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

UMA IA PARA O JOGO COLONIZADORES DE CATAN

BRUNO PAZ E GABRIEL RUBIN

Trabalho de Conclusão I 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

IBM – *International Business Machines*

MDP – *Markov Decision Process*

MCTS – *Monte-Carlo Tree Search*

POMDP – *Partially Observable Markov Decision Processes*

SUMÁRIO

AN AI FOR THE GAME SETTLERS OF CATAN

ABSTRACT

Colonizadores de Catan é um dos principais representantes dos jogos estratégicos modernos e tem muitas características que o tornam difícil de ser jogado por uma IA. Este jogo possui poucas implementações de IA disponíveis, as quais não representam desafio para um jogador. Neste trabalho, propomos pesquisar algoritmos de jogo para projetar uma nova IA capaz de jogar Catan com uma boa performance contra outros agentes e humanos.

Keywords: .

1. INTRODUÇÃO

2. 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. Este equilíbrio de jogo permite que um iniciante tenha chances de competir contra outros jogadores mais experientes. Além disso, o jogo possui profundidade estratégica suficiente para fomentar uma forte comunidade competitiva, que cresce desde o seu lançamento.

Existem diversas implementações de agentes que jogam Colonizadores de Catan disponíveis hoje. Podemos destacar 2 implementações como sendo as melhores [SCS10]. A primeira é proprietária da Castle Hill Studios, parte da MSN Games da Microsoft. A outra é um programa *open-source* desenvolvido por Robert S. Thomas em Java chamado JSettlers [TH02].

2.1 *Regras 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 pontos de vitória é considerado o vencedor da partida.

- Para obter pontos de vitória, os jogadores devem colonizar a ilha com construções e obter os recursos valiosos que a ilha oferece.

2.1.1 *Elementos do Jogo*

- O jogo é jogado em um tabuleiro formado por 19 partes hexagonais que representam o terreno de Catan, e 6 cartões que representam o oceano que cerca a ilha.

- O jogo conta com 2 dados de 6 posições e 18 fichas numeradas distribuídas entre os hexágonos. As fichas representam os valores que podem ser obtidos com o rolar destes 2 dados, com a exceção do número 7. Existem 2 fichas para cada número entre 3 e 11 e 1 ficha para os números 2 e 12.

- Cada jogador possui um conjunto de peças que representam suas construções na ilha de Catan. Estas peças são dividias entre 3 tipos: estradas, aldeias e cidades, e são pintadas de 4 cores, cada cor representando um dos jogadores. Além destas, existe uma peça especial que representa os ladrões de recurso da ilha.

- O jogo possui 2 baralhos de cartas distintos, um baralho representa os recursos que podem ser obtidos pelos jogadores e o outro representa cartas de desenvolvimento. Além destes baralhos, existem 2 cartas de pontos de vitória especiais, uma para a "Maior Estrada"e outra para a "Maior Cavalaria".

2.1.2 *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 (Figura - Tabela de preços).

- Cada jogador rola os dados em seu turno, e o valor obtido define quais hexágonos irão produzir recursos naquele turno, de acordo com as fichas numeradas.

2.1.3 *Construções*

- 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 produzem 1 carta de recurso e cidades produzem 2 cartas.

- 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 "a"se não houver nenhuma aldeia ou cidade nas 3 intersecções vizinhas "b", "c"e "d". (Figura). Aldeias podem ser promovidas para cidades.

- Estradas podem ser construídas nas laterais livres de cada hexágono. 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 inin-

terruptas, 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.

- Além de produzir recursos, aldeias e cidades também contam como 1 e 2 pontos de vitória respectivamente.

2.1.4 *Comércio de Recursos*

- Jogadores podem trocar recursos entre si ou com o banco de jogo. Quando o comércio 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 4:1.

- 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: 2:1 de algum tipo específico de recurso ou 3:1 de qualquer tipo de recurso.

- Jogadores que possuírem aldeias ou cidades próximas à algum dos portos podem usufruir da taxa de conversão deste porto.

2.1.5 *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.

2.1.6 *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"

- Cartas de "Cavalaria"podem ser usadas para mudar a posição da peça de ladrões, sem o descarte de recursos.

- Cartas de "Construção de Estrada"podem ser usadas para construção de 2 estradas de uma vez só sem qualquer custo.

- Cartas de "Monopólio"permitem que o jogador roube todas as cartas de um determinado recurso dos outros jogadores.

- Cartas de "Ano de Fartura"permitem 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.

2.1.7 *Começo da Partida e Fluxo de uma Jogada*

- O jogo começa com uma configuração de hexágonos, fichas numeradas e oceano montado, que formam o tabuleiro do jogo, e cada jogador possui um número de edificações no tabuleiro, minimamente, uma aldeia colocada em alguma posição valida. Jogadores podem criar várias configurações iniciais, iremos nos basear a configuração indicada para iniciantes no manual do jogo.

- Os jogadores jogam em turnos, onde cada turno é dividido em 3 partes: 1 - Rolar dos dados para definir a produção de recursos no turno. A produção vale para todos os jogadores. 2 - Troca de recursos com outros jogadores ou com o banco de jogo. 3 - Construção de recursos e compra de carta 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.

2.2 ***JSettlers***

Implementação na linguagem Java do jogo Colonizadores de Catan. Consiste em um cliente local que trata da lógica do jogo e um servidor com função de guardar os dados do jogo corrente, os dois se comunicam por mensagens. Os agentes criados utilizarão este cliente para jogar, e serão coletadas informações sobre os agentes implementados.

2.2.1 Interface

O cliente é composto por um tabuleiro, assim como no jogo original, e com informações mais explícitas de todos os jogadores, como mostra a Figura??.

Figura 2.1: Interface gráfica do cliente JSettlers.

2.2.2 Funcionamento

- Cliente/Servidor.
- As mensagens enviadas e recebidas pelo cliente, são texto com estrutura: "ID | conteúdo da mensagem".
- A classe que cuida do recebimento e envio de mensagens ao servidor é SOCServer.
- A classe SOCMessage manipula as strings recebidas do servidor, facilitando o uso interno (SOCPlayerClient).

- o ID é o identificador da mensagem, exemplo: ID = 1014 -> BOARDLAYOUT. O conteúdo esperado para este ID, por exemplo, é uma matriz do tabuleiro.
- O cliente manipula as mensagens de a cordo com o ID das mesmas.
- Toda ação local é enviada para o servidor atualizar a situação de jogo.
- O servidor guarda todas as informações do jogo atual, o cliente a lógica e regras de jogo.

3. TÉCNICAS DE IA PARA JOGOS

Esta seção apresenta as técnicas de IA que serão usadas na implementação dos agentes.

3.1 Minimax

O Minimax é um algoritmo que serve de base para grande parte dos algoritmos de busca em árvore de jogo [vdW05]. Composto por um conjunto de estados, jogadas e uma função de utilidade, para calcular quanto vale cada estado final do jogo, 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 da seguinte forma:

- De um certo estado do jogo, o algoritmo cria toda a árvore de jogo, e calcula a utilidade dos estados finais da árvore utilizando a função de utilidade.
- Partindo do estado final, a utilidade de cada estado intermediário do jogo é calculado subindo pela árvore, sempre trocando entre uma etapa de maximização de utilidade e outra de minimização, para simular a jogada de ambos os jogadores, partindo do pressuposto de que ambos vão sempre buscar as jogadas ótimas, que possuem maior utilidade.
- Finalmente, quando a raiz é atingida, a maior utilidade é escolhida dentre os estados imediatamente abaixo do estado raiz e uma jogada é feita pela máquina.

Os utilidade de cada estado intermediário gerados a partir dos valores de utilidade dos estados finais, são chamados de *Minimax value*, e são fundamentais para se descobrir a solução ótima do problema.

3.1.1 *Minimax value*

Para determinar a solução ótima do problema a partir da árvore gerada pelo algoritmo, será necessário consultar o valor atribuído para as jogadas terminais, como dito na seção anterior, e pode ser traduzido na Figura??, onde foram gerados todas as jogadas, gerado o valor de utilidade para cada jogada terminal e a partir daí voltar na árvore atribuindo um valor para cada nodo com base nos valores já gerados, este valor é o *Minimax value*. Por exemplo, na Figura??, a primeira jogada, *A*, é representada por uma ação do jogador (MAX), e pode ser seguida por três possíveis jogadas do oponente (MIN), *B*, *C* e *D*, cada

jogada do oponente (MIN) pode ser concluída com mais três jogadas do jogador (MAX), e como essas jogadas são terminais cada uma delas tem um valor de utilidade. Sabendo este valor, poderá ser atribuído os valores para as jogadas *B*, *C* e *D*, e como essas jogadas são do oponente (MIN), escolhe-se o menor valor, por exemplo, os valores terminais logo após a jogada *B* são: 3, 12 e 8, logo o valor de *B* será 3. Repetindo esse processo para todas as outras jogadas, finalmente chegamos na raiz da árvore, e como é uma jogada do jogador (MAX), escolhemos o maior valor gerado até então, no caso 3.

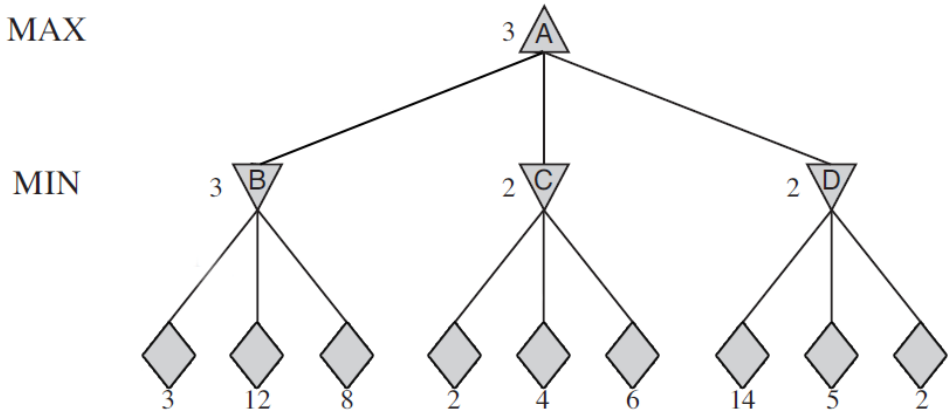

Figura 3.1: Árvore gerada pelo Minimax. [RN10, Cap 5, pp164]

3.1.2 Algoritmo do Minimax

Como dito nas seções anteriores, o Minimax gera toda a árvore de jogo, com todas as possíveis jogadas de todos os jogadores, e isso o torna muito custoso e complexo para jogos com muitas possibilidades, podendo ser avaliado como um algoritmo de força bruta.

Um passo a passo do algoritmo, onde MAX é o jogador e MIN o oponente:

1. A árvore é iniciada com a primeira possibilidade de jogada de MAX, por exemplo, o tabuleiro vazio.

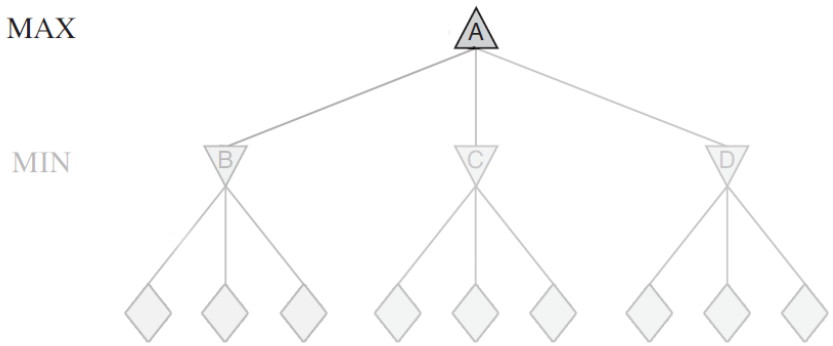

Figura 3.2: Árvore gerada pelo Minimax.

2. O passo seguinte é verificar se a jogada atual é terminal, senão ramifica para uma possibilidade de jogada do MIN.

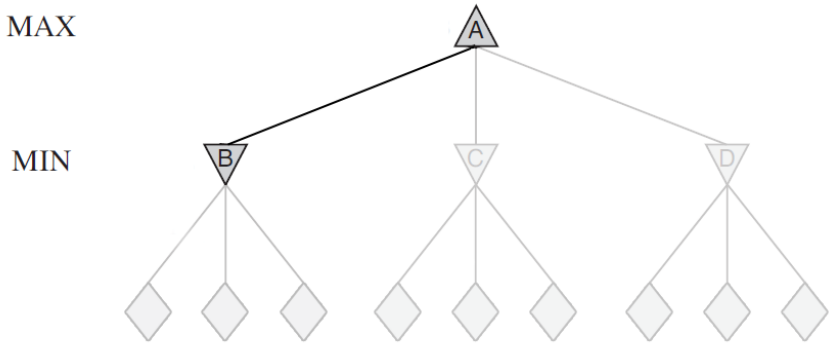

Figura 3.3: Árvore gerada pelo Minimax.

3. Repete-se o passo 2, mas agora ramificando para uma jogada de MAX. Como esta jogada é terminal, é gerado um valor de utilidade.

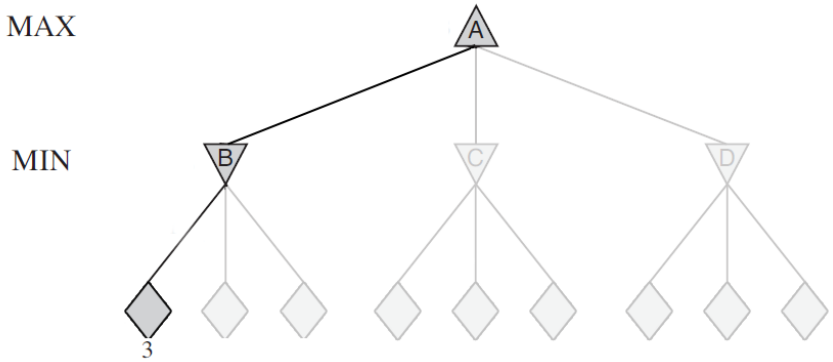

Figura 3.4: Árvore gerada pelo Minimax.

4. Como foi alcançado um nodo terminal da árvore, não existem mais jogadas possíveis para este nodo, então é necessário voltar para o nodo *B* e verificar se ainda existem jogadas possíveis a partir dele. No caso sim, e é repetido o passo 3.

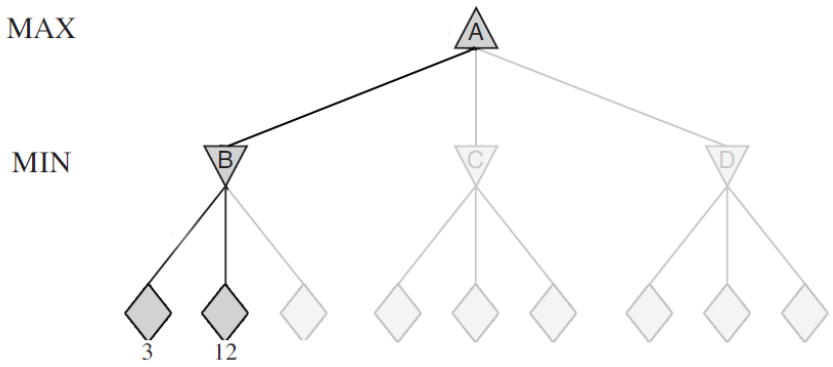

Figura 3.5: Árvore gerada pelo Minimax.

5. Repetição dos passos 3 e 2.

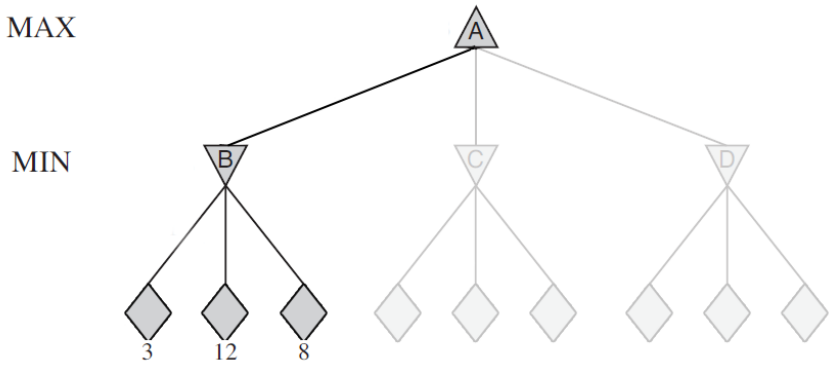

Figura 3.6: Árvore gerada pelo Minimax.

6. Todas as jogadas possíveis a partir do nodo *B* foram ramificadas e avaliadas pela função de utilidade, e sabemos que este nodo é uma possível jogada de MIN, logo atribuímos o Minimax value que será o menor valor de suas jogadas sucessoras para ele, no caso 3.

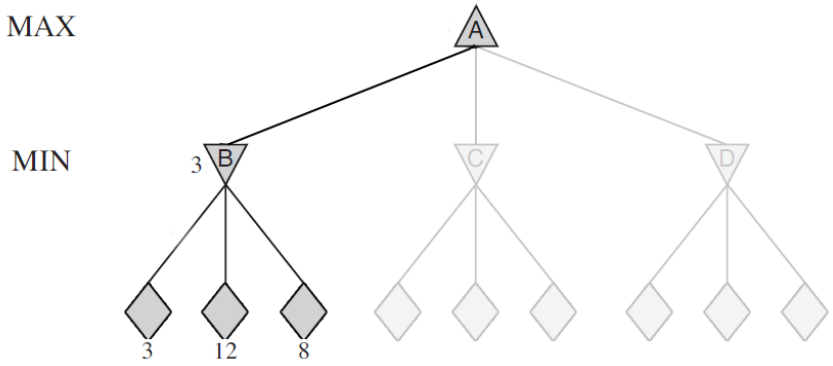

Figura 3.7: Árvore gerada pelo Minimax.

7. Repetindo os passos acima para todas as jogadas restantes, obtemos uma árvore com quase todas as jogadas avaliadas.

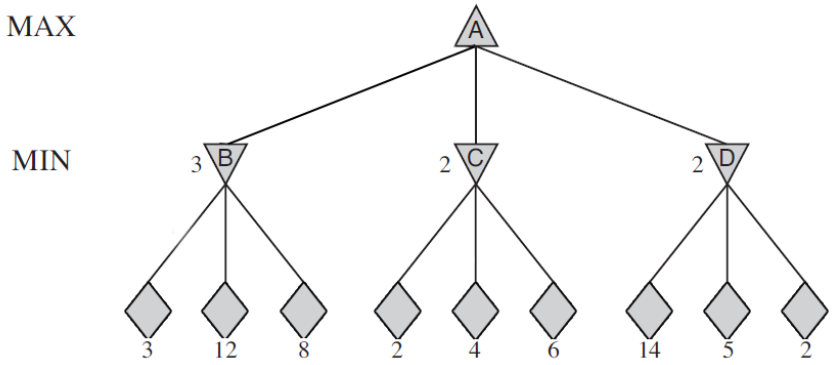

Figura 3.8: Árvore gerada pelo Minimax.

8. Concluindo o algoritmo, o nodo *A* é uma jogada de MAX, logo escolhemos o maior valor de suas jogadas sucessoras, no caso, 3. Este retorno é o passo que o jogador deverá seguir a fim de ganhar o jogo.

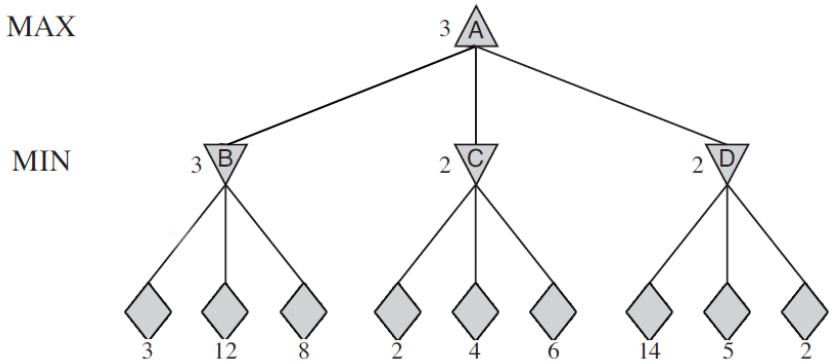

Figura 3.9: Árvore gerada pelo Minimax.

O exemplo mostrado anteriormente é um caso de um jogo para duas pessoas apenas, logo não serve para jogos como o Catan. O que faremos neste caso, é adaptar o algoritmo para permitir tal situação, visto que no caso anterior levamos em consideração apenas as jogadas de MAX em um andar da árvore e MIN logo no próximo e assim sucessivamente. Para um jogo de três jogadores, por exemplo, teremos que adicionar mais um andar para uma possível jogada do segundo oponente, deixando a árvore com um andar para uma jogada de A(MAX), um para uma jogada de B(MIN) e outro para C(MIN) mostrado na Figura??, repetindo o ciclo até o nodo terminal. A função de utilidade agora retorna um vetor de valores, um para cada jogador.

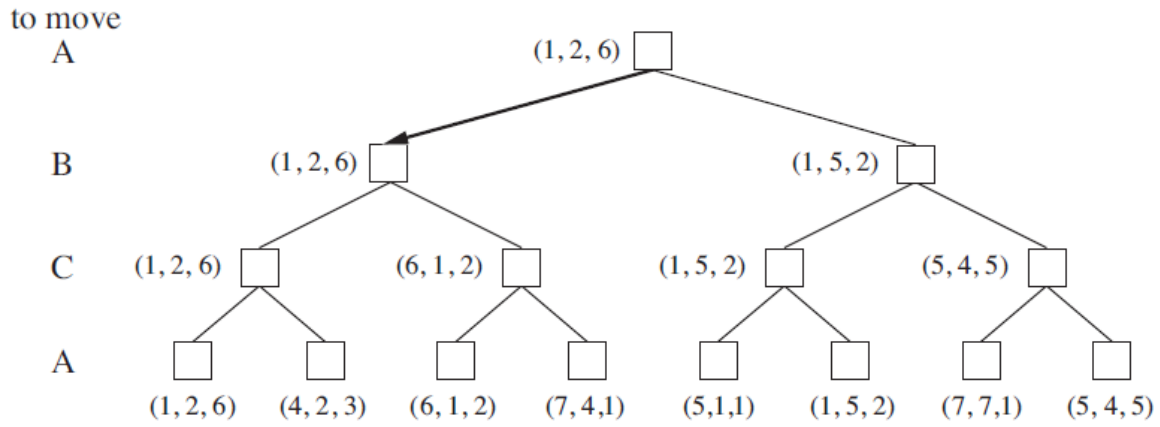

Figura 3.10: Árvore gerada pelo Minimax para mais de dois jogadores (*A*, *B* e *C*). [RN10, Cap 5, pp165-167]

- EXEMPLOS DE CÓDIGOS???.

3.1.3 *Alpha–Beta Pruning*

Alpha-Beta Pruning é uma modificação do algoritmo Minimax original que pode ser utilizada para desconsiderar alguns nodos da árvore que não irão influenciar na escolha da jogada. Devido à natureza do Minimax de maximizar uma etapa e minimizar a seguinte, alguns nodos da árvore não são relevantes durante o processo de seleção e, utilizando esta técnica, podemos descartar eles mesmo antes de considerar-los como é mostrado na Figura??. Conforme o caminhamento na árvore, são definidos dois valores, α e β:

- α = é o valor do melhor caminho já explorado até a raiz para o jogador (MAX).
- β = é o valor do melhor caminho já explorado até a raiz para o oponente (MIN).
- Utiliza o princípio dos máximos e mínimos para decidir continuar em uma ramificação ou descartá-la.

3.1.4 *Evaluation Function*

- O algoritmo assumi saber onde vai terminar a partir de um certo ponto da árvore.
- Jogadas não terminais, podem virar terminais.
- Substituí a função de utilidade por uma função de avaliação heurística no algoritmo original.
- No Alpha–Beta Pruning é adicionada a função de avaliação heurística ao teste de corte.

3.1.5 Expectimax

- Irá ser utilizada na implementação de um dos agentes.
- Leva em consideração elementos de chance, no caso do Catan é o resultado da rolagem dos dados.
- Custo do algoritmo é $O(b_m n_m)$, onde m é profundidade da árvore, b é o fator de ramificação e n é a quantidade de possibilidades na rolagem dos dados.

3.2 *Métodos de Monte-Carlo*

- Métodos de Monte-Carlo são métodos estatísticos muito usados na física e na matemática para aproximar integrais sem solução ou de solução computacionalmente cara. O uso destes métodos em jogos é antigo, Monte-Carlo Simulation foi aplicada no jogo de Blackjack em 1973 [WGM73].

- Monte-Carlo Tree Search (MCTS) é um algoritmo de busca em árvore moderno baseado em Monte-Carlo Simulation que possibilitou a criação de agentes competitivos no jogo Go, um jogo que representou grande desafio para a comunidade de Inteligência Artificial devido a sua complexidade, branching factor elevado, e falta de conhecimento heurístico sobre o jogo [GKS+12].

- Recentemente, o DeepMind do Google conseguiu vencer de um campeão de Go utilizando MCTS e técnicas de aprendizado [SHM+16].

- 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, observáveis e determinísticos [GS11]. Já MCTS é um algoritmo estatístico que pode ser parado à qualquer momento e ainda assim retornar um resultado satisfatório (anytime), é altamente escalável, e necessita de pouco ou nenhum conhecimento do domínio, fazendo dele um ótimo candidato para resolver jogos onde estas técnicas não obtiveram sucesso [BPW+12].

3.2.1 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 [GS11].

- 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 etapa, até atingir um estado final de jogo z_n.

- A cada etapa e da simulação, a política de simulação π, que é um mapeamento de estados e ações $\pi(s, a)$, é utilizada para selecionar uma das ações possíveis do estado atual s_i, e o próximo estado s_{i+1} é gerado a partir desta ação selecionada.

- Após um certo número de simulações ou tempo de processamento, um valor de recompensa estimado, ou *Q-Value*, é obtido para o estado inicial s_0 utilizando a média de todas as recompensas obtidas nas simulações realizadas, seguindo a seguinte fórmula

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

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 de s e $\mathbb{I}_i(s, a) z_i$ é 1 se a ação a foi a ação selecionada na simulação de índice i partindo do estado s e zero caso contrário.

- A política de simulação π é estática e geralmente aleatória, para garantir uma amostragem estatística que considera todo o espaço de pesquisa da árvore.

3.2.2 Monte-Carlo Tree Search

- Monte-Carlo Tree Search (MCTS) é um algoritmo moderno que utiliza Monte-Carlo Simulation para determinar o valor de uma ação em um determinado estado de uma árvore de pesquisa, ajustando a política de seleção a cada simulação, melhorando a política nas simulações subsequentes. O objetivo é chegar a uma política de seleção *best-first*, que sempre seleciona a melhor ação em cada estado visitado durante a simulação [BPW+12].

- O algoritmo constroi uma árvore de pesquisa iterativamente até que um limite de iterações ou tempo de processamento seja atingindo, retornando a melhor ação a ser tomada a partir do estado raiz da árvore. Os nodos da árvore guardam o seu *Q-Value* e a quantidade de vezes em que foram visitados s_v. Conforme o número de simulações aumenta, a árvore cresce e os *Q-Values* dos nodos se tornam mais precisos.

- Diferente de Monte-Carlo Simulation, onde existe uma única política fixa para as simulações, MCTS trabalha com duas políticas: uma *política padrão* fixa π, aplicada sobre

as simulações; e outra *política da árvore* π^* que gradualmente melhora a cada simulação, aplicada sobre a árvore de pesquisa construída.

- A cada iteração, são executados 4 passos, em sequência [BPW+12]:

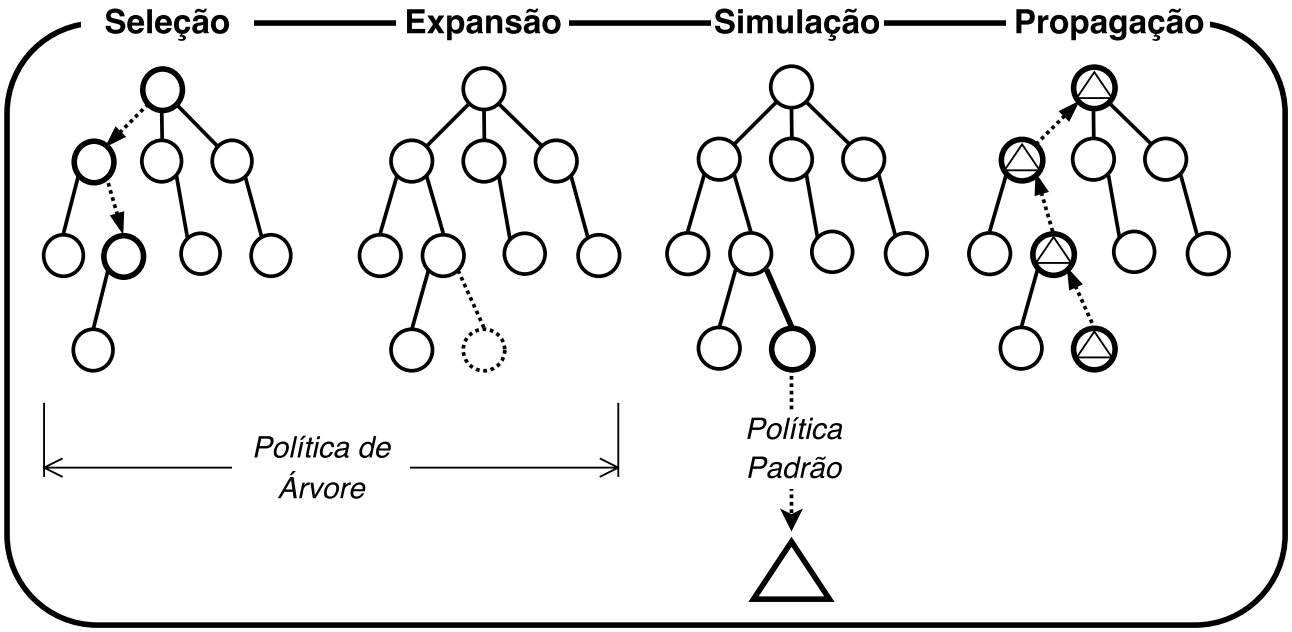

Figura 3.11: Fases do algoritmo de Monte-Carlo Tree Search.

- Seleção: Do nodo raiz, é aplicada recursivamente a política de seleção π^*, até chegar ao nodo expansível de maior urgência. Um nodo é expansível quando ele representa um estado não terminal ainda não visitado.
- Expansão: Um ou mais nodos são adicionados como filhos do nodo selecionado a partir das jogadas válidas possíveis a partir do nodo selecionado.
- Simulação: O jogo é simulado a partir dos nodos gerados na fase de expansão, seguindo uma política de seleção π.
- Propagação Inversa: Com a recompensa obtida na fase de Simulação, o *Q-Value* de cada nodo visitado durante a fase de seleção é atualizada, partindo do nodo adicionado até a raiz.

- Assim que a pesquisa é interrompida, um filho do nodo raiz é escolhido de acordo com uma função *MelhorFilho*(*s*). Existem 3 critérios usuais para escolher o melhor filho utilizando as estatísticas guardadas pelos nodos [Sch09]:

- Filho Max: Seleciona o filho do nodo raiz que tiver a maior recompensa.
- Filho Robusto: Seleciona o filho do nodo raiz com o maior total de visitas.

- Filho Max-Robusto: Seleciona o filho que tiver tanto a maior recompensa quanto o maior número de visitas. Se não exisitr um filho com ambos, continua a pesquisa até encontrar um filho com um total de visitas aceitavel.

- [PSEUDO CÓDIGO MCTS]

- 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 padrão* aleatória, mas sua performance pode ser aumentada significativamente incorporando conhecimento de domínio nesta política. A *política da árvore* tem de balancear exploração e lucro para evitar que o algoritmo fique *guloso* demais e não tenha resultados ótimos.

3.2.3 UCT

- UCT é uma versão de MCTS que utiliza o principio de *otimismo frente à incerteza*, que favorece o maior valor *potencial*, utilizando um método baseado em *multi-armed-bandit problem* como *política da árvore*, garantindo o equilíbrio entre *exploração* e *lucro* [GS11].

- *Multi-armed-bandit problem* é um problema de decisões sequenciais bastante conhecido, em que é preciso escolher ativar um entre várias alavancas possíveis em uma máquina caça-níqueis, buscando maximizar o total de recompensa obtido. Uma política de escolha que resolve este problema deve tentar minimizar o *arrependimento* do jogador. O *arrependimento* é a perda esperada por não escolher a melhor ação constantemente [TLR85].

- Um método chamado UCB1 resolve este problema com o menor crescimento de arrependimento possível frente ao número de escolhas do jogador [ACBF], garantindo o equilíbrio ótimo entre exploração e lucro. Em outras palavras, este método garante que a escolha ótima seja descoberta com a menor quantidade de exploração possível.

- [Método UCB1 e explicações]

- UCT aplica este método de seleção em MCTS tratando cada estado da árvore como um *Multi-armed-bandit problem*, que é resolvido utilizando uma adaptação do método UCB1. A *política da árvore* utilizada pelo UCT é a seguinte:

- [Método utilizado pelo UCT e explicações]

- Uma das grandes qualidades de UCT, é que a árvore gerada por este algoritmo eventualmente converge para uma árvore minimax, e portanto, gera uma árvore de pesquisa ótima [KS06].

- [PSEUDO CÓDIGO UCT]

4. APRENDIZADO DE MÁQUINA

4.1 Aprendizado por Reforço

4.1.1 Q-Learning

Dentre as técnicas de aprendizado por reforço, *Q-Learning* se diferencia das outras por ser uma técnica livre de modelo de transição [WD]. Em *Q-Learning* o agente aprende um valor de recompensa associado à uma tupla ação-estado que converge para resultados similares ao de se aprender o modelo de transição, o que significa que o agente necessita de menos conhecimento do domínio, simplificando o processo de aprendizado, porém restringindo o seu potencial de aprender em ambientes complexos, já que ele não poderá simular os resultados das ações possíveis. Enquanto outras técnicas visam aprender o modelo de transição da MDP desconhecida, esta técnica não necessita do modelo para aprender a resolver o problema.

4.1.2 Exploration and Bandits

4.2 *Deep Learning*

5. OBJETIVOS

Nesta etapa do projeto temos como objetivo implementar um agente na linguagem (Java/Python?) capaz de tomar decisões com base nas técnicas escolhidas, realizar uma comunicação entre o nosso agente com o cliente JSettlers e coletar estatísticas de jogo do agente.

5.1 Objetivo Geral

Implementar uma IA com as técnicas mostradas neste documento, capaz de jogar Colonizadores de Catan através do cliente JSettlers.

5.2 Objetivos Específicos

Foram definidos três grupos para os objetivos: fundamentais, desejáveis e opcionais.

1. Objetivos fundamentais
   (a) Criar um cliente capaz de se conectar ao servidor do JSettlers.
   (b) Criar um agente utilizando a técnica Expectimax.
   (c) Criar um agente utilizando o método de Monte-Carlo.
2. Objetivos desejáveis
   (a) Coletar estatísticas de jogo de cada agente criado.
3. Objetivos opcionais
   (a) Modificar o agente Monte-Carlo implementando Deep Learning.
   (b) Analisar e comparar os agentes Monte-Carlo com e sem Deep Learning.

6. MODELAGEM DO PROJETO

- Será implementado em Python.
- O cliente projetado terá que conectar-se ao servidor do JSettlers, e saber manipular as mensagens recebidas. Tendo a comunicação estabelecida, será possível gerar todos os dados necessários, como o tabuleiro sucessor por exemplo, permitindo que os agentes possam jogar da melhor maneira possível.
- No cliente terá a implementação da lógica de jogo na linguagem Python, semelhante a implementação em Java do JSetllers, onde os agentes consultarão as informações necessárias localmente para tomar decisões corretas com base no jogo atual rodando no servidor.
- O sistema de estatísticas irá em tempo real, guardar informações sobre os agentes.

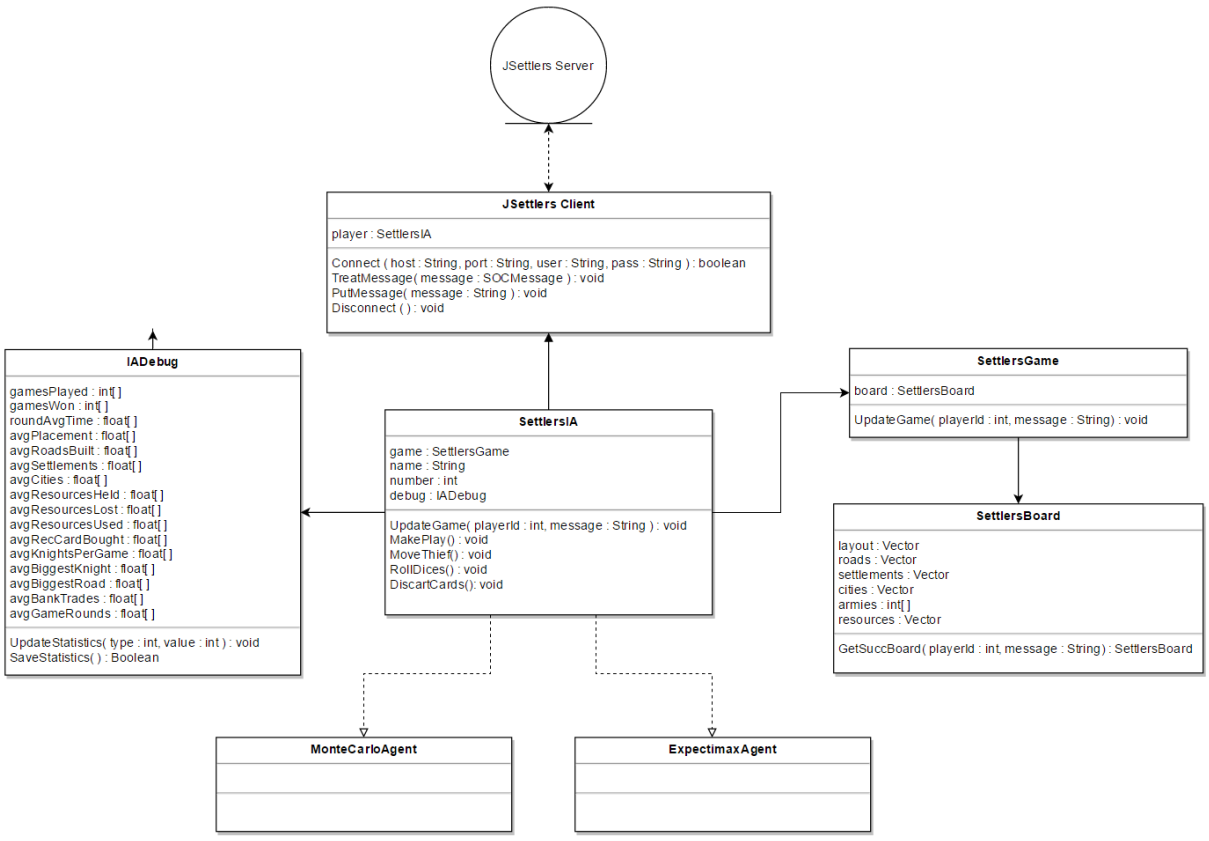

Figura 6.1: UML do nosso cliente.

7. ANÁLISE E PROJETO

Antes da implementação dos agentes será necessário implementar e configurar um cliente que conecte-se ao servidor do JSettlers, que trate as mensagens recebidas do servidor e crie mensagens para enviá-las de volta. Tendo o cliente implementado e configurado, podemos seguir para a próxima etapa, a implementação de um sistema para extrair informações de jogo. Esta será muito importante para a análise de performance dos agentes implementados na próxima e última etapa.

7.1 Atividades

- Configurar o ambiente de trabalho.
- Implementar um cliente capaz de conectar-se ao servidor do JSettlers
- Implementar um sistema capaz de coletar estatísticas de jogo dos agentes.
- Implementar um agente capaz de tomar decisões utilizando a técnica Expectimax.
- Implementar um agente capaz de tomar decisões utilizando o método de Monte-Carlo.
- Analisar as estatísticas dos agentes.

7.2 Cronograma

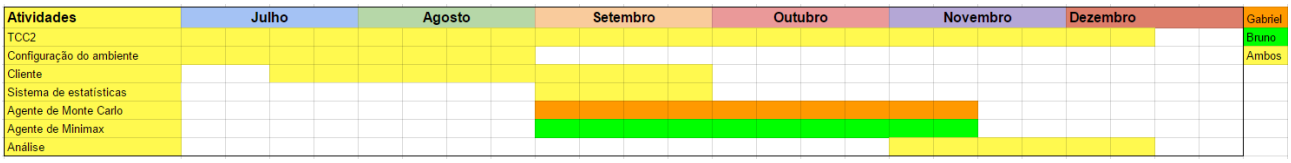

Figura 7.1: Cronograma das atividades planejadas.

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.

[BPW+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.

[GKS+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.

[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.

[KS06] Kocsis, L.; Szepesvári, C. “Bandit based monte-carlo planning”. In: Proceedings of the 17th European Conference on Machine Learning, 2006, pp. 282–293.

[RN10] Russell, S. J.; Norvig, P. “Artificial Intelligence: A Modern Approach (Third Edition)”. Upper Saddle River, New Jersey, USA: Prentice Hall, 2010.

[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.

[SHM+16] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al.. “Mastering the game of go with deep neural networks and tree search”, *Nature*, vol. 529–7587, 2016, pp. 484–489.

[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.

[WD] Watkins, C. J. C. H.; Dayan, P. “Q-learning”, *Machine Learning*, vol. 8–3, pp. 279–292.

[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.