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

ADVERSARIAL HIERARCHICAL-TASK NETWORK INTEGRADO COM APRENDIZADO POR REFORÇO PARA JOGOS EM TEMPO REAL

MATHEUS DE SOUZA REDECKER

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 Rech Meneguzzi

Porto Alegre
2016

ADVERSARIAL HIERARCHICAL-TASK NETWORK INTEGRADO COM APRENDIZADO POR REFORÇO PARA JOGOS EM TEMPO REAL

RESUMO

Jogos de estrategia em tempo real são difíceis ao ponto de vista da Inteligencia artificial(IA) devido ao grande espaço de estados e a limitação do tempo para tomar uma ação. Uma abordagem recentemente proposta é combinar busca adversária com técnicas de HTN, o algoritmo é chamado de *Adversarial Hierarchical-Task Network*. Para tentar melhorar o desempenho do algoritmo propomos uma unificação do algoritmo com técnicas de aprendizado por reforço.

Palavras-Chave: planejamento automatizado, HTN, busca adversária, aprendizado por reforço.

ADVERSARIAL HIERARCHICAL-TASK NETWORK INTEGRATED WITH REINFORCEMENT LEARNING FOR REAL-TIME GAMES

ABSTRACT

Real-time strategy games are hard from an Artificial Intelligence (AI) point of view due to the large state-spaces and the short time to compute each player's action. A recently proposed approach is to combine adversarial search techniques with HTN techniques, in an algorithm called Adversarial Hierarchical-Task Network. To improve the performance, we propose to integrate this algorithm to reinforcement learning techniques.

Keywords: automated planning, HTN, adversarial search, reinforcement learning.

LISTA DE FIGURAS

Figura 2.1 – Representação de um agente 11
Figura 2.2 – Arquitetura simples de um agente. 13
Figura 2.3 – Arquitetura de agentes com estados. 14
Figura 3.1 – Mapa para o exemplo de problema de busca 16
Figura 3.2 – Exemplo de um pedaço de uma *game tree* sobre o jogo da velha . . . 18
Figura 3.3 – Exemplo de game tree utilizando *minimax search* 18
Figura 4.1 – Problema de planejamento clássico. 22
Figura 4.2 – Exemplo de método HTN. 24
Figura 4.3 – Arvore de resolução HTN. 24
Figura 4.4 – Árvore gerada pelo algoritmo de AHTN 28
Figura 7.1 – Um exemplo de tela do MicroRTS 37
Figura 7.2 – Arquitetura MicroRTS 38
Figura 7.3 – Classes do MicroRTS 39
Figura 7.4 – Diagrama de sequência 40

LISTA DE ALGORITMOS

Algoritmo 3.1 – Minimax Search 19
Algoritmo 4.1 – Total-order Forward Decomposition 25
Algoritmo 4.2 – Adversarial hierarchical-task network 26
Algoritmo 5.1 – Q-Learning 32

LISTA DE SIGLAS

IA – Artificial Intelligence

HTN – Hierarchical Task Network

AHTN – Adversarial Hierarchical Task Network

RTS – Real-time Strategy

TFD – Total-order Forward Decomposition

SUMÁRIO

1 INTRODUÇÃO 9

2 AGENTES 11

2.1 AMBIENTES 12

2.2 ARQUITETURAS DE AGENTES 13

3 BUSCA 15

3.1 BUSCA ADVERSARIA 16

3.2 MINIMAX SEARCH 17

4 PLANEJAMENTO 20

4.1 PLANEJAMENTO AUTOMATIZADO 20

4.1.1 REPRESENTAÇÃO DE UM PROBLEMA DE PLANEJAMENTO 20

4.1.2 FORMALIZAÇÃO DE UM PROBLEMA DE PLANEJAMENTO 21

4.2 PLANEJAMENTO HIERÁRQUICO 22

4.3 AHTN 25

5 APRENDIZADO 29

5.1 APRENDIZADO DE MÁQUINA 29

5.2 APRENDIZADO POR REFORÇO 30

5.2.1 APRENDIZADO PASSIVO 30

5.2.2 APRENDIZADO ATIVO 31

5.2.3 GENERALIZAÇÃO 32

6 OBJETIVOS 34

6.1 OBJETIVO GERAL 34

6.2 OBJETIVOS ESPECÍFICOS 34

6.2.1 OBJETIVOS FUNDAMENTAIS 34

6.2.2 OBJETIVOS DESEJÁVEIS 34

7 ANÁLISE E PROJETO 36

7.1 JOGOS 36

7.1.1 JOGOS DE ESTRATÉGIA EM TEMPO REAL 36

7.1.2 MICRORTS 37

7.1.3 ARQUITETURA DO MICRORTS 38
7.2 DESCRIÇÃO DO PROJETO 39

REFERÊNCIAS **41**

1. INTRODUÇÃO

Inteligência artificial (IA) é uma área em ciência da computação que tem como objetivo fazer com que o computador seja capaz de realizar tarefas que precisam ser pensadas, como é feito pelas pessoas. A IA possui uma área de aplicação em jogos, fazendo com que os computadores sejam capazes de jogar jogos como xadrez e jogo da velha, por exemplo. Os Jogos de computador muitas vezes não utilizam algoritmos de decisão autônoma ótimos para simular o comportamento de jogadores; utilizando ao invés disso técnicas que passam a ilusão de que as decisões estão sendo realizadas de forma autônomas, ou ainda utilizam técnicas que se aproveitam das informações provenientes do controle do jogo. Esse tipo de técnicas não pode ser caracterizado como uma técnica de IA totalmente autônoma [6].

Jogos que utilizam técnicas de IA conseguem prover uma melhor interação entre o jogador e o jogo, tornando o jogo mais real e assim prendendo a atenção do jogador [6]. Entretanto, os métodos utilizados, são geralmente mais simples do que os utilizados no meio acadêmico, pelo fato de que o tempo de resposta dos algoritmos é superior ao tempo que se tem para tomar uma ação ótima dentro do jogo e também pelo fato das ações geradas são mais previsíveis [13]. Nos jogos de computador as reações dos oponentes devem ser quase que imediatas, por esse motivo técnicas que tentam explorar todo o espaço de estados possíveis de um jogo se tornam inviáveis para jogos com uma complexidade maior. Por exemplo, no xadrez a quantidade aproximada de estados possíveis é de 10^{40}, isso mostra que o poder de processamento para gerar, de maneira rápida, uma ação precisa ser alto [6]. Então é difícil conseguir gerar uma ação ótima, em alguns casos, são gerados ações sub ótimas para que o tempo de resposta não seja muito alto [13].

Os jogos de estratégia em tempo real são jogos onde os jogadores devem construir uma base, conseguindo coletar recursos e traçar batalhas com o objetivo de derrotar os seus oponentes, jogos muito conhecidos como *StarCraft*[1] e *Age of Empires*[2] são exemplos de jogos desse gênero [12]. Neste gênero de jogos é preciso decidir quais ações precisam ser tomadas, uma maneira de se fazer isso é utilizando técnicas de busca. As técnicas de busca almejam conseguir, uma técnica que consegue definir com essas ações é chamada de busca. As técnicas de busca adversária conseguem determinar qual a próxima ação que deve ser tomada com o objetivo de ganhar o jogo. O problema dessas técnicas é que elas devem explorar pelo menos uma parte do espaço de estados, e com isso jogos que tenham uma grande quantidade de jogadas possíveis, esse tipo de técnica se tornar inviável, pelo tempo que é preciso para gerar uma ação [9].

[1] http://us.battle.net/sc2/pt/
[2] http://www.ageofempires.com/

Na busca de mitigar as limitações de eficiência computacional de abordagens tradicionais de raciocínio em jogos, Ontañón e Buro propuseram o algoritmo chamado *Adversarial Hierarchical Task Network (AHTN)* [11]. Neste algoritmo são combinadas técnicas de planejamento hierárquico com as de busca adversaria. Com o intuito de obter uma melhor performance na escolha das ações, propomos a utilização do algoritmo AHTN em conjunto com um algoritmo de aprendizado por reforço na plataforma MicroRTS[3], que é um jogo de estratégia em tempo real. Com este trabalho pretendemos mostrar que o algoritmo de AHTN apresenta melhores resultados quando aplicado junto com técnicas de aprendizado por reforço.

Este documento é organizado da seguinte forma: No Capítulo 2 é apresentado o conceito de agentes e como podemos representar ele dentro dos diferentes problemas existentes. O Capítulo 3 apresenta como podemos utilizar busca dentro do contexto de jogos, o Capítulo 4 trata o conceito básico de planejamento para conseguir entender como o algoritmo de AHTN funciona. O Capítulo 5 apresenta técnicas de aprendizado de máquina que podem ser usadas em jogos. No Capítulo 6 é proposto os objetivos para este trabalho, enquanto no Capítulo 7 é onde são definidas as atividades que serão realizadas em seguida, junto com a modelagem do problema.

[3]https://github.com/santiontanon/microrts

2. AGENTES

Os agentes são utilizados em jogos como uma abstração que represente os jogadores. Os agentes conseguem absorver informações providas do jogo e assim decidir qual o próximo passo a ser tomado [6].

Formalmente, agentes são entidades que agem de forma continua e autônoma em um ambiente [14]. Os agentes são capazes de receber estímulos do ambiente através de sensores, e assim responder aos estímulos por intermédio de atuadores. Para os agentes os estímulos do ambiente são recebidos como percepções. Os atuadores por sua vez, geram, uma ação considerando as percepções [13]. A interação de um agente com o ambiente pode ser ilustrado pela Figura 2.1.

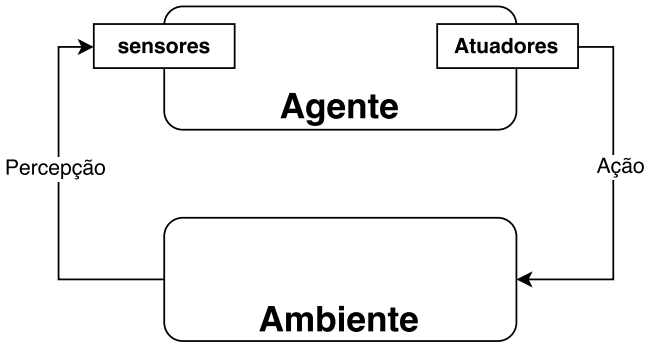

Figura 2.1 – Representação de um agente

O agente deve agir de forma autônoma, para isso ele deve ser capaz de aprender a lidar com situações proporcionadas pelo ambiente, com o intuito de realizar ações em busca do seu objetivo. O agente precisa de três características para conseguir ser autônomo [15]. São elas:

- reatividade, para que os agentes sejam capazes de perceber o ambiente e suas mudanças a fim de levar o ambiente em consideração para a tomada de decisão das ações;
- pró-atividade, para que os agentes consigam ter a iniciativa em tomar as suas ações; e
- habilidade social, para que os agentes sejam capazes de interagir com outros agentes(humanos ou não).

As duas primeiras características são necessárias para que o agente consiga interagir com o ambiente. Um agente sendo reativo, ele consegue, a partir de uma mudança do ambiente, saber como ele deve se comportar. Sendo proativo, o agente pode antecipar suas ações em busca do seu objetivo. Nem sempre um agente vai estar sozinho no ambiente, por esse motivo, a terceira característica é necessária para que o agente consiga

interagir com outros agentes. Sistemas onde existem mais de um agente são chamados de sistema multi agentes. Nesses sistemas, os agentes interagem entre si, podendo ter objetivos em comum ou não. Sendo assim eles terão que cooperar ou negociar entre si [13].

2.1 Ambientes

O agente deve se comunicar com um ambiente para conseguir alcançar seus objetivos. Mas um ambiente é composto por diversas propriedades que podem influenciar como o agente vai agir para chegar ao seu objetivo [13].

Nem sempre todas as informações do ambiente estarão disponíveis, por esse motivo o ambiente pode ser dito como completamente observável, parcialmente observável ou não observável, dependendo da informação disponibilizada. Um ambiente é dito completamente observável se, em qualquer instante de tempo, todas as informações relevantes do ambiente estão disponíveis para os sensores do agente. Caso haja alguma informação que não possa ser acessada, em algum instante de tempo, seja por causa da incapacidade do sensor do agente de captar essas informações ou pelo fato da informação simplesmente não ser disponibilizada, o ambiente é dito parcialmente observável. Agora, se o ambiente não disponibiliza nenhuma informação, o ambiente é tido como não observável [13, 15].

O ambiente pode sofrer modificações, as modificações podem ser provenientes de ações realizadas pelos agentes, ou ainda por mudanças ocasionadas pelo próprio ambiente. O ambiente é determinístico se o estado gerado após a execução de uma ação, em todas as vezes que for executada, levar para o mesmo estado resultante, ou seja, o estado resultante é determinado pelo estado atual e a ação executada pelo agente. Se não há a certeza do estado resultante, o ambiente é estocástico. Quando o ambiente é não determinístico existem chances das ações dos agentes nem lembre levarem para os estados conhecidos [13].

Os estados do ambiente irão mudar ao longo do tempo, seja por uma ação feita por algum agente, ou por alguma mudança que possa ocorrer em razão de outro processo do ambiente. Se o ambiente sofre alguma alteração apenas quando o agente executa alguma ação, o ambiente é estático. Se o ambiente tem a capacidade de mudar independente de uma ação de um agente, o ambiente é dinâmico [15].

Em sistemas multi agentes os agentes podem estar competindo ou cooperando entre si. O ambiente é competitivo quando os agentes estão competindo, como em um jogo de xadrez, por exemplo, ou o ambiente é cooperativo quando os agentes estão cooperando [13].

2.2 Arquiteturas de Agentes

O tipo mais simples de agente é aquele que apenas reage a uma percepção vinda do ambiente. O agente escolhe suas ações baseado no que percebe no momento da decisão, sem levar em consideração ações já tomadas ou percepções anteriores. O agente apenas responde a uma percepção com uma ação, como se houvesse um clausula condicional que determinasse qual a ação a ser tomada se acontecer alguma coisa, como por exemplo, se estiver chovendo eu irei levar um guarda-chuva. A Figura 2.2 ilustra essa arquitetura.

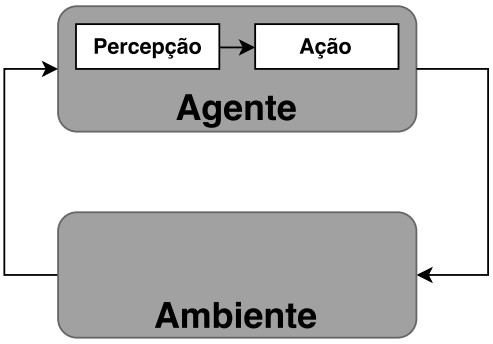

Figura 2.2 – Arquitetura simples de um agente.

Este tipo de agente consegue ser simples, no entendimento e na sua utilização, mas a sua inteligência é limitada. Essa arquitetura é eficaz em ambientes completamente observáveis, pelo fato de que o agente precisa da percepção para realizar a sua ação [13]. A arquitetura pode ser usada, por exemplo, quando for preciso conhecer um conjunto de cidades, o agente após chegar a uma cidade, vai para a próxima cidade ao norte da cidade atual, quando não há norte ele vai para oeste.

Na tentativa de aprimorar as decisões tomadas pelo agente, pode-se usar um estado interno para marcar qual a situação do ambiente. A informação representada no estado pode ser alguma informação que não consiga ser obtida por alguma percepção do ambiente ou de estados que já foram visitados pelo agente, por exemplo [13]. A Figura 2.3 ilustra esta arquitetura.

Este tipo de arquitetura é eficaz para ambientes parcialmente observáveis, pelo fato de que o estado pode guardar informações relevantes para o agente [13]. No exemplo de conhecer as cidades, se guardar as visitas antigas, pode ajudar a não visitar novamente cidades que já tiverem sido visitadas.

Dependendo do intuito do agente, conhecer o estado atual do ambiente não é suficiente. Além de estado, o agente pode precisar de uma informação para saber onde ele quer chegar, ou seja, um objetivo. Um objetivo é usado para descrever o que o agente está almejando alcançar. Para o agente conseguir alcançar o objetivo com uma melhor per-

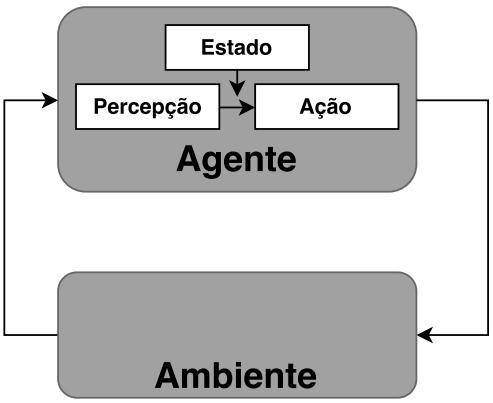

Figura 2.3 – Arquitetura de agentes com estados.

formance pode ser utilizado uma função de utilidade, nesta função é medido o "desejo"do agente em tomar determinada ação. Cada ação exercida pelo agente terá influência no valor de utilidade obtido [13]. Seguindo no exemplo, o objetivo pode ser visitar todas as cidades e a função de utilidade pode medir a distância entre as cidades, e assim podendo alcançar o objetivo com a menor distância percorrida.

3. BUSCA

Com o intuito de encontrar uma sequência de ações que um agente consiga alcançar o seu objetivo, é possível utilizar técnicas de busca. O princípio da busca é encontrar uma solução de um problema através de um conjunto de ações que alcancem o objetivo desejado. Para utilizar as técnicas de busca é preciso formalizar o problema a ser resolvido e o objetivo a ser alcançado. Para utilizar as técnicas de busca, como entrada é recebido um problema e como resolução é retornado um conjunto de ações. Um problema pode ser definido por cinco componentes [13]:

- s_0, o estado inicial, onde o agente inicia no ambiente;
- ações, conjunto das possíveis ações disponíveis no agente;
- *resultado*(s, a), um modelo de transição, que define o estado resultante após a execução da ação a no estado s;
- *objetivo*(s), verifica se o estado é o objetivo do agente; e
- custo do caminho, uma função que define um valor numérico para cada ação realizada em um estado. Essa função pode ser denotada por $c(s, a, s')$, onde s é o estado atual do agente, a é a ação que será aplicada ao estado s e s' é o estado resultante aplicando o modelo de transição resultado(s, a).

Esses elementos são necessários como entrada para que as técnicas de busca consigam ser utilizadas. As técnicas de busca visam encontrar a solução para esse problema, ou seja, a sequência de ações que leve ao objetivo. A solução é obtida quando o agente iniciando no estado inicial do ambiente s_0, utiliza o modelo de transição *resultado*(s, a), através das ações aplicadas nos estados, para chegar a um estado onde satisfaça o objetivo do agente *objetivo*(s). Existem diferentes técnicas para encontrar uma solução. As diferentes técnicas podem encontrar caminhos diferentes para o mesmo problema, isso vem do fato de que o custo do caminho, quando levado em consideração, pode encontrar uma solução ótima, o que significa que a sequência de ações encontrada é a que tem o menor custo de caminho entre todas as soluções [13].

Com a finalidade de exemplificar um problema de busca, considere o mapa apresentado na figura 3.1. Cada círculo representa uma cidade, e as linhas entre as cidades são estradas que ligam as cidades uma a outra. A única ação possível é a de se locomover entre as cidades que tenham ligação. O estado inicial é a cidade de São Jerônimo. O resultado do modelo de transição, é a cidade resultante após se locomover. O objetivo do agente é chegar na cidade de Porto Alegre e o custo de cada ação é o valor definido em cima de cada transição.

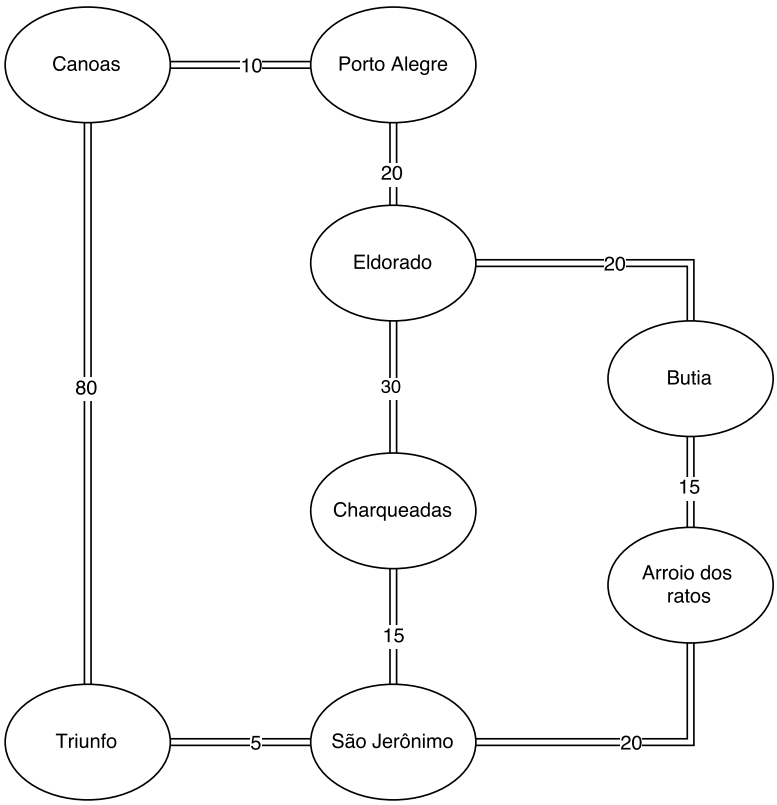

Figura 3.1 – Mapa para o exemplo de problema de busca

Para atingir o objetivo é preciso tentar os possíveis caminhos até o objetivo. Digamos que o agente comece sua viagem indo para a cidade de Triunfo, para nós(humanos) é intuitivo que a escolha não foi a melhor escolha para iniciar, mas a técnica só terá como saber após realizar todas as possíveis opções de caminhos, ou se utilizar uma técnica que utilize uma função de heurística, levando em consideração os custos dos caminhos, que acrescentam um conhecimento extra para a resolução do problema [13].

3.1 Busca adversaria

Jogos são difíceis de resolver com técnicas de IA, pois eles requerem a habilidade de tomar algum tipo de decisão, e as técnicas comuns as vezes não são satisfatórias, seja pelo fato do conjunto de estados possíveis de se atingir ser muito grande, ou pelo curto espaço de tempo para tomar essa decisão. A busca adversaria é utilizada nos jogos que estão situados em ambientes competitivos e de multi agentes. Em um jogo o jogador, preferencialmente, não informa suas jogadas previamente, assim tornando o ambiente imprevisível. Os objetivos dos jogadores entram em conflito, pois ambos estão em busca da vitória. Com o intuito de resolver esse problema é possível gerar uma solução de contingência para tentar antecipar as jogadas do adversário [13].

As técnicas de busca adversária utilizam uma variação da definição de um problema de busca comum. Os componentes devem se adequar ao ambiente competitivo. Por esse motivo os componentes são redefinidos como:

- s_0, sendo o estado inicial, que especifica como o jogo se configura no início;
- *players*(s), define qual jogador tem o movimento no estado s;
- *actions*(s), conjunto das ações possíveis em um estado s;
- *result*(s, a), um modelo de transição, que define o resultado da ação a a aplicada ao estado s;
- *terminal*(s), verifica se o estado s é um estado onde o jogo termina; e
- *utility*(s, p), define um valor numérico, representando o lucro do jogador p ao atingir o estado terminal s.

Com esses componentes descritos é possível formalizar o que é uma árvore de jogadas. A árvore de jogados ou *game tree* contém os estados do jogo e os movimentos possíveis em cada estado. A *game tree* é composta pelo estado inicial s_0, as ações *action*(s) e o modelo de transição *result*(s, a), e possui uma profundidade d, que indica o nível máximo da árvore. A árvore representa em cada nodo um estado do jogo e em cada ligação os estados resultantes após a execução de cada ação possível para o estado. Considerando o jogo de jogo da velha, onde cada jogador realiza uma jogada de cada vez, uma *game tree* que mostra parte das jogadas do jogo da velha é ilustrada na Figura 3.2. O estado inicial do jogo é o campo vazio, a cada nível da árvore todas as possibilidades de jogadas são testadas, a profundidade d dessa árvore chega a 9 quando ela estiver completa, pois a cada nível da árvore uma jogada é marcada no campo.

3.2 Minimax search

O algoritmo de *minimax search* é utilizado como uma técnica de busca adversária. O objetivo do algoritmo é retornar a melhor jogada para o estado atual. Esse método considera dois agentes, chamados de MAX e MIN, onde o jogador MAX representa a perspectiva do agente que está tentando maximalizar suas ações em relação ao agente MIN, que representa o agente adversário do jogador MAX. O algoritmo alterna entre jogadas de MAX e MIN [13].

O algoritmo utiliza a *game tree* para analisar todos os estados possíveis do jogo, e assim decidir qual a ação que quando aplicada ao seu estado atual, trará um melhor benefício no futuro, se caracterizando a melhor jogada. Os nodos folhas da árvore, que

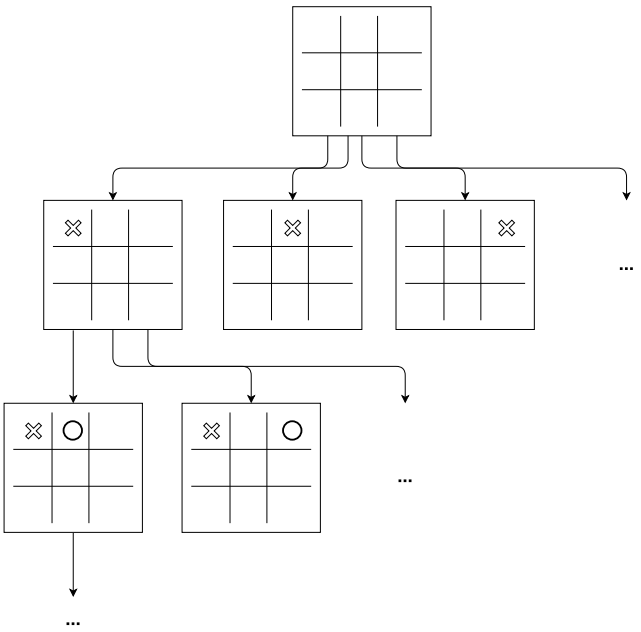

Figura 3.2 – Exemplo de um pedaço de uma *game tree* sobre o jogo da velha

representam o final do jogo, contém um valor de utilidade. Os valores mais altos são as melhores jogadas para MAX, e consequentemente, os valores menores são melhores para MIN. Ao chegar no final da árvore o algoritmo consegue o valor de utilidade para aquele cenário do jogo, quando isso acontece, o algoritmo faz o caminho inverso na árvore analisando os outros possíveis cenários [13]. A Figura 3.3 ilustra esse processo.

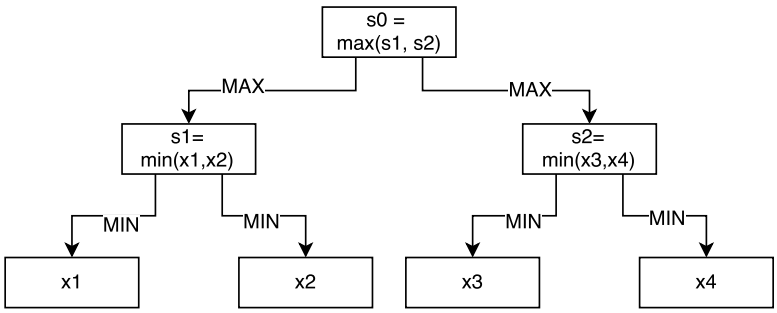

Figura 3.3 – Exemplo de game tree utilizando *minimax search*

O método de *minimax search* assume que os todos os jogadores como sendo racionais, com isso o algoritmo considera que as ações tomadas pelos agentes sempre realizarão uma jogada para tentar ganhar o jogo. O algoritmo de *minimax search* é ilustrado no Algoritmo 3.1. O algoritmo tem como retorno a melhor ação a ser realizada no estado atual. As funções presentes nas linhas 5 e 15 são utilizadas para calcular a jogada na visão de MAX e MIN respectivamente. A função presente na linha 1 é utilizada para iniciar a recursão e ao final retornar o melhor valor de utilidade para o jogador MAX.

O algoritmo de *minimax* deve explorar todo o espaço de estados para conseguir encontrar a ação que deve ser executada. A quantidade de estados possíveis, dependendo da situação, pode ser muito alta, no xadrez esse número chega a 10^{50}, em um jogo de *poker*

Algoritmo 3.1 – Minimax Search

```
1: function MINIMAX(state)
2:     return argmax_{action ∈ actions(s)} min_value(result(state, action))
3: end function
4:
5: function MAX_VALUE(state)
6:     if terminal(state) then
7:         return utility(state)
8:     end if
9:     v = −∞
10:    for all action ∈ actions(state) do
11:        v = max(v, min_value(result(state, action)))
12:    end for
13: end function
14:
15: function MIN_VALUE(state)
16:    if terminal(state) then
17:        return utility(state)
18:    end if
19:    v = ∞
20:    for all action ∈ actions(state) do
21:        v = min(v, max_value(result(state, action)))
22:    end for
23: end function
```

no estilo *texas holdem* esse número pode chegar a 10^{80}. Geralmente, as ações devem ser tomadas em uma quantidade de tempo muito pequena. Por esse motivo, utilizar técnicas de busca para jogos pode ser um problema. Existem algumas abordagens que utilizam busca adversaria com um nível de abstração mais alto para tentar minimax esse problema [12].

4. PLANEJAMENTO

A atividade de planejar alguma coisa é feita muitas vezes por dia. Algumas vezes esse planejamento é muito simples, como organizar as tarefas que serão feitas no próximo dia, outras vezes pode ser mais complexa como organizar uma viagem de final de ano. Planejar implica em elaborar uma sequência de ações com o intuito de alcançar um objetivo, ou seja, o processo de planejar consiste em organizar as ações, antecipando os resultados esperados de cada ação, com o intuito de conquistar um objetivo.

O planejamento, na área da IA, é uma sub área de estudo, onde ele é usado para encontrar um plano que resolva um problema especifico. Como os ambientes nem sempre possuem as mesmas características, existem diferentes técnicas que são usadas para construir um plano.

4.1 Planejamento automatizado

Planejamento automatizado estuda o processo de geração de planos computacionalmente. O objetivo do planejamento é encontrar uma sequência de ações que solucione um problema, a sequência de ações encontrada é chamada de plano. Para construir um plano é utilizado um planejador. O planejador recebe uma descrição formal de um problema de planejamento, e tenta solucionar esse problema utilizando algoritmos de buscas e heurísticas [8, 13].

4.1.1 Representação de um problema de planejamento

Uma das maneiras de representar um problema de planejamento é utilizando lógica matemática. A lógica proposicional é expressada através de sentenças atômicas, que são compostas de proposições. Cada proposição pode assumir um valor de verdadeiro ou falso. Por exemplo, queremos representar que a lâmpada está apagada, para isso pode ser utilizado a proposição *apagada*, se a lâmpada estiver apagada, a proposição assume o valor de verdadeiro. Junto com as proposições podem ser usados conectivos lógicos, como negação (*not*) $\neg$, conjunção (*and*) $\wedge$ e disjunção (*or*) $\vee$. No exemplo da luz, se queremos saber se a luz está apagada e a TV está ligada, podemos representar por *apagada* $\wedge$ *ligada*. A lógica proposicional pode ser considerada simples, mas ela serve de base para as logicas mais expressivas [13].

Como a lógica proposicional tem expressividade limitada, é preciso utilizar uma lógica que consiga resolver esse problema. A lógica de primeira ordem (LPO) estende a

lógica proposicional. Na LPO uma sentença atômica é composta por um predicado seguido de uma lista de termos, denotada por $p(t_0, t_1, ..., t_n)$. Um predicado se refere a uma relação existente entre os termos. Os termos são objetos que se referem a objetos definidos, indefinidos ou a funções [13]. No exemplo da lâmpada, podemos dizer que a lâmpada da cozinha está apagada, representado por *apagada*(*cozinha*). Ou ainda podemos dizer que a lâmpada do quarto está apagada e a TV da sala está ligada, *apagada*(*quarto*) $\wedge$ *ligada*(*sala*).

4.1.2 Formalização de um problema de planejamento

Como nas técnicas de busca, em planejamento também é necessário ter uma descrição do problema. Para realizar a descrição de um problema de planejamento é preciso definir alguns conceitos, são eles [13, 8, 5]:

- estados, onde cada estado é um predicado. Conforme a situação do ambiente os estados assumem os valores de verdadeiro ou falso;
- operadores, cada operador é definido como *op* = (*nome*(*t*), *pre*(*p*), *efeitos*(*p*). O *nome*(*t*) é o nome do operador e t é o conjunto de termos que irão aparecer nas precondições e efeitos. *pre*(*p*) é o conjunto de predicados que representam as precondições do operador. *efeitos*(*p*) é o conjunto de predicados que serão o resultado após a execução do operador; e
- domínio, o conjunto de operadores que podem ser usados para a resolução do problema.

Os operadores também são chamados de ações. As ações causam uma alteração no ambiente. Um exemplo é mover um objeto de um lugar para o outro, precisamos de um estado que defina que um objeto está em determinado lugar. Esse exemplo é representado abaixo:

- Ação(move(from, to))
- Precondição: at(from)
- Efeito: $\neg$ at(from) $\wedge$ at(to)

Uma ação *a* é aplicável em um estado *s*, se todas as precondições forem satisfeitas no estado *s*. O resultado gerado pela execução de *a* no estado *s* é um novo estado *s*′, nesse estado é aplicado todos os efeitos, removendo os predicados negativos e adicionando os positivos.

Formalmente, um problema de planejamento pode ser descrito como $P = (\Sigma, s_0, g)$, onde Σ é o domínio do problema, s_0 é o estado inicial onde o problema começa e g é o objetivo [8]. Voltando ao exemplo do Capítulo 3, onde um agente tenta chegar a cidade de Porto Alegre partindo da cidade de São Jerônimo, podemos formaliza-lo como:

- **estado inicial** (s_0) = $At(São\ Jerônimo)$;
- **objetivo** (g) = $At(Porto\ Alegre)$; e
- **domínio** (Σ) =
  - nome: $move(cityA, cityB)$
  - precondições: $at(cityA) \wedge link(cityA, cityB)$
  - efeitos: $\neg at(cityA) \wedge at(cityB)$.

O processo de geração do plano é feito pelo planejador. Um plano é uma sequência de operadores gerada a partir de um problema. A sequência de operadores quando executados a partir do estado inicial alcança o objetivo. A Figura 4.1 ilustra o comportamento do planejador.

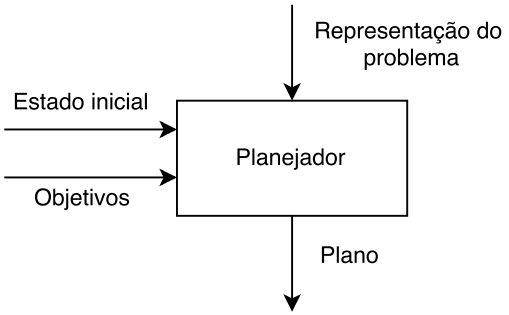

Figura 4.1 – Problema de planejamento clássico.

Um plano é considerado ótimo quando utiliza o menor número de ações para resolver o problema, ou seja, o número de ações que levam o estado inicial até o objetivo é o menor possível. Para gerar planos ótimos o planejador utiliza técnicas de busca combinado com alguma heurística que seja independente de domínio. Uma heurística deve ser computacionalmente eficiente e ter precisão para que a geração do plano seja eficiente [2].

4.2 Planejamento hierárquico

Embora que o planejamento clássico consiga gerar planos ótimos, a quantidade de ações resultante é grande para problemas maiores. O planejamento clássico possui uma expressividade limitada, pois não é possível representar quando uma ação irá ocorrer, por

exemplo [13]. Como alternativa para esses problemas foi proposto o planejamento hierárquico, chamado de *Hierarchical Task Network* (HTN). O planejamento HTN se diferencia do planejamento clássico na forma como os planos são gerados [8]. Enquanto no planejamento clássico é necessário ter heurísticas para a geração de planos, em planejamento HTN é possível expressar conhecimento de domínio junto com os operadores, isso faz com que as ações sejam tratadas em mais alto nível [13].

O objetivo do planejamento HTN é produzir uma sequência de ações que executam determinada tarefa t. As tarefas são completadas decompondo tarefas não primitivas em tarefas menores até só restarem tarefas primitivas. As tarefas primitivas são análogas as ações executáveis em planejamento clássico, e expressam uma atividade que o agente possa executar diretamente no ambiente [13]. Já as tarefa não primitiva representam objetivos que o agente deve alcançar com o intuito de decompor as tarefas não primitivas em primitivas. O exemplo da seção anterior de mover um objeto de lugar é um exemplo de tarefa primitiva, pois altera o estado do ambiente, denotado por (*move* ?*from* ?*to*). Um exemplo de tarefa não primitiva é realizar uma viagem de carro, para conseguir completar essa tarefa é necessário fazer a revisão do carro, arrumar as malas e colocar tudo dentro do carro, que é representado por (*travelByCar* ?*car* ?*stuffs*)).

Para iniciar a geração de um plano HTN, é usado como início uma tarefa de ligação. Onde uma tarefa de ligação HTN é definida como $w = (T, C)$, onde T é um conjunto de tarefas a ser completadas e C é um conjunto ordenado de restrições sobre as tarefas T. As restrições estabelecem a ordem com que as tarefas T devem ser executadas.

Um domínio de planejamento HTN é um par $D = (A, M)$, onde A é um conjunto de predicados, que representam estados no ambiente, e M um conjunto finto de métodos [5]. Um método é utilizado para decompor tarefas não-primitivas em primitivas. Um método é representado por $m = (p, t, w)$, onde p é uma precondição que estabelece o que deve estar presente no estado atual para que a tarefa t consiga ser decomposta por uma tarefa de ligação w [8]. Considerando o exemplo anterior de viajar de carro, a Figura 4.2 ilustra esse método, sendo a tarefa em cinza uma tarefa não primitiva, que posteriormente também será decomposta. O conjunto de tarefas que faz parte da tarefa de ligação está representada pelas sub tarefas, e a ordem caracteriza as restrições.

Um problema de planejamento HTN P é definido como $P = (d, I, D)$, onde D é um domínio, d é a tarefa de ligação inicial e I é um estado inicial como no planejamento clássico. O processo de geração de um plano utilizando planejamento HTN consistem em encontrar um método que consiga ser aplicado na primeira tarefa de d, isso faz com que seja gerado uma tarefa de ligação diferente d', onde a primeira tarefa foi decomposta. Esse processo continua, agora aplicado a d', até que todas as tarefas sejam primitivas [5]. Se em algum ponto, nenhum método consiga ser aplicado, o planejador tem que realizar um retrocesso(*backtracking*), que consiste em voltar a um d anterior a ponto de conseguir tentar outra decomposição [13]. É possível representar a busca do plano por uma árvore N, na

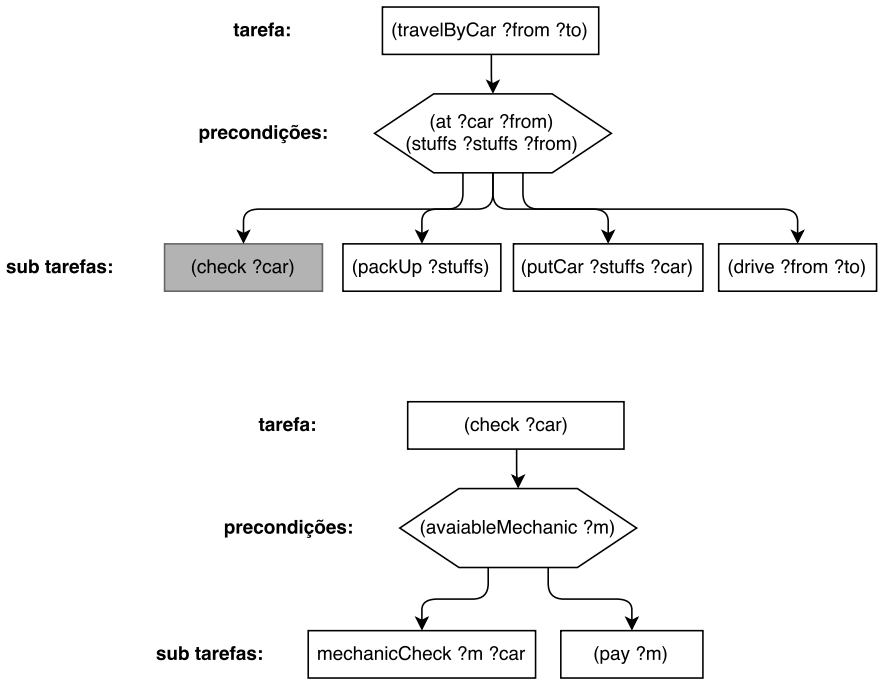

Figura 4.2 – Exemplo de método HTN.

qual os nodos são tarefas ou métodos. Cada tarefa não-primitiva pode ter apenas um filho, que deve ser um método. Cada método deve ter um filho para cada uma das suas sub tarefas. Tarefas primitivas não podem ter filhos, o que significa que elas não podem ser decompostas. Uma árvore totalmente decomposta, é onde todas as folhas de *N* são tarefas primitivas [11]. A Figura 4.3 ilustra a árvore de resolução do exemplo anterior.

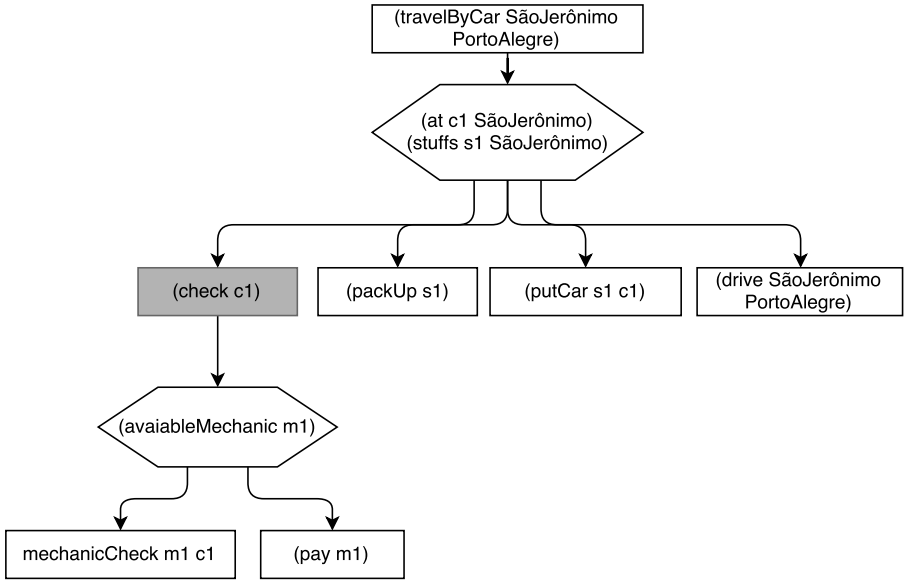

Figura 4.3 – Arvore de resolução HTN.

O algoritmo de *Total-order Forward Decomposition* (TFD) é utilizado para gerar um plano a partir de uma rede de tarefas inicial com ordenação total, como detalhado no

Algoritmo 4.1. O algoritmo gera as ações na mesma ordem que serão executadas, então a cada vez que uma tarefa é alcançada, tudo que antecede a mesma já foi planejado [8].

Algoritmo 4.1 – Total-order Forward Decomposition

```
1: function TFD(s, < t1, ..., tk >, O, M)
2:     if k = 0 then
3:         return <>
4:     end if
5:     if t1 é primitivo then
6:         ativo = {(a, b) | a é uma instancia de O e é aplicável a s e b é uma substituição
       que torna a relevante para b(t1)}
7:         if ativo = ∅ then
8:             return falha
9:         end if
10:        escolhe algum par (a, b) ∈ active
11:        π = TFD(γ(s, a), b(< t2, ..., tk >, O, M)
12:        if π = falha then
13:            return falha
14:        else
15:            return a.π
16:        end if
17:    else if t1 é não primitiva then
18:        ativo = {m| m é aplicável a s e m ∈ M}
19:        if ativo = ∅ then
20:            return falha
21:        end if
22:        escolhe algum par (m, b) ∈ active
23:        w = subtarefas(m).b(< t2, ..., tk >)
24:        return TFD(s, w, O, M)
25:    end if
26: end function
```

4.3 AHTN

Adversarial hierarchical-task network (AHTN) é um algoritmo desenvolvido para lidar com o problema do grande fator de ramificação dos jogos em tempo real [11] utilizando conhecimento de domínio no estilo de planejamento HTN. Nele são combinados técnicas de HTN com o algoritmo *minimax search*. O algoritmo assume jogos totalmente observáveis, baseados em turno e determinísticos.

O Algoritmo 4.2 [11] é a representação da técnica de AHTN, e assume que existem dois jogadores, MAX e MIN, como no algoritmo de *minimax search* apresentado no Capitulo 3. O algoritmo também assume uma busca em uma árvore com uma máxima profundidade d. O intuito do algoritmo de AHTN é gerar os melhores plano para MAX e para

Min, junto com o resultado de uma função de avaliação para quando os planos chegam a um estado terminal.

Algoritmo 4.2 – Adversarial hierarchical-task network

```
1: function AHTNMAX(s, N+, N−, t+, t−, d)
2:     if terminal(s) ∨ d ≤ 0 then
3:         return (N+, N−, e(s))
4:     end if
5:     if nextAction(N+, t+) ≠⊥ then
6:         t = nextAction(N+, t+)
7:         return AHTNMIN((γ(s, t), N+, N−, t, t−, d − 1))
8:     end if
9:     N+* =⊥, N−* =⊥, v* = −∞
10:    ℵ = decompositions+(s, N+, N−, t+, t−)
11:    for all N ∈ ℵ do
12:        (N+', N−', v') = AHTNMax(s, N, N−, t+, t−, d)
13:        if v' > v* then
14:            N+* = N+', N−* = N−', v* = v'
15:        end if
16:    end for
17:    return (N+*, N−*, v*)
18: end function
```

Cada nodo da árvore das jogadas é definido por uma tupla (s, N_+, N_-, t_+, t_-), onde s é o estado corrente do ambiente, N_+ e N_- são a representação de planos HTN para os jogadores MAX e MIN, respectivamente, t_+ e t_- representam ponteiros para qual parte do plano HTN está sendo executada, sendo t_+ para uma tarefa de N_+ e t_- para uma tarefa de N_- [11].

A função *nextAction*(N, t) faz com que, dado uma representação de plano em HTN N e um ponteiro t, seja encontrada a próxima tarefa primitiva que deve ser executada em N após a tarefa t. Se $t = \perp$ então é retornado a primeira tarefa primitiva a ser executada em N. Se existir em N alguma tarefas não primitivas e não existir nenhuma tarefa primitiva em N então *nextAction*$(N, t) = \perp$ [11]. Na linha 5 é testado se existe alguma tarefa primitiva no plano atual, se existir ela é aplicada e passa para o próximo nível da árvore alternando para MIN, através da função *AHTNMin*.

Um nodo MAX $n = (s, N_+, N_-, t_+, t_-)$ é consistente se as ações primitivas que já estão em N_+ e N_- conseguem ser executadas dado um estado s e uma transição γ. A definição de consistência de MIN é análoga [11].

Para um nodo MAX $n = (s, N_+, N_-, t_+, t_-)$, *decompositions*$_+(s, N_+, N_-, t_+, t_-)$ denota o conjunto das decomposições validas que adicionem apenas um novo método em N_+ $(decompositions(N_+, t, m) | m \in applicable(N_+, t))$. A definição de *decompositions*$_-$ é análoga [11]. Na linha 10 é adicionado a $\aleph$ todas as possibilidades de continuação de planos a partir do ponto atual.

Uma função de avaliação e, quando aplicada sobre um estado $s \in S$, retorna a recompensa de MAX em s se ele for um estado terminal ou uma aproximação se s for um estado não-terminal [11]. Na linha 11 são comparados todos os possíveis planos pela função de avaliação e é retornado o melhor plano encontrado para os dois jogadores, junto com o resultado da função de avaliação. Quando um estado terminal ou a profundidade for atingida os planos e a função de avaliação para o estado atual são retornados. É o que representa a Linha 2

A grande diferença entre o algoritmo de *AHTN* e o algoritmo do *minimax search*, é que as chamadas recursivas nem sempre se alternam entre MAX e MIN. O algoritmo troca de nodos MAX para MIN apenas quando os planos estão totalmente decompostos a ponto de gerar uma ação (Linha 7) [11].

Para exemplificar o funcionamento do Algoritmo 4.2, é apresentado a Figura 4.4, que ilustra uma árvore gerada com profundidade $d = 2$. Na raiz da árvore pode ser visto que, para os dois jogadores, apenas uma tarefa não primitiva *win* precisa ser decomposta. Há duas decomposições que o jogador MAX pode aplicar para seu plano HTN, resultando nos nodos n_1 e n_5. A decomposição de n1 não resulta em nenhuma ação primitiva, e por isso n_1 continua um nodo MAX. Uma vez que o jogador MAX decompõe sua HTN para o ponto onde a primeira ação pode ser gerada(nodo n_2 e n_5), é o turno de MIN decompor suas tarefas não primitivas. Quando MIN pode gerar suas ações, a profundidade máxima foi atingida (nodos n_3 e n_4). A função de avaliação e é aplicada para cada um dos estados do jogo para determina o valor das folhas.

Matheus: FINALIZAR PLANEJAMENTO E INTRODUZIR ML

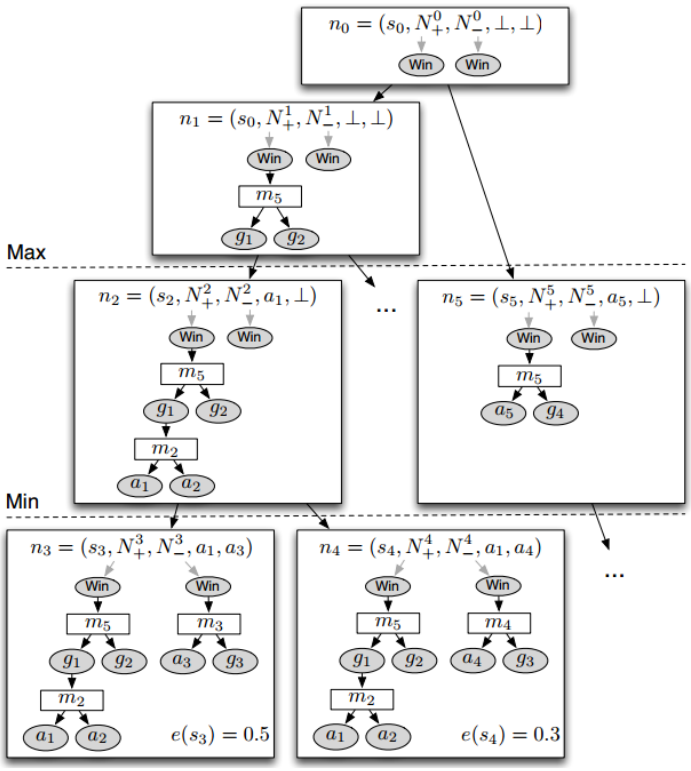

Figura 4.4 – Árvore gerada pelo algoritmo de AHTN

5. APRENDIZADO

Para os humanos o aprendizado ocorre durante toda a vida. O aprendizado é o ato de adquirir novos conhecimentos, ou modificar conhecimentos já existentes ou ainda adquirir uma experiência por repetição do ato de forma incorreta. Aprendizado pode variar de adquirir conhecimento de tarefas simples, como decorando um número de telefone, até tarefas mais complicadas, como a formulação de novas teorias [13].

5.1 Aprendizado de Máquina

A área na computação que estudo esse aprendizado de forma computacional é o aprendizado de máquina, melhor conhecida como *machine learning*. A definição de aprendizado de máquina proposta por Tom Mitchell [7] é a seguinte:

> Definição: Um programa de computador é dito que aprende de uma experiência E com relação a alguma classe de tarefas T, e medida de performance P, se essa performance sobre as tarefas em T, medida por P, melhora com a experiência E.

Essa definição mostra que o sistema aprimora seu conjunto de tarefas T com uma performance P através de experiências E. Ou seja, um sistema baseado em aprendizado de máquina deve, através de experiências, ter um ganho nas informações para solucionar os seus problemas. Para começar a resolver um problema utilizando aprendizado de máquina é preciso escolher qual experiência será aprendida pelo sistema [7]. Para isso existem algumas técnicas que tratam aprendizado de máquina com objetivos diferentes [13]. Alguma das técnicas são:

- aprendizado supervisionado: que consiste em aprender através de algum conjunto de exemplos a realizar a classificação de algum problema. Cada problema é mapeado para uma saída;
- aprendizado não supervisionado: que consistem em aprender através das observações, algum padrão ou regularidade, para classificar em grupos os problemas; e
- aprendizado por reforço: que consiste em aprender, através das execuções de um agente, quais ações possuem maior recompensa média esperada.

Cada tipo de aprendizado é utilizado pode ser usado para uma aplicação especifica, mas existem casos em que a combinação das técnicas se mostra mais eficaz. Um exemplo apresentado por [13] é o reconhecimento de idade por fotos, para essa tarefa são

necessários amostras de fotos com as idades, então a técnica que se encaixa é aprendizado supervisionado, mas existem ruídos aleatórios nas imagens que fazem com que a precisão da abordagem caia, para superar esse problema pode ser combinado aprendizado supervisionado com o não supervisionado.

5.2 Aprendizado por Reforço

O aprendizado por reforço também é conhecido como *reinforcement learning*. Este tipo de aprendizado utiliza *feedbacks*, vindas do ambiente após a sua execução, esse tipo de *feedback*, é chamado de recompensa. O objetivo deste aprendizado é usar as recompensas obtidas nas observações para aprender uma política do ambiente ou determinar o quão boa a política é [13].

Em jogos *reinforcement learning* é um tópico que é bastante utilizado [6]. Em um jogo essa técnica utiliza três etapas, uma para exploração da estratégia para achar as diferentes ações possíveis no jogo, uma função que disponibiliza o *feedback* e diz o quão bom é cada ação, e uma regra de aprendizado que junta as outras duas etapas [6].

Existem dois tipos principais de aprendizado por reforço [13]: aprendizado passivo, e aprendizado ativo; detalhados nas seções a seguir.

5.2.1 Aprendizado passivo

O aprendizado passivo utiliza ambientes completamente observáveis. A política do agente π é fixa, em um estado s, sempre é executado a mesma ação $\pi(s)$. O objetivo desse tipo de aprendizado é aprender o quão bom é a política, o que significa aprender a função de utilidade $U^{\pi}(s)$. Um agente que utiliza aprendizado passivo não conhece o modelo de transição $P(s'|s,a)$, que especifica a probabilidade de alcançar o estado s' a partir do estado s executando a ação a, e também não conhece a função de recompensa $R(s)$, que especifica a recompensa de cada estado [13].

Um agente que utiliza essa técnica realiza várias execuções do ambiente utilizando a politica π. Em cada tentativa o agente inicia no mesmo estado inicial e realiza uma sequência de transições de estados até chegar a um estado terminal. As percepções obtidas com essas execuções, em cada estado, servem para descobrir a recompensa obtida nos estados. O objetivo é utilizar a informação das recompensas para aprender a utilidade esperada $U^{\pi}(s)$ associada a cada estado s não terminal [13].

5.2.2 Aprendizado ativo

O aprendizado ativo diferente do passivo não tem uma política fixa e a mesma deve ser aprendida. Para isso, um agente que utiliza este tipo de aprendizado precisa decidir quais ações tomar, isso faz com que o agente precise aprender o modelo de transição $P(s'|s,a)$ para cada um dos estados e ações [13].

Um método para conseguir definir a política do ambiente é chamado de *Q-learning*. O objetivo desse método é aprender uma utilidade ligada a um par de estado e ação, a notação $Q(s,a)$, representa o valor de executar a ação a no estado s. Este método está relacionado com o valor de utilidade presente na Equação 5.1 [13].

$$U(s) = max_a Q(s,a) \tag{5.1}$$

O algoritmo de *Q-learning* não precisa aprender o modelo de transição $P(s'|s,a)$, por esse motivo ele é chamado de um método livre de modelo. A Equação 5.2 apresenta como é feito o cálculo do valor de $Q(s,a)$.

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma max_{a'} Q(s',a') - Q(s,a)) \tag{5.2}$$

O valor α representa a taxa de aprendizado do agente, variando de 0 a 1, nele é contido a informação se o agente deve considerar as informações obtidas em um novo aprendizado ou não, sendo 1 se considera inteiramente o que foi aprendido, e 0 se for descartar as novas informações. $R(s)$ é a recompensa obtida ao alcançar o estado s. O valor de γ representa o fator de desconto, que descreve a preferência do agente em receber recompensas futuras ou recompensas locais, o seu valor varia entre 0 e 1, quando 0 apenas as recompensas locais são utilizadas, e quando 1 as recompensas futuras são totalmente utilizadas. O Algoritmo 5.1 ilustra o método de *Q-learning* para um agente [13]. A Linha 7 é onde é atualizado o valor de $Q(s,a)$ utilizando a Formula 5.2. O algoritmo leva em conta quantas vezes quantas vezes o valor $Q(s,a)$ foi calculado, incrementando a cada nova passada do algoritmo pela Linha 6. O algoritmo indica a melhor ação para o estado atual.

Um algoritmo que tem grande relação com o *Q-learning* é o algoritmo chamado *State-Action-Reward-State-Action* (SARSA), onde as equações de atualização são bem parecidas. A Equação 5.3 apresenta como é calculado o valor de utilidade no SARSA. A grande diferença entre os dois métodos é que, enquanto o *Q-learning* busca o melhor valor de utilidade do estado na transição observada, o SARSA espera até uma ação ser realmente tomada para calcular o valor para aquela ação. A grande diferença entre os dois métodos é que, enquanto o *Q-learning* não leva em consideração a politica atual, o SARSA utiliza essa informação para saber o que realmente irá acontecer [13].

Algoritmo 5.1 – Q-Learning

```
1: function Q-LEARNING(state, reward)
2:     if terminal(state) then
3:         return Q[s, None] = reward
4:     end if
5:     if state is not null then
6:         increment N[s, a]
7:         Q[s, a] = Q[s, a] + α(N[s, a])(r + γ max_a' Q[s', a'] − Q[s, a])
8:         s = s'
9:         a = argmax_a' f(Q[s', a'], N[s', a'])
10:        r = r'
11:    end if
12:    return a
13: end function
```

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma\ Q(s',a') - Q(s,a)) \tag{5.3}$$

As duas técnicas consistem em percorrer os estados diversas vezes, com o intuito de aprender o valor de utilidade de cada par estado e ação. Entretanto, as técnicas não têm estimativa para pares que não tenha sido visitado. Em caso de ambientes parcialmente observáveis ou não observáveis podem existir estados que não consigam ser alcançados nas observações, e com isso não consigam ser avaliados [7].

5.2.3 Generalização

No aprendizado ativo, o conhecimento sobre as utilidades é feito a partir de um par de estado e ação, e a maneira como é armazenado essa informação é feita através de uma tabela. A maioria dos problemas terá uma grande quantidade de estados, isso faz com que toda a informação tenha que ser calculada e guardada em uma tabela que pode ser muito grande. Com o objetivo de tratar o problema do grande espaço de estados é possível utilizar generalização. A generalização permite compactar o jeito como as informações são armazenadas, e ainda transferir conhecimento entre estados e ações similares [7, 4].

Uma maneira de utilizar generalização é através de uma função de aproximação, que é uma alternativa para a tabela de pesquisa. Essa técnica é vista como uma aproximação, pois não há garantia que ela irá representar a verdadeira função de utilidade. Um algoritmo de aprendizado por reforço pode aprender valores dos parâmetros θ, e gerar uma função U_θ que aproxima a verdadeira função de utilidade. Essa técnica consegue reduzir, para o xadrez, de 10^{40} valores na tabela, para 20 parâmetros aproximados. A compressão alcançada através da função de aproximação permite que o agente consiga generalizar o aprendizado de estados que foram visitados para os que não foram visitados. O problema

desse tipo método é o tempo que leva para convergir a uma função de aproximação que represente o modelo de forma satisfatória [13].

Em aprendizado por reforço a atualização da função é feita a cada nova observação, pelo fato de que a função pode obter valores que precisam ser ajustados. O ajuste é feito por uma função de erro que altera o parâmetro θ, através do cálculo do gradiente. Isso faz com que a função de avaliação permita o aprendizado por reforço a generalizar por meio de suas observações. A Equação 5.4 mostra como fica o cálculo de θ para o *Q-learning* [13].

$$\theta_i = \theta_i + \alpha[R(s) + \gamma\ Q_\theta(s', a') - Q_\theta(s, a)]\frac{\partial Q_\theta(s, a)}{\partial \theta_i} \tag{5.4}$$

As técnicas de aprendizado de máquina têm grande potencial na área de jogos. O aprendizado de máquina consegue que os agentes reproduzam jogadores mais interessante, pelo fato de que os agentes aprendem sobre o ambiente e usam essa informação a seu favor em jogadas futuras. Os agentes aprendem táticas de jogos com suas derrotas e as aperfeiçoam com suas vitorias. Utilizar técnicas de aprendizado de máquina exige um cuidado e um entendimento das necessidades do jogo [6]. O intuito de adicionar um algoritmo de aprendizado de máquina ao trabalho, vem do fato de que com as informações, provenientes das observações, é possível acrescentar um conhecimento extra nas próximas execuções, a fim de não cometer os mesmos erros de observações anteriores.

6. OBJETIVOS

Atualmente, há uma dificuldade na utilização de técnicas de IA para jogos em tempo real. Isso ocorre pelo fato que a IA precisa ser capaz de resolver tarefas, do mundo real, de maneira rápida e satisfatória. Geralmente, o espaço de estados dos jogos é enorme, isso faz com que não se tenha tempo de explorar todas as possibilidades de solução [6].

6.1 Objetivo Geral

O objetivo geral deste trabalho é implementar o algoritmo de planejamento AHTN [11] no jogo MicroRTS. O algoritmo de AHTN combina técnicas de HTN com o algoritmo de *minimax serach*. Após implementado, integrar aprendizado por reforço com o objetivo de melhorar o desempenho do algoritmo. Ao final do trabalho, o objetivo é conseguir comparar resultados obtidos com os resultados [9, 3, 12].

6.2 Objetivos Específicos

Os Objetivos são classificados em dois grupos, fundamentais e desejáveis. Os objetivos fundamentais representam a base do projeto, já os objetivos desejáveis, representam objetivos que devem ser realizados para uma melhor demonstração das técnicas apresentadas, e ainda para tentar melhorar os resultados obtidos nos objetivos fundamentais.

6.2.1 Objetivos fundamentais

- Implementar o algoritmo de AHTN no ambiente do jogo MicroRTS.
- Comparar os resultados obtidos com as implementados já embutidas no MicroRTS.
- Enviar a implementação para o autor do MicroRTS como uma sugestão de IA para o jogo.

6.2.2 Objetivos desejáveis

- Integrar um algoritmo de aprendizado por reforça ao algoritmo de AHTN.

- Comprar os resultados obtidos com a implementação do AHTN.
- Escrever um artigo para um *workshop* mostrando a unificação das técnicas de planejamento com aprendizado por reforço.

7. ANÁLISE E PROJETO

7.1 Jogos

Jogos eletrônicos são muito populares, principalmente pela grande quantidade de gêneros, existem jogos de ação, aventura, esportes, estratégia, entre outros. Hoje em dia, os jogos buscam que quem jogue consiga ficar imerso no dentro do jogo, sem conseguir identificar um padrão nos jogadores fictícios, pois se não o jogo deixa de ser tão interessante. Para que isso aconteça, a IA é associada a diversos jogos, e é comum pensar que quanto mais complexa a IA aplicada dentro do jogo mais difícil jogo irá ficar, mas isso nem sempre é verdade, nem sempre IA complicadas terão melhor desempenho do que as mais simples, uma boa IA dentro do jogo é feita a partir de determinar o comportamento certo para os algoritmo certos [6].

7.1.1 Jogos de estratégia em tempo real

Jogos de estratégia em tempo real, também conhecido por *real-time strategy games* (RTS), é um subgênero de jogos de estratégia, onde os jogadores precisam construir uma base com uma economia, ganhando recursos, construindo edificações, treinando unidades de ataques e tecnologias para elas, tudo isso com o objetivo de destruir uma ou mais bases inimigas [12, 1].

Existem algumas diferenças entre jogos RTS e jogos de tabuleiro, como xadrez. Estas diferenças são [12]:

- movimentos simultâneos, jogadores realizam jogadas ao mesmo tempo;
- tempo real, cada jogador deve realizar suas ações em um curto espaço de tempo;
- parcialmente observável, na maioria dos jogos RTS, o jogador só consegue enxergar parte do ambiente;
- não-determinístico, nem sempre uma ação realizada resulta na saída esperada; e
- complexidade, o espaço de estados e o número de ações possíveis é muito grande.

Pelo fato de existirem essas diferenças, não é possível traduzir automaticamente as técnicas padrões dos jogos de tabuleiro para jogos RTS sem algum tipo de abstração ou simplificação [12].

7.1.2 MicroRTS

Um exemplo deste gênero é o MicroRTS[1], uma simplificação do jogo Starcraft[2], feita por Santiago Ontañón [10] em Java. O MicroRTS foi desenvolvido para fins acadêmicos, com o intuito de aplicar e desenvolver técnicas de IA e para servir como prova de conceito para as técnicas criadas.

O MicroRTS consiste em dois jogadores tentando destruir a base adversaria. Para destruir com o inimigo é preciso eliminar cada unidade e edificações adversarias. A Figura 7.1 mostra uma tela do jogo. Existem quatro tipos de unidades no jogo, são elas:

- *worker*, é responsável por coletar recursos e construir as edificações. Esta unidade também consegue lutar, mas possui um dano muito baixo;
- *heavy*, unidade que pode apenas atacar. Ela possui um alto poder de ataque, mas sua velocidade é lenta;
- *light*, unidade que pode apenas atacar. Ela possui um baixo poder de ataque, mas sua velocidade é rápida; e
- *ranged*, unidade que pode apenas atacar. Ela possui um ataque de longa distância.

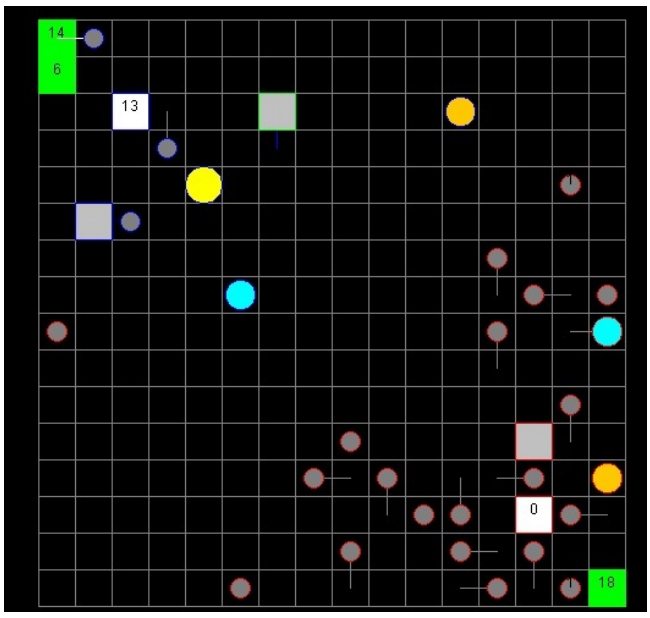

Figura 7.1 – Um exemplo de tela do MicroRTS

Para adquirir as unidades é preciso das edificações e recursos. Existem três tipos de edificações, são elas:

[1] https://github.com/santiontanon/microrts
[2] http://us.battle.net/sc2/pt/

- base, a base é a edificação principal, ela é responsável pela criação dos *workers*, e nela também é guardado os recursos coletados pelos *workers*;
- quartel, o quartel é responsável pela criação das unidades de ataque *heavy*, *light* e *ranged*. Ela pode ser construída pelos *workers* usando recursos; e
- base de recurso, na base de recurso são coletados os recursos pelos *workers*, os a base de recursos é finitos para ser coletada.

No início do jogo, cada jogador inicia com uma base, um *worker* e uma base de recurso para coletar. Todas as unidades podem ser atacas menos a base de recursos.

No ambiente há algumas estratégias implementadas, cada estratégia possui variações dos algoritmos. Algumas das estratégias são:

- *Minimax Alpha-Beta Search Strategies* - O que muda entre as técnicas é o jeito com que é feito a expansão do grafo.
- *Monte Carlo Search Strategies* - Executa jogadas aleatórias para planejar e após utiliza uma heurística para determinar em qual caminho seguir.

A plataforma já foi utilizada para aplicar técnica de IA. Por esse motivo a utilização dela se torna viável para a realização deste trabalho. Nela é possível observar que o fator de ramificação pode ser muito alto dependendo do cenário do jogo.

7.1.3 Arquitetura do MicroRTS

A arquitetura do MicroRTS é composta por 4 componentes principais: O Jogo em si, onde são feitas todas as ações dos jogos. As unidades, onde todas as ações de cada unidade podem ser controladas e acessadas, por exemplo, saber onde está cada unidade inimiga. A Interface gráfica, responsável pela representação gráfica do jogo. E a Inteligência Artificial, onde pode ser acoplado a IA que desejar, a plataforma já possui algumas como foi dito anteriormente. A imagem 7.2 representa os componentes.

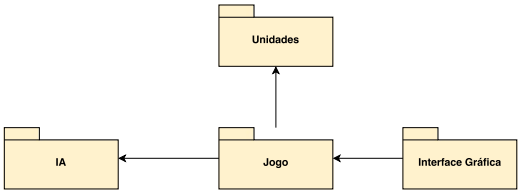

Figura 7.2 – Arquitetura MicroRTS

7.2 Descrição do Projeto

Matheus: REESTRUTURAR ESSA PARTE E ORGANIZAR MELHOR AS IDEIAS

Para realizar a implementação de uma IA para acoplar no MicroRTS, é preciso conhecer as classes responsáveis por cada componente. A Figura 7.3 ilustra as principais classes presentes no jogo. A Classe *GameVisualSimulation* é a interface entre os componentes do jogo e o usuário. A classe *GameState* e *PhysicalGameState*, são responsáveis pelo controle das ações das unidades, e pelo controle do mapa e das unidades dentro dele, respectivamente. A classe *UnitTypeTable* é onde cada unidade é associada as ações possíveis no jogo. A Classe *PhysicalGameStatePanel* é responsável pela interface gráfica. E por fim a classe *AHTN* é onde minha proposta de solução será acoplada ao jogo.

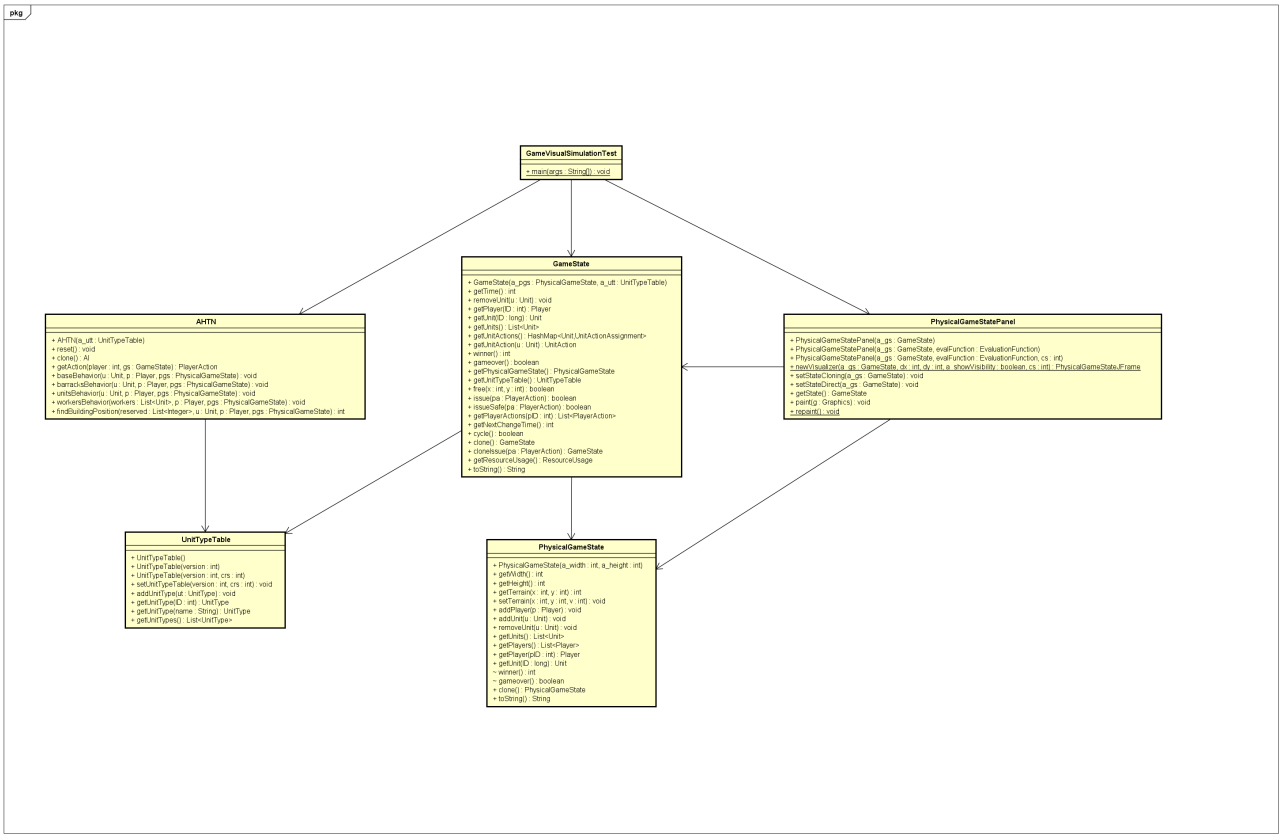

Figura 7.3 – Classes do MicroRTS

A Figura 7.4 apresenta o diagrama de sequência, para entender o funcionamento das classes citadas acima. Os passos de 1 a 6 são apenas para criação dos objetos que serão utilizados pelo jogo. No passo 7 é gerado a ação que deve ser executado no momento atual do jogo, já no passo 8 é confirmado se foi encontrado alguma ação que possa ser executada. No passo 9 a tela é desenha novamente com a atualização das jogadas. Existe um laço do passo 7 até o 9, pois são onde são geradas as ações, esse laço acaba quando termina o tempo limite de jogo, ou algum jogador vence. O diagrama mostra a geração da

jogada para apenas um jogador, mas a forma como as jogadas são geradas é da mesma forma como a apresentada. O algoritmo de AHTN irá ser implementado dentro do método *getAction*, e, portanto, estará presente no passo 7 do diagrama.

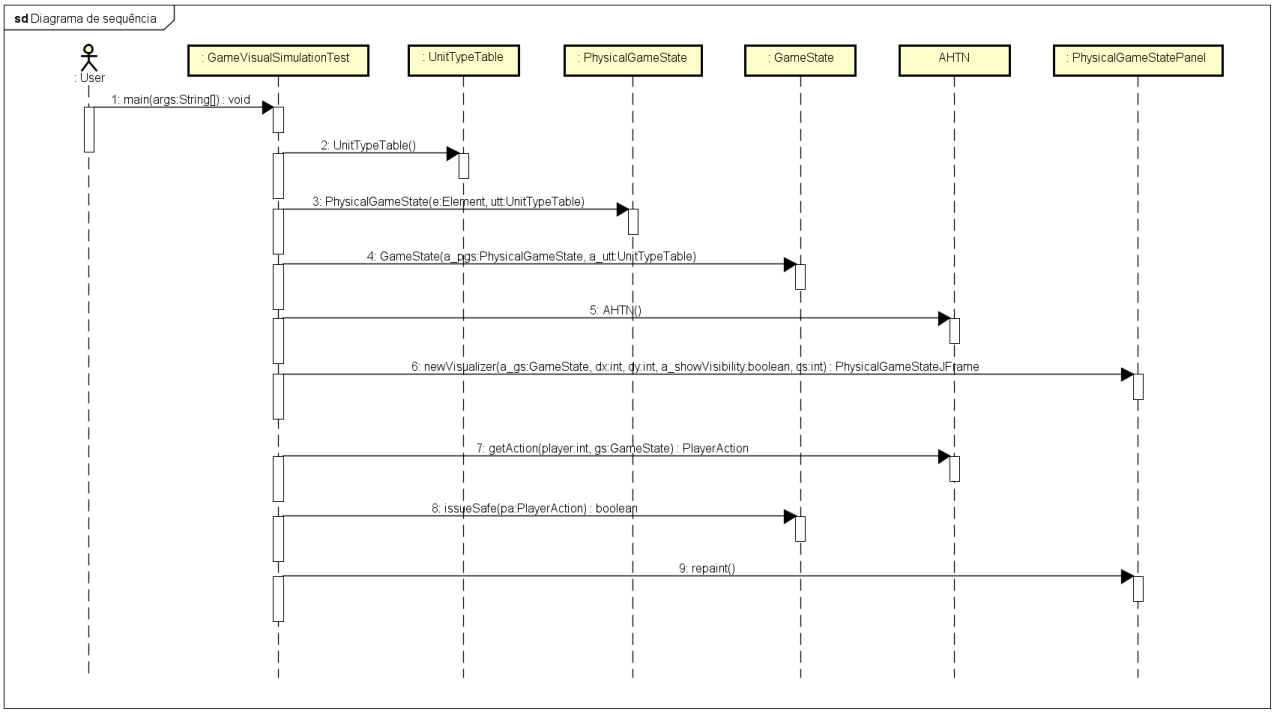

Figura 7.4 – Diagrama de sequência

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Buro, M.; Churchill, D. “Real-time strategy game competitions”, *AI Magazine*, 2012.

[2] Helmert, M.; Haslum, P.; Hoffmann, J.; et al.. “Flexible abstraction heuristics for optimal sequential planning”. In: ICAPS, 2007.

[3] Hogg, C.; Kuter, U.; Munoz-Avila, H. “Learning methods to generate good plans: Integrating htn learning and reinforcement learning.” In: AAAI, 2010.

[4] Kaelbling, L. P.; Littman, M. L.; Moore, A. W. “Reinforcement learning: A survey”, *Journal of artificial intelligence research*, 1996.

[5] Meneguzzi, F.; De Silva, L. “Planning in bdi agents: a survey of the integration of planning algorithms and agent reasoning”, *The Knowledge Engineering Review*, 2015.

[6] Millington, I.; Funge, J. “Artificial Intelligence for Games”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, 2nd ed..

[7] Mitchell, T. M. “Machine Learning”. New York, NY, USA: McGraw-Hill, Inc., 1997, 1st ed..

[8] Nau, D.; Ghallab, M.; Traverso, P. “Automated Planning: Theory & Practice”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[9] Ontanon, S. “Experiments with game tree search in real-time strategy games”, *arXiv*, 2012, 1208.1940.

[10] Ontanón, S. “The combinatorial multi-armed bandit problem and its application to real-time strategy games”. In: Ninth Artificial Intelligence and Interactive Digital Entertainment Conference, 2013.

[11] Ontañón, S.; Buro, M. “Adversarial hierarchical-task network planning for complex real-time games”. In: Proceedings of the 24th International Conference on Artificial Intelligence, 2015, pp. 1652–1658.

[12] Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; Preuss, M. “A survey of real-time strategy game ai research and competition in starcraft”, *Computational Intelligence and AI in Games, IEEE Transactions on*, 2013.

[13] Russell, S.; Norvig, P. “Artificial Intelligence: A Modern Approach”. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009, 3rd ed..

[14] Shoham, Y. “Agent-oriented programming”, *Artif. Intell.*, 1993.

[15] Wooldridge, M. “Intelligent Agents”. The MIT Press, 1999.