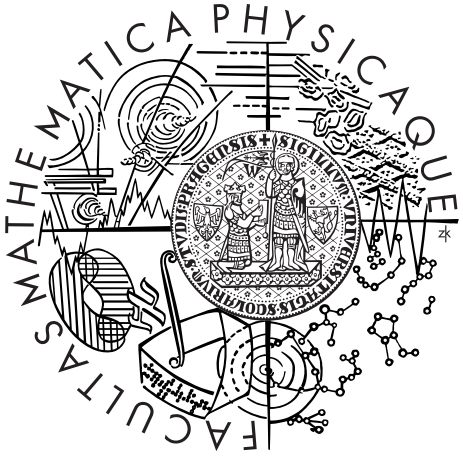
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Dominik Janata

Prostředí pro tvorbu jízdních řádů

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.
Studijní program: Informatika
Studijní obor: Programování

Praha 2014

Poděkování.

Děkuji vedoucímu této práce, RNDr. Tomáši Holanovi, Ph.D., za průběžné konzultace, usměrňování práce a také za to, že mi umožnil významnou část práce vypracovávat v průběhu studijního pobytu Erasmus.

Dále děkuji ing. Miroslavu Janatovi za zpětnou vazbu ke koncepci práce a k logické stavbě jejího textu.

Děkuji také Bc. Zbyňku Jiráčkovi a Josefu Bártovi za zpětnou vazbu k textu práce.

Mé díky také patří Bc. Aleši Zenáhlíkovi za vytvoření grafiky pro webové stránky práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Prostředí pro tvorbu jízdních řádů

Autor: Dominik Janata

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D., Kabinet software a výuky informatiky MFF UK

Abstrakt: Tato práce se zabývá problémem algoritmického generování jízdních řádů. Hlavní aspekt, na který je kladen důraz, je vztah kvality jízdních řádů a požadavků, které na systém mají jeho uživatelé, tedy cestující.

Práce má čtyři hlavní výstupy. Prvním výstupem je modulární systém, který modeluje dopravní síť a umožňuje testovat samostatně dodané generovací algoritmy. Druhým je implementace generování jízdních řádů pomocí genetického algoritmu. Třetím je určité množství vyzkoušených metrik kvality jízdního řádu. Čtvrtým je sada ukázkových implementací, která umožňuje externím programátorům snadno vytvářet vlastní implementace modulů programu; ta je k dispozici na internetových stránkách projektu.

Klíčová slova: Jízdní řády, genetický algoritmus, cestující, hodnocení

Title: Timetable Designing Environment

Author: Dominik Janata

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Tomáš Holan, Ph.D., Department of Software and Computer Science Education

Abstract: The work treats the problem of algorithmic generation of traffic schedules. The emphasised aspect is the relationship between the quality of the schedules and the requirements the customers of the traffic system, being the passengers, put on it.

The work has four major outputs. The first one is a system modelling the traffic network and providing possibility to test generating algorithms. The second is an implentation of genetic algorithm for generating traffic schedules. The third is a set of tested metrics of traffic schedule quality. The fourth is a set of developer packages providing external programmers easy way of implementating the modules of the program; the set is available on the web pages of the project.

Keywords: Traffic schedules, genetic algorithm, passengers, rating

Obsah

Úvod

Výpočetní modely a simulace pomáhají řešit úlohy, na které lidé sami nestačí. Například pomáhají zpřesňovat předpověď počasí či modelovat fyzikální experimenty. Tyto úlohy jsou charakteristické tím, že jejich základem je velké množství prvků, z nichž každý sám o sobě má jasně popsané chování. Počítač umožňuje v rozumném čase vypočítat jejich chování a vzájemné interakce a v případě fyzikálních experimentů ušetřit úsilí a materiál vynaložený na marné pokusy. Zde se pokusíme přiblížit, že i hromadná doprava tvoří spolu s cestujícími systém, který se vyplatí takto simulovat.

Dopravní infrastruktura velkoměst se rozrůstá s tím, jak roste jejich populace, a s tím se zvyšuje i vytíženost městské hromadné dopravy. Ta tomu musí být přizpůsobována, zvyšuje se přepravní kapacita, počet linek i rozsah sítě. Důsledkem je, že není v lidských silách tento systém exaktně upočítat. Plánovači proto sahají k používání modelů, kterými usměrňují toky cestujících městem. Tento přístup je přijatelný, pokud jde o přepravní kapacitu, alespoň v průměrných případech. Problém je, že cestující, jejichž požadavky do těchto modelů nezapadají, jsou nespokojeni. A mají pocit, že jsou „těm nahoře“ lhostejní. Počet takto nespokojených cestujících není zanedbatelný. Poměrně běžná námitka zní: „Proč mi teď přijela přeplněná tramvaj, když touhle dobou na lince XY jezdí co tři minuty poloprázdné vagony?“

V tomto kontextu se použití výpočetní techniky na modelování chování městské hromadné dopravy jeví jako velice přínosné. Ať už k pomocným výpočtům, nebo přímo k plánování jízdních řádů. Pokud budeme za prvky dopravní sítě považovat vozidla a cestující, pak jich řádově budou desítky milionů. Chování vozidel je určené jízdním řádem. Cestující žádný pevný řád nemají, ale jejich významná část má pravidelné trasy, například do práce a z práce. Tyto pravidelné trasy už mohou tvořit základ pro modelování chování cestujících.

Tato práce přináší základ výpočetního systému odpovídajícího výše popsaným požadavkům. Řečený systém naplňuje funkci prostředí pro vytváření jízdních řádů tím, že je zcela modulární, a každý z modulů je možné vyvíjet samostatně. Samotnému vytváření jízdních řádů je zasvěcen jeden modul, ostatní dodávají parametry dopravní sítě, jako jsou topologie sítě či data o cestujících, čímž společně vytváří prostředí pro generátor jízdního řádu. Smyslem práce je, stát se nástrojem pro lidi, kteří chtějí či potřebují vytvářet algoritmy generující jízdní řády. Proto budou na internetu vystaveny některé její části, konkrétně knihovny potřebné pro vývoj modulů, ukázkové implementace modulů a dokumentace knihoven.

Součástí práce je i implementace vytváření jízdních řádů pomocí genetického algoritmu. Tato implementace je doplněna sadou experimentů, které umožnily odpovědět na otázku, zda je genetický algoritmus vhodná metoda pro hledání optimálního jízdního řádu. Experimentem se rozumí, že je ke genetickému algoritmu dodána určitá topologie a data o cestujících, a genetický algoritmus pak vydá nějaký jízdní řád jako nejlepší řešení; tento jízdní řád bude i výstupem experimentu.

V bodech jsou cíle práce následující:

1. Vytvořit model dopravní sítě a vymodelovat vztah cestujících a jízdního řádu, podle kterého vozidla této sítě jezdí.
2. Vytvořit metriku, která při hodnocení jízdního řádu bude zohledňovat zejména jeho soulad s potřebami cestujících.
3. Vyzkoušet genetický algoritmus jako nástroj k hledání optimálního jízdního řádu.
4. Provést experimenty s genetickým algoritmem a jejich výsledky využít k zhodnocení přínosu této metody pro generování JŘ.
5. Vytvořit zázemí pro třetí strany, aby mohly vytvářet vlastní implementace modulů, což znamená
   - Pro každý z modulů vytvořit ukázkovou implementaci, která umožní samostatný vývoj modulu.
   - Vytvořit webový portál, který bude prezentovat projekt a dávat k dispozici ukázkové implementace.

Práce je dále členěna do těchto kapitol:

V kapitole 1 je provedena analýza, která pokrývá strukturu programu, problémy a jejich řešení; některé z těchto problémů nemají jediné správné řešení, v takových případech jde spíš o rozhodnutí, které řešení nejlépe odpovídá filozofii nebo koncepci práce.

Kapitola 2 obsahuje uživatelskou dokumentaci, tedy popisuje popořadě práci uživatele s programem počínaje instalací a konče orientací v grafickém rozhraní.

Kapitola 3 obsahuje programátorskou dokumentaci, je zde popsána struktura zdrojového kódu, u některých částí kódu i myšlenky, které jsou „za ním“ a ke každé třídě je nějaký komentář.

Kapitola 4 pokrývá experimentální část, jsou zde popsány experimenty tak, jak byly postupně prováděny. Často návrh dalších experimentů vycházel z pozorování a závěrů uskutečněných na základě výsledků předcházejících experimentů, tento proces je v této kapitole také průběžně popisován.

Slovník používaných pojmů

Konfigurace	V kontextu genetického algoritmu jde o kombinaci způsobu reprezentace jízdního řádu genomem a hodnotící metriky.
Itinerář [cestujícího]	Posloupnost všech stanic, jimiž cestující projede.
Linka	Pojmenovaná dvojice tras (tam a zpět) mezi dvěma konečnými stanicemi.
Požadavek cestujícího	Tvořený výchozí a cílovou stanicí a časem příchodu do výchozí stanice.
Scénář	Pojmenovaný výčet požadavků cestujících.
Spoj	Jedna cesta soupravy podél jedné trasy linky. Například jeden spoj linky 12 je jedna cesta soupravy ze Sídliště Barrandov na Palmovku.
Stanice	Nejlépe lze pochopit na příkladu: tramvajová superstanice (viz následující heslo) Malostranské náměstí má stanici pro směr Anděl a pro směr Malostranská.
Superstanice	Jak naznačuje definice stanice, superstanice je množina stanic, mezi nimiž vedou virtuální hrany. V rámci superstanice se cestující přesouvají bez využití hromadné dopravy, tedy pokud vezmeme graf dopravní sítě se stanicemi jako vrcholy, pak superstanice je jeho úplný podgraf tvořený výhradně virtuálními hranami. Existují situace, kdy spoj v rámci jedné superstanice projíždí více stanicemi. V takovém případě vede mezi těmito stanicemi zároveň hrana i virtuální hrana.
Virtuální hrana	Hrana grafu dopravní sítě, po níž se cestující přesouvají bez využití spoje.

1. Analýza

1.1 Problematika

V úvodu jsme nastínili hlavní problémy, které budou v práci řešeny. Zde popíšeme, jak budou vyřešeny a proč.

1.1.1 Moduly prostředí

V úvodu je také zdůrazněný požadavek na modularitu výsledného systému. Ta by měla zajistit možnost samostatně implementovat následující moduly (slovem „samostatně“ myslíme bez potřeby znalosti implementace ostatních modulů):

- Topologie dopravní sítě
- Data o cestujících
- Generátor jízdních řádů

Výše uvedené tři moduly si mezi sebou budou vyměňovat data. Tuto výměnu dat by usnadnil nějaký „společný jazyk“ těchto modulů. Např. pokud bude v rámci topologie definována nějaká stanice, bude potřeba, aby na ni bylo možné z dat o cestujících nějak odkázat (např. pokud z/do této stanice pojede cestující). Stejnětak pokud se s ní bude pracovat v generátoru jízdních řádů. Z tohoto důvodu zavedeme knihovnu Core, která bude obsahovat definice tříd používaných napříč více moduly, případně i nějaké knihovny pro práci s těmi třídami.

Už výše popsaná čtveřice modulů by mohla splnit požadavek na vytvoření prostředí pro generování jízdních řádů. Jimi tvořenému systému by ale určitě chyběl jednotící prvek, který by umožňoval porovnávání jízdních řádů, které by v rámci systému vznikaly.

Proto potřebujeme hodnocení. Nemáme k dispozici žádnou metriku, kterou bychom kvalitu jízdního řádu měřili, proto ji budeme muset sami stanovit. V úvodu je řečeno, že tato práce má i experimentální část, čili do ní přidáme i testy různých metrik.

Při hodnocení jízdního řádu je nutné analyzovat data, která máme k dispozici, a z nich spočítat číselný koeficient. Tuto analýzu je možné provést tak, že budeme simulovat chování dopravní sítě a analyzovat budeme data vznikající v průběhu simulace. Tento přístup by mohl být perspektivní z následujících důvodů:

1. Jádrem vznikajícího systému se stane simulátor, jehož vstupy mohou být dodány zvlášť coby moduly.
2. Simulátor zajistí pro generátor jízdních řádů analýzu ostatních vstupních dat, pokud bude jeho výstup předáván do generátoru coby zpětná vazba a základ pro další výpočty.
3. Simulátor zajistí pro hodnocení jízdních řádů analýzu veškerých vstupních dat, včetně výsledného jízdního řádu.

Takto nám k výčtu modulů přibývají ještě dva: simulátor a hodnotič.

K datům o cestující se ještě dostaneme později, v souvislosti s moduly je ale třeba zmínit ještě jednu věc. Data o cestujících popisují jen jejich požadavky, což je (jak blíže ji popisujeme v následujících podkapitolách) nedostatečná informace pro simulátor, který potřebuje itinerář. Z toho důvodu je zapotřebí nějakým způsobem do toku dat zapojit převod požadavku cestujícího na itinerář. Bylo by možné „zadrátovat“ nějaký algoritmus do simulátoru. Hned v následující podkapitole ale ukazujeme, že znalost sítě umožňuje vyvinout pro síť na míru nějaký algoritmus převodu (například je také možné mít cesty předpočítané v databázi a jen je načítat). Z těchto důvodů i převod požadavků cestujících na itineráře oddělíme do samostatného modulu.

V následujících podkapitolách výše jmenované moduly blíže ji rozebereme.

1.1.2 Dopravní síť

Model

Základem modelu dopravní sítě bude graf, jehož vrcholy jsou stanice a hrany jsou koleje nebo úseky silnic, po nichž jezdí autobusy. Protože na tomto grafu bude probíhat simulace pohybu pasažérů, bude muset implementace umožňovat umístění cestujících do stanice a jejich čekání na spoj. Tím se dostáváme k dalším entitám, což jsou spoje, které musí příslušet k nějaké lince a nějaké trase linky. Spoje, stejně jako stanice, musí udržovat výčet cestujících, kteří spojem právě jedou a čekají, až spoj dojede do určité stanice. Pokud je dopravní síť tvořena více autonomními sítěmi (např. tramvaje, autobusy), pak musí model umožňovat rozlišení, do které sítě linka patří.

Prozatím si zjednodušíme práci tím, že trasy linek budeme považovat za pevně dané (tedy za vlastnost sítě) a že budeme pracovat pouze se sítěmi, které nejsou tvořeny více samostatnými sítěmi.

Heuristika

Na podobě většiny dopravních sítí si můžeme všimnout dvou typických rysů:

1. Stanice jsou typicky umisťovány ve dvojicích, kdy spoje projíždějících linek na své trase v jednom směru staví v jedné a v opačném směru v druhé.

2. Pokud sdružíme stanice do superstanic tak, jak jsme si je definovali ve Slovníku, dostaneme graf, v němž jsou dlouhé úseky vrcholů stupně dva, tedy úseky, které se nevětví. Uvnitř těchto úseků jsou mezi dvojicemi superstanic nejrychlejší trasy jednoznačně dané.

Z toho můžeme po řadě vyvodit následující závěry:

1. Graf stanic můžeme nahradit grafem superstanic. Superstanice je pak objekt, který se v kontextu grafu superstanic chová jako vrchol, a navíc obsahuje úplný graf na několika málo (typicky dvou) vrcholech.

2. Graf superstanic můžeme nahradit grafem nějakých složitějších objektů, které se v kontextu grafu chovají jako vrcholy, a navíc obsahují část grafu superstanic. V rámci této části poskytují relativně rychlý výpočet nejrychlejší trasy, protože část obsahuje řádově jednotky vrcholů.

Všimnneme si tedy, že jde o jakousi rekurzivní strukturu nad původním grafem stanic: na nejvyšší úrovni máme graf „nějakých složitějších objektů“ — říkejme jim odteď shluky —, který dělí vrcholy původního grafu do skupin podle určité charakteristiky. Tyto skupiny jsou obaleny do vrcholů tohoto grafu, ty v rámci nich poskytují relativně rychlé vyhledávání nejrychlejší cesty.

Když se podíváme na libovolnou s těchto skupin, zjistíme, že je také složená z vrcholů obsahujících malé grafy: vrcholy jsou superstanice a obsahují úplné grafy na stanicích, mezi nimiž vedou virtuální hrany (mohou mezi nimi vést i hrany „nevirtuální“, ale to v tomto bodě není podstatné).

Připomeňme, že původním zadáním bylo najít nejkratší cestu grafem **stanic**. K tomu nám výše popsané úvahy pomohou takto: pro výchozí a cílovou stanici najdeme příslušný shluk. Shluk obsahuje obvykle kolem 10 superstanic, tedy cca 20 stanic. Vezměme jako příklad pražskou tramvajovou síť, která má v našich datech 578 stanic. Pro ni to dělá 289 superstanic, tedy přibližně 29 shluků.

Při procházení algoritmem se složitostí $N \times log(N)$ to u původního grafu dělá $578 \times 6,359 = 3676$ operací na požadavek. V případě shluků hledáme nejprve nejrychlejší cestu nad $29 \times 3,36 = 98$ operací. Navíc pro každou dvojici shluků v této nalezené cestě musíme najít hranu z původního grafu, která mezi nimi vede. Tu můžeme uložit do hashovací struktury, tedy tato operace může trvat $O(1)$. Navíc musíme uvnitř shluku najít cestu mezi stanicí, v níž do něj trasa vstoupila (tu udává předchozí shluk), a stanicí, v níž jej opustila (tu udává následující shluk. Počítáme 10 superstanic na shluk, tedy tato operace dělá $10 \times 2,3 = 23$. Mezi superstanicemi dohledáme hrany opět v čase $O(1)$, stejně, jako mezi shluky.

Celkem v této heuristice tedy máme $98 + 29 \times 23 = 765$ operací v nejhorším případě. Při celkovém počtu $\binom{578}{2} = 166753$ možných různých požadavků to není zanedbatelná úspora.

Definice 1. *Kontrahovaný graf* $K_G = (V', E')$ *grafu* $G = (V, E)$ *je graf, kde*
$V' = \{v \in V : deg(v) > 2\} \cup \{v_P : \exists P = (W = v_1, ..., v_n = X), n \geq 3, deg(W) \neq 2, deg(X) \neq 2, \forall i \in \{2..n-1\} deg(v_i) = 2\}$
$E' = \{(v, w) : deg_G(v) > 2, deg_G(w) > 2, (v, w) \in E\} \cup \{(v, v_P), (v_P, w) : \exists P\}$ *je cesta jako v definici* V'

Shrňme zde, jaké datové struktury budeme potřebovat pro implementaci shluků nad grafem stanic.

Shluk obsahuje:

- Výčet shluků nižší úrovně hierarchie, které obsahuje (tím bude obsahovat graf).
- Výčet svých sousedů (tím bude implementovat vrchol grafu).
- Pro každý sousední shluk hranu původního grafu, která tyto shluky spojuje (musí existovat jediná, pokud neexistuje, rozdělíme shluk).

Jinak řečeno, shluk funguje zároveň v roli:

- Vrcholu grafu shluků dané úrovně hierarchie.
- Entity obsahující graf (výsek původního grafu).
- Entity poskytující funkcionalitu pro rychlé vyhledávání v grafu shluků nižší úrovně, který obsahuje.

Shluk každé úrovně musí obsahovat metodu, která pro každé dva vrcholy původního grafu, které obsahuje, vrátí cestu mezi nimi listovou úrovní této hierarchie. Listová úroveň hierarchie při implementaci v programu odpovídá jednotlivým stanicím. Protože se dotazy na stejné cesty budou opakovat, bude muset být ve shluku cache na již vyhledané trasy.

Nyní místo obecného případu vezměme náš případ, kdy vrcholy původního grafu jsou stanice. V kořeni hierarchie musí pro každou stanici být seznam shluků

napříč úrovněmi hierarchií, které stanici obsahují. Speciálně shluk na posledním místě tohoto seznamu bude pro všechny stanice společný, protože kořen hierarchie zahrnuje celý původní graf. Hledání cesty mezi dvěma stanicemi S_1, S_2 pak proběhne takto:

1. Nalezení nejnižší úrovně, na níž jsou stanice obsaženy ve stejném shluku; označme tento shluk C
2. Označme C_1 shluk o jedna nižší úrovně, do něhož patří S_1, a analogicky C_2 pro S_2
3. Nalezení cesty $(C_1 = c_1, ..., c_n = C_2)$ mezi C_1 a C_2 v grafu shluků uvnitř C
4. Pro každou dvojici (c_i, c_{i+1}):
   - c_i má uloženou informaci, která stanice slouží pro opuštění c_i směrem do c_{i+1}, označme ji $S_{i_{OUT}}$.
   - c_{i+1} má uloženou informaci, která stanice slouží pro vstup do c_{i+1} z c_i, označme ji $S_{i+1_{IN}}$.
5. Tyto dvojice stanic se stanou součástí výsledné cesty, a mezi těmito dvojicemi dohledáme cestu tak, že pro každou dvojici $(S_{i_{IN}}, S_{i_{OUT}})$ najdeme rekurzivně stejným způsobem cestu v c_i.

V knihovně, která bude sdílená se všemi moduly, implementujeme algoritmus převodu grafu na kontrahovaný graf a vyhledávání trasy v kontrahovaném grafu.

Převod grafu na kontrahovaný graf

Hierarchie stromové struktury nad kontrahovaným grafem je číslována tak, že listová úroveň má pořadí 0 a kořen je v $(n-1)$-ní úrovni, kde n je počet úrovní; v našem případě je pevně dáno, že $n = 4$.

První krok je vytvořit ze stanic nultou, tedy listovou úroveň hierarchie tím, že každou stanici „obalíme“ shlukem. Ten umožní zařazení stanice do hierarchie shluků.

Ve druhém kroku stanice sloučíme shluků reprezentujích superstanice, tím vytvoříme první úroveň.

Po této části musíme stanice, které nejsou připojeny žádnou virtuální hranou, zařadit do samostatných shluků reprezentujících superstanice.

Nakonec musíme zaevidovat hrany vedoucí mezi různými superstanicemi. Tím je druhý krok hotov.

Ve třetím kroku se snažíme hledat posloupnosti superstanic stupně dva.

Nejprve najdeme všechny konečné stanice, to jsou stanice se stupněm jedna. Pokud není žádná, znamená to, že každý vrchol grafu leží na kružnici, a jako konečnou stanici vybereme libovolnou z nich. Není to nic proti ničemu, konečné stanice jsme potřebovali jen jako výchozí body pro spouštění prohledávání do hloubky.

Prohledávání do hloubky bude pracovat takto: Takto máme vytvořenou druhou úroveň a zbývá jen přidat první úroveň, v níž je pouze kořen.

Algoritmus 1 Převod grafu stanic na graf superstanic

```
Seřaď hrany podle ID výchozí stanice vzestupně
for all virtuální hrana e = (v1, v2) do
    if v1 nemá přiřazenou superstanici then
        Vytvoř novou superstanici
        Přiřaď do něj obě stanice spojené hranou
    else if v2 nemá přiřazenou superstanici then
        Přiřaď jej pod superstanici od výchozí stanice
    else if v1.superstation ≠ v2.superstation then
        Vyhoď výjimku
    end if
end for
```

Algoritmus 2 Převod grafu superstanic na kontrahovaný graf

```
while je nenavštívená cílová stanice do
    Vezmi nenavštívenou cílovou stanici a vytvoř pro ni shluk
    if Jsou méně než dvě sousední stanice then
        Pokračuj do té z nich, která není navštívená, pokud taková není, break
    else
        Uzavři shluk
        for all Pro každého ze sousedů do
            Vytvoř shluk
            Pokračuj do souseda
        end for
    end if
end while
```

1.1.3 Trasy cestujících

Řekli jsme, že požadavek cestujícího je trojice výchozího bodu jeho cesty, koncového bodu jeho cesty a času jeho příchodu do výchozího bodu.

Hned v první řadě se nabízí pochybnost, proč čas příchodu do výchozího bodu, když pro skutečné cestující bude čas příchodu do koncového bodu ten předem daný.

V případě, že bychom do požadavku cestujícího dali údaj o zamýšleném příjezdu do koncového bodu cesty, bychom stejně potřebovali dopočítat čas příchodu cestujícího do výchozí stanice, abychom mohli simulovat jeho cestu systémem. Z hlediska simulátoru je smysluplný čas příchodu do výchozí stanice. Tím pádem případná újma realističnosti modelu vznikne jedině v generátoru cestujících v případě, že v jeho implementaci nebude zohledněn tento fakt.

Dále je zde otázka, jak se bude určovat trasa cestujícího. Konzultace se skutečnými cestujícími ukázaly dva přístupy: jeden typ cestujících si předem vyhledává pro svou trasu ideální spojení (například pomocí portálu IDOS), druhý typ má svou trasu a rozhoduje se na místě, tedy ve své aktuální stanici vstoupí do prvního vyhovujícího spoje. Očividně je první přístup mnohem vhodnější pro tento systém, protože umožňuje pro cestující předpočítat trasy na základě jejich požadavků, a v průběhu simulace již žádnou další rozhodovací logiku neimple-

mentovat. Proto byl tento přístup zvolen. Prozkoumáme nyní, jak velká újma realitě byla tímto způsobena.

Itineráře nalezené prvním způsobem minimalizují čistou dobu cesty cestujícího. Předpokládáme, že proto lépe modelují pravidelné cesty, jako jsou například cesty z domova do práce nebo do školy. Naopak, pokud cestující jede, aniž by před tím zjišťoval trasu a jízdní řád, znamená to, že je s dopravní sítí obeznámen, a v takovém případě se bude jeho trasa nejspíše blížit optimální. Vychází ze svých zkušeností, případně ze zažitých stereotypů; v každém případě, v našem světě nehrozí zpoždění spoje, a skutečná doba cesty spoje se rovná teoretické, tedy dané ohodnocením hran.

Z hlediska věrohodnosti modelu má průběžné rozhodování cestujících výhodu v případě nenadálých událostí (nehoda, dopravní zácpa), které mohou vytvořit rozdíl mezi zmíněnou teoretickou a skutečnou dobou cesty spoje. Pokud je itinerář daný napevno, musí cestující čekat na to, až bude tato událost odstraněna. Pokud se cestující rozhoduje průběžně, má šanci na tuto událost reagovat. Náš model ovšem takovéto situace neumožňuje, je to jedna z položek našeho níže uvedeného výčtu zjednodušení reality.

Co by mohlo dále odradit cestujícího od spojů s optimální dobou cesty, je přeplněnost spojů, kterými by měl jet. Té by ale měl předejít generátor jízdních řádů tím, že alokuje adekvátní počet vozidel pro příslušné linky. Neměla by se tedy přizpůsobovat trasa cestujícího alokaci vozidel, ale naopak alokace vozidel by měla vycházet z požadavků cestujících, které se dají převést na itineráře cest s minimální čistou dobou.

Jako závěr z těchto úvah připustíme zjednodušení otázky volby tras cestujících, protože přínos implementace výše uvedeného způsobu chování by znamenal nepřiměřeně náročný zásah do návrhu simulátoru a do algoritmů v něm probíhajících.

1.1.4 Podoba generovaných jízdních řádů

Ve skutečném světě je to, co vidíme z jízdních řádů, jen řada časů příjezdů linek do stanic. Této podobě předchází (alespoň v Praze) seznam odjezdů z výchozích stanic spojů, ze kterého se na základě chronometrických dat dopočítají příjezdy do všech zastávek po cestě. Tyto odjezdy bývají velmi pravidelné, typicky po dobu několika hodin drží stejný interval. To znamená, že jízdní řád jedné linky je inverzní funkce počtu vozidel, která jsou na ni alokovaná. Zjednodušený příklad: jestliže linka má trasu, kterou trvá projet 20 minut v každém směru, a interval je 8 minut, pak tento interval odpovídá pěti vozidlům (pokud započítáme nějakou minimální dobu, kterou řidič musí mít čas si odpočinout na konečné, tak šesti nebo více; s touto okolností bude algoritmus počítat jako s proměnnou, v našich pokusech ji ale necháme nulovou).

Tato transformace je velmi důležitá, protože jedním z nejzásadnějších omezení, které generování jízdních řádů bude mít, bude počet souprav k dispozici. Také validace jízdního řádu, tedy kontrola jeho korektnosti, bude zjišťovat, jestli počet využitých souprav nepřekračuje toto číslo. Touto transformací převádíme problém generování jízdních řádů na problém optimálního rozdělení vozidel mezi linky, a to nám dokonce umožňuje vyčíslit počet možných jízdních řádů. Pokud bychom nechali na linkách konstantní interval celý den, byl by tento počet $\binom{v+l}{l}$, kde v je

počet vozidel a l je počet linek.

Ve skutečnosti je potřebné, aby se v průběhu dne interval měnil, a proto si můžeme den rozdělit na více časových úseků, označme jejich počet p. Snadno vidíme, že s tímto rozdělením je počet možných jízdních řádů $p \times \binom{v+l}{l}$.

1.1.5 Chování simulátoru

Simulátor nejprve musí jízdní řád zvalidovat, tzn. zkontrolovat, jestli spoje jezdí po korektních trasách a jestli není využito více vozidel, než kolik jich je v dopravní síti k dispozici.

Po validaci se spustí simulace, a ta bude probíhat takto:

1. Seřadí položky jízdního řádu podle času odjezdu spoje vzestupně. Tím vznikne simulační kalendář.

2. Seřadí itineráře cestujících podle času příchodu na výchozí stanici vzestupně. Tím vznikne druhý simulační kalendář.

3. Dokud jsou itineráře s časem příchodu menším nebo rovným časem odjezdu spoje, tak jsou pasažéři přiřazováni na stanice

4. Zpracovává položky jízdního řádu, dokud tím neporušuje předchozí podmínku, tedy

   (a) Nechá vystoupit všechny cestující, kteří vystupují v dané zastávce. Ti, kteří končí, jsou odstraněni ze simulace, ostatní jsou zařazeni do fronty k sousední stanici, která je shodná s následující položkou na jejich itineráři.

   (b) Nechá nastoupit všechny cestující z fronty ke stanici, která odpovídá následující stanici na trase spoje, ale jen do počtu odpovídající volné kapacitě spoje. Každý z těchto cestujících je ve spoji zařazen do fronty na výstup ve stanici, která je v jejich itineráři poslední taková, že jí daný spoj projíždí (jinými slovy, spoj odveze cestujícího co nejdál může, tedy co nejdál se jeho trasa shoduje s trasou cestujícího).

1.1.6 Ohodnocovací funkce

Již bylo dříve řečeno, že ohodnocovacích funkcí vyzkoušíme víc. Jednu veličinu ale budou mít společnou, protože je zásadní. Tou je podíl čisté a hrubé doby cesty cestujícího. Čistá doba cesty je doba, kterou cestující v průběhu cesty strávil ve spojích a přestupováním, hrubá navíc obsahuje i dobu čekání v zastávkách. Řekli jsme, že se snažíme maximálně vyhovět požadavkům cestujících. Obecně se dá říci, že požadavku cestujícího maximálně vyhovíme tím, že ho přepravíme v minimálním čase. Musíme vzít v úvahu možnosti sítě, a protože cestujícím jsou přiřazovány nejkratší možné trasy, pak čistá doba cesty cestujícího je vždy stejná a určuje optimální dobu jeho cesty systémem. Hrubá doba cesty pak udává skutečnou dobu.

Když jsme řekli, že čistá doba cesty cestujícího je vždy stejná, dalo by se říct, že potom je čistá doba cesty pro hodnocení nepotřebná, protože pro každou jednu cestu zůstává stejná. Skladba se ale cest v datech o cestujících může lišit, a proto

teprve podíl čisté a hrubé doby cesty je univerzální kritérium, které bude mít stejnou vypovídací hodnotu pro libovolnou skladbu požadavků.

1.1.7 Vývoj genetického algoritmu

Chování genetického algoritmu je závislé na aktuální konfiguraci i na topologii dopravní sítě, proti které bude algoritmus puštěn. V této sestavě není žádná součást, která by se dala považovat za pevně danou a referenční vůči ostatním. Cílem práce je posoudit, zda je genetický algoritmus vhodný nástroj pro vytváření jízdních řádů. Toto posouzení nesmí být komplikováno použitím špatné reprezentace jízdního řádu genomem, nevhodné ohodnocovací funkce či použitím irelevantních modelů. Z toho důvodu je nutné najít správnou sestavu této trojice.

Vhodné je začít u modelů: bez ohledu na konfiguraci je možné vymyslet triviální příklady, kde je možné intuitivně jednoznačně určit, co je správné řešení a po spuštění algoritmu rozhodnout, zda bylo nalezeno toto správné řešení nebo ne.

Pokud jde o genom, problém se může vyskytnout například v následujících aspektech:

- Degenerovaný obor hodnot: způsob reprezentace neumožní vytvoření některých jízdních řádů.
- Přerostlý definiční obor: příliš mnoho různých genomů bude převáděno na stejný jízdní řád.
- Křížení: křížením se nebudou předávat „dobré“ znaky z genomu.

Proto nejprve budeme pozorovat chování genomů na vytvořených modelech, a jako ohodnocovací funkci zvolíme triviální: průměrný poměr doby čisté a hrubé délky cesty cestujících.

Poté, co takto mezi genomy najdeme nejvhodnější, budeme experimentovat s ohodnocovacími funkcemi, kde budeme především sledovat:

- Jestli je funkce dostatečně diferencovaná,
- Jestli algoritmus konverguje k jednoznačnému výsledku,
- Jestli je ten výsledek správný.

Po této fázi, s nejlepším ze zkoušených genomů a nejlepší ze zkoušených ohodnocovacích funkcí, zkusíme vygenerovat náhodný velký model, příliš komplexní na to, aby bylo možné pro člověka rozhodnout, jak vypadá optimální rozdělení vozidel mezi linky. Na něm se pokusíme ukázat, že řešení je dostatečně obecné na to, aby našlo výsledek i na takto složitých modelech a provedeme diskusi, zda je nalezené řešení správné či blízké správnému.

1.1.8 Zjednodušení

Zde shrneme charakteristiky modelu, abychom na jednom místě vypsali všechna zjednodušení, která případně bude zapotřebí v budoucnu v projektu vyřešit:

- Dopravní síť je chápána jako víceúrovňový graf, kde doba cesty spoje mezi dvěma vrcholy na jeho trase je pevně daná hodnocením hrany mezi nimi, tedy nejsou zahrnuty vnější vlivy (jako jsou například dopravní zácpy, mimořádné události a semafory).
- Je podporován pouze jednoúrovňový graf (není vyřešené hledání cesty v grafu s více úrovněmi).
- Cestující mají předpočítanou trasu, tedy jejich chování odpovídá tomu, že si ve skutečném světě před cestou nechali najít spojení od vyhledávače.
- V pokusech se počítá s permanentně jezdícími spoji, což je myslitelné u roboticky ovládaných vozidel, ale ne u vozidel řízených lidmi.
- Množina jízdních řádů je zúžená na takové, kdy je den rozdělený na pásma v rámci nichž mají všechny linky konstantní interval.
- Není ošetřená skoková změna alokace vozidel, která by ve skutečnosti musela být řešena určitými přesuny vozidel dopravní sítí.

S těmito skutečnostmi musíme přijmout, že program poskytuje na výstupu jen přibližný výsledek, s nímž by bylo nutné dále pracovat, aby z něho byl realistický, použitelný jízdní řád.

1.2 Existující řešení

1.2.1 Teoretické práce

V knize Computer Scheduling of Public Transport [4] článek Bus Scheduling Program Development for A.T.A.F., Florence [6] popisuje aplikaci počítače na generování skutečných jízdních řádů, řeší ale problém minimalizace vozidel potřebných k dodržení požadovaných intervalů, což je jiná problematika než jakou pokrývá tato práce.

Další kniha, pokrývající dané téma, je Computer-Aided Scheduling of Public Transport [5]. V jejím úvodu je toto:

> „Crew scheduling is the last in the four-step transit planning process. These steps are network route design, setting frequencies and building timetables, vehicle scheduling and crew scheduling.“

Tato citace nám napovídá, že ani tato práce nepokrývá stejnou oblast jako naše, ale zároveň tu naši zasazuje do kontextu procesu tvorby jízdních řádů. Podle tohoto rozdělení plánovacího procesu naše práce pokrývá druhý ze čtyř kroků.

Extrémní případ principu plánování, který se naše práce snaží prosazovat, je takzvané DRT, Demand Responsive Transport neboli doprava přizpůsobující se požadavkům [7]:

> „A DRT service will be restricted to a defined operating zone, within which journeys must start and finish. Journeys may be completely free form, or accommodated onto skeleton routes and schedules,(...)

> varied as required. As such, users will be given a specified pick-up point and a time window for collection. (...) Some DRT systems may have defined termini, at one or both ends of a route, such as an urban centre, airport or transport interchange, for onward connections.“

Rozdíl DRT a našeho systému je v tom, že DRT reaguje na okamžitou potřebu, okamžitý požadavek jednotlivých cestujících. Náš systém se snaží najít plošně maximální shodu s pravidelnými požadavky všech cestujících, a ty spontánní zohledněné nejsou.

1.2.2 Praktické aplikace

Podle našeho průzkumu je rozšířeným systémem produkt společnosti Trapeze Group. Podle jejích podkladů (viz Příloha 21) se však jedná o řešení pro již naplánovanou síť, kde systém optimalizuje provozní náklady.

K plánování pražské hromadné dopravy používají společnosti DPP i ROPID software společnosti CHAPS. S tímto software jsme nebyli blíže seznámeni, víme, že poskytuje určité funkcionality týkající se samotného plánování jízdních řádů (například na základě naplánovaných odjezdů z konečných stanic a chronometráží dopravní sítě pro jednotlivé části dne umí dopočítat příjezdy spojů do stanic po cestě). Zda obsahuje i nějaké algoritmy pro stanovování časů odjezdů z konečné stanice, nevíme, z debaty s plánovači však vyplynulo, že software tímto způsobem nepoužívají.

2. Uživatelská dokumentace

V této kapitole je postupně vysvětlen postup práce s programem OptimeTable.

2.1 Instalace

Není třeba žádné instalace, stačí nakopírovat všechny soubory do jedné společné složky, nakonfigurovat (viz následující sekce) a spustit.

2.2 Konfigurace

Konfigurace samotného programu vyžaduje následující blok v konfiguračním souboru aplikace:

Ukázka kódu 2.1: Konfigurace adresy externích modulů

```
<applicationSettings>
  <OptimeTable.Properties.Settings>
    <setting name="TimetableGenerator"
    serializeAs="String">
      <value>
        OptimeTable.GeneticTimetableGenerator.dll
      </value>
    </setting>
    <setting name="PassengerGenerator"
    serializeAs="String">
      <value>
        OptimeTable.PassengerGenerator.dll
      </value>
    </setting>
    <setting name="DataSource"
    serializeAs="String">
      <value>
        OptimeTable.Data.Model.dll
      </value>
    </setting>
    <setting name="RequirementToItinerary"
    serializeAs="String">
      <value>
        OptimeTable.Data.Model.dll
      </value>
    </setting>
    <setting name="Benchmarker"
    serializeAs="String">
      <value>
        OptimeTable.Benchmarker.dll
      </value>
    </setting>
```

```
  </OptimeTable.Properties.Settings>
</applicationSettings>
```

Pro každý z externích modulů se zde definuje relativní adresa ke knihovně implementující daný modul. Speciálně ve výše uvedeném případě jsou všechny moduly ve stejné složce, jako EXE soubor programu.

Dále externí moduly mohou vyžadovat vlastní konfiguraci, která se v takovém případě přidá do stejného konfiguračního souboru, to je však záležitostí konkrétní implementace.

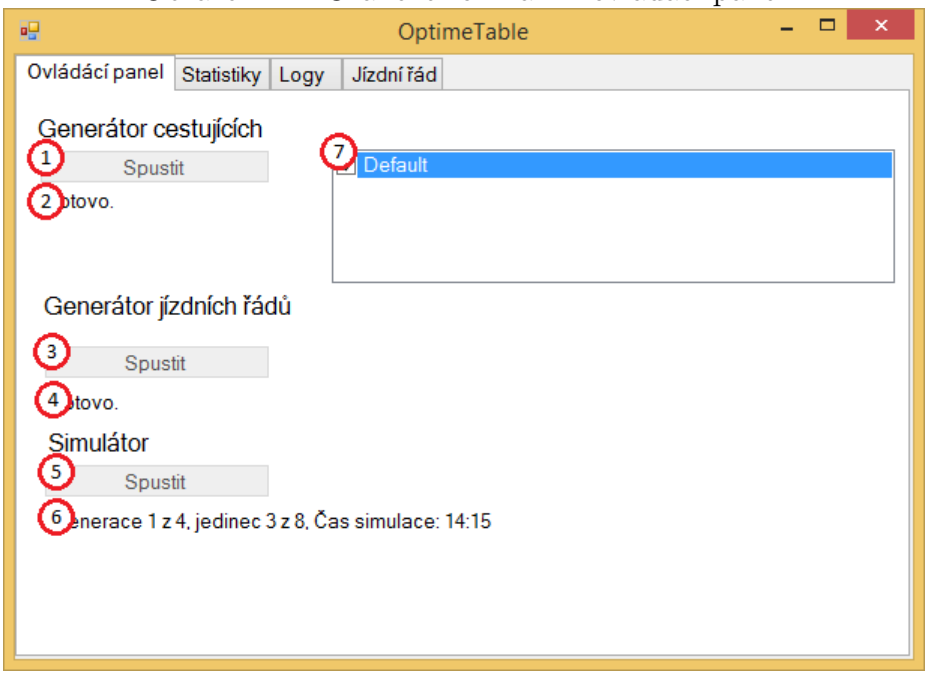
2.3 Uživatelské rozhraní

2.3.1 Ovládání

Program má jednoduché ovládání prostřednictvím grafického rozhraní.

Poznámka: všechny ovládací prvky jsou soustředěny v záložce ***Ovládací panel****, ostatní záložky patří do následující sekce Výstupy.*

Obrázek 2.1: Grafické rozhraní - ovládací panel



Vysvětlivky k popiskám:

1. Tlačítko spouštějící generátor cestujících
2. Stavová popiska generátoru cestujících
3. Tlačítko spouštějící generátor jízdních řádů
4. Stavová popiska generátoru jízdních řádů
5. Tlačítko spouštějící simulátor
6. Stavová popiska simulátoru
7. Výstup generátoru cestujících — seznam scénářů

Logické závislosti jsou: zdroj topologie (který v grafickém rozhraní nemá žádný výstup) poskytne topologii sítě generátoru cestujících, který vygeneruje scénáře pro generátor jízdních řádů, a ten poskytne funkci vracející jízdní řád simulátoru. Tomu by měla odpovídat korektní práce s programem, kdy by postupně měly být provedeny tyto kroky:

1. Spuštění generátoru cestujících
2. Výběr scénářů, s nimiž mají pracovat generátor jízdních řádů a simulátor
3. Spuštění generátoru jízdních řádů
4. Spuštění simulátoru

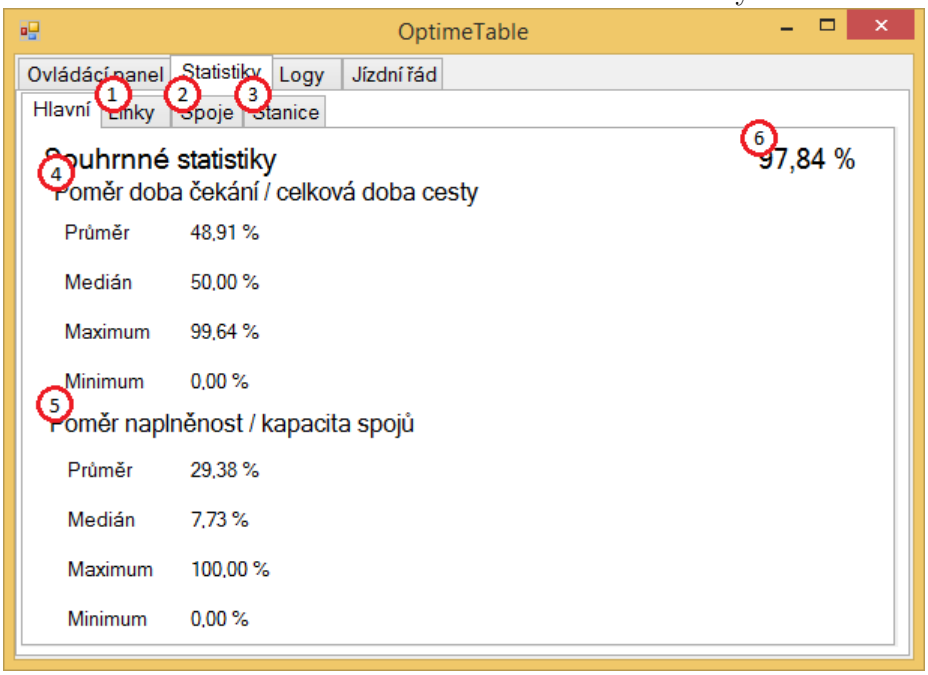
Navíc z kroků 3 a 4 je možné vrátit se na krok 2, změnit výběr scénářů a pokračovat dále.

Konec běhu každého spuštěného modulu je oznámen nastavením příslušné stavové popisky na text „Hotovo“.

2.3.2 Výstupy

Výstupům programu odpovídají zbylé tři záložky uživatelského rozhraní: **Statistiky**, **Logy** a **Jízdní řád**.

Obrázek 2.2: Grafické rozhraní - statistiky



1. Záložka **Linky** obsahuje tabulku se statistikou o linkách - pro každou linku se zobrazí:
   - Poměrná prázdnota - průměr, minimum, maximum. Poměrná prázdnota spoje je procentuální množství míst, která byla nevyužita. Průměr, minimum a maximum se vztahují k množině těchto hodnot získaných od všech spojů linky.

- Maximální agregovaná doba čekání. Agregovaná doba čekání se spočítá při příjezdu spoje do stanice jako průměrná doba, po kterou na spoj cestující, kteří do něj nastoupili, čekali. Maximum je tedy napříč všemi spoji linky.
- Maximální jednotlivá doba čekání. Maximální doba, po kterou cestující čekal na nějaký spoj linky.
- Průměrná jednotlivá doba čekání. Průměrná doba, po kterou cestující čekali na nějaký spoj linky.
- Počet odmítnutých cestujících. Suma počtů cestujících, které odmítly spoje linky (každý odmítnutý cestující se do této statistiky započítá tolikrát, kolik spojů jej odmítlo).

2. Záložka **Spoje** obsahuje tabulku se statistikou o spojích, analogickou k té v záložce **Linky**.

3. Záložka **Stanice** obsahuje tabulku se statistikou o stanicích:

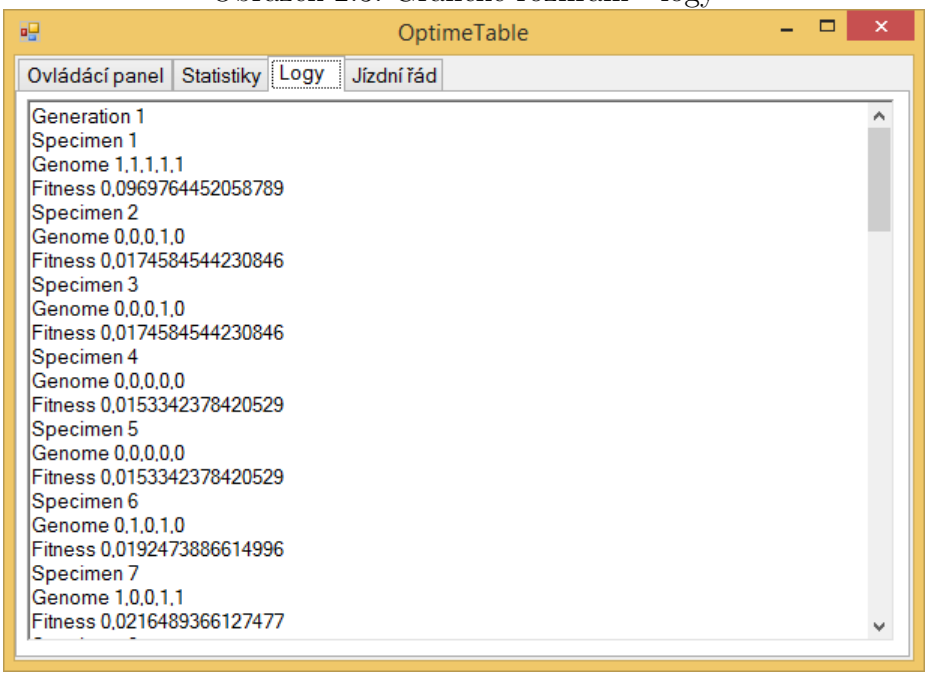
   - Doba čekání - průměr, medián, minimum, maximum. Statistika se vztahuje na množinu dob čekání všech cestujících, kteří čekali ve stanici.
   - Nejdelší čekání - stanice, průměr: každá stanice (když ji chápeme jako vrchol grafu) může mít víc sousedů, statistika Nejdelší čekání - stanice obsahuje z těchto sousedů toho, v jehož směru pasažéři nejdéle čekali. Průměrnou dobu čekání tímto směrem pak obsahuje sloupec průměr.
   - Počet obsloužených cestujících: počet cestujících, kteří ve stanici čekali na spoj.
   - Počet odmítnutých cestujících: kolikrát došlo k tomu, že cestujících nemohl nastoupit do svého spoje, protože byl plný. Jeden cestující se v této statistice může vyskytnout víckrát.

4. Statistika **Poměr doba čekání / celková doba cesty** se vztahuje k množině cestujících - pro tento poměr u cestujících zobrazuje maximum, minimum, medián a průměr.

5. Statistika **Poměr naplněnost / kapacita spojů** se vztahuje stejným způsobem k množině spojů.

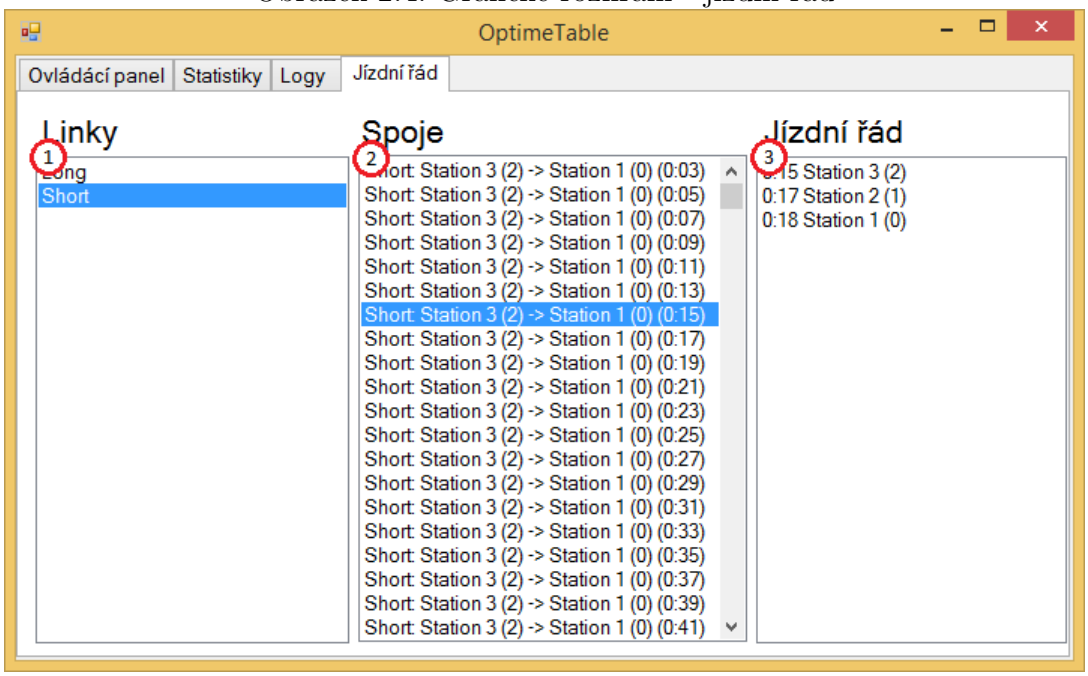
6. Velké číslo s procentem je hodnocení, které výslednému jízdnímu řádu dal hodnotič.

Obrázek 2.3: Grafické rozhraní - logy



Tato záložka je zcela věnována výstupu z externích modulů, tedy obsah této záložky zcela záleží na implementaci těch modulů.

Obrázek 2.4: Grafické rozhraní - jízdní řád



1. Seznam linek. Po nakliknutí jedné položky se aktualizuje seznam spojů (viz následující bod).

2. Seznam spojů. Jedná se o seznam spojů zvolené linky, a to v obou směrech. Po naklinutí spoje se aktualizuje jízdní řád (viz následující bod).

3. Jízdní řád. Jízdní řád zvoleného spoje.

3. Programátorská dokumentace

V této kapitole je nejprve popsána koncepce celého systému a jednotlivých modulů, aby před čtením detailního popisu jednotivých funkcionalit bylo možné seznámit se a pochopit obecné zásady, s nimiž byl kód psán. Dokumentace popisuje nejprve implementaci programu OptimeTable a následně i samostatných knihoven určených k úpravám jiných vývojářů. Implementace OptimeTable je popsána v sekcích Obecné a Moduly, implementace vývojářských balíčků jsou popsány v sekcích Obecné a Vývojářské balíčky.

3.1 Obecné

Program je psaný v jazyce C# nad technologií .NET. Tato volba je motivována velikou mírou podpory, která je vývojářům poskytnuta jak ze strany tvůrců C#, tak ze strany vývojářské komunity - pokročilé jazykové konstrukce, široký výběr knihoven od Microsoftu i od třetích stran. Tato podpora umožňuje pohodlně řešit například rozhraní na databázi nebo dotazování nad výčtovými typy bez nutnosti se těmito — z pohledu úkolů, které má program řešit — podružnostmi hlouběji zabývat a umožňuje soustředit se právě na implementaci samotného programu. Příkladem mohou být knihovny EntityFramework, Moq nebo LINQ.

3.1.1 Kultura kódu

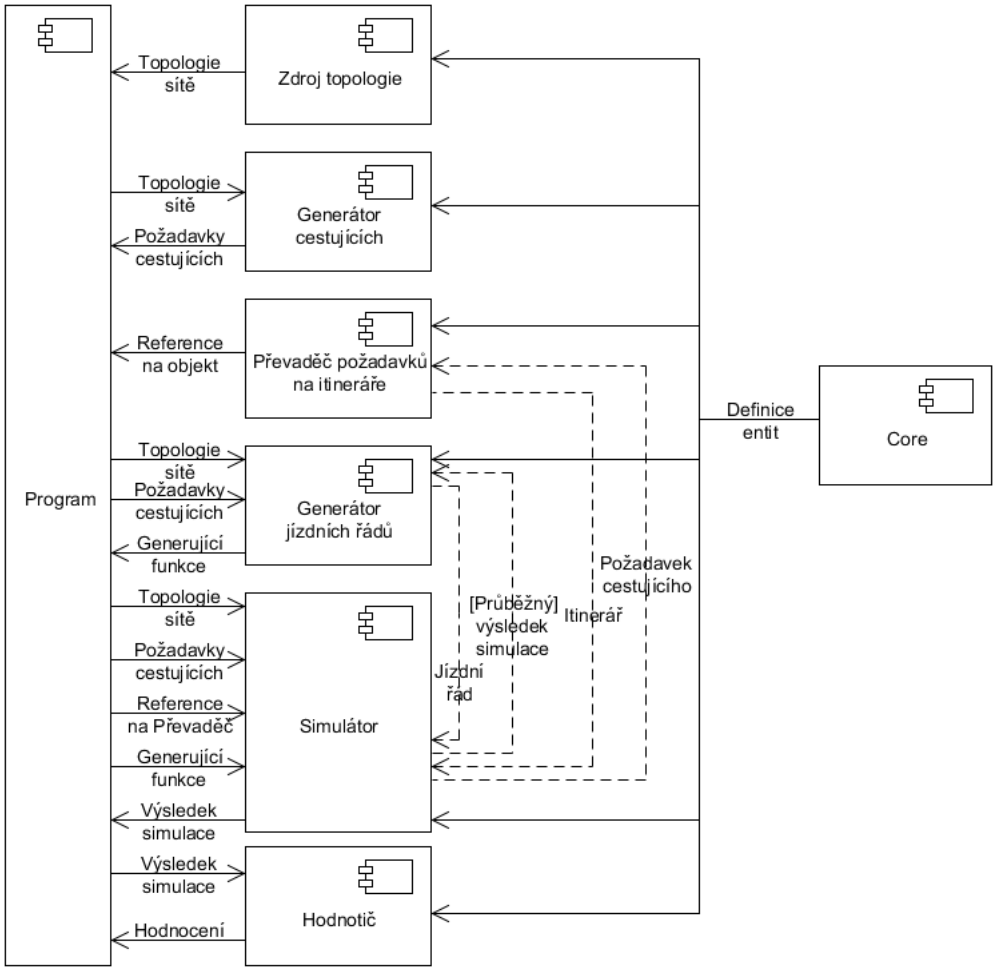
Zdrojový kód je výhradně v angličtině, mimo jiné i v zájmu jeho jazykové konzistence s nativními knihovnami .NET.

Ke kritickým pasážím kódu jsou napsané unit testy; ke každému projektu, pro který existuje alespoň jeden unit test, existuje stejnojmenný projekt s příponou .Test, sdružující všechny testy k projektu. Hierarchie složek tohoto testovacího projektu odpovídá hierarchii složek v původním projektu, tak, aby třída a test ke třídě měly v příslušném projektu stejnou relativní adresu.

3.1.2 Organizace kódu

Program pokrývá implementaci velice různých dílčích úkolů, které samy o sobě jsou netriviální a dávají prostor samostatnému zkoumání a vývoji (například modelování požadavků pasažérů a algoritmické generování jízdních řádů). Právě proto, že tyto úkoly jsou vzájemně oddělené — princip či algoritmus generování požadavků cestujících na dopravní systém nesouvisí s tím, jak se na základě množiny takových požadavků generuje jízdní řád — je program řešený jako soubor mnoha modulů, které je možné vyvíjet jednotlivě.

Obrázek 3.1: Schéma systému



Na výše zobrazeném schématu je výčet všech modulů systému s typy dat, které tvoří jejich vazby. Zároveň, ačkoliv nejde o flowchart, jsou moduly seřazeny v pořadí, v němž jsou typicky volány v průběhu programu.

Toto je úplný výčet modulů programu:

- Spustitelný EXE soubor s grafickým rozhraním
- Knihovna Core s definicemi společných entit
- Zdroj topologie dopravní sítě
- Převaděč požadavků pasažérů na itineráře
- Generátor požadavků pasažérů
- Generátor jízdních řádů
- Simulátor
- Hodnotič
- K některým z výše uvedených projektů navíc testovací projekt obsahující unit testy

Toto řešení architektury umožňuje vytvoření ukázkových implementací určených k tomu, aby vývojáři třetích stran mohli vytvořit modul a ten dodat coby knihovnu programu; o vývojářských balíčcích pojednáme v poslední sekci této kapitoly, zde popisujeme naši implementaci modulů.

Tímto se dostáváme k jedné dvojakosti koncepce programu: na jednu stranu je totiž vyvíjen jako zcela abstraktní systém, který má pouze definovat vztahy modulů, na druhou stranu má dodat jejich implementaci. Tato dvojakost je vyřešena tak, že existuje jedno základní Visual Studio Solution (dále VSS), které obsahuje kompletní sadu modulů včetně vztahů a včetně implementace modulů, a dále VSS ukázkových implementací, které jsou určeny pro samostatný vývoj jednoho modulu.

3.2 Pricipy systému

Před tím, než v následující sekci detailně popíšeme implementace jednotlivých modulů, představíme zde některé obecnější skutečnosti a zásady, z nichž se při implementaci vycházelo.

Je třeba zdůraznit, že opravdu klíčová část návrhu systému je vztah zdroje topologie, převaděče požadavků pasažérů na itineráře, generátoru požadavků pasažérů, generátoru jízdních řádů, simulátoru a hodnotiče. O to, aby tyto moduly spolu mohly „hovořit“ společným „jazykem“, se stará knihovna Core, která poskytuje definice entit, které jsou potom používány v různých modulech. Například třída reprezentující stanici tak může vzniknout ve zdroji topologie, a pak být následně použita v obou generátorech i v simulátoru. Další příklad jsou rozhraní, která musí mít implementována zmíněnými moduly, aby tyto mohly být korektně načteny. Ta jsou také definována v knihovně Core.

Program poskytující systému grafické rozhraní je pak jen způsob prezentace výše uvedeného systému.

V tomto bodě je třeba rozlišovat **program** OptimeTable, který (jak bylo řečeno výše) poskytuje grafické rozhraní na systém, jehož komponenty referencuje, a **systém** OptimeTable, čímž se myslí soustava modulů nebo jejich implementací. Systém OptimeTable vzniknul jako prostředí pro generování jízdních řádů a forma, kterou implementuje toto prostředí, je framework. Z toho důvodu je maximální důraz na nezávislost komponent, tedy na izolaci každé komponenty v samostatném souboru a její následnou referencovatelnost programem, který tyto komponenty propojí, jako je například program OptimeTable.

3.2.1 Vztah komponent

Bližšímu popisu vztahu modulů se věnuje Analýza (viz 1), proto jej zde popíšeme jen velmi stručně. Zdroj topologie dopravní sítě poskytuje — jak název napovídá — celou topologii, což zahrnuje výčet sítí, jednu pro každou z dopravních sítí (autobus, tramvaj...). Síť obsahuje:

- Graf, jehož vrcholy jsou stanice a hrany jsou (v případě tramvají či metra) koleje mezi nimi vedoucí.

- Definice linek, což jsou pojmenované dvojice tras; a trasa je posloupnost stanic v grafu.
- Počet vozidel.

Generátor cestujících přijímá jako vstup topologii sítě a poskytuje scénáře, což jsou pojmenované množiny požadavků cestujících. Požadavek cestujícího je definován jako trojice času příchodu na výchozí stanici, výchozí stanice a cílové stanice. Množina požadavků cestujících je považována za klíčový vstup, jemuž by měl co nejlépe vyhovovat výstupní jízdní řád.

Generátor jízdních řádů přijímá jako vstup topologii sítě a požadavky cestujících. Není nutné, aby spočítal jeden jízdní řád, návrh systému umožňuje i iterativní výpočet. Generátor vrací funkci, která vrací jízdní řád. Bližší popis chování řečené funkce je v následujícím odstavci, v popisu simulátoru.

Simulátor přijímá topologii sítě, požadavky cestujících, převaděč požadavků na itineráře a funkci, kterou vrací generátor jízdního řádu. Simulátor vrací výčet položek simulace, k nimž patří například záznam o přeplněném spoji, neobslouženém cestujícím nebo cestujícím dorazivším do svého cíle. Požadavky cestujících jsou definitivní, jízdních řádů může být více. Dokud funkce vrací jízdní řády, provádí se s každým z nich simulace. Když vrátí speciální definovanou hodnotu, tak je to považováno za pokyn „posledně vrácený jízdní řád je nejlepší“. V tomto okamžiku simulátor vrátí výčet položek simulace tohoto posledně vráceného jízdního řádu.

Simulace samotná probíhá tak, že se požadavky cestujících seřadí vzestupně podle času příchodu a položky jízdního řádu (jejichž jádrem je spoj, stanice a čas odjezdu) se seřadí podle času odjezdu.

Převaděč požadavků pasažérů na itineráře je komponenta, která je potřebná z důvodu návrhu simulátoru. V případě našich implementací je vždy implementována společně se zdrojem topologie, protože právě znalost topologie umožňuje zefektivnit tento výpočet (který je klíčový a typicky časově náročný) pomocí heuristik.

Na okraj poznamenejme, že moduly můžou být napsány jako soustava (například pražská topologie, požadavky pražských cestujících a realistické pražské jízdní řády), kdy konfigurace obsahující jen některé moduly ze soustavy (které doplní jinými) může vést k chybě, nebo obecně.

3.2.2 Principy programu OptimeTable

Program má jednoduché grafické rozhraní, vytvořené prostřednictvím editoru ve Visual Studiu 2013. Ostatní komponenty jsou načítány z DLL souborů pomocí funkcionalit knihovny `System.Reflection`, adresy k DLL souborům jsou v konfiguračním souboru programu.

Pro lepší pochopení funkcí programu je třeba se seznámit s architekturou systému, protože program především zprostředkovává spouštění komponent podle jejich logických závislostí. Přidanou hodnotu poskytuje tím, že zobrazuje výstupní statistiky pro jízdní řád i pro jednotlivé linky, spoje a stanice.

3.2.3 Principy knihovny Core

Knihovna slouží k definici entit používaných napříč všemi moduly, a také některých pomocných algoritmů (typicky používaných buď napříč všemi moduly nebo alespoň v různých implementacích stejného modulu).

Entity jsou organizovány podle modulů, v nichž typicky vznikají (například stanice typicky vzniká ve zdroji topologie, proto je třída `Station` ve jmenném prostoru `Core.Topology`). V případě, že má entita více property nebo proměnných, které jsou zapisovatelné, ale není je nutné inicializovat všechny, má entita konstruktor vynucující inicializaci nezbytného minima.

3.2.4 Principy zdroje topologie

Zdroj topologie je implementován v projektech `OptimeTable.Data.Prague` a `OptimeTable.Data.Model`.

První jmenovaný projekt implementuje pražskou topologii (včetně linek) k roku 2011, kterou načítá z databáze [1]. Zároveň pro Prahu implementuje převod požadavků cestujících na itineráře, který implementuje pomocí shluků (viz popis knihovny Library 3.3.3).

Rozhraní na databázi využívá EntityFramework. Entity načítané z databáze se konvertují na odpovídající entity knihovny Core.

Druhý jmenovaný projekt obsahuje různé pokusné topologie, který byly vytvořeny pro účely testování programu a zejména genetického algoritmu. Každá třída implementující topologii implementuje převod požadavků cestujících na itineráře pomocí shluků.

Pokusné topologie jsou průběžně popisovány v experimentální části (viz 4).

3.2.5 Principy generátoru požadavků cestujících

Generátor byl napsán jako minimální postačující implementace. V žádné studii využíván není, je využíván pouze programem OptimeTable, a očekává se, že nejčastěji bude využit v konfiguraci s pražskou topologií, která je pro prezentaci nejzajímavější (a také nejnázornější). Z těchto důvodů generuje náhodné dvojice stanic, mezi něž se ale nemohou dostat stanice, které běžnými tramvajovými linkami obsluhované nejsou.

Poznámka: linky mají své základní dvě trasy — jednu „tam“, druhou „zpět“ —, a s těmi se také v programu pracuje, ale ve výše zmíněných datech jsou i spoje, které například zatahují do vozovny nebo z ní naopak přijíždí na pravidelnou trasu. Tímto způsobem se do systému dostaly tramvajové zastávky, které neleží na trase žádné pravidelné linky.

3.2.6 Principy hodnotiče

Hodnotič provádí hodnocení na základě výstupu simulátoru. Výstupem simulátoru je v podstatě simulační kalendář, nebo jeho určitá podmnožina. Skládá se z položek o:

[1] viz Příloha 23

- Naplněnosti spoje (pro každý přesun spoje po hraně)
- Plném spoji (pouze pokud nastane)
- Době čekání cestujícího na spoj (při každém přestupu a také po příchodu na výchozí stanici cesty)
- Souhrnu cesty (po skončení cesty)
- Neobsloužení cestujícího z důvodu plného spoje (pouze pokud nastane)

Vychází se ze zásady, že o kvalitě obsloužení cestujícího nejlépe vypovídá poměr čisté a hrubé doby jeho cesty, a všechny funkce hodnotící jízdní řád nějakým způsobem pracují s touto hodnotou agregovanou přes všechny cestující.

3.2.7 Principy generátoru jízdních řádů

Generátor jízdních řádů je implementován v projektech `OptimeTable.TimetableGenerator` a `OptimeTable.GeneticTimetableGenerator`. Jízdní řád je ve všech implementacích poskytován pro celý jeden pracovní den (což však není nezbytné, jízdní řád může obsahovat položky pro rozmezí jedné hodiny nebo třeba dvou týdnů).

První jmenovaný na základě stejných dat, ze kterých čerpá pražská topologie, poskytuje jízdní řády k roku 2009 (neúplné, je to vlastnost výchozích dat).

Druhý jmenovaný implementuje genetický algoritmus nad jízdními řády. Simulátoru poskytuje funkci, která se stará o iterativní procházení generací (tedy že postupně vrací jízdní řády odpovídající aktuálně zkoumanému jedinci aktuální generace) a zpětnou vazbu zařizující přiřazení hodnocení jedincům.

Projekt s genetickým algoritmem obsahuje více hodnotících funkcí a více způsobů formátů reprezentace jízdního řádu genomem, obojí bylo použité v průběhu experimentální části práce.

3.2.8 Principy simulátoru

Simulátor je přímočarou implementací své výše uvedené role v systému. Jeho implementace je vytvořená napevno a jde tak o nedílnou část programu i systému OptimeTable.

Pokud se v kontextu simulátoru mluví o čase, je tím myšlena reprezentace času tak, jak je v simulátoru interně reprezentován, tedy jako nezáporné celé číslo vyjadřující čas v minutách. Typicky je v intervalu $[0, 1440]$, leda s minimálním přesahem do následujícího dne.

3.3 Moduly

V této sekci je pro každý modul napsána jeho koncepce a účel, dále jsou blíže rozepsány detailnější informace.

3.3.1 OptimeTable

Ústřední modul programu. Jako jediný se kompiluje jako spustitelný EXE soubor. Coby ústřední modul referencuje oba generátory a hodnotící modul. Obsahuje veškerý kód grafického rozhraní projektu.

Před inicializací grafického rozhraní se musí provést načtení modulů. Původní návrh počítal s tím, že moduly budou mít daný název souboru, název jmenného prostoru a budou implementovat jednu statickou metodu coby vstupní bod. Toto řešení se ukázalo jako funkční, ale nemoderní a v malé míře také omezující - implementátoři modulů by museli přesně dodržet tyto názvy, což není zásadní překážka, ale byla dána přednost komfortnějšímu řešení, kde taková překážka nebyla. Tímto řešením je konstrukt anglicky nazývaný dependency resolving, česky nejspíš vyhodnocování závislostí. Všechny externí moduly budou obsahovat třídu implementující předem dané rozhraní. Do konfiguračního souboru programu se napíše název souboru knihovny, a pomocí reflexe (funkcionality implementované v `System.Reflection`) se tento soubor načte a najde se v něm třída implementující příslušné rozhraní. Tento proces obstarává metoda `DependencyResolver.Resolve`.

Po vyhodnocení závislostí se inicializují delegátské metody obsluhující uživatelský vstup z grafického rozhraní.

Nakonec je inicializováno grafické rozhraní. V grafickém rozhraní jsou pro uživatelský vstup připravena tlačítka ovládacího panelu. Stisk každého z nich vyvolá zavolání vstupní metody nějakého z externích modulů. Generátor cestujících jako výstup vydá seznam scénářů, které se interně uloží do privátní proměnné scenarios. Dokud není generátor spuštěn znovu, zůstávají tam uložené. Generátoru jízdních řádů je jako vstup předávána množina scénářů zvolených uživatelem. Simulátoru je jako vstup předávána množina scénářů zvolených uživatelem a metoda, kterou vrátil generátor. Výstup každého z modulů je uchováván, a mění se až při opětovném spuštění modulu.

Výstup simulátoru — konečný jízdní řád a statistiky o něm — je po doběhnutí algoritmu zobrazen v příslušných záložkách grafického rozhraní (viz 2.3.1).

Dále jsou vylistovány jednotlivé moduly projektu s detailním popisem. Zatímco popis modulů je seřazen podle jejich abecedního pořadí, popis jejich obsahu vychází z logických závislostí.

3.3.2 OptimeTable.Benchmarker

Implementace hodnotiče. Jako vstup přijímá výstup simulátoru, na základě něhož spočítá hodnocení. Je implementován jako funkce počítající vážený průměr distribuční funkce poměru čisté a hrubé doby cesty u všech cestujících, navíc s elitismem používajícím elitu velikosti 4. Funkční hodnoty jsou pevně dané, jde o deset hodnot v intervalu $[0, 1]$ rozdělených parabolicky, se stejnými vahami.

3.3.3 OptimeTable.Core

Knihovna určená pro všechny ostatní moduly — knihovny i program. Obsahuje definici všech entit, které je potřeba používat napříč všemi moduly, a dává těmto entitám jednotnou, společnou definici. Jakákoliv entita, která se používá v alespoň dvou různých modulech, je definována zde. Organizaci entit do složek

není možné zcela jednoznačně definovat, obecně je to podle toho, ve kterém modulu vzniká, protože typicky se entity generují v jediném modulu (například v datovém zdroji se vytváří instance linek, stanic a dalších entit používaných k definici sítě) a v ostatních modulech jsou pouze čteny. Dále obsahuje i několik algoritmů, které s těmito entitami pracují, například vyhledávání cesty v grafu dopravní sítě. Některé jsou ve složce Library, jiné ve stejné složce, jako entita, se kterou pracují. Nakonec je zde i složka Interface, která pohromadě obsahuje všechna rozhraní definovaná v této knihovně.

U entit implementujících nějakou simulační logiku je navíc zásada, že všechny inicializační hodnoty musí být naplněny jako parametry konstruktoru. Takové entity často obsahují nějaké property, do nichž se nedá zapisovat, ale které Visual Studio přesto zobrazí mezi proměnnými k inicializaci.

Z detailního popisu jednotlivých entit modulu Core jsou zde nejdřív popsána rozhraní, protože jejich vysvětlení nezávisí na žádných ostatních entitách, naopak některé z entit jsou implementacemi nějakého zde specifikovaného rozhraní.

Interface

Obsahuje všechna rozhraní definovaná v této knihovně.

Z hlediska architektury programu jsou nejdůležitější rozhraní:

- `IBenchmarker`
- `IPassengerGenerator`
- `IRequirementToItinerary`
- `ITimetableGenerator`
- `ITopologyProvider`

Rozhraní přímočaře kopírují názvy modulů, pouze `IRequirementToItinerary` ne. Rozhraní `IRequirementToItinerary` je určené pro třídy, které převádí požadavky cestujících (viz níže) na itineráře (viz níže). Počítá se, že pro efektivní provedení tohoto úkolu je výhodná znalost topologie sítě, a tedy že třída, která implementuje rozhraní `ITopologyProvider`, bude implementovat i `IRequirementToItinerary`.

Stručný výklad ostatních rozhraní:

`IEdge` - generické rozhraní, reprezentuje hranu grafu. Využívá se v modulu OptimeTable.Core v grafových algoritmech.

`IIndexedEnumberable` - jde o `IEnumberable` rozšířené o operátor `this[]`.

`IPassengerContainer` - má být implementován jakoukoliv entitou, která má být využívána v simulátoru a pojímat cestující. Vytvoření tohoto rozhraní je motivováno rutinním způsobem přesouvání pasažérů ze stanic do spojů a obráceně, které bylo možné implementací toho rozhraní sjednotit.

`ITimer` - obsahuje pouze informaci o čase. Velice úzce souvisí s `IPassengerContainer`, protože metody implementující toto rozhraní se někdy musí vypořádat s tím, že cestující po jejich opuštění pokračuje po virtuální hraně. Cesta po virtuální hraně se uskuteční mimo jakýkoliv „kontejner“, a ve výsledku funguje stejně, jako by cestující skončil cestu a poté začal po určitém časovém intervalu novou v nějaké stanici. A právě to znovuzahájení cesty v nějaké stanici

vyžaduje informaci o čase a třídu, která spravuje cestující, kteří ještě do dopravní sítě nebyli přidáni.

Topology

Obsahuje entity potřebné k popsání topologie sítě. Z hlediska hierarchie odzdola nahoru to jsou:

- `Station` - stanice. Implementuje rozhraní `IPassengerContainer`, pro každou ze sousedních stanic si udržuje seznam cestujících, kteří přes ní chtějí jet.
- `Edge` a `VirtualEdge` - orientované hrany spojující dvě stanice. VirtualEdge dědí od Edge a nijak ji nerozšiřuje. Typ VirtualEdge pouze slouží k odlišení hran, které jsou virtuální, to znamená že se mezi nimi cestující přesouvají pěšky. V programu jsou virtuální hrany dávány pouze mezi různými stanicemi v rámci jedné superstanice, například v Praze je virtuální hrana mezi tramvajovou stanicí Malostranské náměstí ve směru Anděl a Malostranské náměstí ve směru Malostranská.
- `Graph` - reprezentuje jednu vrstvu dopravní sítě.
- `Network` - reprezentuje celý „svět“ simulace. Počítá se s tím, že bude později podporována vícevrstvá síť, tedy například že zároveň bude obsahovat metro, tramvaje i autobusy. Obsahuje seznam grafů, v simulátoru je vynuceno, aby byl jednoprvkový.

Kromě těchto základních entit jsou zde ještě definovány:
`Line` - linka, která je pojmenovanou dvojicí `LineRoute` (viz následující bod)
`LineRoute` - reprezentuje trasu linky, obsahuje posloupnost stanic s pomocnými metodami pro zjišťování vzdáleností mezi stanicemi.
`Link` - spoj. Spoj je jedna cesta soupravy dané linky z konečné na konečnou.

Passengers

Obsahuje entity související s cestami jednotlivých pasažérů simulací.

Základní z nich jsou:
`Scenario` coby výstupní typ `IPassengerGenerator.Generate`. `Scenario` je vlastně pojmenovaný výčet `PassengerRequirement`.
`PassengerRequirement` je uspořádaná trojice (odkud, kam, kdy), kde „odkud“ a „kam“ jsou stanice (instance třídy `Station`) a „kdy“ je čas.
`PassengerItinerary` je po jednotlivých stanicích rozepsaná cesta pasažéra, typicky vygenerovaná na základě `PassengerRequirement`. Kromě samotné cesty obsahuje `PassengerItinerary` také metody na udržování informace o bodu cesty, v němž se cestující aktuálně nachází.
`ItineraryItem` je položka itineráře, pouze obaluje instanci `Station`. Má dvě dědící třídy, `ItineraryStartItem`, která navíc určuje čas příchodu pasažéra na výchozí stanici, a `ItineraryVirtualItem`, která reprezentuje přesun pasažéra po virtuální hraně. Přesun po virtuální hraně je blíže rozepsán v podkapitole OptimeTable.Simulator 3.3.9.

`Passenger` je entita, se kterou pracuje simulátor. Vlastně jde o `PassengerItinerary` obalený metodami pro udržování statistik o cestě pasažéra, jako doba čekání na zastávce a doba cesty.

Zbývající třída `PassengersHandler` slouží k reprezentaci cestujících, kteří čekají na vstoupení do simulace. Opět bližší popis níže v podkapitole OptimeTable.Simulator 3.3.9.

Simulator

Obsahuje třídy, které se objevují na výstupu simulátoru. Hlavní z nich je `SimulatorResult`, která obaluje seznam `SimulationItem`. Tento seznam se dá chápat jako simulační kalendář obsahující všechny události, které mohou být relevantní pro funkce hodnotící jízdní řád, s nímž byl simulátor spuštěn.

`SimulationItem` je abstraktní třída, od níž všechny ostatní třídy dědí. Slouží pouze k tomu, aby bylo možné všechny položky simulačního kalendáře, které jsou od různých tříd, vložit do společné instance výčtového typu. Tato instance tedy může mít specifičtější generický parametr než `object`, a to právě `SimulationItem`.

`LinkFullItem` nese informaci o počtu cestujících, kteří se nemohli do spoje na dané zastávce dostat, protože byl spoj plný. Jde o uspořádanou trojici (spoj, stanice, počet cestujících).

`LinkStepItem` je záznam o vytíženosti spoje na jedné hraně dopravní sítě. Jde o uspořádanou čtveřici (spoj, stanice, počet volných míst, počet obsazených míst). Použití stanice implikuje, že se nepočítá s jedním spojem projíždějícím jednou stanicí dvakrát.

`PassengerJourneyItem` nese informaci o cestě pasažéra, konkrétně její výchozí a koncovou stanici, celkovou dobu trvání a dobu čekání. Jde o uspořádanou čtveřici (odkud, kam, doba trvání, doba čekání).

`PassengerWaitingItem` nese informaci o době čekání pasažéra v jedné stanici na jeden spoj. Jde o uspořádanou čtveřici (kde, jakým směrem, spoj, doba čekání).

Timetable

Základem jsou třídy: `Timetable` - reprezentuje jízdní řády všech linek v dopravní síti. Reprezentace funguje pomocí výčtu položek `TimetableItem` (viz níže).

`DepartureTimetable` - rozšiřuje `Timetable` o výčet odjezdů spojů z konečných. Počítá se totiž s tím, že velice typický způsob generování jízdního řádu bude pomocí vygenerování odjezdů z konečných, a není proto potřeba zatěžovat autory generátoru jízdního řádu dopočítáváním odjezdů z jednotlivých stanic trasy.

`TimetableItem` - položka jízdního řádu, uspořádaná pětice (spoj, stanice, následující stanice, čas příjezdu, čas odjezdu)

Dále je zde obsažena pomocná třída jsou `TimetableAutomaton`, která usnadňuje implementaci generátoru jízdních řádů tím, že udržuje aktuální stav generátoru vůči simulátoru (pro detaily viz sekci OptimeTable.Simulator 3.3.9). Možné stavy jsou ve výčtu `TimetableStates`.

Library

`Formatters` obsahuje metodu na formátování simulačního času na formát lépe čitelný pro člověka, tedy na hodiny a minuty.

`IEnumerableCopy` obsahuje metody pro kopírování `IEnumerable(T)` a `Queue(T)`. V obou případech jsou zkopírovány pouze odkazy na objekty, nejsou vytvářeny nové instance objektů. Metoda `IEnumerableCopy.Copy(T)` vrací instanci rozhraní `IEnumerable(T)`, která má nejmenší režii vkládání.

`PassengerTransferHandler` implementuje přesuny cestujících mezi dvěma instancemi `IPassengerContainer`. Obsahuje obecnou metodu `Transfer` a dále variantu konkretizovanou pro přesun ze stanice do spoje, protože při tomto přesunu se musí zohlednit omezená kapacita spoje.

`Graph` obsahuje kód pro dvě důležité obecné konstrukce:

- Reprezentace grafu pomocí sousedů
- Vytvoření kontrahovaného grafu

Reprezentace grafu pomocí seznamu sousedů [1] je efektivní pro rychlé procházení grafu do šířky, které potřebujeme ve vytváření kontrahovaného grafu. Implementace této reprezentace, spolu s algoritmem převodu grafu do této reprezentace, je v knihovnách `GraphNeighbourRepresentation(T)`
a `GraphNeighbourRepresentationWithTypedEdges(T)` (druhá jmenovaná je potomkem první). Z hlediska reprezentace jde o přímočarou implementaci, jak je popsána ve Skriptech (viz začátek tohoto odstavce), která je uložena v poli `Neighbours`. Navíc obsahuje pole `Ratings`, kde pro i-tou hranu z pole `Neighbours` obsahuje číslo vyjadřující hodnocení hrany.
`GraphNeighbourRepresentationWithTypedEdges(T)` navíc obsahuje pole `Types`, které stejným způsobem ukládá typy hran. Toto rozšíření (a vlastně zobecnění) základní třídy bylo přidáno proto, že v dopravní síti se využívají virtuální hrany.

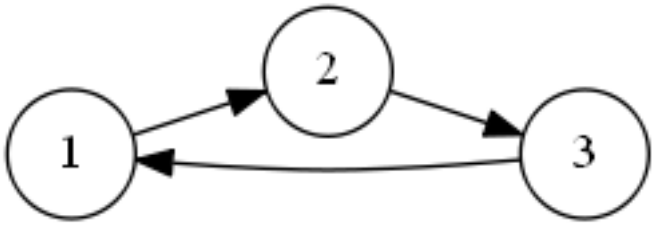
Třída `Cluster` umožňuje implementaci kontrahovaného grafu, reprezentuje shluk (viz 1.1.2). Má rekurzivní strukturu a pro jednotlivé úrovně hierarchie tyto potomky:

- `NetworkCluster` je kořen hierarchie.
- `Cluster` je vrchol kontrahovaného grafu superstanic.
- `SuperStationWrapper` odpovídá superstanici a obaluje podgraf stanic spojených virtuálními hranami.
- `StationWrapper` odpovídá stanici.

`ClusterHelper.Clusterize` vytváří kontrahovaný graf z obyčejného grafu. Přijímá jako parametr instanci `Graph` a callback `Action(Action(Station))`. Toto řešení je ušité na míru modelu pražské tramvajové sítě, ale příliš ničemu nevadí, při implementaci „normálních“ sítí postačuje předat jako hodnotu tohoto parametru lambda funkci s prázdným tělem a funkce `Clusterize` proběhne korektně. Callback je zapotřebí k ošetření možných anomálií ve struktuře sítí, kdy může vzniknout trojúhelník vypadající takto:

Callbacku je předán jako parametr jiný callback (naprogramovaný v rámci třídy `ClusterHelper`), který bere jako parametr výčet stanic, a z těchto stanic udělá jeden shluk. To umožňuje při vytváření kontrahovaného grafu přednostně označit skupiny stanic, které mají být sjednoceny do společného shluku, předtím, než je spuštěn obecný algoritmus.

Obrázek 3.2: Anomální situace na grafu superstanic



3.3.4 OptimeTable.Data.Model

Knihovna obsahující různé modely dopravní sítě. Tyto modely jsou používány v experimentální části práce.

Třída `Data` je implementace `ITopologyProvider` a `IRequirementToItinerary`, která se použije, pokud je modul OptimeTable.Data.Model použit jako zdroj topologie pro hlavní program.

Interface

Obsahuje pouze rozhraní `IModel`. Účel je takový, že ve složce Models (viz níže) bude více modelů, z nichž bude jeden podle kontextu (například hodnoty v konfiguračním souboru) načten a vrácen skrze třídu `Data` do programu coby zdroj topologie a převaděč požadavků na itineráře.

Models

Obsahuje různé topologie, každá z nich využívá pro převod požadavků na itineráře (implementaci `IRequirementToItinerary`) `Cluster`, respektive knihovnu `ClusterHelper`.

3.3.5 OptimeTable.Data.Prague

Knihovna obsahující model pražské tramvajové dopravní sítě. Používá reálná data, která jsou načítána z databáze.

3.3.6 OptimeTable.GeneticTimetableGenerator

Implementace generátoru jízdního řádu pomocí genetického algoritmu. Předpokládáme zde znalost základních principů genetických algoritmů; implementace v programu vychází ze zdroje [2] Obsahuje více reprezentací jízdního řádu pomocí genomu i více ohodnocovacích funkcí. Stejně jako je pomocí vyhodnocování závislostí v programu vyřešen problém s různými moduly dodanými pro program, v tomto modulu je více ohodnocovacích funkcí a více genomů. V experimentální části práce se používají všechny a jsou vzájemně porovnávány jejich výsledky.

Exceptions

Obsahuje výjimky vyvolávané v tomto modulu.

Entities

Obsahuje entity potřebné v celém modulu.

`Generation` - reprezentuje generaci. Kromě výčtu jedinců udržuje ukazatel na aktuálního jedince, aby bylo možné po jedincích generace iterovat, a metodu `AppointResult` pro uložení hodnocení k aktuálnímu jedinci. Jedinci jsou obaleni entitou `SpecimenWrapper`.

`SpecimenWrapper` - představuje uspořádanou dvojici (jedinec, hodnocení).

Interfaces

Obsahuje definice rozhraní, která umožňují realizovat používání různých ohodnocovacích funkcí a různých genomů.

`IFitnessFunction` - Rozhraní pro instance ohodnocovacích funkcí.

`IMultiGenomeSpecimen` - Obecné rozhraní pro reprezentaci jízdního řádu genomem; reprezentace pomocí tohoto rozhraní může obsahovat více různých genomů.

`ISingleGenomeSpecimen` - Specializace předchozího rozhraní pro jediný genom.

FitnessFunctions

Obsahuje implementace ohodnocovacích funkcí. Společnou vlastností těchto implementací je, že to jsou třídy implementující rozhraní `IFitnessFunction`. Funkce mají průběh výpočtu vysvětlený v dokumentačním komentáři, jejich výstupy lze najít v experimentální části práce.

Specimens

Obsahuje různé reprezentace jízdních řádů pomocí genomu. Každá reprezentace je třída implementující rozhraní `IMultiGenomeSpecimen`.

`CountGeneSpecimen` obsahuje jediný genom. Cifra n na i-té pozici genomu znamená: „na linku i je alokováno n vozidel“. Protože křížením by mohlo snadno dojít k přesažení počtu vozidel, která jsou v síti k dispozici (`Graph.AvailableVehicles`), po každé operaci měnící genom (mutace, křížení, inicializace) se provádí harmonizace, zavolání metody `Normalize`. Ta je implementována přímočaře: v případě, že suma genů S je větší než `Graph.AvailableVehicles`, pro každý gen G: $G \leftarrow \lfloor G * \frac{S}{Graph.AvailableVehicles} \rfloor$.

Převod instance `CountGeneSpecimen` na jízdní řád (tělo metody `ToTimetable`) se provádí za pomoci třídy `DepartureTimetable` z knihovny OptimeTable.Core.Library (viz výše). Pro každou linku se vezmou vozidla, která jsou pro ni alokována, a ta se nechají v pravidelných intervalech jezdit. Konkrétně, ať má linka L alokovaných n vozidel a cesta z jedné konečné na druhou a zpět trvá t. Pak interval spojů bude $\lceil {}^{t}/_{n} \rceil$.

`LineGeneSpecimen` obsahuje jediný genom. Cifra n na i-té pozici genomu znamená: „i-té vozidlo je alokováno na linku n“. Zjevná výhoda této reprezentace je, že nemusíme genom normalizovat, a vždy jsou využita právě všechna vozidla.

Převod instance `LineGeneSpecimen` na `Timetable` probíhá stejně, jako u `CountGeneSpecimen`.

`LineGeneSpecimenWithSineTwoGenomes` obsahuje dva genomy. První z nich funguje stejným způsobem, jako `LineGeneSpecimen`. Druhý z nich zajišťuje změnu intervalu linky v čase. Jak název třídy napovídá, využita je funkce sinus. Než budeme pokračovat s bližším popisem, potřebujeme dvě tvrzení s důkazy:

Tvrzení. $\sum_{i=0}^{k-1} cos(\frac{2\pi k}{n}) = 0 = \sum_{i=0}^{k-1} sin(\frac{2\pi k}{n})$

Důkaz je bezprostředním důsledkem věty o součtu mocnin primitivní mocniny, který se nachází ve skriptech doc. Holuba [3]

Tvrzení. *Nechť* $k \in \mathbb{N}, r \in \mathbb{R}$. *Pak* $\sum_{i=0}^{k-1} sin(\frac{2\pi k}{n} + r) = 0$

Důkaz. Pro začátek připomeňme důležitý goniometrický vzorec:
$sin(\alpha + \beta) = sin(\alpha)cos(\beta) + cos(\alpha)sin(\beta)$
Po dosazení hodnot z tvrzení tedy:
$\sum_{i=0}^{n-1} sin(\frac{2\pi k}{n} + r) = \sum_{i=0}^{n-1}(sin(\frac{2\pi k}{n})cos(r) + cos(\frac{2\pi k}{n})sin(r)) =$
$= \sum_{i=0}^{n-1} sin(\frac{2\pi k}{n})cos(r) + \sum_{i=0}^{n-1} cos(\frac{2\pi k}{n})sin(r) =$
$= cos(r)\sum_{i=0}^{n-1} sin(\frac{2\pi k}{n}) + sin(r)\sum_{i=0}^{n-1} cos(\frac{2\pi k}{n}) =$
$= cos(r) \times 0 + sin(r) \times 0 = 0$

□

V interpretaci sinoidního genomu se využije skutečnosti, že funkční hodnoty v bodech rovnoměrně rozložených podél periody sinu sečtou na nulu. Stav vozidel na každé z linek je tedy $V + \lfloor A \times sin(\frac{2\pi}{n} \times O + \frac{t\pi}{720}) \rfloor$, kde V je výchozí stav vozidel podle primárního genomu, A je amplituda odchylky udávané hodnotou sinu, n je počet linek s amplitudou A a $0 \leq O < n$ je pořadí dané linky mezi těmito linkami.

Helpers

Obsahuje pomocné třídy pro vykonávání jednotlivých fází genetického algoritmu: inicializace, vytvoření množiny křížených jedinců, křížení a mutaci.

`InitializationHelper` obsahuje algoritmy pro inicializaci jedinců. Jedinci mohou být různého typu, což je vyřešeno tak, že má třída pro každého podporovaného jedince jednu privátní metodu. Zvenčí se lze dotazovat na statickou generickou metodu `GetNext(T)`.

`MatingPoolHelper` obsahuje algoritmus vytvoření množiny jedinců, kteří budou mezi sebou kříženi. Vstupní metoda je `CreateMatingPool` s parametrem typu `Generation`, popis entity `Generation` následuje o něco níže. Metoda vydá jako výstup stejný počet jedinců, jaký má generace; nechť suma hodnocení všech jedinců je S, pak výběr jedince probíhá takto:

`CrossoverHelper` obsahuje algoritmus křížení jedinců. Jako vstup přijímá očekávaný výstup `MatingPoolHelper.CreateMatingPool`, tedy výčet typu `IMultiGenomeSpecimen`. Průběh vypadá takto: Jak je v pseudokódu vidět, implementace křížení coby operace s genomem je záležitost implementace jedince, pomocná třída pouze vybírá dvojice jedinců. Jedním z důvodů tohoto řešení je snaha zachovat genom privátní proměnnou třídy jedince.

`MutationHelper` obsahuje algoritmus mutace jedince. Implementace je naprosto přímočará – na každém jedinci se pro každý gen každého genomu s pravděpodobností uloženou v property `MutationProbability` zavolá metoda `Mutation`.

Algoritmus 3 Výběr jedince pro fázi křížení

$R \leftarrow Random(S)$
$Sum \leftarrow 0$
while $Sum + CurrentSpecimen.Fitness < R$ **do**
    $Sum \leftarrow Sum + CurrentSpecimen.Fitness$
    $CurrentSpecimen \leftarrow Generation.Next()$
end while
Output.Add($CurrentSpecimen$)

Algoritmus 4 Výběr dvojic jedinců pro skřížení

while Nebyli všichni jedinci skříženi **do**
    $Jedinec1, Jedinec2$
    $Dvojice \leftarrow Jedinec1.Crossover(Jedinec2, PoziceProKazdyGenom)$
    Výstup.Add(Dvojice.První)
    Výstup.Add(Dvojice.Druhý)
end while

3.3.7 OptimeTable.GeneticTimetableGenerator.Test

Unit test project. Testuje funkčnost algoritmů a obsahuje pokusy s jednotlivými konfiguracemi genetického generátoru jízdních řádů.

Pro experimenty jsou zásadní:

Rozhraní `IExperiment` - předepisuje implementaci všech rozhraní, která jsou v programu implementována moduly, tj. `ITopologyProvider`, `IRequirementToItinerary`, `IPassengerGenerator`. Napevno se počítá, že jako implementace `ITimetableGenerator` se využije genetický algoritmus, protože rozhraní obsahuje ještě `SpecimenType` a `IFitnessFunction`.

Třída `ExperimentExecutor` provádí pokusy, jako parametr bere instanci `IExperiment` a název výstupního souboru. Její základní akce je spuštění simulátoru, jako přidanou hodnotu provádí průběžné výpisy do souboru typu CSV tak, že je pak přímo možné vytvořit v některém z tabulkových procesorů graf vývoje počtu alokovaných vozidel pro dané linky. Nakonec testuje, jestli v simulátoru neselhala validace jízdního řádu (příkaz `Assert.IsNull` na konci souboru, viz OptimeTable.Simulator 3.3.9 níže).

Třída `Study` je abstraktní a obsahuje implementace všech pokusů, které byly vytvořeny. Má jen tři abstraktní property - `FitnessFunction`, `SpecimenType` a `StudyName`. S nimi pracují jednotlivé pokusy, kdy `FitnessFunction` a `SpecimenType` se využijí pro vytvoření konfigurace instance genetického generátoru jízních řádů a `StudyName` se použije jako název složky, do níž se ukládají výsledky.

Ve složce Study se dále nachází třída `Experiment`, její účel je pouze přímočará a holá implementace `IExperiment`.

Dále se ve složce Study nachází přímo studie. Každá musí přetěžovat abstraktní tři property, a pro korektní běh také pro každý pokus ve třídě `Study` mít testovou metodu, která pouze volá metodu s tím pokusem. Příklad:

Ukázka kódu 3.1: Study.cs

```
public void TwoLineGraphPassengersOnShortLine()
```

```
{
  var passengerGeneratorMock = (...);
  passengerGeneratorMock
    .Setup(...)
    .Returns(...);

  var model = new OneRailModelTwoLines(2);
  ExperimentExecutor.Experiment(new Experiment
  {
    DataSource = model,
    PassengerGenerator
      = passengerGeneratorMock.Object,
    RequirementToItinerary = model,
    FitnessFunction = FitnessFunction,
    SpecimenType = SpecimenType
  },
  string.Format(
    "{0}/TwoLineGraphPsngrsOnShortLine.csv",
    StudyName));
}
```

Ukázka kódu 3.2: LineGeneSpecimenAverageWaitingRatioFitness.cs

```
[TestClass]
public class LineGeneSpecimenAvgWaitingRatioFitness
          : Study
{
    private AverageWaitingRatioFitness fitnessFunc
    = new AverageWaitingRatioFitness();

    public override IFitnessFunction FitnessFunction
    {
        get { return fitnessFunction; }
     }

    public override Type SpecimenType
    {
        get { return typeof(LineGeneSpecimen); }
    }

    public override string StudyName
    {
        get { return "JmenoTetoStudie"; }
    }

(...)

    [TestMethod]
    public new void TwoLineGraphPassengersOnShortLine()
```

```
    {
        base.TwoLineGraphPassengersOnShortLine();
    }
```

V ukázce kódu 3.2 je navíc názorná ukázka přetížení abstraktních property třídy `Study`.
Studie mají podobu unit testů, protože je potom možné zvlášť spustit jeden pokus v jedné studii, i dávkově všechny pokusy jedné studie a i více studií.

3.3.8 OptimeTable.PassengerGenerator

V projektu `OptimeTable.PassengerGenerator` je pilotní implementace generátoru cestujících. Generuje náhodná data s konstantní velikostí a distribucí v čase.

3.3.9 OptimeTable.Simulator

V projektu `OptimeTable.Simulator` je implementace simulátoru. Jeho vstupem je funkce vracející jízdní řád, požadavky cestujících a popis dopravní sítě. Funkce místo instance jízdního řádu na vstupu umožňuje, aby se generátor jízdního řádu choval dynamicky a na základě výstupu simulátoru jízdní řád zlepšoval, až předá simulátoru poslední, výsledný. V této souvislosti platí konvence, že simulátor i generátor si pamatují posledně předaný jízdní řád. Konec procesu tvorby jízdního řádu je potom naznačen předáním k tomu vyhrazené konstanty z knihovny OptimeTable.Core (viz níže).

Simulátor má jako vstupní bod metodu `Run`. Metoda Run má několik parametrů:

- `ITopologyProvider topologyProvider` - objekt, který poskytuje data o topologii sítě. Měl by to být ten samý objekt, který je poskytnut do generátoru cestujících i jízdních řádů.
- `IRequirementToItinerary requirementToItinerary` - objekt, který umožňuje v kontextu topologie sítě převod požadavku cestujícího na itinerář.
- `IEnumerable(Scenario) scenarios` - seznam scénářů, s nimiž byl spuštěn generátor jízdního řádu, skrze obal třídy `Scenario` seznam požadavků cestujících.
- `Func(..., Timetable) timetableProvider` - viz níže.

Parametr `timetableProvider` je funkce, kterou vrací generátor jízdních řádů. Toto řešení je vhodné, protože pokud by generátor jízdních řádů vracel pouze jeden konečný výsledek - jízdní řád, neumožňovalo by to na straně generátoru implementovat nějakou iterativní metodu, která by využívala výsledky simulátoru. Implementovat simulátor v modulu s generátorem by zase bylo velice neefektivní v situaci, kdy jeden simulátor již implementovaný je. `timetableProvider` je tedy prostředek komunikace mezi simulátorem a generátorem. V rámci této komunikace poskytne simulátor generátoru při každé iteraci tři parametry:

- Výsledek simulace podle posledního jízdního řádu

- Callback pro výpis informace o aktuálním stavu generátoru, očekává se, že callback bude měnit obsah nějakého prvku v grafickém rozhraní
- Callback pro zápis do logu, tedy určen na technické informace, průběžné hodnoty ve výpočtech apod.

Jako první krok se převedou scénáře na výčet itinerářů. Tyto itineráře se budou používat v každé iteraci simulace, tzn. s dalšími jízdními řády se už itineráře cestujících nepřepočítávají.

V druhém kroku se pustí smyčka získávání jízdních řádů z generátoru jízdních řádů. Tato smyčka končí tehdy, když generátor vrátí `null`. Navíc je omezena na počet iterací udaných v konstantě `ITERATIONS_LIMIT`. Poslední jízdní řád vrácený před tím, než generátor vrátil `null`, je považován za „poslední slovo“ generátoru.

Algoritmus 5 Průběh jedné iterace smyčky

Seřazení položek jízdního řádu podle času vzestupně
Vyprázdnění informací o cestujících ve všech spojích
Vyprázdnění informací o cestujících ve všech stanicích
if Jízdní řád je validní **then**
    Rozjeď Simulaci
else
    Vrať null
end if

Algoritmus 6 Validace jízdního řádu

$PocetVozidel \leftarrow 0$
for all položka jízdního řádu= $(Spoj, Stanice, DalsiStanice, Prijezd, Odjezd)$ **do**
    if $Stanice$ není konečná **then**
        Zkontroluj, že hrana $(Stanice, DalsiStanice)$ existuje
    end if
    if Spoj právě vyjel **then**
        $PocetVozidel \leftarrow PocetVozidel + 1$
    end if
    if Spoj právě skončil **then**
        $PocetVozidel \leftarrow PocetVozidel - 1$
    end if
end for
Zkontroluj, že $PocetVozidel \leq Graph.AvailableVehicles$

Kromě toho je v simulaci ještě instance třídy `SystemEnthropy`, která funguje jen pro zkušební provoz a obsahuje údaj o počtu cestujících, aby se ze simulace „neztráceli“.

Algoritmus 7 Průběh simulace

```
Uveď všechny itineráře do výchozího stavu
for all položka jízdního řádu do
    (Spoj, Stanice, DalsiStanice, CasPrijezdu, CasOdjezdu) ← Položka jízdního řádu
    for all itinerář cestujících, který začíná nejpozději v CasOdjezdu do
        Uveď cestujícího na výchozí stanici
        Nastav cestujícímu čas a místo začátku cesty
    end for
    Log.Add(StatistikaOVytíženíSpoje)
    Spoj.VylozitCestujici(Stanice)
    Spoj.NalozitCestujici(Stanice)
    Log.Add(StatistikaODobeCekaniCestujicich)
    if CestujiciSeNevejdou(Stanice, Spoj) then
        Log.Add(StatistikaOPoctuPrespocetnychCestujicich)
    end if
end for
```

3.3.10 OptimeTable.Simulator.Test

`OptimeTable.Simulator.Test` je unit test project. Obsahuje testy funkcionalit simulátoru.

3.3.11 OptimeTable.TimetableGenerator

Projekt `OptimeTable.TimetableGenerator` je implementace generátoru jízdního řádu. Na výstup předává reálný (bohužel neúplný) jízdní řád pražských tramvají z roku 2011.

Celý jízdní řád se načítá z databáze a převádí na entity z OptimeTable.Core.

3.4 Ukázkové implementace

Ke každému balíčku je uvedeno, jaké moduly s ním souvisí (což znamená, jaké moduly ho budou používat), jaké je jeho implementační minimum a na závěr jsou uvedeny tipy pro implementaci. Tipy by měly nabízet pomůcky poskytované modulem OptimeTable.Core, které souvisí s daným balíčkem a které by případně programátor nemusel snadno objevit.

3.4.1 Generátor jízdních řádů

Související moduly

Generátor jízdních řádů dostává jako parametry topologii dopravní sítě a výstup generátoru cestujících a jeho výstup je předáván do simulátoru. K dispozici má entity definované v knihovně OptimeTable.Core.

Výstup generátoru cestujících je v podobě výčtu scénářů, tedy instancí třídy `OptimeTable.Core.Passengers.Scenario`.

Topologie dopravní sítě je zapouzdřená v entitě typu `OptimeTable.Core.Entities.Network`. Ta reprezentuje celou dopravní síť, která se skládá ze samostatných grafů, např. graf tramvajové sítě a graf sítě autobusů. Graf je tvořen stanicemi a hranami, a navíc ještě linkami.

Simulátor dostává od generátoru jízdních řádů funkci, která vrací jízdní řády, a tuto funkci volá, dokud nevrátí hodnotu `null`. Každý vrácený jízdní řád validuje, a v případě, že je validní (odpovídá možnostem dopravní sítě, např. limitům vozidel), provede simulaci dopravního provozu jezdícího podle naposledy vráceného jízdního řádu.

Nutné minimum modulu

Modul musí obsahovat alespoň jednu třídu, která impementuje rozhraní `OptimeTable.Core.Interface.ITimetableGenerator`. V případě, že jich obsahuje víc, je vybrána právě jedna z nich, a není specifikováno, podle jakého klíče. Knihovna, která vznikla kompilací zdrojového kódu obsahující tuto třídu, je technicky postačující pro použití s programem.

Jak je již zmíněno výše, funkce, kterou vrací `ITimetableGenerator.GetTimetable`, vrací instance typu `Timetable`. Běh simulace končí tím, že tato funkce vrátí `null`. Tedy korektní implementace vrací `null` v „rozumném“ čase a po „rozumném“ počtu iterací. Obě veličiny - doba výpočtu a počet iterací - jsou v programu omezeny.

Tipy

- Implementovat v modulu rozhraní `ITimetableGenerator` právě jednou.
- Napsat program tak, aby běžel co nejkratší dobu (řádově nejvýše minuty).
- Využít rozšíření `DepartureTimetable`. Základní třída `Timetable` poskytuje pouze proměnné `Items` a `Links`, tedy pro každý spoj je nutné rozpočítat příjezdy a odjezdy do všech zastávek podél jeho trasy. `DepartureTimetable` navíc přidává výčet `Departures`. Smysl je, že spoj může být určený časem svého odjezdu z konečné, a jeho příjezdy a odjezdy ze zastávek jsou určeny topologií dopravní sítě a případně dopravní situací a zdržením způsobeným výstupem a nástupem cestujících. Proto se tyto odjezdy vloží do `Departures`, a poté, co už tam jsou všechny, se zavolá metoda `Complete`. Ta dopočítá odjezdy a příjezdy do dalších stanic.
- Nahradit nepřítomnost programu, který bude modul využívat, unit testy.
- Využít pro vytvoření instancí rozhraní potřebných na vstup modulu knihovny Moq.
- Využít třídy `TimetableAutomaton`, která umožňuje sledovat význam hodnoty, kterou funkce aktuálně vrací.

3.4.2 Generátor požadavků cestujících

Související moduly

Generátor požadavků cestujících dostává jako parametr topologii dopravní sítě a jeho výstup je předáván do generátoru jízdních řádu a posléze do simulátoru. K dispozici má entity definované v knihovně OptimeTable.Core.

Topologie dopravní sítě je zapouzdřená v entitě typu `OptimeTable.Core.Entities.Network`. Ta reprezentuje celou dopravní síť, která se skládá ze samostatných grafů, např. graf tramvajové sítě a graf sítě autobusů. Graf je tvořen stanicemi a hranami, a navíc ještě linkami.

Nutné minimum modulu

Modul musí obsahovat alespoň jednu třídu, která impementuje rozhraní `OptimeTable.Core.Interface.IPassengerGenerator`. V případě, že jich obsahuje víc, je vybrána právě jedna z nich, a není specifikováno, podle jakého klíče. Knihovna, která vznikla kompilací zdrojového kódu obsahujícího tuto třídu, je technicky postačující pro použití s programem.

Tipy

- Využít možnost rozdělit požadavky cestujících do scénářů. Toto dělení nemá význam pro simulátor, ten požadavky vždy sloučí a nakládá s nimi stejně, ale pro uživatele programu.
- Nahradit nepřítomnost programu, který bude modul využívat, unit testy.
- Využít pro vytvoření instancí rozhraní potřebných na vstup modulu knihovny Moq.

3.4.3 Hodnotič

Související moduly

Hodnotič dostává jako vstup výsledek simulace. Volání hodnotiče je poslední fáze práce programu, poté, co doběhne generátor jízdních řádů.

Nutné minimum modulu

Modul musí obsahovat alespoň jednu třídu, která impementuje rozhraní `OptimeTable.Core.Interface.IBenchmarker`. V případě, že jich obsahuje víc, je vybrána právě jedna z nich, a není specifikováno, podle jakého klíče. Knihovna, která vznikla kompilací zdrojového kódu obsahující tuto třídu, je technicky postačující pro použití s programem.

Tipy

- Výsledek simulace je výčet typu `SimulationItem`. To je abstraktní třída, která má pouze čtyři instancializovatelné potomky, všechny jsou ve jmenném prostoru `OptimeTable.Core.Simulator`. Konkrétně jde o

- `LinkFullItem` - reprezentuje počet cestujících, kteří se v dané stanici nevešli do daného spoje. Pokud k takové situaci nedojde, nepřidává se `LinkFullItem` s nulovým počtem lidí.
- `LinkStepItem` - reprezentuje statistiku o vytížení spoje při jízdě podél jedné hrany grafu.
- `PassengerJourneyItem` - reprezentuje statistiku o celkové cestě jednoho cestujícího (tedy výchozí a cílovou stanici, dobu čekání a cesty).
- `PassengerWaitingItem` - reprezentuje dobu čekání cestujícího v dané zastávce.

Při zpracovávání výsledku je tedy šikovné třídit výčet podle skutečného typu položky (ideálně operátorem `typeof`).

3.4.4 Zdroj topologie

Související moduly

Zdroj topologie nedostává žádný vstup, ale naopak je sám používán jako parametr pro prakticky všechny ostatní moduly: kromě programu ho potřebují i generátor jízdních řádů a generátor cestujících.

Nutné minimum modulu

Modul musí obsahovat alespoň jednu třídu, která impementuje rozhraní `OptimeTable.Core.Interface.ITopologyProvider`. V případě, že jich obsahuje víc, je vybrána právě jedna z nich, a není specifikováno, podle jakého klíče. Knihovna, která vznikla kompilací zdrojového kódu obsahující tuto třídu, je technicky postačující pro použití s programem.

Navíc instance třídy `OptimeTable.Core.Entities.Network`, kterou musí modul z metody `GetNetwork` vracet, musí obsahovat **právě** jeden graf ve výčtu `Nets`.

Počítá se s tím, že výše uvedené pravidlo bude v novějších verzích programu zrušeno.

Tipy

- Typicky bývá zdroj topologie spojen s převaděčem požadavků na itineráře (viz následující sekce), a to i z následujících důvodů:
  - Znalost topologie, kterou modul poskytuje, umožňuje v některých případech efektivnější řešení hledání cest v grafu, než obyčejné variace na prohledávání do šířky a do hloubky.
  - Možnost využít knihovny `OptimeTable.Core.Graph`, konkrétně třídy `ClusterHelper` (viz 3.3.3).
- Jako výstup, a také jako kontrola správnosti vytvořené topologie, se dá využít metoda `GraphHelper.PrintToGVFile`, která vytvoří soubor pro program Graphviz, v němž je možné potom graf vizualizovat.

3.4.5 Převaděč požadavků na itinerář

Související moduly

Převaděč požadavků na itinerář nedostává žádný vstup, ale naopak je sám používán jako parametr pro prakticky všechny ostatní moduly: kromě programu ho potřebují i generátor jízdních řádů a generátor cestujících.

Nutné minimum modulu

Modul musí obsahovat alespoň jednu třídu, která impementuje rozhraní `OptimeTable.Core.Interface.IRequirementToItinerary`. V případě, že jich obsahuje víc, je vybrána právě jedna z nich, a není specifikováno, podle jakého klíče. Knihovna, která vznikla kompilací zdrojového kódu obsahující tuto třídu, je technicky postačující pro použití s programem.

Tipy

Viz podsekce Tipy u předchozí sekce.

4. Experimentální část

V této kapitole se budeme zabývat experimenty, které byly provedeny, aby potvrdily nebo vyvrátily smysluplnost použití genetického algoritmu pro generování jízdních řádů.

V první podkapitole jsou předvedeny elementární experimenty, které využívaly jednoduché modely a jednoduché konfigurace genetického algoritmu. U těchto experimentů je velká přednost v tom, že je možné snadno a přesně intuitivně odhadnout správný výsledek. Následně, pokud daná konfigurace genetického algoritmu dospěla ke jinému výsledku, lze buď zúžit spektrum modelů, na něž se hodí, nebo ji případně zcela vyloučit.

V druhé podkapitole jsou konfigurace nasazeny na složitější modely, kde už lze pouze komentovat výsledek a nelze říci „Správně“ či „Špatně“.

Třetí podkapitola se zabývá nasazením genetických algoritmů na model pražské dopravní sítě a výsledky, k nimž dospěly.

4.1 Elementární experimenty

Každý experiment se skládal z topologie, dat o pasažérech a konfigurace genetického algoritmu (dále konfigurace GA či prostě konfigurace). Pro každou dvojici topologie a dat o pasažérech tedy máme sadu výsledků od různých konfigurací.

4.1.1 Konfigurace GA

V této části experimentování byly využity následující genomy (viz 3.3.6):

- `CountGeneSpecimen` (dále jako CGS)
- `LineGeneSpecimen` (dále jako (LGS)
- `LineGeneSpecimenWithSineGenome` (dále jako LGS(s))
- `OrderedLineGeneSpecimen` (dále jako OLGS)

a následující ohodnocovací funkce (viz 3.3.6):

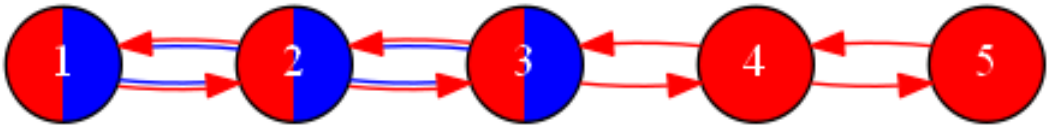
- `AverageWaitingRatioFitness` (dále jako AWR)
- `CubicAverageWaitingRatioFitness` (dále jako CAWR)
- `PassengersOverMinimalRatioFitness` (s různými parametrizacemi, dále jako POMRx, kde x je percentil, na nějž je instance inicializována)

4.1.2 Modely & výsledky

U každého modelu je trasa linky vyznačena posloupností uzlů a hran stejné barvy, tuto barvu má každá linka unikátní. Není-li řečeno jinak, byla nastavena velikost generace 8 a počet generací také 8. Výsledkem je nejlepší jedinec z poslední generace. Význam číselného hodnocení je uvedený u popisu použité hodnotící funkce, z toho také vyplývá nejlepší a nejhorší číselná hodnota. Výstupy algoritmu u každé z tabulek jsou ve složce se jménem odpovídajícím číslu tabulky uvnitř složky tables, jde o přílohy 8 až 20.

Úsečka, dvě linky, pasažéři na delší z nich

Obrázek 4.1: Úsečka, dvě linky, pasažéři jezdí mezi stanicemi 1 a 5



Tabulka 4.1: Data - Úsečka, dvě linky, pasažéři jezdí mezi stanicemi 1 a 5

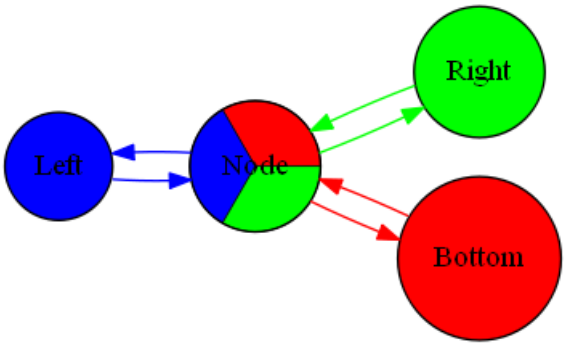
Konfigurace	Červená	Modrá	Hodnocení
CGS, AWR	2	0	0,0037590325
LGS, AWR	2	0	0,0037590325
LGS(s), AWR	2 (1)	0 (0)	0,0037590325
OLGS, AWR	2	0	0,0037590325
OLGS, CAWR	2	0	5,31E-008
OLGS, POMR50	2	0	0,0044
OLGS, POMR70	2	0	0,0044

Pasažéři jezdí výhradně mezi stanicemi 1 a 5.

Zde je výsledek jednoznačný a správný: všechna vozidla byla nasazena na linku, která vozí pasažéry přímo mezi jejich výchozí a cílovou stanicí. V případě, že by jedno vozidlo bylo alokováno na krátkou linku, zdvojnásobil by se interval dlouhé linky a tím by se prodloužila doba čekání cestujících.

Dráha ve tvaru „T“, pasažéři mezi stanicí Bottom a stanicemi Left a Right

Obrázek 4.2: Dráha ve tvaru T, pasažéři mezi spodní stanicí Bottom a stanicemi Left a Right



Tabulka 4.2: Data - Dráha ve tvaru T

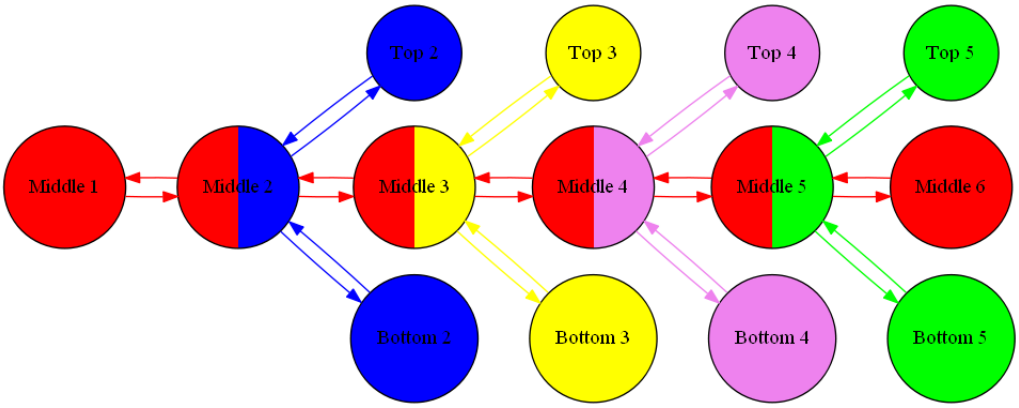
Konfigurace	Červená	Modrá	Zelená	Hodnocení
CGS, AWR	10	4	5	0,49176
LGS, AWR	7	6	7	0,45236
LGS(s), AWR	10 (3)	3 (1)	7 (4)	0,55835
OLGS, AWR	10	5	5	0,49419
OLGS, CAWR	10	5	5	8,68896E-004
OLGS, POMR50	10	3	7	0,6775
OLGS, POMR70	10	3	7	0,2675

Zde už jsou výsledky rozmanitější. Na první pohled vidíme, že kritérium percentilu lidí majících poměr čisté a hrubé doby cesty 70% je v tomto případě nevhodné, proto se žádný jízdní řád u POMR70 nedostal nad nulové hodnocení. Dále vidíme, že všechny ostatní konfigurace dospěly k dominanci centrální linky, což je správně (dokonce i poslední konfigurace, tam je to ale spíše náhoda), ale je na místě otázka, jak moc velká dominance je správná. Intuitivně: pokud každý pasažér potřebuje projet ze stanice Node do stanice Bottom, a zároveň právě polovina potřebuje ze stanice Node do Right a právě polovina z Node do Left, pak by mělo být dvakrát více vozidel na červené lince, než na modré nebo zelené.

V každém případě, z tohoto experimentu pozorujeme, že u konfigurací, kde je nízký průměrný poměr čisté a hrubé doby cesty, vede použití funkce POMR k marginalizaci části cestujících tak, aby se vytvořil dostatečně velký vzorek cestujících, kteří byli přepraveni v dobrém čase.

Dráha ve tvaru žebříku, pasažéři mezi sousedními svislicemi

Obrázek 4.3: Dráha ve tvaru žebříku, pasažéři mezi sousedními svislicemi



Tabulka 4.3: Data - Dráha ve tvaru žebříku, pasažéři mezi sousedními svislicemi

Konfigurace	Modrá	Žlutá	Fialová	Zelená	Červená	Hodnocení
CGS, AWR	3	4	4	2	5	0,63636;
LGS, AWR	4	1	4	5	5	0,6069
LGS(s), AWR	4 (0)	2 (0)	6 (1)	2 (2)	5 (1)	0,6162
OLGS, AWR	4	4	3	4	4	0,6087
OLGS, CAWR	5	3	2	4	5	0,01089
OLGS, POMR50	4	2	4	4	5	1
OLGS, POMR70	4	5	2	2	6	0,4375

Zde netřeba komentovat výsledky s funkcí POMR, ta je pro tuto situaci evidentně nevhodná (příliš mnoho cestujících), ale lze na tom testovat dispozice každého z jedinců maximalizovat hodnocení jedinců.

Jak je také vidět z výpisu průběžných výsledků simulace (viz Příloha 1), funkce AWR je příliš "plochá", což znamená, že silní jedinci nemají velkou šanci se prosadit, jako například jedinec 4 v generaci 2, jehož hodnocení přes 0,17 je nejvyšší, ale mezi šesti jedinci s hodnocením okolo 0,15 se neprosadí. Dá se říct, že je opakem funkce POMR, která naopak dává velice ostrou hranici mezi vhodnými a nevhodnými, někdy ale takovým způsobem, že tuto hranici žádný jedinec nepřekoná.

Z Přílohy 1 také vyčteme další pozorování: v případě, že optimálním řešením je nějaký vyrovnaný poměr vozidel — což obzvláště v reálném použití nepochybně bude — pak generace začnou kolísat na průměrných řešeních, a dobré, vyvážené řešení se neudrží, protože ve fázi crossoveru je skříženo s jiným, které s velkou pravděpodobností rovnováhu naruší.

Dalším poznatkem, který si odnášíme z této sady experimentů, je, že není vhodné mít velkou redundanci genomu - tedy že se „mnoho“ genů serializuje na stejný jízdní řád. V případě formátu LGS je tato redundance až $n!$, při celkovém počtu možných genů k^n, kde k je arita abecedy a n je počet vozidel.

Závěry z této části

Vidíme, že problémem formátu CGS je abeceda, která neumožňuje bez dodatečné harmonizace využít možnosti sítě v plném rozsahu. Problémem formátu LGS je zase — jak je řečeno v posledním odstavci poslední sekce — velká redundance genomu.

Když se pokusíme spočítat možná přiřazení vozidel k linkám, je to $\frac{(v+l)!}{v!l!}$, tedy vlastně kombinační číslo $\binom{v+l}{l}$.

U ohodnocovacích funkcí vidíme dvě protichůdné tendence, každá z nich byla akcentována jinou z funkcí:

1. Maximalizace kvality obsloužení všech cestujících.
2. Maximalizace podílu cestujících, jejichž obsloužení překročilo nějaké minimum.

Na první pohled se zdají tato kritéria být v souladu, ale v okamžiku, kdy není dostatek vozidel pro obsloužení všech cestujících, vedou tyto dva přístupy k dramaticky odlišným výsledkům. První přístup vede k rovnoměrnému obsloužení všech cestujících za cenu toho, že cesta každého jednoho z nich trvá poměrně dlouho. Naproti tomu druhý přístup vede k přednostní alokaci vozidel na určité linky, což na jedné straně zařídí, že kvalita obsluhy části cestujících překročí stanovené minimum, za cenu výrazného zhoršení kvality pro zbytek.

Vidíme také, že nedostatkem ohodnocovací funkce AWR je její příliš pomalý nárůst, který u experimentu s „žebříkovou“ dráhou vedl k upozadnění nejdominantnějšího jedince v generaci. Tento problém vyřešíme zavedením elitismu.

Pro vyčerpání poznatků této části a její dotažení k závěru zúžíme problém: ve skutečnosti jezdí spoje na linkách v pravidelných intervalech, a perioda spoje zůstává stejná řádově hodiny. Tedy můžeme najít rozdělení dne do časových úseků takové, že v rámci každého časového úseku je konstantní perioda na každé lince. Dále víme, že perioda je inverzní funkce k frekvenci spojů, a frekvence spojů je lineární k počtu vozidel alokovaných pro danou linku. Z toho nám vyplývá, že lze v každém časovém úseku popsat jízdní řád jako rozdělení vozidel mezi linky, a tudíž pro každý časový úsek existuje $\binom{v+l}{l}$ jízdních řádů. V závislosti na zvolené jemnosti vzorkování bychom pro časový úsek délky P dostali pro celý den právě $\frac{1440}{P} \times \binom{v+l}{l}$ jízdních řádů. Výsledek je pak teoretické optimum, v průběhu času není možné měnit alokace linek příliš dramaticky.

Do další fáze proto zkusíme vytvořit nový formát genomu. Od formátů CGS a (O)LGS se bude lišit tím, že:

- Na rozdíl od CGS bude využitelný celý definiční obor genomu (ve formátu CGS kvůli harmonizacím omezujícím sumu genů na 20 není)
- Bude mít menší redundanci informace než formát (O)LGS

Na základě výše popsaných experimentů vyzkoušíme nový formát. Nazveme jej PriorityGeneSpecimen a jeho základní myšlenka bude taková, že jedinec bude obsahovat dva typy genů s následujícími funkcemi:

1. První bude mít jen jedinou pozici a bude vyjadřovat minimální počet vozidel alokovaných na všech linkách (tzn. např. pro hodnotu 2 musí mít všechny linky alespoň 2 vozidla).

2. Druhý bude mít tolik pozic, kolik je linek, hodnota na pozici odpovídající lince bude vyjadřovat míru priority, s níž bude vozidlo ze zbytku přiděleno právě této lince.

První typ genu zkusíme s aritou 1 a s aritou $\lfloor \frac{PocetVozidel}{PocetLinek} \rfloor$. V prvním případě dostaneme obdobu formátu CGS s využitím celého definičního oboru a s jistým využitím všech vozidel, která jsou v síti k dispozici. Ve druhém případě bude předchozí pravidlo platit pro vozidla, která nebyla přiřazená podle prvního typu genu. Arita druhého typu genu bude odpovídat maximálnímu možnému podílu dvou nenulových alokací vozidel, zkusíme nastavit hodnotu $min(10, PocetVozidelVSiti - 1)$.

Pro interpretaci druhého genomu je možné zvolit víc přístupů. Je možné:

1. Přesně matematicky rozpočítat počet vozidel na i-té lince $P_i = Round(P \times \frac{g_i}{\sum g_j})$. Touto formulí může vzniknout nedostatek či přebytek vozidel (např. pro $P = 4, g_1 = g_2 = g_3 = 1$). Zbývající vozidla můžeme rozdělit například tak, že je přidělujeme k linkám po jednom podle sestupného pořadí priorit. Pro přebytek vozidel postupujeme naopak.

2. Pravděpodobnostně. Po harmonizaci posloupnosti (g_i) na součet 1 můžeme g_i chápat jako pravděpodobnost přidělení vozidla k i-té lince. Tento přístup má ale velkou nevýhodu, že genom neurčuje rozdělení vozidel jednoznačně, a experimenty ukázaly, že v důsledku může mít jeden genom velmi odlišná ohodnocení. Tuto metodu proto zavrhneme a vozidla budeme rozdělovat na základě matematického vzorce.

Další změnou, kterou provedeme, bude rozšíření generace na víc jedinců: 8 náhodně vygenerovaných jedinců příliš zužuje zkoumaný prostor genomů, zkusíme nastavit počet na 32. Pokud to nebude stačit na odstranění problému s příliš nízkou diferenciací ohodnocovací funkce, zkusíme změnit i tu.

Také nezapomínáme na nutnost změny intervalu linky v průběhu dne, ale zdá se nám správné nejprve odladit dobrou konfiguraci GA pro konstantní interval na jednodušších modelech.

4.1.3 Experimenty s `PriorityGeneSpecimen`

Nejprve se podíváme na výsledky `PriorityGeneSpecimen` (dále již pouze PGS) na modelech, které jsme představili v předchozí sekci. Použijeme i variantu s nulovým počtem společných vozidel, kterou budeme značit PGS0. Pro srovnání s předchozí sekcí použijeme jako měřítko také jedince LGS a OLGS a jako ohodnocovací funkci zatím AWR.

V této části má generace velikost 32, počet generací je 20 a velikost elity 2.

U každého jedince budeme sledovat maximální hodnocení ve výchozí generaci, v poslední generaci a maximum za celý průběh výpočtu.

Odtud vidíme hypotézu, kterou podpoří nebo vyvrátí další experimenty, že LGS a PGS „neudrží“ optimální řešení a křížením se vhodné geny nepředávají, a že mají obecnou tendenci konvergovat k suboptimálním řešením, která se vyznačují vyrovnaným zastoupením vozidel na všech linkách.

Tabulka 4.4: Data - Srovnání jedinců na dráze typu „T“

Jedinec	Červená	Modrá	Zelená	Hodnocení -začátek	**-konec**	-celkem
LGS	10	3	7	0,45236	**0,55834**	0,55834
OLGS	10	3	7	0,4837	**0,55834**	0,55834
PGS0	10	3	7	0,5349	**0,55834**	0,55834
PGS	10	3	7	0,49419	**0,55834**	0,55834

Tabulka 4.5: Data - Srovnání jedinců na dráze Žebřík

Konfigurace	Modrá	Žlutá	Fialová	Zelená	Červená	-začátek	**-konec**	-celkem
LGS	4	4	4	2	5	0,63636	**0,66667**	0,66667
OLGS	4	4	4	2	5	0,63636	**0,66667**	0,66667
PGS0	4	4	4	2	5	0,63636	**0,66667**	0,66667
PGS	4	4	4	2	5	0,63636	**0,66667**	0,66667

V tomto případě cestující chtějí cestovat mezi sousedními svislicemi, tedy například ze stanice Top 2 do Bottom 3 a obráceně, a totéž až po Top 4 s Bottom 5 a Top 5 s Bottom 4.

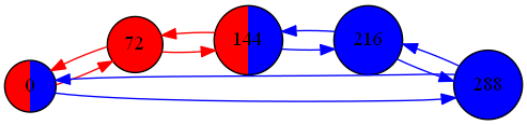
!!! OMIT !!!

Tabulka 4.6: Data - Srovnání jedinců na dráze Žebřík (ii)

Konfigurace	Modrá	Žlutá	Fialová	Zelená	Červená	-začátek	**-konec**	-celkem
LGS	1	1	1	2	0	0,61538	**0,61538**	0,61538
OLGS	1	2	1	1	0	0,61538	**0,61538**	0,61538
PGS0	1	2	1	1	0	0,61538	**0,61538**	0,61538
PGS	1	1	1	2	0	0,61538	**0,61538**	0,61538

Zde ovšem jako dodatek poznamenejme, že při formátu PGS bylo v předposlední generaci maximum 0,57, tedy že optimální jedinci v jedné z posledních fází zcela vypadli, což se dá chápat jako podpora naší hypotézy.

Obrázek 4.4: Cyklická dráha, pasažéři mezi stanicemi 0 a 144



Tento model je poněkud naivní, ale jeho výhoda spočívá v tom, že správné řešení je naprosto jednoznačné a jediné a GA k němu buď dospěje nebo ne.

!!! STEJNÉ !!!

Tabulka 4.7: Data - Srovnání jedinců na kruhové dráze

Konfigurace	Červená	Modrá	-začátek	**-konec**	-celkem
LGS	4	0	1	**1**	1
OLGS	4	0	1	**1**	1
PGS0	4	0	1	**1**	1
PGS	4	0	1	**1**	1

Zde je stejnětak třeba poznamenat, že po dosažení optima v některé z následujících generací toto optimum vypadlo, konkrétně u formátů PGS a OLGS.

Závěry z těchto jednoduchých modelů nejsou napohled tak jednoznačné. Můžeme je ale rozdělit na modely, v nichž je správným řešením „poměrně“ vyrovnané rozdělení vozidel mezi všechny linky, a modely, kde je správné řešení buď potlačení jedné nebo několika málo linek, nebo naopak výrazné posílení jedné nebo několika málo linek. Vidíme, že na vyrovnaných distribucích (žebřík a dráha tvaru „T“) formáty LGS i PGS kolísají a neudrží nejlepší řešení dosažené v průběhu algoritmu. Naopak na trase s jednoznačnou dominancí jedné linky se formát LGS chová velice jednoznačně.

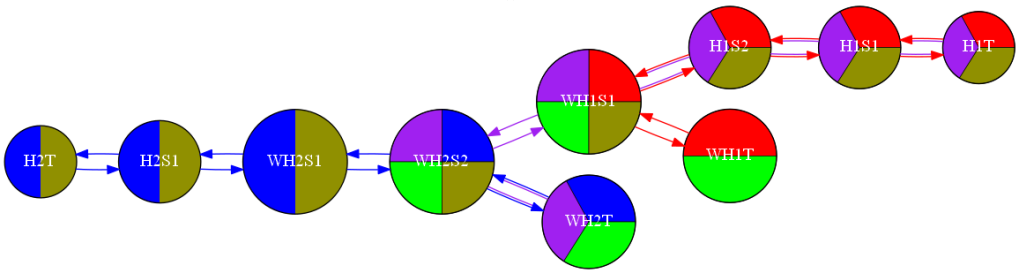
U formátu PGS pozorujeme problematické chování. Problém formátu PGS je, že je složený ze dvou různých genomů, které se oba řídí rovnoměrným rozdělením. V případě, že pro jednu z možných hodnot společného počtu vozidel (dále primárního genomu) existuje několik málo rozdělení (dále sekundárního genomu) s vysokým či nejvyšším hodnocením, ale celkový průměr hodnocení pro tu hodnotu je nižší než pro ostatní, pak formát PGS pravděpodobně konverguje k té jiné hodnotě. Z toho důvodu nedojde k dostatečnému průzkumu sekundárního genomu pro tuto primární hodnotu. Pro maximální hodnotu primárního genomu navíc dochází k významnému snížení významu sekundárního genomu.

Dá se říci, že formát PGS s velkou pravidelností vydává jako výsledek takřka rovnoměrné rozdělení vozidel, což zpravidla není špatná odpověď, ale také není optimální a navíc lze takovou odpověď dát triviálně bez potřeby výpočetní techniky.

U formátu LGS pozorujeme, že často, pokud vyrovnané řešení patří k optimálním nebo aspoň dobrým, je jeho výsledek vygenerovaný už při inicializaci. Toto je důsledkem rovnoměrného rozdělení náhodné funkce. Další experiment byl

o něco komplexnější a přiblížený reálnému světu. Obsahuje dvě oblasti, „sídliště“ a „kanceláře“, přičemž ráno jezdí cestující ze sídliště do kanceláří a večer obráceně. Schéma je níže:

Obrázek 4.5: Model „Sídliště a kanceláře“



Pro 5 vozidel k dispozici byly výsledky následující:

Tabulka 4.8: Data - Srovnání jedinců na modelu „Sídliště a kanceláře“

Konfigurace	Červená	Modrá	Fialová	Zelená	Žlutá	-začátek	**-konec**	-celkem
LGS	0	1	0	1	3	0,5248	**0,5248**	0,5248
OLGS	1	2	2	0	0	0,4995	**0,5079**	0,5079
PGS0	0	1	0	1	3	0,501	**0,5248**	0,5248

a při počtu 20 vozidel

Tabulka 4.9: Data - Srovnání jedinců na modelu „Sídliště a kanceláře“

Funkce	Červená	Modrá	Fialová	Zelená	Žlutá	-začátek	**-konec**	-celkem
LGS	2	5	3	3	7	0,8827	**0,8987**	0,8987
OLGS	2	5	3	3	7	0,88361	**0,89869**	0,89869
PGS0	1	1	1	3	14	0,88199	**0,9069**	0,9069

Různá nastavení pravděpodobnosti mutace a křížení nepřinesla zlepšení: buď se algoritmus neudržel na optimální hodnotě, nebo se k ní ani nedostal. Naopak umocnění hodnotící funkce na vysoký exponent přineslo okamžitý a jednoznačně pozitivní výsledek („Vysokým exponentem“ myslíme exponent, který zvýší podíl hodnocení dvou nejlepších zaznamenaných řešení na 2).

Tento výsledek se dostavil stejně u formátu LGS jako u PGS0. Mezitím byl vyzkoušen ještě `ModulloGeneSpecimen`, dále MGS. Princip: genom má $L-1$ pozic, kde L je počet linek. Arita abecedy je $V+1$, kde V je počet vozidel. Pro první pozici znamená počet přímočaře počet vozidel, která pro první linku v pořadí budou alokována. Pro $(n+1)$-ní pozici znamená hodnota h ve skutečnosti $h \bmod V_n$, kde V_n je počet vozidel, která zbyla po provedení algoritmu na pozicích $1..n$.

Takový návrh byl vyzkoušen, protože u formátu PGS0 se zdál problém ve složitém přepočtu genomu na počty vozidel (přidávání či odebírání vozidel, která chyběla či přebývala důsledkem zaokrouhlovací chyby).

Obecně se dá ale říci, že všechny genomy se chovaly podobně: dosáhly dříve či později stejného optima a následně kolísaly s periodou desítek až stovek generací mezi sadou optimálních a sadou suboptimálních řešení. Tento problém, jak již je popsáno výše, vyřešilo umocnění ohodnocovací funkce. Proto v této fázi opustíme, alespoň dočasně, studium genomů, a zaměříme se na vývoj a porovnávání ohodnocovacích funkcí.

4.1.4 Experimenty s ohodnocovacími funkcemi

Připomeňme, že dosud máme dvě funkce, každou s více variantami:

- `AverageWaitingRatio`, která vrací poměr $\frac{\sum CistaDobaCesty}{\sum HrubaDobaCesty}$, navíc s variantami umocnění tohoto čísla na nějaký přirozený exponent (speciálně, v základní verzi, 1).

- `PassengersOverMinimumRatio`, která vrací podíl cestujících, kteří mají hodnotu $\frac{CistaDobaCesty}{HrubaDobaCesty}$ vyšší než parametricky zadané minimum.

První, o co se pokusíme, je zobecnění druhé z funkcí. Zásadní nevýhoda její původní verze je její velice ostrý přechod od 0 k 1: na velké části definičního oboru nabývala hodnoty 0, v některých případech dokonce na celém. Zkusíme tuto nevýhodu zmírnit tím, že umožníme několik minimálních hodnot, každá bude mít svou váhu. Celkové hodnocení tedy bude vážený průměr hodnot původní funkce pro tato různá minima.

Volba sady minimálních hodnot nemůže být jedna univerzální, přesně daná. Respektive může, ale v takovém případě může nastat jeden ze dvou problémů, z nichž každý způsobí malý rozptyl hodnot funkce v průběhu simulace:

- V simulaci bude „průchodnost“ sítě příliš nízká: pasažéři budou mít poměr času cesty příliš blízký nule a překročí buď žádné nebo jen některá ze spodních minim.

- Přesně opačný problém: poměr se bude blížit 1.

Nyní vyzkoušíme dvakrát dvě varianty: pět a deset minim, rovnoměrně rozdělené a rozdělené s větší hustotou k okrajům intervalu $[0, 1]$. Jako pokusný genom si vezmeme nejprve nejstabilnější PGS0, vyzkoušíme jej pro generaci velikosti 24 a počet generací 100 (případně u komplexnějších modelů méně, z časových důvodů). V tabulce níže bude uveden počet modelů, na nichž došlo ke konvergenci. Konvergencí pro jednoduchost rozumějme situaci, kdy nejlepší jedinec z konečné generace je v této generaci zastoupen alespoň z 50

U nerovnoměrně rozdělených pěti hodnot byly vyzkoušeny tyto tři kombinace:

- Rovnoměrně rozdělené mezní hodnoty, exponenciálně klesající váhy - (0.1, 0.5), (0.3, 0.25), (0.5, 0.125), (0.7, 0.0625), (0.9, 0.0625) (poslední hodnoty stejné, aby se váhy sečetly na 1)

- Nerovnoměrně rozdělené mezní hodnoty, exponenciálně klesající váhy - váhy stejně jako v předchozím případě, hodnoty po řadě 0.3, 0.5, 0.7, 0.8, 0.9

- Rovnoměrně rozdělené mezní hodnoty, exponenciálně rostoucí váhy - (0.1, 0.03125), (0.3, 0.0625), (0.5, 0.125), (0.7, 0.25), (0.9, 0.5) (zde je skutečně respektována exponenciála)

U deseti hodnot jsme využili většího počtu k tomu, abychom zkusili namodelovat „parabolické“ rozmístění bodů: v případě přebytku vozidel se hodnoty soustředí u 1 a v případě nedostatku u 0, protože rozmístíme hustěji body u okrajů intervalu $[0, 1]$. A zkusíme tři varianty vyvážení bodů:

- Rovnoměrné - (0.02, 0.1), (0.08, 0.1), (0.18, 0.1), (0.32, 0.1), (0.5, 0.1), (0.68, 0.1), (0.82, 0.1), (0.92, 0.1), (0.98, 0.1)
- Zesilující prolnutí paraboly (ve smyslu, že na okrajích větší hustota bodů a navíc i vyšší váha bodů) - (0.02, 0.5), (0.08, 0.32), (0.18, 0.18), (0.32, 0.08), (0.5, 0.02), (0.68, 0.08), (0.82, 0.18), (0.92, 0.32), (0.98, 0.5)
- Komplementární k prolnutí paraboly - (0.02, 0.02), (0.08, 0.08), (0.18, 0.18), (0.32, 0.32), (0.5, 0.5), (0.68, 0.32), (0.82, 0.18), (0.92, 0.08), (0.98, 0.02)

Při posouzení, zda algoritmus na daném počtu iterací konvergoval nebo ne, bylo použito pravidlo, že suma hodnocení nejdominantnějších jedinců musí převyšovat sumu hodnocení ostatních jedinců. Z tohoto pravidla vyplývají možné problémy: například mohou vzniknout dvě skupiny jedinců, v jedné ze skupin je hodnocení o malý zlomek menší než ve druhé. Když ovšem uvážíme praktické využití algoritmu, prostor jízdních řádů, který je algoritmem prohledáván, je příliš komplexní na to, aby mohl program vydat jednoznačnou odpověď, a výsledná generace bude spíše vymezovat oblast, v rámci níž se finální řešení bude pohybovat.

Tabulka 4.10: Data - Srovnání ohodnocovacích funkcí založených na počtu cestujících překračujících dané minimální hodnoty

Funkce	Konvergence 10	Konvergence 25	Konvergence 50
5, rovnoměrné rozdělení	5	7	5 !!
5, r. rozdělení, exp. klesající	5	5	8 !!
5, nr. rozdělení, exp. klesající	6	7	8 !!
5, r. rozdělení, exp. rostoucí	5	5	8
10, rovnoměrné rozdělení	5	5	7
10, par. rozmístění, r. rozdělení	5	5	7
10, par. rozmístění, par. rozdělení	5	5	7
10, par. rozmístění, komp. rozdělení	4	7	7

Při pohledu do výstupu „5, rovnoměrně“ si můžeme všimnout, že je tato funkce velice špatně diferencovaná, „5, nerovnoměrně“ jen o málo lépe. To znamená, že i když z hlediska kritéria konvergence vychází pět bodů s rovnoměrným rozdělením a exponenciálně rostoucím hodnocením jako nejlepší, ve skutečnosti je to za cenu nižší diverzity funkčních hodnot. To je dobře vidět na výsledcích u obou žebříkových grafů, kdy algoritmus dokonvergoval k jedné funkční hodnotě, pod kterou ale byli tři různí jedinci. Vzhledem k tomu, že tito jedinci mají u ostatních ohodnocovacích funkcí různé hodnocení, znamená to, že skutečně tato funkce zplošťuje výsledek. Může být ale použita pro dobrý odhad skupiny jedinců, kteří patří mezi optimum.

Další poznatek: všechny funkce konvergovaly zpravidla při stejném počtu iterací na stejné množině modelů.

Dále uděláme několik pokusů s kvantilovou funkcí: funkce vrátí coby hodnocení vážený průměr daných kvantilů (zde se nabízí klást kvantily rovnoměrně nebo nerovnoměrně a váhu jim přiřadit rovnoměrnou nebo nerovnoměrnou, vyzkoušíme všechny možnosti). Uděláme dva jednoduché pokusy: kvantily rovnoměrně pro 5 bodů (0.0, 0.2, ..., 1.0) a po 10 (tedy 0.0, 0.1, ..., 1.0). Potom uděláme dva druhy parabolického rozmístění bodů: nejdřív tak, jako u předchozího experimentu,

tzn. body (0.02, 0.08, ..., 0.98), potom tak, že vrchol paraboly je umístěný v prostředku intervalu, tzn. body (0.0, 0.18, 0.32, 0.42, 0.48, 0.5, 0.52, 0.58, 0.68, 0.82, 1.0).

Tabulka 4.11: Data - Srovnání ohodnocovacích funkcí založených na kvantilové funkci

Funkce	Konvergence 10	Konvergence 25	Konvergence 50
5, rovnoměrně	3	5	5 (-3)
10, rovnoměrně	4	6	6
10, inv. parabolicky	3	5	5
10, parabolicky	3	5	6

Zkusíme ještě kvantilovou funkci vylepšit tím, že parabolické rozmístění hodnot ještě zesílíme parabolickým rozdělením vah jednotlivých kvantilů, a celé navíc umocníme. Je otázkou, na jaký exponent. Příliš nízký exponent nám nepomůže lépe diferenciovat funkci, příliš vysoký způsobí, že hodnocení bude řádově nízké, tedy potenciálně problematické na čtení i na přesnost reprezentace. Proto zvolíme exponent 5, protože hodnocení při tomto exponentu se ukazovalo jako dobrý kompromis těchto protichůdných aspektů.

Tabulka 4.12: Data - Srovnání ohodnocovacích funkcí typu

Funkce	Konvergence 10	Konvergence 25	Konvergence 50
10, par., par., exponent	4	5	5 (-3)

Všechny předchozí výsledky byly zatíženy dvěma hlavními problémy:

- Neschopnost udržet dosažené optimum (optimální hodnota byla málo výrazná, a proto v rámci pravděpodobnostního výběru, kdy byla obklopená mnoha suboptimálními jedinci s rovnou hodnotou nebo i genomem, nakonec vymizela).
- Zacyklení: generace kolísají mezi dvěma lokálními optimy, z nichž jedno bývá i globální.

Jako řešení těchto problémů zkusíme nasadit elitismus. Také upustíme od zkoumání konvergence na 10 generacích, která může celkovou konvergenci vyvrátit, ale ne potvrdit.

Tabulka 4.13: Data - Srovnání ohodnocovacích funkcí s elitismem (elita velikosti 2

Funkce	Konvergence 25	Konvergence 50
10, par., par.,	6	9
10, par., r.,	7	9

Funkce s elitismem dávají velice uspokojivé výsledky, a to bez větších rozdílů na ohodnocovací funkci.

V následující částí proto budeme pracovat s kvantilovou funkcí s 10 parabolicky rozmístěnými hodnotami s rovnoměrnou váhou, umocněnou na pátou. V experimentech použijeme generaci velikosti 24 a uděláme 100 iterací.

4.2 Složitější konfigurace

Vytvářet konfigurace ručně by bylo jednak časově náročné a navíc dost možná neúčelné. Jako složitější konfiguraci v tomto případě potřebujeme náhodné sítě, abychom mohli na nich demonstrovat, že algoritmus je skutečně obecný a zafunguje dobře i na netriviálních sítích. Tyto náhodné konfigurace naopak budou problematické tím, že dost možná budou nerealistické, to nám ale nevadí, protože je potřebujeme pouze k demonstraci obecnosti algoritmu. Na realistických sítích algoritmus vyzkoušíme v následující fázi.

4.2.1 Náhodné generování

Náhodné generování sítě probíhá takto: máme pokusnou síť, které budeme říkat Hřiště. Hřiště je čtvercová mřížka $m \times n$. Umožňuje přidávání linek vedoucích z (x_1, y_1) do (x_2, y_2), kde $0 \leq x_1, x_2 < m$ a $0 \leq y_1, y_2 < n$. Po přidání linky se trasa vytvoří jako náhodná procházka, která je tvořena $|x_2 - x_1|$ horizontálními kroky a $|y_2 - y_1|$ vertikálními kroky. Těchto celkem $|x_2 - x_1| + |y_2 - y_1|$ je náhodně seřazeno, čímž je docílena v daném kontextu náhodná trasa linky.

Linky potom přidáváme tak, že si Hřiště rozdělíme na čtyři kvadranty (důsledkem toho je, že m i n musí být sudé), a celkem l-krát zvolíme dvojici **různých** kvadrantů, v rámci každého z nich vybereme libovolný náhodný bod a tyto body použijeme jako výchozí resp. cílový bod linky.

Potom spustíme simulaci.

4.2.2 Výsledky

Ve všech případech poslední generace byla „víceméně“ zkonvergovaná. Slovem „víceméně“ myslíme, že alokace vozidel pro většinu linek se pohybovala v intervalu velikosti tři (což není ani procento celkového počtu vozidel), a hodnocení jízdních řádů se lišilo o jednotky procent. Tedy se dá argumentovat, že na nahodilé síti algoritmus dává dobrý základ hrubého odhadu správného rozdělení vozidel.

Naopak ani v jednom případě nedošlo k opravdové konvergenci, nejlepší jedinec se vždy vyskytoval pouze v jediném exempláři. Čímž dostáváme závěr, že algoritmus sám o sobě nevydává správné řešení, dá se použít pouze k vymezení prostoru, v němž se optimální řešení nachází.

Tento zúžený prostor se dá brát, že má velikost $\binom{V - \sum_{i=0}^{L-1}(l_{min_i}) + l_0}{l_0}$, kde V je celkový počet vozidel v síti, L počet linek, l_{min_i} je minimální počet vozidel alokovaných pro i-tou linku v poslední generaci a l_0 je počet linek s nenulovým rozdílem maxima a minima. Jedná se ve skutečnosti o to, že zúžením stavového prostoru jsme převedli úlohu na takovou, kde rozdělujeme pouze „sporná“ vozidla mezi linky, které nemají jednoznačně vymezený počet vozidel v poslední generaci.

Model i data jsou příliš velká na to, aby byla vložena sem do textu práce, je možné je nalézt jako přílohy 2 - 7 této práce.

4.3 Praha

Experiment s topologií pražské tramvajové sítě je zde spíš pro zajímavost, případně jako demonstrace toho, že i na reálných topologiích vydává smysluplné

výsledky. Nemáme k dispozici žádná relevantní data pro požadavky pasažérů na pražskou tramvajovou síť, takže necháme jen náhodně vygenerovat dvojice výchozích a cílových stanic v množství přibližně odpovídajícím možnostem sítě. Celý záznam simulace spolu s daty o pasažérech je ve složce Praha. Trasy linek platí k půlce roku 2011.

Tabulka 4.14: Data - Alokace vozidel v Praze

Linka	Počet spojů
1	22
3	28
4	6
5	11
6	11
7	44
8	17
9	33
10	22
11	28
12	22
14	44
15	6
16	17
17	17
18	44
19	22
20	11
21	11
22	28
24	6
25	11
26	28
36	11

Připomeňme, že počet spojů na lince je inverzní funkcí intervalu, tedy že čím více spojů na lince jezdí, tím je nižší interval. Pokud linka má trasu, kterou tam a zpátky trvá projet 80 minut, a je na ní alokováno 40 vozidel, pak z toho v našem zjednodušeném modelu vyplývá interval 2 minuty. V našem případě má spoj kapacitu 150 (pro srovnání, kapacita jednoho vagonu tramvaje Tatra T3, nejrozšířenějšího typu jezdícího v Praze, je 110).

Jak bylo výše řečeno, požadavky pasažérů byly generovány náhodně, a pro výsledek měl větší význam jejich počet a topologie sítě, což jsou:

- „Úzká hrdla“ dopravní sítě, jimiž musí projet větší množství cestujících.
- Návrh linek: při množství cestujících a rovnoměrném rozdělení jejich tras se dá očekávat, že bude potřebná obsluha každé hrany v grafu, což znamená, že pokud nějaká linka je jediná, která obsluhuje některou hranu, pravděpodobně bude muset mít relativně vysokou alokaci vozidel

Výše popsanému by mohla odpovídat vysoká alokace vozidel na následujících linkách, která určitý úsek dopravní sítě obsluhují výlučně:

- 7, Na Knížecí - Radlická
- 11, I. P. Pavlova - Radhošťská
- 19, Špitálská - Nový Hloubětín

Mimo tento aspekt je zajímavé, že linky 3, 9 a 22, které jsou páteřní a nadprůměrně obsazované ve skutečných jízdních řádech, jsou nadprůměrně obsazené i zde. Naopak linka 15 byla od doby pořízení těchto dat zrušena a linka 4 má relativně nízkou prioritu, jezdívala pouze ve špičkách a například pro letní prázdniny roku 2014 byla zrušena.

V souhrnu se dá říci, že tyto výsledky dávají naději na využitelnost genetického algoritmu i celého systému OptimeTable na realistických datech, alespoň pro vymezení prostoru, v němž se pak bude hledat konečný výsledek.

Závěr

Cíle práce

Zde v závěru zhodnotíme práci tak, že po bodech zvážíme, zda jednotlivé cíle práce byly splněny.

Cíl: „Vytvořit model dopravní sítě a vymodelovat vztah cestujících a jízdního řádu, podle kterého vozidla této sítě jezdí.“

Realizace: Projekt je modularizován tak, že požadavky zákazníků jsou poskytovány jedním samostatným modulem, a posléze předávány jako parametry na vstup generátoru jízdních řádů i simulátoru; přitom výstup simulátoru je jediným vstupem hodnotiče. Chování cestujících je jedním ze dvou jevů, o němž jsou v průběhu simulace sbírána data, s tím, že druhým jevem je vytíženost spojů.

Jediné parametry chování generátoru jízdních řádů jsou požadavky cestujících a topologie sítě (k níž patří i údaj o maximálním počtu spojů, které lze zároveň nasadit). Požadavky cestujících také významně ovlivňují výstup simulátoru, který je jediným vstupem hodnotiče jízdních řádů. Tím je naplněn záměr projektu, postavit celý systém tak, aby v první řadě uspokojoval požadavky cestujících.

Cíl: „Vytvořit metriku, která při hodnocení jízdního řádu bude zohledňovat zejména jeho soulad s potřebami cestujících.“

Realizace: Takových metrik bylo vytvořeno několik — viz Experimentální část 4. Vstupem funkce počítající metriku je simulační kalendář, který obsahuje výhradně záznamy o chování cestujících (buď přímo, nebo nepřímo jako údaje o naplněnosti spojů). Ukázalo se, že funkce se chovají dvěma způsoby: buď nejlépe hodnotí jízdní řád, který se snaží rovnoměrně uspokojit všechny cestující - i za cenu, že jsou cestující obslouženi rovnoměrně nekvalitně - nebo nejlépe hodnotí jízdní řád, který má co nejvíc cestujících obsloužených optimálně. Tím se otázka „optimality“ jízdního řádu stává subjektivní a otázkou osobní preference. Druhý jmenovaný přístup v praxi těžko obstojí, proto jako finální metrika byla vybrána jedna z metrik prvního typu.

Cíl: „Vyzkoušet genetický algoritmus jako nástroj k hledání optimálního jízdního řádu.“

Realizace: Zde se ukázalo, že od sebe nelze oddělit problém vývoje metriky, vývoje reprezentace jízdního řádu genomem a vývoje modelů dopravní sítě. Tyto problémy jsou spolu propojené a každý z nich přispívá do výsledku. Dá se říci, že genetický algoritmus vždy konvergoval k rozumnému řešení; zda toto řešení bylo optimální je možné intuitivně rozhodnout pouze v případech triviální dopravní sítě.

Cíl: „Provést experimenty s genetickým algoritmem a jejich výsledky využít k zhodnocení přínosu této metody pro generování JŘ.“

Realizace: O realizaci tohoto cíle pojednává celá kapitola Experimentální část4.

Cíl: „Vytvořit zázemí pro „třetí strany“, aby mohly vytvářet vlastní implementace modulů.“

Realizace: Byly vytvořeny vývojářské balíčky a vývojář dříve nezasvěcený do projektu byl požádán, aby zkusil vytvořit triviální implementaci každého z nich.

Webové stránky fungují, běží na systému Microsoft Azure na adrese `http://optimetable.azurewebsites.net` (jejich grafickou podobu vytvořil Bc. Aleš Zenáhlík, grafika proto není součástí práce).

Nezdary

Genetický algoritmus jako jeden z ústředních bodů práce neposkytoval jednoznačná řešení pro hledání optimálního jízdního řádu. Možné důvody na straně teorie:

- Problematická reprezentace jízdního řádu genomem
  - Některé reprezentace byly nepřesné - převod genomu na jízdní řád obsahuje různé harmonizace a zaokrouhlení, která ztěžují „ruční“ výpočet jízdního řádu z genomu a jsou zdrojem potenciálních chyb
  - Jiné reprezentace jsou redundantní – ten samý jízdní řád je možné vyjádřit velkým množstvím různých genomů
  - Obecně žádná vyzkoušená reprezentace příliš dobře nezachovávala optimalitu, která se křížením vytrácela; toto vyřešilo až použití elitismu
- Relativita otázky, který jízdní řád je optimální: zkoušené metriky se rozdělily na dva typy
  - Jeden typ lépe hodnotil maximální plošné rozdělení vozidel mezi linky tak, že všechny měly stejně nedostatečně, ale žádná nebyla zcela bez obsluhy
  - Druhý typ lépe hodnotil maximální množství optimálně obsloužených cestujících, kdy některé linky mohly být obsluhovány tak, že na nich cestující nečekali vůbec, na úkor jiných, na nichž třeba nejezdilo jediné vozidlo

Reflexe

V souhrnu můžeme říct, že všechny cíle byly splněny, a výstupem práce je produkt, který lze dále rozšiřovat. Náměty pro rozšíření jsou například tyto:

- Podpora víceúrovňových dopravních sítí
- Hlubší průzkum problematiky generování cestujících (například možnosti pořizování dat přímo od cestujících nebo modelování pomocí demografických dat)
- Srovnání výsledků generování jízních řádů genetickým algoritmem s výsledky jiného algoritmu (či více algoritmů)

Seznam použité literatury

[1] Mareš, Martin. *Skripta k Algoritmům a datovým strukturám.* http://mj.ucw.cz/vyuka/1011/ads1/3-grafy.pdf

[2] Goldberg, David E. *Genetic Algorithms in Search, Optimization & Machine Learning*
Addison-Wesley Publishing Company
ISBN 0-201-15767-5

[3] Holub, Štěpán. *Charaktery a Gaussovy součty* http://www.karlin.mff.cuni.cz/~holub/soubory/Charaktery.pdf

[4] Wren, Anthony. *Computer Scheduling of Public Transport* http://www.karlin.mff.cuni.cz/~holub/soubory/Charaktery.pdf

[5] Voss, Stefan, Daduna, Joachim R. *Computer-Aided Scheduling of Public Transport*
http://www.springer.com/business+%26+management/operations+research/book/978-3-540-42243-3
http://books.google.es/books?id=RsvzwUiUBIsC

[6] Brandani, V., Cataoli, G., ORSI, G., Toni, P. *Bus Scheduling Program Development for A.T.A.F., Florence*
http://trid.trb.org/view.aspx?id=297066

[7] Wikipedia *Demand Responsive Transport* http://en.wikipedia.org/wiki/Demand_responsive_transport

Seznam tabulek

Seznam příloh

Všechny níže zmíněné přílohy jsou uložené na přiloženém CD. Všechny adresy souborů jsou proto myšleny vzhledem ke kořenovému adresáři CD.

Ve složce Software/Sources/OptimeTable se nachází kompletní implementace systému OptimeTable, včetně genetického algoritmu a všech ostatních součástí.

Ve složce Software/Sources/SimpleImplementations jsou všechny ukázkové implementace.

Ve složce Software/Sources/Documentation je vygenerovaná dokumentace k implementaci systému OptimeTable.

Ve složce Software/Program je zkompilovaný program s instrukcemi ke spuštění (ty vychází z uživatelské dokumentace této práce, ale konkrétněji popisují, jak spustit program s různými topologiemi).

Následující, číslované přílohy jsou odkazované z textu práce a jsou uloženy ve složce Thesis/Attachments, která je považována za výchozí pro níže uvedené adresy souborů.

Příloha 1: LGSAWRTest.csv

Příloha 2: 4.2.2/model10_200_30.png

Příloha 3: 4.2.2/Playground_10_200_30(4).csv

Příloha 4: 4.2.2/model12_400_50.png

Příloha 5: 4.2.2/Playground_12_400_50(4).csv

Příloha 6: 4.2.2/model12_600_50.png

Příloha 7: 4.2.2/Playground_12_600_50(4).csv

Příloha 8: tables/4.1

Příloha 9: tables/4.2

Příloha 10: tables/4.3

Příloha 11: tables/4.4

Příloha 12: tables/4.5

Příloha 13: tables/4.6

Příloha 14: tables/4.7

Příloha 15: tables/4.8

Příloha 16: tables/4.9

Příloha 17: tables/4.10

Příloha 18: tables/4.11

Příloha 19: tables/4.12

Příloha 20: tables/4.13

Příloha 21: Praha

Příloha 22: Trapeze_SolutionSheet-FXBB_FIN_06192013.pdf

Příloha 23: OptimeTable.bak