VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

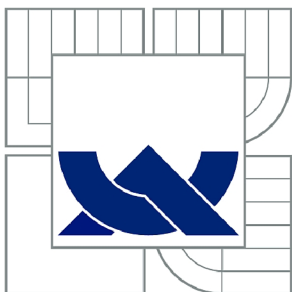
INTELIGENTNÍ ŘÍZENÍ MĚSTA ZA POMOCÍ TELEMETRICKÝCH SÍTÍ

SEMESTRÁLNÍ PRÁCE
SEMESTRAL THESIS

AUTOR PRÁCE
AUTHOR

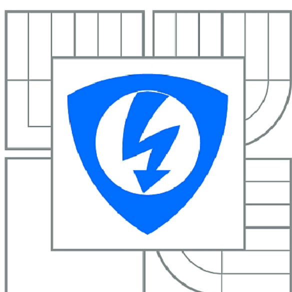
Bc. TOMÁŠ PETRÁK

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

INTELIGENTNÍ ŘÍZENÍ MĚSTA ZA POMOCÍ TELEMETRICKÝCH SÍTÍ

INTELLIGENT CITY MANAGEMENT BY TELEMETRY NETWORKS

SEMESTRÁLNÍ PRÁCE

SEMESTRAL THESIS

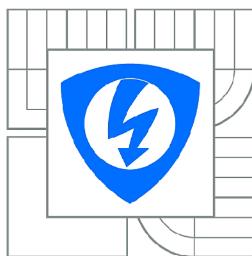
AUTOR PRÁCE Bc. TOMÁŠ PETRÁK

AUTHOR

VEDOUCÍ PRÁCE Ing. RADEK FUJDIAK

SUPERVISOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Semestrální práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Tomáš Petrák	***ID:***	125588
Ročník:	2	***Akademický rok:***	2013/2014

NÁZEV TÉMATU:

Inteligentní řízení města za pomocí telemetrických sítí

POKYNY PRO VYPRACOVÁNÍ:

Úkolem studenta bude orientace v otázkách inteligentního řízení města a dnešních použitelných možností z oblasti telemetrie, specifických pro tuto problematiku. Student se v této části do hloubky zorientuje v problematice a výstupem této části bude kvalitní teoretický základ. V návaznosti na vypracovanou teoretickou část si student vybere konkrétních oblastí řízení, podrobněji rozebere její možnosti a navrhne vlastní způsob (systém) řízení. Následně naprogramuje simulační prostředí, pomocí kterého student ověří svoje navržený systém (ověření bude probíhat na studentem vybraném městě či městské části) a výsledky poté budou v rámci práce diskutovány a porovnány se stávajícími řešeními.

DOPORUČENÁ LITERATURA:

Columbia University. Energy Management in New Yourk City Public Housing. Květen 2009.
ICMA. What is City Management (Conference in San Jose). 2011.
ADELAIDE, C.C. Energy Management Action Plan 2011 - 2014. Květen 2011.
SESAC. Guidelines for sustainable energy management. Květen 2011.

Termín zadání:	11.10.2013	***Termín odevzdání:***	2.1.2014

Vedoucí práce: Ing. Radek Fujdiak
Konzultanti semestrální práce:

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Autor semestrální práce nesmí při vytváření semestrální práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Práce pojednává o řízení města pomocí telemetrických sítě. Je představena problematika telemetrických sítí a multiagentních systémů. V práci je navržen model dopravy v Javě, který umožňuje simulovat a vyhodnocovat konfigurace semaforů v městském provozu.

KLÍČOVÁ SLOVA

Telemetrické sítě, multiagentní systémy, Java, řízení dopravy, optimalizace, genetický algoritmus

ABSTRACT

The thesis is dealing with traffic management using telemetry networks. The problematic of telemetry networks and multiagent systems. A simulation model is proposed in Java which enables configuration simulation and assessment.

KEYWORDS

Telemetry networks, multiagent systems, Java, traffic controlling, optimaliazation, genetic algorithm

PETRÁK, Tomáš *Inteligentní řízení města za pomocí telemetrických sítí*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2013. 56 s. Vedoucí práce byl Ing. Radek Fujdiak

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Inteligentní řízení města za pomocí telemetrických sítí“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno . . . .

(podpis autora)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Radku Fujdiakovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále bych také rád poděkoval Ing. Radimu Burgetovi, Ph.D. za konzulace při návrhu programu.

Brno ....

(podpis autora)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....
(podpis autora)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OBSAH

Úvod **11**

1 Zadání **12**

1.1 Požadavky města Adelaide . . . 12

1.2 Současná řešení . . . 16

2 Telemetrické sítě **17**

3 Multiagentní systémy **18**

3.1 Požadavky na multiagentní systémy . . . 18

4 Volba programovacího jazyka **19**

4.1 Typy programovacích jazyků . . . 19

4.2 Srovnání programovacích jazyků . . . 20

4.3 Java . . . 21

5 Optimalizace **22**

5.1 Telemetrie v dopravě . . . 22

5.2 Genetický algoritmus . . . 23

6 Implementace **24**

6.1 Rozdělení na objekty . . . 24

6.2 Třída Car . . . 25

6.3 Třída Road . . . 27

6.4 Třída CrossRoad . . . 28

6.5 Simulace diskrétních událostí v Javě . . . 30

6.6 Třída Event . . . 33

6.7 Třída EventSwitchLights . . . 34

6.8 Třída EventMoveCar . . . 35

6.9 Třída DiscreteEventSimulation . . . 36

6.10 Příklad simulace . . . 37

6.11 Třída Simulation . . . 38

6.12 Implementace optimalizace . . . 40

6.13 Třída Optimization . . . 42

7 Výsledná Simulace **43**

7.1 Nastavení parametrů . . . 43

7.2 Srovnání dynamického a statického systému . . . 46

8 Závěr **47**

Literatura **48**

Seznam příloh **51**

A Výstup programu **52**

A.1 Výpis z programu pro zvolenou konfiguraci 52

A.2 Zdrojový kód 56

SEZNAM OBRÁZKŮ

1.1 Město Adelaide, zdroj: GoogleMaps 13
1.2 Tlačítko pro chodce, Henley Beach 14
1.3 Vybrané křižovatky pro simulaci, zdroj: GoogleMaps 15
5.1 Schéma genetického algoritmu 23
6.1 UML diagram základních tříd 24
6.2 UML diagram třídy Car 25
6.3 UML diagram třídy Road 27
6.4 UML diagram třídy CrossRoad 28
6.5 UML diagram abstraktní třídy Event 32
6.6 UML diagram třídy EventSwitchLights 34
6.7 UML diagram třídy EventMoveCar 35
6.8 UML diagram třídy DiscreteEventSimulation 36
7.1 Simulovaná situace 43
7.2 Vývoj dopravy ve městě Adelaide 44

SEZNAM TABULEK

2.1 Procento lidí, kteří nepovažují dopravní situaci v centru Adelaide za plynulou [7] 17
4.1 Nejpoužívanější programovací jazyky [15] 20
6.1 Srovnání ArrayList a LinkedList 26
7.1 Dopolední doprava ve městě Adelaide 45
7.2 Odpolední doprava ve městě Adelaide 45
7.3 Porovnání statického a dynamického nastavování semaforů 46

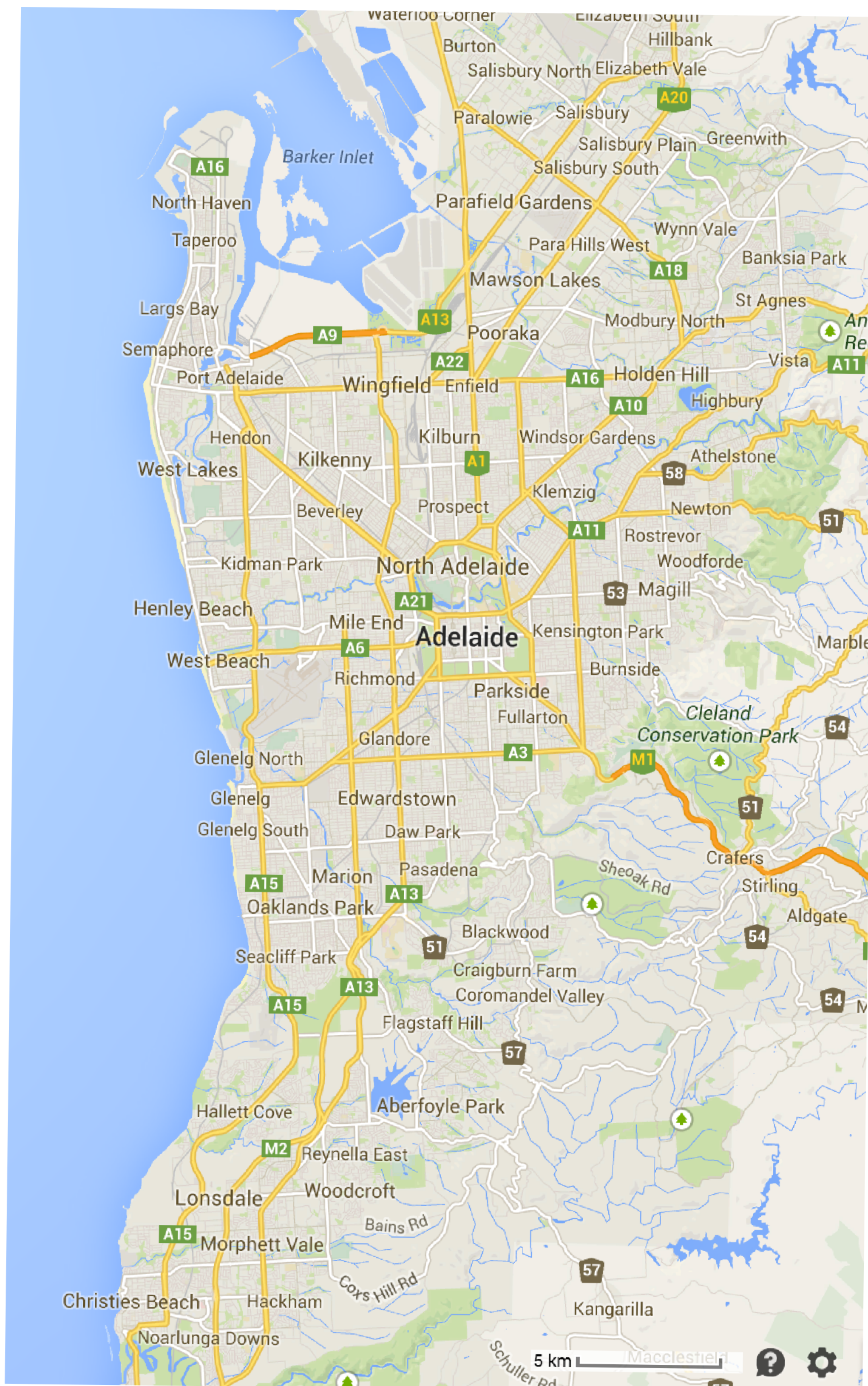
ÚVOD

V době rostoucí urbanizace, zvyšujícího se počtu výškových budov v centrech měst a počtu provozovaných vozidel rapidně stoupá zátěž dopravní sítě ve městech, zvláště pak v jejich centrech. Zácpy způsobují mnohé problémy - hluk, znečištění a snižování dopravní dostupnosti, což snižuje konkurenceschopnost dané oblasti [1].
Města se snaží řešit tuto situaci stavěním obchvatů, tunelů a rozšiřováním stávajících cest. Všechna tato řešení jsou však velmi nákladná a mají své limity, kterých bylo v mnoha případech již dosaženo - stavění obchvatů v zalidněných oblastech je téměř nemožné, stejně tak rozšiřování stávajících cest, stavění tunelů je finančně velice náročné, navíc podloží v některých oblastech jejich stavbu prakticky neumožňuje [2].
První optimalizace dopravy přišla v podobě „zelených vln“, kde byly synchronizovány křižovatky na hlavních tazích tak, aby byl průjezdný celý okruh bez čekání na semaforech. S rostoucí výpočetní silou počítačů a sofistikovaností programovacích jazyků vznikaly nové a nové modely na optimalizaci dopravy. Většina měst má v dnešní době poměrně dobře staticky optimalizovanou dopravu. Dalším krokem v optimalizaci budou telemetrické sítě, do jejichž výzkumu se ročně investují stovky milionů dolarů [8]. Telemetrické sítě umožňují dynamicky optimalizovat dopravu dle aktuálního stavu.
Cílem této práce je navrhnout systém, který umožní simulaci dynamického systému provozu (tj. takového, kde se nastavení jednotlivých semaforů přizpůsobuje dané situaci). Dále je cílem srovnat plynulost dopravy ve staticky a dynamicky optimalizovaném systému.

1 ZADÁNÍ

1.1 Požadavky města Adelaide

Jeden z bodů zadání bylo vytvořit reálnou simulaci (tj. na základě reálných ulic a údajů). Vzhledem k mému ročnímu studiu ve městě Adelaide v Austrálii bylo toto město zvoleno jako modelový příklad pro vytvoření simulace.
Adelaide se poměrně výrazně liší od evropských měst. S 1,3 miliony obyvatel zde žije jenom o pár tisíc lidí více než v Praze, ovšem Praha se rozkládá na 496 km^2, kdežto Adelaide se rozkládá na ploše 1827 km^2 [3].
Na mapě si lze všimnout asymetrického tvaru města - Adelaide leží, stejně jako všechna velká australská města, na pobřeží, navíc je z východu ohraničeno Adelaide Hills (kopcovitou oblastí), což ve výsledku způsobuje „natažení“ města podél pobřeží. Žluté silnice na mapě jsou hlavní tahy - až na výjimky se jedná o 4-proudé silnice, které ohraničují jednotlivé čtvrti města. Tyto cesty vně město pak přecházejí v dálnice.
Menší hustota zalidnění a malý počet lidí žijící v okolních městech (v 500 kilometrovém okruhu okolo Adelaide žije pouze 250 tisíc lidí [4]) vytváří charakteristickou dopravní situaci - lidé dojíždějí do města z větší vzdálenosti, a jsou tak více vytíženy hlavní tahy (dálnice).

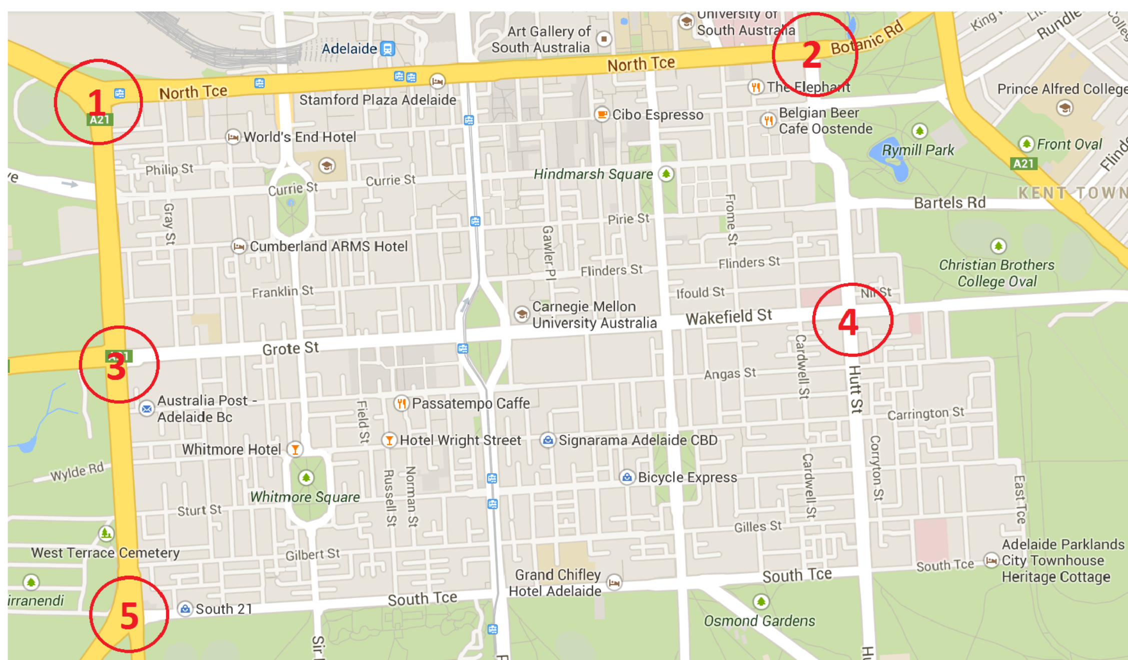


Obr. 1.1: Město Adelaide, zdroj: GoogleMaps

Díky menší hustotě zalidnění pak nedochází k přílišnému vytížení vedlejších tahů mimo centrum. Zde proto k plynulému chodu dopravy stačí přechody, které se přepínají pouze po stisku tlačítka na semaforu. Toto jednoduché řešení je implementováno na všech přechodech ve městě. Zastupitelé města považují řešení v podobě tlačítka pro chodce za dostatečné. Některé čtvrti mimo centrum jsem navštívil a řešení se mi opravdu zdálo dostačující a v těchto čvtrtích by optimalizace dopravy pomocí telemetrických sítí zvýšila plynulost provozu jenom minimálně.
Kde dostačující není, je centrum města. I přesto, že Adelaide má díky své nedlouhé historii široké ulice a silnice s více pruhy, se v centru tvoří zácpy. Dopravní špička je mezi 8 a 9 hodinou ranní a 4 a 5 odpoledne. 4 proudé silnice vedou až na okraj centra, není proto překvapivé, že největší zácpy vznikají v centru, které se nemůže vyrovnat s náporem aut přijíždějícím z šesti hlavních tahů.

Obr. 1.2: Tlačítko pro chodce, Henley Beach

Adelaide Council souhlasil s poskytnutím dat a výběrem křižovatek, které vhodně reprezentují nejvíce vytíženou oblast městkého provozu. Bylo překvapivé, kolik času zabrala diskuze a výběr vhodných křižovatek k simulaci. Adelaide totiž provádí sčíání vozidel na všech křižovatkách v centru (jednou ročně). Všechna data jsou tedy dostupná, bohužel celý dokument v rámci ochrany dat nelze publikovat.
Poznámka: zastupitelé města souhlasili s použitím konkrétních čísel v práci (počtu projížďějících aut vybranými křižovatkami), ne však s publikací celého dokumentu o všech křižovatkách.
Cílem této analýzy měla být úspora, kterou by představovala případná instalace telemetrického systému ve vybrané oblasti. Existují analýzy, jakou ztrátu městu způsobují dopravní zácpy, respektive čekání vozidel. Pokud by takový systém měl mít smysl, tak by měl uspořit více, než bude jeho cena. Pro případné výběrové řízení je proto analýza nezbytná.
Simulované řešení bude využívat telemetrické sítě, do jejichž vývoje se investují ročně stovky milionů dolarů [8], ale které zatím nejsou příliš rozšířené. Jedná se tedy technologii pro města budoucnosti (tzv. future cities). Současná řešení jsou představena v následující kapitole.



Obr. 1.3: Vybrané křižovatky pro simulaci, zdroj: GoogleMaps

1.2 Současná řešení

Většina optimalizačních řešení v dopravě v současné době používá manuálně získaná data, která jsou dále použita jsou vstup počítačové simulace. V této oblasti bylo provedeno mnoho výzkumu a existuje mnoho kvalitních simulačních programů. Navrhovat systém, který by optimalizoval dopravu staticky (tj. jedno nastavení křižovatek po celý den) by nemělo moc smysl, vzhledem k počtu a kvalitě již existujících programů. Například společnost Siemens má celý tým odborníků pracující na této problematice [5].

Tato řešení používají manuální počítání vozidel - jednou ročně se během jednoho týdne 4x vypraví najatý pracovník na danou křižovatku a po dobu 2 hodin zaznamenává počet projíždějících aut [6].

Takovýto přístup má mnoho nedostatků - data jsou sbíráná pouze po relativně krátký časový úsek, a tak objektivně neprezetují dopravní situaci, počítání neprobíhá na všech křižovatkách ve městě a také je drahé - ve státě Jižní Austrálie je minimální mzda 16,75 (australského) dolaru, což při současném přepočtovém kurzu představuje přibližně 312 Kč na hodinu. Další náklady představuje zpracování dat, které také probíhá manuálně.

Se zvyšující se cenou pracovní síly se stále výhodnějším stává plně automatické řešení. Takové řešení navíc dokáže, na rozdíl od manuální počítání vozidel, poskytnout data o počtu vozidel v kterémkoliv okamžiku, která následně mohou být použita pro optimalizaci systému. Systém používající manuálně získaná data nereflektuje a ani nemůže reflektovat dopravní situaci v daném momentě.

Takový systém pracuje s průměrným počtem aut projíždějících danou křižovatkou. Počet aut projíždějící daným úsekem ovšem není konstantní - v ranní a odpolední dopravní špičce projíždí křižovatkami mnohem více vozidel než ve večerním hodinách, jiný je počet projíždějícíh vozidel při slunečném, deštivém nebo zimním počasí, při výstavách ve městě, uzavírkách, státních svátcích, atd.

I sebelépe staticky optimalizovaný systém neumožňuje reflektovat všechny výše zmíněné faktory. Jediný způsob, jak vybudovat systém, který je vždy optimální, je dynamicky optimalizovaný systém se zpětnou vazbou.

2 TELEMETRICKÉ SÍTĚ

Telemetrie je definována jako vysoce automatizovaný komunikační proces, který provádí měření, sbírá data a posílá je centrální entitě. Disciplína jako taková vznikla již v 19. století, kdy byly na Mont Blanc umístěny senzory na měření hloubky sněhu a informace byly přenášeny do měřícího střediska v Paříži v reálném čase [8].
Většinou se jedná o sběr a distribuci dat pomocí bezdrátových sítí, vzhledem k jednoduchosti přenosu informací přes bezdrátovou síť v porovnání s nutností instalace kabelové infrastruktury. Největší rozvoj telemetrie také nastal s rozvojem bezdrátových sítí - mnohé senzory (např. na monitorování počasí) využívají všudypřítomnou GSM síť, která umožňuje sběr dat i v dříve nepřístupných lokalitách. Tyto sítě našly uplatnění téměř ve všech odvětvích od zemědělství po výpravy do vesmíru.
V základní telemetrickou síť tvoří jedná o senzor (např. teplotní nebo tlakový), přenosové zařízení a kontrolní středisko sbírající informace. Tyto systémy slouží buď k monitorování (většinou počasí), k varování (seismické senzory) nebo regulace v reálném čase (čerpadla a dopravní systémy). V případě dopravních systémů se jedná o senzory monitorující dopravu, které jsou napojeny na síť řídící semafory. Ze získaných informací (počtu vozidel na jednotlivých křižovatkách) se potom nastavuje délka periody, po kterou vozidla mohou projíždět křižovatkou, a čas, ve kterém se křižovatka přepne. Křižovatky tvoří tzv. multiagentní systém, který je popsán v následující kapitole.

Tab. 2.1: Procento lidí, kteří nepovažují dopravní situaci v centru Adelaide za plynulou [7]

Rok	Procento
1995	27
1997	29
1999	26
2001	35
2003	40
2005	43
2007	45
2009	44
2011	48
2013	49

3 MULTIAGENTNÍ SYSTÉMY

3.1 Požadavky na multiagentní systémy

Multiagentní systémy jsou systémy s více spolupracujícími počítačovými prvky. Agenti musí splňovat 2 základní požadavky:

- Musí být schopni do jisté míry být autonomní, tj. činit rozhodnutí bez vnějšího zásahu tak, aby byli schopni splnit požadavky na ně kladené.
- Musí být schopni spolupracovat s ostatními agenty.

Multiagentní systémy jsou disciplínou, která se začala formovat až kolem roku 1980 [9]. K jejich hlavnímu rozvoji došlo s počátkem internetu, kde síť agentů představovala možnost, jak simulovat síť autonomních, spolupracujících jednotek (routerů). Podobně jako síť routerů lze simulovat dopravu. Mezi těmito systémy lze najít mnoho paralel – router směruje pakety, které hromadí do front v podobě zásobníku, křižovatka „směruje“ auta, která se hromadí na příjezdové cestě. V případě výpadku některé z linek se hledá alternativní cesta, kterou můžou pakety použít k dosažení cílové adresy, podobně v případě objížděk se hledá v dopravě alternativní cesta. Bandwidth odpovídá kapacitě cesty (počet aut, která mohou daným úsekem projet za hodinu) [10].

Integrace multiagentních systémů se skládá ze 2 základních úkolů – vytvoření individuálního agenta, který bude schopen reprezentovat danou entitu, a vytvoření kolekce těchto agentů, tj. sítě schopné komunikovat a vyměňovat si data.

Celý problém bude reprezentován ve vybraném programovacím jazyku - agenti budou reprezentování třídou, podobně tak auta. Volba jazyka a popis způsobu implementace jsou diskutovány v následující kapitole.

4 VOLBA PROGRAMOVACÍHO JAZYKA

4.1 Typy programovacích jazyků

Vzhledem k nutnosti reprezentovat daného agenta třídou, která je objektem, bude volba zúžena na objektově orientované programovací jazyky. Simulaci by bylo možné implementovat i v jazyce C, implementace bez použití objektů by však byla velice (a zbytečně) náročná. Procedurální jazyky jsou určeny spíše k řešení algoritmických problémů [11]. Bez možnosti vytvořit si vlastní třídu reprezentující agenta by bylo celý problém bylo nutné řešit pouze pomocí funkcí.
Podobně nevhodnými se jeví skriptovací programovací jazyky (například JavaScript, Python, atd.). Tyto jazyky jsou určeny spíše k použití u webových služeb vzhledem k možnosti poměrně rychle a pohodlně vytvářet aplikace a vlastnostem jako stabilitě (při vzniku výjimky skriptovací jazyk ukončí provádění dané části skriptu, ale nevyvolá chybové hlášení), slabému typování a možností jako typová konverze, dynamickému měnění rozsahu polí [12]. Jejich nevýhodou je však neúplnost (tj. předpokládá se spolupráce s jiným programovacím jazykem), špatné strukturování, velmi obtížné ladění a jejich jednostrannost – tyto jazyky jsou často velmi specificky zaměřeny a řešení nestandardního problému je v nich velice obtížné a také neefektivní.

4.2 Srovnání programovacích jazyků

Jako nejvhodnější k implementaci multiagentního systému se jeví objektově orientované, neskriptové programovací jazyk – mezi nejrozšířenější takovéto jazyky patří Java, Objective-C, C++, C# a Delphi (viz. tabulka 4.1).
Je žádoucí, aby kód byl lehce rozšířitelný a lehce použitelný v dalších projektech. To nezahrnuje pouze dobrý návrh a dokumentaci, ale také volbu široce používaného programovacího jazyka. 15 nejpoužívanějších programovacích jazyků podle Tiobe:

Tab. 4.1: Nejpoužívanější programovací jazyky [15]

2013	2012	Programovací jazyk	% uživatelů	Změna za poslední rok
1	1	C	17,890%	-0,81%
2	2	Java	17,311%	-0,26%
3	3	Objective-C	10,202%	-0,91%
4	4	C++	8,268%	-0,94%
5	5	C#	5,620%	+0,07%
6	6	PHP	5,281%	-0,26%
7	7	Visual Basic	3,752%	-1,42%
8	8	Python	2,210%	-1,64%
9	21	Perl	1,877%	+1,30%
10	11	JavaScript	1,852%	+0,53%
11	12	Visual Basic .NET	1,264%	+0,13%
12	10	Ruby	1,242%	-0,43%
13	38	F#	1,030%	+0,79%
14	14	Transact-SQL	1,025%	+0,53%
15	17	Delphi	0,974%	+0,24%

4.3 Java

Ze srovnání vychází Java jako nejpoužívanější programovací jazyk s více než dvojnásobným zastoupením v porovnání s druhým jazykem vhodným pro tuto implementaci (`C++`). Objective-C je jazyk používaný hlavně pro vývoj aplikací pro MacOS, tedy pro tento projekt nevyhovující. V porovnání s `C++` má Java několik dalších výhod:

- Java je multiplatformní – kód napsaný v jazyku Java lze zkompilovat na všechny 3 nejvíce zastoupené PC operační systémy (Windows, Mac OS, Linux) a také například na Android. `C++` lze zkompilovat na Windows, ale kompilace systému na jiný operační systém je problematická.
- Automatickou správu paměti – Java automaticky alokuje a dealokuje paměť pro proměnné (objekty), což ji činí mnohem méně náchylnou na tzv. úniky paměti. V jazyku `C++` má správu paměti má na starosti programátor, což má často za následek špatnou správu paměti.
- Pro jazyk Java bylo napsáno více knihoven, což usnadňuje vývoj a celkově rozšiřuje možnosti práce s tímto jazykem.
- Java je distribuovaná - Java je navržena tak, aby distribuované výpočty byly jednoduché.
- Java je bezpečná - konkrétní srovnání bezpečnosti jednotlivých jazyků je mimo rámec této práce, ale Java se používá v mnoha aplikacích, kde je bezpečnost a bezchybovost klíčová - například jako software do bankomatů [13].
- Java je robustní a spolehlivá - v Javě je kladen velký důraz na kontrolu možných chyb a kompiler je schopen odhalit mnoho chyb, které by u některých programovacích jazyků vyšly najevo až spuštění programu [14].
- Java je více vláknová - vláknění programu a využívání potenciálu vícejádrových procesorů, které jsou dnes již normou i u osobních počítačů, je přímo integrováno do Javy, a jeho použití je tak snažší v porovnání s ostatními programovacími jazyky [13].

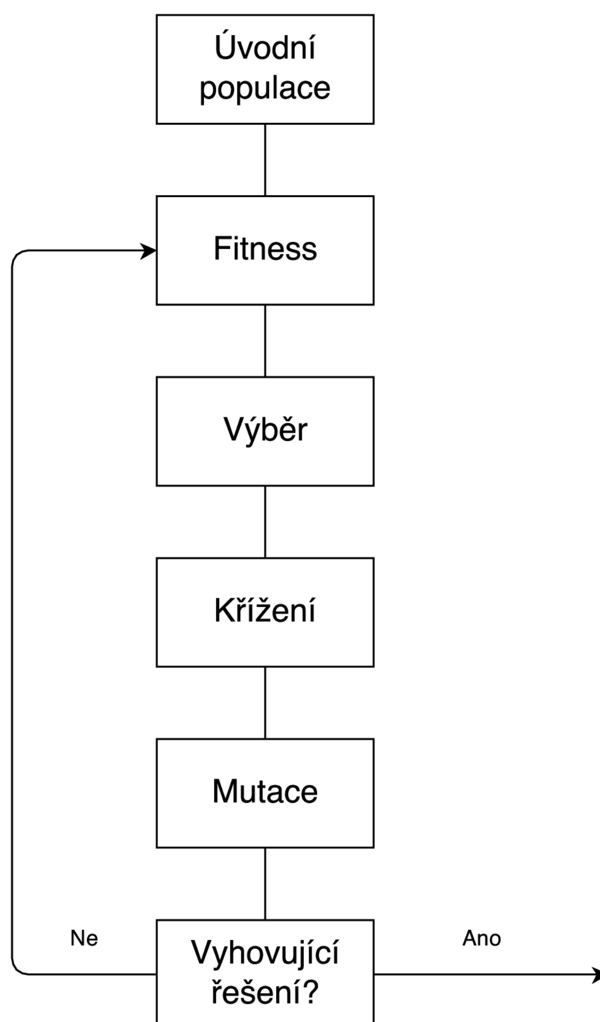
5 OPTIMALIZACE

5.1 Telemetrie v dopravě

Smyslem telemetrických sítí v dopravě není pouze měření provozu, nýbrž i jeho optimalizace. Pouhý sběr dat by byl vhodný k monitorování provozu, jednalo by se však o poměrně neúčelně vynaložené prostředky: v případě, že jsou dostupná data o počtu aut, není pro optimalizaci provozu potřeba další investice do fyzické infrastruktury. Údaj o počtu aut a znalost dopravy ve městě jsou 2 podmínky nutné a dostatečné pro vytvoření optimalizačního modelu (nastavení délky period jednotlivých stavů daného semaforu a jejich vzájemné posunutí). Plynylá optimalizace dopravy je jedním ze základních pilířů tzv. future cities, neboli měst budoucnosti. Analýza výhodnosti takové investice (srovnání celkového zpoždění všech aut) byla to, co zajímalo zastupitele města Adelaide nejvíce. Centrum Adelaide je, vzhledem k pravoúhlému systému ulic, velice vhodné k analýze. Jedná se také oblast s nejvíce vytíženou dopravou a oblast, kde město Adelaide vydává na dopravu miliony dolarů ročně [9].

5.2 Genetický algoritmus

Optimalizační algoritmy jsou používány v případě, že počet všech možných řešení je příliš vysoký na to, aby všechna řešení byla ověřena, a neexistuje heuristika, která by umožňovala konvergenci k optimálnímu/ vyhovujícímu řešení v proveditelném množství iterací.
Genetický algoritmus není název jednoho konkrétního algoritmu - jedná se o celou skupinu algoritmů, které se snaží napodobovat přírodní proces evoluce a množení. Jsou použity 4 základní techniky - dědičnost, mutace, výběr a křížení. V případě genetického algoritmu se jedná o lokální vyhledávání, které na základě řetězců (stringů, polí) kombinovaných pomocí fitness funkce produkuje řešení, která se blíží hledanému (optimálnímu) řešení s danou přesností. Celý průběh optimalizace je zobrazen na následujícím obrázku.

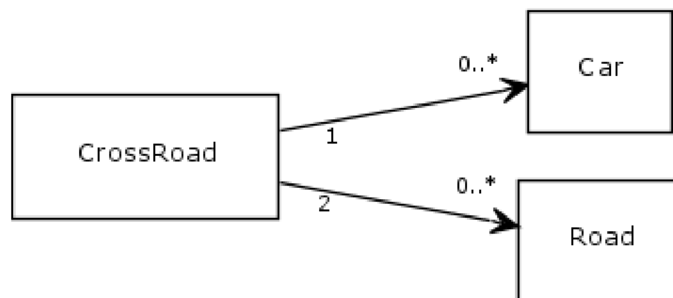


Obr. 5.1: Schéma genetického algoritmu

6 IMPLEMENTACE

6.1 Rozdělení na objekty

Výhodou objektově orientovaného programování je možnost reprezentovat objekty reálného světa třídami. Jedním ze základních úkolů návrhu systému bude rozdělení problému na objekty, které lze pomocí tříd reprezentovat v programovacím jazyku.
Pozn.: Vzhledem k tomu, že simulace je vytvářeno pro město Adelaide, které se nachází v Austrálii, kde převládajícím jazykem je angličtina, bude v kódu pro názvy proměnných a tříd použita angličtina.
Prvním objektem celého návrhu bude třída (agent) reprezentující křižovatku. Tato třída bude obsahovat množinu cest, které spojují křižovatku s ostatními křižovatkami, seznam aut čekajících na dané křižovatce, stavem semaforu (červená nebo zelená) a jménem křižovatky pro její jasnou identifikaci. UML Diagram: Každá kři-



Obr. 6.1: UML diagram základních tříd

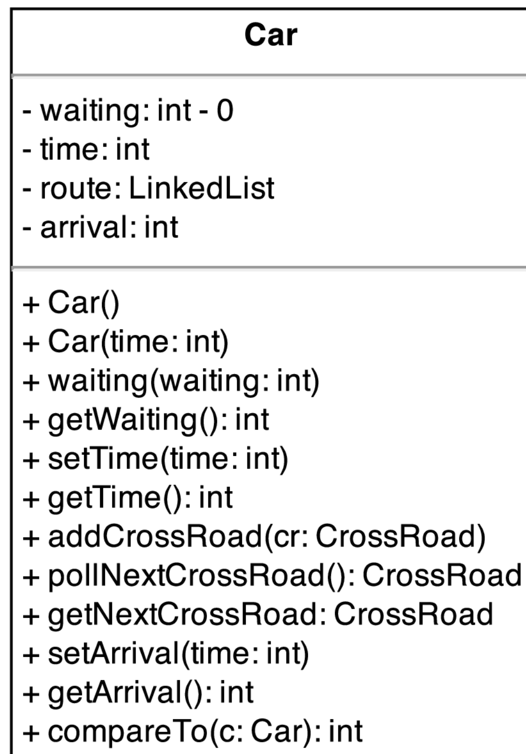
žovatka může mít libovolný počet čekajících aut a příjezdových cest. Každá cesta bude spojovat právě 2 křižovatky a každé auto bude čekat právě na jedné křižovatce. Před implementací třídy CrossRoad (reprezentující křižovatku) je třeba implementovat třídy Car (reprezentující auto) a třídu Road (reprezentující příjezdovou cestu ke křižovatce).

6.2 Třída Car

Cílem třídy Car je reprezentovat auto pohybující se sítí křižovatek. Třída bude obsahovat 4 privátní proměnné:

- time – čas, ve kterém se auto nachází (pro použití v simulaci událostí s diskrétním časem)
- waiting – celkový čas, který stráví auto čekáním. Součet časů čekání všech aut bude použit jako hlavní ukazatel při vyhodnocování nastavení semaforů
- route – trasa, po které se bude auto pohybovat systémem.
- arrival - čas příjezdu na danou křižovatku (použito ve výpočtu zpoždění)

Existuje více možností, jak implementovat interface List (například ArrayList). LinkedList má však některé žádoucí vlastnosti pro tuto implementaci (viz tabulka 2). Vzhledem k tomu, že seznam křižovatek bude postupně plněn (tj. prvek bude vždy přidáván na konec) a bude z něho postupně odebíráno, tj. z listu bude odebrán první prvek, budou tyto operace prováděny s náročností O(1), v porovnání s ArrayListem, kde je náročnost O(n).



Obr. 6.2: UML diagram třídy Car

Tab. 6.1: Srovnání ArrayList a LinkedList

	LinkedList	ArrayList
get(int index)	O(n)	O(1)
add(E element)	O(1)	O(1)
add(int index, E element)	O(n)	O(n-index)
remove(int index)	O(n)	O(n-index)
Iterator.remove()	O(1)	O(n-index)
ListIterator.add(E element)	O(1)	O(n-index)

Dále třída obsahuje konstruktor Car(), který umožní vytvořit novou instanci třídy s časem a čekáním nastaveným na 0 a prázdným listem křižovatek. Kostruktor Car(int time) umožňuje vytvoření instance auta v určitém čase. Auta budou vytvářena ve for cyklu s použitím třídy Random, což bude simulovat náhodný příjezd aut s určitou frekvencí, která bude reprezentována pravděpodobností, nebo staticky vkládána pro simulaci konkrétní situace. U auta by nebylo žádoucí mít identifikátor (například String), protože by to činilo manipulaci (například vytváření objektu) složitější. Třída auto bude obsahovat metody get a set pro každou proměnnou. Pro zjištění další křižovatky budou použity metody getNextCrossRoad, která zjistí následující křižovatku, avšak neodebere ji z listu, a v případě prázdného listu vrátí null. Tato bude sloužit k počítání aut a ke statistikám popsaným v následujících kapitolách. Metoda pollNextCrossRoad odebere první křižovatku z listu a navrátí ji jako proměnnou typu CrossRoad. V případě prázdného listu vrací null. To je další výhoda LinkedListu – obsahuje metodu pollFirst, která obsluhuje výše zmíněný proces. V případě ArrayListu je potřeba celé řešení naprogramovat. Vzhledem k tomu, že auta budou řazena na křižovatce podle času příjezdu, je potřeba implementovat metodu rozhraní Comparable: public class

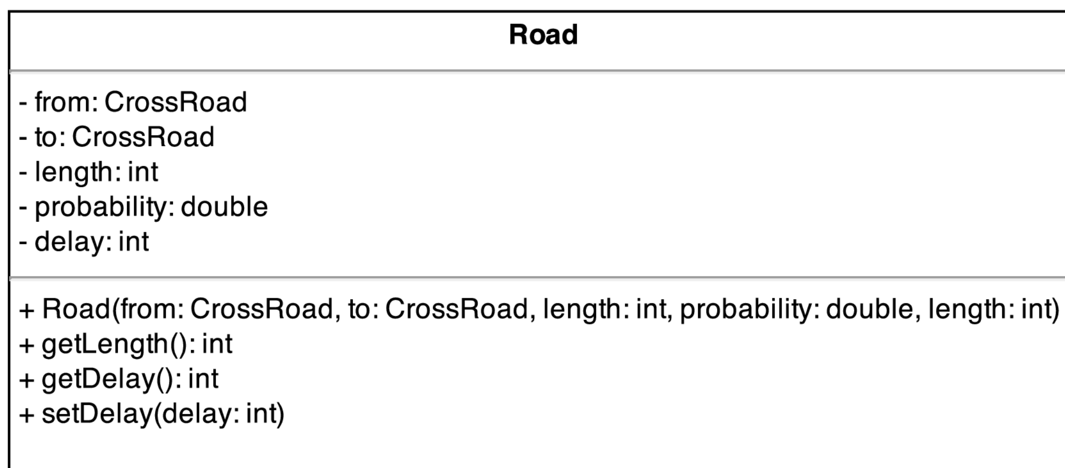
```
Car implements Comparable<Car>
```

Funkce compareTo(Car c) vrací záporné celé číslo v případě, že je příjezdový čas menší než u porovnávaného auta, 0 v případě, že mají obě dvě auta stejný příjezdový čas a kladné číslo v případě, že má auto větší příjezdový čas.
K privátní proměnné nelze standardně přistupovat jinak než přes veřejné metody. V případě implementace Comparable však Java dělá výjimku a umožňuje přímý přístup k dané proměnné pro účely srovnání. Zdánlivě mnohořádkový kód lze tak elegantně zkrátit na pouhý jeden řádek:

```
return new Integer(time).compareTo(new Integer(c.time));
```

6.3 Třída Road

Třída Road reprezentuje cestu spojující jednotlivé křižovatky. Proměnná length obsahuje délku cesty mezi křižovatkami. Tato délka je použita v dalších fázích k výpočtu času, který bude auto potřebovat pro příjezd od jedné křižovatky k druhé. Výpočet bude probíhat při běhu simulace ve třídě DiscreteEventSimulation popsané v následující kapitole. Metoda getLength slouží k přístupu k privátní proměnné length. V mnohých případech je definovat proměnnou jako privátní diskutabilní, vzhledem k tomu, že metody set a get často pouze kopírují hodnoty z jedné proměnné do druhé nebo vracejí danou proměnnou a neprovádějí například žádnou kontrolu v metodě set. V případě třídy Road je však zvláště žádoucí definovat proměnné jako privátní, protože jejich změna za běhu by mohla vyvolat chyby v simulaci. Třída proto ani neobsahuje metodu pro nastavení hodnoty proměnné length.
Tato třída může shromažďovat různé statistické údaje (např. projíždějící vozidla) nebo údaje ovlivňující dopravu - např. zpoždění (v případě nehody, stavebních prací v daném úseku, atd.).

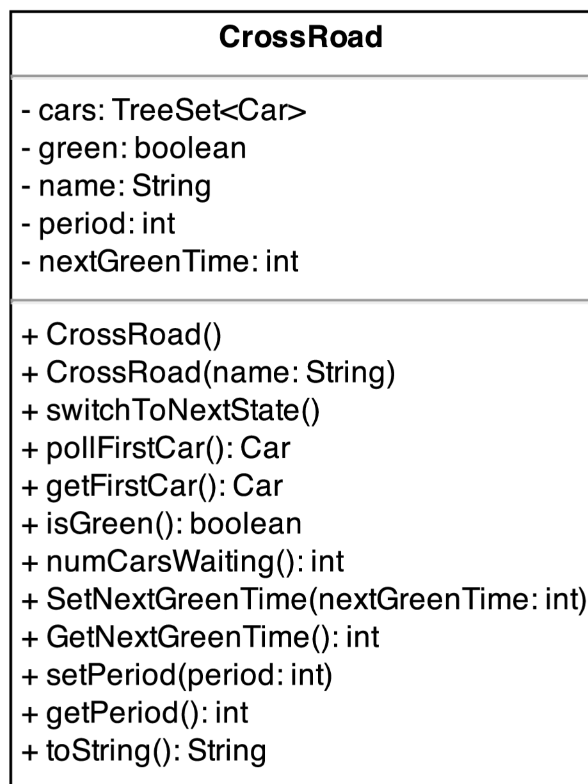


Obr. 6.3: UML diagram třídy Road

6.4 Třída CrossRoad

Po definici tříd Car a Road lze přistoupit k definici třídy reprezentující křižovatku. Auta čekající na křižovatce budou reprezentována proměnnou cars typu TreeSet. TreeSet sice spotřebovává více systémových prostředků než například ArrayList, prvky v TreeSetu jsou ale v kterýkoliv okamžik seřazeny dle velikosti, což je naprosto zásadní pro správné řazení v čase – v každém okamžiku musí být seznam událostí seřazený, aby se nestalo, že budou vykonány v nesprávném pořadí (což by mohlo způsobit cyklení nebo nesouslednost událostí). Ve třídě Car musela být kvůli řazení definována metoda compareTo – aby bylo možné auta jasně porovnat (v tomto případě podle času).
Stav semaforu je reprezentován proměnnou green typu boolean. Pro stav světel by šlo použít i datový typ Integer, popřípadě enumerátor (tj. výčtem stavů, v tomto případě GREEN, RED). Boolean má tu výhodu, že se jedná o proměnnou, která může nabývat pouze 2 stavů (true a false), může být použitá přímo v podmínce if a v případě změny barvy semaforu lze negovat současný stav, tj.:

```
green = !green;
```



Obr. 6.4: UML diagram třídy CrossRoad

Příjezdové cesty jsou reprezentovány ArrayListem, protože se jedná o systémově nejméně náročnou implementaci dynamického pole. Pokud by byl počet příjezdových cest dopředu známý, šlo by reprezentovat příjezdové cesty klasickým polem, které je z hlediska výkonu nejlepší.
U křižovatky je žádoucí mít jasné označení, v tomto případě v podobě proměnné typu String. Jméno křižovatky bude použito ve stavových výpisech. Třída umožňuje použít i konstruktor bez parametru pro lehčí implementaci v kódu. Konstruktor bez parametru lze nedefinovat v případě, že třída neobsahuje konstruktor s parametrem. V takovém případě zdědí třída konstruktor od třídy Object, která je tzv. super třídou, od které dědí všechny objekty v Javě. V případě, že je použit neprázdný konstruktor, je nutné konstruktor bez parametru definovat explicitně v kódu.
Funkce switchToNextState přepíná stav světel. Funkce přiřadí do proměnné green negaci předchozího stavu a vypíše hlášení o přepnutí daného semaforu. Funkce addCar přidá do fronty čekajících aut další auto. Funkce pollFirstCar odebírá první auto z řady. Tato funkce je použita k přiřazení auta na další křižovatku. V případě, že auto nemá další křižovatky, je vráceno null. Funkce getCar funguje stejně, pouze neodebírá auto ze seznamu čekajících aut. Její hlavní použití je při výpisu informací a zjišťování směrů, kterými jednotlivá auta budou pokračovat.
Proměnná nextGreenTime uchovává čas příští zelené, což je údaj nutný pro správné nastavení času odjezdu aut z křižovatky (pokud auto přijede na křižovatku ve chvíli, kdy svítí červená, nemůže „vědět“, na jakou hodnotu nastavit odjezdový čas).

6.5 Simulace diskrétních událostí v Javě

V Javě jsou jednotlivé události modelovány pomocí tříd. Bude potřeba předdefinovat určité vlastnosti všem událostem, aby byla možná jejich obsluha. Společné vlastnosti lze v Javě řešit dvojím způsobem – dědičností a rozhraním. Vzhledem k tomu, že událost (Event) bude sloužit jako předpis, ne konkrétní implementace, bude uvažována pouze abstraktní třída jako možný vzor pro dědičnost. Od abstraktní třídy lze dědit, ale nemůže být vytvořena její instance.

Rozhraní jako takové je používáno jako určitá smlouva, co by měla třída obsahovat, aby dané metody mohly být použity dále v kódu. Nevýhodou rozhraní je, že všechny metody musí být veřejné. Určité metody by měly být privátní, protože jejich nesprávné použití může vyvolat výjimku.

Abstraktní třída poskytuje více struktury a implementace, kterou lze poté použít v podtřídách a není nutné ji pokaždé znovu implementovat. Nevýhodou je to, že některé programy už můžou mít implementovanou nějakou strukturu a v Javě lze dědit pouze od jedné třídy, což může činit implementaci složitou. V tomto případě by takovýto problém neměl nastat vzhledem k tomu, že události budou zpracovávány pouze v hlavním programu a nebudou běžet v jiném vlákně. Jako nejvýhodnější se proto v tomto případě jeví abstraktní třída díky možnosti opětovného použití kódu.

Po definici třídy Event bude možné definovat seznam událostí následovně:

```
TreeSet<Event> events = new TreeSet<>();
```

Pozn.: V novější verzi Javy (7.0) není již nutné specifikovat za klíčovým slovem new typ objektu v listu, protože je implicitní v definici nalevo.

Nyní lze tento list naplňovat libovolnou událostí dědící od třídy Event, což je žádoucí, protože události v systému mají být zpracovány sousledně a nezávisle na typu události. Mít 2 různé listy událostí (což by bylo potřeba ve chvíli, kdy by nebyla definována třída Event) a nějakým způsobem je synchronizovat by bylo velice nešikovné, navíc by lehce mohlo dojít k situaci, že bude událost v čase t + 10 z jednoho listu obsloužena dříve než událost v čase t z listu druhého.
Nyní lze všechny proměnné události držet v jednom listu, ze kterého budou postupně odebírány a zpracovávány. Pokud se budou v simulaci vyskytovat například události EventSwitchLights (přepnutí semaforu), EventWalk (přecházení chodců) a EventMoveCar (jízda auta), lze je všechny přidat do jednoho TreeSetu:

```
TreeSet<Event> events = new Event<>();
events.add(new EventSwitchLights(20));
events.add(new EventWalk(30));
events.add(new EventMoveCar(10));
```

Čísla v kulatých závorkách značí čas, ve kterém má být daná událost vykonána. Události budou vykonány ve správném pořadí, i když událost EventMoveCar přidána do seznamu později než událost EventSwitchLights. Podmínkou jejich správného (a vůbec možného) řazení je opět implementace rozhraní Comparable.
V následném zpracování lze všechny události obsloužit jednotně a bez rozdílu, například ve for cyklu:

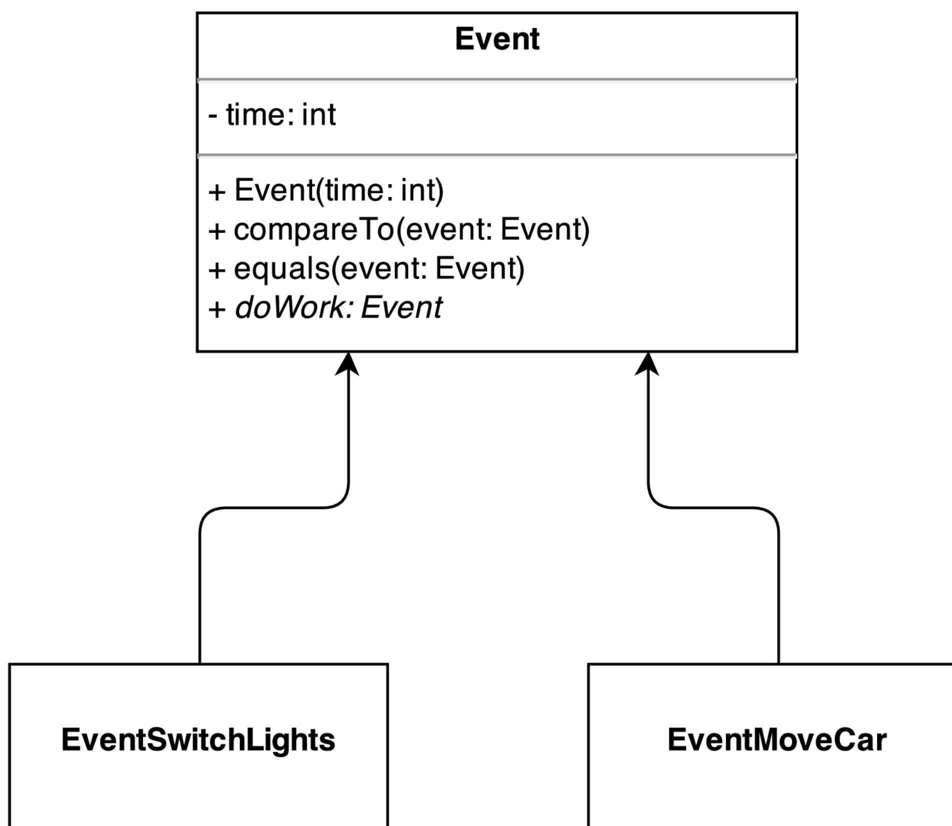
```
for(int i = 0; i < events.size(); i++)
    Events.pollFirst.doWork();
```

Takový for cyklus odebere ze seznamu událostí první událost a vykoná činnost, která je v něm definovaná. Ve třídě Event bude muset být definována metoda `doWork()`, kterou budou mít všechny třídy společnou. Metoda bude definována jako abstraktní, tj. bez těla a její implementace bude ponechána na podtřídách.

Pozn.: I když třídy EventWalk a EventMoveCar nebyly v prvotní fázi implementovány, systém bylo nutné navrhovat tak, aby jejich implementace byla co nejsnazší. Pozdější rozšiřitelnost systému a její pracnost je brána jako důležité kritérium při porovnávání kvality jednotlivých řešení.
Další typy událostí lze přidat bez jakéhokoliv zásahu do kódu pouhým použitín klíčového slova extends v definici nové třídy:

```
public class Walk extends Event
```

Dědičnost je znázorněna na následujícím diagramu:



Obr. 6.5: UML diagram abstraktní třídy Event

6.6 Třída Event

Nejdůležitějším prvkem třídy Event je proměnná time – okamžik, kdy se má daná událost začít vykonávat. Používá se pro řazení událostí a je vlastně základem celého návrhu simulace diskrétních událostí.
Konstruktor bez parametru by v případě třídy Event nedával smysl. Proměnná time není ani inicializována na žádnou hodnotu. Event s chybným časem by mohl způsobit značné problémy v celém systému (například zacyklení). Metoda compareTo je implementací rozhraní Comparable a využívá opět možnosti přímého přístupu k proměnné pro účely srovnávání, i když se jedná o privátní proměnnou.
Metoda doWork je abstraktní, protože každá událost bude vykonávat jinou činnost. Nemělo by smysl ji definovat. Metody definované jako Abstract odvozená třída musí implementovat, eliminuje se tím tedy riziko toho, že programátor zapomene při implementaci tuto metodu definovat a zdědí metodu definovanou v super třídě, která nemusí často pro podtřídu dávat smysl.

6.7 Třída EventSwitchLights

Třída bude obsahovat 2 proměnné - crossRoad a period - křižovatka a perioda, se kterou se bude daná křižovatka přepínat. Je samozřejmě potřebné mít proměnnou time, která slouží k určení času, tak ovšem není definována přímo v třídě, ale je použita v kostruktoru při volání super třídy:

```
public EventSwitchLights(int time, CrossRoad crossRoad, int period) {
        super(time);
```

Už při inicializaci třídy je potřeba nastavit čas další zelené:

```
public EventSwitchLights(int time, CrossRoad crossRoad, int period) {
if (this.crossRoad.isGreen())
    this.crossRoad.setNextGreenTime(time);
else
    this.crossRoad.setNextGreenTime(time + period);
```

Kdyby nebyl čas další zelené nastaven již při vzniku události, mohlo by dojít k tomu, že čas zůstane na hodnotě 0 ještě ve chvíli, kdy už bude potřeba tuto hodnotu znát pro správné nastavení času odjezdu vozidla. Ten by pak byl nastaven na čas 0, což by způsobovalo cyklení.
V metodě doWork jsou přepnuta světla a poté znovu nastavena hodnota další zelené (tuto je nutné udržovat vždy aktuální kvůli nastavení správného odjezdu auta, jedno špatné nastavení vytvoří nekonečnou smyčku generující události).
Na konci je vytvořena nová událost s inkrementací času o jednu periodu:

```
return new EventSwitchLights(getTime() + period, crossRoad, period);
```

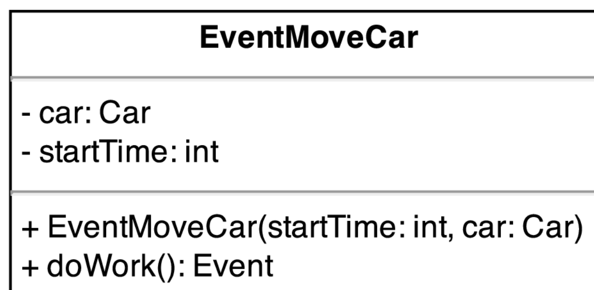
Obr. 6.6: UML diagram třídy EventSwitchLights

6.8 Třída EventMoveCar

Třída opět obsahuje 2 proměnné - auto, u kterého se má pohyb vykonat, a čas, ve kterém se má vykonat. Cílová křižovatka je uložena přímo ve třídě Car, konkrétně v ArrayListu route. V konstruktoru je opět volám konstruktor super třídy. Autu se nastaví nastaví aktuální čas, který je používán při výpočtu zpoždění.
Metoda `doWork` nejprve zkontroluje, jestli je list křižovatek k projetí neprázdný:

```
if (car.getNextCrossRoad() == null)
        return null;
```

V případě, že v listu nejsou další křižovatky k projetí, vrátí se null, event zanikne (auto opouští systém). Čekání je pořád uloženo v dané instanci třídy Car, ta existuje nezávisle na třídě EventMoveCar.
Implementace optimalizace mohou být různé, a proto je vždy kontrolována perioda (nesmí být nulové, jinak by způsobovala cyklení programu). V případě, že je křižovatka červená, odjezdový čas je nastaven na čas další zelené. Zde je vidět potřeba 2 metod get a poll, protože kdyby při zjišťování stavu byla křižoatka odebrána, autu by se zkrátil průjezd systémem a simulace by nefungovala správně.
V případě, že na křižovatce svítí zelená, auto se posouvá na další křižovatku.



Obr. 6.7: UML diagram třídy EventMoveCar

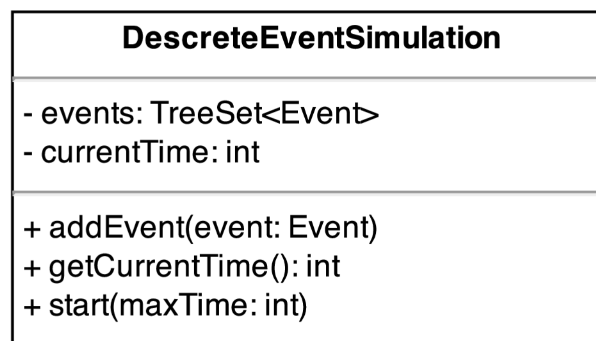
6.9 Třída DiscreteEventSimulation

V této třídě jsou zpracovávány všechny události. Události jsou opět uloženy v TreeSetu kvůli nutnosti jejich seřazení. Metoda addEvent přidává událost do proměnné events. Zde může nastat problém v případě 2 událostí, které budou mít stejný čas. Více v kapitole Rozšíření do diplomové práce. V metodě start se provádí nekonečná smyčka. Je odebrána první událost ze seznamu. Zkontroluje se, zda-li je událost null a případně se příkazem break ukončí provádění programu. Dále následuje kontrola, jestli čas události nepřekročil čas simulace, pokud ano, break ukončí provádění smyčky. Vzhledem k tomu, že jsou události vždy seřazené, další události budou k provedení později, takže lze celý program ukončit a mít jistotu, že byly provedeny všechny události, které v daném časovém rámci měly být provedeny. Následuje výpis času a provedení události. Metoda doWork vrací objekt typu Event, proto je tedy vytvořen nový objekt, kterému je přiřazen objekt vrácený metodou doWork:

```
Event newEvent = e.doWork();
```

V případě, že newEvent není null, je přidán do událostí:

```
events.add(newEvent);
```



Obr. 6.8: UML diagram třídy DiscreteEventSimulation

6.10 Příklad simulace

Možnost simulace s ručně nastavenými parametry se využívá pro ověření vhodnosti dané simulace, lze tak například srovnávat 2 zvažované konfigurace.
Příklad simulace městského okruhu se 4 křižovatkami:
Auta budou přijíždět na křižovatku 1, pokračovat na křižovatky 2 až 4, a poté opustí systém. Příjezd aut bude generován po 10. Mezi každými 2 auty bude rozestup 10 časových jednotek, mezi každými 10 auty potom 100 časových jednotek. Bude předpokládáno, že časový interval zelené a červené je na všech křižovatkách stejný a neměnný, tudíž bude upravována pouze konfigurace času přepnutí křižovatky na zelenou. Předpokládá se, že každému autu bude trvat 20 časových jednotek projetí křižovatky (rozestupy mezi auty, rozjíždění). V případě simulace je výpis poměrně obsáhlý a lze po celou dobu kontrolovat, co se v dané simulaci děje. Pro účely optimalizace je výpis vypnutý, protože by neúměrně prodlužoval dobu výpočtu. Implementace takové simulace je popsána na následující straně.

6.11 Třída Simulation

Pro simulaci bez optimalizace lze vybírat ze 2 tříd - Simulation a Simulation 1. V těchto třídách jde nastavit počet semaforů, jejich propojení, počet přijíždějícíh aut, periody a offsety semaforů. Tyto parametry jsou plně dostačující k reálné simulaci provozu - můžou zde být otestována 2 nastavení a porovnána jejich kvalita. Následuje příklad takové simulace.
V hlavní metodě budou vytvořeny 4 nové křižovatky s popisem, které budou následovně propojeny

```
CrossRoad cr1 = new CrossRoad("cr1");
cr1.add(cr2, 0.25);}
```

Bude naplněn seznam aut ve for cyklu a každému autu bude zadán průjezd okruhem s počátkem na křižovatce 1 a koncem na křižovatce 4:

```
ars.add(index, new Car(time));
cars.get(index).addCrossRoad(cr1);
```

Zde je vidět výhoda možnosti vytvářet auto pouze s časem a bez popisu. V případě nutnosti inicializace proměnné typu String by byl for cyklus zbytečně složitější. Následuje odebrání první křižovatky z cesty každého auta a přidání daného auta do fronty čekajících aut na danou křižovatku (v tomto případě cr1):

```
cars.get(i).pollNextCrossRoad().addCar(cars.get(i));
```

Je vytvořena nová simulace diskrétních událostí a k provedení jsou přidány 4 události EventSwictchLights s počátečním časem 0, 30, 60 a 90:

```
cars.get(i).pollNextCrossRoad().addCar(cars.get(i));
```

Následuje spuštění simulace na 30 minut. Jako základní časová jednotka je zde brána sekunda, takže se simulace spustí na 30*60 základních časových jednotek:

```
des.start(30 * 60);
```

Jako poslední se provede výpis jednotlivých čekacích dob pro každé auto. Vzhledem k tomu, že auto nemá žádné konkrétní označení, jsou auta vypisována v pořadí, v jakém se nachází v seznamu po skončení simulace. Celý výpis je možné nalézt v příloze A.1.

Lze si všimnout, že průměrné čekání auta bylo 405 sekund, tj. přes 101 sekund na každé křižovatce. To je poměrně vysoká hodnota ve srovnání s obvyklou čekací dobou, která se pohybuje mezi 45 a 60 sekundami [17]. Tato hodnota je dána z větší části délkou intervalu červené a zelené. Vzhledem k nízkému počtu vozidel projíždějících semafory a délkou červené a zelené bude v případě 40 aut nejvýhodnější nastavení takové, kde se semafory spustí okamžitě po sobě.
Ověření hypotézy výhodnosti přepnout semaforu naráz ihned na zelenou (výpis ze simulace): The total number of seconds cars waited: 12720s (celkový počet sekund, které auta strávila čekáním)

```
Average waiting time was 318.0s
```

Lze si všimnout, že průměrná čekací doba se zkrátila o 87 sekund, což činí 79,5 sekund na jednu křižovatku, čímž se výrazně přiblížila obvyklé čekací době. Pro dosažení ještě lepšího výsledku bude potřeba nastavit správně délku periody. Optimalizace je diskutována v následující kapitole.

6.12 Implementace optimalizace

Genetický algoritmus jako takový bývá často naprogramován univerzálně, neboli aplikovatelný na celou řadu problémů, jejichž vstup (parametry) lze reprezentovat sérií hodnot (polem) a u nichž lze vytvořit fitness funkci meřící kvalitu daného řešení.

Nejdříve bylo potřeba implementovat knihovnu JGAP - jedná se o volně dostupnout knihovnu pro nekomerční použití [20]. Prvním krokem implementace knihovny je import *.jar souborů. Tyto soubory jsou nalinkovány ke každému projektu, není je teda třeba při přenosu projektu na jiný počítač znovu připojovat. V tomto případě se jedná o 33 knihoven, která se nachází ve složce „Referenced Libraries“.

Po importu knihoven je potřeba vytvořit fitness funkci - na rozdíl od tříd chromozom nebo evolve je fitness funkce vždy jiná - individuální pro každý problém. Fitness funkce hodnotí vhodnost jednotlivých řešení, takže ani nemůže být vytvořena obecně tak, aby byla použitelná v každém projektu.

V tomto případě je hodnotícím parametrem fintness funkce suma individuálních zpoždění vozidel po skončení simulace. Fitness funkce je implementována v podobě třídy TrafficFitnessFunction, která dědí od třídy FitnessFunction a musí implementovat funkci evaluate vracející hodnotu typu double. Ve třídě se nachází prázdný konstruktor (třída při inicializaci nepotřebuje žádné vstupní hodnoty) a funkce evaluate.

V prním kroku je, podobně jako ve třídě Simulation, potřeba nadefinovat křižovatky. Autům je potom přidělena cesta, kterou budou projíždět a jsou přiřazeny na první křižovatku. Každé auto je přiřazeno na patřičnou křižovatku. Příjezd aut je považován jako vstupní hodnoty, které se nebudou optimalizovat - v reálné situaci se počet přijíždějících aut v podstatě nedá regulovat (nelze například rozdělit motoristy na sudé a liché a poslat část jednou cesou a část druhou, jako se tomu děje u load balancingu v případě routerů). Optimalizace bude probíhat v podobě nastavování period zelených a červených a vzájemného posunutí časů jednotlivých semaforů. Tato informace je získána z dané pozice alely:

```
des.addEvent(new EventSwitchLights((Integer)
    configuration.getGene(0).getAllele(), cr1, (Integer)
    configuration.getGene(5).getAllele()));
```

V simulaci je zvažováno 5 křižovatek. Geny 0 až 4 slouží k nastavení délky periody zelených a alely 5 až 9 ke vzájemnému posunutí. Po proběhnutí simulace je spočteno celkové zpoždění:

```
for (Car car : cars)
    totalWaiting += car.getWaiting();
```

Tato hodnota je poté použita ve výpočtu návratové hodnoty:

```
return (NUMBER_OF_CARS * SIMULATION_LENGTH - totalWaiting + 1);
```

Fitness funkce nesmí vracet záporné hodnoty a minimální vrácená hodnota je definována jako 1. Čím vyšší hodnota fitness funkce, tím lepší dané řešení je - tj. nelze přímo vracet hodnotu proměnné totalWaiting, protože zde je logika opačná. Ve výpočtu je třeba zajistit, aby hodnota nebyla záporná - nejdřívě je spočtena maximální možná doba čekání všech aut (čas simulace * počet všech aut, tato situace by nastala v případě, že všechna auta budou po celou dobu běhu simulace čekat). K dané hodnotě je přičteno 1, aby se zajistila minimální hodnota 1 i pro nejhorší případ, že všechna auta budou po celou dobu čekat.
Hodnoty délek period a jejich vzájemné posuny budou nastavovány ve třídě Optimization, kde je fitness funkce volána.

6.13 Třída Optimization

Jako první je potřeba zvolit zvolit konfiguraci - v konfiguraci lze volit genetické operátory, náhodné generátory čísel, atd. Jedná se spíše o možnost úpravy těchto vstupů pro specializované případy, většina příkladů vystačí se standardní konfigurací, která je použita i v této práci:

```
Configuration conf = new DefaultConfiguration();
```

Poté je definována fitness funkce a tato funkce je přiřazena dané konfiguraci:

```
FitnessFunction myFunc = new TrafficFitnessFunction();
conf.setFitnessFunction(myFunc);
```

Následuje definice řešení (jeho délky - počtu v něm obsažených hodnot):

```
Gene[] solution = new Gene[10];
```

Do každé hodnoty je přiřazeno nastavení a určen rozsah hodnot (u první hodnoty v prvním genu se jedná o rozsah 0 až 150):

```
solution[0] = new IntegerGene(conf, 0, 150);
```

Chromozom se skládá z možného řešení a dané konfigurace:

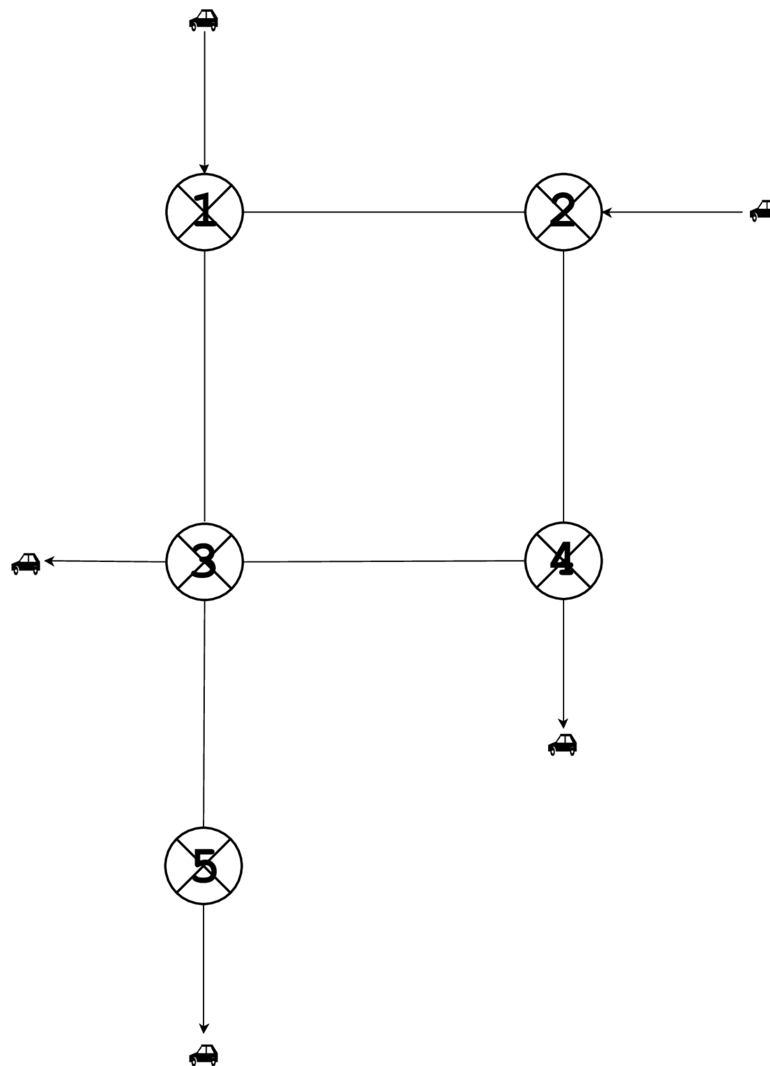
```
Chromosome sampleChromosome = new Chromosome(conf, solution);
```

Po nastavení velikosti populace a definice proměnné bestSolution pro uchování nejlepšího řešení je v daném počtu iterací řešení optimalizováno. V tomto případě nelze určit, jaké je optimální řešení, respektive jeho hodnotu fitness - nelze tedy ukončit optimalizaci při dosažení určité fitness hodnoty. V každé generaci populace je porovnána fitness hodnota nejlepšího řešení s fitness hodnotou aktuálně nejlepšího řešení a případně uloženo současné nejlepší řešení do proměnné bestSolution. Na závěr je vypsáno celkové čekání všech vozidel a průměrné čekání jednoho vozidla.

7 VÝSLEDNÁ SIMULACE

7.1 Nastavení parametrů

Pro simulaci bylo vybráno 5 křižovatek, auta budou přijíždět ze 2 směrů (jeden dálniční sjezd a jedna rychlostní silnice o celkově 4 pruzích). Auta budou sledovaný úsek opouštět 3 křižovatkami (jedná se o 3 velice vytížené křižovatky). Celá situace je znázorněna na následujícím diagramu:



Obr. 7.1: Simulovaná situace

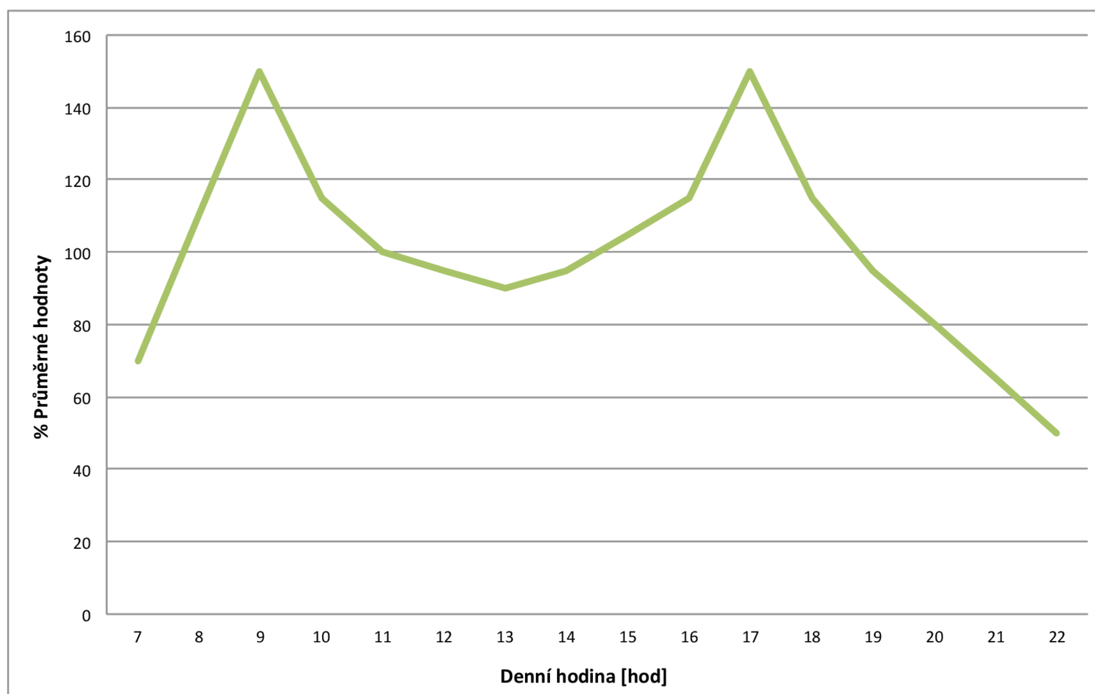
Počty aut vstupujících do systémy jsou známy (počet aut, který danou křižovatkou průměrně projede za hodinu). Auta vstupující křižovatkou 1 budou náhodně volit jednu ze 3 tras:

- 1, 3, výstup ze simulace
- 1, 3, 4, výstup ze simulace
- 1, 3, 5, výstup ze simulace

Auta vstupující křižovatkou 3 budou náhodně volit jednu ze 4 tras:

- 2, 4, výstup ze simulace
- 2, 4, 3, výstup ze simulace
- 2, 4, 3, 5, výstup ze simulace

Ve třídě CrossRoad je uložen seznam aut čekajících na průjezd. V základním nastavení se předpokládá, že autu trvá projetí křižovatky 20 sekund. Tj. v jedné periodě může křižovatkou projet maximálně délka periody/ 20 aut.
Aby srovnávání jednotlivých nastavení bylo objektivní, je na počátku vygenerována jedna konfigurace (zvoleny počty aut a trasy, kterými budou projíždět). Toto nastavení je dále používáno pro všechny simulace. V nastavení, které je použito pro testování, se předpokládá následující průběh dopravy:



Obr. 7.2: Vývoj dopravy ve městě Adelaide

V grafu jsou vidět 2 dopravní špičky (mezi 8 a 9 a mezi 16 a 17 hodinou), v nichž doprava stoupá až ke 145 % v porovnání s průměrem. Procentuální nárůst dopravy oproti průměru:

Tab. 7.1: Dopolední doprava ve městě Adelaide

Čas	6-7	7-8	8-9	9-10	10-11	11-12
[%]	70	110	150	115	100	95

Tab. 7.2: Odpolední doprava ve městě Adelaide

Čas	12-13	13-14	14-15	15-16	16-17	17-18	18-19	19-20	20-21	21-22
[%]	90	95	105	115	150	115	95	80	65	50

Pozn.: U průběhu dopravy se jedná o odhad města Adelaide.
Statický systém byl optimalizován na průměrnou hodnotu dopravy, tj. 100 %. Dynamický systém byl optimalizován na 16 různých nastavení (tj. pro každou hodinu). Poté bylo u obou typů řízení dopravy měřeno zpoždění pomocí nastavení ve třídě Simulation. Výsledky jsou diskutovány v následující kapitole.

7.2 Srovnání dynamického a statického systému

Současné systémy používají jedno nastavení pro celý den, bude zde tedy vzata průměrná hodnota počtu aut za hodinu a pro ni bude optimalizován celý systém. S tímto nastavením bude potom simulováno 16 jednotlivých hodinových úseků - od 6-té hodiny ranní po 10 večerní, což je časový interval, ve kterém je doprava intenzivní, mimo tyto hodiny lze přepnout semafory do tzv. pasivního režimu, ve kterém bliká oranžové světlo a přednosti jsou dány dopravními značkami.
V následující tabulce je srovnáno celkové zpoždění u statického nastavení (CZSN) v dané hodině, celkové zpoždění u dynamického nastavení (CZDN) a jejich procentuální rozdíl, respektive časová úspora u dynamického nastavení.

Tab. 7.3: Porovnání statického a dynamického nastavování semaforů

Hodina	Doprava	CZSN	CZDN	Rozdíl [%]
6 až 7	70%	76	71	0,06
7 až 8	110%	121	107	12,85
8 až 9	150%	171	125	36,92
9 až 10	115%	126	112	12,50
10 až 11	100%	108	108	0,14
11 až 12	95%	103	95	8,32
12 až 13	90%	99	88	12,16
13 až 14	95%	103	95	8,32
14 až 15	105%	113	102	11,18
15 až 16	115%	126	112	12,50
16 až 17	150%	171	125	36,92
17 až 18	115%	126	112	12,50
18 až 19	95%	103	95	8,32
19 až 20	80%	86	77	11,82
20 až 21	65%	70	62	13,47
21 až 22	50%	66	63	5,00
Průměr	100%	111	97	12,69

8 ZÁVĚR

V diplomové práci byl vyvinut systém umožňující simulovat dopravu ve městě. Pro simulaci byl vybrán programovací jazyk Java, hlavně kvůli tomu, že se jedná o jazyk objektový, velmi rozšířený a poskytující dobré možností ladění programu. Jako zvlášť netriviální se ukázala být simulace diskrétních událostí. Návrh byl nejdříve analyzován a rozdělen na jednotlivé objekty, které byly implementovány v podobě tříd. Byl navrhnut systém penalizace v podobě celkového čekání jednotlivých vozidel. Tato penalizace byla použita jako hodnotící parametr vhodnosti nastavení celého systému. Byly simulovány 2 modelové situace a porovnány výsledky jednotlivých nastavení.
Dále bylo řešení rozšířeno rozdělením na události přepnutí semaforu a pohybu auta a byla přidána optimalizace v podobě genetického algoritmu. Po konzultaci se zastupiteli města Adelaide byla vybrána oblast, která byla namodelována a byla porovnáno celkové čekání aut při použití statického a dynamického nastavení systému. Bylo prokázáno, že dynamický systém je při dané dopravní situaci (vycházející z reálných dat) schopen snížit čekání vozidel o 12,69 %, což podle studie, kterou si nechalo město vypracovat, odpovídá úspoře 3 miliony dolarů ročně [22].

LITERATURA

[1] Badger, E. *How Traffic Congestion Affects Economic Growth* [online]. 2013, poslední aktualizace 22. 10. 2013 [cit. 20. 5. 2014]. Dostupné z URL: <http://www.citylab.com/commute/2013/10/how-traffic-congestion-impacts-economic-growth/7310/>.

[2] Harris, W. *Tunnel Construction: Soft Rock and Underwater* [online]. 2008, poslední aktualizace 3. 4. 2008 [cit. 17. 12. 2013]. Dostupné z URL: <http://science.howstuffworks.com/engineering/structural/tunnel4.htm>.

[3] Adelaide Council *Census QuickStats* [online]. 2011, poslední aktualizace 28. 3. 2011 [cit. 17. 12. 2013]. Dostupné z URL: <http://www.censusdata.abs.gov.au/census_services/getproduct/census/2011/quickstat/4GADE>.

[4] Adelaide Council *Census QuickStats* [online]. 2011, poslední aktualizace 28. 3. 2011 [cit. 4. 3. 2014]. Dostupné z URL: <http://www.abs.gov.au/websitedbs/censushome.nsf/home/data?opendocument&navpos=200>.

[5] Siemens *Census QuickStats* [online]. 2014, poslední aktualizace 15. 2. 2014 [cit. 20. 5. 2014]. Dostupné z URL: <http://www.mobility.siemens.com/mobility/global/en/urban-mobility/road-solutions/traffic-control-center-platform/sitraffic-concert-our-traffic-control-center/pages/sitraffic-concert-our-traffic-control-center.aspx>.

[6] Department of Planning, Transport and Infrastructure *Metropolitan Traffic Estimate Maps* [online]. 2014, [cit. 10. 5. 2014]. Dostupné z URL: <http://www.dptiapps.com.au/traffic-maps/aadt_rt1_colour.pdf>.

[7] Smith, P. *Traffic Fluency Survey* [online]. 2014, poslední aktualizace 3. 1. 2014 [cit. 22. 4. 2014]. Dostupné z URL: <http://traffic.sa.gov.au>.

[8] Smith, P. *TELEMETRY NETWORKS* [online]. 2001, poslední aktualizace 3. 5. 2001 [cit. 17. 12. 2013]. Dostupné z URL: <http://www.irig106.org/docs/106-05/irig106part2.pdf>.

[9] Wooldridge, M. *An Introduction to Multiagent Systems* [online]. 2005, poslední aktualizace 6. 11. 2005 [cit. 19. 12. 2013]. Dostupné z URL: <http://books.google.com.au/books?hl=cs&lr=&id=C4_9riKP2kQC&oi=fnd&pg=PR7&dq=multiagent+systems&ots=ovgtfCW91a&sig=wcm6KLQa1gZH396Qo-GSCO4jVzc&redir_esc=y#v=onepage&q=multiagent20systems&f=false>.

[10] Wooldridge, M. *Steganographic Routing in Multi Agent System Environment* [online]. 2007, poslední aktualizace 2. 8. 2007 [cit. 20. 12. 2013]. Dostupné z URL: <http://arxiv.org/pdf/0806.0576.pdf>.

[11] Matthews, F. *Procedural concepts* [online]. 2009, poslední aktualizace 12. 7. 2009 [cit. 8. 1. 2014]. Dostupné z URL: <http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_6/types_language/miniweb/pg3.htm>.

[12] Wayner, P. *From PHP to Perl: What's hot, what's not in scripting languages* [online]. 2011, poslední aktualizace 17. 11. 2011 [cit. 13. 2. 2014]. Dostupné z URL: <http://www.infoworld.com/d/application-development/php-perl-whats-hot-whats-not-in-scripting-languages-175867>.

[13] Maers, G. *Comparison of Java and C++* [online]. 2010, poslední aktualizace 31. 3. 2010 [cit. 13. 2. 2014]. Dostupné z URL: <https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Comparison_of_Java_and_C++.html>.

[14] Darryl, K. *Application Development: 10 Reasons Java Has Supplanted C++ (and 5 Reasons It Hasn't)* [online]. 2012, poslední aktualizace 25. 6. 2012 [cit. 19. 2. 2014]. Dostupné z URL: <http://www.eweek.com/c/a/Application-Development/10-Reasons-Java-Has-Supplanted-C-and-5-Reasons-It-Hasnt-159065/>.

[15] Horáček, M. *TIOBE Index for December 2013* [online]. 2013, poslední aktualizace 16. 12. 2013 [cit. 23. 12. 2013]. Dostupné z URL: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

[16] Horáček, M. *Diskrétní simulace* [online]. 2012, poslední aktualizace 23. 6. 2013 [cit. 23. 12. 2013]. Dostupné z URL: <http://www.simulace.info/index.php/Discrete_event_simulation/cs>.

[17] Jeffrey, C. *How Much Time Do We Spend Waiting at Red Lights?* [online]. 2009, poslední aktualizace 7. 1. 2009 [cit. 23. 12. 2013]. Dostupné z URL: <http://www.ask.com/question/how-much-time-do-we-spend-waiting-at-red-lights>.

[18] Sintes, T. *Vector or ArrayList – which is better?* [online]. 2001, poslední aktualizace 22. 2. 2001 [cit. 23. 12. 2013]. Dostupné z URL: <http://www.javaworld.com/article/2077425/java-se/vector-or-arraylist-which-is-better.html>.

[19] Horswill, Ian. *Australian motorists* [online]. 2013, poslední aktualizace 6. 11. 2013 [cit. 23. 12. 2013]. Dostupné

z URL: <http://www.news.com.au/travel/travel-updates/australian-motorists-spend-94-hours-a-year-in-congestion-tomtom-traffic-inde story-e6frfq80-1226754339030>.

[20] Meffert, Klaus. *Getting Started With JGAP* [online]. 2002, poslední aktualizace 2012 [cit. 23. 12. 2013]. Dostupné z URL: <http://jgap.sourceforge.net/doc/tutorial.html>.

[21] Helsgaun, K. *Discrete Event Simulation in Java* [online]. 2008, poslední aktualizace 16. 9. 2008 [cit. 23. 12. 2013]. Dostupné z URL: <http://www.akira.ruc.dk/~keld/research/javasimulation/JAVASIMULATION-1.1/docs/Report.pdf>.

[22] Adelaide Council. *City Living* [online]. 2012, poslední aktualizace 8. 12. 2012 [cit. 23. 3. 2013]. Dostupné z URL: <http://www.adelaidecitycouncil.com>.

SEZNAM PŘÍLOH

A Výstup programu **52**

A.1 Výpis z programu pro zvolenou konfiguraci 52

A.2 Zdrojový kód 56

A VÝSTUP PROGRAMU

A.1 Výpis z programu pro zvolenou konfiguraci

Running simulation in time 0s
Switching light cr1 to green
The number of cars waiting is 40

Running simulation in time 30s
Switching light cr2 to green
The number of cars waiting is 15

Running simulation in time 60s
Switching light cr3 to green
The number of cars waiting is 15

Running simulation in time 90s
Switching light cr4 to green
The number of cars waiting is 15

Running simulation in time 300s
Switching light cr1 to red
The number of cars waiting is 25

Running simulation in time 330s
Switching light cr2 to red
The number of cars waiting is 0

Running simulation in time 360s
Switching light cr3 to red
The number of cars waiting is 0

Running simulation in time 390s
Switching light cr4 to red
The number of cars waiting is 0

Running simulation in time 600s
Switching light cr1 to green

The number of cars waiting is 25

Running simulation in time 630s
Switching light cr2 to green
The number of cars waiting is 15

Running simulation in time 660s
Switching light cr3 to green
The number of cars waiting is 15

Running simulation in time 690s
Switching light cr4 to green
The number of cars waiting is 15

Running simulation in time 900s
Switching light cr1 to red
The number of cars waiting is 10

Running simulation in time 930s
Switching light cr2 to red
The number of cars waiting is 0

Running simulation in time 960s
Switching light cr3 to red
The number of cars waiting is 0

Running simulation in time 990s
Switching light cr4 to red
The number of cars waiting is 0

Running simulation in time 1200s
Switching light cr1 to green
The number of cars waiting is 10

Running simulation in time 1230s
Switching light cr2 to green
The number of cars waiting is 10

Running simulation in time 1260s

Switching light cr3 to green
The number of cars waiting is 10

Running simulation in time 1290s
Switching light cr4 to green
The number of cars waiting is 10

Running simulation in time 1500s
Switching light cr1 to red
The number of cars waiting is 0

Running simulation in time 1530s
Switching light cr2 to red
The number of cars waiting is 0

Running simulation in time 1560s
Switching light cr3 to red
The number of cars waiting is 0

Running simulation in time 1590s
Switching light cr4 to red
The number of cars waiting is 0

Running simulation in time 1800s
Switching light cr1 to green
The number of cars waiting is 0
List of times car waited
Car 1: 780s
Car 2: 770s
Car 3: 760s
Car 4: 750s
Car 5: 740s
Car 6: 730s
Car 7: 720s
Car 8: 710s
Car 9: 700s
Car 10: 790s
Car 11: 480s
Car 12: 470s

Car 13: 460s
Car 14: 450s
Car 15: 440s
Car 16: 430s
Car 17: 420s
Car 18: 410s
Car 19: 400s
Car 20: 490s
Car 21: 480s
Car 22: 470s
Car 23: 460s
Car 24: 450s
Car 25: 440s
Car 26: 130s
Car 27: 120s
Car 28: 110s
Car 29: 100s
Car 30: 190s
Car 31: 180s
Car 32: 170s
Car 33: 160s
Car 34: 150s
Car 35: 140s
Car 36: 130s
Car 37: 120s
Car 38: 110s
Car 39: 100s
Car 40: 90s
The total number of seconds cars waited: 16200s
Average waiting time was 405.0s

A.2 Zdrojový kód

V příloze se nachází zdrojový kód v jazyce Java. Celé řešení je hojně komentováno pro lehčí čitelnost kódu.