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

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

MATHEUS DE SOUZA REDECKER

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

Orientador: Prof. Felipe Rech Meneguzzi

Porto Alegre
2016

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

RESUMO

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

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

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

ABSTRACT

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

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

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

IA – Artificial Intelligence

HTN – Hierarchical Task Network

AHTN – Adversarial Hierarchical Task Network

RTS – Real-time Strategy

SUMÁRIO

1. INTRODUÇÃO

Inteligencia 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 [3]. A IA possui algumas áreas de aplicação, tais como: planejamento automatizado, jogos, robótica, tradução automática, entre outras [10].

As técnicas de IA utilizadas nos jogos são necessárias para conseguir uma melhor interação com o jogador, tornando o jogo mais real e assim prendendo a atenção do jogador [3]. As técnicas utilizadas nos jogos, geralmente, são mais simples do que as que utilizadas 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 [10]. Nos jogos as reações devem ser quase que imediatas, para isso técnicas que tentam explorar todo o espaço de estados possíveis de um jogo se tornam inviáveis para jogos mais complexos. 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 [3]. 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 [10].

Na busca de mitigar as limitações de eficiência computacional de abordagens tradicionais de raciocínio em jogos, Santiago Ontañón e Michael Buro propuseram o algoritmo chamado *Adversarial Hierarchical Task Network (AHTN)* [8]. Neste algoritmo são combinadas técnicas de HTN com o algoritmo de busca adversaria *minimax search*.

Propomos a utilização do algoritmo AHTN combinado com uma técnica de aprendizado por reforço em um jogo de estrategia em tempo real combinando uma técnica de aprendizado por reforço. Com este trabalho pretendemos mostrar que o algoritmo de AHTN apresenta melhores resultados quando aplicado junto com técnicas de aprendizado por reforço.

Este 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 mostra como podemos utilizar busca dentro do contexto de jogos. O Capítulo 4 apresenta o conceito básico de planejamento para conseguir entender como o algoritmo de AHTN funciona. No Capítulo 6 é proposto os objetivos para este trabalho. O Capítulo 7 é onde são definidas as atividades que serão realizadas em seguida, junto com a modelagem do problema.

2. AGENTES

Formalmente, agentes são entidades que agem de forma continua e autônoma em um ambiente [11]. Os agentes são capazes de receber estímulos do ambiente através de sensores e assim responder aos estímulos por intermédio de atuadores [10]. Para os agentes os estímulos do ambiente são recebidos como percepções [10]. Os atuadores, por sua vez, geram, considerando as percepções, uma ação [10]. Essa definição de agentes pode ser representada pela figura 2.1.

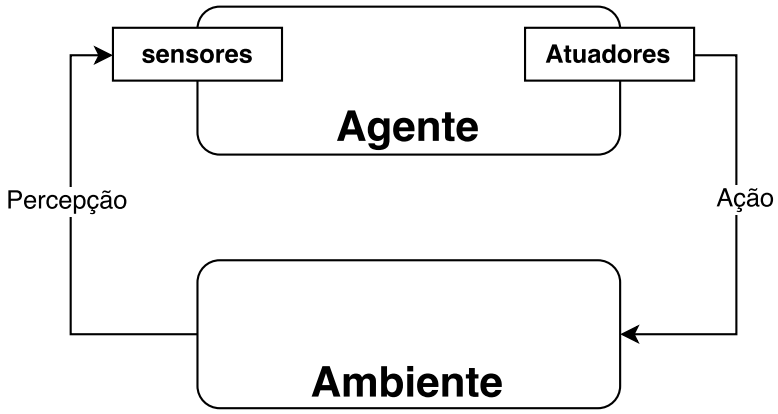


Figura 2.1 – Representação de um agente

Definindo os agentes como autônomos, eles devem ser capazes de aprender a lidar com as situações proporcionadas pelo ambiente, e serem ativos. Para que o agente consiga cumprir estes aspectos é preciso que ele seja capaz de realizar uma ação de forma autônoma e flexível, para que isso aconteça, um agente precisa de três atributos[12]:

- 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;
- Habilidade social - Para que os agentes sejam capazes de interagir com outros agentes(humanos ou não).

Os dois primeiros atributos são necessários para que o agente consiga interagir com o ambiente. Já o terceiro atributo é necessário para que o agente consiga interagir com outros agentes, pois ele nem sempre vai estar sozinho no ambiente, quando há mais de um agente no sistema, podemos considerar o sistema como multi agentes, onde os agentes interagem entre si. Os agentes podem ter objetivos em comum ou não, mas o mais importante é que eles terão que, dependendo da situação, cooperar ou negociar entre si [10].

2.1 Arquiteturas de Agentes

A definição descrita acima é uma forma geral de descrever um agente: um agente recebe um estimulo do ambiente com seus sensores e retorna uma ação por meio de seus atuadores. Mas existem diferentes tipos de arquiteturas de agentes. Cada tipo de arquitetura combina componentes diferentes para gerar as ações [10]. Os quatro tipos básicos de arquiteturas de agentes, que englobam as principais características de agentes são [10]:

- Agentes de simples reflexo (Simple reflex agents);
- Agentes de reflexos baseados em modelo (Model-based reflex agents);
- Agentes baseados em objetivo (Goal-based agents);
- Agentes baseados em utilidade (Utility-based agents).

2.1.1 Simple reflex agents

A arquitetura considerada a mais simples é a de agentes de simples reflexo. Esses agentes escolhem suas ações baseados na percepção atual vinda do ambiente, ignorando qualquer outra percepção que já tenha sido observada [10]. A figura 2.2 representa esse tipo de arquitetura.

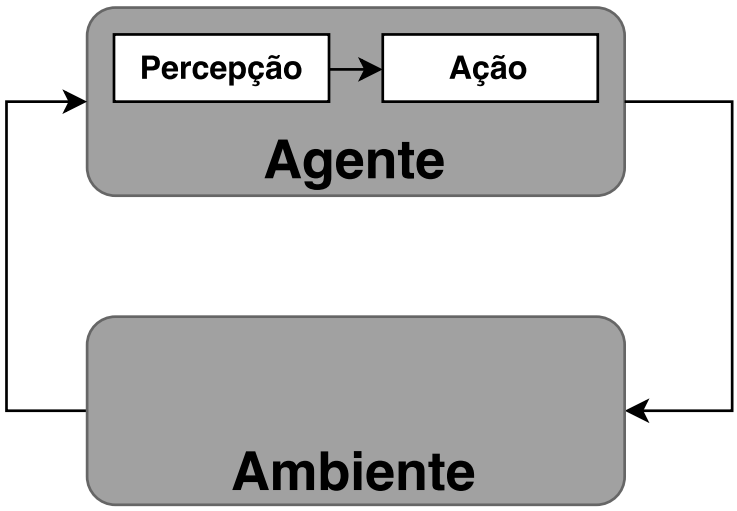


Figura 2.2 – Simple reflex agents

Essa arquitetura pode ser usado em ambientes de alta complexidade, como por exemplo, em carros que são dirigidos por agentes, quando o carro da frente freia, o agente deve frear também para não ocasionar em uma batida. Esse agente utiliza uma regra de ação condicional, se acontecer uma freada no carro da frente então eu devo frear também.

Este tipo de agente tem a propriedade de ser simples, mas com isso sua inteligencia fica limitada [10]. Essa abordagem é eficaz somente se o ambiente for completamente observável, ou seja, os sensores do agente tem acesso a todas as informações do ambiente a qualquer instante de tempo para detectar aspectos que são relevantes para a escolha das ações, por exemplo, um agente precisa se locomover de uma cidade para a outra, se o ambiente é completamente observável o agente consegue ver todas as estradas que chegam a cidade destino, já se o ambiente for parcialmente observável, pode ter alguma estrada que o agente não consiga ver, e se o agente não consegue ver nenhuma estrada o ambiente é não observável [10].

2.1.2 Model-based reflex agents

Os agentes de reflexos baseados em modelo utilizam um estado interno para marcar qual o estado do ambiente ele está. Este estado é utilizado como parte do processo de escolha da ação. A informação do estado pode ser de alguma informação que não pode ser obtida por alguma percepção do ambiente ou de estados que já foram visitados pelo agente. Este tipo de abordagem é eficaz para ambientes parcialmente observáveis, pelo fato de que o estado pode guardar informações relevantes para o agente [10]. Um modelo deste tipo de agente pode ser visto na figura 2.3.

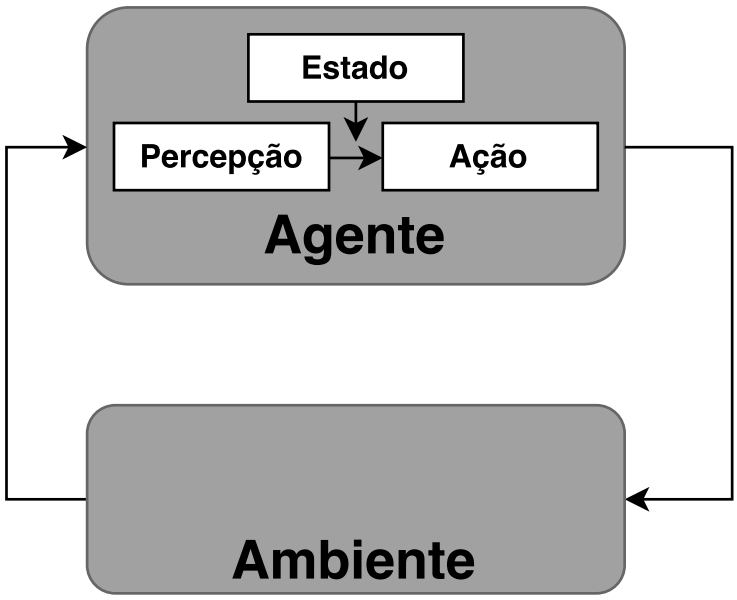


Figura 2.3 – Model-based reflex agents

2.1.3 Goal-based agents

Conhecendo o estado atual do ambiente, as vezes, não é suficiente para saber o que fazer [10]. Além do estado atual, o agente pode precisar de uma informação para saber a onde ele quer chegar, ou seja, um objetivo para descrever o que o agente está buscando alcançar. O objetivo pode ser alcançado com uma ação, outras vezes é mais complicado e leva várias ações. Esta arquitetura é diferente das outras duas apresentadas, pelo fato de se preocupar com o futuro. Para achar a sequencia de ações que alcança o objetivo pode ser usado Busca ou Planejamento, que são sub áreas da IA que tem como objetivo achar a sequencia de ações que levem o agente para o objetivo [10]. A figura 2.4 mostra a arquitetura e como o objetivo influencia na escolha da ação.

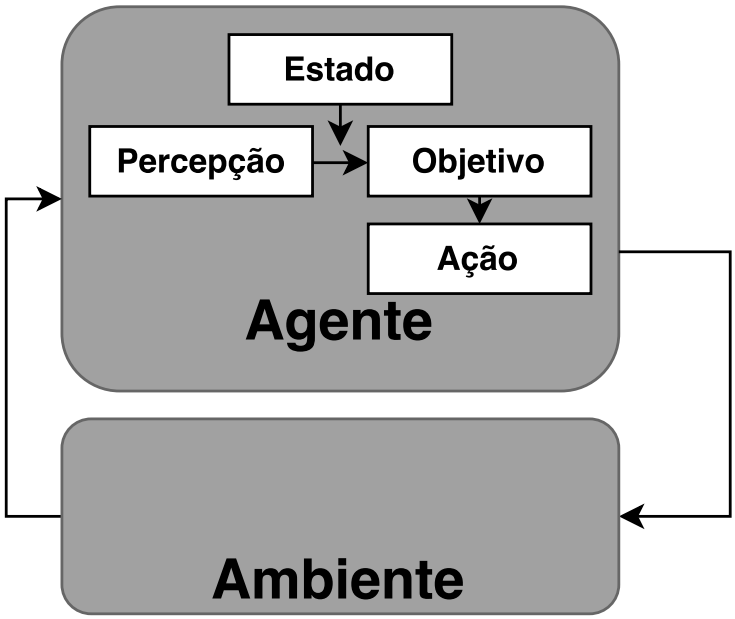


Figura 2.4 – Goal-based agents

2.1.4 Utility-based agents

Apenas alcançar um objetivo pode não ser suficiente para alguns cenários, como por exemplo, um agente que deseja chegar a determinada cidade, existem mais de um caminho que levam para a mesma cidade, indo pelo caminho mais longo o agente ainda estará chegando no objetivo. Para o agente conseguir alcançar o objetivo com uma melhor performance é utilizado uma função de utilidade, nela é medido o "desejo"do agente em tomar determinada ação. Cada ação exercida pelo agente terá influencia no valor de utilidade obtido [10]. Esta arquitetura é melhor utilizada em ambientes parcialmente observáveis e

estocásticos, um ambiente estocástico é onde não há conhecimento preciso do próximo estado dado determinada ação [10]. A figura 2.5 representa essa arquitetura.

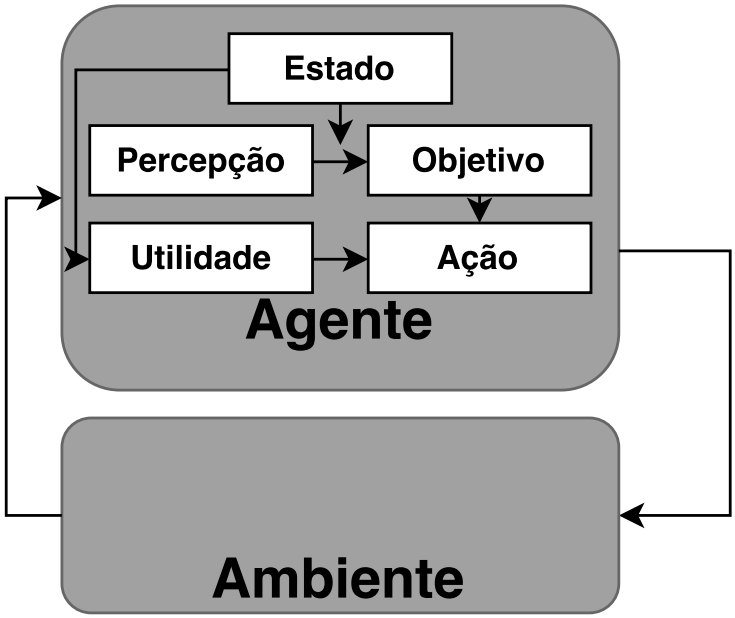


Figura 2.5 – Utility-based agents

3. BUSCA

Para encontrar uma sequência de ações que um agente baseado em objetivo consiga chegar ao seu objetivo é possível utilizar técnicas de Busca. O principio da busca é encontrar uma solução de um problemas através de um conjunto de ações que alcancem o objetivo desejado. Para utilizar as técnicas de busca é preciso formalizar o problema a ser resolvido e o objetivo a ser alcançado, pois na busca, como entrada é recebido um problema e é retornado uma solução na forma do conjunto de ações [10].

3.1 Problema de busca

Para entender melhor como funciona as técnicas de busca é preciso primeiro definir um problema. Um problema pode ser definido por cinco componentes [10]:

- S_0 - O estado inicial, que o agente começa no ambiente;
- Ações- Conjunto das possíveis ações presentes no agente;
- Resultado(s, a) - Um modelo de transição, que define o estado resultando após a execução da ação a no estado s;
- Objetivo(s) - Verifica se o estado é o objetivo do agente;
- Custo do caminho - Uma função que defina um valor numérico para cada ação realizada em um estado. Essa função pode ser denotada por $c(s, a, s')$, onde s é o estado atual do agente, a é a ação que será aplicada ao estado s e s' é o estado resultante aplicando o modelo de transição resultado(s, a).

Esses elementos definem um problema. Uma solução para um problema inicia no estado inicial, e através do modelo de transição utiliza ações para chegar ao objetivo do agente. Pelo fato de que as técnicas de busca aceitam mais de uma soluções para o mesmo objetivo, com conjuntos de ações diferentes, é utilizado para medir a qualidade da solução o custo do caminho, uma solução ótima é tida quando o menor custo do caminho de todas as soluções possíveis é encontrado utilizando o custo do caminho para cada ação realizada [10].

Para exemplificar um problema de busca, considere o mapa apresentado na figura 3.1, cada circulo representa uma cidade, para o exemplo considere o estado inicial como sendo a cidade de São Jerônimo, a ação disponível é se locomover entre as cidades que tenham ligação, o modelo de transição também são as ligações entre as cidades, o objetivo é chegar na cidade de Porto Alegre, e o custo de cada caminho está definido na transição.

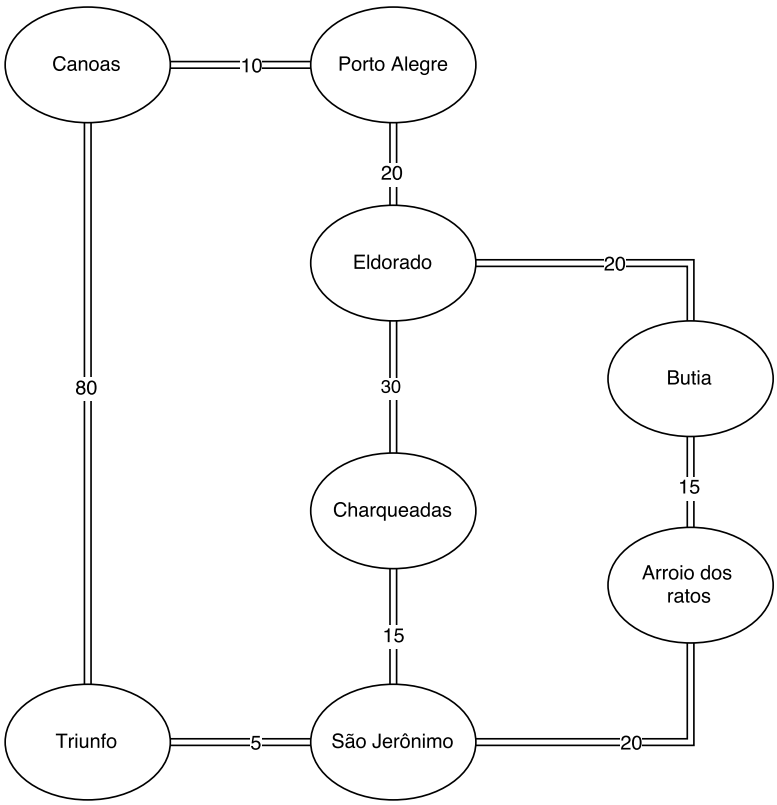


Figura 3.1 – Mapa para o exemplo de problema de busca

Para atingir o objetivo, utilizando uma busca, 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 de inicio, mas a técnica só terá como saber após realizar todas as possíveis opções de caminhos, ou se utilizar de funções heurísticas, que acrescentam um conhecimento adicional para a resolução do problema [10].

3.2 Busca adversaria

A busca adversaria é utilizada para ambientes competitivos, como nos jogos. Como em um jogo o jogador, preferencialmente, não informa suas jogadas previamente, o ambiente se torna imprevisível, e com isso os objetivos dos jogadores entram em conflito, ambos estão em busca da vitoria. Como solução para esse problema é preciso gerar uma solução de contingencia para tentar antecipar as jogadas do adversário. Jogos são difíceis de resolver com técnicas de IA, pois eles requerem uma habilidade de tomar alguma tipo de decisão, e as técnicas comuns as vezes não são satisfatórias, seja pelo fato dos estados que são possíveis de atingir ser muito grande, ou pelo curto espaço de tempo para tomar a decisão [10]. Para resolver esses problemas existem técnicas de busca adversaria para

ambientes competitivos. Os problemas de jogos utilizam uma variação da definição de um problema de busca, totalizando seis componentes, são eles [10]:

- S_0 - O estado inicial, que especifica como o jogo se configura no inicio.
- players(s) - Define qual jogador tem o movimento no estado.
- actions(s) - Conjunto das ações possíveis em um estado.
- result(s, a) - Um modelo de transição, que define o resultado da ação a aplicada ao estado s.
- terminal(s) - Verifica se o estado é um estado onde o jogo terminou.
- utility(s,p) - Define qual é o valor numérico para o jogo quando atingir um estado s terminal por um jogador p.

Com esses componentes descritos é possível entender alguma das técnicas de busca adversarias.

3.3 Minimax search

Para explicar como é resolvido um problema pelo algoritmo de *Minimax search*, primeiro é preciso considerar um jogo com dois jogadores, um é chamado de MAX e o outro de MIN. O jogador MAX representa o jogador que está tentando ganhar o jogo, e o jogador MIN é o jogador que está jogando contra. O jogo é alterna entre jogada de MIN e de MAX até o final do jogo, como em jogos por turnos. Quando são feitas jogadas que beneficiam o jogador MAX é obtido uma recompensa positiva e quando acontece ao contrario, o jogador MIN se beneficia da jogada, é obtido uma recompensa negativa [10].

O estado inicial, as ações e os resultados definem a arvore das jogadas para o jogo. A arvore representa em cada nodo um estado do jogo e cada ligação com os níveis de baixo são os estados resultantes após a execução de cada ação possíveis para o estado. A alternância entre as jogadas de MAX e MIN até chegar as folhas da arvore, que correspondem aos estados terminais. Como o ponto de vista é do MAX, o valor de cada nodo folha representa o valor de utilidade para o MAX, e os maiores valores representam bons resultados para o MAX e ruins para o MIN. Com isso o caminho resultante indica que aquela ação será a melhor ação para o estado atual [10]. Um exemplo de uma arvore pode ser visto na Figura 3.2, considerando os valores contidos nos nodos folhas o valor de utilidade do estado, a figura mostra que a técnica escolhe o melhor valor, neste caso o mais alto, para o jogador.

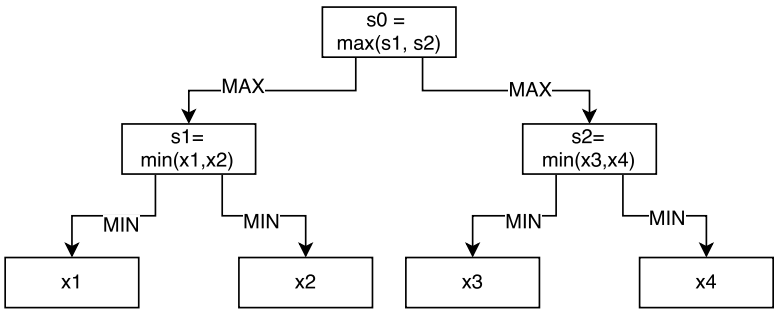


Figura 3.2 – Exemplo de GameTree

Este tipo de busca leva em consideração que o jogador adversário sempre realizará a jogada que mais lhe beneficiará. O algoritmo de *minimax search* pode ser visto no algoritmo 3.1, ele retorna a jogada que tem a melhor chance de resultar em uma vitoria no estado atual, do ponto de vista do jogador MAX [10].

Algoritmo 3.1 – Minimax Search

```
function MINIMAX(state)
    return argmax_{action∈actions(s)} min_value(result(state, action))
end function

function MAX_VALUE(state)
    if terminal(state) then
        return utility(state)
    end if
    v = −∞
    for all action ∈ actions(state) do
        v = max(v, min_value(result(state, action)))
    end for
end function

function MIN_VALUE(state)
    if terminal(state) then
        return utility(state)
    end if
    v = ∞
    for all action ∈ actions(state) do
        v = min(v, max_value(result(state, action)))
    end for
end function
```

O algoritmo sempre acha uma solução e ela é sempre a solução que, dentro das jogadas possíveis, é a melhor para o jogador [10].

4. PLANEJAMENTO AUTOMATIZADO

Planejamento automatizado é uma sub-área da inteligencia artificial que estuda o processo de geração automática de planos. O processo para gerar um plano, é feito a partir da escolha e organização das ações, antecipando os resultados esperados das ações, em busca do seu objetivo. Um plano pode ser descrito como uma sequencia de ações que se forem executadas chegam a um objetivo [5]. O planejamento na computação se diferencia das outras áreas pelo fato de que todo o plano é gerado automaticamente [10].

4.1 Representação do problema

Uma entrada para qualquer técnica de planejamento é uma descrição do problema a ser resolvido. Isso é necessário para não precisar representar todos os estados e transição do problema. Com a descrição do problema os estados e transição não precisam ser explícitos, mas faz com que os estados não descritos possam ser computados [5]. Como foi visto no capitulo 2, os estados são os estados para o agente se situar no ambiente e transições são para a interação com o ambiente. Um problema de planejamento pode ser descrito como $P = (\Sigma, s_0, g)$. Onde [5]:

- Σ- é a representação do problema;
- s_0- é o estado inicial, estado onde o problema começa;
- g- é o objetivo, estado onde o problema deve acabar.

Para representar o problema é necessário representar os estados do ambiente. Uma forma de fazer isso é utilizando lógica matemática. Sendo assim um estado do ambiente é representados por um conjunto de átomos que resultam em verdadeiro ou falso dependendo da interpretação do ambiente [5]. Vamos lembrar do problema do Capitulo 3, chegar a uma determinada cidade, estados para esse problema podem ser estar em determinada cidade e ter ligação entre as cidades, representado respectivamente como *estar(cidade) terLigação(cidadeOrigem, cidadeDestino)*.

Além dos estados do ambiente precisamos determinar as transições, que alteram os estados. As transições utilizam ações e são representadas por operadores de planejamento que alteram os valores dos átomos presentes em determinado estado. Um operador de planejamento é definido como *op* = (nome(*op*), precondições(*op*), efeitos(*op*)), onde cada elemento é definido como [5]:

- nome(*op*) - É o nome do operador de planejamento e *op* é o conjunto de todas as variáveis que irão aparecer qualquer parte do operador de planejamento.

- precondições(*op*) - *op* é o conjunto de átomos ou átomos negativos que representa a precondição do operador de planejamento.
- efeitos(*op*) - *op* é o conjunto de átomos ou átomos negativos que representa o efeito do operador de planejamento

No nosso exemplo, um operador de planejamento seria a mudança de uma cidade A para outra cidade B, que poderia ser representado como:

- nome- mudarDeCidade(cidadeA, cidadeB);
- precondições- estar(cidadeA) $\wedge$ terLigação(cidadeA, cidadeB);
- efeitos- $\neg$ estar(cidadeA) estar(cidadeB).

O nome deve ser único pelo proposito de o nome poder se referir ao operador por completo, ou seja, após definido apenas com o nome pode-se inferir as pré e pós condições, assim escrevendo o nome(*op*) para se referir a todo o operador de planejamento *op* [5].

O processo de geração do plano é feito pelo planejador, para isso ele utiliza a representação do problema, o estado inicial e os objetivos como entrada [5]. A figura 4.1 representa esse processo.

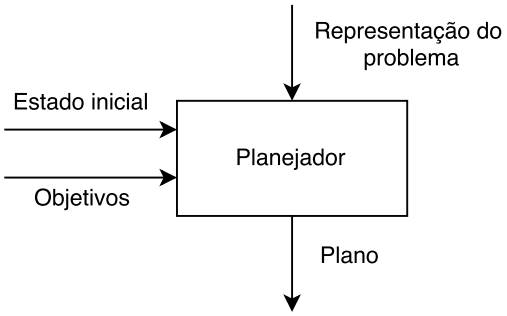


Figura 4.1 – Problema de planejamento

O domínio do problema é a parte do mundo que é expressada pelo conjunto de informações que é usado pelo planejador para gerar o plano [10]. No nosso exemplo, se representássemos as cidades presentes no mapa, apenas aquelas cidades seriam nosso domínio, e a única ação disponível no domínio seria mudar de cidade.

4.2 HTN

Dentro da área de planejamento existe o planejamento hierárquico, chamado de *Hierarchical Task Network* (HTN). Em planejamento HTN as ações são tratadas em mais

alto nível [10]. Essa abordagem de planejamento cria planos decompondo tarefas em tarefas ainda menores [8].

Uma tarefa t é uma representação de uma atividade no ambiente. Existem dois tipos de tarefa [10]:

- Tarefas primitivas- Uma atividade que um agente possa executar diretamente no ambiente.
- Tarefas não-primitivas- Representa objetivos que o agente deve alcançar antes que ele possa executar-las, ou seja, tarefas que devem ser decompostas em tarefas menores até todas serem tarefas primitivas.

Para decompor as tarefas não-primitivas em primitivas é necessário um conjunto de métodos [5]. Um método é composto por $m = (t, C, w)$. Um método m representa uma maneira na qual pode-se decompor as tarefas t em um conjunto de sub-tarefas w se todas as pre-condições C forem satisfeitas [8].

O planejamento HTN N é feito decompondo tarefas não primitivas recursivamente até chegar em tarefas primitivas [5]. N é uma arvore, na qual os nodos são tarefas ou métodos. Cada tarefa não-primitiva pode ter apenas um filho, que deve ser um método. Um método tem um filho para cada uma das tarefas em w. Tarefas primitivas não podem ter filhos. Uma arvore totalmente decomposta, é onde todas as folhas de N são tarefas primitivas [8].

Um problema de planejamento HTN pode ser descrito como, $P = (S, A, \gamma, T, M, s_0, N_0)$. S é o conjunto dos possíveis estados do ambiente. A é o cojunto finitio das ações que podem ser executadas pelo agente. γ é a função de transição que define os efeitos de cada ação no ambiente. T é o cojunto de tarefas. M é o conjunto de métodos. $s_0 \in S$ é o estado inicial do agente. n_0 é estado inicial do ambiente, que servem para definir os objetivos do agente. O proposito do planejamento HTN é achar uma arvore N totalmente decomposta que tenha como estado inicial da arvore n_0 e estado inicial do agente s_0 [8].

A ideia do planejamento HTN é através desta descrição do problema de planejamento HTN, decompor o estado inicial e continuar decompondo as tarefas que restarem pelo conjunto de métodos, até só restarem tarefas primitivas. O plano é composto apenas por tarefas primitivas. Na busca pelo plano, o planejamento HTN começa planejando por um caminho, quando um caminho de resolução leva a um fim de linha é realizado um retrocesso(*backtracking*) até um caminho que tenha uma possibilidade diferente de caminho do que foi tomado anteriormente [10].

4.3 AHTN

Adversarial hierarchical-task network (AHTN) é um algoritmo proposto para tentar solucionar o problema do grande fator de ramificação dos jogos em tempo real [8]. Nele são combinados técnicas de HTN com o algoritmo *minimax search*. O algoritmo assume jogos totalmente observáveis, baseados em turno e determinísticos.

O algoritmo 4.1 [8] é a representação da técnica de AHTN, e assume que existem dois jogadores, MAX e MIN, como no algoritmo de *minimax search* apresentado no capitulo 3. O algoritmo também assume uma busca em uma arvore com uma máxima profundidade d.

Algoritmo 4.1 – AHTN

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

Cada nodo da arvore 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 executado, sendo t_+ para uma tarefa de N_+ e t_- para uma tarefa de N_-. Na raiz da arvore $t_+ = \perp$ e $t_- = \perp$ indicam que nenhuma ação foi executada ainda [8].

A função *nextAction(N,t)* faz com que, dado um HTN N e um ponteiro t, seja encontrada a tarefa primitiva que deve ser executada em N depois da tarefa t. Se $t = \perp$ então é retornado a primeira tarefa primitiva a ser executada em N. Se N ainda não estiver completamente decomposto, ou seja, ainda existem tarefas não primitivas, e existe nenhum tarefa primitiva em N então *nextAction(N,t)* = $\perp$ [8].

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. Formalmente: $nextAction(N_+, t_+) = \perp$, ou $s_0 = \gamma(s, nextAction(N_+, t_+)) \neq \perp$ e o nodo MIN $n_0 = (s_0, N_+, N_-, nextAction(N_+, t_+), t_-)$ seja consistente. A definição de consistência de MIN é análoga [8].

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_+ $(decompositions(N_+, t, m) | m \in applicable(N_+, t))$. $decompositions_-$ é análogo [8].

A partir de uma função de avaliação e, que quando aplicada sobre um estado $s \in S$, retorna a recompensa de max em s se for um estado terminal ou uma aproximação se s for um estado não terminal. A partir destas definições, o algoritmo para AHTNMin é análogo. O algoritmo retorna o melhor plano encontrado para os dois jogadores, e também o resultado da função de avaliação no nodo terminal alcançado após a execução dos planos [8].

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 [8].

A imagem 4.2 mostra uma arvore gerada pelo algoritmo 4.1 com profundidade $d = 2$. Na raiz da arvore pode ser visto que, para os dois jogadores, apenas uma tarefa não primitiva *win* que precisa ser decomposta. Há duas decomposições que o jogador MAX pode aplicar para sua HTN, resultando nos nodes n_1 e n_5. A decomposição de n1 não resulta em nenhuma ação primitiva, e por isso n_1 continua um nodo MAX. Uma vez que o jogador MAX decompõe sua HTN para o ponto onde a primeira ação pode ser gerada(nodo n_2 e n_5), é o turno de MIN de decompor. 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 os estados do jogo para determina o valor das folhas.

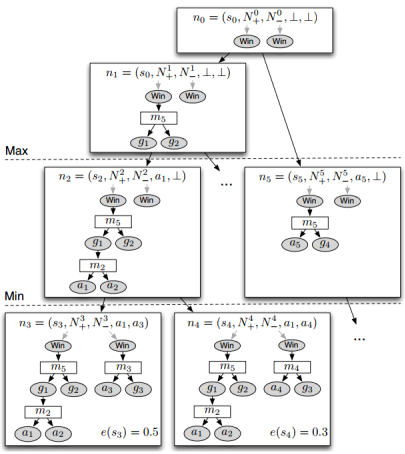


Figura 4.2 – Arvore gerada pelo algoritmo de AHTN

5. APRENDIZADO

Para os humanos o aprendizado ocorre durante toda a vida. O aprendizado é o ato de adquirir novos conhecimentos, ou modificar conhecimentos já existentes ou ainda adquirir uma experiencia por repetição do ato de forma incorreta. Aprendizado pode variar de adquirir conhecimento de tarefas simples, como decorando um numero de telefone, até tarefas mais complicadas, como a formulação de novas teorias [10].

5.1 Aprendizado de Máquina

A área na computação que estudo esse aprendizado de forma computacional é o aprendizado de maquina, melhor conhecida como *machine learning*. A definição de aprendizado de maquina proposta por Tom Mitchell [4] é a seguinte:

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

Essa definição mostra que o sistema aprimora seu conjunto de tarefas T com uma performance P através de experiencias E. Ou seja, um sistema baseado em aprendizado de maquina deve, através de experiencias, ter um ganho nas informações para solucionar os seus problemas. Para começar a resolver um problema utilizando aprendizado de maquina é preciso escolher qual experiencia será aprendida pelo sistema [4]. Para isso existem algumas técnicas que tratam aprendizado de maquina com objetivos diferentes [10]. Alguma das técnicas são:

- Aprendizado supervisionado: Aprender através de algum conjunto de exemplos a realizar a classificação de algum problema. Cada problema é mapeado para uma saída.
- Aprendizado não supervisionado: Aprender através das observações, algum pedrão ou regularidade, para classificar em grupos os problemas.
- Aprendizado por reforço: Aprender através das execuções, bem ou mal sucedidas. Aprende quais ações são melhores de ser executadas.

Cada tipo de aprendizado é utilizado pode ser usado para uma aplicação especifica, mas existem casos em que a combinação das técnicas se mostra mais eficaz. Um exemplo apresentado por [10] é o reconhecimento de idade por fotos, para essa tarefa são necessários amostras de fotos com as idades, então a técnica que se encaixa é aprendizado supervisionado, mas existem ruídos aleatórios nas imagens que fazem com que a

precisão da abordagem caia, para superar esse problema pode ser combinado aprendizado supervisionado com o não supervisionado.

5.2 Aprendizado por Reforço

O aprendizado por reforço também é conhecido como *reinforcement learning*. Este tipo de aprendizado utiliza *feedbacks*, vindas do ambiente após a sua execução, esse tipo de *feedback*, é chamado de recompensa. O objetivo deste aprendizado é usar as recompensas obtidas nas observações para aprender uma politica do ambiente ou determinar o quão boa a politica é [10].

Em jogos *reinforcement learning* é um tópico que é bastante utilizado [3]. Em um jogo essa técnica utiliza três etapas, uma para exploração da estrategia para achar as diferentes ações possíveis no jogo, uma função que disponibiliza o *feedback* e diz o quão bom é cada ação, e uma regra de aprendizado que junta as outras duas etapas [3].

Existem dois tipos principais de aprendizado por reforço [10]:

- Aprendizado Passivo
- Aprendizado Ativo

5.2.1 Aprendizado passivo

O aprendizado passivo utiliza ambientes completamente observáveis. A politica do agente π é fixa, em um estado s, sempre é executado a mesma ação $\pi(s)$. O objetivo desse tipo de aprendizado é aprender o quão bom é a politica, o que significa aprender a função de utilidade $U^{\pi}(s)$. Um agente que utiliza aprendizado passivo não conhece o modelo de transição $P(s'|s,a)$, que especifica a probabilidade de alcança o estado s' a partir do estado s executando a ação a, e também não conhece a função de recompensa $R(s)$, que especifica a recompensa de cada estado [10].

Um agente que utiliza essa técnica realiza várias execuções do ambiente utilizando a politica π. Em cada tentativa o agente inicia no mesmo estado inicial e realiza uma sequencia de transições de estados ate chegar a um estado terminal. As percepções obtidas com essas execuções, em cada estado, servem para descobrir a recompensa obtida nos estados. O objetivo é utilizar a informação das recompensas para aprender a utilidade esperada $U^{\pi}(s)$ associada a cada estado s não terminal [10].

A utilidade é definida para ser a soma esperada das recompensas obtidas seguindo a politica π. Podendo ser representado pela equação 5.1, onde $R(s)$ é a recom-

pensa do estado s, S_t é o estado alcançado no tempo t quando executado a politica π e $S_0 = s$. É preciso de um fator de desconto γ para descrever a performasse do agente para as recompensas atuais sobre as futuras, este fator é um numero entre 0 e 1, quando 1 as recompensas são exatamente equivalentes as recompensas futuras. [10].

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right] \tag{5.1}$$

5.2.2 Aprendizado ativo

O aprendizado ativo diferente do passivo não tem uma politica fixa e a mesma deve ser aprendida. Para isso, um agente que utiliza este tipo de aprendizado precisa decidir quais ações tomar, isso faz com que o agente precise aprender o modelo de transição $P(s'|s,a)$ para cada um dos estados e ações [10].

Um método para conseguir definir a politica do ambiente é chamado de *Q-learning*. O objetivo desse método é aprender uma utilidade ligada a um par de estado e ação, a notação $Q(s,a)$, representa o valor de executar a ação a no estado s. Este método está relacionado com o valor de utilidade presente na equação 5.2 [10].

$$U(s) = max_a Q(s,a) \tag{5.2}$$

Como o *Q-learning* não precisa do modelo de transição $P(s'|s,a)$ ele é chamado de um método livre de modelo. A equação 5.3 representa o calculo do valor de $Q(s,a)$. O algoritmo 5.1 apresenta o método de *Q-learning* para um agente [10].

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

Algoritmo 5.1 – Q-Learning

function Q-LEARNING(*state*, *reward*)
  if *terminal*(*state*) **then**
    return $Q[s, None] = reward$
  end if
  if *state* is not null **then**
    increment $N[s,a]$
    $Q[s,a] = Q[s,a] + \alpha(N[s,a])(r + \gamma max_{a'} Q[s',a'] - Q[s,a])$
    s = s'
    a = $argmax_{a'} f(Q[s',a'], N[s',a'])$
    r = r'
  end if
end function

6. OBJETIVOS

Atualmente, há uma dificuldade na utilização de técnicas de IA para jogos em tempo real. Isso ocorre pelo fato que a IA precisa ser capaz de resolver tarefas, do mundo real, de maneira rápida e satisfatória. Geralmente, o espaço de estados dos jogos é enorme, isso faz com que não se tenha tempo de explorar todas as possibilidades de solução [3].

6.1 Objetivo Geral

O objetivo geral deste trabalho é implementar o algoritmo de planejamento AHTN [8] 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 [6, 2, 9].

6.2 Objetivos Específicos

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

6.2.1 Objetivos fundamentais

- Modelar o domínio do problema de planejamento.
- Implementar o algoritmo de AHTN no ambiente do jogo MicroRTS.
- Executar e testar a implementação para encontrar eventuais falhas.
- Ajustar a implementação para corrigir possíveis erros.
- Criação de *benchmark* para comparações.
- Comparar os resultados obtidos com as implementados já embutidas no MicroRTS.

6.2.2 Objetivos desejaveis

- Integrar um algoritmo de aprendizado por reforço a técnica de AHTN.
- Comprar os resultados obtidos com a implementação do AHTN puro.
- Enviar a implementação para o autor do MicroRTS como uma sugestão de IA para o jogo.
- Escrever um artigo para um *workshop* mostrando a unificação das técnicas de planejamento com aprendizado por reforço.

7. ANÁLISE E PROJETO

7.1 Jogos

Jogos eletrônicos são muito populares, principalmente pela grande quantidade de gêneros, existem jogos de ação, aventura, esportes, estrategia, 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 [3].

7.2 Jogos de estratégia em tempo real

Jogos de estratégia em tempo real, também conhecido por *real-time strategy games* (RTS), é um sub-gênero de jogos de estrategia, 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 [9, 1].

Existem algumas diferenças entre jogos RTS e jogos de tabuleiro, como xadrez. Estas diferenças são [9]:

- 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;
- 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 [9].

7.3 MicroRTS

Um exemplo deste gênero é o MicroRTS[1], uma simplificação do jogo Starcraft[2], feita por Santiago Ontañón [7] 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.

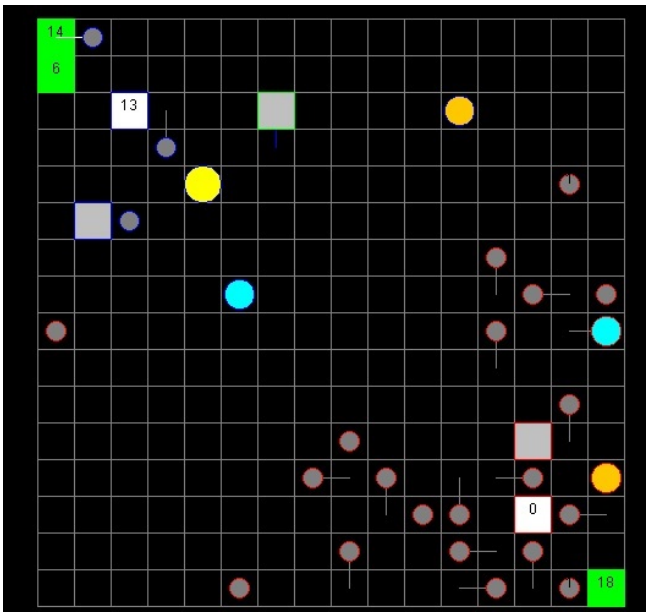


Figura 7.1 – Um exemplo de tela do MicroRTS

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

- Worker- É responsável por coletar recursos e construir as edificações. Esta unidade também consegue lutar, mas possui um dano muito baixo.
- Heavy- Unidade que pode apenas atacar. Ela possui um alto poder de ataque, mas sua velocidade é lenta.
- Light- Unidade que pode apenas atacar. Ela possui um baixo poder de ataque, mas sua velocidade é rápida.
- Ranged- Unidade que pode apenas atacar. Ela possui um ataque de longa distancia.

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

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

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

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

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

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

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

7.4 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 Inteligencia Artificial, onde pode ser acoplado a IA que desejar, a plataforma já possui algumas como foi dito anteriormente. A imagem 7.2 representa os componentes.

Para realizar a implementação de uma IA para acoplar no MicroRTS, é preciso conhecer as classes responsáveis por cada componente. A imagem 7.3 apresenta, as classes principais. A Classe *GameVisualSimulation* é uma *facade* que provê uma interface para a simulação dos outros componentes do jogo. A classe *GameState* e *PhysicalGameState*, são responsaveis pelo controle das ações das unidades, e pelo controle do mapa e das unidades dentro dele, respectivamente. A classe *UnitTypeTable* é onde cada unidade é associada as ações possíveis no jogo. A Classse *PhysicalGameStatePanel* é responsável pela interface gráfica. E por fim a classe *AHTN* é onde minha proposta de solução será acoplada ao jogo.

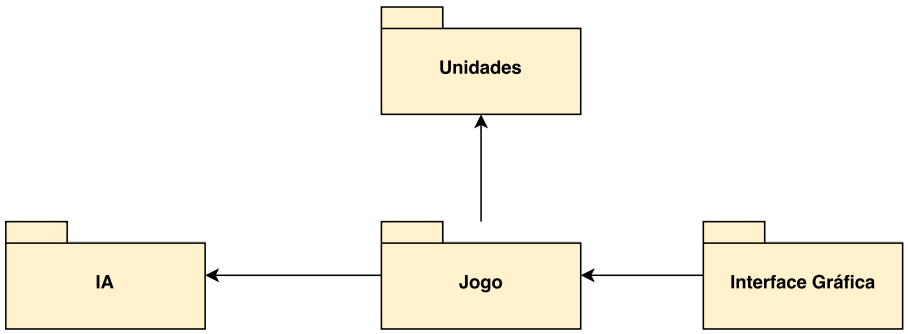


Figura 7.2 – Arquitetura MicroRTS

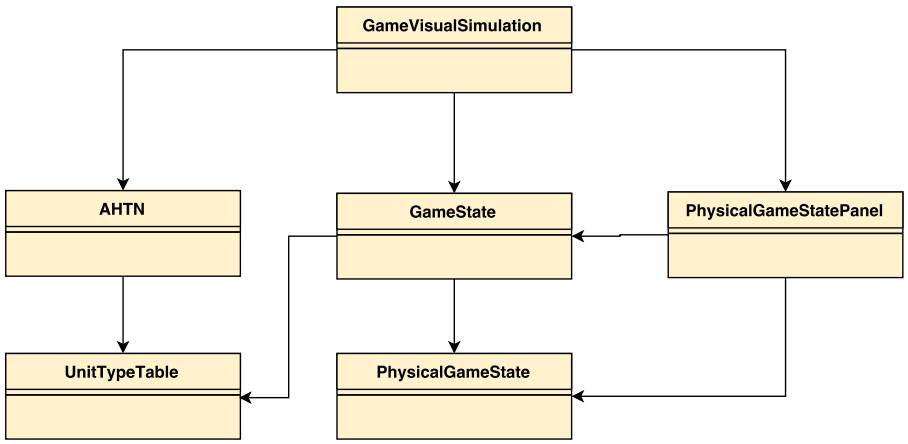
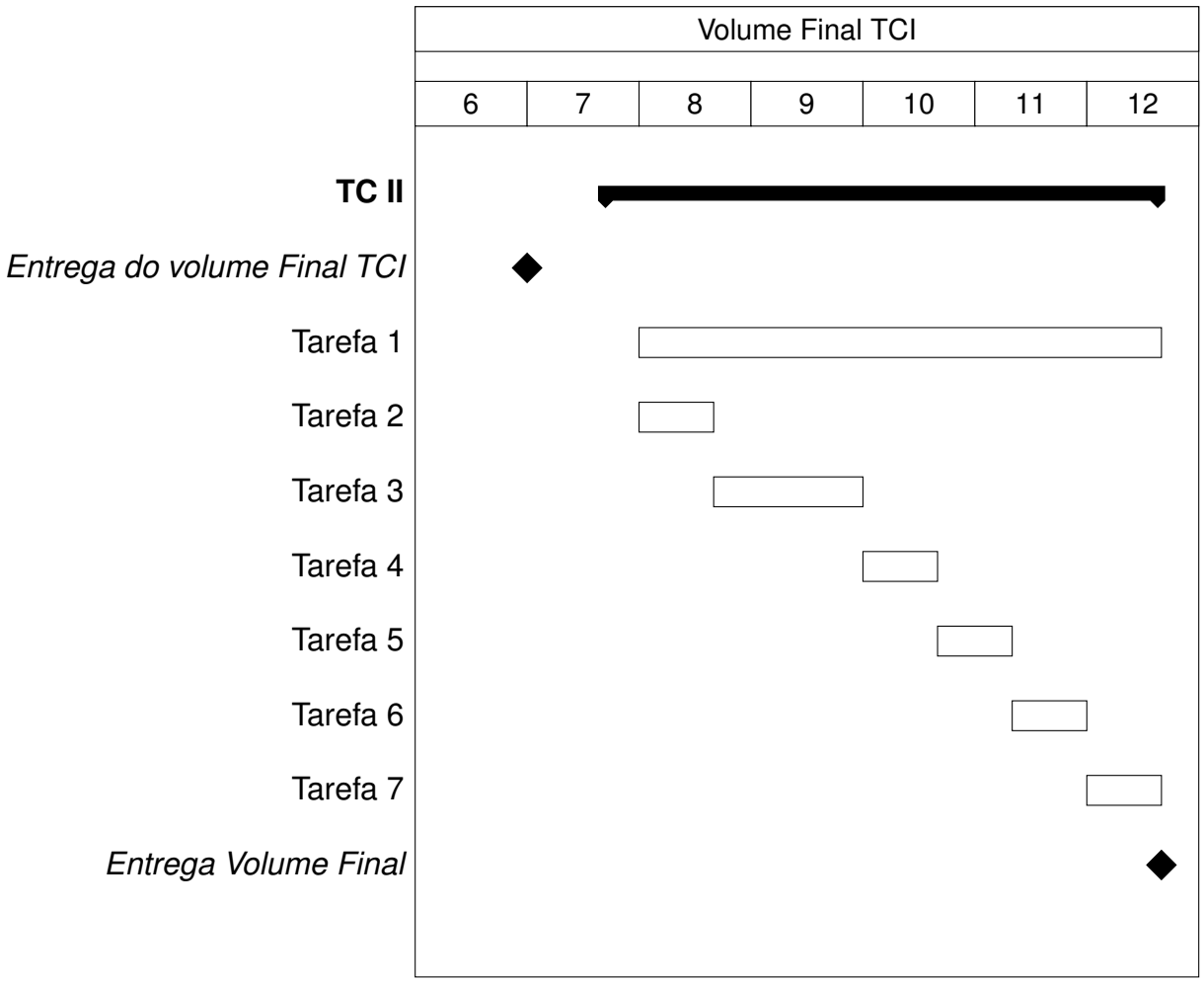


Figura 7.3 – Classes do MicroRTS

7.5 Cromograma

Para está etapa do projeto do Trabalho de Conclusão, foi proposto o plano das atividades apresentado na diagrama de Gantt com a respectiva legenda.

- Tarefa 1 - Escrita do TC II.
- Tarefa 2 - Desenvolvimento do dominio.
- Tarefa 3 - Implementação do algoritmo AHTN.
- Tarefa 4 - Criação dos *benchmark*.
- Tarefa 5 - Integrar o algoritmo de *q-learning* a implementação.
- Tarefa 6 - Realizar a comparação com as outras abordagens.
- Tarefa 7 - Preparar a apresentação.


Volume Final TCI
6
7
8
9
10
11
12
TC II
Entrega do volume Final TCI
Tarefa 1
Tarefa 2
Tarefa 3
Tarefa 4
Tarefa 5
Tarefa 6
Tarefa 7
Entrega Volume Final

REFERÊNCIAS BIBLIOGRÁFICAS

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

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

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

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