Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky
Katedra informačních technologií

Studijní program: Aplikovaná informatika
Obor: Kognitivní informatika

Implementace heuristik pro rozvozní problém s časovými okny

DIPLOMOVÁ PRÁCE

Student : Mgr. Otakar Trunda
Vedoucí : prof. RNDr. Jan Pelikán, CSc.
Oponent : Mgr. Vladimír Holý

2017

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 24. dubna 2017 . . . . . .

Otakar Trunda

Poděkování

Rád bych na tomto místě poděkoval profesoru Pelikánovi za vedení práce a za cenné rady a připomínky ohledně obsahu.

Také bych chtěl poděkovat profesoru Řepovi, který mi pomohl překonat jisté administrativní překážky při zadávání tématu práce a zasloužil se tak o to, že tato práce vůbec vznikla.

Abstrakt

Rozvozní problém s časovými okny patří mezi těžké optimalizační problémy. Přestože má tento typ problémů mnoho praktických aplikací, otázka jeho efektivního řešení stále není uspokojivě objasněná. Tato práce se zabývá studiem *Rozvozních problémů* s *časovými okny a* návrhem nových algoritmů pro jejich řešení.

Jsou zde představené dvě heuristiky a několik navazujících algoritmů, které tyto heuristiky dále rozšiřují. Efektivita navržených postupů je experimentálně ověřena na sadě testovacích dat.

Součástí práce je také vytvoření desktopové aplikace, která implementuje navržené algoritmy a poskytuje další funkcionalitu pro usnadnění řešení rozvozních problémů v praxi. Patří mezi ně například generátor pseudo-náhodných zadání problému, vizualizace řešení a podobně.

Klíčová slova

Rozvozní problém s časovými okny; Heuristiky; Dopravní problémy; Kombinatorická optimalizace.

Abstract

Vehicle Routing Problem with Time Windows is a hard optimization problem. Even though it has numerous practical applications, the question of solving it efficiently has not been satisfyingly solved yet. This thesis studies the *Vehicle Routing Problem with Time Windows* and presents several new algorithms for solving it.

There are two heuristics presented here, as well as several more complex algorithms which use those heuristics as their components. The efficiency of presented techniques is evaluated experimentally using a set of test samples.

As a part of this thesis, I have also developed a desktop application which implements presented algorithms and provides a few additional features useful for solving routing problems in practice. Among others, there is a generator of pseudo-random problem instances and several visualization methods.

Keywords

Vehicle Routing Problem with Time Windows; Heuristics; Transportation Problems; Combinatorial Optimization.

Obsah

1 Úvod

Tato práce se zabývá řešením dopravních problémů, konkrétně *Rozvozního problému s časovými okny*. Rozvozní problémy se často vyskytují v praxi a jejich efektivní řešení je tedy velmi žádoucí. Vysoká výpočetní náročnost úlohy však neumožňuje hledat optimální řešení, proto se u úloh většího rozsahu používají pouze přibližné postupy, zvané heuristiky.

Řešení rozvozního problému spočívá v nalezení takových tras pro rozvoz, aby byly uspokojené všechny požadavky odběratelů a celková délka tras byla co nejmenší.

Tato práce studuje rozvozní problémy a možnosti jejich řešení a představuje několik heuristik, které jsou pro jejich řešení v praxi použitelné.

1.1 Motivace

Problém rozvozu nebo svozu se přirozeně vyskytuje v mnoha oborech podnikání, nejen v oblasti logistiky. Samozřejmostí je řešení těchto typů problémů u provozovatelů logistických služeb, jako jsou kurýři, doručovatelé a přepravci všeho druhu. Rozvoz však musejí optimalizovat také například banky při naplňování bankomatů, internetové supermarkety při dopravě nakoupeného zboží k zákazníkům, nebo malá pekárna, která každé ráno rozváží pečivo k odběratelům.

Optimalizace tras při rozvozu může mít značný ekonomický dopad na úspěšné fungování firmy. Zkrácení délky tras vede nejen k úspoře paliva, ale snižuje se tím i spotřeba pracovního času řidičů. Navíc při lepším plánování rozvozních operací si může společnost vystačit i s menším množstvím vozidel. U velkých rozvozních společností typu DHL může zkrácení délky tras o pouhou desetinu procenta znamenat finanční úspory v řádu tisíců dolarů denně.

Pro řešení klasického rozvozního problému existuje mnoho technik, v praxi často poměrně úspěšných. Klasický rozvozní problém je však značně zjednodušený, a nedá se přímo použít na většinu reálných situací, například zanedbává existenci tzv. časových oken u zákazníků. Časová okna reprezentují otevírací dobu, tedy časový interval, ve kterém je jedině možné příslušného zákazníka navštívit.

Pro řešení *rozvozního problému s časovými okny* zatím neexistuje žádný algoritmus, který by byl všeobecně uznávaný jako nejlepší, a je zde tedy stále prostor pro zlepšování. Navíc *rozvozní problém s časovými okny* je blízký reálným aplikací a jeho efektivní řešení je tedy v praxi žádoucí.

1.2 Cíle práce

Cílem této práce je prozkoumat *Rozvozní problém s časovými okny* a navrhnout novou heuristiku pro jeho řešení. Dalším cílem je výslednou heuristiku, nebo heuristiky implementovat ve formě desktopové aplikace, která bude uživatelsky přívětivá a použitelná pro řešení rozvozních problémů v praxi. Navržená aplikace by měla mimo jiné podporovat vizualizaci nalezených tras a přehledně zobrazovat časové plány jízd jednotlivých vozidel.

1.3 Obsah práce

Kapitoly 2 a 3 představují úvod do problematiky a komentovanou rešerši aktuálně používaných postupů. Kapitoly 4 a 5 pak popisují hlavní přínos této práce: návrh několika nových heuristik pro *rozvozní problém s časovými okny*. Zde je podrobně popsané fungování navržených heuristik včetně popisu reprezentace dat a práce s nimi.

Další, tedy šestá kapitola obsahuje popis navržené aplikace, její funkcionalitu a ovládání. Sedmá kapitola popisuje výsledky provedených experimentů, a práce je poté ukončena závěrem.

2 Dopravní problémy a jejich řešení

Kapitola formálně definuje rozvozní problém a některé související problémy a představuje možnosti, jak lze tyto problémy řešit. Jsou zmíněná také úskalí plynoucí z vysoké výpočetní náročnost problému a některé způsoby, jak je lze překonat.

2.1 Praktické aplikace dopravních problémů

Dopravní problémy se přirozeně vyskytují téměř ve všech oblastech podnikání. Je zřejmé, že je musí řešit dopravci, a společnosti, které se zaměřují na dopravu, jako například PPL, Česká pošta, Uloženka, kurýrní služby, společnosti provozující kamionovou dopravu a podobně.

Dopravu však musejí řešit i společnosti, které se logistikou přímo nezabývají, jako například supermarkety, banky, továrny, těžařské firmy a podobně. Všude tam, kde podnik vyžaduje časté a velkoobjemové dodávky zásob od dodavatelů, a/nebo kde potřebuje pravidelně přepravovat své produkty k odběratelům, se dopravní problémy vyskytují.

V této práci se věnuji úlohám o rozvozu a svozu. Jako příklad těchto typů problémů lze uvést:

- Pekařství rozvážející každé ráno své výrobky do prodejen
- Pizzerii nabízející rozvoz jídla k zákazníkům
- Rozvoz nákupu z internetového supermarketu k zákazníkům
- Svoz mléka z jednotlivých mléčných farem do mlékárny
- Svoz odpadu
- Roznášení pošty
- Rozvoz bankovek při naplňování bankomatů nebo obecně naplňování jakýchkoli automatů, například na kávu, na sladkosti a podobně
- a mnoho dalších

Při řešení rozvozního problému je hlavním úkolem stanovit trasy, po kterých jednotlivá vozidla pojedou tak, aby byli všichni zákazníci obslouženi a vzdálenost, kterou vozidla v součtu ujedou, byla co nejmenší.

Rozvozní problémy lze rozdělit na 2 typy: *jednorázové* a *opakované*. Opakovaný rozvoz se vyznačuje tím, že obsluhu je třeba provádět opakovaně v určitých časových intervalech, přičemž odběratelská místa jsou vždy stejná. Ze zmíněných příkladů lze do této kategorie zařadit situaci s pekařem a také svoz odpadu. V případě jednorázového problému se nao-

pak předpokládá, že stejný rozvoz (tedy se stejnými odběrateli a totožnými objemy zásilek) se nebude pravidelně opakovat. Do této kategorie spadá příklad s pizzerií a lze sem zařadit i rozvoz nákupů z online supermarketů.

Z praktického hlediska je rozdíl v těchto typech problémů zejména v tom, že opakované rozvozní problémy stačí vyřešit jednou, a poté už získané řešení jen opakovaně aplikovat. Je tedy možné procesu řešení věnovat poměrně dlouhou dobu a jakékoli zlepšení kvality řešení (tedy úspora počtu najetých kilometrů) je velmi významné, protože v delším časovém horizontu povede ke značným úsporám. Lze si představit, že plánování cest může v tomto případě trvat několik hodin i dní, obzvlášť pokud se jedná o rozvoz ve velkém objemu (například přeprava ropy pomocí tankerů a podobně).

Naopak u problémů jednorázových je často třeba uskutečnit obsluhu co nejdříve, takže na dlouhé plánování cest není čas a ani případné úspory zde nehrají až tak velkou roli, protože rozvoz se uskuteční jen jednou. Záleží samozřejmě také na velikosti rozvozní úlohy, tj. počtu vozidel a zákazníků, objemu rozváženého zboží a na dalších faktorech.

V praxi se pro řešení otázek spojených s rozvozem používá několik strategií:

1. Využití služeb externí dopravní společnosti
2. Řešení rozvozu vlastními silami – manuálně
3. Řešení rozvozu vlastními silami – pomocí dostupných softwarových nástrojů
4. Řešení pomocí softwarového nástroje vytvořeného na míru

Společnosti, které dopravu řešit nechtějí, mohou využít externích služeb. To je příklad většiny e-shopů a producentů, kteří nabízejí své zboží na internetu. Výhodou tohoto postupu je, že přeprava bude provedena efektivně a odesilatel se jí nemusí zabývat. Nevýhodou je naopak vyšší cena za službu.

Manuální řešení se používají zejména při maloobjemových rozvozech, kdy drobný producent rozváží své výrobky buď přímo ke koncovým zákazníkům, nebo do prodejen a to typicky pomocí jediného vozidla. Výhodou je jednoduchost, nevýhodou pak případná neefektivita, která však je při malém objemu zanedbatelná.

Řešení s využitím dostupných softwarových nástrojů je výhodné pro střední a větší firmy, které disponují flotilou vozidel a kde velikost rozvozního problému už neumožňuje efektivní manuální řešení. Navíc oproti využití externích služeb je tato varianta finančně výhodná.

Vytvoření SW nástroje na míru může být vhodné v případě, kdy je třeba řešit nějakou velmi specifickou variantu rozvozního problému nebo pokud se jedná o velkoobjemový opakovaný rozvoz. Tento způsob také preferují společnosti, které se dopravou přímo zabývají, například PPL a podobně.

2.2 Formulace dopravních problémů

Dopravní, nebo také logistické problémy se obecně zabývají plánováním dopravy a optimalizací nákladů při přepravě zboží. Existuje mnoho variant dopravních problémů, z nichž některé v této kapitole popíšu. Zaměřím se na *Rozvozní problém s časovými okny (VRPTW)*, jehož řešení je náplní této práce, ale popíšu i několik jednodušších problémů, ze kterých VRPTW vychází, stručně zmíním také několik zobecnění.

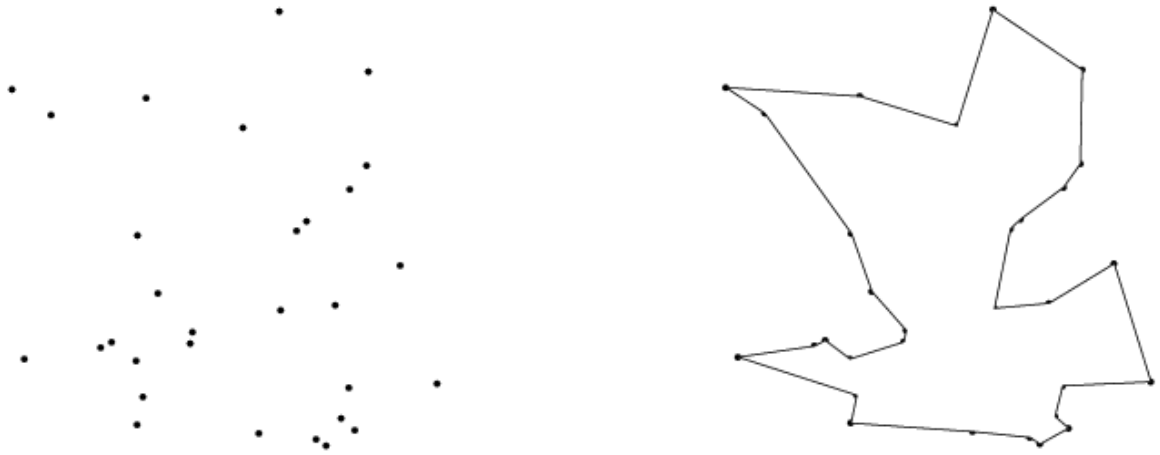
2.2.1 Problém obchodního cestujícího

Problém obchodního cestujícího (*Travelling Salesman Problem – TSP*), v češtině nazývaný také *Okružní dopravní problém*, je jeden z nejznámějších logistických problémů. (Pelikán, 2001, s. 34)

Praktická motivace tohoto problému je následující: Obchodník potřebuje navštívit zákazníky v několika různých městech a poté se vrátit zpátky na svou výchozí pozici. Zná vzdálenost mezi každými dvěma městy. Otázka zní: v jakém pořadí musí města projet, aby celková vzdálenost, kterou ujede, byla co nejmenší?

V řeči teorie grafů lze problém formulovat takto: Je dán hranově ohodnocený úplný orientovaný graf. Najděte v něm Hamiltonovskou kružnici s co nejmenším ohodnocením. V dalším textu budu terminologii z teorie grafů místy využívat. Města budu občas nazývat jako *vrcholy* a spojnice mezi městy jako *hrany*.

Obrázek 1 převzatý z (Wolfram MathWorld, ©1999) ukazuje příklad zadání a řešení.



Obrázek 1. Příklad zadání a řešení problému TSP. Vlevo zadání pomocí polohy měst, vpravo optimální řešení. Převzato z (Wolfram MathWorld, ©1999).

Formální zadání problému:

Je dáno:

- Přirozené číslo *n*, což je celkový počet měst
- Matice D o rozměrech $n \times n$, kde na pozici $D[i, j]$ je nezáporné reálné číslo vyjadřující délku cesty z města i do města j. Případně zde může být čas potřebný na přejezd nebo náklady spojené s přesunem, podle toho, které kritérium je třeba minimalizovat.

Výstupem pak je permutace P, čili uspořádání čísel 1 až *n* takové, že $D[P(1), P(2)] + D[P(2), P(3)] + ... + D[P(n-1), P(n)] + D[P(n), P(1)]$ je co nejmenší. Výrazem $P(i)$ označujeme číslo města, které je ve výsledném uspořádání na pozici *i*.

Občas je u praktických úloh zadání popsané v mírně odlišné podobě. Namísto matice D může být zadaná silniční mapa, která udává, jak jsou města propojená a také vzdálenosti mezi nejbližšími městy. Vzdálenost mezi každou dvojicí měst však lze v tomto případě snadno dopočítat pomocí algoritmu pro hledání nejkratších cest, proto budeme zadání vždy předpokládat ve formě uvedené výše.

V zadání není explicitně uvedené, ve kterém vrcholu má cestující svou jízdu začínat. Tato informace není nutná, protože neovlivňuje podobu výsledné cesty. Řešením je vždy uzavřený okruh a cestujícímu stačí tento okruh sledovat, čímž projede všechna města a vrátí se na počátek, ať už začínal kdekoliv.

Literatura déle rozlišuje několik speciálních případů TSP: (Korte et al., 2010, s 558 - 562)

Symetrické TSP

V této variantě je ohodnocení hran symetrické, čili z města a do města *B* se lze dostat stejně rychle jako z města *B* do *A*. Formálně: $\forall i, j \le n : D[i, j] = D[j, i]$. Potom stačí problém modelovat pomocí klasického, neorientovaného grafu.

V praxi bývá tento předpoklad často splněn, ale nemusí tomu tak být vždy. Symetrii může porušit například to, když se na mapě přejezdů nacházejí jednosměrné ulice.

TSP s metrikou

Jedná se o variantu problému, kde ohodnocení hran splňuje trojúhelníkovou nerovnost, tzn. platí $\forall i, j, k \le n : D[i, j] + D[j, k] \ge D[i, k]$. Tento požadavek, tedy že přímá cesta do cíle je vždy kratší než cesta s nějakým fixním průjezdním místem, je v praxi téměř vždy splněn. Při tvorbě matice vzdáleností nás totiž nezajímá, kudy vozidlo při přesunu pojede, ale pouze délka této trasy. Proto na místo $D[i, j]$ píšeme délku nejkratší cesty z i do j bez ohledu na to, jestli vozidlo projede po cestě nějakými dalšími městy nebo ne.

Předpoklad o trojúhelníkové nerovnosti se může při návrhu algoritmů hodit, proto tuto variantu explicitně zmiňuji. Navíc některé metody řešení poskytují u problému TSP s metrikou prokazatelně lepší výsledky než u obecné varianty.

Euklidovské TSP

V tomto případě odpovídají města bodům v rovině a vzdálenost se počítá euklidovsky, čili jako *vzdálenost vzdušnou čarou*. Na rozdíl od obecného problému, kde je známá jen matice vzdáleností mezi městy, zde máme pro každé město přímo zadanou i jeho přesnou polohu pomocí souřadnic.

Tato situace je v praxi také běžná a dodatečné informace lze s výhodou využít při řešení, jak je popsáno v dalších částech práce. Lze například ukázat, že optimální trasa mezi body v rovině se nikdy nekříží.

2.2.2 Problém obchodního cestujícího s časovými okny

Tento problém se označuje zkratkou *TSPTW* (*Travelling Salesman Problem with Time Windows)*. Jedná se o zobecnění předchozí varianty, kde je u každého města navíc zadané *časové okno,* tj. interval, ve kterém je třeba dané město navštívit.

V zadání je navíc uvedené:

- Pro každé město $i \leq n$ dvě čísla a_i, b_i taková, že $a_i < b_i$
- Pro každou dvojici měst $i, j \leq n$ doba potřebná pro přejezd z města i do města j

Pokud cestující dorazí do města i v čase menším než a_i, pak musí s návštěvou počkat až do času a_i. Pokud by do nějakého města i dorazil později než v čase b_i, pak se mu nepodařilo problém vyřešit.

Řešením je opět permutace, tedy pořadí, v jakém je třeba vrcholy navštívit tak, aby čas návštěvy každého vrcholu byl uvnitř příslušného časového okna a celková délka trasy byla co nejmenší.

Všechny varianty problému TSP popsané výše lze přímo aplikovat i na problém TSPTW. Existují tedy varianty TSPTW s metrikou, euklidovské TSPTW a podobně. Lze si také všimnout, že klasické TSP je speciálním případem TSPTW. Původní problém totiž dostaneme, pokud nastavíme $a_i = 0, b_i = \infty$.

TSPTW je pro modelování reálných aplikací často vhodnější než klasické TSP, protože časová okna se v praxi přirozeně vyskytují. Pokud například řešíme problematiku zásobování, kde je třeba rozvést nějaké množství výrobků do jednotlivých poboček, pak je téměř

jisté, že zásobovač nemůže pobočku navštívit v libovolný čas, ale pouze v době, kdy je na pobočce přítomen pracovník, který může dovezené zboží převzít.

2.2.3 Rozvozní problém

Rozvozní problém (*Vehicle Routing Problem – VRP*), občas nazývaný také *Úloha okružních jízd*, vychází z TSP a přidává navíc několik dalších podmínek. (Pelikán, 2001, s. 37) Prvním rozdílem je, že města jsou rozdělená do dvou kategorií: na *centrálu* a *zákazníky*. Centrála je počáteční a koncový bod trasy. Trasa tedy musí vždy začínat v centrále, navštívit některé zákazníky a poté opět skončit v centrále.

Druhým rozdílem oproti TSP je, že na centrále máme k dispozici větší množství vozidel. Není tedy třeba, aby všechny zákazníky obsloužilo jediné vozidlo, ale je možné zákazníky rozdělit do několika skupin s tím, že každou skupinu navštíví jiné vozidlo.

A konečně posledním rozdílem je zavedení *poptávky* u zákazníků a *kapacity* u vozidel. Při návštěvě zákazníků je třeba jim z centrály přivézt jisté množství zboží. Jaké množství zboží zákazník vyžaduje, je označené jako jeho poptávka a tato informace je předem známá. Vozidlo tedy v centrále naloží zboží a při obsluze jednotlivých zákazníků vždy část zboží vykládá. Množství zboží, které vozidlo uveze, je dané jeho kapacitou.

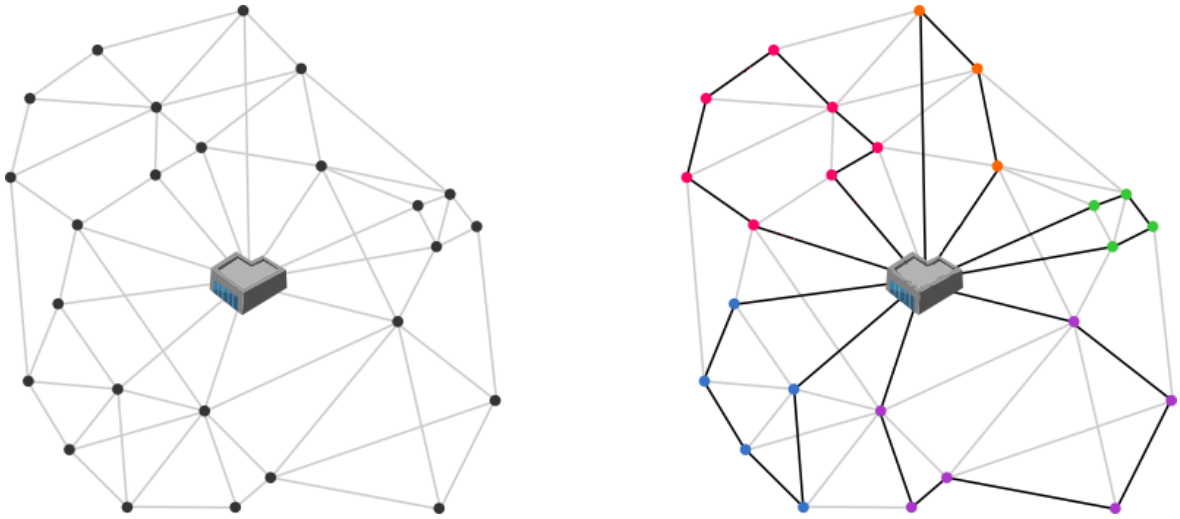
V zadání je oproti TSP navíc uvedené:

- Které město hraje roli centrály
- Kolik vozidel je v centrále k dispozici
- Jaké jsou kapacity jednotlivých vozidel
- Jaké jsou poptávky jednotlivých zákazníků
- Kolik zboží je v centrále k dispozici

Řešením je množina dvojic (v_i, P_i), kde v_i je vozidlo a P_i je permutace obsahující některé zákazníky. Tuto dvojici budu nazývat *okruh*. V řešení vyžadujeme, aby:

- každý zákazník se vyskytoval v právě jedné permutaci P_i
- v každém okruhu byl součet poptávky příslušných zákazníků menší nebo roven kapacitě vozidla, které okruh zajišťuje
- počet okruhů byl menší nebo roven počtu dostupných vozidel
- délka trasy, kterou všechna vozidla v součtu ujedou, byla co nejmenší

Obrázek 2 převzatý z (*NEO: Networking and Emerging Optimization, ©2013)* ukazuje příklad problému VRP. Vlevo je zadání, vpravo řešení. Depo se nachází uprostřed, zákazníci jsou znázornění pomocí černých teček a šedé čáry znázorňují silniční síť. Vpravo je vidět 5 tras, každá je obsloužená jedním vozidlem.



Obrázek 2. Příklad problému VRP. Vlevo zadání, vpravo řešení. Převzato z (NEO: Networking and Emerging Optimization, ©2013).

Na VRP lze opět aplikovat specifické varianty zmíněné u TSP (například lze definovat *symetrické VRP*, *euklidovské VRP* a podobně) a lze si opět všimnout, že TSP je speciálním případem VRP. Klasický problém TSP dostaneme, pokud nastavíme počet vozidel na 1, poptávku zákazníků na 1 a kapacitu vozidla rovnou počtu zákazníků.

VRP už umožňuje modelovat mnoho praktických problému typu rozvozu, zásobování i svozu. Problém svozu, kdy zákazníci něco produkují a jejich produkci je třeba dovézt do centrály, lze totiž jednoduše převést na problém rozvozu tak, že produkci zákazníků nahradíme jejich poptávkou. Rozdíl je pak pouze v tom, že vozidlo vyjede z centrály prázdné a přijede naplněné, ale jeho trasa je při svozu i rozvozu stejná a jsou splněné i kapacitní omezení. V dalším textu budu tedy psát pouze o problému rozvozu.

2.2.4 Rozvozní problém s časovými okny

Problém je v literatuře označovaný zkratkou *VRPTW* z anglického *Vehicle Routing Problem with Time Windows*. Jak už název napovídá, jedná se o kombinaci dvou předchozích modifikací, čili rozšíření VRP o časová okna na straně zákazníků.

Formální zadání problému:

Stejně jako u TSP máme k dispozici informace o počtu cílových míst a jejich vzdálenostech. Navíc je zadané:

- které místo hraje roli centrály / depa
- pro každé cílové místo je daná jeho *poptávka*, tj. množství jednotek zboží, které je třeba dopravit z centrály na dané cílové místo

- pro každé cílové místo je dané právě jedno časové okno, čili dvě čísla: *začátek*, *konec*. Vozidlo musí toto cílové místo navštívit nejdříve v čase *začátek* a nejpozději v čase *konec*.
- kapacita vozidel

Existuje mnoho variant problému VRPTW, některé z nich zmiňuji v sekci 2.2.5. Klasická varianta, kterou se v této práci zabývám, se vyznačuje následujícími charakteristikami:

1. existuje právě jedna centrála/depo, ze kterého všechna vozidla vyjíždějí a také se tam vracejí.
2. v centrále je k dispozici vždy aspoň tolik vozidel, kolik je cílových míst
3. v centrále je k dispozici vždy aspoň tolik jednotek zboží, jako je součet požadavků všech cílových míst
4. všechna vozidla jsou identická
5. poptávka každého místa je nejvýše tak velká, jako je kapacita vozidla
6. každé cílové místo má definované právě jedno časové okno
7. každé cílové místo je obsloužené právě jedním vozidlem
8. není stanovený minimální čas začátku rozvozu, tedy čas, kdy mohou vozidla nejdříve odjíždět z centrály
9. obsluha cílového místa proběhne okamžitě, čili vozidlo se v cílovém místě nijak nezdrží, pouze ho navštíví a ihned pokračuje dál
10. minimalizujeme celkový počet ujetých kilometrů, tedy součet délek tras všech vozidel
11. doba na přejezd mezi dvěma místy je přímo úměrná jejich vzdálenosti

Tato upřesnění zadání problému vyžadují podrobnější komentář. Některá z těchto upřesnění jsou pro problém charakteristická a není možné je odstranit. Jedná se například o požadavek na existenci právě jedné centrály nebo požadavek, aby všechna vozidla byla identická. Existují i varianty problému VRPTW, které umožňují například existenci více centrál a vozidel různého typu, ale ty vyžadují odlišné postupy při řešení a v této práci se jimi nezabývám.

Naopak některá z těchto upřesnění slouží pouze pro pohodlnější zacházení s problémem a nejsou v žádném smyslu omezující. Uvedu několik příkladů.

Požadavek 2: při řešení předpokládám, že vozidel je vždy dostatek. Pokud by se stalo, že program nalezne řešení, které využívá více vozidel, než je v centrále k dispozici, pak by to znamenalo, že s menším počtem vozidel bude kvalita řešení horší, tedy bude potřeba najet více kilometrů. Předpokládám ale, že rozvozní problém se řeší opakovaně a pravidelně, například každý den. Pro firmu je tedy vždy výhodnější učinit jednorázovou investici

a nakoupit nové vozidlo, což povede k tomu, že se sníží každodenní náklady, což se v delším časovém horizontu vyplatí. Při řešení tedy předpokládám, že firma už tyto kroky učinila a vozidel bude vždy tolik, kolik je třeba.

Požadavek 5: pokud by poptávka nějakého cílového místa byla větší, než kapacita vozidla, pak se vždy vyplatí vyslat do tohoto místa několik samostatných plně naložených vozidel, dokud se poptávka nesníží a teprve potom začít řešit rozvozní problém. Například pokud by cílové místo mělo poptávku $M > C$, kde C je kapacita vozidla, pak lze najít číslo k tak, aby $k*C \leq M < (k+1)*C$, vyslat do tohoto místa k plně naložených vozidel, tím poptávku snížit na $M - k*C < C$. Pak už je možné řešit situaci jako klasický problém VRPTW. Lze ukázat, že tento postup nesníží nijak výrazně kvalitu nalezeného řešení.

Požadavek 8, aby vozidla směla vyjíždět z centrály kdykoliv, zajišťuje, že jakékoliv zadání problému bude vždy řešitelné. Pokud by byl stanovený minimální čas zahájení rozvozu (například čas 0), mohlo by se stát, že u některého cílového místa nastane konec časového okna dřív, než kolik je minimální čas na přesun vozidla z centrály do tohoto místa. V takovém případě by vozidlo k zákazníkovi vůbec nestihlo dojet, zákazník by zůstal neobsloužen a problém by tedy neměl přípustné řešení.

Požadavek 9: pokud by bylo třeba, aby se vozidlo v cílovém místě nějakou dobu zdrželo, lze tuto situaci jednoduše převést na případ použitý v této práci (tedy že se vozidlo nezdrží). Tento převod spočívá ve zvětšení času potřebného na přesun o hodnotu, kterou se má vozidlo zdržet. Například pokud přejezd z místa *X* do místa *Y* trvá 10 jednotek času a navíc je dané, že se při obsluze místa *Y* musí vozidlo zdržet 2 jednotky času, pak lze do matice přejezdů rovnou zadat hodnotu 12, tedy počítat s tím, že přejezd z *X* do *Y* trvá 12 jednotek a dále už se délkou trvání obsluhy nezabývat.

Poslední požadavek pouze zjednodušuje formát, ve kterém je popsané zadání problému. Pokud by čas na přejezd nebyl přímo úměrný vzdálenosti, bylo by třeba v zadání uvést kromě matice vzdáleností i matici časů na přejezd. Použité algoritmy by to však nijak neovlivnilo.

Pro přehlednost navíc předpokládám, že čas a vzdálenost jsou uvedené v takových jednotkách, že vozidlo ujede přesně jednu jednotku vzdálenosti za jednu jednotku času, čili že rychlost vozidla je 1. To umožňuje hodnoty času a vzdáleností mezi sebou přímo sčítat a porovnávat bez nutnosti složitějších převodů. Tento předpoklad není nijak omezující, protože jednotky lze změnit pouhým vynásobením vhodnou konstantou, což nemá vliv na fungování řešícího algoritmu ani na podobu výsledného řešení.

V dalším textu budu používat pojmy *město, místo, zákazník, vrchol* a *uzel* ve smyslu *cílové místo, které je třeba obsloužit*. Pojem *město* je inspirovaný situací obchodního cestujícího, který navštěvuje jednotlivá města, ale je samozřejmě možné, že v konkrétní situaci leží všechna cílová místa geograficky uvnitř stejného města. (Například rozvoz peněz z banky do bankomatů po Praze.)

2.2.5 Další zobecnění

Dopravní problém lze dále zobecnit několika různými způsoby. Lze například uvažovat situaci, kdy existuje více než jedno centrální depo, ze kterého vozidla vyjíždějí a kam se vracejí. Tato varianta se nazývá *Multiple Depots VRP (MDVRP).* Ve variantě zvané *Split Delivery VRP* je povolené, aby jednoho zákazníka obsloužilo více vozidel, kde každé vozidlo přiveze část jeho poptávky, což může být v některých situacích výhodné.

Další standardní varianty zahrnují zejména *Stochastic VRP*, *VRP with Backhauls* a *Periodic VRP*. Přesný popis těchto variant i některé techniky pro jejich řešení lze nalézt například v (Han, 2010).

Problém lze zobecňovat ještě dál například tak, že umožníme přepravu nákladu mezi kterýmikoli městy (ne jen z centrály k zákazníkům), zavedeme různé typy stanic (například sklady, přístavy, letiště) a různé typy vozidel (automobily, lodě, letadla) a podobně.

Může také existovat vícero způsobů, jak se lze přepravit mezi stejnou dvojicí míst. Například vozidlo může jet rychle po zpoplatněné dálnici, nebo pomalu po běžné silnici. V některých situacích může být výhodné použít rychlý zpoplatněný úsek – například když je třeba stihnout časové okno zákazníka – jindy může stačit běžná silnice.

Další, v praxi se často vyskytující variantou je situace, kdy je optimalizační kritérium složitější než pouze počet najetých kilometrů. Někdy je třeba uvažovat také celkový čas obsluhy, počet použitých vozidel, nejpozdější čas návratu vozidla do depa, počet špatně obsloužených zákazníků, příplatky za přesčasy řidičů a podobně. V praxi je často třeba minimalizovat nějakou lineární kombinaci všech těchto kritérií.

Takto zobecněné dopravní problémy spadají do kategorie *Automatizovaného plánování* a pro jejich řešení se typicky používají odlišné postupy, než pro VRP. Více informací lze nalézt například v (Helmert, 2008). Další části práce se budou věnovat už pouze problému VRPTW.

2.3 Výpočetní složitost dopravních problémů

Jak bylo zmíněno v předchozí části, dopravní problémy se často vyskytují v praxi, a je proto třeba je umět efektivně řešit. Praktické řešení je však provázeno jistými obtížemi plynoucími z vysoké výpočetní náročnosti těchto typů problémů. Následující část popisuje základní algoritmické přístupy k řešení dopravních problémů, jejich vlastnosti, výhody a nevýhody.

2.3.1 Vlastnosti algoritmů

K praktickému řešení dopravních, i jiných problémů se používají automatizované postupy nazývané algoritmy. Těchto postupů existuje celá řada a mohou se lišit některými svými vlastnostmi. V další části budu jako příklad dopravního problému používat TSP, protože je nejjednodušší. Ostatní dříve zmíněné varianty problému, včetně VRPTW jsou přímým zobecněním TSP a všechny zmíněné výsledky se na ně také vztahují.

Mezi nejdůležitější sledované vlastnosti algoritmů patří: (Arora a Barak, 2009)

- **Korektnost**: algoritmus nikdy nevrátí řešení, které by bylo nesprávné, tzn. takové, které by při reálné aplikaci nevedlo ke splnění daných podmínek. Například nekorektní algoritmus pro VRP by mohl vrátit řešení, kde součet požadavků zákazníků na trase převyšuje kapacitu vozidla.
- **Úplnost**: pokud existuje nějaké řešení, pak ho algoritmus vždy najde.
- **Konečnost**: výpočet algoritmu vždy skončí po provedení konečného počtu kroků.
- **Optimalita**: kvalita řešení, které algoritmus nalezl, je nejlepší možná. Například optimální algoritmus pro TSP nalezne vždy takovou trasu, která má nejmenší možnou délku.
- **Časová složitost**: časová složitost algoritmu je závislost počtu vykonaných operací na velikosti vstupu. U dopravních problémů měříme velikost vstupu počtem měst. Například naivní algoritmus pro řešení TSP, který vyzkouší všechny možné způsoby, jak lze tato města projet (čili všechny permutace), bude vyžadovat počet operací úměrný číslu $(n-1)!$, kde n je počet měst. Časová složitost algoritmu dává odhad, jak dlouho bude řešení konkrétního problému na počítači reálně trvat.
- **Paměťová složitost**: podobně jako u časové složitosti, paměťová složitost vyjadřuje závislost množství paměti, které algoritmus vyžaduje, na velikosti vstupu. Paměťová složitost poskytuje odhad, kolik paměti počítače bude potřeba pro realizaci výpočtu. Paměťovou složitost zde zmiňuji jen pro úplnost. V případě dopravních problémů nepředstavuje paměťová složitost žádnou komplikaci, protože všechny používané algoritmy mají zanedbatelné paměťové nároky. Například pro řešení problému s 1000 městy si všechny známé algoritmy vystačí s pamětí menší než 1GB. Dále se tedy paměťovou složitostí už nebudu zabývat.
- **Aproximační poměr**: pro algoritmy, které negarantují nalezení optimálního řešení, můžeme sledovat, jak moc se liší kvalita jimi nalezeného řešení od kvality řešení optimálního. Označme kvalitu řešení, které algoritmus nalezne jako S a kvalitu optimálního řešení jako OPT, kde $OPT \leq S$. Pak pro číslo $k \in R, k \geq 1$, algoritmus nazýváme *k-aproximační*, pokud pro jakékoli zadání problému platí $\frac{S}{OPT} \leq k$. Číslo k se nazývá *aproximační poměr*. Pokud by například existoval postup, který při řeše-

ní TSP najde vždy trasu, jejíž délka je přinejhorším o 60 % větší než délka nejkratší možné trasy, pak by takový algoritmus byl 1,6-aproximační.

2.3.2 Algoritmy použitelné v praxi

V praxi je možné pro řešení problémů používat pouze algoritmy konečné, korektní a úplné, jinak by jejich řešení nebylo použitelné. V dalším textu už budu vlastnosti konečnost, korektnost a úplnost považovat za samozřejmé a nebudu je nadále zmiňovat.

Je také velice žádoucí, aby algoritmus byl optimální a měl co nejmenší časovou a paměťovou složitost. Pro TSP i všechny zobecněné problémy existují optimální algoritmy, ale jejich složitost (myšleno časová složitost) je neúnosná a není možné je v praxi použít pro řešení rozsáhlejších problémů. Obtíže nastávají už v situacích, kdy se počet měst pohybuje kolem hodnoty 30.

Nejrychlejší známý optimální algoritmus pro TSP má složitost 2^n, kde n je počet měst. Pro představu uvádím tabulku zobrazující čas potřebný k běhu algoritmů v závislosti na velikosti zadání a složitosti algoritmu. Složitost vyjadřuje počet operací, a předpokládám, že algoritmy běží na počítači, který je schopný vykonat 2 miliony operací za sekundu. Srovnání ukazuje Tabulka 1.

Tabulka 1. Čas běhu algoritmů o různé složitosti v závislosti na velikosti zadání

Čas výpočtu		*Velikost zadání (n)*						
		1	*20*	*30*	*50*	*100*	*1000*	*10 000*
Složitost	$\log(n)$	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec
	n	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec
	n^2	1 sec	1 sec	1 sec	1 sec	1 sec	1 sec	51 sec
	n^3	1 sec	1 sec	1 sec	1 sec	1 sec	8 minut	6 dní
	2^n	1 sec	1 sec	9 minut	18 let	> milion let	> milion let	> milion let
	3^n	1 sec	29 minut	3 roky	> milion let	> milion let	> milion let	> milion let
	$n!$	1 sec	38 000 let	> milion let	> milion let	> milion let	> milion let	> milion let

Z tabulky je zřejmé, že algoritmy se složitostí 2^n a větší nejsou v praxi absolutně použitelné pro vstupy větší než 30 a ani rychlejší počítač nebo použití více počítačů na tom nic nezmění. Například nákup 10x rychlejšího počítače by způsobil, že čísla v tabulce budou 10-krát menší, což je ale ve většině problematických případů stále nedostatečné.

V teorii složitosti jsou algoritmy, jejichž složitost má tvar polynomu (tzn. funkce ve tvaru n^c, kde c je přirozené číslo) nazývané *polynomiální*. Naproti tomu algoritmy se složitostí ve tvaru c^n nebo vyšší jsou nazývané *exponenciální* (Arora a Barak, 2009). Pro praktické použití je nezbytné, aby použitý algoritmus byl polynomiální.

2.3.3 Aproximační algoritmy a schémata

Aproximační poměr byl definovaný jako reálné číslo. Obecně však můžeme pro libovolný algoritmus vyčíslit jeho aproximační poměr jako poměr kvality nalezeného řešení ku kvalitě řešení optimálního. Tento poměr však typicky není konstantní, ale závisí na velikosti vstupu (Brecklinghaus et al., 2015).

Můžeme například nalézt algoritmus pro TSP, který garantuje, že kvalita nalezeného řešení bude nejvýše $\log(n)$-krát větší než kvalita optimálního řešení, kde n je počet měst. Takový algoritmus by měl aproximační poměr $\log(n)$. Pro praktické použití je vhodné, aby aproximační poměr byl co nejmenší, nejlépe konstantní a blízký číslu 1. Algoritmům, které garantují konstantní aproximační poměr, říkáme *aproximační algoritmy*.

V teorii složitosti byla intenzivně studovaná otázka, zda je možné aproximační poměr libovolně snižovat, případně i za cenu rostoucí časové složitosti, a jak velká příslušná časová složitost bude. V této souvislosti byl zavedený pojem *polynomiální aproximační schéma (PAS)* (Korte et al., 2010, s. 433).

PAS je posloupnost aproximačních algoritmů taková, že každý algoritmus v posloupnosti má lepší aproximační poměr než algoritmus předchozí a všechny jsou polynomiální vzhledem k velikosti vstupu. Při existenci PAS je tedy možné aproximační poměr snižovat libovolně blízko k hodnotě 1. S nižším aproximačním poměrem však typicky roste složitost algoritmu.

2.3.4 Neexistence efektivních algoritmů pro dopravní problémy

Mohlo by se zdát, že vysoká složitost optimálních algoritmů pro dopravní problémy je způsobená faktem, že zatím nikdo žádné lepší postupy nevymyslel, a že se to může v budoucnu podařit. Situace je však jiná. Je možné dokázat, že všechny optimální algoritmy budou mít vždy vysokou časovou složitost a dokonce ani neoptimální algoritmy, které garantují rozumný aproximační poměr, nemohou být efektivní. Některé z těchto výsledků zde představím.

Budu zde mluvit pouze o složitosti problému TSP. Vzhledem k tomu, že TSP je speciálním případem problémů VRP, TSPTW i VRPTW, vztahují se všechna popsaná omezení i na tyto zobecněné problémy.

1. Neexistuje algoritmus, který řeší optimálně obecnou variantu problému TSP a jeho časová složitost je polynomiální (Arora a Barak, 2009, s. 56).
2. Neexistuje algoritmus, který řeší optimálně Euklidovské TSP (a tedy ani žádnou složitější variantu) a jeho složitost je polynomiální. (Korte et al., 2010, s. 405).
3. Neexistuje polynomiální algoritmus, který by u obecné varianty TSP garantoval konstantní aproximační poměr. (Korte et al., 2010, s. 557).
4. Pro variantu *TSP s metrikou* existuje polynomiální aproximační algoritmus garantující poměr $3/2$, ale neexistuje polynomiální aproximační schéma. Aproximační poměr tedy není možné libovolně snižovat (Korte et al., 2010, s. 560).
5. Pro Euklidovskou variantu TSP existuje polynomiální aproximační schéma. Toto schéma ale není v praxi příliš použitelné, protože poskytované garance jsou příliš nízké a výpočetní čas je sice polynomiální, ale i tak neúnosně vysoký. (Korte et al., 2010, s. 568).

Všechna výše uvedená tvrzení platí pouze za jistého předpokladu, který se v literatuře běžně označuje jako $P \neq NP$. Tento předpoklad zde nebudu detailně popisovat, neformálně však lze říct, že $P \neq NP$ je domněnka z teorie složitosti ohledně výpočetních možností algoritmů. Zatím se ji nepodařilo formálně dokázat, ale většina výzkumníků věří v její platnost. Pokud by se ukázalo, že domněnka je nepravdivá, mělo by to dalekosáhlé teoretické i praktické důsledky, například většina dnes používaných šifrovacích protokolů by přestala být bezpečná. (Arora a Barak, 2009, s. 59).

Je třeba ještě podotknout, že teorie složitosti se zabývá nejhorším možným případem. V praktických aplikacích však nejhorší možný případ nenastává často, a proto i algoritmy, jejichž chování je teoreticky velmi neefektivní, mohou občas v praxi fungovat dobře.

3 Základní algoritmy pro řešení TSP a VRPTW

Algoritmy lze rozdělit do dvou kategorií podle toho, zda garantují nalezení optimálního řešení, nebo ne. Vzhledem k výpočetní složitosti problému lze optimálně řešit jen malé instance a v praxi se proto často používají algoritmy neoptimální. Další části textu shrnuje používané řešící přístupy a popisuje jejich výhody a nevýhody.

3.1 Celočíselné programování

Celočíselné programování (*Mixed Integer Programming – MIP*) je varianta lineárního programování, kde kromě linearity omezujících podmínek a účelové funkce jsou navíc kladeny požadavky na celočíselnost některých proměnných.

Existuje několik způsobů, jak lze VRPTW modelovat pomocí MIP, viz například (Baldacci et al., 2011; 2014; Fukasawa et al., 2006).

Pro optimální řešení úloh celočíselného programování se používá simplexová metoda (nebo jiný algoritmus pro řešení klasického lineárního programování) spolu s technikami, které zajišťují splnění požadavků na celočíselnost. Nejpoužívanějšími technikami jsou:

1. Metoda větví a mezí (*Branch and Bound*)
2. Metoda sečných nadrovin (*Cutting-plane method*)
3. Metoda větví a řezů (*Branch and Cut*)
4. Generování sloupce (*Column Generation*)
5. Metoda větví a cen (*Branch and Price*)

Bližší popis fungování těchto metod je nad rámec této práce a lze jej dohledat v příslušných zdrojích, například (Pelikán, 2001; Fukasawa et al., 2006; Korte et al., 2010, s. 51-127).

Zmíněné techniky garantují nalezení optimálního řešení, ale neposkytují garance na množství potřebného výpočetního času. U rozsáhlých problémů tak může výpočetní čas narůst do obrovských rozměrů. Některé z metod – například metodu Branch and Bound – lze použít i ne-optimálním způsobem tak, že vrátí první nalezené celočíselné řešení, bez garance jeho optimality.

V praxi se často používá tzv. *any-time* varianta (Zhang, 1999, s. 117-120). V takovém případě algoritmus během prohledávání hlásí, jaké nejlepší řešení zatím nalezl a snaží se jej neustále zlepšovat s tím, že uživatel může tento proces kdykoliv přerušit, pokud už nechce výpočtu věnovat více času nebo považuje kvalitu stávajícího řešení za dostačující.

V praxi se k řešení úloh MIP používá specializovaný řešící software. K nejznámějším řešičům patří například CPLEX (https://www.ibm.com/bs-en/marketplace/ibm-ilog-cplex), Gurobi (http://www.gurobi.com/) nebo LINGO (http://www.lindo.com/).

Výhody: představené techniky pro TSP lze snadno upravit i pro problém VPR a VRPTW pouhým přidáním dodatečným omezení do lineárního programu. Lze navíc jednoduše přidat například omezení na maximální počet vozidel a podobně.

Další výhodou je, že využívají standardní formát, pro který existuje mnoho řešičů, takže každý pokrok ve vývoji řešících algoritmů se okamžitě automaticky promítne do lepší kvality nalezených řešení.

Nevýhodou je zejména velká časová náročnost v případě, že je potřeba najít kvalitní řešení u problémů většího rozsahu.

3.2 Heuristiky

Výrazem „heuristika“ je zde obecně míněn postup, který hledá přípustné a pokud možno kvalitní řešení daného problému opakovanou aplikací nějakého poměrně jednoduchého pravidla. Tyto postupy negarantují optimalitu, ani se o to nesnaží, ale pracují typicky velmi rychle.

Příkladem heuristiky pro problém TSP může být následující postup:

1. Začni v libovolném vrcholu.
2. Mezi vrcholy, které ještě nebyly navštívené, vyber ten, který je nejblíž a navštiv ho.
3. Opakuj bod 2, dokud nejsou všechny vrcholy navštívené.
4. Vrať se do počátečního vrcholu.

Tento postup je známý jako *Hladový algoritmus pro TSP* nebo také *Metoda nejbližšího souseda*. Je velmi rychlý a dokáže většinou najít aspoň trochu kvalitní řešení.

Představím zde několik dalších populárních heuristik pro TSP, z nichž většina se dá snadno rozšířit na heuristiky pro VRP. V další kapitole popíšu, jak lze některé z nich zobecnit také pro problém VRPTW. Popis většiny heuristik je převzatý z (Pelikán, 2001).

Metoda Clarke and Wright

Také nazývaná *Metoda výhodnostních čísel*. Pro každou hranu odhadneme, jak výhodné je tuto hranu použít na okružní cestě pomocí tzv. *výhodnostního čísla*. Poté přidáváme hrany od nejvýhodnějších, dokud nevytvoříme cyklus obsahující všechny vrcholy. Formálně:

1. Vyber libovolný vrchol a označ ho číslem 1.

2. Pro každý vrchol $x \neq 1$ vytvoř cykly 1 → x → 1.
3. Pro každou dvojici vrcholů *i, j* > *1*, kde i ≠ *j* spočti výhodnostní číslo $v_{i,j} = d_{i,1} + d_{1,j} - d_{i,j}$, kde $d_{i,j}$ je vzdálenost z vrcholu i do vrcholu *j*.
4. Procházej čísla $v_{i,j}$ v pořadí od největšího a nahrazuj příslušné hrany *(i, 1) a (1, j)* pomocí *(i, j).* Hrany, jejichž přidání by předčasně vytvořilo cyklus, přeskakuj.
5. Jakmile vznikne cyklus, který obsahuje každý vrchol právě jednou, skonči.

Technika *Clarke and Wright* na problému *TSP s metrikou* poskytuje garanci $S/OPT \leq \log_2 n$, kde n je počet měst (Brecklinghaus et al., 2015).

Metoda vkládání

Smyslem je vytvořit cyklus složený pouze ze dvou vrcholů a postupně do něj na vhodná místa vkládat další vrcholy, dokud nebude cyklus obsahovat všechny vrcholy.

1. Začni v libovolném vrcholu, označ ho číslem 1.
2. Vyber vrchol *x,* který je od vrcholu 1 nejvzdálenější a vytvoř z nich počáteční cyklus.
3. Podle zvoleného kritéria vyber vrchol *y*, který ještě není v cyklu, pak vyber vhodnou pozici v cyklu a vlož vrchol *y* na zvolenou pozici.
4. Jakmile vznikne cyklus, který obsahuje všechny vrcholy, skonči.

Kritérií pro volbu vhodného vrcholu a vhodné pozice je několik. Lze zvolit například vrchol, který je nejbližší k některému vrcholu na cyklu nebo vrchol, jehož přidáním se cyklus co nejméně prodlouží a podobně. Podrobnější popis lze nalézt v (Pelikán, 2001, s. 147).

Metoda konvexního obalu

Metoda je podobná předchozí s tím rozdílem, že je uzpůsobená pro konkrétní variantu TSP, a to sice pro *Euklidovské TSP*. Funguje následujícím způsobem:

1. Najdi konvexní obal množiny vrcholů.
2. Vytvoř počáteční cyklus z vrcholů na okraji obalu v pořadí, jak jdou za sebou.
3. Pro každý vrchol k neležící na cyklu a každou dvojici vrcholů *i, j*, ležících na cyklu spočti $s_{i,j,k} = (d_{i,k} + d_{k,j}) / d_{i,j}$, kde $d_{i,j}$ je vzdálenost z vrcholu i do vrcholu *j*.
4. Najdi minimum z čísel $s_{i,j,k}$ a vlož odpovídající vrchol k do cyklu mezi vrcholy *i, j*.
5. Opakuj body 3 a 4 dokud nevznikne cyklus obsahující všechny vrcholy.

Všechny předchozí techniky patřily mezi tzv. *konstrukční heuristiky*, protože se věnovaly vytváření nového řešení. Druhou kategorií jsou tzv. *zlepšující heuristiky*, které se snaží již

známé řešení vylepšit. Příkladem tohoto typu heuristiky je metoda *k-OPT* zvaná také *Metoda výměn*.

Metoda *k-OPT*

Metoda pracuje tak, že zvolí nějakých k hran ve stávajícím řešení a snaží se je nahradit jinými k hranami tak, aby nové řešení bylo lepší než původní. Podrobně popíšeme postup pro $k = 2$.

1. Vyber 2 hrany na cyklu, které nesdílejí žádný společný vrchol, a odeber je.
2. Nahraď je jinou dvojicí hran tak, aby opět vznikl hamiltonovský cyklus, ale jiný než původní.
3. Pokud je nový cyklus lepší než původní, změnu ponechej, jinak změnu vrať a zkus jinou dvojici hran.
4. Pokud už neexistuje žádná dvojice hran, jejímž nahrazením lze řešení zlepšit, tak skonči.

Metoda *k-OPT* se používá pouze pro *symetrické TSP*. Při prohazování hran se totiž může změnit pořadí, v jakém určitý úsek cesty projdu. Prohození dvou hran funguje stejně, jako kdyby se jistý úsek cesty vyjmul a poté vložil do cyklu pozpátku.

V případě Euklidovského TSP je vhodné vybírat na výměnu hrany, které se kříží. Nutno podotknout, že heuristika *k-OPT* negarantuje polynomiální časovou složitost a v nejhorším případě může být počet kroků až $2^{\frac{n}{2}}$ (Aarts a Lenstra, 2003, s. 230-246). V praxi však většinou funguje dobře. U varianty *TSP s metrikou* poskytuje tato technika garanci $S/OPT \leq \log_2 n$, kde n je počet měst (Brecklinghaus et al., 2015).

Výhody: heuristiky jsou rychlé a jednoduché na implementaci. V praxi většinou dokážou nalézt relativně kvalitní řešení, ale ne vždy se to podaří.

Nevýhodou je, že nalezené řešení občas obsahuje části, které jsou očividně neefektivní. Proto je výsledné řešení často třeba ještě nějak dále zpracovat. Heuristiky navíc neposkytují žádné garance ohledně kvality řešení.

3.3 Aproximační algoritmy a schémata

Aproximační algoritmy se od předchozích heuristik liší tím, že poskytují fixní garanci na kvalitu nalezeného řešení.

Metoda minimální kostry

1. Najdi minimální kostru daného (neorientovaného) grafu.
2. Zdvoj všechny hrany v kostře (čili ke každé hraně $x \rightarrow y$ doplň hranu $y \rightarrow x$)
3. Najdi v takto vzniklém grafu Eulerův cyklus, tedy cyklus procházející přes všechny vrcholy.
4. Začni v libovolném vrcholy a projdi celý cyklus. Vrcholy přidávej v tomto pořadí do posloupnosti. Vrcholy, které už v posloupnosti jsou, přeskakuj.

Tato technika najde hamiltonovskou kružnici v grafu a pro variantu *symetrického TSP s metrikou* dává garanci $S/OPT \leq 2$ (Korte et al., 2010, s. 559).

Christofidova metoda

Je také založena na hledání minimální kostry a poskytuje o něco lepší garanci kvality.

1. Najdi minimální kostru daného (neorientovaného) grafu, označ ji T.
2. Vyber vrcholy, které mají v kostře lichý stupeň. Těch je vždy sudý počet.
3. Najdi mezi těmito vrcholy minimální párování (v původním grafu) a hrany z tohoto párování přidej do T. Tím mají všechny vrcholy v T sudý stupeň.
4. V grafu T najdi Eulerův cyklus a ten transformuj na hamiltonovskou kružnici stejně, jako v předchozím případě.

Stejně jako v předchozím případě, také tato metoda využívá neorientovanou kostru a předpoklad o trojúhelníkové nerovnosti vzdáleností, takže poskytuje garance jen u varianty *symetrického TSP s metrikou.* Aproximační poměr je v tomto případě $S/OPT \leq 3/2$ (Korte et al., 2010, s. 559).

Metoda spojování cyklů

Také známá jako metoda zatřiďování. Tato metoda vytvoří několik malých cyklů a ty pak postupně spojuje do stále větších, dokud nezůstane jediný cyklus obsahující všechny vrcholy.

Spojování cyklů probíhá následovně: předpokládejme, že chceme spojit dva cykly $x_1, x_2, x_3, ...x_m, x_1$ a $y_1, y_2, y_3, ...y_p, y_1$ tak, že spojení proběhne přes vrcholy x_i a y_j. V tomto případě zrušíme hrany x_i, x_{i+1} a y_j, y_{j+1}, a přidáme hrany x_i, y_{j+1} a y_j, x_{i+1}. Cena spojení je pak rozdíl ohodnocení hran, které jsme přidali a které jsme odebrali. Cykly spojujeme vždy přes takové vrcholy, aby byla cena spojení minimální.

1. Pro každý vrchol x vytvoř jednoprvkový cyklus, tedy hranu, která začíná i končí v x. Množinu těchto cyklů označ Q.

2. Vyber z množiny Q dva cykly takové, že cena za jejich spojení je minimální.
3. Tyto dva cykly vyjmi z množiny Q, spoj je a výsledek opět vlož do Q.
4. Opakuj body 2 a 3 dokud není v Q jen jeden cyklus.

Tento postup garantuje u varianty *symetrické TSP s metrikou* aproximační poměr 2 (Brecklinghaus et al., 2015).

Výhodou aproximačních algoritmů je, že jsou velmi rychlé, podobně jako heuristiky, ale navíc poskytují garance na kvalitu řešení. Některé z nich mají také teoretické využití, protože ukazují, že problém TSP je možné aproximačně řešit.

Nevýhody: heuristiky i aproximační algoritmy mají nevýhodu v tom, že mohou často uvíznout v tak zvaném lokálním optimu, tedy ve stavu, kde už heuristika nenajde žádné lepší řešení, ale přesto ještě není v optimálním řešení. Také praktické výsledky nejsou vždy tak kvalitní jako například při použití metaheuristik nebo složitějších technik.

3.4 Metaheuristiky

Metaheuristiky jsou v současné době velmi populární pro řešení těžkých úloh v oblasti diskrétní i spojité optimalizace (Rothlauf, 2011). Metaheuristiky jsou složitější než heuristiky: mohou například používat více různých heuristik současně, aplikovat je opakovaně či střídavě, často pracují současně s větším množstvím různých řešení a tato řešení navzájem kombinují a upravují.

Mezi populární metaheuristiky používané pro řešení TSP a VRPTW patří zejména *Genetické algoritmy* a *Ant Colony Optimization*.Viz například (Vaira, 2014; Pellonpera, 2014).

Výhodou těchto postupů je, že většinou dokáží nalézt lepší řešení než heuristiky, a jsou proto často používané v praxi. Nevýhodou je o něco delší čas běhu a absence garancí kvality.

3.5 Další používané techniky

Další možnosti řešení těžkých problémů z oblasti kombinatorické optimalizace zahrnují například využití *hyper-heuristik* (Burke et al., 2013), tzv. *hybridních algoritmů*, které kombinují více přístupů, a přístupy využívající *strojové učení*. Více informací o aktuálně používaných technikách lze nalézt zejména ve sbornících příslušných konferencí, jako jsou například HAIS (http://hais2017.unirioja.es/) a CPAIOR (http://cpaior2017.dei.unipd.it/).

4 Návrh heuristiky pro VRPTW

V této části popisuji návrh heuristik pro VRPTW. Jak bylo zmíněno v předchozích kapitolách, heuristiky lze rozdělit na *konstrukční*, které postupně budují počáteční řešení problému, a *modifikační*, které již známé řešení upravují a zlepšují.

Práce se zaměřuje pouze na modifikační heuristiky, protože pomocí vhodné modifikační heuristiky lze vždy dosáhnout stejného výsledku, jako pomocí kterékoli heuristiky konstrukční. Rozdíl je pouze v tom, že v průběhu výpočtu konstrukční heuristiky je průběžné řešení neúplné.

Například při použití heuristiky nejbližšího souseda pro TSP jsou v průběhu výpočtu některá města nenavštívená a teprve po dokončení celého postupu získáme úplné, validní řešení. Stejného výsledku však lze dosáhnout, pokud v nenavštívené části fiktivně doplníme jakoukoli trasu, například náhodně, a poté provádíme kroky stejně, jako v případě heuristiky nejbližšího souseda. Tím se tento postup stane modifikační heuristikou, protože pouze upravuje již získané řešení, a povede při tom ke stejnému výsledku.

4.1 Obecný formalismus pro popis heuristik

Navrhl jsem formalismus pro obecný popis modifikačních heuristik založený na myšlence lokálního prohledávání.

4.1.1 Konstrukce počátečního validního řešení

Vzhledem k tomu, že modifikační heuristiky pouze upravují již známé validní řešení, je k jejich úspěšnému použití potřeba nějaké počáteční validní řešení již mít. Pro vytvoření počátečního validního řešení využívám jednoduchý postup: ke každému zákazníkovi pojede samostatné vozidlo, které ho obslouží, a poté se vrátí zpátky do centrály. Tím vznikne řešení, které bude mít sice nevalnou kvalitu, ale bude jistě validní.

Při konstrukci takového řešení je použité pravidlo, že vozidlo přijede do prvního místa na trase přesně v době, kdy v tomto místě začíná časové okno. Toho lze vždy dosáhnout, protože vozidlo může z centrály vyjíždět kdykoliv (není stanovený minimální začátek rozvozu), čili čas odjezdu lze stanovit tak, aby po započítání doby na přejezd vozidlo dorazilo přesně v čase, kdy v cílovém místě začíná časové okno.

Lze snadno ukázat, že tento přístup dává nejlepší výsledky. Pokud by vozidlo vyjelo dřív, muselo by v cílovém místě čekat než by mohlo provést obsluhu, což je zbytečné. Pokud by naopak vyjelo později, pak navštíví příslušného zákazníka (a také všechny další zákazníky

v pořadí) později, což nikdy nepřináší žádnou výhodu a naopak to může být nevýhodné, protože hrozí, že vozidlo už nestihne u některého zákazníka na cestě jeho časové možnosti.

4.1.2 Lokální modifikace

Heuristiky budou založené na tzv. *modifikačních operacích*. Modifikační operace dostane na vstupu validní řešení a vrátí jiné validní řešení, které se bude od původního mírně lišit, například navštíví některé dvě místa v opačném pořadí a podobně. Konkrétní způsoby modifikací jsou podrobně popsané v kapitole 5.

Modifikační operace lze poté využít ke zlepšování kvality řešení. Pro dané validní řešení Q označme $N(Q)$ jako množinu validních řešení, které vzniknou z řešení Q jeho modifikací. Množinu $N(Q)$ budu nazývat *okolím řešení* Q.

Jakmile máme k dispozici nástroje pro vytvoření počátečního řešení a pro generování okolí, můžeme pomocí nich popsat několik druhů heuristik:

Hladová heuristika

1. Vytvoř počáteční řešení Q.
2. Vytvoř okolí $N(Q)$ řešení Q.
3. Pokud je množina $N(Q)$ prázdná, skonči.
4. Vyber v množině $N(Q)$ řešení s nejvyšší kvalitou a označ ho P.
5. Pokud má řešení P horší kvalitu než Q, skonči.
6. Jinak nahraď řešení Q řešením P a pokračuj bodem 2.

Náhodná procházka délky *k*

1. Vytvoř počáteční řešení Q.
2. Vytvoř okolí $N(Q)$ řešení Q.
3. Pokud je množina $N(Q)$ prázdná, skonči.
4. Vyber v množině $N(Q)$ libovolné řešení a nahraď jím Q.
5. Opakuj od bodu 2, *k*-krát.

4.2 Reprezentace dat v průběhu výpočtu

Řešení je v průběhu výpočtu reprezentované jako množina *okruhů*. Každý okruh má přiřazeno identifikační číslo, aby bylo možné kontrolovat, jestli jsou daná dvě místa na stejné trase, nebo ne. Jeden okruh (občas budu používat také slovo *trasa*) odpovídá cestě jednoho vozidla. Začíná v centrále, objíždí některé zákazníky a vrací se do centrály.

Ke každému cílovému místu jsou přiřazené tyto informace:

- Identifikační číslo příslušného místa
- Identifikační číslo místa, které předchází tomuto místu na trase
- Identifikační číslo místa, které následuje po tomto místě na trase
- Čas návštěvy
- Identifikační číslo okruhu, ve kterém toto místo leží.

Tyto informace závisejí na konkrétním řešení a v průběhu výpočtu se mění. Budu je nazývat *dynamické*.

Dále mám k dispozici *statické* informace ohledně zadání problému. Tyto se v průběhu výpočtu nemění a nejsou spjaté s konkrétním řešením:

- Celkový počet míst a jejich vzájemné vzdálenosti
- Poptávka jednotlivých míst a kapacita vozidel
- Začátek a konec časového okna pro každé místo

K rozdělení informací na statické a dynamické jsem přistoupil z důvodu efektivity. Dynamická data musí být zkopírovaná v každém řešení, protože toto řešení určují. Naproti tomu statická data jsou uložená pouze jednou a všechna řešení na ně mají odkaz.

4.3 Kontrola validity řešení

Na rozdíl od TSP je při úpravě řešení potřeba zajistit, aby byla dodržená časová okna u všech zákazníků a nebyla překročena kapacita žádného vozidla, čili jestli nově vzniklé řešení je tzv. validní.

Při jakékoli úpravě řešení je tedy třeba validitu kontrolovat. Tuto kontrolu je možné provádět tak, že se při každé modifikaci projde celá trasa od začátku do konce, přepočítají se všechny údaje a zkontroluje se, že vedou k validnímu řešení. Tento způsob je však poměrně neefektivní, například proto, že kontroluje i ty části trasy, které se nezměnily.

Navrhl jsem způsob kontroly validity, který funguje inkrementálně, tedy pouze zkontroluje, že provedená modifikace nezmění validitu současného řešení. Tímto postupem lze za-

jistit, že pokud bylo původní řešení validní, pak modifikované řešení bude také validní. Validitu původního řešení již není třeba ověřovat.

Výhodou tohoto postupu je efektivnější provedení celého výpočtu, nevýhodou je složitější fungování algoritmu a nutnost udržovat dodatečné informace a při každé modifikaci je aktualizovat.

Pracuji s těmito dodatečnými dynamickými informacemi:

- Pro každý okruh: *součet poptávky všech míst na trase*
- Pro každý okruh: *aktuální délka trasy*
- Pro každé místo: *časová rezerva.*

Součet poptávky všech míst na trase umožňuje rychle kontrolovat, jestli při přidání nového cílového místa do trasy bude stačit kapacita vozidla. Při modifikaci řešení se pak tento součet aktualizuje tak, že při přidání nového místa se zvýší o poptávku tohoto místa a při odebrání se naopak odpovídajícím způsobem sníží.

Samozřejmě by bylo možné tuto proměnnou vynechat a při každé modifikaci součet znovu spočítat tak, že projdu celou trasu. To by ale byl přesně ten příklad neefektivity, které se chci vyhnout.

Aktuální délka trasy je užitečná při zjišťování a porovnávání kvality daného řešení a aktualizuje se při každé modifikaci. Opět by bylo možné tuto proměnnou vynechat a počítat délku vždy znovu což by také vedlo k neefektivitě.

Nejzajímavější z těchto dodatečných informací je *časová rezerva* jednotlivých míst. Časovou rezervu místa *X* definuji jako minimum z rozdílu (*konec časového okna – čas návštěvy vrcholu*) přes všechny vrcholy na trase od vrcholu *X* až do konce. Časová rezerva uzlu *X* tedy udává, o kolik nejvýše může vozidlo přijet později do uzlu *X* tak, aby ještě stihlo časové okno v uzlu *X* i ve všech uzlech, které po *X* na trase následují.

Časová rezerva umožňuje efektivně kontrolovat, jestli vložení nového uzlu do trasy nezpůsobí to, že kvůli vzniklému zpoždění nestihne vozidlo obsloužit nějaký z dalších uzlů na své trase. Bez informace o časových rezervách by bylo nutné překontrolovat celý zbytek trasy jeden uzel po druhém.

U časových rezerv je však o něco složitější aktualizace hodnot. Při lokální modifikaci řešení (například vložení nového uzlu *X* do trasy), je třeba aktualizovat časové rezervy u všech uzlů, které *předcházejí* uzlu *X* na trase. Zavedení časových rezerv se však přesto vyplatí, jak je vysvětleno v následující části.

4.4 Množina operací a množina modifikovaných řešení

Okolí řešení *Q* bylo definované jako *množina validních řešení, které vzniknou z Q jeho modifikací*. Pro modifikaci řešení využívám tzv. *operace*. Operace je postup, který lze na daném řešení vykonat tak, že jeho vykonáním vznikne řešení nové. Pokud je toto nové řešení validní, pak říkám, že příslušná *operace* je *validní*.

Jako ukázku modifikační operace lze uvést například výměnu pořadí dvou vrcholů na trase. Operace, které jsem navrhl a které při řešení používám, jsou podrobně popsané v další části práce.

Lze si všimnout, že zdaleka ne každá modifikační operace je validní. Například pokud zůstanu u příkladu výměny pořadí dvou uzlů na stejné trase, pak některé výměny mohou fungovat a dokonce vést ke zlepšení řešení, ale v některých případech výměnu provést nelze z důvodu porušení podmínky časových oken. Vzhledem k tomu, že okolí bylo definované jako množina *validních* řešení, je třeba při generování okolí validitu kontrolovat.

4.4.1 Zlepšující operace

Při modifikaci by se mělo řešení postupně zlepšovat, čili algoritmus by měl směřovat k takovým modifikacím, které vedou na řešení, jehož kvalita je vyšší než kvalita původního řešení. Z tohoto důvodu už při generování okolí sleduji, jaká je kvalita nově vygenerovaných řešení. Definuji pojem *zlepšení řešení* následovně:

Nechť Q je validní řešení a $N(Q)$ je jeho okolí. Pak pro řešení $P \in N(Q)$ je $zlepšení(P) = délka\_trasy(Q) - délka\_trasy(P)$. Zlepšení tedy udává, o kolik je modifikované řešení lepší než původní. V případě, že nové řešení je horší než původní, je jeho *zlepšení* záporné.

Zavedení pojmu *zlepšení* je opět motivované vyšší efektivitou. Zlepšení je možné zjistit přímo z operace, kterou na původním řešení vykonávám, a není proto třeba kvalitu celého řešení přepočítávat. Například při prohození dvou vrcholů na trase je možné zlepšení spočítat pouze ze znalosti vzdáleností mezi prohozenými vrcholy, jejich předchůdci a jejich následníky na trase. Není tedy třeba procházet celou, potenciálně dlouhou trasu, nebo dokonce přepočítávat i jiné trasy. Konkrétní vzorce pro výpočet zlepšení pro jednotlivé operace jsou uvedené v kapitole 5.

4.4.2 Generování okolí pomocí modifikačních operací

Pro generování okolí používám následující postup:

1. V = prázdný seznam.
2. Procházej všechny možné kandidáty na modifikační operace.

(například všechny dvojice vrcholů na stejné trase, které zkusím prohodit)

3. Zkontroluj, jestli je daná operace validní.

(například jestli prohození této dvojice neporuší podmínky na časová okna atd.)

4. Pokud je validní, spočítej *zlepšení* operace.

5. Přidej do seznamu *V* dvojici (kódOperace, zlepšeníOperace).

(například kód „P(4,7)“ může značit operaci prohození míst s čísly 4 a 7)

6. Pokud existují další kandidáti, jdi na bod 2, jinak skonči a vydej seznam *V*.

Seznam *V* pak reprezentuje hledané okolí. Přesněji řečeno, seznam obsahuje pouze operace, nikoli řešení. Prvky okolí je třeba teprve vytvořit tak, že aplikujeme jednotlivé operace na původní řešení. Tento způsob je na první pohled zbytečně komplikovaný ve srovnání s přímočarým postupem, kdy bychom rovnou generovali jednotlivé prvky okolí a vůbec nepotřebovali zavádět pojem *operace*. Ve skutečnosti je však řešení pomocí operací efektivnější.

Aplikace operace (tedy vykonání příslušné změny a vytvoření nového řešení) je časově mnohem náročnější než pouhé zjištění její validity a jejího zlepšení. Kupříkladu při operaci prohození některých dvou míst na trase je možné velmi rychle spočítat, jestli je tato operace validní, nebo ne, pomocí postupu naznačeného výše. Stejně tak je možné rychle zjistit, jaké zlepšení mi aplikace této operace přinese. Samotné vytvoření nového řešení však trvá dlouho.

Pokud by operace modifikovala trasu v nějakém vrcholu *X*, například za něj vložila nový vrchol *Y*, pak tím vznikne zpoždění a změní se časy návštěv u všech vrcholů na trase od *Y* až do konce. Je tedy třeba tuto část trasy celou projít a příslušné informace aktualizovat. Navíc se mohly změnit i časové rezervy míst a tuto informaci je třeba propagovat ke všem vrcholům, které vrcholu *X* předcházejí. Je tedy nutné přepočítat celou trasu.

Heuristiky, které při řešení používám, nikdy nepracují s celou množinou $N(Q)$, ale typicky jen s její malou částí. Například hladová heuristika volí vždy prvek s největším zlepšením a pouze s tím dále pracuje. Je tedy nevýhodné generovat vždy všechny prvky množiny, ale stačí mít k dispozici jen informace potřebné pro zjištění, který prvek množiny je nejlepší (čili informace o validitě a o zlepšení) a tento jediný prvek pak vygenerovat. Použití operací takové fungování umožňuje.

5 Popis jednotlivých typů modifikačních operací

5.1 Formát a značení

Všechny operace pracují s validními řešeními a modifikují některé jejich charakteristiky. Pro popis fungování operací používám následující značení:

- Velká písmena latinské abecedy (např. A, B, C, atd.) označují jednotlivé uzly v grafu. Uzel může reprezentovat buď depo, nebo koncového zákazníka.
- Vzdálenost mezi uzly budu značit pomocí symbolu *d(X, Y),* například vzdálenost míst A a B značím *d(A, B).* Občas budu pro tyto vzdálenosti používat také malá písmena latinské abecedy, například *a=d(A, B).* Vzhledem ke konvencím popsaným v 2.2.4 odpovídá vzdálenost mezi městy také času potřebnému pro přejezd. Budu tedy výrazem *d(A,B)* označovat také čas potřebný k přejezdu z A do B.
- U uzlů, které odpovídají koncovým zákazníkům, budu používat tyto atributy:
    - *Visit*: čas návštěvy příslušného uzlu. (Např. *A.Visit = 345* znamená, že vozidlo provedlo obsluhu uzlu a v čase 345.)
    - *Start*: začátek časového okna v příslušném uzlu. (Např. *A.Start = 123* znamená, že časové oknu v uzlu a začíná v čase 123.)
    - *End*: konec časového okna v příslušném uzlu.
    - *Demand*: velikost poptávky v příslušném uzlu. (Např. *A.Demand = 12* znamená, že do uzlu a je třeba dovézt *12* jednotek zboží.)
    - *Next*: uzel, který následuje po tomto uzlu na trase. (Například *A.Next = C* znamená, že po návštěvě uzlu A bude vozidlo pokračovat do uzlu *C*.)
    - *Prev*: uzel, který předchází tomuto uzlu na trase. (Například *A.Prev = B* znamená, že vozidlo přijelo do uzlu A z uzlu *B*. Pro každý uzel *X* platí, že *(X.Prev).Next = X* a také *(X.Next).Prev = X*.
    - *Reserve*: časová rezerva v příslušném uzlu. (Časová rezerva byla definovaná v sekci 4.3.)
- Výrazem *Capacity* budu označovat kapacitu vozidel.

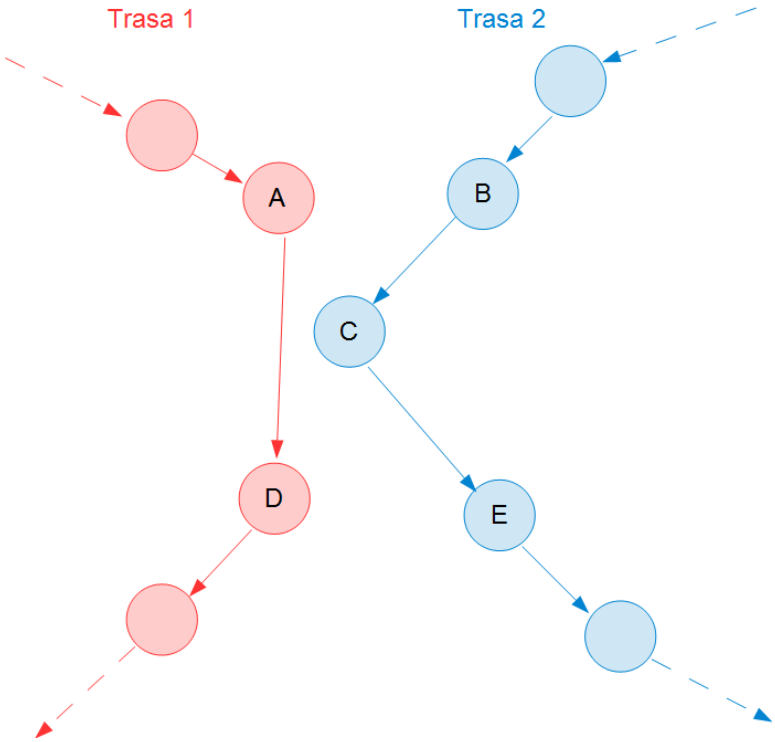
5.2 Operace přesunu

Operace je založená na vložení nového vrcholu do již existující trasy a je podobná vkládací heuristice pro TSP.

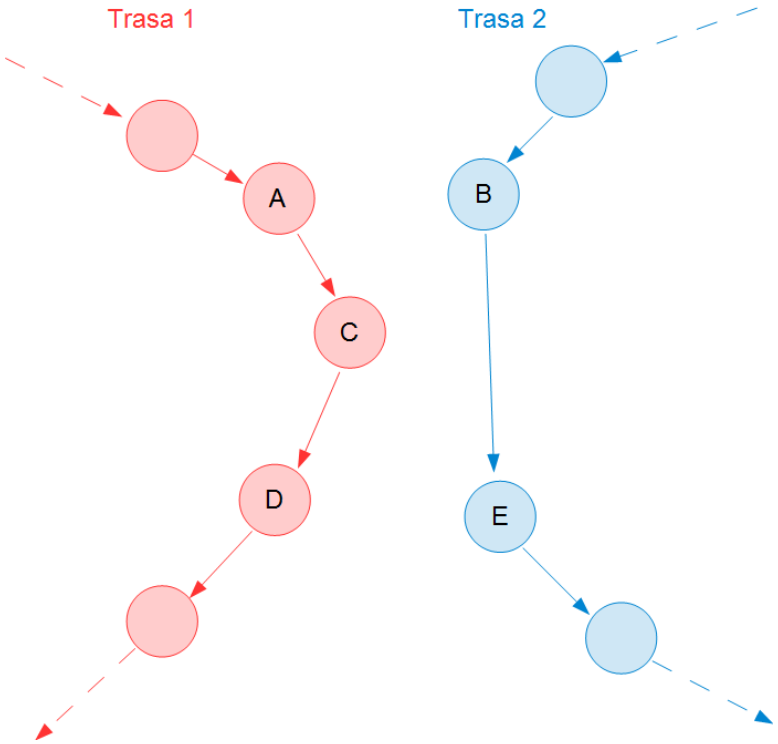
Před vložením vrcholu do nové trasy je tento vrchol nejprve odstraněn z trasy původní. Uvažujeme pouze vkládání vrcholu do *jiné* trasy, tedy situaci, kdy bude cílové místo obslouženo jiným vozidlem než předtím.

Vkládání do stejné trasy by způsobilo pouze změnu pořadí míst na trase vozidla. Tuto operaci neprovádím, protože pro přeuspořádání vrcholů na stejné trase používám speciální operaci popsanou v sekci 5.4.

Následující dva obrázky (Obrázek 3 a Obrázek 4) ukazují příklad použití operace přesunu. Obrázek 3 zobrazuje situaci před použitím operace a Obrázek 4 situaci po použití operace.



Obrázek 3. Ukázka situace před použitím operace přesunu.



Obrázek 4. Ukázka situace po použití operace přesunu. Přesunul se vrchol C za vrchol A.

5.2.1 Kontrola validity

Před uskutečněním přesunu je třeba zkontrolovat, zda vozidlo, ke kterému nový vrchol přidáváme, bude mít dostatečnou kapacitu, a zda zpoždění, které vložením vrcholu vznikne, nezpůsobí problémy u dalších vrcholů na trase, čili zda budou dodržená jejich časová okna. Při odstranění vrcholu z jeho původní trasy naopak není potřeba kontrolovat nic, protože zde nemůže vzniknout žádný problém. Odstraněním nemůže být překročená kapacita ani nevznikne zpoždění ve zbytku trasy.

Označme vrchol, který přesouváme písmenem *C*, jeho předchůdce na původní trase písmenem *B* a jeho následníka písmenem *E*. Vrchol, za který chceme *C* přesunout, označme A a následníka vrcholu A před přesunem označme *D*. Trasu, ve které vrchol *C* leží před přesunem, budu označovat jako *Trasa2,* a trasu, ve které bude ležet po přesunu, označuji *Trasa1*. Značení je stejné, jako na obrázcích. Přesun vrcholu *C* je validní, pokud jsou splněné následující tři podmínky:

1. $\sum_{X \in Trasa1} X.Demand + C.Demand \leq Capacity$
2. $A.Visit + d(A,C) \leq C.End$
3. $\max\left[d(A,C) + d(C,D) - d(A,D); C.Start + d(C,D) - d.Visit\right] \leq D.Reserve$

První podmínka zajišťuje, že vozidlo na *Trase1* bude mít dostatečnou kapacitu, aby stačilo obsloužit navíc ještě vrchol *C*. Druhá podmínka zajišťuje, že vozidlo stihne vrchol *C* obsloužit dřív, než skončí časové okno u toho vrcholu.

Nejsložitější je třetí podmínka, která kontroluje, že zpoždění, které vznikne na *Trase1* v důsledku vložení nového vrcholu, nebude větší, než je časová rezerva následujícího uzlu (tedy uzlu *D*).

Zpoždění ve vrcholu *D* vzniklé přejezdem přes *C* lze spočítat pomocí výrazu $d(A,C)+d(C,D)-d(A,D)$. Navíc se však může stát, že časové okno ve vrcholu *C* bude začínat až později a vozidlo bude muset ve vrcholu *C* nějakou dobu čekat, než bude moci pokračovat dál. V takovém případě bychom nový čas návštěvy vrcholu *D* spočítali jako $C.Start+d(C,D)$. Když od tohoto výrazu odečteme původní čas návštěvy vrcholu *D*, dostaneme vzniklé zpoždění.

Skutečnou hodnotu zpoždění dostaneme jako maximum z těchto dvou možností a výsledek porovnáváme s hodnotou časové rezervy. Aby byla operace přípustná, musí být rezerva aspoň tak velká, jako vzniklé zpoždění.

Zpoždění posune čas návštěv všech vrcholů na trase, ne jen vrcholu *D*. Časovou rezervu však stačí zkontrolovat u vrcholu *D*. Další vrcholy v pořadí není třeba procházet, protože rezerva je definovaná jako minimum z rezerv všech následujících vrcholů (viz sekci 4.3), takže pokud je rezerva dostatečná u vrcholu *D*, pak je dostatečná i u všech dalších vrcholů na trase.

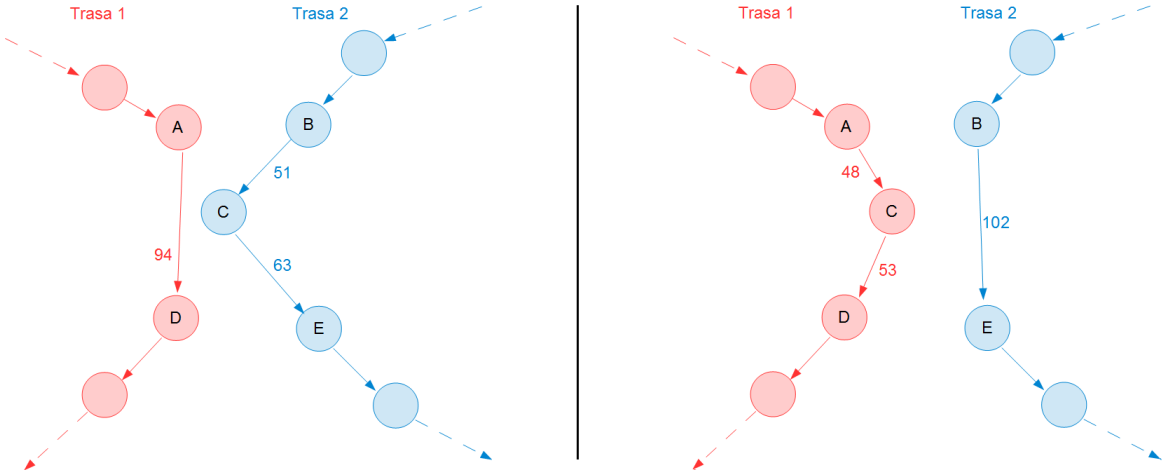
5.2.2 Výpočet zlepšení

Pro validní operace je třeba spočítat jejich zlepšení. V tomto případě je výpočet velmi jednoduchý. Zlepšení lze spočítat podle vzorce:

- *Zlepšení* = $d(A,D)-d(A,C)-d(C,D)+d(B,C)+d(C,E)-d(B,E)$

Jedná se o rozdíl délek hran, na kterých se původní a nové řešení liší. Hrany, které odebíráme, jsou započítané s kladným znaménkem a naopak hrany, které přidáváme, jsou počítané záporně.

Obrázek 5 ukazuje výpočet zlepšení na předchozím příkladu. Úsek z A do *D* se při přesunu prodloužil z 94 na 48+53, čili *Trasa1* je o 7 jednotek delší. Úsek *B*-*E* se zkrátil z původních 51+63 na 102, čili *Trasa2* je o 12 jednotek kratší. Operace tedy přináší zlepšení o velikosti 5.



Obrázek 5. *Výpočet zlepšení operace přesunu.*

Validitu i zlepšení operace lze spočítat velmi rychle. Není k tomu třeba procházet celou trasu, ale stačí k tomu jen informace udržované lokálně.

5.2.3 Aplikace operace

Na základě hodnoty zlepšení je třeba vybrat jednu nebo několik operací, a ty poté aplikovat. Například hladový algoritmus vybírá vždy operaci, která vede k největšímu zlepšení, a tu poté aplikuje, aby získal řešení, které lepší, než původní.

Při aplikaci operace se řešení modifikuje následujícím způsobem:

1. Uprav součet celkové poptávky na trasách:
   - odeber hodnotu *C.Demand* ze součtu poptávky na *Trase2*
   - přidej stejnou hodnotu do součtu poptávky na *Trase1*
2. Uprav aktuální délky obou tras
   - K délce *Trasy1* přičti $d(A,C) + d(C,D) - d(A,D)$
   - K délce *Trasy2* přičti $d(B,E) - d(B,C) - d(C,E)$
3. Odeber vrchol *C* z *Trasy2*:
   - *B.Next nastav na hodnotu E*
   - *E.Prev nastav na hodnotu B*
4. Aktualizuj čas návštěvy u uzlu *E* a u všech dalších uzlů ve zbytku *Trasy2*
   - Čas návštěvy se aktualizuje podle vzorce:
     $X.Visit = min[X.Start; X.Prev.Visit + d(X.Prev, X)]$
5. Aktualizuj časovou rezervu u uzlu *B* a u všech předchozích uzlů v první části *Trasy2*

- Časová rezerva se aktualizuje podle vzorce:
  X.Reserve = min[X.Next.Reserve, X.End-X.Visit]

6. Vlož vrchol *C* do *Trasy1*
   - *A.Next nastav na hodnotu C*
   - *C.Prev nastav na hodnotu a*
   - *C.Next nastav na hodnotu D*
   - *D.Prev nastav na hodnotu C*
7. Aktualizuj čas návštěvy u uzlu *C* a u všech dalších uzlů ve zbytku *Trasy1*
8. Aktualizuj časovou rezervu u uzlu *C* a u všech předchozích uzlů v první části *Trasy1*

Tento postup aktualizuje řešení tak, aby odpovídalo situaci po provedení přesunu, a aktualizuje všechny pomocné proměnné. Aplikace operace je časově náročnější než kontrola validity a výpočet zlepšení, protože je zde nutné projít obě trasy celé, konkrétně v bodech 4, 5 a 7, 8.

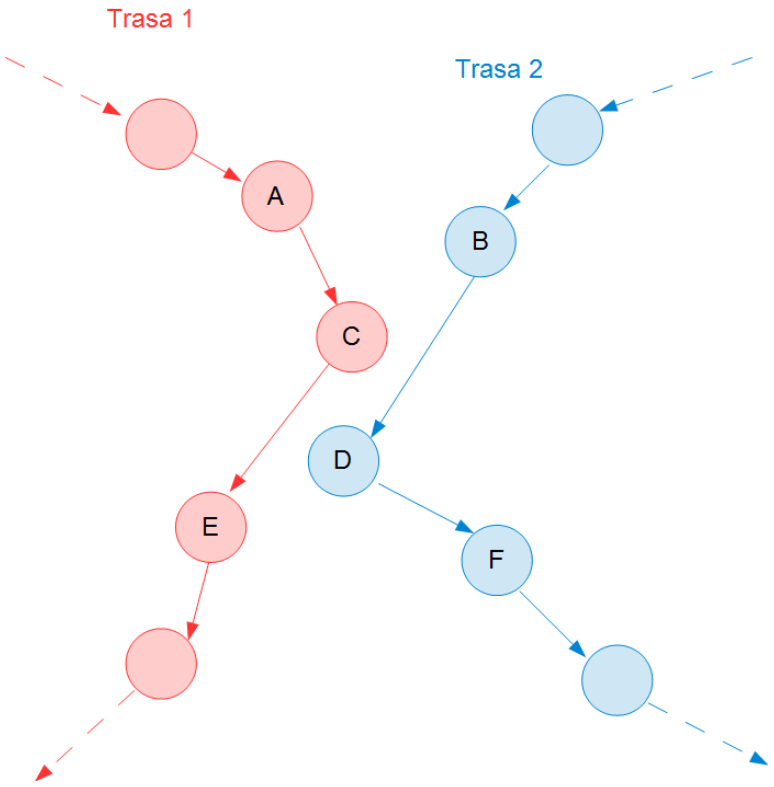
V případě, že vrchol *B* odpovídá depu, je třeba ještě aktualizovat čas příjezdu vozidla do vrcholu *D*. Vrchol *D* se totiž stane prvním vrcholem na trase (po výjezdu z depa), a tedy je možné čas jeho návštěvy nastavit na stejnou hodnotu, jako je začátek jeho časového okna. Viz sekci 4.1.1.

Při generování okolí se zkoušejí všechny dvojice vrcholů *X, Y,* které *neleží* na stejné trase a pro každou dvojici se prozkoumá operace přesunu *X* za *Y*. Pokud je operace validní, spočítá se její zlepšení a je možné ji později použít pro modifikaci. Velikost okolí, které je generované pomocí operace přesunu tak může být až *n*n*, kde *n* je počet cílových míst.

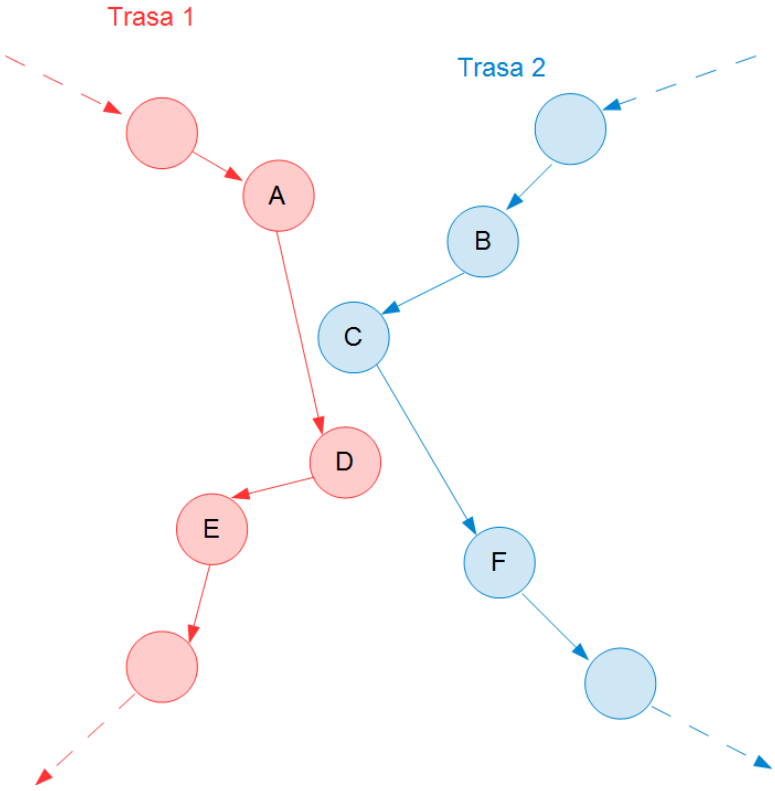
5.3 Operace výměny

Operace je založená na záměně dvou různých vrcholů ve dvou různých trasách. Záměnu vrcholů na stejné trase neprovádíme z důvodů zmíněných výše.

Následující dva obrázky (Obrázek 6 a Obrázek 7) zobrazují ukázku použití operace výměny. Obrázek 6 ukazuje situaci před použitím operace a Obrázek 7 po výměně vrcholů *C* a *D*.



Obrázek 6: Ukázka situace před použitím operace výměny.



Obrázek 7. Ukázka situace po použití operace výměny. Operace byla použitá na vrcholy C a D.

Vrchol *C* leží na *Trase1* a vrchol *D* na *Trase2*. Předchůdce vrcholu *C* na *Trase1* označím A a následníka *C* označím *E*. Předchůdce vrcholu *D* na *Trase2* pak označím *B* a následníka označím *F*. Značení odpovídá situaci na obrázku.

Mohlo by se zdát, že zavedení operace výměny je zbytečné, protože stejného výsledku lze vždy dosáhnout použitím dvou po sobě jdoucích operací přesunu. Například v předchozím příkladu by stačilo nejprve vložit vrchol *C* za vrchol *B,* a poté vložit vrchol *D* za vrchol *A*.

To však není pravda, protože první vložení by mohlo překročit kapacitu vozidla, a tedy vůbec nemusí být proveditelné. Navíc délka trasy se po prvním vložení může výrazně prodloužit a například hladová heuristika by tak tuto možnost vůbec nezkoušela, přestože ve výsledku by celá výměna mohla být výhodná. Tuto intuici potvrzují i provedené experimenty, které ukazují, že použití operace výměn spolu s operací přesunu vede ve většině případů k lepším výsledkům než použití samotné operace přesunu.

5.3.1 Kontrola validity

Výměna vrcholů *C* a *D* je validní, pokud jsou splněné následující podmínky:

1. $\sum_{X \in Trasa1} X.Demand - C.Demand + D.Demand \leq Capacity$
2. $\sum_{X \in Trasa2} X.Demand + C.Demand - D.Demand \leq Capacity$
3. $A.Visit + d(A,D) \leq D.End$
4. $B.Visit + d(B,C) \leq C.End$
5. $\max\left[\max(A.Visit + d(A,D); D.Start) + d(D,E); E.Start\right] - E.Visit \leq E.Reserve$
6. $\max\left[\max(BVisit + d(B,C); C.Start) + d(C,F); F.Start\right] - F.Visit \leq F.Reserve$

Podmínky 1 a 2 kontrolují, že i po prohození vrcholů budou mít příslušná vozidla dostatečnou kapacitu, aby byla schopná zajistit obsluhu. Podmínky 3 a 4 zajišťují, že i po výměně stihnou vozidla obsloužit vrcholy *C* a *D* před koncem jejich časových oken.

Podmínky 5 a 6 jsou složitější a rozeberu je postupně. Nejprve podmínka 5: na levé straně nerovnosti je zpoždění, které vznikne ve vrcholu *E* kvůli výměně. Toto zpoždění se porovnává s časovou rezervou vrcholu *E* podobně jako u operace přesunu a zpoždění se počítá jako *nový čas návštěvy – původní čas návštěvy*.

Nový čas návštěvy se počítá jako čas návštěvy vrcholu *D* + čas potřebný pro přejezd z *D* do *E*. Čas návštěvy vrcholu však nemůže být menší než začátek jeho časového okna, proto se spočítaný výraz ještě porovnává s hodnotou *E.Start* a bere se z nich maximum.

Navíc nový čas návštěvy vrcholu *D*, který je v předchozí rovnici použitý, je třeba nejdříve spočítat, a to jako *čas návštěvy vrcholu A + doba potřebná na přejezd z A do D*. Tato hodnota je poté v případě potřeby zvýšena na minimální hranici *D.Start* pomocí funkce *max* uvnitř výrazu.

Podmínka 6 vykonává stejnou funkci pro *Trasu2*.

5.3.2 Výpočet zlepšení

Zlepšení operace výměny lze spočítat podle vzorce:

- $d(A,D)+d(D,E)+d(B,C)+d(C,F)-d(A,C)-d(C,E)-d(B,D)-d(D,F)$

Opět se jedná o rozdíl délek hran, na kterých se původní a nové řešení liší. Validitu i zlepšení tedy lze i v tomto případě spočítat velmi rychle a není k tomu třeba procházet celou trasu.

5.3.3 Aplikace operace

Při aplikaci operace výměny se řešení modifikuje následujícím způsobem:

1. Aplikuj operaci přesunu: přesuň vrchol *D* za vrchol *A*.
2. Aplikuj operaci přesunu: přesuň vrchol *C* za vrchol *B*.

Aplikace operace přesunu funguje podle postupu popsaného v sekci 5.2.3 na straně 33.

Sice jsem zmínil, že pro účely výpočtu validity není možné nahradit operaci výměny dvěma operacemi přesunu, ale pro samotné provedení modifikačních činností je tato náhrada možná. Navíc se tímto způsobem popis aplikace operace značně zjednodušuje.

Při generování okolí se zkoušejí všechny dvojice vrcholů *X, Y,* které neleží na stejné trase a pro každou dvojici se prozkoumá operace výměny *X* za *Y*. Pokud je operace validní, spočítá se její zlepšení a je možné ji později použít pro modifikaci. Vzhledem k tomu, že výměna je symetrická, zkoušejí se pouze varianty, kde *X.ID* < *Y.ID*. Velikost okolí, které je generované pomocí operace výměny, tak může být až *n*n/2*, kde *n* je počet cílových míst.

5.4 Operace optimalizace tras

Tato operace funguje jinak než předchozí dvě. Negeneruje všechny možné výsledky, ale pouze jeden, a to ten nejlepší. Operace funguje tak, že zafixuje přiřazení míst k trasám, čili seznam míst, která každé z vozidel obslouží, se nezmění, ale pro každé vozidlo se najde nejkratší způsob, jak tato místa obsloužit.

Jedná se opět o modifikační operaci, tedy vstupem bude validní řešení. V prvním kroku se zjistí, která místa vozidlo obsluhuje. Poté se spočítá, v jakém pořadí je třeba tato konkrétní místa navštívit, aby celková délka byla co nejmenší (samozřejmě s přihlédnutím k podmínkám časových oken). Pokud nalezený způsob je lepší než stávající trasa, pak se tato trasa nahradí nově nalezeným okruhem. Takto se postupně přepočítají všechny trasy.

Operace je velmi podobná heuristice *k-OPT*. Optimalizaci trasy si lze představit tak, že odstraníme z aktuálního řešení všechny hrany příslušející jedné konkrétní trase a poté se snažíme tyto hrany doplnit tak, aby celková délka byla co nejmenší. Při doplňování hran je třeba brát ohled na požadavky časových oken u jednotlivých cílových míst.

Naopak požadavky na kapacitní omezení zde není třeba brát v úvahu. Množina míst, které vozidlo obslouží, se nemění, a tedy ani součet poptávky se nezmění. Vzhledem k tomu, že v původním validním řešení ležela tato místa na jedné trase, tak je zaručené, že kapacitní omezení je splněné.

Heuristika tedy funguje tak, že rozdělí problém na jednotlivé trasy a na každé trase vyřeší problém *TSPTW* optimálně. Vzhledem k tomu, že řešení velkých instancí *TSPTW* je časově velmi náročné, omezil jsem délku trasy, která se bude přepočítávat číslem 10. Pokud bude na trase více než 10 míst (včetně depa), tak tato heuristika nebude nic počítat a tuto trasu ponechá beze změn. U kratších tras najde optimální řešení pomocí algoritmu, který vyzkouší všechny možné validní způsoby, jak lze daná místa projet. Výpočetní čas potřebný pro přepočítání trasy o 10-ti místech je únosný a v experimentech se ukazuje, že trasy delší než 10 se vyskytují jen velmi zřídka.

Konkrétní algoritmus zde nebudu popisovat. Jedná se o standardní rekurzivní postup pro generování permutací, který jsem doplnil o průběžnou kontrolu validity.

Použití této operace vygeneruje vždy okolí velikosti 1. Validitu ani zlepšení tohoto nového řešení není třeba ověřovat, protože z povahy použitého algoritmu vyplývá, že získané řešení je vždy validní a vždy aspoň tak dobré, jako bylo původní řešení.

V experimentech používám operaci *optimalizace tras* většinou až jako poslední krok. Po skončení výpočtu všech použitých heuristik se jedenkrát spustí *optimalizace tras*, která zajistí, že každá trasa bude obsloužena co nejefektivněji.

Nutno podotknout, že optimální přepočítání jednotlivých tras nijak negarantuje optimalitu celého řešení. Mohlo se stát, že přiřazení míst k jednotlivým vozidlům v původním řešení bylo nevhodné a přepočítání jednotlivých tras už toto přiřazení nemění.

6 Popis programu

Popsané řešící algoritmy jsem implementoval v podobě desktop-aplikace. Tato aplikace obsahuje základní funkcionalitu potřebnou pro řešení VRPTW a několik dodatečných funkcí užitečných při testování a zobrazování výsledků. V této kapitole představím funkce programu a jeho ovládání a stručně popíšu také objektový návrh a použité datové struktury.

Aplikace je napsaná v jazyce C# s využitím knihovny WinForms a je určená pro platformu Windows. Ke spuštění aplikace je třeba, aby na počítači byl nainstalovaný .NET verze 4 nebo vyšší, který je standardní součástí operačního systému Windows od verze Window 7 ServicePack1 (a dále ve všech novějších verzích).

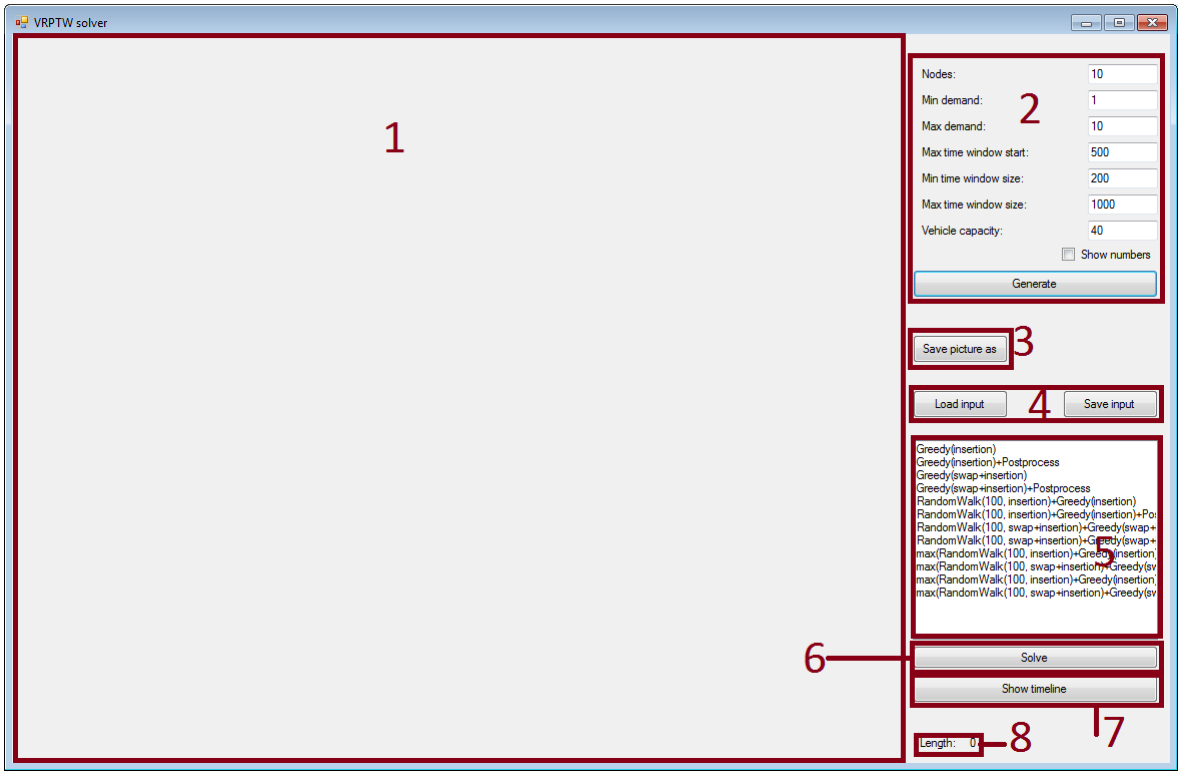
6.1 Použití programu

Aplikace se spouští pomocí souboru *VRPTW.exe*, který se nachází ve složce *Aplikace*. Žádná instalace není nutná. Zdrojové kódy i uživatelské rozhraní programu je psané anglicky, aby byl program snadno přístupný širší skupině uživatelů.

Aplikace umožňuje:

- vygenerovat a vykreslit pseudonáhodné zadání se zadanými parametry
- zvolit některý z řešících algoritmů popsaných v této práci
- vyřešit zadání problému pomocí zvoleného algoritmu
- vypsat nalezené řešení včetně jeho kvality
- vykreslit trasy jednotlivých vozidel do mapy
- vykreslit časový plán jízdy pro jednotlivá vozidla s ohledem na časová okna zákazníků.

Po spuštění aplikace se otevře konzole a okno s několika tlačítky (viz Obrázek 8. Jednotlivé ovládací prvky jsou označené čísly.). K ovládání programu slouží okno s ovládacími prvky. Konzole se využívá pouze pro vypisování výsledků. Při řešení problémů většího rozsahu, například s více než 100 místy, doporučuji okno s konzolí minimalizovat, protože dlouhé výpisy můžou výrazně zpomalovat proces řešení.



Obrázek 8. Uživatelské rozhraní programu.

Pro generování nového pseudonáhodného zadání lze využít tlačítko *Generate* v sekci 2. Při generování se využívají parametry v sekci 2, které je třeba před samotným generováním vhodně nastavit. Význam jednotlivých parametrů je následující:

- *Nodes:* celkový počet cílových míst. Cílová místa i depo se rozmístí na mapu náhodně.
- *Min demand* a *Max demand*: minimální a maximální požadavek cílového místa. Požadavky jsou měřené počtem kusů, které je třeba do cílového místa dopravit z depa. Pro každé cílové místo se jeho požadavek zvolí rovnoměrně náhodně z intervalu <*Min demand*, *Max demand*>.
- *Max time window start:* nejpozdější možný začátek časového okna u cílových míst. Pro každé cílové místo se začátek jeho časového okna zvolí rovnoměrně náhodně z intervalu <0, *Max time window start*>.
- *Min time window size* a *Max time window size:* minimální a maximální velikost časového okna u cílových míst. Pro každé cílové místo se konec jeho časového okna zvolí rovnoměrně náhodně z intervalu <*TimeWindowStart* + *MinTimeWindowSize*, *TimeWindowStart* + *MaxTimeWindowSize*>, kde *TimeWindowStart* je začátek příslušného časového okna.
- *Vehicle capacity*: kapacita vozidla měřená počtem kusů, které může maximálně vézt.

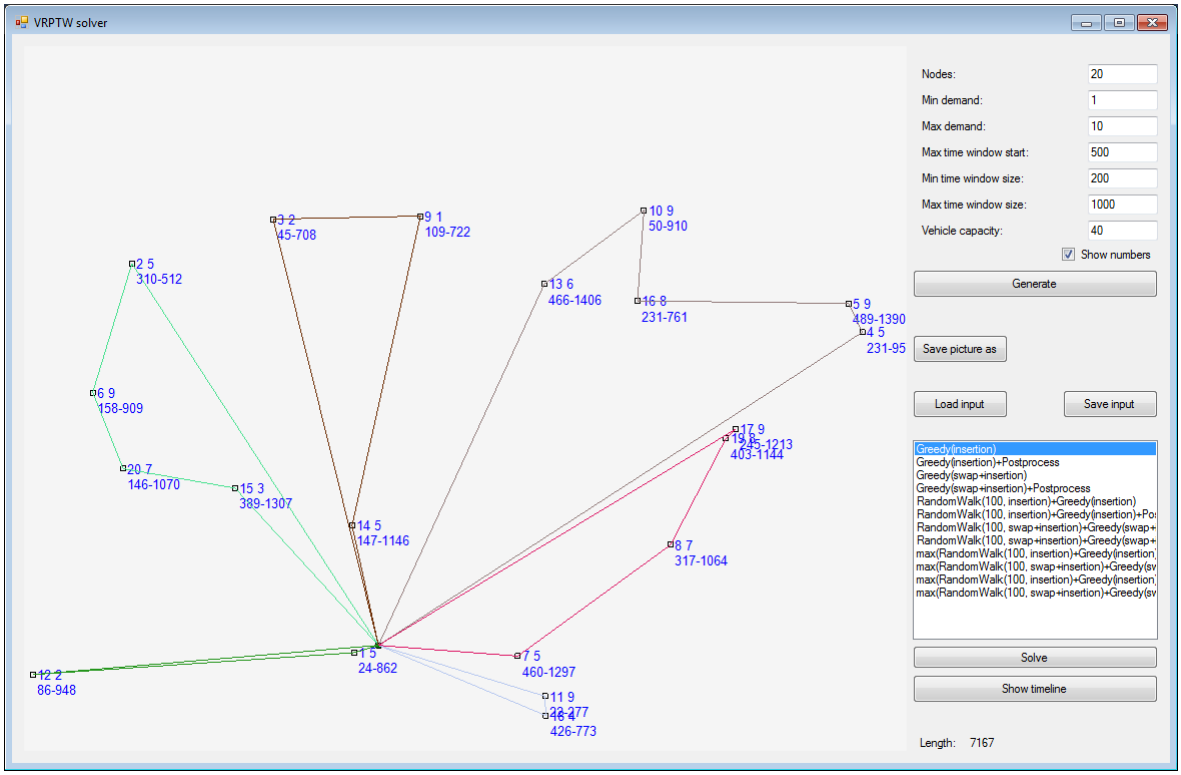
Po vygenerování se vykreslí rozmístění depa a cílových míst na mapě v oblasti 1. Depo je označené zeleným čtverečkem. Při zapnutí volby *Show numbers* se ke každému cílovému

místu vykreslí také dodatečné informace ohledně jeho identifikačního čísla, požadavků a začátku a konce časového okna. Viz například Obrázek 9. První číslo označuje identifikátor místa a druhé jeho požadavek na velikost dodávky. Čísla na dalším řádku pak označují začátek a konec příslušného časového okna. Veškeré informace o zadání se vypíší i do konzole.

Je také možné zadání vstupního problému načíst ze souboru pomocí tlačítka *Load input* v sekci 4. Ve stejné sekci se nachází také tlačítko *Save input*, které umožňuje vygenerované zadání uložit pro pozdější použití. Zde je možné načítat pouze zadání, která byla dříve vygenerovaná a uložená pomocí tohoto programu.

Po získání zadání problému, ať už pomocí generátoru, nebo načtením ze souboru, je možné toto zadání vyřešit. Nejprve je třeba zvolit jeden z nabídnutých algoritmů v sekci 5, a poté kliknout na tlačítko *Solve*, v obrázku pod číslem 6. Přesný popis jednotlivých řešících algoritmů poskytuje kapitola 7.

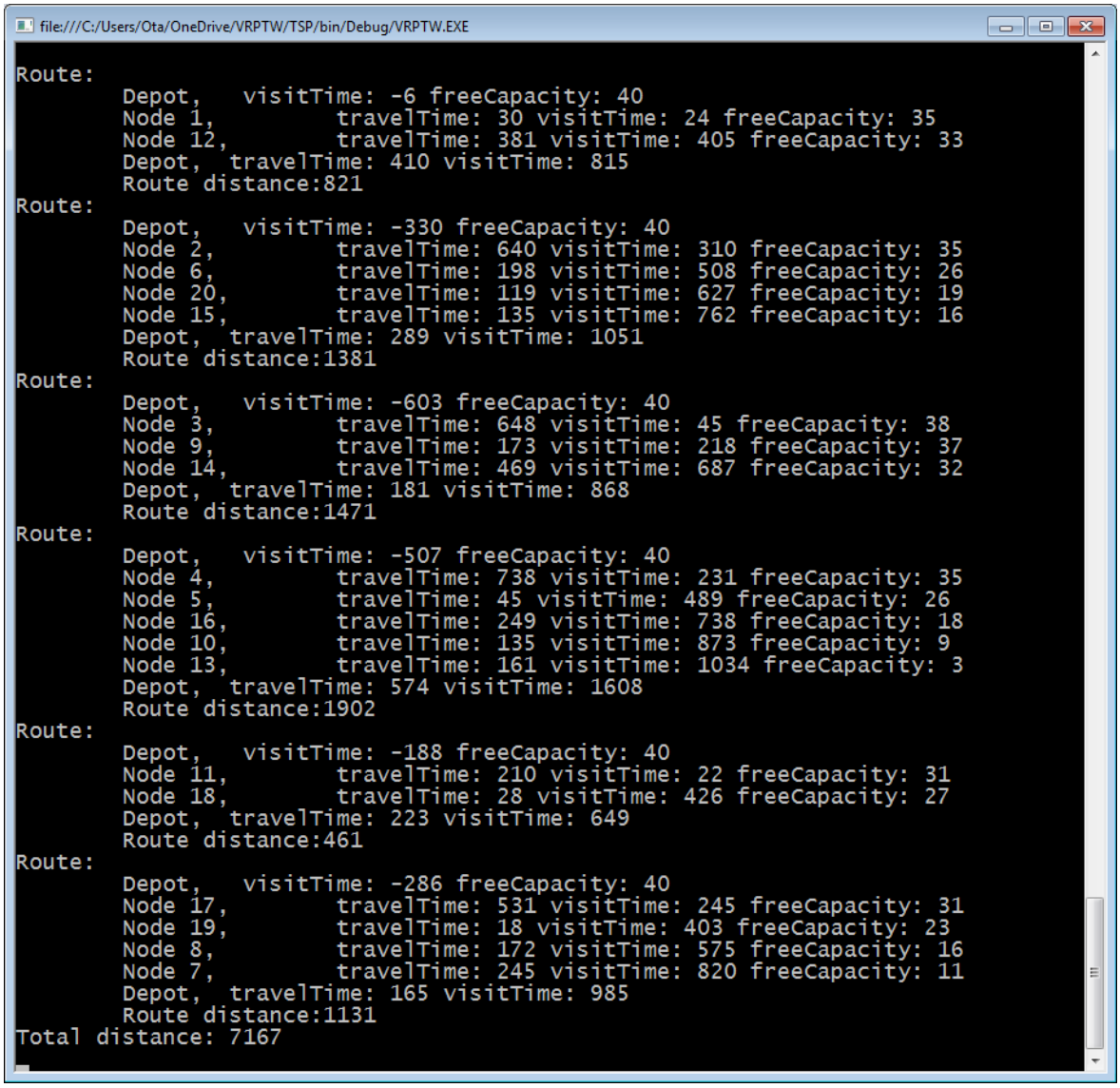
Po doběhnutí řešícího algoritmu se do mapy zakreslí řešení problému. Trasy jednotlivých vozidel jsou znázorněny barevně (viz Obrázek 9).



Obrázek 9. Ukázka vizualizace nalezeného řešení.

Do konzole se poté vypíše přesný popis jednotlivých tras, jak ukazuje Obrázek 10. Popis trasy začíná slovem *Route*, poté následuje seznam navštívených míst. První a poslední místo na trase je vždy depo. U každého místa je popsané jeho identifikační číslo, doba potřebná na přejezd do tohoto místa z předchozího, čas návštěvy a zbývající volná kapacita vozidla po návštěvě příslušného místa. Lze zde snadno zkontrolovat, že čas návštěvy leží

uvnitř daného časového okna a volná kapacita vozidla nikdy neklesne pod nulu. U každé trasy je uvedená její délka a na konci výpisu je uveden údaj o celkové délce všech tras. Údaj o celkové délce se zobrazuje také v hlavním okně v oblasti 8.



Obrázek 10. Popis nalezených tras v konzoli.

Po vyřešení problému je možné pomocí tlačítka *Show timeline* (oblast 7) nechat zobrazit jízdní řád všech vozidel – viz Obrázek 11. Trasa vozidla je označená černou čarou, každý řádek odpovídá jednomu místu, popřípadě depu. Vodorovná osa reprezentuje čas. U každého cílového místa je zeleným obdélníkem označeno jeho časové okno a návštěva místa je označená žlutým kolečkem.

U prvního a posledního místa, které odpovídají depu, je vyznačen čas odjezdu vozidla z depa respektive čas návratu do depa. U vnitřních míst je vyznačen čas návštěvy, a případně i čas příjezdu, pokud se liší od času návštěvy. To se může stát, pokud vozidlo dorazí do uzlu dřív, než je začátek příslušného časového okna, jak je vidět například u uzlu 5. V takovém případě musí vozidlo s návštěvou čekat až na začátek časového okna.

V záhlaví je u každé trasy také uvedena její délka a součet poptávky zákazníků na trase.

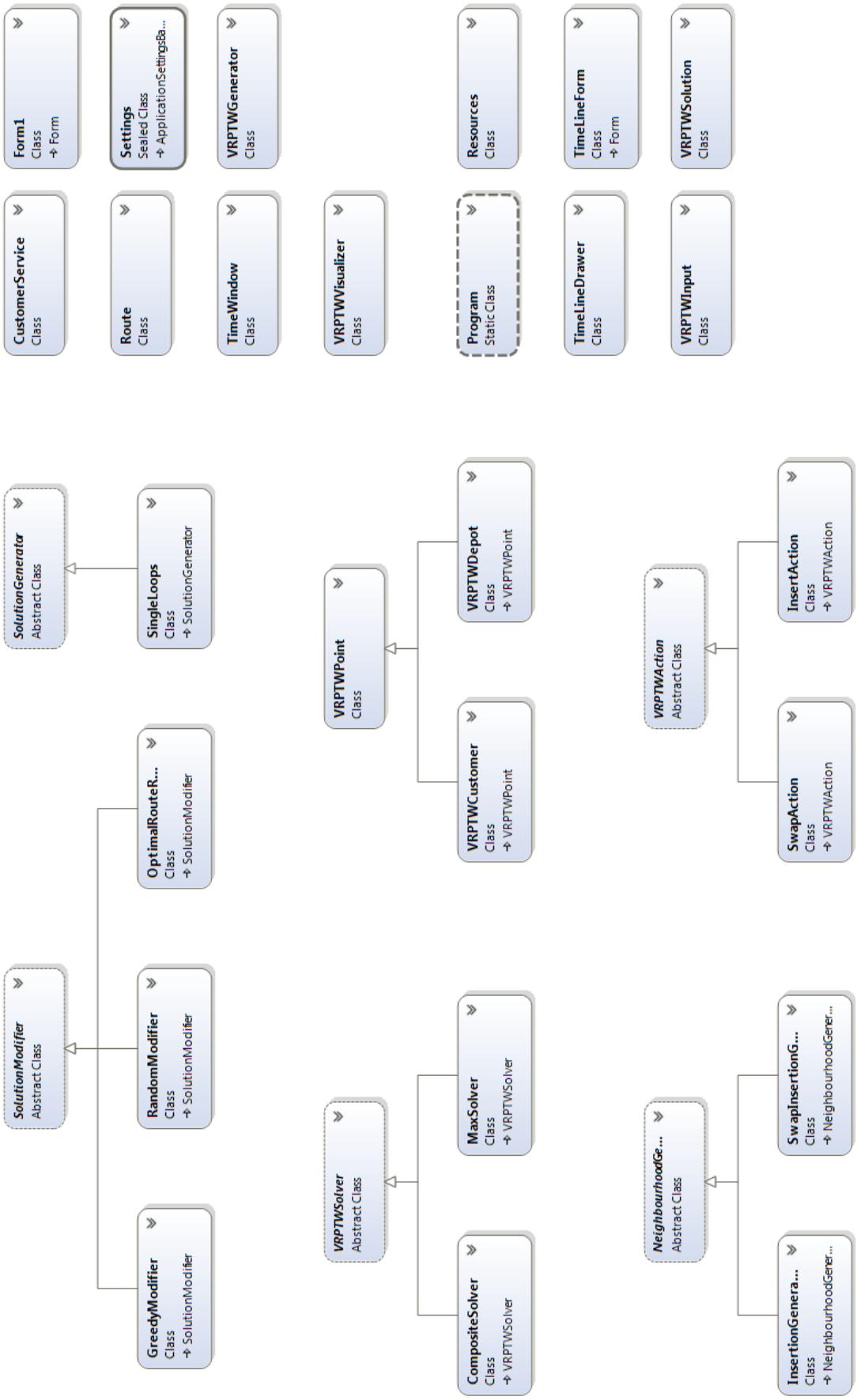
Obrázek 11. Ukázka vizualizace jednotlivých tras.

6.2 Architektura programu

Program je psaný standardním stylem s využitím objektově orientovaného programování. Obrázek 12 ukazuje diagram všech použitých tříd, šipky označují případnou dědičnost. Třídy *VRPTWInput* a *VRPTWSolution* reprezentují zadání a řešení problému. První z těchto tříd udržuje tzv. statické informace o problému, druhá pak dynamické informace (viz sekce 4.2). Třída *VRPTWGenerator* zajišťuje generování nového pseudonáhodného řešení.

Vizualizaci zajišťuje třída *VRPTWVisualizer*, která vykresluje pozice míst na mapě a trasy vozidel, a třída *TimeLineDrawer*, která vykresluje jízdní řád jednotlivých vozidel po kliknutí na tlačítko *Show timeline*.

Třída *SolutionModifier* reprezentuje způsob, jak lze řešení lokálně pozměnit pomocí operace. Má tři potomky: *GreedyModifier* odpovídá hladovému algoritmu a *RandomModifier* odpovídá náhodné procházce (viz sekci 4.1.2). *OptimalRoutesModifier* pak odpovídá operaci optimalizace tras popsané v sekci 5.4.



Obrázek 12. Diagram všech tříd použitých při návrhu programu. Šipky ukazují dědičnost.

Jednotlivé typy operací jsou pak reprezentované třídami *InsertAction* (operace přesunu, viz sekce 5.2) a *SwapAction* (operace výměny, viz sekce 5.3).

Třída *VRPTWPoint* reprezentuje bod na mapě, což může být buď depo (potomek *VRPTWDepot*), nebo zákazník (potomek *VRPTWCustomer*).

Třída *VRPTWSolver* představuje řešící algoritmus. Ten může být dvojího druhu: *CompositeSolver* reprezentuje algoritmus složený z několika jednodušších algoritmů, kde výstup prvního algoritmu je brán jako vstup dalšího atd. Tímto způsobem je reprezentován například řešič, který kombinuje náhodnou procházku, hladový algoritmus a nakonec udělá optimalizaci tras (viz experimenty). Druhým typem pak je *MaxSolver*, který umožňuje použít jednodušší řešič opakovaně a vzít nejlepší řešení ze všech vykonaných běhů. Takto implementované řešiče dosahují v experimentech nejlepších výsledků. (Viz experimenty).

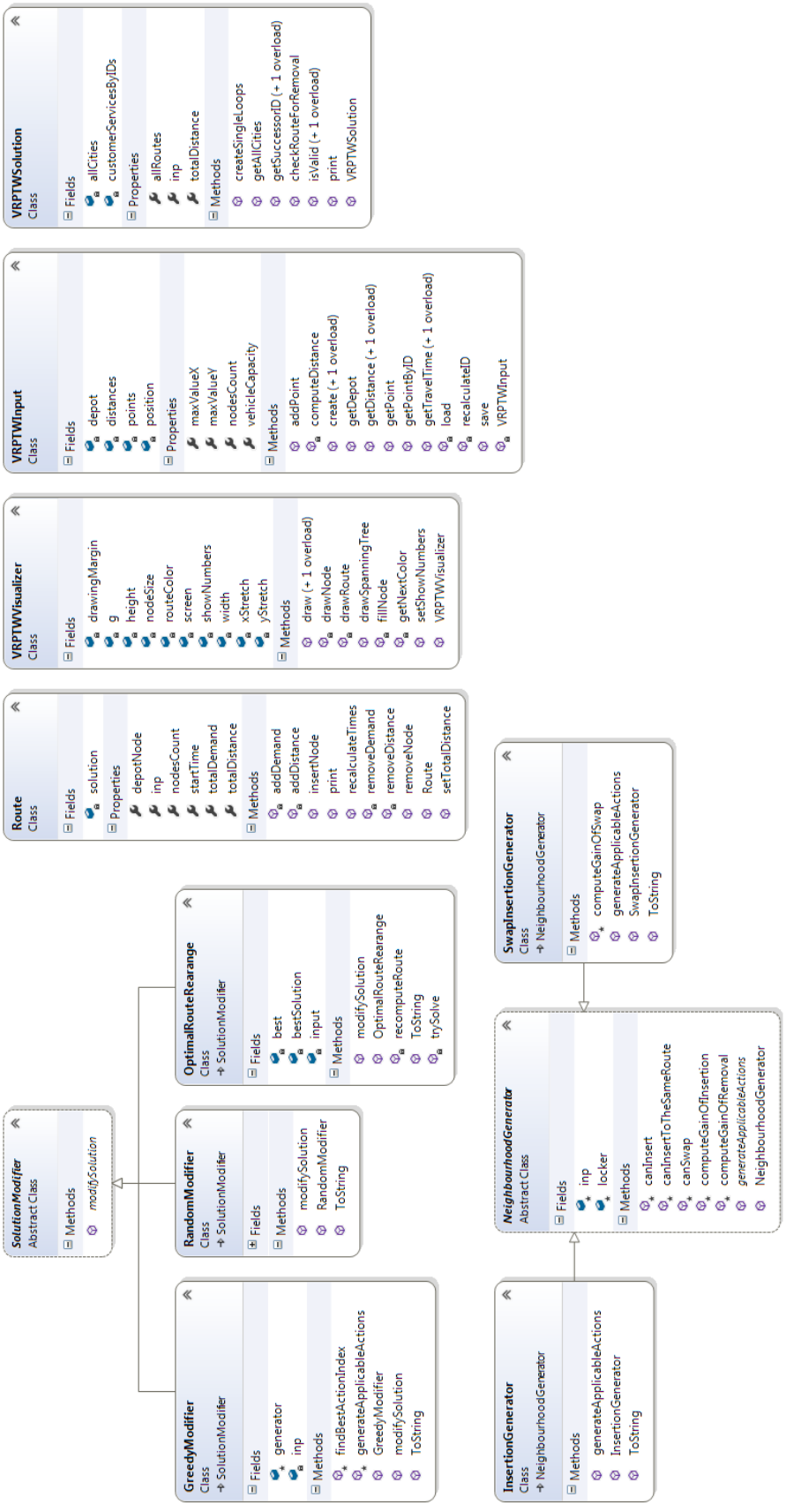
Třída *NeighbourhoodGenerator* implementuje generování okolí daného řešení, jak bylo popsané v sekci 4.4.2. Její dva potomci implementují generování pomocí dvou operací: přesunu a přesunu+výměny.

Třída *Route* pak reprezentuje trasu jednoho vozidla. Zde je uložená většina dodatečných informací zmíněných v sekci 4.2. Cesta vozidla je reprezentovaná jako dvojitě provázaný spojový seznam, kde každý prvek v posloupnosti má odkaz na svého předchůdce a následníka.

Podrobnější popis tříd a fungování jednotlivých algoritmů lze získat pohledem do zdrojových kódů.

Obrázek 13 ukazuje detailněji vlastnosti několika vybraných tříd. Oddíl *Fields* popisuje použité datové položky, *Properties* pak označuje položky určené jen pro čtení a oddíl *Methods* shrnuje názvy přidružených metod.

Oba diagramy byly vygenerované ze zdrojových kódů pomocí vývojového nástroje *Microsoft Visual Studio.*



Obrázek 13. Podrobnější popis některých tříd.

7 Experimentální srovnání

Pro ověření funkčnosti navržených heuristik jsem provedl experimentální srovnání, jehož výsledky v této kapitole představím. Smyslem provedených experimentů bylo otestovat praktické chování navržených algoritmů (například jak kvalitní řešení dokážou najít, jak dlouho reálně běží a podobně) a také porovnat jednotlivé algoritmy mezi sebou, zejména s ohledem na kvalitu nalezeného řešení a čas běhu.

Experimenty běžely na počítači *HP Z840* s procesorem *Intel® Xeon® E5-2650 v4 (2.2 GHz, 30 MB cache, 12 jader)* s operační pamětí *16 GB DDR4*. Všechny algoritmy používaly jen jedno jádro.

7.1 Testované algoritmy

S využitím heuristiky přesunu a heuristiky výměn popsaných v předchozích částech, jsem navrhl několik konkrétních způsobů, jak řešit instance problému VRPTW. Jednotlivé algoritmy fungují následovně:

- Hladový algoritmus využívající přesuny.

  Jedná se o postup popsaný v sekci 4.1.2. Nejprve se vytvoří počáteční řešení, a poté se opakovaně aplikují operace přesunu. Vždy se vybírá taková operace, která co nejvíce zlepší kvalitu řešení. Proces končí ve chvíli, kdy žádný další přesun už kvalitu řešení nezlepší. V programu je tento postup označený jako *Greedy(insertion)* a stejné označení budu používat i zde.

- Hladový algoritmus využívající přesuny a prohození.

  Jedná se opět o hladový algoritmus. Rozdíl oproti předchozímu bodu spočívá v tom, že při generování okolí se využívají jak operace přesunu, tak operace prohození. Z takto vygenerované množiny operací se poté opět vybere taková operace, která vede k největšímu zlepšení kvality. Postup končí ve chvíli, kdy žádná operace už nevede ke zlepšení. Tento algoritmus budu označovat jako *Greedy(swap+insertion)*.

  Hladový algoritmus, který by využíval pouze operace výměny, nepoužívám, protože v počátečním řešení, kde ke každém zákazníkovi jede samostatné vozidlo, by výměny neměly žádný efekt.

- Náhodná procházka fixní délky a poté hladový algoritmus

  Při experimentech se ukázalo, že hladový algoritmus začínající v počátečním stavu (kde ke každému zákazníkovi jede samostatné vozidlo), nemusí dávat tak dobré výsledky, jako když se tento algoritmus pustí z nějakého jiného stavu. Proto jsem zkombinoval náhodnou procházku s hladovým algoritmem.

Náhodná procházka vygeneruje náhodný stav (čili náhodné validní řešení) a z něj se poté aplikuje hladový algoritmus stejně jako v předchozích bodech.

Náhodná procházka funguje podle postupu popsaného v sekci 4.1.2. Vygeneruje se okolí v závislosti na použitých operacích a poté se z množiny operací zvolí jedna náhodně bez ohledu na její zlepšení. Tento postup se opakuje tolikrát, jaká je zamýšlená délka procházky. V programu i v experimentech používám hodnotu 100.

Popsaný postup používám ve dvou verzích: buď pouze s operátorem přesunu, nebo s operátory přesunu a prohození. Při náhodné procházce používám stejnou množinu operátorů jako při následném hladovém algoritmu. Tyto dvě verze označuji jako *RandomWalk(100, insertion) + Greedy(insertion)* a *RandomWalk(100, swap + insertion) + Greedy(swap + insertion).*

Vzhledem k tomu, že tyto algoritmy mohou při opakovaném běhu vracet různé výsledky, provedl jsem 10 pokusů a zveřejňuji jejich průměr.

- Opakované použití náhodné procházky spolu s hladovým algoritmem.

  Jak už jsem zmínil, postup popsaný v předchozím bodě je randomizovaný, čili různé běhy algoritmu mohou vést na různě kvalitní řešení. Nabízí se tedy myšlenka tento postup několikrát zopakovat a ze získaných výsledků vzít ten nejlepší. V programu jsem tuto myšlenku implementoval.

  Předchozí postup opakuji 10-krát a vracím nejlepší dosažený výsledek. V programu i v experimentech označuji takový algoritmus jako *max(RandomWalk(100, insertion) + Greedy(insertion), 10-times)* a *max(RandomWalk(100, swap + insertion) + Greedy(swap + insertion), 10-times).*

  Tento algoritmus je stále randomizovaný, ale vzhledem k tomu, že bere nejlepší dosaženou hodnotu z 10-ti, měly jeho výstupy už být dostatečně stabilní. Opakované běhy tedy v tomto případě neprovádím.

- Optimalizace tras

  Všechny předchozí algoritmy lze rozšířit o optimalizaci tras, popsanou v sekci 5.4. V experimentech používám u každého algoritmu dvě verze: s optimalizací tras a bez. Verzi s optimalizací tras označuji tak, že za názvem je přípona *+Postprocess*.

Celkem tedy testuji 12 algoritmů:

1. *Greedy(insertion)*
2. *Greedy(insertion)+Postprocess*
3. *Greedy(swap+insertion)*
4. *Greedy(swap+insertion)+Postprocess*
5. *RandomWalk(100, insertion)+Greedy(insertion)*
6. *RandomWalk(100, insertion)+Greedy(insertion)+Postprocess*

7. *RandomWalk(100, swap+insertion)+Greedy(swap+insertion)*
8. *RandomWalk(100, swap+insertion)+Greedy(swap+insertion)+Postprocess*
9. *max(RandomWalk(100, insertion)+Greedy(insertion), 10-times)*
10. *max(RandomWalk(100, swap+insertion)+Greedy(swap+insertion), 10-times)*
11. *max(RandomWalk(100, insertion)+Greedy(insertion)+Postprocess, 10-times)*
12. *max(RandomWalk(100, swap+insertion)+Greedy(swap+insertion)+Postprocess, 10-times)*

Algoritmus *max(RandomWalk(100, insertion)+Greedy(insertion)+Postprocess, 10-times)* funguje tak, že při každém z deseti pokusů se provede náhodná procházka, hladový algoritmus i optimalizace tras a z výsledků se bere ten nejlepší. Jinou možností by bylo provést deset pokusů náhodné procházky a hladového algoritmu, z nich vzít nejlepší výsledek a teprve potom provést optimalizaci tras. Optimalizace by se tedy prováděla jen jednou. Tuto možnost však nepoužívám, protože optimalizace tras je poměrně rychlá.

7.2 Testovací data

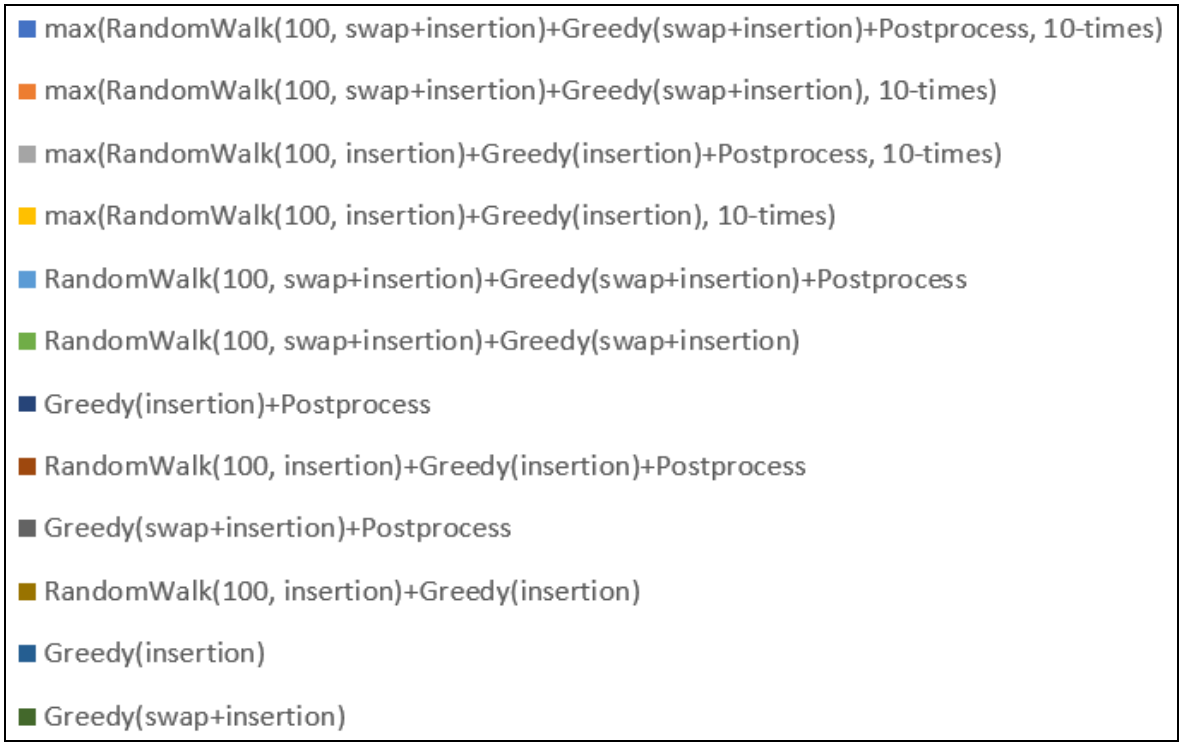
Pro testování jsem vygeneroval celkem 18 zadání problému pomocí náhodného generátoru, který je součástí programu. Jednotlivé příklady se liší celkovým počtem cílových míst, který se pohybuje od 50 do 300. Ostatní parametry byly zafixované na hodnotách:

- *Min demand = 1*
- *Max demand = 10*
- *Max time window start = 500*
- *Min time window size = 200*
- *Max time window size = 1000*
- *Vehicle capacity = 40*

Parametr *Nodes*, tedy *Celkový počet míst* nabývá u testovacích příkladů hodnot 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 250 a 300. Použité testovací příklady jsou přiložené ve složce *Experimenty* a je možné je do programu načíst pomocí tlačítka *Load input*. Všechny algoritmy byly testované na stejných datech. Tabulka se všemi výsledky je ve složce *Experimenty* také přiložená.

7.3 Výsledky

Výsledky budu prezentovat formou grafů. Jednotlivé algoritmy jsou v grafech odlišeny barevně podle legendy, kterou ukazuje Obrázek 14. Vzhledem k tomu, že legenda je poměrně rozměrná, není v dalších grafech už znovu zobrazovaná. Pořadí algoritmů v legendě je určené jejich průměrnými výsledky (viz Obrázek 19) a ve stejném pořadí jsou algoritmy prezentované i v následujících grafech.



Obrázek 14. Legenda k následujícím grafům.

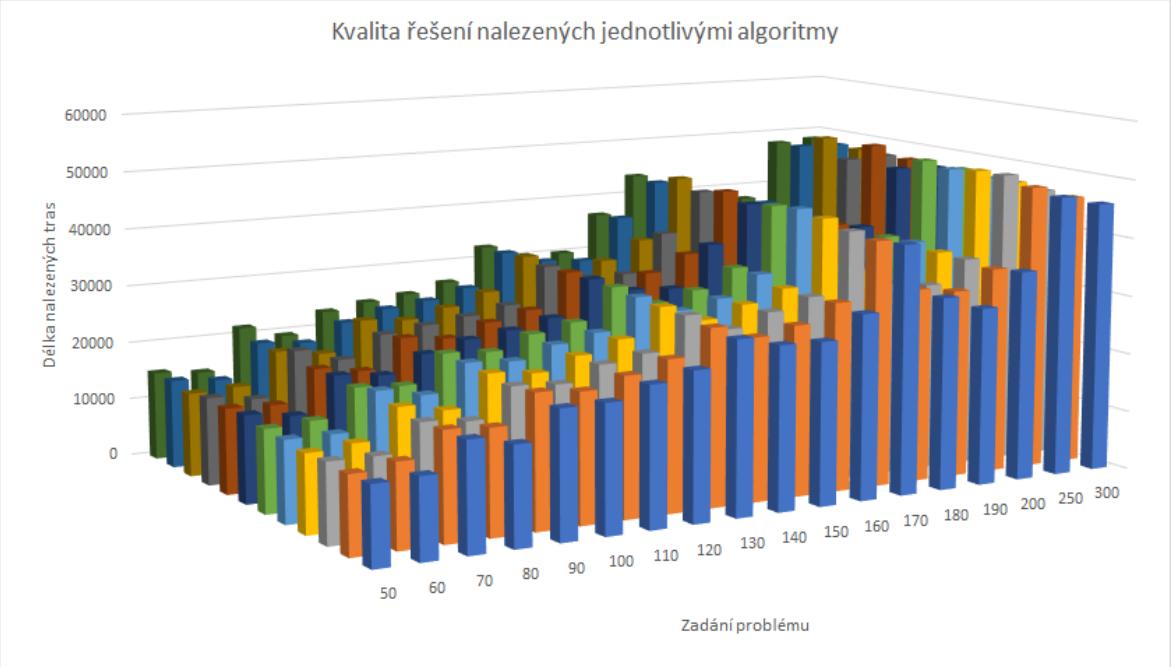
7.3.1 Kvalita řešení

Obrázek 15 ukazuje kvalitu řešení nalezených jednotlivými algoritmy na testovacích datech. Vodorovná osa odpovídá testovacím příkladům a barvy odlišují jednotlivé algoritmy podle legendy uvedené výše. Výška sloupce pak odpovídá kvalitě řešení, čili součtu délky tras. Menší sloupec tedy znamená lepší řešení.

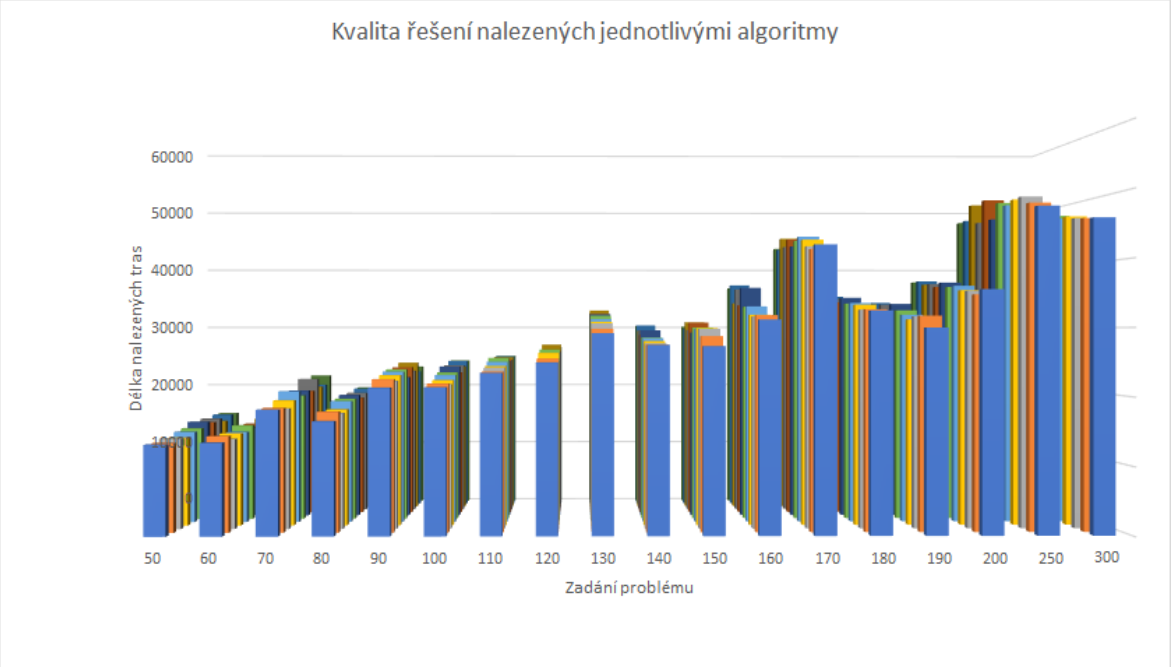
Nejlepších výsledků na většině testovacích příkladů dosahuje algoritmus *max(RandomWalk(100, swap + insertion) + Greedy(swap + insertion) + Postprocess, 10-times)*, zobrazený světle modrou barvou. Jedná se o řadu zobrazenou nejblíže k pozorovateli.

Lépe je možné výsledky posoudit z pohledu, jež ukazuje Obrázek 16. Zde je dobře vidět, že řada nejblíže k pozorovateli je ve většině případů nejnižší.

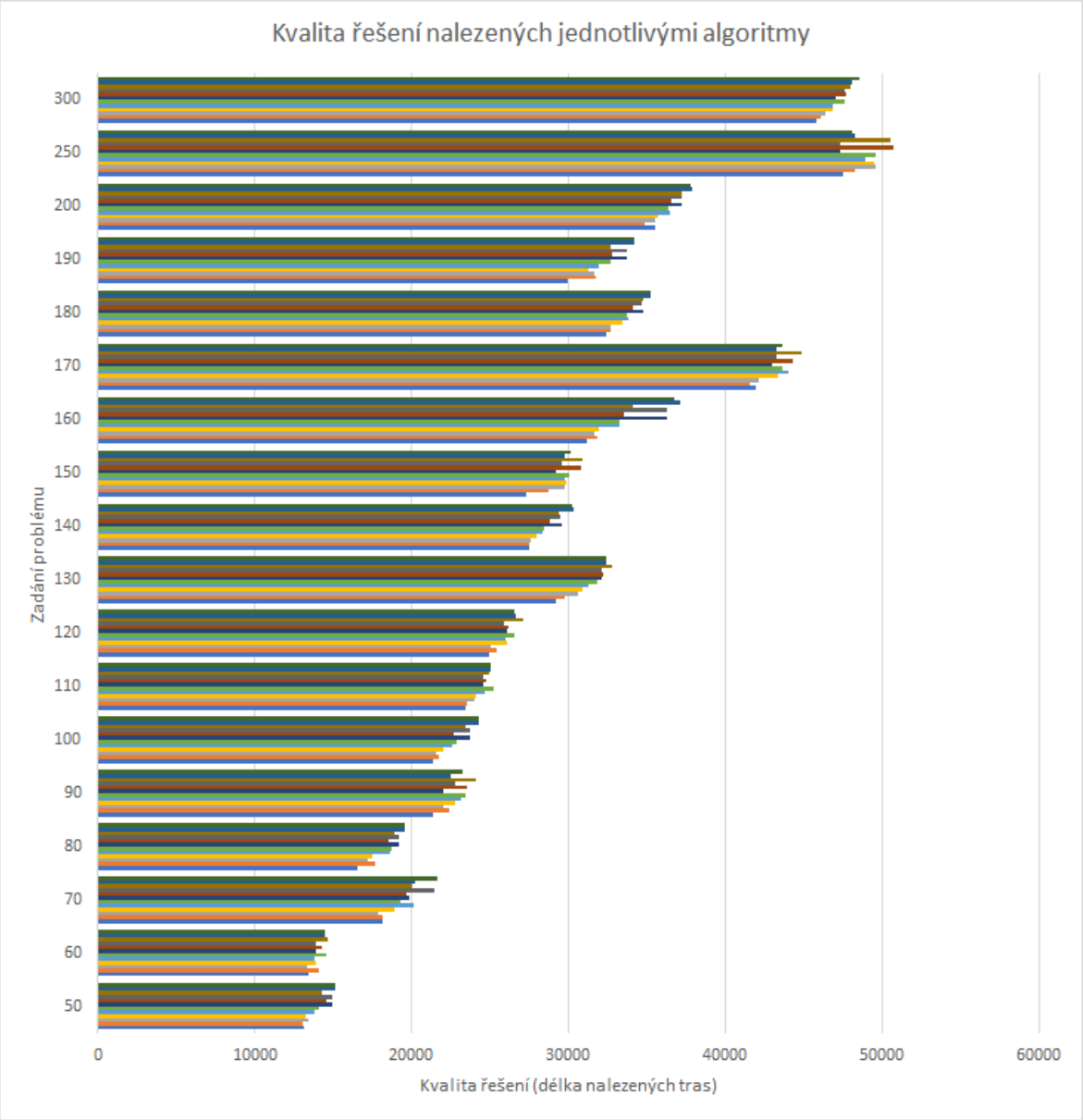
Přehled všech výsledků ukazuje Obrázek 17 a detail výsledků na dvou největších instancích pak zobrazuje Obrázek 18. Zde je vidět zajímavý fakt. Ačkoliv na většině instancí je algoritmus označený světle modrou nejlepší, u problému 250 ho překonávají dva jiné algoritmy, a to sice *Greedy(swap + insertion) + Postprocess a Greedy(insertion) + Postprocess*. Tento jev je tím víc překvapivý, že tyto dva algoritmy na většině instancí dávaly jedny z nejhorších výsledků.



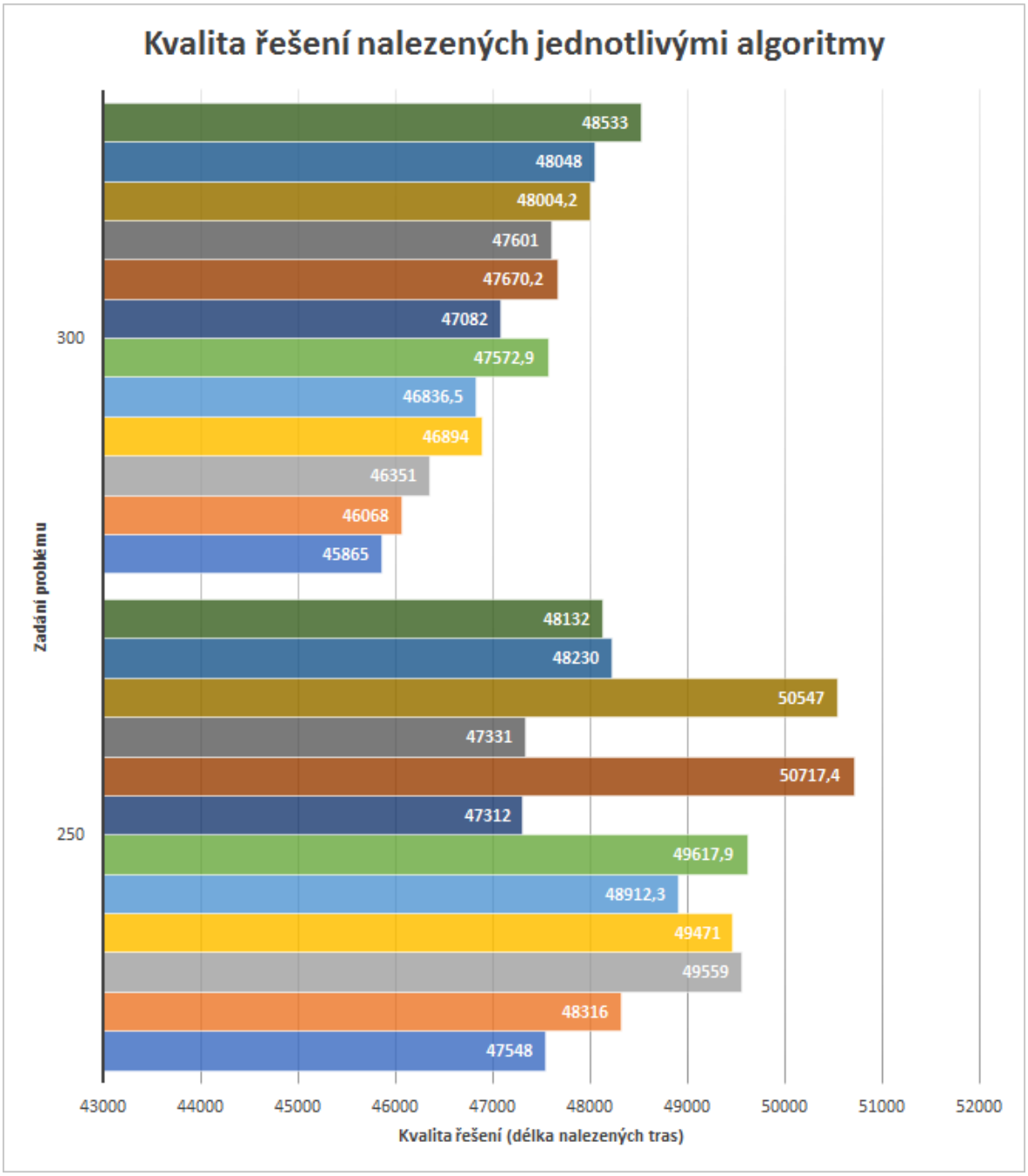
Obrázek 15. Kvalita řešení nalezených jednotlivými algoritmy, 3D graf.



Obrázek 16. Kvalita řešení nalezených jednotlivými algoritmy, 3D graf.



Obrázek 17. Kvalita řešení, 2D graf.

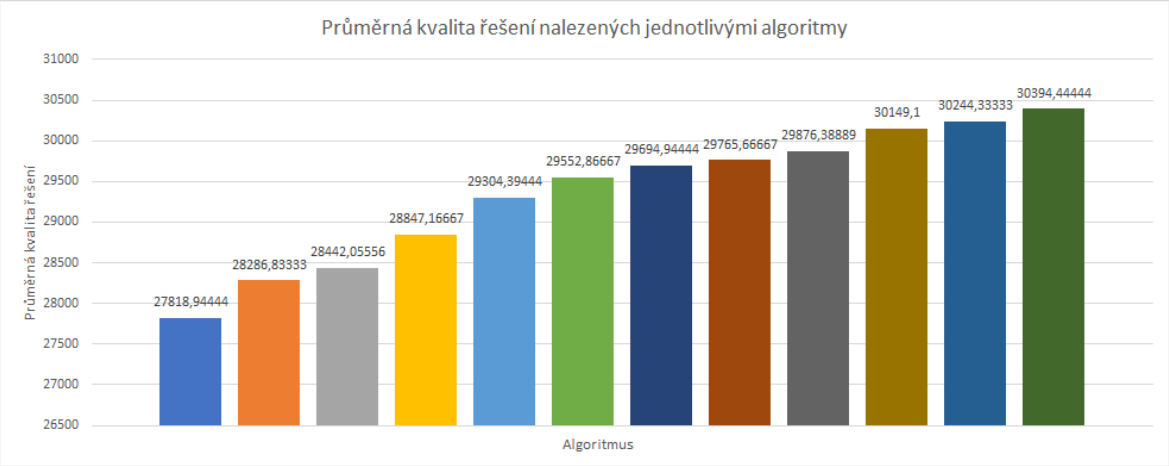


Obrázek 18. Kvalita řešení, detail výsledků na dvou největších problémech.

Obrázek 19 ukazuje průměr výsledků jednotlivých algoritmů přes všechny testovací příklady. Nejlepších výsledků v průměru dosahuje algoritmus *max(RandomWalk(100, swap + insertion) + Greedy(swap + insertion) + Postprocess, 10-times)*, který je nejlepší i na většině testovacích příkladů.

Nejhorších výsledků v průměru dosáhl algoritmus *Greedy(swap+insertion)*, což je poměrně překvapivé, protože jednodušší algoritmus *Greedy(insertion)* se umístil lépe.

Rozdíl mezi nejlepším a nejhorším algoritmem co do kvality činí 9,15 %.

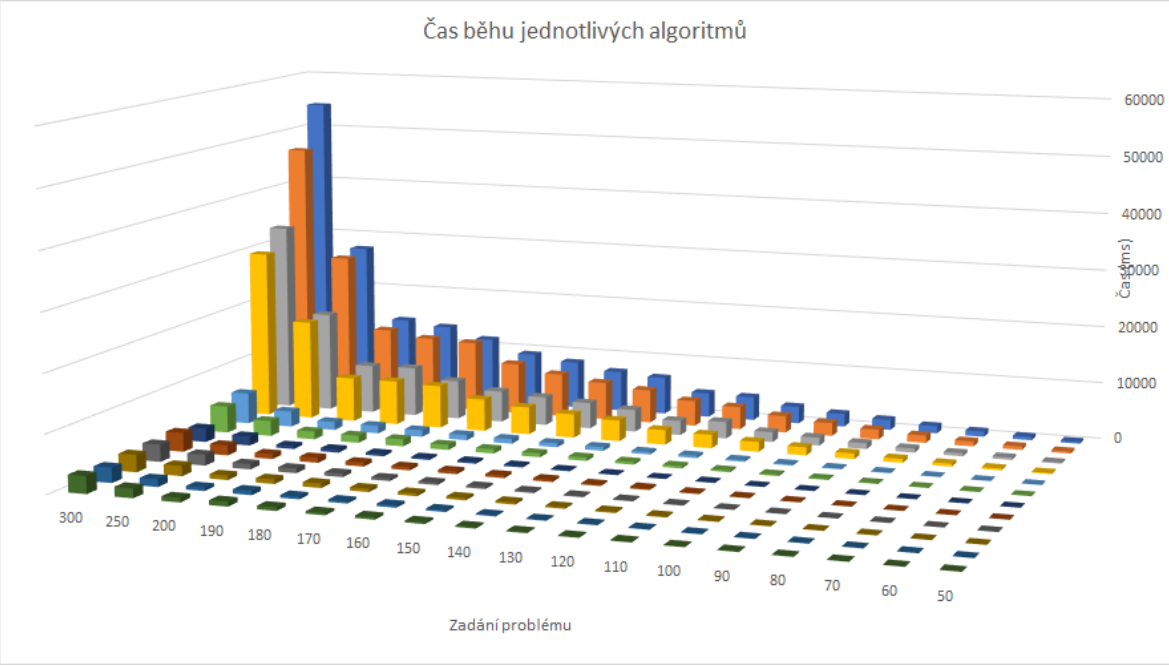


Obrázek 19. Průměr výsledků jednotlivých algoritmů přes všechny testovací příklady

7.3.2 Čas běhu jednotlivých algoritmů

Čas běhu jednotlivých algoritmů na všech problémech ukazuje Obrázek 20. Na vodorovné ose jsou instance problému a barevně jsou odlišené jednotlivé algoritmy podle legendy zmíněné výše. Výška sloupce odpovídá délce běhu příslušného algoritmu na daném problému. Délka běhu je měřena v milisekundách.

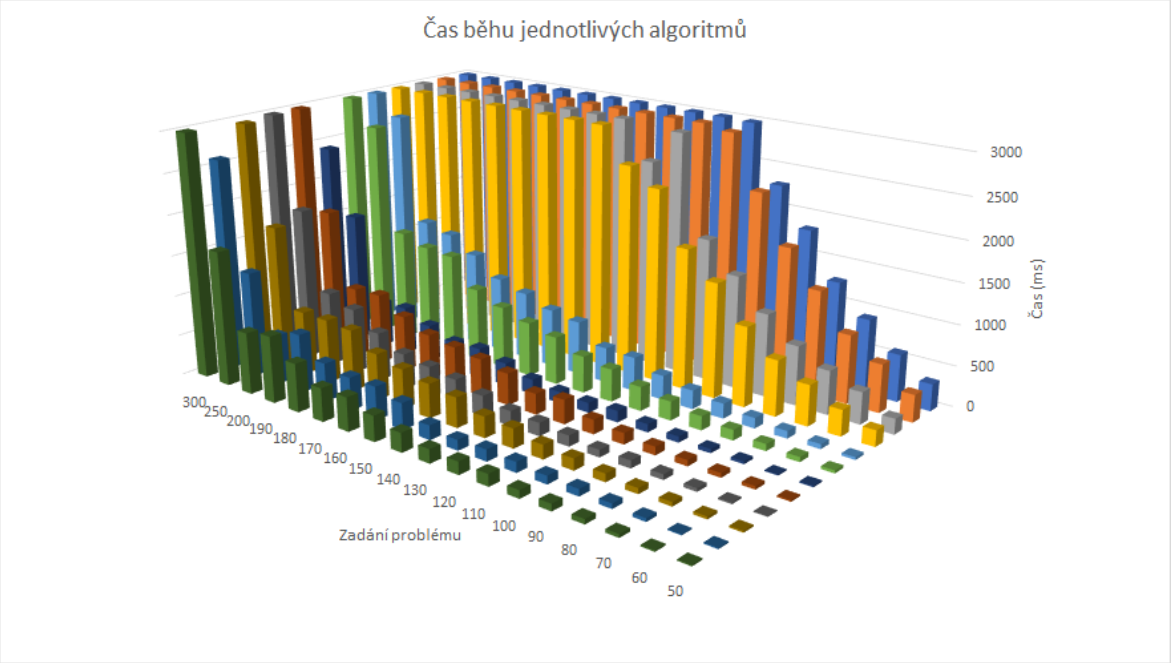
Lze si všimnout, že algoritmy, které nalézají kvalitní řešení, mají zdaleka nejvyšší náročnost na výpočetní čas. Více se této otázce věnuje sekce 7.3.4.



Obrázek 20. Čas běhu algoritmů na jednotlivých problémech, měřeno v milisekundách.

Vzhledem k velkým rozdílům v čase běhu mezi jednotlivými algoritmy jsou v grafu dominantní hodnoty u posledních čtyř algoritmů (algoritmy využívající maximum z několika běhů). Rozdíly mezi zbylými kategoriemi nejsou v grafu patrné.

Obrázek 21 ukazuje detailní záběr na graf času běhu, kde už jsou patrné i rozdíly mezi ostatními algoritmy. Většina z algoritmů běží na většině problémů v čase menším než jedna sekunda. Srovnání jednotlivých typů algoritmů je vidět na grafu, který zobrazuje Obrázek 24.



Obrázek 21. Čas běhu jednotlivých algoritmů, detail.

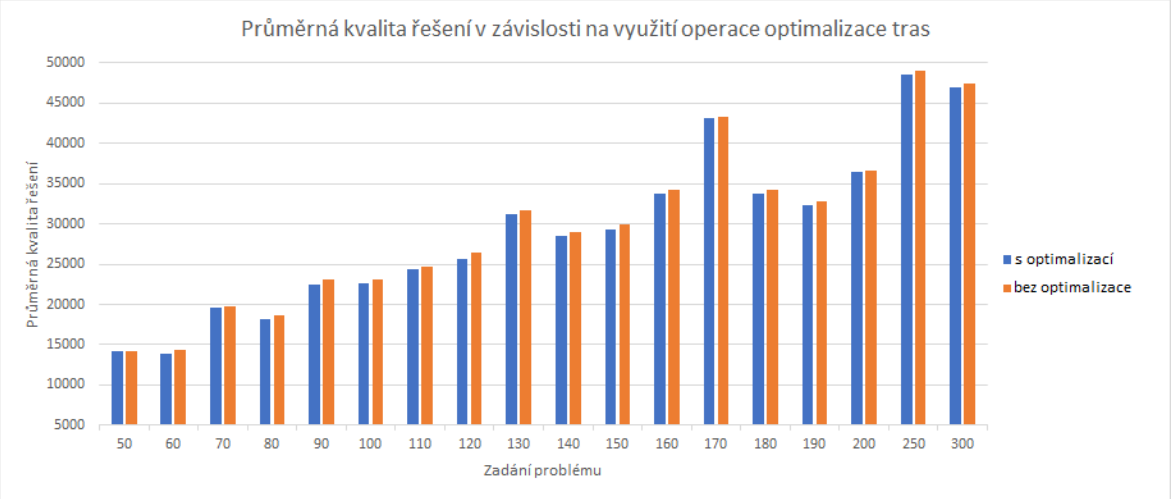
7.3.3 Kvalita řešení podle jednotlivých kategorií

Použité algoritmy lze rozdělit do několika kategorií podle toho, jestli využívají operaci optimalizace tras, zda používají operaci přesunu, nebo operaci přesunu i prohození, a podle toho, zda využívají pouze hladový algoritmus, náhodnou procházku nebo navíc i maximalizaci přes několik běhů.

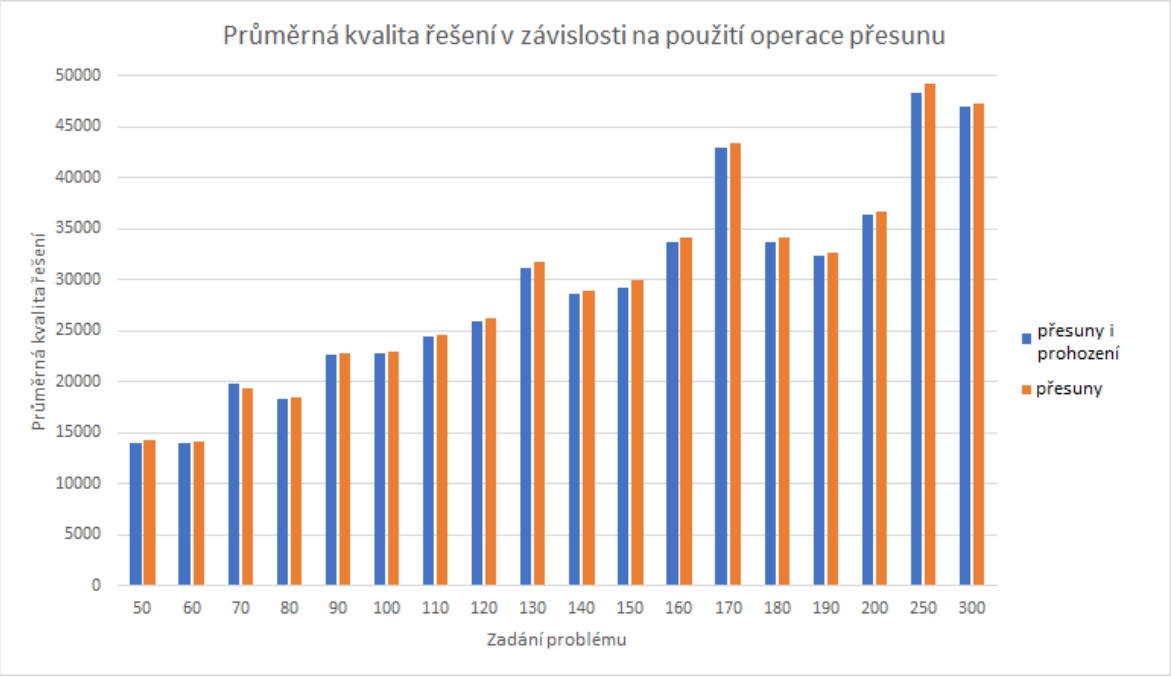
Vliv operace optimalizace tras na kvalitu řešení ukazuje Obrázek 22. Výška sloupce ukazuje průměrný výsledek u algoritmů využívajících optimalizaci tras a u těch, které optimalizaci tras nevyužívají. Je vidět, že optimalizace tras přináší vždy lepší výsledky než standardní algoritmus, což není překvapivé, protože optimalizace tras nemůže nikdy uškodit, může kvalitu řešení jedině vylepšit.

V průměru přináší použití operace optimalizace tras zlepšení kvality o 1,4 %.

Srovnání výsledků při použití operace přesunu oproti použití operací přesunu i prohození ukazuje Obrázek 23. Výška sloupce označuje průměrnou kvalitu řešení u algoritmů využívajících pouze přesuny (oranžová barva) a u těch, využívajících přesuny i prohození (modrá barva). Operace prohození ve většině případů vede k lepším výsledkům. V průměru se přidáním operace prohození zlepší kvalita řešení o 1,5 %.

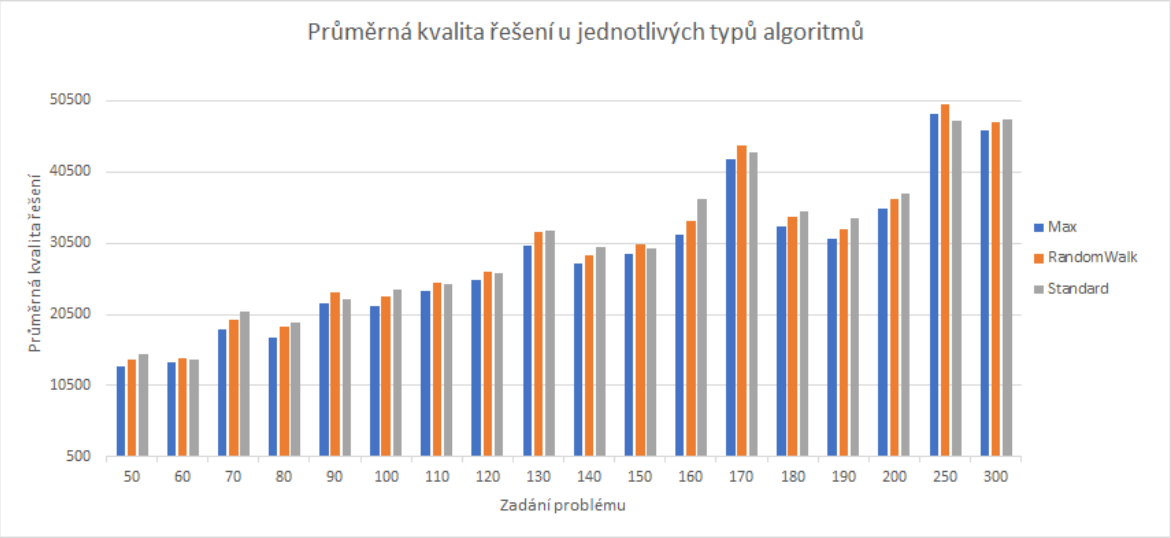


Obrázek 22. Vliv použití operace optimalizace tras na kvalitu řešení.



Obrázek 23. Vliv použití operace prohození na kvalitu řešení.

Je možné kvalitu řešení rozlišit také podle kategorie algoritmu. Algoritmy jsem rozdělil do tří kategorií: *standardní* – hladový algoritmus, případně doplněný o optimalizaci tras, *RandomWalk* algoritmy, které využívají náhodnou procházku a poté hladový algoritmus (s operacemi přesunu i přesunu a prohození a případně rozšířené o optimalizaci tras) a *Max*: algoritmy, které provádějí několik opakovaných pokusů a berou z nich ten nejlepší. Průměrné výsledky podle jednotlivých kategorií ukazuje Obrázek 24. Na vodorovné ose jsou jednotlivé testovací příklady a na svislé pak průměrná kvalita řešení. Barevně jsou odlišené jednotlivé kategorie algoritmů. Výška sloupce ukazuje vždy průměr v dané kategorii přes všechny algoritmy, které do kategorie spadají.



Obrázek 24. Kvalita řešení podle kategorií algoritmů.

Z výsledků je vidět, že algoritmy z kategorie *Standard*, dosahují na většině instancí nejhorších výsledků. Výjimku tvoří zejména problém s číslem 250, kde algoritmy typu *Standard* nalezly nejlepší řešení ze všech. Nejlepších výsledků naopak dosahují ve většině případů algoritmy typu *Max*.

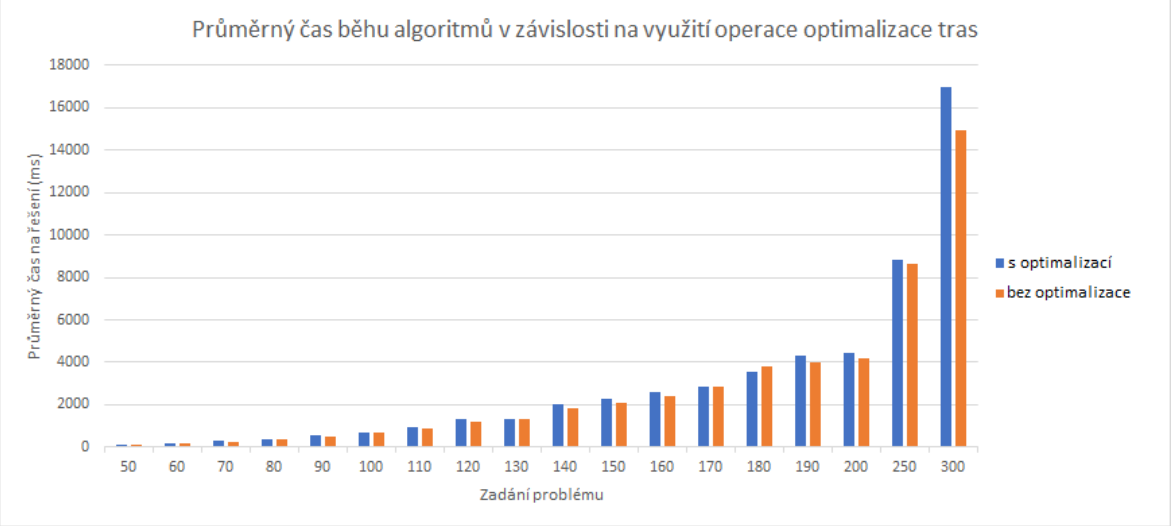
V průměru jsou algoritmy typu *Max* nejlepší, algoritmy typu *RandomWalk* jsou v průměru o 4,5 % horší a algoritmy typu *Standard* jsou o 6 % horší než ty typu *Max*.

7.3.4 Čas běhu v závislosti na kategorii algoritmu

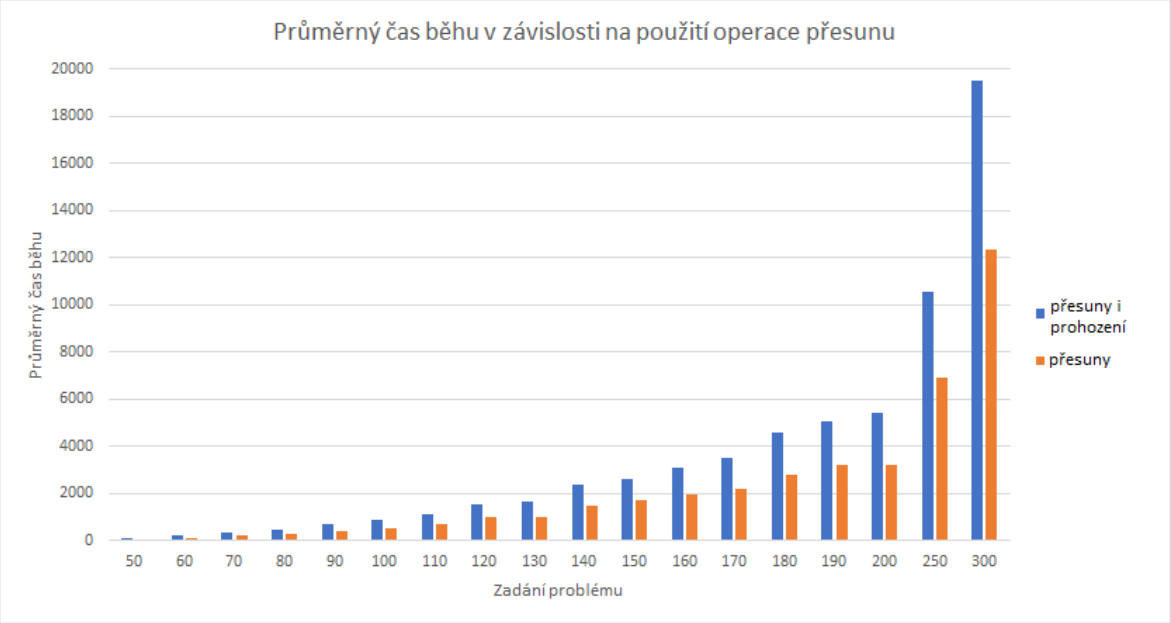
Kromě kvality řešení je možné analyzovat i čas běhu v závislosti na kategoriích. Obrázek 25 ukazuje, jak se liší průměrný čas běhu u algoritmů využívajících operaci optimalizace tras, oproti těm, které ji nevyužívají. Výška sloupce v tomto případě ukazuje průměr z času běhu všech algoritmů, které využívaly optimalizaci tras (modrá barva) a z těch, které ji nevyužívaly (oranžová barva). Z grafu je vidět, že optimalizace tras je časově náročnější, což není překvapivé, protože je s ní spojená dodatečná práce.

Výjimku tvoří problém 180, kde vedlo použití optimalizace tras ke snížení celkového výpočetního času. Tento efekt lze vysvětlit tak, že se dostavil výkyv v rychlosti použitého počítače. Ten může být způsoben například tzv. *přerušením*, kdy je prováděná činnost krátkodobě přerušena jinou činností s vyšší prioritou, například požadavkem operačního systému, nebo se mohl aktivovat tzv. *Garbage Collector*, což je automatizovaná procedura frameworku .NET pro vyčištění nepoužívané paměti.

V průměru vede přidání operace optimalizace tras ke zvýšení výpočetního času o 6 %.

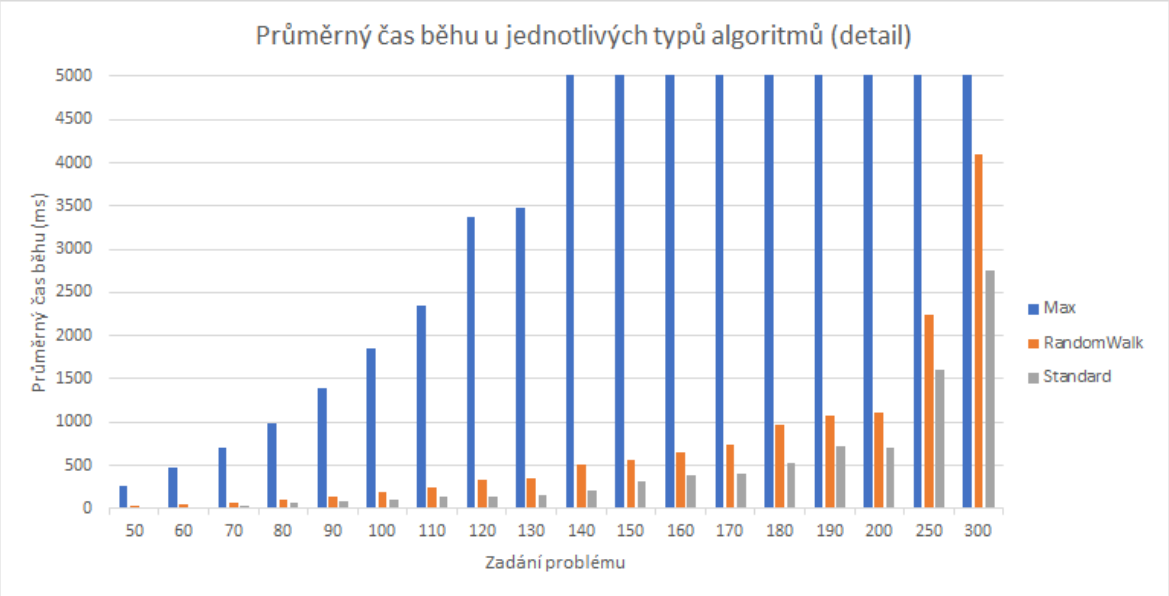


***Obrázek 25.* Průměrný čas běhu v závislosti na použití operace optimalizace tras.**



Obrázek 26. Průměrný čas běhu v závislosti na využití operace přesunu a prohození.

Obrázek 26 ukazuje vliv použití operace výměny (v grafu označené jako prohození) na čas běhu. Z grafu je vidět, že přidání operace výpočetní čas vždy zvyšuje a to v průměru o 59 %.



Obrázek 27. Čas výpočtu v závislosti na kategorii algoritmů (detail).

Obrázek 27 zobrazuje průměrný čas běhu algoritmů v závislosti na kategorii. Jedná se o detail, hodnoty větší než *5000 ms* nejsou v grafu zobrazené. Z grafu je vidět, že algoritmy typu *Standard* běží nejrychleji, následované algoritmy typu *RandomWalk*. Tato kategorie má vždy vyšší výpočetní čas, protože přidání náhodné procházky výpočet vždy prodlouží.

Algoritmy typu *Max* mají zhruba desetkrát vyšší čas běhu, což není překvapivé, protože provádějí 10 pokusů a berou z nich ten nejlepší.

7.4 Vyhodnocení experimentů

Z výsledků experimentů lze učinit několik pozorování:

Operace optimalizace tras má celkově pozitivní efekt. Srovnáním výsledků dosažených algoritmem bez využití optimalizace a stejného algoritmu, který ji využívá lze zjistit, že využití optimalizace se vždy vyplatí a přináší průměrně zlepšení o 1,4 %. Fakt, že optimalizace tras řešení zlepšuje, ukazuje, že předchozí dvě operace nehledají trasy optimálně.

Časová náročnost operace optimalizace tras je poměrně malá. V průměru se využitím optimalizace tras zvyšuje výpočetní čas o 6 %.

Randomizace celkově přináší pozitivní efekt. Při srovnání původní verze algoritmů *Greedy(Insertion)* a *Greedy(Swap+Insertion)* s verzemi, které přidávají náhodnou procházku lze zjistit, že verze s náhodnou procházkou jsou na většině zadání lepší a dávají průměrně o 1,2 % lepší řešení. To platí bez ohledu na využití optimalizace tras.

Toto chování je zajímavé, protože znamená, že náhodná trasa je lepším výchozím bodem pro následný hladový algoritmus než trasa vzniklá tak, že ke každému zákazníkovi pojede samostatné vozidlo. Tento efekt se mi nepodařilo vysvětlit.

Randomizace je časově poměrně nenáročná, celkově přidání randomizace zvyšuje výpočetní čas průměrně o 60 %. Toto zvýšení není nijak hrozivé, protože standardní algoritmy běží velmi rychle – typicky méně než jednu sekundu.

Nejlepších výsledků jednoznačně dosahují varianty, které provádějí deset randomizovaných pokusů a berou z nich ten nejlepší. V průměru dává tento postup ve srovnání s jediným opakováním o 5,5 % lepší výsledky. Tato strategie je již o poznání složitější než původní verze heuristik, jedná se už spíše o meta-heuristiku. To se projevuje také vyšším výpočetním časem. Vzhledem k tomu, že se jedná o deset opakování, je výpočetní čas v průměru zhruba 10-krát vyšší.

Nárůst výpočetního času by bylo možné redukovat využitím více jader procesoru. Deset nezávislých pokusů by totiž bylo možné vykonat paralelně a výpočetní čas by se tak zvýšil jen minimálně. Navíc moderní procesory už minimálně deset jader běžně poskytují. Předchozí techniky však více jader nevyužívají, tak jsem v rámci férovosti srovnání tuto možnost nevyužil ani zde.

Úplně nejlepší řešení našla u většiny zadání nejsložitější varianta algoritmu, tedy *max(RandomWalk(100, swap + insertion) + Greedy(swap + insertion) + Postprocess, 10-times)*. Tato varianta spotřebovávala stabilně také nejvíce výpočetního času.

Využití pouze operace přesunu (tedy varianty algoritmu obsahující *Insertion*) je rychlejší než využití obou operací (tedy varianty obsahující *Swap+Insertion*). To není překvapivé, protože se generuje větší okolí, což vede ke zpomalení výpočtu. V průměru se výpočetní čas zvětší o 59 %. Zvětšení zhruba odpovídá i nárůstu velikosti okolí. Samotná operace přesunu totiž generuje okolí o velikosti řádově n^2 a operace výměny přidává dalších řádově $n^2/2$ což je nárůst ve velikosti okolí o 50 %.

Ve většině případů však dokáže kombinace obou operací nalézt lepší řešení, než využití pouze jediné operace. Průměrně dávají varianty využívající obě operace o 1,5 % lepší řešení než varianty s jedinou operací.

Nejhorší výsledky dává v celkovém srovnání algoritmus *Greedy(swap + insertion)*. Využití vícero různých algoritmů se vyplatilo, protože vykazují dost rozdílné chování i výsledky. Rozdíl mezi nejhorším a nejlepším algoritmem co do kvality řešení činí v průměru 9,5 %, což už by v praktických aplikacích mělo značný ekonomický dopad.

Velké rozdíly jsou také v čase běhu jednotlivých algoritmů. Nejrychlejší je algoritmus *Greedy(insertion)*, který dokázal vyřešit všechny testovací problémy dohromady v čase 7690ms (7 a půl sekundy). Nejpomalejší je naopak algoritmus *max(RandomWalk(100, swap + insertion) + Greedy(swap + insertion) + Postprocess, 10-times)*, který pro vyřešení všech testovacích problémů potřeboval v součtu čas 171245ms (necelé tři minuty).

Toto dává uživateli možnost zvolit algoritmus vhodný pro jeho účely. Buď může vybrat časově náročnější variantu, která pravděpodobně najde lepší řešení, nebo naopak rychlou variantu, která však pravděpodobně najde řešení horší kvality.

Celkově je výpočetní čas přijatelný a navržené algoritmy jsou v praxi dobře použitelné i na velkých instancích problému.

8 Závěr

Tato práce představuje několik heuristik pro řešení rozvozního problému s časovými okny. Problém je studován také z teoretického hlediska a jsou zmíněná i formální omezení na efektivitu řešících algoritmů.

V práci se podařilo vytvořit dvě jednoduché heuristiky a několik složitějších algoritmů, které tyto heuristiky využívají. Svou povahou odpovídají tyto složitější postupy spíše metaheuristikám.

V práci je také popsaná navržená desktopová aplikace, která implementuje vytvořené algoritmy a umožňuje efektivně řešit rozvozní problémy v praxi. V textu práce je analyzovaný způsob efektivní implementace algoritmů, včetně výběru vhodných datových struktur, návrhu objektové architektury a dalších implementačních detailů.

Celkem práce představuje 12 algoritmů a efektivita těchto algoritmů je otestovaná na sadě 18-ti testovacích příkladů. Algoritmy jsou porovnané z hlediska kvality nalezeného řešení i času potřebného pro výpočet. Je studován také vliv různých parametrů algoritmů (například přidání náhodné procházky a podobně) na kvalitu nalezených řešení i na čas běhu.

Algoritmy jsou testované na poměrně velkých příkladech, největší z nich obsahoval 300 měst, což už je výrazně více než kolik se běžně vyskytuje v praktických problémech. Čas běhu použitých algoritmů je přijatelný. Na největších problémech běžel nejrychlejší algoritmus zhruba 2 sekundy a nejpomalejší zhruba 54 sekund. Algoritmy jsou tedy dobře použitelné v praxi.

8.1 Další možná rozšíření

Aplikace v současné době neumožňuje načítat zadání problému přímo vytvořeného uživatelem. Tuto možnost by bylo možné poměrně jednoduše přidat, ale v takovém případě by program nebyl schopný vykreslovat polohy míst do mapy. Zadání by totiž probíhalo pomocí matice vzdáleností a nelze nijak garantovat, že zadané vzdálenosti budou odpovídat vzdálenostem vzdušnou čarou nějakých bodů na mapě. V případě, že by aplikace měla být reálně nasazená, bylo by samozřejmě třeba načítání zadání dodělat. K tomu účelu by také bylo třeba přesně definovat formát vstupních dat.

V současné době je možné aplikaci ovládat pouze graficky klikáním na příslušná tlačítka. Tento způsob je velmi vhodný pro vývoj a testování algoritmů, ale pro reálné nasazení by bylo vhodnější, aby aplikace podporovala i dávkové zpracování, tzn. spuštění z příkazové řádky, automatické zpracování většího množství vstupních problémů a uložení výsledků. Dávkové zpracování by bylo možné přidat poměrně jednoduše, jednalo by se pouze o drobnou úpravu zdrojových kódů.

Jiné rozšíření aplikace může spočívat v návrhu dalších druhů heuristik a zejména v kombinacích již implementovaných postupů, čili přidání metaheuristik. V aplikaci je již k dispozici hladový algoritmus i náhodná procházka a také jejich jednoduchá kombinace. Bylo by možné dopracovat například algoritmus *Simulovaného žíhání*, který hladový algoritmus kombinuje s náhodnou procházkou mnohem sofistikovanějším způsobem.

Poslední důležitou věcí, která v práci chybí, je experimentální srovnání navržených algoritmů s jinými technikami, například s MIP řešiči, a případně i se specializovaným komerčním softwarem. Pokud by navržené postupy v tomto srovnání obstály, bylo by možné zde dosažené výsledky případně publikovat i formou odborného článku na některé z tematicky blízkých konferencí.

Terminologický slovník

Termín	Zkratka	Význam (zdroj)
Aproximační algoritmus		Algoritmus, jehož aproximační poměr je konstantní, čili je to reálné číslo větší než jedna. Viz sekci 2.3.3. (Arora a Barak, 2009, s. 351 - 368)
Aproximační poměr algoritmu		Horní mez na poměr kvality řešení, které algoritmus nalezne, ku kvalitě nejlepšího možného řešení. Viz sekci 2.3.3. (Arora a Barak, 2009, s. 351 - 368)
Časová složitost algoritmu		Závislost počtu kroků algoritmu na velikosti vstupu. (Arora a Barak, 2009, s. 17)
Časové okno		Časový interval, který udává, kdy je možné zákazníka navštívit.
Heuristika		Jednoduchý postup pro řešení optimalizačního problému, který negarantuje nalezení optimálního řešení, ale je poměrně rychlý.
Mixed Integer Programming	MIP	Celočíselné programování. Varianta lineárního programování, kde jsou navíc kladeny požadavky na celočíselnost některých proměnných (Korte et al., 2010, s. 101 - 122)
Modifikační operace		Procedura, která přetváří jedno řešení problému na jiné řešení.
Okruh		Viz Trasa.
Okružní dopravní problém		Viz Problém obchodního cestujícího.
Polynomiální algoritmus		Algoritmus, jehož časová složitost je funkce ve tvaru polynomu. Tyto algoritmy jsou rychlé a dobře použitelné v praxi.
Poptávka zákazníka		Množství zboží, které je třeba k zákazníkovi dopravit.
Problém obchodního cestujícího		Dopravní problém zabývající se optimalizací délky cesty při okružní jízdě přes několik zadaných míst. Viz sekci 2.2.1. (Pelikán, 2010)
Problém obchodního cestujícího s časovými okny		Dopravní problém zabývající se optimalizací délky cesty při návštěvě několika zadaných míst, kde je každé místo možné navštívit jen ve vymezeném časovém intervalu. Viz sekci 2.2.2. (Pelikán, 2010)
Problém okružních jízd		Viz Rozvozní problém.
Rozvozní problém		Dopravní problém zabývající se optimalizací délky cest při rozvozu zboží z centrály k zákazníkům. Viz sekci 2.2.3. (Pelikán, 2010)
Rozvozní problém s časovými okny		Dopravní problém zabývající se optimalizací délky cest při rozvozu zboží z centrály k zákazníkům, kde je možné každého zákazníka navštívit jen ve vymezeném časovém intervalu. Viz sekci 2.2.4. (Pelikán, 2010)
Řešení problému		Množina tras, která udává, jakým způsobem budou zákazníci obslouženi.
Trasa		Posloupnost vrcholů, které vozidlo v tomto pořadí navštíví.

Termín	Zkratka	Význam (zdroj)
Traveling Salesman Problem	TSP	Viz Problém obchodního cestujícího.
Traveling Salesman Problem with Time Windows	TSPTW	Viz Problém obchodního cestujícího s časovými okny.
Vehicle Routing Problem	VRP	Viz Rozvozní problém.
Vehicle Routing Problem with Time Windows	VRPTW	Viz Rozvozní problém s časovými okny.

Seznam literatury

AARTS, E. H. L. a J. K. LENSTRA. *Local search in combinatorial optimization*. Princeton: Princeton University Press, 2003. ISBN 9780691115221.

ARORA, Sanjeev. a Boaz. BARAK. *Computational complexity: a modern approach*. New York: Cambridge University Press, 2009. [cit. 19.04.2017]. ISBN 9780521424264. Dostupné z: <http://theory.cs.princeton.edu/complexity/book.pdf>.

BALDACCI, Roberto, Nicos CHRISTOFIDES a Aristide MINGOZZI. *An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts*. *Mathematical Programming* [online]. 2008, **115**(2), 351-385 [cit. 19.04.2017]. DOI: 10.1007/s10107-007-0178-5. ISSN 0025-5610. Dostupné z: <http://link.springer.com/10.1007/s10107-007-0178-5>.

BALDACCI, Roberto, Aristide MINGOZZI a Roberto ROBERTI. *New Route Relaxation and Pricing Strategies for the Vehicle Routing Problem*. *Operations Research* [online]. 2011, **59**(5), 1269-1283 [cit. 19.04.2017]. DOI: 10.1287/opre.1110.0975. ISSN 0030-364x. Dostupné z: <http://pubsonline.informs.org/doi/abs/10.1287/opre.1110.0975>.

BRECKLINGHAUS, Judith a Stefan HOUGARDY. *The Approximation Ratio of the Greedy Algorithm for the Metric Traveling Salesman Problem*. *Operations Research Letters* [online]. 2015, **43**(3), 259-261 [cit. 19.04.2017]. DOI: 10.1016/j.orl.2015.02.009. ISSN 01676377. Dostupné z: <http://linkinghub.elsevier.com/retrieve/pii/S0167637715000280>.

BURKE, Edmund K, Michel GENDREAU, Matthew HYDE, Graham KENDALL, Gabriela OCHOA, Ender ÖZCAN a Rong QU. *Hyper-heuristics: a survey of the state of the art*. *Journal of the Operational Research Society* [online]. 2013, **64**(12), 1695-1724 [cit. 19.04.2017]. DOI: 10.1057/jors.2013.71. ISSN 0160-5682. Dostupné z: <http://link.springer.com/content/pdf/10.1057 %2Fjors.2013.71.pdf>.

FUKASAWA, Ricardo, Humberto LONGO, Jens LYSGAARD, Marcus Poggi de ARAGÃO, Marcelo REIS, Eduardo UCHOA a Renato F. WERNECK. *Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem*. *Mathematical Programming* [online]. 2006, **106**(3), 491-511 [cit. 19.04.2017]. DOI: 10.1007/s10107-005-0644-x. ISSN 0025-5610. Dostupné z: <http://link.springer.com/10.1007/s10107-005-0644-x>.

HAN, Jian. *Evaluation of Optimization Algorithms for Improvement of a Transportation Companies (in-house) Vehicle Routing System.* Nebraska, 2010. Diplomová práce. University of Nebraska - Lincoln. [cit. 19.04.2017]. Dostupné také z: <http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1012&context=imsediss>.

HELMERT, Malte. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*. Berlin: Springer, 2008. Algorithms and combinatorics, 21. ISBN 978-3-540-77722-9.

KORTE, Bernhard a Jens VYGEN. *Combinatorial optimization: theory and algorithms, Fifth Edition*. Berlin: Springer, 2010. Algorithms and combinatorics, 21. ISBN 3-540-67226-5.

PELIKÁN, Jan. *Diskrétní modely v operačním výzkumu*. 1. vyd. Praha: Professional Publishing, 2001. ISBN 80-86419-17-7.

PELLONPERA, Tuomas. *Ant colony optimization and the vehicle routing problem.* Tampere, 2014. Diplomová práce. University of Tampere, School of Information Sciences. [cit. 19.04.2017]. Dostupné také z: <http://uta32-kk.lib.helsinki.fi/bitstream/handle/10024/95400/GRADU-1401262041.pdf?sequence=1>.

ROTHLAUF, Franz. *Design of modern heuristics: principles and application*. New York: Springer, 2011. Natural computing series. ISBN 9783540729624.

Traveling Salesman Problem -- from Wolfram MathWorld. *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource* [online]. Copyright © 1999 [cit. 19.04.2017]. Dostupné z: <http://mathworld.wolfram.com/TravelingSalesmanProblem.html>.

VAIRA, Gintaras. *Genetic Algorithm for Vehicle Routing Problem.* Vilniaus, 2014. Dizeratční práce. Vilniaus Universitetas. [cit. 19.04.2017]. Dostupné také z: <http://old.mii.lt/files/mii_dis_2014_vaira.pdf>.

Vehicle Routing Problem. *NEO: Networking and Emerging Optimization* [online]. Copyright © 2013 [cit. 19.04.2017]. Dostupné z: <http://neo.lcc.uma.es/vrp/vehicle-routing-problem/>.

ZHANG, Weixiong. *State-space search: algorithms, complexity, extensions, and Applications*. New York: Springer, 1999. ISBN 978-1-4612-7183-3.

Seznam obrázků a tabulek

Seznam obrázků

Seznam tabulek

Příloha A: Obsah přiloženého archivu

Elektronickou přílohou práce je archiv ve formátu *zip*, který obsahuje zejména zdrojové kódy navržené aplikace, spustitelný soubor aplikace, testovací příklady použité v experimentech a tabulku s výsledky všech experimentů.

Detailní obsah přiloženého archivu:

- Složka *Aplikace* obsahuje spustitelný soubor, který umožňuje používat navrženou aplikaci
- Složka *Zdrojové kódy* obsahuje všechny zdrojové kódy programu v jazyce C#. Zdrojové kódy lze prohlížet, případně upravovat otevřením souboru *VRPTW.sln* pomocí *Visual Studia* nebo jiného vývojového nástroje
- Složka *Experimenty* obsahuje 18 zadání problému, které byly použité při testování a soubor *Vysledky.xlsx*, který obsahuje tabulku všech dosažených výsledků. z této tabulky byly vytvořené všechny zde prezentované grafy.
- Soubor *text.pdf* obsahuje elektronickou verzi tohoto textu.