



Civilization Project

Agentes e Inteligência Artificial Distribuída
Mestrado Integrado em Engenharia Informática e Computação

Amadeu Pereira - up201605646

Diogo Yaguas - up201606165

João Lima - up201605314

Descrição do problema

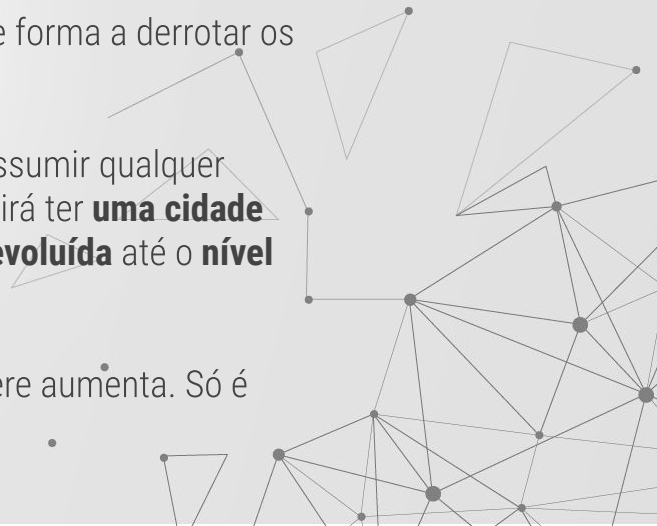
O **jogo de estratégia** consiste em diversas comunidades conseguirem **apoderar-se** das outras através de **diversas estratégias**.

Cada comunidade irá **receber dinheiro passivamente**, diretamente proporcional ao número de cidades que possui. Ao mesmo tempo, as comunidades poderão **comprar recursos** a troco de dinheiro.

Será necessário que **giram** e façam **uso dos recursos / dinheiro** de forma a derrotar os oponentes.

O mapa do jogo está **dividido** numa grelha de **$N \times M$** no qual pode assumir qualquer tamanho, sendo cada **espaço** da grelha é uma **cidade**. Cada comunidade irá ter **uma cidade inicialmente**. Cada cidade, no início do jogo, começa do zero e pode ser **evoluída** até o **nível máximo de 10**, a troco de dinheiro.

À medida que a cidade evolui, a quantidade de dinheiro que esta gere aumenta. Só é possível **interações entre cidade adjacentes**.



ESQUEMA GLOBAL

01

**INTERAÇÃO E
PROTOCOLOS**

02

**ARQUITETURAS E
ESTRATÉGIAS**

03

ÍNDICE

04

**OUTROS MECANISMOS E
SOFTWARE UTILIZADO**

05

**EXPERIÊNCIAS
REALIZADAS E
RESULTADOS**

06

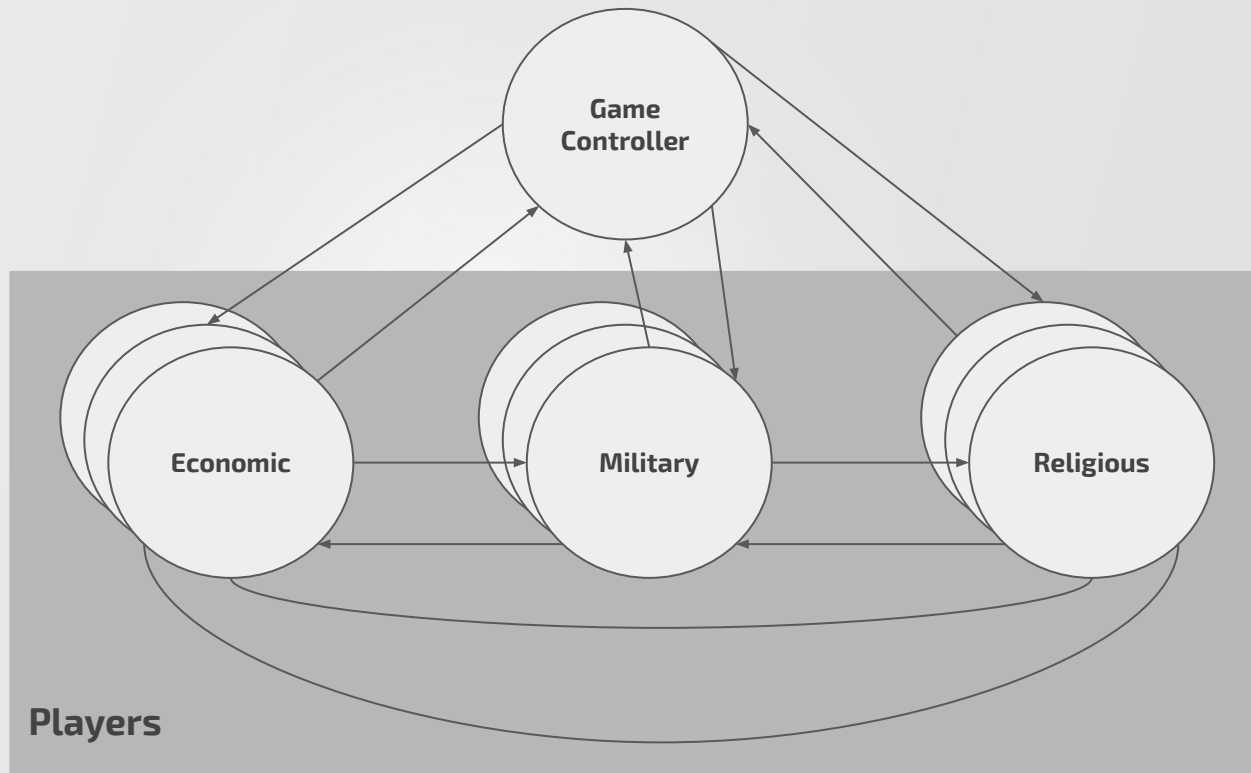
CONCLUSÕES

01

ESQUEMA GLOBAL

Foram considerados 4 tipos de agentes:

- **Game Controller;**
- **Economic;**
- **Military;**
- **Religious.**





02

INTERAÇÃO E PROTOCOLOS

Interação Controller - Player

Controller utiliza protocolo **Subscription Initiator** para ser notificado sempre que um *player* se juntou ao jogo. Além disso, o Controller está responsável por **informar o jogador** da sua **vez de jogar** (Turn) e responde a todas as **perguntas** que o *player* pode colocar (Request).

No que toca às interações entre agentes, são utilizadas **ACL Messages**. As mensagens são enviadas e recebidas em behaviours como:

- FSMBehaviour** ●
- Simple Behaviour** ●
- CyclicBehaviour** ●
- ParallelBehaviour** ●

Interação Player - Player

Um *Player* **questiona e informa** outros *players* sobre **ações que poderá ou irá realizar**, como por exemplo, preço de uma cidade, que irá realizar um ataque, que irá comprar a cidade ou que a cidade se converteu a si.



03

ARQUITETURAS E ESTRATÉGIAS

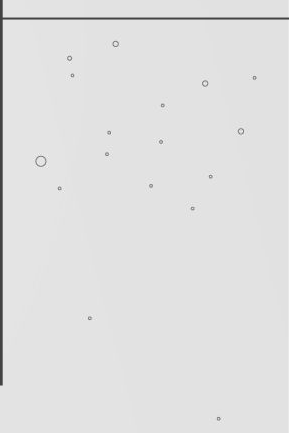
Game Controller

Escolher a informação que passa aos jogadores: Primeiramente passa a informação que é a sua vez de jogar. De seguida, quando questionado, informa sobre as cidades adjacentes (se estão vazias ou ocupadas).

Atualizar o estado do jogo: Quando a vez de um jogador termina, recebe as informações sobre as novas cidades adquiridas e atualiza o estado do jogo e, consequentemente, a gui para rondas futuras.

Players

- 1. Escolher cidades vazias que comprar:** O jogador compra as cidades vazias adjacentes
- 2. Aumentar defesas das próprias cidades:** Das próprias cidades o jogador tem a possibilidade de aumentar as defesas de forma a poder proteger-se contra ataques.
- 3. Atacar cidades adversárias:** Dependendo da estratégia adotada o jogador pode atacar as cidades oponentes adjacentes.
- 4. Contra-atacar religiosamente:** Dependendo do nível de religião dos adversários, o jogador pode aumentar o seu nível religioso nas suas cidades.
- 5. Melhorar o nível das suas cidades:** O jogador tem possibilidade de aumentar o nível das cidades.

- **Ordem do Economics:** 1 - 4 - 3 (Comprar cidades adversárias) - 2 - 5;
 - **Ordem do Military:** 2 - 3 (Atacar com as suas defesas as cidades adversárias) - 1 - 5
 - **Ordem do Religious:** 4 - 3 (Converter cidades adversárias) - 2 - 1 - 5;
- 

04

OUTROS MECANISMOS E SOFTWARE UTILIZADO

O projeto foi desenvolvido em **IntelliJ IDEA** com acesso à framework **JADE**.

Quanto a outros mecanismos, foi implementado uma **estrutura de descoberta de agentes Directory Facilitator (DF)**, onde o agente Game Controller se inscreveu é notificado quando um jogador se junta.



05

EXPERIÊNCIA REALIZADAS E RESULTADOS

1ª Experiência - 10 jogos com mapa de iguais dimensões e 1 jogador de cada estratégia (3 jogadores).

- **100% das vitórias** foram da **estratégia Economics**;
- Primeiro a perder foi **100% da estratégia Religious**;

2ª Experiência - 10 jogos com mapa de iguais dimensões e 2 jogadores de iguais estratégias

- **Economics**: Empate nos 10 jogos ganhando aquele com mais cidades;
- **Religious**: Empate nos 10 jogos ganhando aquele com mais cidades;
- **Military**: 60% das vitórias foram do jogador 1.

3ª Experiência - 10 jogos com mapa de iguais dimensões em 1vs1

- **Military vs Religious**: 100% das vitórias foram da estratégia Military;
- **Military vs Economics**: 80% das vitórias foram da estratégia Economics;
- **Economics vs Religious**: 100% das vitórias foram da estratégia Economics;

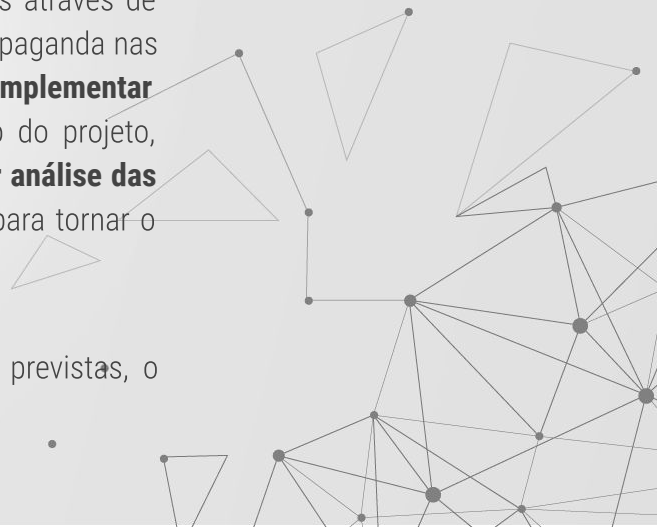
06

Conclusões

O **primeiro projeto** da unidade curricular de **Agentes e Inteligência Artificial Distribuída** teve como objetivo implementar um sistema composto por múltiplos agentes (SMA).

No início do projeto o grupo tinha pensado num **jogo de estratégia** de conquista de comunidades que seriam caracterizadas por **diversas estratégias**. **Economics** tem como objetivo comprar cidades vazias bem como a dos adversários. **Military** obteria as suas cidades através de ataques às mesmas com o uso das defesas das suas cidades. **Religious** recorreria à propaganda nas cidades adversárias de forma a as conquistar. Infelizmente, **não nos foi possível implementar algumas das funcionalidades** de cada uma das estratégias explicitadas na definição do projeto, sendo algo que pretendemos fazer no próximo trabalho, bem como, realizar uma **melhor análise das experiências realizadas** e repensar os valores utilizados em cada uma das estratégias para tornar o jogo o mais equilibrado possível.

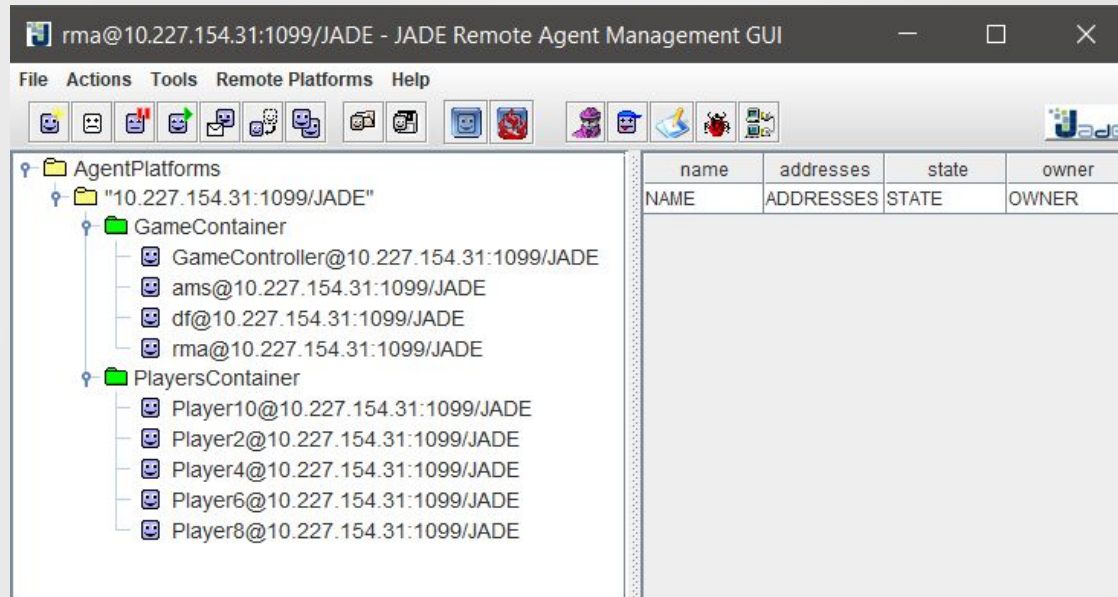
Em suma, apesar de não terem sido desenvolvidas todas as funcionalidades previstas, o projeto foi finalizado com sucesso.



Exemplos detalhados de execução

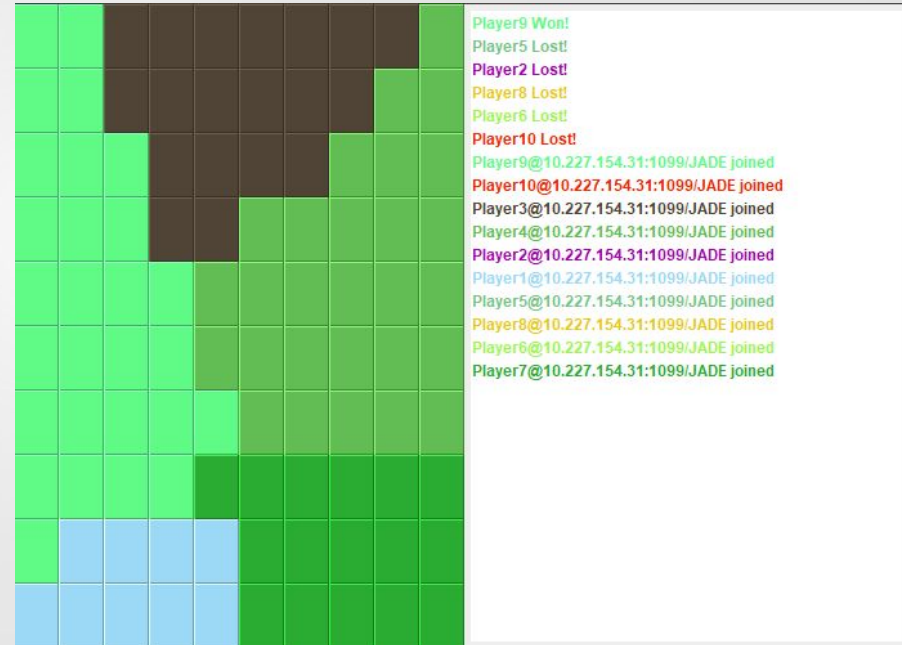
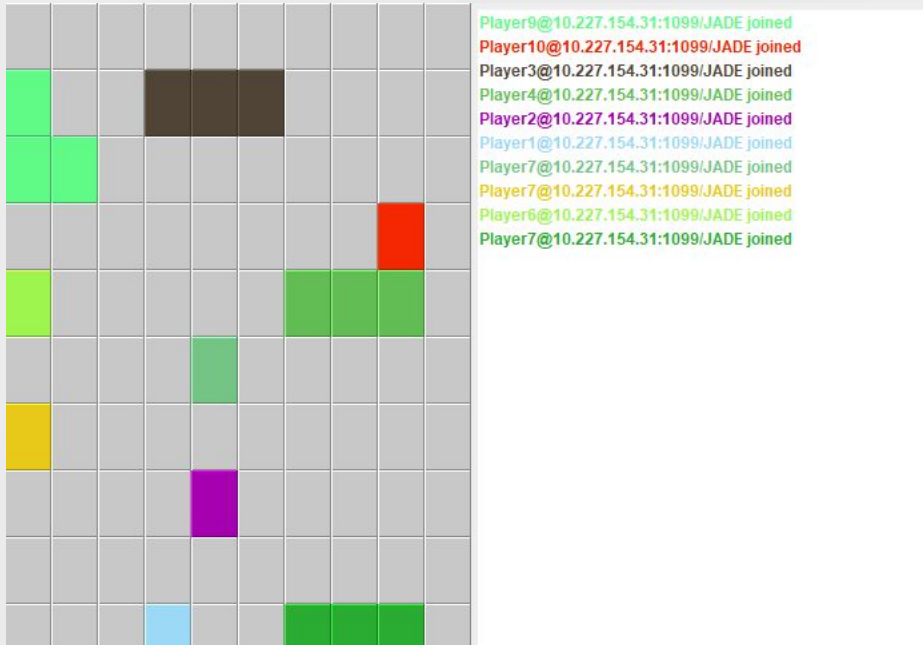
Durante a execução são apresentados dois ecrãs:

- Um ecrã será a **interface do Jade**;



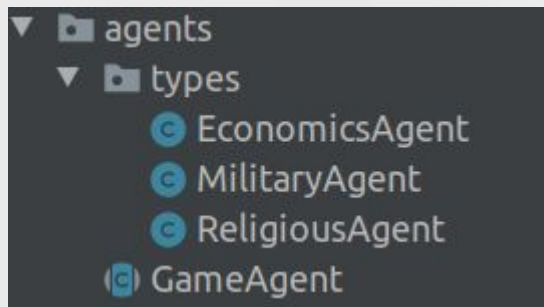
Exemplos detalhados de execução

- O último ecrã é uma **interface com o mapa do jogo** e uma mini consola. O mapa mostra as várias cidades e cada **cor diferente representa um agente diferente** e as cidades que lhe pertencem. A consola serve para poder acompanhar quem está a fazer o turno atual.



Classes implementadas

- **Agents.GameAgent** estende a class Agent do Jade. Este é a classe abstrata que deve ser estendida quando queremos implementar um novo tipo de jogador. Esta classe trata de receber o turno proveniente do Controlador e contém métodos de lógica comuns aos vários agentes. O único método abstrato é logic() que depende do tipo de jogador e do seu método de jogar.
- **Agents.types** é um pacote que contém os vários tipos de agentes e as suas estratégias.



```
/**
 * Buy empty cities with empty cities money.
 *
 * @param new_cities cities that managed to get.
 * @return empty cities that managed to buy.
 */
protected ArrayList<City> buyEmptyCities(ArrayList<City> new_cities) {...}

/**
 * Upgrade cities with upgrade money.
 */
protected void upgradeCities() {...}

/**
 * Wasted all defenses money upgrading evenly every city you own.
 */
protected void upgradeMyDefenses() {...}

/**
 * Tries to defend all cities from religion attacks using the money given.
 */
protected void defendReligion() {...}

/**
 * Logic of player's turn.
 *
 * @return list of new cities conquered.
 */
public abstract ArrayList<City> logic();
/**
 * Player's turn.
 */
private class ReceiveTurn extends CyclicBehaviour {
```

Classes implementadas

- **Agents.GameController** é a classe que controla a lógica do controlador de jogo. Este aceita os vários jogadores novos que vão entrar no jogo, atribuindo um posição no mapa. Ele também mantém um mapa com os donos da várias cidades, para depois poder informar os jogadores de quem são os donos daquelas cidades e como contactá-los. Também é o GameController que é responsável por correr a gui criada e de a atualizar.

```
* Setup game player.
*
* @param player player of the game.
*/
private void setupPlayer(AID player) {...}

/**
 * Create a new message to send.
 *
 * @param player receiver of the message.
 * @param content content of the message.
 * @param type type of message.
 */
private void msg(AID player, String content, int type) {...}

/**
 * Add message to GUI
 *
 * @param msg message to appear on GUI.
 */
private void addActionGUI(String msg) { if (gui != null) gui.addAction(msg); }

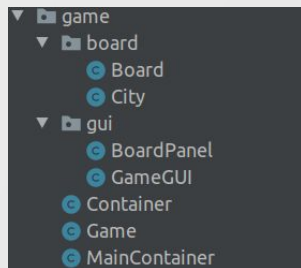
/**
 * Update board GUI.
 */
private void updateBoardGUI() { gui.setBoard(board); }

/**
 * Inform which player's turn.
 */
private class TurnBehaviour extends Behaviour {...}

/**
 * Receives questions of players and answers them.
 */
private class QueriesBehaviour extends Behaviour {...}
```

Classes implementadas

- **Game.Board.City** contem toda a informação sobre a cidade e métodos relacionados com a interação de jogador-cidade.
- **Game.Board.board** contém o mapa com as várias cidades nas suas posições indicadas.
- **Game.Gui** é uma pacote que contém todas as class relativas ao processamento da GUI.



```
private static int maximum_level = 10;
private AID owner;
private Coordinate my_cords;
private int city_price;
private int current_level;
private int cost_to_upgrade;

private int amount_money_producing;
private int defences;
private int amount_on_upgrade;

private ArrayList<Pair<AID, Integer>> religion_attacker;

/**
 * Create a new city.
 *
 * @param owner owner of the city
 * @param cord coordinates of the city.
 */
public City(AID owner, Coordinate cord) {...}

/**
 * Calculates how much money a percentage of this religion is worth
 *
 * @param value The percentage I want to calculate
 * @return The amount of money this percentage costs
 */
public int costOfReligion(int value) { return ((int) Math.ceil(this.amount_on_upgrade * 2 / 100)) * value; }
```

Outras observações

Ao iniciar o programa temos a possibilidade de **fornecer a quantidade de jogadores**, sendo necessário o valor estar **entre 2 e 10**, havendo uma correção de dados caso o **número não seja respeitado**, isto é, se for inferior a 2 o programa irá considerar o número mínimo, enquanto se for superior a 10 irá considerar o número máximo.

Para além disso, temos a **possibilidade de escolher as dimensões do mapa**, sendo que a altura e largura não necessitam de ser iguais, enquanto o **limite mínimo** tem de ser **superior à raiz quadrada do número de jogadores**.

