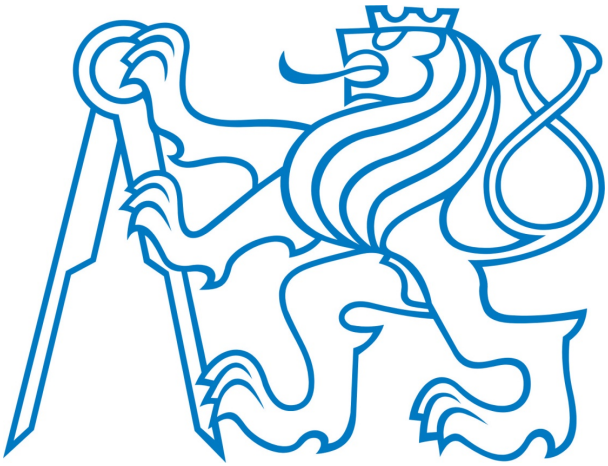
České vysoké učení technické v Praze

Fakulta dopravní



Evoluční techniky pro optimalizaci grafikonu

Diplomová práce

Bc. Valerii Gopak

Praha, 2017

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
d ě k a n
Konviktská 20, 110 00 Praha 1

K614..... **Ústav aplikované informatiky v dopravě**

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Bc. Valerii Gopak

Kód studijního programu a studijní obor studenta:

N 3710 – IS – Inteligentní dopravní systémy

Název tématu (česky): **Evoluční techniky pro optimalizaci grafikonu**

Název tématu (anglicky): Evolution technique for train diagram optimization

Zásady pro vypracování

Při zpracování diplomové práce se řiďte osnovou uvedenou v následujících bodech:

- Analyzujte využití evolučních technik pro řešení úloh typu rozvrhování se zaměřením na řízení železničního provozu.
- Formulujte optimalizační problém.
- Diskutujte možnosti jeho parametrizace (geografický a časový rozsah optimalizace).
- Navrhněte vhodnou kriteriální funkci.
- Navrhněte evoluční algoritmus pro optimalizaci grafikonu vlakové dopravy.

Rozsah grafických prací: dle pokynů vedoucího práce

Rozsah průvodní zprávy: minimálně 55 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)

Seznam odborné literatury:

Zelinka I., Oplatková Z., Šeda M., Ošmera P., Včelař F., Evoluční výpočetní techniky, principy a aplikace, BEN, 2008, Praha, ISBN 80-7300-218-3

Tormos P., et. al, A Genetic Algorithm for Railway Scheduling Problems, Studies in Computational Intelligence (SCI) 128, Springer-Verlag, 2008, Berlin

Vedoucí diplomové práce: **doc. Ing. Vít Fábera, Ph.D.**
Ing. Dušan Kamenický

Datum zadání diplomové práce: **29. července 2016**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání diplomové práce: **30. května 2017**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta dopravní
L.S.

doc. Dr. Ing. Tomáš Brandejský
vedoucí
Ústavu aplikované informatiky v dopravě

prof. Dr. Ing. Miroslav Svítek, dr. h. c.
děkan fakulty

Potvrzuji převzetí zadání diplomové práce.

Bc. Valerii Gopak
jméno a podpis studenta

V Praze dne.....

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci, zpracovanou v závěru studia na ČVUT v Praze, Fakultě dopravní.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorských zákonu).

Praha, 30. května 2017 .....

Valerii Gopak

Poděkování

Rád bych touto cestou vyjádřil poděkování doc. Ing. Vítu Fáberovi, Ph.D. a Ing. Bc. Dušanovi Kamenickému za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnovali.

Evoluční techniky pro optimalizaci grafikonu

Bc. Valerii Gopak

ČVUT v Praze, Fakulta dopravní

Praha, 2017

Abstrakt

Cílem této diplomové práce je analýza problému rozvrhování v železniční dopravě a návrh evolučního algoritmu optimalizujícího grafikon. Vstupem je stávající grafikon,popis infrastruktury, charakteristiky souprav a informace o mimořádných událostech ovlivňujících grafikon. Výstupem je optimalizovaný grafikon. Z větší části se práce zabývá úvahami o optimalizaci grafikonu a detekci konfliktů. Výsledky práce jsou teoretické: rozbor problematiky, rámcový návrh genetického algoritmu a praktické: programový modul pro načítání dat z XML, připojení k MySQL databázi a hledání cest v načteném grafu.

Klíčová slova

Genetický algoritmus, rozvrhování, XML

Evolution technique for train diagram optimization

Bc. Valerii Gopak

CTU in Prague, Faculty of Transportation Sciences

Prague, 2017

Abstract

The goal of this Master's Thesis is to analyse rail transport scheduling problem and to design the evolutionary algorithm that optimises train diagram. Initial state includes the existing train diagram with information about unexpected events, the infrastructure description and train set charachteristics. Final state presents the optimised train diagram. For the most part this thesis is dealing with train diagram optimization and conflict detection. The theoretical results of the thesis are problem analysis and the design of the genetic algorithm. Practical results are the application module for XML data import, MySQL database connection and path findig in the imported graph.

Keywords

Genetic algorithm, scheduling, XML

Obsah

Kapitola 1

Úvod

Evoluční výpočetní techniky se dnes používají v různých odvětvích, masivní uplatnění našly i v dopravním sektoru. Doprava má za úkol přepravu lidí, zboží a informací po dopravních cestách s využitím dopravních prostředků. Proto většina úloh spadající do tohoto odvětví má optimalizační charakter - vyžaduje ve všech případech nalezení optimální trasy od zdroje k cíli, přičemž kritérium optimality může být nejkratší čas, nejkratší vzdálenost nebo minimální energie spotřebovaná na přepravu.

Tato diplomová práce se zabývá optimalizací řízení železniční dopravy. Podstatou je řešení konfliktů v jízdách vlaků způsobeného odchylkami od naplánovaného jízdního řádu z důvodu různých mimořádností a poruch, ať na straně infrastruktury nebo vozidel. Problematika se řeší v rámci celé Evropy, největšího úspěchu dosáhly ve Švýcarsku, kde jsou schopni predikovat konflikty několik hodin dopředu.

Cílem této diplomové práce bude návrh genetického algoritmu, úkolem kterého bude úprava naplánovaného grafikonu, a jeho částečná implementace. Vstupem bude již naplánovaný grafikon, popis železniční infrastruktury a aktuální stav – poloha vlaků, stav infrastruktury atd.. Jedním z úkolů navrhované logiky bude zkontrolovat, zda je potřeba stávající grafikon upravovat. Výstupem algoritmu bude nekolizní grafikon, kde jsou započteny všechny vstupující událostí a veškeré dotčené vlaky dostanou optimální trasy.

Protože jde o téma, které nebývá běžně řešeno, návrhu vlastního algoritmu předcházela podrobná analýza problému. Z ní vyplynulo, že před implementací je potřeba vytvořit několik programových modulů, aby bylo možné vůbec genetický algoritmus reálně implementovat. Jako cílová platforma pro první verzi byl zvolen Dopravní sál Fakulty dopravní. Především, pro implementaci algoritmu bylo potřebné mít podrobně popsanou infrastrukturu - tato informace musí

být digitální a dobře dokumentovaná, aby byla možnost data v programu zpracovávat. Všechny požadavky splňovala právě infrastruktura Dopravního sálu Fakulty dopravní. V rámci své diplomové práce vytvořil Ing. Petr Koutecký popis infrastruktury ve formátu XML, jehož popis bude podrobněji rozebrán v další kapitole. Grafikon pro potřeby vlastních simulací se ukládá v databázi MySQL. Pro implementaci modulů byl použit programovací jazyk C++ a volně dostupné knihovny (zdůvodnění všech použitých prostředků bude následovat v dalších kapitolách). Výstupem práce je jednak analýza problematiky (podmínky a omezení, vstupy) a návrh vnitřní reprezentace infrastruktury, návrh výpočtu simulace pohybu vlaku a rámcový návrh evolučního algoritmu. Implementovány byly následující programové moduly

- načítání dat XML
- načítání dat z MySQL databáze
- hledání alternativních cest v grafu

Kapitola 2

Definice problému

Jedním z problémů železniční dopravy je často vznikající nedodržení naplánovaných jízdních řádů. K odchylkám však dochází ne kvůli nekorektně navrženému grafikonu, ale díky nepředvídaným událostem vznikajícím až na provozní etapě.

Pokud na grafikon nahlížíme jako na systém, identifikujeme objekty jako infrastruktura a vozidla interagující mezi sebou do určité míry v čase. Dopředu v něm definujeme chování jeho prvků (jízdy vlaků dle pravidelného grafikonu), ale na systém má značný vliv jak blízké tak i vzdálené okolí. Neexistují nástroje umožňující takovou interakci predikovat, proto zatím nelze takové odchylky eliminovat. Blízkým okolím ovlivňujícím systém může být jak náhlá porucha infrastruktury, po které se vlaky pohybují, technický stav vozidla, dodržování dopravních předpisů strojvedoucím, tak i mnoho dalších, s tím souvisejících mimořádných událostí. Ke vzdálenému okolí můžeme počítat jak počasí (např. sněhová bouře), zpoždění může také vzniknout mimo řízenou oblast (vlaky přijíždějící ze zahraničí). Odchylka od jízdních řádů nejenom snižuje komfort cestujících, ale má dopad i na kapacitu dopravní cesty. V případě zpoždění vlaku může souprava obsazovat kolejové úseky přidělené v tento čas jiné soupravě, díky čemu vznikne konflikt. Tak vzniká potřeba tyto konflikty řešit.

Moderním trendem v řízení železničního provozu (především na železniční síti ve Švýcarsku, Belgii a nově i Německu) je oddělení strategického řízení (plánování tras vlaků v rámci ročního až denního jízdního řádu), operativního řízení (řešení konfliktů v jízdách vlaků způsobeného odchylkami od naplánovaného jízdního řádu z důvodu různých mimořádností a poruch ať na straně infrastruktury nebo vozidel) a přímého řízení (obsluha zabezpečovacího zařízení s případnou spoluúčastí pro zajištění bezpečnosti provozu, organizace posunu). Právě operativní řízení zajistí výše zmíněnou zpětnou vazbu v systému, tím pádem navrhovaný algoritmus spadá k to-

muto druhu řízení. Podstatného zlepšení řízení železničního provozu dosáhneme tím, že řízení neomezujeme pouze na stavění vlakových cest, ale přímo vozidlu poskytujeme informaci, kde a v jakém čase se má nacházet. To lze považovat za aplikaci klasického přístupu regulace se zpětnou vazbou.

Řešíme-li objevené konflikty, máme řadu možností, co v konkrétním případě učinit. Z množiny vhodných řešení musíme vybrat optimální. Je to velmi široký pojem, definující vlastní měřítko v závislosti na řešeném problému. K jeho stanovení nám budou sloužit určitá kritéria (popsána dále). Nově vygenerované řešení pak musí tato kritéria splňovat.

2.1 Detekování konfliktů

Abychom měli možnost grafikon optimalizovat, musíme nejdřív odhalit vzniklé kolize, pokud takové jsou. Konflikty na železnici se vyskytují díky velkému množství vznikajících neplánovaných zásahů a procesů, které nelze predikovat. Mezi neplánované zásahy patří závady na vozidle a infrastruktuře.

Detekce konfliktů v jízdách vlaků je značně závislá na způsobu zabezpečení jízd vlaků. Je důležité si uvědomit, že každý provozovatel infrastruktury v závislosti na jejím vybavení uvažuje s rozdílnou množinou rizik a jejich následků. Z toho důvodu definuje odlišné funkční požadavky na zabezpečovací zařízení a další postupy pro zajištění bezpečnosti železničního provozu. To může výrazně ovlivňovat podmínky současné jízdy vlaků daným místem. Závisí jak lokálně na vybavenosti infrastruktury, tak na vybavenosti konkrétních vlaků zabezpečovacím zařízením. Nelze tedy uvažovat s pouhým obsazováním jednotlivých úseků příslušnými vlaky, ale je nutné zohlednit i zapevnění pojížděných (případně i dalších) úseků před jízdou vlaků v dostatečném předstihu, případně i se zapevněním těchto úseků po definovanou dobu po jízdě vlaků. Přičemž zapevnění těchto úseků může být podmíněno časově i prostorově. Specifickým případem může být konflikt v jízdách vlaků z důvodu ohrožení cestujících v případě úrovňového přístupu na nástupiště – pak záleží i na druhu vlaků.

Vzhledem k tomu, že kolize jsou příčiny potencionálních mimořádných událostí (nehod), nesmí jejich detekování probíhat ex post. Tedy, jestliže chceme konflikty predikovat, musíme simulovat jízdu všech souprav vyskytujících v grafikonu. Taková simulace vyžaduje poměrně přesné fyzikální modely jak vozidel, tak i infrastruktury. Předmětem řešení není zajištění bezpečnosti – k tomu jsou určeny zabezpečovací zařízení. Ovšem cílem algoritmu je navrhnout řešení, které neobsahuje konflikty v jízdách vlaků, protože v opačném případě sice zabezpečovací zařízení za-

brání nehodě, ale tím pádem jedna souprava bude zastavena. Jako následek změní se upravený grafikon a je vysoká pravděpodobnost vzniku dalších odchylek.

Principiálně kolize detekujeme tak, že zaznamenáme dvě soupravy současně nacházející se (aspoň částečně) na stejném úseku. Abychom byli schopni odhalit tento stav, musíme mít informaci o tom, v jakém čase budou všechny soupravy na každém úseku své trasy. Pokud se podíváme na jízdní řády, tak uvidíme, že obsahují pouze stanice ležící na trase vlaku a časy příjezdu respektive odjezdu. Tím pádem z nich nelze přímo zjistit data o tom, jakými úseky vlak projede v určitém čase, tato informace se musí doplnit. K tomu je potřeba vědět vzájemnou polohu úseků, tj. mít grafový model infrastruktury. Dále potřebujeme na všech úsecích stanovit časy obsazení jednotlivými vlaky. Taková data nejsou dostupná a musíme je dopočítat, v podstatě simulovat jízdu. Narážíme na potřebu kompletního popisu infrastruktury včetně rychlostních omezení na jednotlivých úsecích. Vyplývá z toho první požadavek na navrhovanou aplikaci a to implementace modulu načítajícího infrastrukturu. Potom můžeme přistoupit k samotné simulaci, což představuje s sebou výpočet dynamiky železničního vozidla. Je to poměrně náročná úloha, protože samotná trakční charakteristika má složitý průběh a komplikuje velkým počtem různých druhů lokomotiv (s odlišnými charakteristikami) skutečně pohybujících se po síti. Nelze se však omezit při výpočtu jízdy vlaku pouze na dynamiky samotné lokomotivy, je vždy nutné vzít v úvahu složení celé soupravy (počet a typ vagónů). Tak vzniká další požadavek na návrh modulu vypočítávajícího dynamiku vozidel využívajících infrastrukturu.

Vrátíme-li se zpět k problému doplňování informace, kterými kolejovými úseky vlak projíždí v určitém čase, máme k dispozici informace o tom, jakými stanicemi vlak projíždí a celou síť elementů, ze kterých se infrastruktura skládá. Chceme-li doplnit všechny kolejové úseky mezi dvěma navazujícími stanicemi v jízdním řádu, bez potíží najdeme jejich vzájemnou polohu na infrastruktuře. Budeme pak hledat trasu mezi těmito stanicemi a v tento moment narazíme na problém, ve většině případů existuje více cest je spojujících. Trasa, po níž se vlak skutečně pohybuje, je stanovená jednoznačně, ale není explicitně popsaná, tj. klasické jízdní řády neobsahují dostatek dat nutný pro účely detekování konfliktu. Skutečným vstupem algoritmu bude rozšířený grafikon umožňující jednoznačně určit trasu. V každém případě vzniká další požadavek na algoritmus, je to modul, který bude načítat jízdní řády.

Chtěl bych shrnout primární požadavky na algoritmus:

- Načítání infrastruktury včetně vazeb mezi jednotlivými elementy
- Načítání rozšířeného grafikonu

- Detekování konfliktů
- Odstranění konfliktů prostřednictvím optimalizace nebo triviální úpravou grafikonu (pokud se povede)

2.2 Definování optimalizačních kritérií

Představme si situaci: díky zpoždění jedné ze souprav dva vlaky současně chtějí obsadit stejný kolejový úsek [1]. Tím pádem vzniká konflikt; jedním z očividných řešení je změna trasy jedné ze souprav, ale které? Na tuto otázku existuje více odpovědí, strategicky nejvhodnější dle mého názoru je rozdělení vlaků do tříd a zřejmě souprava z vyšší třídy bude mít přednost, pak pro druhý vlak se musí najít alternativní trasa. Takže všechny vlaky musíme zařadit do určité třídy. Z toho vyplývající otázkou je, na základě čeho bude rozřazení probíhat? Je zde opět více možností: základem může posloužit druh vlaku (osobní, nákladní, rychlík atd.), vazby na jiné soupravy, nabízet to jako komerční službu nebo jejich kombinace. Co se týče vazeb, je pod tím představována relace na navazující spoj, v tomto případě vlak, na který by čekala další souprava, by měl přednost. V případě komerční služby dopravcům by bylo nabídnuto zvýšit svoji kvalitu služeb prostřednictvím zaplacení poplatku a jejich vlaky by v případech konfliktu měli přednost. Dalším kritériem, jak můžeme určit přednost souprav, je jejich destinace. Nejdřív rozřadíme stanice do kategorií například dle počtu kolejí, potom vlak, jehož následující zastávka je uzlová stanice bude mít přednost před vlakem mířicím k malé zastávce.

Za kritérium můžeme také považovat energetické ztráty souprav. Například z důvodu konfliktu bude muset jeden z vlaků úplně zastavit, v tomto případě ztráty budou maximální (brždění a následující rozjezd). Účelem daného kritéria je maximálně omezit snížení rychlostí, protože rozjezd má největší energetické nároky. Jsou i jiné způsoby jak stanovíme, který vlak musí svoji trasu změnit. Například nalezneme optimální (dále pojem bude podrobněji rozebrán) alternativní trasy pro obě soupravy. Použije se ta varianta, která buď bude mít menší vliv na celý grafikon, nebo zpoždění soupravy bude menší, v nejlepším případě budou splněné obě podmínky. Tento způsob přijde na řadu také v případě, kdy soupravy budou patřit do jedné třídy nebo jejich cílová stanice bude spadat do stejné kategorie.

Pokud se podíváme na problematiku konfliktu trochu podrobněji, uvidíme, že existují méně závažné kolize. Například, na určitém úseku došlo ke konfliktu, ve kterém jedna souprava nárokuje na jeho obsazení v době, kdy druhé zbývá minuta k jeho opuštění. Takový konflikt je

[1] V rámci této práce je celá železniční síť rozdělená do úseků a každý je tvořen dvěma železničními návěstidly

výhodnější řešit v časové dimenzi, a to malým snížením rychlosti jedné ze souprav na několika předcházejících úsecích se můžeme kolizi úplně vyhnout. Proto hledání řešení nesmí brát, jako výhradně prostorová záležitost, protože časové posuny hrají při tom také velkou roli. Na závěr dané kapitoly bych chtěl definovat optimální trasu. Pod pojmem optimální si intuitivně představujeme nejkratší cestu, ale v kontextu daného problému má větší váhu doba jízdy (minimalizace případného zpoždění). Tyto dva pojmy na železnici nemusejí být ekvivalentní, protože delší trasa může mít vyšší rychlostní omezení a doba jízdy je potom kratší. Ve většině případů je výhodnější projet delší úsek, než odbočit ve výhybce, kde nemusí se tak snižovat rychlost.

Po zhodnocení všech výše uvedených úvah jsme dospěli k názoru, že nejvhodnější pro náš návrh bude použití kombinace výše uvedených metod. Při doručené zprávě například o zpoždění vlaku a detekci kolize na určitém úseku jedním z řešení je zpomalení jedné ze souprav na několika úsecích předcházejících konfliktnímu, další možnosti je náhradní trasa pro jednu respektive obě soupravy. Po každé úpravě grafikonu včetně této bude probíhat kontrola, zda se kolize ještě vyskytují. Pokud žádné detekované nejsou, algoritmus končí. Ve všech případech se nejdřív musí určit, pro jaký vlak se bude hledat objízdní cesta. Rozřadíme je do několika tříd dle priority: rychlíky, vlaky osobní a nákladní. Souprava vyšší třídy bude mít přednost a tím pádem se bude hledat trasa pro jejího „konkurenta“. Budou-li vlaky patřit do stejné třídy, náhradní cesta se nalezne pro oba a vybere se ta, při níž zpoždění „kolizní“ soupravy a všech dotčených bude minimální. Jelikož na nově nalezené trase mohou konflikty také vznikat, hovoříme o úsecích ovlivněných primární změnou grafikonu. Veškeré úvahy v této kapitole nenaznačují způsoby řešení detekovaných konfliktů. Jelikož nelze řešení vyhledat hrubou sílou, budou za tímto účelem použité heuristické metody, ve které mají stochastický přístup. To znamená, že výše popsána optimalizační kritéria budou mít nepřímý vliv a to tak, že řešení splňující určitá kritéria budou ohodnocená lépe, než ta co je nesplňují.

2.3 Možnosti parametrizace problému

První parametr, který musíme definovat, je měřítko řešené úlohy. Optimalizovat grafikon železniční sítě z hlediska rozsahu můžeme buď lokálně, nebo globálně. Přičemž pod pojmem globálně rozumíme celou síť ať už je to DSFD nebo území České Republiky. Co se týče lokálního přístupu, může to být jak část infrastruktury mezi dvěma stanicemi, tak i celá oblast zahrnující více stanic. Výhradně lokální přístup neumožní zaprvé nalezení nejlepšího řešení, zadruhé v některých případech nedokáže nabídnout žádné řešení konfliktu. Nejlepší řešení není možné te-

oretický najít, protože alternativní trasy můžeme hledat pouze v rámci vymezeného území a vlastní časové odstupy (zrychlení respektive zpomalení vlaku) můžeme vytvářet výhradně na úsecích nehraničících s jinou oblastí. Důvodem je to, že nesmíme vlastními změnami ovlivnit grafikon nezkoumané oblasti, protože tím může zapříčinit vznik nového konfliktu. V takovém případě nejlepším výsledkem hledání bude lokální optimum. Budeme-li řešit konflikt omezený dvěma stanicemi, nemusí v takovém případě existovat žádná alternativní cesta, tudíž nemáme šanci najít za daných okolností řešení. Při řešení konfliktu musíme být také opatrní, protože pokud řešíme pouze jednu kolizi a najdeme pro jednu z „kolizních“ souprav najdeme trasu značně lišící od původní, bude vysoká pravděpodobnost ovlivnit jiné vlaky a vytvořit tak další konflikt. Z toho vyplývá, že řešení máme hledat lokálně, pokud to jde, ale zohledňovat při tom celou síť.

Je potřeba také vymezit časový horizont hledání. Opět se podíváme na dva extrémy, tj. dlouhý a krátký časový interval. Pokud budeme navrhovat grafikon za delší dobu například jeden den je velké riziko vzniku mimořádné události, výsledkem je neplatnost vytvořeného modelu. Dalším nedostatkem je řádově vyšší výpočetní náročnost oproti krátkodobým predikcím. Pokud detekujeme konflikty v krátkém horizontu tak jsme schopni reagovat na všechny kolize, ale pořad bude vysoká výpočetní náročnost, kvůli tentokrát frekvenci spuštění algoritmu. Vhodnější je simulace s například dvou-hodinovým časovým intervalem, nabízí nižší zatíženi a relevantní výsledky. Švýcarská železnice disponuje systémem schopným predikovat ve stejném intervalu.

2.4 Složitost řešené úlohy

Výše byly stručně uvedeny nejpodstatnější komplikace problému rozvrhování. Pokud se podíváme pouze na vstupy algoritmu, uvidíme, že už v etapě načítání dat potřebujeme zpracovávat velký objem informace. Pro představu, železniční síť ohraničená na několik malých nebo středně velkých stanic bude mít tisíce kolejových úseku. Proto implementace navrhovaného algoritmu musí být ve velké míře zaměřená na efektivitu.

Přestavme si situaci, kdy díky zpoždění jedné soupravy vznikne několik kolizí a jednoduché zrychlení respektive zpomalení souprav nacházejících v konfliktu, nepřineslo žádaný výsledek. Musíme pak najít optimální náhradní trasy pro vybrané dle určitých pravidel vlaky. Což znamená, že pro každou takovou soupravu je potřeba prohledat všechny možné uzavřené cesty na daném úseku (například od „konfliktního“ úseku do nejbližší stanice). Při rostoucí vzdálenosti „konfliktního“ úseku od nejbližší stanice roste počet alternativních tras s exponenciální rychlostí. Tím pádem na více místech musíme řešit velmi známý problém obchodního cestujícího, který

patří mezi tzv. NP-těžké úlohy. Tato skutečnost vylučuje metodu hrubé síly. Vzhledem k tomu, že nalezení optimální trasy se velice podobá úloze obchodního cestujícího, je vhodné použít stejné nástroje. V praxi se obvykle používají heuristické metody zejména genetické algoritmy, které budou podrobněji rozebrané dále.

Kapitola 3

Evoluční výpočetní techniky

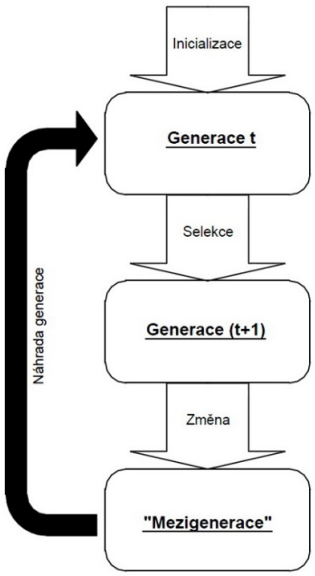
EVT řadíme ke stochastickým algoritmům, jejichž vlastností je práce s tzv. jedinci a jejich postupná evoluce. Pod pojmem jedinec se nejčastěji skrývá zakódované potenciální řešení konkrétní úlohy. Nepracuje se pouze s jedním jedincem, ale s celou populací jedinců vyjádřenou stejnou datovou strukturou. Algoritmus začíná téměř vždycky sestavením (vygenerováním) počáteční generace jedinců. Daný proces většinou se provádí náhodně s určitými omezeními. Dále v cyklu následuje ohodnocování kvality všech jedinců generace. Kvalitu jedince měříme výpočtem tzv. fitness (kriteriální) funkce (zároveň můžeme vyhledat nejlepšího jedince a statisticky vyhodnotit danou generaci). Poté probíhá selekce jedinců, což znamená výběr jedinců do nové generace obvykle na základě jejich kvalitativní (fitness) hodnoty. Navíc neurčujeme, zda bude konkrétní jedinec vybrán do další generace, ale pouze pravděpodobnost jeho přežití. Bude-li jedinec ohodnocen jako kvalitnější, znamená to, že má větší pravděpodobnost postupu (výběru) do následující generace. Tím teoreticky dosáhneme postupného zlepšení průměru ohodnocení generace. Následující fáze – změna, zahrnuje v sobě různé rekombinační operátory použité v závislosti na reprezentaci jedinců. Dva základní typy však se používají nezávisle na reprezentaci, jsou to mutace a křížení.

Mutace je unární operace pracující pouze s jedním jedincem, která podle příslušných pravidel mění jeho část.

Křížení tvoří jednoho či dva jedince (potomky) z nejméně dvou jedinců, takzvaných rodičů. Běžně generujeme potomky ze dvou rodičů a operace je tím pádem binární.

Postup algoritmu EVT je zobrazen níže.

Po změně generace se objevuje pojem „mezigenerace“ – je to generace, ve které proběhl proces selekce a je obohacená o jedinci vygenerované rekombinačními operátory. Tudíž obsahuje



Obrázek 3.1: Schéma algoritmu EVT

jak „rodiče“ tak i „potomky“. Za tím následuje proces náhrady generace, je to část algoritmu, jejíž součásti je vývojová strategie. Pod tímto pojmem rozumíme výběr jedinců ze stávající množiny, kteří budou v další iteraci tvořit novou populaci. Zpravidla je velikost populace po dobu běhu algoritmu konstantní.

Nejrozšířenějšími typy vývojové strategie jsou: generační a postupné. Podstatou generačního typu je úplná náhrada jedné populace populací následující. Při typu postupném mění se pouze část populace, která ve výsledku má v sobě jak rodiče, tak i potomky.

Všechny algoritmy EVT ukončují svou iterativní činnost na základě zastavovacího pravidla. Obecně podmínkou zastavení algoritmu je buď výsledek, který splní naše požadavky nebo maximální počet generací, respektive čas, během kterého musí být úloha vyřešena. Například máme za úkol najít nejkratší cestu mezi vrcholy A a B, zastavovací pravidlo můžeme nadefinovat, buď jako počet kilometrů, se kterým budeme spokojeni, nebo maximální čas běhu algoritmu, který jsme ochotni čekat na výsledek.

3.1 Genetické algoritmy

Významnou odlišností GA od ostatních EVT je reprezentace jedinců ve tvaru řetězců konečné délky, které se nazývají chromozomy. Je to posloupnost symbolů, ve které se každá pozice podle terminologii jmenuje alela. Element takového řetězce se nazývá gen. Genetickou informaci kódu-

jící chromozomy nazýváme genotyp a skutečná informace zakódovaná v konkrétním chromozomu je fenotyp.

Chromozomy fixní délky obsahující geny tvořené dvojkovou soustavou jsou základem SGA (standardní genetický algoritmus). Chromozom představuje zakódované řešení nějaké úlohy. Kvůli dvojkové soustavě použité v SGA pro reprezentaci jedinců musí algoritmus obsahovat dvě funkce navíc, a to jsou funkce kódování a dekódování. Podle složitosti řešení a hodnot, kterých může chromozom nabývat, musíme zvážit, zda je dvojkové kódování vhodné. Při náročném řešení se silně omezeným intervalem nabývajících hodnot chromozomu může funkce kódování stát zbytečným zdrojem chyb a zpomalit běh algoritmu, proto se volí kódování genů celými, resp. reálnými čísly.

GA se přidržují vývojového schéma zobrazeného výše, proto po vygenerování populací následuje její ohodnocení, neboli přiřazení kvality (fitness). Funkce hodnoticí jednotlivé jedince je zobrazením množiny jedinců do množiny reálných čísel.

3.2 Selekce

Účel selekce je dávat přednost jedincům s vyšší kvalitou. Popíšeme si několik nejrozšířenějších metod selekce.

Ruletová selekce a její definitivní rys spočívá v pravděpodobnosti přežití jedince, která je přímo úměrná jeho kvalitě. Osud jedince závisí na náhodném pokusu, ale čím je jedinec kvalitnější, tím je větší pravděpodobnost, že se dostane do další generace. Ukázalo se však, že toto schéma má řadu problémů, mezi něž patří velké vzorkovací chyby a nutnost mít relativně velkou populace pro správnou funkci.

Turnajová selekce je v dnešní době nejpoužívanější selekční strategie. Pro naplnění nové generace provede se turnaj tolikrát, kolik je rozměr populace. Podle předem určeného parametru n ze staré generace se vezme n chromozomů a nejlepší z nich postoupí do generace nové. Nastavením n určujeme selekční tlak – čím větší n, tím větší selekční tlak. V případě zvolení velkého n, například když n se rovná velikosti populace, by se nová generace skládala z nejlepších jedinců a jednalo by se o předčasnou konvergenci.

Pořadová selekce (ranking selection) se vyznačuje omezením předčasné konvergence a odstraněním problému se vzorkováním u malých populací. Na rozdíl od ruletové selekce pravděpodobnost přežití jedince souvisí pouze s jeho umístěním v posloupnosti, ve které jsou chromozomy seřazeny podle kvality.

3.3 Rekombinační operátory

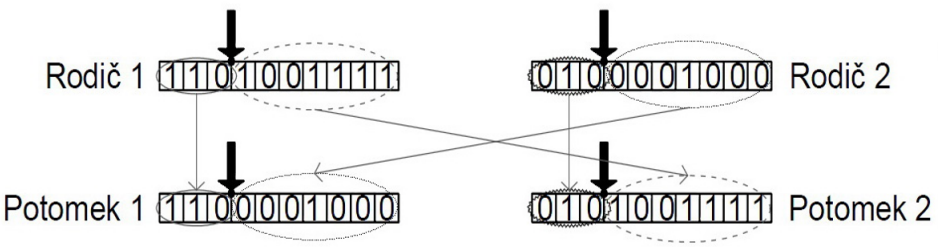
Selekce je důležitý parametr GA, má velký vliv na kvalitu výsledného řešení, ale jeho podstatou je pouze kopírováni některých jedinců do nové generace. K vytváření změn v původně vygenerované populaci slouží změnové neboli rekombinační operátory. K základním rekombinačním operátorům GA patří křížení a mutace. Na rozdíl od selekce jedinci nemusejí být ovlivněni změnovými operátory. Každý jedinec bude zasažen s určitou pravděpodobností. U křížení se nejčastěji jedná o pravděpodobnost v intervalu 0,6 až 1, pravděpodobnost mutace je vždy výrazně menší než pravděpodobnost křížení, běžně 0,05 až 0,15. Obecné schéma křížení je zachyceno na obrázku 3.2.

Obrázek 3.2: Křížení v GA

V SGA křížení kombinuje části pouze dvou rodičů, z nichž vytváří dva potomky. Pro určení jedinců, které podléhají křížení, provede se pokus nad každým jedincem generace a s předem nastavenou pravděpodobností bude zařazen do množiny, nad kterou se provede křížení. Proces křížení proběhne tak, že se z této množiny vybírá pár buď náhodně, nebo se definuje jako dva po sobě jdoucí jedinci. V případě lichého počtu se zkrátí množina o jeden prvek. Další možnou variantou implementace křížení je výběr jedinců do mezigenerace, která se mutuje a následovně nahrazuje starou generaci.

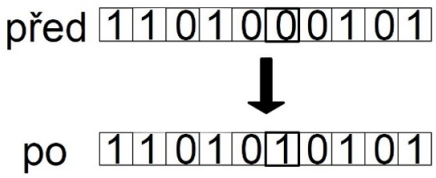
Implementace v SGA, kde jsou jedinci reprezentované bitovou posloupností, se provádí pomocí jednobodového křížení. Bod křížení se zvolí náhodně v jakékoli části posloupnosti na intervalu 1 až n-1, kde n je délka posloupnosti (jedince, chromozomu). Po zvolení bodu bude první potomek obsahovat bity ležící vlevo od bodu křížení prvního rodiče, bity vpravo budou převzaty od druhého rodiče. Druhý potomek naopak ve své levé části má bity zkopírované od druhého rodiče a v pravé od prvního. Na následujícím obrázku je znázorněn tento postup.

Druhým rekombinačním operátorem je mutace. Tento operátor pracuje s jedním jedincem a provádí změnu v jedné alele. Bude-li chromozom zařazen do množiny, nad kterou se provede mutace, náhodně se v něm vybere alela a změní se příslušný gen na opačný 0 na 1, respektive 1



Obrázek 3.3: Příklad křížení v SGA

na 0.



Obrázek 3.4: Chromozom před a po mutaci

Používá se také varianta průchodu celého chromozomu každého jedince, kde s určitou pravděpodobnosti je procházený gen invertován, v tomto případě musí být pravděpodobnost mutace výrazně nižší než v předchozím případě (doporučená hodnota p $<$ 0,1).

3.4 Diferenciální evoluce

Metoda hledající optimální řešení iterativním způsobem, snaží se v každé iteraci vylepšit dosud optimálnější nalezené řešení, s ohledem na definované měřítko kvality. Daná metoda patří k heuristickým algoritmům, které jsou schopné na základě malého objemu informace o řešeném problému najít relevantní řešení. Avšak tento heuristický algoritmus zaručí nalezení pouze lokálního optima, který se může značně lišit od optima globálního.

V jeho základní podobě, algoritmus lze popsat následujícím způsobem. Prvním krokem generuje se množina vektorů předem stanovené velikosti, která se jmenuje generace. Vektorem rozumí se body n-dimenzionálního prostoru, ve kterém je definována cílová funkce $f(x)$, kterou je potřeba minimalizovat. Během každé iterace algoritmus generuje novou generaci vektorů, náhodně kombinující vektory z předchozí generace. Počet vektorů v každé generaci je pevný a předem stanovený na základě předchozích zkušeností nebo omezený z časových důvodů.

Nová generace vektorů vytváří se následujícím způsobem. Pro každý vektor xi z předchozí generace jsou vybrány tři náhodné vektory v_1, v_2, v_3 ze stejné generace, a žádný z nich se nesmí rovnat x_i, dále je také generován mutovaný vektor dle vztahu:

$$v = v_1 + F * (v_2 - v_3)$$

kde F – jeden z parametrů metody, nezáporná konstanta z intervalu $[0, 2]$. Nad mutovaným vektorem v vykonává se operace křížení, která spočívá v tom, že některé jeho souřadnice jsou nahrazeny odpovídajícími koordinátami z původního vektoru x_i (každá koordináta je nahrazena s určitou pravděpodobnosti, je to další parametr této metody – pravděpodobnost křížení v literatuře označovaná jako CR). Výsledný vektor se jmenuje testovací (trial vector). V přídě lepšího ohodnocení tohoto vektoru kriteriální funkci, v nové generaci vektor x_i je nahrazen testovacím vektorem, v opačném případě x_i se zůstává. Hodnota parametru F má značný vliv na značný vliv na chování algoritmu stejně tak i pravděpodobnost nahrazení jednotlivých koordinát. Proto důležitou podmínkou pro správné fungování algoritmu jsou korektně zvolené konstanty. Existuje řada názoru na to v jakém intervalu se musejí tyto konstanty nacházet. Jsou i další způsoby určení těchto konstant nejjednodušší z nich je empirická metoda, která ale není praktická. Proto používají se sofistikovanější nástroje, metody adaptivního řízení těmito parametry. Jednou z nich je metoda samo-adaptace (self-adaptive), kde hodnoty F a pravděpodobnost nahrazení je buď následována od rodičů nebo náhodně vygenerované číslo. Diferenciální evoluce je vhodná pro numerické optimalizační problémy

3.5 Job-Shop rozvrhování

Job-Shop rozvrhování je optimalizační problém, ve kterém se přiřazují prostředky pro zpracování úkolů v příslušném čase. Pro základní pochopení si můžeme představit n úkolů, které je potřebné splnit v různých časových intervalech, ke splnění těchto úkolů se musí rozvrhnout m strojů s různým výrobním výkonem a minimalizovat při tom čas, který uplyne od začátku práce nad prvním úkolem do splnění posledního. Účelem této metody je optimalizace využití času a prostředků.

Kapitola 4

Využití evolučních technik při rozvrhování v železniční dopravě

4.1 Použití genetických algoritmů pro generování jízdních řádů

Ve svém článku [4] se autoři zabývají popisem problematiky rozvrhování v železniční dopravě. Samotné rozvrhování se skládá ze dvou základních částí: generování periodických (neboli cyklických) a neperiodických jízdních řádů. Jízdní řád je periodickým v případě, kdy má přesně stanovenou časovou frekvenci výskytu, v opačném případě je neperiodický. Například rozvrhování soupravy pohybující se každé úterý z Prahy do Ústí nad Labem je periodické, protože má týdenní frekvenci výskytu.

Pří návrhu algoritmu pro generování jízdních řádů se musí zohlednit několik druhů omezení: uživatelské požadavky (parametry rozvrhovaných vlaků), dopravní předpisy a topologie infrastruktury. K uživatelským požadavkům patří čas odjezdu, čas příjezdu a maximální zpoždění. Tyto veličiny nesmějí překračovat tolerovaný interval. Mezi dopravní omezení patří doba jízdy mezi navazujícími stanicemi, křížení – jízda dvou souprav po jednokolejné trati, při níž jedna z nich se pohybuje v protisměru. Doba zastavení, kdy všechny vlaky mají předem určenou dobu stání ve stanici, která v součtu s časem příjezdu nesmí překročit čas odjezdu. Dále je zakázané předjíždění na jakémkoli kolejovém úseku. Také se vyskytuje omezení časového zpoždění vznikající mimo jiné z důvodu dávání přednosti při předjíždění nebo křížení s jinou soupravou. Doba příjmu – časový rozdíl mezi dobou příjezdu do stanice dvou navazujících vlaků, je definována pro vlak, který se objeví ve stanici dříve. Výpravní doba – definována pro dva navazující vlaky ve stejné stanici jako absolutní hodnota rozdílu času odjezdu prvního vlaku a příjezdu

druhého, je určená pro druhý vlak. Současný odjezd dvou vlaků ve stejné stanici do protisměru, musí být posunut o předem danou dobu.

K omezením infrastruktury patří konečná kapacita stanic, souprava může do stanice přijet pouze v případě, jestliže je ve stanici aspoň jedna volná kolej. Časové intervaly, ve kterých dochází k údržbovým zásahům ve stanicích a souprava v nich nesmí zastavovat, pouze projíždět nebo v závislosti na velikosti zásahu nemůže ani projíždět. Časový odstup pro odjíždějící vlaky ze stejné stanice ve stejném směru nesmí být záporný a počítá se jako rozdíl času odjezdu vlaku, který přijel na stanici dřív, jiným slovy nesmí dojít k předjíždění ve stanici. Výše uvedené požadavky platí pouze pro neperiodicky vyskytované jízdy.

Studie, která vznikla na Technické Univerzitě Valencie [4] se zaměřuje na jednu specifickou úlohu, a to vložení dodatečných neperiodických jízdních řádů do daného grafikonu s periodickými jízdními řády. Prvním krokem při rozvrhování jízdních řádů je započítávání souprav pohybujících se po určitých úsecích pravidelně, což klade další omezení na rozvrhování neperiodických jízd. Výše uvedené omezení v plném rozsahu kvůli dopravním předpisům platí pouze pro Španělsko.

V daném návrhu je vygenerované řešení ohodnoceno na základě času zpoždění všech vlaků na síti s ohledem na odchylku od každé soupravy od její optimální doby jízdy.

Proces samotného řešení je založen na základě Job-Shop přístupu. Pro implementaci této metody byl každý vlak rozdělen do řady úkolů, kde jednotlivý úkol představuje přemístění soupravy mezi dvěma body infrastruktury. Skládá se ze dvou částí: vlaku a kolejového úseku, který náleží množině všech úseků trasy příslušné soupravy.

Při generování jízdních řádů se používá standartní schéma genetických algoritmů. Jedinci jsou kódováni pomocí výše zmíněných úkolů, tj. obsazenost kolejového úseku daným vlakem. Chromozom potom představuje sebou řetězec takových úkolů, kde jsou zachycené všechny vlaky a kolejové úseky, kterými tyto vlaky projíždějí.

Počáteční populace se generuje náhodně, ale při tom se vytvářejí jedinci z celé škály ohodnocení (jak nízko tak i vysoko ohodnocení jedinci), kromě toho jedinci, kteří mají větší zpoždění, dostávají větší prioritu (tato metoda je pojmenována jako Regret-Based Biased Random Sampling neboli RBRS). Důvodem vybraného způsobu generování je větší rozmanitost populace, zpravidla vykazující lepší výsledky. Implementovaný genetický algoritmus ve [4] používá jenom klasické rekombinační operátory: křížení a mutaci. Křížení je klasické jednobodové. Mutace je implementovaná netradičním způsobem, s pravděpodobnosti 0.05 každý úkol je přemístěn na jinou pozici v chromozomu. Jako selekční mechanismus byla použitá osvědčená turnajová selekce.

Algoritmus byl otestován na železničním úseku mezi Madridem a Jaénem, trať pokrývá 54

stanic a má celkovou délku 369 kilometrů. Grafikon byl generován pro dvanáctihodinový interval, autoři popisují hardware, na kterém byl program spouštěn, pouze generací a frekvencí procesoru stroje - byl to Pentium 4 s frekvencí 3.6 GHz. Dle uvedených výsledků byl splnitelný grafikon nalezen po pěti minutách od začátku běhu programu. Algoritmus se testoval také na několika dalších problémech, jednalo se o úseky s délkou řádově stovky kilometrů, desítky stanic, desítky periodických i neperiodických vlaků a stovky kolejových úseků, pro které je potřeba vytvořit grafikon. Řešení se porovnávalo s náhodně vygenerovanými jízdními řády, bohužel autoři neuvádějí, zda takové generování bylo něčím omezeno. Genetický algoritmus se také porovnával s řešením obdrženým metodou RBRS. Porovnávané výsledky byly získané za stejný výpočetní čas. Pomocí navrženého algoritmu vygenerované jízdní rády vykazovaly nejlepší výsledky v porovnání s ostatními metodami. Algoritmy se oceňovaly podle odchylky od optimálního jízdního řádu (v daném článku je to nejkratší možná jízdní doba pro každou soupravu) a GA vygeneroval grafikon s nejnižší průměrnou odchylkou.

Výsledky studie potvrzují předpoklad o tom, že GA je vhodným nástrojem při řešení takového druhu úloh.

4.2 Diferenciální evoluce s dvojitou populaci použitá pro řešení problému periodických JŘ na železnici

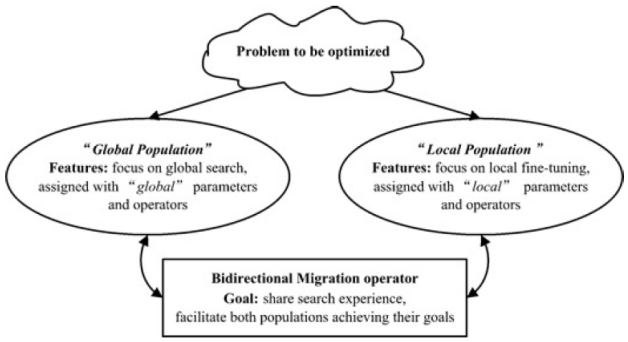
V této části práce popíši výzkum čínských odborníků v oblasti umělé inteligence [5]. Studie se zabývá problémem pravidelného plánování železničních jízdních řádů (periodic railway timetable scheduling neboli PRTS), cílem kterého je minimalizace průměrné doby čekání přestupujících cestujících. Nástrojem k dosažení daného cíle je rozvrhování časů odjezdu všech vlaků v každé stanici. Na rozdíl od tradičních PRTS modelů, uvažujících pouze o linkách s pevně danou dobou dopravního cyklu, tato studie nabízí flexibilnější model umožňující použití odlišných časů dopravních cyklů v závislosti na počtu přestupujících cestujících. Rozšířený algoritmus diferenciální evoluce (DE) s dvojitou populací, nazvanou „dual-population DE“ (DP-DE), byl vyvinut pro řešení PRTS problému. V DP-DE dvě populace spolupracují během evoluce. První je zaměřená na globální vyhledávání, řízení evolučních parametrů a selekční strategií pomáhající udržovat rozmanitost populace, zatímco druhá je zaměřená na regulaci ve směru vyhledávání lokálního optima. Nový dvousměrný migrační operátor nabízí sdílení vyhledaných zkušenosti mezi dvěma populacemi. Navrhována DP-DE byla použitá pro optimalizaci grafikonu kantonského metra v Čině a v několika dalších umělých železničních systémech.

4.2.1 Definování problému

Předpokládá se síť obsahující D linek $l_1, l_2, \ldots, l_D$, kde vlaky na všech linkách odjíždějí ze startovní stanice periodicky. Jestliže dvě linky obsluhují stejnou stanici, cestující mohou přestoupit z jedné linky na druhou. Tato stanice je definována jako „přestupní". Podmnožinou všech stanic je tedy množina přestupných stanic M $O_1, O_2, \ldots, O_M$. Cílem PRTS je zajištění optimálního času odjezdu všech vlaků v každé stanici a zkrácení průměrné doby čekání přestupujících cestujících.

4.2.2 Diferenciální evoluce s dvojitou populaci

Navrhovaná metoda obsahuje dvě populace s různými účely a s dvousměrným migračním operátorem. Jak je ukázáno na obr. 4.1, první populace se jmenuje globální populace (GP), je zaměřena globální hledání řešení a proto její parametry a operátory jsou nastaveny tak, aby rozmanitost populace byla maximální. Druhá populace se jmenuje lokální (LP) a zaměřená na hledání lokálního optima a má patřičně stanovené operátory a parametry.



Obrázek 4.1: Schéma diferenciální evoluce s dvojitou

Dvousměrný migrační operátor umožňuje předání nejlepších jedinců mezi oběma populacemi, s cílem zabezpečení vysoké rozmanitosti populace GP a zároveň rychlou konvergenci LP. Dále bude následovat stručný popis implementace algoritmu. Jedinci jsou kódováni jako vektory čísel s plovoucí řádovou čárkou:

$$X_i^g = [x_{i,1}^g, x_{i,2}^g, \ldots, x_{i,D}^g],$$

kde g je číslo generace evolučního procesu, i je index jedince, D je dimenze problému (tj. počet linek), $x_{i,j}^g$ – doba jízdy j-té linky.

Generování jak lokální tak i globální populace je náhodné, ale z definovaného intervalu pro každou linku, jehož hranice nebyly v článku popsané. Obnovení „globální populace" je zaměřeno

na udržování rozmanitosti, největší roli hrají konstanty F a CR, pro generování jejích hodnot byly zvoleny následující schémata:

$$F = rand(0.01, 1),\ CR = rand(0, 1).$$

Vysoká pravděpodobnost křížení zaručuje to, že se populace zůstává heterogenní. Cílem „lokální populace“ je rychlá konvergence řešení, proto pravděpodobnost křížení je nulová, konstanta F je stanovená jako náhodné číslo s normální distribucí. Obousměrný migrační operátor sdílí zkušenosti GP a LP populací. Při tomto procesu jsou vybráni nejlepší a nejhorší jedinci z každé populace. Jestliže označíme chromozomy GP písmenem A, populace LP písmenem B a jejich ohodnocení fitness funkci $f(A)$ respektive $f(B)$, budou platit následující rovnice pro směr GP →LP:

$$\text{if } f(A_{best}) < f(B_{best}) \text{ then } B_{best} \leftarrow A_{best},$$
$$\text{else if } f(A_{best}) < f(B_{worst}) \text{ then } B_{worst} \leftarrow A_{best},$$

Při splnění daných podmínek bude nahrazen nejlepší jedinec GP. Pro směr GP ←LP platí následující podmínka:

$$\text{if } f(A_{best}) > f(B_{best}) \text{ then } A_{best} \leftarrow B_{best}$$

Algoritmus byl testován na síti metra města Kanton, skládající se z 16 linek a 12 přestupních stanic. Pro vyhodnocení účinnosti se výsledky porovnávají s šesti dalšími metodami umělé inteligence: metoda větví a mezi (dále označována B&B), hladový algoritmus (GH), genetický algoritmus (GA), optimalizace hejnem částic (CLPSO), diferenciální evoluce (DE), diferenciální evoluce s volitelným externím archivem (JADE), diferenciální evoluce se strategií generování kompozitních zkušebních vektorů (CoDE), kovariantní matice strategii adaptace (CMA-ES) a mnoha populační diferenciální evoluce (MPDE). Všechny metody byly naprogramované v prostředí Visual C++ 6.0 a spouštěly se na stroji s procesorem Intel Core 2 Quad s frekvenci 2.4 GHz a 1.96 GB paměti. Všechny simulace spouštěly se nezávisle 30 krát počet iteraci byl omezen na 10 000.

Z uvedeného ve studii grafu vyplývá, že řešení s nejkratší dobou čekání bylo nalezeno právě pomocí diferenciální evoluce s dvojitou populací. Další pozorování, stojící za upomínku, jsou výsledky ostatních metod, které měly sice horší výsledky, ale v některých případech se jedná pouze o několik procent. Bohužel autoři uvádějí, že konstanty ostatních metod byly zvoleny na základě podobných výzkumů, což by dle mého názoru nemělo znamenat, že zvolené parametry

jsou optimální pro daný konkrétní případ. Tím pádem lze teoreticky dosáhnout lepších výkonů i u ostatních metod. Algoritmus má i svoje očividné výhody oproti například metodě MPDE vývoj je neustálený, kdyžto po nalezení MPDE nejlepšího řešení dosáhne se lokálního maxima a faktický vývoj končí.

Dále byly algoritmy testovány na deseti linkách a při různé vytíženosti metra, tj. během špičky, sedla a průměrné zatíženosti.

Při testování na jiných infrastrukturách celková tendence je stejná, jako v prvním testovacím příkladu. Navrhována metoda určitě bude vhodná pro použití v optimalizačních úlohách, kde by se mělo vyhýbat lokálním optimům. Na základě uvedených simulaci nelze jednoznačně říct, že tento algoritmus je nejefektivnější, protože chybí podrobnější popis metod, se kterými byl porovnáván. Je také možné, že díky vyladění jejích parametrů by se mohlo dosáhnout lepších výsledků.

Výsledky výzkumu ukázaly, že výběr heuristických metod je správný směr hledání řešení problému podobného druhu. Není zatím výsledné řešení znatelně lepší v některých případech, což svědčí o tom, že je potřeba opírat se na získané zkušenosti, zlepšovat navržené řešení. V případě heuristických metod a obzvlášť u GA lepších výsledků občas lze dosáhnout úpravou algoritmických konstant (např. pravděpodobnost křížení, velikost populace atd.), který se bohužel mohou stanovit pouze empiricky. V práci bylo také vyzkoušeno málo rekombinačních operátoru a jenom jedna selekční strategie.

Kapitola 5

Řešení

Chceme-li optimalizovat grafikon, první otázka vznikající na etapě návrhu, je rozsah sítě, se kterou budeme pracovat. Velikost infrastruktury v značné míře ovlivňuje požadavky na algoritmus. Tak prvním krokem vlastního návrhu bylo zvolení infrastruktury.

Při popisu infrastruktury byly ve spolupráci s doktorandem Ing. Adamem Hlubučkem analyzovány v současné době dostupné metody, používané v podmínkách ČR i nově vyvíjené především v západní Evropě. Sdružením dopravců, správců infrastruktury, univerzit a firem zapojených do tvorby telematických aplikací, návrhu dopravních modelů a simulací, vznikl standard pro výměnu informací, nazývaný railML. V současné době platná verze railML 2.2 naráží na problematiku jednoznačně popsané infrastruktury, proto pod záštitou UIC nově vzniká relační model TopoModel, na jehož základě je vyvíjena nová kompatibilní verze railML 3.0.

V době prováděných analýz nebylo ještě možné využít rozpracovanou verzi railML 3.0, zároveň bylo rozhodnuto jako nepraktické věnovat značné úsilí k popisu kolejiště DSFD neperspektivní verzí railML 2.2. Ani stávající metody popisu infrastruktury v ČR nebyly zhodnoceny jako perspektivní.

Z tohoto důvodu bylo rozhodnuto využít stávajícího, nestandardizovaného popisu infrastruktury DSFD, vytvořeného Ing. Petrem Kouteckým v rámci bakalářské a diplomové práce, za účelem konfigurace elektronického stavědla či automatického vedení vlaku, realizovaného v DSFD. Protože původní účel tohoto popisu infrastruktury byl jiný, bylo zapotřebí provést jeho úpravu podle potřeb pro optimalizaci řízení provozu.

Dopravní sál představuje laboratoř pro návrh a vývoj železničního zabezpečovacího zařízení. Hlavním dílem sálu je dostatečně přesný model kolejiště, umožňující simulovat reálnou infrastrukturu. Kolejiště tvoří hlavní trať se čtyřmi dopravnami s kolejovým rozvětvením (žst.

Strančice, žst. Senohraby, žst. Čerčany a odbočka Pyšely), která je doplněna o regionální trať se dvěma dopravnami s kolejovým rozvětvením (žst. Sedlnice a žst. Davle). Tato řízená oblast je oboustranně napojená na dvě odstavná nádraží, která jsou vzájemně propojená a umožňují přistavování vlaků do řízené oblasti podle naplánovaného grafikonu vlakové dopravy. Obě odstavná nádraží mají zřízenu vratnou smyčku pro možnost otáčení vlakových souprav.

5.1 Postup řešení

Návrh nového grafikonu můžeme rozdělit na několik základních etap: simulace jízdy vlaku, detekce konfliktů a řešení konfliktů. Pro výpočet jízdy vlaku je potřeba navrhnout fyzikální model dynamiky, tj. definovat fyzické vlastnosti jako zrychlení a odrychlení. Při analýze bylo zjištěno, že samotná problematika trakční charakteristiky vlaků překračuje rámce diplomové práce, proto bylo rozhodnuto reálnou charakteristiku aproximovat. Zjednodušení spočívalo v použití konstantních hodnot zrychlení a odrychlení. Jakákoli simulace je modelování reálných procesů s určitou přesností, v našem případě požadavky určuje metoda detekce konfliktů.

Pro připomenutí se zmíním o tom, že trasa soupravy se skládá z kolejových úseků a výhybek stanovené délky. Při výskytu odchylky od grafikonu musíme zkontrolovat, nedošlo-li ke kolizi na nějakém úseku, tj. potřebujeme zjistit, jestli se dvě nebo více souprav nebude nacházet současně na jednom úseku. Abychom byli schopni kontrolu udělat, potřebujeme vědět časové intervaly na všech úsecích, ve kterých je úsek obsazen. Konflikt bude potom detekován v případě průniků dvou nebo více intervalů na jednom úseku. Tak pro všechny soupravy musíme dopočítat časy příjezdu a odjezdu na jednotlivých úsecích jeho trasy. Budeme k tomu potřebovat data, jak o infrastruktuře tak, i o vozidlech. Co potřebujeme vědět o infrastruktuře: délky úseku, rychlostní omezení a případný sklon. Pak musíme znát dynamické vlastnosti soupravy, které vyplynou z konkrétního typu lokomotivy a hmotnosti tažených vagonů, nezanedbatelnou je i délka soupravy.

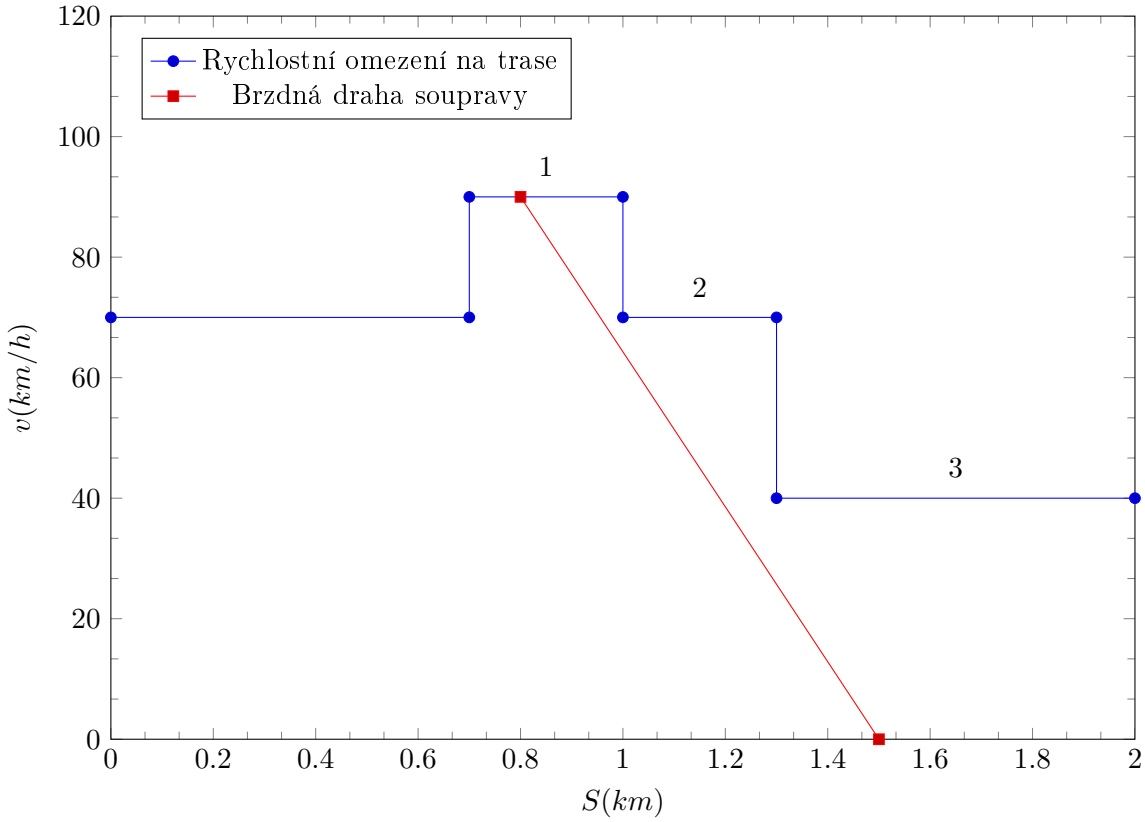
Můžeme se vrátit k simulaci, je teď definovaný její výstup, časy obsazení a uvolnění soupravou všech úseků jeho trasy. Na první pohled máme jednoduchou úlohu, známe zrychlení respektive odrychlení, čas odjezdu ze zdrojové stanice a čas příjezdu do cílové stanice. Potřebujeme tento interval rozdělit mezi všemi úseky, kterými vlak projíždí. Ale při implementaci bylo zjištěno několik komplikací. Začneme iterativním postupem výpočtem času uvolnění na všech úsecích. Čas uvolnění definujeme jako čas, ve kterém se konec posledního vagonu bude nacházet na začátku navazujícího úseku. První problém, na který narazíme, že my obecně nevíme, jak se vlak na úseku pohybuje, jestli zrychluje, jede s ustálenou rychlostí nebo zpomaluje. Nicméně můžeme

říci, že pokud vlak nedosáhl maximální povolené rychlosti na úseku tak se bude zrychlovat a až ji dosáhne, tak pojede ustálenou rychlostí, v případě nižší maximální rychlosti na navazujícím úseku od určité vzdálenosti začne zpomalovat, aby jí nepřekročil. Bohužel tento teoretický předpoklad nefunguje ve skutečnosti, protože ve většině případů zjistíme, že soupravě ke snížení rychlosti nestačí jenom jeden úsek. V případě popisované situace vypočítáme vzdálenost potřebnou k zpomalení a budeme porovnávat jí s délkou úseku, na kterém se vlak nachází, případně i s přecházejícími úseky, tak zjistíme, že máme vrátit se zpátky na několik úseků. Tak teď víme, kdy máme začít brzdit, ale narazíme na další problém, pokud postupujeme iterativně tak neznáme, s jakou rychlosti souprava pohybovala se dřív. Tím pádem máme vypočítanou brzdnou dráhu na základě počáteční rychlosti ne úseku, ze kterého bude souprava skutečně brzdit, ale ze stávajícího. Proto nalezená rychlost neodpovídá reálné, dostáváme se k tomu, že abychom našli skutečnou brzdnou dráhu, musíme nejdřív zjistit počáteční rychlost, můžeme jí ale zjistit pouze v přídě informace o místě, kde je potřeba brzdit. Ve výsledku máme jednu rovnici s dvěma neznámými.

Zřejmým bylo to, že nelze současně spočítat brzdnou dráhu a počáteční rychlost. Tak bylo rozhodnuto předpočítávat místa, kde vlak musí rychlost snižovat a tuto informaci ukládat k již načteným datům o infrastruktuře. Princip navrhované metody spočívá v průchodu všech kolejových úseků od konce trasy a porovnávání jejích rychlostních omezení, jestli (v obraceném pořadí) rychlost v navazujícím úseku je nižší to znamená, že je potřeba vypočítat brzdnou dráhu. Zvláštním případem bude poslední kolejový úsek, kde bude explicitně definovaná nulová maximální povolená rychlost (vlak se musí zastavit). Jak bylo, jíž zmíněno většinou nestačí soupravě pouze jeden úsek ke snížení rychlosti, pro přehlednost popisované metody je uveden následující graf 1.

Na grafu je zobrazené: zastavení vlaku ve stanici (červeně) a změna rychlostí na jednotlivých úsecích. Jak bude popsáno podrobněji dále, všechny kolejový úseky obsahují několik druhů rychlosti, výhybky navíc obsahují rychlost odbočení, proto si musíme explicitně definovat, jaká rychlost platí pro danou soupravu. Rychlost zobrazena v grafu platí pouze pro vybraný druh vlaku a vybranou trasu.

Vidíme, že pro zastavení vlak potřebuje tři kolejové úseky, toto musíme vypočítat algoritmicky. Jak bylo zmiňováno výše, procházíme úseky od konce. Poslední úsek je cílovou stanici proto neplatí omezení dané infrastrukturou, ale explicitně definovaná nulová rychlost. Dané omezení bude platit ne na celém úseku, ale pouze v místě nástupiště, tj. musíme nejdřív dopočítat, s jakou rychlosti musí souprava na tento úsek vjet, aby stihla zastavit se v potřebném místě.



Graf 1. Ukázka brzdné dráhy vlaku

Předpokládáme, že vlak vjede na úsek s maximální povolenou rychlostí 40 km/h rychlost a potřebuje dosáhnout nulové rychlosti. Zjistíme, že brzdná dráha přesahuje délku části úseku, kde se máme zastavit. Tak definujeme vlastní povolenou rychlost k tomu, aby se souprava stihla zastavit na potřebné vzdálenosti v našem případě je to 33 km/h. Teď porovnáváme rychlostní omezení třetího a druhého úseku. Zjistíme opět, že rychlost předchozího úseku je větší ($70 > 33$ km/h), což pro nás znamená, že budeme muset brzdit. Brzdná dráha přesahuje délku úseku č. 2, proto pro ní definujeme rychlostní omezení 66 km/h. Přejdeme k následujícímu úseku, máme snížit rychlost z 90 na 66 km/h, ale v tomto případě délka úseku je postačující tak, že teď pouze definujeme místo, kde vlak musí začít brzdit, v našem případě to budou 83 metry od začátku úseku.

Ke každému úseku doplníme atributy maximální vjezdové rychlosti a vzdálenosti, ve které souprava musí začít brzdit. Pokud jsou oba atributy nulové to znamená, že není potřeba na úseku brzdit. V případě nenulové rychlosti, musí vlak začít snižovat rychlost, jíž od začátku úseku, má-li úsek nenulový atribut brzdné vzdálenosti, začne souprava brzdit v této vzdálenosti.

Dostali jsme se k tomu, že potřebujeme určit, jestli na příslušném úseku souprava bude

zrychlovat nebo jet ustálenou rychlostí. Další komplikace dosud nepopsaná je vliv délky vlaku na rychlostní omezení. Vjíždí-li souprava na kolejový úsek s vyšší rychlosti oproti předchazejícímu, začíná pro ní platit rychlost uvedená na návěsti ne v době průjezdu čela, ale konce posledního vagonu za výjimkou výskytu speciální doplňující návěsti. V síti jsou kolejové úseky se vzdáleností od začátku do konce 50 metrů, všechny výhybky mají stejnou vzdálenost, délka soupravy běžně překračuje 200 metrů. Představme si několik po sobě jdoucích takových úseků a soupravu s délkou 200 metrů, když bude vlak těmito úseky projíždět, začne pro ní platit návěst nařizující rychlost z již opouštěného lokomotivou úseku. Při iterativním průchodu máme informaci pouze o rychlosti v úseku, kde se nachází lokomotiva, případně sousedních. Jako následek nevíme, jaké omezení platí v danou chvíli pro soupravu. Tak byla zavrhnuta iterativní metoda.

5.2 Programové moduly

Před samotnou simulaci byla potřeba shromáždit informace o infrastruktuře, vlaku a jízdních řádech. Abychom mohli s daty manipulovat, museli jsme je z příslušných zdrojů načíst. Informace o železniční sítí je uložena do XML souboru (podrobnější popis bude rozebrán dále), proto byl vytvořen modul pro jeho analýzu. Z přečtené infrastruktury byl vytvořen interní model představující sebou orientovaný graf, kde uzly jsou: návěstidla, izolované styky, kolejové úseky a výhybky; orientace hran je stanovená na základě přečtené vazby.

Jak už bylo napsáno v popisu problému, klasické jízdní řady neumožňují jednoznačně určovat trasy souprav, proto bylo vyžadováno je rozšířit, tak Ing. Petr Koutecký při spolupráci s Ing. Bc. Dušanem Kamenickým navrhli upravenou variantu SQL schématu obsahujícího grafikon. Dopravní sál pro vlastní potřeby ukládá jízdní řády v databázi MySQL, proto vznikl požadavek na vytvoření modulu připojujícího a komunikujícího s databází. Model vlaku z hlediska dynamických charakteristik je celkem složitá úloha značně překračující rámce diplomové práce, proto základní charakteristiky byly implicitně definovány ve zdrojovém kódu. Pro navazující práci je vhodné vytvořit samostatný modul popisující jízdu vlaku. Detailně vytvořené moduly a použité nástroje jsou popsané v implementační části.

5.3 Optimalizační kritéria

Jedním ze vstupů algoritmu bude grafikon obsahující odchylky. Cílem je vytvoření nekonfliktního jízdního řádu, který bude brát ohled na tyto odchylky, a provedené změny by měly být minimální. Primárně se zjistí, je nutné bude upravovat grafikon detekováním konfliktů, pokud

žádné nebudou, tak se předpokládá, že původní JŘ je optimální, tím pádem ho nelze vylepšit a nebude se do něho zasahovat. V případě výskytu aspoň jedné kolize se bude grafikon editovat. Vzhledem k tomu, že my nedokážeme odhadnout rozsah konfliktu, efektivním může být jak jednoduché zkrácení respektive prodloužení časových intervalů průjezdů jedné z konfliktních souprav na přecházejících úsecích, tak i komplexnější zásah zahrnující změnu původně naplánované trasy. Tak budeme muset vybrat vhodné řešení stanovené na základě určitých kritérií, které máme definovat. Jak bylo napsáno výše, vlaky budou rozděleny do několika tříd dle přednosti při řešení konfliktu. Řešení budeme hodnotit na dvou úrovních: celého grafikonu a lokálně řešeného konfliktu, přičemž první bude významnější.

Při řešení kolize prvním krokem najdeme všechny alternativní trasy pro oba vlaky. Na této rozlišovací úrovní optimálním bude řešení s nejkratší dobou jízdy soupravy nižší třídy. Upozorním na to, že ho nebudeme hledat metodou hrubé síly (jedná se o NP-těžkou úlohu), proto našim úkolem je určit jaké řešení nás víc uspokojí. Tak cesta s menší jízdní dobou bude lépe váhově ohodnocená, než s dobou delší, zřejmým je i vliv třídy na ohodnocení.

Každé nalezené řešení může v různé míře ovlivňovat ostatní vlaky nacházející se na síti. Například jedna alternativní cesta může vyvolat další konflikty, zatímco druhá nezpůsobí žádnou. Z daného hlediska má řešení větší dopad na systém jako takový. V souladu s tím kvalitnější (způsobující co nejmíň kolizi) trasa na této úrovni bude mít vyšší váhové ohodnocení než na úrovni lokální.

Na základě výše uvedených úvah můžeme stanovit kriteriální funkci, která bude ohodnocovat řešení nalezené genetickým algoritmem. Návrh grafikonu, kromě vyřešení logistických úkolů, vede k efektivnějšímu využíti energie a vytíženosti infrastruktury. Při úpravě jízdních řádů se budeme snažit tyto cíle splnit s ohledem na kvalitu služeb poskytovanou železnicí. Budeme snižovat zpoždění jednotlivých vlaků ve stanici, jinými slovy kriteriální funkce bude definována jako vážený součet všech zpoždění.

Kapitola 6

Návrh genetického algoritmu

Řešená problematika v rámci této diplomové práce patří mezi tzv. NP-těžké úlohy, kde evoluční techniky jsou výkonnými nástroji schopnými ji řešit. Celkový počet odlišných tras na železniční síti je obrovský, proto nemůžeme hledat alternativní trasu pro soupravu metodou hrubé síly. Ale počet cest mezi dvěma stanicemi je poměrně malý (od jedné do několika cest). Proto by bylo neefektivní prohledávat množinu všech cesty, vhodnějším je primárně řešit konflikt lokálně. Ne vždy může takové řešení existovat, například v situaci, kdy konflikt vznikne na mezi dvěma stanicemi spojených pouze jednou cestou. Můžeme se pokusit takovou kolizi řešit na jiných mezistaničních úsecích, ale vzniká otázka, v jaký čas se tam budou konfliktní soupravy nacházet. Například je možná situace (uvažujeme pořád stejný případ konfliktu), ve které nalezneme alternativní trasy pro obě soupravy spočívající v použití různých koleji ve stanici nacházející před konfliktním úsekem. Při simulace zjistíme, že konflikt vznikne na stejném úseku, ale v jiné části, kvůli změnám dob jízdy souprav. Ale pokud by se jedna souprava zpomalila, vyhnuli bychom se této kolizí díky změně tras obou vlaků a rychlosti jedné z nich. Bohužel nemůžeme takové časové změny odhadnout, protože je to také NP-těžká úloha. Dostáváme se k tomu, že problém rozvrhování musíme řešit ve dvou dimenzích: prostorové (alternativní trasy) a časové (okamžik obsazení respektive opuštěni kolejového úseku).

Pro zmíněné hledání tras jsou vhodným nástrojem genetické algoritmy umožňující v konečném čase najít buď správné nebo uspokojivé řešení. Vzhledem k tomu, že řešení potřebujeme hledat ve dvou dimenzích, kde sena každou kladou různé požadavky, navrhuje se použit genetický algoritmus se dvěma populacemi. První populace bude zodpovědná za hledání alternativních tras a druhá za úpravu časových intervalů úseků, na kterých vlak se bude nacházet.

Ke zvýšení pravděpodobnosti nalezení optimální trasy ne bude hledání úplně náhodné. Pri-

márně se vytvoří ohraničená množina alternativních cest, omezená bude, protože úplná množina je příliš rozsáhlá (jedná se o velký počet kombinaci i v případě malé síti). Proto pro danou infrastrukturu budou nalezené všechny cesty mezi sousedícími stanicemi. Řešení pro jednu populaci by se pak hledalo v této množině. Máme-li pouze jednu kolizi v grafikonu, budeme pak hledat alternatívní trasy pro dvě soupravy. Jedinec se bude skládat ze dvou tras, jedna pro každý vlak. V první populaci bude několik druhů jedinců: obsahující alternativní trasu pro první soupravu (druha bude mít původně naplánovanou), obsahující alternativní trasu pro druhou soupravu, alternativní trasy pro obě soupravy a jedinec představující sebou původní cesty pro oba vlaky. Každý jedinec v této populaci bude sestavován na základě stanic, kterými musí souprava projet, budou se mezi nimi náhodně vybírat trasy z předpočítané množiny alternativních cest s tím, že první generaci bude aspoň jeden jedinec každého druhu.

Druhá populace (časová populace nebo ČP) by se zabývala výhradně časem stráveným vlakem na jednotlivých úsecích. Nejedná se o přesné definování časových intervalů, jejich náhodné generování by neumožnilo nalezení smysluplných řešení. Jsou to časové posuny již definovaných nebo vypočítaných tras pomocí simulace. Rozsah posunu bude předem definován, například od nuly do dvou minut. Tak každý gen bude představovat číslo. Posun bude přiřazován všem úsekům s určitou pravděpodobností.

Dvě tyto populace budou mezi sebou spolupracovat. Nejdřív, podle výše popsaných pravidel bude vygenerována prostorová populace (PP). Dále se bude sestavovat časová populace a to následujícím způsobem: pro každý chromozom prostorové populace vygeneruje odpovídající časový. Tak každý gen ČP bude odpovídat genu PP. Řešení se bude pak skládat ze dvou jedinců popisujících trasu a časové posuny na některých úsecích.

Primárně vyvíjet se bude ČP, protože účelnějším v této úloze je změna času průjezdu, která má nižší pravděpodobnost ovlivnit jiné soupravy a způsobit další konflikty. Kvalita řešení se bude hlídat a pokud rozvoj pouze ČP povede ke stagnaci řešení proběhne jedna změna generace PP. Jinými slovy PP se bude vyvíjet pouze v případu horších výsledků fitness funkce.

Dále budeme vygenerované populace selektovat stejným způsobem. Při tom by se použila turnajová selekce, jako jedna z nejefektivnějších a nejrozšířenějších. Počet turnajových kol bude ekvivalentní velikosti populace, v každém se budou porovnávat dva náhodně vybrané jedince a do další generace postoupí líp ohodnocený chromozom.

6.1 Rekombinační operátory

Kvůli různé struktuře jedinců budou se používat jinak nastavené rekombinační operátory pro každou populaci. Pro PP by se mělo použít více bodové křížení, přičemž počet bodů závisí na počtu konfliktů. Máme-li dvě kolize hledáme čtyři trasy, tím pádem potřebujeme čtyři body k tomu aby potomek měl pouze nové trasy. Tyto body se budou umísťovat náhodně, ale vždy v rámci jedné trasy a pouze v místě výhybek. Mutace v PP se používat nebude, protože nemůžeme měnit kolejové úseky trasy, protože tak bychom spojovali nenavazující na sebe úseky.

Pro ČP populaci použilo by se nejrozšířenější jednobodové křížení s vysokou pravděpodobnosti například 0.7, místo rozdělení rodičů volilo by se náhodně. Všechny jedince budou mutovat s pravděpodobnosti 0.1. Tento proces bude vypadat následujícím způsobem: náhodně se vybere gen a jeho hodnota se nahradí náhodným číslem z dříve definovaného rozsahu pro časové odstupy.

6.2 Fitness funkce

Ohodnocení jedinců je jednou z nejpodstatnějších části algoritmu, značně ovlivňující vývoj jedinců. Jak už bylo zmiňováno dřív, dle nabízených řešení, budou jedince odpovídajícím způsobem ohodnoceny. Právě v této části se definují parametry, na základě kterých, budou určité chromozomy mít větší váhu. Pro ohodnocení jedinců se navrhuje použít dvou-rozměrová fitness funkce. První etapou ohodnocení páru jedinců bude kontrola přípustnosti řešení, tj. jestli je odstraněn konflikt a nevznikl-li nový. Tak jedinci neobsahující správné řešení budou potlačovány. Na následující úrovni největší vliv na ohodnocení bude mít jedinec nabízející nebo blížící se k tomu lokální řešení konfliktu. Dále bude lépe ohodnocené řešení pro soupravu nižší kategorie, protože tak vlaku z vyšší kategorie bude nechána jeho původní trasa. Potom se budou oceňovat podle cílové stanice soupravy, pro kterou je nalezená alternativní trasa, princip je stejný, jako v předchozím případě, tj. větší ohodnocení dostane jedinec s cestou soupravy mířící k menší stanici. Výše uvedená kritéria negarantují jednoznačné upřednostňování jedince, ale lepší váhové ohodnocení, zvyšující pravděpodobnost přežiti. Popsaná kritéria neohodnocují jedince, ale pouze určují jeho váhu. Kvantitativní ohodnocení bude spočívat ve výpočtu váženého zpoždění na celé síti, čím menší je zpoždění tím lépe je jedinec ohodnocen.

Kapitola 7

Popis infrastruktury

Infrastruktura DSFD, jak už bylo zmíněno, je popsaná ve formátu XML. Je to výhodnější pro strojové zpracování, protože všechny rozšířené jazyky programování mají speciální nástroje umožňující čtení takového souboru. Striktně definovaná hierarchie dovoluje přistupovat k dokumentu jako ke stromu, jehož vrcholy jsou objekty, což je také přívětivější pro objektově orientované jazyky jako C++.

Samotný XML soubor se skládá ze dvou základních párových značek „dopravny“ umísťující v sobě seznam všech dopraven a „strom“, tato značka zahrnuje v sobě všechny elementy železniční sítí, které jsou jeho potomky. Následující úrovni uzlů jsou vazby definující id navazujícího prvku nebo více prvků v případě výhybky. Pro názornost uvádím popis jednoho potomka „stromu“.

```
<p t="KolejovyUsek" id="0" druh="2105" n="1BC2" ef="80" d="0" lg="
    1100" vk="120" vns="120" v3="120">
  <v d="VazbaLichySudy">
    <pp id="1" />
  </v>
</p>
```

Atribut *t* udává třídu prvku v daném případě je to kolejový úsek, *id* udává identifikační číslo prvku, které mimo jiné slouží k navazování vazeb mezi elementy. Dále následuje *druh* prvku, v dokumentaci k XML popisu můžeme zjistit, že se jedná o traťový úsek. Atribut *n* nese v sobě informaci o názvu prvku, *d* vyjadřuje příslušnost prvku k dopravně, kde číslo je id dopravny, *lg* vyjadřuje délku úseku v metrech. Pak následují různé rychlostní omezení:

- *vk* – maximální rychlost

- *vns* – rychlost naklápěcí skříně
- *v*3 – rychlost pro šestinápravové vozy
- *vo* – rychlost odbočkou (pouze u výhybek)

Potomkem je $< v >$ značka, která definuje vazbu prvku, atribut *d* určuje druh vazby v daném případě je to navazující prvek ve směru rovně. Její potomkem je nepárová značka $< pp >$ obsahující id elementu, na něhož je prvek navázán. Nalezneme v infrastruktuře ještě dva druhy vazeb: „VazbaOdbockaSudy“ a „VazbaOdbockaLichy“. Je zřejmé, že obě vazby patří pouze k výhybce a popisují směr odbočení.

Kapitola 8

Jízdní řády

Nezbytnou části algoritmu je vstupní grafikon, uložený jak bylo zmíněno výše v MySQL databázi. Uchovávání dat v databázi má řádu výhod oproti použití lokálních medií. Hlavním je přístupnost neboli dostupnost. Informace je uložená na serveru centrály i v případě, že například strojvedoucí vlaku zjistí neschopnost soupravy splnit jízdní řád a může pomocí příslušných nástrojů přidat tuto informaci do databáze. Systém pak detekuje mimořádnou událost a spustí v této práci navrhovaný algoritmus pro detekci respektive odstranění vzniklých konfliktů. V případě, že v grafikonu dojde k nějakým změnám, oznámí všem dotčeným soupravám změnu v jejích jízdním řádu. Tím pádem takový systém je schopný pracovat praktický autonomně s minimálním lidským zásahem. Zřejmým je, že všechny soupravy musejí být vybavené příslušnými nástroji a mít schopnost nepřerušeně komunikovat ve společné síti.

Takový způsob ukládání jízdních řád odpovídá vývojovým trendům v železniční dopravě. Postupně se v celé Evropě přechází od klasického způsobu řízení železničního provozu, kdy ovládání zabezpečovacích zařízení je distribuováno do jednotlivých stanic, k centrálnímu způsobu řízení železničního provozu. Například v ČR by většina významných železničních tratí měla být řízena ze dvou dispečerských pracovišť v Přerově a v Praze.

V současné době jízdní řády DSFD představují sebou schéma v databázi MySQL. Pro tento projekt bylo vyvinuto speciální schéma skládající ze dvou tabulek: *Vlaky* a *Casy*. Tabulka *Casy* slouží pro uložení dat o trasách jednotlivých vlaků a tabulka *Vlaky* obsahuje parametry lokomotiv potřebné k výpočtu. Na obr. 8.1 je zobrazen jízdní řád vlaku mezi dvěma stanicemi v tabulce *Casy*.

Atribut *Vlak* je identifikačním číslem soupravy a zároveň cizím klíčem tabulky *Vlaky*, *DopravniBod* je identifikačním číslem kolejového úseku přes který musí souprava ve stanovený

#	Vlak	DopravniBod	Cas	DruhZaznamu	CisloSekvence	NasledujiciSekvence	CasPorizeni	Autor
1	71	51	60	21	1	2	2017-04-10 22:11:01	NULL
2	71	88	NULL	51	2	3	2017-04-10 22:08:57	NULL
3	71	118	NULL	51	3	4	2017-04-10 22:08:57	NULL
4	71	136	211	21	4	5	2017-04-10 22:12:17	NULL
5	71	153	NULL	51	5	6	2017-04-10 22:08:57	NULL
6	71	191	420	11	6	NULL	2017-04-10 22:14:45	NULL

Obrázek 8.1: Tabulka obsahující jízdní řády

čas projet, nejsou to pouze stanice, jako v klasickém jízdním řádu, ale i zmíněné úseky umožňující jednoznačně určit trasu vlaku mezi stanicemi. Dále je atribut *Cas* definující čas průjezdu úsekem. Pak následuje atribut *DruhZaznamu*, popis jednotlivých druhů:

- 11 - příjezd - dlouhodobý plán
- 12 - příjezd - operativní plán
- 18 - příjezd skutečný
- 21 - odjezd - dlouhodobý plán
- 22 - odjezd - operativní plán
- 28 - odjezd skutečný
- 41 - minimální pobyt - dlouhodobý plán
- 42 - minimální pobyt - operativní plán
- 51 - úsek pro jednoznačné určení cesty

Potom vidíme atribut *CisloSekvence*, které je vlastně identifikačním číslem záznamu v databázi. Názvy ostatních atributů určují i jejich význam. Dále na příkladu dvou záznamu popíšeme tabulku *Vlaky*.

#	Cislo	Druh	Delka	Hmotnost	HV	PrechaziNa	Otacet
1	71	EC	200	450	380	0	0
2	8520	Os	70	150	751	0	0

Obrázek 8.2: Tabulka obsahující parametry vlaků

Atribut *Cislo* je identifikačním číslem soupravy, jak je zřejmé z názvu, *Druh* je kategorii vlaku v našem případě EuroCity a osobní vlak. Další dva atributy určují délku respektive hmotnost soupravy. Atribut *HV* popisuje řádu hnacího vozidla, *PrechaziNa* definuje případné navazující spoje a atribut *Otacet*, říká jestli se vlak otáčí v jedné ze stanic.

Kapitola 9

Implementace

Ústav dopravní telematiky založil dopravní sál, představující laboratoř pro návrh a vývoj železničního zabezpečovacího zřízení. Hlavním dílem sálu je dostatečně přesný model kolejiště, umožňující simulovat reálnou infrastrukturu. Software pro řízení prostředků pohybujících se po kolejišti včetně grafického rozhraní byl implementován Ing. Petrem Kouteckým. Program je napsán v objektově orientovaném jazyce C# s využitím frameworku .NET, obsahuje veškeré objektové modely kolejových prvků včetně zabezpečovací techniky, rozložení trati se načítá z externího souboru. Popis samotné infrastruktury, tím myšleno všech jejích součástí jako kolejový úsek, traťová zabezpečovací zařízeni atd., je umístěn v souboru XML. Nalezneme tam také některé charakteristiky těchto součástí jako například rychlost průjezdu, sklon nebo délku u kolejových úseků. Pro účely simulace popsané dále v této práci bude využito tohoto souboru s popisem infrastruktury.

9.1 Volba programovacího jazyka a vývojového prostředí

Z hlediska efektivity práce v rámci tohoto projektu a časových nároků na jeho implementaci by bylo vhodné preferovat jazyk C#, ve kterém jsou již popsané objekty vhodné pro naše účely Ing. Kouteckým. Ale při optimalizaci jízdních řádů evolučními technikami je potřeba vygenerovat velké množství jedinců (jízdních řádů), je to dáno základními principy fungování heuristických metod. Pro samotné generování jedinců by byl jazyk C# dostačující, ale následné výpočty (ohodnocení) klade velké výpočetní nároky. Z toho důvodu musí být zvolen jazyk, kde převládá vysoká efektivita výsledného programu. Proto byl zvolen jazyk C++, umožňující na nižší úrovni pracovat s paměti a optimalizovat její využití pro vlastní potřeby a vyznačující se výsledným efektivním kódem. Diagram implementovaných tříd je v příloze A.

Před zahájením implementace projektu byla potřeba také rozhodnout, jaké vývojové prostředí se použije. Obecně tento nástroj slouží ke zjednodušení vývoje programu, proto jedním z kritérií byly podpůrné nástroje pro tvorbu a editaci kódu a ladění. Veškerá vývojová prostředí dnes obsahují velké množství klávesových zkratek zrychlujících práci. Mezi další přednosti patří například automatické doplňování metod patřících do uživatelem vytvořených tříd nebo standardních knihoven, což také zvyšuje pohodlí při vývoji. Názorné zobrazení části algoritmu obsahující chyby vzniklé během kompilace zdrojového kódu umožňuje rychlé odstraninění syntaktické chyby a tím se také poněkud zkracuje doba potřebná pro odladění programu. Největší výhodou oceňovanou nejenom začátečníky je vestavěný ladicí nástroj neboli debugger. Vývojová prostředí má i nedostatky, prvním je nepřenositelnost projektu. Samotné vývojové prostředí může existovat ve verzích pro více operačních systémů. Prostředí je většinou shodné, ale tzv. pozadí (back-end), např. vlastní překladač, jsou ve své podstatě odlišné programy a tím pádem nelze přenášet výsledný projekt, a také současně vyvíjet v různých operačních systémech. Zmíním také výpočetní nároky prostředí, jsou celkem nízké, ale mohou se projevit jako nedostatečné u starších počítačů. Mapování externích knihovenje na první pohled celkem jednoduché, protože tuto operaci provedeme několika klinutími, ale jak se ukázalo v praxi, občas je potřeba nastavení změnit nebo dodatečně nastavit, čímž se proces značně komplikuje. Současně s výběrem prostředí bylo rozhodováno i o operačním systému, ve kterém se bude vyvíjet. Původně bylo vybráno prostředí Code::Blocks, protože je volně přístupné, má nízké systémové požadavky a disponuje všemi výše uvedenými přednosti vývojových prostředí.

Dále bylo rozhodnuto použít operační systém MS Windows 10, protože je spolehlivější a uživatelsky přívětivější než jakákoli distribuce Linuxu a používá jej autor pro všechny účely s výjimkou pracovních. Jedním z prvních kroků implementace projektu bylo zprovoznění nezbytně nutných externích knihoven pro načítání souboru XML obsahujícího popis infrastruktury a knihovny pro spojení s databází SQL nesoucí v sobě jízdní řády. Byly vybrány příslušné knihovny Xerces a Mysqlcppconn, důvod takového rozhodnutí bude popsán níže. Prvním pokusem bylo načítání souboru XML s popisem infrastruktury a tedy využití knihovny Xerces. Přestože knihovna byla nainstalována a připojená správně, objevila se chyba linkování, při pokusu jí odstranit bylo zjištěno, že pro tento systém (myslíš tím Windows?) je také vyžadováno použití další dynamické knihovny, tímto byla chyba odstraněna. Dále jsem se snažil zprovoznit knihovnu Mysqlcppconn, po přečtení mnoha návodů na instalaci včetně informací na oficiálních webovým stránkách se nepovedlo dopracovat ke kladnému výsledku. Kvůli špatným výsledkům bylo třeba vyzkoušet jiný systém, na řadě byl Linux a jeho distribuce Ubuntu 16.04, který se dříve

ukázal jako stabilní a celkem intuitivní. Na rozdíl od systému MS Windows byly obě knihovny zprovozněny do hodiny. Na základě této zkušenosti jsem si potvrdil předpoklad říkající, že výběr operačního systému od firmy Microsoft je kompromisním řešením, kde sice dostáváte přívětivé uživatelské rozhraní, intuitivní ovládání, správu systému jednoduchou pro začátečníky , ale na úkor vývojových nástrojů.

9.2 Objekty popisující infrastrukturu

Jak jíž bylo zmíněno v kapitole o popisu infrastruktury, železniční síť se skládá z určitých prvků. Pro implementaci daného projektu využijeme pouze některé z nich (kolejový úsek, výhybka, izolovaný styk a návěstidlo). S ohledem na to, že každý prvek je samostatnou entitou, byly pro ně vytvořeny odpovídající třídy. Všechny elementy mají společné vlastnosti, které v sobě nese třída *PrvekStromu* jako předek, ostatní prvky jsou potomci.

Třída *PrvekStromu* obsahuje základní informace o elementu. Je to jeho identifikační číslo, příslušnost k jedné ze tříd infrastruktury v podobě proměnné výčtového typu (může nabývat hodnot pouze výše definovaných prvků a hodnoty *JINY* pro zachycení výjimek). Z důvodu, který bude napsán v kapitole popisující načítání a zpracování jízdních řádů, byla přidán k této třídě informace o tom, zda na daném elementu vlak odbočuje. Kromě toho *PrvekStromu* obsahuje ukazatele na navazující prvek v síti a element, který ukazuje na něj, tj. je předchozím prvkem. Dále následuje seznam metod třídy:

```
PrvekStromu();
PrvekStromu(unsigned int id);
~PrvekStromu();
int getId();
PrvekStromu * get_lichy_smer();
PrvekStromu * get_sudy_smer();
void set_lichy_smer(PrvekStromu * lichy_smer);
void set_sudy_smer(PrvekStromu * sudy_smer);
Tridy_Prvku get_trida_prvku();
void set_odboceni(bool odboceni);
bool get_odboceni();
```

První dvě metody jsou konstruktory třídy, které nastavují výchozí hodnoty instance třídy. Obě metody definují předchozí a následující prvek jako NULL, a potom v závislosti na tom, jaká z metod byla zavolána, buď se nastaví id prvku na hodnotu -1 (implicitní konstruktor), nebo na hodnotu argumentu id. V rámci daného projektu je označován následující prvek jako lichý a

předchozí jako sudý. Zbytek metod, jak je patrné z názvu, jsou tak zvané get a set metody. Jsou to jednoduché funkce, respektive procedury, které buď nastavují datovou složku instance nebo naopak vracejí její hodnotu. Například metoda

```
void set_lichy_smer(PrvekStromu * lichy_smer);
```

Nastaví hodnotu ukazatele lichého směru na element, který je argumentem funkce.

```
PrvekStromu * get_lichy_smer();
```

Metoda vrací ukazatel na prvek nacházející v lichém směru.

Všechny atributy třídy jsou definovány jako protected, což znamená, že nelze jejích hodnotu měnit mimo třídu. Tento styl programování se doporučuje z bezpečnostních důvodů, aby nedocházelo k nežádoucímu přepisu hodnot atributů. Proto je zapotřebí používat get a set metody, které jsou schopné pracovat s obsahem atributů.

Třída Navestidlo je potomkem třídy PrvekStromu, obsahuje navíc informace o tom, k jaké dopravně návěstidlo přísluší v podobě jeho id, a o jeho druhu. Má následující metody:

```
inline Navestidlo(unsigned int id): PrvekStromu(id){};
Navestidlo(unsigned int id, int dopravna, eDruhNavestidla druhNavestidla);
int get_dopravna();
eDruhNavestidla get_druh_navestidla();
void get_dopravna(int dopravna);
void get_druh_navestidla(eDruhNavestidla druhNavestidla);
```

Tato třída také nemá v sobě žádnou logiku, pouze *get* a *set* metody pro svoje atributy. Třída *IzolovanyStyk* je potomkem třídy *PrvekStromu* a nemá žádnou přidanou hodnotu oproti předchůdci. Byla vytvořena z důvodu zachování struktury a případného rozšíření v budoucnu. Třída *CastKO* neboli část kolejového obvodu je také potomkem třídy *PrvekStromu* a je společným předkem pro třídy KolejovyUsek a Vyhybka, obsahuje jejich společné vlastnosti. Jsou to: sklon úseku respektive výhybky, délka, tři druhy rychlosti (vk – rychlostník n, vns – rychlostník ns pro soupravy s naklápěcími skříněmi, $v3$ – rychlostník 3 pro vozidla skupiny přechodnosti 3) a identifikační číslo dopravny, ke které přísluší. Její metody jsou:

```
inline CastKO(): PrvekStromu(){};
inline CastKO(unsigned int id): PrvekStromu(id){};
CastKO(unsigned int id, unsigned int delka, unsigned int druh, unsigned int
    dopravna, unsigned int rychlostKlasiky, unsigned int rychlostTrida3, unsigned
    int rychlostNS);
int get_delka();
int get_dopravna();
```

```
int get_rychlostKlasiky();
int get_rychlostNS();
int get_rychlostTrida3();
void nastav_delku(unsigned int delka);
```

Stejně jako u jiných prvků infrastruktury nemá v sobě tato třída žádnou logiku, pouze konstruktory a *get/set* metody.

Třída *KolejovyUsek* je potomkem třídy *CastKO* a neliší se od něj na, protože zatím má pouze společné s výhybkou vlastnosti. Byla vytvořena pro zachování logické struktury a pro případné rozšíření v budoucnu. Má pouze konstruktory zděděné od *CastKO*.

Třída *Vyhybka* je potomkem třídy *CastKO*. Na rozdíl od kolejového úseku obsahuje jeden směr pohybu navíc, tj. musí ukazovat nejenom na předchozí a následující prvek, ale na prvek, na který vlak může odbočit. Pro jednoznačnost rozlišení směru odbočeni příslušné výhybky, třída obsahuje ukazatele na odbočky jak v lichém tak i sudém směru a jeden z nich má vždy hodnotu *NULL*. Například, pokud výhybka má odbočku v sudém směru, bude její atribut $odbocka_sudy$ ukazovat na *PrvekStromu*, a ukazatel $odboka_lichy$ bude mít hodnotu *NULL*. Kromě toho každá instance této třídy nese v sobě informace o maximální rychlosti odbočení. Seznam její metod:

```
Vyhybka(unsigned int id);
Vyhybka(unsigned int id, unsigned int delka, unsigned int druh, unsigned int
    dopravna, unsigned int rychlostKlasiky,
        unsigned int rychlostTrida3, unsigned int rychlostNS, unsigned int
            rychlostOdbocka);
~Vyhybka();
void set_odbocka_lichy(PrvekStromu *p);
void set_odbocka_sudy(PrvekStromu *p);
PrvekStromu * get_odbocka_lichy();
PrvekStromu * get_odbocka_sudy();
int get_rychlost_odbocky();
```

Oproti třídě *CastKO* je seznam rozšířen o *get/set* metody atributů, které jsou definovány pouze v této třídě.

9.3 Analýza (parsing) souboru XML s popisem infrastruktury

Pro přečtení železniční infrastruktury je potřeba analyzovat soubor XML obsahující její popis, podrobněji popsaný v kapitole 7. Programovací jazyk C++ neposkytuje nástroje umožňující práci se soubory XML, proto bylo potřeba buď vytvořit vlastní, nebo použít již existující volně

dostupné nástroje. Vzhledem ke složitosti takové úlohy není první varianta vhodná z hlediska efektivity, protože již existují hotová řešení splňující požadavky na analýzu souboru ve formátu XLM. Dalším strategickým řešením byl tedy výběr knihovny pro analýzu (parsing) XML. Existuje celá řada open source knihoven s různým zaměřením, nejrozšířenější jsou: LibXML2, Xerces a RapidXML. LibXML2 je napsaná v jazyce C, je vysoce přenositelná a výkonná, ale nezachovává DOM (Document Object Model) model dokumentu a není objektově orientovaná, protože jazyk C nepodporuje takové paradigma programování. RapidXML patří k nejrychlejším „parserům“, má vysokou úroveň přenositelnosti, je napsaná v jazyce C++, ale také nepodporuje DOM model. Knihovna Xerces na rozdíl od výše uvedených umožnuje zachovat strukturu dokumentu (DOM) a je napsána v jazyce C++. Byla zvolená právě knihovna Xerces ze dvou důvodů: první je objektově orientovaný přístup, tím pádem se nemusejí míchat dva styly programování v projektu. Druhým důvodem je zachování struktury DOM, což umožní ponechat definovanou hierarchii prvků a vazeb tak, jak je uložena v původním dokumentu.

9.4 Využití knihovny Xerces při načítaní infrastruktury

Pro načítání dat ze souboru XML do vnitřní datové struktury byla použita knihovna Xerces z důvodů uvedených výše. Dále bylo potřebné vytvořit třídu zastřešující tento proces třídu XMLInfrastructReader. Její účelem je načítání kolejových úseků, výhybek, izolovaných styků a návěstidel do příslušných vektorů (myšleno kontejner z knihovny <vector>) a následné navazování vazeb mezi nimi. Kromě zmíněných elementů má atributy jako: počet dopraven, ukazatel na pole dopraven, vektor vazeb (struktury definované v této třídě) a ukazatele na objekty knihovny Xerces potřebné pro zpracování souboru XML.

V hlavičkovém souboru *InfrastructReader.h* jsou definovány dvě struktury *Dopravna* a *Vazba_Info*. První popisuje dopravnu, má její identifikační číslo, název a zkratku přebranou ze zdrojového souboru. Všechny tyto vlastnosti jsou uloženy do datového typu string. *Vazba_Info* je pomocná struktura vazeb prvku. Potřeba vytvořit tuto strukturu vznikla po rozhodnutí o způsobu načítání vazeb. Problém jejích přečtení ze souboru spočívá v tom, že při postupném čtení prvků infrastruktury ze souboru, nejsou elementy seřazené dle své návaznosti. To znamená, že například prvek s id 13 má navazující element s id 11, ale v souboru příští element má id 14 (pro názornost je na obrázku n je část XML souboru). Což by při současném načteni prvků a vazeb znamenalo, že je potřeba přeskočit zpátky (v tomto konkrétním případě) na prvek 11. Obecně poloha prvku souboru není známa, tj. při každém navazování vazeb by se musel hledat

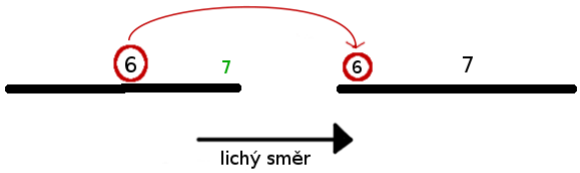
prvek v souboru, což by znamenalo zbytečný růst složitosti algoritmu (nelze určit přesný rozdíl složitosti, protože záleží na tom, jaký by byl vybrán algoritmus hledání).

```
<p t="KolejovyUsek" id="11" druh="2101" n="V2-3" ef="80" d="0" lg="50" vk="110"
    vns="110" v3="110" />
<p t="Vyhybka" id="12" druh="4001" n="1" ef="180" d="0" lg="50" vk="110" vns="110
    " v3="110" vo="40">
  <v d="VazbaOdbockaSudy">
    <pp id="13" />
  </v>
  <v d="VazbaLichySudy">
    <pp id="14" />
  </v>
</p>
<p t="IzolovanyStyk" id="13" druh="2201" n="" d="0">
  <v d="VazbaLichySudy">
    <pp id="11" />
  </v>
</p>
<p t="KolejovyUsek" id="14" druh="2101" n="V1" ef="80" d="0" lg="50" vk="110" vns
    ="110" v3="110">
  <v d="VazbaLichySudy">
    <pp id="15" />
  </v>
</p>
```

Proto bylo rozhodnuto nejprve načíst všechny potřebné prvky do příslušných vektorů a současně do samostatného vektoru ukládat informace o vazbách prvků. Je to vektor, který se skládá z elementů typu $Vazba\_Info$. Jak bylo zmíněno výše, je to struktura skládající z id prvku, ke kterému vazba přísluší, jeho třídy, id elementu s nímž nějakou vazbu tvoří a typ vazby. Poslední je proměnná výčtového typu definovaná pouze pro danou třídu, může nabývat hodnot: $VazbaNeznama$, $VazbaLichySudy$, $VazbaOdbockaLichy$ a $VazbaOdbockaSudy$. Samotné navázání vazeb probíhá až v následujícím kroku pomocí metody $definuj\_vazby()$. Princip je následující: v cyklu jsou procházeny postupně prvky vektoru obsahujícího vazby a v každé iteraci je nejdřív rozpoznán druh vazby. Pak v závislosti na vazbě se vyhledává zdrojový prvek vazby dvěma způsoby. První se použije v případě vztahu „lichý-sudý“ (tj. předchůdce a následník v přímém směru), na základě třídy zdrojového prvku, která je součásti struktury $Vazba\_Info$ hledá se tento prvek v pouze příslušném vektoru. Například, pokud element prvku patří ke třídě izolovaných styků, hledá se tento prvek pouze v předem načteném vektorů izolovaných styků.

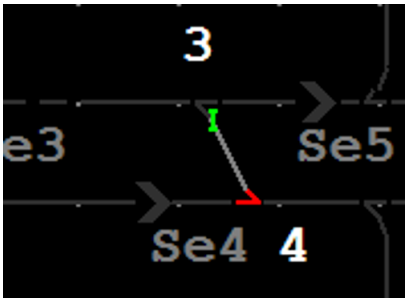
Druhý způsob se použije v ostatních vztazích – odbočka sudá nebo lichá. Vzhledem k tomu, že zdrojem této vazby může být pouze výhybka, vyhledávání probíhá pouze v příslušném vektoru. Potom, když už byl zdrojový prvek vazby nalezen, hledá se cílový. Protože při načítání vazby nelze určit třídu navazujícího prvku, musí se jeho instance hledat ve všech vektorech. Pro optimalizaci, hledání probíhá postupně a začíná na vektoru kolejových úseku, protože těchto elementů je v síti nejvíc, tím pádem pravděpodobnost toho, že náhodný prvek patří k této třídě, je největší. Dále jsou prohledávány kolekce návěstidel, výhybek a izolovaných styků. Pořadí bylo vybráno na základě frekvence výskytu třídy v infrastruktuře. Z popsaného postupu vzniká logická otázka, pokud se prvky musejí každopádně hledat, proč je nehledat rovnou při načítání souboru s infrastrukturou? Jednak se musí při hledání náhodného prvku procházet celý soubor od začátku, když v případě navrženého způsobu ve většině případu je zkoumán pouze jeden kontejner, obsahující mnohem méně dat a je přizpůsobenější pro hledání. Také, při hledání v souboru probíhá přenos informace mezi diskem (kde se příslušný soubor nachází) a pamětí počítače, na rozdíl od hledání v kontejnerech, kde přístup k datům probíhá pouze v rámci paměti. Tyto dvě úvahy vedou k tomu, že zvolený způsob bude teoreticky rychlejší.

Následující kroky opět závisejí na druhu vazby. Jednodušší je druh vazby „lichý-sudý“, kde se ukazateli zdrojového prvku na následující element přiřadí cílový, a v případě existence cílového prvku stane se jeho předchozím prvkem zdrojový. Následující obrázek schematicky zobrazuje tento proces.



Obrázek 9.1: Princip přiřazování ukazatelů

Postup je odlišný kvůli specifickým vlastnostem infrastruktury jeho XML popisu. Pro názornost je použitá část infrastruktury zobrazená v programu JOPEdit vyvinutá Ing. Petrem Kouteckým a odpovídající jemu část XML dokumentu.



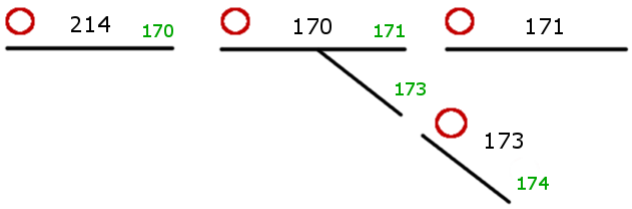
Obrázek 9.2: Část infrastruktury v programu JOPEdit

Podsvícenou šedou barvou je zobrazen kolejový úsek s id 29, červenou barvou zobrazená výhybka s id 28, která na něho navazuje. Dále následuje XML popisující vztah těchto prvků.

```
<p t="Vyhybka" id="28" druh="4002" n="4" ef="80" d="0" lg="50" vk="110" vns="110"
    v3="110" vo="60">
  <v d="VazbaOdbockaLichy">
    <pp id="29" />
  </v>
  <v d="VazbaLichySudy">
    <pp id="33" />
  </v>
</p>
<p t="KolejovyUsek" id="29" druh="2101" n="V4-6" ef="80" d="0" lg="50" vk="110"
   vns="110" v3="110" />
<p t="Vyhybka" id="30" druh="4001" n="3" ef="80" d="0" lg="50" vk="110" vns="110"
    v3="110" vo="60">
  <v d="VazbaOdbockaSudy">
    <pp id="31" />
  </v>
  <v d="VazbaLichySudy">
    <pp id="32" />
  </v>
</p>
```

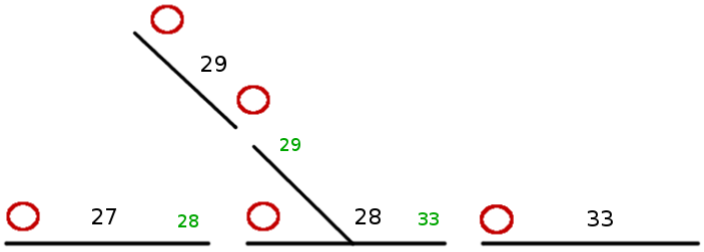
Prvek má v souboru explicitně referenci pouze na následný prvek v jednom směru, avšak pro potřeby prohledávání grafu je nutné doplnit i referenci na „předchozí prvek“ – v opačném směru. Pro srozumitelnost je níže uvedeno běžné schéma přiřazení ukazatelů

Černou barvou zobrazena vlastní čísla prvků, zelenou čísla navazujících elementů explicitně popsaných v souboru a červené kroužky ukazují, kde je potřeba vazby doplnit. Například u výhybky číslo 170 je potřeba doplnit její předchůdce, tj. kolejový úsek č. 214. Dále si schematicky



Obrázek 9.3: Schéma přiřazení ukazatelů

ukážeme výše popsanou situaci s chybějícím ukazatelem.



Obrázek 9.4: Chybějící ukazatele u kolejového úseku č. 29

Na tomto obrázku vidíme, že kolejový úsek č. 29 nemá žádnou explicitně zapsanou vazbu, ale skutečně má navazující výhybku. Tento vztah byla potřeba doplnit na základě informace o navazujících prvků výhybky č. 28. Tímto krokem, ale je vytvořena obousměrná vazba, která je ve skutečnosti jednosměrná (směr odbočení výhybky ukazuje na kolejový úsek, který pak zpět ukazuje na výhybku).

Navazování ukazatelů je pouze části třídy *InfrastructReader*, dále se budu zabývat její podrobnějším popisem. Na začátek bych uvedl seznam metod třídy:

```
protected:
char *precti_atribut(const char *jmeno, DOMElement *el);
int precti_ciselny_atribut(const char *jmeno, DOMElement *el);
bool nacti_dopravny();
bool nacti_strom();
```

```
Tridy_Prvku zjisti_tridu(DOMElement *prvek);
void zpracuj_prvek_stromu(DOMElement *prvek);
bool zpracuj_kolejovy_usek(DOMElement *usek);
bool zpracuj_navestidlo(DOMElement *navest);
bool zpracuj_vyhybku(DOMElement *vyhybka);
bool zpracuj_izolovany_styk(DOMElement *iz_styk);
void definuj_vazby();
void nacti_vazby(DOMElement *prvek,Vazba_Info vaz[]);
void nacti_pp(DOMElement *vazba, Vazba_Info *v);

public:
PrvekStromu * najdi_zdrojovy_prvek(int id, Tridy_Prvku tr);
PrvekStromu * najdi_prvek(int id, Tridy_Prvku *tp);
CastKO * najdi_kolejovy_usek(int id);
Vyhybka * najdi_vyhybku(int id);

string jmeno_souboru;
void nahraj() ;
void analyza();

vector<KolejovyUsek*> get_vector_kolejovych_useku();
vector<Navestidlo*> get_vector_navestidel();
vector<Vyhybka*> get_vector_vyhybek();
vector<IzolovanyStyk*> get_vector_iz_styku();

void vypis_vector();
unsigned int vrat_pocet_dopraven();
string vrat_id_dopravny(int index);
string vrat_nazev_dopravny(int index);
string vrat_zkratku_dopravny(int index);

XMLInfrastructReader();
~XMLInfrastructReader();
```

Dále budou popsané jednotlivé metody.

```
XMLInfrastructReader()
```

Je konstruktorem třídy, nastavuje výchozí nulovou hodnotu atributů třídy, které jsou typu ukazatel, a vytváří instanci třídy *XercesDOMParser*, potřebnou pro zpracování souboru.

```
void nahraj()
```

Načte soubor XML s názvem uloženým do atributu *jmeno_souboru*. Provede syntaktickou analýzu voláním metody *parse* ze třídy *XercesDOMParser* a vyzvedne kořenový element dokumentu, který se přiřadí ukazateli na objekt *DOMElement*.

```
char * precti_atribut(const char *jmeno, DOMElement *element)
```

Funkce přečte atribut tagu (značky), na který ukazuje *element*, název požadovaného atributu je dalším argumentem funkce - jmeno. Po přečtení vrací metoda ukazatel na řetězec s hodnotou atributu. Například u elementu *<p id="3">* chceme načíst jeho id, do argumentu funkce jmeno vložíme atribut „id“ a ukazatel na tento element *< p >*. Metoda vrátí ukazatel na řetězec obsahující „3“. Definice této metody je výhodná, neboť hodnoty atributů se čtou opakovaně. Metoda vracející řetězec se používá např. při čtení druhu návěstidel, neboť hodnotou je právě znakový řetězec. Naopak se nehodí v uvedeném příkladu nebo při čtení rychlosti úseku, kde hodnotu atributu je číslo. Proto pro zkrácení zdrojového kódu byla napsaná následující metoda:

```
int precti_ciselny_atribut(const char *jmeno, DOMElement *el)
```

Má stejné argumenty jako předchozí metoda, ale místo ukazatele na řetězec vrací číslo, jinými slovy převede přečtenou hodnotu atributu ve formě řetězce na číslo. Volá se pouze pro načítání číselných atributů. Například, opět chceme vědět id elementu *<p id="123">*, vrátí funkce hodnotu *123*.

```
bool nacti_dopravny()
```

Metoda pro načítání dopraven, v případě nalezení alespoň jedné dopravny vytvoří dynamické pole dopraven (velikostí odpovídající počtu dopraven) a vrátí hodnotu *true*. Každý prvek pole bude obsahovat id dopravny, její název a zkratku. Pokud nejsou v souboru dopravny, vrátí hodnotu *false*.

```
bool nacti_strom()
```

Postupně načítá celý strom infrastruktury a následně definuje vazby mezi nimi prostřednictvím volání příslušných metod. V případě úspěšného načtení a provázání vrátí hodnotu *true* v opačném *false*.

```
void analyza()
```

Metoda řídí načítání dopraven a pak i celého stromu. Je veřejnou metodou a stejně jako *nahraj*() se musí volat v hlavním programu pro korektní práci algoritmu a to v pořadí po metodě *nahraj*(), kterou předchází pouze konstruktor.

```
Tridy_Prvku zjisti_tridu(DOMElement *prvek)
```

Na základě jednoho z atributů metoda zjistí, ke které třídě patří element, jehož ukazatel je argumentem. Zjištěná třída je následně vrácena.

```
void zpracuj_prvek_stromu(DOMElement *prvek)
```

Řídí zpracování jednotlivých prvků stromu a to tak, že zavoláním výše popsané metody zjistí třídu prvku a pak na tomto základě zavolá příslušnou metodu pro jeho zpracování.

```
bool zpracuj_kolejovy_usek(DOMElement *usek)
```

Metoda se volá pro načtení a zpracování kolejového úseku. Zpracováním je v daném případě myšleno vytvoření nového objektu odpovídající třídy, načítání potřebných atributu a jeho vazby, přidání nově vytvořené instance ke kolekci, která je atributem třídy $XMLInfrastructReader$. Jak bylo popsáno výše, vazby prvku jsou navazovány v dalším kroku tak, že informace o vztahu elementu je prozatím ukládána zvlášť. Po úspěšném načteníprvku je vrácena hodnota *true*, v opačném případě $false$.

```
bool zpracuj_navestidlo(DOMElement *navest)
bool zpracuj_izolovany_styk(DOMElement *iz_styk)
```

Princip těchto metod je stejný jako funkce zpracovávající kolejový úsek, liší se atributy načítané funkcí a používanou kolekcí nových instancí (odpovídají názvu metod).

```
bool zpracuj_vyhybku(DOMElement *vyhybka)
```

Od předchozích metod se liší pouze počtem načítaných vazeb, který je u výhybky samozřejmě větší.

```
void nacti_vazby(DOMElement *prvek, Vazba_Info vaz[])
```

Metoda načítá vazby prvku a to tak, že na základě ukazatele přečte jeho následníka z hlediska hierarchie (element $< v >$ a jeho jediný atribut, který je druhem vazby). Jako vedlejší efekt funkce uloží do pole $vaz[]$ je informaci o vazbě, ale v následujícím kroku, kde se volá metoda načítající potomka tohoto následníka (pro připomenutí obsahuje id navazujícího prvku).

```
void nacti_pp(DOMElement *vazba, Vazba_Info *v)
```

Právě tato metoda načte veškerou informaci o vazbě prvku a uloží ji do ukazatele na strukturu $Vazba\_Info$. Procedura má za úkol načíst druh vazby, id prvku, ke kterému vazba přísluší a id elementu na něj navazujícího. Je pomocnou procedurou pro metodu $nacti\_vazby$, tam se volá buď dvakrát nebo jedenou v závislosti jestli se jedna o výhybku nebo ostatní prvky.

```
void definuj_vazby()
```

Metoda pro navazování vazeb, byla podrobně popsána v kapitole o načítání infrastruktury.

```
PrvekStromu * najdi_zdrojovy_prvek(int id, Tridy_Prvku tp)
```

Jedna z pomocných metod, hledá prvek infrastruktury na základě jeho třídy a identifikačního čísla. Podle třídy pozná, v jaké kolekci se má element hledat, id jednoznačně stanoví jaký element je potřeba najít. V případě, že prvek byl nalezen, vrátí na něho ukazatel, jinak vrátí *NULL*.

```
PrvekStromu * najdi_prvek(int id, Tridy_Prvku *tp)
```

Na rozdíl od předchozí metody prochází postupně všechny kontejnery obsahující prvky infrastruktury. Pořadí je určeno pravděpodobností výskytu určité třídy na infrastruktuře, která je přímo úměrná počtu prvku na infrastruktuře spadající k této třídě. Kromě samotného hledání elementu se do ukazatele *tp* uloží třída, ke které prvek patří.

```
CastKO * najdi_kolejovy_usek(int id)
```

Na rozdíl od ostatních vyhledávacích metod vrací instanci typu *CastKO*. Danou metodu byla potřeba zavést, protože objekt *PrvekStromu* vracený výše uvedenými metody, nevlastni potřebnými atributy pro práci jako s kolejovým úsekem. Jedna se o jeho délku, sklon a rychlosti pro jednotlivé druhy vlaků. Princip vyhledávání je jednoduchý, postupně se od začátku prochází kolekce kolejových úseku až do hledaného úseku. Pokud hledaný úsek v kolekci není je návratová hodnota *NULL*.

```
Vyhybka * najdi_vyhybku(int id)
```

Metoda je vytvořena ze stejných důvodů jako předchozí, akorát její cílem je vyhledáváni výhybky.

```
vector<KolejovyUsek*> get_vector_kolejovych_useku()
vector<Navestidlo*> get_vector_navestidel()
vector<Vyhybka*> get_vector_vyhybek()
vector<IzolovanyStyk*> get_vector_iz_styku()
```

Sada metod sloužících pouze pro návrat konkrétní kolekce objektu *XMLInfrastructReader*.

```
~XMLInfrastructReader()
```

Destruktor třídy, je v něm uvolněna paměť všech dynamicky alokovaných proměnných respektive polí instanci třídy.

Tak je patrně třída *XMLInfrastructReader* se zabývá výhradně načítáním infrastruktury. Dále byla potřeba vytvořit separátní třidu, která by umožnila dle potřeby manipulovat s infrastrukturou. Tento požadavek vzniká kvůli potřebě jak doplňování chybějících v jízdním řádu kolejových úseků (odkaz na popis jízdních řády) tak i při hledání nejoptimálnější cesty. Pojem optimální cesty je zavádějící v rámci železniční sítě, protože představujeme si pod tím nejkratší

trasu, ale nemusí tak tomu ve skutečnosti být. Díky vyšším rychlostem mohou být delší úseky vhodnější, protože mají menší jízdní dobu, tj. optimální cesta je definována jako trasa s nejkratším časem potřebným na její překonání.

Procházení sítě, hledání nejkratší cesty a další funkcionality nabízí třída *Infrastruktura*. Vzhledem k tomu, že cílem této třídy je zastřešení infrastruktury, jejími atributy jsou kolekce načtených a následovně provázaných prvků ze vstupního souboru.

Při zpracování vstupního souboru třídou *XMLInfrastructReader* pro každý přečtený prvek vytváří se pro něj vlastní instance, ale do kolekce třídy prvku (kolejový úsek, návěstidlo, výhybka nebo izolovaný styk) ukládá se pouze ukazatel na tento objekt. Takový způsob zrychluje případnou manipulaci s kolekcí, její přenos respektive kopírování uskutečňuje se za kratší dobu díky menšímu obsaženému objemu dat. Tak při předávání kolekcí pomocí *get* metod kopírují se pouze adresy instancí. Následkem je to, že v případě delegování vektorů třídou *XMLInfrastructReader* třídě *Infrastruktura*, nesmí se dříve zavolat destruktor předávající třídy, protože při uvolnění paměti byly by skutečné instance smazány a zkopírované do přijímající třídy ukazatele obsahovaly by neinicializované adresy.

Pro podrobnější popis třídy *Infrastruktura* uvádím seznam její metod, dále bych stručně popsal účel a princip jednotlivých funkcí a procedur:

```
Infrastruktura( XMLInfrastructReader * reader );
bool najdi_cestu( int zdroj, int cil );
vector<int> najdi_cestu_vh( int zdroj, int cil, bool *cesta );
vector<int> najdi_cestu_vh_alt( int zdroj, int cil, bool *cesta );
void vypis_trasu( int zdroj, int cil );
void vypis_vsechny_trasy( int zdroj, int cil );
vector<vector<PrvekStromu*>> vrat_vsechny_trasy_stack( int zdroj, int cil );
vector<PrvekStromu*> vrat_nejkratsi_trasu( int zdroj, int cil );
vector<KolejovyUsek*> vrat_ku_misto_ps( vector<PrvekStromu*> vect_ps );
vector<Vyhybka*> vrat_vh_misto_ps( vector<PrvekStromu*> vect_ps );
~Infrastruktura();
```

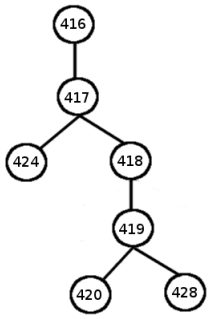
```
Infrastruktura( XMLInfrastructReader * reader )
```

Konstruktor třídy, inicializuje ukazatel na objekt *XMLInfrastructReader*, který je atributem třídy, dále na základě tohoto ukazatele jsou inicializované kolekce třídy *Infrastruktura*.

```
bool najdi_cestu(int zdroj, int cil)
```

Argumenty funkce jsou identifikační čísla prvků infrastruktury. Algoritmus prochází graf do šířky v jednom a to lichém směru od zdroje k cíli, pokud během průchodu narazí na cílový

prvek nebo zdrojový a cílový prvek se shodují vrátí hodnotu *true* v opačném případě *false*. Na obr. 9.5 je schematicky zobrazena část infrastruktury, pro přehlednost na obrázku jsou pouze kolejové úseky a výhybky. V tomto případě pořadí průchodu je následující: 416, 417, 424, 418, 419, 420, 428.



Obrázek 9.5: Schéma části infrastruktury

Algoritmus byl implementován pomoci fronty. Princip je následující do fronty se nejdřív vloží potomek zdrojového prvku, pak vezme se první element fronty (v případě první iterace fronta se skládá pouze z jednoho prvku) a zkontroluje se, není-li cílem. Pokud cílový prvek nebyl nalezen, do fronty se postupně vkládají následníky posledního elementu. Cyklus skonči v moment, kdy fronta bude prázdná.

```
vector<int> najdi_cestu_vh_alt(int zdroj, int cil, bool *cesta
```

Metoda nalezne všechny výhybky mezi cílovým a zdrojovým prvkem a vrátí vektor obsahující jejích id. Jako vedlejší efekt uloží do proměnné typu boolean zda existuje cesta mezi těmito elementy. Může vzniknout otázka jestli by nestačilo vracet prázdný vektor v případě neexistující trasy mezi prvky, v tomto případě nelze zachytit situaci, kdy cílový element bod je dosažitelný, ale nejsou v této cestě výhybky.

Pomoci předchozí metody se nejdřív zkontroluje, jestli existuje cesta mezi zdrojovým a cílovým prvkem, pokud ano, prochází se do šířky část grafu omezená těmito prvky. Všechny výhybky při tom nalezené respektive jejích id jsou uložené do vektoru, který je následně vracen. Tato metoda bude dále využitá pro nalezení všech cest mezi dvěma body grafu.

```
vector<vector<PrvekStromu*>> vrat_vsechny_trasy_stack(int zdroj, int cil)
```

Funkce hledá všechny trasy mezi zadanými prvky infrastruktury. Vstupními argumenty jsou identifikační čísla elementů, mezi nimiž se cesty hledají, výsledkem je dvojrozměrný vektor, ob-

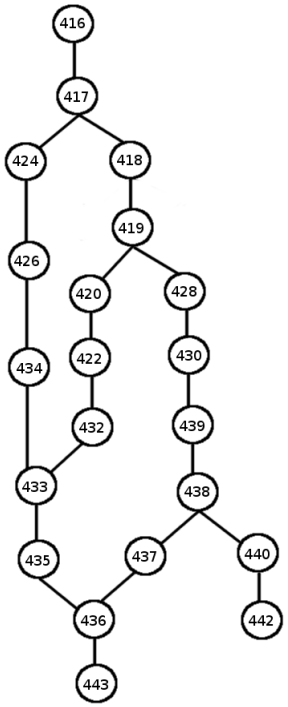
sahující nalezené cesty. Každá cesta představuje sebou vektor ukazatelů na prvky infrastruktury, z nichž se trasa skládá.

Vzhledem ke specifickým vlastnostem infrastruktury, při hledání tras mezi dvěma náhodnými vrcholy grafu můžou vyskytnout cesty délka, kterých několikanásobně překračuje nejkratší trasu. Jednou z příčin proč k tomu dochází, jsou tzv. „zavřené okruhy“, kde vlak může změnit svůj faktický směr pohybu, ale dle definovaných výše vztahů bude to pořád směr lichý. Příkladem může být odbočení vlaku na jedné z výhybek.

Aby k tomu nedocházelo, byl navržen následující algoritmus. Hlavní úvahou při hledání alternativních cest, byla podstata jejich vzniku. Více než jedna cesta mezi dvěma náhodnými vrcholy grafu může vznikat tehdy a jenom tehdy, když mezi nimi existuje alespoň jedna výhybka. Tato skutečnost nastává, protože graf se větví pouze ve výhybkách. Logickým předpokladem na základě tohoto pozorování bylo to, že lze omezit prohledávanou část grafu. Hledání by se ohraničilo cílovým, zdrojovým vrcholem a všemi výhybkami ležícími mezi nimi.

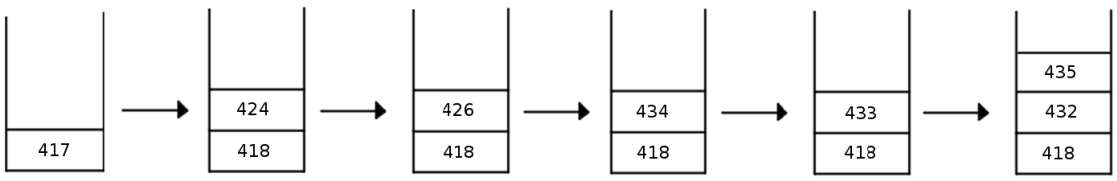
Princip hledání všech tras mezi dvěma úseky je následující, první krok je ověření nejsou-li tyto dva vrcholy shodné a existuje-li mezi nimi cesta. Pokud trasa existuje, hledají se všechny výhybky mezi zadanými prvky. Jestli neleží žádná výhybka, projde se a uloží do vektoru jediná existující trasa. Při výskytu alespoň jedné výhybky může, ale nemusí existovat více cest. Prvním krokem prohledávání všech alternativních cest, je nalezení nejbližší ke zdrojovému prvku výhybky. Ostatní výhybky můžou být pouze její potomky, proto postačujícím je průchod grafu do hloubky. Implementace průchodu grafem je implementována klasicky pomoci datové struktury zásobník, při použití této metody komplikace tvořilo ne nalezení tras samotných, ale doplňování chybějících na zásobníku prvků, podrobněji bude rozebráno na příkladu. Pro názornost detaily metody budou také ukázány na níže uvedeném přikladu. Na obr. 9.6 je zobrazená část infrastruktury, ze které byly pouštěny všechny elementy, kromě výhybek a kolejových úseku, aby nebylo schéma příliš rozsáhlé.

Například budou se hledat trasy mezi kolejovým úsekem č. 416 a 443. Z obrázku je zřejmé, že cesta stejně jako výhybky mezi elementy existují. Dalším krokem je nalezení všech výhybek, v daném příkladu to jsou prvky 417, 419, 433, 438 a 436. Nejbližší ke zdrojovému prvku je výhybka č. 417, proto prvním na zásobníku objeví se právě tento element. V každé iteraci horní prvek na zásobníku uloží se do vektoru $vsechny\_trasy$, na základě kterého pak se samotné cesty prokládají. Vzhledem k tomu, že skutečná infrastruktura neomezená zobrazenými elementy na zásobníku objevují se i výhybky, které nejsou mezi zdrojovým a cílovým prvkem. V tomto případě algoritmus nevloží na zásobník žádný z jeho potomků a do vektoru $vsechny\_trasy$ vloží



Obrázek 9.6: Schéma infrastruktury popisující stanici

se hodnota *NULL* aby potom bylo možné rozpoznat, že sem cesta k cílovému prvku nevede. Pokud se na vrcholu zásobníku vyskytne cílový prvek, to jeho potomky zřejmě nebudou se zpracovávat. Na obr. N zobrazeno prvních šest iteraci práci se zásobníkem.



Obrázek 9.7: Vrcholy grafu na zásobníku

Až zásobník vyprázdní se, je potřeba z uložené do vektoru posloupnosti čísel vytvořit skutečné trasy. V našem případě to bude následující posloupnost:

```
417 418 419 420 421 422 423 432 433 435 436 443 428 429 430 431 439 438 1184 437
    436 443 424 425 426 427 434 433 435 436 443
```

Obecně číselná řada obsahuje nuly, protože vyskytují se výhybky neležící mezi zdrojem a cílem, odstranění takových tras je následujícím krokem algoritmu. Řetězec čísel má více ele-

mentu, než jsou na obrázku, protože byl použit reálný výpis, ve kterém jsou další elementy jako návěstidla a izolované styky. Dále podle počtu výskytu cílového prvku se urči i počet tras. Pro každou se vytvoří vlastní vektor a původní tím pádem rozdělí se na několik, v našem případě tři zobrazené dále.

```
417 418 419 420 421 422 423 432 433 435 436 443
428 429 430 431 439 438 1184 437 436 443
424 425 426 427 434 433 435 436 443
```

Potom obecně může nastat případ, kdy jeden vektor bude podmnožinou jiného, taková situace je ošetřena dále, a to tak, že vektor obsahující méně elementů bude smazán. Jestli začneme studovat výsledné vektory, uvidíme, že jsou to pouze části tras a je potřeba tyto vektory doplnit chybějícími elementy síti. Postup je následující vezme se první element vektoru a bude se procházet infrastruktura v „sudém“ směru pokud nepotká se zdrojový prvek, při tom tyto elementy se ukládají do zkoumaného vektoru. Dále jsou nalezené metodou vektory vráceny.

Tato funkce mezi jiné používá se při doplňování chybějících v jízdních řádech kolejových úseku. Pro připomenutí zmíním, co vlastně JŘ obsahují, jsou to stanice, kterými vlak projíždí, kolejové úseky jednoznačně určující trasu soupravy a odpovídající časy v těchto místech. Po doplnění kolejových úseku a výhybek potřebujeme určit časy příjezdu a odjezdu na každém z nich. K tomu je potřeba vědět dodatečnou informaci, jako například rychlost v jednotlivých úsecích, a nelze jí bohužel získat z nalezených cest metodou *vrat_vsechny_trasy_stack*, protože ona vrací kolekcí typu *PrvekStromu*, které v sobě danou informaci nenesou. Proto byla vytvořena metoda *vrat_ku_misto_ps*.

```
vector<KolejovyUsek*> vrat_ku_misto_ps( vector<PrvekStromu*> vect_ps )
```

Metoda prohledává vektor kolejových úseků a vrací kolekci úseků, které se vyskytují ve vektoru *vect_ps*.

```
vector<Vyhybka*> vrat_vh_misto_ps( vector<PrvekStromu*> vect_ps )
```

Funkce má stejný princip jako předchozí prohledává kolekci výhybek.

9.5 Načítání dat z MySQL databáze

Pro načítání dat z databáze byla vytvořena třída TimeTables dále následuje seznam atributů a implementovaných metod:

```
protected:
  Infrastruktura *infr;
```

```
  vector<Train> trains;
  vector<vector<Track_time>> time_table;
  vector<vector<Track_time>> train_track;
public:
  TimeTables( Infrastruktura *infr );
  void read_tt();
  vector<Train> get_trains();
  vector<vector<int>> get_track_sections();
  void adding_missing_sections();
```

V atributech jsou použité následující struktury:

```
struct Track_time
{
  int track;
  int time_come = 0;
  int time_live = 0;
  int tr_length;
  int tr_vk;
  int tr_vns;
  int tr_v3;
  int tr_vo = 0;
  bool turn = false;
  int brake_ln;
  int brake_speed = 0;
};
struct Train
{
  int train_id;
  int train_ln;
};
```

Struktura $Track_time$ je vytvořená pro ukládání dat o kolejových úsecích: id, čas příjezdu a odjezdu, délka atd. Tato struktura používá se také i pro ukládání výhybek, proto obsahuje atribut *turn*, který určuje, zda vlak na výhybce odbočuje. Struktura *Train* slouží k ukládání informace o vlaku.

```
TimeTables( Infrastruktura *infr );
```

Jízdní řády jsou bezprostředně provázány s infrastrukturou například při doplňování chybějících úseků v jízdních řádech, proto odkaz na infrastrukturu je argumentem konstruktoru.

```
void read_tt();
```

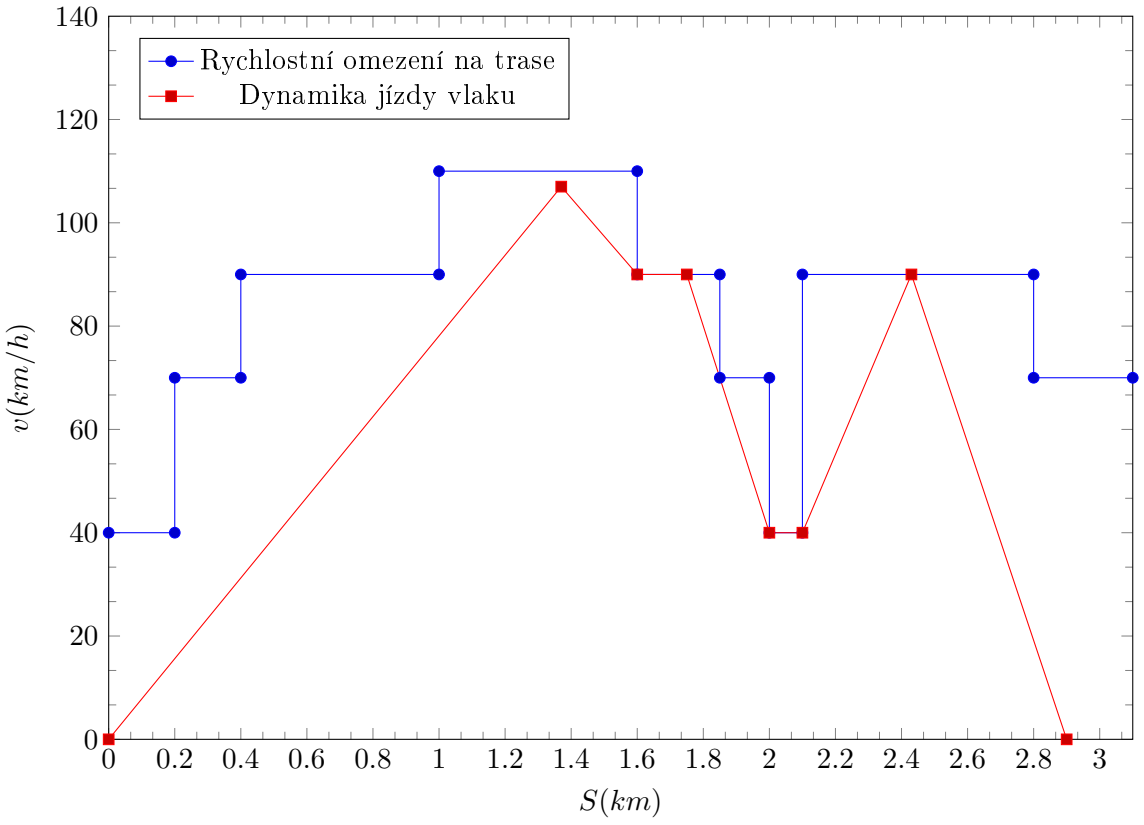
Metoda pomocí MySQL konektoru se připojuje k lokální databázi a pomocí explicitně definovaného selektu ve zdrojovém kódu načítá informace o vlacích a ukládá jí do vektoru *trains*. Díky dalšímu selektu načítá informace o stanicích, dopravních bodech, kterými soupravy projíždějí, a časech průjezdů definovaných v jízdních řádech, táto informace se ukládá do dvojrozměrného vektoru *train_track*.

```
void adding_missing_sections();
```

Jak je zřejmé z názvu, metoda doplňuje kolejové úseky chybějící v jízdním řádu, využívá při tom příslušných metod třídy *Infrastruktura*.

9.6 Návrh výpočtu dynamiky vlaku

Jak se zjistilo při pokusu dopočítat časy vjezdu a opouštění jednotlivých úseků, objem informace o vlaku je nedostatečný pro iterativní výpočet. Následující obrázek ukazuje, jak by vypadal průběh zrychlení respektive zpomalování vlaku při přejezdu mezi dvěma stanicemi.

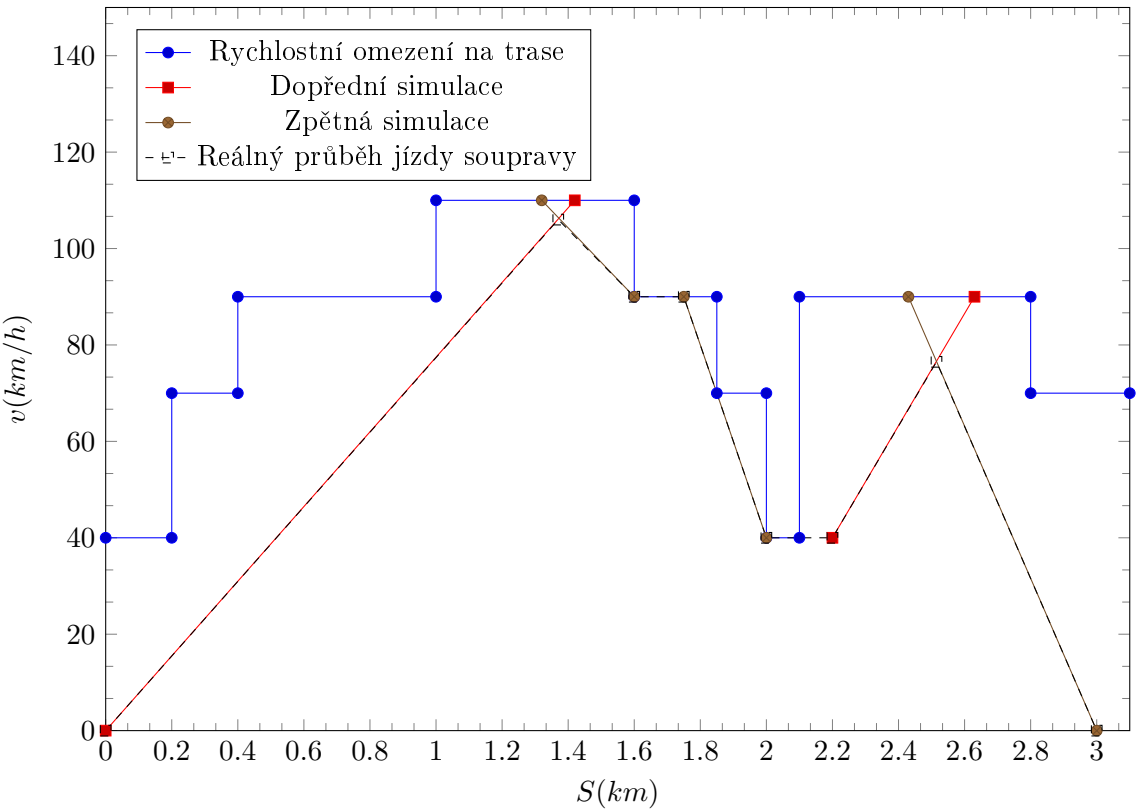


Graf 2. Jízda vlaku mezi dvěma stanicemi

Ve skutečnosti nebude takový graf platit, protože počítá s platnosti omezení rychlosti po vjezdu lokomotivy na kolejový úsek. V reálu při zvyšování rychlosti bude návěstidlo platit pro

soupravu až po vjezdu konce posledního vagonu, s výjimkou speciálních případů (např. kde je doplňující povolující návěst). U takového problému je jednoduché, ale výpočetně náročné řešení. Pro každý vlak se bude předpočítávat rychlostní omezení na infrastruktuře. Například pro soupravu s délkou 200 metrů se omezení posunou o odpovídající vzdálenost. Komplikace spočívá v tom, že se pro každý vlak musí táto informace ukládat zvlášť. Také to znamená, že se bude přepočítávat rychlostní omezení části infrastruktury použité v grafikonu, tj. téměř celá železniční síť. Další komplikaci je různá délka kolejových úseků, v síti se vyskytují úseky s délkou od padesáti metrů do několika kilometrů, což znamená, že na velkých úsecích vznikne několik rychlostních omezení.

Tak že implementace algoritmu, počítajícího časy příjezdu a odjezdu iterativním způsobem, je teoreticky možná, ale prakticky je zbytečně složitá. Při řešení takového druhu úloh je vhodnější metodou diskrétní simulace. Přičemž kvůli brzdné dráze vlaku, která může být delší než jeden kolejový úsek, je potřeba simulovat jízdu ve dvou směrech. Dopředná simulace by modelovala zrychlení soupravy, kdyžto zpětná brzdnou dráhu, jak je ukázáno na následujícím grafu.



Graf 3. Diskrétní simulace

Samotná simulace spočívá v diskrétním posouvání vlaku po infrastruktuře s krokem například jedna sekunda. V každém kroku budeme ukládat informaci o rychlostním omezení úseku, na

kterém se souprava nachází, omezení platné pro soupravu v danou chvíli, aktuální rychlost vlaku, čas a aktuální polohu. Nejdřív se provede dopředná simulace, tj. vlak bude zrychlovat od začátku své trasy, až dosáhne maximální povolené rychlosti na nějakém úseku a v případech, kdy navazující úsek bude mít vyšší rychlostní omezení, protože souprava v takovém případě bude zrychlovat. Potom od konce cesty se bude modelovat snížení rychlosti vlaku. Na začátku se spočítá brzdná dráha soupravy, pak vzdálenosti potřebné pro snížení rychlosti. Reálný průběh vznikne na základě analýzy obou simulací. Výsledky simulací se budou ukládat do struktury obsahující informace o dotčených úsecích. V případech, kdy budou k dispozici výsledky obou simulaci, bude se hledat jejich průnik, jak je ukázáno na dvou úsecích 1 – 1.6 km a 2.1 – 2.8 km z výše uvedeného grafu. Dále, na základě této dynamiky, se provede finální simulace. Během každé iterace se bude kontrolovat, zda vlak vjel nebo opustil kolejový úsek, pokud ano, čas se uloží do příslušného úseku.

Kapitola 10

Závěr

Úkolem této práce byla analýza evolučních technik, jejich aplikace při rozvrhování v železniční dopravě a návrh vlastního evolučního algoritmu pro optimalizaci grafikonu. V práci je podrobně analyzována problematika automatického vyhodnocení jízdního řádu, detekování konfliktů a různé aspekty optimalizace grafikonu obsahujícího odchylky od původně naplánovaného. Na začátku práce s tímto projektem bylo rozhodnuto nejenom navrhnout evoluční algoritmus, ale pro ověření efektivity implementovat. Při detailnějšímu výzkumu se zjistilo, že algoritmus vyžaduje velký objem vstupních informací, kterou je potřeba načíst a dopočítat. Zahrnuje v sobě mimo jiné rozsáhlou úlohu, jako dynamika jízdy vlaku. Dospělo se k závěru, že řešená problematika značně překračuje rámce diplomové práce. Přesto byl vytvořen detailní teoretický rozbor, který by mohl sloužit základem pro navazující práce nad tímto projektem.

Pro shrnutí, vstupy potřebné pro optimalizaci grafikonu: popis topologie infrastruktury včetně parametrů jednotlivých úseků (např. rychlostních omezení), rozšířené jízdní řády umožňující jednoznačně určit trasu soupravy a samostatný modul schopný vypočítat dynamiků více druhů různě zatížených lokomotiv.

Analytické výstupy práce: analýza optimalizace grafikonu, návrh výpočtu dynamiky soupravy a rámcový návrh genetického algoritmu. Praktické: programový modul načítající infrastrukturu, jízdní řády, doplňování chybějící informace o trase soupravy v grafikonu a hledání alternativních cest na infrastruktuře.

Literatura

[1] MAŘÍK, V., O. ŠTĚPÁNKOVÁ, J. LAŽANSKÝ a kol.: *Umělá inteligence (III)*, Praha, Academia, 2001. ISBN 80-200-0472-6.

[2] MAŘÍK, V., O. ŠTĚPÁNKOVÁ, J. LAŽANSKÝ a kol.: *Umělá inteligence (IV)*, Praha, Academia, 2003. ISBN 80-200-1044-0.

[3] ZELINKA I. a kol.: *Evoluční výpočetní techniky*, Praha, BEN 2009, ISBN 978-80-7300-218-3

[4] TORMOS P., et. al, *A Genetic Algorithm for Railway Scheduling Problems*, Studies in Computational Intelligence (SCI) 128, Springer-Verlag, 2008, Berlin

[5] ZHONG J., ZHANG J., CHUNG H.S., SHI Y., LI Y.: *A Differential Evolution Algorithm With Dual Populations for Solving Periodic Railway Timetable Scheduling Problem*, IEEE Transactions on Evolutionary Computation vol. 17, 2013

[6] KOUTECKÝ P.: *Elektronické stavědlo pro dopravní sál FD ČVUT*, diplomová práce, ČVUT FEL, 2011

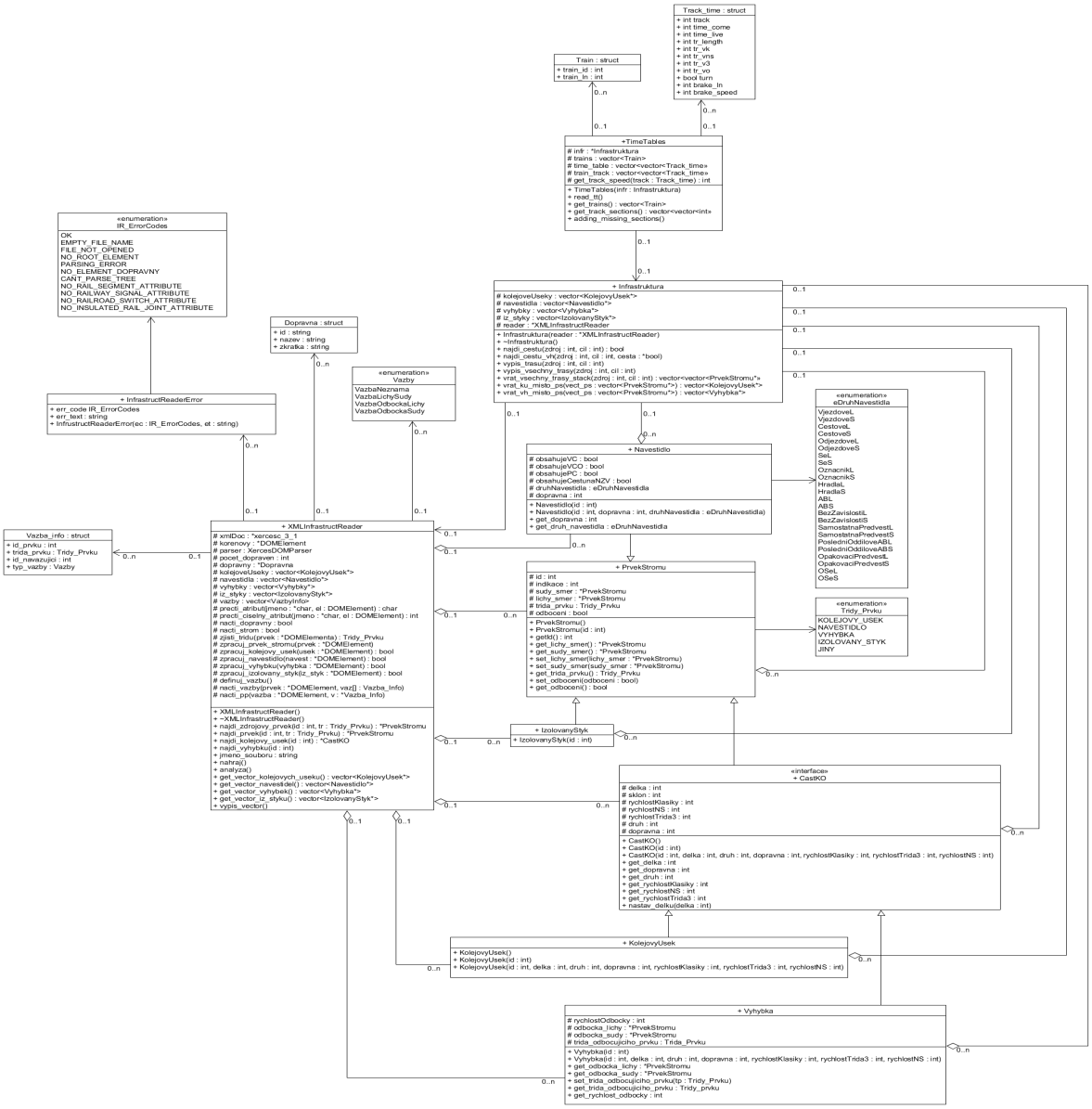
Seznam zkratek

EVT	evoluční výpočetní technika
GA	genetický algoritmus
SGA	standardní genetický algoritmus
DP-DE	diferenciální evoluce s dvojitou populací
GP	globální populace
LP	lokální populace
DSFD	Dopravní sál Fakulty dopravní
ČP	časová populace
PP	prostorová populace

Seznam obrázků

Seznam grafů

Příloha A


Track_time : struct
+ int track
+ int time_come
+ int time_live
+ int tr_length
+ int tr_vk
+ int tr_vns
+ int tr_v3
+ int tr_vo
+ bool turn
+ int brake_ln
+ int brake_speed
Train : struct
+ train_id : int
+ train_ln : int
+TimeTables
infr : *Infrastruktura
trains : vector<Train>
time_table : vector<vector<Track_time»
train_track : vector<vector<Track_time»
get_track_speed(track : Track_time) : int
+ TimeTables(infr : Infrastruktura)
+ read_tt()
+ get_trains() : vector<Train>
+ get_track_sections() : vector<vector<int»
+ adding_missing_sections()
«enumeration»
IR_ErrorCodes
OK
EMPTY_FILE_NAME
FILE_NOT_OPENED
NO_ROOT_ELEMENT
PARSING_ERROR
NO_ELEMENT_DOPRAVNY
CANT_PARSE_TREE
NO_RAIL_SEGMENT_ATTRIBUTE
NO_RAILWAY_SIGNAL_ATTRIBUTE
NO_RAILROAD_SWITCH_ATTRIBUTE
NO_INSULATED_RAIL_JOINT_ATTRIBUTE
+ Infrastruktura
# kolejoveUseky : vector<KolejovyUsek*»
# navestidla : vector<Navestidlo*»
# vyhybky : vector<Vyhybka*»
# iz_styky : vector<IzolovanyStyk*»
# reader : *XMLInfrastructReader
+ Infrastruktura(reader : *XMLInfrastructReader)
+ ~Infrastruktura()
+ najdi_cestu(zdroj : int, cil : int) : bool
+ najdi_cestu_vh(zdroj : int, cil : int, cesta : *bool)
+ vypis_trasu(zdroj : int, cil : int)
+ vypis_vsechny_trasy(zdroj : int, cil : int)
+ vrat_vsechny_trasy_stack(zdroj : int, cil : int) : vector<vector<PrvekStromu*»
+ vrat_ku_misto_ps(vect_ps : vector<PrvekStromu*») : vector<KolejovyUsek*»
+ vrat_vh_misto_ps(vect_ps : vector<PrvekStromu*») : vector<Vyhybka*»
Dopravna : struct
+ id : string
+ nazev : string
+ zkratka : string
«enumeration»
Vazby
VazbaNeznama
VazbaLichySudy
VazbaOdbockaLichy
VazbaOdbockaSudy
+ InfrastructReaderError
+ err_code IR_ErrorCodes
+ err_text : string
+ InfrastructReaderError(ec : IR_ErrorCodes, et : string)
+ Navestidlo
# obsahujeVC : bool
# obsahujeVOD : bool
# obsahujePC : bool
# obsahujeCestunaNZV : bool
# druhNavestidla : eDruhNavestidla
# dopravna : int
+ Navestidlo(id : int)
+ Navestidlo(id : int, dopravna : int, druhNavestidla : eDruhNavestidla)
+ get_dopravna : int
+ get_druh_navestidla : eDruhNavestidla
«enumeration»
eDruhNavestidla
VjezdoveL
VjezdoveS
CestoveL
CestoveS
OdjezdoveL
OdjezdoveS
SeL
SeS
OznacnikL
OznacnikS
HradlaL
HradlaS
ABL
ABS
BezZavislostiL
BezZavislostiS
SamostatnaPredvestL
SamostatnaPredvestS
PosledniOddiloveABL
PosledniOddiloveABS
OpakovaciPredvestL
OpakovaciPredvestS
OSeL
OSeS
Vazba_info : struct
+ id_prvku : int
+ trida_prvku : Tridy_Prvku
+ id_navazujici : int
+ typ_vazby : Vazby
+ XMLInfrastructReader
# xmlDoc : *xercesc_3_1
# korenovy : *DOMElement
# parser : XercesDOMParser
# pocet_dopraven : int
# dopravny : *Dopravna
# kolejoveUseky : vector<KolejovyUsek*»
# navestidla : vector<Navestidlo*»
# vyhybky : vector<Vyhybky*»
# iz_styky : vector<IzolovanyStyk*»
# vazby : vector<VazbyInfo*»
# precti_atribut(jmeno : *char, el : DOMElement) : char
# precti_cislelny_atribut(jmeno : *char, el : DOMElement) : int
# nacti_dopravny : bool
# nacti_strom : bool
# zjisti_tridu(prvek : *DOMElementa): Tridy_Prvku
# zpracuj_prvek_stromu(prvek : *DOMElement)
# zpracuj_kolejovy_usek(usek : *DOMElement) : bool
# zpracuj_navestidlo(navest : *DOMElement) : bool
# zpracuj_vyhybku(vyhybka : *DOMElement) : bool
# zpracuj_izolovany_styk(iz_styk : *DOMElement) : bool
# definuj_vazbu()
# nacti_vazby(prvek : *DOMElement, vaz[] : Vazba_Info)
# nacti_pp(vazba : *DOMElement, v : *Vazba_Info)
+ XMLInfrastructReader()
+ ~XMLInfrastructReader()
+ najdi_zdrojovy_prvek(id : int, tr : Tridy_Prvku) : *PrvekStromu
+ najdi_prvek(id : int, tr : Tridy_Prvku) : *PrvekStromu
+ najdi_kolejovy_usek(id : int) : *CastKO
+ najdi_vyhybku(id : int)
+ jmeno_souboru : string
+ nahraj()
+ analyza()
+ get_vector_kolejovych_useku() : vector<KolejovyUsek*»
+ get_vector_navestidel() : vector<Navestidlo*»
+ get_vector_vyhybek() : vector<Vyhybka*»
+ get_vector_iz_styku() : vector<IzolovanyStyk*»
+ vypis_vector()
+ PrvekStromu
# id : int
# indikace : int
# sudy_smer : *PrvekStromu
# lichy_smer : *PrvekStromu
# trida_prvku : Tridy_Prvku
# odboceni : bool
+ PrvekStromu()
+ PrvekStromu(id : int)
+ getId() : int
+ get_lichy_smer() : *PrvekStromu
+ get_sudy_smer() : *PrvekStromu
+ set_lichy_smer(lichy_smer : *PrvekStromu)
+ set_sudy_smer(sudy_smer : *PrvekStromu)
+ get_trida_prvku() : Tridy_Prvku
+ set_odboceni(odboceni : bool)
+ get_odboceni() : bool
«enumeration»
Tridy_Prvku
KOLEJOVY_USEK
NAVESTIDLO
VYHYBKA
IZOLOVANY_STYK
JINY
+ IzolovanyStyk
+ IzolovanyStyk(id : int)
«interface»
+ CastKO
# delka : int
# sklon : int
# rychlostKlasiky : int
# rychlostNS : int
# rychlostTrida3 : int
# druh : int
# dopravna : int
+ CastKO()
+ CastKO(id : int)
+ CastKO(id : int, delka : int, druh : int, dopravna : int, rychlostKlasiky : int, rychlostTrida3 : int, rychlostNS : int)
+ get_delka : int
+ get_dopravna : int
+ get_druh : int
+ get_rychlostKlasiky : int
+ get_rychlostNS : int
+ get_rychlostTrida3 : int
+ nastav_delku(delka : int)
+ KolejovyUsek
+ KolejovyUsek()
+ KolejovyUsek(id : int)
+ KolejovyUsek(id : int, delka : int, druh : int, dopravna : int, rychlostKlasiky : int, rychlostTrida3 : int, rychlostNS : int)
+ Vyhybka
# rychlostOdbocky : int
# odbocka_lichy : *PrvekStromu
# odbocka_sudy : *PrvekStromu
# trida_odbocujiciho_prvku : Trida_Prvku
+ Vyhybka(id : int)
+ Vyhybka(id : int, delka : int, druh : int, dopravna : int, rychlostKlasiky : int, rychlostTrida3 : int, rychlostNS : int)
+ get_odbocka_lichy : *PrvekStromu
+ get_odbocka_sudy : *PrvekStromu
+ set_trida_odbocujiciho_prvku(tp : Tridy_Prvku)
+ get_trida_odbocujiciho_prvku : Tridy_prvku
+ get_rychlost_odbocky : int

Příloha B

Všechny zdrojové kódy včetně modulů pro načítání infrastruktury, jízdních řádů a práci s načteným grafem se nachází v komprimovaném souboru, který je k dispozici v elektronické podobě. Tento soubor také obsahuje XML soubor s popisem infrastruktury, SQL soubor pro vytvoření uživatele (definovaného ve zdrojovém kódu) a schématu obsahující JŘ (včetně vytvoření tabulek, vazeb a naplnění základními daty), a Makefile řídicí kompilaci a linkování zdrojových kódů.