FAKULTA INFORMATIKY A MANAGEMENTU
UNIVERZITA HRADEC KRÁLOVÉ

Grafové databáze v muti-agentovém systému

Forma studia:	Denní
Obor:	AI3
Ročník:	3. ročník
Vedoucí práce:	Ing. Barbora Tesařová, Ph.D.
Datum zpracování:	19. 4 2016

Autor:	Radek Salay

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové Radek Salay

Poděkování:

Tímto bych chtěl poděkovat paní Ing. Barboře Tesařové, Ph.D. za odborné vedení, poskytnuté informace a cenné rady při zpracování mé bakalářské práce.

Anotace

Práce je zaměřena na grafové databáze v multi-agentovém systému a začlenění grafové databáze do projektu ASE, vyvíjeném na UHK, ve kterém se provádějí měření a testování hypotéz. Cílem této práce je porovnání dvou různých způsobů práce v multi-agentním systému. Jedním způsobem jsou grafové databáze a druhým přístupem bude uchovávání a práce s daty přímo v simulaci ASE, pomocí několika tabulek.

Součástí práce je výsledná zpráva z měření obou způsobů práce s daty, nástin možné spolupráce ASE s Neo4j a postřehy při práci s databázovým jazykem Cypher.

Klíčová slova

Grafová databáze, Neo4j, Multi-agentní simulace, Graf, Testování grafové databáze

Annotation

Title: Graph database in multi-agent systém

Bachelor thesis is focused on graph database in multi-agent system and integration of graph database to project ASE developing on UHK, in which are doing mensuration and testing hypothesis. Goal of this bachelor thesis is comparison two difrend way of work in multi-agent system. First way is graph database and second way will be store and work with data directly in symulation ASE using severel tables.

Keywords

Graph database, Neo4j, Multi-agentn simulation, Graph, Testing gaph database

Obsah

1. Úvod

Multi-agentní systémy, které vycházejí ze snah o vytvoření umělé inteligence, jsou dnes velmi rozšířeným vědním oborem, s širokým uplatněním na poli ekonomiky, biologie a IT. S myšlenkou umělé inteligence si lidstvo hraje už velmi dlouhou dobu. Dnes nejpoužívanější jméno takovým strojům dal však až Karel Čapek ve své hře R.U.R (7) . Od té doby se technika velmi posunula. Místo lidem podobným strojům, se čím dál více zaměřujeme, na lidem podobně smýšlející roboty - soft-boty[1] . Tito virtuální roboti dosahují stále častějších úspěchů.

Lidé tvoří multi-agentní systémy nejčastěji proto, aby za ně vyřešili jejich problémy. Avšak tyto problémy jsou reprezentovány jistými daty, ke kterým je potřeba se nějak dostat. Tradičním způsobem jsou tabulky, které se používají již desetiletí. A popravdě pořád jsou dostačující pro většinu úloh. Jenže s narůstajícím množstvím dat a jejich komplexitou, ve smyslu vazeb mezi danými daty, se samotné tabulky stávají nepoužitelnými. Jako řešením byly vyvinuty No-SQL[2] databázové systémy. Tyto nově vzniklé systémy se skládají z více podkapitol, kde každá má své vlastní řešení. Tato práce se zabývá využitím No-SQL v podobě grafových databází.

Grafová databáze se v této práci bude implementovat do multi-agentové simulace ASE[3].

V první kapitole se budeme věnovat vysvětlení základních pojmů a teoretickým základům multi-agentních systémů, na které dále navážeme, a které budou stěžejní v následujících kapitolách.

ASE je projekt Univerzity Hradec Králové, přesněji Fakulty Informatiky a managementu, pod vedením Dr. Pavla Čecha. Tento projekt započali a publikovali v diplomových pracích v roce 2014 Ing. Lulek, který se věnoval problematice komunikační vrstvy a Ing. Vlach (5), který se

[1] Robot, který je tvořený pouze virtuálně. Bez fyzické schránky.

[2] Ve významu not onli SQL (ne jenom SQL)

[3] ASE (Agent System Enviroment)

v práci zmiňuje o synchronizaci agentů. Na základech těchto prací se projekt ASE dále vyvíjel. O jeho vývoji a současném stavu budeme hovořit ve druhé kapitole.

Později byla do ASE přidána grafová databáze, která je hlavním tématem bakalářské práce. Základy teorie grafů pro přiblížení této problematiky, jsou obsahem třetí kapitoly.

Cílem této práce je seznámit čtenáře s fungováním simulace a jejím účelem, přiblížení celkové problematiky multi-agentních systémů, začlenění grafové databáze do projektu ASE a nástin fungování grafových databází.

Dalším cílem je porovnání účinnosti obou dvou přístupů k získávání dat. Jedním přístupem budou již zmiňované grafové databáze a druhým přístupem bude uchovávání a práce s daty přímo v simulaci ASE, pomocí několika tabulek.

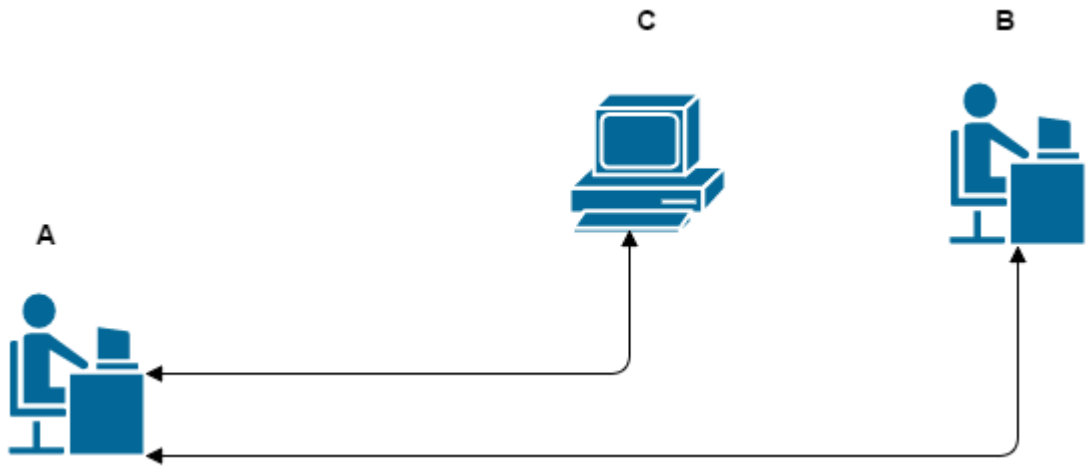
Autor práce předpokládá velmi základní znalosti v oboru informačních technologií a snaží se danou problematiku co nejvíce přiblížit neznalému čtenáři.

2. Multi-agentní systémy

Multi-agentní systémy jsou součástí vývoje umělé inteligence. Tato snaha o vytvoření umělé inteligence sahá až do roku 1950, kdy první test pro takovou inteligenci popsal ve své knize Alan Turing (2) .

Test je na principu, že inteligentní objekt **A** (člověk) pouze komunikuje a nevidí ostatní dva objekty. Prvním objektem, který nevidí, je člověk **B** a druhým objektem, který nevidí, je umělá inteligence **C** (počítač). Test je splněn, pokud **A** nedokáže rozeznat, kdo je člověk a kdo stroj. V tom případě je stroj opravdu inteligentní.

Tento test byl sice již v minulosti zpochybněn, jelikož se podařilo člověka **A** zmást počítačem, který nebyl inteligentní, ale dokázal velmi věrně komunikovat. Avšak princip zůstává a nádherným příkladem ještě efektivnějšího testu na tomto principu je film Ex Machina (13). Od Turingova teoretického testu se povedlo v oblasti umělé inteligence velmi pokročit, přesto jsme od vize skutečné umělé inteligence ještě daleko.



Obrázek 1 Turingův test, vlastní tvorba

Milníkem, který dal vzniknout multi-agentním systémům je rok 1988, kdy byla definována distribuovaná umělá inteligence:

„Distributed Artificial Inteligence (DAI) is the sub field of AI concernd with coordinated, concurent action and problem solving“

Wooldridge (16)

Z této definice vycházejí tři podskupiny:

- distribuované řešení problému
- paralelní umělá inteligence
- multi-agentní systémy

Řešit problémy rychleji a efektivněji, je vždy velmi žádané. Ať už v informatice, biologii nebo ekonomice. Multi-agentní systémy se v těchto oborech využívají především k tvorbě prostředí s danými pravidly a aktéry, kteří prostředí ovlivňují, díky čemuž můžeme studovat danou problematiku odkudkoli a jakkoli dlouho, bez omezení reálného světa. Využít této volnosti můžeme například v biologii, kde nás v takovém případě neomezuje ani to, že dané zvíře již vyhynulo. Časté využití MAS je také v případech, když dané prostředí nelze uchopit a je to čistě nehmotná věc, jako jsou například lidské vztahy a jejich provázanost.

1.1 Agent

V multi-agentních systémech hrají hlavní roli agenty[4]. Jedná se o základní jednotku systému. Agenty jsou dále nedělitelní. Ve smyslu abstrakce[5] by je nemělo cenu dále dělit. Skládají se však z různých částí, které budou dále specifikovány. Agenty dělíme na fyzické a virtuální. Pod fyzickým agentem si může čtenář představit třeba sám sebe. Virtuálního agenta je dost obtížné si představit. Naštěstí i virtuální agenty mají často předlohu v reálném světě. V ASE

[4] Používám neživotný tvar, jelikož práce pojednává o neživých agentech

[5] Zjednodušení problematiky na únosnou mez

je to například firma, kterou si musíme představit jako jednu nedělitelnou jednotku. Tudíž zaměstnanci, stroje a znalosti jsou zapouzdřeny v jedné entitě.

Definic, co přesně představuje agenta je několik a příliš se od sebe neliší:

> *"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors "*
>
> *(Russel & Norvig)(15)*

Podle Russela a Norviga je tedy agentem cokoli, o čem se dá říci, že vnímá své prostředí pomocí senzorů a jedná v prostředí pomocí efektorů.

> *„An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives“*
>
> *(Wooldridge, 2002)(16)*

Wooldridge zase popisuje agenta jako počítačový systém, který je situovaný v nějakém prostředí a je schopný autonomních akcí s prostředím, aby splnil své cíle.

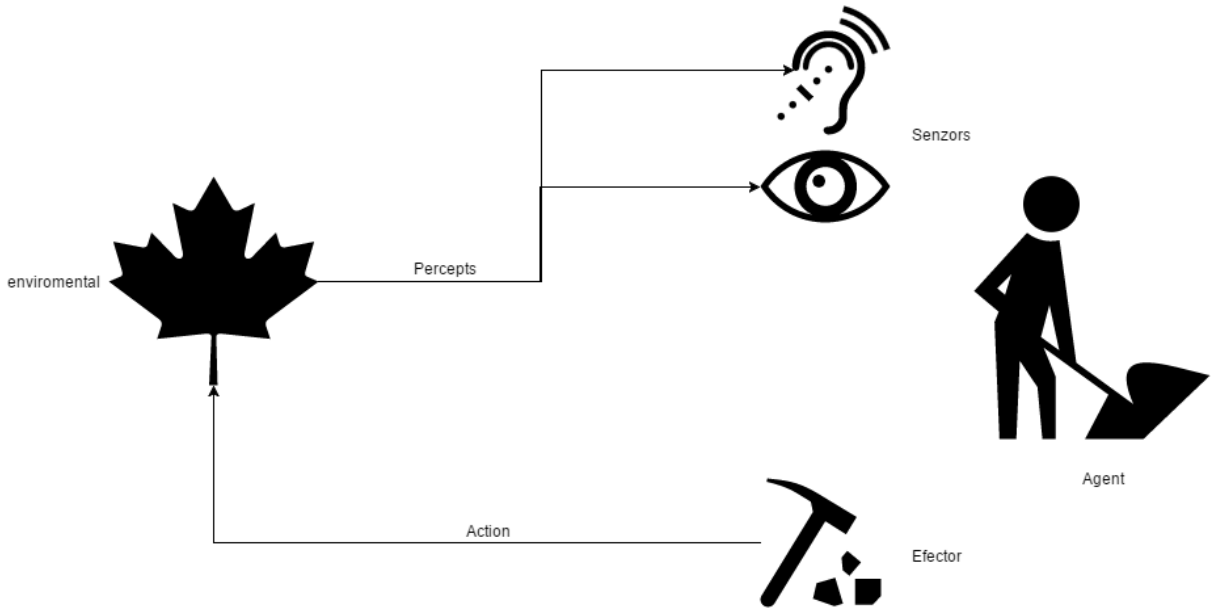
V této práci budeme spíše pracovat s definicí od Wooldridge, jelikož se více blíží definici pro softwarové agenty.

Senzory (Senzors) jsou části agenta, kterými vnímá okolní **prostředí** (Enviroment).

Vnímání okolí (Percepts) je nezbytné pro fungování multi-agentních systémů. Bez této vlastnosti by byly agenty jako neslyšící na koncertě. Senzory u fyzických agentů jsou například oči, zmíněné uši, kamery, nebo podobná zařízení.

Efektory (Efectors) představují na druhé straně agentovy nástroje, kterými ovlivňuje dané prostředí, popřípadě jiné agenty, kteří se v daném prostředí nacházejí. Agenty fyzického typu mají například ruce, nebo ramena pro uchopení objektů. Virtuální to mají o něco jednoduší –

jim stačí spustit příslušnou metodu s algoritmem. Schopnost **ovlivňovat své prostředí**(Actions) je pro agenty velmi důležitá, i když jí nemusí mít všichni.



Obrázek 2 Agent v simulaci, vlastní tvorba

1.1 Prostředí

Důležité pro funkci agentů je prostředí. Kdyby byl agentem člověk, tak by prostředím pro něj byl okolní svět, nebo město ve kterém žije. Prostředí v multi-agentních systémech má také své vlastnosti. To, jestli je prostředí virtuální či fyzické, nehraje příliš velkou roli.

V často citované knize Artificial Inteligent od Russela & Norviga(16) se prostředí přiřazují určité vlastnosti. Je důležité zmínit, že tyto vlastnosti se vztahují pouze k danému agentovy. Tudíž, pokud máme v multi-agentním systému více druhů agentů, každý může dané prostředí vnímat trochu jinak. Jako příklad můžeme uvést známé sousoší tří opic.

Jedna opice své prostředí nevidí, ale může ho slyšet a mluvit s ostatními. Druhá opice své prostředí neslyší, ale vidí ho a může s ostatními také mluvit. Třetí opice své prostředí vidí i slyší, ale nemůže s ostatními mluvit. Zároveň mohou všechny tři se svým prostředím manipulovat pomocí svých rukou. K těmto třem opicím však patří ještě méně známá čtvrtá opice, která své prostředí slyší, vidí, může s ostatními mluvit, ale nemůže používat ruce, tudíž

s prostředím nemůže nijak manipulovat. Tyto opice jsou skvělým příkladem, že to, co se může zdát jako jeden druh agenta - opice, jsou ve skutečnosti čtyři odlišné agenty. A proto je nutné, pro každého zvlášť definovat jeho vlastní vazby k danému prostředí.

Vlastnosti prostředí:

Russel & Norvig(16) ve své knize Artificial Inteligent definují následující parametry:

přístupný / nepřístupný

Přístupné prostředí je v okamžiku, kdy senzory agenta mohou zachytit celé prostředí, ve kterém se agent nachází. V reálném příkladě bychom se ptali, jestli myš v bludišti vidí celé bludiště. Jelikož myš vidí jen nejbližší stěny, je pro ní prostředí nepřístupné.

efektivně přístupné prostředí

Jedná se o postačující podmínku, při které stačí, když agent vnímá vše, co ovlivňuje jeho chování a rozhodování.

deterministické / stochastické

Prostředí je deterministické v případě, pokud agent dokáže plně předvídat následující stav ze stavu, ve kterém se právě nachází.

Tento atribut velmi souvisí s předchozím atributem, jestli agent dokáže celé prostředí sledovat. Avšak není jím plně ovlivněn.

epizodní / průběžné

Dalo by se říci, že následujícím atributem je vnímání času. Při epizodním prostředí se agent vždy nachází v určité, na ostatních nezávislé, epizodě. Toto prostředí může usnadnit agentovo jednání. Jelikož jsou epizody nezávislé, agent si nemusí uchovávat informace o předchozích epizodách. Stejně tak nemusí plánovat příliš dopředu. Délka potřebného plánování je určena délkou dané epizody.

Průběžné prostředí je takové, které se odehrává od začátku až do konce bez přerušení. Dalo by se tedy říct, že se jedná o jednu velkou epizodu.

Statické / dynamické

Zde hovoříme o proměnlivosti prostředí. O dynamickém prostředí hovoříme, pokud se prostředí mění bez agentova přičinění. Nejčastěji k tomu dochází, když agent není v takovém prostředí sám.

Statické prostředí je takové, které se mění pouze přičiněním daného agenta.

Můžeme se zároveň také setkat s postačující podmínkou, kdy se prostředí časem nemění (bez agentova přičinění), ale agentova výkonnost ano. Takové prostředí nazýváme ***polo dynamickým.***

diskrétní / spojité

Agenty v prostředí obyčejně provádějí nějakou funkcionalitu. Pohybují se v prostředí, nebo ho nějak ovlivňují. Jestliže je v tomto ohledu agent nějak limitován, například agent se může nacházet pouze na **N** místech, a $N \in \mathbb{N}$, tak je prostředí **diskrétní.**

Naproti tomu **spojité** prostředí je, když $N \in \mathbb{R}$.

Záleží opět na míře abstrakce, s jakou je prostředí modelované. Pokud je prostředí tvořeno kamerovým záznamem, který je z dále nedělitelných pixlů, dostáváme se do jistého sporu, jestli je pořád prostředí spojité. Toto se může měnit systém od systému.

1.2 Agenty

Dosud jsme se zabývali pouze „fyzickou“ stránkou agentů a také tím, v jakém prostředí se agenty pohybují. Nyní se budeme věnovat tomu, jak agenty řeší situace, se kterými se v svém prostředí setkají a jestli komunikují s ostatními agenty. Postup bude zdola nahoru. Od nejjednodušších typů agentů, až po ty se složitějšími mechanismy, jakými je plánování a komunikace s ostatními.

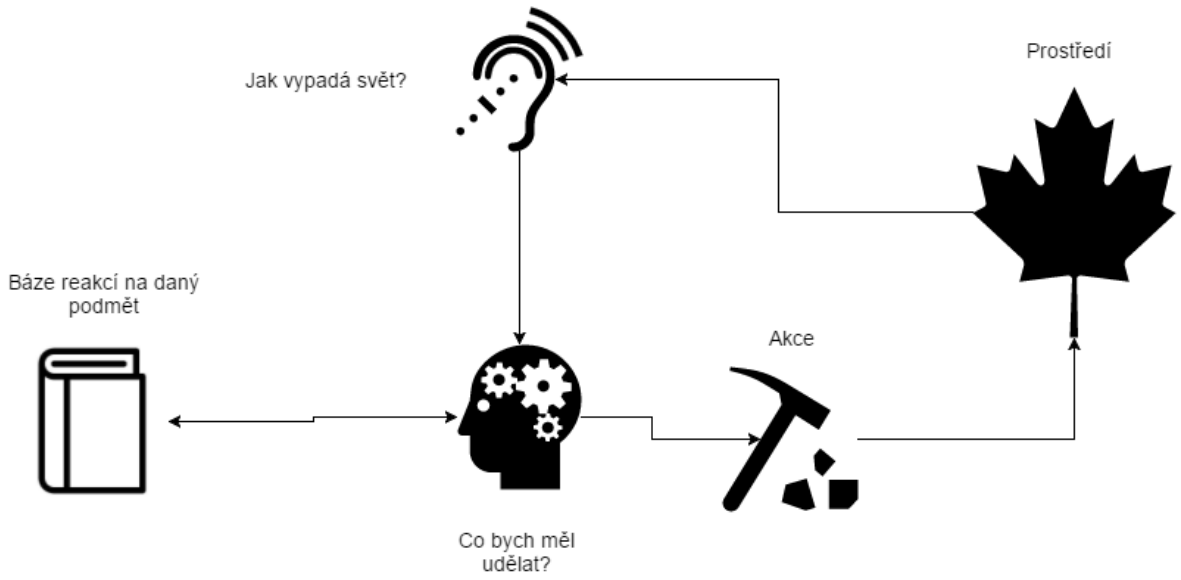
Agenty jsou podle Russela & Norviga(16) ***autonomní***, jestliže po vložení agenta do jeho prostředí, bude agent svou činnost vykonávat sám, bez zásahu zvenčí.

V případě sociálních agentů schopných komunikace, by se autonomnost ztratila pouze v případě, že by komunikovaly mimo prostředí s člověkem. Pokud by však byl člověk součást prostředí, ničemu by vadit neměl, protože by měl být brán jako další agent.

1.2.1 Čistě reaktivní/reflexní agenty

Agent s čistě reaktivní architekturou bývá označován za nejednoduší typ agenta. Takový agent má na každý podnět, který dostane z prostředí, předem připravenou akci. Nezaznamenává své stavy, protože ani nemá kam, neobsahuje totiž paměť. Často se skládá z několika nezávislých modulů, které dohromady tvoří celého agenta (9).

Každé akci náleží reakce $A_n \rightarrow R_n$.



Obrázek 3 Reflexní agent, vlastní tvorba

Nejznámějším příkladem takového agenta je Kelemenova beruška (6).

Jedná se o robotickou berušku. Ta je poháněna hlavním kolečkem pořád dopředu, což je jeden z jejích efektorů. Toto kolečko se nachází v zadní části berušky. Dalším beruščiným efektorem je kolečko o něco menší v přední části. Toto kolečko otáčí beruškou do prava. Avšak je ve vzduchu a musí se aktivovat. Dále má beruška senzor v podobě tykadla připevněného na konci hlavy. Toto tykadlo se dotýká stolu. Beruška se pohybuje v prostředí, které je reprezentováno stolem. Pokaždé, když se beruška dostane ke kraji stolu, tak se přední tykadlo přestane dotýkat desky stolu, protože tam stůl končí. Beruška se převáží, díky čemuž se přední kolečko dotkne stolu a vykoná svou funkci otočení berušky do prava. To, jestli je beruška inteligentní je sporné. Chová se sice inteligentně, nespadne ze stolu, ale ona sama o tom neví, ani to nedělá záměrně, jelikož se prostě jen převáží.

1.2.2 Uvažující agenty

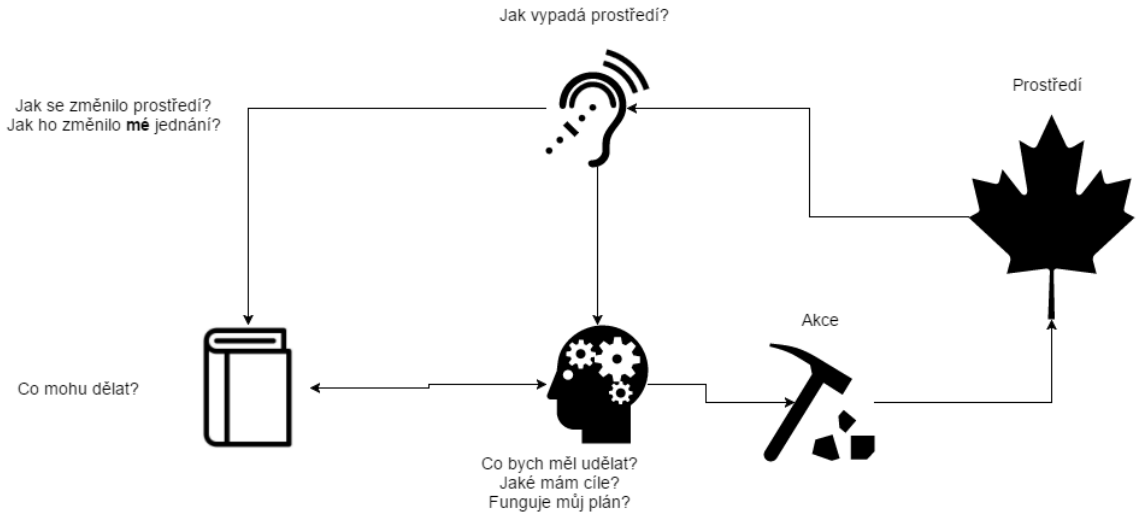
V předešlé části práce je zmiňována reflexivní beruška. To, co berušce chybělo k tomu, abychom o ní mohli říct, že je inteligentní, byla paměť. Zároveň jejímu konání chyběl nějaký vyšší smysl. Beruška jen existovala, ale nic nedělala, pokud tedy nepočítáme to, že se udržela chtě-nechtě na stole.

Po přidání modulu, do kterého může agent ukládat informace, získá agent velmi cenou vlastnost - **paměť.** Nyní si může ukládat „mapu“ svého prostředí, čímž dokáže určit i svou polohu a polohu dalších objektů, se kterými toto prostředí sdílí. To, mít mapu, nebo tvořit si mapu prostředí, přirovnávají Russel & Norvig (15) k taxikáři. Ten by byl bez schopnosti určit svou polohu ve městě naprosto ztracený. Dále si agent dokáže ukládat stav prostředí, ve kterém se nachází. To je počátek plánování a uvažování, jelikož agent bude moci zaznamenávat historii prostředí. Díky této historii bude také zároveň moci předpovídat budoucí stavy prostředí.

Další částí, která reflexní berušce chyběla, byl nějaký vyšší smysl proč tu je. V multi-agentních systémech to označujeme jako mít **cíl**. Agentu je před jeho vložením do prostředí

určen cíl, popřípadě hned několik cílů. U epizodních prostředí je vhodné definovat cíl pro každou epizodu zvlášť.

To, že má agent určitý cíl, samo o sobě nestačí. Agentu je potřeba přidat další modul. Tento modul bude představovat **plánování.** Agent bude předávat informace o prostředí, které získá ze svých senzorů. Dále tomuto modulu předá svůj cíl. Modul by měl vybrat nejvhodnější akci, která povede ke splnění cíle. Místo plánování se může použít **prohledávání**. To, kdy se používá která varianta, je popsáno v dalším kapitole uvažujících agentů.



Obrázek 4 Uvažující agent, vlastní tvorba

1.2.2.1 Inteligence

U berušky bylo řečeno, že její chování bylo inteligentní, i když ona sama inteligentní nebyla. Odpověď na to můžeme najít v těchto definicích:

> *„Intelligent behaviour can be generated without explicit representations of the kind that symbolic AI proposes.“*
>
> *„Intelligent behaviour can be generated without explicit abstract reasoning of the kind that symbolic AI proposes.“*
>
> Wooldridge (16)

Volně můžeme tyto věty přeložit jako: Inteligentní chování může být vytvářeno i bez přímé reprezentace umělé inteligence.

> „*Intelligence is an emergent property of certain complex systems“*
>
> Wooldridge (16)

Což opět volně přeloženo znamená: Inteligence je emergentní vlastnost complexního systému.

Na základě těchto definic, které definují inteligenci jako komplexní vlastnost systému, i když systém nemá žádné inteligentní procesy, bychom tedy mohli původní berušku považovat za inteligentní. V zásadě se dá každý systém, který produkuje inteligentní chování označit za inteligentní.

> „*Situatedness and embodiment. 'Real' intelligence is situated in the world, not in disembodied systems such as theorem provers or expert systems.“*
>
> Wooldridge (16)

Volně přeloženo - skutečná inteligence je ve světě okolo nás a ne v bezduchých systémech, jako jsou důkazoví asistenti a expertní systémy.

> „*Intelligence and emergence. 'Intelligent' behaviour arises as a result of an agent's interaction with its environment. Also, intelligence is 'in the eye of the beholder' - it is not an innate, isolated property.“*
>
> Wooldridge (16)

Tedy - inteligentní chování vzniká jako výsledek agentovy interakce s prostředím. Inteligence záleží na úhlu pohledu a není to izolovatelná vlastnost.

V těchto dodatcích se tedy dočteme, že „skutečnou“ inteligenci není možné vložit, nebo vytvořit v umělých systémech, ale je potřeba jí hledat v reálném světě. S tímto názorem jde souhlasit ve smyslu, že nejsme schopni skutečnou inteligenci vytvořit. Avšak jsme schopni jí zkopírovat. Zatím sice nevíme, co přesně lidi dělá inteligentními, ale víme, že je to proces

v našem mozku. Tudíž, velmi zjednodušeně ho stačí celý zkopírovat tak, jak bylo nádherně předvedeno ve filmu *Transcendence* (3)

To, že nemůžeme vytvořit skutečně umělou inteligenci, v podstatě nepředstavuje překážku. Plnou inteligenci už máme sami. Tudíž nám už jen stačí vytvořit systém, který bude z pohledu problematiky inteligentní a tuto problematiku zvládne řešit.

1.2.2.2 Prohledávání

V části o uvažujících agentech je zmíněno, že se agenty snaží dosáhnout svého cíle. To jestli agent používá k dosažení cíle **plánování,** nebo **prohledávání,** záleží v zásadě na jednom faktoru. Tímto faktorem je prostředí. Pokud se agent nachází v prostředí, které je z jeho pohledu statické, tak k dosažení cíle používá prohledávání.

Proces prohledávání se skládá z určitých částí. Nejdříve si agent ze svého seznamu cílů některý vybere. Pokud jsou na sobě závislé, tak si musí samozřejmě nejprve vybrat v pořadí první cíl. Agent začne s nejnáročnější činností - tím je prohledávání prostoru prostředí. Dále započne s aplikováním akcí, kterých je schopen a jsou pro splnění daného cíle vyžadovány. Takto postupuje, dokud nedosáhne svého cíle(15).

Toto je možné přirovnat k bloudění v bludišti. Agent prochází (bloudí) jednotlivými uličkami. Pokud je potřeba zatáhnout za páku, tak zatáhne. Tuto činnost dělá tak dlouho, dokud nenajde východ. Toto je zdlouhavý proces a při nesprávné strategii může být i nemožné ho splnit. I když agent nalezne cestu z bludiště, tak při pozorném prozkoumání můžeme zjistit, že agent prošel jednou uličkou více krát. Proto se často agentům přidává požadavek na nalezení nejlepšího řešení. Tento požadavek však zvyšuje náročnost na agenta. Ten by však musel prohledat celý svůj prostor, aby s jistotou mohl říct, že nejlepší řešení je toto. Většinou nám však stačí, dostat se pod určitou mez. V případě bludiště by cíl zněl: „Nalezni cestu z bludiště, která tě dostane ven za méně než dvě hodiny“. Takových cest sice může být milion, ale agentu bude stačit najít jednu jedinou. Výstupem prohledávání bývá v celku „jasný“ výsledek - nějaká cesta, číslo, nebo odpověď: Ano, je to možné / Ne, není to možné.

O nalezení nejlepšího, nebo aspoň dostačujícího výsledku nám často jde nejvíce. Dnes se k tomuto využívají metody typu "učenlivých agentů". Tyto metody využívají poznatků z genetických algoritmů.

1.2.2.3 Plánování

Předcházející strategii "prohledávání" agent používal, když se nacházel v prostředí, které pro něj bylo statické. V případě, že je prostředí dynamické, agentům prohledávání přestává stačit. Jako důkaz můžeme uvést znovu labyrint. Víme, že nejkratší cesta z labyrintu trvá hodinu a pět minut. Avšak labyrint se mění každou hodinu. Pokud by agent pořád jen prohledával, tak i při nalezení východu nebude mít požadovaný výsledek, jelikož cesta, kterou předtím šel, už neexistuje.

Při prohledávání je učení nástroj, pro nalezení lepších výsledků. U plánování je učení a jemu podobné postupy, takřka nezbytností. Agentu v dynamickém prostředí nestačí zvolit si cíl a strategii, jak k tomuto cíli dojít a podle této strategie se řídit. Prostředí se mění a tím se může stát, že jeho strategie přestává fungovat. Agent se v této situaci musí adaptovat a svou strategii (plán) měnit(15). Výstupy plánování jsou obecné algoritmy a postupy k dosažení cíle.

1.2.2.4 BDI

Teorie BDI (Belive, Desire, Interest) byla založena M. Bratmanem v roce 1987. Jedná se o koncept architektury uvažujících agentů. Tento koncept počítá s tím, že každý agent systému bude mít své **B,D,I**.

Belive / představy: agent na základě informací získaných se senzorů vytvoří své představy o tom, jak vypadá prostředí, ve kterém se nachází. Je možné, že se představy budou lišit agent od agenta, jelikož každý bude prostředí vnímat jinak (9).

Desire / tužby: agenty mají stejně jako většina lidí má své tužby. Jedná se o seznam cílů, kterých chce agent dosáhnout (9)

Interest / zájmy: jsou tužby, na které se agent bude soustředit. Bude chtít přetvořit prostředí, aby odpovídalo jeho dané tužbě (9)

1.2.3 Sociální agent

V této části se dostaneme k jádru toho, co definuje multi-agentový systém. Do této chvíle jsme hovořili o samostatných agentech. Agenty s uvažující architekturou už jsou sice schopny sami zvládnout teoreticky cokoli, co jim zadáme, avšak jsou na to pořád sami. A podle rčení „ve dvou se to lépe táhne“ je lepší, když jim přidáme spolupracovníka. Je to vlastně nutné, pokud chceme simulovat reálný systém.

Agenty ve virtuálním prostředí spolu nejčastěji komunikují pomocí zpráv. To, jak bude zpráva fyzicky vypadat, se liší systém od systému. Avšak existují jisté doporučené postupy. Jedním z takových je standart ACL[6], který vyvíjí společnost FIPA[7]. Tento standard zahrnuje více než jen komunikaci, ale i to, jak by měla architektura agenta vypadat a jak by měl probíhat agentův životní cyklus. Dále ukazuje architektonické standardy celého multi-agentového systému.

To, že agenty mohou komunikovat jeden s druhým, vytváří možnost využívat schopností jiného agenta a jeho informací. Například agent **A** hledá, kde se v prostředí nachází nějaký předmět. Buď prohledá celé prostředí, což mu zabere dost dlouho (záleží na velikosti prostředí a prostředí pro agenta musí být nepřístupné), nebo se agent **A** může zeptat ostatních agentů, jestli neznají polohu tohoto předmětu.

Agenty mohou nejen využívat něčí služby, ale zároveň mohou uzavírat „spojenectví“. Takovéto skupiny agentů pak pracují na společném cíli, který stihnou uskutečnit mnohem rychleji. Spolupracující agenty mezi sebou mají určitý řád. Stejně jako spolupracují lidé ve firmě. Existuje několik architektur, které popisují průběh, jakým takové společenství funguje. Typy těchto společenství jsou rozebrány v následujících kapitolách.

[6] Agent comunication language
[7] Foundations for inteligent Physical Agents

1.2.3.1 FIPA

Společnost FIPA vznikla v roce 1996. Jedná se o podobně jako konsorcium W3C[8] o sjednocení firem, které se zabývají podobnou problematikou. Dnes do tohoto konsorcia patří mnoho firem a vědeckých institucí (12).

1.2.4 Způsoby spolupráce v multi-agentních systémech

Mít schopnost předávat si mezi sebou informace, agentům sama o sobě nestačí. Pokud chceme, aby agenty skutečně něčeho dosáhly a přitom efektivně spolupracovaly, je potřeba jim do této společné snahy přidat i řád, nebo spíše další architekturu.

Proto, aby agenty věděly, jak mezi sebou mají komunikovat, jsou vytvořeny protokoly a architektury. ACL od FIPA, které bylo zmíněno, představuje právě tuto část, bez které by následující část Koordinace nebyla možná.

Umělá inteligence, do které agenty patří, se dá rozlišit na dva podtypy.

Distribuovaná: Tento typ umělé inteligence se používá, když je potřeba řešit jeden rozsáhlý problém. Pak se části této úlohy rozdělí do několika výpočetních procesů. Procesy často bývají specificky zaměřeny na určitý úkol. Můžeme to přirovnat například k sekcím ve firmě. Je tu vedení, které rozděluje úlohy, IT, pracovníci atd. Úlohám spadajícím do této kategorie se říká distribuované řešení problému[9]. Při využití distribuovaného řešení, je potřeba, aby agenty byly vždy ochotní spolupracovat - předpokládá se benevolence[10]. Agenty musí chtít dosáhnout onoho společného cíle, vyřešit problém a neměly by klást vlastní zájmy nad tento cíl (9).

[8] WordlWide Web Consortium

[9] Distributed problom solving

[10] Benevolence assumption

Decentralizovaná: Název napovídá, že u decentralizovaného přístupu agenty nemají jednotu distribuovaného systému. Agenty této architektury mají své vlastní individuální cíle. To jim však nijak nebrání spolupracovat s ostatními. Tato spolupráce je často velmi krátkodobá, po splnění daného cíle se agenty přesunou k dalšímu a k tomu už se daná skupina nemusí hodit. U distribuovaného typu byl u agentů předpoklad benevolence. V decentralizovaném tento předpoklad chybí. Agenty mohou odmítnout spolupráci. Popřípadě se můžou vyskytnout i agenty protichůdných cílů, kteří si navzájem škodí (9).

Umělý život: je disciplína zabývající „Zkoumáním života, jaký je a také jaký by mohl být". Často se v této souvislosti zkoumají složité ekosystémy. Ať už živé, třeba mraveniště, nebo umělé, jako je například ekonomika (9).

Simulace ASE spadá právě do této kategorie zkoumání umělého života.

1.2.5 Koordinace

Koordinace mezi agenty může vzniknout za určitých předpokladů:

- Počet agentů je roven, nebo je větší dvěma. Je nemožné, aby agent komunikoval sám se sebou (pokud tedy nemodelujeme schizofrenního agenta).

- Agenty musejí být schopny komunikovat.

H. Mintzberg (4) ve své knize popisuje několik typů koordinace, které mohou nastat v závislosti na typech agenta a dále na vztazích, které agenty mezi sebou mají (4).

Nejvolnějším vztahem je **vzájemná dohoda**[11]**.** Agenty jsou v tomto vztahu na stejné úrovni, není mezi nimi nikdo nadřazený, ani podřízený. Agenty si sami mezi sebou, bez třetí strany, vykomunikují spolupráci. Agenty mezi sebou sdílí zdroje a vědomosti potřebné k dosažení

[11]Mutual adjustment

cíle. H. Mintzberg(4) tuto koordinaci přirovnává k dvěma lidem v kánoi, kteří spolu pádlují po řece (4).

Dalším typem koordinace popsaným H. Mintzbergem (4) je **přímý dozor[12].** V tomto přístupu už mají různá postavení. Jeden agent je vybrán, aby korigoval celý tým - „One brain coordinates several hands“ (4). Agenty se lehce vzdají své autonomnosti, aby splnily daný cíl. Tento systém je přirovnáván k sportovním týmům, kdy každý hráč má svou roli, ale jeden řídí celý tým a říká ostatním, kam mají jít a co po nich chce. Avšak realizace toho, jak to udělají, zůstává na agentech.

Posledním, i když rozvětvenějším typem je **standardizace.** V přechozích dvou případech mezi agenty docházelo neustále ke komunikaci. Ať už agenty komunikovaly mezi sebou, nebo s nějakou nadřazenou autoritou. Jakmile se však skupina rozroste, tak je komunikace neúčinná - není na ní čas. H.Mintzberg (4) uvádí dobrý příklad operačního sálu. Kdyby se doktoři řídili vzájemnou dohodou, tak by spolu museli pořád mluvit a domlouvat se na postupu, v krizové situaci by musel chirurg čekat na vyjádření ostatních, než by mohl zašít prasklou tepnu. Při přímém dozoru by vůdce sice mohl stihnout dávat všem příkazy, avšak jen do jisté míry. Jakmile by bylo více operací najednou, byl by přehlcen a nestíhal by ani on. Tomuto stavu se říká přetížení centrálního prvku. V takových situacích je potřeba, aby každý věděl, co má dělat a jakými pravidly se má řídit. Všichni mají právě své standardní postupy. Tyto postupy jsou však obecné. Neříkají, jak má chirurg operovat. Spíš říkají: "Nenech pacienta umřít".

K. Aleš (9) ve své knize toto přirovnává naopak k pravidlům silničního provozu. Na křižovatce potkáte auto a vy i druhý řidič víte, kdo má přednost a jak se chovat. Avšak pokud agent, z nějakého důvodu, chce třeba dosáhnout jiného cíle, nebo urychlit svou práci, nerespektuje daná pravidla. Pak na to může doplatit celý systém. Pacient umře - stane se dopravní nehoda(9).

[12] Direct supervision

2 ASE

V úvodu je zmíněno, že v dnešní době se multi-agentní systémy využívají pro řešení složitých problémů. Část hovořící o způsobech spolupráce v multi-agentních systémech zmiňuje **umělý život**. A přesně o takovém umělém životu projekt ASE je. ASE simuluje fungování, životní proces a ekonomiku. Přesněji její makro-ekonomické části, které se skládají ze subjektů. Tyto subjekty nejsou menší než firma. Tento projekt má na katedře informačních technologií již svou tradici. Tato práce navazuje na práci předchozích studentů Ing. Radima Lulka(8), Ing. Jana Vlacha(5) a Tomáše Kolingra. Z odborných prací těchto studentů UHK je v některých pasážích této práce čerpáno.

Framework pro ASE se podařilo vytvořit. Později do něj byli zakomponováni sociální agenty, kteří spolu obchodují a tím provádí onu simulaci. Pro odlehčení zátěže, která je v této simulaci vyvíjena na práci s daty, byla implementována No-SQL databáze, v podobě grafové databáze. Na základě dat získaných z grafové databáze, je možné provádět další výzkum zaměřený na sociální vazby mezi danými firmami.

V následující kapitole se budeme věnovat právě simulaci ASE. Tomu, jaký typ architektury je použit u agentů. Komunikační záležitosti při obchodu a podmínky jeho uskutečnění, zde budou také uvedeny.

2.1 Agenty v ASE

To, že ASE je multi-agentní simulace, bylo již řečeno. V této části práce bude přiblíženo, o jaké agenty se vlastně jedná, co představují a jaké mají vlastnosti.

Jak je napsáno v úvodu, základním agentem ASE je nějaká obecná firma. Modely, které jsou v práci uvedeny podléhají jisté abstrakci. Myšleno, že v implementaci se rozrostly o jisté

podpůrné privátní[13] atributy. Každá firma má atributy, kterými je definována. Stejně tak je mají i agenty v simulaci ASE.

Ekonomika je postavena na třech pilířích. Nabídka, poptávka a peníze. Bylo potřeba zařídit, aby agenty dokázaly pracovat v tomto obecném pravidlu.

Poptávka: Poptávka představuje, co agent potřebuje. Každý agent spotřebovává své zdroje k tomu, aby mohl „žít“. Tyto zdroje si musí někde opatřovat. Nezáleží, jestli si je vytvoří sám, nebo je získá od ostatních. Avšak žádný agent neumí vytvořit vše, co potřebuje. Tím jsou agenty donuceni spolupracovat s prostředím, respektive s ostatními agenty.

Nabídka: Nabídka představuje, co agenty produkují. A následně, co nabízejí k odkupu ostatním, když mají přebytek. Díky nabízení svých produktů ostatním agentům, získají agenty „peníze“, za které mohou odkoupit zdroje, které sami nemají a neumí je vytvořit.

Peníze: V simulaci ASE jsou peníze zastoupeny virtuální měnou, která nemá v reálném svůj vzor. Avšak agenty ji využívají k obchodům a plnění svých cílů. Tato měna je uložena v atributu finance a pracuje se s ní pouze na oboru celých čísel.

2.2 Atributy agentů

ID: Představuje agentovo jedinečné označení v simulaci.

Platform: Je podobného významu jako ID, až na to, že se jedná o jedinečné označení platformy, na které agent žije.

Inventory: Agenty mají svůj inventář, do kterého si ukládají své zdroje, „produkty“.

Partners: Představuje seznam agentů, které agent zná. Agent nezná všechny agenty v simulaci, ale pouze určitou část. Tento seznam však není neměnný. Pokud agent usoudí, že je někdo v seznamu zbytečný, vyřadí ho. Následně se pomocí zbylých kontaktů pokusí najít vhodnou náhradu.

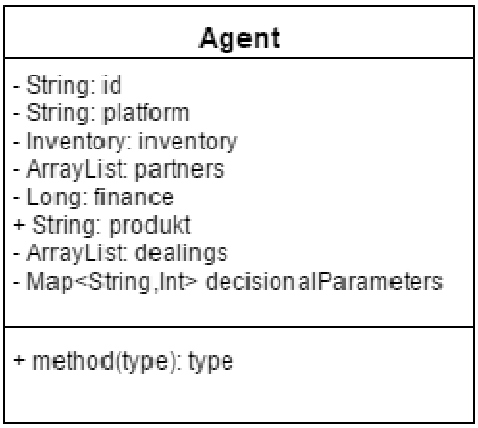
[13] Na venek neviditelné a pro systém a zkoumání dané problematiky nepotřebné.

Finance: Tento atribut představuje množství „peněz“, které agent vlastní a může s nimi nakládat.

Produkt: Je spíše atribut pro ostatní agenty, který udává, co je agent schopný vytvářet. Atribut poptávka neexistuje, jelikož agent poptává všechny zdroje v simulaci.

Dealings: Je seznam obchodů, které se agenta aktuálně týkají. Jsou to obchodní nabídky ostatním agentům, u kterých se čeká na odpověď.

DecisionParameters: Agenty mají svůj osobní, pro každého agenta individuální, přístup k tomu, co je dobrý obchod. Například agent "A" považuje dobrý obchod, když prodá dané zboží za 100. Agent "B" však chce za to samé už 150. V této mapě[14] se dané preference agenta ukládají.



Obrázek 5 Zápis agenta v UML, vlastní tvorba

2.2.1 Metody agentů

Jedná se o aktivity agentů - o to, co agenty mohou dělat. Jelikož jsou agenty virtuální, tak do této kapitoly spadají i části agenta, jako jsou senzory a efektory.

Ze zástupců **senzorů** to jsou metody recive(), findAgent()

[14] Datový typ

Recive(): Jedná se o metodu, jejímž výstupem je zpráva pro daného agenta. Agenty nemají své zprávy přímo u sebe, ale jsou uloženy ve frontách na platformě. Agent požádá o zprávu a vždy jí dostane. Pokud platforma nemá další zprávu pro agenta, pošle mu null[15].

FindAgent(): Této metodě na vstupu agent předá, co hledá. Například, že hledá agenta, který produkuje (prodává) jídlo. Prohledá se seznam agentů, které agent, který zavolal metodu, zná. Tudíž jsou v jeho seznamu Partners.

Pokud agent ve svém seznamu Partners žádného takového agenta nenajde, tak provede širší hledání, ve kterém se prohledají seznamy přilehlých agentů. Agent **A** prohledá svůj seznam a pak prohledá seznam agenta **A**, ve kterém se pokusí najít agenta **C.**

To by byly senzory, kterými agenty získávají informace z prostředí. Dále následují efektory. Opět se bavíme o virtuálních efektorech, tudíž o metodách, které agent provádí.

Spotřeba: Agent na počátku každého ticku[16] spotřebuje své zdroje, což ho nutí plánovat. Agent si musí spočítat, jestli má zdroje na příští ticky a na kolik. Jakmile mu začnou ubývat začne zdroje opět shánět.

Work(): Jak bylo napsáno, agenty spotřebovávají zdroje, nebo je také prodávají někomu jinému. Naštěstí si některé zdroje umí vytvořit sami. Touto metodou tyto zdroje tvoří. To, co tvoří, je obsaženo v atributu produkce. Agent ve finále každý tick provádí tuto metodu, jelikož se stává jen málokdy, že by stav tohoto zdroje byl plný. Už kvůli tomu, že je spotřebováván jím samým a také tím, že to je jeho hlavní obchodní artikl.

Obchod: Toto už sice není metoda, ale spíše činnost více agentů. Avšak jejím výsledkem je změna prostředí.

Obchod v simulaci ASE probíhá mezi dvěma různými agenty, bez zásahu třetí strany. Jedná se zpravidla o vytvoření nabídky agentem **A**, kterou pošle agentu **B**. Ten jí zhodnotí a na základě svých preferencí odpoví. Může nabídku přijmout, odmítnout, nebo jí pozměnit a poslat na zpět jako novou nabídku.

[15] Prázdnou zprávu

[16] Časová jednotka

2.2.2 Čas v ASE

V simulaci ASE je čas reprezentován vymezeným úsekem, který je nazýván tick. Problematika nemožnosti vytvoření souvisle probíhajícího času, je řešena v diplomové práci J. Vlacha (5), kde je uvedeno úskalí ASE v počtu agentů. Se stávajícími technologiemi není možné vytvořit simulaci o dosti velkém počtu agentů, kteří existují v souvislém čase a všichni jsou spuštěni najednou.

Řešení ASE spočívá ve spuštění vždy určitého počtu agentů na paralelních vláknech. Agenty dostanou přidělené časové body. Každá jejich akce je stojí určité množství časových bodů. Jakmile agent vypotřebuje své body, ukončí svou činnost a předá pracovní vlákno dalšímu agentu. Tento postup připomíná herní kola z tahových strategií, kdy se hráči střídají v tazích.

2.2.3 PEAS

Úkolové prostředí se označuje zkratkou PEAS[17]. Každému typu agenta v simulaci se specifikuje jeho PEAS. Definování PEAS pomáhá určení účelu (role) agenta v simulaci.

V simulaci ASE se vyskytuje zatím jeden druh agenta.

Performenc / výkon: Výkon agenta v simulaci ASE se určuje podobně jako v reálném světě výkon firmy - v penězích. Agent, který má nejvyšší finance, je nejvýkonnější. Později by bylo možné přidat přepočet nehmotného majetku na finance. To však v době psaní této práce zakomponované v ASE není.

Enviroment / prostředí**:** Prostředí pro daného agenta je tvořeno ostatními agenty. Pro agenta **A** jsou agenty **B,C,D** prostředím, se kterým komunikuje a získává z něj zdroje.

Prostředí je pro agenta **nepřístupné,** jelikož nezná všechny agenty v simulaci, jen několik svých známých.

[17]Performenc, Envirometn, Actions, Sensors

Dále je prostředí pro agenta **stochastické**, což znamená, že agent není schopný určit, jak se prostředí vyvine, jelikož nezná strategii ostatních agentů.

Agent se pohybuje v prostředí, které je pro něj **průběžné**. Přítomnost časového ticku sice poukazuje na epizodní prostředí, avšak agenty vycházejí ze svých předešlých zkušeností.

Prostředí se mění bez toho, aniž by do něj agent zasahoval. Zasahují totiž ostatní agenty. Tudíž se jedná o prostředí **dynamické**.

Agent se pohybuje ve **spojitém** prostředí, jelikož není nijak omezen počtem stavů, v jakých se může nacházet.

Actions / akce: Akce agenta, které ovlivňují prostředí, byly podrobněji popsány výše. Jedná se o **Work(), Spotřeba, Obchod.**

Sensors/senzory*:* Senzory na vnímání prostředí byly stejně jako akce, podrobně popsané v předchozích kapitolách. Jedná se o **FindAgent(), Recive().**

3 Grafy

V minulých kapitolách byly představeny multi-agentní technologie a projekt ASE. V této kapitole bude tématem teorie grafů, ze které vycházejí grafové databáze.

Grafy, jsou nedílnou součástí grafových databází. Je tedy velmi podstatné pochopit strukturu a fungování samotných grafů a až poté přejít ke grafovým databázím. Následující kapitola pojednává o grafech samotných a o vlastnostech, které jsou u nich pozorovány.

3.1 Úvod do grafů

Grafy jsou tvořeny dvěma částmi - uzly a hrany. Každý uzel může mít nula hran, v tom případě se jedná o primitivní graf. Klasický graf však může mít až nekonečno hran. Hrana naproti tomu musí vycházet a končit v uzlu, i kdyby začátkem a koncem měl být tentýž uzel(14).

Uzly a hrany mohou mít své vlastnosti. Uzly představují objekty z reálného světa, většinou jsou to námi zkoumané objekty. Každý objekt má určité vlastnosti, kterými ho popisujeme. Barva, velikost, finance. Objekty mají též mezi sebou určité vazby. Lidé mezi sebou mají například rodinné vazby, např. rodič a dítě. Tyto vazby jsou atributy vazeb. Vazby jsou nějakým směrem orientované. Například Jarda je kamarád Franty a naopak. Ale Jarda může být otcem Franty, ale naopak už ne (11).

Ve výsledné grafové databázi budou grafy znázorňovat mapu virtuálního prostředí, ve kterém se simulace odehrává. Uzly budou agenty, kteří představují firmy obchodující a žijící v simulaci. Hrany budou cesty mezi nimi. Zároveň hrany budou reprezentovat i vztahy mezi agenty. Tím, že hranám můžeme dát jejich vlastní atributy, budeme schopni rozlišit druhy vztahů.

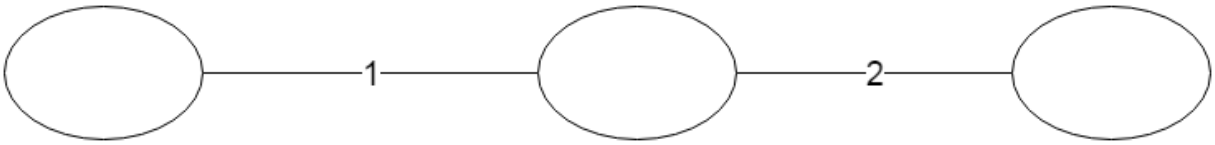
3.2 Vlastnosti grafů

To, jak grafy vypadají a z čeho se skládají, bylo popsáno v předchozí části. Tato se bude zaměřovat na vlastnosti, které grafy mají. Tyto vlastnosti nám pomohou v práci s grafy.

3.2.1 Váženost grafu

Neohodnocený graf je takový, ve kterém nemá žádná hrana svou hodnotu. Všechny hrany jsou si rovny. V takovém grafu je cena cesty[18] určena podle počtu projitých hran, jako by každá hrana měla hodnotu rovna jedné. Takové grafy ve zkoumané simulaci nebudou, jelikož právě rozdíly v ceně cest budou klíčové.

Ohodnocený graf má v sobě zakomponované hodnoty pro každou hranu. Díky tomu se určuje, jaká cesta z uzlu A do B je nejvhodnější. Je to právě podle porovnávání ceny cest (14).



Obrázek 6 Zápis váženosti grafu, vlastní tvorba

3.2.2 Souvislost grafu

Pokud z jakéhokoli uzlu v grafu můžeme vést cestu do jakéhokoli uzlu, tak můžeme říct, že graf je souvislý (14).

Tato podmínka v simulaci ASE musí být vždy splněna. Důsledkem této podmínky je, že žádný uzel, nebo skupina uzlů, nebude separovaná od ostatních. Kdybychom tento jev nehlídali, nejspíše by nám takový izolovaný uzel brzy umřel. Popřípadě bychom vytvořili ve skupině prostor pro monopolní firmu.

[18] Cena cesty - součet hodnot všech projitých hran

Obrázek 7 Ukázka rozdílu souvislosti a nesouvislostí grafu, vlastní tvorba

3.2.3 Vyšší stupně souvislosti

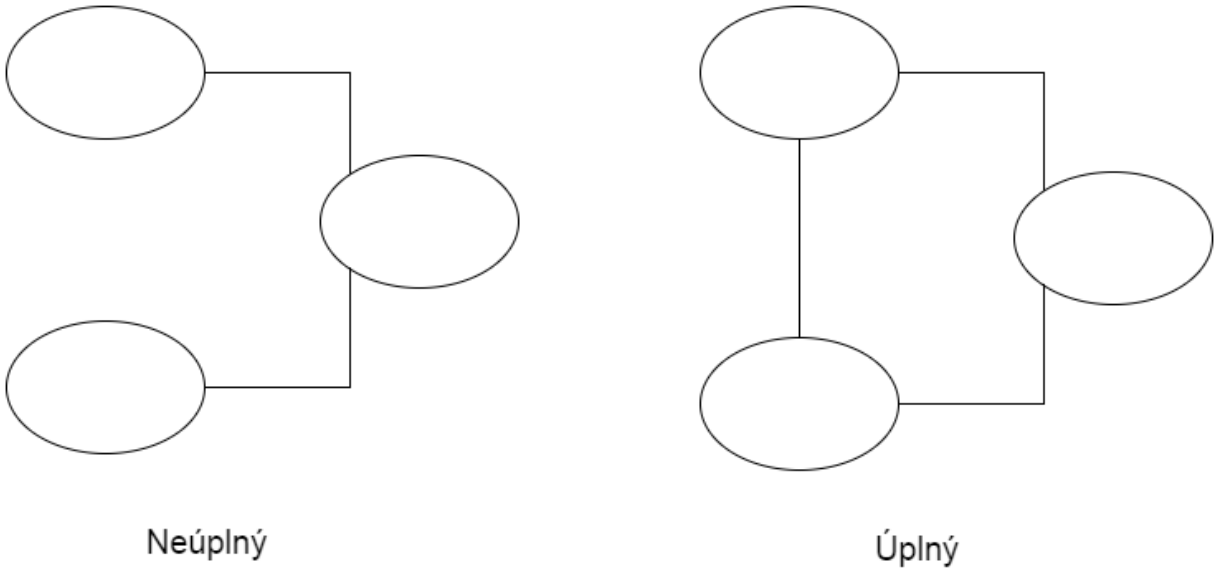
Stupeň souvislosti se určuje jako počet cest mezi dvěma libovolnými vrcholy (14).

Tato vlastnost zajišťuje, že i po zrušení jedné cesty, se nám graf nerozpadne na dva oddělené grafy. V ASE se totiž bude velmi často - vlastně pořád stávat, že některé firmy zaniknou a jiné budou stvořeny. Zánikem firmy nám z grafu zmizí uzel a jeho hrany. Pokud budeme hlídat stupeň souvislosti, aby neklesl pod určité minimum, nikdy se nám graf nestane nesouvislým. Například pokud budeme udržovat stupeň souvislosti $k \geq 2$, máme jistotu, že se vyhneme rozdělení grafu.

3.2.4 Úplnost grafu

O grafu se dá říct, že je úplný tehdy, když z jakéhokoli uzlu vede přímá cesta do jiného uzlu(14).

Tento typ grafů v simulaci ASE použit nebude, jelikož jeho komplexnost neodpovídá realitě. Z každého místa na světě nevede specifická cesta do jiného.



Obrázek 8 Ukázka rozdílu úplného a neúplného grafu, vlastní tvorba

3.2.5 Orientovanost grafu

Grafy jsou neorientované a orientované. V neorientovaných grafech nemají hrany směr, dalo by se říct, že jsou obousměrné. V orientovaných grafech už tyto směry jsou(14).

Grafy simulace ASE jsou orientované, tudíž může nastat stav: A zná B, ale B nezná A.

Obrázek 9 Ukázka rozdílu orientovaného a neorientovaného grafu, vlastní tvorba

4 Grafové databáze

Grafové databáze spadají do skupiny s názvem NoSQL[19] databází. Tyto databáze se začaly rozšiřovat v devadesátých letech dvacátého století, kdy přestávaly tradiční metody v podobě SQL databází stačit. Tím není myšleno, že by SQL nemohly tuto činnost zvládnout také, ale čas a prostředky se nepřijatelně zvýší. Grafové databáze jsou tedy odvětví NoSQL, které se zaměřuje na problematiku udržování vazeb mezi daty. Tím, že jsou data zpracovávána v podobě grafů, odpadá nutnost udržování si referencí „Vlastní klíč“ a „Cizí klíč“[20] na zbytek databáze. Jelikož samotná struktura tyto vazby nahrazuje (11).

Jako grafová databáze byla vybrána Neo4j, kterou vyvinula společnost Neo Software v roce 2007. Neo4j je vytvářeno v programovacím jazyce Java. Díky faktu, že v Neo Sofware pro vývoj své databáze použili programovací jazyk Java, je zajištěna multi-platformnost celé databáze(10).

Dalším důvodem pro výběr Neo4j jako grafové databáze pro projekt ASE, bylo množství studijních materiálů v podobě online tutoriálů (1), dále knihy pojednávající o grafech a Neo4j, jako například *Graph Databases* od autorů Robinsona, Webbera a Eifrema(11) a v neposlední řadě i široké komunitě lidí, kteří poskytují na internetu dostatek vlastních řešení v této oblasti.

4.1 Neo4j

Níže budou popsány teoretické principy databáze Neo4j. Dále v kapitole Práce s Neo4j bude jejich praktické využití i bližší specifikací.

[19]NoSQL – Not only SQL

[20]Vlastní klíč a cizí klíč jsou pojmy z SQL databází, které zajištují odkazování se na jiné tabulky.

4.1.1 Node

Jak bylo napsáno v kapitole **Grafy**, samotné grafy se skládají ze dvou klíčových komponent. Tím jsou vrcholy a hrany. Vrcholy jsou v Neo4j zapsány v podobě **nodů**. Jsou to nejčastěji objekty z reálného světa, nebo z domény, která je zpracovávaná. V simulaci ASE to jsou tedy agenty reprezentující firmy. Tudíž každý agent v simulaci bude zaznamenán jako jeden vrchol v databázi.

4.1.2 Relationship

Vztahy mezi danými nody jsou druhou klíčovou částí grafových databází. V grafových databázích jsou reprezentovány hranami mezi danými vrcholy. Hrany vyjadřují vztah mezi nody. V simulaci ASE to je právě vztah, že node **A** má ve svém seznamu partnerů daný node **B**. Tím je vyjádřeno, že **A** spolupracuje a chce nodu **B** často posílat nabídky k obchodu. Vazbám jde stejně jako nodům přiřazovat určité vlastnosti, které danou vazbu více upřesní. V simulaci ASE by se jako vlastnost vztahu mohla uchovávat informace o tom, kolik obchodů **A** navrhl **B** a kolik jich bylo úspěšných. Tato informace zatím však není uchovávána.

4.1.3 Property

Vlastnosti jsou důležitou součástí databázové struktury a udržují většinu důležitých informací, které je možné z databáze získat. Vlastnosti jsou uchovávány v podobě mapové struktury, vyjádřené klíčem a hodnotou ke klíči odpovídající.

Nodům je možné přiřadit vlastnosti podle potřeb uživatele. Avšak atribut ID si Neo4j vytváří nezávisle samo. Tudíž se v databázi vyskytuje jak atribut ID, tak atribut IdAgenta. Toto by mohlo být pro někoho zmatečné, ve smyslu jaký identifikátor slouží k čemu. IdAgenta je unikátní identifikátor agenta v simulaci ASE. ID je tudíž unikátní identifikátor v databázi Neo4j.

4.1.4 Label

Label pomáhá k orientaci v grafové struktuře. Označuje buďto typ nodu, nebo typ vazby mezi nody.

V době psaní této práce, se v ASE používal label typu:

- FIRM

- WorkWith

Pro lepší vysvětlení však bude uveden jiný přiklad domény. Kdybychom měli databázi vozidel, tak label bude označení typu, tedy - jedná se o auto, motorku, nebo přívěs ?

V případě vazeb by label znamenal typ spojení obou objektů, tedy například, že někdo vozidlo ***řídí***, nebo někdo vozidlo ***opravuje***. Díky tomu se pak dá vyhledávání omezit jen na určité nody, kteří mají určitý label. Node a vazba může mít pouze jeden label.

4.1.5 Cypher

Jelikož je grafová struktura dosti odlišná od ostatních databázových struktur, bylo potřeba vytvořit vlastní jazyk na práci s touto strukturou. Tímto jazykem se stal **Cypher**. Cypher se velmi inspiroval u jazyku SQL a byl přizpůsoben tak, aby práce s ním velmi připomínala práci s tradičním SQL (1).

Nejjednodušší cesta pro získání dat z databáze, je přes vytvoření dotazu v jazyce Cypher. Příkladem takového dotazu je následujíc kód:

MATCH (n:FIRM)	Slovem MATCH určujeme, co má být hledáno. To něco je určeno v závorkách. V této části se určuje struktura toho, co je hledáno. V další části, kde budou použity vazby, je tato část zřejmější. V tomto případě je to **n**, které má label FIRM.
WHERE (n.Finance< 30)	WHERE je upřesňující část dotazu. Určují se zde určitá kritéria, která mají být také splněna.
RETURN n LIMIT 5	Za částí RETURN se udává, co nám má databáze ve skutečnosti vrátit. Je sice potřeba najít node, který splňuje několik vlastností a zároveň jeho okolí splňuje další požadavky, ale na konci je potřeba vrátit jen určitá vlastnost takového nodu. Toto je řečeno

právě v RETURN. V databázi se ale může vyskytovat několik nodů (vazeb), které splňují takové požadavky. Pro případ, aby dotazovaný nebyl výsledky přehlcen, je možné určit omezení výsledků. Toto omezení se určuje číslem kladným a nenulovým za slovem LIMIT.

Pokud je potřeba pracovat s vazbami mezi nody, tak k tomu je v jazyku Cypher vytvořena speciální konvence.

MATCH (n)-[v:WorkWith*y..z]->(B)

V kulatých závorkách jsou zapsány nody. V hranatých závorkách jsou zapsány vazby mezi těmito nody. Směr vazby je naznačen šipkou (->). Standardně Neo4j vyhledá agenta **B,** který je přímo spojený s **n** vazbou WorkWith. Jde však zařídit, aby se hledal i vzdálenější agent, nebo naopak, aby se první agent přeskočil. Toto se umožní přidáním symbolu ***** za název vazby. Za * následuje **y**, které určuje, od které úrovně se má vyhledávat. **Z** určuje, do které úrovně se má vyhledávat.

WHERE (v.Value = 5) Princip upřesnění pomocí WHERE, je i u vlastností vazeb totožný s vlastnostmi nodu.

RETURN v V případě, že se dotazujeme na vrácení nodu, je nám vrácen jen daný node. Avšak pokud se dotazujeme na vazbu, je vrácena samotná vazba, ale s ní i oba nody, které spojuje.

5 Práce s Neo4j

Pro práci s Neo4j v Java, je společností Neo Software, vytvořena knihovna. Tato knihovna bude dále v práci popsána. Pro práci s Javou byly metody přímo napsány autorem.

5.1 Knihovna Neo4J pro práci s Javou

V práci se bude používat standardní knihovna od Neo Software, která obsahuje předem připravené metody, poskytující ulehčení práce s grafovou databází.

5.1.1 Vytvoření databáze

Prvním krokem v práci s daty, je tato data nějak získat, nebo vytvořit. Následující příkaz je právě tímto prvním krokem.

```
GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase(storeDir);
```

Tento příkaz vytvoří novou databázi na místě určeném **storeDir**. Pomocí instance **db** se pak pracuje s danou databází. Je možné pracovat s více databázemi na jednou, tím že bude vytvořeno více instancí typu GraphDatabaseService, avšak každá musí odkazovat na jinou databázi - parametr **storeDir** musí být různý. Při pokusu o připojení k databázi, kterou obsluhuje jiný program, třeba grafický editor, nám tak překladač vrátí výjimku.

Pokaždé, když se v kódech pracuje s instancí **db,** je nutné daný kód „obalit“ Try/cache .

```
try (Transactiontx = db.beginTx()){
     db.someThing();
     tx.success();
                              }
```

Toto zaobalí daný požadavek na databázi jako transakci. Díky tomu bude muset dotaz splnit pravidla ACID, aby byl vykonán. Pokud je nesplní, tak se databáze vrátí do stavu před ním. S tímto zabezpečením se již není třeba obávat, že by se databáze poškodila.

5.1.2 Tvorba Nodů a jeho vlastností

Do databáze je potřeba vložit agenty ze simulace jako nody. K tomu má knihovna Neo4j vlastní metody, které jsou níže.

```
(1) Node newNode = db.createNode(labelName);
(2) newNode.setProperty(propertyName,propertyValue);
```

Vytvoření nového nodu se stává z řádku (1). Při vytvoření se zadává i **Label**, do kterého daný Node patří. Řádek (2) ukazuje vytvoření vlastnosti pro Node, vytvořený o řádek výše. V příkazu *setProperty* se na místo prvního argumentu vloží název dané vlastnosti. *propertyValue* je hodnota k dané vlastnosti.

propertyName vždy přijímá pouze argument typu String[21].

propertyVaule má však rozsah o něco větší:

```
boolean/boolean[]
byte/byte[]
short/short[]
int/int[]
long/long[]
float/float[]
double/double[]
char/char[]
String/String[]
```

Jedná se tedy o všechny primitivní[22] typy a pole s elementy těchto hodnot.

Vlastnosti jsou však ve skutečnosti později přetypovány na kořenový typ **Object**[23]. V typu **Object** databáze vlastnosti později i vrací. Neo4j si takto tvoří pro každý Node jeho **mapu**[24] vlastností.

[21] Datový typ String.

[22] Základní datové struktury jsou označeny primitivní.

5.1.3 Tvorba Vazeb a jejich vlastností

Tvorba vazeb je podobně jako přiřazování vlastností metoda volaná na nějakém nodu, ten se automaticky stane výchozím pro danou vazbu. Do parametrů metody se nejdříve vloží cílový node a jako druhý parametr se vloží label vazby, kterou chceme tvořit.

```
startNode.createRelationshipTo(endNode,labelName);
```

Pokud je potřeba vazbě přiřadit nějaké vlastnosti, je potřeba na tuto vazbu získat odkaz. To je možné více způsoby. Buď se vazba vyžádá přímo od databáze dotazem na vrácení vazby mezi nodem **A** a **B**, nebo je možné si vazbu vytvořit samostatně, jak je ukázáno níže a k nodům jí přidat až později.

```
(1)Relationship r;
(2)r.setProperty(propertyName,PropertyValue);
```

Stejně jako v případě vlastností nodů si i u vazeb Neo4j vytváří mapu vlastností dané relace.

---

[23] Prvotní třída v jazyce Java, z které dědí všechny zbylé třídy.

[24] Myšleno jako datový typ Map (key,value).

6 Testování

Při testování byl kladen důraz na rychlost zpracování daného dotazu. Čas, za který se dotaz provede, bude měřen tak, že se před spuštěním dané metody zaznamená aktuální čas (1).

Pak se provede daná metoda (2). Metody budou mít vždy za účel získat parametr **IdAgenta**, nebo více agentů.

Po skončení se znovu změří aktuální čas (3).

Od druhého měření času se odečte hodnota prvního měření (4). Tím získáme dobu, která byla nutná ke zpracování. Čísla uváděná v tabulkách budou tudíž v řádech milisekund. Testů bylo vždy provedeno třicet a poté byly zprůměrovány. Tento průměr je zapsán v tabulkách níže.

```
(1)long start_time = System.nanoTime();
(2) giveMeAgent();
(3)long end_time = System.nanoTime();

(4)doubl edifference = (end_time - start_time)/1e6;
```

Testování bylo prováděno na notebooku Samsung s parametry:

Tabulka 1 Parametry testovacího počítače

Operační systém	Windows 10 64bit
Procesor	Intel i7-3630 2,40GHz
RAM	8GB

6.1 Zapsání dat do databáze

V této podkapitole bude věnována pozornost náročnosti zapsání dat do databáze.

6.1.1 Zapsání nodů:

Ukázka kódu, je uvedena v přílohách: příloha 1

Metoda na zápis nodů vyžaduje některé parametry. V těchto parametrech je seznam všech agentů, které je potřeba zapsat do databáze. Dále je také předáván odkaz na danou databázi. Do pomocného parametru part, který je tvořen arrayListem[25], je načtena vždy část agentů, která bude zapsána. V závislosti na hardwarových prostředcích, které jsou k dispozici, se udává množství agentů, které má být zpracováno. Podmetoda writePartNode už zajišťuje skutečné zapsání do databáze. Toto zapsání probíhá podle postupu popsaném v předchozí části.

6.1.2 Zapsání vazeb:

Ukázka kódu, je uvedena v přílohách: příloha 2

Zapsání vazeb je podmíněno tím, že byly předem zapsány všechny Nody, na které se budou vazby vázat. Pokud by se vazby vytvářely rovnou s Nody, došlo by k výjimce - odkazování se na neexistující Node.

Vazby jsou vytvářeny postupně. Vždy se vezme část agentů, protože při pokusu o zapsání všech najednou, často docházelo k překročení fyzických možností paměťových médií. Následně je postupně procházen seznam agenta **A**, který obsahuje odkazy na agenty **B, C** atd. Je potřeba zjistit ID agenta **A**, které má v databázi. To je možné několika způsoby. Byl vybrán způsob, který využívá toho, že agenty jsou zapisování do databáze postupně. Tudíž v databázi Neo4j mají agenty stejné ID[26], jako je index v seznamu všech agentů v simulaci. Stačí tedy

[25] Datová struktura představující pole prvků daného typu.

[26] ID v databázi se přiděluje automaticky a jedná se o jiný parametr než IdAgenta.

zjistit, jaký index má agent v seznamu ASE. Následně si vyžádat node na pozici označené tímto indexem. Pak už se vytvoří vazba podle postupu popsaným v předchozích částech práce.

Tabulka 2 Měření zápisu dat do Neo4j

Nodů	100000	200000
Vazeb	5	5
Doba zápisu Nodů	6599(ms)	11410.782883(ms)
Doba zápisu vazeb	802795(ms)	3606388.015636(ms)
Doba celkového zápisu	809795(ms)	3617800.113922(ms)

Z tabulky výsledků je možné vyčíst, že zapsání do databáze je velmi náročné. Část, ve které se zapisují samotné **Nody** je obsloužena dostatečně rychle. Avšak jakmile program začne pracovat na zapisování vazeb mezi danými **Nody,** situace se značně zkomplikuje a potřebný čas rapidně stoupá. Počet všech vazeb je závislý na počtu nodů **N** a počtu vazeb **V**.

Ve vztahu $N * V = X$.

Kde **X** je výsledný počet vazeb. Když se porovnají výsledky z prvního a druhého měření, je vidět, že když se zvedl počet agentů dvojnásobně, tak doba zápisu Nodů se nezdvojnásobí, ale je pouze jeden a půl krát větší. Zato čas potřebný na zápis vazeb, byl potřeba čtyři a půl krát větší.

6.2 Získání odpovídajícího agenta

Hlavním důvodem pro nasazení Neo4j je hypotéza, že získávání potřebných informací, v podobě agentů, kteří splňují určité požadavky, bude s Neo4j rychlejší, než s využitím čisté Javy. V té to podkapitole bude tato domněnka testována.

6.2.1 Vyhledávací algoritmy

ShortestPath

Tento algoritmus nalezne nejkratší cestu mezi dvěma nody.

Pokud v dotazu poslaném na databázi není uveden algoritmus, implicitně je použit ShortestPath.

AllSimplePaths a AllPaths

Tyto algoritmy jsou na hledání více možných cest mezi danými nody. Avšak pro potřeby simulace ASE, stačí najít jednu cestu a to tu nejkratší. Další hledání by vytvořilo pouze časovou zátěž a přidaná hodnota by zde nebyla.

Dijkstra

Dijkstrův algoritmus se využívá k hledání cest tvořené vazbami, které jsou ohodnoceny. To, jak se ohodnocují vazby, bylo popsáno v části *Váženost grafu*. Jelikož v simulaci ASE nejsou cesty ohodnoceny, není tento algoritmus využíván.

Pro hledání agentů bude v práci použit algoritmus ShortestPath.

6.2.2 Vyhledání agenta daného typu a vlastností

Jedná se o základní získání agenta, který odpovídá daným parametrům. V tomto případě bude hledán agent, který nabízí nějaký produkt. To, o jaký produkt přesně jde, už není podstatné a metoda se v závislosti na daném produktu nemění.

Neo4j:

Ukázka kódu, je uvedena v přílohách: příloha 3

Metoda spustí připojení na databázi přes odkaz **db**. Tam je pomocí metody **execute** spuštěn dotaz v jazyce **Cypher:**

```
MATCH (A:FIRM) WHERE (A.Produkt = {AProdukt} RETURN A LIMIT 1.
```

Hledaný produkt je parametrem metody a ten je předáván spolu s příkazem metodě **execute**. Tento dotaz obslouží databáze a vrátí výsledek. V tomto případě jeden, jelikož je určen limit výsledků na jeden. Následně je procházeno pole výsledků a je hledán sloupec s názvem **A**. **A** proto, že v dotazu byl takto výsledek označen za slovem **RETURN**. Následně je vyhledána vlastnost **IdAgent** a ta je odeslána volajícímu. Pokud není nalezen žádný takový agent, tak je vrácen prázdný řetězec.

Java:

Ukázka kódu, je uvedena v přílohách: příloha 4

V parametrech metoda potřebuje znát produkt, který má agent vytvářet a seznam všech možných agentů. Přes cyklus forEach[27] je pak prohledáván seznam agentů **register,** dokud není nalezen takový, který má požadovanou vlastnost. Následně je hledání ukončeno a nalezené **IdAgenta** posláno volajícímu.[28]. Pokud není nalezen žádný takový agent, tak je vrácen prázdný řetězec.

[27] Typ cyklu v jazyce Java pro procházení prvků seznamu.

[28] Tímto volajícím je nejčastěji některý z agentů.

Výsledky měření:

Tabulka 3 Měření získání prvního vhodného agenta

Počet nodů	1000
Java nativ	0.00290789999999999967(ms)
Neo4j	61.43441923333335(ms)

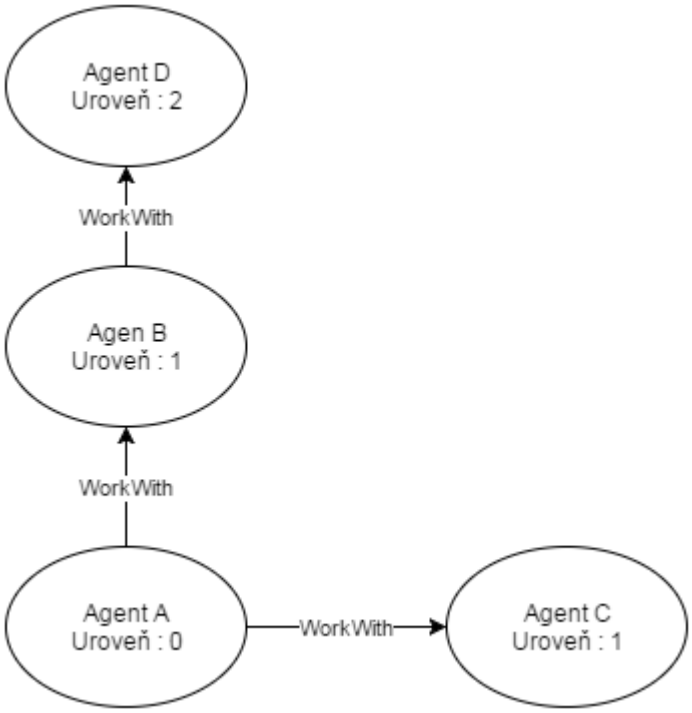
Jak je vidět na výsledcích, rozdíl mezi danými metodami je až extrémně veliký. I když obě metody vracejí totéž.

Klasický způsob využívající čistou Javu, své hledání ukončí, jakmile najde prvního možného agenta.

Neo4j tu nejspíše má problém s přílišnou obecností dotazu a tím, že vyhledává všechny možné výsledky a až pak z nich vytvoří podmnožinu podle parametru **LIMIT,** kterou vrátí volajícímu.

6.2.3 Získání nejbližších vhodných agentů

Vhodný agent se bude hledat do hloubky dvou. Tím je myšleno, že pokud agent **A** bude středem, tak nalezený agent může být vzdálen nejvíce do druhé úrovně. Jak je naznačeno na obrázku níže.



Obrázek 10 Ilustrace hloubky prohledávání

Neo4j

Ukázka kódu, je uvedena v přílohách: příloha 5

Tato metoda vrací seznam nejbližších agentů, kteří odpovídají zadaným parametrům. V tuto chvíli je tímto parametrem produkt. Metoda si na začátku vyžádá jméno produktu, který mají hledaní agenty vytvářet. Dále si vyžádá odkaz na databázi, ve které má tyto agenty hledat. Nakonec bude také žádat odkaz na agenta, který tuto metodu volá.

Metoda pošle dotaz na databázi:

```
MATCH (A:FIRM)-[:WorkWith*1..2]-(B) WHERE (A.IdAgent ={idA}) AND (B.Produkt ={produktB}) RETURN B.
```

S parametry, ve kterých se nachází **ID** volajícího agenta a jméno produktu, který mají vytvářet agenty, které hledáme. V tomto případě nenese dotaz část, která omezuje počet výsledků. Tudíž databáze vrátí všechny možné, ze kterých si až agent vybere ty, které uzná, podle svých preferencí, za vhodné. Agenty jsou hledány do hloubky dva. To znamená mezi agentem **A** a agentem **B** se může nacházet nanejvýše právě jeden agent **C**. Pokud v dotazu

nastane chyba, nebo takový node neexistuje, dotaz se vrátí prázdný. ASE pak vypíše chybovou hlášku fail.

Java

Ukázka kódu, je uvedena v přílohách: příloha 6

Metoda jako její protějšek v Neo4j variantě potřebuje název produktu, který mají hledaní agenty vyrábět. Dále také seznam agentů, který agent **A** zná. Metoda je rozdělena do dvou částí. V první **FindMeAgentNativLevelOne** se prohledá seznam partnerů, který zná samotný agent **A.** Pak se procházejí znovu agenty, které zná agent **A**, ale tentokrát se pošle jejich seznam agentů metodě **FindMeAgentNativLevelTwo** spolu se seznamem už nalezených agentů a jménem hledaného produktu. Prohledá se daný seznam a odpovídající agenty se přidají do seznamu výsledků, který se následně pošle volajícímu agentu **A**.

Výsledky měření:

Tabulka 4 Měření: nalezení vhodných agentů v hloubce 2

Počet agentů/Nodů	1000	1000	1000	1000	100000
Počet vazeb	5	15	30	100	100
Neo4j	25,5	30	39	128,5	123
Java nativ	0,01	0,1	0,48	7	49,2

Z výsledků je patrné, že pro simulaci ASE, bude výhodnější použít vlastní systém vyhledávání a nespoléhat se na Neo4j. Avšak také je zde dobře vidět, že se vzrůstající složitostí, tzn. se vzrůstajícím počtem vazeb na ostatní agenty, vzrůstá časová náročnost čisté Javy v jednotkách řádů. Zatímco Neo4j má pozvolný nárůst. Složitost se zvedla dvacetkrát a Neo4j potřebovalo pouze čtyřikrát více času. Naopak klasická Java potřebovala pět tisíckrát více času. Dále je možné na posledních dvou testech sledovat, že množství agentů (nodů) nemá na Neo4j příliš velký dopad. Naproti tomu u čisté Javy má vyšší množství agentů

značný dopad. Nejspíše kvůli různorodosti agentů a tudíž i nutnosti častějšího zaznamenávání nového agenta do seznamu.

7 Závěr

Při psaní této práce, bylo nutné se velmi důkladně seznámit s fungováním a realizací multi-agentních systémů, což bylo zároveň i důležité k vytvoření logiky agentů, kteří se nacházejí v projektu ASE.

Zapojení grafové databáze Neo4j probíhalo bez obtíží. Hlavním výhodou byla velmi dobrá dokumentace ze strany Neo Software a dále také velké množství tutoriálů, které jsou k dispozici na stránkách Neo Software[29]. Tyto pomocné materiály hrály roli při výběru Neo4j, jako zástupce grafových databází.

Z výsledků provedených testů vyplývá, že simulaci ASE bude v tuto chvíli lépe bez grafové databáze Neo4j. Nejspíše je však jen otázkou času, kdy bude nutné grafovou databázi využít. Grafová databáze má totiž oproti klasickému řešení v Java mnohem menší nárůst časové náročnosti při zvýšení složitosti.

Dále je otázkou, jak by se změnily podmínky při zadání požadavku prohledávat okolí agenta do hloubky tří, nebo více vrstev. V Neo4j stačí změnit pouze jeden parametr, ale v jazyce Java, by bylo nutné přepsat celé metody na vyhledávání agentů. Největší nevýhoda Neo4j pro simulaci ASE, je doba potřebná pro zapsání dat do databáze, která překračuje únosné meze.

[29]http://neo4j.com/docs/stable/

8 Bibliografie

1. **Neo Software.** What is Cypher? *The Neo4j Manual.* [Online] Neo Software, 2016. [Citace: 12. 4 2016.] http://neo4j.com/docs/stable/cypher-introduction.html.

2. **Turing, Alan.** turing.pdf. *Computer Science and Electrical Engineering.* [Online] 2015. [Citace: 12. 25 2015.] http://www.csee.umbc.edu/courses/471/papers/turing.pdf. 49: 433-460..

3. **Pfister, Wally.** *Transcendence.* Alcon Entertainment, Syncopy Films, 2014.

4. **Mintzberg, Henry.** *The Structuring of organizations.* Michigan : Prentice-Hall, 1979. 0138552703.

5. **Vlach, Jan.** *Synchronizace a chování v multiagentním systémů.* Hradec Králové : Univerzita Hradec Králové, 2014.

6. **Kelemen, Jozef.** *Strojovia a agenty.* Bratislava : Archa, 1994. 80-7115-089-4.

7. **Čapek, Karel.** *R.U.R.* 1921. 9788831720946.

8. **Radim, Lulek.** *Komunikace v multiagentním systému.* Hradec Králové : Univerzita Hradec Králové, 2014.

9. **Kubík, Aleš.** *Inteligentní agenty.* Brno : Computer Press, 2004. 9788025103234.

10. **Software, Neo.** graphacademy. *Neo4j.* [Online] Neo Software, 2016. [Citace: 15. 3 2016.] http://neo4j.com/graphacademy/.

11. **Robinson, Ian, Webber , Jim a Eifrem, Emil.** *Graph Databases.* Sebastopol : O'Reilly Media,, 2015. 978-1-491-93200-1.

12. **FIPA.** FIPA. *http://www.fipa.org.* [Online] FIPA, 2016. [Citace: 20. 2 2015.] http://www.fipa.org/.

13. **Garland, Alex.** *Ex Machina.* Andrew Macdonald, 2015.

14. **Hliněný, Petr.** Diskrétní Matematika. *fi.muni.cz.* [Online] 23. únor 2005. [Citace: 10. 1 2016.] http://www.fi.muni.cz/~hlineny/Vyuka/Old/DIM-distext05.pdf. 456-533 DIM.

15. **Russel, Stuart a Norvig, Peter.** *Artificial Inteligent.* New Jersey : Prentice-Hall, Inc., 1995. 0-13-103805-2.

16. **Wooldridge, Michael.** *An Introduction to Multiagent Systems.* Liverpool : JOHN WILEY & SONS, LTD, 2002. 0,7149691 X.

17. **Bond, Alan a Gasser, Les.** *A survey of distributed artificial intelligence.* Los Angeles : Artificial Intelligence Group, 1988. 90089-0782.

8.1 Seznam Tabulek

8.2 Seznam Obrázků

8.3 Přílohy

8.3.1 Příloha 1: Zapsání Nodů

```
Static ArrayList<Agent>part = new ArrayList<>(100000);
public void writeNodeToDB(List<Agent>registr,
GraphDatabaseServicedb) {
          part.clear();
          for (inti = 0; i<registr.size(); i++) {
               part.add(registr.get(i));
               if (i % 100000 ==0) {
                    writePartNode (path, part, db);
                    part.clear();
               }
          }
          zapisPulNodu(path, part, db);

     }
Private void writePartNode(List<Agent>half, GraphDatabaseServicedb)
{
          try (Transactiontx = db.beginTx()){
               for (Agent agent : half) {
                    Node newNode = db.createNode(Neo4jLabel.FIRM);

                    newNode.setProperty("IdAgent", agent.getId());
                    newNode.setProperty("IdPlatform",
agent.getPlatform());
                    newNode.setProperty("Finance",
agent.getInventory().getFinance());
                    newNode.setProperty("Produkt",
agent.getProdukt());
                    tx.success();
               }
          }

     }
```

8.3.2 Příloha 2: Zapsání vazby

```
Static ArrayList<Agent>part = new ArrayList<>(100000);
Public void writeRelationshipToDB(List<Agent>registr,
GraphDatabaseServicedb) {
          part.clear();
          for (int i = 0; i<registr.size(); i++) {
               part.add(registr.get(i));
               if (i % 5000 ==0) {
                    writePartRel(path, part, db,registr);
                    part.clear();
               }
          }
          zapisCastVazeb(path, part, db, registr);
     }
     Private void writePartRel(List<Agent>part,
GraphDatabaseServicedb,List<Agent>registr) {
          try (Transactiontx = db.beginTx()){
               for (Agent agent : part) {
                    intindexZdrojovehoNodu =
registr.indexOf(agent);
                    Node zdrojovyNode =
db.getNodeById(indexZdrojovehoNodu);
                    for (Agent agentuvPartner :
agent.getPartners()) {
                         intindexCilovehoNodu =
registr.indexOf(agentuvPartner);
                         Node cilovyNode =
db.getNodeById(indexCilovehoNodu);

     zdrojovyNode.createRelationshipTo(cilovyNode,
Neo4jRel.WorkWith);
                    }
               tx.success();
               }
          }
     }
```

8.3.3 Příloha 3: Nalezení agenta Neo4j

```
Public StringfindAgent(GraphDatabaseServicedb,Stringprodukt) {
          String finis = "";
          ExecutionEngineengine = new ExecutionEngine(db);
          Stringquery = "MATCH (A:FIRM) WHERE (A.Produkt =
{AProdukt} RETURN A LIMIT 1"
          Map<String, Object>params = new HashMap<String,
Object>();
          params.put("AProdukt", produkt

          try (Transactiontx = db.beginTx()){
               ExecutionResultresult = engine.execute(query, params);
```

```
                Iterator<Node>it = result.columnAs("A");
                if (it.hasNext()) {
                      finis = (String) it.next().getProperty("IdAgent");
                }else {
                      System.out.println("Fail");
                }
                tx.success();
          }
}
```

8.3.4 Příloha 4: Nalezení agenta Java

```
Public String findMeAgentNativTest1(String produkt,
List<Agent>register) {
          Stringrespond = "";

          for (Agent agent : register) {
                if (produkt == agent.getProdukt().toString()) {
                      respond = agent.getId().toString();
                      break;
                }
          }
          Return respond;
```

8.3.5 Příloha 5: Vyhledání agentů Neo4j

```
public void writeRelationshipToDB(String path, List<Agent> registr,
GraphDatabaseService db) {
          part.clear();
          ArrayList<Node> nodeList=new ArrayList<>(registr.size());
          try (Transaction tx = db.beginTx()){
                nodeList = Lists.newArrayList(db.getAllNodes());
                tx.success();
          }
          for (int i = 0; i < registr.size(); i++) {
                part.add(registr.get(i));
                if (i % 5000 ==0) {
                      zapisCastVazeb(path,part, db,registr,nodeList);
                      System.out.println("Zapsána část vazeb");
                      part.clear();
                }
          }
          zapisCastVazeb(path, part, db, registr,nodeList);
          System.out.println("Zapsána část vazeb");
     }
     private void zapisCastVazeb(String path, List<Agent> part,
GraphDatabaseService db,List<Agent> registr,ArrayList<Node>
nodeList) {
```

```
          try (Transaction tx = db.beginTx()){
               for (Agent agent : part) {
                    NodezdrojovyNode=
nodeList.get(indexZdrojovehoNodu);
                    for(Agent agentuvPartner:agent.getPartners()) {
                         int          indexCilovehoNodu          =
registr.indexOf(agentuvPartner);
                         Node cilovyNode=
nodeList.get(indexCilovehoNodu);

     zdrojovyNode.createRelationshipTo(cilovyNode,Neo4jRel.WorkWith)
;
                    }
               tx.success();
               }
          }
     }
```

8.3.6 Příloha 6: Vyhledaní agentů Java

```
Public ArrayList<String>findMeAgentNativLevelOne(String produkt,
ArrayList<Agent>partners) {
          ArrayList<String>respond = new ArrayList<>(1000);

          for (Agent agent : partners) {
               if (produkt == agent.getProdukt().toString()) {
                    if (respond.contains(agent.getId()) == false) {
                         respond.add(agent.getId());
                    }
               }
          }
               for (Agent agentTwo : partners) {
                         respond =
FindMeAgentNativLevelTwo(agentTwo.getPartners(), respond, produkt,
howMuch);
                    }
          Return respond;
     }
     publicArrayList<String>FindMeAgentNativLevelTwo(ArrayList<Agent
>partners,ArrayList<String>respond,Stringprodukt) {
               for (Agent agent2 : partners) {
                    if (produkt == agent2.getProdukt().toString())
{
                         if (respond.contains(agent2.getId()) ==
                         false) {
                              respond.add(agent2.getId());
                         }
                    }
               }

          Return respond;
```