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

# ADVERSARIAL HIERARCHICAL-TASK NETWORK PARA JOGOS EM TEMPO REAL

**MATHEUS DE SOUZA REDECKER**

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

Orientador: Prof. Felipe Rech Meneguzzi

**Porto Alegre**
**2016**

## AGRADECIMENTOS

Inicialmente agradeço ao meu orientador prof. Felipe Rech Meneguzzi pela oportunidade de realizar este trabalho, sob sua orientação. Agradeço aos meus pais, Ana e Hélio, pelo suporte oferecido ao longo da vida acadêmica. E como não poderia deixar de mencionar, minha companheira, Carolina, pelo apoio, carinho, e pela compreensão em relação ao meu esforço na elaboração deste trabalho.

# ADVERSARIAL HIERARCHICAL-TASK NETWORK 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*. Propomos a utilização deste algoritmo em um jogo de estrategia em tempo real.

**Palavras-Chave:** planejamento automatizado, planejamento hierarquico, busca adversária.

# ADVERSARIAL HIERARCHICAL-TASK NETWORK 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. We propose the utilization of this algorithm in one real-time strategy game.



**Keywords:** automated planning, hierarchical planning, adversarial search.

# LISTA DE FIGURAS

# LISTA DE ALGORITMOS

# 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

A Inteligência artificial (IA) possui aplicação em jogos, fazendo com que os computadores sejam capazes de jogar jogos como xadrez e jogo da velha sem intervenção humana. Os Jogos de computador muitas vezes não utilizam algoritmos para simular o comportamento dos jogadores, utilizando ao invés disso, técnicas que passam a ilusão de que as decisões estão sendo realizadas de forma autônomas, ou ainda, aproveitam das informações provenientes do controle do jogo para tomar as suas decisões. Esse tipo de técnicas não pode ser caracterizado como uma técnica de IA totalmente autônoma [5].

Jogos que utilizam técnicas de IA conseguem prover uma melhor interação entre o jogador e o jogo, e assim prendendo a atenção do jogador. 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 serem previsíveis [5]. Nos jogos de computador as reações das jogadas 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 é preciso algoritmos eficientes para gerar uma ação de maneira rápida [5]. Em alguns casos, são gerados ações sub ótimas para que o tempo de resposta não seja muito alto [11, Capítulo 3].

Os jogos de estratégia em tempo real são jogos onde os jogadores devem construir uma base, coletar recursos e traçar batalhas que tem com o objetivo derrotar os seus oponentes. Jogos muito conhecidos como *StarCraft*[1] e *Age of Empires*[2] são exemplos de jogos desse gênero [10]. Uma maneira de decidir quais ações precisam ser tomadas, é utilizando técnicas de busca. As técnicas de busca almejam definir qual a ação deve ser escolhida através de uma busca pelas possibilidades disponíveis para o jogo. O problema dessas técnicas é que elas devem explorar pelo menos uma parte do espaço de estados, e com isso o tempo que é preciso para gerar uma ação é alto [7]. Outra maneira de escolher uma jogada é pensando em como conquistar um objetivo por vez, por exemplo, para ganhar do adversário é preciso criar tropas ofensivas, mas antes de ter tropas é preciso ter um quartel para as treinar. Então, planejar as ações pensando em algum objetivo é uma maneira de determinar qual ação deve ser escolhida no momento. Técnicas de planejamento podem ser utilizadas para gerar planos para alcançar esses objetivos.

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)* [9]. Neste algoritmo são combinadas técnicas de

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

planejamento hierárquico com as de busca adversária. O intuito deste trabalho é explorar eficientemente o espaço de ações disponíveis usando conhecimento de domínio, a fim de definir qual a próxima ação que deve ser executada em um jogo. A plataforma escolhida foi o MicroRTS[3], que é um jogo de estratégia em tempo real.

Este documento constitui a parte final do trabalho de conclusão de curso, relatando o desenvolvimento do que foi proposto no Trabalho de Conclusão I. O documento está 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 automatizado e planejamento hierárquico, junto com a explicação do algoritmo de AHTN. O Capítulo 5 apresenta os recursos necessários para a realização deste trabalho. No Capítulo 6 é apresentado como foi feita a implementação do trabalho. O Capítulo 7 apresenta a avaliação dos resultados. Finalmente, no Capítulo 8 é feita a conclusão do trabalho, junto com os trabalhos futuros.

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

# 2. AGENTES

Os agentes são utilizados em jogos como uma abstração que represente os jogadores, eles conseguem absorver informações providas do jogo e assim decidir qual o próximo passo a ser tomado [5]. Formalmente, agentes são entidades que agem de forma continua e autônoma em um ambiente, sendo capazes de receber estímulos do ambiente através de sensores, e assim responder aos estímulos por intermédio de atuadores [12]. Para os agentes os estímulos do ambiente são recebidos como percepções. Os atuadores por sua vez, geram uma ação considerando as percepções [11, Capítulo 7]. A interação de um agente com o ambiente pode ser ilustrado pela Figura 2.1.

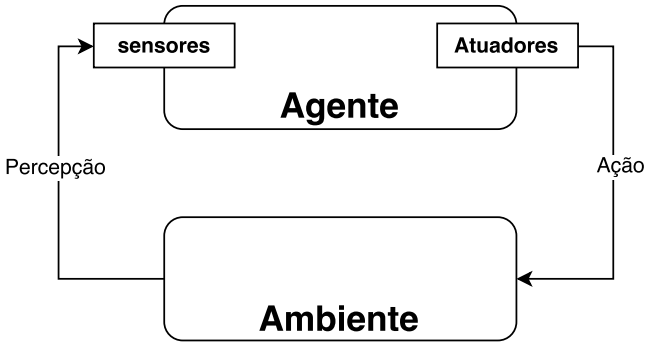


Figura 2.1 – Representação de um agente

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 [13]. 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, ele 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 [11, Capítulo 7].

## 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 [11, Capítulo 2].

Nem sempre todas as informações do ambiente estarão disponíveis, por esse motivo o ambiente pode ser completamente observável, parcialmente observável ou não observável, dependendo da informação disponibilizada. Um ambiente é 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 parcialmente observável. Agora, se o ambiente não disponibiliza nenhuma informação, o ambiente é não observável [11, Capítulo 2] [13].

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 [11, Capítulo 2].

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 [13].

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. O ambiente é cooperativo quando os agentes cooperam entre si [11, Capítulo 2].

## 2.2 Arquiteturas de Agentes

O tipo mais simples de agente é aquele que apenas reage a uma percepção vinda do ambiente. Ele 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.

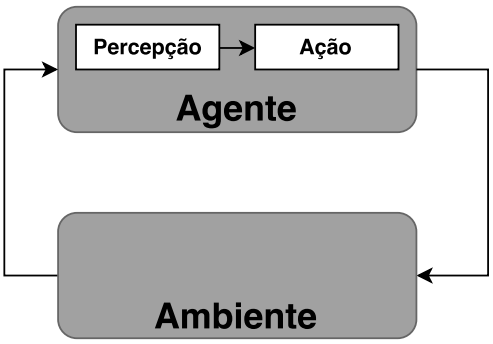


Figura 2.2 – Arquitetura simples de um agente.

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 [11, Capítulo 7]. 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 cidade ao norte da 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 [11, Capítulo 7]. 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 [11, Capítulo 7]. No exemplo de conhecer as cidades, se guardar as visitas antigas, pode ajudar a não visitar novamente cidades que já foram 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 alcançar tal objetivo com uma melhor performance pode ser utilizado uma função de utilidade, nesta função o "desejo"de tomar determinada ação é

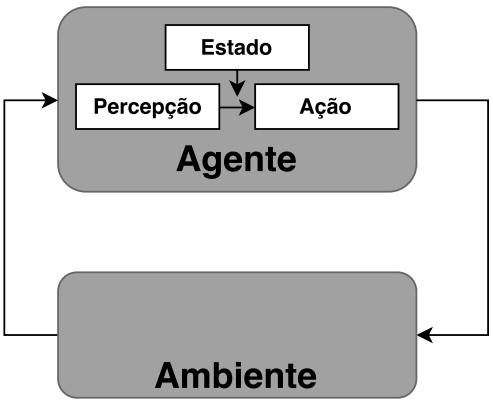


Figura 2.3 – Arquitetura de agentes com estados.

medido. Cada ação exercida pelo agente terá influência no valor de utilidade obtido [11, Capítulo 7]. Seguindo no exemplo, o objetivo pode ser visitar todas as cidades e a função de utilidade pode medir a distância entre elas, assim podendo alcançar o objetivo com a menor distância percorrida.

# 3. BUSCA

Técnicas de busca visam encontrar sequências de ações para um agente alcançar um objetivo. Para utilizar as técnicas de busca é preciso formalizar o problema a ser resolvido e o objetivo a ser alcançado. Para isso, é preciso ter um problema e como resolução é retornado um conjunto de ações. Um problema pode ser definido por cinco componentes [11, Capítulo 3]:

- $s_0$, 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*($s$), uma função que verifica se o estado $s$ satisfaz o objetivo do agente e;
- Custo do caminho, uma função que define um valor numérico para cada ação realizada em um estado. O custo é definido dependendo do ambiente, pois o custo pode ser tanto a distância entre duas localidades, ou o custo monetário de se locomover.

Tais elementos são necessários como entrada para as técnicas de busca. 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 estratégias de busca 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 [11, Capítulo 3].

Considere o mapa apresentado na Figura 3.1 para exemplificar um problema de busca. Cada círculo representa uma cidade, e as linhas entre as cidades são estradas que as ligam umas às outras. A ação possível neste problema 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.

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 para iniciar, mas a técnica só tem como saber após realizar todas as possíveis opções de caminhos, ou se utilizar junto a técnica uma

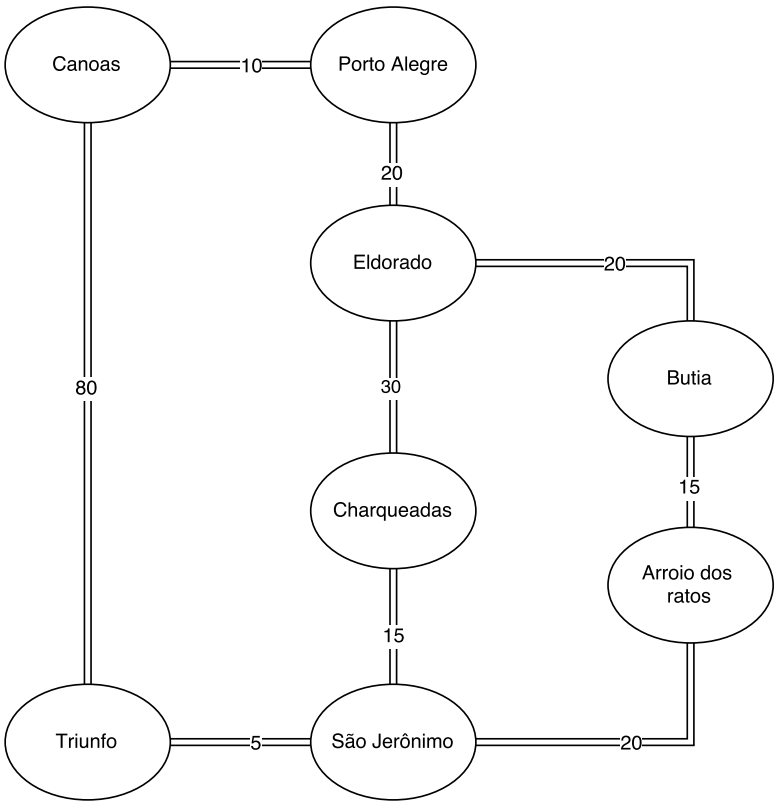


Figura 3.1 – Mapa para o exemplo de problema de busca

função de heurística, levando em consideração os custos dos caminhos, que acrescentam um conhecimento extra para a resolução do problema [11, Capítulo 3].

## 3.1 Busca adversária

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 pode ser utilizada nos jogos que estão situados em ambientes competitivos e de multi agentes. Em um jogo, o jogador não informa suas jogadas previamente, mas está sempre buscando vencer o jogo, por essa razão é possível prever as suas ações. No xadrez, os jogadores alternam jogadas até que um jogador ganha, e o outro perde. Uma maneira de representar uma vitória é atribuindo um valor de utilidade positivo para aquela configuração do jogo, e consequentemente um valor de utilidade negativo para uma derrota. Isto faz com que no final do jogo, quando os valores de utilidade dos agentes forem somados, a sua soma seja sempre zero. Com o intuito de resolver esse problema, é possível gerar uma solução de contingência para tentar antecipar as jogadas do adversário [11, Capítulo 5].

As técnicas de busca adversária utilizam uma variação da definição de um problema de busca comum. Os componentes sofrem algumas mudanças para 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 $s$. Também é chamado de função de avaliação.

Com estes 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. Cada nodo da árvore representa um estado do jogo. Os nodos tem uma ligação para cada ação possível. As ligações representam as ações, o nodo resultante da ligação é o novo estado do jogo após a execução da ação. Um nodo folha é alcançado quando há uma configuração de fim de jogo. 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 *minimax search* é um algoritmo de busca adversária, seu objetivo é retornar a melhor jogada para o estado atual. Este método considera dois agentes, chamados de MAX e MIN, onde o jogador MAX representa a perspectiva do agente que está tentando maximizar a recompensa de suas ações em relação ao agente MIN, que representa o agente adversário do jogador MAX. O algoritmo alterna entre jogadas de MAX, e MIN [11, Capítulo 5].

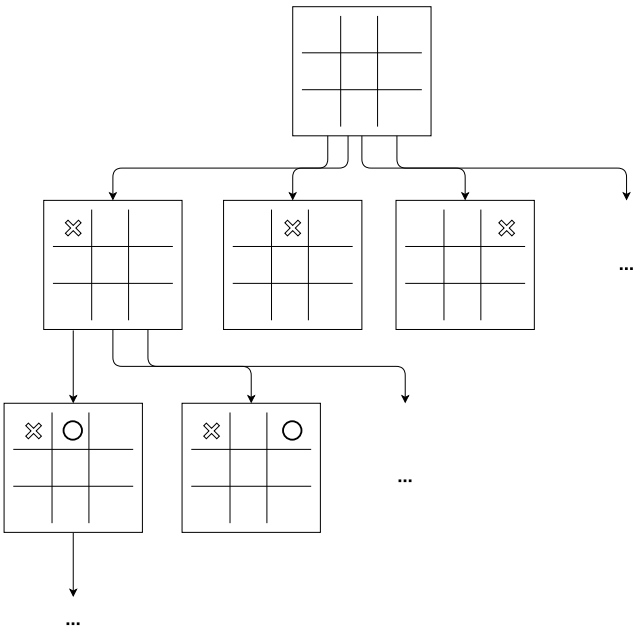

Figura 3.2 – Exemplo de um pedaço de uma *game tree* sobre o jogo da velha

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 representam o final do jogo, contém um valor de utilidade, obtido pela função de avaliação. 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 [11, Capítulo 5]. A Figura 3.3 ilustra uma árvore de jogo, que simula o comportamento do algoritmo de *minimax search*. Nessa árvore é possível observar que os nodos folhas contém o valor de utilidade para as configurações de final de jogo de cada estado, e nos nodos superiores é escolhida a jogada que minimiza as chances do jogador MIN ganhar, e aumenta as chances do jogador MAX ganhar.

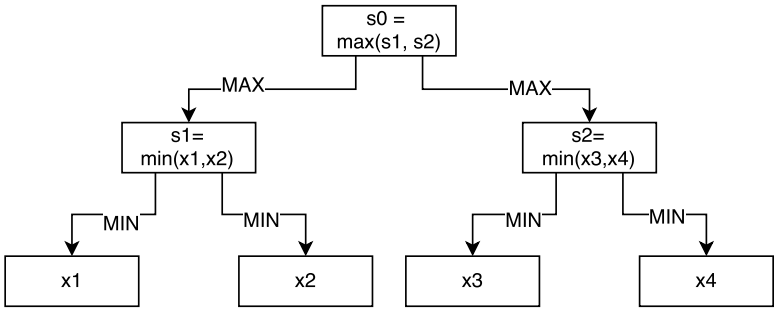


Figura 3.3 – Exemplo de game tree utilizando *minimax search*

O método de *minimax search* assume que todos os jogadores são racionais. Com isso o algoritmo considera que os 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.

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
```

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* no estilo *texas holdem* esse número pode chegar a $10^{80}$. Geralmente, as ações devem ser tomadas em um curto período de tempo. 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 [10]. Outras abordagens diminuem o espaço de estados, cortando caminhos que não influenciam no resultado final [11, Capítulo 5]

# 4. PLANEJAMENTO

As pessoas realizam a atividade de planejar alguma coisa 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 para conquistar um objetivo.

O planejamento na área da IA, é uma subárea de estudo, que se usa 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, para isto pode ser utilizado algoritmos de buscas e heurísticas [6][11, Capítulo 10].

### 4.1.1 Representação de um problema de planejamento

Uma das maneiras de representar um problema de planejamento é utilizando lógica proposicional. 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, para representar que uma lâmpada está apagada, 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, pode-se representar que a luz está apagada e a TV está ligada, uma forma de fazer isto é *apagada* $\wedge$ *ligada*. A lógica proposicional pode ser considerada simples, mas ela serve de base para as lógicas mais expressivas [11, Capítulo 10].

Como a lógica proposicional tem expressividade limitada, é preciso utilizar uma lógica que consiga expressar mais sobre os estados do ambiente que desejamos repre-

sentar. A lógica de primeira ordem (LPO) estende a lógica proposicional, e diferente da lógica proposicional a LPO consegue representar relação entre preposições, ou seja, as preposições podem ter aridade mais que um, um exemplo pode ser um objeto estar sobre o outro ($sobre(A, B)$). Outra diferença da LPO é a possibilidade de utilizar quantificadores para demonstrar fatos comuns sobre as preposições, como objetos que tenham a mesma cor ($\forall x\ azul(x)$).

A descrição formal de sentenças em LPO 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, que são objetos que se referem a objetos definidos, indefinidos ou a funções [11, Capítulo 10]. No exemplo da lâmpada, pode-se dizer que a lâmpada da cozinha está apagada, pelo predicado $apagada(cozinha)$. Ou ainda, 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

Alguns conceitos são importantes para entender a formalização de um problema de planejamento. Os estados são um conjunto de átomos que não podem ser divididos em sub fórmulas, eles assumem valores de verdadeiro ou falso, dependendo da interpretação do ambiente. As ações alteram o estado do ambiente, e para que sejam aplicadas é preciso que um conjunto de precondições seja satisfeito, se for, é aplicado um conjunto de efeitos sobre o ambiente [11, Capítulo 10]. Por exemplo, mover um objeto de um lugar para o outro, é preciso de um predicado 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 $s'$. Nele é aplicado todos os efeitos, removendo os predicados negativos e adicionando os positivos [4].

Os operadores são definidos como $op = (nome(t), precondies(p), efeitos(p)$. O $nome(t)$ é o nome do operador e $t$ é o conjunto de termos que compõem as precondições e os efeitos. $precondies(p)$ é o conjunto de predicados que representam as precondições do operador. $efeitos(p)$ é o conjunto de predicados que serão aplicados ao ambiente após a execução do operador. Todas os operadores disponíveis para a resolução do problema formam a descrição das ações [6].

Como nas técnicas de busca, em planejamento também é necessário ter uma descrição do problema. Formalmente, um problema de planejamento pode ser descrito como $P = (\Sigma, s_0, g)$, onde $\Sigma$ é o domínio, onde consta a descrição do problema com as ações disponíveis, $s_0$ é o estado inicial onde o problema começa e $g$ é o objetivo [6]. Voltando ao exemplo do Capítulo 3, onde um agente tenta chegar a cidade de Porto Alegre partindo da cidade de São Jerônimo, podemos formalizá-lo como:

- **estado inicial** ($s_0$) = $At(São\ Jerônimo)$;
- **objetivo** ($g$) = $At(Porto\ Alegre)$; e
- **domínio** ($\Sigma$) =
  - 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 que, quando executada partindo do estado inicial, modifica o estado de forma que o objetivo seja válido no estado resultante. A Figura 4.1 ilustra o comportamento do planejador.

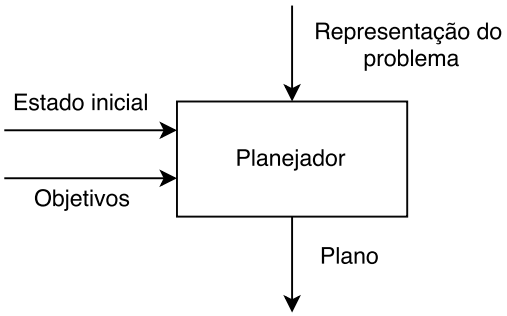


Figura 4.1 – Problema de planejamento clássico.

Um plano é considerado ótimo quando não há nenhum outro plano que atinga o objetivo utilizando um número menor de ações que ele. Uma forma de gerar planos ótimos é utilizando 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

O planejamento clássico consegue gerar planos ótimos. Mas o tratamento de problemas de grande complexidade ou com o espaço de estados grande pode se tornar

intratável. Isso acontece por conta do poder computacional necessário para que o plano consiga ser gerado. O planejamento clássico possui uma expressividade limitada, pois não é possível representar quando uma ação irá ocorrer, por exemplo [11, Capítulo 11]. 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 [6]. 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 alto nível. O conhecimento de domínio contém informações que determinam o espaço de estados possíveis para determinado ambiente. O planejador utiliza o conhecimento de domínio para percorrer entre os estados e encontrar os planos possíveis. Para isso, o conhecimento de domínio deve conter informações relativas as ações, disponíveis no ambiente, e as possíveis transições de estados [11, Capítulo 11].

O objetivo do planejamento HTN é produzir uma sequência de ações que executem determinada tarefa $t$, está tarefas está completa quando todas as tarefas não primitivas são decompostas 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 [11, Capítulo 11]. Já as tarefas não primitivas 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, podendo ser representado em HTN como: `(move ?from ?to)`. Um exemplo de tarefa não primitiva é realizar uma viagem de carro, para completar essa tarefa é necessário fazer a revisão do carro, arrumar as malas e colocar as bagagens dentro do carro, uma representação destas tarefas pode ser: `(travelByCar ?car ?stuffs)`.

Para iniciar a geração de um plano HTN, deve ser usado como início uma tarefa de ligação. 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 [4]. 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 $w$ [6]. 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.

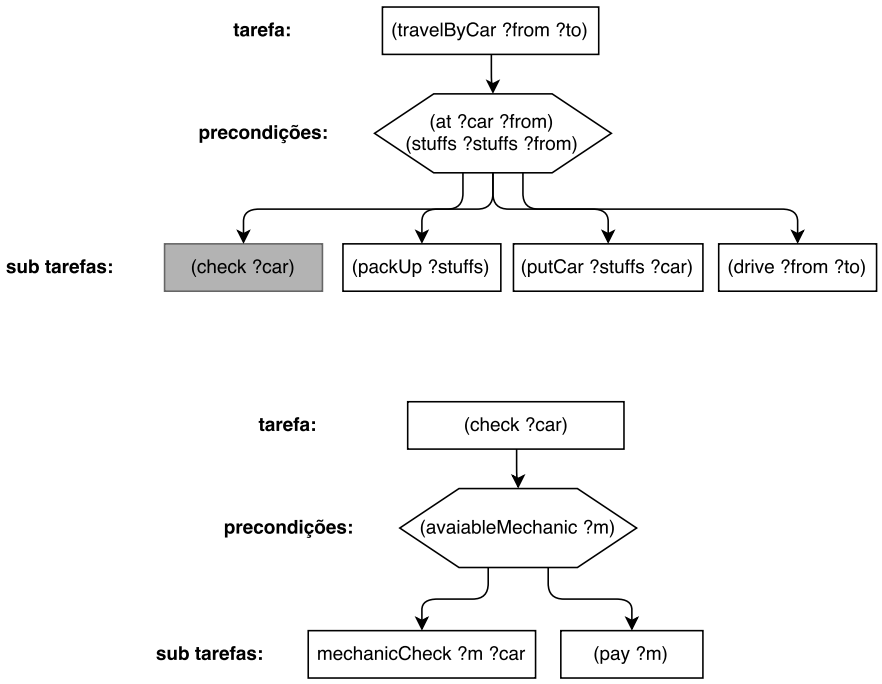


Figura 4.2 – Exemplo de método HTN.

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 [4]. Se em algum ponto, nenhum método consiga ser aplicado, o planejador deve realizar um retrocesso(*backtracking*), que consiste em voltar a um $d$ anterior a ponto de conseguir aplicar outra decomposição [11, Capítulo 11]. É possível representar a busca do plano por uma árvore $N$, na 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 [9]. A Figura 4.3 ilustra a árvore de resolução do exemplo anterior.

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 [6].

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
```

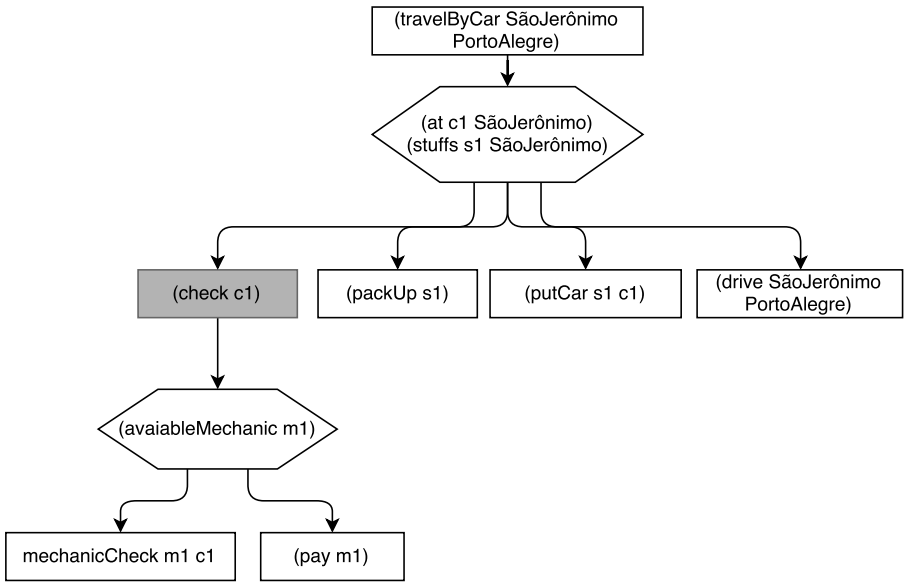


Figura 4.3 – Arvore de resolução HTN.

## 4.3 Adversarial hierarchical-task network

*Adversarial hierarchical-task network* (AHTN) é um algoritmo desenvolvido para lidar com o problema do grande fator de ramificação dos jogos em tempo real [9] 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 [9] é a representação da técnica de AHTN, e assume que existem dois jogadores, MAX e MIN, como no algoritmo de *minimax search* apresentado na Seção 3.2. 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.

O algoritmo de AHTN gera uma árvore das jogadas. Cada nodo da árvore 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 $N_-$. Cada nodo da árvore representa um estado do jogo [9].

O algoritmo alterna a perspectiva dos jogadores sempre que existir uma tarefa primitiva no plano atual, e assim avança na profundidade da árvore das jogadas. A função *AHTNMin* é responsável pela mudança de perspectiva, e ela está presente no bloco que inicia na Linha 5. No mesmo bloco há a função `nextAction(N,t)`, que é responsável por

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
```

verificar é possível a troca de perspectiva. Ela recebe como parâmetro um plano ($N$) e um ponteiro ($t$). Com isso ela retorna um ponteiro para a próxima tarefa primitiva, caso ainda não haja uma tarefa primitiva no plano, a função retorna $t = \perp$.

Caso não seja possível trocar de perspectiva, o algoritmo de AHTN decompõe o plano. O método *decompositions* é responsável por decompor o plano atual a partir do estado do jogo na árvore. O método apenas adiciona novos planos que utilizem apenas um novo método ao plano atual. A chamada deste método ocorre na Linha 10 do algoritmo gerando novos planos. O algoritmo utiliza as decomposições para comparar todos os planos gerados por uma função de avaliação. O plano que tiver a melhor função de avaliação é escolhido como melhor caminho. O plano escolhido é retornado junto com o resultado da função de avaliação. Este processo está presente na Linha 11.

O plano pode acabar quando a profundidade limite for atingida, ou quando um estado terminal for alcançado. Quando alguma dessas condições for alcançada, é retornado o plano das duas perspectivas e a função de avaliação para o estado do jogo atual. A Linha 2 mostra essa condição.

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) [9].

A Figura 4.4[1] é utilizada para exemplificar o funcionamento do algoritmo 4.2. Na raiz da árvore há apenas uma tarefa não primitiva *win* para ser decomposta, para os dois

[1] Figura extraída de [9]

jogadores. Há duas decomposições que o jogador MAX pode aplicar para seu plano HTN, resultando nos nodos $n_1$ e $n_5$. A decomposição $n_1$ não resulta em nenhuma ação primitiva, e por isso $n_1$ continua um nodo MAX. Uma vez que o jogador MAX decompõe seus planos e há tarefas primitivas para serem executadas (nodos $n_2$ e $n_5$), é o turno de MIN decompor suas tarefas não primitivas. Após MIN gerar suas ações, a profundidade máxima ($d = 2$) 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.

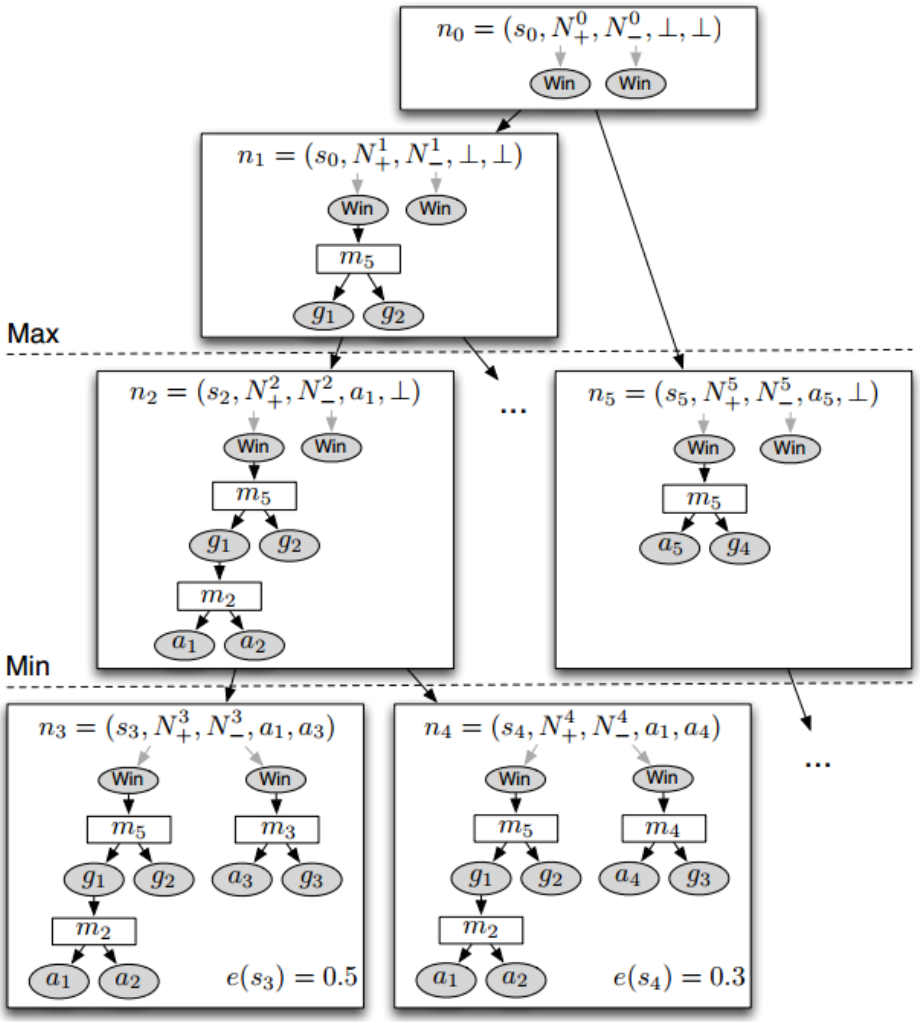


Figura 4.4 – Árvore gerada pelo algoritmo de AHTN.

O problema que o algoritmo de AHTN tenta lidar é o grande fator de ramificação e o pouco tempo para determinar a próxima ação do agente. O algoritmo utiliza planejamento através do conhecimento de domínio para diminuir a combinação de ações que não tem nenhuma chance de ser usadas em um jogo real [9].

# 5. PROJETO DE IMPLEMENTAÇÃO

Este capítulo apresenta o jogo MicroRTS, que foi escolhido para ser utilizado como plataforma para a implementação do algoritmo de AHTN. Este capítulo também apresenta como funciona o planejador HTN, que será utilizado para a geração de planos.

## 5.1 MicroRTS

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 os jogadores consigam ficar imersos 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. Não é sempre que uma IA complicada terá melhor desempenho do que uma mais simples. Uma boa IA, para jogos, é feita a partir do comportamento desejado para o jogo com os algoritmo certos [5].

Jogos de estratégia em tempo real, também conhecidos por *real-time strategy games* (RTS), é um subgênero de jogos de estratégia. Nesse gênero de jogo os jogadores devem construir uma base, buscar recursos, construir edificações, treinar unidades de ataques e aprimorar suas tecnologias. O objetivo final do jogo é destruir uma ou mais bases inimigas. Alguns fatores dificultam o desenvolvimento de IAs para jogos RTS. Os fatores estão relacionados com a complexidade dos jogos, pois os jogos RTS possuem um grande espaço de estados. Além disso os jogadores realizam as jogadas ao mesmo tempo, fazendo com que cada jogador tenha um curto espaço de tempo para realizar as suas ações. Por essa razão, não é possível traduzir automaticamente as técnicas de IA para jogos RTS sem algum tipo de abstração ou simplificação [10, 1].

Um exemplo deste gênero é o MicroRTS[1], uma simplificação de jogos como Starcraft[2]. O MicroRTS foi desenvolvido por Santiago Ontañón [8] em Java 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 ganhar o jogo é preciso eliminar cada unidade e edificação do adversário. O jogo termina quando um dos dois jogadores não tem mais unidades, ou quando o jogo atinge o limite de tempo. A Figura 5.1 mostra um exemplo de tela do jogo.

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

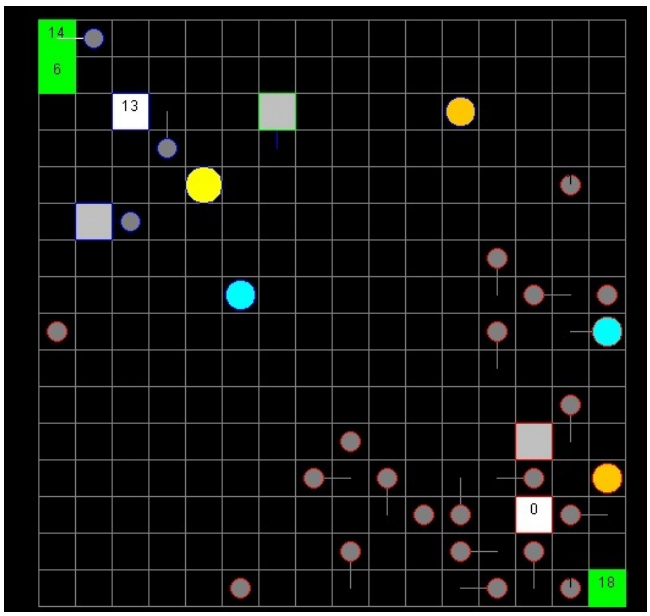


Figura 5.1 – Um exemplo de tela do MicroRTS

### 5.1.1 Unidades e construções

No MicroRTS existem quatro tipos de unidades no jogo, um trabalhor chamado de *worker*, e três unidades de ataque distintas. A unidade *worker*, é responsável por coletar recursos e construir as edificações. Esta unidade também consegue lutar, mas possui um dano muito baixo. As unidades de ataque podem apenas atacar, elas custam a mesma quantidade de recursos para serem produzidas, mas suas características são diferentes. A unidade *heavy* possui um alto poder de ataque, mas sua velocidade é baixa. Já a unidade *light* possui um baixo poder de ataque, mas sua velocidade é maior. Por fim, a unidade *ranged* consegue atacar de longa distância.

Para treinar as unidades é preciso ter recursos e edificações. A base é a edificação principal, e ela é responsável pela criação dos *workers*. Os *workers* coletam os recursos e armazenam na base. Os recursos são necessários para treinar e construir tudo dentro do jogo. O quartel é construído apenas por *workers*, e é responsável pela criação das unidades de ataque. Todas as unidades e edificações tem um custo em recursos para serem produzidos.

### 5.1.2 Arquitetura

O MicroRTS conta com cinco classes principais para funcionamento do jogo. Cada classe é responsável por um controle dentro do jogo. As classes *GameState* e *PhysicalGameState* são responsáveis pelo controle das ações das unidades dentro do mapa. A

*UnitTypeTable* é utilizada para associar cada unidade com as suas ações possíveis. A classe *PhysicalGameStatePanel* é responsável pela interface gráfica. A classe *GameVisualSimulation* é utilizada para unir todos os componentes do jogo, e configurar as informações de mapa, jogadores, humanos ou IAs, e tempo de duração da partida. O diagrama de classes presente na Figura 5.2 ilustra como as classes se relacionam e seus principais métodos.

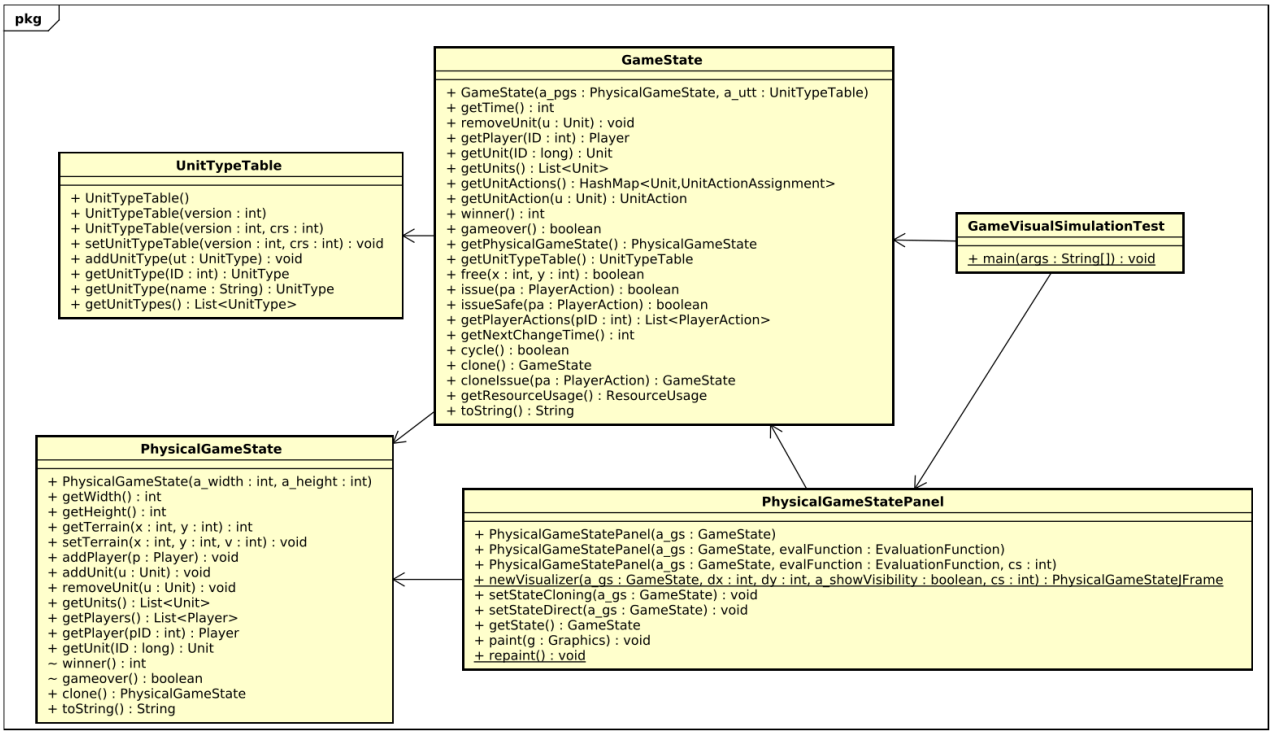


Figura 5.2 – Classes principais do MicroRTS

O MicroRTS fornece uma camada de abstração para que se possa utilizar as unidades, sem precisar ter conhecimento de como elas estão estruturadas. Essa camada se chama *AbstractionLayerAI*. Ela apresenta métodos para treinar unidades, construir edificações, coletar recursos, atacar, e movimentar as unidades. Esta camada oferece todos os recursos necessários para que seja implementado uma IA.

Toda a IA, que for implementada no MicroRTS, deve conter o método *getAction*. Este método é responsável por determinar qual ação deve ser realizada pela IA em determinado momento do jogo. Cada IA pode assumir o lugar de um jogador no MicroRTS. A classe *GameVisualSimulation*, que é responsável por gerenciar a simulação, inicializa os jogadores, e a cada período de tempo chama o método para realizar as ações dos jogadores. A Figura 5.3 ilustra o diagrama de sequência do funcionamento do método `main` da classe *GameVisualSimulation*, e o Algoritmo 5.1 apresenta o pseudo código do método.

A classe primeiro obtém as instâncias dos objetos necessários para a simulação do jogo: esse processo está representado a partir da linha 2 até a linha 6. A linha 7 é onde o algoritmo chama a classe da interface gráfica para desenhar a tela do jogo. Após

isso, o jogo entra em um laço para que os jogadores façam suas jogadas. As linhas 9 e 10 são onde os jogadores escolhem qual as suas jogadas. Nas linhas 11 e 12, as jogadas são testadas para que não haja nenhuma violação quanto as restrições do jogo. Isso é necessário para que não haja nenhuma inconsistência no jogo, como por exemplo, acessar uma posição que não é possível, ou ainda construir uma edificação no lugar de outra. Na linha 13 é responsável por determinar se o jogo ainda não acabou. A linha 14 desenha a tela novamente com as jogadas executadas. Assim o laço da linha 8 é executado até que o jogo termine com algum vencedor, ou quando o tempo de simulação acabar.

Algoritmo 5.1 – Pseudo código da classe *GameVisualSimulation*.

```
 1: function MAIN(String[])
 2:     utt = UnitTypeTable()
 3:     pgs = PhysicalGameState()
 4:     gs = GameState(pgs, utt)
 5:     player1 = Player1()
 6:     player2 = Player2()
 7:     drawScreen()
 8:     while !gameover()||endTime() do
 9:         action1 = player1.getAction()
10:         action2 = player2.getAction()
11:         issueSafe(action1)
12:         issueSafe(action2)
13:         gameover = gs.cycle()
14:         repaintScreen()
15:     end while
16: end function
```

### 5.1.3 Técnicas de IA

O MicroRTS possui algumas técnicas de IA já implementadas. Algumas destas técnicas simulam algum comportamento dentro do jogo, sem ter um algoritmo de IA, essas técnicas são chamadas de estratégias hard-coded. Estratégias hard-coded consistem em um certo tipo de script com um objetivo inflexível definido em tempo de programação. No MicroRTS existe algumas estratégias hard-coded implementadas: RandomAI e RandomBiased, LightRush, HeavyRush, RangedRush e WorkerRush. A RandomAI executa movimentos totalmente aleatórios. Já a RandomBiasedAI também executa movimentos aleatórios, mas com uma probabilidade maior de realizar três movimentos: ataque, extração de recurso, e criação de tropas. As demais estratégias são baseadas no mesmo estilo de jogo, mas com unidades de ataque diferentes. Elas utilizam um *worker* para extrair recursos, e treinam um único tipo de unidade de ataque para atacar. As técnicas são chamadas de LightRush, HeavyRush, e RangedRush, elas utilizam as unidades *light*, *heavy*, e ranged,

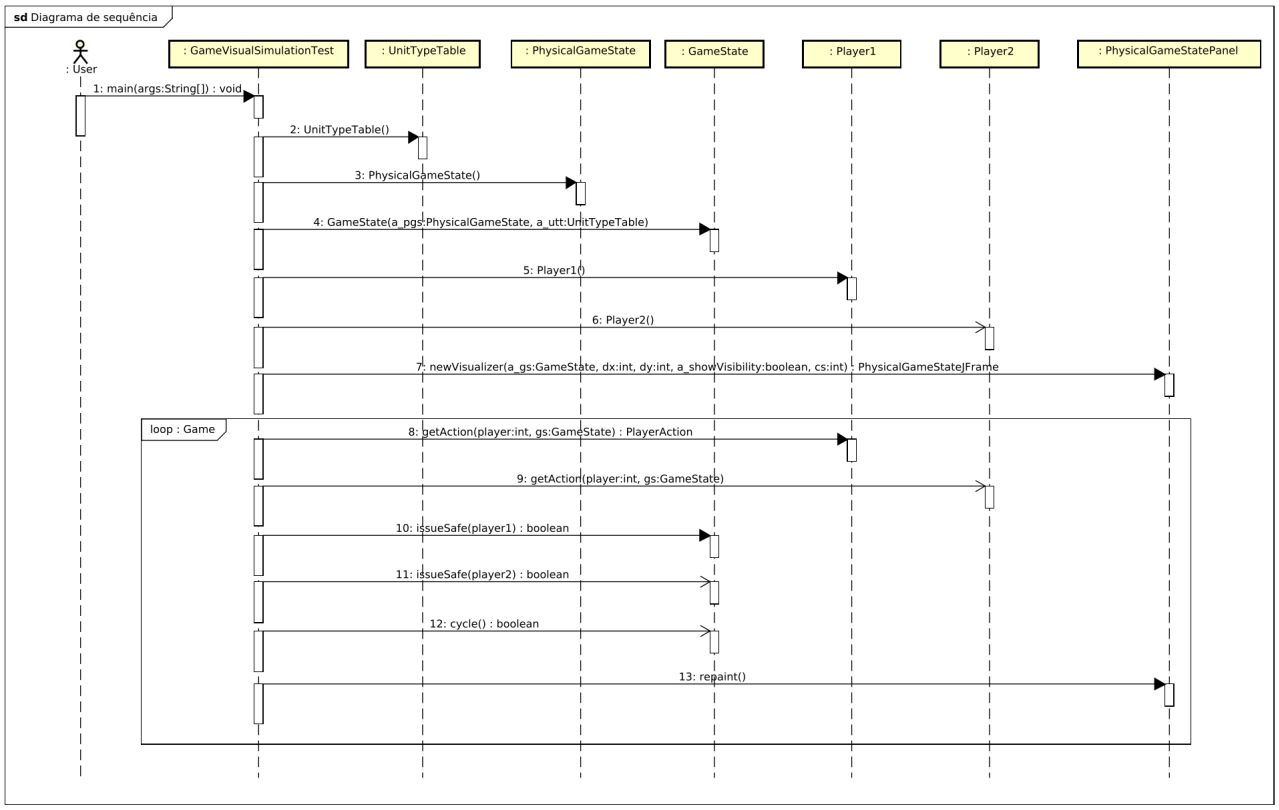


Figura 5.3 – Diagrama de sequência do método `main` da classe *GameVisualSimulation*

respectivamente. A WorkerRush consiste em ter um *worker* para realizar a extração de recursos, mas ao invés de treinar unidades de ataque, ela treina outros *workers* para realizar os ataques, isso faz com que a técnica consiga criar muitas unidades de maneira rápida.

Outras técnicas utilizam algum algoritmo de IA para decidir qual ação deve ser realizada. Como é o caso das estratégias de Minimax[7], Monte Carlo, e Portfolio Search[3]. A estratégia de Minimax utiliza o algoritmo de minimax para uma determinada profundidade. A profundidade não é em relação aos turnos, como no algoritmo comum de minimax, mas sim ao tempo disponível para tomar uma decisão. A estratégia de Monte Carlo verifica todas as ações possíveis para o estado, e simula jogos aleatórios para aquela jogada até um determinado tempo. Uma heurística é utilizada para determinar qual foi a ação que obteve o melhor valor. Já estratégia de Portfolio Search possui um conjunto de estratégias (o portfólio) e utiliza um algoritmo de busca para decidir qual a melhor estratégia para ser usada em determinado momento. O conjunto é composto pelas estratégias Hard-Coded.

[3]https://github.com/santiontanon/microrts/wiki/Artificial-Intelligence

## 5.2 Java Simple Hierarchical Ordered Planner 2

Java Simple Hierarchical Ordered Planner 2 (JSHOP2) [3] é um sistema de planejamento independente de domínio baseado em HTN. O JSHOP2 foi desenvolvido em Java por Dana Nau e sua equipe de pesquisa. O JSHOP2 recebe como entrada uma descrição do domínio e um problema de planejamento. A descrição do domínio contém a formalização das ações dos agentes, as tarefas e métodos que as decompõem. O planejador realiza a geração do plano decompondo os métodos até que só restem tarefas primitiva, a Seção 4.2 explica como é feita a decomposição. Na descrição do domínio as tarefas primitivas são descritas como operadores. Na descrição do domínio os operadores são compostos por um nome de operador, uma lista de precondições que deve ser verdadeira para a execução do operador, uma lista de elementos que serão removidos do estado, e uma lista de elementos que serão adicionados ao estado. Por exemplo, um operador que representa o deslocamento de uma pessoa de um lugar origem para um lugar destino, para isso é preciso estar no lugar origem, e assim que a pessoa se locomover ela não está mais neste local e está no local destino. Este exemplo de operador é apresentado a seguir:

```
1  (:operator (!move ?from ?to)
2    ((at ?from)) ;; Precondition
3    ((at ?from)) ;; Delete list
4    ((at ?to))   ;; Add list
5  )
```

Os métodos são utilizados para decompor tarefas de alto nível para níveis mais baixos. No JSHOP2 os métodos são identificados por um nome de método, que é único para cada método. Dentro de um método há uma lista de precondições, e uma lista de tarefas, que podem ser primitivas ou não primitivas, para quantos casos forem necessários. Por exemplo, o problema de ir de um local para outro, se o local destino tem um caminho para o local origem, é possível se locomover para o local, mas se não houver caminho é preciso ir para outra cidade que tenha caminho para o local destino. Este exemplo pode ser visto abaixo:

```
1  (:method (travel ?from ?to)
2    ((at ?from) (path ?from ?to)) ;; precondition
3    ((!move ?from ?to)) ;; task list
4
5    ((not (path ?from ?to)) (path ?from ?somewhere)) ;; precondition
6    ((!move ?from ?somewhere) (travel ?somewhere ?to)) ;; task list
7  )
```

A descrição do domínio precisa de um nome. O nome é necessário para que o planejador consiga fazer a ligação da descrição do domínio com o problema de planeja-

mento. A descrição do domínio com métodos e operadores para o exemplo citado acima ficaria assim:

```
(defdomain move
  (
    (:operator (!move ?from ?to)
      ((at ?from))
      ((at ?from))
      ((at ?to))
    )

    (:method (travel ?from ?to)
      ((at ?from) (path ?from ?to))
      ((!move ?from ?to))

      ((not (path ?from ?to)) (path ?from ?somewhere))
      ((!move ?from ?somewhere) (travel ?somewhere ?to))
    )
  )
)
```

O problema de planejamento contém as informações do estado do ambiente e qual é o objetivo do agente. O estado do ambiente é composto por predicados que são usados nos métodos e operadores, como no exemplo `at ?x` e `path ?a ?b`, como é apresentado na Seção 4.1. O objetivo do agente é um ou mais método do domínio, no exemplo `travel ?from ?to`. Um possível problema de planejamento para o exemplo pode ser o apresentado abaixo:

```
(defproblem problem move
  (
    (path PortoAlegre Charqueadas)
    (path Charqueadas SaoJeronimo)
    (at PortoAlegre)
  )

  (
    (travel PortoAlegre SaoJeronimo)
  )
)
```

O planejador necessita do domínio e do problema para gerar os planos. O plano gerado inclui o custo para executar este plano, junto com os operadores em ordem que devem ser executados para que o objetivo seja alcançado. Caso existam mais planos que levem ao mesmo objetivo, o JSHOP2 apresenta os planos na mesma estrutura. O plano gerado para o exemplo acima é o seguinte:

```
[Plan cost: 2.0

```

```
3 (!move portoalegre charqueadas)
4 (!move charqueadas saojeronimo)
5 --------------------
6 ]
```

No próximo capítulo é apresentado a modelagem do domínio, a implementação do algoritmo de AHTN no MicroRTS, e como é feita a tradução do cenário do jogo para um problema de planejamento.

# 6. IMPLEMENTAÇÃO

A implementação do algoritmo de AHTN dentro do MicroRTS foi feita em duas etapas. Primeiro foi feita a modelagem do domínio e criado qual heurística define o valor de utilidade para os estados. Na segunda etapa foi acoplado o JSHOP2 com o MicroRTS, e logo após foi feita a implementação do algoritmo AHTN.

## 6.1 Modelagem do domínio

Antes de realizar a implementação do algoritmo de AHTN, é preciso criar um domínio de planejamento. O domínio de planejamento serve para que o JSHOP2 consiga gerar os planos que serão usados pelo algoritmo.

A modelagem do domínio inicia com a especificação de qual estratégia o domínio vai adotar. O domínio é pensado em uma situação de jogo, onde o jogador quer treinar uma unidade de ataque para destruir o seu adversário. Sendo assim, o jogador precisa de um quartel para treinar as unidades, e uma base para os recursos serem guardados. O domínio tem apenas uma de cada edificação. Apenas um *worker* é utilizado, ele fica responsável pela extração dos recursos e construção das edificações. Quando o quartel está criado, e há recursos para treinamento de um unidade de ataque, a unidade *ranged* é criada. Assim que uma unidade de ataque estiver pronta, ela é enviada para atacar o adversário. Apenas quando a unidade *ranged* é destruída que outra unidade é criada. Todos os mapas testados iniciam com uma base, pois a base é necessária para treinar *workers* e para guardar os recursos.

A modelagem do domínio começa pela descrição dos operadores. Cada ação que pode ser executada dentro do jogo é descrita como um operador no domínio. Por exemplo, as ações de construir um quartel, e coletar recursos, são transformadas nos operadores !*buildbarrack* e !*getresource*, respectivamente. Já os métodos servem para determinar ações de mais alto nível, e devem respeitar a estratégia definida. Por exemplo, no domínio apenas um quartel é construído, então o método deve ter uma precondição para verificar se não existe outro quartel no jogo, caso não tenha, ainda é preciso checar se há recursos suficientes para construir. Cada método pode encadear a chamada de outros métodos do domínio. Isso acontece pois pode ser necessário realizar outra ação antes de conseguir realizar a ação que o método está tentando fazer. Por exemplo, é necessário um quartel para treinar uma unidade de ataque, quando o método de treinar unidade é chamado, ele verifica que não há um quartel, e ele chama o método de construir o quartel.

Para ganhar o jogo é preciso atacar o adversário, por essa razão o método utilizado como objetivo no problema de planejamento é o *ataqueranged*. Esse método desencadeia

todas as outras chamadas de método dependendo do estado do ambiente. No Apêndice A está a descrição deste domínio, que é chamado de domínio 1.

O domínio 1 treina apenas uma unidade de ataque, o que acaba limitando o poder de ataque da estratégia. Para tentar solucionar esse problema, outro domínio foi modelado. Esse domínio tem a mesma estratégia do domínio anterior, com a diferença de que mais de uma unidade de ataque é criada. Assim, enquanto uma unidade está atacando, outra pode estar sendo treinada. No Apêndice B está a descrição deste domínio, que é chamado de domínio 2.

## 6.2 Heurística

O algoritmo de AHTN requer uma função de avaliação para determinar o valor de utilidade para cada estado do jogo. Essa função utiliza uma heurística para determinar o valor de utilidade. Foram criadas duas heurísticas. A primeira leva em consideração apenas as unidades do jogador. Quanto maior o número de unidades disponíveis no ambiente melhor a função de avaliação do estado. As unidades são ponderadas pelo custo de treinamento ou de construção, isso é necessário porque o custo de treinamento ou construção de cada unidade é diferente. A Equação 6.1 ilustra a formula para essa heurística.

$$avaliacao(s) = (1 * worker) + (5 * quartel) + (10 * base) + (2 * unidadesDeAtaque) \quad (6.1)$$

O problema dessa heurística é que as unidades do adversário não estão sendo levadas em consideração. Se um jogador tem o dobro de unidades que o outro, isso é um forte indicio que ele está ganhando o jogo. Para resolver o problema outra heurística foi criada. Essa heurística utiliza a mesma maneira de calcular da Equação 6.1, mas também calcula o mesmo valor para as unidades do adversário. O valor das suas unidades que o jogador possui é subtraído do valor das unidades do adversário. Assim em um cenário onde o jogo está com mais tropas a favor, o valor de utilidade é positivo, e caso o contrário negativo. Esta heurística foi a escolhida para ser utilizada na implementação.

Se um jogador não tem uma base, e não tem recursos para construir uma, o jogo está perdido, pois a base é necessário para armazenar os recursos. Nesse caso, as duas heurísticas retornam um valor negativo, indicando o fim do jogo.

## 6.3 Implementação

### 6.3.1 Ações do MicroRTS

Antes de iniciar a implementação do algoritmo de AHTN, é preciso criar os métodos em Java que executam as ações dentro do MicroRTS. A camada de abstração fornece as classes que são responsáveis por exercer essas funções, mas ainda assim é preciso informar qual unidade o método está chamando. Para isso foram criados 6 métodos que utilizam as classes da camada. Cada método fica responsável por apenas uma ação dentro do jogo. Os métodos realizam as seguintes ações: construir uma base, construir um quartel, treinar um *worker*, um *worker* coleta recursos, treinar unidade de ataque, e uma unidade de ataque procura unidades adversárias para atacar.

### 6.3.2 Geração dos planos

O algoritmo de AHTN decompõem todas as possibilidades de plano para um objetivo com determinada configuração do jogo. Para gerar os planos foi utilizado o planejador JSHOP2. O JSHOP2 gera duas classes java, uma representando o domínio, e outra representando o problema. A classe do domínio gerada traduz as informações relativas aos métodos, operadores, e as ligações entre eles, para estruturas de dados do Java. Já a classe do problema contém o estado do ambiente e as tarefas que desejam ser alcançadas. A classe do problema gera o plano utilizando as estruturas da classe do domínio.

A classe que representa o domínio não precisa ser alterada, pois são sempre os mesmo operadores e métodos que são utilizados para gerar o plano. Mas o problema muda a cada nova configuração do ambiente. Nesse caso a classe do problema deve ser alterado para que represente um novo estado. Essa mudança foi feita alterando a classe java originada pelo JSHOP2. A mudança consiste em gerar novos predicados a cada nova configuração. Para isso foi preciso uma análise de como é guardado internamente cada um dos componentes do problema de planejamento na classe do JSHOP2. A partir dessa análise houve o mapeamento dos estados do jogo para os predicados do problema.

Uma classe chamada de *EstadoDoJogo* foi criada para representar o estado em que o jogo está em determinado momento. A classe contém informações das unidades que cada jogador tem em determinada configuração do jogo, podendo ser alterada caso alguma ação seja executada. A cada novo estado é gerado um conjunto de predicados que representam o jogo, para que o planejador consiga gerar quais planos são possíveis de serem feitos.

A Figura 6.1 ilustra a comunicação necessária para geração dos planos. A geração de um plano é feita a partir da chamada do método `getPlanos`. Para isso, é necessário informar qual o estado do jogo. A classe do problema, gerada pelo JSHOP2, pega as informações do domínio e gera os planos a partir do estado do jogo informado. Por fim, os planos gerados são retornados para a classe dentro do MicroRTS.

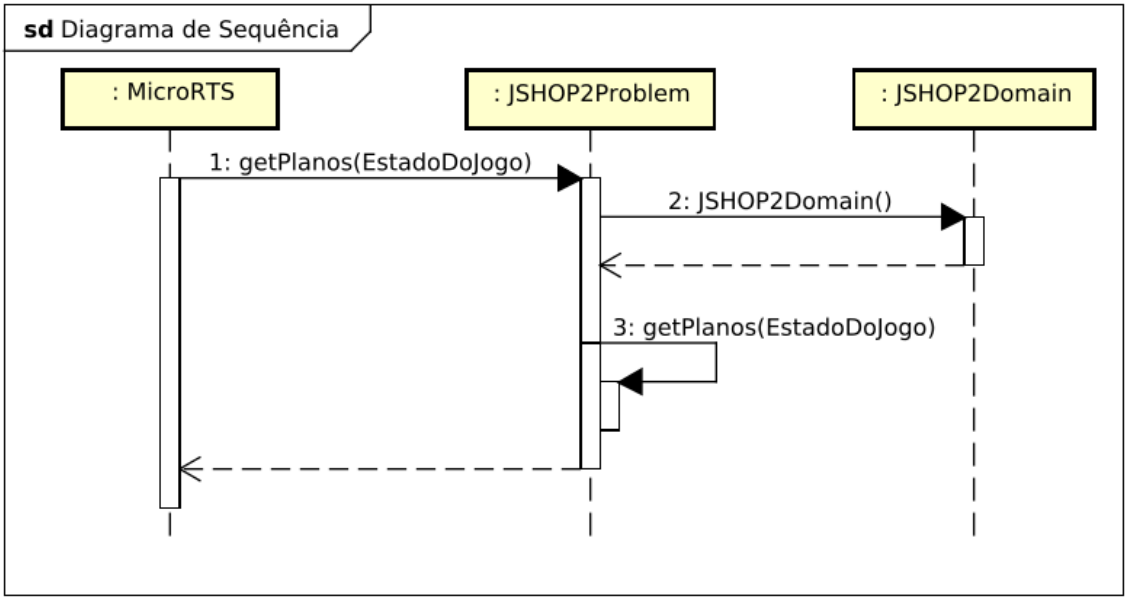


Figura 6.1 – Comunicação entre o MicroRTS e as classes do JSHOP2

### 6.3.3 Algoritmo de AHTN

Para que o MicroRTS reconheça uma IA, ela deve ter o método *getAction* implementado. Quando este método é invocado, a técnica deve realizar a sua operação e retornar qual movimento deve ser realizado. Sendo assim, a classe que implementa o algoritmo de AHTN deve utilizar este método para retornar à ação encontrada pelo algoritmo de AHTN. O algoritmo de AHTN implementado seguiu o padrão do algoritmo apresentado na Seção 4.2, mas com algumas alterações.

O JSHOP2 gera os planos apenas com os operadores utilizados para alcançar o objetivo. Com isso, o plano gerado não tem a informação de quais métodos foram utilizados para chegar aos operadores. Neste caso, o plano contém os operadores ordenados que chegam ao objetivo. No algoritmo de AHTN original o plano vai sendo decomposto através dos métodos a medida que a arvore das jogadas está expandindo, e só realiza a troca de perspectiva entre MAX e MIN quando há uma tarefa primitiva a ser executada. No algoritmo de AHTN implementado há uma pequena diferença do algoritmo original. No algoritmo implementado sempre acontece a troca de perspectiva a cada rodada. A troca de perspectiva

é feita para cada plano que é gerado a partir do estado do jogo. O Algoritmo 6.1 ilustra o pseudo código do algoritmo de AHTN implementado.

O algoritmo recebe um estado, representando qual a configuração do ambiente, um plano para a perspectiva MAX, um plano para a perspectiva MIN, e uma profundidade. A Linha 2 verifica se o estado é uma configuração de final de jogo, ou ainda se a profundidade configurada não chegou ao fim. Caso uma das duas coisas aconteça, o algoritmo retorna o plano de cada perspectiva, e também a função de avaliação para a configuração em que o jogo se encontra.

O algoritmo avança caso não haja uma configuração de final de jogo. Na linha 5 o algoritmo aplica a próxima ação primitiva do plano no estado. Neste ponto, o jogador realiza uma ação no estado. Após alterar o estado, o algoritmo gera todos os planos para MIN a partir do novo estado, e realiza a chamada para o método *AHTNMin*. O método de *AHTNMin* realiza as mesmas operações que o método de *AHTNMax* mas com a perspectiva de MIN. Seguindo, na linha 9, o algoritmo verifica qual dos planos obteve a melhor função de avaliação. Com isso, o algoritmo decide qual a melhor opção de plano. O melhor plano é retornado na linha 13.

Algoritmo 6.1 – Pseudo código do algoritmo de AHTN implementado.

```
 1: function ATHNMAX(estado, planoMax, planoMin, deph)
 2:     if terminal(estado) ∨ d ≤ 0 then
 3:         return (planoMax, planoMin, avaliacao(estado))
 4:     end if
 5:     nextAction(planoMax)
 6:     (Pmax′, Pmin′, ev′) =⊥, ⊥, −∞
 7:     for all plano ∈ getPlanos(estado) do
 8:         (Pmax, Pmin, ev) = AHTNMIN((estado, planoMax, planoMin, deph − 1))
 9:         if ev′ > ev then
10:             (Pmax′, Pmin′, ev′) = (Pmax, Pmin, ev)
11:         end if
12:     end for
13:     return (Pmax′, Pmin′, ev′)
14: end function
```

O método *getAction* é responsável pela chamada do algoritmo de AHTN. Ele realiza a chamada para o método de *AHTNMax* para todos os planos possíveis no estado atual. O retorno indica qual plano tem a melhor função de avaliação. A primeira ação do plano com a melhor função de avaliação é a escolhida para ser executada.

O método *getAction* é chamado a todo o ciclo para decidir qual a jogada deve ser realizada. As ações geralmente não são realizadas em um clico. Por exemplo, treinar uma unidade leva alguns ciclos. Assim o algoritmo gera muitas vezes a mesma ação. Foi definido um tempo de 50 ciclos para gerar uma nova ação. Com isso o número de ações geradas repetidamente foi reduzido.

# 7. RESULTADOS - AVALIAÇÃO DE DESEMPENHO

Foram utilizados três mapas para avaliar o desempenho do algoritmo. A avaliação dos resultados está dívida por mapa. Em cada mapa é possível jogar em dois lados. O lado azul começa o jogo na parte superior do mapa, e o lado vermelho começa na parte inferior. Em cada mapa é avaliado a porcentagem de vitórias para cada domínio. Os adversários, utilizados para testar o algoritmo, são as técnicas presentes no MicroRTS apresentadas na Seção 5.1.3. Cada adversário foi utilizado dez vezes em cada mapa, e em cada lado do jogo.

## 7.1 Mapa 1

No primeiro mapa, cada jogador inicia com uma base, um *worker* e dois recursos próximos a sua base. O mapa é formado por 16x16 posições. O lado azul e o lado vermelho têm a mesma formação dos componentes iniciais. A Figura 7.1 ilustra a tela do jogo com a configuração inicial do mapa.

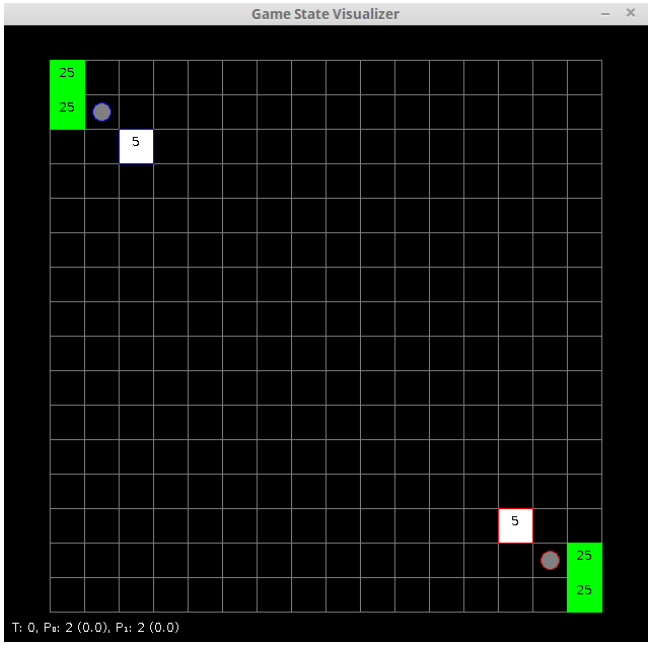


Figura 7.1 – Configuração inicial do mapa 1.

A Tabela 7.1 ilustra os resultados obtidos com o algoritmo de AHTN contra as técnicas do MicroRTS. Quando o domínio 1 é executado do lado vermelho, ocorre a criação de duas unidades de ataque ao invés de uma. Isso ocorre porque o tempo de criação da unidade é maior que o da geração de outra ação do algoritmo. Quando o algoritmo é invocado novamente, ele não tem conhecimento da unidade que está prestes a ser criada,

e assim ocorre a duplicação dessa outra unidade. O tempo de espera para que o algoritmo gere outra ação foi alterado para tentar remover esse problema. Mas com um tempo maior, as outras ações eram afetadas, e o algoritmo se torna menos vitorioso.

Tabela 7.1 – Porcentagem de vitórias do AHTN no mapa 1.

| | **Domínio 1** | | **Domínio 2** | |
|---|---|---|---|---|
| **Adversário** | **Lado Azul** | **Lado Vermelho** | **Lado Azul** | **Lado Vermelho** |
| RandomIA | 100% | 100% | 100% | 100% |
| RandomBiasedIA | 80% | 100% | 100% | 100% |
| RangedRush | 0% | 100% | 100% | 100% |
| HeavyRush | 0% | 100% | 0% | 100% |
| LightRush | 0% | 100% | 0% | 100% |
| WorkerRush | 0% | 0% | 0% | 0% |
| MonteCarlo | 60% | 80% | 100% | 100% |
| Minimax | 100% | 100% | 100% | 100% |
| Portfolio | 0% | 0% | 0% | 0% |

A partir dos dados da tabela é possível observar que há um ganho do lado vermelho em relação ao azul nos dois domínios. A ação de construir o quartel, gerada pela camada de abstração do MicroRTS, encontra uma posição mais favorável do lado vermelho. Fazendo com que o lado vermelho tenha uma melhor fluidez na hora da criação das tropas. O domínio 2 consegue melhores resultados do que o domínio 1. Adversários que não são vencidos no lado azul passam a ser vencidos com o segundo domínio, e os resultados do lado vermelho são consolidados.

## 7.2 Mapa 2

Neste mapa, cada jogador inicia com uma base, um *worker*, um quartel e um recurso perto de sua base. O mapa é composto por 8x8 posições. O quartel está construído no limite superior do mapa para o jogador azul, e no limite inferior do mapa para o jogador vermelho. A Figura 7.2 ilustra a tela do jogo com a configuração inicial do mapa.

A Tabela 7.2 ilustra os resultados obtidos neste mapa. Por conta do mapa ser pequeno em relação ao anterior, não há grande diferença entre os domínios. O jogo acaba rápido, pois as técnicas criam unidades e logo encontram as unidades adversárias. Mais uma vez o lado vermelho tem larga vantagem sobre o lado azul. Isso acontece pelo fato de que, no lado vermelho, as técnicas do MicroRTS atacam rapidamente, o que não acontece quando as técnicas estão do lado azul, pois após as unidades serem criadas, elas ficam muito tempo paradas antes de atacar. A técnica do Minimax por exemplo, no lado azul ela cria unidades de ataque, mas não envia para atacar, no lado vermelho as tropas são imediatamente postas ao ataque.

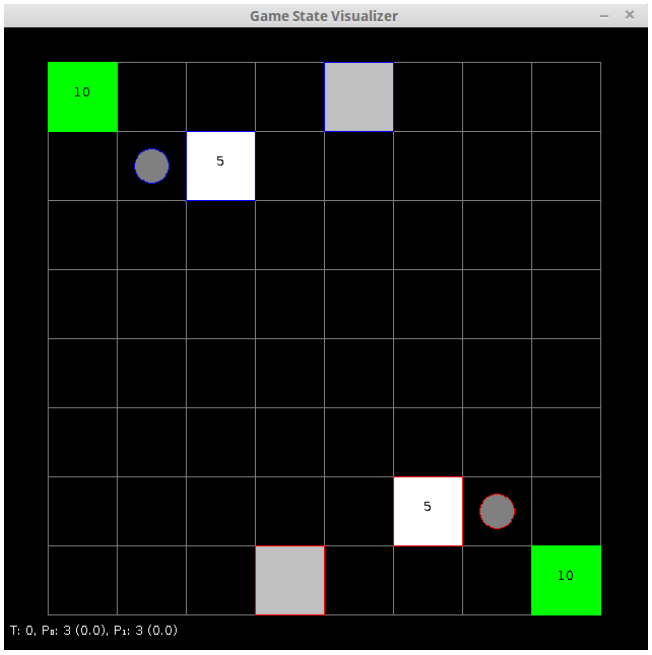


Figura 7.2 – Configuração inicial do mapa 2.

Tabela 7.2 – Porcentagem de vitórias no mapa 2.

| | **Domínio 1** | | **Domínio 2** | |
|---|---|---|---|---|
| **Adversário** | **Lado Azul** | **Lado Vermelho** | **Lado Azul** | **Lado Vermelho** |
| RandomIA | 100% | 100% | 100% | 100% |
| RandomBiasedIA | 40% | 80% | 80% | 100% |
| RangedRush | 0% | 100% | 0% | 100% |
| HeavyRush | 0% | 100% | 0% | 100% |
| LightRush | 0% | 100% | 0% | 100% |
| WorkerRush | 0% | 0% | 0% | 0% |
| MonteCarlo | 0% | 0% | 0% | 0% |
| Minimax | 0% | 100% | 0% | 100% |
| Portfolio | 0% | 60% | 0% | 80% |

## 7.3 Mapa 3

Neste mapa, cada jogador inicia com uma base, um *worker*, e um recurso perto de sua base. O mapa é composto por 8x8 posições. Além dos componentes dos jogadores, há posições obstáculos no centro do mapa. Os obstáculos não são posições validas para os jogadores. A Figura 7.3 ilustra a tela do jogo com a configuração inicial do mapa.

A Tabela 7.3 ilustra os resultados obtidos neste mapa. Os resultados mostram, outra vez, a vantagem do lado vermelho no jogo. Do mesmo modo que no mapa 1, a construção do quartel é mais favorável do lado vermelho. A colocação de obstáculos no mapa não alterou a maneira com que as técnicas se comportam.

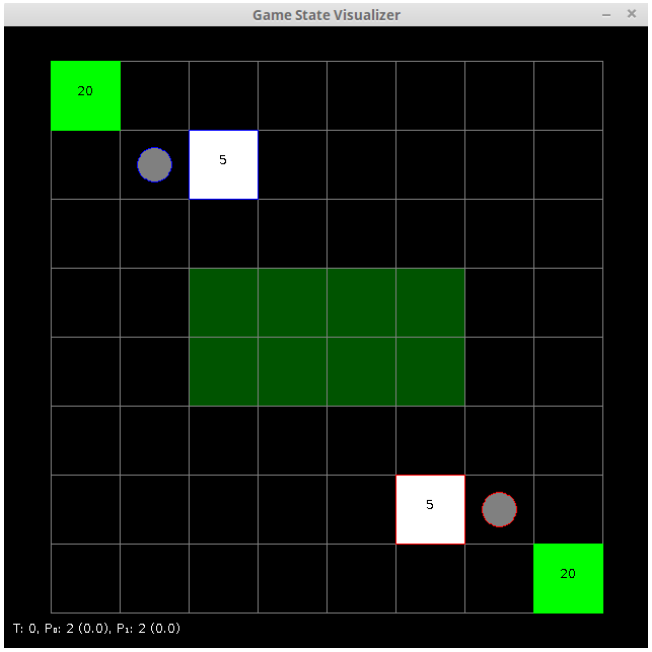


Figura 7.3 – Configuração inicial do mapa 3.

Tabela 7.3 – Porcentagem de vitórias no mapa 3.

| | **Domínio 1** | | **Domínio 2** | |
|---|---|---|---|---|
| **Adversário** | **Lado Azul** | **Lado Vermelho** | **Lado Azul** | **Lado Vermelho** |
| RandomIA | 100% | 100% | 100% | 100% |
| RandomBiasedIA | 100% | 80% | 100% | 80% |
| RangedRush | 0% | 100% | 0% | 100% |
| HeavyRush | 0% | 100% | 0% | 100% |
| LightRush | 0% | 100% | 0% | 100% |
| WorkerRush | 0% | 0% | 0% | 0% |
| MonteCarlo | 0% | 0% | 0% | 0% |
| Minimax | 0% | 80% | 80% | 80% |
| Portfolio | 0% | 0% | 0% | 0% |

A técnica de WorkerRush não é vencida em nenhuma configuração de jogo. Isso acontece porque, a técnica treina *workers*, que é a unidade que custa menos recurso no jogo, e logo as envia para atacar, por essa razão o algoritmo não tem tempo de ter algum tipo de reação. A técnica de Portfolio perde apenas no mapa 2, pois as tropas ficam um tempo parada antes de atacar, o que não acontece nos outros mapas. O tempo parado faz com que o algoritmo de AHTN consiga gerar as ações necessárias para que unidades de ataque sejam criadas.

O grande problema dos domínios modelados é não ter um plano de contingência. O adversário pode estar criando muitas tropas, mas o domínio segue o plano de criar suas unidades de ataque, o que leva o adversário a vitória. Os domínios perdem contra técnicas que treinam unidades de ataque de forma rápida, por causa do tempo que leva para construir um quartel e revidar ao ataque.

## 7.4 Tamanho de cada IA

Algumas técnicas podem precisar de muito esforço de programação para serem implementadas, já outras podem ser feitas de maneira mais rápida e ter a mesma eficiência. Uma maneira de comparar as técnicas é através do tamanho do código. Assim é possível ver o esforço empregado na criação das técnicas. Pois quanto maior o tamanho, maior o número de linhas de código necessários para a implementação da técnica. A Tabela 7.4 ilustra o tamanho de cada técnica após aplicado um algoritmo de compressão de dados.

Tabela 7.4 – Tamanho do código de cada técnica.

| **Adversário** | **Tamanho** |
|---|---|
| Domínio HTN 1 | 19,9 kB |
| Domínio HTN 2 | 20,1 kB |
| RandomIA | 4,0 kB |
| RandomBiasedIA | 4,6 kB |
| RangedRush | 13,7 kB |
| HeavyRush | 14,0 kB |
| LightRush | 14,0 kB |
| WorkerRush | 13,6 kB |
| MonteCarlo | 18,1 kB |
| Minimax | 12,9 kB |
| Portfolio | 14,4 kB |

A implementação do algoritmo de AHTN é a técnica com maior tamanho. Isso vem do fato de que é preciso integrar com o algoritmo as classes do JSHOP2 e a camada de abstração do MicroRTS. Fazendo com que o tamanho do código cresça.

A técnica com maior tamanho do MicroRTS é a MonteCarlo. Contra o algoritmo de AHTN, essa técnica venceu sempre nos mapas 2 e 3, e perdeu apenas no mapa 1 para o domínio 2. A técnica WorkerRush é a única que não perde em nenhum cenário de jogo. Ela possui tamanho perto da media dos tamanhos. As técnicas mais simples, que apenas utilizam movimentos aleatórios, são as técnicas de menor tamanho, e ainda em alguns casos conseguem vencer.

## 7.5 Tempo de geração das ações

Em jogos RTS, o tempo de geração de uma ação é fundamental. Técnicas que demoram muito tempo para gerar as ações tendem a ser superadas. Isso ocorre porque o jogador adversário tem mais tempo para executar suas ações.

A técnica de LightRush foi escolhida como adversário para avaliar o tempo de geração das ações no algoritmo de AHTN. Essa técnica foi escolhida por ter o mesmo

comportamento contra os dois domínios. O mapa 1 foi o mapa utilizado para este teste. A Figura 7.4 ilustra o gráfico com os tempos obtidos para os domínios.

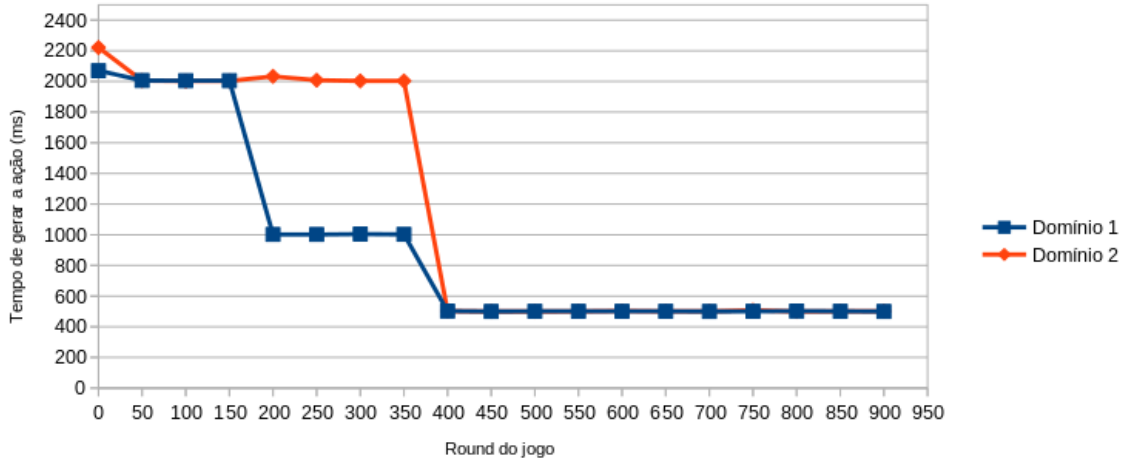


Figura 7.4 – Tempo de geração das ações para os dois domínios.

O gráfico mostra que no início do jogo o tempo para gerar uma ação é elevado. No início do jogo, o jogador tem apenas uma base e um *worker*. Para conseguir vencer o adversário é preciso construir um quartel, e treinar unidades de ataque. Ao passar das jogadas, o tempo para gerar as jogadas diminui. Isso ocorre porque há menos ações que podem ser realizadas no jogo, o que implica em menos níveis na árvore de busca que o algoritmo tem que percorrer. No final do jogo, o jogador já tem tropas de ataque e apenas manda elas atacar.

No domínio 1 o tempo de gerar a unidade de ataque é menor do que no domínio 2. Isso acontece, pelo fato de que no domínio 1, apenas uma unidade é criada, e com isso os planos gerados são diferentes, já no domínio 2 mais de uma unidade é criada. O tempo de geração ao final do jogo é semelhante, pois as duas abordagens são parecidas, e se diferem apenas na maneira como as unidades são criadas.

As técnicas do MicroRTS conseguem gerar as ações em poucos milissegundos. Já o algoritmo de AHTN implementado leva no mínimo meio segundo para determinar qual ação deve ser realizada. Isso ocorre por causa de que a cada troca de perspectiva, o algoritmo tem que gerar os planos novamente. Pois a cada novo nível na árvore de busca algum componente do ambiente mudou.

# 8. CONCLUSÕES

A realização deste trabalho me proporcionou aprendizado sobre agentes, técnicas de busca, planejamento, e jogos RTS. A implementação do MicroRTS está bem estruturada, por isso não houve dificuldades no entendimento da plataforma. O planejador JSHOP2 é de fácil acesso, e sua documentação é bem rica. Mas os exemplos de descrição do domínio e problema disponíveis com o planejador não cobrem todas as suas funcionalidades. Por essa razão a descrição do domínio foi a grande dificuldade deste trabalho. O tempo na união do MicroRTS com o JSHOP2 foi maior do que o previsto, e por essa razão, não foi possível acoplar ao algoritmo de AHTN um algoritmo de *Machine Learning*.

A avaliação dos resultados mostrou que algumas técnicas que a princípio não pareciam ser capazes de derrotar o algoritmo conseguiram ser bastante eficientes. Com tudo, a implementação do algoritmo se mostrou valida em relação as outras técnicas do MicroRTS. O grande problema foi que, em todos os casos, o tempo de geração das ações é maior no algoritmo de AHTN do que nas outras técnicas. Por exemplo, a técnica de WorkerRush, que não perdeu em nenhum dos testes, leva 1 milissegundo para determinar as suas ações. Além do tempo de geração das ações, os conhecimentos de domínio não avaliam todas as possibilidades relevantes do jogo, deixando o algoritmo limitado em relação as ações.

O algoritmo de AHTN como alternativa para abordagens tradicionais de raciocínio em jogos se mostrou uma alternativa viável para que o jogador ganhe a partida. Entretanto, o algoritmo não foi tão eficiente quanto a mitigar as limitações de eficiência computacional. Para que o algoritmo de AHTN consiga ser utilizado de melhor forma em jogos RTS, é preciso de um mecanismo para limitar o tempo de busca para geração das ações. Pois técnicas que vencem o algoritmo necessitam de menos tempo para escolher uma ação. Outro fator que pode melhorar o desempenho do algoritmo é ter outras descrições de domínio. Talvez com domínios que avaliam melhor as possibilidades de ações, e que tenha uma visão mais completa do ambiente do jogo, consiga ser mais eficiente na escolha das ações.

Para trabalhos futuros pretende-se modelar outros conhecimentos de domínio, e implementar um tempo limite para que o algoritmo gere suas ações. Além disso, pretende-se realizar o acoplamento do algoritmo de AHTN com alguma técnica de *Machine Learning*.

# APÊNDICE A – DOMÍNIO HTN DA ESTRATÉGIA 1

## A.1 Domínio

```
(defdomain ahtn-estrategia1
  (
    (:operator (!build-base ?worker ?recursoBase)
      ((worker ?worker) (have ?recursoBase))
      ((have ?recursoBase))
      ((base ?base))
    )

    (:operator (!train-worker ?base ?recursoworker)
      ((base ?base) (have ?recursoworker))
      ((have ?recursoworker))
      ((worker ?worker))
    )

    (:operator (!build-barrack ?worker ?recursoQuartel)
      ((worker ?worker) (have ?recursoQuartel))
      ((have ?recursoQuartel))
      ((quartel ?quartel))
    )

    (:operator (!train-ranged ?quartel ?recursoRanged)
      ((quartel ?quartel) (have ?recursoRanged))
      ((have ?recursoRanged))
      ((ranged ?ranged))
    )

    (:operator (!get-resource ?worker ?base ?recurso)
      ((worker ?worker) (base ?base))
      ()
      ((have ?recurso))
    )

    (:operator (!attack-ranged ?ranged)
      ((ranged ?ranged))
      ()
      ()
    )

    (:method (construir-base ?base)
      ((not (base ?base)) (recurso ?base ?recursoBase) (have ?recursoBase) (
    worker ?worker))
```

```
41        ((!build-base ?worker ?recursoBase))
42      )
43
44      (:method (treinar-worker ?worker)
45        ((not (worker ?worker)) (recurso ?worker ?recursoWorker) (have ?
      recursoWorker) (base ?base))
46        ((!train-worker ?base ?recursoWorker))
47      )
48
49      (:method (construir-quartel ?quartel)
50        ((not (quartel ?quartel)) (recurso ?quartel ?recursoQuartel) (have ?
      recursoQuartel) (worker ?worker))
51        ((!build-barrack ?worker ?recursoQuartel))
52
53        ((not (quartel ?quartel)) (recurso ?quartel ?recursoQuartel) (not (
      have ?recursoQuartel)) (worker ?worker) (base ?base))
54        ((!get-resource ?worker ?base ?recursoQuartel) (construir-quartel ?
      quartel))
55
56        ((not (quartel ?quartel)) (not (worker ?worker)) (base ?base))
57        ((treinar-worker ?worker) (construir-quartel ?quartel))
58
59        ((not (quartel ?quartel)) (not (base ?base)) (worker ?worker))
60        ((construir-base ?base) (construir-quartel ?quartel))
61      )
62
63      (:method (treinar-ranged ?ranged)
64        ((not (quartel ?quartel)))
65        ((construir-quartel ?quartel) (treinar-ranged ?ranged))
66
67        ((quartel ?quartel) (recurso ?ranged ?recursoRanged) (not (have ?
      recursoRanged)) (worker ?worker) (base ?base))
68        ((!get-resource ?worker ?base ?recursoRanged) (treinar-ranged ?ranged)
      )
69
70        ((quartel ?quartel) (recurso ?ranged ?recursoRanged))
71        ((!train-ranged ?quartel ?recursoRanged))
72      )
73
74      (:method (ataque-ranged ?ranged)
75        ((not (ranged ?ranged)))
76        ((treinar-ranged ?ranged) (ataque-ranged ?ranged))
77
78        ((ranged ?ranged))
79        ((!attack-ranged ?ranged))
80      )
81    )
82 )
```

# APÊNDICE B – DOMÍNIO HTN DA ESTRATÉGIA 2

## B.1 Domínio

```
(defdomain ahtn-estrategia2
  (
    (:operator (!build-base ?worker ?recursoBase)
      ((worker ?worker) (have ?recursoBase))
      ((have ?recursoBase))
      ((base ?base))
    )

    (:operator (!train-worker ?base ?recursoworker)
      ((base ?base) (have ?recursoworker))
      ((have ?recursoworker))
      ((worker ?worker))
    )

    (:operator (!build-barrack ?worker ?recursoQuartel)
      ((worker ?worker) (have ?recursoQuartel))
      ((have ?recursoQuartel))
      ((quartel ?quartel))
    )

    (:operator (!train-ranged ?quartel ?recursoRanged)
      ((quartel ?quartel) (have ?recursoRanged))
      ((have ?recursoRanged))
      ((ranged ?ranged))
    )

    (:operator (!get-resource ?worker ?base ?recurso)
      ((worker ?worker) (base ?base))
      ()
      ((have ?recurso))
    )

    (:operator (!attack-ranged ?ranged)
      ((ranged ?ranged))
      ()
      ()
    )

    (:operator (!train-attack-ranged ?ranged)
      ((ranged ?ranged) (quartel ?quartel) (have ?recursoRanged))
      ((have ?recursoRanged))
```

```
42      ((ranged ?ranged))
43    )
44
45    (:method (construir-base ?base)
46      ((not (base ?base)) (recurso ?base ?recursoBase) (have ?recursoBase) (
      worker ?worker))
47      ((!build-base ?worker ?recursoBase))
48    )
49
50    (:method (treinar-worker ?worker)
51      ((not (worker ?worker)) (recurso ?worker ?recursoWorker) (have ?
      recursoWorker) (base ?base))
52      ((!train-worker ?base ?recursoWorker))
53    )
54
55    (:method (construir-quartel ?quartel)
56      ((not (quartel ?quartel)) (recurso ?quartel ?recursoQuartel) (have ?
      recursoQuartel) (worker ?worker))
57      ((!build-barrack ?worker ?recursoQuartel))
58
59      ((not (quartel ?quartel)) (recurso ?quartel ?recursoQuartel) (not (
      have ?recursoQuartel)) (worker ?worker) (base ?base))
60      ((!get-resource ?worker ?base ?recursoQuartel) (construir-quartel ?
      quartel))
61
62      ((not (quartel ?quartel)) (not (worker ?worker)) (base ?base))
63      ((treinar-worker ?worker) (construir-quartel ?quartel))
64
65      ((not (quartel ?quartel)) (not (base ?base)) (worker ?worker))
66      ((construir-base ?base) (construir-quartel ?quartel))
67    )
68
69    (:method (treinar-ranged ?ranged)
70      ((not (quartel ?quartel)))
71      ((construir-quartel ?quartel) (treinar-ranged ?ranged))
72
73      ((quartel ?quartel) (recurso ?ranged ?recursoRanged) (not (have ?
      recursoRanged)) (worker ?worker) (base ?base))
74      ((!get-resource ?worker ?base ?recursoRanged) (treinar-ranged ?ranged)
      )
75
76      ((quartel ?quartel) (recurso ?ranged ?recursoRanged))
77      ((!train-ranged ?quartel ?recursoRanged))
78    )
79
80    (:method (ataque-ranged ?ranged)
81      ((not (ranged ?ranged)))
82      ((treinar-ranged ?ranged) (ataque-ranged ?ranged))
```

```
83
84        ((ranged ?ranged) (quartel ?quartel) (recurso ?ranged ?recursoRanged)
      (have ?recursoRanged))
85        ((!train-attack-ranged ?ranged))
86
87        ((ranged ?ranged) (recurso ?ranged ?recursoRanged) (not (have ?
      recursoRanged)))
88        ((!attack-ranged ?ranged) (treinar-ranged ?ranged))
89     )
90   )
91 )
```

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] Buro, M.; Churchill, D. “Real-time strategy game competitions”, *AI Magazine*, vol. Vol 33, No 3, 2012, pp. 106–108.

[2] Helmert, M.; Haslum, P.; Hoffmann, J.; et al.. “Flexible abstraction heuristics for optimal sequential planning”. In: International Conference on Automated Planning and Scheduling, 2007, pp. 176–183.

[3] Ilghami, O.; Nau, D. S. “A general approach to synthesize problem-specific planners”, *University of Maryland*, 2003.

[4] Meneguzzi, F.; De Silva, L. “Planning in bdi agents: a survey of the integration of planning algorithms and agent reasoning”, *The Knowledge Engineering Review*, vol. Vol. 30:1, 2015, pp. 1–44.

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

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

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

[8] 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.

[9] 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.

[10] 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, pp. 1–19.

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

[12] Shoham, Y. “Agent-oriented programming”, *Artificial Intelligence*, vol. 60–1, 1993, pp. 51–92.

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