České vysoké učení technické v Praze

Fakulta jaderná a fyzikálně inženýrská

BAKALÁŘSKÁ PRÁCE

Kamil Ondrák

Praha – 2010

České vysoké učení technické v Praze

Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky

Decentralizované řízení dopravní signalizace: optimalizace zelené vlny

BAKALÁŘSKÁ PRÁCE

Vypracoval: Kamil Ondrák

Vedoucí práce: Ing. Václav Šmídl, Ph.D.

Konzultant: Dr. Ing. Jan Přikryl, Ph.D.

Rok: 2010

Zadání práce s podpisem děkana.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

Praha, 7. července 2010 Kamil Ondrák

Poděkování

(. . .) Děkuji Mgr. Markovi Hryciowovi za pozorné pročtení a podnětné připomínky.
(. . .)

Kamil Ondrák

Název práce:

Decentralizované řízení dopravní signalizace: optimalizace zelené vlny

Autor: Kamil Ondrák

Obor: Inženýrská informatika

Druh práce: Bakalářská práce

Vedoucí práce: Ing. Václav Šmídl, Ph.D.
ÚTIA AV ČR

Konzultant: Dr. Ing. Jan Přikryl, Ph.D.
ÚTIA AV ČR

Abstrakt: Popis práce

Klíčová slova:

Title:

Decentralized control of traffic lights: green wave optimization

Author: Kamil Ondrák

Abstract: Popis práce anglicky

Key words:

Obsah

Úvod **1**

1 Popis situace **3**

1.1 Konstrukce a řízení křižovatky 3

1.2 Oblast Zličína 4

1.3 Simulátor Aimsun 5

1.3.1 Getram Extensions 6

1.4 Emulátor řadiče ELS3 7

2 Agentní systémy **9**

2.1 Inteligentní agent 9

3 Algoritmus řízení **12**

3.1 Návrh algoritmu 12

3.2 Použité knihovny 15

3.2.1 IT++ 16

3.2.2 BDM 17

3.2.3 Libconfig 18

3.3 VGS API 18

3.4 Struktura programu 19

3.5 Program `main_loop.exe` 20

3.6 Implementace navrženého algoritmu 23

4 Výsledky simulací **34**

Závěr **35**

Seznam použitých zdrojů **36**

Přílohy **37**

Úvod

Na mnoha místech silničních komunikací dnes nastává problém s jejich ucpáváním příliš silným automobilovým provozem, přičemž nejčastějším úzkým hrdlem bývají křižovatky. Teoreticky nejjednodušší se jeví přestavba dotčených míst. To je však často také řešení nejnákladnější a v některých místech to ani není možné, například z důvodu nedostatku prostoru. Další možností je omezení počtu vozidel přijíždějících do předmětné oblasti. To sice zlepší průjezdnost v kritickém místě, ale provoz se tím přesouvá jinam, kde tak může vzniknout podobný problém. Jako zajímavá možnost se jeví co nejlepší řízení provozu pomocí světelných signalizačních zařízení.

V současné době je většina vytížených křižovatek řízena pomocí poměrně sofistikovaných signálních plánů, které se i dokáží přizpůsobovat aktuální dopravní situaci. Tyto plány bývají vypočítány různými dlouhodobě vyvíjenými programy na základě změřených dat o hustotě dopravy v daném místě. Pro izolované křižovatky se chovají velmi dobře, nevýhodou toho způsobu řízení však bývá právě izolovanost – každá křižovatka má informace jen o svém nejužším okolí a chybí jakákoli koordinace mezi sousedícími křižovatkami.

Na některých místech se používá jistá forma centralizovaného řízení. Ústředna sbírá údaje z určité oblasti a na základě těchto informací vysílá pokyny do řadičů příslušných křižovatek. Tento postup vede k velmi dobrým výsledkům, ale většímu rozšíření brání malá univerzálnost algoritmů používaných v ústřednách.

Relativně novým přístup je decentralizovaném řízení dopravní signalizace. Ta funguje na principu multiagentních systémů. Každá křižovatka se pak stává jedním agentem, který jedná s ostatními agenty za účelem vytvoření společné strategie řízení vedoucí ke zlepšení průjezdnosti.

Cílem této práce je seznámit se s tímto decentralizovaným způsobem řízení, navrh-

nout komunikační strategii, která by pomocí nastavení tak zvaných offsetů mohla vézt ke zlepšení průjezdnosti oblastí, toto řešení implementovat na počítači a prozkoumat jeho důsledky v simulátoru dopravy Aimsun. Toto vše je prováděno na modelu skutečné oblasti, konkrétně Řevnické ulice v Praze – Zličíně. Jako referenční stav pak slouží řízení dopravní signalizace signálními plány expertně navrženými metodou Monte Carlo.

Na začátku práce je představen způsob řízení světelných křižovatek, používané detektory a popis signálních plánů. Následuje popis modelované oblasti a po té sekce o simulátoru Aimsun. Ta představuje některé možnosti, které Aimsun nabízí, ukazuje jaká vstupní data do simulace vstupují a jaké zní vystupují. Součástí je také popis rozhraní Getram Extensions, které se používá pro komunikaci mezi simulátorem a externími programy.

Následující kapitola pak představuje úvod to teorie multiagentních systému ... (atd, podle toho, co v kapitole bude)

Ve třetí kapitole je popsána nejprve teoreticky navrhovaná strategie komunikace mezi agenty a pak i představena konkrétní implementace. Ta spočívá v rozšíření stávajícího řešení pro simulace dopravy vyvíjené v Ústavu teorie informace a automatizace (ÚTIA) Akademie věd ČR.

V poslední kapitole se nacházejí výsledky simulací navrženého algoritmu a srovnání stavu v oblasti před a po jeho aplikaci.

Kapitola 1

Popis situace

1.1 Konstrukce a řízení křižovatky

Křižovatka řízená světelným signalizačním zařízením se skládá z několika prvků. Vedle samotných semaforů je v její blízkosti připojen řadič, který se stará o ovládání signalizačních prvků. K řadiči pak může být připojeno několik detektorů, poskytujících mu informace a aktuální dopravní situaci.

Detektory se dělí dělí do tří kategorií podle jejich umístění. (i) výzvové, (ii) prodlužovací a (iii) strategické. (i) jsou umístěné na stop čáře, (ii) cca 30 metrů před ní a (iii) na vzdálenějších místech, typicky alespoň 100 metrů před stop čárou. Ne vždy musí být na všech ramenech křižovatky instalovány všechny typy detektorů.

Technicky je nejčastějším řešením detektoru indukční smyčka umístěná ve vozovce, kterou prochází střídavý proud. Ta pomocí změn magnetického pole ve své okolí zaznamenává projíždějící vozidla. V případě umístění dvou smyček blízko sebe lze získat i podrobnější informace o rychlosti, druhu vozidla a další údaje. Další možností sledování provozu jsou infračervené detektory. Jejich hlavní součástí je kamera, snímající sledovaný prostor v infračervené oblasti. Ve výsledném obrazu pak poměrně snadno rozezná automobily i chodce jakožto zdroje tepelného záření.

Dále existují například video detektory založené na kamerách snímající v oblasti viditelného světla (videodetektory), mikrovlnné nebo ultrazvukové detektory. Tyto však nejsou v současné době v simulované oblasti instalovány.

Detektory umístěné na Zličíně poskytují dva údaje. Prvním je počet vozidel, která přes detektor projela a druhým čas, po který se v detekované oblasti vyskytovalo nějaké vozidlo. Bohužel, tato data nejsou naprosto přesná.

(!chyby detektorů!)

Řízení křižovatky probíhá pomocí signálních plánů uložených v řadiči nebo dle pokynů z ústředny. Signální plány mohou být pevné nebo rámcové. Signální plán se skládá z několika fází, které mají buď pevně nebo rámcově dány časy začátků a délku. Fáze jsou složené ze signálních skupin. Signální skupina pak značí skupinu světelných signalizačních zařízení, která rozsvěcuje současně zelenou. Signální plány mají pevnou a jednotnou délku jednoho cyklu, značí se T_c. V předmětné oblasti je to konkrétně 80 sekund. Posledním důležitým pojmem je *offset*. Ten říká o kolik sekund je začátek jednoho cyklu signálního plánu posunut oproti jistému počátečnímu času t_0.

(možná obrázek sem)

V případě pevných signálních plánů se přepínání fází řídí předem danou tabulkou, která určuje kdy která fáze začíná a jak dlouho má trvat. Navíc může být u fází uložena možnost je prodloužit v případě, že údaje z detektorů oznamují další přijíždějící vozidla v daném směru.

Rámcové signální plány dovolují řadiči větší volnost při zařazování fází. Na základě aktuálních naměřených údajů může řadič volit nejvhodnější fázi z několika momentálně přípustných, může nastat i situace, že některá fáze nebude v rámci cyklu zařazena vůbec.

1.2 Oblast Zličína

Ověření možnosti decentralizovaného řízení dopravní signalizace je v této práci provedeno na modelu skutečné oblasti: ulici Řevnické v Praze – Zličíně. V ulici se nachází pět světelných křižovatek situovaných v jedné linii. Pro zjednodušení jsou v této práci simulovány jen dvě severní křižovatky: 5.495 a 5.601. První jmenovaná je trojramenná křižovatka ulic Řevnická a Na Radosti, druhá je potom čtyřramenná a je tvořena napojením autobusového terminálu na jedné a obchodního centra Metro-

pole Zličín na druhé straně na Řevnickou.

1.3 Simulátor Aimsun

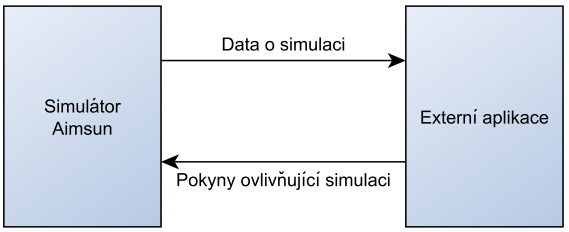
K simulaci dopravní situace se používá softwarový balík GETRAM/AIMSUN od společnosti TSS (Transport Simulation Systems) ve verzi 4.2.16. Jádrem celého nástroje je program Aimsun, který slouží k samotné simulaci.

Aimsun (Advanced Interactive Microscopic Simulator for Urban and Non-Urban Networks, [7]) je mikrosimulátorem dopravy[1]. To znamená, že modeluje polohu jednotlivých vozidel spojitě během celé doby simulace. Každé vozidlo se řídí určitým modelem chování. Lze nastavit různé styly předjíždění, prudkost brzdění a rozjezdů, ale třeba i ochotu čekat. Je možné simulovat různé druhy vozidel, od osobních přes nákladní auta až po autobusy a hromadnou dopravu vůbec. Vedle vozidel Aimsun samozřejmě nabízí simulaci většiny objektů, které se vyskytují v dopravních sítích: světelných signalizační zařízení, detektorů, atd.

Vstup pro simulátor se skládá ze scénáře a parametrů simulace. Scénář obsahuje popis dopravní sítě, údaje o dopravní poptávce, programy řízení dopravy a plány veřejné dopravy. Popis dopravní sítě představuje geometrickou reprezentaci zkoumané oblasti, tedy rozměry a tvar jednotlivých silničních pruhů a křižovatek, umístění dopravní signalizace, detektorů, zastávek hromadné dopravy a podobně. Dopravní poptávka může být zadaná dvěma způsoby. Buď ve formě intenzity dopravy na vstupech, poměrů odbočení na křižovatkách a počátečního stavu sítě, nebo pomocí takzvané O-D matice, která zachycuje počet uskutečněných cest mezi dvojicemi vstupních a výstupních bodů. Programy řízení dopravy zahrnují popisy fází a jejich trvání pro řízené křižovatky, přednosti v jízdě pro křižovatky neřízené. Konečně plány veřejné dopravy se skládají z popisů tras, zastávek a jízdních řádů linek hromadné dopravy v simulované oblasti. Parametry simulace pak představují pevné hodnoty popisující experiment (jako doba simulace) a proměnné hodnoty určené pro kalibraci modelu.

Výstup za simulace je spojitá grafická reprezentace zkoumené oblasti, statistická data o provozu (tok vozidel, počty zastavení, průměrná rychlost, atd.) a údaje sesbí-

[1] Současná verze Aimsun 6 dovoluje již mikro, meso i makrosimulaci



Obrázek 1.1: Aimsun a externí aplikace

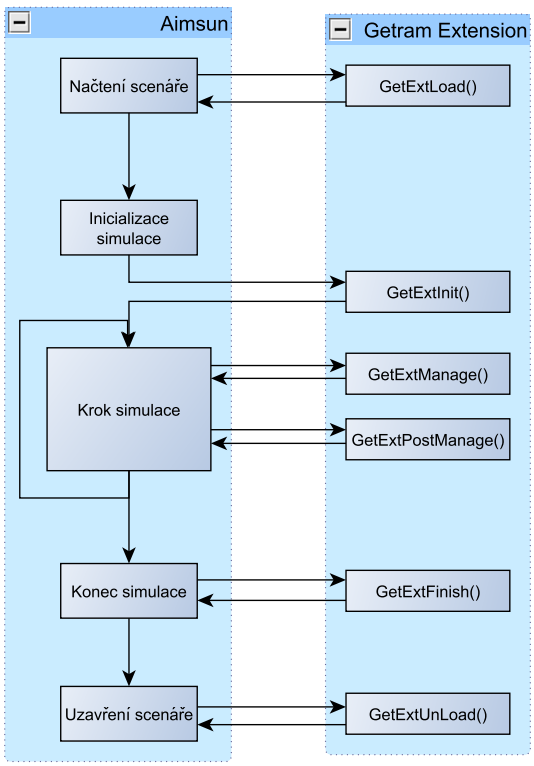
raná z detektorů. Výsledky mohou být vykresleny do grafů, nebo uloženy v textové podobě do souborů či do databáze pro další zpracování.

1.3.1 Getram Extensions

Aimsun má aplikační rozhraní (API), které umožňuje tvorbu rozšířujících modulů jménem Getram Extensions. Aimsun pomocí tohoto rozhraní poskytuje v reálném čase data ze simulace a naopak přijímá data pro její ovlivňování, jak je naznačeno na obrázku (1.1). Rozšíření se píší buď jako DLL knihovny napsané v C/C++ nebo ve formě skriptů v Pythonu.

Komunikace mezi rozšířením (Getram Extension) a simulátorem Aimsun probíhá pomocí šesti funkci:

1. *GetExtLoad()* je volána v okamžiku nahrání rozšíření Aimsunem.
2. *GetExtInit()* je volána na začátku simulace.
3. *GetExtManage()* se volá na začátku každého kroku simulace. Slouží k získání aktuálních údajů z detektorů, dat o vozidlech a dalších údajů. Také umožňuje naopak ovládání aktivních prvků v simulované oblasti.
4. *GetExtPostmanage()* je funkce podobná předchozí, jen tato je volána na konci simulačního kroku.
5. *GetExtFinish()* se volá na konci simulace a v ní rozšíření dokončuje všechny operace, které si to ještě žádají.
6. *GetExtUnLoad()* je zavolána v okamžiku, kdy Aimsun ukončuje komunikaci s rozšířením.



Obrázek 1.2: Podrobný nákres komunikace mezi Aimsunem a Getram Extension

Průběh komunikace mezi Aimsunem a rozšířením je vidět na obrázku (1.2)

1.4 Emulátor řadiče ELS3

Pro simulace funkcí řadiče křižovatky se používá emulátor ELS vyvinutý firmou Eltodo. Ten z reálného řadiče obsahuje algoritmus řízení. Ovládání HW periferií (modul detektorů, spínačů signálních skupin, a podobně) jsou nahrazeny výše popsaným simulátorem Aimsun.

V INI souboru emulátoru řadiče jsou uloženy parametry dopravního řešení použitého v simulované oblasti. Dopravní řešení je určeno parametry křižovatky (její konstrukce, umístění detektorů, fáze, signální skupiny, ...) a dopravními vztahy mezi těmito parametry (signální plány, dynamické řízení, ...). Návrh dopravního řešení je dílem dopravního inženýra a jeho struktura přesahuje rámec této práce.

ELS3 má pro komunikaci s okolím vlastní API. V každém simulačním kroku přes něj

obdrží z Aimsunu stavy svých detektorů. Od knihovny BDM (která zde nahrazuje modul dopravní ústředny) řadič pak přijímá data pro ovlivnění řízení. Výstupem řadiče je obraz barevné kombinace signálních skupin. Ten se zasílá do Aimsunu na konci každého simulačního kroku.

Kapitola 2

Agentní systémy

2.1 Inteligentní agent

Pokud mluvíme o agentních systémech, je nutné předem si vyjasnit základní pojem agent. Bohužel neexistuje přesná obecně přijímaná definice. Pro účely této práce můžeme použít například tu, kterou používá Michael Wooldrige v [8, kap. 1]: „*Agent* je počítačový program, který je *umístěn* v nějakém *prostředí* a který je schopen *autonomního jednání* v tomto prostředí za účelem dosažení určených cílů.“ Autonomií se zde myslí schopnost samostatného jednání bez zásahu člověka nebo jiných programů. Tato definice se zatím vyhýbá pojmu *inteligentní*, tomu se budeme věnovat později.

Agent z definice má tedy určitý druh čidel, kterými může získat některé údaje o svém okolí, omezený seznam akcí, které může provádět a jistý algoritmus, který určuje jakou z akcí (pokud vůbec nějakou) v aktuálním čase provést.

Komplexnost rozhodovacího procesu závisí mimo jiné i na prostředí. To můžeme klasifikovat z mnoha pohledů. Zda lze či nelze získat úplné informace o jeho současném stavu, jestli je deterministické nebo nedeterministické, tedy jestli stejná akce povede vždy ke stejnému konečnému stavu. Dále je možné rozlišit mezi statickým a dynamickým prostředí. Ve statickém prostředí jsou veškeré změny stavu dílem pouze agenta, zatímco dynamické okolí se mění ať agent jedná nebo ne. Podobných rozdělení existuje mnoho, ale toto pro představu postačí. Nejlépe řiditelné je

samozřejmě prostředí, které je plně popsatelné, deterministické a statické, bohužel řízení dopravy je případem přesně opačným.

Naprosto typickým příkladem jednoduchého agenta umístěného v prostředí je termostat ovládající topení za účelem udržení stále teploty v místnosti. Tento agent má jedno čidlo, kterým určuje jestli je teplota nižší než nastavená požadovaná, a dvě možné akce: zapnutí a vypnutí vytápění. Rozhodovací proces se pak skládá ze dvou pravidel:

$$\text{nízká teplota} \longrightarrow \text{zapnout vytápění}$$
$$\text{správná teplota} \longrightarrow \text{vypnout vytápění}$$

Takto postavený program se dá považovat za agenta, ale pravděpodobně jen málokdo by ho označil za agenta inteligentního. Problém je už samotná definice inteligence. Proto se omezíme na to, že inteligentní agent je takový, který dokáže flexibilně reagovat. Flexibilita znamená tři vlastnosti:

reaktivita – inteligentní agent vnímá své okolí a dle takto získaných údajů reaguje na jeho změny tak, aby splnil své nastavené cíle;

proaktivita – inteligentní agent je schopný předvídat chování systému a na základě těchto předpovědí aktivně jednat tak, aby splnil své nastavené cíle;

sociální schopnosti – inteligentní agenti na sebe mohou vzájemně působit tak, aby splnili své nastavené cíle.

Požadavky na vysokou reaktivitu a proaktivitu jdou proti sobě. Je třeba najít vyvážený poměr mezi oběma. Extrémně reaktivní agent totiž na základě neustále přijímaných podmětů stále koná nějakou akci. Ovšem tímto způsobem se může dostat do stavu, kdy podměty způsobují zaměření agenta na různé cíle, kterou nejsou slučitelné a tím pádem ani jeden z cílů nemůže být nikdy uskutečněn. Na druhou stranu příliš proaktivní agent se zaměří na jeden cíl a kvůli tomu ignoruje, že počáteční podmínky, které ho k plnění cíle dovedly, už nejsou platné. Najít proto tu správnou míru je obtížná úloha.

Mezi sociální schopnosti nepatří jen rutinní výměna informací, kterou dnešní počítače provádějí prakticky neustále, ale především schopnost vyjednávat a spolupracovat při plnění společných nebo individuálních cílů. Mezi běžné akce tak patří

navrhování změn spolu s přínosem, které by tato změna přinesla, jejich zvažování a následné přijetí, odmítnutí či protinávrh vedoucí ke kompromisu.

(...pokračování...)

Kapitola 3

Algoritmus řízení

3.1 Návrh algoritmu

Úkolem v této práci bylo navrhnout algoritmus vyjednávání mezi jednotlivými křižovatkami tak, aby koordinovanou změnou svých offsetů vytvořili zelenou vlnu a tedy aby projelo co nejvíce vozidel bez zbytečného zastavování.

Jedním z nosných prvků návrhu je funkce, která ohodnotí konkrétní nastavení offsetu a umožní tak tedy porovnat různé jeho hodnoty a vybrat tu možná nejlepší. Za tuto míru kvality byl zvolen odhad počtu aut, která projedou křižovatkou bez zastavení.

Pro výpočet hodnocení (v programu nazývaném `rating`) je třeba pro každý jízdní pruh na vjezdech do křižovatky znát délku fronty a očekávané časy příjezdů vozidel od sousední křižovatky. Délka fronty se v současné době získává ze simulátoru, příjezdy pak přímo od souseda. Ten je počítá podle vztahu

$$t_z = t_{zz} + \frac{d}{v_P} + \mathit{offset} \tag{3.1}$$

a

$$t_k = t_z + t_{dz} \ , \tag{3.2}$$

kde t_z je čas začátku příjezdu aut a t_k čas konce. Dále pak t_{zz} je čas začátku svícení zelené, d vzdálenost ke křižovatce, která žádá o časy příjezdů, v_P průměrná rychlost vozidel, *offset* nastavený offset a t_{dz} délka svícení zelné. K těmto dvěma vypočítaným hodnotám se ještě přidává předpokládaný počet aut. Jeho odhad je získán z hustoty

provozu v minulém cyklu. Křižovatka zasílající odhady takovýchto trojic údajů předává několik – podle toho, kolik fází rozsvěcí zelenou ve směru připouštějícím jízdu k sousedovi žádajícímu o odhady.

Po shromáždění všech předpokladů od sousedů může agent přistoupit k samotnému výpočtu hodnocení. Ten spočívá v tom, že agent spočítá příspěvky všech jízdních pruhů, u kterých má nějaké informace o časech příjezdů. Pro každý pruh se pak nejprve spočítá délka fronty v čase, kdy se na jeho odjezdu rozsvítí zelená. To znamená:

$$Q_V = Q + \sum_{i=1}^{k} |T_i| \cdot n_i \tag{3.3}$$

kde Q_V je délka tak zvané virtuální fronty, zde jde o délku fronty při rozsvícení zelené, Q je odhadovaná délka fronty, $|T_i|$ je délka i-tého intervalu, ve kterém přijíždějí vozidla, n_i je počet těchto vozidel a k je počet intervalů s příjezdy před rozsvícením zelené.

Z tohoto odhadu pak vychází výpočet předpokládané délky fronty v okamžiku, kdy zelená přejde zpět na červenou. K tomu je třeba rozdělit interval od prvního očekávaného příjezdu nebo rozsvícení zelené (podle toho, co nastane dříve) po zhasnutí zelené (tento interval označíme T) na posloupnost intervalů $(T_i)_{i=1}^{k}$. T_i jsou intervaly typu $\langle a_i; b_i \rangle$, kde $b_i = a_{i+1}\ \forall i \in \{1, \ldots, k-1\}$, a_1 a b_k jsou krajní body intervalu T a a_i jsou chronologicky řazené všechny významné body, tedy body, ve kterých se mění signalizovaný znak nebo ve kterých začíná či končí příjezd vozidel od souseda.

S pomocí tohoto dělení pak můžeme počítat

$$Q_K = Q_V + \sum_{i=1}^{k} u(T_i)\,, \tag{3.4}$$

kde $u(T_i)$ je funkce $u:\ \langle a; b \rangle \to \mathbb{R}$

$$u(T_i) = \begin{cases} n_i & \text{pokud v intervalu } T_i \text{ pouze přijíždějí vozidla} \\ -|T_i| \cdot c_o & \text{pokud v intervalu } T_i \text{ pouze odjíždějí vozidla} \\ n_i - |T_i| \cdot c_o & \text{pokud v intervalu } T_i \text{ přijíždějí i odjíždějí vozidla} \end{cases} . \tag{3.5}$$

Q_K značí konečnou délku fronty, Q_V je virtuální fronta z rovnice (3.3), k je počet intervalů „s různými vjezdy a výjezdy“. $|T_i|$ je délka intervalu T_i, n_i počet vozidel,

která přijedou v intervalu T_i a c_o je empiricky zjištěná konstanta – počet vozidel, která za sekundu opustí křižovatku.

Definice funkce $u(T_i)$ jak je zapsána vztahem (3.5) není úplná. Ještě je nutné doplnit omezení

$$u(T_i) \begin{cases} \geq -Q_i & \text{pokud v intervalu } T_i \text{ pouze odjíždějí vozidla} \\ \leq n_i & \text{pokud v intervalu } T_i \text{ přijíždějí i odjíždějí vozidla} \end{cases}, \tag{3.6}$$

kde Q_i je délka fronty na začátku intervalu T_i.

Tyto podmínky zarčují, že pokud z jízdního pruhu jen odjíždějí vozidla, nemůže jich odjet více než jich čeká na začátku ve frontě a že pokud vozidla zároveň přijíždějí a odjíždějí, projede jich bez zastavení maximálně tolik, kolik dorazí od souseda.

Pokud vyjde Q_K záporné, představuje (odhadovaný) počet aut, která mohou křižovatkou projet bez zastavení a tato hodnota se tedy odečte od hodnocení dané křižovatky (a tím se hodnocení zvýší).

Vyvstává otázka jak zjistit aktuální délky fronta na jednotlivých ramenech jen pomocí údajů z detektorů, které jsou k dispozici. Ukazuje se, že toto není triviální problém a jeho řešení přesahuje rámec této práce. Z toho důvodu nejsou v programu délky front odhadovány tak, jak by to bylo nutné v reálné situaci, ale používají se hodnoty, které lze získat ze simulátoru Aimsun. Podrobnější informace o modelování délky fronty představuje například [3].

Při samotném vyjednávání pak mají agenti dvě role, jeden z nich je označen jako *pasivní*, zatímco druhý je v pozici aktivního. Pasivní agent má pevně nastavený offset a jen reaguje na pokyny aktivního. Pokyny tvoří buď žádost o očekávané časy příjezdů nebo návrh na změnu offsetu. Na první z nich agent vždy odpovídá zasláním požadovaných údajů, u druhého pak zvažuje, jestli by změna v celkovém součtu přinesla zlepšení `ratingu`. Z tohoto popisu už pak vyplývá role aktivního agenta. Ten stejně pracuje se žádostmi o časy příjezdů, navíc ale aktivně mění svůj offset a pokouší se vyjednat změnu u sousedů.

Vyjednávací cyklus pak probíhá v několika krocích. Nejprve si všichni agenti vyžádají očekávané příjezdy vozidel na základě offsetů nastavených v minulém cyklu. S přihlédnutím k těmto očekáváním pak aktivní agenti spočítají `rating` svého nastavení offsetu a pokusí se zjistit, jestli by nějaká změna vlastního offsetu nevedla ke

zlepšení hodnocení.

Hledání nejlepšího vlastního offsetu probíhá ve třech krocích. Nejprve se porovná aktuální offset, offset zvýšený o 8 sekund a offset snížený o 8 sekund. Vybere se nejlépe hodnocený ze tří možností a pokračuje se s ním stejným způsobem, jen další uvažovaná změna je ±4 sekundy. V posledním kroku je pak otestován posun o ±2 sekundy.

Když aktivní agenti naleznou své nejlepší offsety, rozešle se všem účastníkům zpráva o nalezení stabilního stavu, která obsahuje nové hodnoty očekávaných příjezdů vozidel. Nyní aktivní agenti vyzkouší, jestli by k dalšímu zlepšení nevedla změna offsetu u některého z jejich sousedů. Stejným způsobem jako při hledání vlastního nejlepšího offsetu zkusí odhadnout změnu `ratingu` při posunu sousedova offsetu o ± 4, 2 a 1 sekundu. Pokud má nejlepší hodnocení nenulová změna, zašle se sousedovi žádost o tuto změnu spolu se změnou `ratingu`, kterou by přinesla.

Každý pasivní agent poté sesbírá všechny návrhy a otestuje, který z nich má největší součet změny ratingu u něj samotného a změny u navrhovatele a ten potom přijme za vlastní. Pokud by všechny návrhy přinesly zápornou změnu, jsou zamítnuty a žádná změna nenastává. V každém případě jsou pak rozeslány informace o novém stabilním stavu a s nimi opět očekávané příjezdy.

Tímto každá křižovatka nalezne svůj konečný offset. Problém nastává při zaslání tohoto offsetu do řadiče křižovatky. Od toho není garantována okamžitá akce, způsob jakým žádaného offsetu dosáhne je jen v jeho režii a než se tak stane může trvat i několik cyklů. Z tohoto důvodu se vypočtený offset neposílá hned po nalezení, ale agent vždy v pěti cyklech napočítá optimální offset, z těchto pěti hodnot spočítá průměr a až ten se následně předá řadiči pro zpracování. Tím je také snížena reaktivnost agenta a zamezí se případným přehnaným reakcím na chvilkové změny poptávky, kterým stejně není možné se přizpůsobit.

3.2 Použité knihovny

Pro usnadnění práce a také z důvodu lepšího začlenění do současného řešení se v programu používají tři volně dostupné knihovny: `IT++`, zjednodušující práci s vektory,

maticemi a poli, `BDM` (Bayesian Decision Making), která obsahuje užitečné nástroje pro práci s popisy k vektorům a obsahuje také nástroj pro ukládání průběžných hodnoot z experimentu a `libconfig`, sloužící především pro práci s konfiguračními soubory. První dvě knihovny jsou distribuovány pod GPL licencí, třetí pak pod licencí LGPL.

3.2.1 IT++

`IT++` je knihovna pro C++, která obsahuje třídy a funkce pro provádění některých matematických operací, zpracování signálů a další. Pro účely této práce jsou zajímavé právě funkce matematické.

Pro zacházení s vektory jsou v knihovně mimo jiné zavedeny typy `vec` a `ivec`. První jmenovaný je vektor obsahující prvky typu `double`, tedy čísla s desetinou čárkou, druhý je pak složen z prvků `int`, čili čísel celých. Práce s vektory je velmi intuitivní, navíc lze často používat syntaxi podobnou jako v programu MATLAB.

Vektor můžeme nadefinovat jedním z těchto způsobů

```
vec my_vector;
vec my_vector(10);
```

přičemž první z nich pro vektor nealokuje paměť. To je pak nutné udělat funkcí setsize(). Následně je pak možné uložit do vektoru jednotlivé prvky a to například jedním z následujících příkazů.

```
vec   a = "0 0.7 5 9.3";          // tedy a = [0 0.7 5 9.3]
ivec  b = "0:5";                  // tedy b = [0 1 2 3 4 5]
vec   c = "3:2.5:13";             // tedy c = [3 5.5 8 10.5 13]
ivec  d = "1:3:5,0:2:4";          // tedy d = [1 3 5 0 2 4]
vec  e("1.2,3.4,5.6");            // tedy e = [1.2 3.4 5.6]
```

Navíc lze i-tému prvku vektoru `a` přistupovat pomocí a(i) nebo a[i].

Díky přetížení operátorů lze s vektory přímo provádět běžné matematické operace, jako jsou:

```
a+b       // součet vektorů
```

```
a+5      // přičtení čísla 5 ke všem prvkům vektoru
a*b      // skalární součin vektorů
a*9      // vynásobení všech prvků vektoru číslem 9
```

a tak podobně.

Práce s maticemi pak funguje podle obdobných pravidel.

Dále se v knihovně nachází třída `Array`. Díky ní je možné vytvářet pole prvků libovolného typu (včetně výše popsaných vektorů) a provádět s nimi mnohé operace. Příkladem užití právě pro pole vektorů je například následující kód:

```
Array<vec> pole_vektoru(2);
my_vec_array(0) = "2 4 5 0 3";;
my_vec_array(1) = "0.1 0.2 0.3 0.4 0.3 0.2 0.1";
```

U takto definovaného pole pak lze pak intuitivně přistupovat k jeho prvkům pomocí pole_vektoru(i) a navíc provádět operace jako jsou posuny prvků, výběr podmnožiny prvků nebo třeba prostředního prvku a mnohé další.

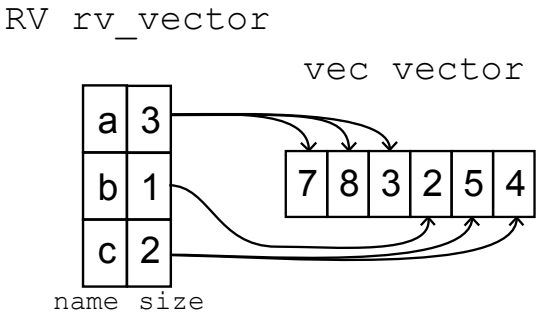
3.2.2 BDM

Knihovna `BDM` (Bayesian Decision Making) se zabývá, jak název napovídá, bayesovským rozhodováním. Ovšem v této práci jsou z ní přímo použity jen některé třídy, jmenovitě `UI`, `RV`, `datalink` a `logger`.

Třída `UI` (User Info) slouží pro ukládání a čtení libovolných uživatelských dat. Zde se používá pro načtení konfigurace pro simulátor a pro jednotlivé agenty a dále jako formát pro ukládání a posílání zpráv mezi agenty.

Účelem třídy `RV` je poskytovat popisné informace k datovému vektoru. Proměnná typu `RV` se skládá z jedné nebo několika položek. Každá taková položka má svoje jméno (`name`) a délku – počet prvků, které tomuto jménu odpovídají. Součet délek všech takovýchto položek se potom musí rovnat délce vektoru, který je danou proměnnou typu `RV` popisován. Fungování je ilustrováno na obrázku 3.1.

Pro kopírování dat mezi vektory existuje třída `datalink`. Vytváří spojení mezi



Obrázek 3.1: Vektor typu `RV` (vlevo) fungující jako popis k vektoru typu `vec` (vpravo). Každá položka z `rv_vector` má v levém sloupci své jméno (`name`) a v pravém velikost (`size`). V tomto případě se tedy podvektor `c` rovná vektoru [5 4]

dvěma datovými vektory na základně shodně pojmenovaných prvků v k nim příslušných popisných vektorech. Po propojení vektoru s podvektorem pak funkce datalinku umožňují snadné kopírování dat oběma směry.

Na závěr `Logger` je třída určená pro ukládání dat z programu. Tvoří abstraktní vrstvu mezi programem a samotným zápisem dat. Při použití se jen na začátku nastaví v jaké formě chceme získat výsledné údaje (např. uložit do paměti, do souboru, do databáze, atd.) a dále třídu používáme bez ohledu na tuto volbu. Je tím zajištěna flexibilita pro případ, že se změní požadavky na výstupy z programu.

3.2.3 Libconfig

Konfigurační parametry pro simulaci se ukládají do souboru ve formátu, který zavádí knihovna `libconfig`. Jde o textový formát, který je stručnější a pro člověka lépe čitelný, než XML.

(...rozvést nebo zrušit...)

3.3 VGS API

Velmi důležitým úkolem při simulaci reálné oblasti v mikrosimulátoru Aimsun je vložení reálných vstupních intenzit dopravy do zkoumaného modelu. Běžným po-

stupem je ruční sčítání vozidel v dané oblasti, zpravidla v hodinovém rastru. Takto získaná data je možné vložit do simulátoru Aimsun jako hodinové zátěže, ovšem toto je třeba provést také ručně.

Přesnější možností je získat intenzity provozu z dat získaných přímo z dopravních detektorů instalovaných v předmětné oblasti. To však znamená ruční zadání stovek údajů do simulátoru. Právě z tohoto důvodu vzniklo VGS API. To se stará o nastartování Aimsunu a vpouštění vozidel do oblasti. Vjezdy vozidel se při tom řídí údaji v externím souboru, obsahujícím intenzity na jednotlivých ramenech. Uživatel pak jen zadá jméno a umístění tohoto souboru a vše ostatní se děje automaticky. Navíc lze volit mezi různými rozdělení pravděpodobnosti vjezdů, přičemž nejčastěji používané je rovnoměrné či Poissonovo rozdělení.

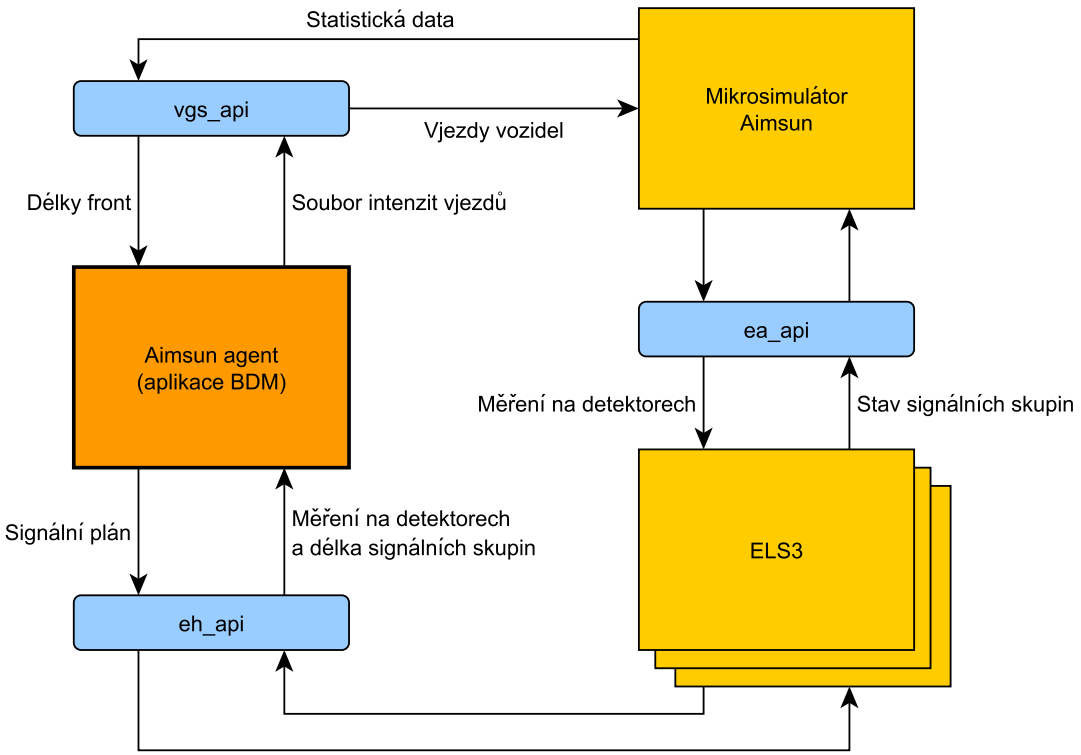
Vedle práci se vstupní daty pro simulaci se VGS API stará i o zpracování dat výstupních. Aimsun sice zvládá export výsledků simulací do textových souborů a obsahuje i nástroje pro jejich vizualizaci, neumožňuje přímo ale některé důležité věci jako je například porovnání výsledků ze dvou různých scénářů. VGS proto v průběhu celého experimentu ukládá údaje o sledované oblasti a to jak pro jednotlivé sekce, tak pro celý systém. Ve výsledku uživatel získává informace o počtu zastavení vozidel, o jeho zpoždění, průměrné rychlosti, době jízdy a době stání, o dopravím toku a hustotě dopravy na jednotlivých segmentech či o délkách front vozidel. Údaje o délkách front jsou specialitou VGS, Aimsun tento údaj pro jednotlivé jízdní pruhy přímo neposkytuje a VGS má proto implementován vlastní algoritmus pro jejich sčítání.

Implementací VGS API je DLL knihovna napsaná v jazyce C pro 32 a 64 bitové systémy Windows XP a výše. Navíc je součástí VGStoolbox, sada skriptů pro zpracování výstupních dat v programu Matlab.

3.4 Struktura programu

Fungování simulačního prostředí tak jak bylo navrženo v ÚTIA je na obrázku (3.2). Skladá se ze tří hlavních bloků, mikrosimulátoru Aimsun, emulátorů řadičů ELS3 a z tzv. Aimsun agenta.

Aimsun agent představuje řídící prvek celého systému. Přes `vgs_api` spouští Aim-



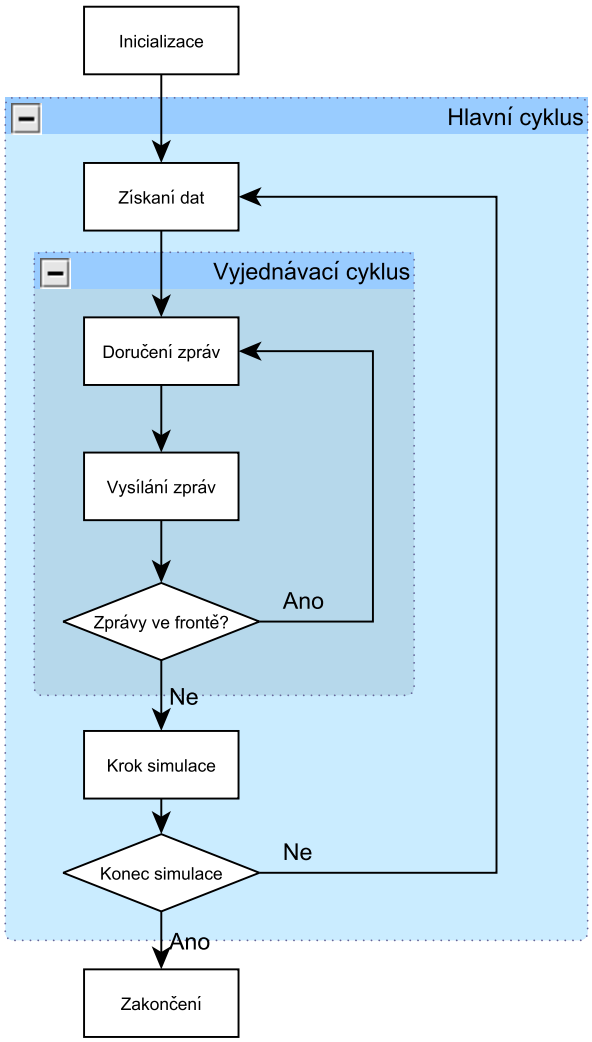
Obrázek 3.2: Struktura celé simulace

sun a stejným způsobem od něj průběžně přijímá hodnoty sledovaných dopravních veličin. Dále prostřednictvím `eh_api` získává údaje z řadičů křižovatek. Na základě všech sesbíraných údajů pak sestavuje pokyny pro změny v signálních plánech a opět pomocí `eh_api` tyto pokyny zasílá řadičům.

Na závěr komunikace mezi mikrosimulátorem a emulátory řadičů, které obsahuje údaje z detektorů v jednom směru a ze stavů signálních skupin ve druhém, probíhá přes `ea_api`.

3.5 Program `main_loop.exe`

Simulace je obsluhována z programu *main_loop.exe*. Ten se stará o načtení konfigurace, spuštění simulátoru Aimsun a zajišťuje komunikačního prostředníka mezi jednotlivými agenty. Program se dá rozdělit na jednotlivé bloky jak je zobrazeno na obrázku 3.3. Z tohoto nákresu bude vycházet následující popis.



Obrázek 3.3: Strukuta fungování programu `main_loop.exe`, podrobněji popsaná v podkapitole 3.5

Během *inicializace* je načten konfigurační soubor, jehož jméno se předává jako jediný parametru při spouštění `main_loop.exe`. Konfigurační soubor je ve formátu stanoveném knihovnou `libconfig`. Po načtení jsou parametry ze skupiny `system` předány VGS API, které se postará o spuštění Aimsunu s těmito parametry. Jde především o intenzitu provozu na vstupech do dopravní sítě a délku simulace.

Následně je dle konfigurace vytvořeno pole ukazatelů na agenty `Ags`. Při konstrukci jednotlivých agentů je volána jejich funkce `from_setting()` načítající konkrétní parametry každého agenta.

Poté se vytváří instance třídy `logger` a její provázání s agenty. V tu chvíli se pouští

i funkce `ds_register()` umožňující aktualizaci datalinků pro spojení mezi agenty a simulátorem. Na závěr tohoto bloku programu je provedena inicializace fronty zpráv a zaregistrování vektorů které se budou v průběhu simulace logovat.

Díle se pokračuje ústředním for cyklem. Část `získání dat` v sobě zahrnuje uložení logovaných hodnot ze simulátoru a pak především uložení vektor d_t do proměnné `glob_dt` a jeho odeslání agentům ke zpracování funkcí `adapt()`. Poté přichází na řadu vyjednávací cyklus.

Vyjednávací cyklus se zabývá obsluhou fronty zpráv. Nejprve proběhne *doručení zpráv*, které jsou aktuálně ve frontě. V prvním kroku vyjednávacího cyklu je fronta zpráv samozřejmě prázdná, v následujícím se od konce prochází a adresovaným agentům jsou předány příslušné zprávy. To se děje zavoláním funkce `recieve()` se zprávou jako parametrem. V případě, že adresát není nalezen a zprávu tedy nelze doručit je oznámeno varování, ale pokračuje se dál v činnosti.

Po vyprázdnění fronty nastává čas pro *vysílání zpráv*. To spočívá ve spuštění funkce `broadcast()` u všech agentů. V ní mají příležitost vložit do fronty zpráv libovolný počet sdělení pro ostatní agenty, nemusí ovšem vysílat žádné.

Další postup závisí na stavu fronty zpráv. Pokud je prázdná, tedy žádný agent nevyslal žádnou zprávu, končí vyjednávací cyklus a pokračuje se v běhu programu. V případě, že fronta prázdná není, cyklus se opakuje a proběhne další doručování zpráv z fronty. Vyjednávání může být ukončeno ještě v případě, kdy proběhne stanovený maximální počet vyjednávacích cyklů, nastavený v proměnné `max_cycles`. Limit by však měl být nastaven tak, aby k jeho dosažení nedocházelo, pokud se tak stane, mělo by to být signálem problému v programu.

Poslední část hlavního cyklu je nazvaná *krok simulace*. Spočívá nejprve v zavolání funkce `act()` u každého agenta. To je okamžik, kdy mohou agenti ovlivnit řízení signalizace. Právě zde probíhá zkopírování vypočítaných hodnot do vektoru `glob_ut`, tedy do proměnné představující vektor vstupních hodnot u_t. Na závěr hlavního cyklu je celá simulace posunuta o jeden krok dále voláním funkcí `step()` u loggeru `L`, `Ds` a u všech agentů.

Tato smyčka se opakuje dokud není dosažen v konfiguračním souboru nastavený čas simulace. *Zakončení* spočívá již jen v zapsání log souboru.

3.6 Implementace navrženého algoritmu

Algoritmus popsaný v kapitole 3.1 je implementován ve třídě `GreenWaveTrafficAgent`. Instance této třídy představují agenty přiřazené k jednotlivým křižovatkám (respektive jejím řadičům).

Třída je odvozena od předka jménem `BaseTrafficAgent`. Ten slouží jen jako prakticky prázdný agent, po přiřazení k nějaké křižovatce nekoná žádnou akci a umožňuje tak simulovat systém bez decentralizovaného řízení. Předkem této třídy je třída `Participant` z knihovny BDM. `Participant`, tedy účastník, představuje základní jednotku v rozhodovacím procesu. Každý účastník má své jméno (`name`) a je schopen ukládat výsledky. Už právě třída `Participant` deklaruje existenci funkcí `adapt()`, `broadcast()`, `recieve()`, `act()` a `step()`. Tyto jsou v něm definovány jako virtuální (`virtual`), tedy pokud je v potomkovi redefinujeme, je zaručeno volání těchto nových funkcí.

První po vytvoření instance agenta proběhne zavolání funkce `from_setting()` vypsanou jako kód 3.1. V té se nejprve volá stejnojmenná funkce předka, která načítá z konfigurace základní parametry, jmenovitě:

lanes informace o jízdních pruzích,

neighbours – jména agentů u sousedních křižovatek,

green_names – jména jednotlivých signálních skupin,

green_starts – čas, ve který příslušné signální skupiny rozsvěcí zelenou,

green_times – délka trvání svícení zelené dané signální skupiny, uváděna jako poměr doba svícení/délka cyklu,

a některé další, které pro tuto práci nejsou důležité.

Dále se v této funkci nastavují hodnoty konstant a výchozí hodnoty některých proměnných. Konkrétně jde o

car_leaving – čas v sekundách, ze který jedno auto opustí frontu

VP – průmerná rychlost jízdy automobilu mezi křižovatkami na volné silnici

actual_time – uchovává aktuální čas simulace (v sekundách)

negot_cycle – počet již proběhlých vyjednávacích cyklů od posledního (! zformulovat !)

cycle_count – hranice, po které nastává průměrování konečného offsetu

total_offset – hodnota součtu dosud vyjednaných offsetů od posledního průměrování

negot_offset – krok, kterým se začíná při hledání ideálního offsetu u souseda

negot_limit – krok, kterým se končí při hledání ideálního offsetu u souseda

passive – indikuje zda agent zastává pasivní nebo aktivní roli

V závěru jsou je pak z konfigurace načten výchozí offset, role agenta a připraven vektor výstupních hodnost `outputs` spolu s jeho popisem `rv_outputs`. Na úplném konci je ještě zapnuto logování hodnot offsetů pro pozdější vyhodnocení.

```
587  void from_setting(const Setting &set) {
588    BaseTrafficAgent::from_setting(set);
589
590    RV rv_exp;
591
592    car_leaving=2;
593    VP=45;
594    actual_time=0;
595
596    negot_cycle=1;
597    cycle_count=5;
598    total_offset=0;
599
600    negot_offset=4;
601    negot_limit=1;
602
603    passive=0;
604
```

```
605     UI::get(last_offset, set, "offset", UI::compulsory);
606     UI::get(passive, set, "passive", UI::optional);
607
608     for (int i=0;i<lanes.length();i++) {
609       for (int j=0;j<lanes(i).outputs.length();j++) {
610         if (lanes(i).outputs(j)!="DUMMY_DET") {
611           rv_exp=RV(lanes(i).outputs(j)+"-"+name+"_"+
    to_string(i),3);
612           rv_outputs.add(rv_exp);
613         }
614       }
615     }
616     outputs.set_size(rv_outputs._dsize());
617
618     log_level[logoffset]=true;
619   }
```

Kód 3.1: Funkce from_setting()

Další volanou funkcí v agentech je `adapt()`. Podobně jako `from_setting()` nejprve spouští stejnomenou funkci u svého předka. V té probíhá vyplnění vektorů `inputs` a `queues` daty z Aimsunu, uloženými ve vektoru `glob_dt`. Údaje z `queues` se předají příslušným `LaneHandler`ům. Dále je nastaven `planned_offset` na hodnotu získanou v minulém cyklu a stavové proměnné jsou nastaveny do svých výchozích hodnot.

```
415   void adapt(const vec &glob_dt) {
416     BaseTrafficAgent::adapt(glob_dt);
417
418     for (int i=0;i<lanehs.length();i++) {
419       lanehs(i)->queue=queues(lanehs(i)->queue_index);
420     }
421
422     planned_offset=last_offset;
423
424     //set state variables to default values
425     final_state=false;
426     new_stable_state=false;
427     send_requests=false;
```

```
428     need_exps=true;
429     negot_offset=4;
430   }
```

Kód 3.2: Funkce adapt()

Funkce `recieve` zpracovává přijaté zprávy od ostatních agentů. Na začátku je přijatá zpráva rozdělena do jednotlivých proměnných.

```
491   void receive(const Setting &msg){
492     string what;
493     string to;
494     string from;
495     vec value;
496     RV *rv;
497
498     UI::get(what, msg, "what", UI::compulsory);
499     UI::get(to, msg, "to", UI::compulsory);
500     UI::get(from, msg, "from");
501     UI::get(rv, msg, "rv");
502     UI::get(value, msg, "value");
```

Kód 3.3: Fragment funkce `recieve()`, inicializace

Dále se činnost liší podle „předmětu“ zprávy, tedy podle řetězce uloženého v části `what`. V případě přijetí žádosti o očekávané časy příjezdů je jen uloženo jméno odesílatele do seznamu žadatelů `requesters`.

```
503     if (what=="expected_times_request"){
504       requesters.push_back(from);
505     }
```

Kód 3.4: Fragment funkce `recieve()`

Pokud jsou naopak obsahem přijaté zprávy očekávané časy příjezdů automobilů od souseda, aktivní agent najde nový optimální offset pro svůj signální plán a spočítá `rating`, pasivní provede jen tento výpočet. Na konci je stavová proměnná nastavena `new_stable_state` na `true`, což určuje další činnost agent ve fázi vysílání zpráv.

```
507     else if (what=="new_expected_cars") {
```

```
508       rv_recieved_exps=*rv;
509       recieved_exps=value;
510
511       if (!passive) {
512         planned_offset=find_best_offset(planned_offset,8);
513         planned_offset=normalize_offset(planned_offset);
514       }
515
516       planned_rating=count_rating(planned_offset,
      recieved_exps, rv_recieved_exps);
517
518       // we have new stable state to broadcast
519       new_stable_state=true;
520     }
```

Kód 3.5: Fragment funkce `recieve()`

Obdržení zprávy s hlavičkou „`stable_state`“ znamená, že některý ze sousedů změnil své nastavení offsetu a zasílá tedy nové hodnoty očekávaných příjezdů. Pokud je aktuální hodnota `negot_offset` větší nebo rovna `negot_limit`, zkouší aktivní, která z hodnot 0, +`negot_offset` nebo −`negot_offset` by po přičtení k sousedově offsetu znamenala nejlepší rating. Hodnota `negot_offset` je snížena na polovinu a je připraveno odeslání žádosti pro souseda. Pokud Pokud byla proměnná `negot_offset` již nižší než než `negot_limit`, přepíná se agent do konečného stavu, neb již nemá další prostor k vyjednávání se sousedy.

```
521     else if (what=="stable_state") {
522       rv_recieved_exps=*rv;
523       recieved_exps=value;
524
525       planned_rating=count_rating(planned_offset,
      recieved_exps, rv_recieved_exps);
526
527       if (!passive) {
528         for (int i=0;i<neighbours.length();i++) {
529           rv_change_request.add(RV(neighbours(i)+"_change"
      ,2));
530           change_request.set_size(rv_change_request._dsize()
```

```
    );
531         ivec ind=RV(neighbours(i)+"_change",2).dataind(
    rv_change_request);
532
533         // offset change
534         change_request(ind(0))=find_best_exps(negot_offset
    ,neighbours(i),rating_change);
535         // rating change
536         change_request(ind(1))=rating_change;
537       }
538
539       if (negot_offset>=negot_limit) {
540         negot_offset/=2;
541         send_requests=true;
542       }
543       else {
544         final_state=true;
545       }
546     }
547     else {
548       final_state=true;
549     }
550   }
```

Kód 3.6: Fragment funkce `recieve()`

Při přijetí se zprávy s žádostí o změnu offsetu agent zkoumá, zda je součet změny hodnocení po aplikaci navrženého offsetu a zlepšení hodnocení u souseda, které od změny offsetu očekává, větší než nula. Takováto změna se přijímá a podle ní se upravuje `planned_offset`. V každém případě se agent připraví na zaslání aktuálních hodnot očekávaných příjezdů.

```
551   else if (what=="offset_change_request") {
552     double final_rating_diff;
553
554     rv_recieved_changes=*rv;
555     recieved_changes=value;
556
```

```
557       for (int i=0;i<rv_recieved_changes.length();i++) {
558
559         ivec ind=RV(rv_recieved_changes.name(i),2).dataind(
      rv_recieved_changes);
560
561         final_rating_diff=-planned_rating+count_rating(
      planned_offset+(int)recieved_changes(ind(0)),
      recieved_exps, rv_recieved_exps)+recieved_changes(ind(1))
      ;
562         if (final_rating_diff>=0) {
563           planned_offset+=(int)recieved_changes(ind(0));
564           planned_offset=normalize_offset(planned_offset);
565           planned_rating+=final_rating_diff;
566         }
567       }
568
569       new_stable_state=true;
570     }
```

Kód 3.7: Fragment funkce `recieve()`

Pokud přijde zpráva s předmětem, který `GreenWaveTrafficAgent` neumí zpracovat, předává ji předkovi. Kdyby se stalo, že ani ten si se zprávou neporadí, pošle se zpráva jeho předkovai (tedy třídě `bdm::Participant`), který v takovém případě vyvolá varování o nezpracovatelné zprávě.

```
571     else {
572       BaseTrafficAgent::receive(msg);
573     }
574   }
```

Kód 3.8: Fragment funkce `recieve()`

Funkce `broadcast()` vysílá zprávy do fronty zpráv a to v závislosti na hodnotách agentových stavových proměnných. Pokud není prázdný seznam `requests`, sestaví odpověď pro každého z agentů uvedeného v seznamu a ten následně vyprázdní. Dále pak za každou ze stavových proměnných, která mají hodnotu `true` sestaví příslušnou zprávu a hodnotu změní na `false`.

```
429 void broadcast(Setting& set){
430
431   //ask neighbours for exptected arrive times
432   if (need_exps) {
433     for (int i=0; i<neighbours.length(); i++){
434       Setting &msg =set.add(Setting::TypeGroup);
435       UI::save ( neighbours(i), msg, "to");
436       UI::save (name,msg,"from");
437       UI::save ( (string)"expected_times_request", msg, "
    what");
438     }
439     need_exps=false;
440   }
441
442   // broadcast expected cars
443   if (!requesters.empty()) {
444     double a;
445     expected_cars();
446     do {
447       Setting &msg =set.add(Setting::TypeGroup);
448       UI::save ( requesters.back(), msg, "to");
449       UI::save ( name, msg, "from");
450       UI::save ( (string)"new_expected_cars", msg, "what")
    ;
451       UI::save ( &(rv_outputs), msg, "rv");
452       UI::save ( outputs, msg, "value");
453       requesters.pop_back();
454       a=outputs (10);
455     } while (!requesters.empty());
456   }
457
458   // broadcast new stable state (new stable expectations)
459   if (new_stable_state) {
460     expected_cars();
461     for (int i=0;i<neighbours.length();i++) {
462       Setting &msg = set.add(Setting::TypeGroup);
```

```
463         UI::save ( neighbours(i), msg, "to");
464         UI::save ( name, msg, "from");
465         UI::save ( (string)"stable_state", msg, "what");
466         UI::save ( &(rv_outputs), msg, "rv");
467         UI::save ( outputs, msg, "value");
468       }
469       new_stable_state=false;
470     }
471
472     // broadcast requests to change offset(s)
473     if (send_requests) {
474       for (int i=0;i<neighbours.length();i++) {
475         Setting &msg = set.add(Setting::TypeGroup);
476         UI::save ( neighbours(i), msg, "to");
477         UI::save ( name, msg, "from");
478         UI::save ( (string)"offset_change_request", msg, "
    what");
479         UI::save ( &(rv_change_request), msg, "rv");
480         UI::save ( change_request, msg, "value");
481       }
482       send_requests=false;
483     }
484
485     // reached final offset.
486     if (final_state) {
487       final_state=false;
488     }
489   }
```

Kód 3.9: Funkce broadcast()

Ve funkci `act()` se spočítaná hodnota `planned_offset` přičítá k hodnotě proměnné `total_offset` a to až do chvíle, než je v něm tolik hodnot, kolik stanovuje limit `cycle_count`. Když se tohoto limitu dosáhne, vydělí se `total_offset` tímto limitem, čímž se získá průměrná hodnotná offsetu z posledních cyklů. Průměr se pak znormalizuje, aby byl z intervalu $\langle 0; T_c \rangle$ a uloží se do vektoru u_t, představovaným proměnou `global_ut`. Tím se tato hodnota dostane k řadiči křižovatky.

```
615 void act(vec &glob_ut){
616   if (negot_cycle==cycle_count) {
617     vec action;
618     action.set_size(rv_action._dsize());
619
620     ivec index = RV(name+"_offset",1).dataind(rv_action);
621
622     action(index(0))=normalize_offset(total_offset/
    cycle_count, false);
623     action2ds.filldown(action,glob_ut);
624
625     total_offset=0;
626     negot_cycle=1;
627   }
628   else {
629     total_offset+=planned_offset;
630     negot_cycle++;
631   }
632   last_offset=planned_offset;
633 }
```

Kód 3.10: Funkce act()

O zápis aktuálních hodnot do logovacího souboru se stará `log_write()`. Ukládá se dvousložkový vektor. První prvek obsahuje vypočítaný offset `planned_offset`, druhý jeho hodnocení `planned_rating`.

```
644 void log_write() const {
645   if (log_level[logoffset]){
646     vec offset_vec(2);
647     offset_vec(0)=planned_offset;
648     offset_vec(1)=planned_rating;
649     log_level.store(logoffset, offset_vec);
650   }
651 }
```

Kód 3.11: Funkce log_write()

Poslední volaná veřejná funkce agenta je `step()`, která jen posouvá interní ukazatel času v agentovi o délku kroku simulace.

```
635  void step() {
636    actual_time+=step_length;
637  }
```

Kód 3.12: Funkce step()

Kapitola 4

Výsledky simulací

Závěr

Seznam použitých zdrojů

[1] Bazzan, A. L. C.: Opportunities for multiagent systems and multiagent reinforcement learning in traffic control. *Auton Agent Multi-Agent Syst*, 2009: s. 342–375.

[2] Lindner, M. A.: *libconfig - A Library For Manipulating Structured Configuration Files*. Říjen 2007.

[3] Pecherková, P.; Duník, J.; Flídr, M.: *Robotics, Automation and Control*, kapitola Modelling and Simultaneous Estimation of State and Parameters of Traffic System. InTech, Croatia, 2008, ISBN 978-953-7619-18-3, s. 319–336.

[4] Roozemond, D. A.: Using intelligent agents for pro-active, real-time urban intersection control. *European Journal o Operational Research*, 2001: s. 293–301.

[5] TSS: *Getram v4.2 getting started - User's manual*. Říjen 2003.

[6] TSS: *GETRAM Extensions VERSION 4.2 - User's manual*. Květen 2004.

[7] TSS: *Getram v4.2 - User manual*. Únor 2004.

[8] Weiss, G. (editor): *Multiagent systems: a modern approach to distribuated artificial intelligence*. MIT Press, 1999, ISBN 0-262-23203-0.

Přílohy