UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RICARDO BOSCHINI ALBUQUERQUE PASSARELLA

DESENVOLVIMENTO DE UMA ABORDAGEM DE COOPERAÇÃO EM SISTEMAS MULTIAGENTES

FLORIANÓPOLIS
2016

RICARDO BOSCHINI ALBUQUERQUE PASSARELLA

DESENVOLVIMENTO DE UMA ABORDAGEM DE COOPERAÇÃO EM SISTEMAS MULTIAGENTES

Trabalho de conclusão de curso apresentado como pré-requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal de Santa Catarina.
Orientador: Prof. Dr. Elder Rizzon Santos

FLORIANÓPOLIS
2016

Ricardo Boschini Albuquerque Passarella

DESENVOLVIMENTO DE UMA ABORDAGEM DE COOPERAÇÃO EM SISTEMAS MULTIAGENTES

O presente Trabalho de Conclusão do Curso (TCC) foi julgado adequado e aprovado, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação pela Universidade Federal de Santa Catarina.

Florianópolis, 1 de dezembro de 2016.

_____

Prof. Dr. Mário Antônio Ribeiro Dantas
Coordenador do Curso de Ciência da Computação

Banca Examinadora

_____

Prof. Dr. Elder Rizzon Santos

_____

Prof. Dr. Ricardo Azambuja Silveira

_____

Profa. Dra. Jerusa Marchi

RESUMO

Neste trabalho foi realizado um estudo sobre sistemas multiagentes, e particularmente, as abordagens para cooperação entre agentes. A partir desse estudo foi criado um cenário com o seguinte problema: como carros podem cooperar entre si em um cruzamento de modo que não haja necessidade de semáforos. Para solucionar este problema foi desenvolvida uma abordagem baseada em planos compartilhados. Para verificar a eficiência deste plano, foi implementado esta abordagem utilizando o framework JaCaMO. E como comparativo, foi implementado um agente sem capacidade de cooperação, necessitando do uso de semáforos em cruzamentos. Ambos os agentes foram testados no cenário desenvolvido. Os testes mostraram que a abordagem baseada em planos compartilhados leva menos tempo para os agentes se locomoverem até seu destino final.

Palavra-chave: Inteligência artificial, Sistemas multiagentes, Cooperação, JaCaMo, Planos compartilhados

ABSTRACT

This paper has been studied multiagent system, and in particular, some approaches on how agents can cooperation. Using this study, It was possible to create a scenario with the following problem: How can autonomous cars cooperate between themselves in someway that it not necessary the use of semaphore at intersection. It was used an approach of shared plans to solve this problem. And It was implement this approach using the framework JaCaMo so it was possible to analyze this approach efficiency. The cooperative agent was comparable with an agent that didn't know to cooperate and rely on semaphore at intersection. The tests showed that the approach with shared plans takes less time on average to move to their final destination.

Key words: Multiagent system, Cooperation, JaCaMo, Shared plans

LISTA DE ABREVIATURAS E SIGLAS

CDI	Crenças-desejos-intenções
CNET	Contract Net, protocolo para compartilhar tarefas.
FIPA	Foundation for Intelligent Physical Agents
FIPA-ACL	Agent communication language
GPGP	Generalized Partial Global Planning
IA	Inteligência artificial
KQML	Knowledge Query and Manipulation Language
SMA	Sistema multiagente
SPDC	Solução de Problemas Distribuídos e Cooperativos (Cooperative Distributed Problem Solving)
TAEMS	Task Analysis, Environment Modeling, and Simulation (Análise de Tarefas, Modelação de Ambiente e Simulação)

LISTA DE ILUSTAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Os sistemas de computadores tem se tornado cada vez mais eficazes, possibilitando as mais variadas proezas, desde ganhar dos melhores jogadores de xadrez (ou recentemente dos melhores jogadores de Go), até piloto automático de carro.

Atualmente, estamos entrando em um mundo totalmente conectado, o termo regularmente empregado "Internet das Coisas", demonstra que teremos inúmeros computadores conectado entre si, trocando informações e precisando realizar as mais variadas funções em conjunto.

O problema é que para muitas dessas funções, as técnicas de sistemas distribuídos tradicionais não conseguem solucioná-las. Por isso, a área de sistemas multiagentes estuda, entre outros assuntos, como agentes autônomos podem se relacionar a fim de solucionar um determinado problema.

Um sistema multiagentes precisa de mecanismos para que os agentes possam se comunicar, principalmente se o sistema for um mundo aberto e possuir agentes heterogêneos. Por isso, linguagens específicas para agentes precisam ser desenvolvidas e implementadas.

As técnicas de negociação e argumentação fornecem mecanismos para solucionar conflitos entre agentes, como no caso de um agente tentando convencer outro agente realizar uma tarefa.

Os conceitos de teoria dos jogos fornecem uma base teórica para os relacionamentos entre agentes. Embora não consiga fornecer uma resposta devido a complexidade computacional NP-completo na maioria dos casos.

Inevitavelmente, embora os agentes sejam autônomos, eles terão que trabalhar em conjunto em algumas situações. Seja para atingir um objetivo inviável, ou para economizar recursos ao dividir as tarefas em vários agentes.

Várias abordagens estão sendo aplicadas em cooperação de agentes, seja aprendizagem, planos, alocação de tarefas e sociedade. Cada uma tentando solucionar um dos problemas em sistemas multiagentes.

Ao mesmo tempo, está sendo desenvolvido modelos e frameworks como Jadex, MOISE+, Jade, JaCaMo, etc., para estudar e desenvolver sistemas multiagentes.

2. OBJETIVO

2.1. OBJETIVO GERAL

O objetivo geral do projeto é analisar abordagens para a cooperação entre agentes e concretizar um modelo para cooperação em um estudo de caso.

2.1.1 Objetivos Específicos

- Analisar as abordagens e modelos para cooperação em SMA;
- Desenvolver um cenário na área de sistemas multiagentes que exija a cooperação entre agentes;
- Desenvolver uma abordagem para a cooperação de agentes;
- Testar a abordagem de cooperação no cenário definido;
- Analisar e descrever a contribuição deste trabalho em cooperação de agentes em SMA.

2.1.2 Objetivo Específico 1

- Compreender os conceitos gerais de sistemas multiagentes para contextualizar o problema;
- Estudar os conceitos gerais de cooperação de agentes;
- Estudar o estado da arte em cooperação de agentes.

2.1.3 Objetivo Específico 2

- Estudar as principais ferramentas de modelagem e implementação em SMA;
- Escolher uma ferramenta;
- Escolher uma ou mais abordagens;
- Definir os requisitos do cenário;

- Definir métricas para a cooperação do cenário.

2.1.4 Objetivo Específico 3

- Dado o levantamento obtido na 1ª etapa do trabalho (objetivo 1), identificar as abordagens ou modelos mais adequados e interessantes;
- Especificar a aplicação que satisfaça os requisitos definidos no estudo de caso.

2.1.5 Objetivo Específico 4

- Implementação do modelo;
- Implementação do cenário;
- Utilizar uma ferramenta de simulação de multiagentes (JaCaMo é a plataforma provável) para testar a abordagem proposta no estudo de caso proposto;
- Usar as métricas especificadas no Objetivo 2 para analisar os resultados da implementação.

3. SISTEMA MULTIAGENTES

O conceito de sistemas multiagentes (SMA) em inteligência artificial é bem debatido, não tendo uma definição formal. No entanto, uma explicação mais abrangente seria: um sistema que possui múltiplas entidades autônomas com informações e/ou interesses divergentes (SHOHAM, LEYTON-BROWN, 2010). Ou pela definição de WOOLDRIDGE (2009), um sistema multiagente é um conjunto de agentes que interagem entre si trocando mensagem.

Simplificando, um agente pode ser considerado como um sistema de computador capaz de realizar ações independentemente, executando-as em nome do seu dono, ou seja, um agente tem que descobrir o que precisa fazer de modo que satisfaça os objetivos planejados (WOOLDRIDGE, 2009).

Embora não haja um consenso sobre o termo agente, a maioria dos pesquisadores em inteligência artificial concorda com a versão de Wooldridge com definição fraca e forte de agentes (TWEEDALE, ICHALKARANJE et al, 2006).

A definição fraca define o agente como tendo a habilidade para autonomia simples, sociabilidade, reatividade e proativo. Enquanto que a definição forte o agente é considerado um sistema de computador que além das características da definição fraca, é conceitualizado e implementado com características humanas, apresentando noções cognitivas (conhecimento, crenças, intenções), obrigações e até emoções.

A área de sistemas multiagentes (SMA) é multidisciplinar: envolvendo economia, filosofia, lógica, ciências sociais, sistemas distribuídos, inteligência artificial, etc.

A inteligência artificial estuda os componentes de inteligência: a habilidade de aprender, planejar, entender imagens, etc. O estudo de SMA utiliza essas técnicas para construir os agentes (WOOLDRIDGE, 2009).

Sistemas distribuídos fornecem várias características para o estudo de SMA pelo fato de ambos lidarem com vários componentes interagindo entre si, além de apresentarem problemas similares como exclusão mútua, compartilhamento de recursos e deadlocks.

No entanto, sistemas multiagentes se diferem dos sistemas distribuídos tradicionais devido a algumas diferenças. Umas das principais diferenças é que os agentes são considerados autônomos, e assume-se que eles tomam as decisões independentemente ao invés de seguir os

procedimentos definidos durante implementação como em sistemas distribuído tradicional, ou seja, os agentes poderiam realizar ações que os próprios desenvolvedores não pensaram (WOOLDRIDGE, 2009).

Outra diferença importante, é que nem sempre os agentes terão o mesmo proprietário e os mesmo objetivos. Por causa disso, é necessário técnicas de negociação e argumentação para que consigam trabalhar de forma cooperativa ou coordenada. Essa situação normalmente se caracteriza como um *jogo*, conceito amplamente estudado em teoria dos jogos.

Teoria dos jogos é uma teoria matemática que estuda a interação entre agentes independentes (BINMORE, 1992). Recentemente, as técnicas e ferramentas de teoria dos jogos são encontrados em várias aplicações computacionais feitas em pesquisas de sistemas multiagentes.

No entanto, várias técnicas em teoria dos jogos, como Nash equilibrium, foram criadas sem uma visão computacional, e por causa disso, computar a solução destes problemas se mostraram ser computacionalmente muito difícil (NP-completo ou pior) (WOOLDRIDGE, 2009).

3.1 COMUNICAÇÃO

Na visão linguística tradicional, a comunicação tem uma forma (sintaxe), carrega algum significado (semântica), e pode ser influenciado por várias circunstância da comunicação (pragmático).

Na definição de Sperber e Wilson (1996), comunicação é o processo envolvendo duas entidades capaz de processar informação. Sendo que uma entidade modifica o ambiente físico do outro e como resultado, a segunda entidade constrói representações similares ao que já existia para a primeira entidade. Comunicação é significado, informações, proposições, pensamentos, ideias, crenças, emoções e atitudes.

Segundo W. Lambert Gardiner (2008), professor do departamento Communication Studies em Concordia University, comunicação é a cola que suporta todos os sistemas.

Os cientistas da computação já tentam solucionar o problema de comunicação há algum tempo, e vários regras formais foram desenvolvidas para representar as propriedades de comunicação em sistemas concorrentes (HOARE, 1978; MILNER, 1989).

Em um cenário orientado a agentes, as entidades não podem forçar outros agentes a fazer uma ação ou escrever uma informação diretamente dentro do outro agente como se fosse um sistema de distribuído tradicional. Para isso, um agente precisa de ações comunicativas, com o objetivo de alterar as crenças do outro agente.

O filósofo John Langshaw Austin trata as várias formas de comunicação como se fossem ações que tem um determinado objetivo, dando origem à teoria de atos de fala (Speech Acts). Posteriormente, essa teoria se tornou a base para diversas linguagens desenvolvidas especialmente para a comunicação de agentes como, por exemplo, KQML e FIPA-ACL.

Além disso, atos de fala também influenciaram o desenvolvimento de paradigmas de desenvolvimento ao ser usada de forma direta em aplicações de software. Chamado de programação racional, este paradigma de programação se baseia nas motivações dos agentes ao invés de lidar com objetos informacionais. Neste paradigma os atos de fala são tratadas como primitivas, permitindo a construção de programas mais poderosos, fácil para criar e entender (SHOHAM, LEYTON-BROWN, 2010).

3.1.1 Atos de Fala

A teoria atos de fala lida com a comunicação como se fossem ações. Assumindo que estas ações de fala, feitas por agentes, têm um objetivo previamente planejado. Esses atos possuem a mesma característica de outros feitos humanos, no sentido que elas alteram o estado do mundo do mesmo modo que ações físicas.

O filósofo Austin, reconhecido como o criador da teoria dos atos de fala (Austin, 1962), identificou diferentes tipos de verbos performativos. Estes verbos não são verdadeiros ou falsos, pois eles não descrevem, não relatam, nem constatam nada. Tais verbos são ditos por uma pessoa quando estão realizando uma ação. Por exemplo, eu te batizo; Eu te condeno a dez meses de prisão; Declaro aberta a sessão; Ordeno que você saia; Eu te perdôo.

Posteriormente, John Searle estendeu o trabalho de Austin em seu livro (Searle, 1969). Searle identificou várias propriedades que devem ocorrer num ato de fala entre o locutor e o ouvinte para a comunicação ser bem sucedida.

Os pesquisadores de inteligência artificial identificaram que se a teoria atos de fala é importante para interação entre humanos, então agentes autônomos também deveriam usá-la.

Um das teoria com maior inflûencia na teoria de agentes foi desenvolvida por Cohen e Levesque. Eles utilizaram a teoria da intenção (*intent theory*) descrita em (COHEN, LEVESQUE, 1990a) para desenvolver uma teoria na qual os atos de fala são modelados como ações e executadas por agentes racionais com um objetivo previamente planejado (COHEN, LEVESQUE, 1990b).

3.1.2 KQML

Desenvolvida pela Knowledge Sharing Effort (financiada pela DARPA), a KQML (knowledge query and manipulation language) é uma linguagem baseada em mensagens para comunicação de agentes. Esta linguagem define um formato para mensagens de modo que o agente possa mostrar a intenção da mensagem ilocutória.

A KQML possui uma arquitetura baseada em camadas: A camada inferior apresenta funcionalidades para transporte de mensagens ou comunicação. A camada do meio, provê as primitivas com o quais os agentes podem trocar mensagens com significado. E a camada superior

especifica o conteúdo das mensagens, normalmente em alguma linguagem formal como Knowledge interchange Format (KIF) (GENESERETH, 1991) ou SQL (ELMASRI & NAVATHE, 1994).

Uma mensagem em KQML pode ser considerado como um objeto (no sentido de programação orientada a objetos): cada mensagem tem um enunciado performativo (como se fosse a classe da mensagem), e um número de parâmetro (pares de atributos e valores, que podem ser tratados com instâncias das variáveis) (WOOLDRIDGE, 2009).

Um exemplo de mensagem em KQML, seria:

```
(ask-one
        :content (PRICE IBM ?price)
        :receiver stock-server
        :language LPROLOG
        :ontology NYSE-TICKS
)
```

Fonte: (WOOLDRIDGE, 2009).

O significado desta mensagem é que o emissário está perguntando o preço de uma ação da IBM. O enunciado performativo é *ask-one*, essa performativa significa que o agente irá perguntar a outro agente e espera uma resposta. Os outros componentes são os atributos: o campo *:content,* especifica qual é o conteúdo da mensagem, neste exemplo, o preço da ações da IBM; o *:receiver* define para quem é a mensagem; *:language* especifica qual é a linguagem que a mensagem foi escrita, neste exemplo, é LPROLOG; e por último, *:ontology* usa o conhecimentos de ontologia para definir qual a terminologia que será usada na mensagem (ver Ontologia). (WOOLDRIDGE, 2009).

A KQML apresenta o paradigma de comunicação assíncrono, pois embora o transporte das mensagens sejam confiáveis e mantenham a ordem das mensagens, ela não pode garantir a entrega em tempo. Por isso, o sincronismo deve ser feito em nível de aplicação (Reading in Agents, 1998).

Os agentes de KQML possuem uma base de conhecimento virtual (Virtual Knowledge Base - VKB) onde contém as crenças e objetivos do agente. Essas informações armazenadas

internamente são únicas para cada agente e representadas através de diferentes linguagens e paradigmas. Então, nenhum agente pode assumir que outro agente irá usar a mesma representação (Reading in Agents, 1998).

A principal falha de KQML é a falta de semântica bem definida. Isso resultou em implementações únicas. E consequentemente, tornou as comunicações difíceis, pois os agentes não eram entendidos. A especificação definida pela FIPA tenta corrigir este problema (Reading in Agents, 1998).

3.1.3 FIPA-ACL

A linguagem ACL (agent communication language) foi desenvolvida seguindo as especificações definidas pela FIPA (Foundation for Intelligent Physical Agents).

A ACL possui algumas similaridade com KQML: tem uma linguagem externa para mensagens; tem 20 performativas para a definição das intenções dessas mensagens de modo que sejam interpretadas corretamente, e não especifica o uso de uma linguagem no conteúdo da mensagem. Além disso, ela possui uma estrutura de mensagem e de atributos similar ao KQML (WOOLDRIDGE, 2009).

Exemplo de mensagem uma FIPA ACL (FIPA, 1999):

```
(inform
   :sender agent1
   :receiver agent2
   :content (price good2 150)
   :language sl
   :ontology hpl-auction
)
```

Fonte: (WOOLDRIDGE, 2009).

A diferença entre ACL e KQML está nas coleções de performativas disponível em cada linguagem e na definição semântica para a linguagem.

Os desenvolvedores do FIPA ACL sabiam os problemas do KQML e por isso, resolveram dar um compreensão semântica formal para a linguagem utilizando a teoria de Cohen e Levesque sobre atos de fala (como ações racionais) (COHEN, LEVESQUE, 1990b) e particularmente as melhorias realizadas no trabalho de (BRETIER e SADEK, 1997).

A linguagem SL (Semantic Language), uma linguagem formal, foi usada para definir as semânticas da ACL. Com SL, cada representação das crenças, desejos, incertezas e ações que os agentes executem, podem ser formalmente definidos.

As semânticas da ACL mapeia cada mensagem ACL para uma fórmula em SL que define uma restrição que o emissário da mensagem deve satisfazer para ser considerado dentro do padrão FIPA ACL. Essa restrição é definido como condições de viabilidade (*feasibility*).

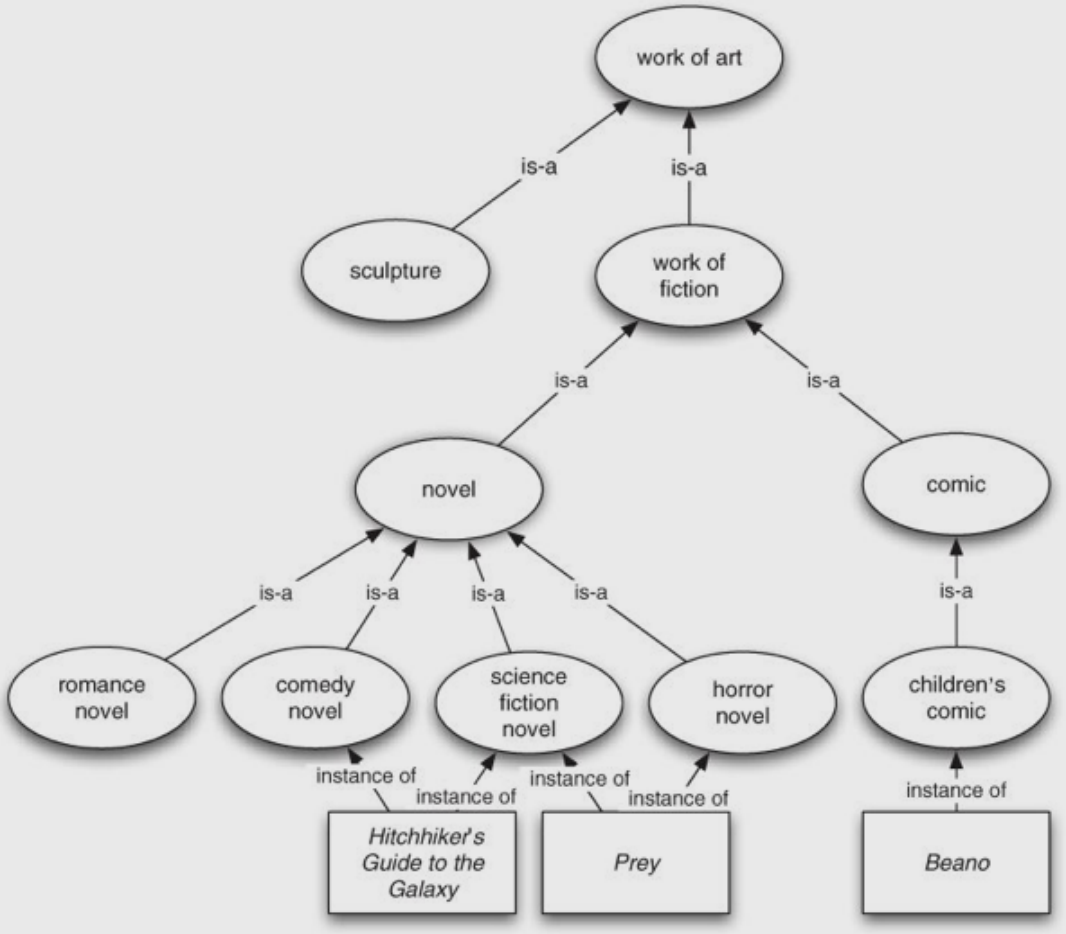
Diversas plataformas para desenvolvimento de sistemas multiagentes foram criadas usando o FIPA ACL. Umas das mais usadas é o JADE (Java Agent Development Environment), que é um framework em java para desenvolver sistemas com agentes FIPA.

3.1.4 Ontologias

Para que agentes consigam se comunicar de forma bem sucedida, eles precisam se fazer entender. O estudo de ontologias tenta resolver este problema. (WOOLDRIDGE, 2009).

Ontologia é uma definição formal de uma base de conhecimento. Essencialmente, o conhecimento é organizado de forma estrutural, sendo que as relações entre estas definições geram classes e subclasses. (HENDLER, 2001).

Figura 1 - Exemplo de uma base de conhecimento e como elas se relacionam. Classes são oval, instâncias são retângulos e as flechas indicam o relacionamento entre as entidades.



Fonte: (WOOLDRIDGE, 2009).

Ontologias podem ser representados com diferentes linguagens como por exemplo, XML ou OWL (Ontology Web Language).

OWL é uma linguagem com a maior importância e influência em linguagens de ontologia (BENCHHOFER et al., 2004). Essencialmente, OWL é uma coleção de frameworks baseados nos frameworks de ontologia em XML.

Figura 2 - Exemplo de uma base de conhecimento em OWL.

```
1. Ontology(
2.    Class(WorkOfArt partial owl:Thing)
3.    Class(Sculpture partial WorkOfArt)
4.    Class(WorkOfFiction partial WorkOfArt)
5.    Class(Novel partial WorkOfFiction)
6.    Class(Comic partial WorkOfFiction)
7.    Class(RomanceNovel partial Novel)
8.    Class(ComedyNovel partial Novel)
9.    Class(ScienceFictionNovel partial Novel)
10.   Class(HorrorNovel partial Novel)
11.   Class(ChildrensComic partial Comic)
12.   DisjointClasses(Sculpture WorkOfFiction)
13.   ObjectProperty(author
14.      domain(Novel) range(Person))
15.   ObjectProperty(content
16.      domain(Novel) range(String))
17.   Individual("Hitchhiker's Guide to the Galaxy"
18.      type(ScienceFictionNovel)
19.      value(author "Douglas Adams")
20.      value(content "Far out in the uncharted
21.      backwaters of the unfashionable end of
22.      the Western Spiral Arm of the Galaxy..."))
23.   Individual("Prey"
24.      type(intersectionOf(HorrorNovel ScienceFictionNovel))
25.      value(author "Michael Crichton")
26.      value(content "Things never turn out
27.         the way you think they will...."))
28.   Individual("Beano"
29.      type(ChildrensComic))
30.   DifferentIndividuals(
31.      "Hitchhiker's Guide to the Galaxy"
32.      "Prey")
33.)
```

Fonte: (WOOLDRIDGE, 2009).

Definindo as classes:

Class (WorkOfFiction partial WorkOfArt)

ou seja, WorkOfFiction é um subconjunto de workOfArt.

As instâncias:

Individual ("Prey" type(intersectionOf(HorrorNovel ScienceFictionNovel)) ...)

Prey é uma instância de HorrorNovel e ScienceFictionNovel.

E as relações:

DisjointClasses(Sculpture WorkOfFiction)

Especificando que as duas entidades são duas *coisas* diferentes.

Em uma base de conhecimento de OWL, é possível analisar e inferir informações novas. Esta análise permite verificar por consistência, ou seja, verificar se existe contradições implícitas ou explícitas; ou tentar resolver o problema de classificação de instâncias através da inferências dos conhecimentos existentes.

3.2 COORDENAÇÃO

A coordenação é um problema de gerenciamento das atividades dos agentes que dependem um do outro. E o mecanismo de coordenação é essencial se as tarefas que os agentes precisam executar estão correlacionadas. Por isso, coordenação é considerado um dos principais problemas em trabalhos cooperativos (WOOLDRIDGE, 2009).

Frank von Martial sugeriu uma topologia para relacionar os tipos de coordenação. Separando essas relações entre atividades positivas ou negativas.

Relações positivas são todas as relações entre dois planos de agentes distintos na qual seja possível ter algum benefício para pelo menos um dos planos ao combiná-los (von Martial, 1990). Essas relações podem ter sido pedida (por exemplo, "Eu explicitamente pedi para você me ajudar com minhas atividades") ou não pedida (Ao trabalhar em conjunto, é possível encontrar uma solução que irá beneficiar pelo menos uma das partes, sem prejudicar o outro).

As relações negativas foram divididas em três tipos (von Martial, 1990):

- Relação de ações iguais: Quando os planos de ambos os agentes são idênticos, e ao reconhecer isso, uma das partes pode executar a ação sozinho e salvar o esforço da outra;
- Relação por consequência: As ações do plano de um agente tem o efeito colateral de completar um objetivo de outro agente, salvando a necessidade desse segundo agente em explicitamente realizá-la;
- Relação favorável: Alguma parte do plano de um agente tem o efeito colateral em contribuir para completar um objetivo de outro agente, por exemplo, solucionando alguma pré condição.

Segundo Frank von Martial, coordenação em sistemas multiagentes deve ocorrer em tempo de execução, ou seja, os agentes precisam ser capazes de identificar essas relações e quando necessário, gerenciar como parte de suas atividades (VON MARTIAL, 1992).

O problema de coordenação de atividades em Distributed Vehicle Monitoring Testbed (DVMT) é tratado com planejamento parcial global (partial global planning) (DURFEE, 1988,

1996; DURFEE, LESSER, 1987). O objetivo principal é a cooperação dos agentes ao trocar informação de modo que encontrem um consenso na solução do problema. O planejamento é *parcial*, pois o sistema não consegue gerar o plano para o problema inteiro. E é *global,* porque os agentes compartilham seus planos locais a fim de atingir uma visão mais ampla.

Outra abordagem realizada é a coordenação através de intenções conjuntas. As intenções apresentam um papel crítico na coordenação ao prover tanto a estabilidade, quanto a previsibilidade do que é necessário para interação social; além de fornecer a flexibilidade e a reatividade necessária para alterar o ambiente.

Em (LEVESQUE et al., 1990), foi observado que fazer parte de um time resulta em algum tipo de responsabilidade com os outros membros. Por exemplo, se um indivíduo descobre que uma tarefa não irá funcionar, ele deve pelo menos tentar avisar os outros.

Assim, Jennings define a diferença em commitment (compromisso) que sustenta uma intenção e a convenção relacionada a este pensamento (JENNINGS, 1993a). O *compromisso* é uma promessa; e *convenção* (convention) é o monitoramento do compromisso, ou seja, especifica quando um compromisso pode ser abandonado e como os agentes irão lidar caso isso aconteça.

Quando um grupo de agentes fazem parte de atividades cooperativas eles devem ter um compromisso com o objetivo geral, além dos seus compromissos individuais para as tarefas que foram atribuídos.

Levesque definiu este tipo de cooperação em noções de joint persistent goal (JPG - Objetivo de persistência em conjunto) (LEVESQUE et al. 1990). Em JPG, um grupo de agentes tem um compromisso coletivo em relação a algum objetivo e uma motivação para este objetivo.

Jennings notou que os compromissos e convenções poderiam ser codificado num sistema baseado em regras ao estudar o uso de compromisso em JPG no sistema de cooperação industrial chamado ARCHON (JENNING, 1993a, 1995).

Em (WOOLDRIDGE, JENNINGS, 1994, 1999), é apresentado um modelo formado por quatro etapas para solução de problemas distribuídos e cooperativos (SPDC):

1. Reconhecimento: SPDC começa quando algum agente numa comunidade multiagente tem um objetivo e reconhece a vantagem da cooperação para atingir seu intento.

2. Formação do time: Ao concluir o *Reconhecimento*, o agente irá solicitar ajuda, se bem sucedido, haverá um grupo de agentes com um *compromisso* coletivo para atingir um objetivo.

3. Formação do plano: O time precisa chegar um acordo sobre quais ações eles devem seguir para atingir seu objetivo. Nessa situação os agentes podem usar técnicas de negociação.

4. Ação em equipe: os agentes executam o plano em conjunto. Mantendo uma relação definida por uma convenção (*convention*) que todo agente deve seguir.

A formação de times é muito comum na sociedade humana. Naturalmente, há vários casos que um único indivíduo é incapaz de solucionar uma tarefa sozinho, seja devido a falta de especialidade, recursos ou informações. Consequentemente, a única alternativa é a cooperação.

(OKIMOTO, 2015) apresentou um framework para analisar a robustez do time, particularmente nos casos de entrada e saída de agentes. Os autores apresentam o algoritmo ART capaz de computar a robustez de um time, e um outro algoritmo que busca em computar toda a troca de um time.

3.3 NEGOCIAÇÃO

O objetivo da negociação (ou barganha) é chegar em um acordo entre as partes, e principalmente, um acordo a fim de resolver os objetivos e as preferências em conflito.

Wooldrige (2009) apresenta como a negociação pode ser formalizada e implementada: o agente teria um *conjunto de negociação* que contém as propostas possíveis que o agente pode fazer; haveria um protocolo definindo as propostas permitidas para o agente; cada agente teria uma *estratégia* privada que determina qual proposta o agente irá fazer; e por fim, uma *regra de acordo* que estabelece quando houve um acordo na negociação.

Dessa forma, em uma negociação, as propostas do agente são definidas pela estratégia, o agente escolhe uma proposta do conjunto de negociação que seja permitida, assim dito pelo protocolo. E caso haja um acordo, como definido pela *regra de acordo,* a negociação termina em acordo. Normalmente, uma negociação continua por várias rodadas, sendo apresentando várias proposta em cada.

A complexidade da negociação aumenta caso estiver lidando com múltiplas questões, neste caso, o agente precisa negociar vários atributos muitas vezes correlacionadas. Por exemplo, na compra de uma casa, o comprador precisa negociar, não só o preço, mas também a forma de pagamento (entrada e financiamento), extras como móveis e assim por diante. A existência de múltiplos atributos podem levar a um crescimento exponencial de possíveis soluções. Bem diferente quando somente tem uma única questão: o comprador só pensa em pagar o menor possível e o vendedor o máximo possível, criando uma simetria nas preferências dos agentes.

Além disso, outra forma de aumentar a complexidade é a quantidade de agentes envolvidos na negociação: Negociação Um-para-um, um agente negocia com outro agente, ou seja, o caso de simetria de preferências é um tipo de negociação Um-para-um. Negociação Muitos-para-um, um único agente negocia com vários agentes. Leilões é um tipo de negociação muitos-para-um. Negociação Muitos-para-Muitos, vários agentes negociam com outros simultaneamente. No pior caso, com n agentes envolvidos na negociação no total, significa que pode até $n(n-1)/2$ tópicos de negociação ocorrendo, podemos considerar as negociações em bolsa de valores como muitos-para-muitos.

O processo de negociação pode se tornar ainda mais complexa se levarmos em consideração a possibilidade de um agente tentar enganar os outros.

Embora a alta complexidade, o processo de negociação auxilia na solução de vários problemas de sistemas multiagentes. Por exemplo, (Kraus, 2001) aplica negociação para o problema de alocação de recursos: os agentes precisam negociar o uso de algum recurso que, embora seja renovado infinitamente (no sentido de quando um agente terminar de usar, o próximo pode começar), não pode ser usado por mais de um agente ao mesmo tempo. A negociação ocorre para determinar quando cada agente terá acesso ao recurso, levando em consideração as preferências individuais dos envolvidos.

3.4 ARGUMENTAÇÃO

Existe certas situações que nós precisamos convencer uma ou mais pessoas sobre algo. Por exemplo, na situação de um aluno apresentando seu trabalho de conclusão de curso para uma banca de avaliadores, os membros da banca precisam estabelecer uma posição racionalmente justificável no que diz respeito aos vários argumentos que foram colocados. Estes precisam decidir se o aluno mostrou domínio no assunto exposto e resultados compatível com um aluno de graduação. Caso todos argumentos são coerentes entre si, então apresentar um posição será fácil, pois não há nenhuma discordância. No entanto, nem sempre isso acontece. Para chegar numa posição racionalmente justificável é necessário rejeitar (ou pelo menos ignorar) alguns dos argumentos.

Isto é, argumentação lida com a inconsistência nas crenças de múltiplos agentes. (Wooldrige, 2009) apresenta argumentação como sendo um provedor de princípios técnicos para lidar com incoerência, e em principalmente, para extrair as posições racionalmente justificáveis de um conjunto de argumentos inconsistentes.

Em (Dung, 1995; Vreeswijk and Prakken, 2000), os autores abstraem a argumentação identificando quais argumentos "atacam" outros argumentos, além de tentar identificar quais são os argumentos fortes. Posteriormente, (Besnard and Hunter, 2008) desenvolvem a argumentação dedutiva, baseando os argumentos em fórmulas lógicas e o conceito de "ataque" entre argumentos é construído a partir de uma prova lógica. A argumentação dedutiva ajuda a resolver a falta de estrutura na abstração argumentativa, por causa disso, o argumento era considerado atômico, o que não é comum na prática.

O conhecimento em argumentação é amplamente usado em sistemas multiagentes, principalmente quando há agentes heterogêneos. Em (El-Sisi, MOUSA, 2014), os autores utilizam argumentação para resolver um problema de negociação entre agentes. Ao invés de usar somente os conceitos de barganha na qual há ofertas e contraofertas de preços, os conceitos de argumentação proporcionam a possibilidade de alterar as crenças das outras parte envolvida, garantindo um maior poder de barganha.

3.5 TEORIA DOS JOGOS

As técnicas e ferramentas de teoria dos jogos são encontrados em várias aplicações computacionais feitas em pesquisas de sistemas multiagentes. Por isso, será apresentado alguns conceitos de teoria dos jogos. Embora várias técnicas em teoria dos jogos, como Nash equilibrium, se mostraram computacionalmente muito difícil (NP-completo ou pior) (WOOLDRIDGE, 2009).

A teoria dos jogos começou como uma teoria econômica, no entanto, hoje em dia, ela é aplicada nos mais variados campos, como ciência política, biologia, psicologia e mais recentemente, na ciência da computação (SHOHAM, 2008). A inteligência Artificial aproveitou o conhecimento já existente em teoria dos jogos para estudar a interação entre agentes independentes aplicando uma visão matemática (SHOHAM, LEYTON-BROWN, 2010).

A teoria dos jogos estuda o problema de como estratégias de interações podem ser criadas para maximizar o bem estar de um agente quando se relacionar com outros agentes. Além de definir como protocolos e mecanismos podem ser arquitetados com propriedades desejadas (PARSONS, WOOLDRIDGE, 2002).

Um problema em teoria dos jogos, ou um jogo, ocorre quando cada jogador tem um número possível de estratégias, criando a possibilidade de várias situações finais diferentes, dependendo qual estratégia cada agente escolher. Como cada agente é livre para escolher sua estratégia, ele o fará considerando a estratégia mais provável que os outros agentes escolheriam. (SHOHAM, LEYTON-BROWN, 2010).

Um jogo pode ser solucionado ao definir funções de utilidade para cada estratégia dos agentes envolvidos e cada consequência ser representada por um número real, indicando se uma estratégia é boa ou ruim para um determinado agente (PARSONS, WOOLDRIDGE, 2002).

A função de utilidade é uma quantificação dos interesses de um agente envolvido no jogo, sendo que estes interesses são influenciado pelas preferências individuais e as incertezas de cada agente. (SHOHAM, LEYTON-BROWN, 2010).

Esta função é fundamentada no conceito de preferências de um agente, sendo que a teoria com maior influência é a Neumann-Morgenstern:

Seja O definido como um conjunto finito de resultados. Para qualquer par $o1,o2 \in O$, seja $o1 \succcurlyeq o2$ definido como proposição que o agente fracamente prefere o1 em relação a o2. Seja $o1 \sim$

o2 definido a proposição que o agente é indiferente entre o1 e o2. Finalmente, por o1 $\succ$ o2, define a proposição que o agente estritamente prefere o1 em relação a o2 (SHOHAM, LEYTON-BROWN, 2010).

Em sistemas multiagentes, a utilidade fornece uma forma conveniente para codificar as preferências de agentes. No entanto, para construir agentes mais razoáveis, os pesquisadores costumam combinar probabilidades com cálculos de utilidade para cada ação e calcular a *utilidade esperada* (expected utility) para cada. (PARSONS, WOOLDRIDGE, 2002).

Agentes sempre vão querer maximizar suas funções de utilidade, ou seja, um jogador irá sempre escolher a maior função de utilidade. No entanto, em um jogo com mais de um jogador, a decisão de cada agente irá influenciar os possíveis resultados do jogo, podendo criar uma situação de incerteza.

Um jogo na forma normal (*normal form*), ou forma estratégica, é a representação mais comum das possíveis estratégias de interação em teoria dos jogos. Nesta forma, cada estado representa uma situação no mundo real, e a combinação das estratégias de cada jogador tem um estado como resultado (SHOHAM, LEYTON-BROWN, 2010).

A forma mais comum de representar um jogo é através de uma matriz de n dimensões, onde cada linha representa uma possível ação para o jogador 1, cada coluna representa uma possível ação para o jogador 2 e cada célula mostra a função de utilidade do jogadores (SHOHAM, LEYTON-BROWN, 2010).

Figura 3 - Dilema do prisioneiro.

	C	D
C	$-1, -1$	$-4, 0$
D	$0, -4$	$-3, -3$

Fonte: (SHOHAM, LEYTON-BROWN, 2010).

No exemplo acima, é o famoso exemplo conhecido como Dilema do Prisioneiro: duas pessoas foram presas e elas tem duas opções, dedurar o amigo (escolher D) ou ficar em silêncio (escolher C). Na situação que os dois se deduram, ambos resultam na função de utilidade -3, se

ambos ficarem em silêncio, uma função de utilidade -1, e se um dedurar e outro ficar em silêncio, uma utilidade zero para quem dedurar e -4 para quem ficar em silêncio. O melhor resultado deste jogo seria se ambos os jogadores cooperassem a fim de chegar no resultado *CC*.

A estratégia é um conjunto de opções de um jogador onde o resultado final depende das ações desse jogador e dos outros jogadores envolvidos. Em teoria dos jogos a estratégia é um algoritmo completo de como um jogador deve agir em cada situação do jogo. (SHOHAM, LEYTON-BROWN, 2010).

Uma estratégia é chamada de Eficiência (ótimo) à Pareto quando um jogador não pode encontrar um resultado melhor sem piorar algum outro jogador. (SHOHAM, LEYTON-BROWN, 2010).

Nash Equilibrium ocorre quando nenhum agente gostaria de mudar sua estratégia se soubesse a estratégia dos outros agentes envolvidos.

Strict Nash Equilibrium: Uma estratégia s=(s1,...,sn) é estritamente um Nash equilibrium se, para todos os agentes i e para todos as estratégias:

$s'i \neq si$, $ui(si,s{-}i) > ui(s'i, s{-}i)$.

Todo jogo com um número finito de jogadores e resultados tem pelo menos um Nash Equilibrium (Nash, 1951).

3.6 COOPERAÇÃO

Em biologia evolucionária, cooperação é qualquer tipo de adaptação que evoluiu, ao menos em parte, para aumentar o sucesso dos parceiros sociais do ator. (GARDNER et al, 2009)

Na visão de Stephen Jay Gould, biólogo evolucionista, não há nada na natureza que exige competição. O sucesso em deixar mais descendentes se deve a várias estratégias, como mutualismo e simbiose (ambas são formas de cooperação). Não há, a princípio, uma preferência geral para a seleção natural, seja um comportamento competitivo ou cooperativo.

A comunicação também pode ser tratada como uma tarefa cooperativa, pois as entidades envolvidas querem ser entendidas, ou seja, é o melhor interesse para ambas as partes comunicarem de forma clara e eficientemente. Com essa ideia e se baseando nos atos de fala, o filósofo Paul Grice criou o que é chamado de princípio para cooperação (*Cooperative principle)*. Este princípio define regras, chamadas de Gricean Maxims, para que uma comunicação entre dois agentes seja bem sucedida.

Em um sistema multiagente cada participante precisa interagir um com outro para completar suas tarefas. No entanto, há distinções entre um sistema distribuído tradicional e um sistema multiagente:

- Agentes em um sistema multiagente podem ter sido arquitetados e implementados por diferentes pessoas e podem não compartilhar o mesmo objetivo. Esta situação pode ser classificada como um *jogo*, na qual cada agente age de forma estratégica para atingir o melhor objetivo possível.
- Como agentes agem de forma autônoma (as decisões são feitas em tempo real ao invés de serem decididas em tempo de desenvolvimento), eles precisam ser capaz de coordenar suas atividades e cooperar uns com os outros.

Tradicionalmente, em SMA, a cooperação apresenta os seguintes tipos de atividade: resolução de problemas distribuídos e cooperativos, compartilhamento de tarefas e compartilhamento de resultados.

O estudo de resolução de problemas distribuído e cooperativo (Cooperative Distributed Problem Solving) analisa sistemas que contém agentes com conhecimento distintos mas relacionados e que a solução de problemas do sistema está além da capacidade individual de uma entidade (DURFEE et al., 1989b).

Um sistema de cooperação seria simples se todos os agentes tivessem um objetivo em comum e não se importassem com seus objetivos individuais, mesmo que um agente seja prejudicado no processo.

No entanto, a área de sistemas multiagentes foca em problemas relativos a agentes com interesses próprios. Como cada agente tem seus próprios objetivos, podem ocorrer conflitos de interesses, assim como nas sociedades humanas. Mas como um agente não consegue resolver seu objetivo sozinho, este terá que cooperar com os outros agentes.

A pesquisa em sistema multiagente está concentrada em vários problemas ao arquitetar sociedades de agentes autônomos, por exemplo: como e porque agentes cooperam (WOOLDRIDGE, JENNINGS, 1994), como agentes podem identificar e resolver conflitos (ADLER et al., 1989; GALLIERS, 1988b; LANDER et al. 1991), como agentes podem negociar e entrar em acordo em situações que estão aparentemente em desacordo (EPHRATI, ROSENSCHEIN, 1993; ROSENSCHEIN, ZLOTKIN, 1994).

A literatura propõe dois tipos de problemas que precisam ser considerados para analisar o sucesso de uma implementação de agentes artificiais:

- Coerência (Coherence): pode ser medido na qualidade da solução, na eficiência dos recursos usados, na clareza da operação, e quão bem o sistema degrada na presença de incerteza ou fracasso. (WOOLDRIDGE, 1994).
- Coordenação: a presença de conflitos entre agentes, no sentido de agentes interferindo de modo destrutivo um com outro (o que será necessário tempo e esforço para resolver), é um indicador de má coordenação. Um sistema de coordenação perfeito, os agentes não precisariam se comunicar explicitamente, pois eles seriam previsíveis entre si, uma abordagem seria manter um modelo interno um do outro. (Ver o capítulo 3.2 de coordenação).

As principais questões em resolução de problemas distribuídos e cooperativos são:

- Como um problema pode ser dividido em tarefas menores de forma distribuídas entre os agentes?
- Como pode uma solução do problema ser efetivamente sintetizada a partir dos resultados dos subproblemas ?
- Como podem as atividades gerais de solução de problemas dos agentes serem otimizadas de modo a produzir uma solução que maximiza a métrica de coerência?
- Quais técnicas podem ser utilizadas para coordenar as atividades de agentes de modo a evitar ações destrutivas e maximizar a eficiência.

O compartilhamento de tarefas ocorre quando um problema é decomposto em subproblemas e alocado para diferentes agentes. A principal questão é saber como as tarefas podem ser alocadas em cada agente, considerando que eles costumam ter capacidades diferentes e possuem a autonomia de recusar tarefas, então para a alocação de tarefas precisa haver um acordo entre os envolvidos. Para tais acordos costuma-se usar técnicas de negociação.

O Contract Net (CNET protocol ou CNP) é um protocolo para compartilhar tarefas. Neste protocolo, um agente, agindo como gerente pela duração desta tarefa, informa aos outros agentes da rede sobre a tarefa a ser feita (*task announcement*). Ao receber o anúncio, um agente irá verificar se consegue resolver, considerando seus recursos de hardware e software, e irá fazer uma proposta ao gerente caso afirmativo (*bid*). O gerente, irá considerar todas as propostas recebidas e irá escolher o(s) agente(s) com maior capacidade para resolver o problema.

O CNET se tornou em um dos frameworks mais estudados e implementados, Wooldridge acredita que este fato se deve a sua simplicidade.

Em compartilhamento de resultados os agentes trocam informações de suas soluções. Esta troca pode ser proativa, caso um agente distribua as informações que acredita serem relevantes para outro agente. Ou reativo, quando um agente solicita a informação a outro agente, usando linguagem performativa.

Segundo (DURFEE, 1999), o compartilhamento de informação aumenta a eficiência do grupo devido aos seguintes fatores: *confiança* (soluções independentes podem ser trocadas, identificando possíveis erros); *plenitude* (os agentes podem compartilhar suas visões locais para chegar numa visão global da situação); *precisão* (os agentes podem compartilhar os resultados a fim de aumentar a precisão da solução global); *oportunidade* (mesmo que um agente possa resolver o problema sozinho, ao compartilhar, a solução pode ser encontrada mais rapidamente).

Além dessas questões, a formação de times também é um aspecto estudado tanto em cooperação quanto em coordenação. Essa abordagem está descrita na seção 3.2.

As questões acima constituem os problemas clássicos em cooperação em SMA. No entanto, o estado da arte em cooperação de agentes está utilizando-se de várias abordagens para solucionar problemas em cooperação, como aprendizagem, planos, alocação de tarefas, e sociedade.

Em aprendizagem, os agentes estão recebendo novas informações sobre o sistema e inferindo novas crenças. Estes agentes terão que decidir, por exemplo, quando compartilhar e coletar as informações com outros agentes.

Os agentes começam a trabalhar em conjunto pois o objetivo dos agentes é inalcançável de outra maneira ou porque eles podem economizar recursos (materiais, tempo) caso o façam. Para trabalharem em conjunto eles terão alocar as tarefas entre eles.

Em vários casos, o agente precisa planejar suas ações para atingir o objetivo maior. Seja ordenando as ações ou compartilhando seus planos com outros agentes. Esse caso seria uma abordagem baseada em planos.

E por fim, o agente está inserido em um sistema. Em vários casos, o sistema precisa definir regras que todos os agentes devem seguir, principalmente se estivermos lidando com um sistema aberto, onde agentes heterogêneos podem entrar e sair a vontade. Nessa situação, a abordagem baseada em sociedade utiliza-se de normas para limitar as ações dos agentes.

3.6.1 ABORDAGENS BASEADAS EM APRENDIZAGEM

Um sistema multiagente possui mais de um agente que interagem entre si, e possui restrições no sistema, por exemplo, como os agentes podem a qualquer momento saber tudo sobre o mundo que os outros agentes conhecem (incluindo os estados internos dos outros agentes). Essas

restrições são importantes para definir problemas em sistema multiagentes, caso contrário, os agentes distribuídos poderiam agir em sincronia, sabendo exatamente qual situação os outros agentes estão realizando e qual o resultado esperado. Principalmente porque, na maioria dos casos reais, os agentes não terão todas as informações do sistema. (PANAIT, LUKE, 2005).

Como os agentes não possuem conhecimento total do sistema, eles terão que descobrir sozinhos, através de várias tentativas, como solucionar uma tarefa ou minimizar erros. isso é conhecido como aprendizagem de máquina, uma automatização do processo indutivo.

Comparado com o problema de aprendizagem de máquina em geral, o problema de aprendizagem em sistema multiagentes se destaca por dois fatores. Em primeiro, como no sistema multiagentes o domínio do problema contém múltiplos agentes, o espaço de busca abrangido pode ser extraordinariamente grande. E devido a interação desses agentes, pequenas alterações no comportamento de aprendizagem podem resultar em mudanças imprevisíveis no grupo multiagente como um todo. Segundo, aprendizagem com múltiplos agentes pode envolver múltiplos aprendizes, cada um aprendendo e adaptando no contexto dos outros. Isto introduz questões da teoria dos jogos para o processo de aprendizagem que ainda não são totalmente compreendidos. (PANAIT, LUKE, 2005).

Dependendo do tipo de resposta que o aprendiz recebe para suas ações, existe três abordagens principais em aprendizagem:

- Supervisionada, o aprendiz recebe o resultado exato.
- Sem supervisão, não há nenhuma resposta.
- Baseado em recompensa, o aprendiz recebe uma avaliação da qualidade ("recompensa").

O método mais comum na literatura é a abordagem baseada em recompensa. Essa abordagem pode ser dividida em dois: aprendizado por reforço (estima a função valor). E o método de busca estocástica, como computação evolucionária, recozimento simulado (simulated annealing) ou Subida pela Encosta mais Íngreme (stochastic hill-climbing).

Panait e Luke apresentam uma taxonomia da aprendizagem na cooperação do sistema multiagentes, classificando os problemas em duas classificações maiores, aprendizagem em time e aprendizagem concorrente.

Em aprendizagem em time há somente um aprendiz envolvido. E depois este agente reporta as informações para o time de agentes. Como essa abordagem não possui múltiplos aprendizes, as técnicas tradicionais de aprendizagem podem ser utilizadas, no entanto, elas podem ter problemas de escalabilidade caso o número de agentes aumente devido a interação entre estes agentes.

A aprendizagem de time pode ser dividida em homogênea, quando todos os agentes são atribuídos o mesmo comportamento (mesmo com agentes heterogêneos), neste caso, como todos os agentes possuem o mesmo comportamento, o espaço de busca para o processo de aprendizagem é drasticamente reduzido. E em aprendizagem de time heterogênea, quando o time possui agentes com diferentes comportamentos. Essa abordagem apresenta melhores resultados devido a especialização de agente, mas precisa de um espaço de busca maior. E o caso misto, chamado de híbrido, que busca unir os agentes com comportamentos semelhantes em pelotões, e cada pelotão se comporta como um time de aprendizagem homogêneo. (PANAIT, LUKE, 2005)

A outra categoria de aprendizagem na cooperação do sistema multiagentes é o caso de aprendizagem concorrente, quando há simultaneamente vários aprendizes. Essa abordagem é preferível nos problemas que é possível aplicar decomposição, e quando é útil tentar cada subproblema com certo grau de independência dos outros. Neste caso, haverá uma drástica redução do espaço de busca e complexidade computacional. (PANAIT, LUKE, 2005).

O maior desafio nos problemas de aprendizagem concorrente é que o aprendiz está adaptando seu comportamento ao mesmo tempo que outros agentes estão se adaptando. Esse caso se torna complexo porque um agente está se adaptando ao um ambiente que também está se alterando, logo a adaptação do agente se torna inválida. Esse ambiente dinâmico infringe suposições básicas nas técnicas tradicionais de aprendizagem de máquina. (PANAIT, LUKE, 2005)

Em (ECK, SOH, 2015), os autores utilizam a abordagem de aprendizagem para compartilhamento de informações. Os agentes precisavam decidir se pedem, coletam ou compartilham a informação. Para simplificar o processo de aprendizagem, os autores transformam o estado atual do ambiente em um problema mais simples de melhoramento do conhecimento ao

longo do tempo, numa transformada nomeada de Knowledge State MDP. Essa formulação do problema permite os agentes realizar decisões baseadas no estado atual do seu conhecimento, ao invés do ambiente. Com essa transformada, os autores podem utilizar aprendizagem por reforço para aprender sobre o ambiente e sobre os outros agentes, inviável anteriormente devido a alta complexidade.

3.6.2 LOCAÇÃO DE TAREFAS

A divisão de tarefas é muito importante para qualquer sociedade biológica ou artificial que deseja que seus membros cooperem entre si a fim de atingir algum objetivo em comum. Naturalmente, o problema de alocação de tarefas é muito estudado em sistemas multiagentes, seja um grupo de robôs com a função de explorar uma área (LAGOUDAKIS et al, 2004) ou uma equipe de resgate com bombeiros e policiais precisando lidar com inúmeras emergências na cidades (PUJOL-GONZALEZ et al, 2015). Em ambos os casos os agentes querem otimizar a distribuição de tarefas entre os agentes envolvidos.

Gerkey e Matarić (2004) definem uma tarefa como sendo um dos vários objetivos necessário para completar a meta global do sistema. Tarefas podem ser discretas (por exemplo, entregue esta carta) ou contínua (por exemplo, vigie esta porta).

A complexidade na alocação de tarefas pode variar dependendo do problema: as tarefas são realizadas instantaneamente ou é preciso alocar o tempo de uso dos recursos; os membros são homogêneos ou heterogêneos; as tarefas podem ser realizadas independentes ou estão relacionadas em múltiplas camadas de tarefas; a quantidade de agentes envolvidos; as decisões são centralizadas ou descentralizadas. (KORSAH, DIAS, STENTZ, 2013) (Gerkey, Matarić, 2004).

Um agente precisa ter a capacidade de identificar qual tarefa deve realizar. Para isso, é usado o conceito de teoria dos jogos conhecido como utilidade. Como não existe uma padronização de como calcular a utilidade de uma tarefa, cada autor apresenta a sua forma. Mas de modo geral, a utilidade irá se basear em dois princípios: a qualidade na execução da tarefa, ou seja, o quanto uma tarefa é executada corretamente, dado um ideal hipotético. E o quanto de recursos são gastos na execução da tarefa, podendo considerar recursos financeiros, temporais, energéticos, etc. Por exemplo, um robô que realiza a limpeza do ambiente, o quão limpo o chão deve ficar (qualidade)

e consumo de energia elétrica (gastos de recurso). A relação entre dois fatores irá definir a função de utilidade. O agente usar esta função para calcular as várias possibilidades de utilidades a fim de encontrar a solução ótima, ou seja, a melhor solução possível dado as informações conhecidas e analisadas.

Gerkey e Matarić apresentaram uma taxonomia para problemas de alocação de tarefas em sistema de multi robôs. Caracterizando os problemas em três eixos:

- Tarefa-Única (TU) vs Tarefa-Múltiplas (TM): TU significa que o robô é capaz de executar somente uma tarefa por vez, enquanto que TM significa que o robô pode executar múltiplas tarefas ao mesmo tempo.
- Tarefa Robô-Único (RU) vs Tarefa Robô-Múltiplos (RM): RU significa que cada tarefa precisa de somente um robô para completá-la. Enquanto que RM significa que uma tarefa precisa de múltiplos robôs.
- Atribuição Instantânea (AI) vs Atribuição Prolongada de Tempo (AT): AI significa que toda a alocação ocorre instantaneamente, não existe planejamento futuro. Enquanto que em AT, existe preocupação com as tarefas atuais e com as futuras, além da possibilidade de que algumas tarefas precisam ser executadas numa determinada ordem.

Com essas definições, os autores analisaram cada combinação dos eixos. O caso de TU-RU-AI (Tarefa-Única, Tarefa Robô-Único, Atribuição Instantânea) é considerado uma instância do *Problema de Atribuição Ótimo* (Gale, 1960) - um problema bem estudado em teoria dos jogos e em pesquisas de operações, no contexto de atribuições pessoais. Além disso, é o único problema que pode ser solucionado em tempo polinomial. Já as outras combinações foram consideradas fortemente NP-difícil.

Em TU-RU-AT (Tarefa-Única, Tarefa Robô-Único, Atribuição Prolongada de Tempo) é um problema que envolve determinar um cronograma de tarefas para cada robô como uma instância do problema de agendamento (Brucker, 1998).

O problema TU-RM-AI (Tarefa-Única, Tarefa Robô-Múltiplos, Atribuição Instantânea) é significamente mais difícil e é também comparado com o problema de formação de coalizões.

Apresentado como sendo um problema de dividir conjunto de robôs numa coalizão para solucionar uma tarefa específica, este problema é matematicamente equivalente ao conhecido *problema de particionamento de conjuntos* em otimização combinacional.

O TU-RM-AT (Tarefa-Única, Tarefa Robô-Múltiplos, Atribuição Prolongada de Tempo) pertence a classe de problemas que inclui tanto formação de coalizão quanto agendamento de tarefas.

Já os casos TM-RU-AI e TM-RU-AT (Tarefa-Múltiplas, Tarefa Robô-Único) são bem incomuns. Os autores relacionam o caso TM-RU-AI com TU-RM-AI, na qual as tarefas e os robôs seriam invertidos na formulação do *problema de particionamento de conjuntos*. E o caso TM-RU-AT seria equivalente ao TU-RM-AT.

No caso TM-RM-AI (Tarefa-Múltiplas, Tarefa Robô-Múltiplos, Atribuição Instantânea), o objetivo é tentar computar a coalizão de robôs para executar uma ação, e um determinado robô pode ser alocado para mais de uma coalizão, ou seja, este robô poderá realizar mais de uma tarefa. Os autores associam este problema como sendo um caso de *cobertura de conjuntos,* um problema bem conhecido em otimização combinacional.

E por último, o caso TM-RM-AT é considerado um problema de agendamento de tarefas com múltiplos processadores e máquinas multifuncionais. Os autores apresentam este problema como sendo fortemente NP-difícil.

Devido a complexidade deste problema, há vários estudos e propostas de como resolver problema de alocação de tarefas. Abordagens baseadas em leilão já são bem antigas, em (LAGOUDAKIS et al, 2004), por exemplo, os autores tentam resolver o problema de exploração de robôs na qual cada tarefa consiste numa localização que o robô precisa visitar e eles desenvolvem um algoritmo chamado de alocação PRIM que é mais rápido que leilões combinacionais, e diferente de leilões de item único, produz alocações cujo total seja comprovadamente, no máximo, duas vezes maior que o custo total mínimo.

Outra abordagem foi proposta por (PUJOL-GONZALEZ et al, 2015), os autores usam o algoritmo max-sum para a coordenação entre times. Que é um algoritmo de passagem de mensagem baseado na lei genérica de distribuição (generalised distributive law - GDL)). Aplicando este algoritmo no problema do Robocup 2013, os autores mostram empiricamente que o algoritmo max-sum binário obtém melhores resultados que outros problemas de otimização distribuídos

restritos (distributed constraint optimization problem - DCOP) no estado da arte, conseguindo prevenir o dobro de dano na cidade.

Novas propostas continuam surgindo, a ideia de (WICKE, FREELAN, LUKE, 2015) foi utilizar o conceito de caçadores de recompensa para solucionar o problema de alocação de tarefas. Esta definição é um sistema comum no estados unidos na qual qualquer agente pode se comprometer com uma tarefa, mas somente o primeiro que completa-la ganha a recompensa. A vantagem dessa abordagem, segundo os autores, é que as técnicas tradicionais baseadas em leilões, o agente que ganhou o leilão não necessariamente é o mais capacitado para realizar a tarefa, ou mesmo, se consegue concluí-la. Já na abordagem de caçadores de recompensa, uma mesma tarefa pode estar sendo realizada por vários agentes ao mesmo tempo. O algoritmo proposto, ComplexP, demonstra melhores resultados que leilões tradicionais na simulação de um futebol de robôs, principalmente porque uma tarefa não era exclusiva de um único agente.

3.6.3 ABORDAGENS BASEADAS EM PLANOS

Planejamento é um comportamento natural para qualquer pessoa que deseja atingir um objetivo a longo prazo. Por exemplo, se um estudante tem interesse em terminar a Universidade, ele precisa entregar o trabalho de conclusão de curso. Logo, é necessário planejar os horários que serão dedicados para ler, estudar e escrever o trabalho.

Em inteligência artificial, os agentes podem ser classificados em duas categorias dependendo das técnicas utilizadas para tomada de decisão: agentes reativos utiliza somente as informações atuais para decidir sua próxima ação; e agentes planejadores que consideram situações futuras, inclusive as consequências causadas por suas próprias ações.

Em um ambiente de múltiplos agentes, o problema de planejamento pode ser definido como sendo um problema de planejamento por um grupo de agente e para o grupo de agentes. Exceto no caso de um planejador central, cada agente teria um plano individual (CLEMENT, WEERDT, 2009). Continuando o exemplo do estudante, ele teria que planejar o andamento do trabalho junto com seu orientador, apresentando o andamento do projeto, discutindo o conteúdo e o caminho a ser seguido, ou seja, ambos teriam que criar um plano.

Um plano é uma sequência ordenada de ações que, caso executadas corretamente, resulta em um conjunto de objetivos atingidos para alguns agentes. (CLEMENT, WEERDT, 2009). A maioria dos problemas de planejamento em sistemas multiagentes coincide em três eixos:

- Relação de dependência de uma tarefa. Se uma tarefa é totalmente independente, sem nenhum recurso compartilhado, ou se é totalmente relacionado, e os agentes precisam executar ações em conjunto e possuem recursos em comum.
- Cooperação ou agentes individualista. Em alguns casos o agente está interessado em otimizar somente a sua própria utilidade.
- Comunicação. Se os agentes conseguem se comunicar durante as execuções das tarefas ou se será preciso coordenar as ações previamente.

Várias técnicas já foram usadas para solucionar problemas de planejamento. No caso de planejamento central, o planejador geralmente segue o seguinte algoritmo, generalização de (DURFEE, 2001):

1. Dado um objetivo, um conjunto de operadores, e um estado de inicial, crie um plano parcialmente ordenado;
2. Separe o plano em subplanos;
3. Programe as subtarefas ao considerar os recursos de alocação (inclusive os agentes que irão realizá-la) e as restrições temporais;
4. Comunique os subplanos aos agentes, identificando e resolvendo eventuais conflitos.
5. Inicie a execução do plano.

Essas etapas não são totalmente isoladas, por exemplo, é possível coordenar as ações enquanto os planos estão sendo construídos (combinação das etapas 2, 3 e 4) ou adiar a coordenação para a fase de execução (união de 4 e 5).

Embora a existência de um planejador central, existe outros métodos para distribuir as tarefas e ao mesmo tempo, permitindo uma maior autonomia e privacidade dos agentes, por

exemplo, ao utilizar de leilões (WALSH, WELLMAN, YGGE, 2000) e (WELLMAN, et. al., 2001), e simulações de mercado (WELLMAN, 1993) e (WELLMAN, et. al., 1998).

No caso de planejamento descentralizado, a diferença principal seria que a segunda e terceira etapa do algoritmo acima seriam executadas localmente por cada agente. Sendo que as técnicas de alocação podem ser diferentes para cada indivíduo.

O framework Generalized partial global planning (Planejamento Global Parcial generalizado) agrupa as cinco etapas, definindo que cada agente tem conhecimento parcial dos planos dos outros agentes utilizando uma representação especializada do plano. Em essência, o método seria: Se um agente A informa ao agente B parte de seu plano, B integra essas informações a seu próprio plano parcial global. O agente B pode melhorar o plano global, por exemplo, ao remover redundâncias e apresentar ao outros agentes. Eles podem aceitá-las, rejeitá-las ou modificá-las. Essas etapas ocorrem durante a execução da primeira parte do plano local. (CLEMENT, WEERDT, 2009)

Um importante caso de estudo em planejamento é a coordenação (etapa 4) antes de criarem seus planos (etapa 2 e 3). Um forma dos agentes se coordenarem é usando leis sociais. As leis sociais são convenções aceitas e seguidas por todos envolvidos, por exemplo, todo mundo dirige na pista da direita, não havendo necessidade de comunicar com o outro motorista explícita. As leis sociais podem ser usadas para reduzir o tempo de comunicação e planejamento.

Em problemas de planejamento distribuído contínuo, geralmente, exige que agente quebre e refaça compromissos durante execução, normalmente causados por eventos inesperados ou alterações de objetivos. Em (FARINELLI et al, 2015), os autores apresentam uma forma de interromper o plano para que humanos consigam terminar uma tarefa (seja porque o agente não tem conhecimento completo ou porque ele não tem a capacidade de realizá-la), e em seguida, o agente pode continuar o plano inicial, sendo essa interrupção o mais suave possível.

3.6.4 ABORDAGEM BASEADA EM SOCIEDADE

Um sistema multiagente pode ser estruturado em três níveis: organizacional, coordenação e ambiente, segundo Boissier et al(2012):

- Nível Organizacional: a relação de organizações e normas é chamado de organização normativa. Uma organização define a estrutura e arquitetura do sistema multiagente em termos de grupo, papéis, padrão de cooperação entre atores, recursos, ontologias. E normas definem a governança e controle para uma organização, e dentro dela. Além disso elas podem ser considerado como regulamentações restringindo o comportamento do agente ou suas expectativas.
- Nível de Coordenação: Neste nível, as entidades (agentes ou organizações) interagem entre si. No contexto social dessas entidades, acordos e contratos conectam essas entidades com o nível Organizacional e com outras entidades.
- Nível do Ambiente: O Ambiente é o lugar em que é possível a existência de serviços, recursos e entidades.

O estado normativo possui informações sobre violações, cumprimento e conselhos em onde e quando a organização precisa mudar. Além disso, ao considerar as influências entre o nível Organizacional e o Nível de Coordenação, é possível considerar dois fluxos em busca por controle: fluxo emergente de processamento (vem do sentido de baixo para cima); ou fluxo normativo de processamento (no sentido de cima para baixo). (BOISSIER et al, 2012)

Normas são usadas em sistemas multiagentes principalmente para auxiliar o desenvolvimento de coordenação. Por exemplo, normas podem ser usadas para regulamentar sistema multiagente aberto na qual agentes heterogêneos podem entrar e sair a qualquer momento. (BOISSIER, et al 2012)

Vários autores também utilizam o conceito de política no desenvolvimento de sociedades. Este conceito possui várias interpretações, segundo Kagal et al. (2003), políticas são guias de comportamento para entidades dentro de um particular domínio, guiando o caminho na qual a entidade deve agir ao fornecer regras para restringir seu comportamento.

Embora não tenha um acordo na comunidade sobre a definição, políticas normalmente são consideradas como regras privadas de uma entidade (agente ou organização), enquanto que normas

seriam as políticas que uma comunidade concorda, relacionado um grupo numa sociedade de agentes.

Boella and van der Torre (2004) classifica as normas entre normas regulativas que descreve obrigações, proibições e permissões; e normas constitutivas que regulamenta a criação de fatos institucionais como propriedade, casamento, dinheiro, e modificações no próprio sistema normativo.

Harmon et al. (2008) distingue em dois tipos de política: políticas legais (precisam ser obedecidas) e políticas de orientação (não precisam ser obedecidas).

Já as organizações podem fornecer um caminho para incorporar diferentes políticas ou normas explícitas, seja através de um contexto social ou um gerenciamento do sistema organizacional, além de garantir a supervisão, cobrança ou classificação das normas e políticas.

Uma sociedade de agentes muda e evolui devido a inúmeros aspectos, seja devido as organizações ou as normas que governa o sistema. Elas podem ser endógeno (causado pelos próprios agentes membro da organização) ou exógeno (causado por observadores fora da organização). Neste contexto, a evolução das normas é um importante aspecto da mudança organizacional, tanto normas que fazem parte do contexto externo da organização, quando normas internas (políticas).

Há várias propostas de como usar normas em sistemas multiagentes, sendo que a principal utilidade delas é em coordenar e controlar o comportamento dos agentes. Em (ALECHINA, BULLING, DASTANI, 2015), os autores tentam responder como e quando restringir o comportamento dos agentes a fim de obter a solução desejada. Para isso, eles estudam o problema prático de normas executivas e criam o conceito de *Guarda*. O Guarda são funções que restringe as possíveis ações depois de algum evento. Os autores também propõem um modelo computacional formal de normas, guardas e normas executivas, baseada em lógica temporal em tempo linear com operadores do passado. E concluem que nem todas as normas podem ser aplicadas com funções como Guardas, mesmo com recursos computacionais ilimitado para raciocinar eventos futuros.

(KING et al, 2015) propõem usar organizações para governar e revisar outras organizações, seguindo os exemplos que encontramos no mundo social, como tratados, legislações, acordos. Para isso, é estabelecido uma espaço regulatório definido por normas na forma de um corpo de regulação. Segundos os autores, cada organização projeta suas normas para seus próprios objetivos,

por isso, ao criar uma autoridade maior guiando as regras organizacionais, cada organização terá que coordenar seus projetos com as outras organizações e não poderá impor limites inaceitáveis ao direitos dos agentes. Os autores propõem um framework computacional formal baseado em camadas de organizações, na qual as normas de cada camada é governada e monitoradas pela camada superior.

3.6.5 RESUMO DAS ABORDAGENS

Neste tópico será apresentado um resumo das abordagens descritas acima na forma de tabela. Cada linha desta tabela representa um artigo representando o estado da arte em cooperação de agentes. Os artigos foram destaque da 14ª conferência de Agentes Autônomos e Sistemas Multiagentes (Autonomous Agents and Multiagent Systems - AAMAS) em 2015.

As colunas dividem os tópicos da publicação:

1. Publicação: A primeira coluna é Publicação, apresenta qual é a publicação que aquela linha está se referindo.
2. Abordagem: Em seguida, a coluna Abordagem mostra como os autores da publicação tentaram solucionar o problema proposto em aspectos gerais.
3. Detalhamento da Abordagem: Na terceira coluna é explicado exatamente como os autores lidaram com o problema.
4. Desenvolvimento: Em Desenvolvimento aparece qual foi a proposta de solução pelos autores (normalmente um algoritmo ou framework).
5. Granularidade: Em Granularidade, assume-se qual o nível de atuação que o algoritmo/framework foi desenvolvido. Se foi no agente, no time ou no sistema (afetando também como o agente recebe as tarefas, por exemplo).
6. Ambiente: é apresentado qual o mundo que os agentes estão situados no estudo de caso. Se o ambiente é aberto (os agentes podem entrar e sair); se é dinâmico (o ambiente se altera).
7. Estudo de Caso: descreve o caso específico que o sistema foi validado.
8. Conclusões: É apresentado as conclusões dos autores de seus estudos

Tabela 1 - Abordagens descritas

Continua

Publicação	Abordagem	Detalhamento da abordagem
ECK, Eck; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Aprendizagem	Aprender quando o agente deve coletar e quando ele deve pedir/compartilhar informações
FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Planos (compartilhados)	Permitir interrupções nos planos dos agentes de forma suave, ou seja, o(s) agente(s) consigue(m) continuar a tarefa após a interrupção terminar.
WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)	Alocação de tarefas	Alocação de tarefas baseado em sistemas de caçadores de recompensa: sistema comum no estados unidos na qual qualquer agente pode se comprometer com uma tarefa, mas somente o primeiro que completa-la ganha a recompensa. A vantagem dessa abordagem, segundo os autores, é que as técnicas tradicionais baseadas em leilões, o agente que ganhou o leilão não necessariamente é o mais capacitado para realizar a tarefa, ou mesmo, se consegue concluí-la. Já na abordagem de caçadores de recompensa, uma mesma tarefa pode estar sendo realizada por vários agentes ao mesmo tempo.

Tabela 1 - Abordagens descritas.

Conclusão

Publicação	Abordagem	Detalhamento da abordagem
OKIMOTO, Tenda; SCHWIND, Nicolas; CLEMENT, Maxime; RIBEIRO, Tony; INOUE, Katsumi;	Formação de times	Formação de time orientado a tarefa é o problema de formar o melhor time orientado e

MARQUIS, Pierre; How to Form a Task-Oriented Robust Team, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015		capaz de completar alguma tarefa dado recursos limitados.
PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015) , 2015	Alocação de tarefas	Considerando o problema de simulação de resgate RoboCub (um grupo de policiais e bombeiros precisam trabalhar em conjunto para diminuir os dados na cidade depois de um desastre natural), os autores buscam uma forma eficiente para que diferentes times consigam cooperar e trocar informações.

Fonte: Autor (2016).

Tabela 2 - Publicações.

Continua

Publicação	Desenvolvimento
ECK, Eck; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Para compartilhar informações, os autores desenvolvem uma transformada chamada Knowledge State MDP. Essa técnica simplifica o problema de aprendizagem e permitiu usar aprendizagem com reforço para aprender sobre o ambiente e outros agentes, ao invés de depender de uma coordenação antecipada.
FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Usando Rede de Petri
WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)	Os autores desenvolvem um algoritmo nomeado ComplexP baseado no conceito de caçadores de recompensa

Tabela 2 – Publicações

Conclusão

Publicação	Desenvolvimento
OKIMOTO, Tenda; SCHWIND, Nicolas; CLEMENT, Maxime; RIBEIRO, Tony; INOUE, Katsumi; MARQUIS, Pierre; How to Form a Task-Oriented Robust Team, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Este artigo propõe um framework formal para analisar problemas de robustez ao formar um time orientado a tarefas: Um time é visto com k-robusto (para um inteiro k não negativo), se ao remover algum k agentes do time ainda consegue completar as tarefas determinadas.
PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015) , 2015	Os autores desenvolveram um modelo exclusivo do Tractable Higher Order Potentials (THOPs) para o problema de alocação de tarefas de bombeiros dentro do domínio da simulação do Robocup. Esse modelo permite ser simplificado em problemas com várias áreas funcionais inter-relacionadas, como times de resgates. E por fim, foi aplicado este modelo para coordenação de um time multi funcional de policiais e bombeiros no problemas do RoboCup. Os autores usam o algoritmo max-sum para a coordenação entre times. (Max-Sum é algoritmo de passagem de mensagem baseado na lei genérica de distribuição (generalised distributive law - GDL))

Fonte: Autor (2016).

Tabela 3 - Publicações.

Publicação	Granularidade	Ambiente
ECK, Eck; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Agente	Mundo aberto (agentes entrando e saindo) e ambiente ad hoc.
FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Time	O domínio estudado foi um time de robôs aquáticos coletando informações num corpo d'água.
WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)	Sistema	Ambiente dinâmico - complexidade das tarefas eram alteradas
OKIMOTO, Tenda; SCHWIND, Nicolas; CLEMENT, Maxime; RIBEIRO, Tony; INOUE, Katsumi; MARQUIS, Pierre; How to Form a Task-Oriented Robust Team, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Time	Mundo aberto
PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015) , 2015	Sistema	Problema de resgate do Robocub 2013: mundo aberto heterogêneo.

Fonte: Autor (2016).

Tabela 4 - Estudos de caso.

Continua

Publicação	Estudo de caso
ECK, Eck; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Experimento empírico: quantidade variada de agentes; a rede foi criada aleatoriamente usando um modelo Erdos-Renyi; mundo aberto (10% dos agentes saiam e outros entravam no lugar); agentes tinham sensores com capacidades diferentes;
FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Validado a abordagem numa aplicação de embarcações robóticas
WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)	Testes em uma simulação baseado em experimentos em futebol de robôs. 1-Ambiente estático: agentes não podem abandonar uma tarefa que se compromissou a fazer; 2-ambiente dinâmico: agentes são removidos e realocados no jogo periodicamente; 3-a complexidade das tarefas variavam para cada agente periodicamente; 4-Alguns agentes eram mais lentos em ir atrás de tarefas; 5-Algumas tarefas foram modificadas para serem mais difícil de completá-las para agentes em particular.

Tabela 4 - Estudos de caso.

Conclusão

Publicação	Estudo de caso
OKIMOTO, Tenda; SCHWIND, Nicolas; CLEMENT, Maxime; RIBEIRO, Tony; INOUE, Katsumi; MARQUIS, Pierre; How to Form a Task-Oriented Robust Team, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Dois algoritmos para solucionar este problema foram propostos e avaliados: ART - um algoritmo que computa um time c-custo e k-robusto quando existe. E AORT - um algoritmo otimizado "bi-objective constraint" que busca computar toda troca de time (solução Pareto ótimo).
PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015) , 2015	Problema de resgate do Robocub 2013: Este problema é um ambiente de simulação para benchmark que simula um cenário de busca e resgate urbano quando policiais, ambulâncias e bombeiros precisam coordenar suas ações.

Fonte: Autor (2016).

Tabela 5 - Conclusões.

Continua

Publicação	Conclusões
ECK, Eck; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	A abordagem via KSMDP + modelo baseado em aprendizagem reforçada apresentou maior certeza nas crenças comparado com somente uma abordagem (somente coletar informação ou somente pedir informação); e a certeza aumentava mais rapidamente com um número maior de agentes no mundo. No entanto, com o maior número de agentes no mundo (10 agentes), houve pouco agentes com a resposta correta, isso ocorreu, segundo os autores, devido a memória institucional, ou seja, os agentes davam prioridade a informação compartilhada ao invés de sentir a informação, e o fato dos agentes não terem um sistema de confiança de informação.
FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	O mecanismo de interrupção diminuiu o tempo para completar a missão (em até 48% de redução), e diminuiu a carga no operador (até 80% redução de ações humanas).

Tabela 5 – Conclusões

Conclusão

Publicação	Conclusões
WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015)	Simulações 1, 2 e 3, todas as técnicas convergiram para resultados similares a técnica gulosa de referência, apresentando diferença estatística insignificante; 4- A técnica baseada em caçadores em recompeza, chamada de ComplexP, mostrou resultados superiores comparados com a técnica baseada em leilão, segundo os autores, isso se deve a capacidade de ComplexP compartilhar tarefas e um agente saber a chance dos outros agentes em completar uma tarefa. A técnica de leilão simples não consegue impedir que agentes mais lentos atrasem em completar a tarefa. 5 - ComplexP novamente vence o teste, principalmente pelo fato que uma tarefa mais difícies para um determinado agente pode ser resolvida por outro, não acontecendo o caso de agentes ficarem presos, sem ajuda, tentando completar uma tarefa.
OKIMOTO, Tenda; SCHWIND, Nicolas; CLEMENT, Maxime; RIBEIRO, Tony; INOUE, Katsumi; MARQUIS, Pierre; How to Form a Task-Oriented Robust Team, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015	Experimentos mostraram que (i) um padrão de transição de fase fácil-difícil-fácil pode ser observado com problemas de decisão e (ii) para problemas de otimização "bi-objetive constraint", o número de troca de times aumenta superficialmente com um número maior de agentes.
PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015) , 2015	A modelagem de problemas de interação entre time inter dependente em THOPs permitiu o algoritmo Max-Sum operar em tempo polinomial num ambiente complexo. Os experimentos com o algoritmo alcançam 2.5 vezes melhores resultados que outros algoritmos no estado da arte.

Fonte: Autor (2016).

4. ARQUITETURAS E FERRAMENTAS

Neste capítulo será apresentado o modelo BDI utilizado na concepção da maioria dos agentes. Em seguida, será detalhado o funcionamento do Jadex, um framework para criação de agentes orientados a objetivos seguindo o modelo crença-desejo-intenção (BDI).

A importância de organização em SMA nos faz apresentar o funcionamento de MOISE+, um modelo organizacional para sistema multiagentes baseado nas definições de papéis, grupos e missões.

E por último, o Framework JaCaMo, uma junção de três tecnologias para desenvolvimento de SMA.

4.1 MODELO BDI

O modelo belief–desire–intention (BDI, crença-desejo-intenção) foi inicialmente proposto por Bratman (1987) como uma teoria de raciocínio humano (BRATMAN, 1987). Neste modelo, as causas para ações estão somente relacionadas com desejo ignorando outros aspectos do processo cognitivo (como emoções).

O sucesso desse modelo se deve a sua simplicidade em explicar o complexo comportamento humano, além de apresentar os fundamentos psicológicos que corresponde como as pessoas conversam sobre comportamento humano. (POKAHR, BRAUBACH, LAMERSDORF, 2005)

Rao e Georgeff (1995) definiram crenças, desejos, e intenções como atitudes mentais e representadas, quando possível, como estados mundo. As três atitudes mentais da arquitetura (WIKIPEDIA, 2016).

- Crenças: Crenças representam o estado informacional do agente. Crenças também pode inferir novas regras, e consequentemente, criar novas crenças. Um crença não precisa ser uma verdade do sistema, mesmo que o agente acredite que seja.
- Desejos: Desejos representam o estado motivacional dos agentes. Eles representam os objetivos e situações que o agente gostaria de realizar ou alcançar.

- Intenções: Intenções representam o estado deliberativo do agente, ou seja, o que o agente escolheu fazer.

As intenções de um agente são subconjuntos das crenças e desejos, ou seja, um agente age em busca de um estado mundo que deseja e que acredita ser possível. Para ser computacionalmente tratável, Rao e Georgeff (1995) também propuseram várias simplificações na teoria, a principal é somente as crenças são representadas explicitamente. Desejos são reduzidos para eventos, podendo ser resolvidos em planos pré definidos, e intenção são representados implicitamente, durante execução, pela pilha de planos a serem executados.

4.2 JADEX

Jadex é um framework para criação de agentes orientados a objetivos seguindo o modelo crença-desejo-intenção (CDI). O objetivo é construir uma camada de agentes racionais que sobreponha a infraestrutura mediadora e permita a construção de agentes inteligentes usando os conceitos de engenharia de software (POKAHR, BRAUBACH, LAMERSDORF, 2005).

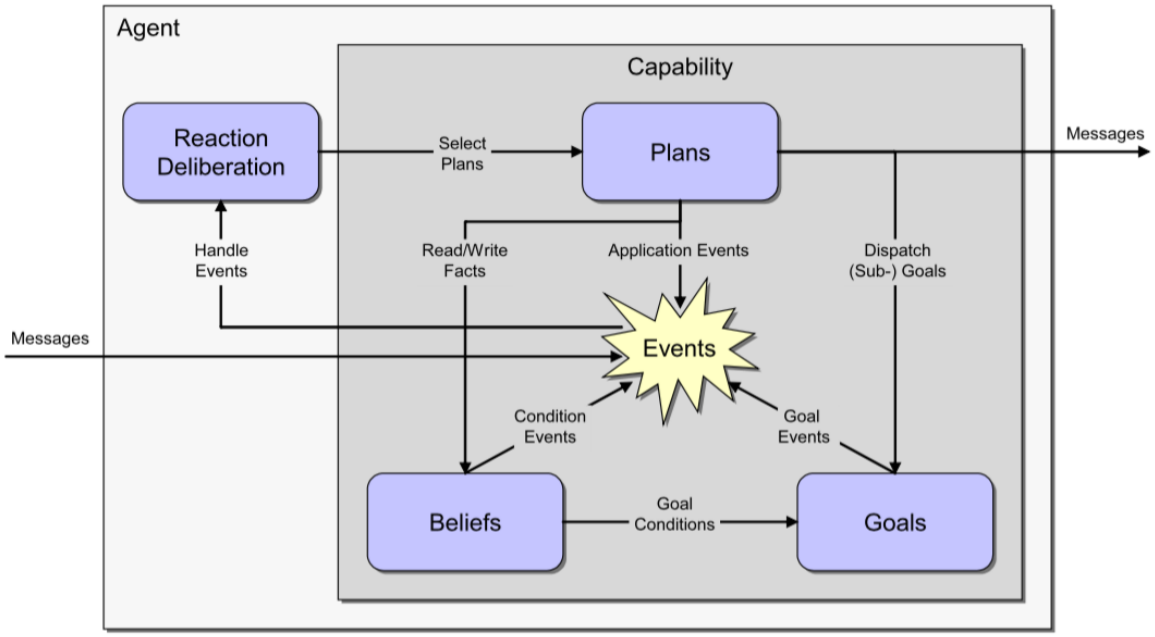
A maioria dos problemas em agentes exige as seguintes competências de uma plataforma para desenvolver aplicações multiagentes: abertura, mediator, e raciocínio. Abertura significa a interconexão de aplicações não relacionadas; mediator enfatiza nos serviços de manutenção, segurança e persistência. E raciocínio visa o processo de decisão interno do agente e principalmente tenta mapear esse processo da arquitetura natural como insetos e humanos (POKAHR, BRAUBACH, LAMERSDORF, 2005).

Além de ser compatível com os objetivos clássicos de mediator e raciocínio BDI, o desenvolvimento de Jadex é guiado por dois fatores. Por um lado, o desenvolvimento da engine de raciocínio recebe várias melhorias na arquitetura BDI de modo em geral, como por exemplo, apresentar uma representação explícita dos objetivos e uma forma de integrar os mecanismos de determinar os objetivos. Por outro lado, o sistema respeita o estado da arte atual em relação a orientação a objetivos em engenharia de software, a fim de atrair não somente especialistas em inteligência Artificial, mas também desenvolvedores de software habilidosos. Por causa disso, o

desenvolvimento de agentes é baseado em Java e XML, e apresenta suporte a aspectos de engenharia de software, como módulos reusáveis e ferramentas de desenvolvimento.

Analisando a arquitetura de Jadex, um agente é uma caixa preta que recebe e envia mensagens. Todos os tipos de eventos, mensagens recebidas ou eventos objetivo serve como entrada para a reação interna e o mecanismo deliberativo, este mecanismo dispara eventos para os planos selecionados da biblioteca de planos. Em Jadex, a reação e o mecanismo deliberativo são os únicos componentes globais de um agente. Os outros componentes estão agrupados em módulos reusáveis chamados de capacidades.

Figura 4 - Arquitetura abstrata Jadex



Fonte: (POKAHR; BRAUBACH; LAMERSDORF, 2005).

As crenças em Jadex apresentam representações em orientação a objetos, onde objetos arbitrários podem ser armazenados como fatos (crenças) ou em conjunto de fatos (conjunto de crenças) (POKAHR; BRAUBACH; LAMERSDORF, 2005).

Para qualquer objetivo (desejo) que um agente possui, o agente irá iniciar as ações necessárias até considerar que o objetivo foi alcançado, é inalcançável, ou não deseja mais

completa-lo. Em Jadex, os objetivos são representados como objetos explícitos e são armazenado em uma base de objetivos. Essas bases são acessíveis para o componente de raciocínio assim como para os planos se eles precisarem saber ou querem alterar o estado atual dos objetivos do agente. Como os objetivos são representados separadamente de outros planos, o sistema pode manter objetivos que não associados a nenhum plano. Por causa disso, diferentemente de outros sistemas CDI, Jadex não exige que todos os objetivos adotados estejam consistentes entre si, desde que somente um subconjunto consistente de objetivos sejam seguidos ao mesmo tempo. (POKAHR; BRAUBACH; LAMERSDORF, 2005).

Planos representam o comportamento de um agente e são compostos de cabeça e corpo. A cabeça do plano especifica as circunstância na qual o plano pode ser selecionado, por exemplo, qual evento ou objetivo gerenciado pelo plano e as condições para a execução do plano. Já o corpo do plano fornece um curso pré definido de ações, dado uma linguagem procedural. Esse curso de ação é para ser executado pelo agente quando o plano selecionar a execução, e pode conter ações fornecidas pelo sistema API, como enviar mensagens, manipular crenças, criar sub-objetivos. (POKAHR, BRAUBACH, LAMERSDORF, 2005)

A implementação em Jadex consiste em dois componente: um arquivo de definição do agente para a especificação das crenças, objetivos e planos, além dos valores iniciais. Esse arquivo é criado em XML e segue o modelo Jadex CDI. A parte dos planos (corpo dos planos) são realizados na linguagem de programação Java e tem acesso as CDI do agente através de uma API. (POKAHR, BRAUBACH, LAMERSDORF, 2005)

4.3 MOISE+

Um tópico importante em sistemas multiagentes é a coordenação e controle de agentes autônomos a fim de garantir um resultado em nível macro. Assim como falado na abordagem baseada em sociedade, os conceitos de sociedade (normas, políticas, organização) tentam resolver este problema.

Várias pesquisas em sistemas multiagente propuseram linguagens para modelar organizações, permitindo especificar aspectos de uma organização de tal modo que pode ser usado por agentes. Como (HUBNER, 2009) descreve: permitir os agentes ficarem "organização-ciente".

MOISE+ é um modelo organizacional para sistema multiagentes baseado nas definições de papéis, grupos e missões. Essas especificações pode ser usado tanto para agentes gerenciar e usar a organização, como para uma organização obrigar que os agentes sigam as especificações (HUBNER, SICHMAN, BOISSIER, 2007).

O MOISE+ propõe uma linguagem de modelagem organizacional que explicitamente divide as especificações da organização em três dimensões: estrutural, funcional, e deôntica (HUBNER, SICHMAN, BOISSIER, 2007).

A dimensão estrutural especifica os papéis, grupos e conexões de uma organização. Neste modelo um objetivo global é realizado enquanto os agente precisam seguir as obrigações e permissões que estão em seus papéis, ou seja, um agente precisa desempenhar um papel para fazer parte do grupo.

A dimensão funcional específica como o objetivo global coletivo deve ser atingido, por exemplo, como os objetivos estão divididos (em planos globais), as políticas para alocar tarefas, a coordenação dos planos de execução e a qualidade (tempo consumido, uso de recursos) de um plano.

As duas dimensões anteriores são tratadas independentemente, por isso, a dimensão deôntica serve para vincular a dimensão estrutural com a dimensão funcional ao definir papéis (permissão e obrigação) para as missões. Com essa separação, o SMA pode trocar sua estrutura (dimensão estrutural) sem alterar seu funcionamento (dimensão funcional) e vice versa.

A Especificação Estrutural (EE) do MOISE+ é construída em três níveis (HUBNER, SICHMAN, BOISSIER, 2007):

1. Nível individual: a conduta que um agente deve seguir quando recebe um papel.
2. Nível social: conecta os papéis na categoria de conhecimento, comunicação e autoridade.
3. Nível coletivo: o agrupamento dos papéis em grupos.

Fazendo uma analogia com um time de futebol, no nível individual o time de futebol define os papéis como goleiro, zagueiro, capitão, atacante, treinador, etc., e as heranças das relações entre

eles. Um agente pode ter dois ou mais papéis somente se eles são compatíveis. Por exemplo, um capitão goleiro.

No nível coletivo, os jogadores são divididos em dois grupos (defensor e atacante) e estes grupos são subgrupos do grupo time.

E no nível social, os papéis são conectados. Cada conexão tem uma fonte e um alvo, por exemplo, o treinador (que é fonte) tem uma conexão de autoridade sobre todos os jogadores (alvo). O mesmo ocorre nas conexões de conhecimento e de comunicação.

A Especificação Funcional (EF) é composta de um conjunto de esquemas que representam como um SMA normalmente atinge seu objetivo global (organizacional) ao definir como esses objetivos são divididos (planos) e distribuídos entre os agentes (missões).

O esquema pode ser representado por uma árvore onde a raiz é o objetivo global e as folhas são os objetivos que podem ser atingidos pelos agentes. Essa decomposição pode tanto ser estabelecida pelos projetistas do SMA ou pelos próprios agentes. O esquema é formado por missões que representam um conjunto coerente de objetivos que um agente pode atingir. Por exemplo, continuando o exemplo de futebol, um agente poderia estar comprometido a seguir o seguinte conjunto de missões: "ficar na área de gol do adversário", "chutar para o gol do adversário" e o objetivo comum "marcar um gol".

E a dimensão deôntica estabelece o grau de autonomia dos agentes ao deixar explícito o que é permitido e o que é obrigatório na organização. Uma *permissão*(p, m) declara o que o agente desempenhando o papel p é autorizado em comprometer-se para a missão m. Além disso, uma *obrigação*(p, m) declara o que o agente fazendo p deveria se comprometer com m.

S-MOISE+ é uma implementação de código aberto de um mediator organizacional, utilizando o modelo MOISE+ como fundamentação. Esse mediator é uma interface entre o agente e o sistema de níveis, fornecendo acesso para o nível de comunicação, para as informações sobre o estado atual da organização (criar grupo, esquemas, atribuir papéis, etc.), e permite os agentes alterar a organização e suas especificações (com restrições, o agente precisa respeitar as especificações da organização). (HUBNER et al, 2009), (HUBNER, SICHMAN, BOISSIER, 2007).

4.4 JaCaMo

JaCaMo é um framework para programação de sistemas multiagentes combinando três tecnologias: *Jason,* para programação de agentes autônomos; *Cartago*, para programação de artefatos de ambiente; e *MOISE+* para programação de organizações multiagentes (JaCaMo, 2015).

Jason é uma plataforma de desenvolvimento de sistemas multiagentes que incorpora a programação orientada a agentes. Utiliza a linguagem baseada em lógica e arquitetura BDI chamada AgentSpeak. Essa linguagem foi criado Rao (RAO, 1996), mas depois foi estendido por vários autores. Jason possui a versão extendida do AgentSpeak (JaCaMo, 2015).

Por padrão, Jason implementa as comunicações numa linguagem baseada em KQML. No entanto, Jason também provê distribuição e comunicação usando Jade, baseado em FIPA-ACL (JaCaMo, 2015).

CarTaGo é um framework e infraestrutura para programação de ambientes e execução em sistemas multiagentes. Cartago se baseia na meta modelagem de Agentes & Artefatos (A&A). Essa modelagem introduz metáforas de alto nível identificadas em ambientes de trabalho humano de forma cooperativa: agentes como entidades computacionais que realiza alguma tarefa relacionadas com seu objetivo; e artefatos como um recursos e ferramentas construídas, usadas e manipuladas dinamicamente por agentes para realizar suas tarefas (JaCaMo, 2015).

E Moise+ é um modelo organizacional para sistemas multiagentes baseado na noção de papéis, grupos e missões. Moise+ estabelece uma explícita especificação para a organização que pode ser usado por agentes para inferir sobre sua organização ou pela própria plataforma organizacional para forçar os agentes a seguir a especificação (JaCaMo, 2015).

5. COOPERAÇÃO VIA PLANOS COMPARTILHADOS

Analisando as várias abordagens de cooperação estudados, foi desenvolvido uma abordagem baseada em planos compartilhados utilizando como base o framework *Generalized Partial Global Planning* (Planejamento Global Parcial generalizado, ou sigla GPGP) apresentado no capítulo 3.6.3.

Primeiramente, na seção 5.1 será exposto o problema escolhido e o cenário criado a partir dele. Em seguida será explicado como cada abordagem descrita no capítulo 3.6 se relaciona com este problema e porque foi escolhido planos compartilhados.

Na seção 5.2 erá descrito com maiores detalhes o framework Generalized PGP utilizado como base para o algoritmo criado e descrito na seção 5.3.1, um algoritmo específico para solucionar o problema de trânsito.

Na seção 5.3.2 será explicado com detalhes o funcionamento do sistema, suas classes e principalmente o agente, implementados com a ferramenta JaCaMo para criação de SMA.

5.1 ESTUDO DE CASO

Foram estudados vários problemas em que a cooperação entre agentes poderia melhorar o resultado final. O caso clássico como o problema dos mineradores (compartilhar com os outros mineradores se achou ouro ou não) foi um dos primeiros problemas analisados. No entanto, com o intuito de encontrar problemas que a humanidade podem entrar no futuro, foi analisado os vários problemas que uma cidade inteligente teria: compartilhamento de energia, veículos, escritório, casa, etc. E em um futuro mais próximo, a cooperação entre múltiplos dispositivos conectado a internet, normalmente conhecido como internet das coisas. Outro assunto que vem ganhando destaque nos últimos anos é o desenvolvimento de carros autônomos, e nos últimos meses quase toda empresa automobilística anunciou que está desenvolvendo algum tipo de carro inteligente.

Analisando o problema de tráfego, existem várias situações em que carros autônomos poderiam cooperar entre si. Dentro deste contexto, foi escolhido para estudo um problema comum

no tráfego de carro: como carros poderiam cooperar entre si em cruzamentos de modo que não haja necessidade de semáforos?

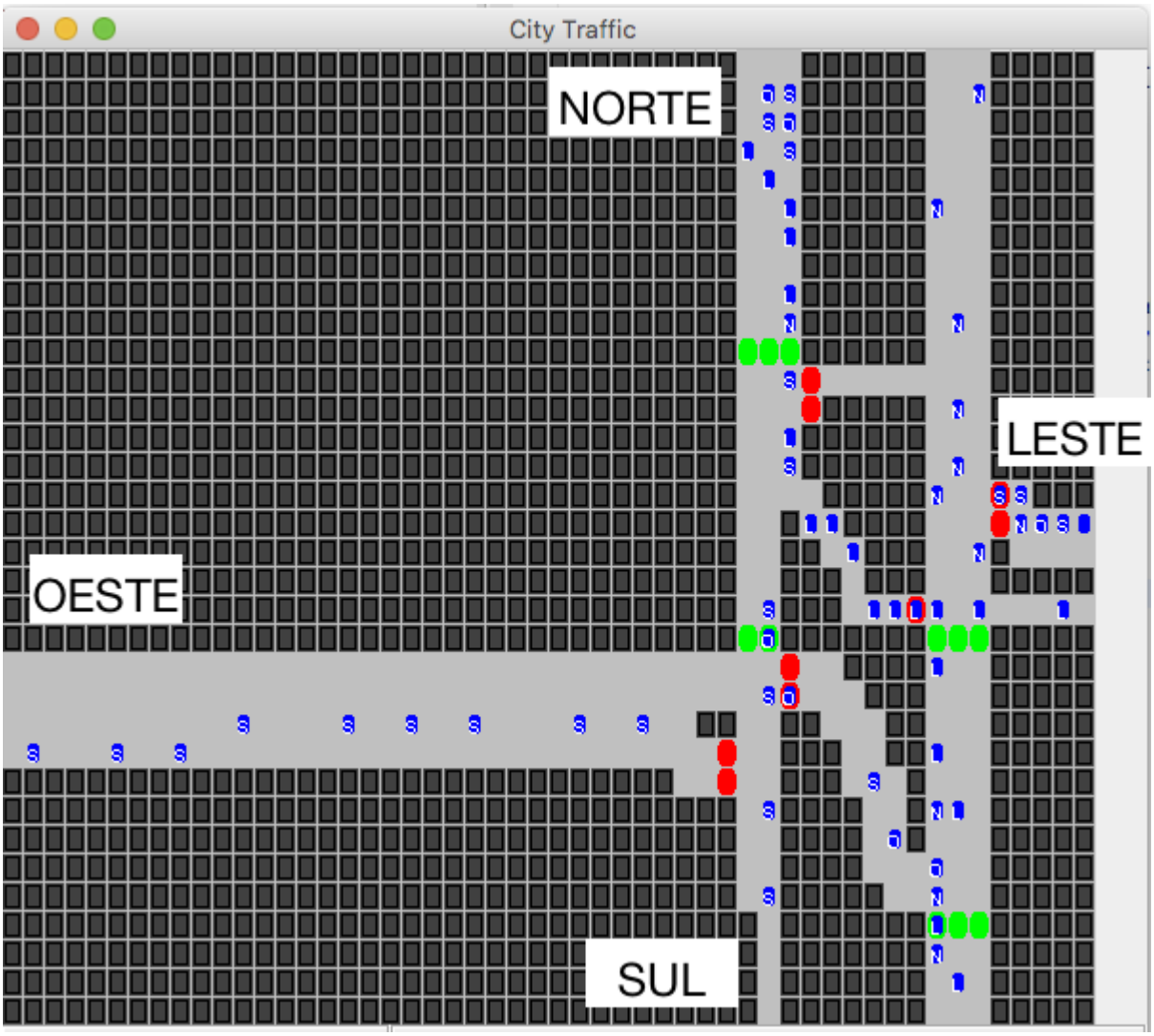
A partir deste problema, foi criado o ambiente para testar o modelo de cooperação proposto, utilizando como base o cruzamento existente ao redor da UFSC. A Beiramar está ao Norte, o Córrego Grande ao Leste, a UFSC a Oeste, e o pantanal ao Sul.

Figura 5 - Mapa

Fonte: (GOOGLE MAPS, 2016).

E a partir do mapa acima, foi criado o ambiente abaixo na ferramenta JaCaMo.

Figura 6 - Ambiente



Fonte: Autor (2016).

Ambiente criado usando o JaCaMo no qual os agentes foram aplicados. As cores vermelho e verde são o semáforo no seu estado atual. A cor azul representa o agente, a letra em cima do agente informa qual para onde o agente está indo (a primeira letra de cada ponto cardeal)

No cenário proposto, cada agente pertence a uma das doze rotas abaixo:

- Do Norte para Sul
- Do Norte para Oeste
- Do Norte para Leste
- Do Norte para Norte
- Do Oeste para Sul
- Do Leste para Norte
- Do Leste para Sul
- Do Leste para Oeste
- Do Sul para Norte
- Do Sul para Sul
- Do Sul para Leste
- Do Sul para Oeste

Cada uma dessas 12 rotas possui uma quantidade total de agentes variável, assim como a frequência com que estes agentes aparecem. Essas informações são descritas na Seção 5.4 (*Testes*).

O agente tem a capacidade de observar o ambiente e perceber que o semáforo está amarelo ou vermelho. Nessa situação o agente sabe que não pode avançar, e fica parado, observando o ambiente até o semáforo ficar verde.

5.1.1 Análise sobre as Abordagens

Com o problema de tráfego descrito acima, foi analisado como cada abordagem discutida no capítulo 3.6 poderia ser utilizada para auxiliar na solução do problema.

Inicialmente, parece ser um problema de alocação de tarefas. Um agente carro tem como objetivo sair do ponto A e ir até o ponto B. Para cumprir esse objetivo, ele irá criar uma rota entre estes dois pontos. Se visualizarmos o mapa como um plano cartesiano, essa rota pode ser dividida como uma sequência de movimentos (tarefas) que o agente precisa realizar até concluir o objetivo.

Usando a taxonomia de Gerkey e Matarić descrita no capítulo 3.6.2 Alocação de tarefas. Este problema poderia ser considerado um problema TU-RU-AT (Tarefa-Única, Robô-Único, Atribuição Prolongada de Tempo).

No entanto, o cenário de tráfego possui múltiplos agentes e cada um deles está criando sua própria rota para chegar em seu destino. Como somente um agente pode cumprir a tarefa de se mover para o ponto (x,y) no mesmo instante de tempo t, o real problema neste cenário é como os agentes podem cooperar entre si para evitar colisões.

Dado este problema, a abordagem baseada em planos compartilhados se adequa como uma tentativa de solução. Afinal, um agente planejador está pensando nas tarefas futuras e como elas são afetadas.

O conceito de Sociedade também poderia aparecer num problema de tráfego, no entanto, neste trabalho este conceito ficou de forma implícita. Por exemplo, as regras de trânsito como andar pela direita, não andar na contramão, parar nos semáforos são normas conhecidas por todos os agentes previamente.

Neste trabalho, quando uma rota é calculada, já se sabe se qual é a direção da rua, ou seja, o agente não precisa se questionar se está na contramão (mais detalhes na seção 5.3.2 - implementação). Além disso, o próprio algoritmo utilizado pelos agentes para compartilhar seus planos poderia ser considerado um norma da sociedade, ou seja, uma política interna que todos os agentes compartilham.

A abordagem baseada em aprendizagem também foi considerada. Analisando as propostas do Google ou do Tesla Motors para carros autônomos, conceitos de aprendizagem são utilizados principalmente para reconhecimento de padrão (descobrir se o ambiente mudou comparado com a última coleta de dados), encontrar rotas alternativas (em caso de acidentes, obras ou desvios em geral), direção defensiva (uma bola invade a pista, qual a probabilidade de uma criança vir atrás). Este tipo de proposta pode se adequar mais a trabalhos futuros.

5.2 ESPECIFICAÇÃO DO MODELO GENERALIZED PARTIAL GLOBAL PLANNING

Este trabalho utilizou o framework Generalized Partial Global Planning (GPGP) como base para o desenvolvimento de um modelo de cooperação adaptado para trânsito. Por isso, nesta seção, serão descritos as principais características desse modelo.

O GPGP foi desenvolvido como um framework sem domínio definido para coordenação em tempo real de agentes que precisam trabalhar juntos para cumprir objetivos difíceis.

Um dos maiores problemas em coordenação é a existência de incerteza nas tarefas que não são locais, ou seja, o cronograma e execução de tarefas de um agente pode impactar a realização de outra tarefa realizada por outro agente.

Para que a coordenação seja eficiente, decisões precisam ser feitas sobre qual sub-objetivo um agente deve cumprir, quando ele irá cumprir esses objetivos e quanto esforço ele gastará em cada objetivo.

A ideia geral do GPGP é construir um módulo coordenado para fornecer informação para um agendador local, removendo a incerteza na execução da tarefa e com isso, construindo planos melhores. E ao realizar isso para cada agente, irá melhorar a eficiência global na solução dos objetivos dos agentes.

No GPGP cada agente possui um banco de dados para armazenar as crenças sobre as tarefas atuais. Armazenando informações das tarefas como método, interdependências e parâmetros que afetam duração, prazos e qualidade. GPGP utiliza TAEMS (Task Analysis, Environment Modeling, and Simulation) como linguagem de modelagem para descrever as estruturas das tarefas do agente.

Quando os agentes se encontram, eles podem iniciar um diálogo, comparando suas tarefas ou buscando interação entre suas atividades. Desse modo cada agente pode construir uma visão parcial do plano global.

Caso esses agentes encontrem uma relação entre suas tarefas, por exemplo, a tarefa do agente A precisa ser executada antes do agente B. Nesse caso, agente A pode reagendar sua tarefa usando seu agendador interno e oferece ao agente B um comprometimento em cumprir essa tarefa em um determinado tempo t. O agente B reagenda sua própria tarefa para depois do tempo t, mas antes do prazo final da tarefa.

Em essência o GPGP segue o seguinte algoritmo:

1. O agente, utiliza seu planejador interno para dividir seu objetivo em várias tarefas.

2. Ao entrar em contato com outros agentes, eles iniciam diálogo, fazendo com que o agente obtenha uma visão parcial do plano global.

3. Esses planos não coordenados (com incerteza) viram a entrada do módulo de coordenação do GPGP.

4. Esse módulo aplica algum método de coordenação e no final repassa para o agendador interno do agente que otimiza os agendamentos.

O papel do mecanismo de coordenação é fornecer informação aos agendadores de cada agente de modo que eles possam fazer o melhor agendamento possível. Dependendo do mecanismo utilizado, o tipo de informação transmitido aos outros agentes varia. Por exemplo, um mecanismo que informa redundância entre as tarefas. Um outro mecanismo que informa a relação entre tarefas, mostrando se uma tarefa precisa ser executada antes de outra (coordenação difícil) ou se irá auxiliar ao executada antes da outra tarefa (coordenação fácil). Ou mesmo, um terceiro mecanismo que informa sobre as tarefas já realizadas pelo agente.

5.3 IMPLEMENTAÇÃO DO MODELO GPGP PARA TRÂNSITO

Nessa seção será explicado o modelo desenvolvido para a solução do problema de trânsito descrito na seção 5.1. Esse modelo foi baseado no modelo GPGP, no entanto, somente algumas características foram utilizadas desse modelo, ficando de fora até algumas características essenciais para considerar este um caso de GPGP, como por exemplo, a utilização de TAEMS como linguagem de modelagem para as estrutura de tarefas dos agentes. Implementar o modelo GPGP completamente iria aumentar significamente o tempo necessário para realizar este trabalho, assim como o aumento na complexidade do sistema (no capítulo 6.1, é sugerido como trabalho futuro a generalização desse modelo).

O GPGP foi escolhido como base desta implementação principalmente pelo fato de não possuir um domínio definido. Deste modo, foi possível adequar o algoritmo para solucionar o problema de trânsito proposto.

Em seguida, será descrita a implementação desse modelo realizada no JaCaMo, apresentando suas classes, funcionamento e problemas encontrados.

5.3.1 Algoritmo adaptado ao Trânsito

A abordagem com planos desenvolvida neste trabalho é descentralizada, ou seja, todos os agentes possuem um plano global e ao mesmo tempo seu estado mental particular. A abordagem da sociedade (SMA) pode ser resumida pelo seguinte algoritmo (baseado no funcionamento do GPGP):

1. Agente entra no sistema
2. Ele cria uma rota até seu destino final.
3. Ele tenta adequar sua rota ao plano global, criando um plano privado coordenado.
   a. Caso haja conflito no ponto x, y no instante de tempo t, espera um instante de tempo na posição anterior ao conflito.
   b. Caso haja um segundo conflito na mesma posição, altere o plano do agente que iria causar o segundo conflito.
4. Notifique os outros agentes sobre o plano global foi atualizado.
5. Cada agente atualiza seu plano privado caso haja alteração.

A seguir cada uma das etapas do algoritmo é descrita com mais detalhes:

1. No modelo GPGP, quando os agentes se encontram, ele pode iniciar um diálogo e compartilhar informações. No estudo de caso proposto, temos carros autônomos capazes de cooperar entre si, no entanto, um carro não precisa comunicar com todos os carros da cidade se o problema é somente no cruzamento à frente. Então, considera-se que quando o agente entra no sistema, o agente está iniciando um diálogo com os outros agentes na mesma região, ou seja, os agentes que podem afetar suas tarefas e vice versa. E nesse diálogo o novo agente recebe o plano global dos agentes presentes.
2. Do mesmo modo que o modelo GPGP, o agente cria uma sequência de tarefas internamente de modo que solucione seu problema que é sair do ponto x, y e ir até x1, y1. Essa rota é um conjunto de movimentos que o agente deve fazer, mas ainda não

possui nenhuma forma de cooperação. Para encontrar a menor distância entre esse dois pontos, o algoritmo Dijkstra encontra o caminho mais curto.

3. Com a sequência de tarefas e o conhecimento sobre o plano global adquirido ao entrar no sistema, o agente analisa possíveis conflitos entre suas tarefas e as tarefas dos outros agentes. O plano global é estruturado como um mapeamento em que um ponto x, y e instante de tempo t indicam se aquela posição e naquele instante está ocupado, e caso esteja ocupado, qual é o nome do agente.

   a. Com esse mapeamento, é possível descobrir se haverá um conflito. No primeiro conflito o agente espera um instante de tempo na posição anterior ao conflito.

   b. Caso ocorra um segundo conflito no instante t+1, ou seja, no próximo instante, o agente pode utilizar aquela posição no instante t+1. E altera o plano do agente que teve conflito para esperar o instante t+1 na posição anterior. E para que o agente que teve seu plano alterado saiba que seu plano foi alterado, a versão do plano é incrementada.

   c. No final, o agente irá criar um plano privado para si, esse plano é a rota criada anteriormente só que com ações coordenadas. O plano privado é estruturado como um mapeamento de instante t para a posição que o agente precisa estar.

4. E seguindo o modelo GPGP, depois de reagendar suas tarefas, o agente notifica os outros agentes. Quando termina de escalonar suas tarefas (movimentos) no plano global de forma que tenha nenhum conflito, o agente compartilha o plano global modificado com os agentes no sistema.

5. Todos os agentes no sistema recebem a notificação de que o plano global foi alterado. Eles verificam se a versão do seu plano privado é diferente com a versão do plano global modificado. Caso seja, o agente atualiza seu plano.

O plano global não possui uma entidade centralizadora, ou seja, cada agente possui sua visão deste plano. De certa forma, o plano global será igual para todos os agente no ambiente, pois quando um agente altera o plano, ele o compartilha com os outros agentes.

Além disso, o plano global informado no algoritmo é representado por duas estruturas de dados. A primeira é um conjunto com todos os planos privados. Um plano privado é representado por uma tabela hash - a chave é o instante de tempo t e o valor é o ponto x,y. Por exemplo, o agente A tem um plano privado que informa que no instante de tempo 13, ele deve se mover para o ponto (36, 15) e no tempo 14, ele deve se mover para para o ponto (36, 16).

A segunda estrutura de dados do plano global é uma tabela hash onde a chave é o instante de tempo t e o ponto (x,y); e o valor é o agente localizado nessa posição e nesse instante de tempo. Utilizando o exemplo acima, no instante de tempo 13 e ponto (36, 15), tem-se o agente A. Uma outra forma de armazenar essa informação seria uma matriz de três dimensões, mas manipular uma matriz como estrutura de dados seria mais difícil alterar os dados no caso de replanejamento.

Para que este algoritmo seja executado corretamente é essencial a sincronização dos agentes, ou seja, é necessário que todos os agentes estejam seguindo um mesmo relógio. Por ser um problema em tempo real, não existe margem para a realização da tarefa (movimento), caso um agente se atrase ou se adiante, todo o plano global seria afetado, principalmente com centenas de agentes. Por causa disso, é essencial a existência de um relógio global que todos os agentes obedeçam.

Outro detalhe importante deste algoritmo é que quando o agente A está coordenando seus planos, caso haja um segundo conflito, o agente A irá alterar o plano do agente B, no entanto, o agente A desconhece se o agente B também teve conflito e também estava esperando um instante de tempo, por causa disso, pode ocorrer do agente B esperar um segundo instante de tempo. Ou seja, não há garantia que um agente irá esperar somente um instante de tempo.

5.3.2 Implementação no JaCaMo

Para a implementação do modelo acima foi utilizado o framework JaCaMo, descrito no capítulo 4.4. Foi utilizado o módulo de Jason para o desenvolvimento dos agentes (driver.asl para o cenário com semáforos e cooperative-driver.asl para o cenário com planos compartilhados). Foi utilizado também a parte de ambiente do Jason para mostrar os agentes numa interface gráfica.

O Cartago foi utilizado para desenvolvimento do ambiente, e a interação dos agentes com este ambiente, através da utilização de artefato.

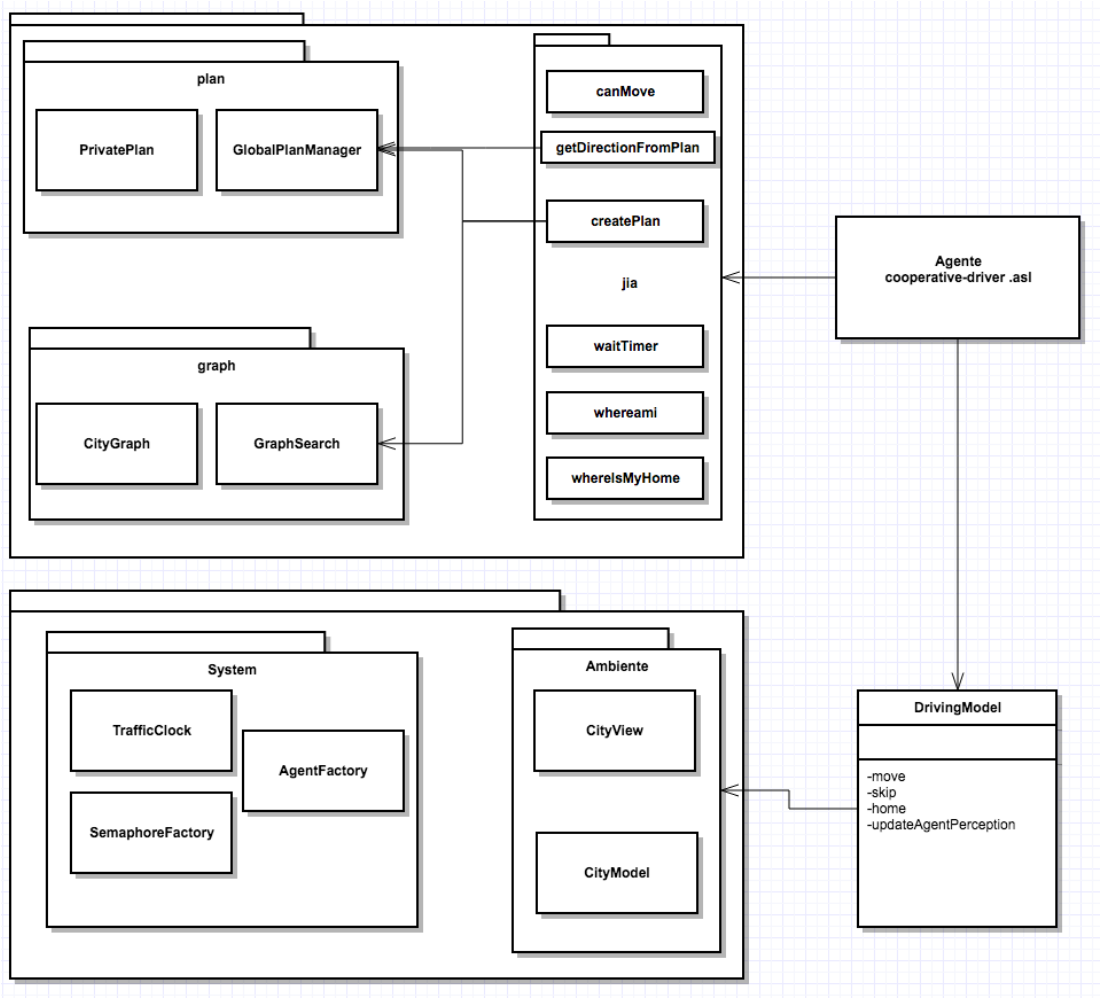
O componente Moise+ não foi utilizado para as definições da sociedade, por causa disso, as normas e papéis foram implementadas de forma implícita.

Foi implementado o Algoritmo (Seção 5.3.1) adaptado a Trânsito, criando agentes capazes de compartilhar seus planos privados, criando um plano parcial global no processo. Além disso, para poder comparar a eficiência da abordagem com planos compartilhados, foram desenvolvidos também agentes que não possuem a habilidade de cooperação, e para que eles consigam passar pelo cruzamento foi implementado o conceito de semáforos, ou seja, seria a situação atual do trânsito.

5.3.2.1 Visão Geral da Implementação

Na figura abaixo, as principais classes do sistema são destacadas. Em seguida será explicado o funcionamento delas.

Figura 7 - Classes



Fonte: Autor (2016).

Todas as classes do diagrama acima foram desenvolvidas em Java com exceção do cooperative-driver.asl que foi desenvolvido em uma versão estendida de AgentSpeak.

As classes na parte superior da Figura fazem parte do agente, enquanto que as classes de baixo fazem parte do ambiente ou auxiliares ao sistema.

O Agente consegue realizar ações internas, elas estão localizada na pasta *jia,* como por exemplo, canMove, getDirectionFromPlan, createPlan, etc. Essas ações por sua vez, aproveitam das classes no pacote de grafos (graph) ou no pacote de planos (plan) para encontrar o menor caminho entre dois pontos ou armazenar e manipular seus planos respectivamente.

O agente consegue interagir com o ambiente através do artefato DrivingModel. E realizar operações *move*, *skip*, *home*. Essas operações também atualizam a percepção do agente sobre seu redor.

5.3.2.2 Artefato, sistema, e outras classes

O sistema é iniciado pelo JaCaMo a partir do arquivo de propriedade nomeado *traffic.jcm*.

```
10
11 mas traffic {
12
13     agent ricardo : cooperative-driver.asl {
14         focus : city.floripa1
15         instances : 100
16     }
17
18     workspace city {
19         artifact floripa1 : traffic.DrivingModel()
20     }
21
```

Neste arquivo é definido o nome do agente (ricardo), qual é o tipo de agente (cooperative-driver.asl), qual o workspace e artefato que ele está localizado (city.floripa1) e quantos agentes terá no sistema (100 agentes, os agentes são nomeados ricardo1 até ricardo100).

O workspace city.floripa1 é definido como a classe Java traffic.DrivingModel(). DrivingModel é um artefato, no conceito de CarTaGo, artefatos são recursos e ferramentas construídos, usados e manipulados por agentes a fim de auxiliar a realização de suas atividades individuais ou coletivas.

Nesta implementação, o artefato DrivingModel cria o ambiente representado pelo CityModel (cria um cenário como uma grade, definindo a posição das ruas e dos semáforos) e apresenta esse ambiente através de uma interface gráfica gerado pelo CityView. DrivingModel também fornece métodos para que o agente consiga interagir com o ambiente, ou seja, ele faz a conexão dos agentes com o ambiente.

DrivingModel também inicia três classes fundamentais no projeto: TrafficClock, AgentFactory, SemaphoreFactory.

TrafficClock: Todo o sistema é sincronizado por essa classe. Essa classe garante que todos os agentes executarão uma e somente uma ação antes de mudar para o próximo instante de tempo. Na seção de sincronização (Seção 5.3.2.4) será aprofundado sobre essa classe e a necessidade dela.

AgentFactory: Essa classe coloca o agente dentro no ambiente dentro do tempo estipulado. Cada um dos 12 caminhos possíveis (por exemplo, de norte-a-sul ou leste-a-oeste) possui uma quantidade total de agentes e a frequência com que estes agentes devem aparecer, por exemplo, o caminho Do Sul pro Norte possui 30 agentes e eles entram no sistema a cada 3 instantes de tempo do TrafficClock.

SemaphoreFactory: Nessa classe os semáforos são criados. Os semáforos são usados somente no caso dos agentes que não utilizam planos compartilhados. Os semáforos ficam alternando seus estados (vermelho, verde, amarelo). A frequência para trocarem seus estados foi criado baseado em observação e obedece a tabela abaixo.

Tabela 6 - Semáforos

Direção dos semáforos										
De norte a Sul 1										
De Norte a sul 2										
Sul para o Norte 1										
Sul para norte 2										
Leste para Oeste 1										
Leste para Oeste 2										
Oeste para Leste 1										
Oeste para Leste 2										
Oeste para Leste 3										

Fonte: Autor (2016).

A contagem dos semáforos começa de cima para baixo olhando no mapa. Cada célula representa um instante de tempo t. Ou seja, em 10 t os semáforos voltam para o estado inicial.

5.3.2.3 O Agente

Nesse tópico será explicado o funcionamento do agente cooperativo (cooperative-driver.asl) implementado na versão extendida do AgentSpeak (existem várias similaridades com prolog).

Quando o sistema começa, o agente também começa a operar.

Já de início, ele precisa adicionar três desejos: saber onde é sua casa (*!where_is_home*), onde está (*!where_is_my_position*) e indo para casa (*!going_home*).

```
15
16  +!start
17      <-  !where_is_home;
18          !where_is_my_position;
19          !going_home.
20
```

Começando por !where_is_home, o agente tem três possíveis planos:

```
21
22  +!where_is_home : .my_name(N) & jia.whereIsMyHome(N,X,Y)
23      <-  .print(N, " home is at (",X,",", Y,")");
24          +home(X,Y);
25          !where_is_home.
26
27  +!where_is_home : not home(X,Y) & .my_name(N)
28      <-  !where_is_home.
29
30  +!where_is_home.
31
```

O primeiro plano na linha 22, possui a condição *jia.whereIsMyHome(N, X, Y)*. Essa condição é uma operação interna do agente. N representa o nome do agente obtido em *.my_name(N)*. X e Y são variáveis que representa um dos possíveis destinos finais.

Caso o agente já esteja dentro do sistema (a classe AgentFactory.java já criou este agente), a condição *jia.whereIsMyHome(N, X, Y)* terá sucesso. E o plano da linha 17 irá virar uma intenção, ou seja, o agente está comprometido com este desejo (Uma intenção no Jason é um plano iniciado o qual o agente está executando a fim de atingir um determinado desejo).

Para cumprir com o desejo da linha 22, o agente precisa executar corretamente as ações após <-. A ação *+home(X, Y)* significa adicionar as crenças do agente a posição da sua casa (por exemplo, home(45, 0)), as variáveis X e Y foram definidas na operação interna anterior.

A próxima ação é o desejo *!where_is_home* novamente. O agente precisa se questionar novamente, pois podem existir várias posições possíveis para chegar em casa (por

exemplo, um agente indo pro norte, as posições (44, 0), (45, 0), (46, 0) são destinos finais possíveis. Quando o agente tiver todas as posições possíveis a operação *jia.whereIsMyHome(N, X, Y)* irá falhar. Neste caso, outro plano terá que ser analisado, o plano da linha 27 também será falso, pois o agente possui a crença de *home(X, Y)*. Como o plano da linha 30 não possui nenhuma condição, ele será escolhido, este plano não possui nenhuma ação para ser executada, finalizando o desejo *!where_is_home*.

Caso o agente ainda não faça parte do sistema, a condição *jia.whereIsMyHome(N, X, Y)* irá voltar falso na primeira vez, e nenhuma crença home(X, Y) será adicionado. O plano +*!where_is_home* na linha 27 terá a condição *not home(X, Y)* verdadeira, e este plano irá virar uma intenção. Para executar essa intenção com sucesso, o agente terá que cumprir com o desejo *!where_is_home*, ou seja, o agente irá ficar se questionando onde é sua casa até ele ser adicionado no sistema.

Voltando a intenção *+!start*, a primeira ação !*where_is_home* foi concluída. A próxima ação é *!where_is_my_position*.

```
40
41  +!where_is_my_position : .my_name(N) & jia.whereami(N,X,Y)
42      <-  -+pos(X,Y).
```

Esse desejo só tem um plano capaz de executá-la. A operação interna do agente *jia.whereami(N, X, Y)* o agente questiona sua posição. E a variável X e Y retorna com a posição que o agente se encontra. Com a condição satisfeita, o agente pode atualizar sua posição a partir das informações X,Y.

E a última ação de *+!start* é *!going_home.* Esse desejo também só tem um plano.

```
31
32  +!going_home : home(X,Y) & pos(AgX, AgY) & .my_name(N)
33      <-  jia.createPlan(N, AgX, AgY, X, Y);
34          !update_global_plan;
35          !go_home.
36
```

Esse plano tem a condição que o agente tenha a crença sobre sua casa (*home(X, Y)*) e sobre sua posição *pos(AgX, AgY)*. Como o agente já adquiriu essas crenças nos desejos anteriores, o agente pode começar a execução deste plano. A primeira ação é a operação interna *jia.createPlan(N, AgX, AgY, X, Y)* para o agente criar seu plano, essa operação informa o nome do agente (N), qual é a posição do agente (AgX, AgY) e qual é a posição (X, Y) que leva o agente para casa.

A operação interna *createPlan* funciona da seguinte maneira:

```
26
27      @Override
28      public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws Exception {
29              String name = ((Atom) terms[0]).toString();
30              int iagx = (int) ((NumberTerm) terms[1]).solve();
31              int iagy = (int) ((NumberTerm) terms[2]).solve();
32              int itox = (int) ((NumberTerm) terms[3]).solve();
33              int itoy = (int) ((NumberTerm) terms[4]).solve();
34
35              Location from = new Location(iagx, iagy);
36              Location to = new Location(itox, itoy);
37
38          try {
39              List<Location> route = GraphSearch.findShortestPath(from, to);
40
41              String s = "";
42              for (Location location : route) {
43                  s += " new Location(" + location + "), ";
44              }
45              logger.info(s);
46
47              PrivatePlan plan = GlobalPlanManager.add(name, route);
48
49              PrivatePlanManager.save(name, plan);
50
51              return true;
52
53          } catch (Throwable e) {
54              for (StackTraceElement stackTraceElement : e.getStackTrace()) {
55                  logger.info(stackTraceElement.toString());
56              }
57
58              e.printStackTrace();
59              return false;
60          }
61      }
```

Depois do Jason converter as variáveis criadas em AgentSpeak e representadas em Java como uma Interface denominada como Term. O agente pode criar seu plano. Na linha 39, o agente

utiliza-se de seu GPS interno para encontrar o caminho mais curto entre sua posição atual (variável Location from) e a posição de sua casa (variável Location to). O algoritmo utilizado para realizar esse cálculo foi Dijkstra.

Com a rota criada. O agente precisa realizar a terceira etapa do algoritmo especificado na seção 5.3, para isso, ele tenta adequar sua rota ao plano global na linha 47 *GlobalPlanManager.add(name, route)*. Assume-se que o agente recebeu o plano Global quando entrou no sistema, ou seja, ele já tem o plano dos outros agentes nesse momento.

O escalonamento dos planos segue a regra do algoritmo:

1. Caso haja conflito no ponto x, y no instante de tempo t, espera um instante de tempo na posição anterior ao conflito.
2. Caso haja um segundo conflito na mesma posição, altere o plano do agente que iria causar o segundo conflito.

Depois de escalonar, o agente salva seu novo plano privado em *PrivatePlanManager.save(name, plan);* O nome do agente é utilizado aqui e em outras classes para encontrar as informações do agente nas estrutura de dados em Java, foi desenvolvido dessa maneira para facilitar a manipulação dos dados.

Voltando a intenção *!going_home.* A próxima ação é notificar os outros agentes sobre alteração dos planos em *!update_global_plan*.

```
36
37 +!update_global_plan
38     <-  .print("tell others agents to update global plan");
39         .broadcast(tell,updatedGlobalPlan).
40
```

Esse desejo tem a ação *.broadcast(tell,updatedGlobalPlan)*. Ao realizar essa ação todos os agentes recebem a crença *updatedGlobalPlan*. Na implementação desse agente não se está compartilhando os dados do plano global via Jason, foi analisado a possibilidade de realizar essa transferência, mas a complexidade seria alta. Então os agentes recebem uma notificação e depois consultam estrutura de dados em Java.

```
82
83 +updatedGlobalPlan[source(S)] : .my_name(N) & home(_,_)
84     <- .print(S, " atualizou o plano global. Preciso verificar meu plano.");
85        jia.updateMyPlan(N).
86
```

Quando um agente recebe essa crença, ele precisa realizar a operação interna jia.updateMyPlan(N) para verificar seu plano.

```
23     @Override
24     public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws Exception {
25         try {
26             String name = ((Atom) terms[0]).toString();
27
28             new Object();
29
30             synchronized (TrafficClock.COUNTER) {
31                 PrivatePlan privatePlanOnGlobalPlan = GlobalPlanManager.getPrivatePlan(name);
32
33                 if (privatePlanOnGlobalPlan != null) {
34                     int myPlanVersion = PrivatePlanManager.getMyPlanVersion(name);
35
36                     int newVersion = privatePlanOnGlobalPlan.getVersion();
37                     if (myPlanVersion < newVersion) {
38                         PrivatePlanManager.save(name, privatePlanOnGlobalPlan);
39                         logger.info(name + " updating my plan from version " + myPlanVersion + " to version " + newVersion);
40                     }
41                 }
42             }
43
44             return true;
45         } catch (Throwable e) {
46             e.printStackTrace();
47             return false;
48         }
49     }
50
```

Nessa operação interna, o agente irá, primeiramente, pegar as informações dele que estão no plano global na linha 31. Em seguida, ele irá pegar o pegar a versão do plano privado atual na linha 34. Caso as novas informações do plano global sejam uma nova versão, o agente atualiza seu plano privado com essas novas informações.

O plano global que um agente possui, tem também uma cópia do plano privado de cada agente. Então, quando o agente altera o plano de outro agente, ele incrementa a versão do plano privado daquele agente. Quando o agente receber um novo plano global, ele só precisa verificar a versão do seu plano no plano global para descobrir se precisa atualizar.

Voltando ao intenção *!going_home*. A última ação necessária é o desejo *!go_home*.

```
43
44 +!go_home : home(X,Y) & pos(X,Y) & .my_name(N)
45     <- .print("I'm home");
46        home(N);
47        -home(X,Y);
48        -pos(X,Y).
49
50 +!go_home : pos(AgX, AgY) & .my_name(N)
51     <-  !wait_timer;
52        jia.getDirectionFromPlan(N, AgX, AgY, DirX, DirY);
53        .print("Estou na posicao (",AgX,",",AgY,"). Devo me movimentar para (",DirX,", ",DirY,")");
54        !wait_car(DirX, DirY);
55        !move(DirX, DirY);
56        !go_home.
57
```

Existem dois planos possíveis, sendo que o primeiro ocorre quando a posição do agente é igual sua crença de *home*, e o segundo quando o agentes ainda não chegou ao seu destino. Caso o primeiro plano seja verdade o agente executa operação no ambiente chamado *home(N)*. Operações no ambiente são realizadas no artefato DrivingModel.java

```
69
70      @OPERATION
71      void home(String name) {
72          try {
73              await_time(TrafficClock.AGENT_WAIT_TIME);
74              TrafficClock.increamentAgentCount(name);
75
76              CityAgent agent = AgentFactory.getAgentByName(name);
77
78              if (agent != null) {
79                  int agentId = agent.getId();
80
81                  model.removeAgent(agentId);
82
83                  synchronized (TrafficClock.COUNTER) {
84                      AgentFactory.removeAgentInTheSystem(name);
85                  }
86
87              }
88          } catch (Exception e) {
89              e.printStackTrace();
90          }
91      }
92
```

Diferente das operações internas executadas anteriormente, uma operação no artefato o agente está se relacionando com o ambiente, seja alterando-o ou simplesmente observado.

Na operação home, o agente está sendo removido do ambiente na linha 81. E removido do sistema na linha 84.

Voltando a intenção atual, *!go_home*. Caso o agente ainda não tenha chegado na sua posição final, o agente terá que seguir o segundo plano. A primeira ação é *!wait_timer*.

```
57
58  +!wait_timer : .my_name(N) & jia.waitTimer(N)
59      <- .print("can run").
60
61  +!wait_timer
62      <- .print("wait..");
63          !wait_timer.
64
```

Esse desejo é um sincronizador. Ou seja, força o agente aguardar o próximo instante do relógio antes de executar novamente. Um agente pode realizar somente uma ação por instante: criar seu plano, mover, ou ficar parado.

A próxima ação de !go_home é *jia.getDirectionFromPlan(N, AgX, AgY, DirX, DirY)*, essa operação interna utiliza o instante de tempo atual para descobrir qual é o próxima posição do agente. Salvando essa posição nas variáveis DirX e DirY.

Em seguida, é a ação *!wait_car(DirX, DirY)*. Esse desejo o agente espera o carro da frente se mexer. Embora o ambiente impeça colisões, foi adicionado essa verificação para adicionar realismo.

A próxima ação é *!move(DirX, DirY)*, esse desejo tem dois planos:

```
72
73 +!move(DirX, DirY) : pos(DirX, DirY) & .my_name(N)
74     <- .print("Ficar parado.");
75        skip(N).
76
77 +!move(DirX, DirY) : pos(AgX, AgY) & .my_name(N)
78     <-  .print("Minha posicao (",AgX,", ",AgY,")");
79         .print("Quero me mover para (",DirX,", ",DirY,")");
80         move(N, DirX, DirY);
81         !where_is_my_position.
82
```

No primeiro plano o agente deve se mover para a mesma posição que já se encontra, ou seja, ficar parado. Nesse caso ele executa a operação no ambiente *skip(N),* essa operação serve somente para o agente perceber o ambiente nesse turno. Perceber o ambiente o agente verifica todas as posições ao redor que ele se encontra. Verifica a existência de agentes, semáforos e qual o estado do semáforo (o cooperative-driver.asl ignora semáforos).

O segundo plano, o agente executa o operação no ambiente *move(N, DirX, DirY)*.

```
50
51      @OPERATION
52      void move(String name, int x, int y) {
53          logger.info(name + " DrivingModel.move()");
54
55          CityAgent agent = AgentFactory.getAgentByName(name);
56          int agentId = agent.getId();
57
58          TrafficClock.increamentAgentCount(name);
59
60          boolean moved = model.move(agentId, x, y);
61          if (!moved) {
62              errors.incrementAndGet();
63              logger.info("errors " + errors);
64          }
65
66          updateAgPercept(agentId);
67
68      }
69
```

Nessa operação, o artefato informa na linha 60 que o agente está se movendo para posição x, y. Como o agente esperou a posição ficar livre antes de se mexer, então o movimento terá sucesso. Depois de se mover, na linha 66 o agente verifica o ambiente ao seu redar.

Voltando a intenção *!move(DirX, DirY)*. Depois de se movimentar, o agente terá o desejo de verificar sua posição novamente em *!where_is_my_position*.

Esse é o funcionamento do agente cooperative-driver.asl. Ele irá ficar se movendo até chegar na posição final. E atualizando seu plano caso alguém o notifique.

O agente driver.asl tem algumas diferenças interessantes. Como ele descobre e obedece semáforos, ele precisa verificar a existência de semáforos e qual é o estado dele antes de se mover.

```
52
53 +!move(DirX, DirY) : pos(AgX, AgY) & semaphoreRed(AgX,AgY) & .my_name(N)
54     <- .print("Sinal Vermelho...");
55         .print("Minha posicao (",AgX,", ",AgY,")");
56         .print("Quero me mover para (",DirX,", ",DirY,")");
57         skip(N);
58         !move(DirX, DirY).
59
60 +!move(DirX, DirY) : pos(AgX, AgY) & semaphoreYellow(AgX,AgY) & .my_name(N)
61     <- .print("Sinal Amarelo...");
62         .print("Quero me mover para (",DirX,", ",DirY,")");
63         skip(N);
64         !move(DirX, DirY).
65
66 +!move(DirX, DirY) : pos(AgX, AgY) & semaphoreGreen(AgX,AgY) & .my_name(N)
67     <- .print("Sinal Verde...");
68         .print("Quero me mover para (",DirX,", ",DirY,")");
69         move(N, DirX, DirY);
70         !where_is_my_position.
71
72 +!move(DirX, DirY) : pos(AgX, AgY) & .my_name(N)
73     <-  .print("Minha posicao (",AgX,", ",AgY,")");
74         .print("Quero me mover para (",DirX,", ",DirY,")");
75         move(N, DirX, DirY);
76         !where_is_my_position.
77
```

Para verificar semáforo, o agente tem quatro planos para o desejo *!move(DirX, DirY)*. Quando este agente realiza uma operação no ambiente, como *move* ou skip. O agente está analisando o ambiente ao seu redor, caso haja um semáforo, o ambiente adiciona uma crença a este agente.

```
77
78 +cell(X,Y,semaphoroRed) <-
79     -semaphoreGreen(X,Y);
80     -semaphoreYellow(X,Y);
81     +semaphoreRed(X,Y).
82 +cell(X,Y,semaphoroGreen) <-
83     -semaphoreYellow(X,Y);
84     -semaphoreRed(X,Y);
85     +semaphoreGreen(X,Y).
86 +cell(X,Y,semaphoroYellow) <-
87     -semaphoreGreen(X,Y);
88     -semaphoreRed(X,Y);
89     +semaphoreYellow(X,Y).
90
```

Com essa informação o agente escolhe o plano apropriado para o estado do semáforo ou o quarto plano de *!move(DirX, DirY)* para uma posição sem semáforo. Pode-se observar que no plano 1, em que o semáforo é vermelho o agente executa a operação *skip* ao invés de se mexer, ou seja, ele só verifica se o estado do semáforo alterou para verde, e somente na próxima execução de *!move(DirX, DirY)*, o plano escolhido pode ser o terceiro plano, com semáforo verde.

5.3.2.4 Sincronização dos agentes

Na descrição do algoritmo adaptado ao Trânsito foi ressaltada a importância da sincronização dos agentes para o funcionamento correto. Esse destaque foi feito pois esse foi um dos maiores problemas encontrados na implementação do algoritmo no JaCaMo.

Para solucionar este problema foi criada a classe TrafficClock. Essa classe possui um contador que representa o relógio global do sistema, esse relógio define o instante de tempo t.

Além disso, essa classe sabe quantos agentes existem dentro do sistema em cada instante t. Com essa informação, o TrafficClock espera todos as entidades do sistema (agentes, semáforos, criadores de agentes) executarem, e depois aumenta o contador em um. E todas essas entidades ficam aguardando o contador aumentar caso já executaram.

Para aguardar o contador aumentar, os semáforos e o criador de agentes utilizam de técnicas de programação concorrente, utilizando semáforos para bloquear a execução de uma thread até que outra thread libere, ou seja, congelando a thread até que o TrafficClock informe que elas podem voltar a executar.

Para os agentes no entanto, não foi possível realizar essa técnica. Essa técnica foi a primeira abordagem utilizada para sincronizar os agentes. A ideia era o raciocínio dos agentes serem congelados até o TrafficClock liberar novamente. O problema é que o funcionamento do JaCaMo impede essa ideia pois as *threads* são reutilizadas entre os agentes, caso uma *thread* seja congelada, não seria usada para outros agentes.

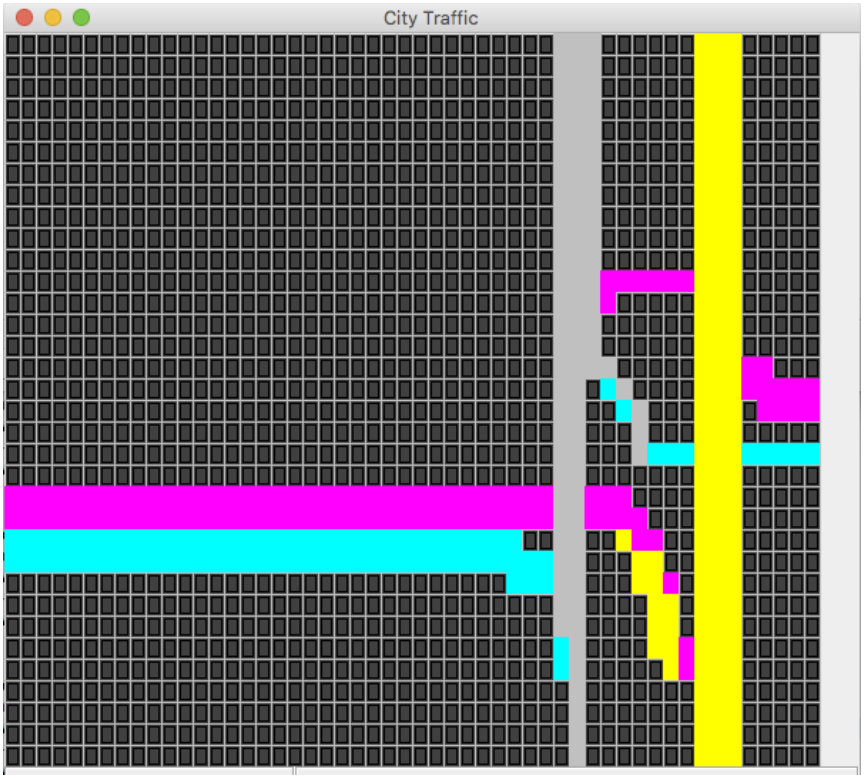
Depois de descobrir essa característica do JaCaMo, foi desenvolvida a abordagem de deixar o raciocínio dos agentes funcionando, mas ao mesmo tempo, bloquear todos os tipos de ações até o TrafficClock libere. Eventualmente o escalonador de thread do JaCaMo passa por todos os agentes e com isso o TrafficClock libera o próximo instante de tempo.

Por isso foi criado o desejo *!wait_timer*. Caso o agente já tenha executado no instante de tempo t, ele ficará em loop até todos os agentes consigam executar.

5.3.2.5 Grafo e movimentação

Um grafo é criado ao mesmo tempo que o ambiente na classe CityModel. Este grafo representa os possíveis caminhos que os agentes podem realizar.

Figura 8 - Printscreen do cenário de trânsito. Cada cor representa uma direção



Cada cor representa uma direção: Amarelo: Norte; Magento: Oeste; Cinza: Sul; Verde-Água: Leste.
Fonte: Autor (2016).

Na imagem acima, a cor amarela representa as estradas indo pro norte. Em magento, a estrada a Oeste. Em cinza, a estrada com direção ao Sul. E em verde-água, a estrada em direção a Leste.

A partir dessas direções é possível criar o grafo que os agentes utilizam para se movimentar. Por exemplo, um ponto amarelo no mapa representa uma estrada indo para norte, então o agente pode se movimentar para norte, mas não pode se movimentar para o sul pois seria contramão. No ponto amarelo também é possível se movimentar para Leste ou Oeste, pois seria o caso de um carro alterar de pista.

O mesmo ocorre com as outras direções, cada ponto-estrada está conectado com o ponto adjacente desde que não seja contramão.

5.4 TESTES

Nesta seção serão apresentados os experimentos realizados para testar a abordagem anterior. Na seção 5.4.1 serão expostos os parâmetros com o que os testes foram executados. E na seção 5.4.2 serão explicado os resultados dos experimentos.

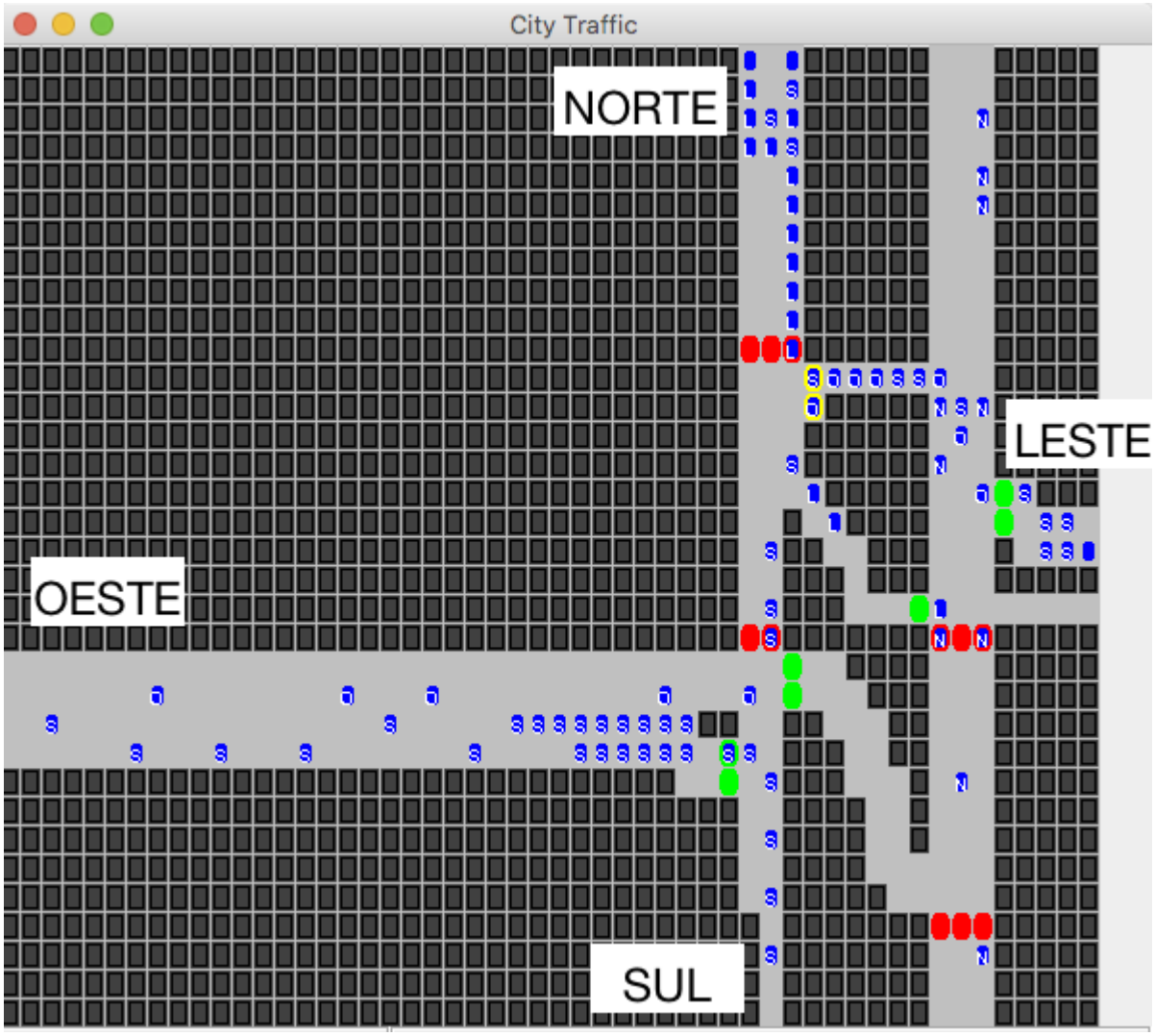
5.4.1 Descrição dos Experimentos

Com o intuito de validar a abordagem de planos compartilhados, foram criado dois tipos de cenários, um caso com poucos agentes e um caso com bastante agentes. Esses dois cenários foram colocados a prova com dois tipos de agentes, um agente com a capacidade de cooperação usando planos compartilhados e um outro agente que não sabe cooperar, precisando usar semáforos.

Em cada cenário proposto, há 12 rotas ativas. E cada uma delas, possui uma quantidade única de agentes. Cada rota possui também uma frequência para a entrada de um agente, representado na unidade de instante de tempo *t* (Ver sobre TrafficClock e Sincronização dos agentes no capítulo 5.3.2.4).

Importante destacar que não existe nenhum tipo de atraso para sair do sistema quando o agente chega em seu destino final. Ou seja, o próprio cruzamento está criando o trânsito.

Figura 9 - Cruzamento



Fonte: Autor (2016).

Ambiente desenvolvido usando o JaCaMo no qual os agentes foram aplicados. As cores vermelho, verde e amarelo são o semáforo no seu estado atual. A cor azul representa o agente, a letra em cima do agente informa qual para onde o agente está indo (a primeira letra de cada ponto cardeal).

O Cenário 1 foi executado com um total de 100 agentes e o Cenário 2 com um total de 400 agentes, distribuídos para cada rota conforme a tabela abaixo:

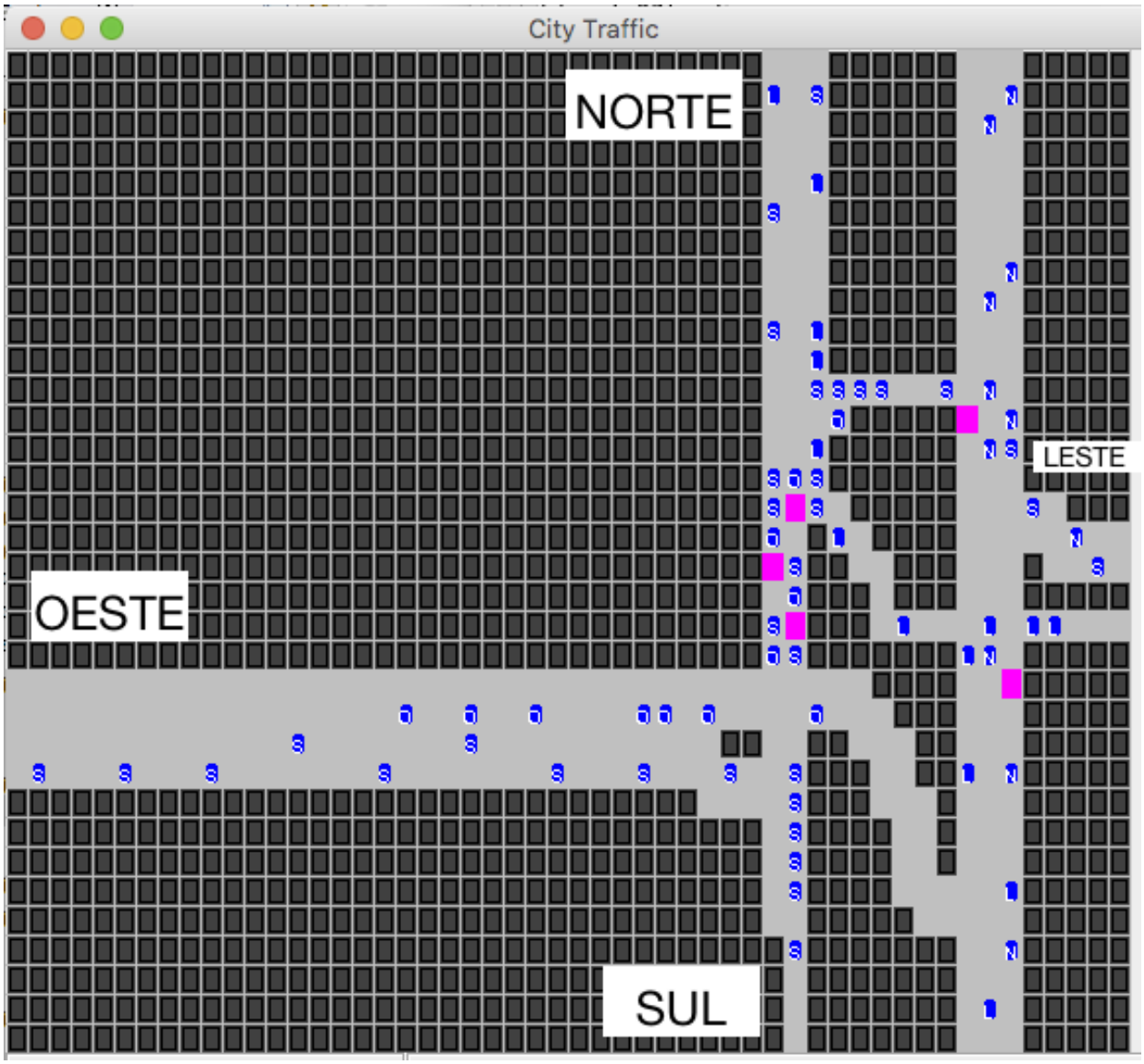
Tabela 7- Cenários

		Cenário 1	**Cenário 2**
	Frequência (em instante *t*)	**Quantidade de agentes**	**Quantidade de agentes**
Do Norte para Sul	4	15	60
Do Norte para Oeste	6	4	16
Do Norte para Leste	4	12	48
Do Norte para Norte	10	1	4
Do Oeste para Sul	4	15	60
Do Leste para Norte	6	10	40
Do Leste para Sul	4	12	48
Do Leste para Oeste	6	4	16
Do Sul para Norte	6	10	40
Do Sul para Sul	10	1	4
Do Sul para Leste	4	12	48
Do Sul para Oeste	6	4	16

Fonte: Autor (2016).

Um segundo cenário em que os agentes foram testados foi uma situação de que exigiu replanejamento dos planos. Para isso, foi criado o conceito de buraco no ambiente.

Figura 10 - Os pontos em rosa representam buracos na estrada e os agentes precisam desviar.



Fonte: Autor (2016).

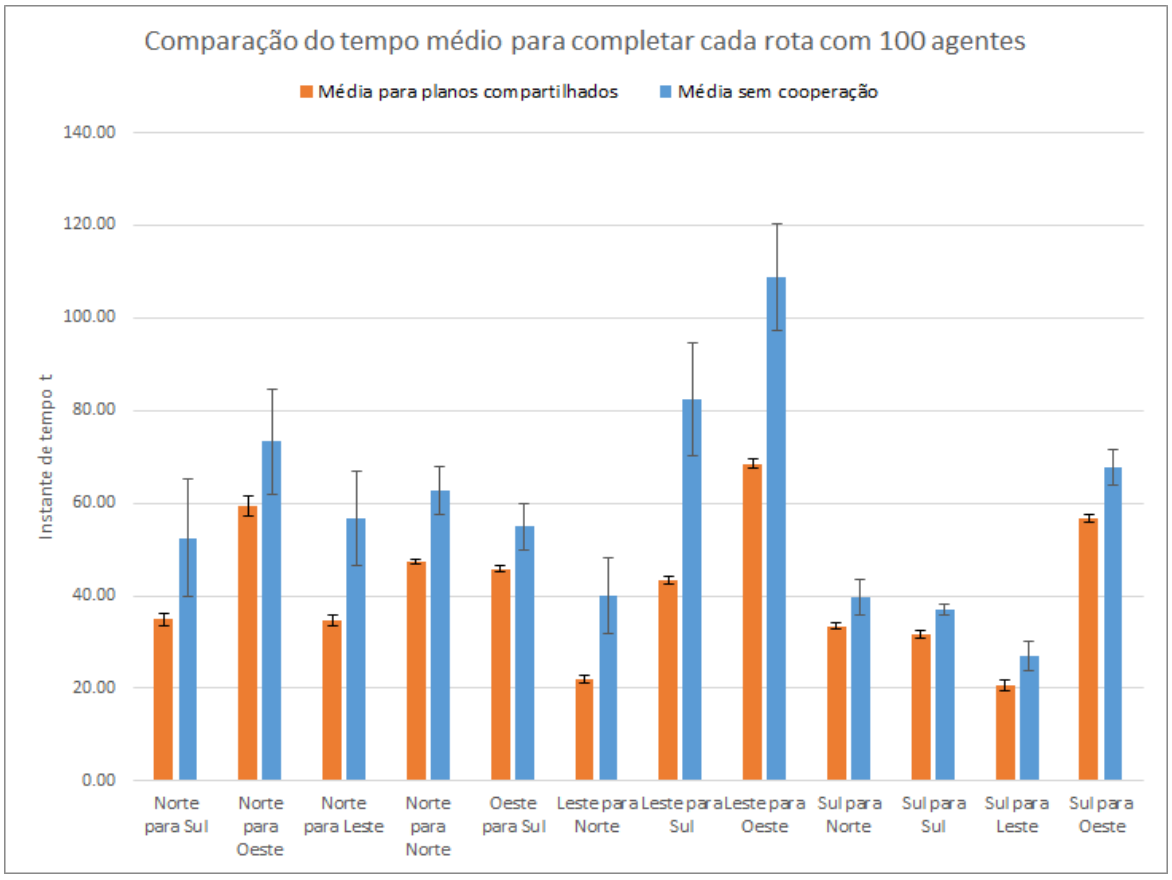
Os buracos são obstáculos no ambiente (em rosa na figura acima) que o agente só descobre quando fica adjacente. O agente sem capacidade de cooperação, quando encontra um buraco, ele desvia e segue em frente. Já o agente com cooperação, ao identificar um buraco, ele altera os planos

de todos os agentes que seriam afetados pelo buraco e compartilha o novo plano global para os outros agentes.

5.4.2 Resultados dos experimentos

Os dois cenários foram testados 5 vezes para os dois tipos de agentes. Foi coletado o instante de tempo t em que o agente entrou no sistema e em qual instante de tempo ele saiu do sistema. E com essas informações, foi feita análise abaixo.

Figura 11 - Comparação de Tempo médio



Fonte: Autor (2016).

No gráfico acima, temos uma comparação do tempo médio para cada rota no experimento com 100 agentes ao mesmo tempo. O primeiro destaque é que todas as rotas concluíram mais

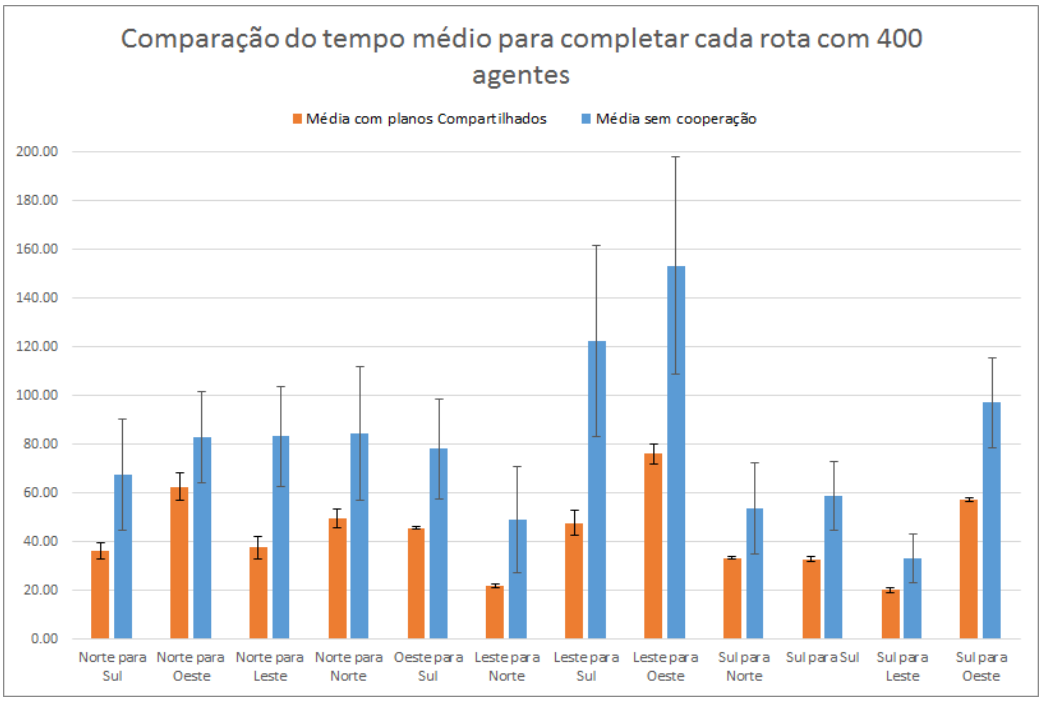
rapidamente com agentes capazes de cooperar entre si. Conseguindo quase metade do tempo de execução com que algumas das rotas, como Leste para Norte (54%) e Leste para Sul (52%).

Outra dado interessante é o desvio padrão elevado quando não há cooperação. Chegando a 12t no caso de Leste para Sul e no caso de Norte para Sul, já a abordagem de cooperação, o desvio padrão se mantém quase zero na maioria das rotas.

O comportamento do desvio padrão pode ser explicado pelo funcionamento de cada abordagem. No caso sem cooperação, os agentes são restritos pelo semáforo, caso o agente tenha a sorte de pegar somente semáforo verde, ele consegue terminar sua rota rapidamente, caso contrário ele terá que esperar o semáforo abrir. Essa característica torna o sistema incerto.

Já no caso de planos compartilhados, os agentes obedecem um algoritmo que define que cada agente só deve esperar um turno, por causa disso, um agente não fica dependendo da sorte. Criando um sistema mais determinístico.

Figura 12 - Comparação de tempo médio



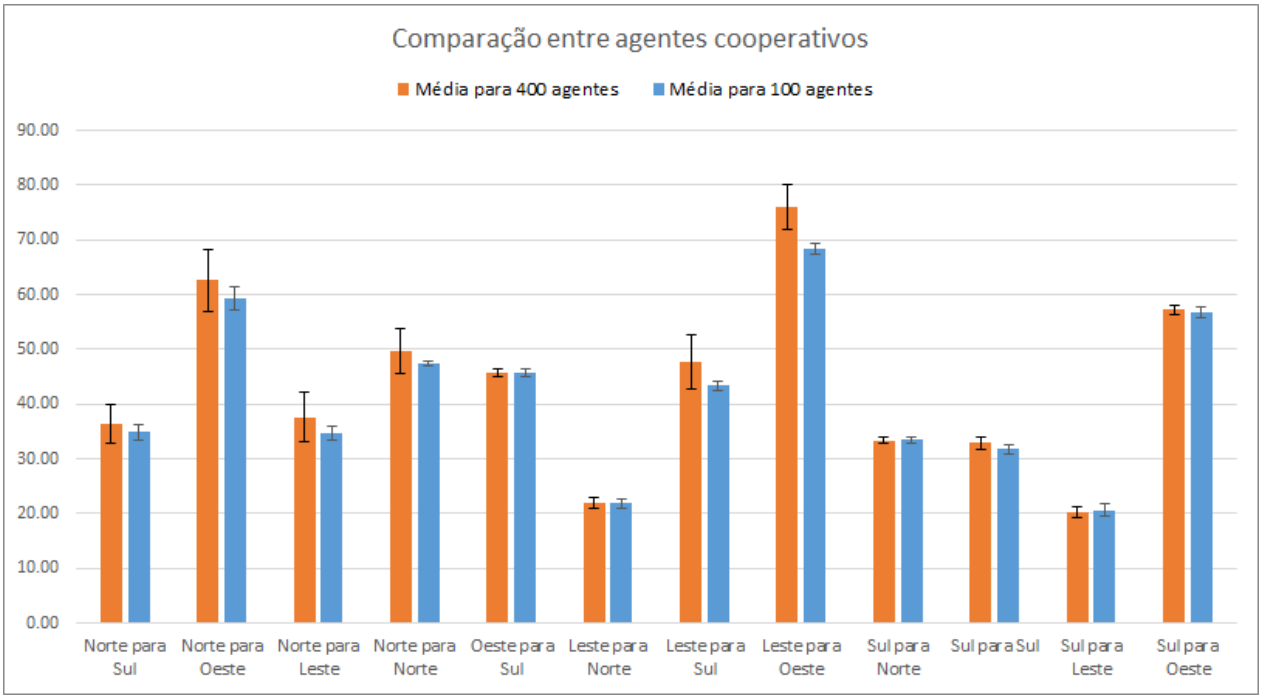
Fonte: Autor (2016).

No experimento com 400 agentes a situação se mostra ainda uma maior vantagem de cooperação com planos compartilhados. A maioria das rotas apresentam cerca de metade do tempo médio e em Leste para Sul, os planos compartilhados termina em 39% do tempo.

A abordagem com semáforo mostra novamente um grande desvio padrão, apoiando novamente a hipótese apresentada anteriormente de que a existência de semáforos cria muita incerteza no tempo total que um agente levaria para chegar em seu destino.

O desvio padrão da abordagem com planos aumentou um pouco, mas ainda sim, nenhum desvio acima de 5t.

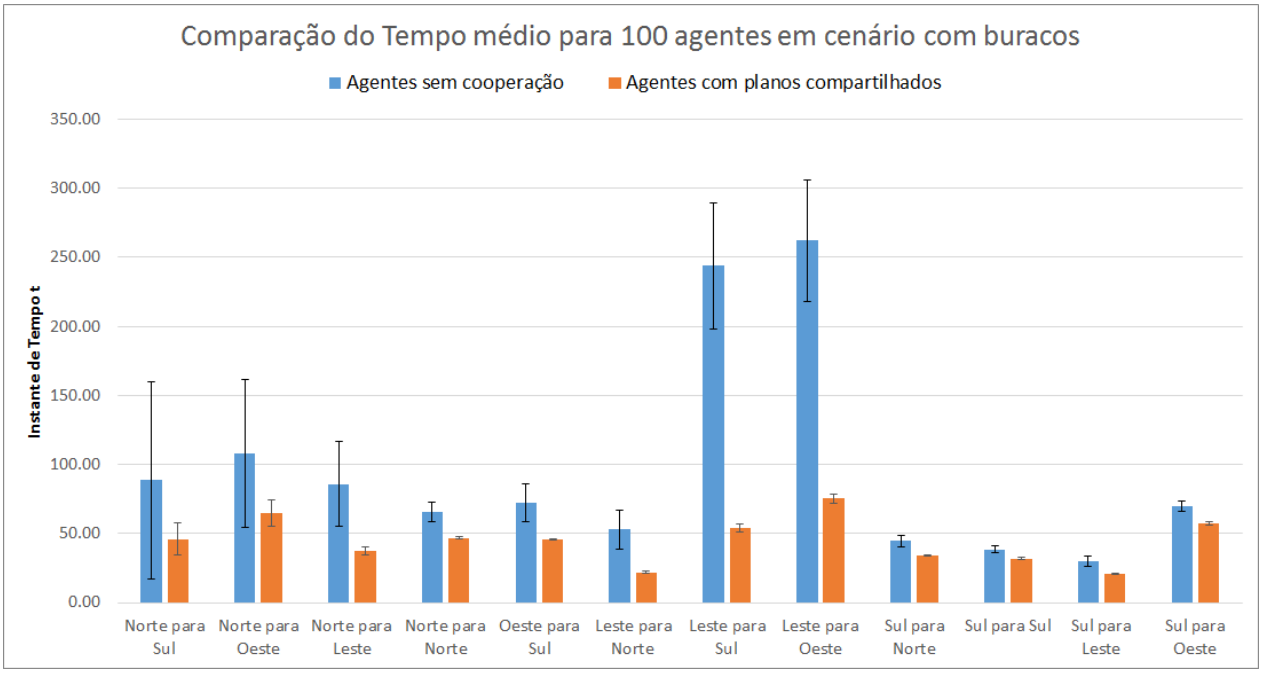
Figura 13 - Comparação entre agentes cooperativos



Fonte: Autor (2016).

E ao comparar os dados do agente cooperativo para o experimento com 100 agentes simultâneos e 400 agentes simultâneos, nota-se que houve um aumento do tempo entre 0% e 11%. Um aumento pequeno para 4 vezes mais agentes no sistema, embora talvez seja possível diminuir essa variação se os agentes evitarem bloquear agentes que estão indo para um destino diferente dele.

Figura 14 - Comparação de tempo médio



Fonte: Autor (2016).

O caso com buraco se repete os resultados. Os agentes com cooperação terminam as rotas com pouca dificuldade, principalmente pois o primeiro agente a encontrar o buraco já atualiza o plano global.

Por fim, depois de comparar e analisar os resultados dos experimentos, é possível considerar a abordagem baseada em planos compartilhados um grande avanço comparado com agentes sem capacidade de cooperação. No entanto, a execução de mais casos de testes, com mais e menos agentes, e com uma frequência de agentes diferente, apresentaria uma diversidade maior de informações sobre a atuação dos agentes.

6. CONSIDERAÇÕES FINAIS

Embora várias melhorias poderiam ser realizadas no algoritmo para compartilhamento de planos globais, ou mesmo melhorias na implementação do sistema no JaCaMo, este trabalho conseguiu expor os vários problemas na cooperação entre agentes e propôs algumas abordagens para solucionar alguns desses problemas.

O estudo em sistemas multiagentes no capítulo 3 e a análise das abordagens aplicadas a cooperação apresentados no capítulo 3.6 serviram como base para o desenvolvimento deste trabalho. Com esse conhecimento foi possível criar um cenário que poderia se beneficiar da cooperação de agentes. Além disso, as abordagens estudadas auxiliaram a identificar qual delas poderia ajudar a solucionar o cenário proposto. Como resultado, foi desenvolvida uma abordagem baseada em planos compartilhados. Para conseguir testá-la, a mesma foi implementada no JaCaMO e comparada com agentes sem capacidade de cooperar.

Os resultados dos testes, embora simples, demonstra que o agente com cooperação apresenta melhores resultados globais. No entanto, a capacidade de cooperação no cenário proposto é limitado principalmente pela quantidade de agentes no sistema. Ao realizar testes com mais de um mil agentes, os agentes não tinham vantagem no planejamento, pois não existia variação na movimentação. E pelo fato que a resolução de conflito não é eficiente, um carro poderia ficar esperando vários instantes de tempo em um cruzamento.

A resolução de conflito não é eficiente, como explicado no Capítulo 5.3.1 Algoritmo adaptado ao Trânsito, pois um agente pode fazer um outro agente esperar mais de dois instante de tempo; e por causa disso, atrasar vários outros agentes. A resolução de conflitos teria que analisar as rotas dos outros agentes antes de alterar o plano e tentar encontrar uma solução em que o menor número de agentes sejam afetados, mas sem que um determinado agente fique esperando.

Este trabalho demonstrou o funcionamento de uma abordagem baseada em planos compartilhados para cooperação de agentes. E apresentou também os problemas e soluções encontrados na implementação dessa abordagem no JaCaMO, uma plataforma que pode ser

utilizada para vários implementações de SMA. Mesmo assim, várias melhorias poderiam ser feitas, desde otimização no algoritmo para encontrar colisões no plano global, até algumas sugestões de trabalhos futuro.

6.1 TRABALHOS FUTUROS

A primeira sugestão seria converter os conceitos de sociedade, que estão implícitos nesta implementação, para um conjunto de papéis e missões criadas no módulo do Moise+. Por exemplo, entrar no sistema, criar um plano, ir para casa, poderiam ser missões que o papel nomeado *trabalhador* teria que cumprir.

Uma terceira sugestão, seria aumentar o tamanho do sistema, criando múltiplas áreas de cooperação. Um agente não precisaria cooperar com todos os agentes do sistema, somente com os agentes que afetam suas tarefas.

E uma última sugestão é a generalização da implementação do algoritmo de tráfego de modo que seja independente do problema. Seria um trabalho complexo, exigindo uma refatoração desta implementação, criando interfaces que deveriam ser implementadas para cada situação.

REFERÊNCIAS

TWEEDALE, J., ICHALKARANJE, N., SIOUTIS C., JARVIS B., CONSOLI A., PHILLIPS-WREN G. Innovations in multi-agent systems **Published by Elsevier Ltd**. 2006.

WOOLDRIDGE, Michael. An Introduction to MultiAgent System, second edition. United Kingdom**. John Wiley & Sons Ltd**, 2009.

SHOHAM, Yoav; LEYTON-BROWN, Kevin. MULTIAGENT SYSTEM, Algorithm, Game-Theoric**, and Logical Foundations**, Revisão 1.1, 2010.

BINMORE, K. G., Fun and Games**: A text on Game Theory**, D.C. Heath, 1992.

SPERBER, Dan; WILSON, Deirdre; Relevance: Communication and Cognition; Second Edition; **Blackwell Publishing**, 1996.

GARDINER, W. Lambert. **The Psychology of Communication; Trafford Publishing;** 2008.

HOARE, C. A. R.. **Communicating Sequential Processes.** 1978.

MILNER, R.; Communication and Concurrency; **Prentice-Hall international series in computer science**; 1989.

SEARLE, John R.; Speech Acts: **An Essay in the Philosophy of Language**; Cambridge University Press, 1969.

COHEN, Philip R.; LEVESQUE, Hector J.; **Intention Is Choice With Commitment**; Elsevier, North-Holland, 1990.

COHEN, P. R., and LEVESQUE, H. J. Rational interaction as the basis for communication. In Cohen, P. R.; Morgan, J.; and Pollack, M. E., eds., **Intentions in Communication** . MIT Press. 1990b.

AUSTIN, J. L., How to do things with words, London**: Oxford University Press,** 1962.

GENESERETH, Michael R 1991 knowledge interchange format. In proceedings of the 2nd **International Conference of Principles of Knowledge Representation and Reasoning,** 1991.

HUHNS, M.N. et al, Reading in Agents, **Morgan Kaufmann Publishers**, Inc., San Francisco, California,1998.

Foundation for Intelligent Physical Agents (FIPA) FIPA Architectural Overview, **FIPA Technical Committee A,San Francisco**,CA. International Standards, 1999.

SADEK, M. D.; BRETIER P.; PANAGET F.; ARTIMIS: natural dialogue meets rational agency IJCAI'97 **Proceedings of the Fifteenth international joint conference on Artifical intelligence -** Volume 2 1997.

BERNERS-LEE, T., HENDLER, J., and LASSILA, O. **The semantic web. Scientific American**, 2001.

BECHHOFER, S., VAN HARMELEN, F., HENDLER, J., HORROCKS, I., MCGUINNESS, D. L., PATEL-SCHNEIDER, P. F., and STEIN, L. A. OWL **web ontology**, 2004.

VON MARTIAL, F.**Interactions among autonomous planning agents**. 1990.

VON MARTIAL, F. **Coordinating Plans of Autonomous Agents** (LNAI Volume 610). Springer-Verlag: Berlin, Germany. 1992.

DURFEE, E. H. Coordination of Distributed Problem Solvers. **Kluwer Academic Publishers: Dordrecht**, The Netherlands. 1988.

DURFEE, E. H. Planning in distributed artificial intelligence. In O'Hare, G. M. P. and Jennings, N. R., editors, **Foundations of Distributed Artificial Intelligence**, John Wiley & Sons. 1996.

DURFEE, E. H. and Lesser, V. R. Using partial global plans to coordinate distributed problem solvers. **In Proceedings of the Tenth International Joint Conference on Artificial Intelligence (**IJCAI-87), Milan, Italy. 1987.

FRANKLIN, S. and Graesser, A. . Is it an agent, or just a program? In Müller, J. P., Wooldridge, M., and Jennings, N. R., editors, **Intelligent Agents III** (LNAI Volume 1193), Springer-Verlag: Berlin, Germany. 1997.

JENNINGS, N. R. Commitments and conventions: The foundation of coordination in multi-agent systems. **The Knowledge Engineering Review,** 8(3): 223–250. 1993a.

KRAUS, S. Strategic Negotiation in Multiagent Environments. **The MIT Press: Cambridge**, MA. 2001.

DUNG, P. M. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. **Artificial Intelligence,** 77: 321–357. 1995.

VREEWIJK, G. A. W. and PRAKKEN, H. Credulous and sceptical argument games for preferred semantics. In Ojeda-Aciego, M., de Guzmán, I. P., Brewka, G., and Pereira, L. M., editors, **Logics in Artificial Intelligence – Proceedings of the Seventh European Workshop,** JELIA 2000 (LNAI Volume 1919), pages 239–253. Springer-Verlag: Berlin, Germany. 2000.

BESNARD, P. and HUNTER, A. Elements of Argumentation. **The MIT Press:** Cambridge, MA. 2008.

EL-SISI, Ashraf; MOUSA, Hamdy, **Argumentation Based Negotiation in Multi-agent System,** 2014.

PARSONS S., WOOLDRIDGE M., AMGOUD L. An analysis of formal interagent dialogues. In Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems (**AAMAS-02), Bologna**, Italy, July 2002.

IEONG, S., and SHOHAM, Y. Bayesian coalitional games. AAAI: Proceedings **of the AAAI Conference on Artificial Intelligence**. 2008.

GARDNER, J, DOWD, A-M., MASON, C. and ASHWORTH, P. A framework for stakeholder engagement on climate adaptation**. CSIRO Climate Adaptation Flagship Working paper** No.3. http://www.csiro.au/resources/CAF-working-papers.html. (2009).

DURFEE, E. H,. LESSER, V.R. and CORKILL, D.D. Trends in cooperative distributed problem solving. **IEEE Transactions on Knowledge and Data Engineering**, 1(1), 63-83. (1989b).

WOOLDRIDGE, M. and JENNINGS, N. R. Formalizing the Cooperative Problem Solving Process. In M. Klein, editor, Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence (IWDAI-93), **Lake Quinalt, WA**, July 1994.

ADLER, M.R. et al. Conflict resolution strategies for non hierarchical distributed agents. In Distributed Artificial Intelligence (eds.L. Gasser and M. Huhns), Volume 2, pp. 139-162. **Pitman, London and Morgan Kaufmann**, San Mateo, Ca. (1989).

GALLIERS, J.R. A theoretical frameqork for computer models of cooperative dialogue,acknowledging multi-agent conflict. **PhD thesis, The Open University**, UK (1988b).

LANDER, S., LESSER, V.R. & CONNEL, M.E. (1991). Conflict resolution strategies for cooperatingexpert agents. In CKBS-90. **Proceedings of the International Working Conference onCooperating Knowledge Based Systems** (ed. S.M. Deen),pp. 183-200. Springer,Berlin.

EPHRATI, E., Rosenschein, J.S. Multi-agent planning as a dynamic search for social consensus. In Proceeding softhe **Thirteenth International Joint Conference on Artificial Intelligence**, 423-429 1993.

ZLOTKIN, Gilad; ROSENSCHEIN, Jeffrey S., Coalition, Cryptography, and Stability: Mechanisms for Coalition Formation in Task Oriented Domains, **In Proc. of AAAI94**, 1994.

DURFEE, E. H.. Distributed problem solving and planning. In Weiß, G., editor, Multiagent Systems, pages 121–164. **The MIT Press: Cambrid** 1999.

PANAIT, Liviu, LUKE Sean Cooperative Multi-Agent Learning - State of the Art, **Springer Science - Netherlands** 2005.

CLEMENT, Brad; WEERDT, Mathijs de; **Introduction to Planning in Multiagent Systems**, 2009.

DURFEE, Edmund H. Distributed Problem Solving and Planning; Artificial Intelligence **Laboratory EECS Department University of Michigan**, Lecture Notes in Computer Science , 2001.

WALSH, William E., WELLMAN Michael P., YGGE F. . Combinatorial auctions for supply chain formation. **In Second ACM Conference on Electronic Commerce**, pages 260–269. ACM Press, 2000. URL http://www-personal.engin.umich.edu/ wew/research.html.

WELLMAN, Michael P. , WALSH William E., WURMAN Peter R., MACKIE-MASON, Jeffrey K. . **Auction protocols for decentralized scheduling. Games and Economic Behavior,** 35(1–2):271–303, 2001. URL http://www-personal.engin.umich.edu/ wew/research.html.

WELLMAN, Michael P. . A market-oriented programming environment and its application to distributed multicommodity flow problems. **Journal of Artificial Intelligence Research,** 1:1–23, 1993.

WELLMAN, Michael P., WALSH William E., WURMAN P.R., MACKIE-MASON J.K.. Auction protocols for decentralized scheduling. In Proceedings of the Eighteenth International **Conference on Distributed Computing Systems,** May 1998..

FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, **Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015), 2015.

PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th **International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015) , 2015.

LAGOUDAKIS, Michail G., BERHAULT Marc, KOENIG Sven, KESKINOCAK Pinar, KLEYWEGT Anton J., Simple auctions with performance guarantees for multi-robot task allocation, Proceeding of 2004 **IEEE - Intelligent Robots and Systems**, 2004.

KORSAH, G. Ayorkor, DIAS, M. Bernardine, STENTZ, Anthony, **A Comprehensive Taxonomy for Multi-Robot Task Allocation, research** 2013.

GERKEY Brian P., MATARIC Maja J, **A formal analysis and taxonomy of task allocation in multi-robot systems**, 2004.

WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, **Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015).

WOOLDRIDGE, JENNING, **The Cooperative Problem Solving Process**, 1999.

BOISSIER , Oliver; COLOMBETTI, Marco;LUCK, Michael;MEYER, John-Jules, POLLERES, Axel; Norms, organizations, and semantics; **The Knowledge Engineering Review**, Vol. 28:1, 107–116. Cambridge University Press, 2012.

KAGAL L, FININ T, JOSHI A. A policy language for pervasive systems. **In: Fourth IEEE** international workshop on policies for distributed systems and networks; 2003.

BOELLA, G., and VAN DER TORRE, L ; The social delegation cycle**. InProcs. of DEON'04** Workshop. 2004.

HARMON, S. J., DELOACH, S. A. & ROBBY. Trace-based specification of law and guidance policies for multi-agent systems. In Engineering Societies in the Agents World VIII, 8th International Workshop, ESAW 2007, Athens, Greece, October 22–24, 2007, Artikis, A., O'Hare, G. M. P., Stathis, K. & Vouros, G. A. (eds), **Revised Selected Papers, Lecture Notes in Computer Science,** 4995, 333–349. Springer, ISBN 978-3- 540-87653-3. 2008

ALECHINA, Natasha, BULLING, Nils, DASTANI, Mehdi; Practical Run-Time Norm Enforcement with Bounded Lookahead, Proceedings of the 14th International Conference on **Autonomous Agents and Multiagent Systems (**AAMAS 2015)

KING, Thomas C.; LI, Tingting; VOS, Marina De, DIGNUM, Virginia; JONKER, Catholijn M., PADGET, Julian, M. RIEMSDIJK, Birna van; A Framework for Institutions Governing Institutions, **Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015)

BRATMAN, M. Intention, Plans, and Practical Reason. Harvard University Press, **Cambridge, Massachusetts**, 1987.

POKAHR, Alexander, BRAUBACH, Lars, LAMERSDORF, Winfried, JADEX: A BDI REASONING ENGINE, **Springer US, University of Hamburg**, 2005.

WIKIPEDIA, **Belief–desire–intention software model**, disponível em: < https://en.wikipedia.org/wiki/Belief%E2%80%93desire%E2%80%93intention_software_model > Acesso em 04.07.2016

RAO, A., GEORGEFF, M.. BDI Agents: from theory to practice. In V. Lesser, editor, Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), pages 312–319**, The MIT Press: Cambridge,** MA, USA., San Francisco, CA, USA, 1995

HUBNER, Jomi F., SICHMAN, Jaime S., BOISSIER, Olivier, Developing Organised Multi-Agent Systems Using the Moise+ Model: Programming Issues at the System and Agent Levels, **International Journal of Agent-Oriented Software Engineering archive, Volume** 1 Issue 3/4, Pages 370-395 , 2007

HUBNER, Jomi F., BOISSIER, Olivier, KITIO, Rosine, RICCI, Alessandro, Instrumenting multi-agent organisations with organisational artifacts and agents "Giving the organisational power back to the agents", 2009
JaCaMo - Aproach, Disponível em < http://jacamo.sourceforge.net/ > Acesso em 12.12.2015

OKIMOTO, Tenda; SCHWIND, Nicolas; CLEMENT, Maxime; RIBEIRO, Tony; INOUE, Katsumi; MARQUIS, Pierre; How to Form a Task-Oriented Robust Team, Proceedings of the **14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015), 2015

ECK, Eck; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the **14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015), 2015.

ANEXOS

O código abaixo também está disponível no site https://github.com/ricpass/tcc de forma pública.

1. traffic.jcm

```
/*

    JaCaMo Project File

    This file defines the initial state of the MAS (initial agents, environment,
organisation, ....)

    (see below the documentation about what can be defined in this file)

*/

mas traffic {

    agent ricardo : cooperative-driver.asl {
//    agent ricardo : driver.asl {
       focus : city.floripa1
       instances : 2
    }

    workspace city {
       artifact floripa1 : traffic.DrivingModel()
//     debug
    }

}
```

2. **TestMap.java**

```
package traffic;

import jacamo.infra.JaCaMoLauncher;
import jason.JasonException;

public class TestMap {

	public static void main(String[] args) {
		TestMap t = new TestMap();

		t.start();
	}

	private void start() {
		try {
			JaCaMoLauncher runner = new JaCaMoLauncher();
			runner.init(new String[] { "./traffic.jcm" });
			runner.getProject().addSourcePath("./src/agt");
			runner.create();
			runner.start();
			runner.waitEnd();
			runner.finish();
		} catch (JasonException e) {
			e.printStackTrace();
		}
	}

}
```

3. DrivingModel.java

```
// CArtAgO artifact code for project traffic

package traffic;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;

import agent.AgentFactory;
import agent.CityAgent;
import cartago.Artifact;
import cartago.OPERATION;
import jason.asSyntax.Atom;
import jason.asSyntax.Term;
import jason.environment.grid.Location;
import semaphore.SemaphoreFactory;
import system.TrafficClock;

public class DrivingModel extends Artifact {

	private static CityModel model;
	private static CityView view;

	private static int sleep = 500;

	private static Term obstacle = new Atom("obstacle");
	private static Term buraco = new Atom("buraco");
	private static Term semaphoroRed = new Atom("semaphoroRed");
	private static Term semaphoroYellow = new Atom("semaphoroYellow");
	private static Term semaphoroGreen = new Atom("semaphoroGreen");
	private static final Logger logger =
Logger.getLogger(DrivingModel.class.getName());

	private static AtomicInteger errors = new AtomicInteger(0);

	@OPERATION
	public void init() {
		logger.info("init world agentId: ");
		initWorld();
	}

	private void initWorld() {
		if (model == null) {
			model = CityModel.createCity();
			view = CityView.createView(model);
			view.setEnv(this);
			SemaphoreFactory.createSemaphores();
			AgentFactory.createAgents();

			TrafficClock.startCounter();
		}
```

```
}

@OPERATION
void move(String name, int x, int y) {
	logger.info(name + " DrivingModel.move()");

	CityAgent agent = AgentFactory.getAgentByName(name);
	int agentId = agent.getId();

	TrafficClock.increamentAgentCount(name);

	boolean moved = model.move(agentId, x, y);
	if (!moved) {
		errors.incrementAndGet();
		logger.info("errors " + errors);
	}

	updateAgPercept(agentId);

}

@OPERATION
void home(String name) {
	try {
		await_time(TrafficClock.AGENT_WAIT_TIME);
		TrafficClock.increamentAgentCount(name);

		CityAgent agent = AgentFactory.getAgentByName(name);

		if (agent != null) {
			int agentId = agent.getId();

			model.removeAgent(agentId);

			synchronized (TrafficClock.COUNTER) {
				AgentFactory.removeAgentInTheSystem(name);
			}

		}
	} catch (Exception e) {
		e.printStackTrace();
	}
}

@OPERATION
void skip(String name) {
	try {
		TrafficClock.increamentAgentCount(name);

		CityAgent agent = AgentFactory.getAgentByName(name);

		if (agent != null) {
			int agentId = agent.getId();
```

```
                        updateAgPercept(agentId);
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
}

private void updateAgPercept(int agentId) {
        // its location
        Location location = model.getAgPos(agentId);

        // what's around
        for (int i = 1; i < 2; i++) {
                updateAgPercept(location.x - i, location.y - i);
                updateAgPercept(location.x - i, location.y);
                updateAgPercept(location.x - i, location.y + i);
                updateAgPercept(location.x, location.y - i);
                updateAgPercept(location.x, location.y);
                updateAgPercept(location.x, location.y + i);
                updateAgPercept(location.x + i, location.y - i);
                updateAgPercept(location.x + i, location.y);
                updateAgPercept(location.x + i, location.y + i);
        }

        // view.update();
}

private void updateAgPercept(int x, int y) {
        if (model == null || !model.inGrid(x, y))
                return;

        // remove all first
        try {
                removeObsPropertyByTemplate("cell", null, null, null);
        } catch (IllegalArgumentException e) {
        }

        String semaphoreName = "cell"; // _"+x+"_"+y;

        if (model.hasObject(CityModel.OBSTACLE, x, y)) {
                defineObsProperty("cell", x, y, obstacle);
        } else if (model.hasObject(CityModel.SEMAPHORO_GREEN, x, y)) {
                // System.out.println("has green at (" + x + ", " + y + ")");
                defineObsProperty(semaphoreName, x, y, semaphoroGreen);

        } else if (model.hasObject(CityModel.SEMAPHORO_RED, x, y)) {
                // System.out.println("has red (" + x + ", " + y + ")");
                defineObsProperty(semaphoreName, x, y, semaphoroRed);

        } else if (model.hasObject(CityModel.SEMAPHORO_YELLOW, x, y)) {
                // System.out.println("has yellow (" + x + ", " + y + ")");
                defineObsProperty(semaphoreName, x, y, semaphoroYellow);
```

```
            } else if (model.hasObject(CityModel.BURACO, x, y)) {
                  defineObsProperty("cell", x, y, buraco);
            }

      }

      public void setSleep(int s) {
            sleep = s;
      }

      public static Location getAgentLocation(String name) {
            CityAgent agent = AgentFactory.getAgentByName(name);

            if (agent != null) {
                  int agentId = agent.getId();
                  return model.getAgPos(agentId);
            }
            return null;
      }

      public static Location[] whereIsMyHome(String name) {
            CityAgent agent = AgentFactory.getAgentByName(name);
            if (agent != null)
                  return agent.getHomes();
            return null;
      }
}
```

4. CityView.java

```
package traffic;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.util.Hashtable;

import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import agent.AgentFactory;
import configuration.TrafficConfiguration;
import jason.environment.grid.GridWorldModel;
import jason.environment.grid.GridWorldView;

public class CityView extends GridWorldView {

	private DrivingModel env;

	private JSlider jSpeed;
	private JLabel jlMouseLoc;
	private static CityView view;

	public static CityView createView(CityModel model) {
		view = new CityView(model);
		return view;
	}

	public CityView(GridWorldModel model) {
		super(model, "City Traffic", 600);
		setVisible(true);
		repaint();
	}

	/**
	 *
	 */
	private static final long serialVersionUID = 1843044389185345713L;

	public void setEnv(DrivingModel env) {
		this.env = env;
```

```
	}

	@Override
	public void initComponents(int width) {
		super.initComponents(width);
		JPanel args = new JPanel();
		args.setLayout(new BoxLayout(args, BoxLayout.Y_AXIS));
		int maxSpeed = 100;
		int normalSpeed = 500;
		int slowSpeed = 1000;

		jSpeed = new JSlider();
		jSpeed.setMinimum(maxSpeed);
		jSpeed.setMaximum(slowSpeed);
		jSpeed.setValue(normalSpeed);
		jSpeed.setPaintTicks(true);
		jSpeed.setPaintLabels(true);
		jSpeed.setMajorTickSpacing(100);
		jSpeed.setMinorTickSpacing(20);
		jSpeed.setInverted(true);
		Hashtable<Integer, Component> labelTable = new Hashtable<Integer,
Component>();
		labelTable.put(maxSpeed, new JLabel("max"));
		labelTable.put(normalSpeed, new JLabel("speed"));
		labelTable.put(slowSpeed, new JLabel("min"));
		jSpeed.setLabelTable(labelTable);
		JPanel p = new JPanel(new FlowLayout());
		p.setBorder(BorderFactory.createEtchedBorder());
		p.add(jSpeed);

		args.add(p);

		JPanel msg = new JPanel();
		msg.setLayout(new BoxLayout(msg, BoxLayout.Y_AXIS));
		msg.setBorder(BorderFactory.createEtchedBorder());

		p = new JPanel(new FlowLayout(FlowLayout.CENTER));
		p.add(new JLabel("(mouse at:"));
		jlMouseLoc = new JLabel("0,0)");
		p.add(jlMouseLoc);
		msg.add(p);

		JPanel s = new JPanel(new BorderLayout());
		s.add(BorderLayout.WEST, args);
		s.add(BorderLayout.CENTER, msg);
		getContentPane().add(BorderLayout.SOUTH, s);

		// Events handling
		jSpeed.addChangeListener(new ChangeListener() {
			public void stateChanged(ChangeEvent e) {
				if (env != null) {
					env.setSleep((int) jSpeed.getValue());
				}
```

```
                }
            });

            getCanvas().addMouseMotionListener(new MouseMotionListener() {
                public void mouseDragged(MouseEvent e) {
                }

                public void mouseMoved(MouseEvent e) {
                    int col = e.getX() / cellSizeW;
                    int lin = e.getY() / cellSizeH;
                    if (col >= 0 && lin >= 0 && col < getModel().getWidth() &&
lin < getModel().getHeight()) {
                        jlMouseLoc.setText(col + "," + lin + ")");
                    }
                }
            });

      }

    @Override
    public void draw(Graphics g, int x, int y, int object) {
      boolean debug = false;

        switch (object) {
        case CityModel.STREET_NORTH:
            if (debug){
                drawStreet(g,x,y, Color.YELLOW);
            } else {
                drawStreet(g,x,y, Color.LIGHT_GRAY);
            }
            break;
        case CityModel.STREET_SOUTH:
            if (debug){
                drawStreet(g,x,y, Color.LIGHT_GRAY);
            } else {
                drawStreet(g,x,y, Color.LIGHT_GRAY);
            }
            break;
        case CityModel.STREET_EAST:
            if (debug){
                drawStreet(g,x,y, Color.CYAN);
            } else {
                drawStreet(g,x,y, Color.LIGHT_GRAY);
            }
            break;
        case CityModel.STREET_WEST:
            if (debug){
                drawStreet(g,x,y, Color.MAGENTA);
            } else {
                drawStreet(g,x,y, Color.LIGHT_GRAY);
            }
            break;
        case CityModel.SEMAPHORO_GREEN:
```

```
                drawSemaphoro(Color.GREEN, g,x,y);
                break;
            case CityModel.SEMAPHORO_RED:
                drawSemaphoro(Color.RED, g,x,y);
                break;
            case CityModel.SEMAPHORO_YELLOW:
                drawSemaphoro(Color.YELLOW, g,x,y);
                break;
            case CityModel.BURACO:
                g.setColor(Color.MAGENTA);
               g.fillRect(x * cellSizeW, y * cellSizeH, cellSizeW, cellSizeH);
            }
        }

        @Override
        public void drawAgent(Graphics g, int x, int y, Color c, int id) {
          if (TrafficConfiguration.USE_ID_ON_AGENT){
                super.drawAgent(g, x, y, c, -1);
                g.setColor(Color.WHITE);
               drawString(g, x, y, defaultFont, String.valueOf(id+1));
          } else {
                super.drawAgent(g, x, y, c, -1);
                String des = AgentFactory.getAgentDestination(id);

                String letter = "";
                if (des != null){
                      switch (des) {
                      case "NORTH":
                            letter = "N";
                            break;
                      case "EAST":
                            letter = "L";
                            break;
                      case "WEST":
                            letter = "O";
                            break;
                      case "SOUTH":
                            letter = "S";
                            break;
                      }
                      g.setColor(Color.WHITE);
                      drawString(g, x, y, defaultFont, letter);
                }

          }
//        update();

        }

          private void drawStreet(Graphics g, int x, int y, Color color) {
                g.setColor(color);
            g.fillRect(x * cellSizeW, y * cellSizeH, cellSizeW, cellSizeH);
          }
```

```
	private void drawSemaphoro(Color color, Graphics g, int x, int y) {
//		System.out.println("CityView.drawSemaphoro() new color: " + color + "
("+x+","+y+")");
//		new Throwable().printStackTrace();
		g.setColor(color);
		g.fillOval(x * cellSizeW, y * cellSizeH, cellSizeW, cellSizeH);
//		update();
	}

}
```

5. CityModel.java

```
package traffic;

import java.util.HashMap;
import java.util.logging.Logger;

import agent.AgentFactory;
import configuration.TrafficConfiguration;
import graph.CityGraph;
import jason.environment.grid.GridWorldModel;
import jason.environment.grid.Location;

public class CityModel extends GridWorldModel {

	private static CityModel model;
	private String cityModelId = "";
	public static final int STREET_NORTH = 16;
	public static final int STREET_SOUTH = 32;
	public static final int STREET_EAST = 64;
	public static final int STREET_WEST = 128;

	public static final int SEMAPHORO_RED = 256;
	public static final int SEMAPHORO_GREEN = 512;
	public static final int SEMAPHORO_YELLOW = 1024;

	public static final int BURACO = 2048;

	private static final Logger logger =
Logger.getLogger(CityModel.class.getName());

	private static HashMap<Location, Object> positionsMutex = new HashMap<>();

	protected CityModel(int w, int h, int nbAgs) {
		super(w, h, nbAgs);
	}

	public static CityModel createCity() {
		logger.info("CityModel.createCity()");

		String[][] map = CityMap.getMap();

		int width = map[0].length; // x - numbers of rows
		int height = map.length; // y - numbers of columns

		model = new CityModel(width, height,
TrafficConfiguration.NUMBER_OF_AGENTS);
		model.setCityModelId("Scenario 1");

		logger.info("y.size " + map.length); // column
		logger.info("x.size " + map[0].length); // row
```

```
			for (int row = 0; row < map.length; row++) {
				for (int collumn = 0; collumn < map[0].length; collumn++) {
					String way = map[row][collumn];

					Location location = new Location(collumn, row);
					// jacamo invert x and y, so I invert again to look like
the matrix at CityMap.class
					// System.out.println(location + " " + pos);
					CityGraph.get().addOnGraph(map, row, collumn, way);

					switch (way) {
					case ".":
						model.add(CityModel.OBSTACLE, location);
						break;
					case "N":
						model.add(CityModel.STREET_NORTH, location);
						break;
					case "S":
						model.add(CityModel.STREET_SOUTH, location);
						break;
					case "E":
						model.add(CityModel.STREET_EAST, location);
						break;
					case "W":
						model.add(CityModel.STREET_WEST, location);
						break;
					case "B":
						model.add(CityModel.STREET_NORTH, location);
						// city.add(CityModel.STREET_NORTH_SEMAPHORE,
location);
						// city.add(CityModel.SEMAPHORO, location);
						break;
					case "A":
						model.add(CityModel.STREET_SOUTH, location);
						// city.add(CityModel.STREET_SOUTH_SEMAPHORE,
location);
						// city.add(CityModel.SEMAPHORO, location);
						break;
					case "K":
						model.add(CityModel.STREET_EAST, location);
						// city.add(CityModel.STREET_EAST_SEMAPHORE,
location);
						// city.add(CityModel.SEMAPHORO, location);
						break;
					case "Q":
						model.add(CityModel.STREET_WEST, location);
						// city.add(CityModel.STREET_WEST_SEMAPHORE,
location);
						// city.add(CityModel.SEMAPHORO, location);
						break;
					default:
						break;
					}
```

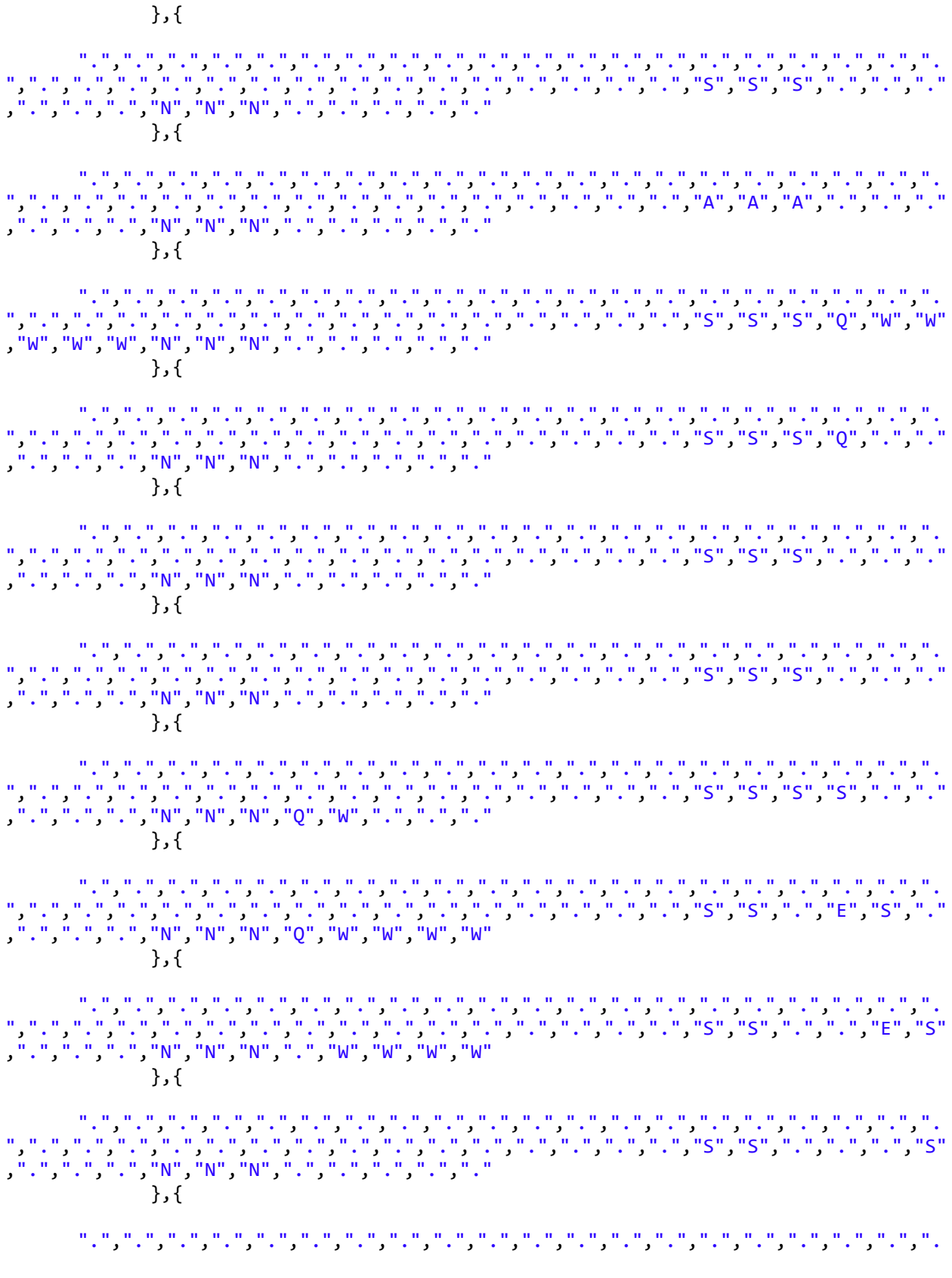
```
                        positionsMutex.put(location, new Object());
                }
        }

//          model.add(CityModel.BURACO, new Location(36, 15));
//          model.add(CityModel.BURACO, new Location(35, 17));
//          model.add(CityModel.BURACO, new Location(36, 19));
//
//          model.add(CityModel.BURACO, new Location(46, 21));
//          model.add(CityModel.BURACO, new Location(44, 12));


        return model;
    }

    private void setCityModelId(String cityModelId) {
        this.cityModelId = cityModelId;
    }

    public String getCityModelId() {
        return this.cityModelId;
    }




//  public static void createAgents() {
//      for (int i = 0; i < NUMBER_OF_AGENTS; i++) {
//          System.out.println("CityModel.createAgents()");
//          // startAgent(i);
//          int agentId = i;
//          String name = "ricardo" + (agentId + 1);
//          int x = -1;
//          int y = -1;
//          Location home;
//          if (agentId == 0) {
//              x = 36;
//              y = 0;
//              home = new Location(0, 22);
//          } else {
//              x = 36;
//              y = 1;
//              home = new Location(45, 0);
//          }
//          model.setAgPos(agentId, x, y);
//
//          agents.put(name, new CityAgent(agentId, home));
//      }
//  }
```

```
	public boolean move(int agentId, int x, int y) {
		return setAgentPosition(agentId, x, y);
	}

	public static boolean addAgent(int agentId, Location location){
		logger.info("CityModel.addAgent() id " + agentId);
		return model.setAgentPosition(agentId, location.x, location.y);
	}

	private boolean setAgentPosition(int agentId, int x, int y) {
		synchronized (AgentFactory.AGENT_FACTORY_MUTEX) {
			if (isFree(x, y)) {
				Location agPos2 = getAgPos(agentId);
				setAgPos(agentId, x, y);

				if (agPos2 != null){
					Object mutex = positionsMutex.get(agPos2);
					synchronized (mutex) {
						mutex.notifyAll();
					}
				}
				return true;
			} else {
				logger.info("postion ("+x + ", " + y + ") is not free for
agent " + agentId);
				logger.info("Agent at position ("+x+","+y+") is id " +
getAgAtPos(x, y));
				return false;
			}
		}
	}

	public void removeAgent(int agentId) {
		synchronized (AgentFactory.AGENT_FACTORY_MUTEX) {
			Location location = getAgPos(agentId);
			remove(CityModel.AGENT, location);
		}
	}

	public static void updateSemaphore(int semaphoreOld, int semaphoreNew,
Location location){
		model.remove(semaphoreOld, location);
		model.add(semaphoreNew, location);
	}

	public static void waitLocation(Location location){
		Object mutex = positionsMutex.get(location);
		synchronized (mutex) {
			try {
				mutex.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
```

```
		}
	}

	public static boolean isLocationFree(int x, int y){
		return model.isFree(x, y);
	}

	public static int getAgentAtLocation(int x, int y){
		return model.getAgAtPos(x, y);
	}


}
```

6. CityMap.java

```
package traffic;

public class CityMap {

	private static final String[][] map = {
		{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
		},{
	".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S","S",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
```



```
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S",".",".",".","S"
,"E","E","E","N","N","N","E","E","E","E","E"
            },{

        ".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","A","A",".",".",".","."
,".",".",".","B","B","B",".",".",".",".","."
            },{

        "W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W
","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","S","S","Q","W","W","."
,".",".",".","N","N","N",".",".",".",".","."
            },{

        "W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W
","W","W","W","W","W","W","W","W","W","W","W","W","W","W","W","S","S","Q","W","W","W"
,".",".",".","N","N","N",".",".",".",".","."
            },{

        "E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E
","E","E","E","E","E","E","E","E","E","E","E","E","E",".",".","S","S",".",".","N","W"
,"W",".",".","N","N","N",".",".",".",".","."
            },{

        "E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E","E
","E","E","E","E","E","E","E","E","E","E","E","E","E","E","K","S","S",".",".",".","N"
,"N",".",".","N","N","N",".",".",".",".","."
            },{

        ".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".","E","E","K","S","S",".",".",".","N"
,"N","W",".","N","N","N",".",".",".",".","."
            },{

        ".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S",".",".",".","."
,"N","N",".","N","N","N",".",".",".",".","."
            },{

        ".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S","S",".",".",".","."
,"N","N",".","N","N","N",".",".",".",".","."
            },{

        ".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","E","S",".",".",".","."
,"N","N","W","N","N","N",".",".",".",".","."
            },{

        ".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","E","S",".",".",".","."
,".","N","W","N","N","N",".",".",".",".","."
            },{
```

```
		".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S",".",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
			},{
		".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S",".",".",".","."
,".",".",".","B","B","B",".",".",".",".","."
			},{
		".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S",".",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
			},{
		".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".
",".",".",".",".",".",".",".",".",".",".",".",".",".",".",".","S",".",".",".","."
,".",".",".","N","N","N",".",".",".",".","."
			}
		};

		public static String[][] getMap(){
			return map;
		}
}
```

7. TrafficReport.java

```
package system;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Set;

import agent.AgentFactory.AgentPath;
import agent.CityAgent;

public class TrafficReport {

	public static void createCSV(HashMap<String, CityAgent> agentsAtHome,
HashMap<String, AgentPath> map) {

		PrintWriter pw;
		try {
			String filename = "/traficReport";
			File file = new File(System.getProperty("user.home") + filename +
".csv");
			int count = 1;
			while (file.exists()){
				file = new File(System.getProperty("user.home") + filename
+ count + ".csv");
				count++;
			}

			pw = new PrintWriter(file);


			StringBuilder sb = new StringBuilder();
			sb.append("Rota Do Agente");
			sb.append(',');
			sb.append("Nome");
			sb.append(',');
			sb.append("Instante Inicial");
			sb.append(',');
			sb.append("Instante Final");
			sb.append('\n');

			Set<String> names = agentsAtHome.keySet();
			for (String n : names) {
				CityAgent cityAgent = agentsAtHome.get(n);

				int startTime = cityAgent.getStartTime();
				int finishTime = cityAgent.getHomeTime();
				String agentRoute = getAgentPath(map, n);
```

```
                        sb.append(agentRoute);
                        sb.append(',');
                        sb.append(n);
                        sb.append(',');
                        sb.append(startTime);
                        sb.append(',');
                        sb.append(finishTime);
                        sb.append('\n');
                }

                pw.write(sb.toString());
                pw.close();
                System.out.println("wrote report at " + file.getAbsolutePath());

        } catch (FileNotFoundException e) {
                e.printStackTrace();
        }
    }

    public static String getAgentPath(HashMap<String, AgentPath> map, String name)
{
        AgentPath agentPath = map.get(name);
        if (agentPath != null) {
                return agentPath.name();
        }
        return "not found";
    }

}
```

8. TrafficClock.java

```
package system;

import java.util.HashMap;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;

import configuration.TrafficConfiguration;

public class TrafficClock implements Runnable{


	private static Logger logger = Logger.getLogger(TrafficClock.class.getName());

	private static final int SYSTEM_SPEED = 300;
	public static final long AGENT_WAIT_TIME = 100;

	private static final int NUMBER_OF_WAIT_BEFORE_PRINT = 20000/SYSTEM_SPEED; //
print if it has waited 20 seconds.

	public static AtomicInteger COUNTER = new AtomicInteger(0);

	private static HashMap<String, Integer> agentsCount = new HashMap<>();

	private static AtomicInteger numberOfExecution = new AtomicInteger(0);

	private static TrafficClock clock = new TrafficClock();

	@Override
	public void run() {
		while (true) {
			int sleepCounter = 0;

			boolean sleep = true;
			while (sleep) {
				try {
					Thread.sleep(SYSTEM_SPEED);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				sleepCounter++;

				sleep = false;
				synchronized (COUNTER) {
					for (String agentName : agentsCount.keySet()) {
						int count = agentsCount.get(agentName);
						int globalCount = getCounter();

						if (count != globalCount) {
							if (sleepCounter ==
NUMBER_OF_WAIT_BEFORE_PRINT) {
```

```
                                                        logger.info(agentName + " has not
executed yet. Its count is " + count + " globalCount is " + globalCount);
                                                        count = 0;
                                                }
                                                sleep = true;
                                        }
                                }
                        }

                }
                incCounter();

                synchronized (TrafficConfiguration.RUN) {
                        if (TrafficConfiguration.IS_EVERYONE_AT_HOME){
                                break;
                        }
                }
        }
        logger.info("All agents finished execution. Traffic Clock stop at " +
getCounter());
    }

    private void incCounter() {
        synchronized (COUNTER) {
                logger.info("AgentsOnTheSystem: " + agentsCount.size());
                logger.info("NumberOfExecution " + numberOfExecution.get());
                numberOfExecution = new AtomicInteger(0);
                COUNTER.incrementAndGet();
                logger.info(" ========== ========== tick: " + COUNTER + "
========== ========== ");
                COUNTER.notifyAll();
        }
    }

    public static void waitForCount(String name) {
        clock.waitForCounter(name);
    }

    private void waitForCounter(String name) {
        synchronized (COUNTER) {
                Integer count = agentsCount.get(name);

                while (count >= getCounter()) {
                        try {
                                COUNTER.wait();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                increamentAgentCount(name);
        }
    }
```

```
	public static int getCounter() {
		return COUNTER.get();
	}

	public static void increamentAgentCount(String name) {
		int totalExecuted = numberOfExecution.incrementAndGet();
		logger.info(name + " numberOfExecution: " + totalExecuted);
		agentsCount.put(name, getCounter());
	}

	public static Integer getAgentCounter(String name) {
		return agentsCount.get(name);
	}

	public static void removeAgentClock(String name) {
		synchronized (COUNTER) {
			agentsCount.remove(name);
		}
	}

	public static void startCounter() {
		new Thread(new TrafficClock()).start();
	}

}
```

9. SemaphoreFactory.java

```
package semaphore;

import java.util.logging.Logger;

import configuration.TrafficConfiguration;
import jason.environment.grid.Location;
import system.TrafficClock;
import traffic.CityModel;

public class SemaphoreFactory {

	private static final int FULL_FREQUENCY = 10;
	private static final SemaphoreFactory instance = new SemaphoreFactory();

	private static Logger logger =
Logger.getLogger(SemaphoreFactory.class.getName());

	public static void createSemaphores() {
		instance.start();
	}

	private void start() {
		if (TrafficConfiguration.HAS_SEMAPHORE){
			logger.info("start SemaphoreFactory");
			for (SemaphorePosition semaphore : SemaphorePosition.values()) {

				new Thread() {
					@Override
					public void run() {
						logger.info("start " + semaphore);
						setInitLocationOnCityModel(semaphore);

						while (true) {

	TrafficClock.increamentAgentCount(semaphore.name());
							int semaphoreOld =
semaphore.getState();

							int semaphoreNew = semaphore.tick();

							Location[] locations =
semaphore.getLocations();
							for (Location location : locations) {

	CityModel.updateSemaphore(semaphoreOld, semaphoreNew, location);
							}
						}

					}

				}.start();
```

```
                }
            }
        }

        private void setInitLocationOnCityModel(SemaphorePosition semaphore) {
            Location[] locations = semaphore.getLocations();
            for (Location location : locations) {
                CityModel.updateSemaphore(semaphore.getState(),
semaphore.getState(), location);
            }
        }

        enum SemaphorePosition {

            NORTH_SOUTH_1(CityModel.SEMAPHORO_GREEN, new Location[] { new
Location(35, 10), new Location(36, 10), new Location(37, 10) }, 5, 4, 1),
            NORTH_SOUTH_2(CityModel.SEMAPHORO_GREEN, new Location[] { new
Location(35, 20), new Location(36, 20) }, 3, 6, 1),
            //
            SOUTH_NORTH_1(CityModel.SEMAPHORO_GREEN, new Location[] { new
Location(44, 20), new Location(45, 20), new Location(46, 20) }, 4, 5, 1),
            SOUTH_NORTH_2(CityModel.SEMAPHORO_GREEN, new Location[] { new
Location(44, 30), new Location(45, 30), new Location(46, 30) }, 3, 6, 1),
            //
            EAST_WEST_1(CityModel.SEMAPHORO_RED, new Location[] { new Location(38,
11), new Location(38, 12) }, 7, 2, 1),
            EAST_WEST_2(CityModel.SEMAPHORO_RED, new Location[] { new Location(37,
21), new Location(37, 22) }, 7, 2, 1),
            //
            WEST_EAST_1(CityModel.SEMAPHORO_RED, new Location[] { new Location(47,
15), new Location(47, 16) }, 6, 3, 1),
            WEST_EAST_2(CityModel.SEMAPHORO_RED, new Location[] { new Location(43,
19) }, 6, 3, 1),
            WEST_EAST_3(CityModel.SEMAPHORO_RED, new Location[] { new Location(34,
24), new Location(34, 25) }, 7, 2, 1),;

            private Location[] locations;
            private int state;
            private int redLightTimer;
            private int yellowLightTimer;
            private int greenLightTimer;
            private boolean firstTime = true;

            private SemaphorePosition(int state, Location[] locations, int
redLightTimer, int greenLightTimer, int yellowLightTimer) {
                this.state = state;
                this.locations = locations;
                this.redLightTimer = redLightTimer;
                this.greenLightTimer = greenLightTimer;
                this.yellowLightTimer = yellowLightTimer;
            }

            public Location[] getLocations() {
```

```
		return locations;
	}

	public int getRedLightTimer() {
		if (firstTime && this == EAST_WEST_1) {
			firstTime = false;
			return 5;
		}

		return redLightTimer;
	}

	public int getYellowLightTimer() {
		return yellowLightTimer;
	}

	public int getGreenLightTimer() {
		if (firstTime && this == SOUTH_NORTH_2) {
			firstTime = false;
			return 4;
		}

		return greenLightTimer;
	}

	public int getState() {
		return state;
	}

	public int tick() {
		int miliseconds = 0;
		switch (state) {
		case CityModel.SEMAPHORO_RED:
			state = CityModel.SEMAPHORO_GREEN;
			miliseconds = getRedLightTimer();
			break;
		case CityModel.SEMAPHORO_GREEN:
			state = CityModel.SEMAPHORO_YELLOW;
			miliseconds = getGreenLightTimer();
			break;
		case CityModel.SEMAPHORO_YELLOW:
			state = CityModel.SEMAPHORO_RED;
			miliseconds = getYellowLightTimer();
		}

		double waitTime = FULL_FREQUENCY * miliseconds * 0.1;
		int waitTime2 = (int) waitTime;

		String name = this.toString();
		for (int count = 0; count <= waitTime2; count++){
			TrafficClock.waitForCount(name);
		}
```

```
                    logger.info("[" + this + "] waitTime: " + waitTime2 + " for state
" + state);
                    return state;
                }

            }

}
```

10. TrafficConfiguration.java

```
package configuration;

import agent.AgentFactory.AgentPath;

public class TrafficConfiguration {

	public static final int NUMBER_OF_AGENTS = 10;

	public static final boolean HAS_SEMAPHORE = false;

	 public static final AgentPath[] AGENTS_AVAILABLE = null;
	// public static final AgentPath[] AGENTS_AVAILABLE = { AgentPath.NORTH_EAST,
AgentPath.SOUTH_NORTH };
	// public static final AgentPath[] AGENTS_AVAILABLE = { AgentPath.NORTH_EAST,
AgentPath.NORTH_NORTH, AgentPath.NORTH_SOUTH, AgentPath.NORTH_WEST };

//	public static final AgentPath[] AGENTS_AVAILABLE = { AgentPath.NORTH_SOUTH };

	public static final boolean USE_ID_ON_AGENT = false ;

	public static boolean IS_EVERYONE_AT_HOME = false;

	public static Object RUN = new Object();

}
```

11. CityAgent.java

```
package agent;

import jason.environment.grid.Location;

public class CityAgent {

	private String name;
	private int id;
	private Location[] homes;
	private Location spawn;
	private int startTime;
	private int homeTime;

	public CityAgent(int id, Location spawn, Location homes[]) {
		this.name = "ricardo"+(id+1);
		this.id = id;
		this.spawn = spawn;
		this.homes = homes;
	}

	public int getId() {
		return id;
	}

	public Location[] getHomes() {
		return homes;
	}

	public Location getSpawn() {
		return spawn;
	}

	public String getName() {
		return name;
	}

	public void setStartTime(int startTime) {
		this.startTime = startTime;
	}

	public int getStartTime(){
		return this.startTime;
	}

	public void setHomeTime(int homeTime){
		this.homeTime = homeTime;
	}

	public int getHomeTime(){
		return this.homeTime;
```

```
        }
    }
```

12. AgentFactory.java

```
package agent;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;
import java.util.stream.Collectors;

import configuration.TrafficConfiguration;
import jason.environment.grid.Location;
import system.TrafficClock;
import system.TrafficReport;
import traffic.CityModel;

public class AgentFactory {

	static Location[] southHomes = { new Location(36, 33)};
	static Location[] westHomes = { new Location(0, 21), new Location(0, 22) };
	static Location[] eastHomes = { new Location(51, 19) };
	static Location[] northHomes = { new Location(44, 0), new Location(45, 0), new
Location(46, 0) };

	static Location[] northSpawn = { new Location(35, 0), new Location(36, 0), new
Location(37, 0) };

	static Location[] northSpawn2 = { new Location(35, 0), new Location(36, 0) };
	// static Location[] northSpawn3 = { new Location(41, 19) };

	static Location[] southSpawn = { new Location(44, 33), new Location(45, 33),
new Location(46, 33) };

	// static Location[] southSpawn2 = { new Location(44, 33) };
	// static Location[] southSpawn3 = { new Location(44, 25) };

	static Location[] westSpawn = { new Location(0, 24), new Location(0, 23) };
	static Location[] eastSpawn = { new Location(51, 17), new Location(51, 16) };

	private static AgentFactory factory = new AgentFactory();

	private static final Logger logger =
Logger.getLogger(AgentFactory.class.getName());

	public static Object AGENT_FACTORY_MUTEX = new Object();

	private static AtomicInteger agentIdCounter = new AtomicInteger(0);
```

```
	private static final HashMap<String, AgentPath> map = new HashMap<>();

	public enum AgentPath {
		NORTH_SOUTH(15, 4, northSpawn, southHomes),
		WEST_SOUTH(15, 4, westSpawn, southHomes),

		NORTH_EAST(12, 4, northSpawn, eastHomes),
		SOUTH_EAST(12, 4, southSpawn, eastHomes),
		EAST_SOUTH(12, 4, eastSpawn, southHomes),

		SOUTH_NORTH(10, 6, southSpawn, northHomes),
		EAST_NORTH(10, 6, eastSpawn, northHomes),

		NORTH_WEST(4, 6, northSpawn, westHomes),
		SOUTH_WEST(4, 6, southSpawn, westHomes),
		EAST_WEST(4, 6, eastSpawn, westHomes),

		NORTH_NORTH(1, 10, northSpawn, northHomes),
		SOUTH_SOUTH(1, 10, southSpawn, southHomes),;

		// private AtomicInteger homeCounter = new AtomicInteger(0);
		// private AtomicInteger spawnCounter = new AtomicInteger(0);
		private Random random = new Random();
		private int percentage;
		private int frequency;
		private Location[] homes;
		private Location[] spawn;
		private List<CityAgent> agentsToEnter = new ArrayList<>();
		private HashMap<String, CityAgent> agentsInTheSystem = new HashMap<>();
		private HashMap<String, CityAgent> agentsAtHome = new HashMap<>();
		private boolean isAlive = true;

		private AgentPath(int percentage, int frequency, Location[] spawn,
Location[] homes) {
			this.percentage = percentage;
			this.frequency = frequency;
			this.homes = homes;
			this.spawn = spawn;

		}

		public CityAgent getFirstAgent() {
			CityAgent agent = null;
			synchronized (AGENT_FACTORY_MUTEX) {
				if (agentsToEnter.size() > 0) {
					agent = agentsToEnter.get(0);
				}
			}
			return agent;
		}

		public void addAgent(CityAgent agent) {
			synchronized (AGENT_FACTORY_MUTEX) {
```

```
                agentsToEnter.add(agent);
        }
}

public void removeFirst() {
        synchronized (AGENT_FACTORY_MUTEX) {
                agentsToEnter.remove(0);
        }
}

// public Location getHome() {
//// int pos = homeCounter.getAndIncrement() % homes.length;
// int pos = random.nextInt(homes.length);
//
// return homes[pos];
// }

public Location getSpawn() {
        // int pos = spawnCounter.getAndIncrement() % spawn.length;
        int pos = random.nextInt(spawn.length);
        return spawn[pos];
}

public void addAgentInTheSystem(CityAgent agent) {
        synchronized (AGENT_FACTORY_MUTEX) {
                agentsInTheSystem.put(agent.getName(), agent);
        }
}

public CityAgent getAgentInTheSystem(String name) {
        synchronized (AGENT_FACTORY_MUTEX) {
                return agentsInTheSystem.get(name);
        }
}

public void removeAgentInTheSystem(String name) {
        synchronized (AGENT_FACTORY_MUTEX) {
                CityAgent removedAgent = agentsInTheSystem.remove(name);
                removedAgent.setHomeTime(TrafficClock.getCounter());
                agentsAtHome.put(name, removedAgent);
        }
}

public boolean isAlive() {
        synchronized (AGENT_FACTORY_MUTEX) {
                return isAlive;
        }
}

public void setNotAlive() {
        synchronized (AGENT_FACTORY_MUTEX) {
                isAlive = false;
        }
```

```
		}

		public int getTotalOfAgents() {
			return (TrafficConfiguration.NUMBER_OF_AGENTS * percentage) /
100;
		}

	}

	public static void createAgents() {
		factory.create();
	}

	private Set<AgentPath> getAgentPathAvailable() {
		AgentPath[] available = TrafficConfiguration.AGENTS_AVAILABLE;
		if (available == null || available.length == 0) {
			available = AgentPath.values();
		}
		Set<AgentPath> values =
Arrays.stream(available).collect(Collectors.toSet());
		return values;
	}

	private void create() {
		assert (getTotalOfAgents() <= TrafficConfiguration.NUMBER_OF_AGENTS);

		for (AgentPath agentPath : getAgentPathAvailable()) {

			new Thread() {
				@Override
				public void run() {
					String name = "AgentFactoryCreate-" + agentPath;
					logger.info("AgentFactory.AgentPath.start() " +
agentPath + " " + agentIdCounter);

					TrafficClock.increamentAgentCount(name);
					int numberOfAgentsInThisPath =
agentPath.getTotalOfAgents();
					synchronized (TrafficClock.COUNTER) {
						for (int i = 0; i < numberOfAgentsInThisPath;
i++) {

							TrafficClock.waitForCount(name);

							int counter =
TrafficClock.getCounter();
							logger.info(agentPath + " counter: " +
counter);
							logger.info(agentPath + "
agentPath.frequency: " + agentPath.frequency);

							if (counter % agentPath.frequency == 0)
{
```

```
                                                synchronized
(AGENT_FACTORY_MUTEX) {
                                                      Location spawn =
agentPath.getSpawn();
                                                      CityAgent cityAgent = new
CityAgent(getNextAgentId(), spawn, agentPath.homes);

      agentPath.addAgent(cityAgent);
                                                }
                                          } else {
                                                i--;
                                          }
                                    }
                                    agentPath.setNotAlive();
                                    TrafficClock.removeAgentClock(name);
                              }
                        }
                  }.start();

                  new Thread() {
                        public void run() {
                              while (getNumberOfAgentsAtHome() <
TrafficConfiguration.NUMBER_OF_AGENTS) {

                                    boolean ok = false;
                                    Location spawn = null;
                                    synchronized (AGENT_FACTORY_MUTEX) {
                                          CityAgent firstAgent =
agentPath.getFirstAgent();

                                          if (firstAgent != null) {
                                                spawn = firstAgent.getSpawn();
                                                ok =
CityModel.addAgent(firstAgent.getId(), spawn);
                                                if (ok) {
                                                      logger.info("Added agent "
+ firstAgent.getName() + " in the system at " + firstAgent.getSpawn());

      TrafficClock.increamentAgentCount(firstAgent.getName());

      firstAgent.setStartTime(TrafficClock.getAgentCounter(firstAgent.getName()));
                                                      agentPath.removeFirst();

      agentPath.addAgentInTheSystem(firstAgent);

      factory.addMap(firstAgent.getName(), agentPath);
                                                }
                                          }
                                    }

                                    if (!ok && spawn != null) {
                                          CityModel.waitLocation(spawn);
                                    }
```

```
                                }
                        };
                }.start();
        }
    }

    protected int getNextAgentId() {
        int id = agentIdCounter.getAndIncrement();
        return id;
    }

    public static CityAgent getAgentByName(String name) {
        synchronized (AGENT_FACTORY_MUTEX) {
            AgentPath agentPath = factory.getMap(name);
            if (agentPath != null) {

                CityAgent agent = agentPath.getAgentInTheSystem(name);
                if (agent != null)
                    return agent;
            }
            return null;
        }
    }

    public static void removeAgentInTheSystem(String name) {
        synchronized (AGENT_FACTORY_MUTEX) {
            logger.info("Removing agent " + name + " from the system");
            AgentPath agentPath = factory.getMap(name);

            if (agentPath != null) {
                CityAgent agent = agentPath.getAgentInTheSystem(name);
                if (agent != null) {
                    agentPath.removeAgentInTheSystem(name);
                }
            }

            TrafficClock.removeAgentClock(name);

            synchronized (TrafficConfiguration.RUN) {
                if (AgentFactory.getNumberOfAgentsAtHome() ==
getTotalOfAgents()) {
                    TrafficConfiguration.IS_EVERYONE_AT_HOME = true;


    TrafficReport.createCSV(AgentFactory.getAgentsAtHome(), map);
                }
            }
        }
    }

    private static int getTotalOfAgents() {
        int n = 0;
        for (AgentPath agentPath : factory.getAgentPathAvailable()) {
```

```
                n += agentPath.getTotalOfAgents();
        }
        return n;
}

public static String getAgentDestination(int id) {
        synchronized (AGENT_FACTORY_MUTEX) {
                String name = "ricardo" + (id + 1);
                AgentPath agentPath = factory.getMap(name);
                if (agentPath != null) {
                        CityAgent agent = agentPath.getAgentInTheSystem(name);
                        if (agent != null) {
                                // System.out.println("==========================
                                // AgentFactory.getAgentDestination()");
                                String s = agentPath.toString();
                                return s.substring(s.indexOf("_") + 1, s.length());
                        }
                }
                return null;
        }
}

public void addMap(String name, AgentPath path) {
        synchronized (AGENT_FACTORY_MUTEX) {
                map.put(name, path);
        }
}

public AgentPath getMap(String name) {
        synchronized (AGENT_FACTORY_MUTEX) {
                return map.get(name);
        }
}

public static int getNumberOfAgentsAtHome() {
        synchronized (AGENT_FACTORY_MUTEX) {
                int n = 0;
                for (AgentPath agentPath : factory.getAgentPathAvailable()) {
                        n += agentPath.agentsAtHome.size();
                }
                return n;
        }
}

public static int getNumberOfAgentsOnTheSystem() {
        synchronized (AGENT_FACTORY_MUTEX) {
                int n = 0;
                for (AgentPath agentPath : factory.getAgentPathAvailable()) {
                        n += agentPath.agentsInTheSystem.size();
                }
                return n;
        }
}
```

```
	public static int howManySpammerIsAlive() {
		synchronized (AGENT_FACTORY_MUTEX) {
			Set<AgentPath> agentPathAvailable =
factory.getAgentPathAvailable();

			int alive = 0;
			for (AgentPath agentPath : agentPathAvailable) {
				if (agentPath.isAlive()) {
					alive++;
				}
			}
			return alive;
		}
	}

	public static Set<String> getRunningAgents() {
		synchronized (AGENT_FACTORY_MUTEX) {
			Set<String> agentsAlive = new HashSet<>();

			for (AgentPath agentPath : factory.getAgentPathAvailable()) {
				Set<String> agentsInTheSystem =
agentPath.agentsInTheSystem.keySet();

				for (String agentName : agentsInTheSystem) {
					agentsAlive.add(agentName);
				}

				String name = "AgentFactoryCreate-" + agentPath;
				if (agentPath.isAlive()) {
					agentsAlive.add(name);
				}
			}

			return agentsAlive;
		}

	}

	public static HashMap<String, CityAgent> getAgentsAtHome() {
		HashMap<String, CityAgent> agentsAtHome = new HashMap<>();
		for (AgentPath agentPath : factory.getAgentPathAvailable()) {
			agentsAtHome.putAll(agentPath.agentsAtHome);
		}
		return agentsAtHome;
	}
}
```

13. driver.asl

```
// Agent driver in project traffic

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start
	<-	!where_is_home;
		!going_home.


+!where_is_home : .my_name(N) & jia.whereIsMyHome(N,X,Y)
	<-	.print(N, " home is at (",X,",", Y,")");
		+home(X,Y);
		!where_is_home.

+!where_is_home : not home(X,Y) & .my_name(N)
	<-	!where_is_home.

+!where_is_home.

+!going_home
	<-	!where_is_my_position;
		!go_home.

+!where_is_my_position : .my_name(N) & jia.whereami(N,X,Y)
	<-	-+pos(X,Y).

+!go_home : home(X,Y) & pos(X,Y) & .my_name(N)
	<- .print("I'm home");
		home(N);
		-home(X,Y);
		-pos(X,Y).

+!go_home : home(X,Y) & pos(AgX, AgY)
	<-	!wait_timer;
		jia.getDirection(AgX, AgY, X, Y, A, A, DirX, DirY);
		.print("Estou na posicao (",AgX,",",AgY,") e quero ir para
(",X,",",Y,"). Devo me movimentar para (",DirX,", ",DirY,")");
		!move(DirX, DirY);
		!go_home.

+!wait_timer : .my_name(N) & jia.waitTimer(N)
	<- .print("can run").

+!wait_timer
```

```
	<- .print("wait..");
		!wait_timer.

+!move(DirX, DirY) : buraco(DirX,DirY) & pos(AgX, AgY) & home(HomeX, HomeY) &
.my_name(N)
	<-
		.print("Minha posicao (",AgX,", ",AgY,")");
		.print("Buraco em (",DirX,", ",DirY,")!!!");
		jia.getDirection(AgX, AgY, HomeX, HomeY, DirX, DirY, NewDirX, NewDirY);
		skip(N);
		.


+!move(DirX, DirY) : pos(AgX, AgY) & semaphoreRed(AgX,AgY) & .my_name(N)
	<- .print("Sinal Vermelho...");
		.print("Minha posicao (",AgX,", ",AgY,")");
		.print("Quero me mover para (",DirX,", ",DirY,")");
		skip(N);
		.

+!move(DirX, DirY) : pos(AgX, AgY) & semaphoreYellow(AgX,AgY) & .my_name(N)
	<- .print("Sinal Amarelo...");
		.print("Quero me mover para (",DirX,", ",DirY,")");
		skip(N);
		.

+!move(DirX, DirY) : pos(AgX, AgY) & semaphoreGreen(AgX,AgY) & .my_name(N)
	<- .print("Sinal Verde...");
		.print("Quero me mover para (",DirX,", ",DirY,")");
		move(N, DirX, DirY);
		!where_is_my_position.

+!move(DirX, DirY) : pos(AgX, AgY) & .my_name(N)
	<-  .print("Minha posicao (",AgX,", ",AgY,")");
		.print("Quero me mover para (",DirX,", ",DirY,")");
		move(N, DirX, DirY);
		!where_is_my_position.

+cell(X,Y,semaphoroRed) <-
	-semaphoreGreen(X,Y);
	-semaphoreYellow(X,Y);
	+semaphoreRed(X,Y).
+cell(X,Y,semaphoroGreen) <-
	-semaphoreYellow(X,Y);
	-semaphoreRed(X,Y);
	+semaphoreGreen(X,Y).
+cell(X,Y,semaphoroYellow) <-
	-semaphoreGreen(X,Y);
	-semaphoreRed(X,Y);
	+semaphoreYellow(X,Y).

+cell(X,Y,buraco)
	<- .print("percebi buraco em ", X, ", ", Y, ")");
```

```
            +buraco(X,Y).

{ include("$jacamoJar/templates/common-cartago.asl") }
{ include("$jacamoJar/templates/common-moise.asl") }
{ include("$jacamoJar/templates/org-obedient.asl") }
```

14. cooperative-driver.asl

```
// Agent driver in project traffic
/*
 * This file was developed using JaCaMo eclipse plugin available at
http://jacamo.sourceforge.net/eclipseplugin/tutorial/
 * This plugin will validate the Jason syntax in this file.
 *
 */

/* Initial beliefs and rules */

/* Initial goals */

!start.

/* Plans */

+!start
	<-	!where_is_home;
		!where_is_my_position;
		!going_home.


+!where_is_home : .my_name(N) & jia.whereIsMyHome(N,X,Y)
	<-	.print(N, " home is at (",X,",", Y,")");
		+home(X,Y);
		!where_is_home.

+!where_is_home : not home(X,Y) & .my_name(N)
	<-	!where_is_home.

+!where_is_home.

+!going_home : home(X,Y) & pos(AgX, AgY) & .my_name(N)
	<-	jia.createPlan(N, AgX, AgY, X, Y);
		!update_global_plan;
		!go_home.

+!update_global_plan
	<-	.print("tell others agents to update global plan");
		.broadcast(tell,updatedGlobalPlan).

+!where_is_my_position : .my_name(N) & jia.whereami(N,X,Y)
	<-	-+pos(X,Y).

+!go_home : home(X,Y) & pos(X,Y) & .my_name(N)
	<- .print("I'm home");
		home(N);
		-home(X,Y);
		-pos(X,Y).
```

```
+!go_home : pos(AgX, AgY) & .my_name(N)
      <-    !wait_timer;
            jia.getDirectionFromPlan(N, AgX, AgY, DirX, DirY);
            .print("Estou na posicao (",AgX,",",AgY,"). Devo me movimentar para
(",DirX,", ",DirY,")");
            !wait_car(DirX, DirY);
            jia.getDirectionFromPlan(N, AgX, AgY, NewDirX, NewDirY);
            !move(NewDirX, NewDirY);
            !go_home.

+!wait_timer : .my_name(N) & jia.waitTimer(N)
      <- .print("can run").

+!wait_timer
      <- .print("wait..");
            !wait_timer.

+!wait_car(DirX, DirY) : .my_name(N) & jia.canMove(N, DirX, DirY)
      <- .print("can move").

+!wait_car(DirX, DirY) : pos(AgX, AgY) & .my_name(N)
      <-    .print("cannot move to (",DirX,", ",DirY,")");
      .wait(50);
      jia.getDirectionFromPlan(N, AgX, AgY, NewDirX, NewDirY);
      !wait_car(NewDirX, NewDirY).

+!move(DirX, DirY) : pos(DirX, DirY) & .my_name(N)
      <- .print("Ficar parado.");
            skip(N).

+!move(DirX, DirY) : buraco(DirX,DirY) & pos(AgX, AgY) & home(HomeX, HomeY) &
.my_name(N)
      <-
            .print("Minha posicao (",AgX,", ",AgY,")");
            .print("Buraco em (",DirX,", ",DirY,")!!!");
            jia.changePlan(N, AgX, AgY, DirX,DirY, HomeX, HomeY);
            !update_global_plan.

+!move(DirX, DirY) : pos(AgX, AgY) & .my_name(N)
      <-  .print("Minha posicao (",AgX,", ",AgY,")");
            .print("Quero me mover para (",DirX,", ",DirY,")");
            move(N, DirX, DirY);
            !where_is_my_position.

+updatedGlobalPlan[source(S)] : .my_name(N) & home(_,_)
      <- .print(S, " atualizou o plano global. Preciso verificar meu plano.");
            jia.updateMyPlan(N).

+updatedGlobalPlan[source(S)]
      <- .print("Ainda nÃ£o entrei no sistema. Do nothing").
```

```
+cell(X,Y,buraco)
       <- .print("percebi buraco em ", X, ", ", Y, ")");
             +buraco(X,Y).

{ include("$jacamoJar/templates/common-cartago.asl") }
{ include("$jacamoJar/templates/common-moise.asl") }
{ include("$jacamoJar/templates/org-obedient.asl") }
```

15. GlobalPlanManager.java

```
package plan;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.logging.Logger;

import jason.environment.grid.Location;
import system.TrafficClock;

public class GlobalPlanManager {

	private static Logger logger =
Logger.getLogger(GlobalPlanManager.class.getName());
	private static GlobalPlanManager manager = new GlobalPlanManager();
	private static HashMap<String, PrivatePlan> privatePlansMap = new HashMap<>();
	private static Map<GlobalPlanMapKey, String> globalPlansMap = new HashMap<>();

	GlobalPlanManager() {
	}

	public static PrivatePlan add(String name, List<Location> oldPrivatePlan) {
		return manager.addPlan(name, oldPrivatePlan);
	}

	public static PrivatePlan add(String name, List<Location> oldPrivatePlan, int
clock) {
		return manager.addPlan(name, oldPrivatePlan, clock);
	}

	public PrivatePlan addPlan(String name, List<Location> oldPrivatePlan) {
		synchronized (TrafficClock.COUNTER) {
			return addPlan(name, oldPrivatePlan, TrafficClock.getCounter());
		}
	}

	public void updateOthersAgentPlan(Location l){
		synchronized (TrafficClock.COUNTER) {



		}
	}

	public PrivatePlan addPlan(String name, List<Location> calculetePath, int
clock) {
```

```
			synchronized (TrafficClock.COUNTER) {
				try {
					logger.info("Adding " + name + " private plan on the
GlobalPlan");
					int instant = clock;
					List<GlobalPlanMapKey> keys = getKeys(globalPlansMap,
name);
					removeKeys(globalPlansMap, keys);

					Map<GlobalPlanMapKey, String> globalPlanTemp = new
HashMap<>();
					HashMap<Integer, Location> privatePlan = new HashMap<>();
					HashSet<String> workingAgent = new HashSet<>();

					workingAgent.add(name);
					interateForNextLocations(name, calculetePath, instant,
globalPlanTemp, privatePlan, workingAgent, false);

					globalPlansMap.putAll(globalPlanTemp);

					PrivatePlan p = null;
					PrivatePlan oldPrivatePlan = privatePlansMap.get(name);
					if (oldPrivatePlan != null){
						p = new PrivatePlan(privatePlan,
oldPrivatePlan.getVersion() + 1);
					} else {
						p = new PrivatePlan(privatePlan);
					}
					privatePlansMap.put(name, p);

					checkGlobalPlan();

					PrivatePlan clone = p.clone();

					if (clone == null){
						new Object();
					}
					return clone;
				} catch (Throwable t) {
					for (StackTraceElement stackTraceElement :
t.getStackTrace()) {
						logger.info(stackTraceElement.toString());
					}
					t.printStackTrace();
				}
			}
			return null;

		}

		private void interateForNextLocations(String name, List<Location>
calculetePath, int instant, Map<GlobalPlanMapKey, String> globalPlanTemp,
```

```
                    HashMap<Integer, Location> privatePlan, HashSet<String>
workingAgent, boolean waitLastInstant) {
             for (int position = 0; position < calculetePath.size(); position++) {

                    Location oldPlanLocation = calculetePath.get(position);
                    GlobalPlanMapKey key = new GlobalPlanMapKey(oldPlanLocation,
instant);

                    String agentNameAtLocation = getAgentAtLocation(globalPlanTemp,
key);

                    if (agentNameAtLocation != null) {

                           boolean canChangeAgent = canAgentChangeHisPlan(name,
instant, globalPlanTemp, agentNameAtLocation);
                           boolean isWorking =
workingAgent.contains(agentNameAtLocation);

                           if (canChangeAgent && waitLastInstant && !isWorking) {
                                  globalPlanTemp.put(key, name);
                                  privatePlan.put(instant, oldPlanLocation);

                                  logger.info(name + ": Changing plan of agent " +
agentNameAtLocation + " at location (" + key.getLocation() + ") in the instant " +
key.getCounter());
                                  changeOtherAgentPlan(agentNameAtLocation,
key.getLocation(), key.getCounter(), globalPlanTemp, workingAgent);

                                  waitLastInstant = false;
                           } else {
                                  position--;
                                  Location previewPlanLocation =
calculetePath.get(position);
                                  GlobalPlanMapKey previewKey = new
GlobalPlanMapKey(previewPlanLocation, instant);

                                  agentNameAtLocation =
getAgentAtLocation(globalPlanTemp, previewKey);

                                  privatePlan.put(instant, previewPlanLocation);
                                  globalPlanTemp.put(previewKey, name);

                                  if (agentNameAtLocation != null) {
                                         boolean c = canAgentChangeHisPlan(name,
instant, globalPlanTemp, agentNameAtLocation);
                                         assertTrue(c);


      assertTrue(!workingAgent.contains(agentNameAtLocation));

                                  logger.info(name + ": Changing plan of agent "
+ agentNameAtLocation + " at location (" + previewKey.getLocation() + ") in the
instant " + previewKey.getCounter());
```

```
                                        changeOtherAgentPlan(agentNameAtLocation,
previewKey.getLocation(), previewKey.getCounter(), globalPlanTemp, workingAgent);
                                }
                                waitLastInstant = true;
                        }
                } else {
                        globalPlanTemp.put(key, name);
                        privatePlan.put(instant, oldPlanLocation);
                        waitLastInstant = false;
                }

                instant++;

                globalPlansMap.putAll(globalPlanTemp);
                globalPlanTemp = new HashMap<>();
        }
    }

    private boolean canAgentChangeHisPlan(String name, int instant,
Map<GlobalPlanMapKey, String> globalPlanTemp, String agentNameAtLocation) {
        PrivatePlan privatePlanFromLocation =
privatePlansMap.get(agentNameAtLocation);
        Location locationBeforeConflict =
privatePlanFromLocation.getLocationAt(instant-1);
        GlobalPlanMapKey beforeConflictKey = new
GlobalPlanMapKey(locationBeforeConflict, instant);
        String agentAtLocationBeforeConflict =
getAgentAtLocation(globalPlanTemp, beforeConflictKey);

        if (agentAtLocationBeforeConflict != null){
                if (agentAtLocationBeforeConflict.equals(agentNameAtLocation)){
                        // o agente na localizÃ§Ã£o ficou parado
                        return false;
                }
                if (agentAtLocationBeforeConflict.equals(name)){
                        // eu estou na localizaÃ§Ã£o anterior do agente que eu
quero alterar, nÃ£o tem como alterar este agente.
                        return false;
                }
        }

        return true;
    }

    private String getAgentAtLocation(Map<GlobalPlanMapKey, String>
globalPlanTemp, GlobalPlanMapKey key) {
        assertTrue(!(globalPlanTemp.get(key) != null && globalPlansMap.get(key)
!= null));
        String agentNameAtLocation = globalPlanTemp.get(key) != null ?
globalPlanTemp.get(key) : globalPlansMap.get(key);

        return agentNameAtLocation;
    }
```

```
	private void changeOtherAgentPlan(String name, Location conflictedLocation,
int conflictedInstant, Map<GlobalPlanMapKey, String> globalPlanTemp, HashSet<String>
workingAgent) {
//		Map<GlobalPlanMapKey, String> map = workingAgent.contains(name) ?
globalPlanTemp : globalPlansMap;

		List<GlobalPlanMapKey> keys = getKeys(globalPlansMap, name);
		removeKeys(globalPlansMap, keys);
		Collections.sort(keys);

		HashMap<Integer, Location> newPrivatePlan = new HashMap<>();
		workingAgent.add(name);

		// ========== add keys until conflict ================
		int pos = 0;
		GlobalPlanMapKey key = keys.get(pos);
		while (key.getCounter() < conflictedInstant) {
			globalPlanTemp.put(key, name);
			newPrivatePlan.put(key.getCounter(), key.getLocation());
			pos++;
			key = keys.get(pos);

			if (pos >= keys.size()){
				new Object();
			}
		}

		// ====================================================

		// ========== replicate location before conflict =================
		GlobalPlanMapKey keyBeforeConflict = keys.get(pos - 1);
		GlobalPlanMapKey conflictedKey = keys.get(pos);

		assertTrue(conflictedKey.getLocation().equals(conflictedLocation));
		// test
		GlobalPlanMapKey bla = new GlobalPlanMapKey(conflictedLocation,
conflictedInstant);
		String a = getAgentAtLocation(globalPlanTemp, bla);
		// <<<<

		GlobalPlanMapKey fixKey = new
GlobalPlanMapKey(keyBeforeConflict.getLocation(), conflictedInstant);

		String locatName = getAgentAtLocation(globalPlanTemp, fixKey);

		globalPlanTemp.put(fixKey, name);
		newPrivatePlan.put(fixKey.getCounter(), fixKey.getLocation());

		if (locatName != null) {
			logger.info(name + ": Changing plan of agent " + locatName + " at
location (" + fixKey.getLocation() + ") in the instant " + fixKey.getCounter());
```

```
                changeOtherAgentPlan(locatName, fixKey.getLocation(),
fixKey.getCounter(), globalPlanTemp, workingAgent);
            }
            // ================================================


            // ========== continue locations =================
            int instant = conflictedInstant + 1;
            List<Location> list = new ArrayList<>();

            Location previewLocation = fixKey.getLocation();
            for (int k = (pos); k < keys.size(); k++) {
                Location nextlocation = keys.get(k).getLocation();

                if (!nextlocation.equals(previewLocation)) {
                    // remove old duplicate.
                    list.add(nextlocation);
                }

                previewLocation = nextlocation;
            }

            interateForNextLocations(name, list, instant, globalPlanTemp,
newPrivatePlan, workingAgent, true);

            PrivatePlan oldPrivatePlan = privatePlansMap.get(name);
            PrivatePlan p = new PrivatePlan(newPrivatePlan,
oldPrivatePlan.getVersion() + 1);
            privatePlansMap.put(name, p);
            workingAgent.remove(name);
        }

        private List<GlobalPlanMapKey> getKeys(Map<GlobalPlanMapKey, String> map,
String name) {
            List<GlobalPlanMapKey> keys = new ArrayList<>();
            for (GlobalPlanMapKey key : map.keySet()) {
                String value = map.get(key);
                if (name.equals(value)) {
                    keys.add(key);
                }
            }

            return keys;
        }

        private void removeKeys(Map<GlobalPlanMapKey, String> map,
List<GlobalPlanMapKey> keys) {
            for (GlobalPlanMapKey k : keys) {
                map.remove(k);
            }
        }

        public static PrivatePlan getPrivatePlan(String name) {
```

```
		synchronized (TrafficClock.COUNTER) {
			try {
				PrivatePlan p = privatePlansMap.get(name);
				if (p != null){
					return p.clone();
				} else {
					return null;
				}
			} catch (CloneNotSupportedException e) {
				e.printStackTrace();
				return null;
			}
		}
	}

	public static Map<GlobalPlanMapKey, String> getGlobalPlan() {
		return globalPlansMap;
	}

	private void checkGlobalPlan() {
		Set<String> names = privatePlansMap.keySet();

		for (String n : names) {
			checkIfAgentIsTeleporting(n);
			checkIfAgentIsDisapering(n);
		}
	}

	private void checkIfAgentIsTeleporting(String name) {
		PrivatePlan privatePlan = GlobalPlanManager.getPrivatePlan(name);
		HashMap<Integer, Location> plan = privatePlan.getPlan();

		Set<Integer> keySet = plan.keySet();
		List<Integer> list = new ArrayList<>(keySet);
		Collections.sort(list);
		int initialClock = list.get(0);
		Location locationPreview = plan.get(initialClock);
		for (int i = initialClock + 1; i < plan.size(); i++) {
			Location nextLocation = plan.get(i);

			assertTrue(locationPreview.equals(nextLocation) ||
locationPreview.isNeigbour(nextLocation));

			locationPreview = nextLocation;
		}
	}

	private void checkIfAgentIsDisapering(String name) {
		PrivatePlan privatePlan = GlobalPlanManager.getPrivatePlan(name);
		HashMap<Integer, Location> plan = privatePlan.getPlan();

		Set<Integer> keySet = plan.keySet();
		List<Integer> list = new ArrayList<>(keySet);
```

```
			Collections.sort(list);

			int previewInstant = list.get(0);
			for (int i = 1; i < list.size(); i++) {
				int count = list.get(i);

				assertTrue((previewInstant + 1) == count);

				previewInstant = count;
			}
		}

		private void assertTrue(boolean t) {
			if (!t) {
				new Object();
			}
		}

		public static HashMap<String, PrivatePlan> getPrivatePlanMap() {
			synchronized (TrafficClock.COUNTER) {
				return privatePlansMap;
			}
		}

		public static HashMap<PrivatePlan, String> getPlansWithLocation(Location
buraco) {
			HashMap<PrivatePlan, String> map = new HashMap<>();
			synchronized (TrafficClock.COUNTER) {
				Set<GlobalPlanMapKey> keys = globalPlansMap.keySet();
				for (GlobalPlanMapKey globalPlanMapKey : keys) {
					if (globalPlanMapKey.getLocation().equals(buraco)){
						String name = globalPlansMap.get(globalPlanMapKey);
						PrivatePlan p = privatePlansMap.get(name);
						map.put(p, name);
					}
				}
				return map;
			}
		}
}
```

16. GlobalPlanMapKey.java

```
package plan;

import jason.environment.grid.Location;

class GlobalPlanMapKey implements Comparable<GlobalPlanMapKey> {
	private Location location;
	private int counter;
	private boolean hasWaited;

	public GlobalPlanMapKey(Location location, int counter) {
		this.location = location;
		this.counter = counter;
	}

	@Override
	public boolean equals(Object obj) {
		if (obj == null)
			return false;
		if (this == obj)
			return true;
		if (!(obj instanceof GlobalPlanMapKey))
			return false;

		GlobalPlanMapKey other = (GlobalPlanMapKey) obj;
		if (other.location.equals(this.location)) {
			if (other.counter == this.counter) {
				return true;
			}
		}
		return false;
	}

	@Override
	public int hashCode() {
		int PRIME = 31;
		int result = 1;
		result = PRIME * result + location.x;
		result = PRIME * result + location.y;
		result = PRIME * result + counter;
		return result;
	}

	@Override
	public int compareTo(GlobalPlanMapKey o) {
		return Integer.compare(this.counter, o.counter);
	}

	public Location getLocation() {
```

```
        return location;
}

public int getCounter() {
        return counter;
}

public boolean isHasWaited() {
        return hasWaited;
}

public void setHasWaited(boolean hasWaited) {
        this.hasWaited = hasWaited;
}



}
```

17. PrivatePlan.java

```
package plan;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Set;

import jason.environment.grid.Location;

public class PrivatePlan implements Cloneable{

	private HashMap<Integer, Location> plan = new HashMap<>();
	private int version;

	public PrivatePlan(HashMap<Integer, Location> plan) {
		this(plan, 1);
	}
	public PrivatePlan(HashMap<Integer, Location> plan, int version) {
		this.plan = plan;
		this.version = version;
	}


	public int getVersion() {
		return version;
	}

	@Override
	protected PrivatePlan clone() throws CloneNotSupportedException {
		PrivatePlan clone = (PrivatePlan) super.clone();
		return clone;
	}

	public void increaseVersion() {
		this.version++;
	}

	public Location getLocationAt(int instant){
		return plan.get(instant);
	}

	public HashMap<Integer, Location> getPlan(){
		return this.plan;
	}

	public Location getLastLocation(){
		Set<Integer> keySet = plan.keySet();
		List<Integer> list = new ArrayList<>(keySet);
```

```
        Collections.sort(list);

        int lastClock = list.get(list.size()-1);

        return plan.get(lastClock);
}

}
```

18. PrivatePlanManager.java

```
package plan;

import java.util.HashMap;

import jason.environment.grid.Location;

public class PrivatePlanManager {

	private static HashMap<String, PrivatePlan> plans = new HashMap<>();

	public static void save(String name, PrivatePlan plan) {
		synchronized (plans) {
			plans.put(name, plan);
		}
	}

	public static int getMyPlanVersion(String name) {
		synchronized (plans) {
			return plans.get(name).getVersion();
		}
	}

	public static Location getNextPositionAt(String name, int instant) {
		synchronized (plans) {
			PrivatePlan p = plans.get(name);
			return p.getLocationAt(instant);
		}
	}



}
```

19. whereIsMyHome.java

```
package jia;

import jason.JasonException;
import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.NumberTermImpl;
import jason.asSyntax.Term;
import jason.environment.grid.Location;
import system.TrafficClock;
import traffic.DrivingModel;

public class whereIsMyHome extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = 7841062451049819417L;

	private int counter = 0;

	@Override
	public Object execute(final TransitionSystem ts, final Unifier un, final
Term[] terms) throws Exception {
		try {
			if (!terms[0].isAtom()) {
				// System.out.println(terms[0].toString());
				throw new JasonException("The first argument of the
internal action 'whereami' is not an Atom.");
			}
			if (!terms[1].isVar()) {
				throw new JasonException("The argument of the internal
action 'whereami' is not a variable.");
			}
			if (!terms[2].isVar()) {
				throw new JasonException("The argument of the internal
action 'whereami' is not a variable.");
			}

			String name = ((Atom) terms[0]).toString();

			Location[] destination = DrivingModel.whereIsMyHome(name);

			if (destination != null) {

				if (counter >= destination.length) {
					return false;
				}
```

```
                        Location des = destination[counter];
                        counter++;
                        int x = des.x;
                        int y = des.y;
                        return un.unifies(terms[1], new NumberTermImpl(x)) &&
un.unifies(terms[2], new NumberTermImpl(y));
                  } else {
                        Thread.sleep(TrafficClock.AGENT_WAIT_TIME);
//                      synchronized (TrafficClock.COUNTER) {
//                            TrafficClock.COUNTER.wait();
//                      }
                        return false;
                  }
            } catch (ArrayIndexOutOfBoundsException e) {
                  throw new JasonException("The internal action 'random' has not
received the required argument.");
            } catch (Exception e) {
                  throw new JasonException("Error in internal action 'random': " +
e, e);
            }
      }

}
```

20. whereami.java

```
package jia;

import jason.JasonException;
import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.NumberTermImpl;
import jason.asSyntax.Term;
import jason.environment.grid.Location;
import traffic.DrivingModel;

public class whereami extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = 7407810088162225140L;

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
		try {
			if (!terms[0].isAtom()) {
//				System.out.println(terms[0].toString());
				throw new JasonException("The first argument of the
internal action 'whereami' is not an Atom.");
			}
			if (!terms[1].isVar()) {
				throw new JasonException("The argument of the internal
action 'whereami' is not a variable.");
			}
			if (!terms[2].isVar()) {
				throw new JasonException("The argument of the internal
action 'whereami' is not a variable.");
			}

			String name = ((Atom) terms[0]).toString();

			Location location = DrivingModel.getAgentLocation(name);

			if (location != null){
				int x = location.x;
				int y = location.y;
				return un.unifies(terms[1], new NumberTermImpl(x)) &&
un.unifies(terms[2], new NumberTermImpl(y));
			} else {
				return false;
			}
		} catch (Throwable e) {
```

```
                    e.printStackTrace();
                    return false;
                }
            }

        }
```

21. waitTimer.java

```
package jia;

import java.util.logging.Logger;

import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.Term;
import system.TrafficClock;

public class waitTimer extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = 3437768761758311883L;
	private Logger logger = Logger.getLogger(waitTimer.class.getName());

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
		try {
			// synchronized (TrafficClock.COUNTER) {
			String name = ((Atom) terms[0]).toString();

			 Thread.sleep(TrafficClock.AGENT_WAIT_TIME);

			int agentCounter;
			int globalCounter;
			boolean canExecute;
			synchronized (TrafficClock.COUNTER) {
				agentCounter = TrafficClock.getAgentCounter(name);
				globalCounter = TrafficClock.getCounter();

				canExecute = (agentCounter + 1) == globalCounter;
//				if (!canExecute) {
//					TrafficClock.COUNTER.wait();
//				}
			}
			return canExecute;
			// }
		} catch (Throwable e) {
			e.printStackTrace();
			return false;
		}
	}

}
```

22. updateMyPlan.java

```
package jia;

import java.util.logging.Logger;

import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.Term;
import plan.GlobalPlanManager;
import plan.PrivatePlan;
import plan.PrivatePlanManager;
import system.TrafficClock;

public class updateMyPlan extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = 4001882026510196767L;
	private Logger logger = Logger.getLogger(updateMyPlan.class.getName());

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
		try {
			String name = ((Atom) terms[0]).toString();

			new Object();

			synchronized (TrafficClock.COUNTER) {
				PrivatePlan privatePlanOnGlobalPlan =
GlobalPlanManager.getPrivatePlan(name);

				if (privatePlanOnGlobalPlan != null) {
					int myPlanVersion =
PrivatePlanManager.getMyPlanVersion(name);

					int newVersion =
privatePlanOnGlobalPlan.getVersion();
					if (myPlanVersion < newVersion) {
						PrivatePlanManager.save(name,
privatePlanOnGlobalPlan);
						logger.info(name + " updating my plan from
version " + myPlanVersion + " to version " + newVersion);
					}
				}
			}

			return true;
```

```
                } catch (Throwable e) {
                        e.printStackTrace();
                        return false;
                }
        }

}
```

23. getDirectionFromPlan.java

```
package jia;

import java.util.logging.Logger;

import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.NumberTerm;
import jason.asSyntax.NumberTermImpl;
import jason.asSyntax.Term;
import jason.environment.grid.Location;
import plan.GlobalPlanManager;
import plan.PrivatePlan;
import plan.PrivatePlanManager;
import system.TrafficClock;

public class getDirectionFromPlan extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = -4621071386925102193L;

	private static Logger logger =
Logger.getLogger(getDirectionFromPlan.class.getName());

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
		try {
			new Object();

			String name = ((Atom) terms[0]).toString();

			int iagx = (int) ((NumberTerm) terms[1]).solve();
			int iagy = (int) ((NumberTerm) terms[2]).solve();
			synchronized (TrafficClock.COUNTER) {
				PrivatePlan privatePlanOnGlobalPlan =
GlobalPlanManager.getPrivatePlan(name);
				if (privatePlanOnGlobalPlan != null) {
					int myPlanVersion =
PrivatePlanManager.getMyPlanVersion(name);

					int newVersion =
privatePlanOnGlobalPlan.getVersion();
					if (myPlanVersion < newVersion) {
						PrivatePlanManager.save(name,
privatePlanOnGlobalPlan);
```

```
                                        logger.info(name + " updating my plan from
version " + myPlanVersion + " to version " + newVersion);
                                    }
                              }
                        }

                        Location myPosition = new Location(iagx, iagy);

                        Location nextPosition =
PrivatePlanManager.getNextPositionAt(name, TrafficClock.getCounter());

                        if (myPosition.isNeigbour(nextPosition)) {
                              int x = nextPosition.x;
                              int y = nextPosition.y;

                              return un.unifies(terms[3], new NumberTermImpl(x)) &&
un.unifies(terms[4], new NumberTermImpl(y));
                        }

                        return false;

                  } catch (Throwable e) {
                        for (StackTraceElement stackTraceElement : e.getStackTrace()) {
                              logger.info(stackTraceElement.toString());
                        }
                        return false;
                  }
            }
}
```

24. getDirection.java

```
package jia;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.logging.Logger;

import graph.GraphSearch;
import jason.JasonException;
import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.NumberTerm;
import jason.asSyntax.NumberTermImpl;
import jason.asSyntax.Term;
import jason.environment.grid.Location;

public class getDirection extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = 1L;

	private static Logger logger = Logger.getLogger(getDirection.class.getName());

	private Set<Location> obstacles = new HashSet<>();

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
		try {
			if (!terms[6].isVar()) {
				throw new JasonException("The argument of the internal
action 'getDirection' is not a variable.");
			}
			if (!terms[7].isVar()) {
				throw new JasonException("The argument of the internal
action 'getDirection' is not a variable.");
			}

			int iagx = (int) ((NumberTerm) terms[0]).solve();
			int iagy = (int) ((NumberTerm) terms[1]).solve();
			int itox = (int) ((NumberTerm) terms[2]).solve();
			int itoy = (int) ((NumberTerm) terms[3]).solve();

			if (!terms[4].isVar()) {
				int buracoX = (int) ((NumberTerm) terms[4]).solve();
				int buracoY = (int) ((NumberTerm) terms[5]).solve();
				Location buraco = new Location(buracoX, buracoY);
```

```
                        obstacles.add(buraco);
                    }
                    Location from = new Location(iagx, iagy);
                    Location to = new Location(itox, itoy);


                    List<Location> path = GraphSearch.findShortestPath(from, to,
obstacles);

                    logger.info("path.size: " + path.size());
                    String locations = "{";
                    for (Location location : path) {
                        locations += " new Location(" +location.toString() + "), ";
                    }
                    logger.info(locations + "};");

                    if (path.size() > 1) {
                        Location nextLocation = path.get(1);

                        int x = nextLocation.x;
                        int y = nextLocation.y;
                        // System.out.println("nextLocation: "+ nextLocation);
                        return un.unifies(terms[6], new NumberTermImpl(x)) &&
un.unifies(terms[7], new NumberTermImpl(y));
                    }
                    return false;
                } catch (Throwable e) {
                    e.printStackTrace();
                    return false;
                }
            }

}
```

25. createPlan.java

```
package jia;

import java.util.List;
import java.util.logging.Logger;

import graph.GraphSearch;
import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.NumberTerm;
import jason.asSyntax.Term;
import jason.environment.grid.Location;
import plan.GlobalPlanManager;
import plan.PrivatePlan;
import plan.PrivatePlanManager;

public class createPlan extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = -2905804114150662912L;

	private static Logger logger = Logger.getLogger(createPlan.class.getName());

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
			String name = ((Atom) terms[0]).toString();
			int iagx = (int) ((NumberTerm) terms[1]).solve();
			int iagy = (int) ((NumberTerm) terms[2]).solve();
			int itox = (int) ((NumberTerm) terms[3]).solve();
			int itoy = (int) ((NumberTerm) terms[4]).solve();

			Location from = new Location(iagx, iagy);
			Location to = new Location(itox, itoy);

		try {
			List<Location> route = GraphSearch.findShortestPath(from, to);

			String s = "";
			for (Location location : route) {
				s += " new Location(" + location + "), ";
			}
			logger.info(s);

			PrivatePlan plan = GlobalPlanManager.add(name, route);

			PrivatePlanManager.save(name, plan);
```

```
                return true;

            } catch (Throwable e) {
                for (StackTraceElement stackTraceElement : e.getStackTrace()) {
                    logger.info(stackTraceElement.toString());
                }

                e.printStackTrace();
                return false;
            }
        }
}
```

26. changePlan.java

```
package jia;

import java.util.HashMap;
import java.util.List;

import graph.GraphSearch;
import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.NumberTerm;
import jason.asSyntax.Term;
import jason.environment.grid.Location;
import plan.GlobalPlanManager;
import plan.PrivatePlan;
import plan.PrivatePlanManager;
import system.TrafficClock;

public class changePlan extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = 3567504326500386213L;

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {

		String name = ((Atom) terms[0]).toString();
		int agentX = (int) ((NumberTerm) terms[1]).solve();
		int agentY = (int) ((NumberTerm) terms[2]).solve();

		int buracoX = (int) ((NumberTerm) terms[3]).solve();
		int buracoY = (int) ((NumberTerm) terms[4]).solve();

		int homeX = (int) ((NumberTerm) terms[5]).solve();
		int homeY = (int) ((NumberTerm) terms[6]).solve();

		Location from = new Location(agentX, agentY);
		Location to = new Location(homeX, homeY);

		Location buraco = new Location(buracoX, buracoY);

		synchronized (TrafficClock.COUNTER) {
			List<Location> route = GraphSearch.findShortestPath(from, to,
buraco);

			int clockTimer = TrafficClock.getCounter() - 1;
```

```
                    PrivatePlan plan = GlobalPlanManager.add(name, route,
clockTimer);

                    PrivatePlanManager.save(name, plan);

                    HashMap<PrivatePlan, String> plansWithProblem =
GlobalPlanManager.getPlansWithLocation(buraco);
                    for (PrivatePlan privatePlan : plansWithProblem.keySet()) {
                          int c = clockTimer;
                          Location locationAt = privatePlan.getLocationAt(c);
                          Location lastLocation = privatePlan.getLastLocation();

                          if (locationAt == null){
                                c++;
                                locationAt = privatePlan.getLocationAt(c);
                          }

                          List<Location> agentRoute =
GraphSearch.findShortestPath(locationAt, lastLocation, buraco);
                          String agentName = plansWithProblem.get(privatePlan);
                          GlobalPlanManager.add(agentName, agentRoute, c);
                    }
             }

             return true;
      }
}
```

27. canMove.java

```
package jia;

import agent.AgentFactory;
import agent.CityAgent;
import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Atom;
import jason.asSyntax.NumberTerm;
import jason.asSyntax.Term;
import system.TrafficClock;
import traffic.CityModel;

public class canMove extends DefaultInternalAction {

	/**
	 *
	 */
	private static final long serialVersionUID = -8905774560823455874L;
//	private Logger logger = Logger.getLogger(canMove.class.getName());

	@Override
	public Object execute(TransitionSystem ts, Unifier un, Term[] terms) throws
Exception {
		try {
			String name = ((Atom) terms[0]).toString();
			int x = (int) ((NumberTerm) terms[1]).solve();
			int y = (int) ((NumberTerm) terms[2]).solve();

			Thread.sleep(TrafficClock.AGENT_WAIT_TIME);

			boolean canMove = false;
			if (CityModel.isLocationFree(x, y)) {
				canMove = true; // free
			} else {
				int agentIdAtPosition = CityModel.getAgentAtLocation(x, y);
				if (agentIdAtPosition == -1) {
					canMove = true;
				} else {
					CityAgent agent = AgentFactory.getAgentByName(name);
					if (agentIdAtPosition == agent.getId()) {
						canMove = true; // If it is myself, I can stay
still
					}
				}
			}

//			logger.info(name + " can move? " + canMove);
			return canMove;
```

```
            } catch (Throwable e) {
                  e.printStackTrace();
                  return false;
            }
      }

}
```

28. GraphSearch.java

```
package graph;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import jason.environment.grid.Location;

public class GraphSearch {

	private static GraphSearch search = new GraphSearch();
	private Set<Location> obstacles = new HashSet<>();

	public static List<Location> findShortestPath(Location from, Location to) {
		return search.findShortestPathByAStar(from, to, new HashSet<>());
	}

	public static List<Location> findShortestPath(Location from, Location to,
Set<Location> obstacles) {
		return search.findShortestPathByAStar(from, to, obstacles);
	}

	public static List<Location> findShortestPath(Location from, Location to,
Location obstacle) {
		search.obstacles.add(obstacle);

		return search.findShortestPathByAStar(from, to, search.obstacles);
	}

	public List<Location> findShortestPathByAStar(Location from, Location to,
Set<Location> obstacles) {

		HashMap<Location, GraphNode> hashMap = CityGraph.get().getHashMap();

		Set<Location> closedSet = new HashSet<>();

		HashSet<Location> openSet = new HashSet<>();
		openSet.add(from);

		HashMap<Location, NodeInfo> graphScore = new HashMap<>();

		for (Location location : hashMap.keySet()) {
			graphScore.put(location, new NodeInfo(Integer.MAX_VALUE));
		}
		graphScore.put(from, new NodeInfo(0));
```

```
		while (!openSet.isEmpty()){
			Location current = null;
			for (Location location : openSet) {
				if (current == null
						|| graphScore.get(location).getValue() <
graphScore.get(current).getValue()){
					current = location;
				}
			}

			if (current.equals(to)){
//				System.out.println("found!!");
				break;
			}

			openSet.remove(current);
			closedSet.add(current);

			for (GraphNode graphNode : hashMap.get(current).getNeigbors()) {
				Location neighborLocation = graphNode.getLocation();

				if (!closedSet.contains(neighborLocation)){

					openSet.add(neighborLocation);

					NodeInfo currentNodeInfo = graphScore.get(current);

					int score = currentNodeInfo.getValue() + 1; //cost
between neighbors is always one

					if (obstacles.contains(neighborLocation)){
						score = Integer.MAX_VALUE;
					}

					NodeInfo neighborNodeInfo =
graphScore.get(neighborLocation);

					if (score < neighborNodeInfo.getValue()){
						neighborNodeInfo.update(score, current);
					}

				}
			}
		}

		List<Location> shortestPath = new ArrayList<>();
		NodeInfo nodeInfo = graphScore.get(to);
		shortestPath.add(to);
		while (nodeInfo.getPreviewLocation() != null){
			Location previewLocation = nodeInfo.getPreviewLocation();
//			System.out.println(previewLocation);

			nodeInfo = graphScore.get(previewLocation);
```

```
                shortestPath.add(previewLocation);
            }

            Collections.reverse(shortestPath);
            return shortestPath;
        }

        class NodeInfo {

            private int value;
            private Location previewLocation;

            public NodeInfo(int value) {
                this.value = value;
            }

            public void update(int score, Location current) {
                this.value = score;
                this.previewLocation = current;
            }

            public int getValue() {
                return value;
            }

            public void setValue(int value) {
                this.value = value;
            }

            public Location getPreviewLocation() {
                return previewLocation;
            }

            public void setPreviewLocation(Location previewLocation) {
                this.previewLocation = previewLocation;
            }

        }

}
```

29. GraphNode.java

```
package graph;

import java.util.ArrayList;
import java.util.List;

import jason.environment.grid.Location;

public class GraphNode {

	private Location location;
	private GraphNode north;
	private GraphNode south;
	private GraphNode east;
	private GraphNode west;
	private String way;

	public GraphNode(Location location, String way) {
		this.setLocation(location);
		this.way = way;
	}

	public GraphNode getNorth() {
		return north;
	}

	public void setNorth(GraphNode north) {
		this.north = north;
	}

	public GraphNode getSouth() {
		return south;
	}

	public void setSouth(GraphNode south) {
		this.south = south;
	}

	public GraphNode getEast() {
		return east;
	}

	public void setEast(GraphNode east) {
		this.east = east;
	}

	public GraphNode getWest() {
		return west;
	}
```

```
	public void setWest(GraphNode west) {
		this.west = west;
	}

	public Location getLocation() {
		return location;
	}

	public void setLocation(Location location) {
		this.location = location;
	}

	public String getWay() {
		return way;
	}

	@Override
	public String toString() {
		return "("+getLocation()+"):"+getWay();
	}

	public List<GraphNode> getNeigbors(){
		List<GraphNode> neighbors = new ArrayList<>();

		if (this.north != null){
			neighbors.add(north);
		}
		if (this.south != null){
			neighbors.add(south);
		}
		if (this.east != null){
			neighbors.add(east);
		}
		if (this.west != null){
			neighbors.add(west);
		}

		return neighbors;
	}

}
```

30. CityGraph.java

```
package graph;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;

import jason.environment.grid.Location;

public class CityGraph {

	private final HashMap<Location, GraphNode> hashMap = new HashMap<Location,
GraphNode>();

	private static CityGraph instance = new CityGraph();

	private CityGraph() {}

	public static CityGraph get(){
		return instance;
	}

	public HashMap<Location, GraphNode> getHashMap(){
		return hashMap;
	}

	public void addOnGraph(String[][] map, int row, int collumn, String way) {
		if (!way.equals(".")) {
			Location location = new Location(collumn, row);

			// jacamo invert x and y, so I invert again to look like the
matrix at CityMap.class
			GraphNode node = hashMap.get(location);
			if (node == null) {
				node = new GraphNode(location, way);
				hashMap.put(location, node);
			}

			checkNorth(node, map, hashMap, row, collumn, way, location);
			checkSouth(node, map, hashMap, row, collumn, way, location);
			checkEast(node, map, hashMap, row, collumn, way, location);
			checkWest(node, map, hashMap, row, collumn, way, location);

			// System.out.println(location + " " + pos);

		}
	}

	public void readGraph() {
		Location l = new Location(47, 19);
```

```
            GraphNode node = hashMap.get(l);

            Set<Location> visited = new HashSet<>();

            print(visited, node);

      }

      private void print(Set<Location> visited, GraphNode node) {
            Location location = node.getLocation();
            if (visited.contains(location)) {
                  return;
            }

            visited.add(location);
//          System.out.println(node);

            if (node.getNorth() != null) {
                  print(visited, node.getNorth());
            }
            if (node.getSouth() != null) {
                  print(visited, node.getSouth());
            }
            if (node.getEast() != null) {
                  print(visited, node.getEast());
            }
            if (node.getWest() != null) {
                  print(visited, node.getWest());
            }
      }

      private void checkWest(GraphNode node, String[][] map, HashMap<Location,
GraphNode> hashMap, int row, int collumn, String pos, Location location) {
            GraphNode nodeWest = check(map, hashMap, row, collumn - 1);

            if (nodeWest != null && !(pos.equals("E") || pos.equals("K") ||
nodeWest.getWay().equals("E") || nodeWest.getWay().equals("K"))) {
                  node.setWest(nodeWest);
            }
      }

      private void checkEast(GraphNode node, String[][] map, HashMap<Location,
GraphNode> hashMap, int row, int collumn, String pos, Location location) {
            GraphNode nodeEast = check(map, hashMap, row, collumn + 1);

            if (nodeEast != null && !(pos.equals("W") || pos.equals("Q") ||
nodeEast.getWay().equals("W") || nodeEast.getWay().equals("Q"))) {
                  node.setEast(nodeEast);
            }
            hashMap.put(location, node);
      }
```

```
	private void checkSouth(GraphNode node, String[][] map, HashMap<Location,
GraphNode> hashMap, int row, int collumn, String pos, Location location) {
		GraphNode nodeSouth = check(map, hashMap, row + 1, collumn);

		if (nodeSouth != null && !(pos.equals("N") || pos.equals("B") ||
nodeSouth.getWay().equals("N") || nodeSouth.getWay().equals("B"))) {
			// if going north, I cannot have a way to south, otherwise it
would
			// be driving on the wrong-way.
			node.setSouth(nodeSouth);
		}
		hashMap.put(location, node);

	}

	private void checkNorth(GraphNode node, String[][] map, HashMap<Location,
GraphNode> hashMap, int row, int collumn, String pos, Location location) {
		GraphNode nodeNorth = check(map, hashMap, row - 1, collumn);

		if (nodeNorth != null && !(pos.equals("S") || pos.equals("A") ||
nodeNorth.getWay().equals("S") || nodeNorth.getWay().equals("A"))) {
			node.setNorth(nodeNorth);
		}
		hashMap.put(location, node);
	}

	private GraphNode check(String[][] map, HashMap<Location, GraphNode> hashMap,
int row, int collumn) {
		GraphNode node = null;

		if (row >= 0 && row < map.length && collumn >= 0 && collumn <
map[0].length) {
			String way = map[row][collumn];

			if (!way.equals(".")) {
				Location location = new Location(collumn, row);

				node = hashMap.get(location);
				if (node == null) {
					node = new GraphNode(location, way);
					hashMap.put(location, node);
				}

			}
		}

		return node;
	}

}
```

Desenvolvimento de uma abordagem de cooperação em sistemas multiagentes

Ricardo Passarella, Elder Rizzon Santos

Curso de Bacharelado em Ciência da Computação – Universidade Federal de Santa Catarina (UFSC) – Campus de Florianópolis
88040-900 – Florianópolis – SC – Brasil

ricpass@gmail.com, elder.santos@ufsc.br

Abstract. *In this paper a study was carried out on multi-agent systems, and in particular, some approaches on how agents can cooperate. Using this study, It was possible to create a scenario with the following problem: How can autonomous cars cooperate between themselves in someway that it not necessary to use a semaphore at an intersection.*

It was adopted an approach of shared plans to solve this problem. And it was implement using the framework JaCaMo so it was possible to analyse this approach efficiency. The cooperative agent was compared with an agent that didn't know to to cooperate and rely on semaphore at intersection. The tests showed that the approach with shared plans takes less time on average to move to their final destination.

Resumo. *Neste trabalho foi realizado um estudo sobre sistemas multiagentes, e particularmente, as abordagens para cooperação entre agentes. A partir desse estudo foi criado um cenário com o seguinte problema: como carros podem cooperar entre si em um cruzamento de modo que não haja necessidade de semáforos.*

Para solucionar este problema foi desenvolvida uma abordagem baseada em planos compartilhados. Para verificar a eficiência deste plano, foi implementado esta abordagem utilizando o framework JaCaMO. E como comparativo, foi implementado um agente sem capacidade de cooperação, necessitando do uso de semáforos em cruzamentos. Ambos os agentes foram testados no cenário desenvolvido. Os testes mostraram que a abordagem baseada em planos compartilhados leva menos tempo para os agentes se locomoverem até seu destino final.

1. Introdução

Os sistemas de computadores tem se tornado cada vez mais eficazes, possibilitando as mais variadas proezas, desde ganhar dos melhores jogadores de xadrez (ou recentemente dos melhores jogadores de Go), até piloto automático de carro.

Atualmente, estamos entrando em um mundo totalmente conectado, o termo regularmente empregado "Internet das Coisas", demonstra que teremos inúmeros computadores conectado entre si, trocando informações e precisando realizar as mais variadas funções em conjunto.

O problema é que para muitas dessas funções, as técnicas de sistemas distribuídos tradicionais não conseguem solucioná-las. Por isso, a área de sistemas multiagentes estuda, entre

outros assuntos, como agentes autônomos podem se relacionar a fim de solucionar um determinado problema.

Um sistema multiagentes precisa de mecanismos para que os agentes possam se comunicar, principalmente se o sistema for um mundo aberto e possuir agentes heterogêneos. Por isso, linguagens específicas para agentes precisam ser desenvolvidas e implementadas.

Inevitavelmente, embora os agentes sejam autônomos, eles terão que trabalhar em conjunto em algumas situações. Seja para atingir um objetivo inviável, ou para economizar recursos ao dividir as tarefas em vários agentes.

Várias abordagens estão sendo aplicadas em cooperação de agentes, seja aprendizagem, planos, alocação de tarefas e sociedade. Cada uma tentando solucionar um dos problemas em sistemas multiagentes.

Em aprendizagem, os agentes estão recebendo novas informações sobre o sistema e inferindo novas crenças. Estes agentes terão que decidir, por exemplo, quando compartilhar e coletar as informações com outros agentes.

Os agentes começam a trabalhar em conjunto pois o objetivo dos agentes é inalcançável de outra maneira ou porque eles podem economizar recursos (materiais, tempo) caso o façam. Para trabalharem em conjunto eles terão alocar as tarefas entre eles.

Em vários casos, o agente precisa planejar suas ações para atingir o objetivo maior. Seja ordenando as ações ou compartilhando seus planos com outros agentes. Esse caso seria uma abordagem baseada em planos.

E por fim, o agente está inserido em um sistema. Em vários casos, o sistema precisa definir regras que todos os agentes devem seguir, principalmente se estivermos lidando com um sistema aberto, onde agentes heterogêneos podem entrar e sair a vontade. Nessa situação, a abordagem baseada em sociedade utiliza-se de normas para limitar as ações dos agentes.

2. Trabalhos relacionados

Levando em consideração as várias abordagens possíveis para cooperação de agentes, foi realizado um levantamento de algumas abordagens apresentadas na conferência de agentes autônomos e sistemas multiagentes de 2015 (AAMAS 2015).

Em (ECK, SOH, 2015), os autores utilizam a abordagem de aprendizagem para compartilhamento de informações. Os agentes precisavam decidir se pedem, coletam ou compartilham a informação. Para simplificar o processo de aprendizagem, os autores transformam o estado atual do ambiente em um problema mais simples de melhoramento do conhecimento ao longo do tempo, numa transformada nomeada de Knowledge State MDP. Essa formulação do problema permite os agentes realizar decisões baseadas no estado atual do seu conhecimento, ao invés do ambiente. Com essa transformada, os autores podem utilizar aprendizagem por reforço para aprender sobre o ambiente e sobre os outros agentes, inviável anteriormente devido a alta complexidade.

Outra abordagem foi proposta por (PUJOL-GONZALEZ et al, 2015), os autores usam o algoritmo max-sum para a coordenação entre times. Que é um algoritmo de passagem de mensagem baseado na lei genérica de distribuição (generalised distributive law - GDL)). Aplicando este algoritmo no problema do Robocup 2013, os autores mostram empiricamente que o algoritmo max-sum binário obtém melhores resultados que outros problemas de otimização distribuídos

restritos (distributed constraint optimization problem - DCOP) no estado da arte, conseguindo prevenir o dobro de dano na cidade.

A ideia de (WICKE, FREELAN, LUKE, 2015) foi utilizar o conceito de caçadores de recompensa para solucionar o problema de alocação de tarefas. Esta definição é um sistema comum no estados unidos na qual qualquer agente pode se comprometer com uma tarefa, mas somente o primeiro que completá-la ganha a recompensa. A vantagem dessa abordagem, segundo os autores, é que as técnicas tradicionais baseadas em leilões, o agente que ganhou o leilão não necessariamente é o mais capacitado para realizar a tarefa, ou mesmo, se consegue concluí-la. Já na abordagem de caçadores de recompensa, uma mesma tarefa pode estar sendo realizada por vários agentes ao mesmo tempo. O algoritmo proposto, ComplexP, demonstra melhores resultados que leilões tradicionais na simulação de um futebol de robôs, principalmente porque uma tarefa não era exclusiva de um único agente.

Em problemas de planejamento distribuído contínuo, geralmente, exige que agente quebre e refaça compromissos durante execução, normalmente causados por eventos inesperados ou alterações de objetivos. Em (FARINELLI et al, 2015), os autores apresentam uma forma de interromper o plano para que humanos consigam terminar uma tarefa (seja porque o agente não tem conhecimento completo ou porque ele não tem a capacidade de realizá-la), e em seguida, o agente pode continuar o plano inicial, sendo essa interrupção o mais suave possível.

Há várias propostas de como usar normas em sistemas multiagentes, sendo que a principal utilidade delas é em coordenar e controlar o comportamento dos agentes. Em (ALECHINA, BULLING, DASTANI, 2015), os autores tentam responder como e quando restringir o comportamento dos agentes a fim de obter a solução desejada. Para isso, eles estudam o problema prático de normas executivas e criam o conceito de Guarda. O Guarda são funções que restringe as possíveis ações depois de algum evento. Os autores também propõem um modelo computacional formal de normas, guardas e normas executivas, baseada em lógica temporal em tempo linear com operadores do passado. E concluem que nem todas as normas podem ser aplicadas com funções como Guardas, mesmo com recursos computacionais ilimitado para raciocinar eventos futuros.

(KING et al, 2015) propõem usar organizações para governar e revisar outras organizações, seguindo os exemplos que encontramos no mundo social, como tratados, legislações, acordos. Para isso, é estabelecido uma espaço regulatório definido por normas na forma de um corpo de regulação. Segundos os autores, cada organização projeta suas normas para seus próprios objetivos, por isso, ao criar uma autoridade maior guiando as regras organizacionais, cada organização terá que coordenar seus projetos com as outras organizações e não poderá impor limites inaceitáveis ao direitos dos agentes. Os autores propõem um framework computacional formal baseado em camadas de organizações, na qual as normas de cada camada é governada e monitoradas pela camada superior.

3. Cooperação via planos Compartilhados

Analisando as várias abordagens de cooperação, foi desenvolvido uma abordagem baseada em planos compartilhados utilizando como base o framework Generalized Partial Global Planning (Planejamento Global Parcial generalizado, ou sigla GPGP) para solucionar o problema de trânsito proposto.

3.1 Estudo de caso

Foram estudados vários problemas em que a cooperação entre agentes poderia melhorar o resultado final. Um assunto que vem ganhando destaque nos últimos anos é o desenvolvimento de carros autônomos, e nos últimos meses quase toda empresa automobilística anunciou que está desenvolvendo algum tipo de carro inteligente.

Analisando o problema de tráfego, existem várias situações em que carros autônomos poderiam cooperar entre si. Dentro deste contexto, foi escolhido para estudo um problema comum no tráfego de carro: como carros poderiam cooperar entre si em cruzamentos de modo que não haja necessidade de semáforos?

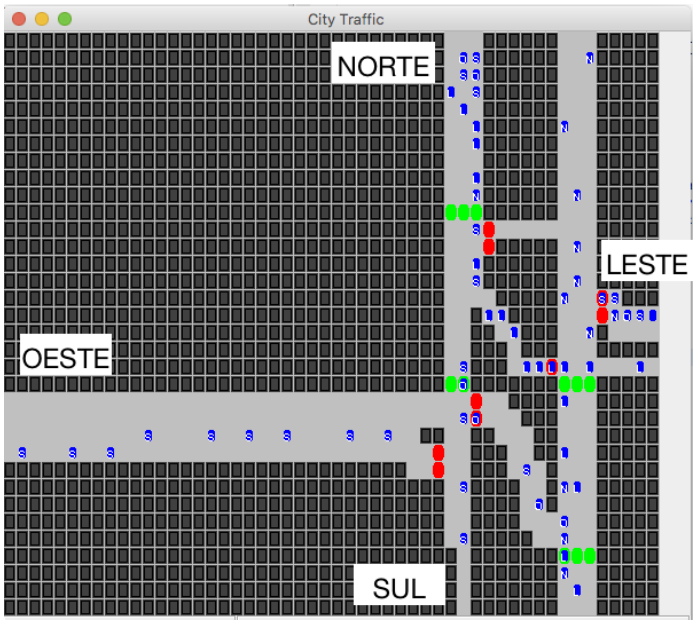


Figura 1. Ambiente criado no JaCaMo

Ambiente criado usando o JaCaMo no qual os agentes foram aplicados. As cores vermelho e verde são o semáforo no seu estado atual. A cor azul representa o agente, a letra em cima do agente informa qual para onde o agente está indo (a primeira letra de cada ponto cardeal).

3.2 Algoritmo adaptado ao trânsito

A abordagem com planos desenvolvida neste trabalho é descentralizada, ou seja, todos os agentes possuem um plano global e ao mesmo tempo seu estado mental particular. A abordagem da sociedade (SMA) pode ser resumida pelo seguinte algoritmo (baseado no funcionamento do GPGP):

1. Agente entra no sistema
2. Ele cria uma rota até seu destino final.
3. Ele tenta adequar sua rota ao plano global, criando um plano privado coordenado.
    a. Caso haja conflito no ponto x, y no instante de tempo t, espera um instante de tempo na posição anterior ao conflito.

   b. Caso haja um segundo conflito na mesma posição, altere o plano do agente que iria causar o segundo conflito.
4. Notifique os outros agentes sobre o plano global foi atualizado.
5. Cada agente atualiza seu plano privado caso haja alteração.

A seguir cada uma das etapas do algoritmo é descrita com mais detalhes:

1. No modelo GPGP, quando os agentes se encontram, ele pode iniciar um diálogo e compartilhar informações. No estudo de caso proposto, temos carros autônomos capazes de cooperar entre si, no entanto, um carro não precisa comunicar com todos os carros da cidade se o problema é somente no cruzamento à frente. Então, considera-se que quando o agente entra no sistema, o agente está iniciando um diálogo com os outros agentes na mesma região, ou seja, os agentes que podem afetar suas tarefas e vice versa. E nesse diálogo o novo agente recebe o plano global dos agentes presentes.
2. Do mesmo modo que o modelo GPGP, o agente cria uma sequência de tarefas internamente de modo que solucione seu problema que é sair do ponto x, y e ir até x1, y1. Essa rota é um conjunto de movimentos que o agente deve fazer, mas ainda não possui nenhuma forma de cooperação. Para encontrar a menor distância entre esse dois pontos, o algoritmo Dijkstra encontra o caminho mais curto.
3. Com a sequência de tarefas e o conhecimento sobre o plano global adquirido ao entrar no sistema, o agente analisa possíveis conflitos entre suas tarefas e as tarefas dos outros agentes. O plano global é estruturado como um mapeamento em que um ponto x, y e instante de tempo t indicam se aquela posição e naquele instante está ocupado, e caso esteja ocupado, qual é o nome do agente.
   a. Com esse mapeamento, é possível descobrir se haverá um conflito. No primeiro conflito o agente espera um instante de tempo na posição anterior ao conflito.
   b. Caso ocorra um segundo conflito no instante t+1, ou seja, no próximo instante, o agente pode utilizar aquela posição no instante t+1. E altera o plano do agente que teve conflito para esperar o instante t+1 na posição anterior. E para que o agente que teve seu plano alterado saiba que seu plano foi alterado, a versão do plano é incrementada.
   c. No final, o agente irá criar um plano privado para si, esse plano é a rota criada anteriormente só que com ações coordenadas. O plano privado é estruturado como um mapeamento de instante t para a posição que o agente precisa estar.
4. E seguindo o modelo GPGP, depois de reagendar suas tarefas, o agente notifica os outros agentes. Quando termina de escalonar suas tarefas (movimentos) no plano global de forma que tenha nenhum conflito, o agente compartilha o plano global modificado com os agentes no sistema.
5. Todos os agentes no sistema recebem a notificação de que o plano global foi alterado. Eles verificam se a versão do seu plano privado é diferente com a versão do plano global modificado. Caso seja, o agente atualiza seu plano.

O plano global não possui uma entidade centralizadora, ou seja, cada agente possui sua visão deste plano. De certa forma, o plano global será igual para todos os agente no ambiente, pois quando um agente altera o plano, ele o compartilha com os outros agentes.

Além disso, o plano global informado no algoritmo é representado por duas estruturas de dados. A primeira é um conjunto com todos os planos privados. Um plano privado é

representado por uma tabela hash - a chave é o instante de tempo t e o valor é o ponto x,y. Por exemplo, o agente A tem um plano privado que informa que no instante de tempo 13, ele deve se mover para o ponto (36, 15) e no tempo 14, ele deve se mover para para o ponto (36, 16).

A segunda estrutura de dados do plano global é uma tabela hash onde a chave é o instante de tempo t e o ponto (x,y); e o valor é o agente localizado nessa posição e nesse instante de tempo. Utilizando o exemplo acima, no instante de tempo 13 e ponto (36, 15), tem-se o agente A. Uma outra forma de armazenar essa informação seria uma matriz de três dimensões, mas manipular uma matriz como estrutura de dados seria mais difícil alterar os dados no caso de replanejamento.

Para que este algoritmo seja executado corretamente é essencial a sincronização dos agentes, ou seja, é necessário que todos os agentes estejam seguindo um mesmo relógio. Por ser um problema em tempo real, não existe margem para a realização da tarefa (movimento), caso um agente se atrase ou se adiante, todo o plano global seria afetado, principalmente com centenas de agentes. Por causa disso, é essencial a existência de um relógio global que todos os agentes obedeçam.

Outro detalhe importante deste algoritmo é que quando o agente A está coordenando seus planos, caso haja um segundo conflito, o agente A irá alterar o plano do agente B, no entanto, o agente A desconhece se o agente B também teve conflito e também estava esperando um instante de tempo, por causa disso, pode ocorrer do agente B esperar um segundo instante de tempo. Ou seja, não há garantia que um agente irá esperar somente um instante de tempo.

4. Testes

Para analisar a eficiência do algoritmo proposto, foi utilizado o JaCaMo, um framework para programação de sistemas multiagentes combinando três tecnologias: Jason, para programação de agentes autônomos; Cartago, para programação de artefatos de ambiente; e MOISE+ para programação de organizações multiagentes (JaCaMo, 2015). A partir dessa implementação, foi possível realizar os experimentos.

4.1 Descrição dos Experimentos

Com o intuito de validar a abordagem de planos compartilhados, foram criado dois tipos de cenários, um caso com poucos agentes e um caso com bastante agentes. Esses dois cenários foram colocados a prova com dois tipos de agentes, um agente com a capacidade de cooperação usando planos compartilhados e um outro agente que não sabe cooperar, precisando usar semáforos.

Em cada cenário proposto, há 12 rotas ativas. E cada uma delas, possui uma quantidade única de agentes. Cada rota possui também uma frequência para a entrada de um agente, representado na unidade de instante de tempo t.

Foram criados três cenários: o primeiro cenário teve um total de 100 agentes no sistema; no segundo cenário foram 400 agentes no sistema; e por fim, foi testado um cenário com 100 agentes no sistema mas com buracos que o obrigavam os agentes a replanejar suas rotas.

Importante destacar que não existe nenhum tipo de atraso para sair do sistema quando o agente chega em seu destino final. Ou seja, o próprio cruzamento está criando o trânsito.

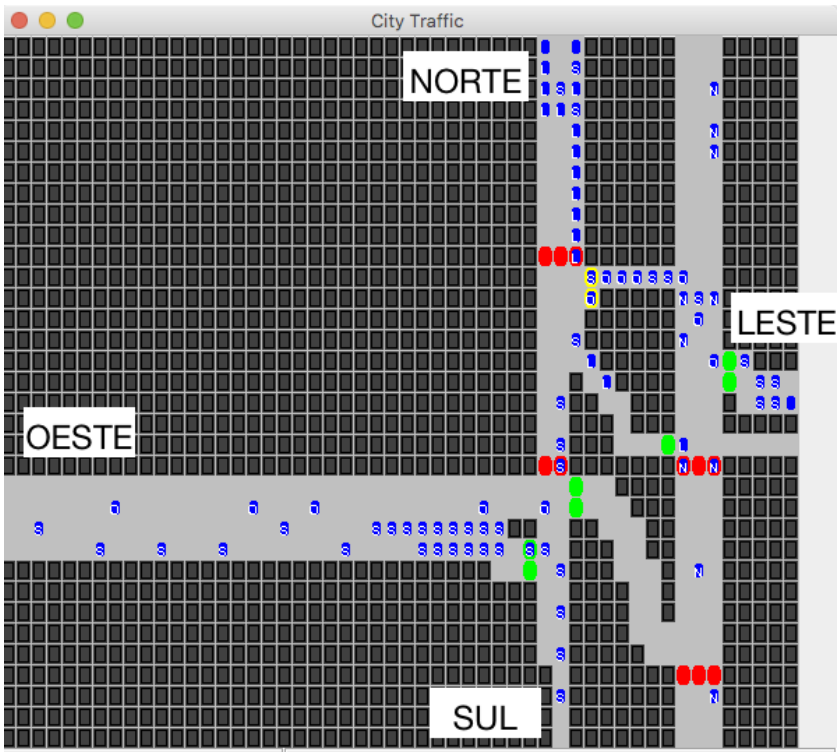


Figura 2. Ambiente criado no JaCaMo

Ambiente desenvolvido usando o JaCaMo no qual os agentes foram aplicados. As cores vermelho, verde e amarelo são o semáforo no seu estado atual. A cor azul representa o agente, a letra em cima do agente informa qual para onde o agente está indo (a primeira letra de cada ponto cardeal).

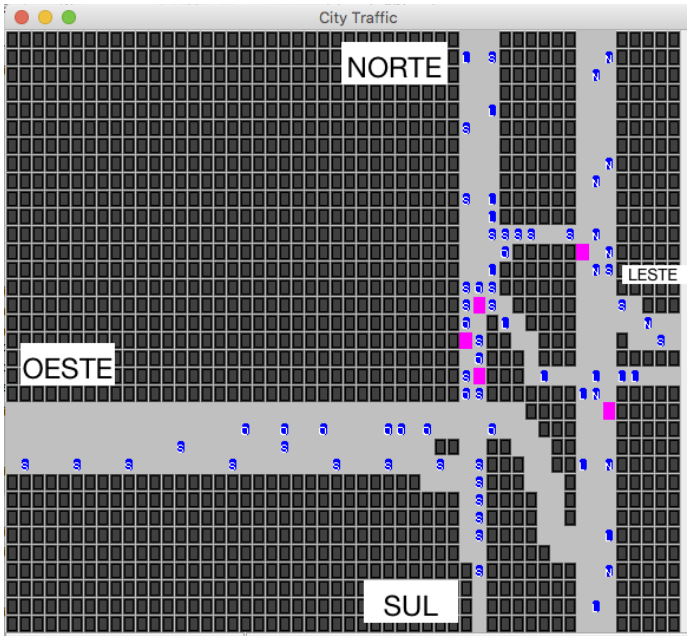


Figura 3. Ambiente criado no JaCaMo com buraco

Os buracos são obstáculos no ambiente (em rosa na figura acima) que o agente só descobre quando fica adjacente. O agente sem capacidade de cooperação, quando encontra um buraco, ele desvia e segue em frente. Já o agente com cooperação, ao identificar um buraco, ele altera os planos de todos os agentes que seriam afetados pelo buraco e compartilha o novo plano global para os outros agentes.

4.2 Resultados dos experimentos

Os dois cenários foram testados 5 vezes para os dois tipos de agentes. Foi coletado o instante de tempo t em que o agente entrou no sistema e em qual instante de tempo ele saiu do sistema. E com essas informações, foi feita análise abaixo.

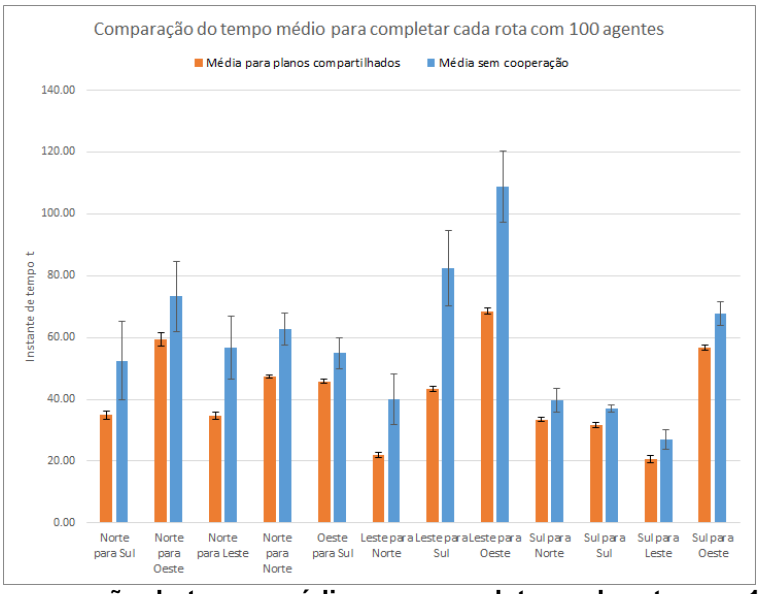


Figura 4. Comparação do tempo médio para completar cada rota com 100 agentes

No gráfico acima, temos uma comparação do tempo médio para cada rota no experimento com 100 agentes ao mesmo tempo. O primeiro destaque é que todas as rotas concluíram mais rapidamente com agentes capazes de cooperar entre si. Conseguindo quase metade do tempo de execução com que algumas das rotas, como Leste para Norte (54%) e Leste para Sul (52%).

Outra dado interessante é o desvio padrão elevado quando não há cooperação. Chegando a 12t no caso de Leste para Sul e no caso de Norte para Sul, já a abordagem de cooperação, o desvio padrão se mantém quase zero na maioria das rotas.

O comportamento do desvio padrão pode ser explicado pelo funcionamento de cada abordagem. No caso sem cooperação, os agentes são restritos pelo semáforo, caso o agente tenha a sorte de pegar somente semáforo verde, ele consegue terminar sua rota rapidamente, caso contrário ele terá que esperar o semáforo abrir. Essa característica torna o sistema incerto.

Já no caso de planos compartilhados, os agentes obedecem um algoritmo que define que cada agente só deve esperar um turno, por causa disso, um agente não fica dependendo da sorte. Criando um sistema mais determinístico.

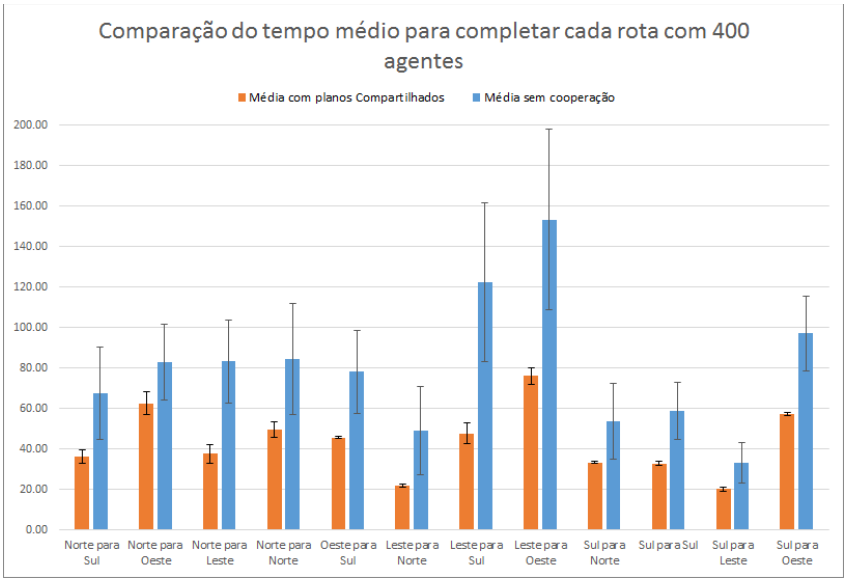


Figura 5. Comparação do tempo médio para completar cada rota com 400 agentes.

No experimento com 400 agentes a situação se mostra ainda uma maior vantagem de cooperação com planos compartilhados. A maioria das rotas apresentam cerca de metade do tempo médio e em Leste para Sul, os planos compartilhados termina em 39% do tempo.

A abordagem com semáforo mostra novamente um grande desvio padrão, apoiando novamente a hipótese apresentada anteriormente de que a existência de semáforos cria muita incerteza no tempo total que um agente levaria para chegar em seu destino.

O desvio padrão da abordagem com planos aumentou um pouco, mas ainda sim, nenhum desvio acima de 5t.

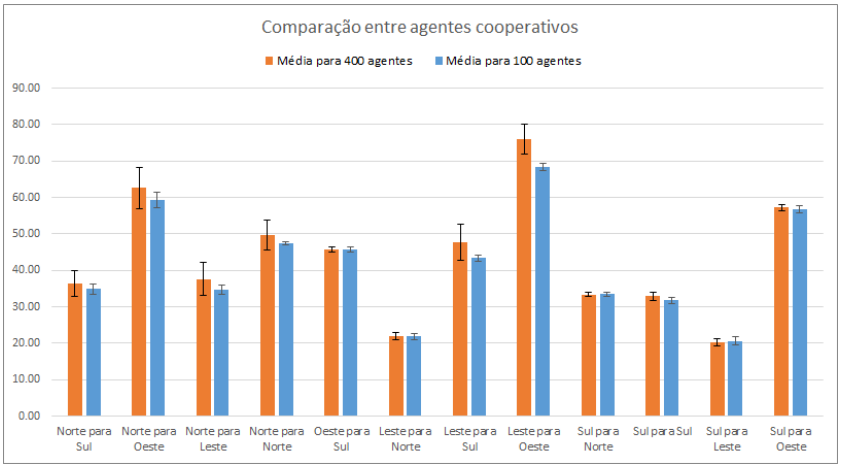


Figura 6. Comparação entre agentes cooperativos

E ao comparar os dados do agente cooperativo para o experimento com 100 agentes simultâneos e 400 agentes simultâneos, nota-se que houve um aumento do tempo entre 0% e 11%. Um aumento pequeno para 4 vezes mais agentes no sistema, embora talvez seja possível diminuir essa variação se os agentes evitarem bloquear agentes que estão indo para um destino diferente dele.

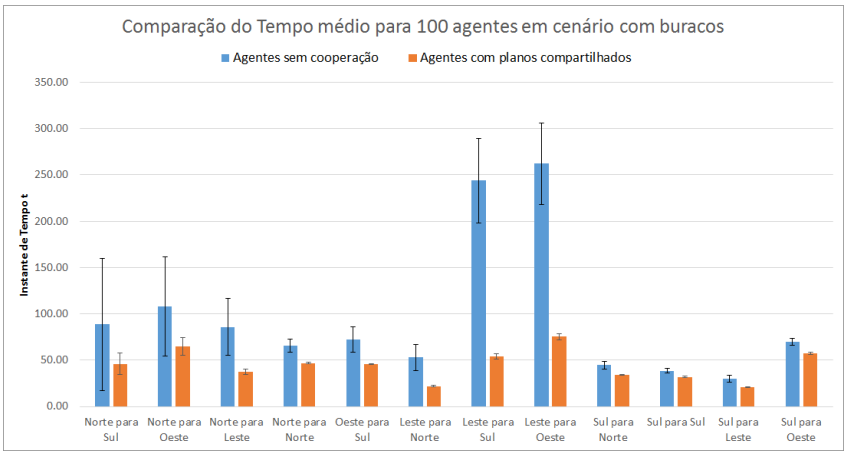


Figura 7. Comparação do Tempo médio para 100 agentes em cenário com buracos

O caso com buraco se repete os resultados. Os agentes com cooperação terminam as rotas com pouca dificuldade, principalmente pois o primeiro agente a encontrar o buraco já atualiza o plano global.

Por fim, depois de comparar e analisar os resultados dos experimentos, é possível considerar a abordagem baseada em planos compartilhados um grande avanço comparado com agentes sem capacidade de cooperação. No entanto, a execução de mais casos de testes, com mais e menos agentes, e com uma frequência de agentes diferente, apresentaria uma diversidade maior de informações sobre a atuação dos agentes.

5. Conclusão

Embora várias melhorias poderiam ser realizadas no algoritmo para compartilhamento de planos globais, ou mesmo melhorias na implementação do sistema no JaCaMo, este trabalho conseguiu expor os vários problemas na cooperação entre agentes e propôs algumas abordagens para solucionar alguns desses problemas.

O estudo em sistemas multiagentes e a análise das abordagens aplicadas a cooperação serviram como base para o desenvolvimento deste trabalho. Com esse conhecimento foi possível criar um cenário que poderia se beneficiar da cooperação de agentes. Além disso, as abordagens estudadas auxiliaram a identificar qual delas poderia ajudar a solucionar o cenário proposto. Como resultado, foi desenvolvida uma abordagem baseada em planos compartilhados. Para conseguir testá-la, a mesma foi implementada no JaCaMO e comparada com agentes sem capacidade de cooperar.

Os resultados dos testes, embora simples, demonstra que o agente com cooperação apresenta melhores resultados globais. No entanto, a capacidade de cooperação no cenário proposto é limitado principalmente pela quantidade de agentes no sistema. Ao realizar testes com mais de um mil agentes, os agentes não tinham vantagem no planejamento, pois não existia variação na

movimentação. E pelo fato que a resolução de conflito não é eficiente, um carro poderia ficar esperando vários instantes de tempo em um cruzamento.

A resolução de conflito não é eficiente, pois um agente pode fazer um outro agente esperar mais de dois instante de tempo; e por causa disso, atrasar vários outros agentes. A resolução de conflitos teria que analisar as rotas dos outros agentes antes de alterar o plano e tentar encontrar uma solução em que o menor número de agentes sejam afetados, mas sem que um determinado agente fique esperando.

Este trabalho demonstrou o funcionamento de uma abordagem baseada em planos compartilhados para cooperação de agentes. E apresentou também os problemas e soluções encontrados na implementação dessa abordagem no JaCaMO, uma plataforma que pode ser utilizada para vários implementações de SMA.

6. Referências

ECK, Adam; SOH, Leen-Kiat; To Ask, Sense, or Share: Ad Hoc Information Gathering; Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), 2015

PUJOL-GONZALEZ, Marc, CERQUIDES, Jesus; FARINELLI, Alessandro, MESEGUER, Pedro, RODRIGUEZ-AGUILAR, Juan A. ;Efficient Inter-Team Task Allocation in RoboCup Rescue, Proceedings of the 14th **International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015) , 2015.

WICKE, Drew, FREELAN, David, LUKE, Sean, Bounty Hunters and Multiagent Task Allocation, **Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015).

FARINELLI, Alessandro, MARCHI, Nicolo', RAEISSI, Masoume M., BROOKS, Nathan, SCERRI, Paul, A Mechanism for Smoothly Handling Human Interrupts in Team Oriented Plans, **Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015), 2015.

ALECHINA, Natasha, BULLING, Nils, DASTANI, Mehdi; Practical Run-Time Norm Enforcement with Bounded Lookahead, Proceedings of the 14th International Conference on **Autonomous Agents and Multiagent Systems (**AAMAS 2015)

KING, Thomas C.; LI, Tingting; VOS, Marina De, DIGNUM, Virginia; JONKER, Catholijn M., PADGET, Julian, M. RIEMSDIJK, Birna van; A Framework for Institutions Governing Institutions, **Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems** (AAMAS 2015)

JaCaMo Project Website, disponível em: < http://jacamo.sourceforge.net/ > Acesso em 04.07.2016