

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL  
FACULDADE DE INFORMÁTICA  
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**AGENTES PARA O JOGO  
COLONIZADORES DE CATAN**

**BRUNO PAZ E GABRIEL RUBIN**

Trabalho de Conclusão II apresentado  
como requisito parcial à obtenção  
do grau de Bacharel em Ciência da  
Computação na Pontifícia Universidade  
Católica do Rio Grande do Sul.

Orientador: Prof. Felipe Meneguzzi

**Porto Alegre  
2016**

## **LISTA DE SIGLAS**

IA – Inteligência Artificial

MCTS – *Monte Carlo Tree Search*

UCB – *Upper Confidence Bound*

UCT – *Upper Confidence Bounds for Trees*

AMAF – *All Moves As First*

RAVE – *Rapid Action Value Estimation*

# SUMÁRIO

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>INTRODUÇÃO</b>                              | <b>7</b>  |
| <b>2</b> | <b>TÉCNICAS DE IA PARA JOGOS</b>               | <b>8</b>  |
| 2.1      | TEORIA DE JOGOS PARA IA                        | 8         |
| 2.1.1    | REPRESENTAÇÃO                                  | 8         |
| 2.1.2    | CARACTERÍSTICAS                                | 10        |
| 2.2      | MINIMAX  | 10        |
| 2.2.1    | ALGORITMO MINIMAX                              | 11        |
| 2.2.2    | <i>ALPHA-BETA PRUNING</i>                      | 13        |
| 2.2.3    | <i>EVALUATION FUNCTION</i>                     | 15        |
| 2.3      | MÉTODOS DE MONTE CARLO PARA JOGOS              | 17        |
| 2.3.1    | MONTE CARLO <i>SIMULATION</i>                  | 18        |
| 2.3.2    | MONTE CARLO <i>TREE SEARCH</i>                 | 18        |
| 2.3.3    | <i>UPPER CONFIDENCE BOUNDS FOR TREES (UCT)</i> | 21        |
| 2.3.4    | MELHORIAS PARA UCT                             | 22        |
| <b>3</b> | <b>COLONIZADORES DE CATAN</b>                  | <b>26</b> |
| 3.1      | ELEMENTOS DO JOGO                              | 26        |
| 3.2      | ANDAMENTO DE UMA PARTIDA                       | 27        |
| 3.3      | RECURSOS                                       | 28        |
| 3.4      | CONSTRUÇÕES                                    | 29        |
| 3.5      | TROCA DE RECURSOS                              | 30        |
| 3.6      | NÚMERO 7 E OS LADRÕES                          | 30        |
| 3.7      | CARTAS DE DESENVOLVIMENTO                      | 31        |
| <b>4</b> | <b>AGENTES JOGADORES DE CATAN</b>              | <b>32</b> |
| 4.1      | AGENTE SIMPLES                                 | 32        |
| 4.1.1    | IMPLEMENTAÇÃO                                  | 32        |
| 4.2      | AGENTE MINIMAX                                 | 33        |
| 4.3      | AGENTE MCTS                                    | 34        |
| 4.3.1    | IMPLEMENTAÇÃO                                  | 34        |
| 4.4      | AGENTE UCT                                     | 35        |

|          |   |           |
|----------|---|-----------|
| 4.4.1    | IMPLEMENTAÇÃO .....                             | 35        |
| <b>5</b> | <b>CLIENTE .....</b>                            | <b>37</b> |
| 5.1      | <i>JSETTLERS</i> .....                          | 37        |
| 5.1.1    | INTERFACE .....                                 | 37        |
| 5.1.2    | TROCA DE MENSAGENS .....                        | 38        |
| 5.2      | ARQUITETURA .....                               | 39        |
| 5.3      | IMPLEMENTAÇÃO .....                             | 40        |
| 5.3.1    | SISTEMA DE MENSAGENS .....                      | 41        |
| 5.3.2    | REPRESENTAÇÃO DE ESTADOS .....                  | 41        |
| 5.3.3    | SISTEMA DE AÇÕES .....                          | 41        |
| 5.4      | FERRAMENTAS DE TESTE .....                      | 42        |
| 5.4.1    | <i>PROFILING</i> .....                          | 42        |
| 5.4.2    | VISUALIZADOR DE ESTADOS .....                   | 43        |
| <b>6</b> | <b>RESULTADOS - PERFORMANCE E ANÁLISE .....</b> | <b>45</b> |
| 6.1      | AGENTE SIMPLES .....                            | 46        |
| 6.2      | AGENTE MCTS .....                               | 47        |
| 6.3      | AGENTE UCT .....                                | 49        |
| 6.4      | ANÁLISE DE RESULTADOS .....                     | 52        |
| <b>7</b> | <b>CONCLUSÃO .....</b>                          | <b>54</b> |
|          | <b>REFERÊNCIAS .....</b>                        | <b>55</b> |

# AGENTES PARA O JOGO COLONIZADORES DE CATAN

## RESUMO

Colonizadores de Catan é um dos principais representantes dos jogos estratégicos modernos. O jogo possui muitas características que o tornam difícil de ser jogado por um agente: é um jogo estocástico, de informação imperfeita e com mais de 2 jogadores. Este jogo possui poucas implementações de agentes disponíveis, as quais não representam desafio para um jogador humano. Neste trabalho, vamos implementar agentes capazes de jogar Catan contra outros agentes existentes e jogadores humanos através de diferentes técnicas de IA para jogos: um agente com heurística simples, como ponto de partida, e outros agentes baseados em Monte Carlo Tree Search. Finalmente, vamos analisar e comparar a performance dessas técnicas no jogo contra o agente JSettlers, considerado uma referência de performance para este jogo, a fim de decidir qual delas é melhor para este jogo.

**Palavras-Chave:** Colonizadores de Catan, Minimax, Alpha beta pruning, Monte Carlo Tree Search, UCT.

# AGENTS FOR THE GAME SETTLERS OF CATAN

## ABSTRACT

Settlers of Catan is one of the main representatives of modern strategic board games. The game is challenging for AI players due to its features: it's a stochastic, imperfect information game with more than 2 players. There are few agents available for it, and they are not challenging against human players. In this paper, we will implement agents capable of playing Catan against other existing agents and human players, using distinct AI techniques for games: one agent with simple gameplay heuristics, and other agents based on the Monte Carlo Tree Search family of algorithms. Finally, we will analyze and compare the performance of these techniques against the JSettlers agent, considered to be a gameplay benchmark for this game, in order to decide which one of them is best suited for this game.

**Keywords:** Settlers of Catan, Minimax, Alpha beta pruning, Monte Carlo Tree Search, UCT.

## 1. INTRODUÇÃO

Jogos de tabuleiro representam um desafio para a comunidade de Inteligência Artificial. O estudo de jogos clássicos como Xadrez, Damas e Go foi importante para o desenvolvimento da área [SCS10]. Muitas técnicas de IA foram desenvolvidas para melhorar a performance de um agente nesses jogos clássicos [Mar87]. Apesar de lidar bem com jogos tradicionais, essas técnicas muitas vezes não são satisfatórias para jogos estratégicos modernos, comumente chamados de *eurogames*, devido à maior complexidade de grande parte desses jogos quando comparados a jogos de tabuleiro tradicionais [SCS10]. Técnicas recentemente desenvolvidas para jogar jogos complexos melhoraram expressivamente a performance de um agente no clássico jogo Chinês Go [GKS<sup>+</sup>12], e trazem novas possibilidades de progresso para agentes que jogam jogos estratégicos modernos [CBSS08].

Jogos estratégicos modernos são de grande interesse para a comunidade de IA devido às características que esses jogos dividem com jogos de tabuleiro clássicos e jogos de videogame [CBSS08]. Escolhemos Colonizadores de Catan para este trabalho por ser um jogo que representa bem o arquétipo de um *eurogame* [SCS10] e por possuir elementos e características de jogo que o tornam desafiador: dados, cartas, mais de 2 jogadores e troca entre os jogadores. Existem implementações de agentes disponíveis para Catan, mas elas não representam desafio para jogadores experientes [CBSS08], ou seja, jogadores com um conhecimento avançado em estratégias vencedoras do jogo. Neste trabalho, implementamos toda a lógica de jogo e jogadores de Colonizadores de Catan, e 2 desses jogadores são competitivos contra o agente JSettlers, considerado uma referência de performance para esse jogo [SCS10]. Nosso projeto é *open-source* e pode ser encontrado no endereço: <https://github.com/GabrielRubin/TC2>.

Este documento é organizado da seguinte forma: No Capítulo 2, introduzimos conceitos de IA para jogos e explicamos em detalhes as técnicas de IA que serão consideradas para este trabalho, além de apontar melhorias e aperfeiçoamentos para estas técnicas; Detalhamos o jogo Colonizadores de Catan e suas regras no Capítulo 3; Explicaremos os agentes implementados no Capítulo 4; Mostramos como foi desenvolvido o cliente de jogo no Capítulo 5; No Capítulo 6, analisamos os resultados obtidos; Finalmente, no Capítulo 7, apresentamos nossa conclusão para este trabalho.

## 2. TÉCNICAS DE IA PARA JOGOS

Este capítulo apresenta teoria de jogos para IA e detalha as técnicas que serão consideradas neste trabalho para jogar Catan. A primeira parte do capítulo define como podemos representar um jogo formalmente. Esse formalismo serve de base para explicarmos os algoritmos de IA para jogos das seções seguintes: os baseados em Minimax e os baseados em Métodos de Monte Carlo.

### 2.1 Teoria de Jogos para IA

Teoria de Jogos [FT91] é um ramo da matemática aplicada que estuda formalmente o cenário de conflito e cooperação entre agentes racionais, e também formaliza diversos aspectos de jogos como: estratégias, tomada de decisão, e a representação formal de um jogo. Esses formalismos servem como base para a pesquisa de jogos no ramo da Inteligência Artificial. Vamos abordar aqui alguns formalismos que serão importantes para explicar outros conceitos que serão apresentados neste trabalho.

#### 2.1.1 Representação

Podemos representar jogos competitivos de maneira abstrata como um ambiente composto por diversos estados onde 2 ou mais agentes competem entre si para vencer o jogo [RN10, Cap 5, pp161-163]. Para nossa representação, utilizaremos a seguinte notação:

- $S$ : é o conjunto de estados do jogo, onde  $s_0$  é o estado inicial
- $S_T \subset S$ : é o conjunto de estados terminais do jogo
- $Player(s)$ : é uma função que define qual jogador  $p$  deve jogar no estado  $s$
- $Actions(s)$ : é uma função que retorna o conjunto de ações  $A$  permitidas a partir de um estado  $s$
- $Result(s, a)$ : é uma função de transição de estados. A partir de um estado  $s$  e uma ação  $a$ , retorna um estado  $s'$  resultante desta ação
- $TerminalTest(s)$ : é uma função que verifica se um estado é terminal. Retorna *True* se um estado  $s \in S_T$  e *False* caso contrário

- $Utility(s, p)$ : é uma função de utilidade, que calcula o valor de utilidade de um estado terminal  $s$  para um jogador  $p$

Uma partida começa do estado inicial  $s_0$  e segue até atingir um estado terminal  $s_T$ . A transição entre os estados de jogo é feita pela função de transição, a partir da ação escolhida pelos jogadores: em cada estado intermediário  $s_i$  entre  $s_0$  e  $s_T$ , um jogador  $p$  definido por  $Player(s_i)$  toma uma ação  $a_i \in Actions(s_i)$  e atualiza o estado de jogo  $s_i$  para o estado  $s_{i+1}$  através de  $Result(s_i, a_i)$ . A estratégia (*política*) de cada jogador determina qual ação  $a$  ele irá tomar a partir de um estado  $s$ . Para vencer a partida, os jogadores buscam atingir um estado terminal  $s_T \in S_T$  de maior utilidade, definida por  $Utility(s_T, p)$ . Um valor definido pela função de utilidade é considerado de maior utilidade em caso de vitória, de menor em caso de derrota e meio termo em caso de empate (+1, -1 e 0, respectivamente) [RN10, Cap 5, pp161-163].

A partida pode ser representada como uma árvore, chamada de *árvore de jogo* [RN10, Cap 5, pp162], onde os nodos representam os estados de jogo  $s$ , e as arestas representam as ações  $a$  que levam de um estado a outro. Os níveis da árvore representam o jogo após a jogada de cada jogador, esses níveis são chamados de *ply* e representam um *turno* da partida [RN10, Cap 5, pp163-165]. A Figura 2.1 mostra um exemplo de árvore de jogo para uma partida do jogo *Tic Tac Toe*, conhecido no Brasil como *jogo da velha*.

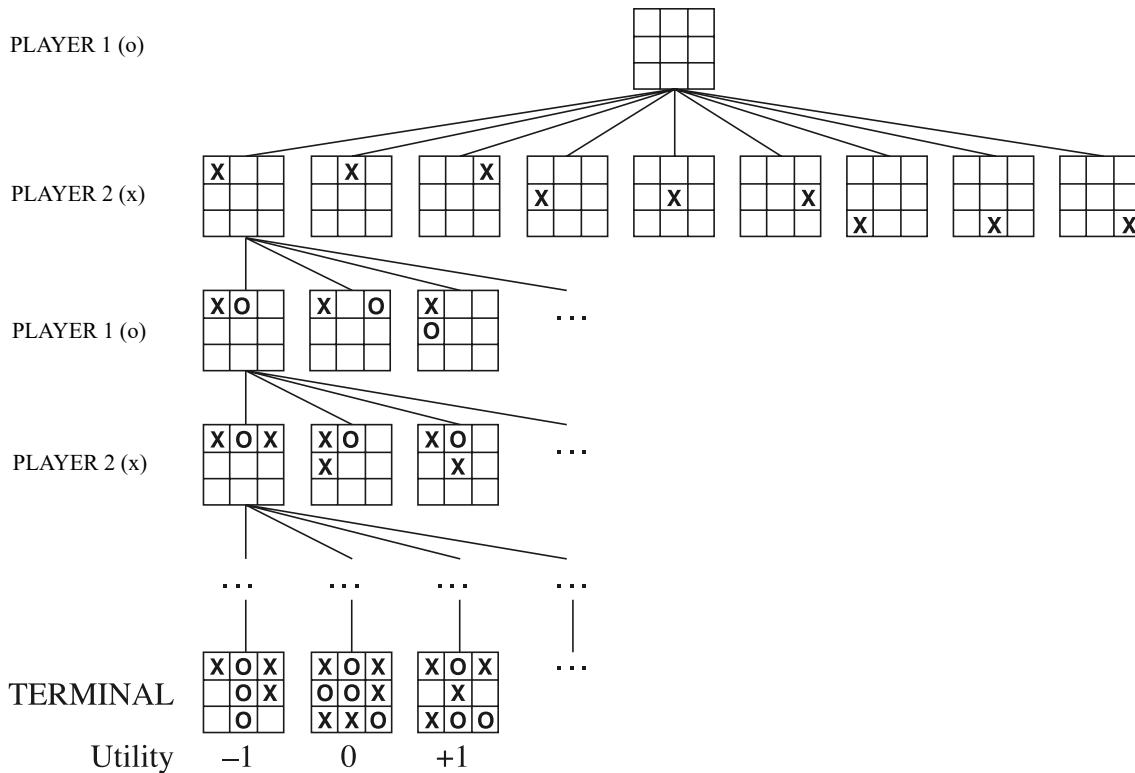


Figura 2.1: árvore de jogo para o *jogo da velha* [RN10, Cap 5, pp163].

### 2.1.2 Características

A Teoria de Jogos define que jogos podem ser divididos em diversas categorias de acordo com suas características [MF09], o que ajuda a classificá-los e determinar a complexidade no desenvolvimento de um agente capaz de jogá-los. Nos parágrafos abaixo, abordamos algumas das classificações que serão importantes para o nosso trabalho.

Jogos determinísticos são aqueles que não possuem nenhum elemento de chance dentro de suas regras, como o rolar de dados para a definição de alguma ação. Jogos que possuem tais elementos são estocásticos, e são mais complexos pela dificuldade de se considerar diversos futuros possíveis para o jogo.

Em jogos de informação perfeita, todas as informações sobre o estado de jogo estão disponíveis para todos os jogadores. Quando existem informações às quais os jogadores não têm acesso, como cartas viradas para apenas um dos jogadores, o jogo é classificado como de informação imperfeita. Estes jogos apresentam maior desafio para os jogadores, já que eles não podem definir com precisão o estado atual do jogo, apenas estimá-lo.

Um jogo é de soma-zero quando a soma final da pontuação de todos os jogadores envolvidos é sempre zero e sempre existe um vencedor. O ganho obtido por um jogador se traduz em perda para os outros jogadores e sempre é possível medir com precisão qual dos jogadores está ganhando.

O número de jogadores também influencia na complexidade de um jogo. Quanto maior o número de jogadores, maior será a árvore de jogo, já que ela deve representar estados e ações para cada jogador, o que aumenta a complexidade do problema.

Apresentaremos a seguir técnicas de IA para jogos que se baseiam nos conceitos vistos nesta seção. Utilizaremos as funções mostradas nesse capítulo nos algoritmos que compõem estas técnicas.

## 2.2 Minimax

Minimax é um algoritmo que serve de base para grande parte dos algoritmos de busca em árvore de jogo [vdW05]. Sua função é encontrar a estratégia ótima, a fim de vencer a partida, construindo uma árvore de jogo que modela a competição entre dois jogadores em um jogo de soma-zero [vNM44].

Em um jogo de dois jogadores, jogando competitivamente, um jogador busca a jogada de maior utilidade em seu turno, *MAX*, enquanto o oponente escolhe a jogada de menor utilidade para o jogador no outro turno, *MIN*. A utilidade é dada nos estados terminais

do jogo, como visto na seção anterior. Já os estados intermediários da árvore de jogo não representam vitória nem derrota, e para se descobrir a utilidade de um nodo intermediário, é necessário expandir as sub-árvores até os nodos terminais, para se entender o impacto de uma decisão antes do final de uma partida.

### 2.2.1 Algoritmo Minimax

O algoritmo Minimax calcula o valor de utilidade para os estados intermediários da árvore de jogo, o Minimax *value*, a partir da utilidade conhecida dos estados finais de jogo [KT53]. Por exemplo, na Figura 2.2.(a), de um estado de jogo  $s_0$ , o jogador pode escolher entre 3 jogadas, cada uma leva para um estado de jogo onde a escolha pertence ao oponente *MIN*:  $s_1$ ,  $s_2$  e  $s_3$ . Em cada um desses estados, o oponente pode atingir 3 estados terminais, a partir de 3 jogadas possíveis em cada estado, e entre essas 3 jogadas, o oponente escolhe aquela que leva ao estado terminal com a menor utilidade para o jogador. O Minimax *value* dos estados em que o oponente jogou,  $s_1$ ,  $s_2$  e  $s_3$ , é atualizado com base na utilidade do estado terminal atingido pela sua jogada, Figura 2.2.(b). O jogador então, partindo do estado  $s_0$ , escolhe a jogada que leva ao estado com o maior Minimax *value* dentre os estados  $s_1$ ,  $s_2$  e  $s_3$ , e atualiza o seu Minimax *value*, de acordo com o Minimax *value* do estado atingido pela sua jogada, Figura 2.2.(c) [RN10, Cap 5, pp164].

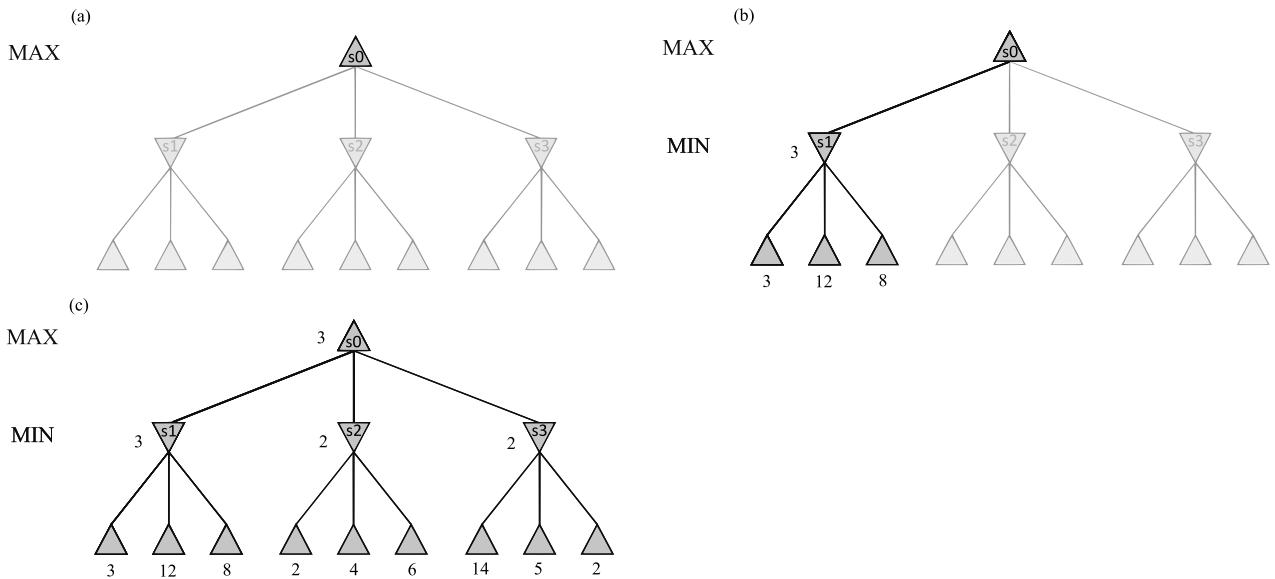


Figura 2.2: Ilustração da árvore gerada pelo Minimax.

O Algoritmo 2.1 percorre a árvore de jogo recursivamente, visitando os estados possíveis, gerados pela função *Result*, a cada recursão. A recursão termina quando a função *TerminalTest* verifica que o estado visitado é terminal, e retorna a utilidade deste estado de acordo com a função *Utility*.

---

**Algorithm 2.1** Minimax decision [RN10, Cap 5, pp164]

---

```

1: function MINIMAX-DECISION( s ) returns an action
2:   return  $\text{argmax}_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$ 

3: function MIN-VALUE( s ) returns a utility value
4:   if TERMINALTEST(s) then
5:     return UTILITY(s, p)
6:   value =  $\infty$ 
7:   for each a in ACTIONS(s) do
8:     value =  $\text{MIN}(\text{value}, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
9:   return value

10: function MAX-VALUE( s ) returns a utility value
11:   if TERMINALTEST(s) then
12:     return UTILITY(s, p)
13:   value =  $-\infty$ 
14:   for each a in ACTIONS(s) do
15:     value =  $\text{MAX}(\text{value}, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
16:   return value

```

---

Esse algoritmo considera apenas dois jogadores [vNM44] e não pode ser usado para jogos como Catan, que possui mais de dois jogadores. O que é feito, nesse caso, é adaptar o algoritmo para permitir tal situação [RN10, Cap 5, pp165]. Com dois jogadores, levamos em consideração apenas as jogadas do jogador como MAX em um *ply* e do seu oponente como MIN no *ply* seguinte e assim sucessivamente, utilizando um valor que representa a utilidade do jogador. Para um jogo com mais jogadores, podemos utilizar um vetor para representar a utilidade de cada jogador em um estado terminal, e cada jogador busca maximizar a sua utilidade em seu *ply*. A Figura 2.3 mostra o início de uma árvore de um jogo de 3 jogadores.

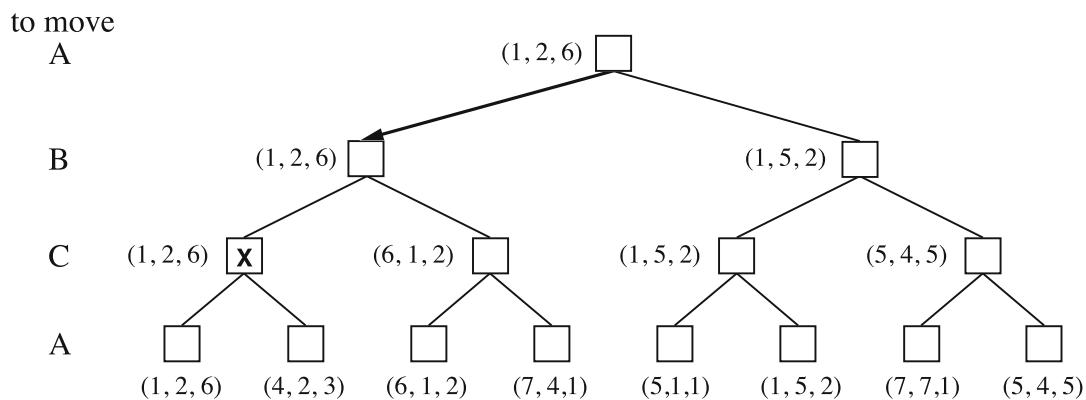


Figura 2.3: Exemplo de uma árvore gerada pelo Minimax para mais de dois jogadores (*A*, *B* e *C*) [RN10, Cap 5, pp165-167].

## 2.2.2 Alpha–beta pruning

Um dos grandes problemas do Minimax é sua passagem por toda a árvore de jogo, ou seja, se um jogo possui muitos estados, o algoritmo examina todos eles, e isso impacta diretamente a performance do algoritmo. Podemos reduzir a quantidade de estados examinados pelo algoritmo, *cortando* caminhos de nodos que não serão relevantes para o resultado antes de examiná-los [HE61].

Considere a equação simplificada do algoritmo Minimax em 2.1, que mostra o resultado do Minimax para o estado raiz  $s_0$  da árvore de jogo representada na Figura 2.4. Note que os dois estados terminais sucessores de  $s_2$ ,  $x$  e  $y$ , não influenciam o resultado da equação, pois o valor  $z$  de  $s_2$  será sempre  $\leq 2$ , devido a função  $\min(2, x, y)$ , e será desconsiderado pela função  $\max(3, z, 2)$  durante o *ply* de MAX. Se o algoritmo realiza a pesquisa da esquerda para a direita, uma vez que o primeiro sucessor de  $s_2$  for calculado, com o valor 2, sabemos que os valores de  $x$  e  $y$  podem ser desconsiderados, pois o valor de  $s_0$  não será maior que 3, já calculado após examinar  $s_1$ , logo, podemos desconsiderar  $x$  e  $y$ , sem prejudicar o resultado final do algoritmo [RN10, Cap 5, pp167].

$$\begin{aligned}
 \text{Minimax}(s_0) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\
 &= \max(3, \min(2, x, y), 2) \\
 &= \max(3, z, 2) \quad \text{onde, } z = \min(2, x, y) \leq 2 \\
 &= 3
 \end{aligned} \tag{2.1}$$

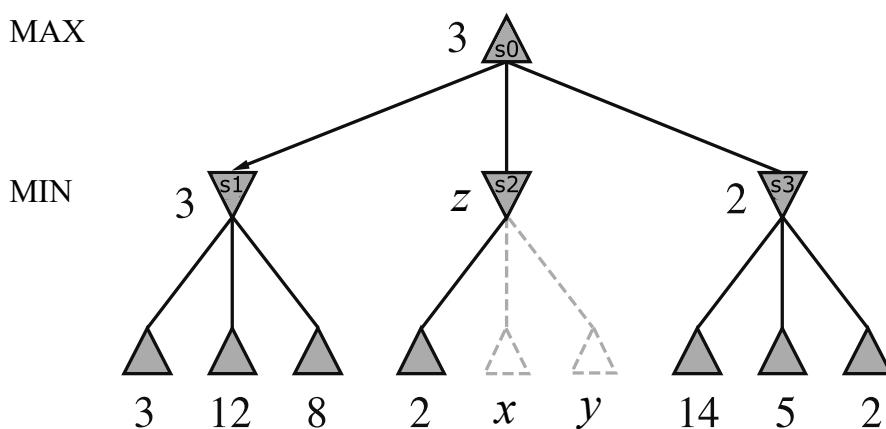


Figura 2.4: Exemplo de árvore de jogo onde pode haver corte: não é necessário examinar  $x$  e  $y$ .

A técnica *alpha-beta pruning* é uma modificação do algoritmo Minimax original [HE61], na etapa de minimização e maximização de estados, e é nesta etapa que é re-

alizada a decisão de corte. São adicionados ao algoritmo do Minimax 2 valores que são atualizados conforme o algoritmo percorre a árvore:

- $\alpha$  = é o valor do melhor caminho já explorado até a raiz para o jogador (*MAX*).
- $\beta$  = é o valor do melhor caminho já explorado até a raiz para o oponente (*MIN*).

A execução do algoritmo *Alpha–beta pruning* em uma árvore de jogo é representada pela Figura 2.5, onde inicialmente  $\alpha$  e  $\beta$  são iniciados com  $-\infty$  e  $+\infty$  respectivamente, como mostra a Figura 2.5.(a). A árvore é ramificada da esquerda para a direita, até chegar a um estado terminal onde  $Utility(s, p) = 3$ . O estado  $s_1$  está em um *ply* onde o jogador *MIN* joga, e como por enquanto não há valores menores que 3 gerados a partir de  $s_1$ , o algoritmo altera o valor de  $\beta$  para 3. Na Figura 2.5.(b) mais uma jogada a partir do estado  $s_1$  é ramificada e avaliada, mas seu valor é maior que o gerado anteriormente, e como o oponente busca o menor valor de utilidade para o jogador, este valor não pode ser considerado para  $\beta$ . Finalmente todas as jogadas de  $s_1$  são ramificadas e avaliadas, como mostra a Figura 2.5.(c), e o valor para *MIN* a partir do estado  $s_1$  é atualizado para 3. Como não existem mais jogadas possíveis a partir de  $s_1$ , a recursão volta para  $s_0$  e é atribuído o valor 3 para  $\alpha$ , pois  $s_0$  é um estado de *MAX*, e por enquanto não há valores maiores que 3 para seus sucessores.

Na Figura 2.5.(d) todos os estados a partir do estado  $s_1$  são conhecidos, e apenas 1 do estado  $s_2$ , com valor de utilidade 2. Como  $\alpha$  é maior que 2, o algoritmo aplica o corte, visto que  $\alpha$  não pode ser menor que 3, seguindo para o próximo estado a partir de  $s_0$ . Na Figura 2.5.(e) um estado terminal do estado  $s_3$  é avaliado em 14, e como 14 é maior  $\alpha$  (melhor caminho para  $s_0$  até então) é preciso continuar as ramificações neste estado. Os dois estados restantes de  $s_3$  são avaliados em 5 e 2 respectivamente, como mostra a 2.5.(f). A utilidade do estado  $s_3$  é definida como 2, por ser o menor valor entre seus estados sucessores, e como 2 é menor que o  $\alpha$ , o algoritmo atualiza o valor de  $s_0$  para 3.

O Algoritmo 2.2 é a implementação da técnica *Alpha–beta pruning*, e sua base é o algoritmo do Minimax. As modificações realizadas são as adições dos valores  $\alpha$  e  $\beta$  nos métodos de maximização (*Max-Value*) e minimização (*Min-Value*), permitindo que o algoritmo compare os estados já avaliados para decidir se deve continuar percorrendo um caminho da árvore, ou para e segue para outra possibilidade, se houver.

Mesmo com a redução de possibilidades de jogadas que seriam ruins, a técnica *alpha-beta pruning* ainda precisa percorrer grande parte da árvore de jogo [Knu75], tornando-a ineficiente para jogos com uma certa complexidade. Uma solução para este problema é parar a exploração da árvore de pesquisa antes de atingir os estados terminais de jogo, e utilizar uma função de estimativa heurística ou *evaluation function* para estimar a qualidade dos estados atingidos.

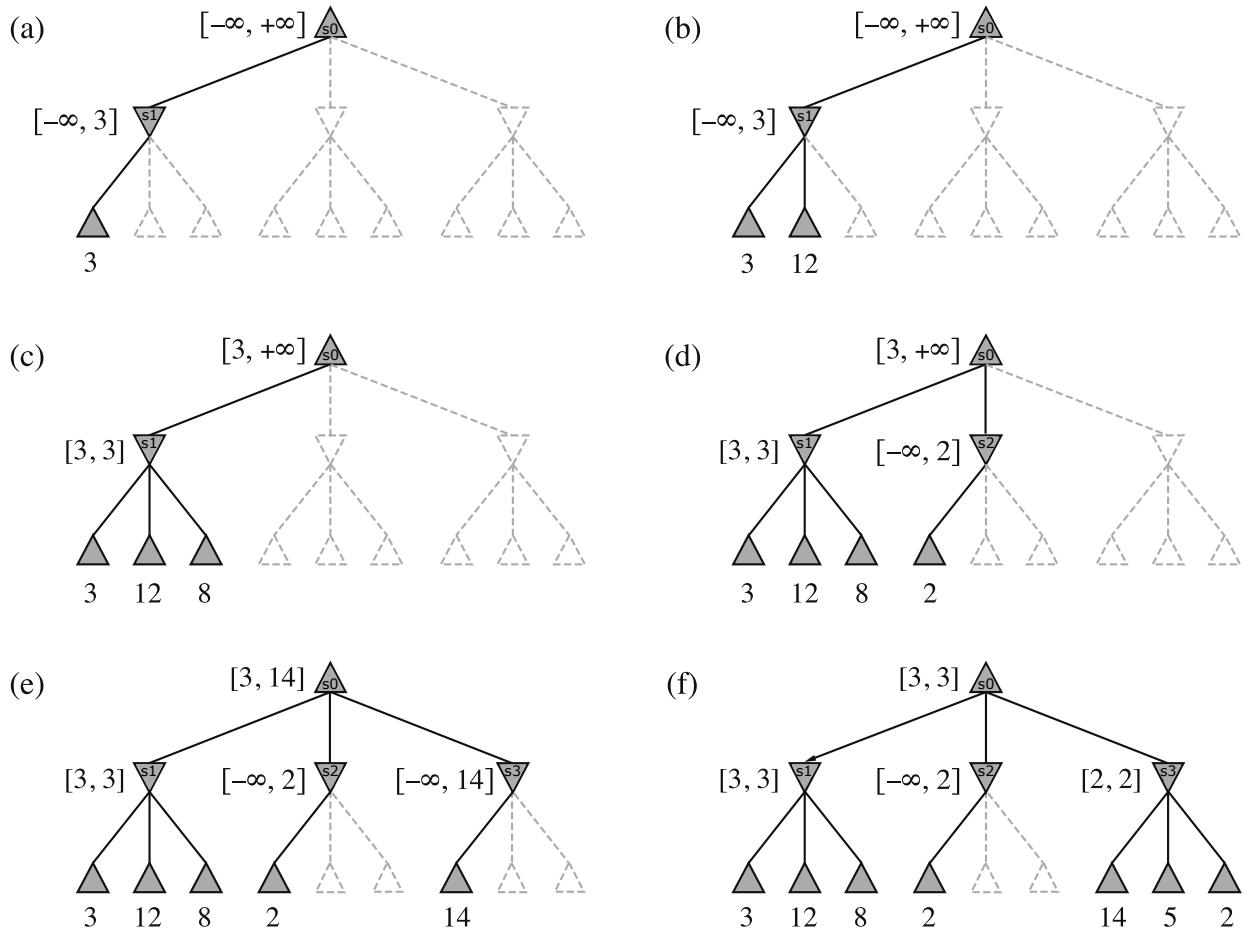


Figura 2.5: Estágios da árvore do algoritmo Minimax com *alpha–beta pruning* [RN10, Cap 5, pp168].

### 2.2.3 Evaluation function

Uma função de estimativa ou *evaluation function* tem como objetivo estimar um valor para um determinado estado de jogo, levando em consideração diversos *fatores* de jogo [RW89]. Estes *fatores* podem ser: o posicionamento das peças no tabuleiro, a pontuação atual de cada jogador, a quantidade de peças ou recursos de cada jogador, ou outros *fatores* que são *decisivos* em um jogo. No xadrez, por exemplo, cada peça possui uma movimentação própria além de uma escala de importância para o resultado da partida, e a função pode estimar o valor de um estado com base nas peças que cada jogador possui e a configuração destas no tabuleiro.

Este cálculo estimativo pode ser descrito de diversas formas, umas delas é pela soma ponderada dos *fatores* descritos anteriormente, representada na Equação 2.2 [RN10, Cap 5, pp172], onde  $w_i$  é o valor de cada peça do jogador presente no tabuleiro e  $f_i$  a quantidade de cada peça presente no tabuleiro, por exemplo. Somando o produto desses fatores temos um resultado estimado do estado  $s$  avaliado.

---

**Algorithm 2.2** Algoritmo Minimax com *alpha–beta pruning* [RN10, Cap 5, pp170]

---

```

1: function ALPHA-BETA-SEARCH( s ) returns an action
2:   v = MAX-VALUE(s,  $-\infty$ ,  $+\infty$ )
3:   return the action in ACTIONS(s) with value v

4: function MIN-VALUE( s,  $\alpha$ ,  $\beta$  ) returns a utility value
5:   if TERMINALTEST(s) then
6:     return UTILITY(s, p)
7:   value =  $\infty$ 
8:   for each a in ACTIONS(s) do
9:     v = MIN(v, MAX-VALUE(RESULT(s, a),  $\alpha$ ,  $\beta$ ))
10:    if v  $\leq \alpha$  then
11:      return v
12:     $\beta$  = MIN( $\beta$ , v)
13:   return v

14: function MAX-VALUE( s,  $\alpha$ ,  $\beta$  ) returns a utility value
15:   if TERMINALTEST(s) then
16:     return UTILITY(s, p)
17:   value =  $-\infty$ 
18:   for each a in ACTIONS(s) do
19:     v = MAX(v, MIN-VALUE(RESULT(s, a),  $\alpha$ ,  $\beta$ ))
20:     if v  $\geq \beta$  then
21:       return v
22:      $\alpha$  = MAX( $\beta$ , v)
23:   return v

```

---

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s) \quad (2.2)$$

O propósito de se aplicar uma função de estimativa é reduzir consideravelmente o caminhamento pela árvore de jogo. Utilizando esta função, o algoritmo estima precoceamente qual será a utilidade de um determinado estado não terminal, ou seja, o algoritmo não percorre todo o caminho até o estado terminal, dando continuidade para a pesquisa em outras jogadas, e repetindo este processo assim que possível [RW89].

Para que uma função de estimativa seja usada pelo Minimax, são necessárias apenas duas alterações no algoritmo, com ou sem *alpha–beta pruning*: substituir a função *Utility* pela função de estimativa e substituir o *TerminalTest* por um teste de corte. O Algoritmo 2.3 representa, de forma geral, como a função *Eval* será chamada para a estimativa de um estado, onde a função *Cutoff-Test* pode chamá-la caso o estado *s* analisado é um bom estado para estimar o resultado de jogo, sem precisar percorrer o resto do caminho até o estado terminal. A função *Cutoff-Test*(*s*, *d*) retorna *True* para os estados terminais e para

todo estado que estiver em uma profundidade  $d$  maior que uma profundidade limite [RN10, Cap 5, pp173].

---

**Algorithm 2.3** Minimax com *evaluation function* [RN10, Cap 5, pp171]

---

```

1: function H-MINIMAX(  $s, d$  ) returns an estimated reward
2:   if CUTOFF-TEST( $s, d$ ) then
3:     return EVAL( $s$ )
4:   else if PLAYER( $s$ ) = MAX then
5:     return  $\max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d + 1)$ 
6:   else if PLAYER( $s$ ) = MIN then
7:     return  $\min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d + 1)$ 

```

---

## 2.3 Métodos de Monte Carlo para Jogos

Métodos de Monte Carlo são métodos estatísticos utilizados em diversas áreas, especialmente na física e na matemática, para aproximar o resultado de integrais sem solução ou de solução computacionalmente cara [BPW<sup>+</sup>12]. Existem diversas técnicas de IA para jogos baseadas nestes métodos, como Monte Carlo Simulation, Monte Carlo Tree Search (MCTS) e Upper Confidence Bounds for Trees (UCT).

Estas técnicas obtiveram sucesso em diversos jogos estocástico e de informação imperfeita: Monte Carlo Simulation foi utilizado em agentes que jogam diversos jogos de cartas, como *Blackjack* [WGM73] e *Bridge* [Gin01]; MCTS e UCT são algoritmos de busca em árvore baseados em Monte Carlo Simulation que possibilitaram a criação de agentes competitivos no jogo Go, um jogo que representava grande desafio para a comunidade de IA devido à sua complexidade, branching factor elevado, e falta de heurísticas efetivas para o jogo [GKS<sup>+</sup>12].

Desde seu sucesso no jogo Go, diversas implementações foram feitas utilizando estas técnicas para resolver outros jogos. A maioria das técnicas de pesquisa em árvore tradicionais, como Minimax e suas variações, são algoritmos de força bruta que dependem de muito conhecimento do domínio, capazes de resolver jogos com pouco espaço de pesquisa, de informação perfeita e determinísticos [GS11]. Já as técnicas baseadas em métodos de Monte Carlo, são algoritmos estatísticos, que podem ser parados a qualquer momento e ainda assim retornar um resultado satisfatório (*anytime*), são altamente escaláveis, e necessitam de pouco ou nenhum conhecimento do domínio, fazendo deles ótimos candidatos para resolver jogos onde estas técnicas tradicionais não obtiveram sucesso [BPW<sup>+</sup>12]. Por isso, consideramos esta técnica uma concorrente de Minimax para nossa implementação.

### 2.3.1 Monte Carlo *Simulation*

Monte Carlo Simulation, ou *Flat* Monte Carlo, é um algoritmo de busca baseado em simulação. O algoritmo pode ser utilizado em uma árvore de jogo para estimar a melhor jogada válida a partir do estado raiz, utilizando simulações que seguem uma *política de simulação*  $\pi_{sim}$  [GS11].

Uma política  $\pi$  define qual ação deve ser tomada a partir de qualquer um dos possíveis estados de uma árvore de jogo, onde  $\pi(s)$  retorna a ação  $a$  que deve ser tomada a partir do estado  $s$  [RN10, Cap 17, pp645-648]. Uma simulação começa do estado raiz  $s_0$  e segue andando pela árvore de jogo em profundidade e sem *backtracking*, alternando o jogador simulado a cada *ply*, até atingir um estado final de jogo  $s_n$ . A cada *ply* da simulação, a política  $\pi_{sim}$  é utilizada para selecionar uma ação  $a_i \in Actions(s_i)$ , e o próximo estado  $s_{i+1}$  é gerado a partir da função *Result*( $s_i, a$ ).

Após um certo número de simulações ou tempo de processamento, um valor estimado é obtido a partir do estado inicial  $s_0$ . Este valor, chamado de *Q-Value*, é uma estimativa para a recompensa obtida tomando a ação  $a$  partindo de um estado  $s$ , representado por  $Q(s, a)$ .

O cálculo do *Q-Value* é realizado utilizando a média de todas as recompensas obtidas nas simulações realizadas, seguindo a Equação 2.3 [GS11], onde  $N(s, a)$  é o número de vezes que uma ação  $a$  foi selecionada partindo do estado  $s$ ,  $N(s)$  é o número de vezes que o jogo foi jogado pelo estado  $s$ ,  $z_i$  é o resultado da simulação de índice  $i$  jogada a partir do estado  $s$  e  $\mathbb{I}_i(s, a)$  é uma função indicadora, que retorna 1 se a ação  $a$  foi selecionada na simulação de índice  $i$  partindo do estado  $s$  e 0 caso contrário.

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s, a) z_i \quad (2.3)$$

### 2.3.2 Monte Carlo *Tree Search*

Monte Carlo Tree Search (MCTS) é um algoritmo moderno baseado em Monte Carlo Simulation. Assim como o anterior, este algoritmo estima a melhor jogada a partir de um estado raiz de uma árvore de jogo, mas diferente dele, MCTS possui uma política dinâmica, que se adapta para realizar simulações apenas nos nodos mais promissores da árvore de jogo, reduzindo o espaço de busca e obtendo performance superior em relação ao algoritmo anterior [CBSS08].

O algoritmo funciona construindo uma árvore de pesquisa iterativamente, que divide o espaço de pesquisa do algoritmo: Inicialmente ele percorre a árvore de pesquisa

construída, utilizando uma *política da árvore*  $\pi_A$ , esta política é dinâmica e melhora conforme a árvore cresce; Utilizando a política  $\pi_A$ , um nodo é escolhido, e a partir dele, uma simulação é feita utilizando uma *política de simulação*  $\pi_{sim}$  fixa, como a de Monte Carlo Simulation. O objetivo é chegar a uma política  $\pi_A$  que seja *best-first*, ou seja, que sempre seleciona a melhor ação para cada estado visitado durante a pesquisa em árvore [BPW<sup>12</sup>].

Cada nodo  $v$  da árvore de pesquisa construída guarda 4 informações: O estado de jogo com o qual é associado  $s(v)$ ; A ação que levou até este estado  $a(v)$ ; O seu *Q-Value*  $Q(v) = Q(s(v_p), a(v))$ , onde  $s(v_p)$  é o estado do nodo pai de  $v$ ; A quantidade de vezes em que este nodo foi visitado  $N(v)$ . Conforme o número de simulações aumenta, a árvore cresce e os *Q-Values* dos nodos se tornam mais precisos através da propagação dos valores obtidos nas simulações. Podemos dividir uma iteração do algoritmo em 4 fases, executadas em sequência, como na Figura 2.6 [BPW<sup>12</sup>]:

- Seleção: O algoritmo percorre a árvore de pesquisa construída em profundidade, partindo do nodo raiz  $v_r$ , utilizando a política  $\pi_A$  para selecionar os nodos que serão visitados, buscando o nodo expansível de maior urgência  $v$ . Um nodo é expansível quando ele representa um estado não terminal e possui filhos ainda não visitados.
- Expansão: O algoritmo cria um nodo  $v'$  a partir de uma ação selecionada segundo  $\pi_A(s(v))$ , e adiciona este nodo como filho de  $v$ , selecionado na fase anterior.
- Simulação: A partir do estado  $s(v')$ , o algoritmo simula o jogo utilizando a política  $\pi_{sim}$  para selecionar as ações a cada *ply*, até que um estado terminal de jogo seja atingido.
- Propagação: A partir do resultado da simulação, o algoritmo calcula o *Q-Value* do nodo  $v'$  e atualiza os *Q-Values* dos nodos visitados durante a fase de Seleção.

Assim que a pesquisa é interrompida, um filho do nodo raiz é escolhido de acordo com uma função *BestChild(v)*, que retorna o nodo filho  $v'$  mais promissor a partir do nodo  $v$ . Existem 3 critérios usuais para escolher o melhor filho utilizando as estatísticas guardadas pelos nodos [Sch09]:

- Filho Max: o método seleciona o filho que tiver o maior *Q-Value*.
- Filho Robusto: o método seleciona o filho com o maior número total de visitas.
- Filho Max-Robusto: o método seleciona o filho que tiver tanto o maior *Q-Value*, quanto o maior número de visitas. Se não existir um filho com ambos, o algoritmo continua a pesquisa até encontrar um filho com um total de visitas aceitável.

O Algoritmo 2.4, apresenta um exemplo de implementação para o algoritmo MCTS em um jogo que tenha mais de 2 jogadores, como Catan, onde  $\Delta$  é um vetor de *Q-Values* para cada jogador do jogo. Consideramos a sintaxe  $a(v)$  como acesso de uma variável

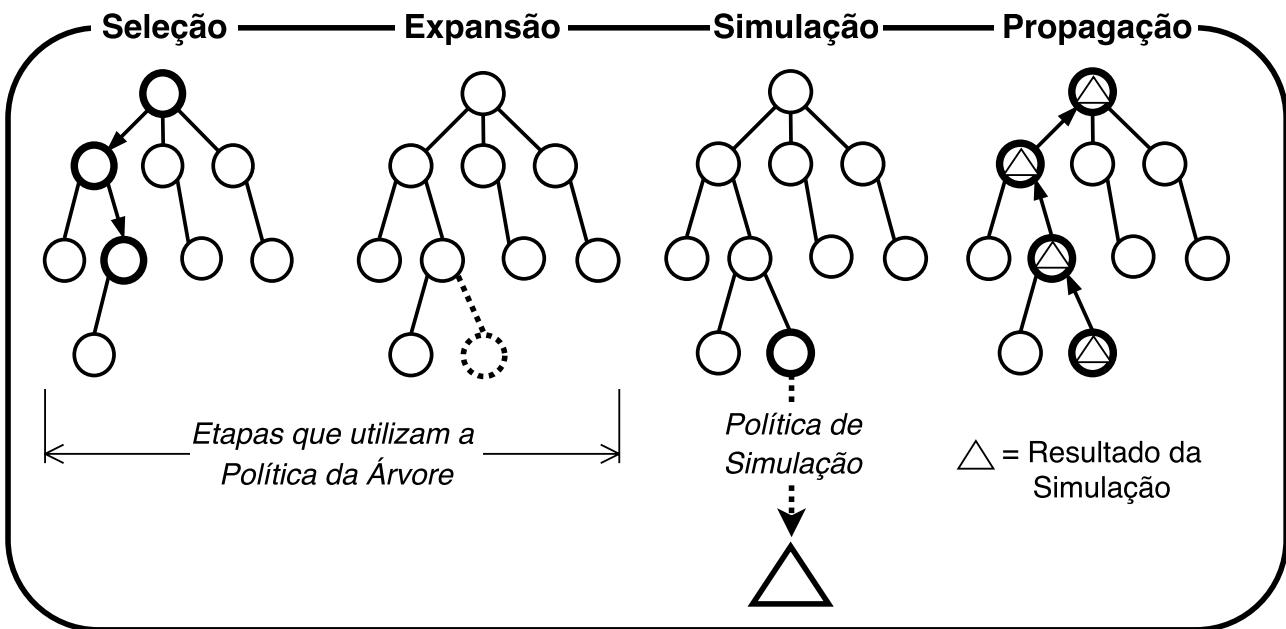


Figura 2.6: As 4 fases de uma iteração do algoritmo Monte Carlo Tree Search [BPW<sup>+</sup>12].

local  $a$  de  $v$ , por exemplo: Na linha 4, estamos acessando a variável local de estado  $s$  do nodo  $v_n$ ; E na linha 7, retornamos a variável local de ação  $a$  do nodo resultado da operação  $\text{BestChild}(v_0)$

---

#### Algorithm 2.4 Algoritmo MCTS simplificado [BPW<sup>+</sup>12]

---

```

1: function MONTECARLOTREESEARCH(  $s_0$  ) returns an action
2:   create root node  $v_0$  with state  $s_0$ 
3:   while within computational budget do
4:      $v_n \leftarrow \text{TREEPOLICY}( v_0 )$ 
5:      $\Delta \leftarrow \text{SIMULATIONPOLICY}( s(v_n) )$ 
6:     BACKUP(  $v_n$  ,  $\Delta$  )
7:   return  $a(\text{BESTCHILD}(v_0))$ 

```

---

As duas políticas de seleção utilizadas pelo algoritmo podem ser alteradas para melhorar a sua performance. O algoritmo pode obter bons resultados utilizando uma política  $\pi_{sim}$  aleatória, mas sua performance pode ser aumentada significativamente incorporando conhecimento de domínio nesta política [Bou05]. Veremos a seguir uma técnica baseada em MCTS que utiliza uma política  $\pi_A$  específica para melhorar o processo de construção da árvore de pesquisa.

### 2.3.3 Upper Confidence Bounds for Trees (UCT)

A política  $\pi_A$  é essencial para o funcionamento do algoritmo MCTS, pois é com esta política que o algoritmo escolhe quais nodos da árvore de pesquisa devem ser expandidos, ou seja: esta política define como a árvore é construída. É necessário equilibrar *exploração* e *lucro* durante a seleção de nodos da árvore: O algoritmo deve explorar nodos de baixa recompensa imediata, pois a estratégia ótima pode estar *escondida* por trás destes nodos; O algoritmo também deve saber quando lucrar para não perder tempo explorando nodos que só levam a baixas recompensas. *Upper Confidence Bounds for Trees* (UCT) é uma versão de MCTS que utiliza uma política  $\pi_A$  que garante o equilíbrio ótimo entre *exploração* e *lucro* utilizando um método que resolve problemas de *Multi-armed-bandit* [GS11].

*Multi-armed-bandit problem* é um problema de decisões sequenciais bastante conhecido, em que é preciso escolher ativar uma entre várias alavancas possíveis em uma máquina caça-níquel repetidas vezes, buscando maximizar o total de recompensa obtido. Uma política de seleção que resolve este problema deve tentar minimizar o total de recompensa perdido pelo jogador por não escolher a melhor ação constantemente, buscando equilíbrio entre *exploração* e *lucro* [TLR85].

*Upper Confidence Bound 1* (UCB1), resolve o *Multi-armed-bandit problem* utilizando uma política que garante um equilíbrio ideal entre *exploração* e *lucro* [ACBF]. O método está representado na Equação 2.4 [BPW<sup>+</sup>12], onde  $\bar{X}_j$  é a recompensa média obtida escolhendo a alavanca  $j$ ,  $n_j$  é o número de vezes que a alavanca  $j$  foi escolhida e  $n$  é o numero total de jogadas realizadas. O termo  $\bar{X}_j$  encoraja a escolha de jogadas lucrativas, enquanto o termo  $\sqrt{\frac{2 \ln n}{n_j}}$  encoraja a exploração de jogadas pouco exploradas.

$$UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \quad (2.4)$$

UCT aplica este método de seleção em MCTS tratando a seleção de nodos da árvore (a política  $\pi_A$ , no caso) como um *Multi-armed-bandit problem*, que é resolvido utilizando uma adaptação do método UCB1, como visto na Equação 2.5 [BPW<sup>+</sup>12].

$$UCT = \bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}} \quad (2.5)$$

É adicionado no método UCB1 uma constante de exploração  $C_p > 0$ , que pode ser ajustada para forçar mais exploração. Caso o número de visitas a um nodo  $n_j = 0$ , UCT utiliza o valor de  $\infty$ , para garantir que todos os filhos de um nodo sejam considerados pelo menos uma vez antes que outros filhos deste sejam expandidos.

Uma das grandes qualidades de UCT, é que, além dele realizar uma pesquisa mais *focada* que MCTS, equilibrando *exploração e lucro*, a árvore gerada por este algoritmo eventualmente converge para uma árvore minimax, dado um número suficiente de simulações, e portanto, é capaz de gerar uma árvore de pesquisa ótima [KS06].

O Algoritmo 2.5 apresenta um exemplo de implementação de UCT baseado no algoritmo MCTS. Como consideramos UCT um algoritmo mais promissor para resolver Catan, apresentamos este código em mais detalhes. Neste código, assim como no de MCTS,  $\Delta$  representa um vetor de *Q-Values*, e  $a(v)$  é um acesso de uma variável local  $a$  de  $v$ .

---

#### **Algorithm 2.5** Algoritmo UCT [BPW<sup>+</sup>12]

---

```

1: function UCTSEARCH(  $s_0$  ) returns an action
2:   create root node  $v_0$  with state  $s_0$ 
3:   while within computational budget do
4:      $v_n \leftarrow \text{TREEPOLICY}( v_0 )$ 
5:      $\Delta \leftarrow \text{SIMULATIONPOLICY}( s(v_n) )$ 
6:     BACKUP(  $v_n$  ,  $\Delta$  )
7:   return  $a(\text{BESTCHILD}( v_0 , 0 ))$ 
```

---

#### 2.3.4 Melhorias para UCT

Existem diversas outras técnicas baseadas em UCT que podem ser consideradas. Em geral essas outras técnicas especializam o algoritmo básico para algum problema específico ou melhoram sua performance com ajustes na sua implementação. Iremos comentar a seguir algumas dessas técnicas que podem ser interessantes para implementarmos nosso agente que jogará Catan.

#### *All Moves As First* (AMAF)

*All Moves As First* é uma heurística para UCT criada com a ideia de que uma ação tem um valor médio, não importando o estado em que esta ação foi tomada. Esta técnica melhorou muito a performance de agentes que jogam Go [GKS<sup>+</sup>12]. Ela funciona com um valor AMAF  $\tilde{Q}(s, a)$ , que é a média de recompensa obtida por todas as simulações onde a ação  $a$  é selecionada em *qualquer turno* depois que o estado  $s$  foi adicionado à árvore de pesquisa, como visto na Equação 2.6 [GS11], que é semelhante a função que calcula o *Q-Value* em MCTS e UCT, onde  $\tilde{\mathbb{I}}_i(s, a)$  é uma função indicadora, que retorna 1 se a ação  $a$  foi selecionada a partir de qualquer outro estado  $s$  já simulado anteriormente, fazendo parte ou não do caminho desta simulação, e 0 caso contrário, e  $\tilde{N}(s, a)$  é o número total de simulações usadas para estimar o valor AMAF.

---

**Algorithm 2.5** Algoritmo UCT [BPW<sup>+</sup>12] - Continuação

---

```

8: function TREEPOLICY(  $v$  ) returns a node
9:   while TERMINALTEST( $v$ ) is False do
10:    if  $v$  not fully expanded then
11:      return EXPAND(  $v$  )
12:    else
13:       $v \leftarrow \text{BESTCHILD}( v, C_p )$ 
14:    return  $v$ 

15: function EXPAND(  $v$  ) returns a node
16:   choose  $a \in$  untried actions from ACTIONS(  $s(v)$  )
17:   add a new child  $v'$  to  $v$ 
18:   with  $s(v') = \text{RESULT}( s(v) , a )$ 
19:   and  $a(v') = a$ 
20:   return  $v'$ 

21: function BESTCHILD(  $v$  ,  $c$  ) returns a node
22:   return  $\arg \max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$ 

23: function SIMULATIONPOLICY(  $s$  ) returns a utility value
24:   while TERMINALTEST( $s$ ) is False do
25:     choose  $a \in$  ACTIONS(  $s$  ) uniformly at random
26:      $s \leftarrow \text{RESULT}( s , a )$ 
27:   return UTILITY( $s$  ,  $p$ )

28: function BACKUP(  $v$  ,  $\Delta$  )
29:   while  $v$  is not null do
30:      $N(v) \leftarrow N(v) + 1$ 
31:      $Q(v) \leftarrow Q(v) + \Delta( v , p )$ 
32:      $v \leftarrow \text{parent of } v$ 

```

---

$$\tilde{Q}(s, a) = \frac{1}{\tilde{N}(s, a)} \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s, a) z_i \quad (2.6)$$

Desta forma, o valor estimado de uma ação escolhida  $a$  é atualizado para todos os nodos da árvore de pesquisa, mesmo que eles não façam parte do caminho que chegou até  $a$ , fazendo com que  $a$ , partindo de qualquer estado  $s$  já conhecido, tenha seu valor estimado atualizado.

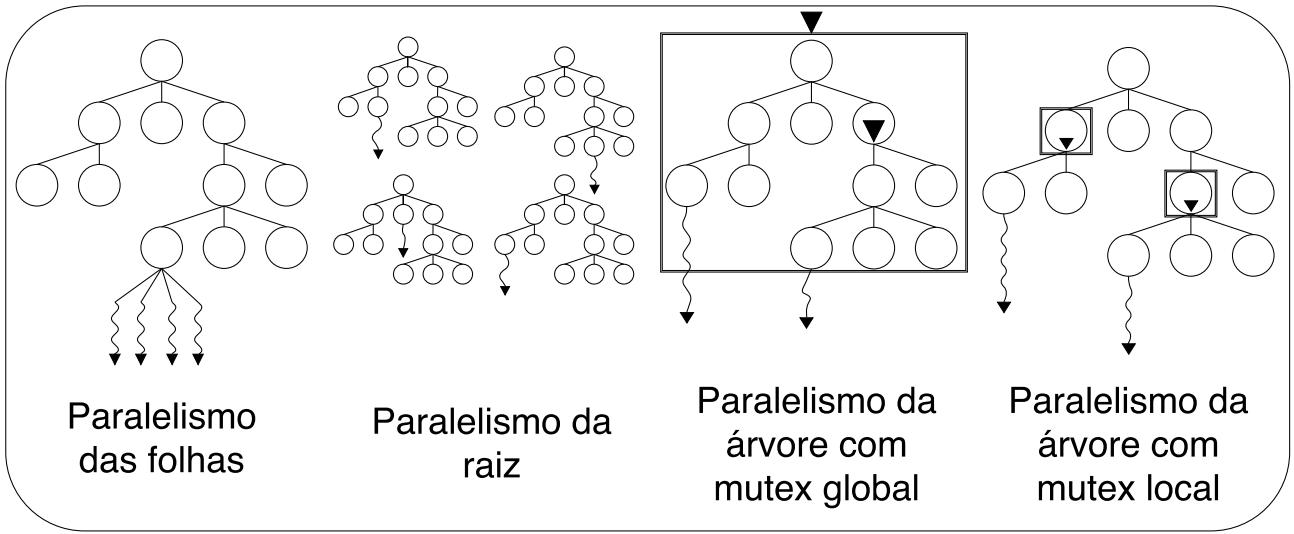
*Rapid Action Value Estimation* (RAVE) é uma melhoria para AMAF, geralmente aplicada sobre o algoritmo básico de UCT. Esta técnica mistura o valor estimado AMAF  $A$ , com o valor estimado pelo algoritmo UCT  $U$ , utilizando a Equação 2.7 [BPW<sup>+</sup>12], onde  $V > 0$  representa um número mínimo de visitas necessárias até que os valores estimados  $A$

deixem de ser considerados, e apenas estimativas mais concretas oferecidas pelo valor  $U$  sejam utilizadas. Essa seletividade é representada por  $\alpha = \max \left\{ 0, \frac{V - v(n)}{V} \right\}$ , onde  $v(n)$  é o número de visitas total do estado  $s$ . Até esse valor  $V$  não ser atingido, uma mistura das estimativas  $A$  e  $U$  é utilizada como estimativa, de acordo com  $\alpha A + (1 - \alpha)U$ .

$$\alpha A + (1 - \alpha)U \text{ onde, } \alpha = \max \left\{ 0, \frac{V - v(n)}{V} \right\} \quad (2.7)$$

## Paralelismo

A independência entre as simulações de uma UCT faz deste algoritmo um bom candidato a paralelismo [BPW<sup>+</sup>12]. Melhorar a performance do algoritmo é interessante porque também melhora a qualidade das suas estimativas, diminuindo o tempo das simulações, mas simulações podem ser feitas. Existem diversos meios de paralelizar o algoritmo, veremos alguns dois mais populares, ilustrados na Figura 2.7.



○ = Nodo da árvore      { = Simulação      □ = Região Exclusiva      ▼ = Thread

Figura 2.7: 4 formas de paralelizar algoritmos baseados em UCT [BPW<sup>+</sup>12].

- **Paralelismo das folhas:** Sempre que a política  $\pi_A$  de uma UCT criar ou chegar até nodos folha da árvore de pesquisa, a fase de expansão e simulação destes nodos folha será realizada em paralelo, adicionando e simulando vários filhos de uma só vez. O problema desta técnica é que ela está *presa* pela simulação que demorar mais.
- **Paralelismo da raiz:** Essa técnica cria várias UCTs que são executadas em paralelo, e fazem jogadas independentemente. Uma vez que o tempo (ou número total de simulações) máximo for atingido, um algoritmo de votação pode ser utilizado para escolher qual UCT fez a melhor estimativa [SKW10]. A vantagem desta técnica é que

as UCTs são independentes e podem ser paradas a qualquer momento, mantendo a característica *anytime* de UCT.

- Paralelismo da árvore: Várias simulações são executadas em paralelo em uma mesma árvore de pesquisa de MCTS. Para proteger as *threads* simultâneas de tentarem acessar as mesmas informações, um mecanismo de exclusão mútua (*mutex*) pode ser utilizado em nível global, ou diversos mecanismos em nível local, encadeando recursos utilizados a nodos específicos. Esta técnica é mais difícil de ser implementada e pode gerar *overhead* de comunicação entre as *threads* concorrentes [BPW<sup>+</sup>12].

### Aprender uma política $\pi_{sim}$

A política  $\pi_{sim}$  pode ser melhorada adicionando heurísticas guiadas por conhecimento acerca do domínio, como regras e estratégias de jogo. Um agente pode aprender uma política  $\pi_{sim}$  melhor do que a simplesmente aleatória utilizando técnicas de aprendizado de máquina.

Uma técnica chamada Move-Average Sampling Technique (MAST) melhora a política  $\pi_{sim}$  tornando ela mais tendenciosa a certas ações [BPW<sup>+</sup>12]. Esta técnica funciona guardando a recompensa média  $Q_m(a)$  de cada ação  $a$ , independente do estado em que a ação foi tomada, em uma tabela, atualizando os dados da tabela a cada propagação feita durante as iterações do algoritmo de UCT. Os dados da tabela são utilizados nas fases de simulação para tornar a seleção de ações mais tendenciosa para ações que geralmente geram boas recompensas.

### 3. COLONIZADORES DE CATAN

Colonizadores de Catan é um jogo de tabuleiro moderno criado pelo alemão Klaus Teuber e publicado em 1995. O jogo se popularizou rapidamente e ganhou notoriedade fora da Alemanha, fazendo com que o gênero conhecido como *eurogame* fosse reconhecido em todo o mundo [SCS10]. Uma das principais razões do sucesso de Catan é a sua jogabilidade: as regras e funcionamento do jogo foram desenvolvidas buscando o equilíbrio entre estratégia, política entre jogadores e sorte.

Consideramos este jogo para este trabalho, por ser um dos principais representantes do gênero *eurogame* e apresentar características de jogo que o tornam interessante para a pesquisa em IA [SCS10]. Dentre estas características, podemos destacar: o jogo é estocástico, devido aos dados, e é parcialmente observável, devido as cartas que os jogadores possuem.

Diversas versões e expansões foram criadas para o jogo principal ao longo dos anos. Este trabalho considera apenas a versão atualizada do jogo original, lançado em 1996 e com a última revisão em 2015, sem considerar as expansões. Nesta ultima revisão, o nome do jogo foi simplificado de *Colonizadores de Catan* para apenas *Catan*, e este nome também é considerado neste trabalho. As regras descritas a seguir se baseiam nas regras da quinta edição do jogo [May15].

#### 3.1 Elementos do Jogo

Em Colonizadores de Catan, cada jogador controla um grupo de colonizadores que pretendem se instalar em uma remota ilha chamada Catan. O jogo é uma corrida por pontos: o primeiro jogador a obter 10 ou mais pontos de vitória é considerado o vencedor da partida e o grande colonizador da ilha. Para obter pontos de vitória, os jogadores devem colonizar a ilha com construções e através delas obter os recursos valiosos que a ilha oferece. A ilha possui diversos tipos de terreno, e cada um destes produz um tipo recurso que pode ser utilizado pelos jogadores para construir suas edificações na ilha.

- Terreno: 19 partes hexagonais que podem ser configuradas com 18 fichas numeradas, representando tanto o tipo de terreno quanto a probabilidade deste terreno gerar recursos.
- Oceano: 6 cartões que são colocados ao redor dos hexágonos e contém os portos da ilha.

- Construções: conjunto de peças que representam suas construções na ilha de Catan. Estas peças são divididas entre 3 tipos: estradas, aldeias e cidades, e são pintadas de 4 cores, cada cor representando um dos jogadores
- Dados: 2 dados de 6 lados numerados que definem quais terrenos da ilha gerarão recursos.
- Cartas: 2 baralhos de cartas distintos, um baralho representa os recursos que podem ser obtidos pelos jogadores e o outro representa cartas de desenvolvimento.
- Cartas Especiais: 2 cartas de pontos de vitória especiais, uma para a *Maior Estrada* e outra para a *Maior Cavalaria*.
- Ladrões: peça especial que representa os ladrões de recurso da ilha.
- Troca: jogadores podem trocar recursos entre si ou com o banco de jogo.

### **3.2 Andamento de uma partida**

O jogo começa com uma configuração de hexágonos, fichas numeradas e oceano montado, que formam o tabuleiro do jogo. Com o tabuleiro de jogo pronto, todos os jogadores devem então colocar suas peças iniciais para a partida. Essa fase é chamada de rodada de preparação.

Durante a rodada de preparação, começando por um jogador aleatório, todos os jogadores devem posicionar 1 aldeia e 1 estrada no tabuleiro, alternando em sentido horário. O último jogador a colocar sua primeira aldeia e sua primeira estrada no tabuleiro, se torna o primeiro a colocar uma segunda aldeia e uma segunda estrada, obrigatoriamente conectada a segunda aldeia, seguido pelos outros jogadores em sentido anti-horário até atingir o primeiro jogador escolhido. Ao final desta fase, todos os jogadores devem ter 2 aldeias e 2 estradas construídas no tabuleiro. A Figura 3.1 mostra um exemplo de configuração inicial para o jogo.

Após terminada a rodada de preparação, os jogadores se revesam no sentido horário em *plies*, onde o *ply* de cada jogador é dividido em 3 fases:

1. Fase de produção: o jogador rola os dados para definir a produção de recursos no turno. A produção vale para todos os jogadores.
2. Fase de trocas: o jogador pode iniciar trocas com outros jogadores ou o banco de jogo.

3. Fase de compras: o jogador pode construir recursos e comprar cartas de desenvolvimento. Não existe limite para o número de construções, nem o número de cartas compradas, desde que o jogador possa pagar por tudo.

O jogo segue até que um dos jogadores obtenha 10 pontos de vitória. Quando isso ocorrer, o jogo obrigatoriamente acaba e os outros jogadores contam os seus pontos para decidir as posições em que ficaram na partida. Apresentaremos a seguir as regras detalhadas que são seguidas durante o decorrer da partida.



Figura 3.1: Exemplo de tabuleiro montado com um *setup* inicial para 4 jogadores.

### 3.3 Recursos

A ilha de Catan possui 6 tipos de terreno, e cada um destes produz um tipo diferente de recurso que pode ser captado pelos colonizadores:

- Floresta: produz madeira.
- Pasto: produz ovelha.
- Lavoura: produz trigo.
- Colina: produz tijolo.
- Montanha: produz minério.
- Deserto: não produz recursos.

Cada hexágono representa 1 destes 6 tipos de terreno, e dentre os 19 hexágonos do jogo existem: 4 florestas, 4 pastos, 4 lavouras, 3 colinas, 3 montanhas e 1 deserto. Jogadores podem utilizar estes recursos para construção de edificações e estradas no tabuleiro ou a compra de cartas de desenvolvimento. Os recursos acumulados por um jogador não ficam expostos aos outros jogadores.

### 3.4 Construções

Estradas podem ser construídas nas laterais livres de cada hexágono e custam 1 tijolo e 1 madeira. Uma lateral é livre quando não houver outra estrada construída nela. As estradas ligam as aldeias e cidades de um jogador. O jogador que possuir o maior caminho formado por estradas ininterruptas, com no mínimo 5 estradas, recebe a carta de *Maior Estrada*, que vale 2 pontos de vitória. Esta carta é única e pode trocar de dono caso um outro jogador possua o maior caminho de estrada no tabuleiro.

Aldeias e cidades são necessárias para a produção de recursos. Estas edificações podem ser construídas nas intersecções entre hexágonos, produzindo recursos de acordo com os 3 hexágonos que formam a intersecção. Aldeias custam 1 tijolo, 1 madeira, 1 trigo e 1 ovelha, produzem 1 recurso e contam como 1 ponto de vitória. Cidades custam 2 trigos e 3 pedras, produzem 2 recursos e contam como 2 pontos de vitória.

Um jogador pode construir uma aldeia sempre que este possuir uma estrada na lateral de algum dos 3 hexágonos que formam a intersecção desejada e que esta posição respeite a regra da distância. Segundo a Regra da Distância, só é possível construir uma aldeia em uma intersecção se não houver nenhuma aldeia ou cidade em até 2 intersecções de distância, de acordo com a Figura 3.2. Aldeias que estiverem construídas no tabuleiro e pertencerem ao jogador, podem ser promovidas para cidades.

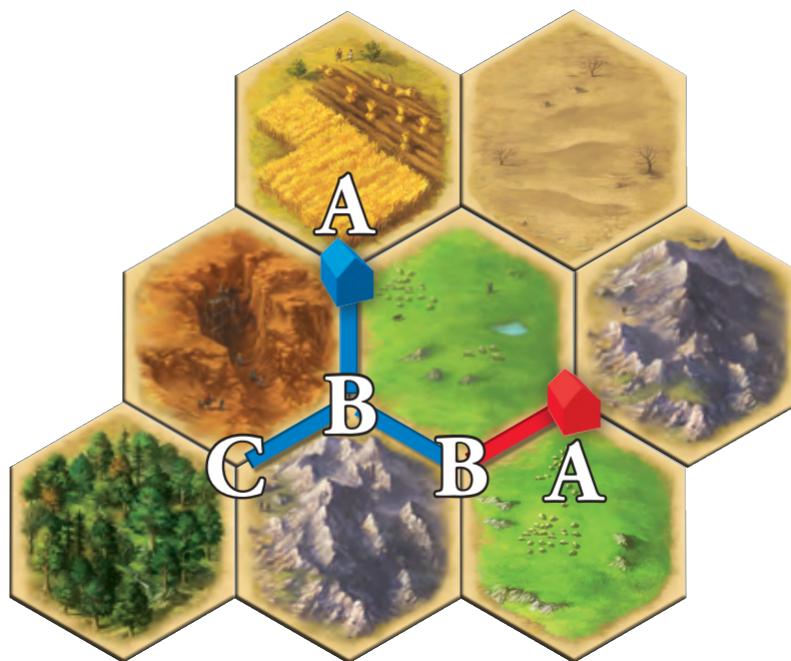


Figura 3.2: Exemplo da Regra da Distância, ambos os jogadores só podem construir aldeias no ponto "C"

### 3.5 Troca de Recursos

Trocas acontecem de acordo com alguma taxa de  $n_1R_1 : n_2R_2$ , onde  $n_1$  e  $n_2$  são quantidades dos recursos  $R_1$  e  $R_2$  respectivamente, que podem ser qualquer um dos 5 recursos disponíveis. Quando a troca de recursos for entre os jogadores, as taxas de troca ficam a critério dos jogadores envolvidos. Trocas com o banco de jogo tem taxa de  $4R_1 : 1R_2$ .

Em cada 1 das 6 abas de oceano existe pelo menos 1 porto. Cada porto possui uma taxa de conversão de recurso, estas podem ser:  $2R_x : 1R_2$ , onde  $R_x$  é um tipo específico específico de recurso definido pelo porto, ou  $3R_1 : 1R_2$ , onde  $R_1$  e  $R_2$  são de qualquer tipo de recurso. Jogadores que possuírem aldeias ou cidades próximas à algum dos portos podem usufruir da sua taxa de conversão.

### 3.6 Número 7 e os Ladrões

A peça de ladrões impede que um hexágono produza recursos e é colocada acima de algum dos hexágonos do jogo. Os jogadores podem mover esta peça quando obtiverem o valor 7 nos dados.

Quando este valor for obtido, a peça de ladrões é obrigatoriamente movida da posição em que estiver do tabuleiro, e todos os jogadores que possuírem mais de 7 cartas de recursos, inclusive o jogador que rolou os dados, devem escolher metade de seus recursos e descarta-los. Se o número for ímpar, é feito arredondamento para cima.

### 3.7 Cartas de Desenvolvimento

Existem 25 cartas de desenvolvimento disponíveis no jogo, divididas em 5 tipos: 14 Cartas de *Cavalaria*; 2 Cartas de *Construção de Estrada*; 2 Cartas de *Monopólio*; 2 Cartas de *Ano de Fartura*; 5 Cartas de *Ponto de Vitória*.

Estas cartas podem ser compradas com recursos e cada uma apresenta um efeito diferente no jogo. Os efeitos das cartas são: *Cavalaria* pode ser utilizada para mudar a posição da peça de ladrões, sem o descarte de recursos; *Construção de Estrada* possibilita a construção de 2 estradas, sem qualquer custo; *Monopólio* permite que o jogador roube todas as cartas de um determinado recurso dos outros jogadores; *Ano de Fartura* permite que o jogador pegue 2 cartas de recurso do banco de jogo sem qualquer custo. Cada jogador só pode utilizar uma carta de desenvolvimento por turno, a qualquer momento do turno, mas não no mesmo turno em que esta carta foi comprada.

O primeiro jogador que utilizar 3 cartas *Cavalaria* recebe a carta de *Maior Cavalaria*, que vale 2 pontos de vitória. Esta carta é única e pode trocar de dono caso um outro jogador possua o maior número de cartas *Cavalaria* utilizadas no jogo.

## 4. AGENTES JOGADORES DE CATAN

Neste capítulo, vamos abordar os diversos agentes que foram implementados neste trabalho. Primeiramente, será detalhada a implementação de um agente *simples*, seguido pelos agentes que utilizam as técnicas estudadas: Minimax, MCTS e UCT.

Todos os agentes são capazes de jogar Catan sem regras especiais e sem vantagens durante o jogo. Os agentes desenvolvidos neste trabalho executam trocas apenas com o banco de jogo, e não com outros jogadores, pois consideramos que esta negociação é um tema complexo que poderia ser considerado como foco de um trabalho de conclusão, logo, está fora do escopo do nosso trabalho, mas pode ser uma melhoria futura para nossos agentes.

### 4.1 Agente Simples

Este agente joga Catan tomando decisões aleatórias com pré-seleção de ações. Optamos por não implementar ações verdadeiramente aleatórias devido ao fato de que muitas das ações possíveis em Catan podem prejudicar o decorrer da partida de um jogador se tomadas sem nenhum critério. A construção de cidades e aldeias, por exemplo, são essenciais para a progressão durante o jogo, já que os recursos gerados por estas construções permitem que o jogador realize mais jogadas no decorrer da partida. Sem um número razoável destas construções, o jogador pode ficar sem ações por vários turnos.

As jogadas podem ser mais prejudiciais durante a rodada de preparação do jogo, descrita na Seção 3.3, já que a posição inicial das aldeias é essencial para a distribuição de recursos do resto da partida [GL14]. Será apresentado a seguir a implementação destes agentes, além de detalharmos como a pré-seleção impede que jogadas *prejudiciais* sejam consideradas.

#### 4.1.1 Implementação

Durante a rodada de preparação, este agente seleciona, dentre todas as posições validas para a construção de aldeias, as posições que estejam adjacentes a no mínimo 2 recursos do tabuleiro. Isso impede que o jogador aleatório construa uma aldeia em uma posição que produza apenas um recurso, o que minimiza a possibilidade dele ficar sem produção de recursos suficiente para progredir durante a partida.

Após esta rodada de preparação, este agente prioriza a construção de aldeias e cidades, separando estas ações das demais da seguinte forma: se for possível construir uma

cidade, ele desconsidera as outras ações, o mesmo vale para aldeias, caso ele não possa construir uma cidade. Esta seleção é realizada já que ambas ações concedem pontos de vitória, que aproximam o agente da vitória, e geram recursos, que permitem ao agente realizar mais ações durante a partida.

Caso o agente não possua recursos o suficiente para realizar uma ação em seu turno, ele troca os recursos mais abundantes em sua *mão* com o banco de jogo por recursos que não possui, se possível, de acordo com as taxas de troca. Esta é mais uma medida para evitar que o jogador fique ocioso durante a partida.

## 4.2 Agente Minimax

O agente Minimax se mostrou inapto para jogar Catan. O principal fator para isso é o branching factor elevado do jogo [Roe12]. Estudamos diversas estratégias de corte da árvore, como *alpha-beta pruning*, que corta a árvore de pesquisa consideravelmente, mas esta técnica só é possível em um jogo de 2 jogadores, com mais jogadores é necessário empregar uma versão específica de Minimax, o  $\text{Max}^N$ , com estratégias de *pruning* menos eficazes [Fri14].

Optamos então por verificar a capacidade deste agente em uma versão diferenciada de Catan, implementada em nosso cliente, onde apenas 2 jogadores jogam em uma partida. Implementamos Minimax com *alpha-beta pruning* e o colocamos contra nosso agente simples.

Mesmo limitando o jogo para apenas 2 jogadores, o *branching factor* ainda é elevado demais para considerarmos Minimax como alternativa viável para o jogo. Devido a esse fator, mesmo com um limite de profundidade na pesquisa pequeno, como 1 jogada a frente, o agente tem muitas jogadas para considerar na árvore de pesquisa. A *explosão* de estados é maior no começo do jogo, durante a rodada de preparação, onde diversas posições do tabuleiro devem ser consideradas, e esta é a fase mais importante para o agente, limitar o espaço de pesquisa durante essa fase pode significar perdas catastróficas para o agente [GL14].

Constatamos através de nossos testes que durante as fases inciais do jogo, cada jogada gera em média 30 nodos na árvore de jogo, e nosso cliente leva em média 30 segundos para computar 1 ply. Estes resultados demonstram o desafio que jogos estratégicos modernos como Catan representam para técnicas de IA tradicionais.

## 4.3 Agente MCTS

Este agente joga Catan utilizando o algoritmo Monte Carlo *Tree Search*, descrito na Seção 2.3.2. Este algoritmo escolhe uma ação a ser tomada realizando uma amostragem na árvore de jogo, através de diversas simulações de jogo, reduzindo o espaço de busca consideravelmente, o que possibilita que este algoritmo seja utilizado em Catan, mesmo com seu *branching factor* elevado.

Durante esta seção, sempre que nos referirmos a *simulações*, estamos nos referindo a simulações de jogo realizadas pelo algoritmo MCTS, e partidas realizadas serão referidas como *executadas* pelo cliente. A seguir, descreveremos a implementação deste agente além de seus resultados.

### 4.3.1 Implementação

Implementamos o algoritmo MCTS descrito neste trabalho, utilizando o *Filho Max*, descrito na Seção 2.3.2, como método de seleção de nodos utilizado pela política da árvore. Com este método, nosso agente sempre prioriza nodos da árvore de pesquisa que tiverem o maior *Q-Value* calculado.

Utilizamos os pontos de vitória obtidos pelo agente ao final da partida simulada para calcular o *Q-Value* de um nodo, e adicionamos pontos *virtuais* caso ele tenha sido o vencedor da partida simulada. Estes pontos *virtuais* tornam a pesquisa mais tendenciosa para ações que levaram a vitórias do agente. A Equação 4.1 detalha como calculamos o *Q-Value* de cada jogador, onde  $Q^i$  é o *Q-Value* do jogador  $i$ ,  $vp^i$  são os pontos de vitória obtidos pelo jogador  $i$  e  $V$  representa os pontos *virtuais* do jogador, onde  $t$  representa o turno em que a partida simulada terminou e  $\mathbb{I}_w(i)$  é uma função indicadora que retorna 1 caso o jogador  $i$  foi o vencedor da partida e 0 caso contrário. A parte  $\frac{100}{t}$  faz com que estados de jogo com menos turnos, que terminaram mais cedo, tenham valor maior do que estados de jogo com muitos turnos, onde 100 é a média observada de turnos em cada partida.

$$Q^i = \frac{vp^i}{10} + V, \text{ onde } V = \frac{100}{t} 2\mathbb{I}_w(i) \quad (4.1)$$

Com este cálculo de *Q-Value*, nosso agente valoriza ações que levam a vitória e ações que levam à mais pontos de jogo, impedindo que o agente considere muitas jogadas sem utilidade durante sua pesquisa.

Como o algoritmo toma decisões baseadas no resultados de  $n$  simulações, ou no número de simulações realizadas em um tempo  $t$ , a velocidade destas simulações está di-

retamente relacionada com a velocidade de resposta deste agente, no caso de um limite de  $n$  simulações, ou na sua performance, caso houver um limite de tempo  $t$  para sua resposta. Por conta disso, otimizamos nosso código Python para acelerar as simulações de jogos, utilizando memória para guardar diversas informações sobre o estado de jogo e o resultado de operações, minimizando a quantidade de operações realizadas a cada simulação. Outra medida que tomamos para acelerar as simulações, foi a utilização da pré-seleção do agente simples na política de simulação do algoritmo. A pré-seleção acelera a construção de aldeias e cidades durante as simulações, fazendo com que os jogos simulados terminem antes. Com estas otimizações, nosso agente é capaz de simular cerca de 80 jogos por segundo.

Para evitar que nosso agente utilize uma quantidade exagerada de memória, optamos por serializar nodos recém criados pela política da árvore do algoritmo através do módulo de serialização para Python chamado *cPickle*, e guardamos apenas o objeto serializado em memória. Caso um nodo seja visitado muitas vezes, substituimos o objeto serializado pelo estado de jogo desserializado, evitando que os nodos mais visitados necessitem ser desserializados repetidas vezes, o que consumiria processamento. Através desta medida, mantemos em memória apenas os estados dos nodos mais *importantes* para a pesquisa, ou seja, aquelas mais visitados, permitindo uma redução de cerca de 10 vezes no consumo de memória do algoritmo, sem prejudicar sua performance.

## 4.4 Agente UCT

Para este agente, utilizamos como base a mesma implementação utilizada para o agente MCTS. A diferença deste agente para o anterior é o método de seleção utilizado pela política da árvore: este agente utiliza o método UCB1 descrito na Seção 2.3.3. Com este método de seleção, os resultados obtidos por este agente foram muito superiores ao dos agentes anteriores.

Este agente é nossa referência de performance contra o agente JSettlers, como demonstraremos a seguir na Seção de resultados. Vamos apresentar a sua implementação em detalhes, além de possíveis modificações e melhorias.

### 4.4.1 Implementação

Implementamos este agente com todos os métodos do agente MCTS, especializando apenas os métodos necessários para o algoritmo UCT, como a seleção da política da árvore. Em nossa implementação, utilizamos valores de *Q-Value pequenos*, não muito maiores que 1, para não afetar a equação UCB1 [BPW<sup>+</sup>12], já que valores muito *distantes*

1 fazem com que a parte de exploração da equação nunca seja relevante. Para esta parte de exploração da equação, utilizamos  $C_p = 1$  como valor da *constante de exploração*. Não existe uma regra para a definição da *constante de exploração*  $C_p$  na equação utilizada pelo método UCT, ela varia de jogo para jogo e entre implementações diferentes, e geralmente é escolhida baseada em testes de performance [BPW<sup>+</sup>12]. Escolhemos este valor baseado em nossas experimentações com o algoritmo e, devido a restrições de tempo para obtenção e análise de resultados para este trabalho, não analisamos o impacto desta variável no jogo Colonizadores de Catan neste trabalho e pretendemos realizar esta análise em trabalhos futuros.

Além de implementarmos o UCB1 para este agente, implementamos uma função para a estimativa rápida de nodos recém criados na árvore de pesquisa. Como este algoritmo *explora* mais a árvore de jogo, criamos um método que atribui um *Q-Value* estimado durante a criação dos nodos da árvore de pesquisa, fazendo com que o algoritmo explore primeiro ações com o *potencial* de dar mais pontos de vitória, como a construção de aldeias e cidades, sem forçar a seleção destas ações, como na pré-seleção do agente simples.

## 5. CLIENTE

O cliente, desenvolvido em Python 2.7, tem função de realizar a conexão entre os agentes desenvolvidos neste projeto com o servidor do cliente JSettlers. A comunicação é feita através de mensagens, tratadas por um sistema próprio, e as ações geradas pelas mensagens também são realizadas por um sistema próprio.

Além da conexão, o cliente também contém toda a lógica de jogo do Catan e é capaz de simular uma partida do inicio ao fim, com todas as ações possíveis no jogo. A seguir descreveremos os detalhes de nossa implementação e do servidor JSettlers.

### 5.1 JSettlers

JSettlers é uma implementação do jogo Colonizadores de Catan que possui uma interface gráfica e possibilita a realização de partidas entre jogadores via um servidor de jogo. Desenvolvido por Robert S. Thomas na linguagem Java [TH02], seu código fonte é *open-source* e possui uma das melhores implementações de agente para o jogo [SCS10]. É por conta deste agente que escolhemos esta implementação para este projeto, e utilizamos este agente como referência para a performance dos agentes desenvolvidos. Além disso, JSettlers permite a fácil visualização do jogo, através de sua interface, além da participação de jogadores humanos em uma partida.

O servidor do JSettlers tem como função guardar os dados da partida corrente, e realizar a lógica de Catan. Os jogadores jogam a partida atuando como clientes deste servidor, e se comunicam com ele através de troca de mensagens. Nosso cliente contém uma interface de comunicação para o servidor JSettlers, que traduz ações de jogo para mensagens que podem ser recebidas e enviadas para o JSettlers.

#### 5.1.1 Interface

A interface do cliente é composta por um tabuleiro, assim como no jogo original, e com informações mais explícitas de todos os jogadores, como mostra a Figura 5.1. Além de representar os detalhes do jogo, a interface contém botões e controles que permitem com que jogadores humanos joguem Catan, além de um chat para possibilitar a negociação.

Outro aspecto interessante do cliente, é a possibilidade de um jogador humano substituir um agente durante a partida através de um comando da interface, facilitando no *debug* de nossos agentes. Utilizamos este recurso para *entrar* no lugar de um agente JSettlers em algumas partidas de *debug* e acompanhar as estratégias de nossos agentes

enquanto participávamos do jogo, permitindo uma análise mais profunda da qualidade das decisões de nossos agentes.



Figura 5.1: Interface gráfica do cliente JSettlers.

### 5.1.2 Troca de mensagens

Toda a comunicação entre cliente/servidor se dá por mensagens do tipo texto com um formato padrão: "ID | conteúdo da mensagem", onde o "ID" é o identificador da mensagem. A classe do cliente JSettlers responsável por receber e mandar mensagens é a SOCSERVER. Uma vez que a classe SOCSERVER recebe uma mensagem, ela imediatamente é enviada para a classe SOCMessage que manipula e realiza uma ação conforme o identificador da mesma. Esta ação está relacionada diretamente ao nosso cliente, modificando tabuleiro, dados, informações dos jogadores, etc. Quando o cliente precisa enviar uma mensagem ao servidor, a classe SOCPlayerClient envia um comando para a classe SOCMessage, que monta a mensagem de acordo com a solicitação do cliente, retornando para

SOCPlayerClient enviar para SOCServer. Uma amostra da relação entre cliente/servidor pode ser vista na Figura 5.2:

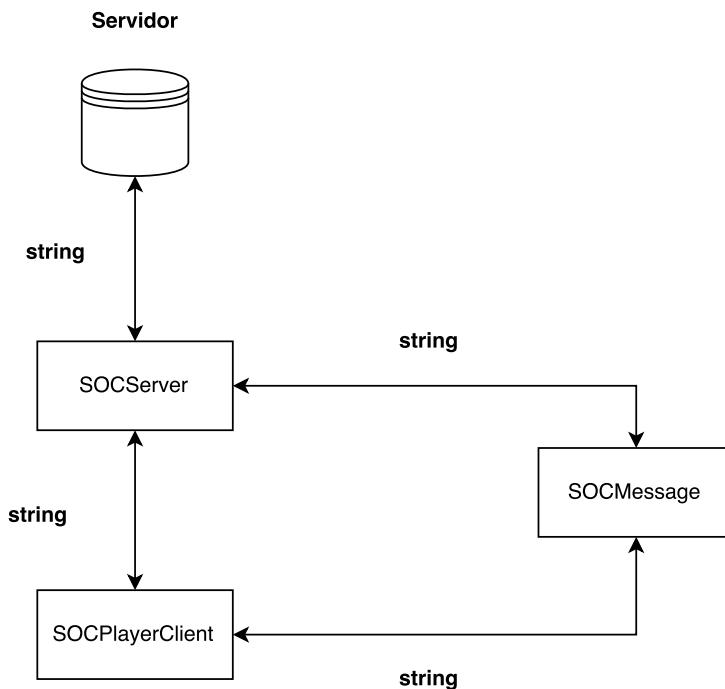


Figura 5.2: Diagrama da comunicação cliente/servidor do JSettlers.

## 5.2 Arquitetura

O cliente que desenvolvemos neste trabalho é dividido em quatro aspectos fundamentais: conectividade com o servidor, sistema de mensagens, representação de estados e os sistemas de ações. A conectividade com o servidor é feita através de uma interface de comunicação com o servidor JSettlers, chamada *Client*, e um sistema de mensagens que possibilita a comunicação com o servidor de jogo, mantendo as informações recebidas em uma classe que representam os estados de jogo, chamada *GameState* e informações sobre os jogadores em uma classe que representa um jogador, chamada *CatanPlayer*, que é a classe base para nossos agentes. Para simular o jogo localmente, criamos um sistema de ações capazes de mudar o estado da classe *GameState*, que é utilizado pelos nossos agentes para simular partidas (no caso de agentes baseados em simulação) e selecionar uma ação. Estas ações podem ser transformadas em mensagens e enviadas de volta para o servidor. Uma visão geral da arquitetura é mostrada na Figura 5.3.

Além do cliente, foram implementadas diversas ferramentas paralelas para facilitar nossos testes e a depuração de nosso código, como: um script próprio para executar partidas totalmente locais; um sistema capaz de salvar estados de jogo (*GameState*) e recuperá-los do disco; um visualizador de estado de jogo. Todas estas ferramentas, assim com o cliente, foram implementadas na linguagem Python 2.7, com auxílio de módulos próprios da linguagem.

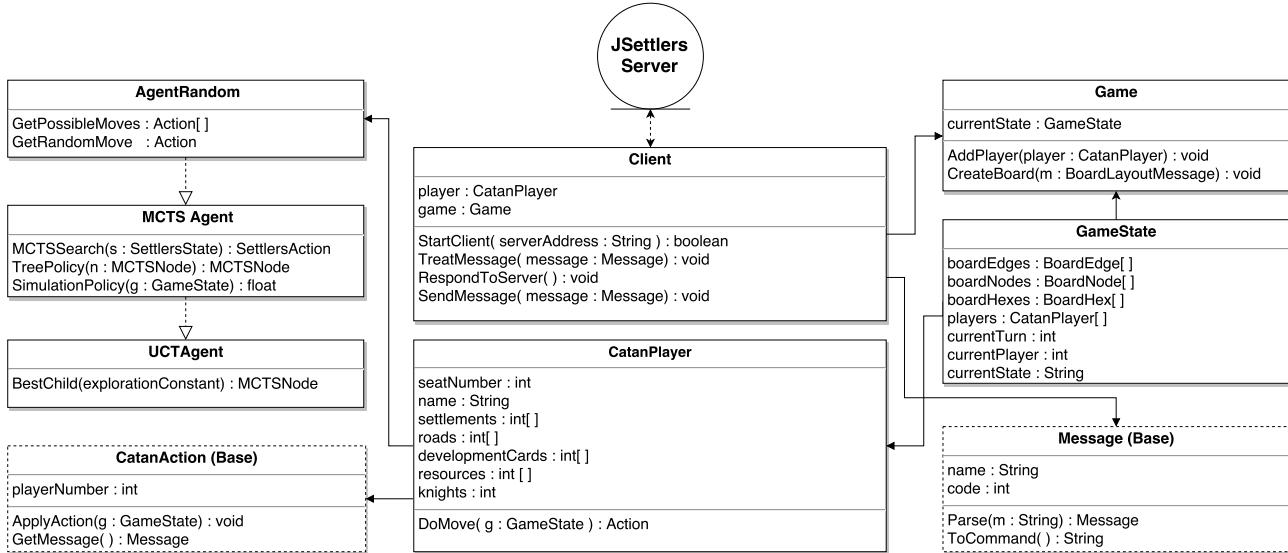


Figura 5.3: Visão geral da arquitetura do nosso cliente.

### 5.3 Implementação

Escolhemos a linguagem Python 2.7 [Pyt16] para este projeto por ser uma linguagem que permite rápida prototipação e implementação, com uma comunidade ativa e diversos módulos disponíveis para adicionar funcionalidades à linguagem. Essas características foram fundamentais para este trabalho, devido a complexidade do sistema desenvolvido, e o tempo disponível para sua implementação.

A desvantagem de Python frente outras linguagens é a sua velocidade. Nossa implementação poderia executar partidas de Catan mais rapidamente em outra linguagem, e, por conta disso, procuramos otimizar o código o máximo possível. Implementamos módulos de extensão para Python específicos para Catan, utilizando a linguagem Cython [Cyt16], capaz de compilar módulos para Python na linguagem C, de execução mais rápida do que código nativo na linguagem Python.

### 5.3.1 Sistema de mensagens

Este sistema tem como função organizar as mensagens recebidas do servidor, para o uso do cliente. Cada tipo de mensagem é tratada como uma classe específica, contendo todos os dados mandados pelo servidor.

As mensagens implementadas para o nosso cliente, são todas equivalentes as implementações do cliente JSettlers, assim foi possível seguir um mesmo padrão e facilitar o entendimento. Não foram implementadas todas as mensagens, apenas as essenciais para que os agentes pudessem jogar o jogo de maneira competitiva.

### 5.3.2 Representação de estados

Foram implementados os elementos principais do jogo Catan, tanto de jogadores, quanto de lógica e elementos de jogo. Por exemplo, o tabuleiro do jogo é representado por um dicionário de hexadecimais, assim como no cliente JSettlers [TH02]. Cada número hexadecimal é o índice de um hexágono do tabuleiro. Cada hexágono possui vértices, que são representados também por hexadecimais, e cada vértice possui arestas, que são também hexadecimais.

O estado de jogo é mantido em uma classe chamada *GameState*, que implementa esta representação de tabuleiro, com todos os dados relativo ao tabuleiro de jogo em um determinado estado. Esta classe também mantém as informações sobre os jogadores da partida, representados pela classe *CatanPlayer*, que contém os dados de todos os jogadores, como suas estradas, aldeias, cartas de recurso, cartas de desenvolvimento e pontos de vitória.

### 5.3.3 Sistema de ações

Existem várias jogadas possíveis no Catan, a fim de facilitar a aplicação delas pelos agentes, foi criado este sistema para unificar as ações que podem ser executadas pelo simulador e recebidas/enviadas pelo servidor. Cada ação é representada por uma classe, assim como as mensagens, e uma ação pode ser instanciada a partir de uma mensagem e vice-versa, o que facilita no mapeamento da lógica de jogo entre o servidor JSettlers e nosso cliente.

As classes *CatanAction* recebem um estado de jogo e produzem o estado sucessor esperado, através da função *DoMove(GameState g)*, que implementa a funcionalidade

da função  $Result(s, a)$ , descrita na Seção 2.1.1, e necessária para a execução de jogos dentro do cliente. É com esta função que os agentes MCTS e UCT implementam a simulação de jogo, logo, sua implementação foi essencial para este trabalho.

## 5.4 Ferramentas de teste

Para o auxílio do desenvolvimento dos agentes e do cliente, foram criadas ferramentas de testes. Estas ferramentas possibilitaram otimizar o código, verificar erros, validar respostas e obter resultados mais facilmente.

Dentre as ferramentas implementadas para auxiliar este projeto, podemos destacar: ferramenta de *profiling*; visualização de estados de jogo; *logs* salvos das partidas, contendo todo o histórico de ações e estados de jogo; gerador de arquivos .csv, para facilitar a visualização de dados gerados. A velocidade com que o cliente executa jogos também foi medida a cada modificação feita no código, através de um *profiler* de velocidade, para garantir que a performance não fosse afetada inadvertidamente.

### 5.4.1 Profiling

Ferramenta utilizada para identificar possíveis gargalos de performance em nosso código. Ela é capaz de gerar um *log* contendo as informações detalhadas sobre: tempo, quantidade de chamadas de cada função e de cada arquivo presente no projeto. A biblioteca utilizada foi a *cProfiler*, própria do Python [Pyt16]. Com a ajuda desse *profiler*, realizamos diversas otimizações em nossa implementação.

O Log 5.1, é um exemplo de como é a saída da ferramenta. Onde a coluna *ncalls* representa a quantidade de vezes que o arquivo, representado pela coluna *filename:lineno(function)*, é executado, e o tempo total de execução do mesmo, representado pela coluna *tottime*. Com essas informações, foi possível identificar trechos de códigos não otimizados com mais precisão.

| ncalls  | tottime | percall | cumtime                       | percall | filename : lineno (function ) |
|---------|---------|---------|-------------------------------|---------|-------------------------------|
| 40361   | 49.939  | 0.001   | 49.939                        | 0.001   | {cPickle.loads}               |
| 20337   | 21.745  | 0.001   | 23.208                        | 0.001   | {cPickle.dumps}               |
| 9307508 | 21.143  | 0.000   | 23.459                        | 0.000   | CatanPlayer.py:455(           |
|         |         |         | DepthSearch)                  |         |                               |
| 1532675 | 6.710   | 0.000   | 22.621                        | 0.000   | AgentRandom.py:101(           |
|         |         |         | GetRandomAction_RegularTurns) |         |                               |
| 1570271 | 5.473   | 0.000   | 5.473                         | 0.000   | CatanGame.py:312(             |
|         |         |         | GetPossibleRoads)             |         |                               |

```

408916    4.621    0.000    30.155    0.000    CatanPlayer.py:364(
    CountRoads)
20000     3.751    0.000    81.230    0.004    AgentMCTS.py:347(
    SimulationPolicy)
1550178   2.240    0.000    2.240     0.000    CatanGame.py:333(
    GetPossibleSettlements)
...
...
\label{profilerResult}

```

Listing 5.1: Log gerado pelo cProfiler.

Outro script de *profiling* implementado foi utilizada para medir a velocidade de execução de  $n$  jogos entre agentes simples em um tabuleiro de jogo fixo. Com este *script*, pudemos medimos a quantidade de partidas executadas por segundo e obter comparação rápida de performance entre diferentes versões do cliente.

#### 5.4.2 Visualizador de estados

Pelo fato de nosso cliente não possuir interface gráfica, foram encontradas algumas dificuldades para saber rapidamente se um estado de jogo era válido ou não. Para lidar com este problema, desenvolvemos um visualizador de estados capaz de salvar e carregar estados de jogo do disco, e renderizar uma imagem que representa as aldeias, cidades e estradas construídas por cada jogador, além de informar quem é o jogador com a maior estrada.

A implementação de uma interface gráfica baseada neste visualizador de estados é uma possível melhoria futura para nosso cliente. Optamos por não incluir esta melhoria já que ela foge do escopo deste trabalho, além de questões de tempo de projeto, mas acreditamos que com este visualizador como base, uma interface gráfica é uma evolução possível para nosso cliente. A Figura 5.4 é um exemplo de imagem gerada pelo visualizador para *debug*, onde é indicado qual é o jogador com a maior estrada da partida.



Figura 5.4: Exemplo do estado de jogo visto no visualizador de imagem.

## 6. RESULTADOS - PERFORMANCE E ANÁLISE

Medimos a performance de jogo de nossos agentes em Catan através de partidas realizadas em nosso cliente e partidas realizadas no servidor JSettlers, contra os agentes JSettlers. Estas partidas foram divididas em diferentes cenários de teste, onde colocamos 1 agente implementado contra 3 agentes diferentes, afim de obter e comparar dados referentes a performance de jogo dos agentes em diferentes configurações de partida. Nas seções a seguir, vamos analisar os seguintes dados obtidos durante as partidas realizadas em cada um dos cenários de teste:

- $V\%$  : é a porcentagem de vitórias de um agente
- $\bar{t}$  : é a média de turnos que um agente leva para vencer uma partida
- $p\%(n)$  : é a porcentagem de partidas onde um agente obteve  $n$  pontos de vitória
- $\bar{pts}$  : é a média de pontos que um agente obteve durante as partidas
- $\sigma pts$  : é o desvio padrão de pontos que um agente obteve durante as partidas
- $\bar{R}, \bar{S}, \bar{C}, \bar{K}$  : são as médias de estradas, aldeias, cidades e cavaleiros, respectivamente, realizadas durante as partidas
- $LA\%, LR\%$  : são as porcentagens de partidas onde o jogador obteve as conquistas de maior cavalaria e maior estrada, respectivamente

Consideramos que  $V\%$  é o maior indicador de performance de jogo de um agente, enquanto os demais dados demonstram o comportamento de um agente durante a partida, e são importantes para analisar os agentes. Nas seções que seguem, vamos representar graficamente  $p\%(n)$  de cada agente em cada um dos cenários de teste, onde  $p\%(n > 10) = V\%$ , já que a partida termina sempre que um agente obtiver  $n > 10$  pontos de vitória, além de comentar e analisar cada gráfico. Ao final deste capítulo, apresentaremos tabelas contendo os dados obtidos para os 3 cenários de teste que envolvem o agente JSettlers: 1 agente MCTS contra 3 agentes JSettlers; 2 cenários contendo versões diferentes do agente UCT, um com um agente que simula 1.000 partidas para sua estimativa e outro com um agente que simula 10.000, contra 3 agentes JSettlers em ambos.

Devido a restrições de tempo para obtenção dos resultados, não analisaremos o impacto de todas as variáveis disponíveis para os agentes implementados neste trabalho e pretendemos estudar o impacto destas variáveis em trabalhos futuros. Dentre estas variáveis, podemos destacar:

- Outras fórmulas e valores de Q-Value para nossos agentes MCTS e UCT

- O refinamento do valor  $N$  utilizado no algoritmo de serialização da árvore gerada pelos algoritmos MCTS e UCT
- O impacto da constante de exploração  $C_p$  para o algoritmo UCT
- A influencia da função de estimativa rápida de nodos recém criados na árvore de pesquisa na performance do algoritmo UCT

Todos os testes foram realizados em um computador pessoal de bom desempenho; Intel Core i7-4790K CPU @ 4.00GHz; 16GB RAM DDR3; HD 2TB 7200 RPM; Windows 10 EDU.

## 6.1 Agente Simples

Medimos a performance de nosso agente simples contra todos os outros agentes apresentados neste trabalho. Dentre os principais atributos do nosso agente, está a capacidade rápida de decisão sem ficar *trancado* durante a partida, ou seja, sem ações disponíveis por diversos turnos seguidos. Apesar dos critérios de pré-seleção de jogadas, este agente é bastante inferior aos demais e serve como caso de teste, para medir a performance dos outros agentes implementados neste trabalho.

Nosso cliente é capaz de executar cerca de 100 partidas entre agentes simples por segundo, em nossa máquina de testes. A Figura 6.1 foi criada com base nos resultados obtidos após a realização de 1.000 partidas entre 4 agentes simples e apresenta a relação entre pontos obtidos por cada agente e a porcentagem de partidas realizadas, onde 10 ou mais pontos de vitória caracterizam vitórias dos agentes, de acordo com as regras de jogo vistas na Seção 3.1.

Durante nossos testes, mantivemos as *cadeiras* (ou posições dos agentes) fixas. Com base nos resultados mostrados na Figura 6.1, podemos concluir que a posição dos agentes não tem impacto em sua performance de jogo, já que com cadeiras fixas, em jogo entre 4 agentes simples, cada agente obtém uma porcentagem de vitórias  $V_{\%} \approx 25\%$ , o que demonstra o equilíbrio entre os agentes. Por conta disso, não levaremos este dado em conta durante a medição de performance de nossos agentes.

A Figura 6.1 também mostra que mesmo com a pré-seleção de estados, estes agentes obtém poucos pontos de vitória na maioria dos jogos. Acreditamos que o motivo disso é que estes agentes realizam trocas com o banco de jogo e descarte de recursos de forma aleatória, desta forma, a *mão* de recursos deste agente não é *consistente* e ele *perde* a oportunidade de realizar diversas jogadas por descartar ou trocar recursos que seriam estratégicos.

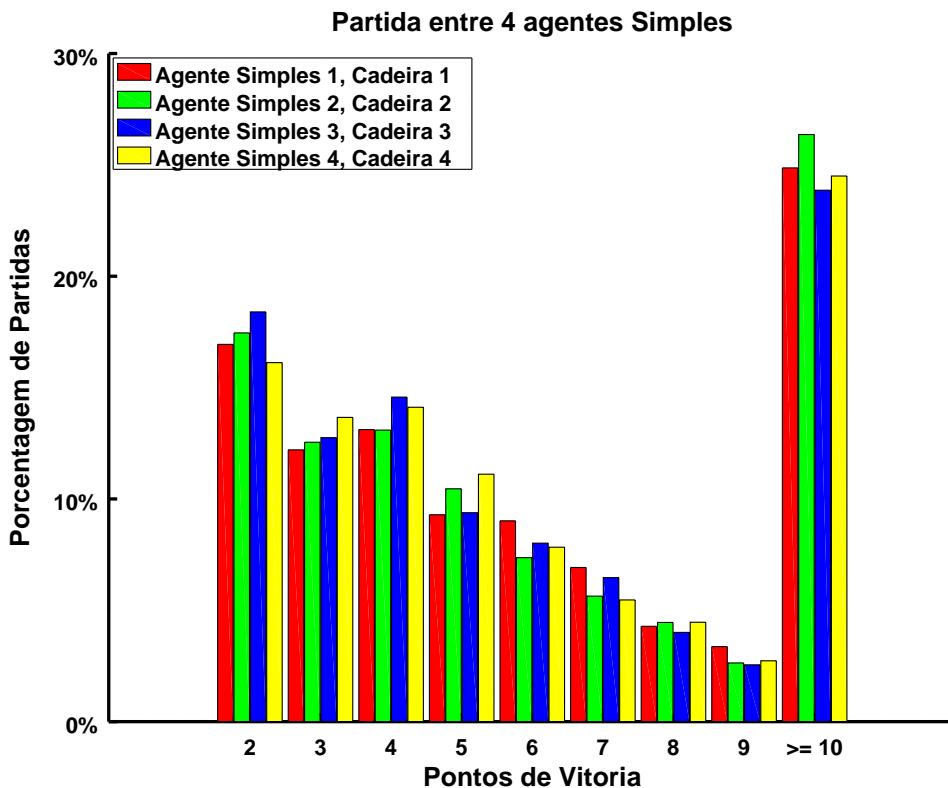


Figura 6.1: Performance de 4 Simples jogando em cadeiras fixas.

## 6.2 Agente MCTS

Medimos a performance deste agente em jogos contra 3 de nossos agentes simples e em jogos contra 3 agentes JSettlers. No decorrer desta seção, quando nos referirmos a *simulações*, estamos nos referindo as simulações realizadas pelo algoritmo MCTS durante sua fase de simulação. Com a finalidade de padronizar nossos testes e avaliações, vamos utilizar um número fixo de 1.000 simulações por decisão do agente. Durante esta Seção, chamamos este agente de *MCTS1000* para evidenciar sua quantidade de simulações. Decidimos por implementar estes valores fixos de simulações pois um limite de tempo não garante um número de simulações, que pode variar entre máquinas diferentes. Consideramos 1.000 simulações como um padrão para nosso agente, pois, com base em nossos testes, acreditamos que é um número suficiente de simulações para que o algoritmo realize uma estimativa satisfatória. Outro fator é o tempo demorado que um número maior de simulações levariam: rodando em nossa máquina de testes, o agente *MCTS1000* leva em média 40 segundos para tomar uma decisão. Com 10.000 simulações, este agente pode levar até 2 minutos e 30 segundos por decisão.

Para acelerar a realização de testes entre este agente e os agentes simples, não utilizamos o servidor do JSettlers, executamos estas partidas localmente, utilizando a implementação de Catan contida em nosso cliente com multiprocessamento. O tabuleiro destes

jogos foi mantido fixo, e o jogador inicial escolhido aleatoriamente, assim nenhum agente é favorecido e apenas as suas decisões, e não a configuração do tabuleiro, influenciam no resultado de cada partida. Nossa agente *MCTS1000* venceu cerca de 60 das 100 partidas jogadas contra estes agentes, ficando com uma porcentagem de vitórias  $V_{\%} \approx 60\%$ . A Figura 6.2 representa os valores  $p_{\%}(n)$  obtido por cada agente nessas 100 partidas. Com estes resultados, podemos concluir que este agente é superior ao simples e capaz de formular uma estratégia competitiva.

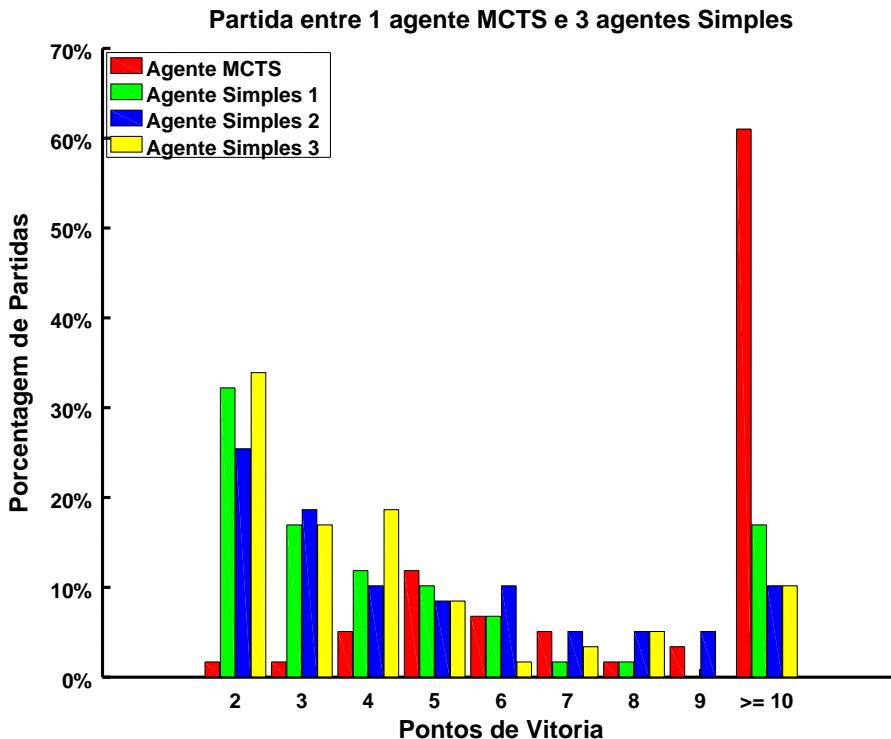


Figura 6.2: Performance do agente *MCTS1000* contra 3 agentes Simples.

Os testes contra o agente *JSettlers* foram executados utilizando o servidor *JSettlers*, com tabuleiro de jogo aleatório. Apesar da vantagem do agente *MCTS1000* frente ao agente simples, este agente não representa desafio para o agente *JSettlers*: em 100 partidas contra 3 agentes *JSettlers*, o agente *MCTS1000* obteve  $V_{\%} \approx 10\%$ , como mostra a Figura 6.3.

Acreditamos que essa desvantagem frente ao *JSettlers* se deve ao fato do agente *JSettlers* ser capaz de fazer planos: *planos imediatos* e *planos futuros* [TH02]. Já nosso agente *MCTS1000* é *guloso*, pois não leva em consideração a exploração da árvore de jogo, devido ao seu critério de seleção na política da árvore, que só leva em consideração ações com maior *utilidade* imediata, sem explorar outras jogadas disponíveis que podem ser mais vantajosas no decorrer da partida. O agente a seguir é capaz de obter resultados superiores utilizando uma política de árvore aperfeiçoada.

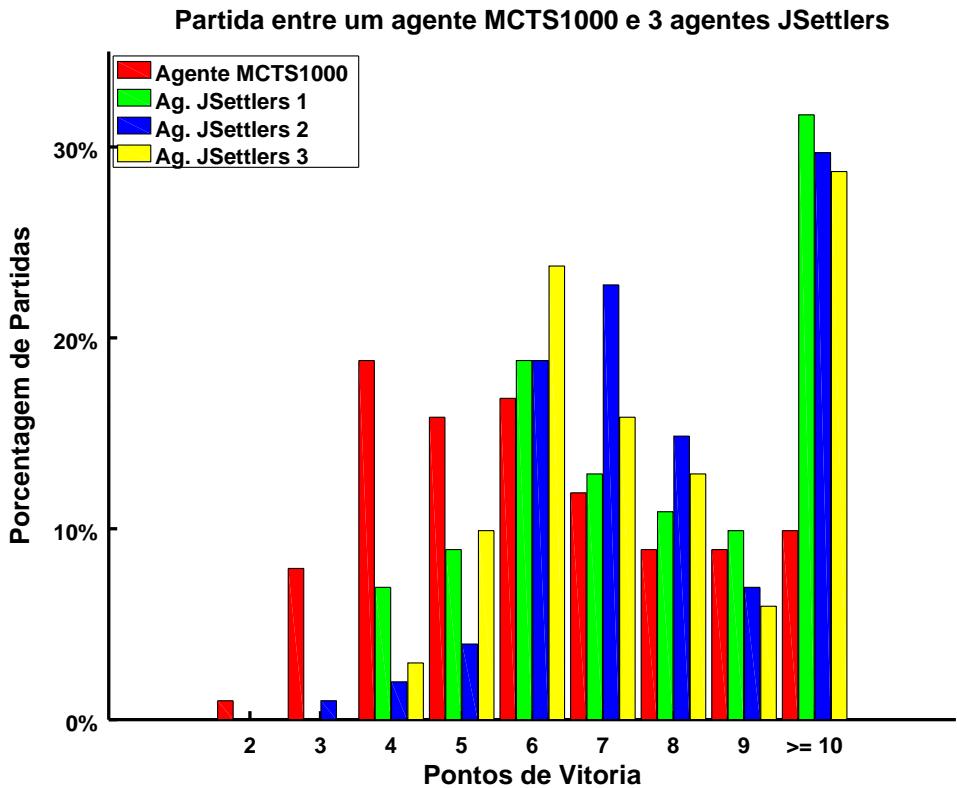


Figura 6.3: Performance do agente MCTS1000 contra 3 agentes JSettlers.

### 6.3 Agente UCT

Colocamos este agente em partidas contra agentes simples e contra agentes JSettlers. A configuração dos testes é a mesma do agente *MCTS1000*: colocamos este agente em partidas contra 3 agentes simples e em partidas contra 3 agentes JSettlers. Novamente, quando nos referirmos à *simulações*, são simulações de partidas realizadas pelo algoritmo UCT para sua estimativa. Além de 1.000 simulações, como implementado no agente *MCTS1000*, consideramos também 10.000 simulações para este agente em nossos testes, a fim de medir quanto o número de simulações afeta em sua performance contra o agente JSettlers. Vamos nos referir ao número de simulações feitas pelo agente como *UCT1000* para 1.000 simulações e *UCT10000* para 10.000 simulações. Os tempos de resposta deste algoritmo são os mesmos do MCTS: cerca de 40 segundos por decisão para o agente *UCT1000* e 2 minutos e 30 segundos para o *UCT10000*.

Ambas as versões deste agente tiveram performance de jogo superior à do agente MCTS. Nosso agente *UCT1000* obteve uma porcentagem de vitórias  $V_{\%} \approx 90\%$  de vitórias nas 100 partidas jogadas contra 4 agentes simples, o que demonstra a sua superioridade estratégica contra estes agentes simples e ao agente MCTS, comparando suas performances neste teste. A Figura 6.4 mostra a média de pontos obtida pelos agentes nestas 100 partidas.

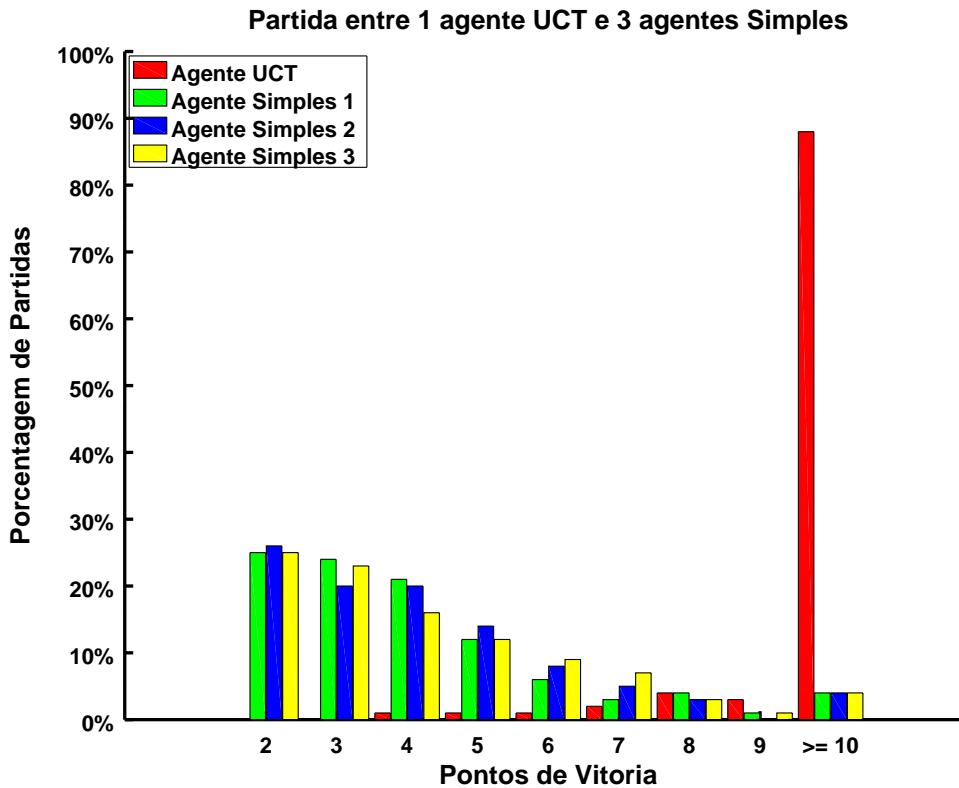


Figura 6.4: Performance do agente UCT1000 contra 3 Agentes Simples.

Nosso agente *UCT1000* pode ser considerado equivalente em performance ao agente JSettlers, pois obteve uma porcentagem de vitórias  $V_{\%} \approx 27\%$  em 125 partidas contra 4 agentes JSettlers, como demonstrado na Figura 6.5. Podemos perceber através dos valores de  $p_{\%}(n)$ , representados pela Figura 6.5, que seu comportamento é *diferente* do comportamento do agente JSettlers, já que os valores de  $p_{\%}(n)$  do agente UCT variam mais do que os valores  $p_{\%}(n)$  dos agentes JSettlers. Acreditamos que isso se deve, em parte, à falta de negociação de nosso agente: através de repetidas trocas, um agente pode obter um recurso que não possui em *mãos* ou que tem poucas, ou nenhuma, chance de produzir, para realizar alguma ação do seu interesse. Nosso agente *UCT1000* está limitado a realizar trocas apenas como banco de jogo, que são mais raras devido às taxas de troca elevadas, já na negociação entre jogadores, as taxas de troca podem ser mais baixas, permitindo que os agentes JSettlers troquem com maior frequência.

Outro motivo que pode estar contribuindo para a maior variação dos pontos de vitória deste agente, é a seleção de posições *ruins* para as aldeias durante a rodada de preparação do jogo. Essa seleção é de extrema importância para a partida, já que a distribuição inicial de recursos é definida por estas primeiras construções. A seleção destas aldeias iniciais é mais complexa do que as demais seleções durante a partida, devido a 2 fatores que aumentam o espaço de pesquisa consideravelmente: ela acontece no começo da partida, quando a árvore de jogo é *maior*, e existem mais posições possíveis para a construção aldeias, característica especial da rodada inicial. Este problema poderia ser

solucionado através de uma heurística que selecione posições sabidamente superiores, utilizando estratégias específicas para Catan, reduzindo assim o espaço de pesquisa e, consequentemente, o número de simulações necessário para obter uma *boa estimativa*.

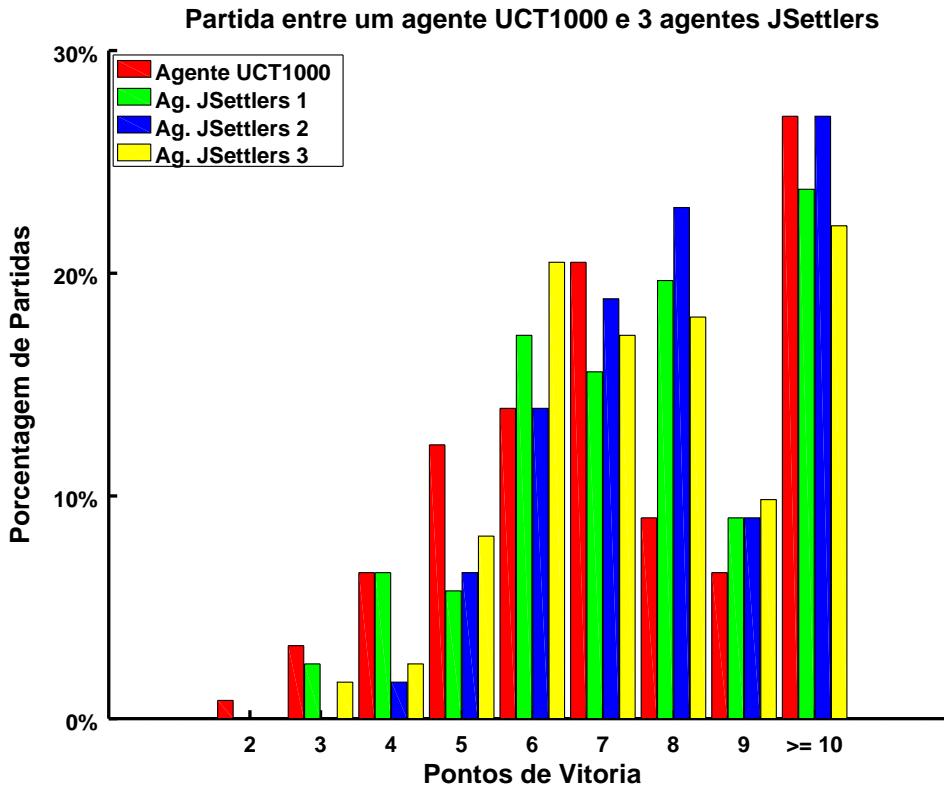


Figura 6.5: Performance do agente UCT1000 contra 3 agentes JSettlers.

O agente *UCT1000* foi superior ao JSettlers em nossos testes, e obteve uma porcentagem de vitórias  $V_{\%} \approx 37\%$  em 125 partidas jogadas contra 4 agentes JSettlers, como mostra a Figura 6.6. Percebemos que este agente manteve valores de  $p_{\%}(n)$  próximos a vitória (onde  $6 < n < 9$ ), menor do que o agente *UCT1000*. Acreditamos que isso se deve ao fato de que este agente consegue *decidir* a partida mais rapidamente, vencendo grande parte das partidas onde está *próximo* a vitória. Apesar disso, observamos que existe uma taxa elevada de partidas onde este agente fica com *poucos* pontos de vitória, especialmente os valores  $p_{\%}(5)$  e  $p_{\%}(6)$ , como o agente *UCT1000*. Concluímos que os mesmos motivos descritos anteriormente para o agente *UCT1000* podem ser a causa deste comportamento, pois a diferença entre estes agentes é apenas na quantidade de simulações. Mais simulações poderiam melhorar a seleção de aldeias iniciais, mas 10.000 delas podem não ser o suficiente para se obter uma melhoria de performance expressiva em relação ao último agente.

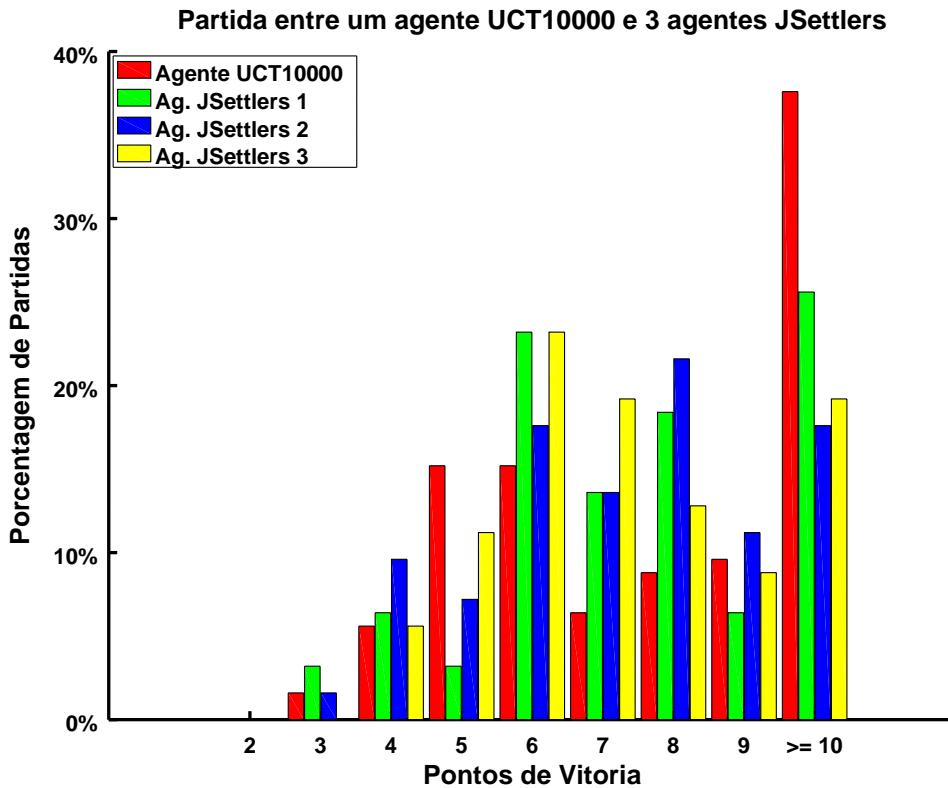


Figura 6.6: Performance do agente UCT10000 contra 3 agentes JSettlers.

## 6.4 Análise de Resultados

Analisamos a performance de nossos agentes baseados em *Monte Carlo Tree Search* nos cenários de teste contra os agentes JSettlers através da medição dos dados coletados, descritos no começo deste capítulo. A Tabela 6.1 apresenta a porcentagem de vitórias  $V\%$  e a média de turnos que um agente leva para vencer uma partida  $\bar{t}$ , onde nosso agente esta caracterizado como  $A$  e  $J1, J2$  e  $J3$  representam os agentes JSettlers. Essa Tabela mostra que nosso agente *UCT1000* obteve resultados similares aos dos agentes JSettlers, enquanto o *UCT10000* obteve uma média de vitórias significativamente maior. Observamos também que, embora tenha ganhado poucas partidas, nosso agente *MCTS1000* obteve a menor média de turnos para vencer o jogo, devido ao seu comportamento *guloso*.

| Agente   | Partidas | $V\%$ (A) | $V\%$ (J1) | $V\%$ (J2) | $V\%$ (J3) | $\bar{t}$ (A) | $\bar{t}$ (J1) | $\bar{t}$ (J2) | $\bar{t}$ (J3) |
|----------|----------|-----------|------------|------------|------------|---------------|----------------|----------------|----------------|
| MCTS1000 | 101      | 9.90%     | 28.71%     | 23.76%     | 37.62%     | 74.70         | 84.52          | 83.88          | 85.24          |
| UCT1000  | 125      | 27.20%    | 20.80%     | 24.00%     | 28.00%     | 81.59         | 82.08          | 87.97          | 83.03          |
| UCT10000 | 125      | 37.60%    | 20.80%     | 24.00%     | 17.60%     | 78.64         | 81.69          | 84.50          | 76.32          |

Tabela 6.1: Porcentagem de vitórias dos agentes vs JSettlers.

Analisamos também a média de pontos de vitória obtidos durante nossos testes  $\bar{pts}$  além do desvio padrão desta média  $\sigma_{pts}$ , demonstrados na Tabela 6.2, onde compara-

mos os valores obtidos por todos os agentes com os obtidos apenas pelos nossos agentes. Nesta tabela, podemos observar que o único de nossos agentes que obteve uma média de pontos superior aos agentes JSettlers foi o agente *UCT10000*, enquanto o agente *UCT1000* teve uma média próxima. Quando levamos em conta o desvio padrão das médias, os valores obtidos foram novamente próximos aos dos agente JSettlers. Estas médias podem ser utilizadas como métricas de performance no jogo, mas não indicam superioridade de um agente em relação ao outro. Assim como os gráficos apresentados, estas métricas indicam que os agentes *UCT1000* e *UCT10000* possuem comportamentos *diferentes* do agente JSettlers durante a partida, devido aos possíveis problemas apontados na Seção 6.3, e pela maneira com que estes agentes realizam suas decisões, através de amostragem, que pode levar a maiores variações de performance, enquanto os agentes JSettlers jogam através de planejamento.

| Agente   | Partidas | $\bar{pts}$ (geral) | $\sigma pts$ (geral) | $\bar{pts}(A)$ | $\sigma pts(A)$ |
|----------|----------|---------------------|----------------------|----------------|-----------------|
| MCTS1000 | 101      | 7.33                | 2.11                 | 6.16           | 2.18            |
| UCT1000  | 125      | 7.58                | 1.94                 | 7.33           | 2.23            |
| UCT10000 | 125      | 7.46                | 2.06                 | 7.78           | 2.29            |

Tabela 6.2:  $\bar{pts}$  e  $\sigma pts$  de todos os agentes vs  $\bar{pts}$  e  $\sigma pts$  de nossos agentes.

A Tabela 6.3 mostra a quantidade de construções feitas por nossos agentes durante nossos testes, além da porcentagem de partidas em que nossos agentes obtiveram as cartas de pontos de vitória especiais: de *Maior Estrada LR%*, descrita na Seção 3.4 e de *Maior Cavalaria LA%*, descrita na Seção 3.7. Esta tabela mostra que o agente *UCT10000* é capaz de realizar mais ações em relação aos demais agentes implementados neste trabalho. A única média inferior deste agente, foi a de aldeias construídas  $\bar{S}$ , pois este agente converte um número maior de aldeias em cidades, como mostra a sua média de cidades  $\bar{C}$ , que é superior aos demais. Nossos agentes *UCT1000* e *UCT10000* obtiveram uma alta taxa de cartas especiais, e esse comportamento pode estar diretamente relacionado às suas altas taxas de vitória, pois estas cartas podem ser decisivas em uma partida, já que ambas concedem 2 pontos de vitória para o jogador que as detém.

| Agente   | Partidas | $\bar{R}$ | $\bar{S}$ | $\bar{C}$ | $\bar{K}$ | $LR\%$ | $LA\%$ |
|----------|----------|-----------|-----------|-----------|-----------|--------|--------|
| MCTS1000 | 101      | 7.65      | 1.92      | 1.50      | 1.00      | 31.68% | 9.90%  |
| UCT1000  | 125      | 7.67      | 1.90      | 1.74      | 1.50      | 45.60% | 24.80% |
| UCT10000 | 125      | 7.90      | 1.81      | 1.86      | 1.59      | 55.20% | 28.80% |

Tabela 6.3: Dados de jogo de nossos agentes.

## 7. CONCLUSÃO

Colonizadores de Catan é um jogo desafiador para agentes com heurísticas simples e possui um *branching factor* elevado, que dificulta a implementação do algoritmo Minimax neste jogo. Ao final deste trabalho, demonstramos empiricamente a capacidade do nosso agente de formular estratégias competitivas através do resultado das partidas realizadas em nossos experimentos. Concluimos com base nestes resultados, que o algoritmo *Monte Carlo Tree Search* e suas variações são eficazes neste jogo, além de ter potencial competitivo contra outras implementações que utilizam estratégias *hard-coded*. Em nossos testes, nosso agente *UCT10000* obteve mais vitórias que o agente JSettlers, que é uma referência em performance neste jogo [SCS10]

Durante nossa pesquisa, encontramos poucas implementações *open-source* para o jogo, além da implementação JSettlers na linguagem Java. Por conta disso, desenvolvemos nossa própria implementação para o jogo, além de um cliente para a comunicação com o servidor JSettlers, desenvolvidos na linguagem Python, que carece de implementações contendo toda a lógica de jogo. A linguagem Python é muito utilizada dentro da comunidade de IA e suas extensões que facilitam a implementação de agentes baseados em *Machine Learning*. Logo, nossa implementação, que contém os agentes desenvolvidos neste trabalho e a lógica de jogo, pode servir como base para a pesquisa de novos agentes que utilizem tais técnicas em Python. Nossa implementação está hospedada no endereço: <https://github.com/GabrielRubin/TC2>

Apesar de obter uma performance superior a do agente JSettlers, nosso agente *UCT10000* possui 2 desvantagens frente a este agente: ele leva mais tempo do que este agente para tomar uma decisão e não realiza trocas com outros jogadores, apenas com o banco de jogo. O tempo de resposta de nosso agente pode ser melhorado através de otimizações em nosso código Python e em nossos algoritmos, e pretendemos realizar estas mudanças futuramente. A negociação com outros jogadores é o único aspecto de jogo que não foi considerado para este trabalho, pois consideramos a negociação estratégica um tema complexo que exige demasiada pesquisa para entrar no escopo deste trabalho, mas sua implementação pode melhorar a performance de jogo de nossos agentes e nos comprometemos a pesquisar formas de solucionar este problema.

Estratégias de jogo específicas para Catan podem ser consideradas para aperfeiçoar nossos agentes MCTS e UCT [BPW<sup>+</sup>12], como a utilização de *estratégias de construção* [GL14] para aprimorar o posicionamento das primeiras aldeias durante a rodada de preparação do jogo. Com base em nossos resultados neste trabalho, acreditamos que a escolha de posições *ruins* para as primeiras aldeias foi um dos fatores que afetou a performance de nossos agentes durante nossos testes, e pretendemos estudar essa e outras heurísticas capazes de melhorar a sua performance.

## REFERÊNCIAS BIBLIOGRÁFICAS

- [ACBF] Auer, P.; Cesa-Bianchi, N.; Fischer, P. “Finite-time analysis of the multiarmed bandit problem”, *Machine Learning*, vol. 47–2, pp. 235–256.
- [Bou05] Bouzy, B. “Associating domain-dependent knowledge and monte carlo approaches within a go program”, *Information Sciences*, vol. 175–4, 2005, pp. 247 – 257, heuristic Search and Computer Game Playing {IV}.
- [BPW<sup>+</sup>12] Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; Colton, S. “A survey of monte carlo tree search methods”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4–1, March 2012, pp. 1–43.
- [CBSS08] Chaslot, G.; Bakkes, E.; Szita, I.; Spronck, P. “Monte-carlo tree search: A new framework for game ai”. In: In Proceedings of AIIDE-08, 2008, pp. 216–217.
- [Cyt16] Cython, O. “Site oficial da linguagem cython”. Capturado em: <http://cython.org/>, Nov 2016.
- [Fri14] Fridenfalk, M. “Equalization and variation enhancement in strategic multiplayer games”. In: The IEEE Games, Entertainment, Media Conference (GEM), 2014 IEEE International, Toronto, Canada, October 2014, 2014, pp. 284–285.
- [FT91] Fudenberg, D.; Tirole, J. “Game Theory”. Cambridge, MA: MIT Press, 1991, translated into Chinesse by Renin University Press, Bejing: China.
- [Gin01] Ginsberg, M. L. “Gib: Imperfect information in a computationally challenging game”, *J. Artif. Int. Res.*, vol. 14–1, Jun 2001, pp. 303–358.
- [GKS<sup>+</sup>12] Gelly, S.; Kocsis, L.; Schoenauer, M.; Sebag, M.; Silver, D.; Szepesvári, C.; Teytaud, O. “The grand challenge of computer go: Monte carlo tree search and extensions”, *Commun. ACM*, vol. 55–3, Mar 2012, pp. 106–113.
- [GL14] Guhe, M.; Lascarides, A. “Game strategies for The Settlers of Catan”. IEEE Computer Society Press, 2014, pp. 1–8.
- [GS11] Gelly, S.; Silver, D. “Monte-carlo tree search and rapid action value estimation in computer go”, *Artificial Intelligence*, vol. 175–11, 2011, pp. 1856 – 1875.
- [HE61] Hart, T. P., Edwards, D. J. “The tree prune (tp) algorithm. artificial intelligence project memo 30”. In: Massachusetts Institute of Technology, 1961.
- [Knu75] Knuth, D. E. “An analysis of alpha-beta pruning”. In: AIJ, 6(4), 1975, pp. 293–326.

- [KS06] Kocsis, L.; Szepesvári, C. “Bandit based monte-carlo planning”. In: Proceedings of the 17th European Conference on Machine Learning, 2006, pp. 282–293.
- [KT53] Kuhn, H. W.; Tucker, A. W. “Extensive games and the problem of information”. In: Kuhn, H. W. and Tucker, A. W. (Eds.), Contributions to the Theory of Games II. Princeton University Press, 1953.
- [Mar87] Marsland, T. A. “Computer chess methods”. In: Shapiro, S., ed.: Encyclopaedia of Artificial Intelligence, 1987, pp. 157–171.
- [May15] Mayfair Games, I. “Catan - 5th edition, game rules almanac”. Capturado em: [http://www.catan.com/files/downloads/catan\\_5th\\_ed\\_rules\\_eng\\_150303.pdf](http://www.catan.com/files/downloads/catan_5th_ed_rules_eng_150303.pdf), 2015.
- [MF09] Millington, I.; Funge, J. “Artificial Intelligence for Games, Second Edition”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, 2nd ed..
- [Pyt16] Python, O. “Site oficial da linguagem python”. Capturado em: <https://www.python.org/>, Nov 2016.
- [RN10] Russell, S. J.; Norvig, P. “Artificial Intelligence: A Modern Approach (Third Edition)”. Upper Saddle River, New Jersey, USA: Prentice Hall, 2010.
- [Roe12] Roelofs, G. “Monte carlo tree search in a modern board game framework”, *research paper available at umimaas. nl*, 2012.
- [RW89] Russell, S. J.; Wefald, E. H. “On optimal game-tree search using rational meta-reasoning”. In: IJCAI-89, 1989, pp. 334—340.
- [Sch09] Schadd, F. C. “Monte-carlo search techniques in the modern board game thurn and taxis”, Tese de Doutorado, Maastricht University, 2009.
- [SCS10] Szita, I.; Chaslot, G.; Spronck, P. “Monte-carlo tree search in settlers of catan”. In: Proceedings of the 12th International Conference on Advances in Computer Games, 2010, pp. 21–32.
- [SKW10] Soejima, Y.; Kishimoto, A.; Watanabe, O. “Evaluating root parallelization in go”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2–4, Dec 2010, pp. 278–287.
- [TH02] Thomas, R.; Hammond, K. “Java settlers: A research environment for studying multi-agent negotiation”. In: Proceedings of the 7th International Conference on Intelligent User Interfaces, 2002, pp. 240–240.
- [TLR85] T. L., L.; Robbins, H. “Asymptotically efficient adaptive allocation rules”. In: Advances in Applied Mathematics, 1985, pp. 4–22.

- [vdW05] van der Werf, E. "AI Techniques for the Game of Go". UPM, Universitaire Pers Maastricht, 2005.
- [vNM44] von Neumann, J.; Morgenstern, O. "The theory of games and economic behavior". In: Princeton University Press, 1944.
- [WGM73] Widrow, B.; Gupta, N. K.; Maitra, S. "Punish/reward: Learning with a critic in adaptive threshold systems", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3–5, Sept 1973, pp. 455–465.