlib.pyplot* (Matplotlib.org., 2022) na vizualizáciu grafu. Najskôr sa vytvorí orientovaný graf G pomocou *nx.DiGraph(),* potom sa pridajú hrany medzi uzlami spolu s váhami, ktoré predstavujú vzdialenosť. Následne sa vypočíta centralita uzlov pomocou metódy *nx.betweenness_centrality*, čo umožňuje určiť, cez ktoré uzly prechádza najviac najkratších ciest. Uzol s najvyššou hodnotou *centrality* sa považuje za najviac zaťažený ((Nair et al., 2024); (Feng et al., 2022)). Výsledky sa vypíšu do konzoly, potom sa vykoná vizualizácia grafu. Rozloženie uzlov sa určuje pomocou funkcie *nx.spring_layout(),* samotný graf sa vykreslí pomocou *nx.draw()*, pričom popisy váh hrán sa pridajú cez *nx.draw_networkx_edge_labels()*. Výsledkom je grafické zobrazenie štruktúry siete so zaznačenými váhami a kľúčovými uzlami, čo umožňuje analyzovať jej efektivitu.



Obr. 1. Vizualizácia výsledkov spustenia kódu z Prílohy č.1.

Okrem toho kód vypíše ďalšie číselné výsledky.

Centralita uzlov (betweenness): {'Bratislava': 0.0,

'Trnava': 0.25,

'Nitra': 0.41666666666666663,

'B.Bystrica': 0.3333333333333333,

'Košice': 0.25}

Najviac zaťažený uzol: Nitra

Tieto čísla ukazujú *betweenness centrality*, teda ako často sa uzol nachádza na najkratších cestách medzi inými uzlami. Čím vyššia hodnota, tým dôležitejší je uzol ako tranzitný bod v sieti. Táto veličina nemá jednotky merania, pretože je normalizovaná a odráža relatívny význam uzla v celkovej štruktúre dopravnej siete. V uvedených výsledkoch Nitra (0.4167) je najdôležitejším uzlom, pretože cez ňu prechádza najviac najkratších trás, čo ju robí potenciálnym „úzkym miestom“ siete. Banská Bystrica (0.3333) zohráva tiež významnú úlohu, ale v menšej miere. Trnava a Košice (0.25) majú strednú úroveň zaťaženia, zatiaľ čo Bratislava (0.0) sa takmer vôbec nepoužíva ako tranzitný uzol. To môže znamenať, že jej poloha alebo spojenia sú menej dôležité pre celkový tok dopravy.

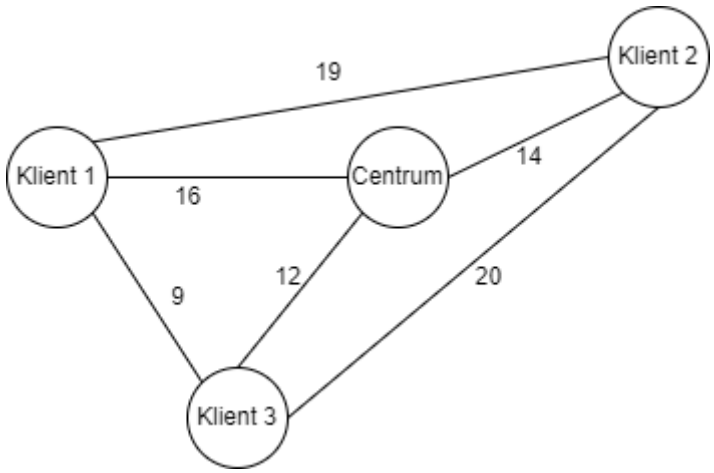
V reálnej praxi môžu vstupné údaje naznačiť potrebu zavedenia nových riešení. V prípade tejto úlohy by bolo potrebné zvážiť optimalizáciu presmerovaním časti trás cez Bratislavu, aby sa vyvážilo zaťaženie. Ďalšou možnosťou je pridanie nového uzla alebo priamej trasy medzi mestami, aby sa znížila závislosť od Nitry a zlepšila priepustnosť siete.

Medzi nevýhody tohto algoritmu patrí zobrazenie uzlov v grafe. Algoritmus ich nedokáže umiestniť tak, aby graf vizuálne zodpovedal skutočnému rozloženiu miest, a čo viac, pomery dĺžok hrán a hodnôt ich zaťaženia sa nezachovávajú. Pri každom opätovnom spustení kódu sa navyše dizajn grafu mení, niekedy na menej prehľadnú verziu. Okrem toho sa vo väčšine prípadov pri vykreslení grafu hrany prekrývajú kvôli nevhodnému umiestneniu uzlov, kde sa jeden nachádza medzi dvoma inými.

1.1.2. Optimalizácia trasy

Optimalizácia trasy je proces hľadania optimálnych alebo takmer optimálnych trás pre dopravné prostriedky s ohľadom na sadu obmedzení, ako sú vzdialenosť, čas, náklady alebo kapacita. Tento proces môže pomôcť skrátiť čas na ceste, znížiť spotrebu paliva či emisie a zároveň zlepšiť spokojnosť zákazníkov alebo kvalitu služieb (Yuan, 2024). V Pythone existuje niekoľko knižníc, ktoré môžu pomôcť riešiť problémy optimalizácie trás, napríklad OR-Tools, PuLP alebo Pyomo (Solver Max Optimal Solutions, 2025). Tieto knižnice umožňujú formulovať a riešiť rôzne typy optimalizačných modelov, ako sú lineárne programovanie, celočíselné programovanie alebo programovanie s obmedzeniami.

Uvažujme riešenie klasického problému obchodného cestujúceho (The Travelling Salesman Problem (TSP)) (Dumka et al., 2025): kuriérska služba má doručiť tovar z distribučného centra trom zákazníkom, ktorí sa nachádzajú na rôznych miestach. Je potrebné prejsť všetky vrcholy grafu najkratšou trasou, aby sa znížili náklady na palivo a čas na ceste a vrátiť sa do východiskového bodu. Pre jednoduchý príklad použijeme nasledujúce údaje: vzdialenosť od centra k prvému zákazníkovi je 16 km, k druhému 14 km a k tretiemu 12 km; vzdialenosť medzi prvým a druhým zákazníkom je 19 km, medzi prvým a tretím 9 km, a medzi druhým a tretím 20 km. Vizualizácia tohto grafu môže vyzerať nasledovne.



Obr. 2. Vizualizácia grafu úlohy typu TSP z príkladu

V Prílohe č.2 je uvedený kód na riešenie danej úlohy. Na začiatku sa importujú potrebné moduly z knižnice PuLP na formuláciu a riešenie úlohy lineárneho programovania (PuLP, 2025). Potom sa vytvorí zoznam *locations*, ktorý obsahuje uzly trasy (Centrum, Klient 1, Klient 2, Klient 3), a slovník *distances*, v ktorom sú definované vzdialenosti medzi uzlami. Následne sa vytvorí objekt *problem* triedy LpProblem, ktorý definuje úlohu

minimalizácie celkovej vzdialenosti. Potom sa určia dve skupiny premenných: *x[i, j]*, ktorá nadobúda hodnotu 1, ak sa medzi uzlami *i* a *j* vyberie trasa, a *u[i]* – pomocná premenná na zabránenie vzniku podcestných cyklov. Cieľová funkcia minimalizuje celkovú vzdialenosť trasy, pričom systém obmedzení zaručuje, že každý uzol bude navštívený práve raz. Aby sa predišlo pod cestným cyklom (prípad, keď program rozdelí riešenie na niekoľko samostatných ciest, z ktorých nemožno vytvoriť jeden optimálny okruh, ktorý sa skončí v počiatočnom bode (Jackovich et al., 2020)), používa sa špeciálne obmedzenie *subtour elimination constraints*, ktoré zabezpečuje vytvorenie jedinej uzavretej trasy. Potom sa zavolá *problem.solve()*, ktoré nájde optimálne riešenie, po čom sa vypíše stav riešenia. Daný kód vygeneruje nasledujúce riešenie úlohy.

Status riešenia: Optimal

Optimálna trasa:

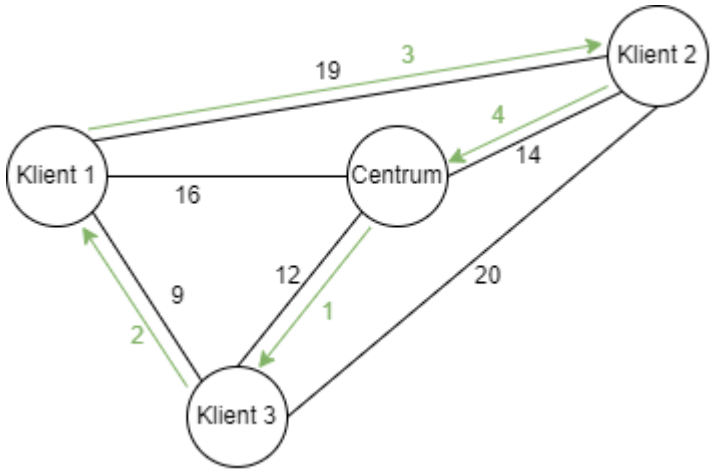
Centrum → Klient 3

Klient 3 → Klient 1

Klient 1 → Klient 2

Klient 2 → Centrum

Na existujúcom grafe môžeme túto cestu znázorniť nasledovne.



Obr. 3. Vizualizácia grafu riešenia úlohy typu TSP z príkladu

Riešenie úlohy nájdenia optimálnej trasy je možné dosiahnuť pomocou najviac praktizovanej metódy lineárneho programovania, ktorá však má svoje výhody aj nevýhody. Jednou z hlavných výhod je, že tento prístup zaručuje nájdenie globálne optimálneho

riešenia, ak má úloha riešenie. Vďaka využitiu dobre preskúmaných optimalizačných metód je riešenie spoľahlivé a model je možné ľahko upraviť pridaním nových obmedzení. Ďalšou výhodou je univerzálnosť metódy, pretože ju možno použiť nielen pre klasický problém obchodného cestujúceho, ale aj pre širšie spektrum dopravných a logistických problémov.

Tento prístup má však aj nevýhody. Pri veľkom počte uzlov sa dimenzia úlohy rýchlo zvyšuje, čo ju robí výpočtovo náročnou, najmä pri rozsiahlych dopravných sieťach. Okrem toho si metóda vyžaduje dodatočné obmedzenia, ako je to v prípade tejto úlohy (spomínané vyššie *subtour elimination constraints*), inak algoritmus môže nájsť riešenie, ktoré obsahuje nesprávne trasy alebo pod cesty. Lineárne obmedzenia používané v tomto modeli môžu tiež byť nedostatočne flexibilné pre reálne podmienky, kde je potrebné brať do úvahy dynamické faktory, ako napríklad zmeny cestnej premávky alebo nepredvídané oneskorenia.

1.1.3. Optimalizácia rozvrhu

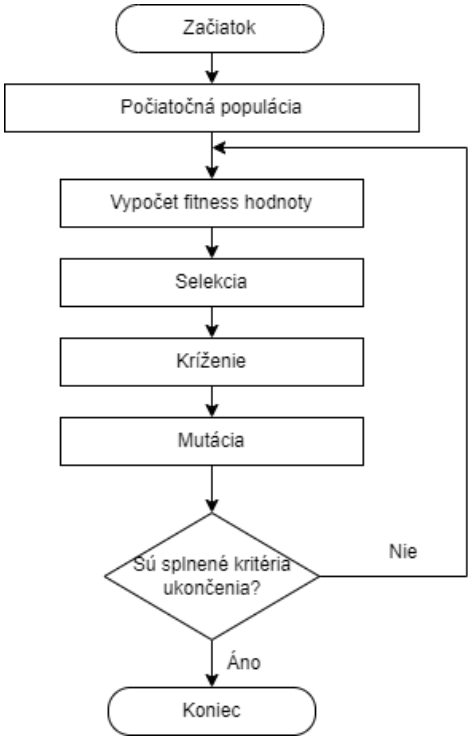
Optimalizácia rozvrhu je proces hľadania optimálneho alebo takmer optimálneho harmonogramu pre dopravné prostriedky alebo cestujúcich, pričom sa zohľadňuje súbor obmedzení, ako sú dostupnosť, dopyt alebo preferencie. Tento proces môže pomôcť vyvážiť ponuku a dopyt, maximalizovať využitie alebo príjmy, prípadne minimalizovať oneskorenia či poruchy (Schwarz et al., 2021).

V jazyku Python existuje niekoľko knižníc, ktoré môžu pomôcť pri riešení problémov s optimalizáciou rozvrhu, napríklad SimPy, DEAP alebo OptaPlanner ((OptaPlanner, 2023)&(SimPy, 2025)&(Advancedor Academy, 2024)). Tieto knižnice umožňujú modelovať a optimalizovať zložité systémy, ako sú rady, skladovanie alebo logistika, a uplatňovať rôzne metódy, ako je diskrétna simulácia udalostí, evolučné algoritmy alebo metaheuristiky.

Rozoberme príklad jednoduchej úlohy optimalizácie rozvrhu. Predpokladajme, že dopravný podnik sa snaží optimálne rozdeliť 10 autobusov medzi tri trasy tak, aby vyvážil zaťaženie a dodržal stanovené obmedzenia. Každá trasa má minimálny a maximálny počet autobusov, ktoré na nej môžu byť pridelené: minimálne požiadavky sú 4, 2 a 2 autobusy, zatiaľ čo maximálne sú 7, 6 a 5. Cieľom optimalizácie je nájsť také rozdelenie autobusov, ktoré minimalizuje maximálne zaťaženie na jednej trase a zároveň zabezpečí efektívne využitie zdrojov. Hlavné obmedzenia spočívajú v tom, že celkový počet autobusov nemôže

presiahnuť 10 a každá trasa musí dostať aspoň minimálny a najviac maximálny počet vozidiel.

Na vyriešenie tejto úlohy je vhodné použiť genetický algoritmus, ktorý pomocou prirodzeného výberu, kríženia a mutácií postupne nájde najvyváženejšie rozdelenie autobusov, spĺňajúce všetky stanovené podmienky (Miao & Zhao, 2024). Nižšie na obrázku je uvedený postup algoritmu.



Obr. 4. Algoritmus riešenia úlohy optimalizácie rozvrhu

V Prílohe č.3 je uvedený kód, ktorý vykonáva danú úlohu. Na vyriešenie bola použitá knižnica DEAP (DEAP, 2024). Najprv sa realizuje hlavná podmienka použitia genetického algoritmu – vytvorenie počiatočnej populácie. Volaním funkcie *toolbox.population()* sa nastaví počet, napríklad 50 jedincov, pričom každý z nich predstavuje zoznam autobusov, ktoré sú náhodne rozdelené medzi trasami. Táto inicializácia sa vykonáva vo funkcii *inicializuj_individual()*, ktorá generuje platné varianty rozdelenia a zaručuje, že každá trasa dostane počet autobusov v prijateľných medziach. Na hodnotenie každého jedinca sa používa funkcia *hodnotenie()*, ktorá spočíta počet autobusov na každej trase a pridáva penalizácie, ak sú porušené minimálne alebo maximálne obmedzenia. Počas behu algoritmu sa používajú hlavné operácie genetického algoritmu (DEAP, 2024): *výber*, *kríženie* a *mutácia*. Výber sa vykonáva cez *toolbox.select = tools.selTournament(tournsize=3)*, kde najlepšie jedince súťažia o postup do ďalšej generácie. Operácia kríženia je implementovaná

cez *toolbox.mate = tools.cxTwoPoint*, ktorá mieša gény rodičovských riešení, čím vytvára nové kombinácie. Mutácia, definovaná v *toolbox.mutate = tools.mutUniformInt(low=0, up=trasy-1, indpb=0.1)*, náhodne mení trasu niektorých autobusov, aby sa zachovala rozmanitosť populácie. Hlavný evolučný cyklus trvá 50 generácií, pričom v každej z nich sa vykonáva funkcia *algorithms.eaSimple()*, ktorá realizuje proces selekcie, kríženia a mutácie, čím aktualizuje populáciu. Po každej generácii sa vypočíta najlepšie nájdené riešenie *tools.selBest(pop, k=1)[0]* a jeho rozdelenie sa uloží. Výstupom algoritmu je optimálne rozdelenie autobusov medzi trasami, ktoré zabezpečuje efektívne využitie zdrojov a dodržanie všetkých obmedzení. Výstup programu:

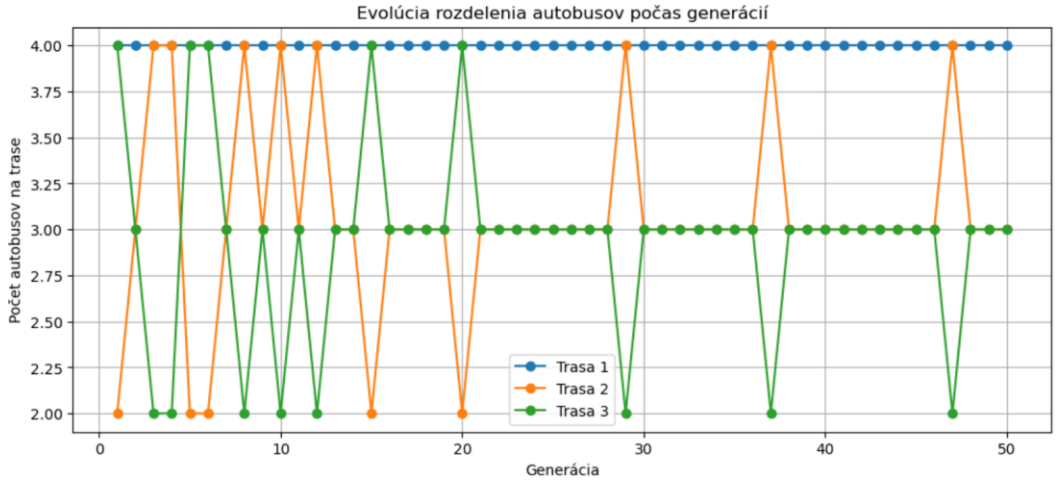
Najlepšie rozdelenie autobusov:

Trasa 1: 4 autobusov

Trasa 2: 3 autobusov

Trasa 3: 3 autobusov

Aby sa mohla sledovať dynamika zmeny riešenia počas generácií, rozšírime kód o vizualizačné nástroje z knižnice *matplotlib.pyplot* (Príloha č.4). Kód sa líši iba tým, že obsahuje zoznam *best_distributions*, ktorý uchováva najlepšie riešenie pre každú generáciu. Po dokončení evolúcie sa vygeneruje graf, ktorý zobrazuje, ako sa menil počet autobusov na jednotlivých trasách od prvej až po poslednú generáciu. Graf je uvedený nižšie.



Obr. 5. Vizualizácia výsledkov evolúcie

Na počiatočných generáciách možno pozorovať vysokú variabilitu rozdelenia, pretože genetický algoritmus náhodne generuje počiatočné riešenia. Avšak v procese evolúcie algoritmus postupne upravuje rozdelenie autobusov, snažiac sa minimalizovať

maximálne zaťaženie na jednotlivých trasách a zároveň splniť všetky stanovené obmedzenia. Práve preto je na grafe viditeľná počiatočná fáza stabilizácie v rozmedzí od 13. do 20. generácie, pričom po 20. generácii riešenie zostáva prakticky nezmenené s výnimkou niekoľkých ojedinelých prípadov. Stabilizácia čiary grafu počas dlhého obdobia generácií znamená, že algoritmus dosiahol optimálne alebo kvázioptimálne riešenie a ďalšie mutácie už neprinášajú významné zlepšenie.

Výhodou genetického algoritmu je že umožňuje nachádzať približne optimálne riešenia aj pri prísnych obmedzeniach. Vďaka použitiu selekcie, kríženia a mutácie dokáže preskúmať široké spektrum možných riešení a vyhnúť sa lokálnym minimám. Má však aj svoje nevýhody. Hlavným problémom je relatívne pomalá konvergencia, keďže na dosiahnutie stabilného riešenia je potrebný veľký počet generácií (Katoch et al., 2021). Okrem toho algoritmus nezaručuje nájdenie globálne optimálneho riešenia, ale iba sa k nemu snaží priblížiť. Ďalším problémom je fluktuácia v neskorších generáciách, keď algoritmus pokračuje v úprave riešení aj po tom, čo sa zdá byť optimálne. Medzi nevýhody patrí aj citlivosť algoritmu na zmeny parametrov, ako je počet generácií, pravdepodobnosť mutácie alebo fitness funkcia. Tieto parametre je potrebné starostlivo nastavovať, pretože aj pri najmenších zmenách môže algoritmus zlyhať a dávať nekonzistentné alebo chaotické výsledky.

1.1.4. Optimalizácia nákladov

Optimalizácia nákladov je proces hľadania optimálnych alebo takmer optimálnych nákladov na dopravnú činnosť so zohľadnením množstva obmedzení, ako sú rozpočet, kvalita alebo riziko. Tento proces môže pomôcť znížiť výdavky, zvýšiť zisk alebo dosiahnuť strategické ciele (Dumka et al., 2025).

V jazyku Python existuje niekoľko knižníc, ktoré môžu pomôcť pri optimalizácii nákladov, napríklad NumPy, SciPy alebo Pandas (PyQueantNews, 2024). Tieto knižnice umožňujú manipulovať a analyzovať údaje, ako sú matice, polia alebo tabuľky, a aplikovať rôzne metódy, ako je regresia, interpolácia alebo optimalizácia.

Pozrime si jednoduchú klasickú úlohu optimalizácie nákladov na distribúciu tovaru zo skladov do obchodov, keď sú k dispozícii informácie o nákladoch na dodanie jednotky

tovaru z každého skladu do každého obchodu, ako aj o ich zásobách a potrebách. Nižšie sú uvedené vstupné údaje pre túto úlohu.

Sklad\Obchod	Nitra	Prešov	Trnava	B. Bystrica	Zásoba
Bratislava	2	3	1	4	100
Košice	3	1	2	3	150
Žilina	4	2	5	1	200
Potreba	80	90	110	170	450

Tabuľka 1. Vstupné údaje príkladu úlohy optimalizácia nákladov

V Prílohe č.5 je uvedený kód, ktorý vykonáva danú úlohu. Algoritmus začína definíciou vstupných údajov, vrátane množstva tovaru v skladoch, požiadaviek obchodov a matice prepravných nákladov. Následne sa tieto údaje pripravia pre lineárne programovanie (spracovanie metódou *flatten()* (NumPy.org, 2024), kde sa nastavujú obmedzenia pre dodávky zo skladov a dopyt obchodov. Kombináciou týchto obmedzení sa vytvára matica obmedzení, ktorá je potrebná pre optimalizáciu. Pomocou metódy lineárneho programovania *linprog()* z knižnice *scipy.optimize* sa vypočíta optimálny plán prepravy (SciPy.org, 2025), ktorý minimalizuje celkové náklady. Ak je riešenie úspešné, výsledky sa zobrazia vo forme prehľadnej tabuľky s množstvami prepraveného tovaru medzi konkrétnymi skladmi a obchodmi. V závere algoritmus zobrazí optimalizovaný prepravný plán a celkové náklady na prepravu. Ak optimálne riešenie neexistuje, zobrazí sa chybové hlásenie. Nižšie sú uvedené výsledky vykonania úlohy týmto kódom.

Prepravný plán:

Obchod Nitra Obchod Prešov Obchod Trnava Obchod Banská Bystrica

Sklad Bratislava 0.0 0.0 100.0 0.0

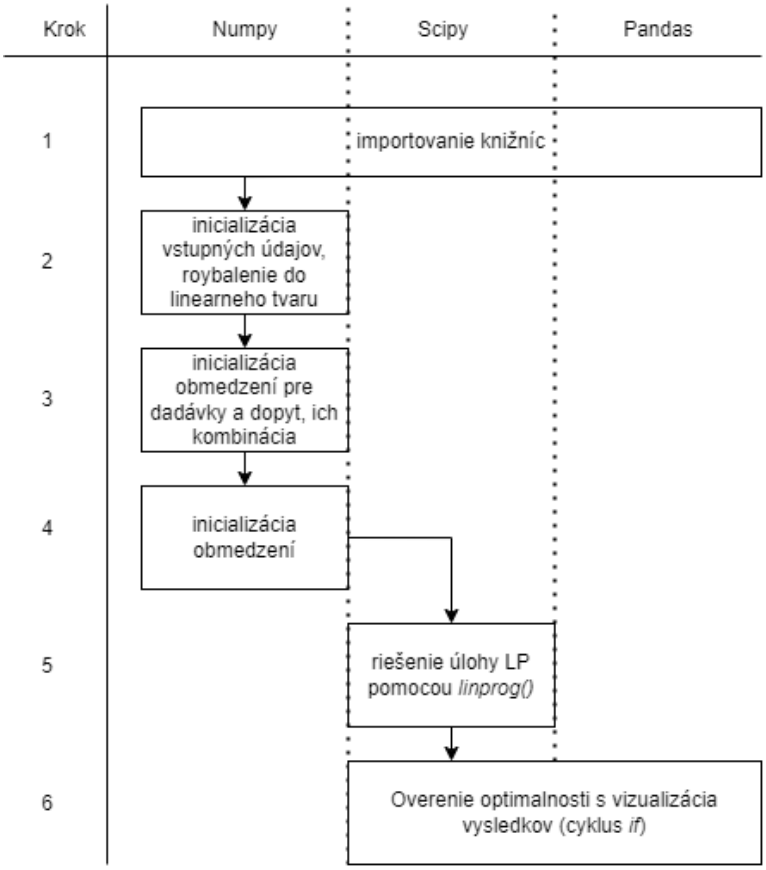
Sklad Košice 50.0 90.0 10.0 0.0

Sklad Žilina 30.0 0.0 0.0 170.0

Celkové náklady na prepravu: 650.00

Aby sa zabezpečilo vykonanie úlohy, boli použité až tri knižnice Python: Numpy, Scipy a Pandas. Každá knižnica plní presne definovanú úlohu, čím zabezpečuje efektivitu a

presnosť optimalizácie nákladov na prepravu. Poradie a účel použitia každej z knižníc počas vykonávania algoritmu sú uvedené na nasledujúcom obrázku.



Obr. 6 Schéma použitia knižníc podľa ich úlohy v algoritme

Použitie knižnice NumPy umožňuje jednoduchú manipuláciu s dátovými poľami, čo je dôležité pre prípravu vstupných údajov a tvorbu obmedzení úlohy. Knižnica SciPy, konkrétne jej modul *optimize.linprog*, implementuje metódy lineárneho programovania, ktoré umožňujú nájsť optimálny plán prepravy minimalizovaním celkových nákladov. Vďaka použitiu metódy *'highs'* algoritmus zabezpečuje vysokú výkonnosť aj pri úlohách s veľkým množstvom obmedzení. Dôležitou súčasťou algoritmu je cyklus *if*, na začiatku ktorého sa nájdené riešenie úlohy overuje podľa kritérií optimálnosti (či riešenie minimalizuje cieľovú funkciu a či sú splnené všetky obmedzenia) pomocou metódy *success* z knižnice SciPy. Ak sú kritériá splnené, cyklus pomocou knižnice Pandas zobrazí výsledky vo formáte tabuľky, čo umožňuje štruktúrované zobrazenie získaných údajov, čím sa zvyšuje prehľadnosť a názornosť výpočtov.

Medzi hlavné výhody tohto algoritmu patrí rýchlosť výpočtov, univerzálnosť a možnosť adaptácie na rôzne typy prepravných úloh. Použitie týchto štandardizovaných

knižníc Pythonu zaručuje stabilitu a spoľahlivosť výsledkov a taktiež uľahčuje integráciu algoritmu do softvérových riešení, ak je to potrebné. Algoritmus však má aj určité nevýhody, medzi ktoré patrí obmedzenie použitia iba na lineárne optimalizačné úlohy, čo môže zúžiť oblasť jeho použitia. Rovnako v prípade, že neexistuje prípustné riešenie, algoritmus nemusí poskytnúť jasné odporúčania na korekciu vstupných údajov alebo alternatívne stratégie riešenia.

1.2. Analýza súčasnej literatúry

1.2.1. Integrácia Pythonu s inými nástrojmi

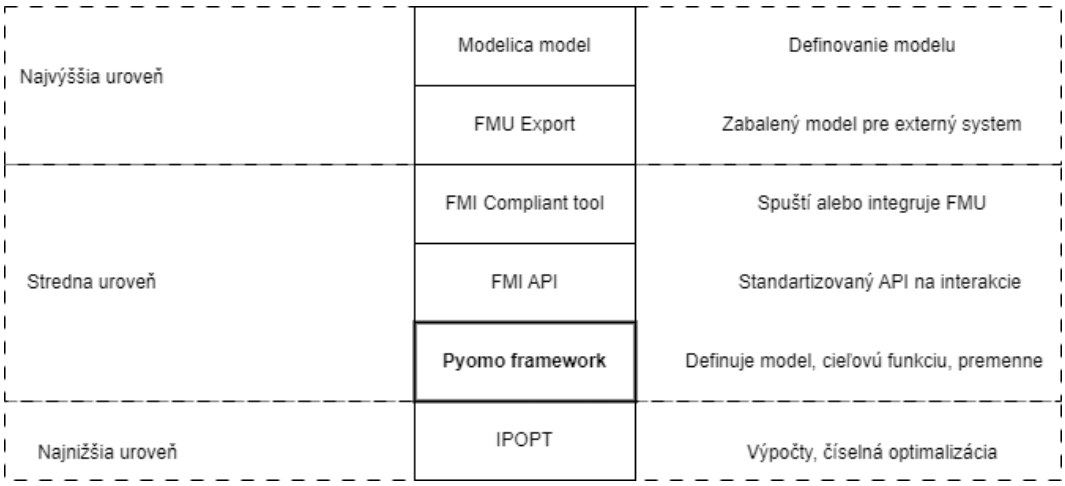
Jedným z príkladov využitia už skôr spomínanej knižnice Python Pyomo je prezentovaný v práci *„Steady-state Optimization of Modelica Models and Functional Mockup Units with Pyomo“* (Gohl et al., 2025). Autori v tejto štúdii porovnávajú dva prístupy k riešeniu optimalizačnej úlohy ako alternatívu k zložitejšiemu softvérovému riešeniu, ktoré si vyžadovalo integráciu väčšieho množstva nástrojov a metód na dosiahnutie požadovaného cieľa.

Jeden z prístupov zahŕňa použitie frameworku Pyomo na spracovanie vstupných údajov, ktoré boli formalizované vo forme exportovaného súboru zo simulovaného modelu vytvoreného v jazyku Modelica. Nástroj kompatibilný so štandardom FMI (Functional Mockup Interface (Modelon, 2025) zabezpečuje načítanie tohto súboru tak, aby bol dostupný pre Pyomo prostredníctvom štandardizovaného rozhrania (FMI ako API).

FMI bol prepojený s nástrojovým ekosystémom knižnice Pyomo pomocou knižnice PyFMI (Pypi.org., 2018), ktorá vytvára prostredie na získanie vstupných údajov vo formáte FMU (MATLAB Help Center, 2024) pre ďalšie spracovanie. Pyomo framework tu funguje ako sprostredkovateľská vrstva, v ktorej sa definuje optimalizačný problém – cieľová funkcia, obmedzenia a rozhodovacie premenné. Používateľ má v tomto prostredí pohodlný prístup k manipulácii s modelom, pričom po úpravách sú údaje odoslané do výpočtového nástroja IPOPT.

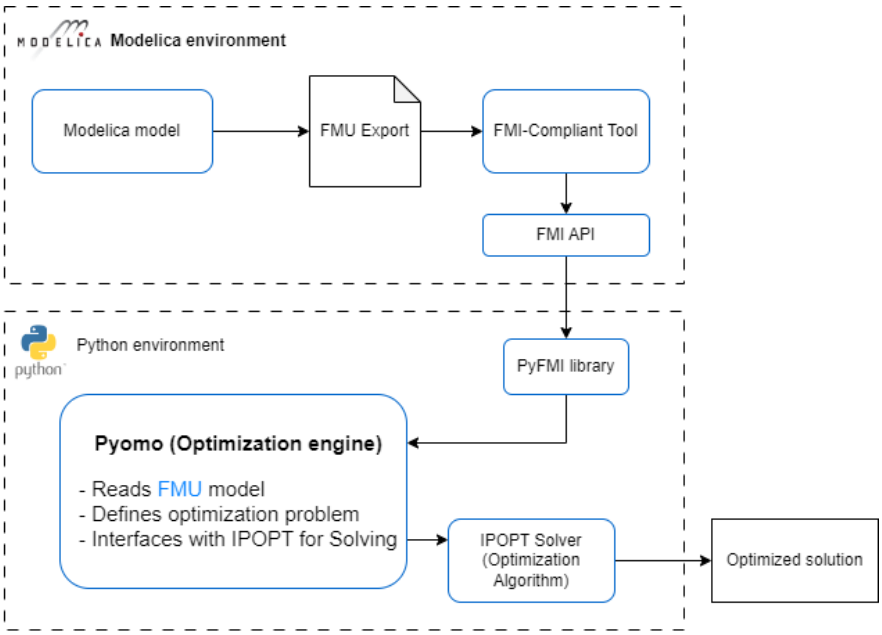
IPOPT (Interior Point OPTimizer) je nízkoúrovňová softvérová knižnica určená na riešenie rozsiahlych nelineárnych optimalizačných problémov (Pypi.org, 2021). V prostredí Python je volaná priamo z frameworku Pyomo (pričom je dostupná aj pre iné programovacie jazyky), a výpočty sa tak vykonávajú na nižšej úrovni softvérovej architektúry.

Nižšie na schéme je znázornená hierarchia softvérových komponentov riešenia vzhľadom na jednotlivé architektonické vrstvy.



Obr. 7. Prehľad komponentov riešenia na architektonických úrovniach

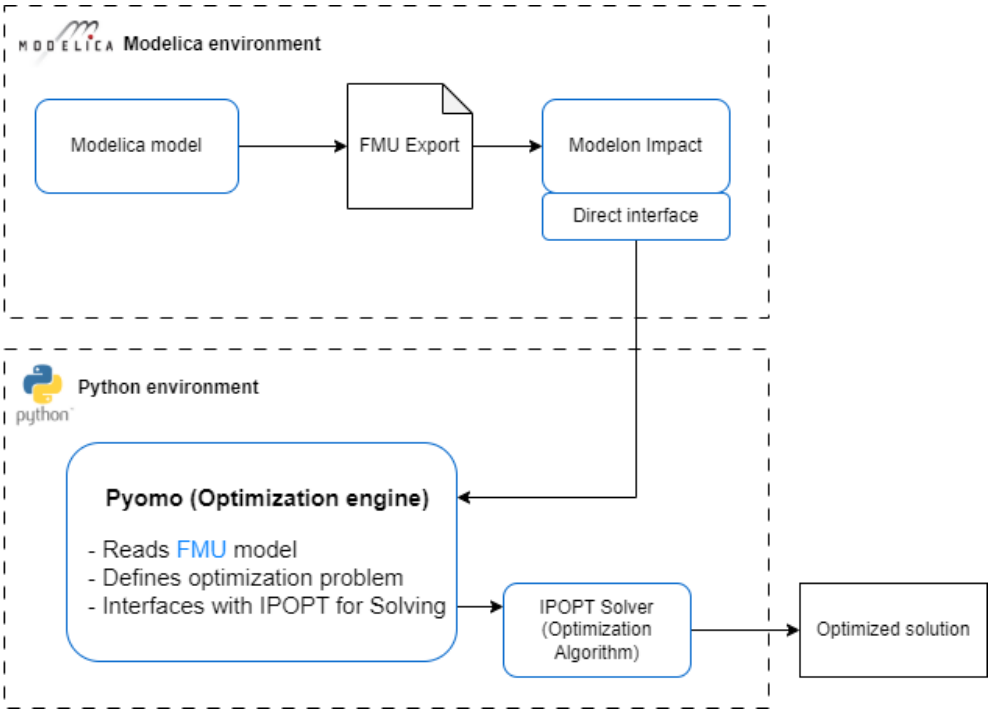
Pyomo framework sa nachádza na strednej architektonickej úrovni a zabezpečuje definovanie kľúčových atribútov modelu pri riešení optimalizačných úloh. Z iného pohľadu môžeme vidieť, že využitie jazyka Python v tomto riešení pokrýva súčasne strednú aj nižšiu architektonickú úroveň: pomocná knižnica PyFMI, samotný rámec Pyomo a výpočtová knižnica IPOPT predstavujú komponenty spoločného softvérového prostredia riadeného práve týmto programovacím jazykom. Tento architektonický pohľad z pohľadu prostredí je uvedený na obrázku nižšie.



Obr. 8. *Prehľad komponentov riešenia s PyFMI*

Vedci porovnávajú tento prístup, ktorý využíva nástroje FMI Compliant Tool a FMI API, s integráciou riešenia Modelon Impact. Tento druhý softvér rovnako umožňuje importovanie a čítanie modelu, no je komplexnejší a využíva vlastné priame rozhranie na interakciu s rámcom Pyomo, čo znamená, že použitie knižnice PyFMI nie je v tomto prípade potrebné.

Nižšie je predstavený architektonický pohľad prispôsobený tomuto alternatívnemu prístupu.



Obr. 9. *Prehľad komponentov riešenia s Modelon*

Pyomo sa teda používa v oboch prípadoch a plní rovnaké funkcie. Tento Python framework je štandardizovaný a flexibilný, čo umožňuje jeho relatívne jednoduchú integráciu s iným softvérom. Autori však dospeli k záveru, že kľúčovú úlohu zohráva práve integrovaný softvér, pretože od neho závisia možnosti a výsledky modelovania.

Prvý prístup je technicky jednoduchší, ľahší a rýchlejší na implementáciu, umožňuje integráciu ďalších riešení na báze programovacieho jazyka Python, je lacnejší, ale má nižšiu presnosť a vyžaduje viac iterácií počas optimalizácie. To robí tento prístup vhodným pre optimalizačné úlohy nižšej až strednej zložitosti v rámci menších projektov.

Druhý prístup umožňuje dosahovanie presnejších a rýchlejších výsledkov a môže sa používať pri riešení komplexných nelineárnych problémov vo veľkých priemyselných

podnikoch, ako je ukázané na príkladoch v štúdii. Jeho implementácia je však náročnejšia a vyžaduje vyššiu technickú kvalifikáciu používateľa pri nastavovaní modelu a celkovo je finančne nákladnejšia.

Táto štúdia je dobrým príkladom využitia štandardizovaných knižníc jazyka Python v kombinácii s individuálne prispôsobeným riešením a integráciou s iným softvérom.

1.2.2. Používanie Pythonu na backende a frontende

Python môže výborne zvládať riešenie optimalizačných úloh prostredníctvom kódovej implementácie algoritmu. Tento programovací jazyk je však známy svojou univerzálnosťou v rôznych oblastiach použitia, vrátane tvorby rozhraní. Odborníci, ktorí používajú Python na riešenie rôznorodých úloh, majú možnosť v prípade potreby vyvíjať aj používateľské rozhrania bez nutnosti rozširovania svojich znalostí o ďalšie programovacie jazyky. Týmto spôsobom môže Python pokryť obe architektonické úrovne softvéru: backend aj frontend.

Dobrým príkladom je článok *«Practice and Research Optimization Environment in Python (PyPROE)»* (Jaus et al., 2025), venovaný vývoju a implementácii optimalizačného prostredia PyPROE, určeného pre vedecký výskum a praktické využitie v oblasti optimalizácie. Hlavným cieľom PyPROE je vytvoriť univerzálnu platformu, ktorá uľahčuje tvorbu matematických modelov a poskytuje možnosť riešenia širokého spektra optimalizačných úloh, vrátane lineárneho, nelineárneho a viackriteriálneho programovania.

V práci sa kladie dôraz na interaktívne možnosti platformy, ktoré umožňujú používateľom operatívne hodnotiť vplyv zmien parametrov úlohy na konečný výsledok. Platforma poskytuje pohodlné rozhranie, ktoré podporuje vizualizáciu a analýzu získaných údajov. To výrazne zvyšuje efektivitu práce používateľov s optimalizačnými modelmi, umožňuje ľahko interpretovať výsledky a prijímať informované manažérske rozhodnutia.

Autori porovnávajú vyvinutú platformu s inými populárnymi softvérmi a priznávajú, že PyPROE zaostáva v oblasti výpočtovej efektívnosti, pretože Python je vysokoúrovňový programovací jazyk a využíva štandardizované knižnice. Avšak pokiaľ ide o návrh optimalizácie zložitých úloh, platforma vyniká vďaka svojej jednoduchosti a intuitívnosti funkcionality, čo urýchľuje prácu a minimalizuje prah zručností potrebných pre používateľa na riešenie úloh.

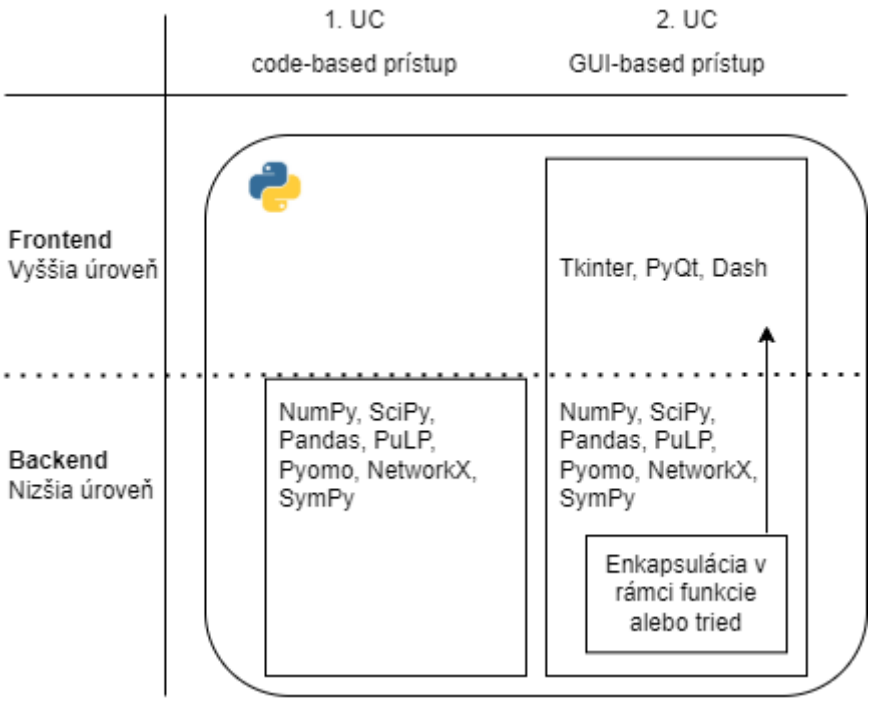
Softvér	Výpočtová účinnosť	Jednoduchosť použitia	Cena
GimOPT	Vysoká	Nizká	Bezplatné
HiPPO	Vysoká	Stredná	Bezplatné
Matlab	Vysoká	Nizká	Stredná
MathCAD	Vysoká	Nizká	Stredná
NEOSIS id8	Stredná	Stredná	Vysoká
PyPROE	Stredná	Vysoká	Bezplatné

Tabuľka 2. Porovnanie výkonu PyPROE z inými nástrojmi

Vedci v článku neuvádzajú informácie o tom, pomocou ktorých knižníc bolo riešenie vytvorené, ale každá z existujúcich knižníc je určená na vykonávanie konkrétnych úloh. Napríklad na tvorbu optimalizačných modelov, manipuláciu s dátami a lineárne programovanie sa používajú knižnice ako NumPy, SciPy, Pandas, PuLP, Pyomo a NetworkX (Umesh, 2023). Na tvorbu grafických používateľských rozhraní sú určené nástroje ako Tkinter, PyQt a Dash ((RealPython, 2025)&(PythonGUIs, 2025)).

Zaujímavé je porovnanie takéhoto prístupu, kde je v prostredí Python implementovaný frontend aj backend, s jednoduchším riešením, kde sa kód píše a spúšťa priamo v konzole, ako v príkladoch uvedených v predchádzajúcich častiach tejto práce. V prípade PyPROE, ktoré predpokladá vytvorenie integrovaného prostredia pre optimalizačné úlohy s využitím Pythonu pre obe architektúrne úrovne aplikácie, prístup umožňuje používateľom nielen vykonávať optimalizačné výpočty, ale aj interagovať s používateľsky prívetivým grafickým rozhraním, prezerať vizualizácie údajov a dynamicky meniť parametre úlohy. Na rozdiel od tohto prístupu, spôsob písania kódu v Pythone a jeho vykonávania priamo v konzole alebo v prostrediach ako Jupyter Notebook poskytuje vývojárovi maximálnu flexibilitu, čo mu umožňuje rýchlo testovať optimalizačné modely a vykonávať zmeny v kóde.

Pozrime sa na prístupy k riešeniu optimalizačných úloh pomocou spúšťania kódu cez konzolu a vývoja plnohodnotnej aplikácie s grafickým používateľským rozhraním z pohľadu architektúry. Nižšie je uvedený prehľad rozdelenia použitia populárnych knižníc v závislosti od architektonickej úrovne pre každý z týchto prípadov použitia.



Obr. 10. Rozdelenie knižníc Python podľa frontendu a backendu

Výhodou prístupu s grafickým používateľským rozhraním (GUI) je interaktivita, unifikované prostredie a jednoduchosť používania aj pre ľudí bez hlbokých znalostí programovania. Zároveň však vývoj rozhrania v Pythone vyžaduje dodatočné zdroje a možnosti dizajnu a prispôsobenia sú obmedzené v porovnaní s modernými webovými frameworkmi, ktoré sú štandardizované a široko používané. Možnosť písania kódu priamo v konzole je vhodná pre profesionálov, ktorí potrebujú prístup k plnej funkcionalite Pythonu a možnosť rýchleho spracovania údajov. Na druhej strane, absencia pohodlného grafického rozhrania môže sťažovať prácu s optimalizačnými modelmi pre neodborníkov a vytvárať problémy v prípade potreby vizualizácie zložitých údajov.

Každý z týchto dvoch prístupov má teda svoje výhody a nevýhody, a preto v praxi neexistuje ideálne riešenie. Výber prístupu závisí od kombinácie mnohých faktorov, cieľov a biznis požiadaviek.

1.2.3. Použitie Python v úlohách dynamickej optimalizácie.

Optimalizačné úlohy niekedy predpokladajú hľadanie riešení v reálnom čase s neustálym monitorovaním zmien parametrov modelu a prispôsobovaním rozhodnutí podľa nich. Subjekt teda má prístup k flexibilnému nástroju, ktorý zohľadňuje rôzne faktory. V článku „*A novel dynamic route optimization method and its implementation using Python to*

optimize ship voyages sailing time based on weather routing techniques“ (Moussa et al., 2024) vedci opisujú vývoj najnovšieho optimalizačného nástroja založeného na dynamických údajoch v reálnom čase. Nástroj sa používa na hľadanie optimálnej trasy lodí so zohľadnením poveternostných podmienok na základe údajov zozbieraných z otvorených zdrojov. Používa sa tu zložitý TBS algoritmus s podporou algoritmov Greedy a Divide and Conquer. Neoddeliteľnou súčasťou riešenia sú aj integrované knižnice Python.

Algoritmus TBS (Time Boundary Semicircles Algorithm) bol vyvinutý vedcami špeciálne na optimalizáciu lodných trás v reálnom čase so zohľadnením meniacich sa poveternostných podmienok. Hlavným cieľom tohto algoritmu je minimalizovať čas plavby výberom trasy, ktorá má najmenší negatívny vplyv počasia na rýchlosť plavidla. Takýto prístup umožňuje skrátiť čas cesty, a tým zvýšiť efektívnosť spotreby paliva a znížiť emisie CO_2 v námornej doprave. Autori neuvádzajú presné detaily konštrukcie algoritmu, avšak je známe, že využíva koncept tzv. „časových polkruhov“ (*Time Boundary Semicircles*), z ktorého pochádza aj názov metódy. Tieto polkruhy reprezentujú maximálnu vzdialenosť, ktorú môže plavidlo prejsť v určitom časovom intervale so zohľadnením reálnych poveternostných podmienok, ako je výška vĺn, smer vetra a ďalšie faktory.

Konštrukcia algoritmu začína zberom meteorologických údajov zo služby Copernicus Marine Environment Monitoring Service (CMEMS), ktorá je verejne dostupná na internete. Údaje sa ukladajú vo formáte NetCDF a spracúvajú sa pomocou knižníc jazyka Python: xarray, Pandas a NumPy. Tieto údaje obsahujú kľúčové ukazovatele námorných podmienok a umožňujú algoritmu zohľadniť meniace sa faktory pri plánovaní trasy. Následne sa vytvárajú tzv. časové polkruhy, ktoré určujú prípustné oblasti pre pohyb plavidla v každom časovom okamihu. Polomer týchto polkruhov zodpovedá maximálnej vzdialenosti, ktorú môže plavidlo prekonať za pevne stanovený čas (napríklad za jednu hodinu) pri optimálnych podmienkach. V rámci každého polkruhu sa identifikujú možné body trasy, pričom každý z týchto bodov sa vyhodnocuje na základe aktuálnych poveternostných podmienok.

Na vyhodnotenie jednotlivých bodov trasy sa používajú tri matematické modely: Bowditch, Artessen a Khokhlov. Tieto špecifické modely, využívané v námornej doprave, vypočítavajú zníženie rýchlosti plavidla v dôsledku vĺn, vetra a ďalších poveternostných faktorov. Následne Greedy algoritmus (Garcia, 2025) vyberá najvhodnejší bod pre pokračovanie trasy na základe minimalizácie strát rýchlosti. V tomto kontexte Greedy algoritmus znamená, že v každom kroku sa vyberá najlepšie lokálne riešenie, bez zohľadnenia globálneho výsledku.

Algoritmus zároveň využíva prístup Divide and Conquer, ktorý umožňuje rozdeliť celkovú trasu na menšie segmenty (Kordic et al., 2024). To zjednodušuje optimalizačný proces, pretože každý segment môže byť spracovaný samostatne s ohľadom na aktuálne poveternostné podmienky. Takýto prístup zaručuje vyššiu flexibilitu a adaptabilitu algoritmu, keďže každá časť trasy môže byť priebežne prispôsobená na základe nových údajov o počasí.

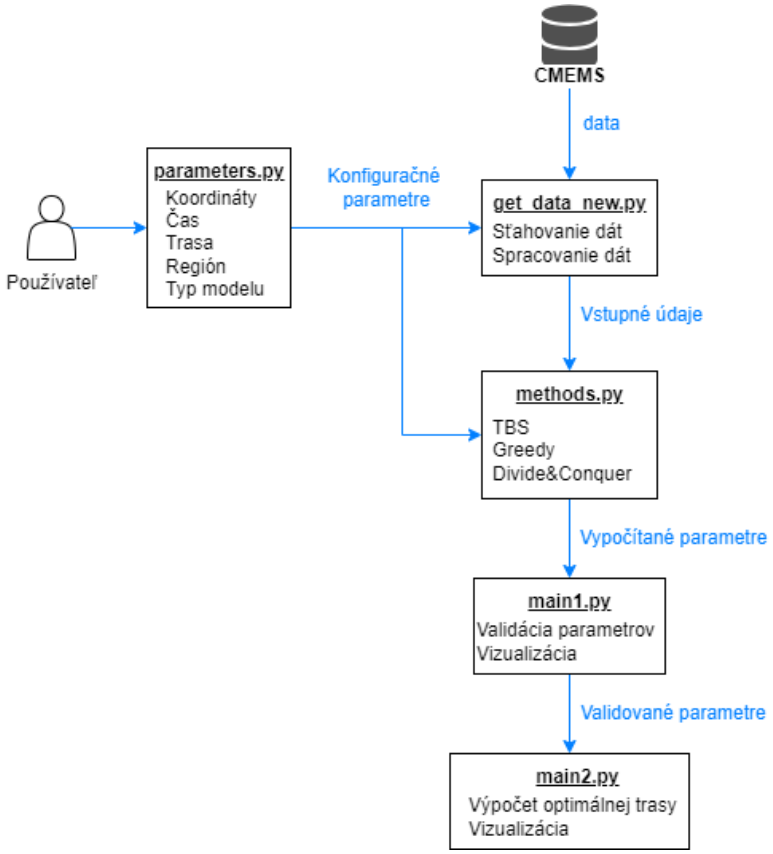
Po vypočítaní optimálnej trasy je jej vizualizácia realizovaná pomocou knižníc Matplotlib a Cartopy. To umožňuje nielen graficky znázorniť optimalizovanú trasu, ale aj vyhodnotiť jej efektivitu v porovnaní s alternatívnymi riešeniami. Nižšie v tabuľke je uvedený všeobecný prehľad použitia jednotlivých knižníc podľa ich funkcie v rámci algoritmov.

Knižnica\Algoritmus	TBS	Greedy	Divide and Conquer
NumPy	Spracovanie údajov o počasí (výška vĺn, smer vetra).	Výpočet najlepšieho bodu	Výpočty s koordinátmi a spracovanie údajových polí
Pandas		Uloženie možností bodov a ich porovnávanie	Štruktúrovanie údajov
Matplotlib	Vizualizáciu trasy na mape	-	Vizualizáciu každého segmentu trasy
Cartopy		-	-
xarray	Práca s meteorologickými údajmi z CMEMS vo formáte NetCDF	Výber optimálnych poveternostných podmienok medzi dostupnými údajmi	-

Tabuľka 3. Účasť knižníc Python v rámci algoritmov

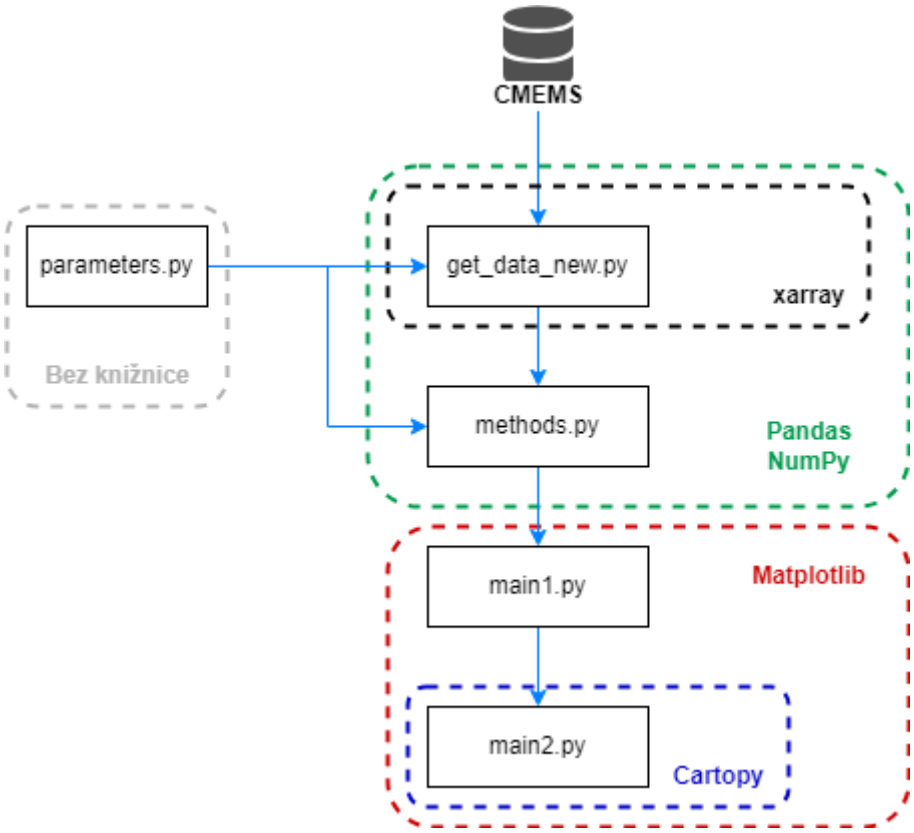
Implementácia softvérového riešenia je zabezpečená pomocou piatich kľúčových skriptov napísaných v jazyku Python, pričom každý z nich zohráva svoju špecifickú úlohu.

Nižšie je uvedený architektonický diagram zobrazujúci postupnosť použitia skriptov, ich poradie a funkcionalitu z pohľadu celkového procesu.



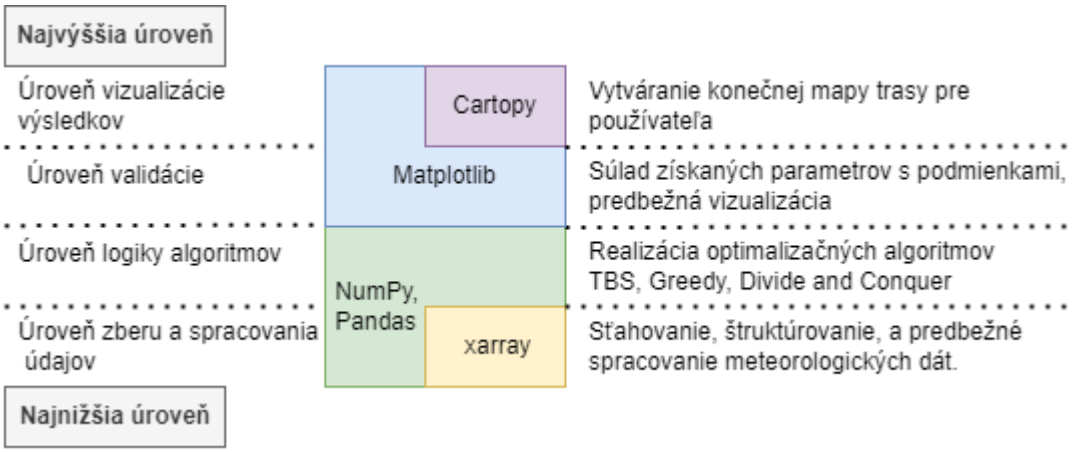
Obr. 11. Postup použitia skriptov Pyhton v TBS algoritme

Každá knižnica Pythonu má svoj špecifický účel, rovnako ako každý skript v rámci daného algoritmu. Nasledujúci architektonický diagram znázorňuje postup použitia skriptov z pohľadu pripojených knižníc. Zobrazená je hierarchia, v ktorej každá knižnica zodpovedá za svoju časť realizácie algoritmu.



Obr. 12. Využitie knižníc v skriptách TBS algoritmu

Zároveň Python pokrýva realizáciu algoritmu na rôznych logických úrovniach. Na každej z týchto úrovní sa využívajú konkrétne knižnice. Napríklad z nižšie uvedeného obrázka môžeme vidieť, že niektoré knižnice, ako napríklad Matplotlib alebo NumPy spolu s Pandas, majú širšiu oblasť použitia a každá z nich pokrýva viacero vrstiev. Naopak, Cartopy a xarray majú užšiu špecializáciu a používajú sa iba na špecifické úlohy v rámci vykonávania algoritmu – ako napríklad vizualizácia výsledkov na geografickej mape alebo spracovanie meteorologických údajov v neštandardnom formáte.



Obr. 13. Využitie knižníc na rôznych logických úrovniach

Výsledky ukázali, že využitie algoritmu TBS umožnilo skrátiť čas plavby o 7 % až 27,25 % v porovnaní s existujúcim softvérovým riešením SIMROUTE, najmä v podmienkach zložitého počasia.

Toto softvérové riešenie je jedinečné tým, že zohľadňuje špecifické podmienky typické výlučne pre námornú dopravu – poveternostné podmienky, vietor, výšku a smer vĺn. Využitie tohto prístupu však zároveň otvára perspektívu adaptácie algoritmu aj na leteckú dopravu a pozemnú logistiku. V leteckej doprave majú poveternostné podmienky podobný vplyv na trasovanie letov.

V prípade pozemnej dopravy sa dopravné a logistické spoločnosti stretávajú s inými výzvami. Medzi ne patrí aj počasie, ktoré môže ovplyvniť priechodnosť ciest v závislosti od ročného obdobia a klimatickej oblasti. Klasickým problémom pozemnej dopravy sú však najmä dopravné zápchy. V takomto prípade by sa model dynamickej optimalizácie trasy mohol rozšíriť o integráciu služieb ako Google Maps, ktoré poskytujú informácie o aktuálnej dopravnej situácii a hustote premávky. Na tieto účely Google Maps poskytuje prístup k svojim zdrojom prostredníctvom API (Google Maps Platform, 2025).

2. Praktická časť

2.1. Formulácia úlohy

V praktickej časti tejto práce bola analyzovaná možnosť riešenia transportného problému pomocou jazyka Python, pričom úloha bola zostavená na základe reálnych údajov. Dáta sú verejne známe a boli zhromaždené z verejne dostupných zdrojov na internete. Ide o klasickú úlohu obchodného cestujúceho (TSP), ktorá predpokladá nájdenie najrýchlejšej optimálnej trasy, po ktorej má dopravné vozidlo prejsť, pričom každé miesto navštívi práve raz a následne sa vráti do východiskového bodu. Predpokladá sa existencia fiktívnej kuriérskej služby v meste Bratislava, ktorá má v určitý deň a čas doručiť tovar na rôzne adresy v rámci mesta. Hlavným faktorom ovplyvňujúcim rýchlosť doručenia dopravným prostriedkom je dopravná situácia na cestách, ktorá sa líši v závislosti od dňa v týždni, dennej doby a konkrétnej trasy. V závislosti od biznisových procesov môže navyše vzniknúť určitý poriadok alebo priorita, podľa ktorej má byť vykonané prechádzanie uzlami grafu. Cieľom je preskúmať, ako tieto dva faktory – dopravná vyťaženosť a priorita doručenia – ako aj ich kombinácia môžu ovplyvniť optimálne riešenie.

2.2. Zber vstupných údajov

Na zber vstupných údajov o náhodných adresách v meste Bratislava, ktoré slúžili ako vrcholy grafu dopravnej úlohy, bola použitá verejne dostupná platforma Mapy.cz (Mapy.cz, 2025). Na začiatku bol podniknutý pokus využiť najpopulárnejšiu službu Google Maps, no čoskoro sa zistilo, že z nej nie je možné získať stabilné údaje, keďže už obsahuje vstavané algoritmy, ktoré automaticky vyhľadávajú najoptimálnejšiu trasu v závislosti od dopravnej situácie a prítomnosti zápch. To znamená, že v rôznych časoch poskytoval Google Maps rôzne „optimálne“ trasy medzi rovnakými adresami, ktoré však neboli vždy najkratšie alebo najrýchlejšie. Služba Mapy.cz tieto faktory nezohľadňuje, a preto poskytuje stabilné údaje o najrýchlejších trasách bez ohľadu na cestnú premávku. Ako východiskový a zároveň konečný bod (t. j. fiktívny sklad kuriérskej služby) bola zvolená adresa Dudvážska 1. Ako doručovacie adresy bolo vybraných desať ďalších lokalít, pričom výber sa zameriaval na rôzne časti mesta, aby sa vytvorila dynamika. Pre modelovanie úlohy boli takisto určené úzke miesta grafu – mosty. Na týchto miestach v Bratislave často vznikajú dopravné zápchy a preťaženie ciest. Keďže to ovplyvňuje čas jazdy, a nie vzdialenosť, vstupné údaje sú uvádzané v minútach. Vzdialenosti ako také v tejto úlohe nie sú predmetom záujmu. Nižšie

sú uvedené vstupné údaje úlohy vo forme času cesty medzi všetkými adresami podľa údajov zo služby Mapy.cz. Úzke miesta, t. j. miesta, kde trasa vedie cez most a v špičke môže dôjsť k zdržaniu, sú vyznačené tučným písmom.

	Dudvážska 1	Wolkrova 2	Šintavská 2	Starohájska 8	Obchodná 52	Žatevná 4	Dvořákovo nábrežie 10	Parková 29	Kubačova 19	Karadžičova 8	Pasienková 1
Dudvážska 1	-	13	**16**	**13**	**15**	**21**	**15**	9	19	12	4
Wolkrova 2	13	-	5	4	**6**	**14**	**6**	**9**	**20**	**7**	**10**
Šintavská 2	**16**	5	-	6	**9**	**15**	**10**	**13**	**23**	**11**	**14**
Starohájska 8	**13**	4	6	-	**9**	**16**	**9**	**10**	**21**	**8**	**11**
Obchodná 52	**15**	**6**	**9**	**9**	-	15	7	9	15	4	13
Žatevná 4	**21**	**14**	**15**	**16**	15	-	11	**17**	26	16	**18**
Dvořákovo nábrežie 10	**15**	**6**	**10**	**9**	7	11	-	11	19	6	**13**
Parková 29	9	**9**	**13**	**10**	9	**17**	11	-	18	7	7
Kubačova 19	19	**20**	**23**	**21**	15	26	19	18	-	16	16
Karadžičova 8	12	**7**	**11**	**8**	4	16	6	7	16	-	11
Pasienková 1	4	**10**	**14**	**11**	13	**18**	**13**	7	16	11	-

Obr. 14. Vstupné údaje o čase cestovania medzi adresami v meste Bratislava

2.3. Zber údajov o dopravnej vyťaženosti

Zdrojom údajov o dopravnej vyťaženosti ciest v Bratislave bola verejne dostupná služba Tomtom.com (Tomtom.com, 2025). Informácie sú poskytnuté vo forme tabuľky, v ktorej je pre každú hodinu dňa uvedené, koľko času je v priemere potrebné na prejdenie 10 kilometrov po meste. Napríklad v pondelok medzi 4.00 a 5.00 sú cesty v meste najmenej vyťažené počas celého týždňa, zatiaľ čo v utorok medzi 8.00 a 9.00 sú najviac preťažené. Nižšie je uvedená časť tabuľky s údajmi zo spomínanej webovej stránky. Niektoré z týchto hodnôt sa stali dôležitým východiskovým bodom pre následné výpočty.

	Sun	Mon	Tue
12:00 AM	13 min 51 s	12 min 21 s	12 min 54 s
	13 min 56 s	12 min 1 s	12 min 37 s
02:00 AM	13 min 49 s	11 min 39 s	12 min 8 s
	13 min 33 s	10 min 54 s	11 min 23 s
04:00 AM	13 min 24 s	10 min 8 s	10 min 27 s
	12 min 36 s	10 min 48 s	11 min 6 s
06:00 AM	11 min 56 s	13 min 8 s	13 min 15 s
	12 min 6 s	19 min 11 s	19 min 44 s
08:00 AM	12 min 23 s	20 min 46 s	22 min 25 s
	12 min 49 s	16 min 28 s	17 min 57 s

Obr. 15. Údaje o prejdenie 10 kilometrov po meste Bratislava podľa Tomtom.com

Ako bolo uvedené vyššie, Mapy.cz poskytuje údaje o vzdialenostiach a čase bez zohľadnenia dopravnej vyťaženosti, teda za ideálnych podmienok. Zároveň podľa najnižšej zaznamenanej dopravnej vyťaženosti v dátach (pondelok o 4:00) podľa Tomtom.com, dopravný prostriedok prejde 10 kilometrov za 10 minút 8 sekúnd. Dôležitou otázkou bolo určiť dynamiku zmeny času potrebného na prekonanie najviac zaťažených úsekov trasy v rôznych hodinách. Ako časové rámce pre úlohu boli zvolené tri hodiny utorka: 7:00, 8:00 a 9:00, počas ktorých dochádza najprv k nárastu dopravného zaťaženia až do maximálne existujúcej hodnoty, a následne k jeho poklesu.

Na určenie tejto dynamiky bol vypočítaný koeficient, ktorý ukazuje, o koľko je v jednotlivých hodinách zaťaženie vyššie v porovnaní s najnižšou zaznamenanou hodnotou, ktorá bola považovaná za východiskový bod dopravného zaťaženia. Výpočet bol vykonaný podľa nasledujúceho vzorca:

$$b = \frac{a}{608}$$

, kde b – koeficient hľadanej vyťaženosti vzhľadom na najnižšiu existujúcu hodnotu pre konkrétnu hodinu,

a – čas potrebný na prekonanie 10 km v danej hodine,

608 – hodnota najnižšej zaznamenanej vyťaženosti (10 minút a 8 sekúnd) vyjadrená v sekundách.

Týmto spôsobom boli vypočítané koeficienty pre jednotlivé časové úseky.

Hodiny	Čas	V sekundách	Koeficient
7:00	19 min 44 s	1184	1.947368
8:00	22 min 25 s	1345	2.212171
9:00	17 min 57 s	1077	1.771382

Tabuľka 4. Koeficienty vyťaženosti v rôznych hodinách

Následne boli pomocou týchto koeficientov upravené pôvodné údaje o časoch cesty medzi adresami. Údaje boli spracované takým spôsobom, že všetky najviac zaťažené úseky trasy, ktoré boli v texte vyznačené tučným písmom, boli násobené jednotlivými koeficientmi postupne. Týmto spôsobom vznikli tri nové vstupné údaje s časmi ciest pre každú z hodín dopravnej špičky, ktoré sa líšia len v časoch prejazdu cez úzke miesta (mosty). Sú uvedené nižšie.

	Dudvážska 1	Wolkrova 2	Šintavská 2	Starohájska 8	Obchodná 52	Žatevná 4	Dvořákovo nábrežie 10	Parková 29	Kubačova 19	Karadžičova 8	Pasienková 1
Dudvážska 1	-	13	**31.15789**	**25.31579**	**29.21053**	**40.89474**	**29.21053**	9	19	12	4
Wolkrova 2	13	-	5	4	**11.68421**	**27.26316**	**11.68421**	**17.52632**	**38.94737**	**13.63158**	**19.47368**
Šintavská 2	**31.15789**	5	-	6	**17.52632**	**29.21053**	**19.47368**	**25.31579**	**44.78947**	**21.42105**	**27.26316**
Starohájska 8	**25.31579**	4	6	-	**17.52632**	**31.15789**	**17.52632**	**19.47368**	**40.89474**	**15.57895**	**21.42105**
Obchodná 52	**29.21053**	**11.68421**	**17.52632**	**17.52632**	-	15	7	9	15	4	13
Žatevná 4	**40.89474**	**27.26316**	**29.21053**	**31.15789**	15	-	11	**33.10526**	26	16	**35.05263**
Dvořákovo nábrežie 10	**29.21053**	**11.68421**	**19.47368**	**17.52632**	7	11	-	11	19	6	**25.31579**
Parková 29	9	**17.52632**	**25.31579**	**19.47368**	9	**33.10526**	11	-	18	7	7
Kubačova 19	19	**38.94737**	**44.78947**	**40.89474**	15	26	19	18	-	16	16
Karadžičova 8	12	**13.63158**	**21.42105**	**15.57895**	4	16	6	7	16	-	11
Pasienková 1	4	**19.47368**	**27.26316**	**21.42105**	13	**35.05263**	**25.31579**	7	16	11	-

Obr. 16. Údaje o čase cestovania medzi adresami v meste Bratislava o 7.00

	Dudvážska 1	Wolkrova 2	Šintavská 2	Starohájska 8	Obchodná 52	Žatevná 4	Dvořákovo nábrežie 10	Parková 29	Kubačova 19	Karadžičova 8	Pasienková 1
Dudvážska 1	-	13	**35.39474**	**28.75822**	**33.18257**	**46.45559**	**33.18257**	9	19	12	4
Wolkrova 2	13	-	5	4	**13.27303**	**30.97039**	**13.27303**	**19.90954**	**44.24342**	**15.4852**	**22.12171**
Šintavská 2	**35.39474**	5	-	19.90954	**33.18257**	**22.12171**	**28.75822**	**50.87993**	**24.33388**	**30.97039**	
Starohájska 8	**28.75822**	4	6	-	**19.90954**	**35.39474**	**19.90954**	**22.12171**	**46.45559**	**17.69737**	**24.33388**
Obchodná 52	**33.18257**	**13.27303**	**19.90954**	**19.90954**	-	15	7	9	15	4	13
Žatevná 4	**46.45559**	**30.97039**	**33.18257**	**35.39474**	15	-	11	**37.60691**	26	16	**39.81908**
Dvořákovo nábrežie 10	**33.18257**	**13.27303**	**22.12171**	**19.90954**	7	11	-	11	19	6	**28.75822**
Parková 29	9	**19.90954**	**28.75822**	**22.12171**	9	**37.60691**	11	-	18	7	7
Kubačova 19	19	**44.24342**	**50.87993**	**46.45559**	15	26	19	18	-	16	16
Karadžičova 8	12	**15.4852**	**24.33388**	**17.69737**	4	16	6	7	16	-	11
Pasienková 1	4	**22.12171**	**30.97039**	**24.33388**	13	**39.81908**	**28.75822**	7	16	11	-

Obr. 17. Údaje o čase cestovania medzi adresami v meste Bratislava o 8.00

	Dudvážska 1	Wolkrova 2	Šintavská 2	Starohájska 8	Obchodná 52	Žatevná 4	Dvořákovo nábrežie 10	Parková 29	Kubačova 19	Karadžičova 8	Pasienková 1
Dudvážska 1	-	13	**28.34211**	**23.02796**	**26.57072**	**37.19901**	**26.57072**	9	19	12	4
Wolkrova 2	13	-	5	4	**10.62829**	**24.79934**	**10.62829**	**15.94243**	**35.42763**	**12.39967**	**17.71382**
Šintavská 2	**28.34211**	5	-	6	**15.94243**	**26.57072**	**17.71382**	**23.02796**	**40.74178**	**19.4852**	**24.79934**
Starohájska 8	**23.02796**	4	6	-	**15.94243**	**28.34211**	**15.94243**	**17.71382**	**37.19901**	**14.17105**	**19.4852**
Obchodná 52	**26.57072**	**10.62829**	**15.94243**	**15.94243**	-	15	7	9	15	4	13
Žatevná 4	**37.19901**	**24.79934**	**26.57072**	**28.34211**	15	-	11	**30.11349**	26	16	**31.88487**
Dvořákovo nábrežie 10	**26.57072**	**10.62829**	**17.71382**	**15.94243**	7	11	-	11	19	6	**23.02796**
Parková 29	9	**15.94243**	**23.02796**	**17.71382**	9	**30.11349**	11	-	18	7	7
Kubačova 19	19	**35.42763**	**40.74178**	**37.19901**	15	26	19	18	-	16	16
Karadžičova 8	12	**12.39967**	**19.4852**	**14.17105**	4	16	6	7	16	-	11
Pasienková 1	4	**17.71382**	**24.79934**	**19.4852**	13	**31.88487**	**23.02796**	7	16	11	-

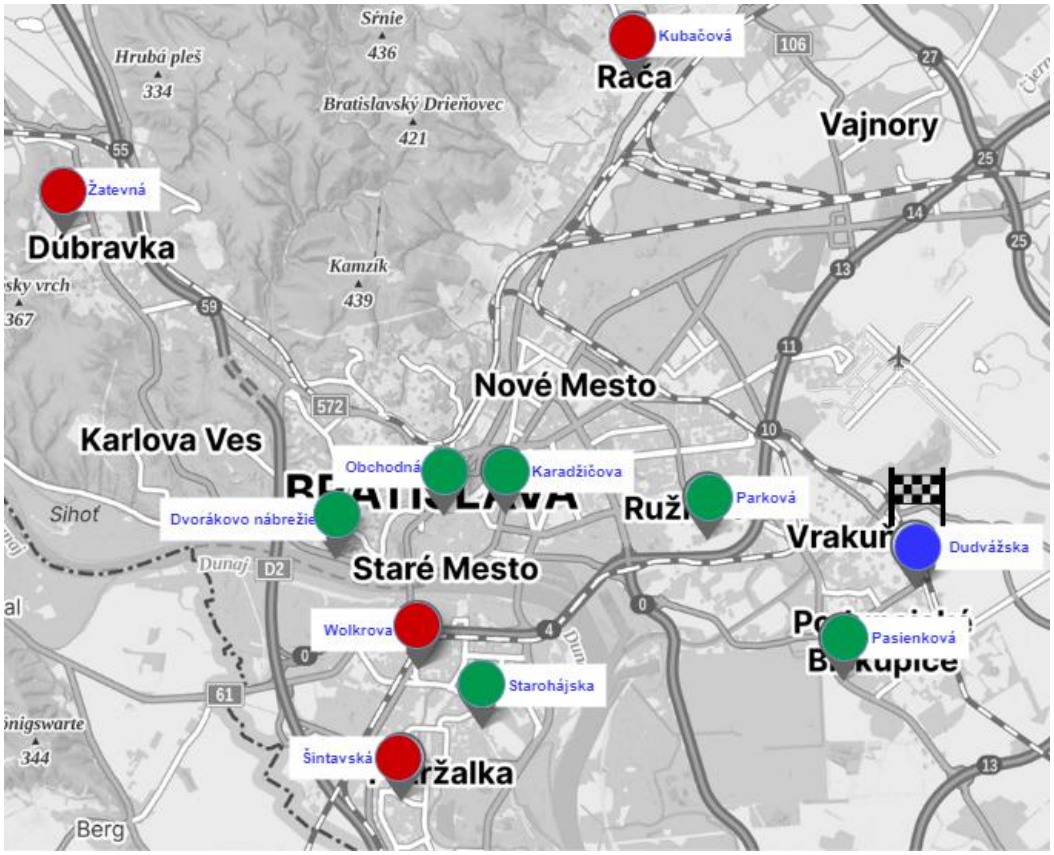
Obr. 18. Údaje o čase cestovania medzi adresami v meste Bratislava o 9.00

Týmto sa ukončila príprava vstupných údajov úlohy. Nasledujúcim krokom bola programová implementácia s využitím všetkých vytvorených dát.

2.4. Programová implementácia

2.4.1. Popis prístupov

V rámci programovej implementácie riešenia úlohy boli preskúmané dva varianty úlohy, v ktorých boli zohľadnené rôzne faktory ovplyvňujúce optimalizáciu. V prvom variante bola modelovaná prioritizácia adries, ktorá môže nastať aj v reálnom živote. Predpokladalo sa, že existujú adresy s vyššou prioritou, ktoré je potrebné navštíviť v prvom rade, a až následne prejsť všetkými ostatnými, ktoré majú nižšiu prioritu. Tieto adresy boli zvolené ako štyri najviac vzdialené body od seba navzájom, aby sa v simulácii vytvorila dostatočná dynamika pohybu. Ide o adresy: Šintavská 2, Wolkrova 2, Kubačova 19 a Žatevná 4. Nižšie je uvedená grafická vizualizácia tejto úlohy na mape mesta. Počiatočná (a zároveň konečná) adresa je vyznačená modrou farbou, adresy s vyššou prioritou červenou, a tie s nižšou prioritou zelenou.



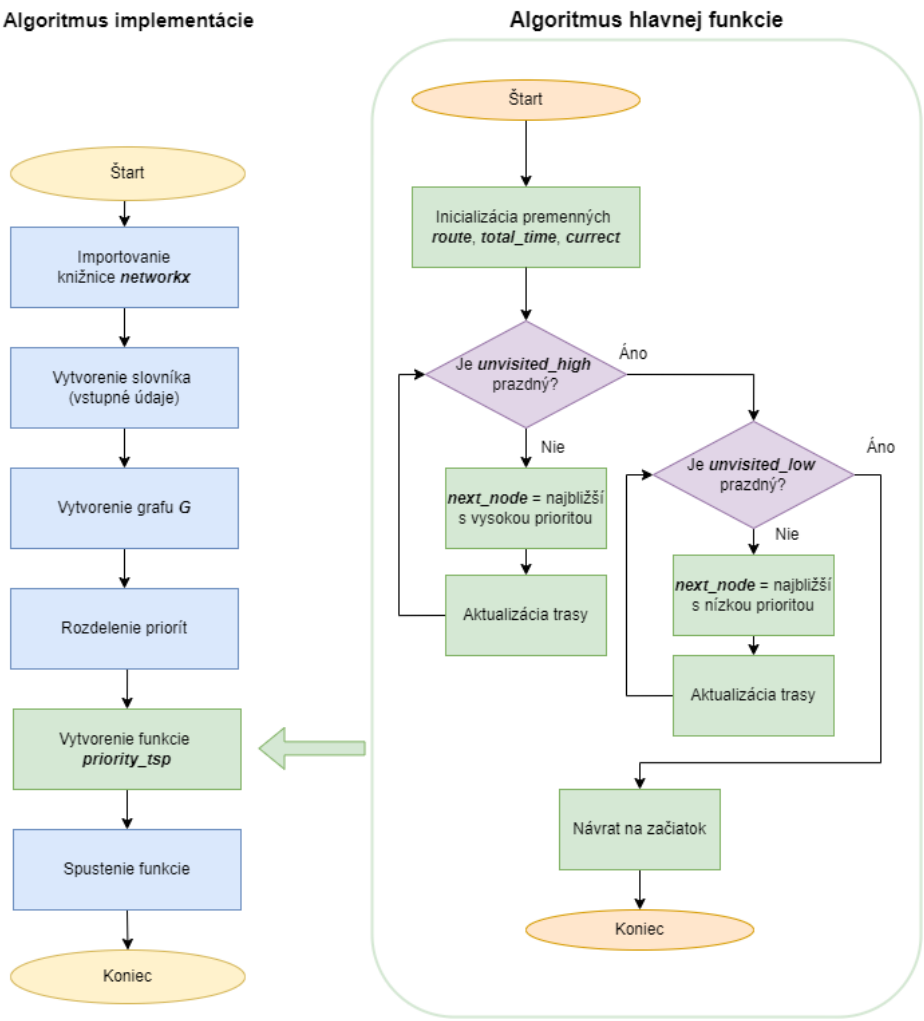
Obr. 19. Umiestnenie adries s prioritami v meste Bratislava

V druhom variante úlohy bola namiesto prioritizácie zavedená podmienka, že ak počas prechodu určitým bodom existuje neprejdená susedná adresa, ku ktorej čas jazdy nepresahuje 7 minút, je potrebné prednostne navštíviť túto adresu. Ide o pravidlo „Ak je to

blízko – zastaň tam.“ Je dôležité poznamenať, že tento variant sa líši od klasickej úlohy obchodného cestujúceho, kde sa hľadá globálne optimálne riešenie. V tomto prípade ide o lokálne optimálne rozhodovanie, kde sa uprednostňuje nasledujúci najbližší bod, ak sa nachádza v rádiuse do 7 minút. Ak takýto bod neexistuje, vyberie sa najbližší z ostatných dostupných uzlov v grafe.

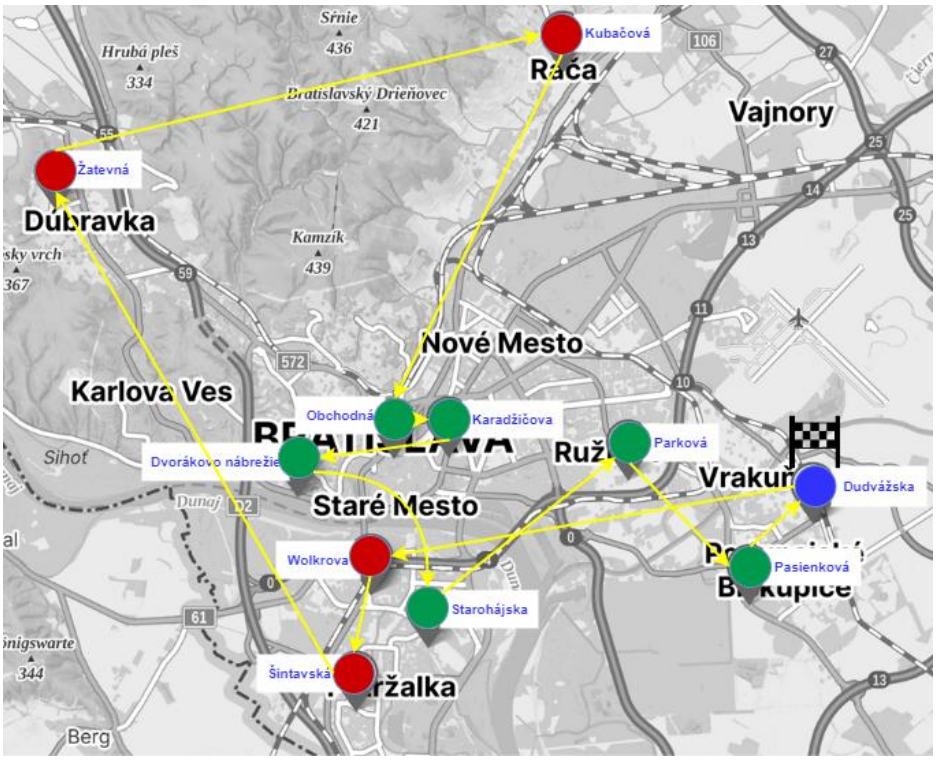
2.4.2. Variant 1

Najprv bola vyriešená úloha nájdenia optimálnej trasy za ideálnych podmienok bez dopravného preťaženia. Zdrojový kód tejto úlohy je uvedený v Prílohe č. 6. Algoritmus začína importom knižnice *networkx* (NetworkX, 2025), vytvorením slovníka *cisty_cas* s vstupnými údajmi a následne konštrukciou grafu na ich základe. Následne sú body rozdelené na tie s vyššou a nižšou prioritou. Základom riešenia je funkcia *priority_tsp*, ktorá implementuje samotný algoritmus hľadania trasy. Najprv prechádza všetky body s vyššou prioritou, pričom v každom kroku vyberá najbližší ešte nenavštívený bod z tejto množiny. Po ich prejdení sa algoritmus presúva k bodom s nižšou prioritou, kde opäť v každom kroku vyberá najbližší bod. Na konci sa trasa uzatvára – z posledného bodu sa kuriér vracia do východiskového bodu Dudvážska. Funkcia sa volá s vytvoreným grafom a zadanými prioritami. Výsledkom je zoznam bodov v poradí prechodu a celkový čas potrebný na prejdenie trasy. Tento prístup simuluje reálnu logistiku, kde niektoré doručenia majú vyššiu naliehavosť a vyžadujú prioritné vybavenie. Nižšie je uvedená schéma algoritmu.



Obr. 20. Algoritmus kódu uvedeného v Prílohe č. 6

Spustenie kódu s údajmi za ideálnych podmienok bez dopravného zaťaženia poskytlo optimálne riešenie trasy: Dudvážska – Wolkrova – Šintavská – Žatevná – Kubačova – Obchodná – Karadžičova – Dvořákovo nábrežie – Starohájska – Parková – Pasienková – Dudvážska, pričom celkový čas cesty predstavuje 114 minút. Na lepší prehľad riešenia bol taktiež vytvorený graf na reálnej mape mesta.



Obr. 21. Graf riešenia prvej úlohy na mape mesta Bratislavy

Následne bol kód postupne modifikovaný a spustený s vstupnými údajmi o dopravnej vyťaženosti o 7., 8. a 9. hodine. Zdrojové kódy sa nachádzajú v Prílohách č. 7, 8 a 9 respektíve a líšia sa iba použitými údajmi. Výsledky riešenia úlohy s rôznymi údajmi, ktoré znázorňujú zmenu optimálneho riešenia v závislosti od hodiny, pre ktorý sa trasa optimalizuje, sú uvedené nižšie v prehľadnej porovnávacej tabuľke. Časti trasy, ktoré sa odlišujú od ostatných riešení, sú vyznačené červenou farbou.

Udaje	Cistý čas	Čas po 7.00	Čas po 8.00	Čas po 9:00
Cesta	Dudvážska	Dudvážska	Dudvážska	Dudvážska
	Wolkrova	Wolkrova	Wolkrova	Wolkrova
	Šintavská	Šintavská	Šintavská	Šintavská
	Žatevná	Žatevná	Žatevná	Žatevná
	Kubačova	Kubačova	Kubačova	Kubačova
	Obchodná	Obchodná	Obchodná	Obchodná
	Karadžičova	Karadžičova	Karadžičova	Karadžičova
	Dvořákovo nábrežie	Dvořákovo nábrežie	Dvořákovo nábrežie	Dvořákovo nábrežie
	Starohájska	Parková	Parková	Parková
	Parková	Pasienková	Pasienková	Pasienková
	Pasienková	Starohájska	Starohájska	Starohájska
	Dudvážska	Dudvážska	Dudvážska	Dudvážska
Čas	114	162.95	173.27	156.09

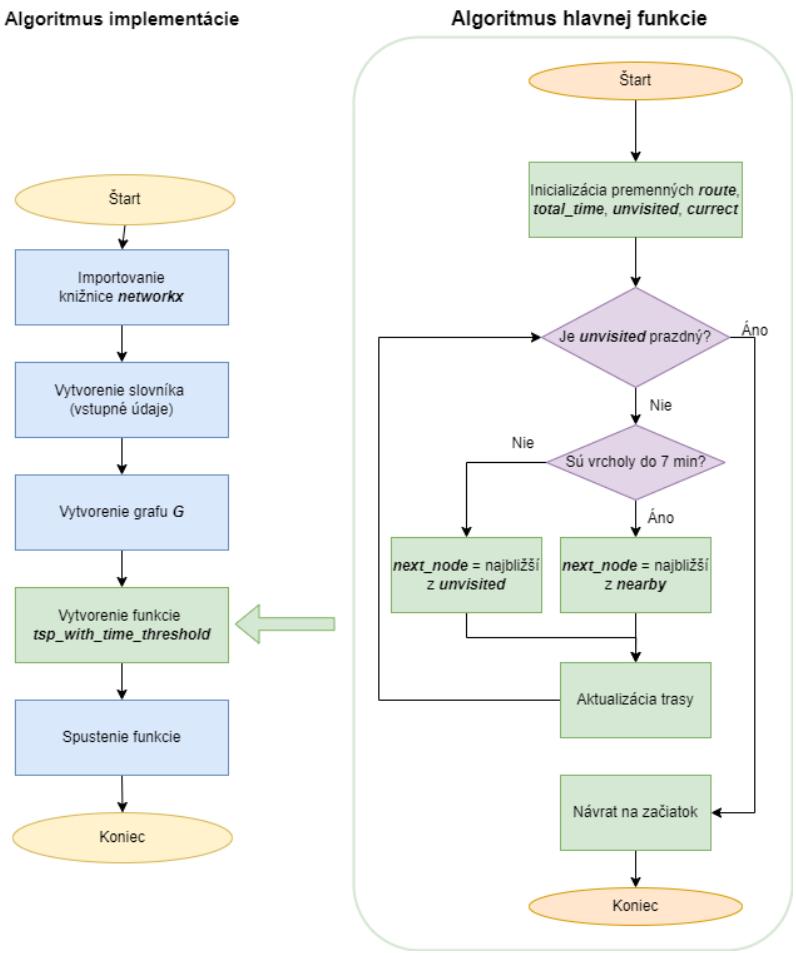
Obr. 22. Porovnanie výsledkov optimalizácie pre rôzne hodiny v prvej úlohe

Z výsledkov vyplýva, že dopravná vyťaženosť má vplyv na optimálnu trasu. Bez ohľadu na konkrétnu hodinu sa optimálne riešenie nemení v rámci skúmaného časového intervalu, avšak líši sa od trasy určenej za ideálnych podmienok bez dopravného zaťaženia. Napriek tomu, že samotná trasa zostáva rovnaká, čas potrebný na jej prejdenie sa pri dopravnom zaťažení mení. V praxi možno na základe týchto výsledkov prijať rozhodnutie, napríklad o úprave pracovného času doručovacej služby. Optimálne je začať trasu v utorok ráno najskôr o 9:00, čím možno nielen skrátiť čas doručenia, ale aj znížiť spotrebu paliva a úroveň emisií.

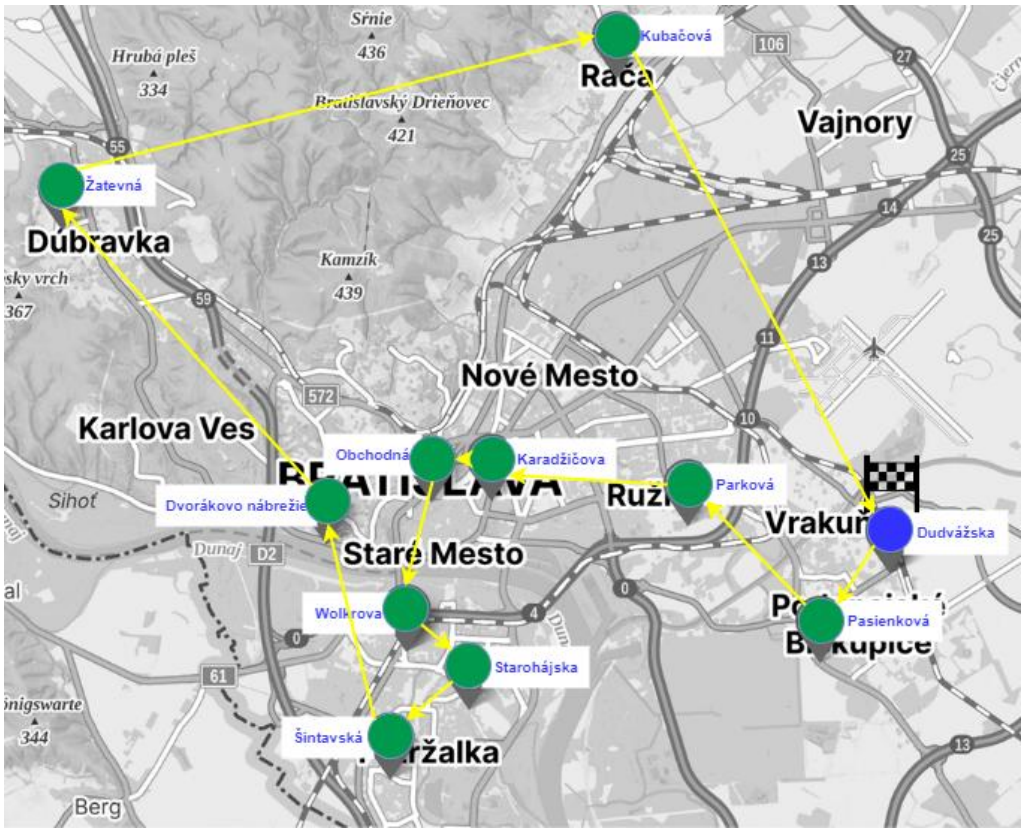
Ak by sa optimálna trasa v priebehu času menila, bolo by potrebné zmeniť prístup. V takom prípade by bolo vhodné implementovať počítadlo, ktoré bude uchovávať informáciu o čase jazdy a aktualizovať sa v každom cykle výpočtu. Vzhľadom na to, že doba jazdy sa pri rôznych podmienkach pohybuje v rozmedzí 2 až 3 hodín, možno podľa aktuálneho času v počítadle použiť rôzne vstupné dáta podľa hodiny. To by umožnilo v reálnom čase aplikovať relevantné údaje o dopravnej situácii a priebežne aktualizovať optimálnu trasu, ktorú ešte treba prejsť.

2.4.3. Variant 2

Na začiatku riešenia úlohy podľa druhého variantu bolo rovnako vykonané hľadanie optimálnej trasy za ideálnych podmienok bez dopravného preťaženia. Zdrojový kód riešenia je uvedený v Prílohe č. 10. Algoritmus implementovaný v tomto kóde hľadá trasu medzi zadanými bodmi pomocou tzv. „greedy“ prístupu s časovým prahom [14]. Pri prechode každým bodom grafu algoritmus overuje, či medzi ešte nenavštívenými bodmi existujú také, ktoré sa nachádzajú v rámci definovaného prahu – teda bližšie ako 7 minút jazdy. Ak také body existujú, algoritmus vyberie ten najbližší z nich. Ak sa žiadny bod nenachádza v danom rádiuse, algoritmus pokračuje výberom najbližšieho bodu spomedzi všetkých zostávajúcich. Potom sa aktualizuje aktuálna poloha, bod sa pridá do trasy a zároveň sa odstráni zo zoznamu nenavštívených bodov. Tento proces pokračuje, kým nie sú navštívené všetky body grafu. Tento prístup simuluje správanie kuriéra, ktorý sa vždy snaží zvoliť najbližšiu adresu, na ktorej ešte nebol. Nižšie je uvedená schéma algoritmu a vizuálne znázornenie grafu na reálnej mape mesta.



Obr. 23. Schéma algoritmu z kódu v Prílohe č. 10



Obr. 24. Graf riešenia prvej úlohy na mape mesta Bratislavy

Následne bol kód opäť postupne upravený a – rovnako ako v predchádzajúcom prístupe – spustený so vstupnými údajmi o dopravnej vyťaženosti o 7., 8. a 9. hodine. Kódy sa nachádzajú v Prílohách č. 11, 12 a 13 respektíve. Výsledky riešenia úlohy s rôznymi údajmi, ktoré znázorňujú zmenu optimálneho riešenia v závislosti od hodiny, pre ktorú sa optimalizácia vykonáva, sú uvedené v tabuľke nižšie na porovnanie. Časti trasy, ktoré sa líšia od ostatných riešení, sú vyznačené modrou farbou.

Udaje	Cistý čas	Čas po 7.00	Čas po 8.00	Čas po 9:00
Cesta	Treshhold = 7	Treshhold = 7	Treshhold = 7	Treshhold = 7
	Dudvážska	Dudvážska	Dudvážska	Dudvážska
	Pasienková	Pasienková	Pasienková	Pasienková
	Parková	Parková	Parková	Parková
	Karadžičova	Karadžičova	Karadžičova	Karadžičova
	Obchodná	Obchodná	Obchodná	Obchodná
	Wolkrova	Dvořákovo nábrežie	Dvořákovo nábrežie	Dvořákovo nábrežie
	Starohájska	Žatevná	Žatevná	Wolkrova
	Šintavská	Kubačova	Kubačova	Starohájska
	Dvořákovo nábrežie	Wolkrova	Wolkrova	Šintavská
	Žatevná	Starohájska	Starohájska	Žatevná
	Kubačova	Šintavská	Šintavská	Kubačova
	Dudvážska	Dudvážska	Dudvážska	Dudvážska
Čas	104	146.11	155.63	121.2

Obr. 25. Porovnanie výsledkov optimalizácie pre rôzne hodiny v prvej úlohe

Tieto výsledky taktiež ukazujú, že dopravná vyťaženosť má vplyv na optimálnu trasu, keďže sa líši pre každú hodinu v porovnaní s riešením bez zaťaženia. Jedným z pozoruhodných výsledkov je, že pri maximálnej vyťaženosti o 8. hodine sa trasa nemení v porovnaní s predchádzajúcou hodinou, mení sa len celkový čas jazdy, ktorý sa mierne predlžuje. Naopak, po začiatku poklesu dopravného zaťaženia po 9. hodine sa zmení polovica trasy a zároveň sa výrazne skráti celkový čas jazdy.

2.5. Analýza výsledkov

Každý z dvoch analyzovaných prístupov má svoje výhody aj nevýhody. Hlavný rozdiel medzi týmito dvoma algoritmami spočíva v spôsobe výberu nasledujúceho bodu. Prvá metóda prechádza všetky možnosti s cieľom nájsť globálne najkratšiu trasu, pričom ignoruje lokálne výhody. Algoritmus najskôr kompletne prejde všetky body s vysokou prioritou, pričom ich navštevuje podľa najkratšej vzdialenosti od aktuálnej polohy. Až následne prechádza body s nižšou prioritou, opäť výberom vždy najbližšieho bodu. Tento prístup lepšie zodpovedá reálnym logistickým scenárom, kde niektoré objednávky alebo klienti majú vyššiu dôležitosť a vyžadujú prednostné doručenie, bez ohľadu na to, ako ďaleko sa nachádzajú. Nezohľadňuje časové prahy, ale dôsledne rešpektuje určené priority.

Naopak, druhá metóda s časovým prahom implementuje greedy lokálny prístup, pri ktorom sa nasledujúci bod vyberá spomedzi najbližších dostupných. Tým sa minimalizujú straty času na krátkych úsekoch, hoci nie je zaručené globálne optimálne riešenie. Trasa sa

tak zostavuje s cieľom minimalizovať celkový čas presunov, pričom sa uprednostňujú blízke body, ale zároveň sa nezasekne, ak žiadne také nie sú. Tento prístup je vhodný pre situácie, kde je cieľom znížiť dlhé presuny, no neexistuje pevne stanovená priorita medzi doručovacími bodmi. Na vizuálne porovnanie výsledkov oboch prístupov bola vytvorená spoločná porovnávacia tabuľka.

Udaje	Cistý čas		Čas po 7.00		Čas po 8.00		Čas po 9:00	
		Treshhold = 7	Prioritizácia	Treshhold = 7	Prioritizácia	Treshhold = 7	Prioritizácia	Treshhold = 7
Cesta	Dudvážska 1	Dudvážska	Dudvážska	Dudvážska	Dudvážska	Dudvážska	Dudvážska	Dudvážska
	Wolkrova 2	Pasienková	Wolkrova	Pasienková	Wolkrova	Pasienková	Wolkrova	Pasienková
	Šintavská 2	Parková	Šintavská	Parková	Šintavská	Parková	Šintavská	Parková
	Žatevná 4	Karadžičova	Žatevná	Karadžičova	Žatevná	Karadžičova	Žatevná	Karadžičova
	Kubačova 19	Obchodná	Kubačova	Obchodná	Kubačova	Obchodná	Kubačova	Obchodná
	Obchodná 52	Wolkrova	Obchodná	Dvořákovo nábrežie	Obchodná	Dvořákovo nábrežie	Obchodná	Dvořákovo nábrežie
	Karadžičova 8	Starohájska	Karadžičova	Žatevná	Karadžičova	Žatevná	Karadžičova	Wolkrova
	Dvořákovo nábrežie 10	Šintavská	Dvořákovo nábrežie	Kubačova	Dvořákovo nábrežie	Kubačova	Dvořákovo nábrežie	Starohájska
	Starohájska 8	Dvořákovo nábrežie	Parková	Wolkrova	Parková	Wolkrova	Parková	Šintavská
	Parková 29	Žatevná	Pasienková	Starohájska	Pasienková	Starohájska	Pasienková	Žatevná
	Pasienková 1	Kubačova	Starohájska	Šintavská	Starohájska	Šintavská	Starohájska	Kubačova
	Dudvážska 1	Dudvážska	Dudvážska	Dudvážska	Dudvážska	Dudvážska	Dudvážska	Dudvážska
Čas	114	104	162.95	146.11	173.27	155.63	156.09	121.2

Obr. 26: Porovnanie výsledkov pre rôzne časové úseky v rámci oboch úloh

V prvom rade je zrejmé, že v rámci riešenia dopravných úloh je nevyhnutné zohľadniť vyťaženosť cestnej premávky, ako aj fakt, že dopravná situácia sa mení v priebehu dňa. Kľúčovým faktorom je existencia spoľahlivého zdroja údajov, ktorý poskytne aktuálne informácie o zmenách v tomto dynamickom prostredí.

V tomto výskume boli analyzované stabilné podmienky, pri ktorých sa údaje menia v závislosti od času štartu, no nemenia sa počas samotného pohybu, keďže nájdené optimálne riešenia sa vo väčšine prípadov nemenili v rámci sledovaného časového okna – s výnimkou poslednej hodiny v druhom prístupe. V prípade, že sa optimálna trasa počas prepravy mení (čo je pravdepodobné pri dlhších časových intervaloch), vzniká potreba zásadne zmeniť prístup k riešeniu problému. V takom prípade možno v programovom riešení implementovať počítadlo, ktoré bude uchovávať informácie o čase jazdy a aktualizovať sa pri každom cykle algoritmu. Keďže doba jazdy sa v rôznych podmienkach pohybuje v rozmedzí 2 až 3 hodín, pri zmene času v počítadle je možné použiť rôzne vstupné údaje podľa aktuálnej hodiny. To umožní v reálnom čase využívať relevantné údaje o dopravnej situácii a priebežne aktualizovať optimálnu trasu, ktorú je ešte potrebné prejsť.

Čo sa týka dvoch analyzovaných prístupov, výber vhodného riešenia môže závisieť od modelu služieb, ktorý by daná kuriérska spoločnosť ponúkala. Celkovo druhý prístup vykazuje rýchlejší prechod trasou v ktorúkoľvek hodinu, čo šetrí nielen čas a palivo, ale aj znižuje emisie.

Zatiaľ čo náklady na čas a palivo sa dajú kompenzovať nastavením cien služieb podľa ich rýchlosti a komfortu, emisie CO_2 nie je možné kompenzovať. Zvlášť v kontexte nových smerníc EÚ (European Commission, 2023), ktoré majú v súčasnosti motivačný charakter, no v budúcnosti môžu firmy zaväzovať k zníženiu dopadu na životné prostredie, sa výber modelu doručovania musí prispôsobiť aj týmto ekologickým faktorom.

Z výsledkov tiež vyplýva, že má zmysel prispôsobiť pracovný harmonogram doručovacej služby vyťaženosti ciest, a to bez ohľadu na zvolený model poskytovania služieb. V tomto prípade sa odporúča začínať doručovanie najskôr o 9:00, pretože tak možno predísť menej výhodným podmienkam.

3. Záver

V rámci tejto diplomovej práce boli preskúmané možnosti využitia programovacieho jazyka Python pri riešení ekonomických optimalizačných úloh, totiž v oblasti logistiky a dopravy. Pozornosť hlavne bola venovaná modelovaniu rozvozových trás v mestských podmienkach s využitím algoritmov na grafoch.

V teoretickej časti boli rozobrané všeobecné typy úloh a príklady ich riešení spolu s opisom algoritmov. Osobitný dôraz bol kladený na analýzu vedeckej literatúry, ktorá prezentuje reálne prípady využitia Python pri riešení zložitejších úloh, jeho integráciu do riešení s ďalšími softvérovými komponentmi, ich vzájomnú interakciu a miesto Pythonu v architektúre celého systému. Python disponuje knižnicami pre takmer každý typ úloh, ktoré sa už v praxi často považujú za štandardizované. Štandardizácia predstavuje efektívne a rýchle riešenie najmä v oblastiach, kde sa úlohy často opakujú alebo sú si veľmi podobné. Na druhej strane však takýto prístup môže znamenať istú nepružnosť pri riešení nových, komplexných alebo špecifických problémov, ktoré si vyžadujú prispôsobenie alebo použitie úplne iných nástrojov.

V praktickej časti boli implementované dva varianty algoritmov:

1. Problém obchodného cestujúceho s prioritou, kde sa najskôr navštevujú uzly s vyššou prioritou a až následne tie s nižšou. V rámci každej skupiny sa ďalší bod vyberá na základe minimálneho času presunu.

2. Problém obchodného cestujúceho s časovým prahom, ktorý iteratívne vyberá najbližší bod doručenia v prípade, že sa nachádza v rámci definovaného časového limitu. Ak takýto bod nie je dostupný, algoritmus pokračuje k celkovo najbližšiemu dostupnému bodu.

Na implementáciu bola použitá knižnica NetworkX, ktorá umožnila konštrukciu grafu trasy a uchovávanie časových váh medzi uzlami.

Bol vytvorený model reálneho prípadu — výpočet doručovacích trás v meste Bratislava, kde uzly reprezentovali skutočné adresy a váhy medzi nimi zodpovedali odhadovanému času presunu. Na základe tohto modelu bolo možné porovnať rôzne prístupy k plánovaniu trás.

Stanovený cieľ práce bol úspešne dosiahnutý — praktickým príkladom bola preukázaná vhodnosť použitia Pythonu na modelovanie optimalizačných úloh v oblasti

logistiky. Implementované algoritmy vykazujú dobrú flexibilitu, rozšíriteľnosť a praktickú použiteľnosť v menších firmách alebo logistických oddeleniach.

Praktický prínos tejto práce spočíva v návrhu základného, no funkčného systému, ktorý môže slúžiť ako východisková platforma pre ďalší vývoj sofistikovanejších logistických aplikácií. Potenciálne smery ďalšieho rozvoja zahŕňajú integráciu s geografickými dátami a mapovými API službami za účelom využitia dynamických údajov v reálnom čase, vytvorenie interaktívneho používateľského rozhrania pre plánovanie trás v reálnom čase, ako aj aplikáciu metaheuristických algoritmov, konkrétne genetických algoritmov, algoritmu mravčích kolónií či tabu search.

Zoznam príloh

Príloha č. 1 – Riešenie príkladu analýzy sietí

Príloha č. 2 – Riešenie príkladu optimalizácie trasy

Príloha č. 3 – Riešenie príkladu optimalizácie rozvrhu

Príloha č. 4 – Vizualizácia výsledkov optimalizácie rozvrhu

Príloha č. 5 – Riešenie príkladu optimalizácie nákladov

Príloha č. 6 – Riešenie prvej praktickej úlohy za ideálnych podmienok bez dopravného preťaženia

Príloha č. 7 – Riešenie prvej praktickej úlohy za podmienok dopravného preťaženia o 7.00

Príloha č. 8 – Riešenie prvej praktickej úlohy za podmienok dopravného preťaženia o 8.00

Príloha č. 9 – Riešenie prvej praktickej úlohy za podmienok dopravného preťaženia o 9.00

Príloha č. 10 – Riešenie druhej praktickej úlohy za ideálnych podmienok bez dopravného preťaženia

Príloha č. 11 – Riešenie druhej praktickej úlohy za podmienok dopravného preťaženia o 7.00

Príloha č. 12 – Riešenie druhej praktickej úlohy za podmienok dopravného preťaženia o 8.00

Príloha č. 13 – Riešenie druhej praktickej úlohy za podmienok dopravného preťaženia o 9.00

Príloha č. 1

```
import networkx as nx
import matplotlib.pyplot as plt

# Vytvorenie grafu
G = nx.DiGraph()
edges = [
    ("Bratislava", "Trnava", 47), ("Bratislava", "Nitra", 94),
    ("Trnava", "Nitra", 118), ("Trnava", "B.Bystrica", 235),
    ("Nitra", "B.Bystrica", 71),
    ("B.Bystrica", "Košice", 188),
    ("Košice", "Trnava", 165)
]

G.add_weighted_edges_from(edges)

# Analýza centrality uzlov na základe najkratších ciest
betweenness = nx.betweenness_centrality(G, weight='weight')
most_central = max(betweenness, key=betweenness.get)

print("Centralita uzlov (betweenness):", betweenness)
print(f"Najviac zaťažený uzol: {most_central}")

# Vizualizácia grafu
pos = nx.spring_layout(G)
plt.figure(figsize=(6, 4))
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000,
edge_color='gray')
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.title("Analýza siete: betweenness centrality")
plt.show()
```

Príloha č. 2

```
from pulp import LpProblem, LpMinimize, LpVariable, lpSum, LpStatus


# Vzdialenosti medzi uzlami
locations = ["Centrum", "Klient 1", "Klient 2", "Klient 3"]
distances = {
    ("Centrum", "Klient 1"): 16, ("Centrum", "Klient 2"): 14, ("Centrum", "Klient 3"): 12,
    ("Klient 1", "Klient 2"): 19, ("Klient 1", "Klient 3"): 9, ("Klient 2", "Klient 3"): 20
}


# Definícia problému
problem = LpProblem("Optimalizacia_trasy", LpMinimize)


# Premenné rozhodovania
x = LpVariable.dicts("x", [(i, j) for i in locations for j in locations if i != j], cat='Binary')
u = LpVariable.dicts("u", locations, lowBound=0, cat='Continuous')


# Cieľová funkcia: minimalizácia celkovej vzdialenosti
problem += lpSum(x[i, j] * distances.get((i, j), distances.get((j, i), 0)) for i in locations for j in locations if i != j)


# Obmedzenia: Každé miesto musí byť navštívené práve raz
for i in locations:
```

```
    problem += lpSum(x[i, j] for j in locations if i != j) == 1
    problem += lpSum(x[j, i] for j in locations if i != j) == 1

# Obmedzenie na elimináciu podcestných cyklov (subtour elimination constraints)
for i in locations[1:]:
    for j in locations[1:]:
        if i != j:
            problem += u[i] - u[j] + len(locations) * x[i, j] <= len(locations) - 1

# Riešenie problému
problem.solve()

# Výstup výsledkov
print("Status riešenia:", LpStatus[problem.status])
if LpStatus[problem.status] == 'Optimal':
    print("Optimálna trasa:")
    route = []
    current = "Centrum"
    while len(route) < len(locations):
        for j in locations:
            if current != j and x[current, j].varValue > 0.5:
                route.append((current, j))
                current = j
                break
```

```
    for step in route:
        print(f"{step[0]} → {step[1]}")
else:
    print("Nenašlo sa optimálne riešenie.")
```

Príloha č. 3

```
import random
from deap import base, creator, tools, algorithms

# Nastavenie pevného semena pre reprodukovateľnosť
random.seed(42)

# Počet autobusov a trás
autobusy = 10
trasy = 3

# Minimálne a maximálne požiadavky na autobusy na trasách
min_autobusov = [4, 2, 2]  # Každá trasa musí mať aspoň toto množstvo autobusov
max_autobusov = [6, 4, 4]  # Každá trasa nesmie mať viac ako toto množstvo autobusov

# Definícia optimalizačného problému
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
```

```
# Inicializácia jednotlivcov - generovanie platného rozdelenia autobusov
def inicializuj_individual():
    rozdelenie = [None] * autobusy
    zostavajuci_autobusy = autobusy
    dostupne_trasy = list(range(trasy))

    for i in dostupne_trasy:
        min_val = min_autobusov[i]
        max_val = min(max_autobusov[i], zostavajuci_autobusy)
        if max_val < min_val:
            pridelenie = min_val
        else:
            pridelenie = random.randint(min_val, max_val)
        zostavajuci_autobusy -= pridelenie

        volne_indexy = [idx for idx, val in enumerate(rozdelenie) if val is None]
        for _ in range(pridelenie):
            if volne_indexy:
                index = random.choice(volne_indexy)
                rozdelenie[index] = i
                volne_indexy.remove(index)

    for idx in range(len(rozdelenie)):
        if rozdelenie[idx] is None:
```

```
                rozdelenie[idx] = random.choice(dostupne_trasy)

            return creator.Individual(rozdelenie)

        # Hodnotiaca funkcia - minimalizácia rozdielu medzi trasami
        def hodnotenie(individual):
            zatazenie = [individual.count(i) for i in range(trasy)]
            penalizacia = sum(max(0, min_autobusov[i] - zatazenie[i]) + max(0, zatazenie[i]
- max_autobusov[i]) for i in range(trasy))
            return max(zatazenie) + penalizacia,

        # Inicializácia DEAP
        toolbox = base.Toolbox()
        toolbox.register("individual", inicializuj_individual)
        toolbox.register("population", tools.initRepeat, list, toolbox.individual)
        toolbox.register("evaluate", hodnotenie)
        toolbox.register("mate", tools.cxTwoPoint)
        toolbox.register("mutate", tools.mutUniformInt, low=0, up=trasy-1, indpb=0.1)
        toolbox.register("select", tools.selTournament, tournsize=3)

        def optimalizuj():
            pop = toolbox.population(n=50)
            generations = 50

            for gen in range(generations):
```

```
        algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.1, ngen=1, stats=None, halloffame=None, verbose=False)

    najlepsie_riesenie = tools.selBest(pop, k=1)[0]
    print("Najlepšie rozdelenie autobusov:")
    for i in range(trasy):
        print(f"Trasa {i+1}: {najlepsie_riesenie.count(i)} autobusov")

optimalizuj()
```

Príloha č. 4

```
import random
import matplotlib.pyplot as plt
from deap import base, creator, tools, algorithms

# Nastavenie pevného semena pre reprodukovateľnosť
random.seed(42)

# Počet autobusov a trás
autobusy = 10
trasy = 3

# Minimálne a maximálne požiadavky na autobusy na trasách
min_autobusov = [4, 2, 2]  # Každá trasa musí mať aspoň toto množstvo autobusov
```

```
max_autobusov = [6, 4, 4]  # Každá trasa nesmie mať viac ako toto množstvo autobusov


# Definícia optimalizačného problému
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)


# Inicializácia jednotlivcov - generovanie platného rozdelenia autobusov
def inicializuj_individual():
    rozdelenie = [None] * autobusy
    zostavajuci_autobusy = autobusy
    dostupne_trasy = list(range(trasy))

    for i in dostupne_trasy:
        min_val = min_autobusov[i]
        max_val = min(max_autobusov[i], zostavajuci_autobusy)
        if max_val < min_val:
            pridelenie = min_val
        else:
            pridelenie = random.randint(min_val, max_val)
        zostavajuci_autobusy -= pridelenie

        volne_indexy = [idx for idx, val in enumerate(rozdelenie) if val is None]
        for _ in range(pridelenie):
            if volne_indexy:
```

```
                index = random.choice(volne_indexy)
                rozdelenie[index] = i
                volne_indexy.remove(index)

        for idx in range(len(rozdelenie)):
            if rozdelenie[idx] is None:
                rozdelenie[idx] = random.choice(dostupne_trasy)

        return creator.Individual(rozdelenie)

    # Hodnotiaca funkcia - minimalizácia rozdielu medzi trasami
    def hodnotenie(individual):
        zatazenie = [individual.count(i) for i in range(trasy)]
        penalizacia = sum(max(0, min_autobusov[i] - zatazenie[i]) + max(0, zatazenie[i]
- max_autobusov[i]) for i in range(trasy))
        return max(zatazenie) + penalizacia,

    # Inicializácia DEAP
    toolbox = base.Toolbox()
    toolbox.register("individual", inicializuj_individual)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("evaluate", hodnotenie)
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutUniformInt, low=0, up=trasy-1, indpb=0.1)
    toolbox.register("select", tools.selTournament, tournsize=3)
```

```
def optimalizuj():
    pop = toolbox.population(n=50)
    generations = 50
    best_fitness_values = []
    avg_fitness_values = []
    best_distributions = []

    # Uloženie inicializačného rozdelenia
    best_individual = tools.selBest(pop, k=1)[0]
    best_distributions.append([best_individual.count(i) for i in range(trasy)])
    best_fitness_values.append(hodnotenie(best_individual)[0])
    avg_fitness_values.append(sum(hodnotenie(ind)[0] for ind in pop) / len(pop))

    for gen in range(generations):
        algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.1, ngen=1, stats=None,
halloffame=None, verbose=False)
        best_individual = tools.selBest(pop, k=1)[0]
        fitness_values = [hodnotenie(ind)[0] for ind in pop]
        best_fitness_values.append(hodnotenie(best_individual)[0])
        avg_fitness_values.append(sum(fitness_values) / len(fitness_values))
        best_distributions.append([best_individual.count(i) for i in range(trasy)])

    najlepsie_riesenie = tools.selBest(pop, k=1)[0]
    print("Najlepšie rozdelenie autobusov:")
```

```
        for i in range(trasy):
            print(f"Trasa {i+1}: {najlepsie_riesenie.count(i)} autobusov")

        # Vizuálna reprezentácia evolúcie rozdelenia autobusov
        fig, ax = plt.subplots(figsize=(12, 5))
        generations_range = range(1, generations + 1)  # +1 pre inicializačné rozdelenie
        for i in range(trasy):
            ax.plot(generations_range, [dist[i] for dist in best_distributions[1:]], marker='o', linestyle='-', label=f'Trasa {i+1}')

        ax.set_xlabel("Generácia")
        ax.set_ylabel("Počet autobusov na trase")
        ax.set_title("Evolúcia rozdelenia autobusov počas generácií")
        ax.legend()
        ax.grid()
        plt.show()

    optimalizuj()
```

Príloha č. 5

```
import numpy as np
from scipy.optimize import linprog
import pandas as pd

# Vstupné dáta
```

```
supply = np.array([100, 150, 200])  # Množstvo tovaru v skladoch Bratislava, Košice, Žilina

demand = np.array([80, 90, 110, 170])  # Požiadavky obchodov v mestách Nitra, Prešov, Trnava, Banská Bystrica


# Prepravné náklady (matica 3x4)

cost = np.array([

   [2, 3, 1, 4],  # Zo skladu Bratislava do obchodov Nitra, Prešov, Trnava, Banská Bystrica

   [3, 1, 2, 3],  # Zo skladu Košice do obchodov Nitra, Prešov, Trnava, Banská Bystrica

   [4, 2, 5, 1]  # Zo skladu Žilina do obchodov Nitra, Prešov, Trnava, Banská Bystrica

])


# Počet skladov a obchodov

num_supply = len(supply)

num_demand = len(demand)


# Rozbalenie nákladov do lineárneho tvaru pre linprog

c = cost.flatten()


# Obmedzenia pre množstvo dodávok (<= supply)

A_supply = np.zeros((num_supply, num_supply * num_demand))

for i in range(num_supply):

   A_supply[i, i * num_demand:(i + 1) * num_demand] = 1

b_supply = supply
```

```
# Obmedzenia pre dopyt (>= demand)
A_demand = np.zeros((num_demand, num_supply * num_demand))
for j in range(num_demand):
    A_demand[j, j::num_demand] = 1
b_demand = demand

# Kombinácia obmedzení dodávok a dopytu
A = np.vstack([A_supply, A_demand])
b = np.hstack([b_supply, b_demand])

# Parametre pre lineárnu optimalizáciu
bounds = [(0, None)] * (num_supply * num_demand)

# Riešenie úlohy lineárneho programovania
result = linprog(c, A_eq=A, b_eq=b, bounds=bounds, method='highs')

# Formátovanie výsledkov do prehľadnej tabuľky
if result.success:
    transportation_plan = result.x.reshape((num_supply, num_demand))
    total_cost = result.fun

    # Zobrazenie výsledkov
    transport_df = pd.DataFrame(transportation_plan,
```

```
                index=[f'Sklad {city}' for city in ['Bratislava', 'Košice', 'Žilina']],
                columns=[f'Obchod {city}' for city in ['Nitra', 'Prešov', 'Trnava',
'Banská Bystrica']])

    print('Prepravný plán:')
    print(transport_df)
    print(f'Celkové náklady na prepravu: {total_cost:.2f}')
else:
    print('Nepodarilo sa nájsť optimálny prepravný plán')
```

Príloha č. 6

```
# Preimportujeme potrebnú knižnicu
import networkx as nx

# Vytvoríme slovník s časom
cisty_cas = {
    "Dudvážska": {"Wolkrova": 13, "Šintavská": 16, "Starohájska": 13, "Obchodná":
15, "Žatevná": 21,
                  "Dvořákovo nábrežie": 15, "Parková": 9, "Kubačova": 19,
"Karadžičova": 12, "Pasienková": 4},
    "Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 6, "Žatevná": 14,
"Dvořákovo nábrežie": 6,
                 "Parková": 9, "Kubačova": 20, "Karadžičova": 7, "Pasienková": 10},
    "Šintavská": {"Starohájska": 6, "Obchodná": 9, "Žatevná": 15, "Dvořákovo
nábrežie": 10,
                  "Parková": 13, "Kubačova": 23, "Karadžičova": 11, "Pasienková": 14},
```

```
        "Starohájska": {"Obchodná": 9, "Žatevná": 16, "Dvořákovo nábrežie": 9, "Parková": 10,
                        "Kubačova": 21, "Karadžičova": 8, "Pasienková": 11},
        "Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15,
                     "Karadžičova": 4, "Pasienková": 13},
        "Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 17, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 18},
        "Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 13},
        "Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},
        "Kubačova": {"Karadžičova": 16, "Pasienková": 16},
        "Karadžičova": {"Pasienková": 11}
    }

    # Vytvoríme graf
    G = nx.Graph()

    # Pridáme uzly
    for location in cisty_cas.keys():
        G.add_node(location)

    # Pridáme hrany (čas medzi bodmi)
    for start, destinations in cisty_cas.items():
        for end, distance in destinations.items():
            G.add_edge(start, end, weight=distance)
```

```
# Určíme body s vyššou prioritou a všetky ostatné
high_priority = {"Šintavská", "Wolkrova", "Kubačova", "Žatevná"}
low_priority = set(G.nodes) - high_priority - {"Dudvážska"}

# Funkcia na vyhľadanie trasy s ohľadom na prioritu
def priority_tsp(graph, start, high_priority, low_priority):
    route = [start]
    total_time = 0
    unvisited_high = set(high_priority)
    unvisited_low = set(low_priority)
    current = start

    # Najprv prejdeme cez body s vyššou prioritou
    while unvisited_high:
        next_node = min(unvisited_high, key=lambda node: graph[current][node]['weight'])
        total_time += graph[current][next_node]['weight']
        route.append(next_node)
        unvisited_high.remove(next_node)
        current = next_node

    # Potom prejdeme cez body s nižšou prioritou
    while unvisited_low:
```

```
            next_node = min(unvisited_low, key=lambda node: graph[current][node]['weight'])
            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited_low.remove(next_node)
            current = next_node

        # Vrátime sa do východiskového bodu
        total_time += graph[current][start]['weight']
        route.append(start)

        return route, total_time

    # Nájdeme trasu s ohľadom na prioritu
    best_priority_route, min_priority_time = priority_tsp(G, "Dudvážska", high_priority, low_priority)

    # Výstup výsledkov
    best_priority_route, min_priority_time
```

Príloha č. 7

```
# Preimportujeme potrebné knižnice po reštarte prostredia
import networkx as nx

# Obnovíme slovník s časom
```

```
cas_Tue_7_am = {
    "Dudvážska": {"Wolkrova": 13, "Šintavská": 31.16, "Starohájska": 25.32, "Obchodná": 29.21, "Žatevná": 40.89,
                  "Dvořákovo nábrežie": 29.21, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},
    "Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 11.68, "Žatevná": 27.26, "Dvořákovo nábrežie": 11.68,
                 "Parková": 17.53, "Kubačova": 38.95, "Karadžičova": 13.63, "Pasienková": 19.47},
    "Šintavská": {"Starohájska": 6, "Obchodná": 17.53, "Žatevná": 29.21, "Dvořákovo nábrežie": 19.47,
                  "Parková": 25.32, "Kubačova": 44.79, "Karadžičova": 21.42, "Pasienková": 27.26},
    "Starohájska": {"Obchodná": 17.53, "Žatevná": 31.16, "Dvořákovo nábrežie": 17.53, "Parková": 19.47,
                    "Kubačova": 40.89, "Karadžičova": 15.58, "Pasienková": 21.42},
    "Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15, "Karadžičova": 4, "Pasienková": 13},
    "Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 33.11, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 35.05},
    "Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 25.32},
    "Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},
    "Kubačova": {"Karadžičova": 16, "Pasienková": 16},
    "Karadžičova": {"Pasienková": 11}
}
```

```
# Vytvoríme graf
G = nx.Graph()

# Pridáme uzly
for location in cas_Tue_7_am.keys():
    G.add_node(location)

# Pridáme hrany (čas medzi bodmi)
for start, destinations in cas_Tue_7_am.items():
    for end, distance in destinations.items():
        G.add_edge(start, end, weight=distance)

# Určíme body s vyššou prioritou a všetky ostatné
high_priority = {"Šintavská", "Wolkrova", "Kubačova", "Žatevná"}
low_priority = set(G.nodes) - high_priority - {"Dudvážska"}

# Funkcia na vyhľadanie trasy s ohľadom na prioritu
def priority_tsp(graph, start, high_priority, low_priority):
    route = [start]
    total_time = 0
    unvisited_high = set(high_priority)
    unvisited_low = set(low_priority)
    current = start
```

```
        # Najprv prejdeme cez body s vyššou prioritou
        while unvisited_high:
            next_node = min(unvisited_high, key=lambda node: graph[current][node]['weight'])
            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited_high.remove(next_node)
            current = next_node

        # Potom prejdeme cez body s nižšou prioritou
        while unvisited_low:
            next_node = min(unvisited_low, key=lambda node: graph[current][node]['weight'])
            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited_low.remove(next_node)
            current = next_node

        # Vrátime sa do východiskového bodu
        total_time += graph[current][start]['weight']
        route.append(start)

        return route, total_time

    # Nájdeme trasu s ohľadom na prioritu
```

best_priority_route, min_priority_time = priority_tsp(G, "Dudvážska", high_priority, low_priority)

Výstup výsledkov

best_priority_route, min_priority_time

Príloha č. 8

Preimportujeme potrebné knižnice po reštarte prostredia

import networkx as nx

Obnovíme slovník s časom

cas_Tue_8_am = {

"Dudvážska": {"Wolkrova": 13, "Šintavská": 35.39, "Starohájska": 28.76, "Obchodná": 33.18, "Žatevná": 46.46,

"Dvořákovo nábrežie": 33.18, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},

"Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 13.27, "Žatevná": 30.97, "Dvořákovo nábrežie": 13.27,

"Parková": 19.91, "Kubačova": 44.24, "Karadžičova": 15.49, "Pasienková": 22.12},

"Šintavská": {"Starohájska": 6, "Obchodná": 19.91, "Žatevná": 33.18, "Dvořákovo nábrežie": 22.12,

"Parková": 28.76, "Kubačova": 50.88, "Karadžičova": 24.33, "Pasienková": 30.97},

"Starohájska": {"Obchodná": 19.91, "Žatevná": 35.39, "Dvořákovo nábrežie": 19.91, "Parková": 22.12,

"Kubačova": 46.46, "Karadžičova": 17.70, "Pasienková": 24.33},

```
        "Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15, "Karadžičova": 4, "Pasienková": 13},
        "Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 37.61, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 39.82},
        "Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 28.76},
        "Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},
        "Kubačova": {"Karadžičova": 16, "Pasienková": 16},
        "Karadžičova": {"Pasienková": 11}
    }

    # Vytvoríme graf
    G = nx.Graph()

    # Pridáme uzly
    for location in cas_Tue_8_am.keys():
        G.add_node(location)

    # Pridáme hrany (čas medzi bodmi)
    for start, destinations in cas_Tue_8_am.items():
        for end, distance in destinations.items():
            G.add_edge(start, end, weight=distance)

    # Určíme body s vyššou prioritou a všetky ostatné
    high_priority = {"Šintavská", "Wolkrova", "Kubačova", "Žatevná"}
```

```
low_priority = set(G.nodes) - high_priority - {"Dudvážska"}

# Funkcia na vyhľadanie trasy s ohľadom na prioritu
def priority_tsp(graph, start, high_priority, low_priority):
    route = [start]
    total_time = 0
    unvisited_high = set(high_priority)
    unvisited_low = set(low_priority)
    current = start

    # Najprv prejdeme cez body s vyššou prioritou
    while unvisited_high:
        next_node = min(unvisited_high, key=lambda node: graph[current][node]['weight'])
        total_time += graph[current][next_node]['weight']
        route.append(next_node)
        unvisited_high.remove(next_node)
        current = next_node

    # Potom prejdeme cez body s nižšou prioritou
    while unvisited_low:
        next_node = min(unvisited_low, key=lambda node: graph[current][node]['weight'])
        total_time += graph[current][next_node]['weight']
        route.append(next_node)
```

```
            unvisited_low.remove(next_node)
            current = next_node

        # Vrátime sa do východiskového bodu
        total_time += graph[current][start]['weight']
        route.append(start)

        return route, total_time

    # Nájdeme trasu s ohľadom na prioritu
    best_priority_route, min_priority_time = priority_tsp(G, "Dudvážska", high_priority, low_priority)

    # Výstup výsledkov
    best_priority_route, min_priority_time
```

Príloha č. 9

```
    # Preimportujeme potrebné knižnice po reštarte prostredia
    import networkx as nx

    # Obnovíme slovník so vzdialenosťami
    cas_Tue_9_am = {
        "Dudvážska": {"Wolkrova": 13, "Šintavská": 28.34, "Starohájska": 23.03, "Obchodná": 26.57, "Žatevná": 37.20,
```

"Dvořákovo nábrežie": 26.57, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},

"Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 10.63, "Žatevná": 24.80, "Dvořákovo nábrežie": 10.63,

"Parková": 15.94, "Kubačova": 35.43, "Karadžičova": 12.40, "Pasienková": 17.71},

"Šintavská": {"Starohájska": 6, "Obchodná": 15.94, "Žatevná": 26.57, "Dvořákovo nábrežie": 17.71,

"Parková": 23.03, "Kubačova": 40.74, "Karadžičova": 19.49, "Pasienková": 24.80},

"Starohájska": {"Obchodná": 15.94, "Žatevná": 28.34, "Dvořákovo nábrežie": 15.94, "Parková": 17.71,

"Kubačova": 37.20, "Karadžičova": 14.17, "Pasienková": 19.49},

"Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15, "Karadžičova": 4, "Pasienková": 13},

"Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 30.11, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 31.88},

"Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 23.03},

"Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},

"Kubačova": {"Karadžičova": 16, "Pasienková": 16},

"Karadžičova": {"Pasienková": 11}

}

Vytvoríme graf

G = nx.Graph()

```
# Pridáme uzly
for location in cas_Tue_9_am.keys():
    G.add_node(location)

# Pridáme hrany (čas medzi bodmi)
for start, destinations in cas_Tue_9_am.items():
    for end, distance in destinations.items():
        G.add_edge(start, end, weight=distance)

# Určíme body s vyššou prioritou a všetky ostatné
high_priority = {"Šintavská", "Wolkrova", "Kubačova", "Žatevná"}
low_priority = set(G.nodes) - high_priority - {"Dudvážska"}

# Funkcia na vyhľadanie trasy s ohľadom na prioritu
def priority_tsp(graph, start, high_priority, low_priority):
    route = [start]
    total_time = 0
    unvisited_high = set(high_priority)
    unvisited_low = set(low_priority)
    current = start

    # Najprv prejdeme cez body s vyššou prioritou
    while unvisitedc:
        next_node = min(unvisited_high, key=lambda node:
graph[current][node]['weight'])
```

```
            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited_high.remove(next_node)
            current = next_node

        # Potom prejdeme cez body s nižšou prioritou
        while unvisited_low:
            next_node = min(unvisited_low, key=lambda node: graph[current][node]['weight'])
            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited_low.remove(next_node)
            current = next_node

        # Vrátime sa do východiskového bodu
        total_time += graph[current][start]['weight']
        route.append(start)

        return route, total_time

    # Nájdeme trasu s ohľadom na prioritu
    best_priority_route, min_priority_time = priority_tsp(G, "Dudvážska", high_priority, low_priority)

    # Výstup výsledkov
```

best_priority_route, min_priority_time

Príloha č. 10

```
# Importujeme knižnicu NetworkX
import networkx as nx


# Vytvoríme graf zo zadaných údajov
cisty_cas = {
    "Dudvážska": {"Wolkrova": 13, "Šintavská": 16, "Starohájska": 13, "Obchodná": 15, "Žatevná": 21,
                  "Dvořákovo nábrežie": 15, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},
    "Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 6, "Žatevná": 14, "Dvořákovo nábrežie": 6,
                 "Parková": 9, "Kubačova": 20, "Karadžičova": 7, "Pasienková": 10},
    "Šintavská": {"Starohájska": 6, "Obchodná": 9, "Žatevná": 15, "Dvořákovo nábrežie": 10,
                  "Parková": 13, "Kubačova": 23, "Karadžičova": 11, "Pasienková": 14},
    "Starohájska": {"Obchodná": 9, "Žatevná": 16, "Dvořákovo nábrežie": 9, "Parková": 10,
                    "Kubačova": 21, "Karadžičova": 8, "Pasienková": 11},
    "Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15,
                 "Karadžičova": 4, "Pasienková": 13},
    "Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 17, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 18},
```

```
        "Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 13},
        "Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},
        "Kubačova": {"Karadžičova": 16, "Pasienková": 16},
        "Karadžičova": {"Pasienková": 11}
    }

    # Vytvoríme graf
    G = nx.Graph()
    for start, destinations in cisty_cas.items():
        for end, time in destinations.items():
            G.add_edge(start, end, weight=time)

    # Algoritmus trasy bez priorít, ale s časovým prahom
    def tsp_with_time_threshold(graph, start, time_threshold=7):
        route = [start]
        total_time = 0
        unvisited = set(graph.nodes) - {start}
        current = start

        while unvisited:
            nearby = [node for node in unvisited if node in graph[current] and graph[current][node]['weight'] <= time_threshold]
            if nearby:
                next_node = min(nearby, key=lambda node: graph[current][node]['weight'])
```

```
            else:
                next_node = min(unvisited, key=lambda node: graph[current][node]['weight'])

            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited.remove(next_node)
            current = next_node

        if start in graph[current]:
            total_time += graph[current][start]['weight']
            route.append(start)

        return route, total_time

    # Spustenie algoritmu
    optimal_route, total_time = tsp_with_time_threshold(G, "Dudvážska", time_threshold=7)
    optimal_route, total_time
```

Príloha č. 11

```
# Preimportujeme potrebné knižnice po reštarte prostredia
import networkx as nx

# Obnovíme slovník so vzdialenosťami
```

```
cas_Tue_7_am = {
    "Dudvážska": {"Wolkrova": 13, "Šintavská": 31.16, "Starohájska": 25.32, "Obchodná": 29.21, "Žatevná": 40.89,
                  "Dvořákovo nábrežie": 29.21, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},
    "Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 11.68, "Žatevná": 27.26, "Dvořákovo nábrežie": 11.68,
                 "Parková": 17.53, "Kubačova": 38.95, "Karadžičova": 13.63, "Pasienková": 19.47},
    "Šintavská": {"Starohájska": 6, "Obchodná": 17.53, "Žatevná": 29.21, "Dvořákovo nábrežie": 19.47,
                  "Parková": 25.32, "Kubačova": 44.79, "Karadžičova": 21.42, "Pasienková": 27.26},
    "Starohájska": {"Obchodná": 17.53, "Žatevná": 31.16, "Dvořákovo nábrežie": 17.53, "Parková": 19.47,
                    "Kubačova": 40.89, "Karadžičova": 15.58, "Pasienková": 21.42},
    "Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15, "Karadžičova": 4, "Pasienková": 13},
    "Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 33.11, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 35.05},
    "Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 25.32},
    "Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},
    "Kubačova": {"Karadžičova": 16, "Pasienková": 16},
    "Karadžičova": {"Pasienková": 11}
}
```

```
# Vytvoríme graf
G = nx.Graph()
for start, destinations in cas_Tue_7_am.items():
    for end, time in destinations.items():
        G.add_edge(start, end, weight=time)

# Algoritmus trasy bez priorít, ale s časovým prahom
def tsp_with_time_threshold(graph, start, time_threshold=7):
    route = [start]
    total_time = 0
    unvisited = set(graph.nodes) - {start}
    current = start

    while unvisited:
        nearby = [node for node in unvisited if node in graph[current] and graph[current][node]['weight'] <= time_threshold]
        if nearby:
            next_node = min(nearby, key=lambda node: graph[current][node]['weight'])
        else:
            next_node = min(unvisited, key=lambda node: graph[current][node]['weight'])

        total_time += graph[current][next_node]['weight']
        route.append(next_node)
        unvisited.remove(next_node)
```

```
            current = next_node

        if start in graph[current]:
            total_time += graph[current][start]['weight']
            route.append(start)

        return route, total_time

    # Spustenie algoritmu
    optimal_route, total_time = tsp_with_time_threshold(G, "Dudvážska", time_threshold=7)
    optimal_route, total_time
```

Príloha č. 12

```
    # Preimportujeme potrebné knižnice po reštarte prostredia
    import networkx as nx

    # Obnovíme slovník s časom
    cas_Tue_8_am = {
        "Dudvážska": {"Wolkrova": 13, "Šintavská": 35.39, "Starohájska": 28.76, "Obchodná": 33.18, "Žatevná": 46.46,
                      "Dvořákovo nábrežie": 33.18, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},
        "Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 13.27, "Žatevná": 30.97, "Dvořákovo nábrežie": 13.27,
```

"Parková": 19.91, "Kubačova": 44.24, "Karadžičova": 15.49, "Pasienková": 22.12},

"Šintavská": {"Starohájska": 6, "Obchodná": 19.91, "Žatevná": 33.18, "Dvořákovo nábrežie": 22.12,

"Parková": 28.76, "Kubačova": 50.88, "Karadžičova": 24.33, "Pasienková": 30.97},

"Starohájska": {"Obchodná": 19.91, "Žatevná": 35.39, "Dvořákovo nábrežie": 19.91, "Parková": 22.12,

"Kubačova": 46.46, "Karadžičova": 17.70, "Pasienková": 24.33},

"Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15, "Karadžičova": 4, "Pasienková": 13},

"Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 37.61, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 39.82},

"Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 28.76},

"Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},

"Kubačova": {"Karadžičova": 16, "Pasienková": 16},

"Karadžičova": {"Pasienková": 11}

}

```
# Vytvoríme graf
G = nx.Graph()
for start, destinations in cas_Tue_8_am.items():
    for end, time in destinations.items():
        G.add_edge(start, end, weight=time)
```

```
# Algoritmus trasy bez priorít, ale s časovým prahom
def tsp_with_time_threshold(graph, start, time_threshold=7):
    route = [start]
    total_time = 0
    unvisited = set(graph.nodes) - {start}
    current = start

    while unvisited:
        nearby = [node for node in unvisited if node in graph[current] and graph[current][node]['weight'] <= time_threshold]
        if nearby:
            next_node = min(nearby, key=lambda node: graph[current][node]['weight'])
        else:
            next_node = min(unvisited, key=lambda node: graph[current][node]['weight'])

        total_time += graph[current][next_node]['weight']
        route.append(next_node)
        unvisited.remove(next_node)
        current = next_node

    if start in graph[current]:
        total_time += graph[current][start]['weight']
        route.append(start)
```

```
    return route, total_time


# Spustenie algoritmu
optimal_route, total_time = tsp_with_time_threshold(G, "Dudvážska", time_threshold=7)
optimal_route, total_time
```

Príloha č. 13

```
# Preimportujeme potrebné knižnice po reštarte prostredia
import networkx as nx



# Obnovíme slovník so vzdialenosťami
cas_Tue_9_am = {
    "Dudvážska": {"Wolkrova": 13, "Šintavská": 28.34, "Starohájska": 23.03, "Obchodná": 26.57, "Žatevná": 37.20,
                  "Dvořákovo nábrežie": 26.57, "Parková": 9, "Kubačova": 19, "Karadžičova": 12, "Pasienková": 4},
    "Wolkrova": {"Šintavská": 5, "Starohájska": 4, "Obchodná": 10.63, "Žatevná": 24.80, "Dvořákovo nábrežie": 10.63,
                 "Parková": 15.94, "Kubačova": 35.43, "Karadžičova": 12.40, "Pasienková": 17.71},
    "Šintavská": {"Starohájska": 6, "Obchodná": 15.94, "Žatevná": 26.57, "Dvořákovo nábrežie": 17.71,
                  "Parková": 23.03, "Kubačova": 40.74, "Karadžičova": 19.49, "Pasienková": 24.80},
```

```
    "Starohájska": {"Obchodná": 15.94, "Žatevná": 28.34, "Dvořákovo nábrežie": 15.94, "Parková": 17.71,
                    "Kubačova": 37.20, "Karadžičova": 14.17, "Pasienková": 19.49},
    "Obchodná": {"Žatevná": 15, "Dvořákovo nábrežie": 7, "Parková": 9, "Kubačova": 15, "Karadžičova": 4, "Pasienková": 13},
    "Žatevná": {"Dvořákovo nábrežie": 11, "Parková": 30.11, "Kubačova": 26, "Karadžičova": 16, "Pasienková": 31.88},
    "Dvořákovo nábrežie": {"Parková": 11, "Kubačova": 19, "Karadžičova": 6, "Pasienková": 23.03},
    "Parková": {"Kubačova": 18, "Karadžičova": 7, "Pasienková": 7},
    "Kubačova": {"Karadžičova": 16, "Pasienková": 16},
    "Karadžičova": {"Pasienková": 11}
}

# Vytvoríme graf
G = nx.Graph()
for start, destinations in cas_Tue_9_am.items():
    for end, time in destinations.items():
        G.add_edge(start, end, weight=time)

# Algoritmus trasy bez priorít, ale s časovým prahom
def tsp_with_time_threshold(graph, start, time_threshold=7):
    route = [start]
    total_time = 0
    unvisited = set(graph.nodes) - {start}
    current = start
```

```
        while unvisited:
            nearby = [node for node in unvisited if node in graph[current] and graph[current][node]['weight'] <= time_threshold]
            if nearby:
                next_node = min(nearby, key=lambda node: graph[current][node]['weight'])
            else:
                next_node = min(unvisited, key=lambda node: graph[current][node]['weight'])

            total_time += graph[current][next_node]['weight']
            route.append(next_node)
            unvisited.remove(next_node)
            current = next_node

        if start in graph[current]:
            total_time += graph[current][start]['weight']
            route.append(start)

        return route, total_time

    # Spustenie algoritmu
    optimal_route, total_time = tsp_with_time_threshold(G, "Dudvážska", time_threshold=7)
    optimal_route, total_time
```

Zoznam použitej literatúry

Vedecké články

HAGBERG, Aric – SWART, Pieter – CHULT, Daniel. Exploring Network Structure, Dynamics, and Function Using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy 2008)*. Los Alamos National Laboratory, June 2008. Dostupné na: https://doi.org/10.25080/TCWV9851

JOVANOVIC, Aleksandar – UZELAC, Ana – KUKIC, Katarina – TEODOROVIC, Dušan. The shortest-path and bee colonyoptimization algorithms for trafficcontrolatsingle intersection with NetworkX. In *Demonstratio Mathematica*, February 2024. Dostupné na: https://doi.org/10.1515/dema-2023-0160

NAIR, Shilpa – ABHIRAM, Manoj – KUZHIYAMKUNNATH, Bhavathrathan. Betweenness Centrality Measures as Potential Predictors of Crash Frequency. In *Transportation in Developing Economies*, Indian Institute of Technology Bombay, April 2024, vol. 10. Dostupné na: https://doi.org/10.1007/s40890-024-00205-1

FENG, Yelai – WANG, Huaixi – CHANG, Chao – LU, Hongyi. Intrinsic Correlation with Betweenness Centrality and Distribution of Shortest Paths. In *MDPI Mathematics*, July 2022, vol. 10, no. 14, art. 2521. Dostuplné na: https://doi.org/10.3390/math10142521

YUAN, Feier. Data Analysis and Optimization Strategies for Delivery Operations in a Shenyang-Based Convenience Store Network. In *SHS Web of Conferences*, December 2024, vol. 208. Dostupné na: https://doi.org/10.1051/shsconf/202420804013

DUMKA, Pankaj – CHAUHAN, Rishika – MISHRA, Dhananjay R. Travelling Salesman Problem: A Python Implementation. Jaypee University of Engineering and Technology, March 2025. Dostupné na:

https://www.researchgate.net/publication/390092650_Travelling_Salesman_Problem_A_Python_Implementation

JACKOVICH, Petar – COX, Bruce – HILL, Raymond R. Comparing greedy constructive heuristic subtour elimination methods for the traveling salesman problem. In *Journal of Defense Analytics and Logistics*, November 2020, vol. 4, no. 2, Dostupné na: https://doi.org/10.1108/JDAL-09-2020-0018

SCHWARZ, Sebastian – UERLICH, Sebastian Alexander – MONTI, Antonello. pycity_scheduling—A Python framework for the development and assessment of optimisation-based power scheduling algorithms for multi-energy systems in city districts. Institute for Automation of Complex Power Systems, E.ON Energy Research Center, RWTH Aachen University, December 2021. Dostupné na: https://www.researchgate.net/publication/355967423_pycity_scheduling-A_Python_framework_for_the_development_and_assessment_of_optimisation-based_power_scheduling_algorithms_for_multi-energy_systems_in_city_districts

MIAO, Wenkang – ZHAO, Xingdong. Traffic Congestion Scheduling for Underground Mine Ramps Based on an Improved Genetic Scheduling Algorithm. In *Applied Sciences*, October 2024, vol. 14, no. 21, art. 9862. Dostupné na: https://doi.org/10.3390/app14219862

KATOCH, Sourabh – CHAUHAN, Sumit Singh – KUMAR, Vijay. A review on genetic algorithm: past, present, and future. In *Multimedia Tools and Applications*, vol. 80, February 2021. Dostupné na: https://doi.org/10.1007/s11042-020-10139-6

GOHL, Jesse – TUMMESCHEIT, Hubertus – ANDERSSON, Robin – STUBER, Matthew. Steady-state Optimization of Modelica Models and Functional Mockup Units with Pyomo. In *American Modelica Conference 2024*. Modeon Inc., University of Connecticut, January 2025. Dostupné na: https://doi.org/10.3384/ecp207109

JAUS, Christopher – HAYNIE, Kaelyn – MULLIGAN, Michael – FANG, Howie. Practice and Research Optimization Environment in Python (PyPROE). In *MDPI Computers*. Liberty University, Department of Mechanical Engineering and Department of Computer Science, February 2025. Dostupné na: https://doi.org/10.3390/computers14020054

MOUSSA, Ahmad A. – ALI, Amman A. – EL SAYEH, Mohi Eldeen – SHEHATA, Ahmad S. A novel dynamic route optimization method and its implementation using Python to optimize ship voyages sailing time based on weather routing techniques. Arab Academy for Science Technology and Maritime Transport, December 2024. Dostupné na: https://doi.org/10.21203/rs.3.rs-5707487/v1

GARCÍA, Andrea. Greedy algorithms: a review and open problems. In *Journal of Inequalities and Applications*, February 2025, vol. 2025, no. 1. Dostupné na: https://doi.org/10.1186/s13660-025-03254-1

KORDIĆ, Stevan – KOVAČ, Nataša – DAVIDOVIĆ, Tatjana. Divide and Conquer Approach to Discrete Berth Allocation Problem. In *Scientific Bulletin of Naval Academy*, Constanta, Romania, January 2024, vol. XVIII, no. 2. Dostupné na:https://www.researchgate.net/publication/377087071_Divide_and_Conquer_Approach_to_Discrete_Berth_Allocation_Problem

REN, Fangtao. Economic Analysis of Logistics Network in the Context of Supply Chain Management. In *International Research in Economics and Finance* 6(1):7, City University of Macau, Macau, China, April 2022. Dostupné na: https://doi.org/10.20849/iref.v6i1.1112

Internetové zdroje

Varun Tyagi. (2024, February 9). *Simplifying Logistics with Python: A Practical Guide to Bundling Moves*. https://medium.com/operations-research-bit/simplifying-logistics-with-python-a-practical-guide-to-bundling-moves-2f0d937dfb19

NwtworkX. (2015, October 27). *NetworkX documentation*. https://networkx.org/documentation/networkx-1.10/index.html

Matplotlib.org. (2022, August 10). *Matplotlib 3.10.1 documentation*. https://matplotlib.org/stable/

Solver Max Optimal Solutions. (2025). *Optimization modelling in Python.* https://www.solvermax.com/resources/links/optimization-modelling-in-python

PuLP. (2025). *Optimization with PuLP Documentation.* https://coin-or.github.io/pulp/index.html#optimization-with-pulp

OptaPlanner. (2023, September 6). *OptaPlanner User Guide.* https://docs.optaplanner.org/latest/optaplanner-docs/html_single/index.html

SimPy. (2025). Time and Scheduling with SimPy https://simpy.readthedocs.io/en/latest/topical_guides/time_and_scheduling.html

Advancedor Academy. (2024). *Production Scheduling with Genetic Algorithms* https://advancedoracademy.medium.com/production-scheduling-with-genetic-algorithms-74f7ed08e10e

DEAP. (2024). *DEAP documentation*. https://deap.readthedocs.io/en/master/index.html

PyQueantNews. (2024, June 13). *Mastering Python for Finance: NumPy, pandas, SciPy.* https://www.pyquantnews.com/free-python-resources/mastering-python-for-finance-numpy-pandas-scipy?utm_source=chatgpt.com

NumPy.org. *NumPy fundamentals*. (2024). https://numpy.org/doc/stable/user/basics.html

Scipy.org. (2025, February 17). *SciPy documentation. Version: 1.15.2*. https://docs.scipy.org/doc/scipy/

Pypi.org. (2021, April 7). *Ipopt 1.0.3*. https://pypi.org/project/ipopt/#description

Pypi.org. (2018, December 16). *PyFMI 2.5 Documentation*. https://pypi.org/project/PyFMI/

Modelon. (2025). *What is FMI?* https://modelon.com/blog/functional-mock-up-interface-fmi/

MATLAB Help Center. (2024). *Include Functional Mockup Unit (FMU) in model*. https://www.mathworks.com/help/simulink/ref_extras/fmu.html

RealPython. (2025, February 2). *Develop Data Visualization Interfaces in Python With Dash*. https://realpython.com/python-dash/

PythonGUIs. (2025, March 10). *PyQt vs. Tkinter — Which Should You Choose for Your Next GUI Project?* https://www.pythonguis.com/faq/pyqt-vs-tkinter/

Umesh S. (2023, February 11). *Python libraries for scientific computing (e.g., NumPy, SciPy, Matplotlib, Pandas, Seaborn).* https://umeshsl.medium.com/python-libraries-for-scientific-computing-e-g-numpy-scipy-matplotlib-pandas-seaborn-ab064e73d3bc

Google Maps Platform. *Routes API*. (2025, February 5)
https://developers.google.com/maps/documentation/routes

Mapy.cz. *Addresses in Bratislava.* (2025)
https://mapy.com/s/cejakemepe

Tomtom.com. *Bratislava live traffic*. (2025)
https://www.tomtom.com/traffic-index/bratislava-traffic/

European Commission. *CO_2 emission performance standards for cars and vans* (2023, April 19)
https://climate.ec.europa.eu/eu-action/transport/road-transport-reducing-co2-emissions-vehicles/co2-emission-performance-standards-cars-and-vans_en