

**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 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 PARA JOGOS EM TEMPO REAL

RESUMO

Jogos de estratégia em tempo real são difíceis ao ponto de vista da Inteligência 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 estratégia 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

Figura 2.1 – Representação de um agente	10
Figura 2.2 – Arquitetura simples de um agente.	12
Figura 2.3 – Arquitetura de agentes com estados.	13
Figura 3.1 – Mapa para o exemplo de problema de busca	15
Figura 3.2 – Exemplo de um pedaço de uma <i>game tree</i> sobre o jogo da velha . . .	17
Figura 3.3 – Exemplo de game tree utilizando <i>minimax search</i>	17
Figura 4.1 – Problema de planejamento clássico.	21
Figura 4.2 – Exemplo de método HTN.	23
Figura 4.3 – Árvore de resolução HTN.	25
Figura 4.4 – Árvore gerada pelo algoritmo de AHTN	27
Figura 6.1 – Um exemplo de tela do MicroRTS	32
Figura 6.2 – Arquitetura MicroRTS	33
Figura 6.3 – Classes do MicroRTS	34
Figura 6.4 – Diagrama de sequência	35

LISTA DE ALGORITMOS

Algoritmo 3.1 – Minimax Search	18
Algoritmo 4.1 – Total-order Forward Decomposition	24
Algoritmo 4.2 – Adversarial hierarchical-task network	26

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	8
2	AGENTES	10
2.1	AMBIENTES	11
2.2	ARQUITETURAS DE AGENTES	12
3	BUSCA	14
3.1	BUSCA ADVERSARIA	15
3.2	MINIMAX SEARCH	16
4	PLANEJAMENTO	19
4.1	PLANEJAMENTO AUTOMATIZADO	19
4.1.1	REPRESENTAÇÃO DE UM PROBLEMA DE PLANEJAMENTO	19
4.1.2	FORMALIZAÇÃO DE UM PROBLEMA DE PLANEJAMENTO	20
4.2	PLANEJAMENTO HIERÁRQUICO	22
4.3	AHTN	25
5	OBJETIVOS	29
5.1	OBJETIVO GERAL	29
5.2	OBJETIVOS ESPECÍFICOS	29
5.2.1	OBJETIVOS FUNDAMENTAIS	29
5.2.2	OBJETIVOS DESEJÁVEIS	29
6	ANÁLISE E PROJETO	31
6.1	JOGOS	31
6.1.1	JOGOS DE ESTRATÉGIA EM TEMPO REAL	31
6.1.2	MICRORTS	32
6.1.3	ARQUITETURA DO MICRORTS	33
6.2	DESCRIÇÃO DO PROJETO	34
	REFERÊNCIAS	36

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 as 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 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, definir qual a ação deve ser escolhida através de uma busca pelas possibilidades disponíveis para o jogo. Nesse gênero de jogos há um adversário com quem se joga contra, e as jogadas do seu adversário afetam as suas jogadas, direta ou indiretamente. Levando isso em consideração as técnicas de busca adversária conseguem determinar qual a próxima ação que deve ser tomada com o objetivo de ganhar o jogo, levando em consideração o seu adversário. O problema dessas técnicas é que elas devem explorar pelo menos uma parte do espaço de estados, e com isso jogos que tenham

¹<http://us.battle.net/sc2/pt/>

²<http://www.ageofempires.com/>

uma grande quantidade de jogadas possíveis, esse tipo de técnica se tornar inviável, pelo tempo que é preciso para gerar uma ação [9]. Outra maneira de escolher uma jogada é pensando em como conquistar seus objetivos, por exemplo, para ganhar do adversário é preciso criar tropas ofensivas, mas para isso é preciso ter um quartel. Então, planejar as ações necessárias, pensando em algum objetivo é uma maneira de pensar em qual ação deve ser escolhida. As técnicas de planejamento automatizado geram planos com o intuito de chegar a um objetivo.

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 na plataforma MicroRTS³, 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 constitui a parte final do trabalho de conclusão de curso, relatando o desenvolvimento do que foi proposto no trabalho de conclusão 1. 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 para conseguir entender como o algoritmo de AHTN funciona. O Capítulo ?? apresenta técnicas de aprendizado de máquina que podem ser usadas em jogos. No Capítulo 5 é proposto os objetivos para este trabalho, enquanto no Capítulo 6 é onde são definidas as atividades que serão realizadas em seguida, junto com a modelagem do problema. –colocar os novos capitulos aqui–

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 contínua 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, Capítulo 7]. A interação de um agente com o ambiente pode ser ilustrado pela Figura 2.1.

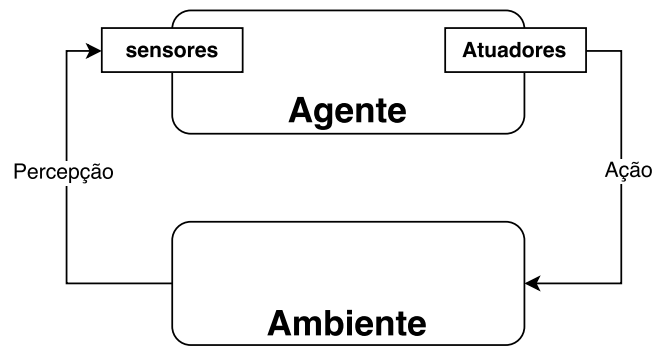


Figura 2.1 – Representação de um agente

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 [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, 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 [13, Capítulo 2].

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, Capítulo 2] [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 sempre levarem para os estados conhecidos [13, 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 [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, Capítulo 2].

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.

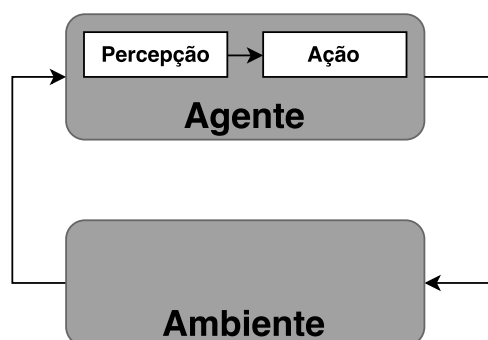


Figura 2.2 – Arquitetura simples de um agente.

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, 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 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, 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 [13, Capítulo 7]. 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-

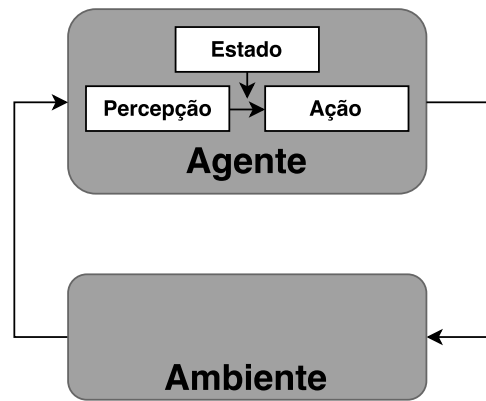


Figura 2.3 – Arquitetura de agentes com estados.

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, Capítulo 7]. 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

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 como entrada é recebido um problema e como resolução é retornado um conjunto de ações. Um problema pode ser definido por cinco componentes [13, 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 distancia entre duas localidades, ou o custo monetário de se locomover.

Esses 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 [13, 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 ligam as cidades uma a outra. 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 escolha para iniciar, mas a técnica só terá como saber após realizar todas as possíveis opções de caminhos, ou se utilizar uma técnica que

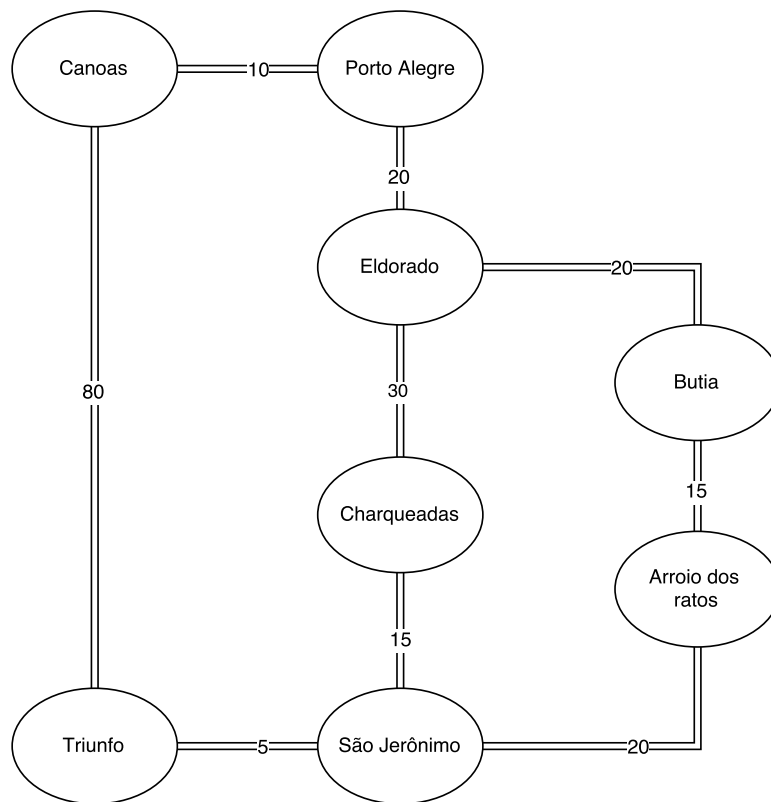


Figura 3.1 – Mapa para o exemplo de problema de busca

utilize uma função de heurística, levando em consideração os custos dos caminhos, que acrescentam um conhecimento extra para a resolução do problema [13, Capítulo 3].

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 pode ser utilizada nos jogos que estão situados em ambientes competitivos e de multi agentes. Em um jogo o jogador, preferencialmente, não informa suas jogadas previamente, mas está sempre buscando vencer o jogo, por isso 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. Isso faz com que no final do jogo, quando os valores de utilidade dos agentes forem somados, a sua soma é sempre zero. Com o intuito de resolver esse problema é possível gerar uma solução de contingência para tentar antecipar as jogadas do adversário [13, 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 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 jogadas 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. 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 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 [13, Capítulo 5].

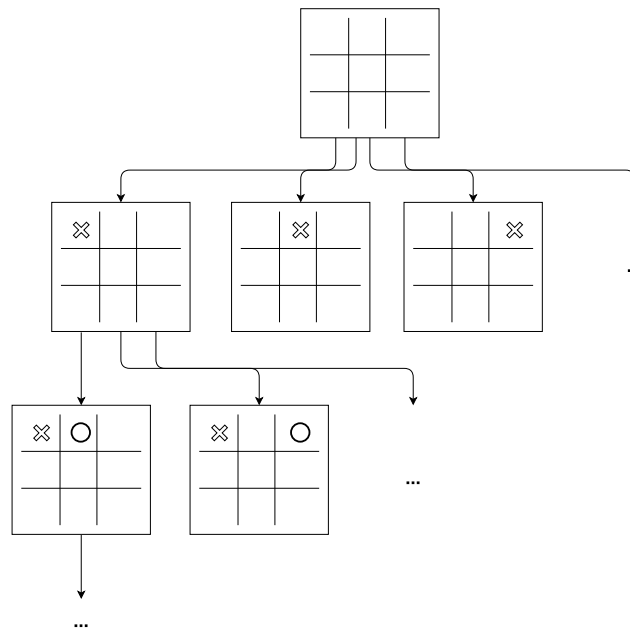


Figura 3.2 – Exemplo de um pedaço de uma *game tree* sobre o jogo da velha

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. 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, Capítulo 5]. A Figura 3.3 ilustra uma árvore de jogo, que simula o comportamento do algoritmo de *minimax search*. Nesta 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 escolhido a jogada que minimiza as chances do jogador MIN ganhar, e aumenta a chance do jogador MAX ganhar.

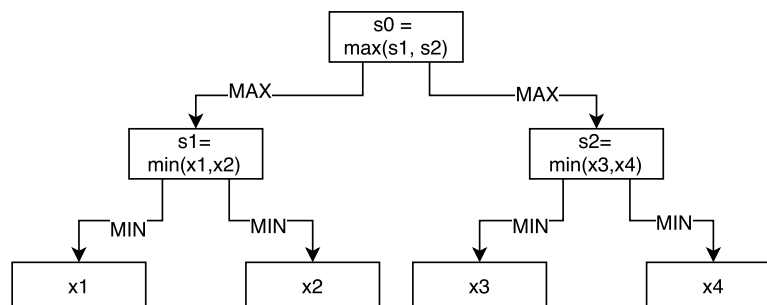


Figura 3.3 – Exemplo de game tree utilizando *minimax search*

O método de *minimax search* assume que 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.

Algoritmo 3.1 – Minimax Search

```

1: function MINIMAX(state)
2:   return  $\operatorname{argmax}_{action \in actions(s)} \min\_value(result(state, action))$ 
3: end function
4:
5: function MAX_VALUE(state)
6:   if terminal(state) then
7:     return utility(state)
8:   end if
9:    $v = -\infty$ 
10:  for all action  $\in actions(state)$  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:    $v = \infty$ 
20:  for all action  $\in actions(state)$  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 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].

Felipe

E alpha-beta
pruning?

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 específico. 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 isso pode ser utilizado algoritmos de buscas e heurísticas [8][13, 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, 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 lógicas mais expressivas [13, 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 represen-

tar. 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*)). Uma outra diferença da LPO é a possibilidade de utilizar quantificadores para demonstrar fatos comuns sobre as preposições, como objetos que tenham a mesma cor ($\forall x \text{ azul}(x)$).

A descrição formal de sentenças em LPO é composta por um predicado seguido de uma lista de termos, denotada por $p(t_0, t_1, \dots, t_n)$. 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

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, esses átomos assumem valores de verdadeiro ou falso, dependendo da interpretação do ambiente. As ações alteram o estado do ambiente. Para que elas sejam aplicadas é preciso que um conjunto de precondições seja satisfeito, se for é aplicado um conjunto de efeitos sobre o ambiente [13, Capítulo 10]. Um exemplo é mover um objeto de um lugar para o outro, precisamos 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: $\neg \text{at}(\text{from}) \wedge \text{at}(\text{to})$

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 [5].

Os operadores são definidos como $op = (\text{nome}(t), \text{precondies}(p), \text{efeitos}(p))$. 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 o resultado após a execução do operador; Todas os operadores disponíveis para a resolução do problema formam a descrição das ações [8].

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 Σ é a descrição do problema com as ações disponíveis, 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 formalizá-lo como:

- **estado inicial** (s_0) = $At(S\grave{a}o\ Jer\^o{n}imo)$;
- **objetivo** (g) = $At(Porto\ Alegre)$; e
- **domínio** (Σ) =
 - nome: $move(cityA, cityB)$
 - condições: $at(cityA) \wedge link(cityA, cityB)$
 - 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. Para realizar a formalização de um problema de planejamento é preciso definir alguns conceitos.

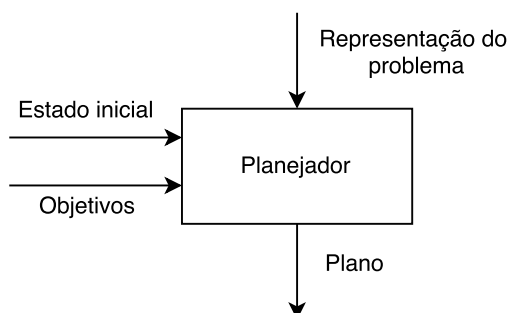


Figura 4.1 – Problema de planejamento clássico.

Um plano é considerado ótimo quando não há nenhum outro plano que atinga o objetivo utilizando um número menor de ações. 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

Embora que o planejamento clássico consiga gerar planos ótimos, o tratamento de problemas de grande complexidade ou com o espaço de estados grande pode se tornar intratável 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 [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].

Felipe: Se possível, fale um pouco mais sobre conhecimento de domínio

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, podendo ser representado em HTN como: $(move\ ?from\ ?to)$. 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 pode ser representado por $(travelByCar\ ?car\ ?stuffs)$.

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 finito 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.

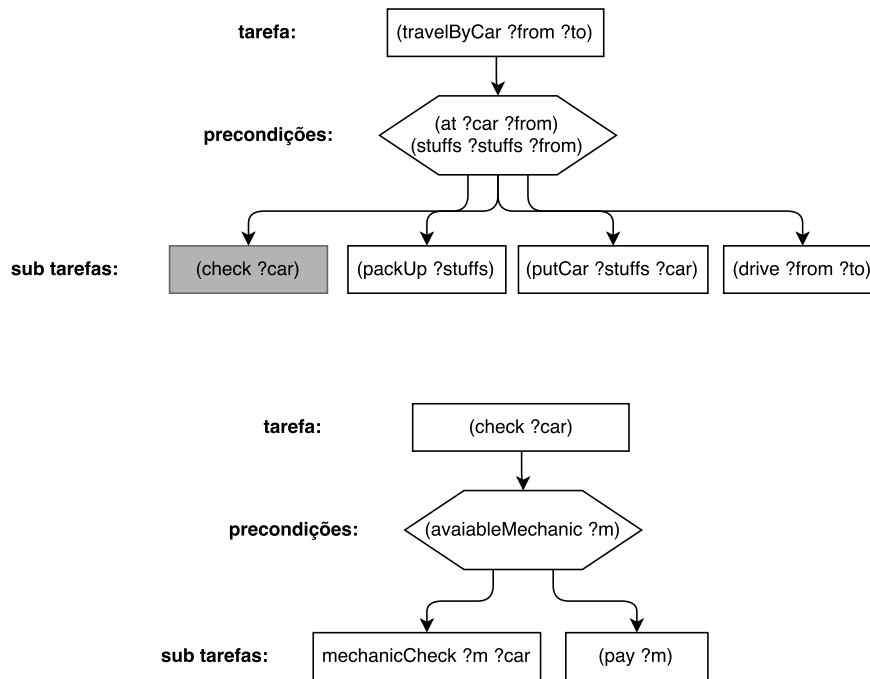


Figura 4.2 – Exemplo de método HTN.

Um problema de planeamento 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 planeamento clássico. O processo de geração de um plano utilizando planeamento 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 deve realizar um retrocesso(*backtracking*), que consiste em voltar a um d anterior a ponto de conseguir aplicar outra decomposição [13]. 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 [11]. 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 [8].

Algoritmo 4.1 – Total-order Forward Decomposition

```

1: function TFD( $s, \langle t_1, \dots, t_k \rangle, O, M$ )
2:   if  $k = 0$  then
3:     return  $\langle \rangle$ 
4:   end if
5:   if  $t_1$  é primitivo then
6:      $ativo = \{(a, b) \mid a \text{ é uma instancia de } O \text{ e é aplicável a } s \text{ e } b \text{ é uma substituição}$ 
        $\text{que torna } a \text{ relevante para } b(t_1)\}$ 
7:     if  $ativo = \emptyset$  then
8:       return falha
9:     end if
10:    escolha algum par  $(a, b) \in ativo$ 
11:     $\pi = TFD(\gamma(s, a), b(\langle t_2, \dots, t_k \rangle, O, M)$ 
12:    if  $\pi = falha$  then
13:      return falha
14:    else
15:      return  $a.\pi$ 
16:    end if
17:  else if  $t_1$  é não primitiva then
18:     $ativo = \{m \mid m \text{ é aplicável a } s \text{ e } m \in M\}$ 
19:    if  $ativo = \emptyset$  then
20:      return falha
21:    end if
22:    escolha algum par  $(m, b) \in ativo$ 
23:     $w = subtarefas(m).b(\langle t_2, \dots, t_k \rangle)$ 
24:    return  $TFD(s, w, O, M)$ 
25:  end if
26: end function

```

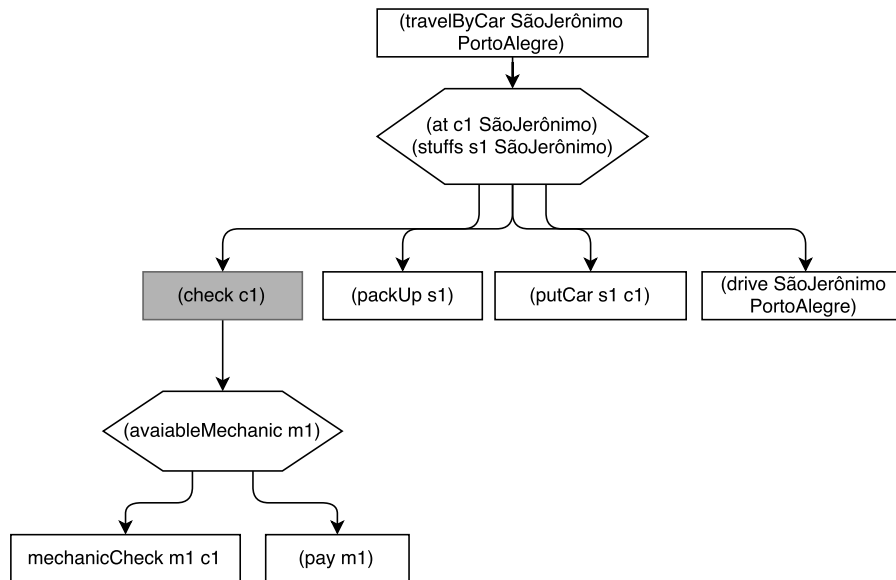



Figura 4.3 – Árvore de resolução HTN.

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 Capítulo 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.

Felipe: Não entendi a organização dos parágrafos abaixo, escreve abaixo os bullet points com o teu plano desta seção, por favor...

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, dada 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

Felipe

Referencia a seção, já que tu tens uma seção só para isto

Felipe

Parágrafo solto

Felipe

Evite a voz passiva a todo o custo

Algoritmo 4.2 – Adversarial hierarchical-task network

```

1: function AHTNMAX( $s, N_+, N_-, t_+, t_-, d$ )
2:   if  $terminal(s) \vee d \leq 0$  then
3:     return ( $N_+, N_-, e(s)$ )
4:   end if
5:   if  $nextAction(N_+, t_+) \neq \perp$  then
6:      $t = nextAction(N_+, t_+)$ 
7:     return AHTNMIN( $(\gamma(s, t), N_+, N_-, t, t_-, d - 1)$ )
8:   end if
9:    $N_+^* = \perp, N_-^* = \perp, v^* = -\infty$ 
10:   $\aleph = decompositions_+(s, N_+, N_-, t_+, t_-)$ 
11:  for all  $N \in \aleph$  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

```

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

Felipe

Evite a voz passiva a todo o custo

Felipe

Evite a voz passiva a todo o custo

Felipe

Análoga a que?

Felipe

Idem

Felipe

Evite a voz passiva a todo o custo, olhar a concordância.

Felipe

Esta questão da função de avaliação seria melhor tu ter falado lá na explicação do minimax (veja o H-MiniMAX no Russel e Norvig). Daí isto facilitaria a tua referência

Felipe

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 n_1 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.

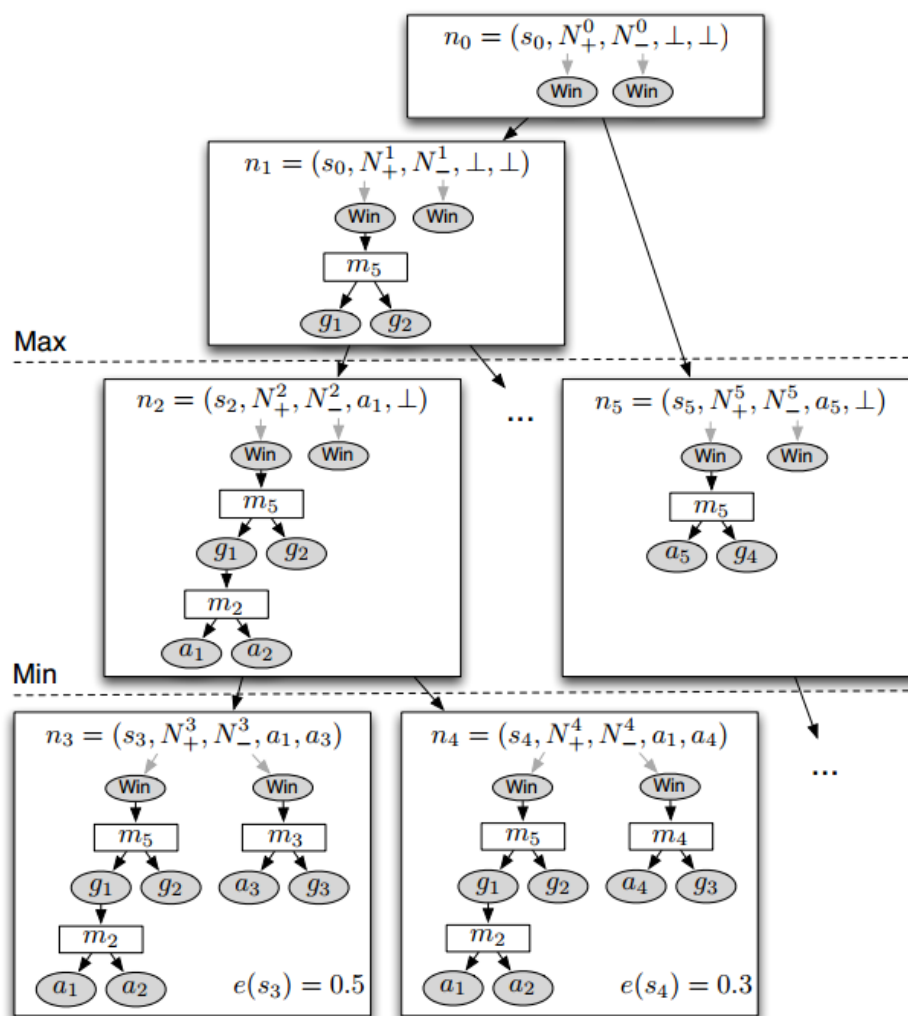


Figura 4.4 – Árvore gerada pelo algoritmo de AHTN

Em jogos, um agente pode ter que seguir um objetivo, para isso é possível utilizar técnicas de planejamento. O grande problema, continua sendo o grande fator de ramificação e o pouco tempo para determinar a próxima ação do agente []. O algoritmo de AHTN

tenta lidar com esses problemas.

Com o intuito de aumentar o desempenho do algoritmo, é possível utilizar técnicas de aprendizado de máquina na geração dos planos. No processo de geração do plano é possível ter mais de um método que decompõe a mesma tarefa não primitiva, nesses casos o planejador precisa escolher um dos métodos para tentar chegar ao fim da decomposição. Como dito anteriormente, se em algum ponto, não for possível decompor uma tarefa não primitiva, um retrocesso é feito até chegar a um ponto que outro método possa ser executado. Uma maneira de melhorar a decisão de escolha dos métodos é obtendo uma recompensa a cada decomposição de uma tarefa não primitiva, para que com as informações obtidas nas execuções, as recompensas auxiliem nas escolhas dos métodos, com o intuito de minimizar os retrocessos [3].

Felipe: E o capítulo termina assim, no mais?

Felipe

Quais problemas? É um problema o agente ter um objetivo? Na minha cabeça, se usa HTN/Planejamento para eliminar combinações de ações que não tem nenhuma chance de ser usadas em um jogo real (adicionar conhecimento de domínio), mas tu tens que verificar no paper

Felipe

O que tem a ver aprendizado de máquina com o resto do parágrafo? Com planejamento?

Felipe

Passivo!!

5. 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].

5.1 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].

5.2 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.

5.2.1 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.

5.2.2 Objetivos desejáveis

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

- Comparar 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.

6. ANÁLISE E PROJETO

6.1 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].

6.1.1 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].

6.1.2 MicroRTS

Um exemplo deste gênero é o MicroRTS¹, uma simplificação do jogo Starcraft², 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.



Figura 6.1 – Um exemplo de tela do MicroRTS

O MicroRTS consiste em dois jogadores tentando destruir a base adversária. Para destruir com o inimigo é preciso eliminar cada unidade e edificações adversárias. A Figura 6.1 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.

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

¹<https://github.com/santiontanon/micorts>

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 atacadas 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.

6.1.3 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 6.2 representa os componentes.

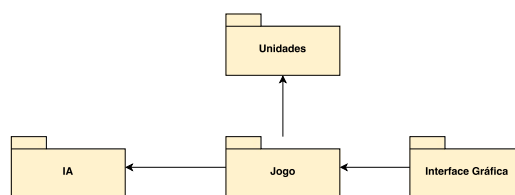


Figura 6.2 – Arquitetura MicroRTS

6.2 Descrição do Projeto

Durante o semestre foi avaliado a possibilidade de acoplar uma nova técnica de IA a plataforma do MicroRTS. O MicroRTS conta com cinco classes principais para funcionamento do jogo e uma para controle da IA. O diagrama de classes presente na Figura 6.3 ilustra os principais métodos das classes originadas do MicroRTS, e alguns dos possíveis métodos necessários para a integração da minha proposta de solução. As funcionalidades de cada classe encontra-se à seguir:

- *GameVisualSimulation* é a interface entre os componentes do jogo e o usuário;
- *GameState* e *PhysicalGameState* são responsáveis pelo controle das ações das unidades dentro do mapa;
- *UnitTypeTable* é onde cada unidade esta associada as ações possíveis no jogo;
- *PhysicalGameStatePanel* é responsável pela interface gráfica; e
- *AHTN* é onde minha proposta de solução será acoplada ao jogo.

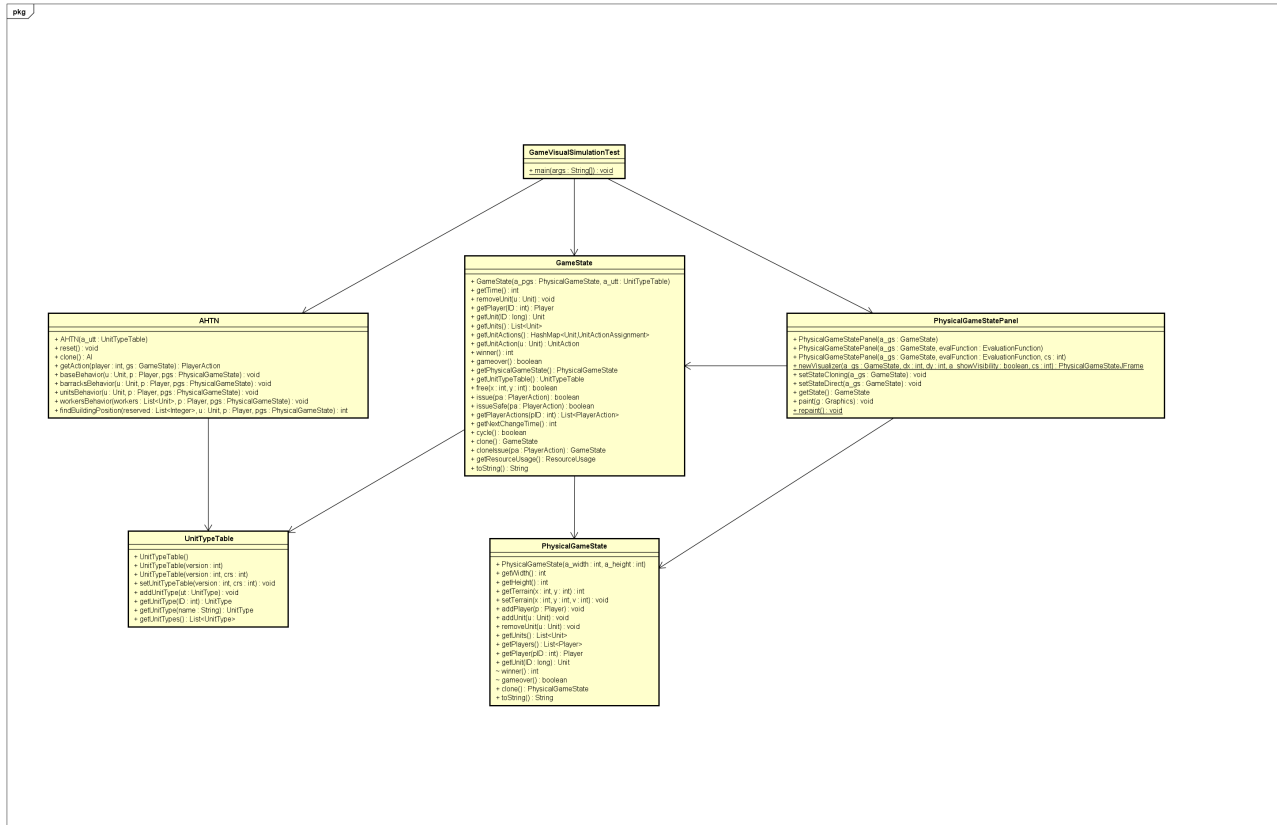


Figura 6.3 – Classes do MicroRTS

A Figura 6.4 ilustra o diagrama de sequência, com o intuito de entender o funcionamento das classes citadas anteriormente. A funcionalidade dos passos são explicados abaixo:

- Os passos de 1 a 7 são utilizados para a criação dos objetos que serão utilizados ao longo do jogo;
- Dos passos 8 a 13 fazem parte do laço do jogo, e cada passo é descrito abaixo:
 - Os passos 8 e 9 são onde cada jogador escolhe a ação que deseja tomar no momento atual do jogo;
 - Os passos 10 e 11 são onde cada jogada é testada, esse passo é necessária para que nenhuma jogada viole alguma restrição do jogo;
 - O passo 12 é necessário para verificar se o jogo não acabou; e
 - O passo 13 serve para desenhar novamente a tela com as jogadas executadas.

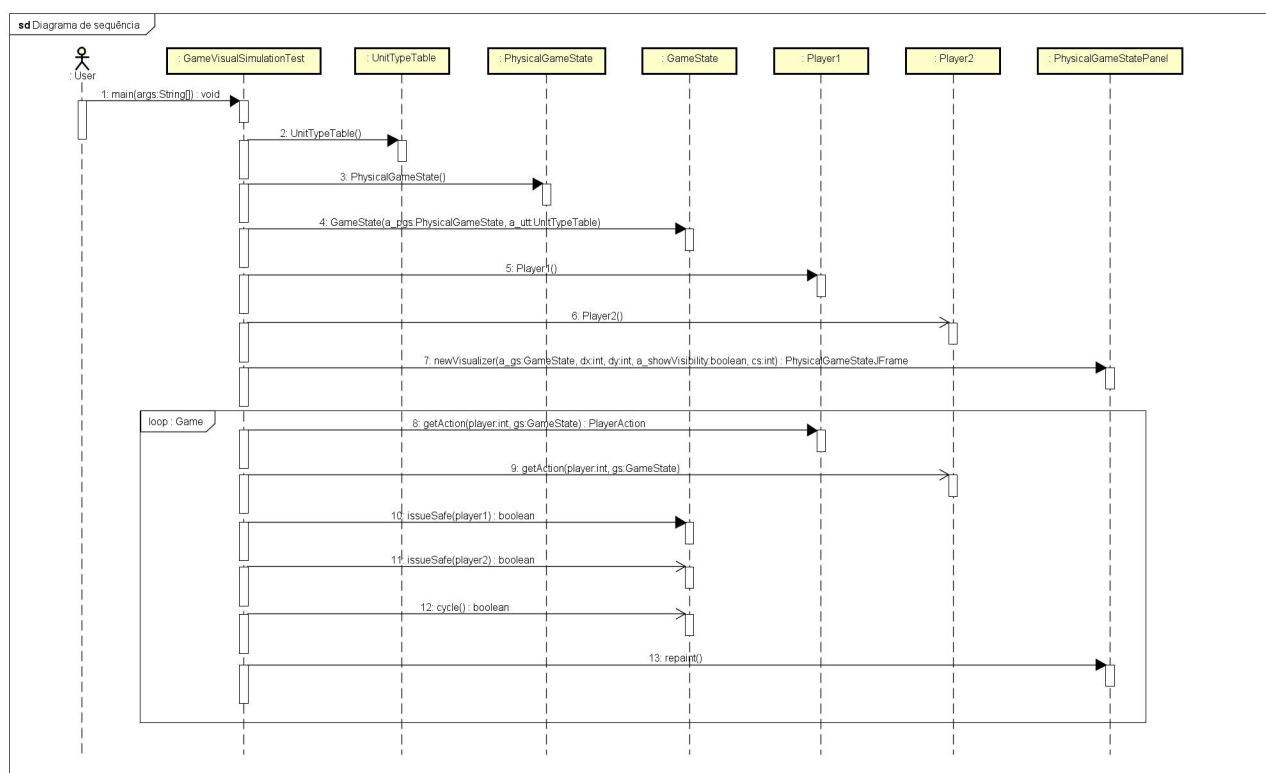


Figura 6.4 – Diagrama de sequência

O algoritmo de AHTN irá encaixar no modulo da IA na classe de *AHTN*, onde através do método *getAction* deverá retornar a ação para o estado atual do jogo a cada nova chamada.

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.