UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Antonio Luis Cardoso Silva

Técnicas de aprendizado de máquina aplicadas em Jogos RTS

Belo Horizonte
2020

Antonio Luis Cardoso Silva

Técnicas de aprendizado de máquina aplicadas em Jogos RTS

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Luiz Chaimowicz

Belo Horizonte
2020

© 2020, Antonio Luis Cardoso Silva.
Todos os direitos reservados

Silva, Antonio Luis Cardoso.

S586t Técnicas de aprendizado de máquina aplicadas em Jogos RTS [manuscrito] / Antonio Luis Cardoso Silva — 2020.
71 f. il.

Orientador: Luiz Chaimowicz
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação
Referências: f. 54-67

1. Computação – Teses. 2. Jogos digitais – Teses. 3. Inteligencia artificial – Teses. 4. Redes neurais – Teses. I. Chaimowicz, Luiz. II.Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV.Título.

CDU 519.6*82 (043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região º 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

TÉCNICAS DE APRENDIZADO DE MÁQUINA APLICADAS EM JOGOS RTS

ANTONIO LUIS CARDOSO SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. GISELE LOBO PAPPA
Departamento de Ciência da Computação - UFMG

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 4 de Dezembro de 2020.

Resumo

O domínio de Jogos de Estratégia em Tempo Real ou RTS apresenta grandes desafios para a área de inteligência artificial por ser um ambiente dinâmico, em tempo real, adversarial e incerto. Um método para abordar esses desafios é através do uso de algoritmos de aprendizagem de máquina onde o agente inteligente aprende utilizando dados de partidas jogadas por humanos. Nesse trabalho foram implementadas diversas técnicas de aprendizado na criação de um agente capaz de jogar Starcraft e predizer o resultado da partida.

Para cuidar da parte estratégica e econômica da partida pelo agente inteligente foi utilizado planejamento online baseado em casos. Já para cuidar da parte de combate foi utilizado mapas de influência.Por fim, para a predição de resultados foram utilizadas redes neurais recorrentes. Para isso, também foi utilizada a base de dados STARDATA que possui informações de mais de 65000 partidas de Starcraft. O módulo de predição de partida conseguiu obter uma precisão entre 67% e 86% de acordo com o tempo de partida. Além disso, os módulos estratégicos, econômicos e de combate obteve melhores resultados que os respectivos trabalhos que serviram de referência. Já o agente inteligente competiu contra outros agentes da competição AIIDE 2017 e observou-se que ele consegue adaptar a diferentes situações do jogo.

Palavras-chave: Inteligência Artificial, Mineração de Dados, Jogos

Abstract

The Real Time Strategy Games (RTS) domain presents great challenges for the artificial intelligence area because it is a dynamic, real time, adversarial and uncertain environment. One method for addressing these challenges is through the use of machine learning algorithms where an intelligent agent learns using data of games played by humans. In this work, several machine learning techniques were implemented in the creation of an agent capable of playing Starcraft and predicting the outcome of the match.

To take care of the strategic and economic part of the game by the agent, online case-based planning was used. To take care of the combat module, influence maps were used. To predict the match result we used recurrent neural networks. The STARDATA database was also used, which contains information on more than 6500 Starcraft games. The prediction module was able to obtain an accuracy between 63% and 80% according to the game time. Also, the strategic, economic and combat modules were more accurate than the works we used as reference. The intelligent agent competed against other agents in the AIIDE 2017 competition and it was observed that it manages to adapt to different situations in the game.

Keywords: Artificial Intelligence, Data Mining, Games

Lista de Figuras

3.1 Interface do jogo Starcraft: Broodwar. 20
3.2 Ciclo de um sistema RBC. 23
3.3 Ciclo do planejamento online baseado em casos (POBC) 28
3.4 Topologia de uma rede neural recorrente em uma camada escondida. 30

4.1 Ontologia de objetivos para ações em Starcraft 37
4.2 Exemplo de um episódio e um trecho de plano. 38
4.3 Representação de um plano expandido. 40
4.4 Exemplo de um plano para o treinamento de uma unidade do tipo *Marine* mostrando as macro ações e pré-condições. 43
4.5 Resumo do método proposto. 44
4.6 Mapa de influência criado pela a unidade Marine 45
4.7 Mapa de influência criado pela a unidade *Siege Tank* em *Siege Mode*. 46
4.8 Uma batalha entre Marines e Zealots e o mapa de influência correspondente. . 46
4.9 Mapa de influência criado pela a unidade *Siege Tank* em *Siege Mode*. 50
4.10 Ciclo de execução do UFMGBot 50

5.1 Comparação entre o erro de treino e teste para a RNR simples. 54
5.2 Comparação entre os recursos coletado pelo bot criado contra o ZZZKBOT . . 56
5.3 Comparação entre a quantidade de unidades criadas em partida do bot criado contra ZZZKBOT 57
5.4 Comparação entre a quantidade de construções criadas em uma partida do bot criado contra ZZZKBOT 57
5.5 Comparação entre os recursos em partida do bot criado contra o UAlbertaBot 58
5.6 Comparação entre a quantidade de undiades criadas em partida do bot criado contra UAlbertaBot 58
5.7 Comparação entre a quantidade de construções criadas em uma partida do bot criado contra UAlbertaBot 59

A.1 Ciclo de execução do UFMGBot 64
A.2 Representação de um plano expandido. 65
A.3 Exemplo de um episódio e um trecho de plano. 65
A.4 Uma batalha entre Marines e Zealots e o mapa de influência correspondente. . 66
A.5 Construção dos 2 Gateways na partida 66

A.6 Uma batalha entre Dragoons, Tanks e um bunker e o mapa de influência correspondente. 67

Lista de Tabelas

4.1 Eventos extraídos da base de dados. 44
4.2 Número de jogos por confronto na base Stardata. Legenda: P = Protoss, T = Terran, Z = Zerg 49

5.1 Comparação da taxa de vitória entre os dois agentes. 51
5.2 Comparação da taxa de vitória para diferentes números de partidas de demonstração. 52
5.3 Resultados comparando diferentes RNRs para o intervalo de tempo entre 3 e 6 54
5.4 resultado obtido para diferentes intervalos de tempo usando uma RNR simples 54
5.5 Taxa de vitória contra bots da competição AIIDE 2017. 55

Sumário

1 Introdução **10**
- 1.1 Motivação 10
- 1.2 Desafios em Jogos RTS 12
- 1.3 Objetivos 13
- 1.4 Estrutura do Trabalho 14

2 Trabalhos Relacionados **15**
- 2.1 Agentes inteligentes em jogos RTS 15
- 2.2 Técnicas de aprendizado por imitação 17

3 Referencial Teórico **19**
- 3.1 Jogos RTS 19
- 3.2 Raciocínio baseado em Casos 22
- 3.3 Planejamento online baseado em casos 27
- 3.4 Redes Neurais Recorrentes 29

4 Metodologia **32**
- 4.1 Planejamento Online Baseado em Casos em Jogos de Estratégia em Tempo Real 32
- 4.2 POBC para Starcraft 33
- 4.3 Sistema de combate usando mapas de influência 43
- 4.4 Predição de resultados usando Redes Neurais Recorrentes 48

5 Experimentos e Resultados **51**
- 5.1 Mapas de Influência 51
- 5.2 Planejamento Online Baseado em Casos 52
- 5.3 Predição do Resultado de Partidas 53
- 5.4 Avaliação do bot 55

6 Conclusão **60**

Referências Bibliográficas **61**

Apêndice A Exemplo de execução do UFMGBOT **64**

Capítulo 1

Introdução

Nessa seção serão introduzidos os conceitos básicos de jogos RTS, a definição do problema e a motivação e objetivos deste trabalho.

1.1 Motivação

Jogos há muito tempo vêm servindo como ambiente de pesquisa para novas técnicas de Inteligência Artificial (IA) devido à complexidade inerente nos mesmos, já que são criados para desafiar a mente humana, ao mesmo tempo que possuem regras bem definidas e que podem ser implementadas facilmente no computador. Com o avanço do poder de processamento dos computadores, os gráficos de jogos digitais se tornam cada vez mais realistas e com isso o comportamento dos personagens mostrados também tem que evoluir para evitar a quebra de imersão por parte do jogador.

A área de IA em jogos avançou muito durante os últimos 15 anos em que existe como um campo de pesquisa específico. Durante esse tempo, surgiram novos encontros anuais voltados especificamente nessa área como: IEEE Conference on Computational Intelligence and Games (CIG), recentemente renomeada para IEEE International Conference on Games (COG) e a conferência AAAI Artificial Inteligence and Interactive Digital Entertainment (AIIDE). Durante um seminário Dagstuhl em Inteligência Artificial e Computacional em Jogos realizado em 2012, foram definidas as principais áreas de pesquisa em IA para jogos [Yannakakis Togelius, 2015]. Sendo elas:

1. *nonplayer character (NPC) behavior learning*
2. *search and planning*
3. *player modeling*
4. *games as AI benchmarks*
5. *procedural content generation*

6. *computational narrative*
7. *believable agents*
8. *AI-assisted game design*
9. *General game AI*
10. *AI in commercial games*

A área de *believable agents* ou agentes críveis se preocupa em criar agentes com aspecto mais humano e com isso aprimorar o comportamento dos mesmos. Geralmente o personagem é um inimigo um aliado ou um personagem neutro (personagens que não podem ser controlados e têm como papel simplesmente a interatividade com o jogador). O jogo, tendo inimigos com comportamento mais humano, pode ser desafiante ao mesmo tempo que impede a sensação do jogador estar sendo "trapaceado" caso ele perca. Aliados com comportamento humano permitem que jogos multijogador utilizem bots quando jogadores reais não estão presentes. Já personagens neutros realistas permitem que a história seja contada sem a perda de imersão. Agentes que buscam simular o comportamento humano são chamados de agentes críveis.

Uma forma de se criar *believable agents* é através da utilização de técnicas de aprendizagem por imitação. Nesse tipo de aprendizado o comportamento desejado é conseguido se baseando em demonstrações de estratégias de especialistas para se atingir um objetivo. Essa estratégia é generalizada para estados não visitados antes. Geralmente o especialista é um humano e a demonstração pode vir de um ou mais especialistas [Osa et al., 2018].

Uma IA com comportamento mais humano pode ser utilizada em diversos gêneros de jogos, como: Jogos de Estratégia em Tempo Real (RTS) [Weber B.G et al., 2011], Tiro em Primeira Pessoa (FPS) [Hingston, 2009] e jogos de plataforma [Ortega et al., 2013]. O foco desse trabalho será em jogos RTS. Nesse tipo de jogo o objetivo é construir, posicionar e controlar unidades e estruturas durante a partida para assegurar áreas do mapa e destruir tropas e bases inimigas. Isso normalmente é limitado por algum recurso que deve ser obtido no mapa utilizando unidades e estruturas especializadas. As tarefas a serem realizadas durante o jogo exigem muito do jogador pois deve-se controlar a posição de dezenas de unidades em diferentes pontos do mapa para realizar tarefas como combater o inimigo, coletar recursos e construir estruturas.

Jogos RTS têm sido explorados por pesquisadores para a criação de novas técnicas de inteligência artificial por requerer planejamento de alto nível e a longo prazo ao mesmo tempo que requer o controle específico de unidades, chamados de gerenciamento macro e micro, respectivamente. Além disso o jogo é parcialmente observável e o número de estados e ações possíveis é muito grande para técnicas clássicas de IA poderem ser efetivas.

Starcraft é o jogo mais popular na pesquisa de IA para jogos RTS devido a API de Brood War (BWAPI) que permite código externo obter o estado do jogo e executar ações como se fosse um jogador em uma partida.

1.2 Desafios em Jogos RTS

Starcraft vem sendo utilizado como ambiente de pesquisa em Inteligência Artificial há bastante tempo pois apresenta desafios complexos que são de interesse da área. Problemas como os destacados em Buro [2004] e Buro Furtak [2003]:

- **Planejamento adversarial em tempo real**: Em jogos RTS o agente não pode perder muito tempo pensando em como executar as ações, devido ao ambiente ser dinâmico e imprevisível. Por isso uma abstração do estado do mundo deve ser encontrada para permitir que buscas pelas as ações sejam executadas de forma eficiente.

- **Tomada de decisão sob incerteza**: O agente não tem conhecimento da localização do inimigo no começo do jogo. Por isso estratégias para o reconhecimento do mapa devem ser executadas com a utilização de unidades específicas e caso não haja informação suficiente, hipóteses plausíveis devem ser formadas e deve-se agir de acordo com elas.

- **Aprendizado e modelagem do oponente**: Jogadores humanos sabem como detectar pontos vulneráveis no oponente e usar essa fraqueza no futuro. Muitas vezes sistemas de inteligência artificial têm dificuldade de realizar o mesmo.

- **Gerenciamento de recursos**: O jogador deve coletar recursos espalhados no mapa e deve usá-los para construir forças de ataque e defesa, melhorar a infantaria e se desenvolver tecnologicamente. Além disso, o jogador tem que saber balancear o uso de recursos entre essas categorias para sair vencedor no jogo. Por exemplo, o jogador pode usar todos seus recursos para construir um grande exército mas ele pode não ser suficiente para vencer o oponente apesar da vantagem numérica.

- **Combate**: Combate é parte fundamental desse tipo de jogo, seja para adquirir novos territórios ou para enfraquecer o exército adversário. Os jogadores devem controlar as unidades taticamente, buscando utilizá-las da melhor forma possível para vencer o oponente. Dois subproblemas em combates de jogos RTS são descritos em Laursen Nielsen [2005]:

- ⋆ **Posicionamento de unidades**: é a necessidade de movimentar e posicionar as unidades de tal forma que obtenha-se maior vantagem possível levando em consideração o terreno onde o combate ocorre e o posicionamento inimigo. Por exemplo, é preferível posicionar unidades de forma a controlar corredores pois obriga o inimigo a afunilar suas unidades e combater com poucas unidades por vez.
- ⋆ **Escolha de alvo**: Se refere a decisão de qual unidade adversária focar em combate de forma que o adversário seja derrotado mais rapidamente. Por exemplo, tanques possuem alto dano em área por isso é preferível que ataquem agrupamentos de inimigos.

Jogos RTS são complexos e possuem várias nuances que precisam ser endereçadas para a construção de agentes inteligentes (bots) eficientes, por isso há vários anos existem pesquisas em IA interessadas em resolver esses problemas.

1.3 Objetivos

Como mostrado na Seção 1.2, jogos RTS representam um grande desafio para o desenvolvimento de agentes inteligentes devido ao ambiente dinâmico, a incerteza sobre o ambiente e a velocidade com que as decisões têm que ser tomadas. Uma forma de desenvolver agentes inteligentes é utilizando técnicas de aprendizado por imitação, onde o objetivo é o agente aprender a executar uma tarefa através de demonstração. Nesse trabalho a demonstração que é dada para o agente aprender é vinda de uma base de dados que contém dados de partidas de jogadores humanos.

Objetivo Geral: A partir de uma base de dados, utilizar técnicas de aprendizado de máquina, principalmente raciocínio baseado em casos e mapas de influência para criação de um bot capaz de jogar Starcraft e para criar uma rede capaz de predizer o resultado da partida em diferentes intervalos de tempo.

Objetivos Específicos: Mais especificamente, pretende-se:

- Utilizar uma base de dados de partidas de Starcraft como a fonte de demonstração para as técnicas de aprendizado de máquina.
- Usar Planejamento online baseado em casos para cuidar da parte estratégica e econômica do bot.

- Usar uma base de dados com mapas de influência representando estados do jogo e a sequência de ações executadas a partir daquele estado para cuidar da parte de combate do bot.
- Avaliar a performance do bot desenvolvido realizando partidas contra outros bots.
- Criar uma rede neural recorrente capaz de predizer o resultado de uma partida à medida que ela acontece.

1.4 Estrutura do Trabalho

O trabalho está organizado da seguinte forma: no Capítulo 2 se discute trabalhos relevantes na pesquisa de IA em jogos e na criação de bots para jogos RTS. No Capítulo 3, são explicados os conceitos teóricos utilizados para a criação do bot e para a predição do resultado da partida. No Capítulo 4 se descreve a metodologia utilizada, detalhando o modelo de Raciocínio Baseado em Casos usado, o uso de mapas de influência para a descoberta de padrões de combate e a uso de redes neurais recorrentes para a predição de resultados. No Capítulo 5 são descritos os experimentos realizados e os resultados obtidos e finalmente, no Capítulo 6 é apresentado a conclusão do trabalho

Capítulo 2

Trabalhos Relacionados

Nesse capítulo são apresentados artigos e trabalhos relacionados à inteligência artificial aplicada em jogos RTS e uso de aprendizado por imitação.

2.1 Agentes inteligentes em jogos RTS

As pesquisas em jogos de tabuleiros complexos como Go obtiveram resultados extraordinários nos últimos anos, conseguindo vencer do melhor jogador do mundo [Silver et al., 2017]. Agora os pesquisadores voltam-se seus esforços para o desenvolvimento de inteligência artificial em jogos digitais. Há um grande interesse na pesquisa em jogos RTS devido à complexidade dos mesmos, relacionada à grande quantidade de ações disponíveis, ao ambiente parcialmente observável, a necessidade de planejamento em curto e longo prazo e muitos outros desafios. Em uma parceria da *Blizzard* e *DeepMind* foi lançado uma API para Starcraft 2 permitindo a criação de agentes inteligentes para o jogo de forma mais fácil [Vinyals et al., 2017].

Porém pesquisas de agentes inteligentes em jogos RTS vêm sendo feitas há muito tempo tendo como base o jogo Starcraft: Broodwar. Por exemplo, algoritmos de *Reinforcement Learning*, como o Sarsa, já foram utilizados para aprender a controlar unidades em combates com poucas unidades. Para atingir esse objetivo uma rede neural é utilizada para descobrir a recompensa esperada por atacar ou fugir com uma unidade específica e escolhe-se a unidade com a maior recompensa esperada dentro do jogo. O sistema aprendeu a derrotar a IA já contida em Starcraft para combates com três unidades [Shantia et al., 2011].

No trabalho de Sailer et al. [2007] é utilizada uma abordagem baseada na teoria dos jogos ao procurar o equilíbrio de Nash em um conjunto de estratégias já conhecidas em um jogo RTS simplificado. Essa versão simplificada foca apenas no aspecto tático de jogos RTS concentrando apenas no movimento em grupo de unidades. Foram usadas simulações para comparar o resultado esperado de se utilizar cada estratégia contra o

oponente usando estratégias do mesmo conjunto e então selecionando a estratégia ótima de acordo com o equilibrio de Nash.

Existem competições para estimular a criação de IA para o jogo, como a organizada pela *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (AIIDE). Nessa competição existem 4 categorias: microgestão, combate em pequena escala, jogo com tecnologia limitada e jogos completos o que atrai muitos de participantes. Outras competições ocorrem durante a conferência *Computational Intelligence and Games* (CIG), atual *IEEE International Conference on Games* (COG) e a *Student Starcraft AI Tournament* [Farooq et al., 2016].

Como exemplos de artigos que buscam a criação de agentes com comportamento humano para Starcraft temos: Weber B.G et al. [2011], que cria um bot chamado EISBot onde sua arquitetura é dividida nos gerenciadores de estratégia, renda, construção, tática e de reconhecimento. Temos também trabalhos que focam no desenvolvimento de comportamentos isolados como o de Tavares et al. [2014], que modela a coordenação dos múltiplos agentes do jogo como um problema de alocação, em que tarefas específicas são dadas aos agentes mais apropriados para executá-las. Liu et al. [2014] utiliza algoritmos genéticos para a criação de gerenciamento de micro ações de alta qualidade.

No final do ano de 2018 houve um grande avanço na área de agentes inteligentes para Starcraft com o lançamento do *Alphastar* um agente criado pela empresa DeepMind que foi capaz de derrotar jogadores profissionais de Starcraft 2 em condições iguais às apresentadas em partidas oficiais e sem restrições. Para isso foi utilizada uma rede neural profunda que inicialmente foi treinada usando aprendizado supervisionado com dados de partidas de humanos. Isso permitiu o agente aprender o básico de estratégias micro e macro. Depois essa base foi utilizada como fonte de um processo de aprendizado por reforço multi-agente onde os agentes lutavam entre si em uma competição contínua [Team, 2019].

No trabalho de Tavares et al. [2019] é utilizado seleção de algoritmos, que é o mapeamento de um problema para algoritmos, para a criação de um bot capaz de jogar Starcraft. Para a seleção do algoritmo foi utilizado o minimax-Q tanto em condições quando o algoritmo pode se adaptar em tempo real as estratégias do oponente e quando o algoritmo é estático. Ao ser colocado pra competir contra outros bots foi percebido a melhora nas taxas de vitórias com o avanço do campeonato.

2.2 Técnicas de aprendizado por imitação

Nesse trabalho é usado uma definição ampla de aprendizado por imitação como toda técnica que utilizam dados de ações realizadas por humanos para que o aprendizado ocorra, que é o foco desse trabalho. Existem vários trabalhos que utilizam esses tipos de técnicas, como, por exemplo, raciocínio e planejamento baseado em casos e redes neurais.

No trabalho de Bakkes et al. [2011] é descrito um sistema de aprendizado por observação baseado em casos que é customizado para jogar um jogo RTS criado usando a engine Spring RTS [1] em nível estratégico, enquanto decisões táticas são feitas por um script. É utilizado raciocínio baseado em casos (RBC), com casos extraídos de replays. Também é gravado um valor de fitness para cada estado para que o sistema possa intencionalmente selecionar estratégias sub ótimas quando estiver vencendo para fazer o jogo mais competitivo e divertido de jogar. Isso requer uma boa métrica de fitness o que pode ser difícil de se criar em jogos RTS.

No trabalho de Mehta et al. [2009] é descrito um sistema de Raciocínio Baseado em Casos (RBC) e de planejamento que é capaz de aprender a jogar Wargus, um mod para o jogo Warcraft 2, a partir de replays de jogos humanos com anotações. Essas anotações possuem informação dos objetivos que o jogador estava tentando atingir durante a partida. O sistema pode agrupar sequências de ações em um comportamento para atingir objetivos específicos e aprender a hierarquia de objetivos e a possível ordem que eles ocorrem. Os comportamentos aprendidos são armazenados em uma base de comportamentos que pode ser usada pelo sistema para atingir objetivos durante a partida. Isto resulta em um agente que requer menos do programador da IA porque o sistema pode ser treinado para perseguir objetivos e comportamentos.

O sistema descrito em Weber Ontañón [2010] analisa replays de Starcraft para determinar os objetivos sendo perseguidos pelo jogador para cada ação. Usando uma definição de objetivos criados por especialistas, o sistema aprende quais sequências de ações levam a objetivos específicos e em que situações essas ações ocorrem. Dessa forma, os replays podem ser anotados automaticamente com os objetivos sendo perseguidos a cada momento e esse conhecimento pode ser armazenado em uma base de casos em que pode ser usada em um sistema de planejamento baseado em casos.

Em Oh et al. [2014] são coletados replays de situações de combate em Starcraft: Broodwar. Eventos relevantes do jogo são extraídos de cada *frame*. Para cada frame o valor do campo potencial de cada cena é calculado para criar mapas de influência e armazenados em uma base de dados. Durante o jogo, o agente busca na base de dados por mapas de influência que são mais similares à situação atual. Então os movimentos

[1] https://www.springrts.com

de unidades individuais são imitados de acordo com a semelhança entre unidades. No trabalho de Prado Prandini Faria et al. [2019] é aplicado a estratégia de aprendizado por Imitação Profunda Ativa (AIP) aplicado ao ambiente dinâmico do jogo FIFA. AIP é um segmento de aprendizado por imitação que consiste em uma estratégia de treinamento de *Deep Learning* supervisionado onde os agentes aprendem observando e copiando o comportamento de especialistas humanos.

Capítulo 3

Referencial Teórico

Nesse capítulo é explicada a fundamentação teórica necessária para a compreensão do trabalho, falando sobre Jogos RTS, Raciocínio Baseado em Casos e Redes Neurais Recorrentes.

3.1 Jogos RTS

Jogos de Estratégia em Tempo Real (RTS) são um subgênero de jogos de estratégia onde o progresso do jogo não é feito em turnos. Em um RTS os jogadores posicionam e manuseiam unidades e estruturas durante o decorrer do jogo. Para a construção dessas unidades e estruturas geralmente é necessário utilizar recursos que podem ser encontrados no mapa. Esses recursos são recolhidos em pontos especiais e exigem unidades específicas para essa tarefa. Um típico jogo do gênero RTS geralmente contém características como coleta de recursos, construção de bases, desenvolvimento tecnológico e controle de unidades. A Figura 3.1 mostra a interface de Starcraft: Broodwar, um dos jogos RTS mais populares.

As tarefas a serem executadas em jogos de estratégia em tempo real muitas vezes são complexas, pois geralmente há diversas unidades que devem ser controladas e cada uma executa funções e objetivos diferentes dentro do jogo. Pode ser que uma unidade seja responsável pela coleta de recursos, outra para combate e outra para construção. Além disso deve-se considerar as evoluções tecnológicas que podem ser feitas e qual delas é mais importante a ser priorizada no contexto da partida.

Os principais elementos em jogos RTS podem ser divididos em: coleta de recursos, criação de construções, treinamento de unidades, pesquisa de tecnologias e exploração do mapa. Coleta de recursos é uma das tarefas principais no jogo. Apenas com os materiais coletados é possível aumentar seu exército, construções, obter novas tecnologias entre outras atividades. Os recursos a serem coletados variam de jogo para jogo. Em Warcraft III, por exemplo, temos ouro e madeira como recurso que servem para construir novas

Figura 3.1: Interface do jogo Starcraft: Broodwar.

unidades e avançar tecnologicamente. Existe um limite de unidades no jogo e nas unidades que são capazes de coletar os recursos. Por isso elas devem ser distribuídas eficientemente entre os recursos e os locais de coleta para que o jogador possa continuar aumentando seu exército e promovendo melhorias tecnológicas.

Outro importante elemento em jogos de estratégia em tempo real é a criação de construções. Elas são importantes para treinar unidades de combate e coleta de recursos e para o desenvolvimento de novas tecnologias. Para serem construídas é necessário certa quantidade de recursos coletados. Geralmente o jogador começa a partida com uma construção chamada de base e algumas unidades. A partir dessa base pode-se criar novas unidades e construções. Geralmente, diferentes tipos de população ou raças no jogo possuem diferentes tipos de construções. O treinamento de unidades também é muito importante em jogos RTS porque é com elas que se constrói o exército para combater o inimigo, coletar recursos e prover visão do inimigo.

Não adianta ter um exército com muitas unidades, mas sem as evoluções tecnológicas necessárias para deixar as unidades mais fortes. Por isso a pesquisa de tecnologias é outro aspecto crucial em jogos de estratégia em tempo real. Pesquisas também podem ajudar a tornar as construções mais fortes e protegidas contra ataques inimigos e devem fazer parte da estratégia planejada pelo jogador para vencer o jogo. Já a exploração do mapa é essencial para o planejamento de estratégias pelo jogador pois nesse tipo de jogo, geralmente se usa o modelo de *Fog of War*, onde o jogador só consegue ver uma porção limitada do mapa e é necessário ordenar unidades para explorar o mapa concedendo visão das unidades e construções inimigas. Isso permite saber quando um ataque está sendo planejado ou saber que tipos de unidades estão sendo construídas pelo inimigo. Dessa forma, é possível construir uma composição de exército eficiente contra o inimigo.

3.1.1 Starcraft

O jogo mais popular do gênero RTS é o Starcraft, lançado em 31 de Março de 1998 pela Blizzard Entertainment. O jogo possui três raças, cada uma possuindo unidades e construções diferentes, porém bastante equilibradas. As raças são:

- **Terran**: Representa os humanos e é a raça mais equilibrada por possuir unidades versáteis e baratas e com a capacidade de atacar os inimigos à distância com unidades como tanques, por exemplo.
- **Protoss**: Representa uma raça futurista com armas e estruturas avançadas. Suas unidades são fortes, porém caras para serem construídas.
- **Zerg**: Raça de seres primitivos com unidades baratas que podem construir várias unidades de combate e capaz de expandir territórios rapidamente. Geralmente procura sobrecarregar os inimigos atacando com vantagem numérica de exércitos.

O jogo possui três tipos de unidades:

- ***Workers***: São responsáveis pela a coleta de gás e minérios.
- **Unidades de combate**: São especializadas em atacar o oponente, com algumas capazes de ataques de longo alcance (*ranged*) e outras apenas de curta distância (*melee*). Algumas têm a capacidade de voar podendo atacar apenas outros inimigos voadores ou inimigos terrestres e voadores.
- **Unidades de Suporte**: Servem para auxiliar outras unidades de alguma forma, seja curando outras unidades, providenciando escudo, transportando ou dando visão do mapa.

O jogo possui um modo campanha para um único jogador e um modo multiplayer. O seu modo multiplayer foi um grande sucesso permitindo o surgimento do cenário competitivo (*e-sports*), especialmente na Coreia do Sul onde partidas são transmitidas na TV e jogadores profissionais são tratados como estrelas. O Starcraft vendeu mais de 11,5 milhões de cópias no mundo sendo um terço das vendas na Coreia do Sul.

3.2 Raciocínio baseado em Casos

Raciocínio baseado em casos (RBC) é o processo de resolver novos problemas baseado em soluções de problemas similares no passado. A origem desse processo se deu devido aos trabalhos de Roger Schank [Schank, 1983] onde ele propôs uma visão diferente no raciocínio baseado em modelos, inspirado por raciocínio humano e organização da memória. Schank sugeriu que o nosso conhecimento sobre o mundo é organizado em pacotes de memória que são postos juntos com episódios particulares nas nossas vidas que tiveram impacto suficiente para ser lembrado. Estes pacotes de organização de memória (POM) e seus elementos não estão isolados, mas conectados pela expectativa da progressão normal dos eventos que são chamados de scripts por Schank. Existe uma hierarquia de POM onde POMs maiores compartilham POMs menores. Se um POM contém uma situação em que um problema foi resolvido com sucesso e a pessoa se encontra em situação similar, então a experiência passada é reobtida e a pessoa pode tentar seguir os mesmos passos para tentar chegar em uma solução. Então, ao invés de seguir um conjunto geral de regras, são reaplicados esquemas de soluções que tiveram sucesso anteriormente em um contexto novo e similar. Usando essas observações sobre o processo de raciocínio humano, Schank propôs o modelo de *raciocínio baseado em memória* e os *sistemas especialistas baseados em memória*. Eles são baseados nos seguintes princípios:

- A base de conhecimento utilizada é formada principalmente da enumeração de casos específicos e experiências. Isso se baseia no fato de um especialista ser mais capaz de recordar experiências do que de articular regras internas.

- Os problemas que são apresentados a um sistema especialista baseado em memória podem não ter nenhum caso específico ou regra que é correspondido exatamente, mas o sistema pode raciocinar a partir de similaridades gerais, procurando casos similares e os adaptando para produzir uma resposta. Isto é baseado no poder de generalização do raciocínio humano.

- A memória de experiências utilizada pelo sistema é mudada e aumentada de acordo com cada caso que é apresentado. O aprendizado automático representa um marco em modelos de raciocínio baseados em memória. O sistema lembra de problemas que foram resolvidos e usa essa informação para resolver problemas futuros.

A área de IA relacionada com raciocínio baseado em casos coloca o modelo de Schank de raciocínio baseado em memória em prática.

3.2.1 O ciclo RBC

Para a resolução de problemas usando RBC, são utilizados os passos mostrados na Figura 3.2 baseada na descrição de Kolodner [1992]. Primeiro, dado o problema atual é feita uma pesquisa na base de casos por problemas similares resolvidos anteriormente. Depois recupera-se a solução para esses problemas. Caso os problemas não sejam exatamente iguais ao problema atual é necessário adaptar as soluções obtidas. Em seguida a solução é adaptada e aplicada, verifica se o problema foi resolvido satisfatoriamente e, em caso positivo, esse problema e a solução são armazenados na base de dados. Essa solução então pode ser usada para a resolução de futuros problemas. De acordo com Kolodner, o ciclo CBR pode ser descrito em 4 estágios:

1. *Recuperação de caso*: depois do problema ter sido abordado, o caso mais parecido é procurado na base de casos e uma solução aproximada é retornada.

2. *Adaptação de caso*: A solução recuperada é adaptada para se adequar ao novo problema.

3. *Avaliação da solução*: A solução adaptada é aplicada ao problema e depois o resultado é avaliado. Caso o resultado não seja satisfatório a solução recuperada deve ser adaptada novamente ou novos casos devem ser recuperados.

4. *Atualização da base de casos*: Se a solução foi verificada corretamente, o novo caso deve ser adicionado a base de casos.

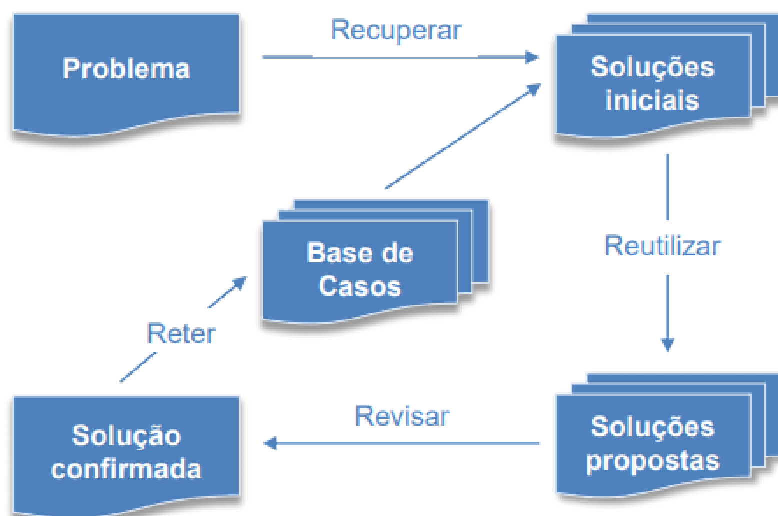


Figura 3.2: Ciclo de um sistema RBC.

Já Aamodt Plaza [1996] descrevem os quatro passos do ciclo RBC mostrados anteriormente de forma um pouco diferente, chamando de *quatro REs*, mostrado na Figura 3.2:

1. RECUPERE o(s) caso(s) mais parecido(s).
2. REUSE o(s) caso(s) para tentar resolver o problema.
3. REVISE a solução proposta se necessária.
4. RETENHA a nova solução como parte de um novo caso.

3.2.2 Representação de casos

Um caso é um conhecimento que representa uma experiência. Ele contém a descrição do problema que foi resolvido, que geralmente é representado pelo o estado do mundo quando o caso ocorreu. Contém também a solução do problema, composta de ações realizadas para atingir o objetivo e em alguns casos, o resultado descrevendo o estado do mundo depois que o caso foi solucionado.

A descrição do problema contém informação suficiente sobre o problema e o contexto necessário para que a recuperação de casos seja eficiente e precisa. É útil armazenar estatísticas de recuperação como o número de vezes que um caso foi recuperado e a quantidade de vezes que foi usado. Estas estatísticas podem ser úteis para priorizar casos mais usados na hora de realizar a busca ou remover casos que raramente são usados e para manutenção geral da base de casos.

A solução do problema pode ser atômica ou composta. Soluções atômicas podem ser vistas em sistemas RBC usados para classificação e diagnóstico. Já soluções compostas são usadas em sistemas RBC usadas para planejamento [Aamodt Plaza, 1996]. Uma solução composta contém sequências de ações, ou um arranjo de vários componentes. Uma solução pode ajudar a chegar a novas soluções que são adicionadas à base de casos, por isso os passos de uma solução de um problema são tão importantes quanto a solução em si.

Casos podem ser representados por vetores de características ou usando algum método formal de IA como frames, objetos, redes semânticas ou regras. O formalismo escolhido depende da informação armazenada em cada caso. Não existe consenso na comunidade de RBC sobre qual informação deve ser armazenada em cada caso e por consequência qual é a melhor forma dele ser representado.

3.2.3 Indexação

Indexação é o processo de dar índices a casos de modo a facilitar sua recuperação. Pesquisadores em RBC propuseram diferentes diretrizes para a indexação de casos [Aamodt Plaza, 1996]. Índices devem ser: preditivos para a relevância do caso, reconhecíveis de modo que seja compreensível porque ele está sendo usado, abstratos para permitir a expansão da base de casos no futuro e concretos o suficiente para permitir uma recuperação eficiente e correta.

Existem métodos manuais e automáticos para a seleção de índices. Quando se seleciona índices manualmente, se decide qual é o propósito e em qual circunstância cada caso vai ser mais útil. Pesquisadores dizem que a escolha manual de índices é melhor do que a escolha automática, porém existe um número cada vez maior de algoritmos automáticos para executar essa tarefa. Os índices podem não ser fixos, ou seja, eles podem mudar durante o uso do sistema. A mudança de índices pode ser encarada como uma forma de aprendizado. Por exemplo, se um caso errado é recuperado ou um problema totalmente novo pode ser encontrado, então é possível mudar os pesos das características que definem os índices. Pode-se também mudar as características totalmente ou então adicionar ponteiros para outros casos na base de casos.

3.2.4 Organização da base de casos

Existem três abordagens principais para a organização da base de casos: organização plana, organização em grupo e organização hierárquica. Também pode se fazer uma combinação dessas abordagens.

- **Organização plana**: A forma mais fácil de se organizar uma base de casos é em uma estrutura planar da base de casos. Tem a vantagem de ser simples e fácil de adicionar e apagar casos porem a recuperação de casos é feita olhando caso por caso em toda a base, o que faz se perder muito tempo para bases maiores.

- **Organização em grupo**: A base de casos é organizada em grupos baseando-se na similaridade entre eles. A vantagem desse tipo de organização é que a seleção de em qual grupo o caso se encontra é mais fácil porém requer algoritmos mais complexos para a adição e o subtração de casos comparado com a organização plana.

- **Organização hierárquica**: A memória de casos é organizada em uma estrutura em forma de rede de categorias, relações semânticas, casos e índices. Cada caso é associado a uma categoria e as categorias são conectadas por uma rede semântica contendo características e estados intermediários. A organização hierárquica facilita a recuperação de casos, porém possui complexidade na adição e exclusão de casos e a reorganização e manutenção da base é muito custosa.

3.2.5 Recuperação de Casos

Existem várias formas de se realizar a busca de casos na base de dados. A mais simples dela é o método *primeiro vizinho mais próximo* que executa testes de similaridade para todos os casos na base e retorna apenas o melhor resultado. Esse método requer longo tempo para executar a busca, especialmente para bases de casos maiores. Para acelerar o processo de busca de casos pode-se realizar uma pré-seleção com uma medida de similaridade mais simples usando indexação dos casos.

Outra forma de se executar a busca mais rapidamente é fazendo o ranqueamento de casos. O método mais simples de ranqueamento é usando estatísticas de busca. Os casos que são mais buscados na base, podem ser priorizados na busca. Os resultados da busca podem retornar um ou vários casos que melhor correspondem à busca. Os mecanismos de busca tendem a ser simples e mais rápidos se um número maior de casos são recuperados. Se todos os resultados são usados para encontrar a solução e no final a melhor a solução é escolhida.

Outra forma de acelerar a busca é utilizando paralelismo. Buscas em paralelo podem ser realizadas pois a correspondência de casos não requer muita comunicação entre os processos executando em paralelo. A implementação de busca em paralelo é simples em bases de casos em organização plana ou em grupos, e é mais difícil de ser implementada em bases de casos hierárquicas.

3.2.6 Adaptação

Quando um caso é recuperado da base, na maioria das vezes ele não irá corresponder ao problema que se deseja encontrar a solução, por isso o caso recuperado deve ser adaptado. O processo de adaptação procura por diferenças entre o caso

recuperado e o caso atual e aplica fórmulas ou um conjunto de regras para corrigir essas diferenças quando apresentada uma solução. Em geral, existem dois tipos de adaptações em RBC:

- **Adaptação estrutural**: Regras de adaptação são aplicadas diretamente nas solução armazenadas em casos. Se a solução é representada por apenas um valor ou uma coleção de valores independentes, adaptação estrutural pode ser aplicada modificando certos parâmetros na direção apropriada ou fazendo uma junção de vários casos recuperados da base. Porém, se existe interdependência entre os componentes da solução, a adaptação estrutural é mais complicada por requerer uma compreensão maior do problema e um modelo melhor definido
- **Adaptação derivacional**: A adaptação é feita reusando algoritmos, métodos, ou regras que geraram a solução original para produzir a nova solução para o problema em questão. Este tipo de adaptação as vezes é chamada de *re-instanciação* e pode ser usada para problemas com domínios que são bem compreendidos.

3.3 Planejamento online baseado em casos

O Planejamento Online Baseado em Casos (POBC) é uma adaptação do Raciocínio Baseado em Casos discutido na Seção 3.2 e foi proposto por Ontanon em Ontanón et al. [2010], esse algoritmo é mais adequado pra resolução de problemas mais complexos como os apresentados no domínio de jogos de estratégia em tempo real. Os problemas que surgem ao se utilizar RBC nesse tipo de domínio são devidos principalmente a duas suposições que são feitas nesse modelo. A primeira suposição é que o problema pode ser resolvido de uma vez só, ou seja, em uma única passada do ciclo RBC. No Planejamento Baseado em Casos, para resolver um problema pode ser necessário a resolução de vários subproblemas e monitorar a sua execução para o caso de ocorrer mudanças no ambiente .

A segunda suposição que é feita é que a execução e a resolução do problema são separadas, ou seja, o ciclo RBC produz a solução mas a execução da solução delegada para um módulo externo. Em domínios estratégicos de tempo real a execução do problema faz parte da sua resolução especialmente quando o modelo interno do mundo não é perfeito e ter certeza que a solução foi executa corretamente faz parte da solução. Por exemplo, ao se executar uma solução o sistema pode encontrar detalhes inesperados ou mudanças no mundo que tornam aquela solução inválida e por isso outra solução deve ser proposta.

A Figura 3.3 mostra o ciclo de Planejamento Baseado em Casos que é uma extensão do ciclo de Raciocínio Baseado em Casos. Foram adicionados dois processos para lidar

com o planejamento e execução de soluções em domínios em tempo real. Os dois processos adicionados foram:

- **Expansão**: Esse processo obtém a solução proposta pelo sistema para um problema e busca por subproblemas nele. Se houver, estes subproblemas são enviados para o processo de recuperação para que eles possam ser solucionados. Outro objetivo desse processo é monitorar mudanças no mundo para, caso haja mudança significativa que atrapalhe a resolução do problema, a solução seja enviada novamente para a etapa de adaptação. Essa etapa é chamada de *adaptação atrasada*. Esta é uma característica importante quando se trabalha com ambientes dinâmicos.

- **Execução**: Este processo é encarregado de executar a solução atual e atualizar o progresso de acordo com o resultado da execução. Se em algum passo a solução falhar o que causa um subproblema também a não ser resolvido então a solução é atualizada para refletir esse acontecimento. Quando isso ocorre o módulo de expansão é responsável por busca uma nova solução para o subproblema.

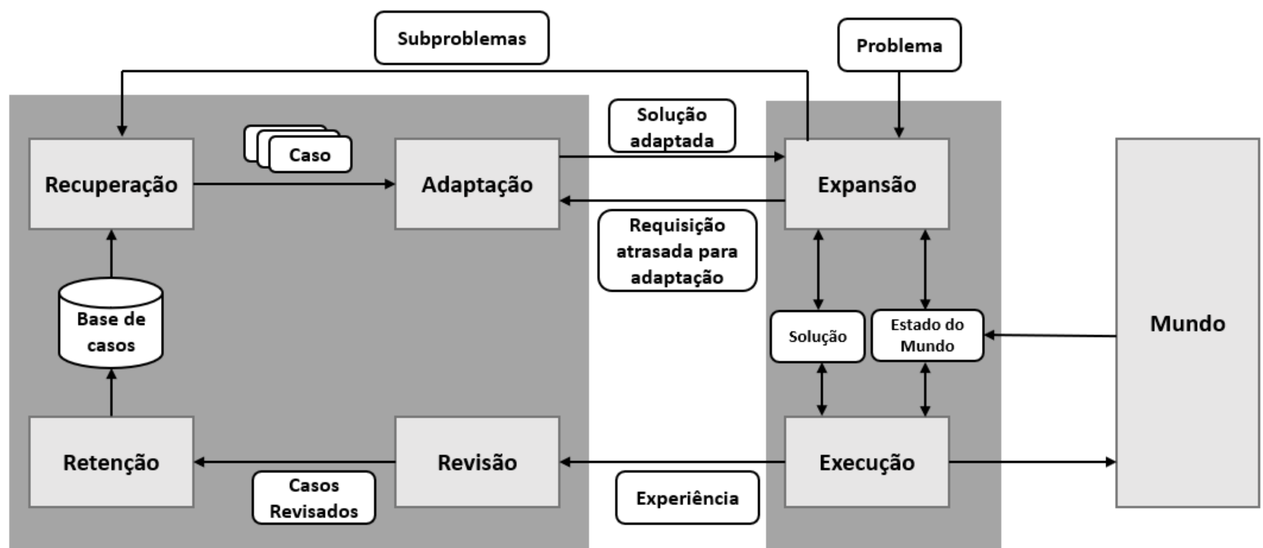


Figura 3.3: Ciclo do planejamento online baseado em casos (POBC)

O fluxo no Planejamento Baseado em casos é o seguinte: problemas chegam no processo de expansão que decompõe em subproblemas se necessário. Cada um desses subproblemas é enviado para o processo de recuperação que recupera casos relevantes da base de casos. O processo de adaptação gera uma solução para os subproblemas reusando e adaptando as soluções nos casos recuperados. Estas soluções são enviadas de volta para o processo de expansão que incorpora na solução atual. Ao mesmo tempo em que o processo de expansão constrói a solução, o processo de execução executa partes da solução que já tenham sido completamente expandidas. O resultado da execução dessas soluções é enviado para o processo de revisão que verifica a solução proposta baseado no

resultado obtido no ambiente. Finalmente, o processo de retenção decide se deve reter novas experiências ou não.

Existem quatro outras melhorias feitas quando comparado com o ciclo do Raciocínio Baseado em Casos:

1. Problemas entram no ciclo através do processo de expansão. Quando um problema chega no sistema, este problema é colocado como o plano atual. Assim a primeira coisa que o módulo de expansão faz é enviar este problema para o processo de recuperação

2. O ciclo RBC é dividido em duas partes: a primeira é composta de recuperação e adaptação e a segunda parte é composta de revisão e retenção. A primeira parte tem a função de achar soluções para novos problemas e a segunda parte é responsável por aprender com a experiência. A revisão obtém como entrada o resultado da execução da solução e a revisa para verificar se atingiu o objetivo. Soluções revisadas são transferidas para o processo de retenção que decide se retém novos casos ou não.

3. O novo ciclo incorpora o *mundo* ao seu projeto uma vez que é parte de qualquer processo de resolução de problema em tempo real.

4. O novo ciclo contém um novo processo de adaptação atrasada. Em um ambiente que muda dinamicamente, nós queremos uma adaptação atrasada para o último momento para ter certeza que os planos são adaptados de acordo com a última informação. Assim o componente de expansão pode enviar de volta planos para adaptação se o ambiente muda muito desde que o plano foi adaptado da última vez. Assim é importante que o processo de adaptação seja eficiente.

3.4 Redes Neurais Recorrentes

Redes neurais artificiais são modelos computacionais baseado na biologia do cérebro e os seus neurônios. Nesse modelo os neurônios são representados por nós e esses nós podem receber dados, realizar cálculos e atualizar seus pesos. Então esses dados são propagados para nós de saída onde é possível ver o resultado de processamento da rede. No modelo padrão de redes neurais artificiais, dados fluem da camada de entrada, através da camada oculta para a camada de saída, então o dado não passa pelo mesmo nó da rede duas vezes. Por causa disso, a rede não tem memória da entrada que ela recebeu anteriormente e não tem noção da ordem de entrada. Nas Redes Neurais Recorrentes (RNR) o ciclo da informação passa por um loop levando em consideração a entrada

atual e o que aprendeu de entradas passadas. Esta configuração de rede neural permite a presença de memória de curta duração e é mais adequada para tarefas que envolvem dados sequenciais, como séries temporais, textos, dados financeiros, áudio e outros. A Figura 3.4 mostra um exemplo de topologia de uma rede neural recorrente. Observe como o dado da camada de saída volta para alimentar a camada oculta.

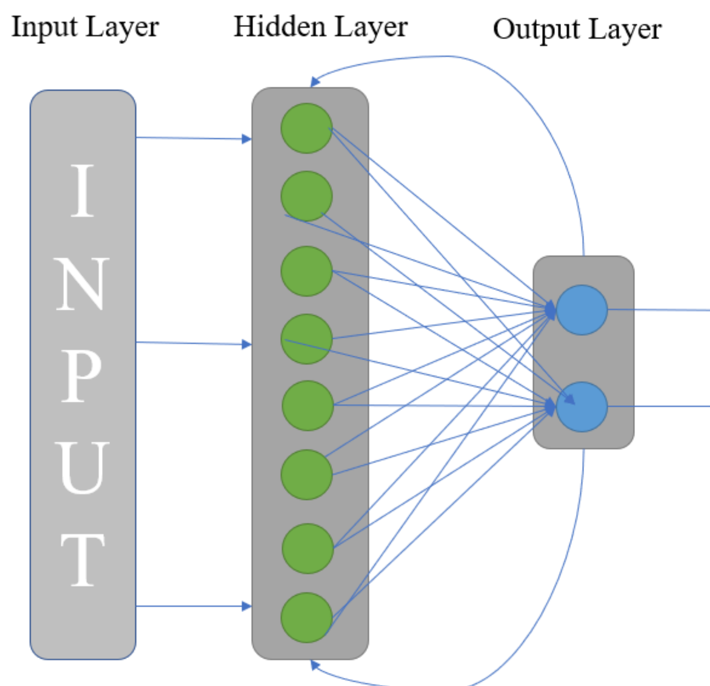


Figura 3.4: Topologia de uma rede neural recorrente em uma camada escondida.

Um tipo específico de RNR é a rede *Long Short-Term Memory* (LSTM) que permite a RNR estender sua memória tornando-a mais adequada para tarefas onde existe um intervalo entre experiências importantes. LSTMs têm unidades similares a células que decidem que informação é armazenada e quais são esquecidas de acordo com sua importância [Gers et al., 1999]. A importância é determinada através de pesos que também são aprendidos através de *backpropagation*. Isto significa que a rede aprende que informação é mais importante que outras durante treinamento. As células podem ser de três tipos: entrada, esquecimento e saída. As células de entrada definem que informações da entrada podem passar pela rede, as células de esquecimento apagam informações que não são importantes e as células de saída definem o quanto de impacto a saída tem considerando a entrada no intervalo de tempo atual. Outro tipo de RNR é a *Gated Recurrent Unit* (GRU), que é parecida a LSTM mas tem apenas dois tipos de células: atualização e reset [Cho et al., 2014]. A célula de atualização determina quanto de informação do passado é passada para o futuro e a célula reset determina quanto de informação é esquecida.

As redes neurais recorrentes são utilizadas nesse trabalho para prever o resultado de partidas de *Starcraft* a partir de uma base de dados que contém mais de 65000 replays de partidas de humanos.

Capítulo 4

Metodologia

4.1 Planejamento Online Baseado em Casos em Jogos de Estratégia em Tempo Real

Jogos de estratégia em tempo real têm diversas características que dificultam a implementação de uma abordagem tradicional de planejamento.

- Eles têm grandes espaços de decisão (ou seja, a quantidade de ações que podem ser tomadas em determinado momento é muito grande).
- Eles possuem um grande espaço de estados.
- Eles são não-determinísticos.
- Eles são jogos de informação incompleta, onde o jogador somente pode ver parte do mapa que já foi explorado e inclui oponentes imprevisíveis.
- Eles são dinâmicos. Assim enquanto o sistema está decidindo qual ação executar o jogo continua executando e o estado do jogo muda constantemente.

Por exemplo, Starcraft (Figura 3.1) é um jogo de estratégia em tempo real onde o objetivo do jogador é continuar vivo e destruir a base inimiga. Cada jogador tem uma série de tropas e construções e tem que obter recursos (minerais e gás) para produzir mais tropas e construções. Construções são necessárias para produzir tropas e tropas são requeridas para atacar o inimigo. Portanto, Starcraft envolve raciocínio complexo para determinar quando e como construir quais tropas e construções. Uma estratégia padrão para um mapa padrão para a raça *Terran* envolve: construir *workers* para coletar minérios, construir uma refinaria para coletar gás, construir barracas para treinar *marines* e *medics*, construir fábricas para criar *Siege Tanks* e finalmente atacar o inimigo.

Porém, estratégias padrão não são efetivas contra especialistas humanos. Por exemplo, no começo do jogo *marines* são fracos contra as unidades básicas das outras raças como *zealots* e *zerglings*. Se o inimigo decide produzir essas unidades em massa no

começo do jogo, o jogador deve colocar suas construções posicionados em certas passagens estreitas para obrigar os inimigos a se enfileirarem para passar por esse ponto assim permitindo que os marines ataquem os inimigos um a um ao invés de serem cercados.

Existem diversas razões pelas quais abordagens padrão de planejamento baseado em busca não podem ser diretamente aplicadas em domínios como Starcraft. Por exemplo, centenas de milhares de comandos diferentes podem ser usados na situação mostrada na Figura 3.1. Portanto, busca competitiva usando minimax não é possível. Planejamento utilizando a técnica STRIPS não pode ser diretamente aplicado pois a quantidade de estados é muito grande. O algoritmo de planejamento HTN pode suportar problemas maiores do que algoritmos de planejamento tradicionais. Porém o benefício vem com o custo de ter que definir um certa quantidade de conhecimento do domínio.

Mesmo se a complexidade computacional dos algoritmos de planejamento fosse baixa o suficiente para ser aplicada em Starcraft, algoritmos clássicos de planejamento assumem domínios determinísticos. Portanto, técnicas de planejamentos probabilístico são exigidas. Técnicas de planejamento probabilístico são baseados em MDPs (Processo de Decisão de Markov ou Markov Decision Process) ou POMDPs (Processo de Decisão de Markov Parcialmente Obsérvavel). Estas técnicas providenciam uma fundação firme para o planejamento em domínios probabilísticos mas têm a desvantagem de custar caro computacionalmente.

O ciclo POBC foi projetado se pensando em domínios como jogos RTS. Esta abordagem foi implementada na IA do UFMGBOT projetado para jogar Starcraft.

4.2 POBC para Starcraft

Nesta seção será apresentado o sistema implementado na IA do UFMGBOT que utiliza a arquitetura de planejamento online baseado em casos descrita anteriormente. A IA do UFMGBOT aprende a jogar Starcraft observando como humanos jogam. A IA aprende o que chamamos de trecho de planos observando o que humanos fazem e armazenam esses trechos no sistema em forma de casos. A execução da IA pode ser dividida em dois estágios principais.

- **Aprendizado**: Durante o aprendizado, a IA observa ações tomadas durante o jogo para aprender trechos de planos que será armazenada na base de casos. No nosso experimento os jogos foram obtidos da base de dados STARDATA. Conjuntos de ações tomadas nas partidas são anotados de forma automática indicando o objetivo

que o jogador perseguia naquele momento. Para isso são identificados quais os objetivos mais comuns perseguidos por jogadores durante a partida, como: produzir *workers*, obter recursos, criar construções, atacar o inimigo e etc. Usando essas anotações um conjunto de trechos de planos são extraídos da sequência de ações e armazenado na base de casos. Para cada trecho é armazenado: a situação em que foi executado, o objetivo que estava perseguindo e o seu sucesso ou falha.

- **Execução**: O processo de execução para se ganhar a partida é feito pelos processos de recuperação de planos, adaptação de planos, expansão e execução. O módulo de execução de planos fica encarregado de executar o plano atual e atualizar o seu estado indicando quais ações foram bem sucedidas ou não. O módulo de expansão é encarregado de identificar subobjetivos no plano atual e executá-los. Mas para fazer isso ele precisa do módulo de Recuperação de planos que, dado um subobjetivo e o estado atual do jogo ele recupera o plano mais apropriado na base de dados. E, finalmente, temos o módulo de adaptação que adapta os planos recuperados de acordo com o estado atual do jogo.

Um dos principais aspectos desse sistema é que ele intercala planejamento e execução para lidar com domínios dinâmicos. É realizada uma combinação de recuperação de planos baseado em casos e adaptação de planos sem a realização de buscas para achar trechos que são capazes de atingir os objetivos de cada plano.

A principal diferença dessa implementação do planejamento online baseado em casos da implementação descrita por Ontanón et al. [2010] é que nesse trabalho os planos são representados por macro-ações onde se abstrai as micro ações realizadas pelo o jogador. Por exemplo, nesse trabalho ao invés de aprender que para se coletar o mineral é preciso executar as ações de selecionar uma unidade e direcionar até um mineral e depois voltar pra base, esse processo é abstraído com a criação de um script que executa essa tarefa básica e com isso o planejamento pode se preocupar em aprender a visão macro do gerenciamento estratégico e econômico do jogo.

As próximas subseções irão explicar cada módulo com mais detalhes. Primeiro será mostrada a linguagem utilizada para representar planos. Depois será descrito o processo de aprendizado, incluindo o processo de revisão e aprendizado de casos. Em seguida será descrito o módulo de recuperação de casos. Depois o processo de expansão e execução de planos e, finalmente, o módulo de adaptação.

4.2.1 Representação de planos

Nesta seção será mostrada o formalismo usado para representação de planos que tem como objetivo permitir à IA aprender novos planos, representar e raciocinar sobre os mesmos. O componente básico são os trechos de planos. Esses trechos são compostos de três elementos:

- Um conjunto de precondições que devem ser satisfeitas antes que o plano possa ser executado. Por exemplo, um trecho pode ter como pré-condição que uma certa quantidade de *workers* esteja presente e que a refinaria não esteja sendo usada por trabalhadores.
- Um conjunto de condições ativas que representam as condições que devem ser satisfeitas para que a execução do plano tenha chance de sucesso. Durante a execução do plano, se as condições ativas não são satisfeitas, o plano pode ter sua execução parada uma vez que não vai conseguir atingir o objetivo. Por exemplo, um *worker* criando uma construção deve permanecer vivo senão a construção não será criada.
- O plano em si, que contém as ações que podem ser executadas, por exemplo, em Starcraft, essas ações seriam atacar, mover, construir etc. E possível também conter subobjetivos, o que significa que a aplicação tem que achar um trecho de plano que o resolva antes de resolver o objetivo principal.

Trechos de planos possuem objetivos, por exemplo, "ter uma torre". Para cada domínio, um conjunto de objetivos possíveis deve ser definido. Diferente de abordagens clássicas de planejamento, pós-condições não podem ser especificadas para um trecho de plano pois não há garantia que ele será bem sucedido. Então, só pode ser determinada sua condição de sucesso. Para usar POBC no domínio de Starcraft é preciso três coisas:

- Um conjunto de macro ações como: atacar inimigo, criar construção e etc.
- Um conjunto de sensores que obtém informação sobre o estado atual do mundo que são usadas para especificar as pré-condições, as condições ativas e objetivos dos trechos de plano. Para starcraft exemplo de sensores são: *numeroDeTropas*. *numeroDeMinerais*, *numeroDeWorkers*.
- Um conjunto de objetivos, que podem ser estruturados de forma hierárquica para especificar a relação entre eles. Um objetivo pode ter parâmetros e cada objetivo tem uma condição de sucesso. Por exemplo, *TemUnidades(Torres)* é um objetivo válido para starcraft e tem uma condição de sucesso: *UnidadeExiste(Torre)*.

4.2.2 Aquisição de planos

Para adquirir os planos necessários para realizar o planejamento online baseado em casos foi utilizada a base de dados STARDATA que contém replays de mais de 65000 partidas de Starcraft jogadas por humanos. Primeiro, foi necessário estabelecer uma ontologia de objetivos que é utilizada para descrever os objetivos que estão sendo perseguidos dado ações executadas em determinado tempo de jogo.

A ontologia de objetivo foi formulada baseada em análise de partidas profissionais de Starcraft e é mostrada na Figura 4.1. O subobjetivo de economia contém tarefas que alcançam o objetivo de gerenciar a infraestrutura de recursos, enquanto o subobjetivo de estratégia contém tarefas para alcançar os objetivos de expandir a árvore de tecnologias e produzir unidades de combate.

Cada objetivo na ontologia tem um conjunto de parâmetros que são usados para avaliar se o objetivo atual é aplicável a situação atual do jogo. Por exemplo, o objetivo econômico recebe como entrada os parâmetros: tempo atual de jogo, o número de unidades controladas pelo agente, o número máximo de unidades que o agente suporta, o número de unidades *worker*, expansão e refinarias e o número de *workers* coletando gás.

Para extrair os casos das ações executadas e anotar com os objetivos sendo perseguidos, é executado dois processos. Na primeira parte do processo, as ações de uma sequência específica são colocadas em diferentes categorias baseados em subobjetivos que alcançaram e então agrupados em casos baseados em localidade temporal. Por exemplo, a ação de atacar o inimigo seria colocada na categoria e agrupada em um caso onde foram executados outros ataques ao inimigo que ocorreram em um espaço de tempo específico. O segundo passo do processo é a fase de anotação, onde os casos são rotulados baseados no estado do jogo onde a primeira ação ocorreu no caso. Essa abordagem agrupa ações em três categorias que correspondem aos subobjetivos principais da ontologia com o objetivo de vencer uma partida de Starcraft.

Primeiro, a sequência de ações do jogador são convertidas em sequência de macro ações, depois eles são agrupadas em casos usando o modelo baseado em reconhecimento de objetivos que se aproveita da estrutura linear dos rastros de macro ações do jogador. Dado uma macro ação no tempo t em uma sequência, nós podemos computar o estado do jogo no tempo $t + n$ recuperando diretamente da sequência de macro ações onde n é o número de macro ações que ocorreram a partir do tempo t. O estado de jogo em $t + n$ representado por S_{t+n} pode ser inferido como o objetivo do jogador no tempo t dado que o jogador tem um plano de tamanho n. Este modelo de reconhecimento de objetivo assume que o jogador tem um plano linear para atingir um objetivo específico. Isso é corroborado pelo que é chamado de "ordem de construção"que é bastante popular em Starcraft, que se refere sequências de ações predefinidas executadas pelo jogador na abertura do jogo.

Planos são construídos iterando sobre o conjunto de macro ações em uma sequência e construindo novos planos a cada n macro ações usando o seguinte algoritmo:

$$P.S = S_t$$
$$P.G = grafo(A_t, A_{t+1}, ..., A + t + n)$$
$$P.O = categoria.O$$
$$P.O.parametros = computar_parametros(S_{t+n})$$

onde P é o plano gerado, S é o estado do jogo, G é o grafo contendo as macro ações, pré-condições e condições de sucesso, como mostrado na Figura 4.4, A é uma macro ação executada pelo jogador na sequência, *categoria.O* é o subobjetivo, *P.O.parametros* são os parâmetros dos subobjetivos e uma função que computa os parâmetros dos objetivos dado um estado de jogo.

- Vencer Starcraft
  - ⋆ Economia
    - ∗ Produzir unidades *workers*
    - ∗ Coletar recursos
    - ∗ Construir instalações de recursos
    - ∗ Gerenciar suprimentos
  - ⋆ Estratégia
    - ∗ Construir instalações de produção
    - ∗ Criar construção tecnológicas
    - ∗ Produzir unidades de combate
    - ∗ Pesquisar melhorias
  - ⋆ Tática
    - ∗ Atacar oponente
    - ∗ Usar habilidade tecnológica ou feitiço

Figura 4.1: Ontologia de objetivos para ações em Starcraft

4.2.3 Recuperação de planos

Para resolver uma tarefa de planejamento complexa, vários subproblemas têm que ser resolvidos. Por exemplo, em Starcraft para atingir o objetivo de ganhar o jogo,

primeiro deve-se coletar recursos, construir a base e depois atacar o inimigo. Estes problemas são diferentes entre si e na base de casos pode haver diferentes trechos de plano para resolver cada um desses problemas em diferentes circunstâncias. Com isso, a base de casos será heterogênea. Como um trecho pode funcionar em uma situação e em outras não, será necessário adicionar informação em que estado um trecho foi bem sucedido. Por isso a base de casos pode ser organizada em dois elementos:

- Trechos de plano: um procedimento composto de conjunto de ações, subobjetivos em sequência ou paralelo.
- Episódios: Um episódio é uma tupla $e = (p, G, S, O)$ onde $e.p$ é um trecho de plano, $e.G$ é um objetivo, $e.S$ é uma situação ou estado do jogo e $e.O$ é o resultado de executar o plano que é um número real entre 0 e 1 indicando quão bem o trecho alcançou seu objetivo.

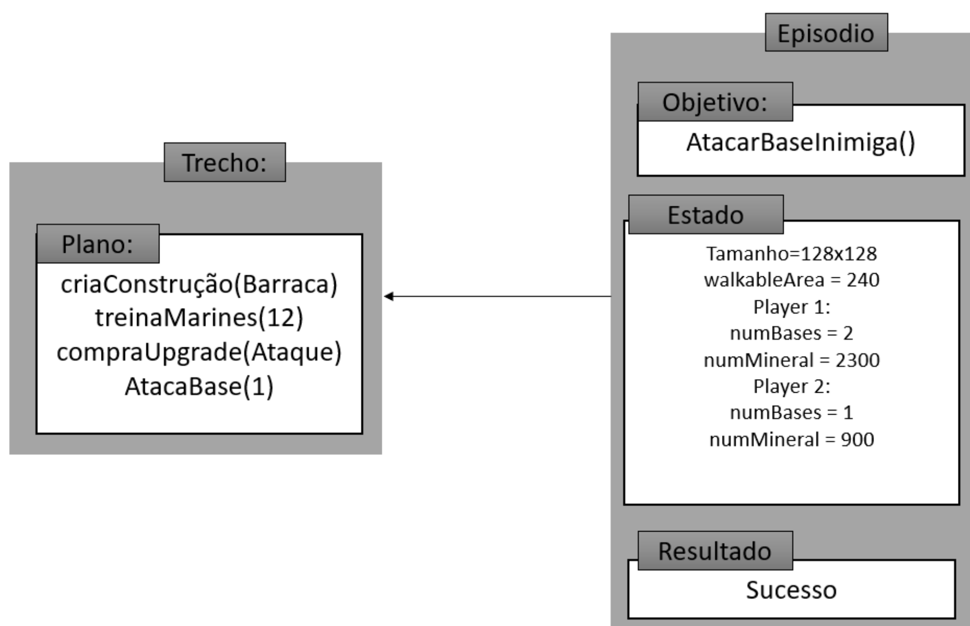


Figura 4.2: Exemplo de um episódio e um trecho de plano.

Depois do processo de aprendizado de plano, cada trecho na base de casos está associada com um episódio. Quando o agente jogar uma partida, novos episódios serão adquiridos e um trecho pode estar associado a vários episódios, armazenando a experiência de executar esse trecho em diferentes contextos. Assim, o agente irá aprender por experiência que trecho é mais adequado para cada situação.

A Figura 4.2 mostra um exemplo de trecho de plano associado a um episódio, com quatro elementos: um objetivo, o estado do jogo, o resultado no episódio e o trecho de plano que contém um procedimento para atingir o objetivo específico. O estado do jogo é representado por 40 características sendo 16 pra representar a quantidade de cada

tropa pois o número máximo de tipo diferentes de tropas é 16 entre as raças, 16 pra representar a quantidade de cada construção pois, novamente é o número máximo de tipos de construções diferentes entre as raças, 2 pra representar a quantidade de minério e gás, e 6 características para representar o mapa: o tamanho de cada dimensão, a porcentagem de terreno que pode ser caminhado, a porcentagem de terreno plano, porcentagem de terreno elevado e a quantidade de bases.

Dado um estado do jogo e um objetivo, o ideal seria recuperar um trecho de plano que tenha a melhor performance dado esse estado e objetivo. Para predizer a performance do trecho p, o módulo de recuperação usa os episódios associados com p. Para recuperar os episódios, o sistema usa uma medida de relevância de episódio $ER(e, S, G)$ que computa a relevância dado um episódio e, o estado atual S e o objetivo G, e é definido como:

$ER(e, S, G) = \alpha GS(e.G, G) + (1 - \alpha)SS(e.S, S)$

Onde GS é a similaridade entre o objetivo do trecho e o objetivo atual e SS é a similaridade entre estados. α é um parâmetro que foi definido como 0.75 nos experimentos. A similaridade entre objetivos GS é 1 se os objetivos são diferentes e 0 se são iguais.

Já a similaridade entre estados SS retorna a similaridade entre dois estados representado por um número entre 0 e 1. Para isso, é calculado o inverso da distância euclidiana simples entre as características dos estados do jogo normalizada entre 0 e 1.

Para predizer a performance de um trecho de plano, é definido $RE(p, S, G) = \{e_1, ..., e_k\}$, como o conjunto de k episódios mais relevantes definidos pela a função ER associados a um trecho p. Onde o $k \leq 5$. Usando essa definição para predizer a performance de um trecho de plano é utilizado a fórmula 4.1:

$$PP(p, S, G) = \frac{1 + \sum_{e \in RE(p,S,G)} ER(e, S, G) \times e.O}{2 + \sum_{e \in RE(p,S)} ER(e, S, G)} \tag{4.1}$$

A fórmula diz que a performance prevista é a média ponderada dos resultados que o trecho p obteve em estados do jogo similar ao estado atual S com objetivos similares. Os valores 1 e 2 foram adicionados ao numerador e denominador respectivamente seguindo a regra de estimativa de probabilidade laplace que enviesa a performance prevista para 0.5 quando se tem poucos episódios. O resultado do processo de recuperação é o trecho p que tem a mais alta performance prevista $PP(p, S, G)$.

4.2.4 Expansão e Execução de plano online

Durante a execução, a expansão, execução e adaptação de planos ocorrem em paralelo para manter uma árvore de planejamento parcial. A árvore de planejamento parcial consiste em dois tipos de nós: objetivos e trechos de plano.

Inicialmente o plano consiste de um único objetivo que corresponde a tarefa de planejamento atual. No nosso sistema o objetivo inicial é sempre "Vencer o jogo". Assim o módulo de expansão de planos pergunta ao modulo de recuperação para buscar um trecho para aquele objetivo. Este trecho pode haver vários subobjetivos, assim o módulo de expansão irá perguntar novamente para o módulo de recuperação para buscar novos trechos de plano. Na Figura 4.3 mostra um exemplo de um plano onde o objetivo no topo é "Vencer", porém ele tem três subobjetivos: "Construir uma base", "Construir um exército"e "Atacar". O objetivo de "construir base"já foi atribuído um trecho de plano e ele não tem subobjetivos. O resto ainda não foi atribuído um trecho. Quando um objetivo ainda não tem um trecho atribuído se diz que ele está em aberto.

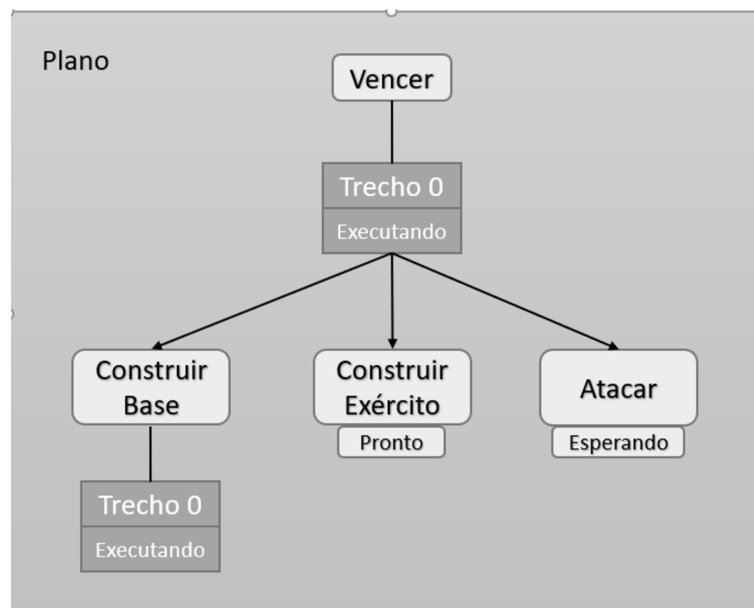


Figura 4.3: Representação de um plano expandido.

Cada trecho de plano tem um estado associado: pendente, executando, sucesso ou falha. Se ele não foi executado ainda, considera-se pendente e depois de executado pode-se atribuir o estado de sucesso ou falha. Quando um trecho falha, ele também é considerado em aberto pois é necessário buscar um novo trecho para este objetivo. Objetivos abertos podem estar tanto prontos ou esperando. Um objetivo em abeto está pronto quando todos os trechos que devem ser executados antes desse objetivo são bem sucedidos, caso contrário ele fica esperando.

Algorithm 1 Algoritmo de expansão de plano, onde p é o plano atual e S é o estado atual do jogo.

function ExpansaoDePlano(p, S)
  G = ObtemObjetivosAbertos(p)
  for $g \in G$ **do**
    p_g = RecuperaPlano(g, S)
    p'_g = AdaptaPlano(g, S)
    p_g = InsereTrechoNoPlano(g, S)
  end for
  return p
end function

O algoritmo 1 mostra o algoritmo do módulo de expansão de plano que executa o ciclo de execução dado um plano p e um estado de jogo S. O módulo de expansão de plano está a todo momento perguntando ao plano atual se existe um objetivo em aberto. Quando há um objetivo em aberto, é enviado para o módulo de recuperação de planos para recuperar um trecho de plano apropriado. Então, o trecho é enviado para o módulo de adaptação para inserir no plano atual.

O algoritmo 2 mostra o algoritmo que o módulo de execução de plano executa a cada ciclo e é composto de quatro etapas:

- *ComecaTrechosProntos*: Para cada trecho pendente, o módulo de execução avalia as pré-condições e quando elas são finalizadas, o trecho é executado. Se o estado de jogo for mudado, o trecho é enviado para o módulo de adaptação.

- *enviaAçõesParaExecucao*: Se qualquer trecho sendo executado tem macro ações e estas macro ações tem as pré-condições satisfeitas, elas são enviadas para o agente executar. Se as pré-condições não forem satisfeitas o trecho é enviado para o módulo de adaptação para tentar ser reparado. Se não for possível, o trecho é marcado como falho e o objetivo é tido como aberto, buscando um novo trecho. Se for encontrado inimigos o módulo de combate descrito na Seção 4.3 é posto em ação.

- *atualizaStatusDeTrecho*: O módulo de execução periodicamente avalia as condições ativas e de sucesso de cada trecho. Se as condições ativas não são satisfeitas o trecho é marcado como falho e o objetivo é considerado aberto novamente. Se as condições de sucesso são satisfeitas, o trecho é marcado como bem sucedido.

- *atualizaStatusDaAcao*: Quando uma macro ação é bem sucedida ou falha o módulo de execução atualiza o status do trecho que o contém. Quando a macro ação é bem sucedida o trecho executado pode continuar para o próximo passo, quando falha o trecho é marcado como falho e o objetivo é colocado em aberto.

Algorithm 2 Algoritmo de execução do plano, onde p é o plano atual e S é o estado atual do jogo.

```
function ExecucaoDePlano(p, S)
    p = comecaTrechosProntos(p, S)
    p = enviaAcoesParaExecucao(p, S)
    p = atualizaStatusDeTrecho(p, S)
    p = atualizaStatusDaAcao(p, S)
    return p
end function
```

4.2.5 Adaptação de plano

Tradicionalmente adaptação de plano consiste em dois subprocessos: adaptação de parâmetros e adaptação estrutural do plano. Nesse trabalho o primeiro não é necessário pois ações básicas foram abstraídas em macro ações que já realiza essa adaptação. Por exemplo, uma ação básica que pode ser realizada é a construção de um *gateway* pelo probe 1 na coordenada 112x48. Na macro ação isso é representado pela função *criarConstrução(gateway)* que escolhe o *probe* e a coordenada que achar mais conveniente. Assim, esses parâmetros não precisam ser adaptados.

Para realizar a adaptação estrutural primeiro é preciso entender que um plano pode ser representado por um grafo que contém macro ações, pré-condições e condições de sucesso, onde as macro ações são os nós e se uma macro ação depender do resultado de outra existe uma aresta entre eles.. A Figura 4.4 mostra um exemplo desse tipo de grafo. Os quadrados cinza representa as ações, os quadrados brancos as pré-condições. As condições de sucesso vai ser cada unidade sendo criada ou treinada por cada ação, existir ao final do processo.

Um plano pode falhar no meio da execução quando uma ação não consegue atingir as condições de sucesso ou uma pré-condição. Por isso é necessário adaptar o plano pra obter novos trechos de plano que resolva o problema. Por exemplo, para a pré-condição de ter 150 minerais para a construção da barraca, é enviado uma requisição para o módulo de recuperação de planos para achar um trecho que consiga resolver esse problema. Caso após um tempo ainda haja falha pra satisfazer uma pré-condição então todo o plano é descartado e o módulo de recuperação tenta buscar outro plano pra satisfazer o objetivo.

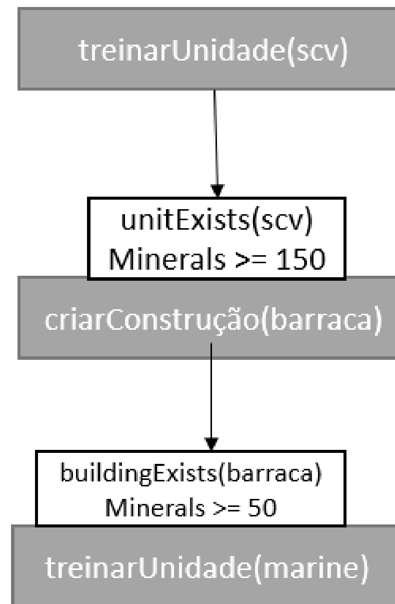


Figura 4.4: Exemplo de um plano para o treinamento de uma unidade do tipo *Marine* mostrando as macro ações e pré-condições.

4.3 Sistema de combate usando mapas de influência

Nessa seção será explicado como mapas de influência extraídos das partidas contidas na base de dados *Stardata* são utilizados para auxiliar no sistema de combate. A base de dados contém informações relevantes de eventos do jogo capturados a cada 3 *frames*. Usando a informação da posição das unidades, seu tipo e quantidade de vida é possível criar vários mapas de influência que representam o estado do jogo em determinado instante. Esses mapas são armazenados em uma base de dados juntamente com informação das ações executadas pelas unidades. Durante o jogo o bot cria um mapa de influência, pesquisa na base de dados qual é o mapa mais similar e imita as ações executadas pelas unidades. Com isso o bot pode ser simplificado como um problema de criar uma base de dados com mapas de influência e implementar algoritmos de imitação. O método proposto é mostrado de forma resumida na Figura 4.5.

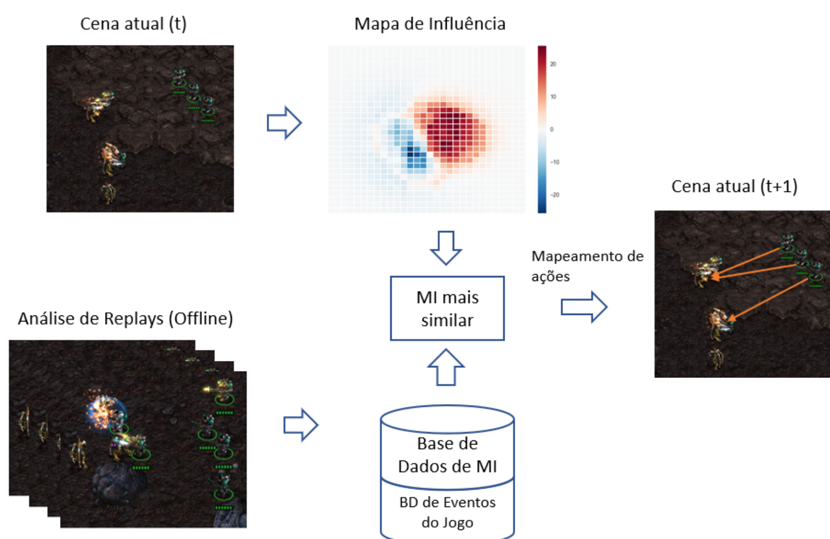


Figura 4.5: Resumo do método proposto.

Tabela 4.1: Eventos extraídos da base de dados.

ID	Tipo	Posição	Vida	Comando	Direção
0	Marine	120,200	40	Move	Right
1	Marine	120,212	40	Move	Right-Up
2	Siege Tank	112,192	150	Attack	Left-Up
...	...	...	...	..	...

4.3.1 Criação dos mapas de influência

A base de dados *Stardata* possui informação de eventos da partida exemplificado na Tabela 4.1. Além de informações básicas como o tipo e posição da unidade também possui informação do comando e direção dados a cada unidade. Além disso, há uma base de dados separada que contem informações de onde ocorrem batalhas em cada partida em um área de 200x200, as unidades envolvidas e por quantos frames a batalha dura. Informação que ajuda na criação dos mapas de influência explicado adiante.

Mapas de influência (MI) são baseados no conceito de que objetos no jogo criam uma zona de influência de acordo com sua força e essa influência está presente na sua posição atual e ao seu redor. Cada unidade do jogo vai influenciar diferentemente o MI levando em consideração o seu alcance de ataque, a quantidade de dano por segundo (DPS), sua velocidade de movimento e sua vida atual. Por exemplo, a Figura 4.6 mostra o campo de influência criado pela unidade do tipo Marine. Essa unidade possui alcance de ataque de valor 4 e DPS igual a 9.5. Consequentemente em um raio de 4 células possui uma influência de valor 9.5. Além disso, a unidade pode se movimentar. Logo, a influência da unidade deve se espalhar para além do seu alcance de ataque. O decaimento

da sua influência é determinado pela sua velocidade de movimento: quanto mais rápido a unidade menos decaimento sofrerá. Outra variável a ser considerada é a vida, se a unidade possui 80% sua influência será reduzida para apenas 80%. A fórmula 4.2 mostra como o mapa de influência é calculado para cada unidade de acordo com sua posição, dano por segundo, velocidade de movimento e vida atual.

$$MI(i,j) = \begin{cases} unit_dps \times \frac{current_health}{maximum_health} & \text{if } distance \leq \text{unit_range} \\ \frac{unit_dps}{falloff} \times \frac{current_health}{maximum_health} & \text{otherwise} \end{cases} \tag{4.2}$$

Onde a variável *falloff* é determinado pela a equação 4.3 onde a variável *steps* indica a distância além do alcance de ataque se encontra certa posição (i, j).

$$\begin{aligned} steps &= distance - unit_range \\ falloff &= (\frac{6}{unit_movement})^{steps} \end{aligned} \tag{4.3}$$

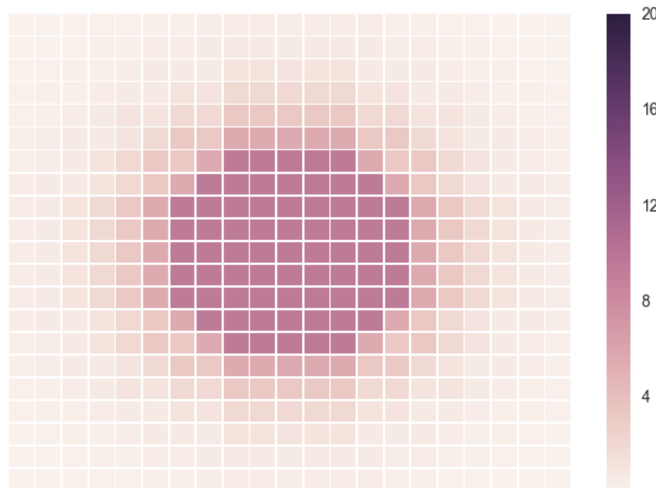


Figura 4.6: Mapa de influência criado pela a unidade Marine

A unidade do tipo *Siege Tank* cria um campo de influência interessante quando está em *siege mode* como mostrado na Figura 4.9. Normalmente a unidade *Siege Tank* possui um alcance de ataque de 7 e DPS igual a 19 porém em *siege mode* ele passa a ter alcance igual a 12, DPS igual a 22 e um alcance mínimo igual a 2. Como ele não pode atacar unidades muito próximas ele não projeta no centro um campo de influência. A unidade também não pode se mover nesse modo requerendo um longo tempo de transformação, por isso a projeção de sua influência fora do seu alcance de ataque é drasticamente reduzido.

A Figura 4.8 mostra a batalha entra 4 unidades do tipo *Zealot* e 5 *Marines* e o mapa de influência criado. Unidades inimigas tem o valor do seu campo influência invertido. O valor individual dos campos de influência de cada unidade é somado. *Zealots* são unidades

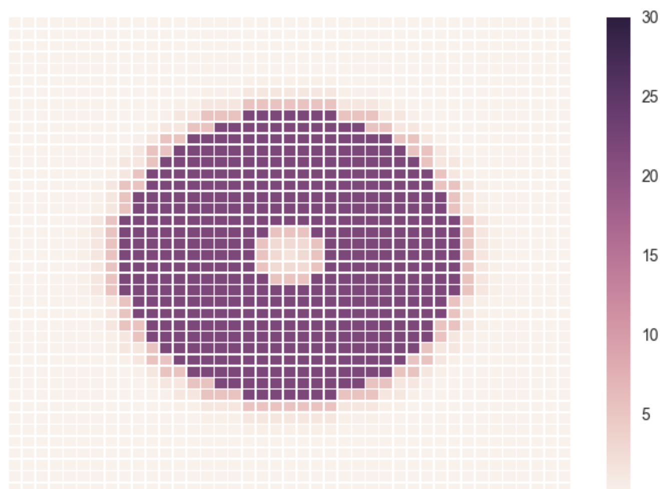
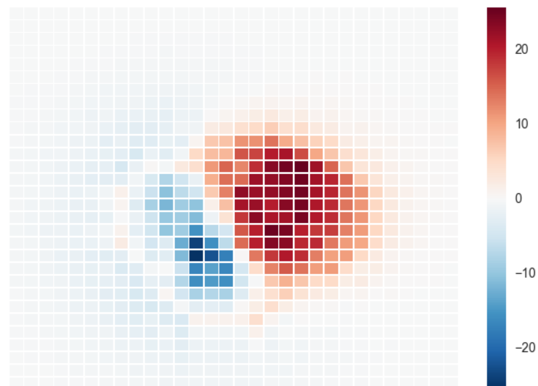


Figura 4.7: Mapa de influência criado pela a unidade *Siege Tank* em *Siege Mode*.

que apenas atacam corpo a corpo porém causam mais DPS que os *Marines* por isso seu campo de influência acaba eclipsando o campo dos dois *Marines* sendo atacados. Os mapas de influência são armazenados em uma base de dados juntamente com as ações executadas nos próximos 40 frames.

(a)



(b)

Figura 4.8: Uma batalha entre Marines e Zealots e o mapa de influência correspondente.

4.3.2 Processo de imitação

Recuperação de casos

Neste passo o bot produz um mapa de influência da situação atual do jogo. Depois, o MI atual é comparado com os outros MIs armazenados na base de dados. A comparação é feita calculando a distância euclidiana do mapa de influência atual com todos os outros mapas de influencia da base de dados, para realizar esse cálculo primeiro os pontos dos mapas de influência são considerados lineares e cada ponto do mapa representa uma coordenada. Como o mapa tem tamanho 200x200 então no cálculo da distância euclidiana irá ter 40000 coordenadas. A Fórmula 4.4 mostra como a distância euclidiana é calculada onde o primeiro número subscrito representa a coordenada do ponto no mapa de influência linearizado e o segundo número representa se é do primeiro ou segundo mapa de influência sendo comparado.

$$d = \sqrt{(p_{1,1} - p_{2,1})^2 + (p_{2,1} - p_{2,2})^2 ... + (p_{40000,1} - p_{40000,2})^2} \quad (4.4)$$

Os 5 MIs da base de dados com menor distância para o MI atual são escolhidos. Desses 5, o que tiver a composição mais similar, com unidades do mesmo tipo em quantidades similares, é escolhido para a etapa de imitação de ações.

Para tornar o preprocessamento mais rápido todos os MIs são armazenados em memória.Além disso é realizado o agrupamento de MIs de acordo com as composições mais comuns de exércitos. Por exemplo em uma partida de Terran contra Zergs é comum ver *Marines*, *Medics* e *Siege Tanks* contra *Zerglings*, *Mutalisks* e *Lurkers* então mapas de influência que contém confrontos entre esses tipos de unidades são agrupados conjuntamente. Assim, quando o bot produzir um mapa de influência que contém batalhas entre esses mesmos tipos de unidade, será feita a busca nesse subconjunto.

Imitação de ações

O MI mais similar na base de dados é recuperado junto com as ações executadas pelas unidades nos próximos 40 *frames*. Essas ações recuperadas são adaptadas pelo bot que tenta reproduzi-las pelas unidades atuais. As ações recuperadas de 6 unidades do mesmo tipo são agrupadas e o comando dado mais frequente, seja movimentar, atacar, etc, é reproduzido pelas unidades controladas atualmente pelo bot. Para escolher quais unidades executará essas ações é calculado a distância euclidiana entre as 6 unidades do MI recuperado e todas as unidades do mesmo tipo do MI atual aquelas que mais próximas reproduzirão a ação.

Caso a ação seja de movimento, o local de deslocamento é calculado com a média do deslocamento das 6 unidades. Caso não haja unidades do mesmo tipo, primeiro se

procura por unidades que se desloca no mesmo meio, ou seja, por terra ou pelo ar. Como unidades aéreas podem sobrevoar terrenos elevados essa priorização evita que, caso a ação dada seja de movimento, esse comando continue sendo reproduzido. E depois é priorizado unidades que tenha o alcance de ataque mais parecido, para assim evitar, por exemplo, que uma unidade que ataca corpo-a-corpo execute o comando de ataque que está longe a colocando em perigo desnecessariamente. E por último é priorizado unidades que possuam a quantidade máxima de vida mais similar.

4.4 Predição de resultados usando Redes Neurais Recorrentes

A metodologia usado para a predição dos resultados consistem em treinar uma Rede Neural Recorrente (RNR) com dados de uma base de partidas de Starcraft e avaliando sua performance em predizer o resultado da partida. Primeiro foram feito testes preliminares para definir a arquitetura da rede neural. Depois disso, foram implementados três tipos de redes neurais recorrentes: uma RNR simples, uma LSTM e uma GRU. A rede com o melhor resultado foi treinada novamente, dessa vez utilizando intervalos de tempos diferentes para poder investigar como dados de diferentes períodos da partida influencia a acurácia da predição.

O *dataset* usado neste trabalho está disponível no Github[1] e contém informação de 65646 *replays* de partidas de Starcraft contendo 1535 milhões de frames e 496 milhões de ações de jogadores. O estado do jogo foi gravado a cada 3 quadros o que é perfeito para o uso em diferentes tarefas de aprendizado de máquina como classificação de estratégia, aprendizado por reforço, aprendizado por imitação e outros [Lin et al., 2017].

A base de dados possui partidas que cobrem todas as combinações de confrontos entre raças possíveis. A tabela 4.2 mostra a quantidade de partidas por confronto. A base possui informações gerais da partida como as raças envolvidas, o mapa escolhido e os jogadores participantes além de informações específicas do que acontece no decorrer do jogo como a quantidade de minerais e gás, quantidade de unidades, o tipo de cada uma, sua vida, escudo, posição e etc.

Para esse trabalho foi utilizado 27550 partidas do dataset onde as partidas possuiam uma média de 15 minutos de duração. Devido a natureza sequencial dos dados uma rede neural recorrente foi escolhido para a tarefa de predizer o resultado de partidas dado o que acontece em um minuto específico do jogo. Para a base de dados

[1]https://github.com/TorchCraft/StarData

Tabela 4.2: Número de jogos por confronto na base Stardata. Legenda: P = Protoss, T = Terran, Z = Zerg

PvZ	TvZ	PvT	TvT	PvP	ZVZ
18016	14531	17385	2550	7015	6149

poder ser usada na RNR antes foi feito pré-processamento da informação reorganizando em instâncias que representam um minuto especifico de um jogo. Para cada partida foi criado N instâncias onde N é a quantidade de minutos de uma partida. Informações gerais da partida como as raças envolvidas, jogador e mapa foram mantidos em todas as instâncias da mesma partida. Os dados coletados a cada minuto da partida e armazenado em dimensões diferentes em cada instância foram:

- Quantidade de minério e gás.
- Quantidade de unidades e a capacidade máxima.
- Tipo de unidade e sua quantidade.
- Soma da vida máxima de unidades do mesmo tipo.
- Soma da vida atual de unidades do mesmo tipo.
- Tipo de construção e a quantidade.
- Soma da vida atual de construções do mesmo tipo.

No final do pré-processamento a nova base de dados possuía 413250 instâncias e 244 dimensões. As RNR foram implementadas em Python usando as biliotecas Keras e Theano.

A Figura 4.10 mostra o ciclo de execução do UFMGBot como um todo que é uma adaptação do ciclo de planejamento online com a adição dos módulos de combate e predição. A parte de predição não está integrado ao módulo, a idéia é que como trabalho futuro ele possa ser utilizado para auxiliar na decisão dos casos a serem recuperados de acordo com o resultado predito. O módulo de combate é chamado sempre que uma unidade inimiga é encontrada por outra, assim o mapa de influência é gerado e comparado com os mapas da base de caso e para obter as ações a serem executadas.

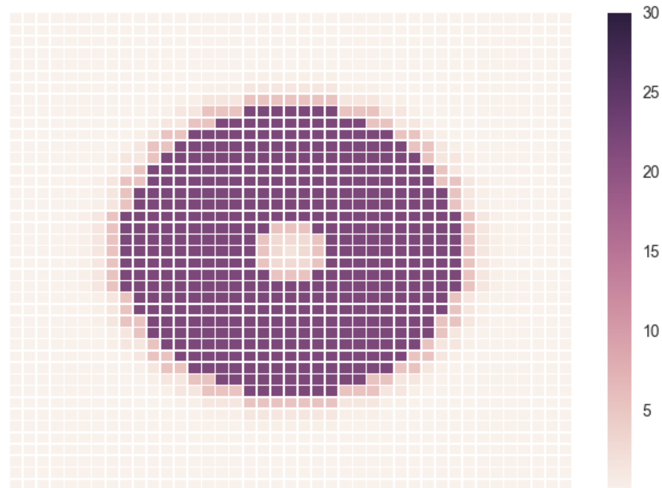


Figura 4.9: Mapa de influência criado pela a unidade *Siege Tank* em *Siege Mode*.

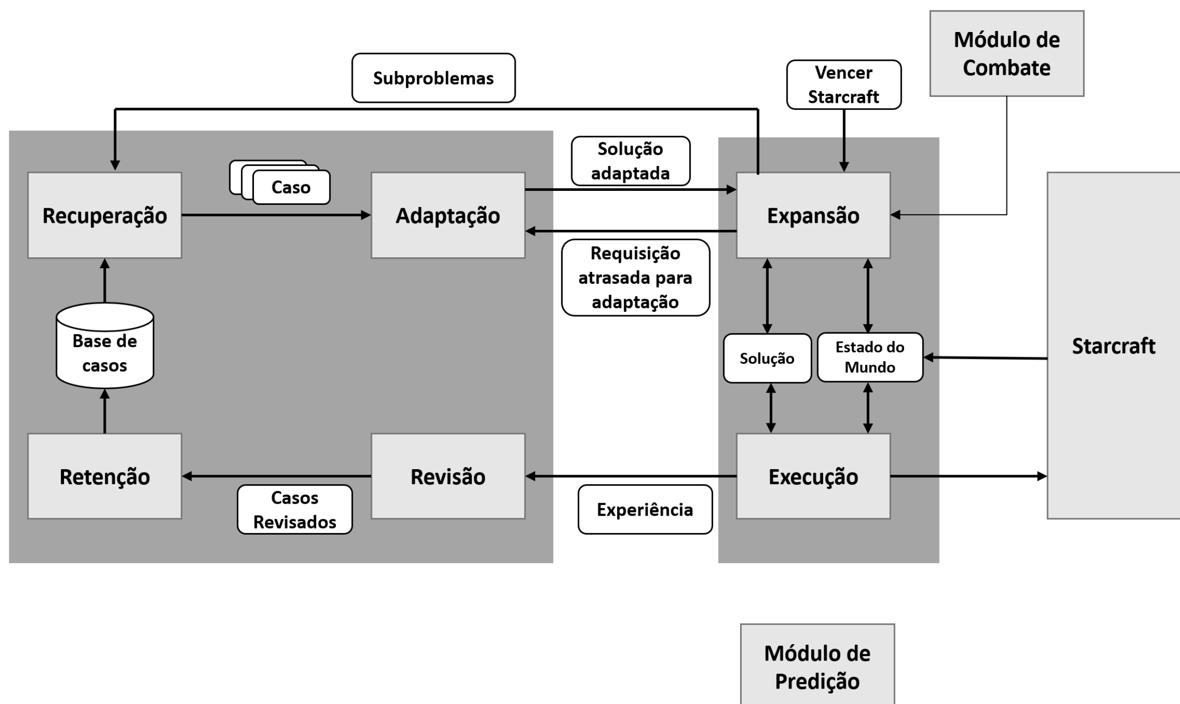


Figura 4.10: Ciclo de execução do UFMGBot

Capítulo 5

Experimentos e Resultados

5.1 Mapas de Influência

Para avaliar a performance do uso de base de dados de mapas de influência, foi feita a comparação com a implementação uilizada como referência para esse trabalho descrito em Oh et al. [2014]. A diferença entre os dois trabalhos é que para o UFMGBOT são levadas em consideração as ações em conjunto da tropa ao invés do controle de unidade por unidade como foi feito em Oh et al. [2014]. Para realizar a comparação foi utilizado o modelo de torneio de micro-gerenciamento de Starcraft da conferência *Artificial Intelligence and Interactive Digital Entertainment* (AIIDE). Foram utilizados 40 partidas jogadas por humanos em situação de combate entre tropas do tipo *Dragoons* da raça *Protoss* para a criação da base de dados de mapas de influência resultando em 14647 mapas de influência. Foram realizadas 20 partidas de batalhas entre a mesma unidade da raça Protoss chamada de *Dragoons* com tamanho 12 vs. 12 e 24 vs. 24 contra a IA padrão do jogo e foi obtida a média de vitórias. A Tabela 5.1 compara os resultados.

Tabela 5.1: Comparação da taxa de vitória entre os dois agentes.

Agente	**12 vs. 12**	**24 vs. 24**
Oh et al. [2014]	70%	75%
UFMGBOT	78.5%	84%

Como pode ser visto o agente UFMGBOT obteve melhores resultados, uma possível explicação é que o agrupamento de unidades permitiu que as tropas tivessem maior foco de ataque em inimigos específicos, os eliminando mais rápido, além de melhor se posicionarem evitando que tropas ficassem isoladas sendo um alvo mais fácil para o inimigo.

5.2 Planejamento Online Baseado em Casos

Para avaliar a performance de POBC, é feito a comparação com a implementação original de POBC proposto em Ontanón et al. [2010]. Originalmente o POBC foi utilizado para criar um agente inteligente capaz de jogar Warcraft II, chamado de Wargus. A diferença da implementação são que em Ontanón et al. [2010] as ações são consideradas individualmente ao invés de macro ações e as sequências de ações das partidas são anotadas manualmente para indicar o objetivo sendo perseguido naquele momento ao invés de ser feito de forma automática.

Foram realizados 48 partidas contra a IA padrão nos dois trabalhos e obtido a média de vitórias. Foram utilizados diferentes números de partidas como demonstração para extração de casos. Para Wargus, foram feitos testes utilizando até 10 partidas de demonstração para obtenção de casos, resultando em uma taxa de vitória média igual a 41,67%. Para esse trabalho utilizando 10 partidas de demonstração foi obtida uma taxa de vitória de 36.25%. Porém vale levar em consideração que as anotações sobre objetivos sendo perseguidos foi feito automaticamente como descrito na Seção 4.2.2 o que leva a mais imprecisão quando a comparada a forma manual ao qual o especialista pode ser muito mais preciso porém demanda muito mais tempo, podendo levar até 10 minutos por partida, enquanto que o processo automático ocorre em segundos. A tabela 5.2 compara a média de vitória pro agente Wargus utilizando de 5 a 10 demonstrações e para o agente desse trabalho de 5 a 50 demonstrações.

Tabela 5.2: Comparação da taxa de vitória para diferentes números de partidas de demonstração.

Num. Demo. (Wargus/UFMGBOT)	**Wargus**	**UFMGBOT**
5/5	22.91%	16.37%
6/10	29.17%	36.25%
7/20	41.67%	53.18%
8/30	31.25%	67.56%
9/40	43.75%	78.83%
10/50	41.67%	85.17%

No trabalho de Ontanón et al. [2010] é notado que o maior problema com o agente Wargus é sua habilidade com o microgerenciamento de unidades e que a taxa de vitória se mantém próximo dos 40% devido a isso. O agente UFMGBOT não sofre desse problema devido ao microgerenciamento ser feito pelas macro ações e pela base de dados de mapas de influência. Com isso, é possível perceber a evolução da taxa vitória com o aumento do uso de partidas como demonstração para a extração de casos.

5.3 Predição do Resultado de Partidas

Como comentado anteriormente foram utilizadas rede neurais recorrentes para a predição do resultado de partidas de acordo com informações da base de dados STARDATA. Porém existem diversos tipos de RNRs como mostrado na Seção 3.4, por isso foi necessário fazer experimentos para saber qual era a mais adequada para a tarefa e qual topologia a ser usada. Para achar a melhor topologia da rede foram feitos testes empíricos utilizando uma RNR simples, inicialmente com redes com complexidade alta com duas camadas ocultas e 32 neurônios em cada. Essa rede acabou não tendo boa acurácia por isso a complexidade foi diminuída até obter resultados satisfatórios. A entrada da rede vai ser passada em blocos de dados referentes a 5 minutos da partida, representado pelo time steps na configuração da topologia da rede abaixo. Os dados são passados a cada minuto onde nos primeiros 5 minutos que não há informação dos 5 minutos anteriores é feito uma cópia do primeiro minuto da partida. A informação de um minuto específico da partida é representado por um vetor de 244 dimensões como discutido na seção 4.4. Além disso os dados foram normalizados para valores entre 0 e 1. A saída da rede é composta de duas unidades correspondendo a porcentagem de vitória e de derrota. A topologia final da rede se encontra abaixo, e foi utilizada em todos os outros experimentos.

- Uma camada oculta com 16 unidades recorrentes
  - ⋆ Regularizador: Dropout 0.2
- Uma camada de saída com duas unidades
  - ⋆ Ativação: Softmax
- time steps: 5
- Função de perda: Binary Cross-Entropy
- Otimizador: RMSProp

Depois desses experimentos preliminares, foram implementados diferentes tipos de RNR: LSTM, GRU e uma Rede Neural Recorrente (RNR) Simples usando um batch de tamanho 64 e 50 épocas de treinamento. Para o experimento, 67% das partidas foram escolhidas para treino e 33% para teste. Dos 33% apenas dados referentes aos minutos 3 a 6 foram escolhidos devido a serem dados intermediários da partida onde não se tem nem pouca informação para realizar uma predição no começo da partida e nem também muita informação no final o que tornaria a predição mais fácil com o jogador ganhador obtendo vantagens de ouro, gás, unidades etc. Cada teste foi executado 5 vezes e a

Tabela 5.3: Resultados comparando diferentes RNRs para o intervalo de tempo entre 3 e 6

Rede	RNR Simples	GRU	LSTM
Acurácia	67.32%	64.78%	62.81%
Desvio Padrão	1.04%	1.29%	1.42%

Tabela 5.4: resultado obtido para diferentes intervalos de tempo usando uma RNR simples

Minutos	0-3	3–6	6-9	9-12
Acurácia	67.76%	70.95%	79.19%	86.48%
Desvio Padrão	1.63%	1.31%	1.04%	1.08%

média e o desvio padrão se encontra na Tabela 5.3. Nós podemos observar que a Rede Neural Recorrente Simples obteve uma maior acurácia. Isto provavelmente se deve ao problema não ser tão complexo. Redes mais complexas como GRU e LSTM podem requerer mais dados para uma melhor performance. A Figura 5.1 mostra que a topologia para este tipo de rede não estava causando overfitting uma vez que o erro de treino e teste se manteve decrescente. Quando há overfitting é esperado que o erro de teste aumente enquanto o erro de treino diminui.

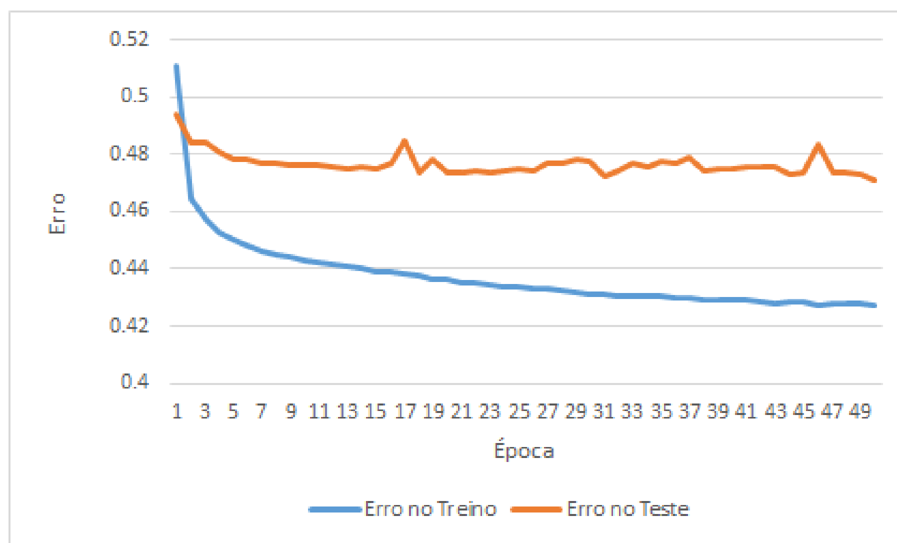


Figura 5.1: Comparação entre o erro de treino e teste para a RNR simples.

Uma vez que a RNR simples obteve um melhor resultado, ela foi escolhida para os próximos experimentos que avalia o impacto de treinar a RNR com dados de diferentes intervalos de tempo do jogo. Foram usados 4 diferentes intervalos de tempo: de 0 a 3, 3 a 6, 6 a 9 e 9 a 12. Lembrando que o tempo médio das partidas na base de dados é de 15 minutos. O treinamento foi feito usando validação cruzada k-fold com k igual a 5. Embora na base de dados cada instância represente um minuto em uma partida, a divisão entre treino e teste foi feita considerando partidas inteiras para que o teste seja feito com partidas que a rede não tenha visto antes. O resultado é mostrado na Tabela 5.4.

A acurácia aumentar com o passar do tempo de jogo é esperado pois o time vencedor vai ganhando vantagens como maior número de *workers*, recursos coletados, tropa de combate e etc. Eessa diferença entre jogadores ajuda a rede neural a realizar a classificação.

5.4 Avaliação do bot

O bot foi configurado para jogar como Protoss e para avaliar a performance do bot, foram feitas partidas contra 6 bots participantes da competição entre bots da AIIDE 2017. Os bots escolhidos foram: ZZKBOT, PurpleWave, IRON, UalbertaBot, KillAll, ZiaBot. Eles se encontraram na posição: 1, 2, 3, 14, 18 e 29 na competição respectivamente. Foram feitas 30 partidas contra cada um e obteve-se a taxa de vitória mostrada na Tabela 5.5. As raças correspondem a Terran (T), Protoss (P) e Zerg (Z)

Tabela 5.5: Taxa de vitória contra bots da competição AIIDE 2017.

Bot (Rank) (Raça)	**Taxa de vitória**
ZZZKBOT (1) (Z)	32.7%
PurpleWave (2) (P)	35.3%
IRON (3) (T)	36%
UalbertaBot (14) (P)	60.7%
KillAll (18) (Z)	78.1%
ZiaBot (29) (Z)	87.5%

Como pode-se obervar o bot não foi tão bem contra os melhores bots da competição porém foi competitivo contra bots que se encontram do meio da tabela pra baixo. As figuras 5.2, 5.3 e 5.4 mostram dados de uma partida contra o ZZZKBOT obtidas pelo programa *BWChart* onde o UFMGBOT perdeu. A Figura 5.2 mostra os recursos obtidos pelos dois bots onde as cores mais fracas representam o UFMGBOT e as cores mais fortes representam o ZZZKBOT. O azul representando a quantidade de minerais, verde a quantidade de gás e vermelho a quantidade de unidades produzidas. Como pode-se perceber, o UFMGBOT foi capaz de coletar mais minerais porém produziu menos unidades e recolheu menos gás. As outras duas imagens explica parcialmente porque isso pode ter acontecido.

A Figura 5.3 mostra a quantidade de unidades criadas por cada bot. Percebe-se a grande quantidade de *Zerglings* criados pelo o ZZZKBOT e poucos drones. Já o UFMGBOT possui muitos **Probes** e poucas unidades de ataque: apenas 9 *Zealots*, 4 *Corsairs*, unidade aérea que foi criada para combater os *Mutalisks*, unidade aérea dos

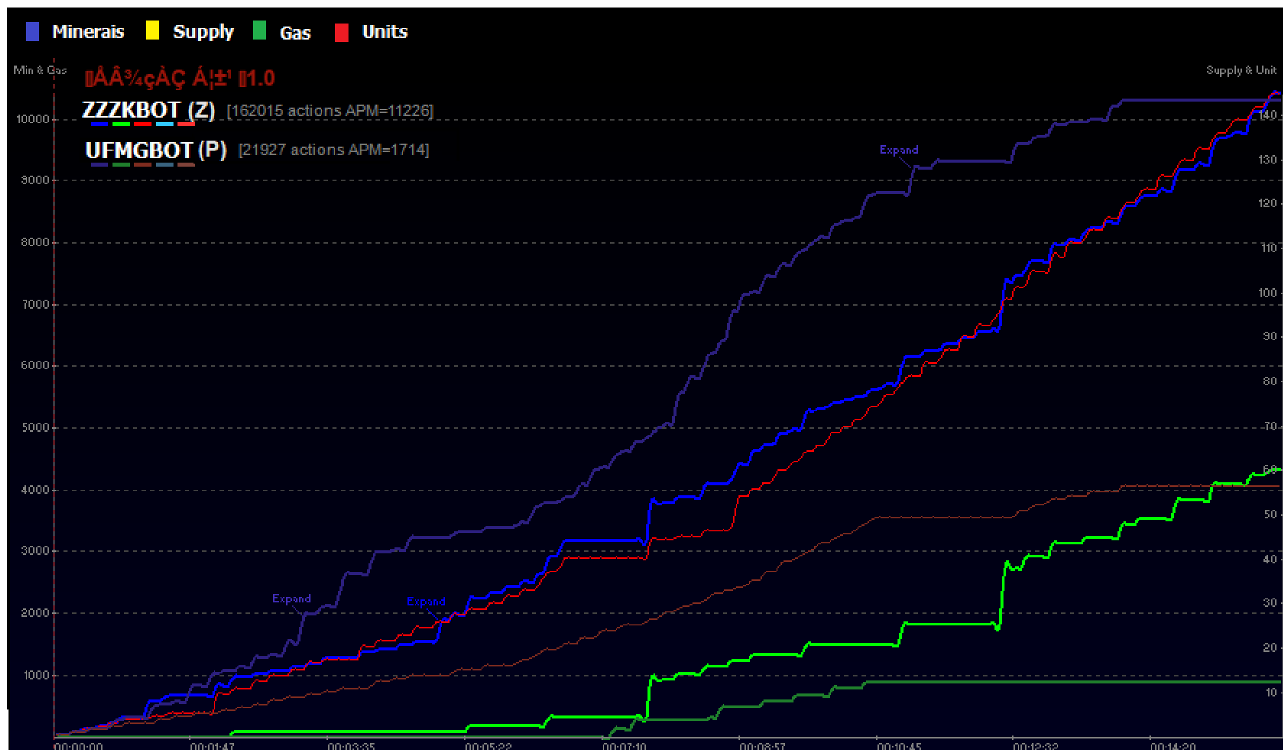


Figura 5.2: Comparação entre os recursos coletado pelo bot criado contra o ZZZKBOT

Zergs. A estratégia adotada pelo UFMGBOT só é revelada na Figura 5.4. É mostrado que o UFMGBOT optou por utilizar uma grande quantidade de defesa estática na forma de *Photon Cannons*. Isso levou o ZZZKBOT a sofrer as grandes perdas de *Zerglings* se suicidando para os *Photon Cannons* o que explica os 294 *Zerglings* criados. Após um tempo o ZZZKBOT foi capaz de adaptar a estratégia e mudou para a construção de unidades aéreas, o UFMGBOT foi capaz de perceber isso e criar os *Corsairs* mas acabou perdendo.

Já as figuras 5.5, 5.6 e 5.7 mostram dados de partida contra o UAlbertaBot onde o UFMGBOT ganhou. Ambos jogaram de Protoss. Na Figura 5.5 as cores mais fortes representam o UFMGBOT e as mais fracas o UAlbertaBot. Pode-se perceber que o UFMGBOT conseguiu coletar mais minerais e gás e produzir mais unidades do que seu adversário. A Figura 5.6 mostra que o bot produzido nesse trabalho construiu mais do que o dobro de *Probes* para a coleta de recurso, o que explica a maior quantidade coletada.

Além disso, percebe-se que enquanto o UAlbertaBot construiu apenas *Dragoons*, o UFMGBOT construiu inicialmente *Zealots* e depois *Dragoons*. Esses *Zealots* foram importantes para garantir a vitória pois foram responsáveis por destruir vários *Probes* do time inimigo no começo da partida. A Figura 5.7 mostra que o UFMGBOT avançou mais na árvore de tecnologia criando construções mais avançadas e que possibilitam obter melhorias de ataque e defesa.

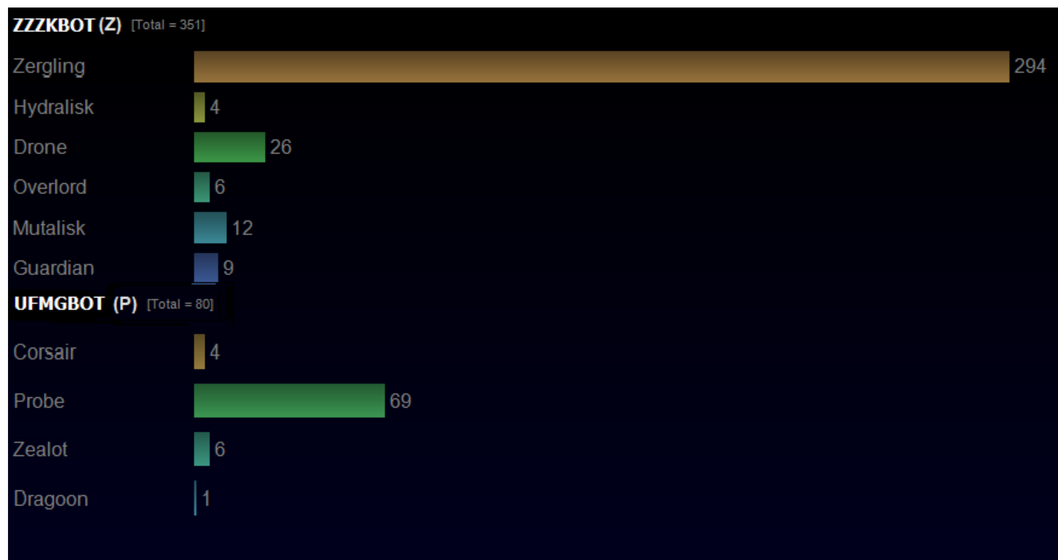


Figura 5.3: Comparação entre a quantidade de unidades criadas em partida do bot criado contra ZZZKBOT

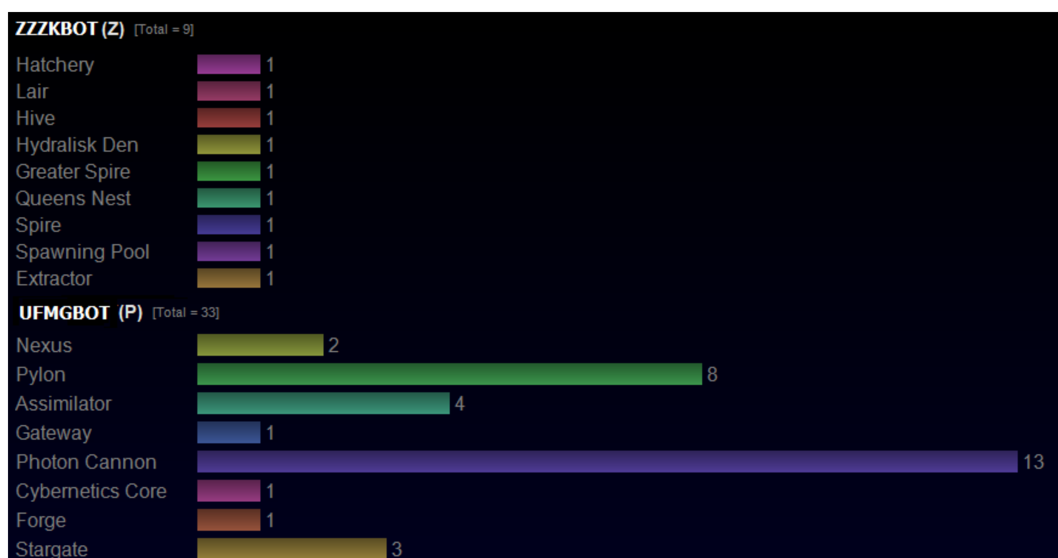


Figura 5.4: Comparação entre a quantidade de construções criadas em uma partida do bot criado contra ZZZKBOT

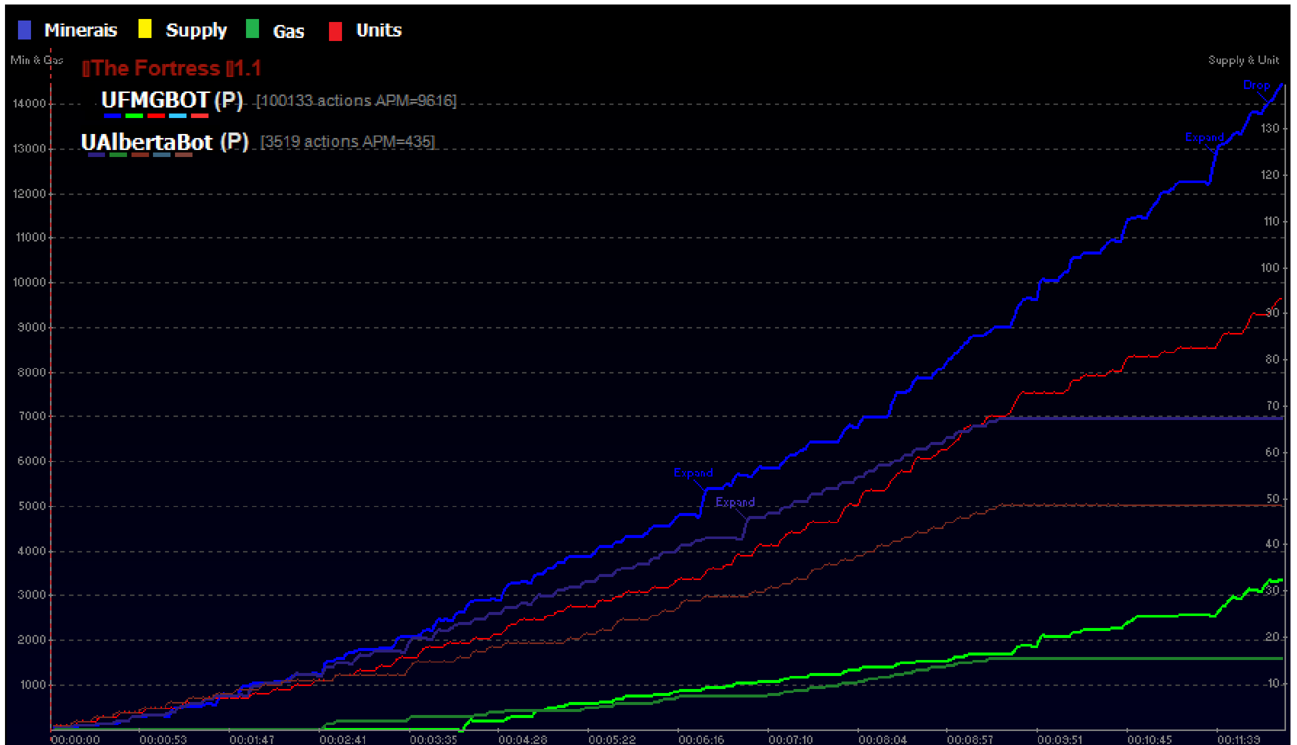


Figura 5.5: Comparação entre os recursos em partida do bot criado contra o UAlbertaBot

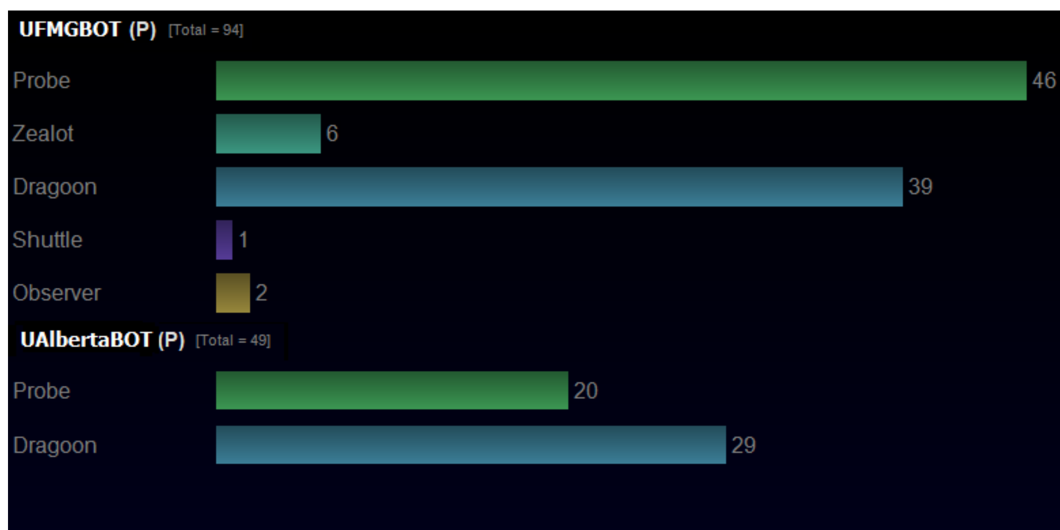


Figura 5.6: Comparação entre a quantidade de undiades criadas em partida do bot criado contra UAlbertaBot

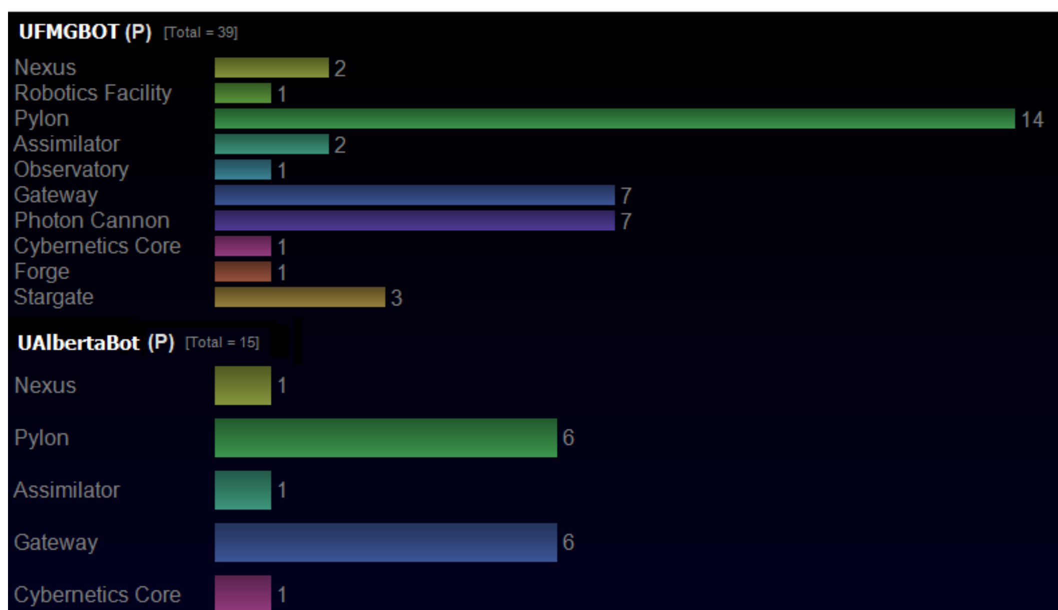


Figura 5.7: Comparação entre a quantidade de construções criadas em uma partida do bot criado contra UAlbertaBot

Capítulo 6

Conclusão

Nesse trabalho foram estudados e implementados várias técnicas de aprendizado por imitação para a criação de uma inteligência artificial capaz de jogar e prever resultados de partida de Starcraft. Aprendizado por imitação tem o objetivo de ensinar a inteligência artificial a executar alguma tarefa observando como humanos realizaram essa tarefa. Nesse sentido, foi utilizado a base de dados Stardata que contém mais de 65000 replays de partidas jogada por humanos. Como algoritmos foram utilizados Planejamento Online Baseado em Casos que ficou responsável pela gerenciamento econômico e estratégico da partida de Starcraft. Um módulo utilizando mapas de influência foi criado para gerenciar o combate. Além de um módulo de previsão continua do resultado da partida utilizando Redes Neurais Recorrentes.

O módulo de previsão continua foi testado usando três tipos de redes neurais recorrentes: uma RNR simples, uma GRU e uma LSTM. A RNR simples obteve melhores resultados e conseguiu predizer o resultado das partidas com 63% e 80% de acurácia dependendo de quão avançado na partida se estava. Os módulos de POBC e base de dados de mapas de influência foram comparados com as implementações que serviram de referência para esse trabalho, e a implementação do UFMGBOT obteve melhores resultados em ambos. Já o bot em si foi comparado com outros bots da competição da AIIDE 2017 e não conseguiu boas taxas de vitória contra os melhores bots da competição mas obteve bons resultados contra os outros selecionados para teste. O bot mostrou ser capaz de se adaptar as circunstâncias do jogo, mudando de estratégia quando necessário além de progredir na árvore tecnológica buscando melhorias de ataque e defesa das tropas.

Como trabalho futuro pode-se pensar em melhor integrar o módulo de predição da partida ao módulo estratégico permitindo adaptar a estratégia de acordo com a taxa de vitória prevista.

Referências Bibliográficas

Aamodt, A. Plaza, E. (1996). CBR: foundational issues, methodological variations and systems approach. AI Communications, 7(1).

Bakkes, S. Spronck, P. Van Den Herik, J. (2011). A CBR-inspired approach to rapid and reliable adaption of video game ai. Case-Based Reasoning for Computer Games Workshop at the International Conference on Case-Based Reasoning (ICCBR), 17--26. Citeseer.

Buro, M. (2004). Call for AI research in RTS games. Proceedings of the AAAI-04 Workshop on Challenges in Game AI, 139--142. AAAI press.

Buro, M. Furtak, T. (2003). RTS games as test-bed for real-time AI research. Proceedings of the 7th Joint Conference on Information Science (JCIS 2003), volume 2003, 481--484.

Cho, K. van Merrienboer, B. Bahdanau, D. Bengio, Y. (2014). On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. arXiv:1409.1259 [cs, stat]. arXiv: 1409.1259.

Farooq, S. Oh, I.-S. Kim, M.-J. Kim, K. (2016). StarCraft AI Competition: A Step Toward Human-Level AI for Real-Time Strategy Games. Ai Magazine, 37:102--106.

Gers, F. A. Schmidhuber, J. Cummins, F. (1999). Learning to forget: continual prediction with LSTM. 1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470), volume 2, 850--855 vol.2.

Hingston, P. (2009). A Turing Test for Computer Game Bots. IEEE Transactions on Computational Intelligence and AI in Games, 1(3):169--186. ISSN 1943-068X.

Kolodner, J. L. (1992). An introduction to case-based reasoning. Artificial Intelligence Review, 6(1):3--34. ISSN 1573-7462.

Laursen, R. Nielsen, D. (2005). Investigating small scale combat situations in real-time-strategy computer games. Master's thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark.

Lin, Z. Gehring, J. Khalidov, V. Synnaeve, G. (2017). STARDATA: A StarCraft AI Research Dataset. arXiv:1708.02139 [cs]. arXiv: 1708.02139.

Liu, S. Louis, S. J. Ballinger, C. (2014). Evolving effective micro behaviors in RTS game. 2014 IEEE Conference on Computational Intelligence and Games, 1--8.

Mehta, M. Ontanón, S. Amundsen, T. Ram, A. (2009). Authoring behaviors for games using learning from demonstration. Proceedings of the Workshop on Case-Based Reasoning for Computer Games, 8th International Conference on Case-Based Reasoning (ICCBR 2009), L. Lamontagne and PG Calero, Eds. AAAI Press, Menlo Park, California, USA, 107--116.

Oh, I.-S. Cho, H. Kim, K. (2014). Imitation learning for combat system in RTS games with application to starcraft. 2014 IEEE Conference on Computational Intelligence and Games, 1--2.

Ontanón, S. Kinshuk Mishra Sug, N. Ram, A. (2010). On-Line Case-Based Planning. Computational Intelligence, 26:84--119.

Ortega, J. Shaker, N. Togelius, J. Yannakakis, G. N. (2013). Imitating human playing styles in Super Mario Bros. Entertainment Computing Entertainment Computing, 4(2):93--104. ISSN 1875-9521. OCLC: 4912954722.

Osa, T. Pajarinen, J. Neumann, G. Bagnell, J. A. Abbeel, P. Peters, J. (2018). An Algorithmic Perspective on Imitation Learning. now.

Prado Prandini Faria, M. Maria Silva Julia, R. Bononi Paiva Tomaz, L. (2019). Evaluating the Performance of the Deep Active Imitation Learning Algorithm in the Dynamic Environment of FIFA Player Agents. 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), 228--233.

Sailer, F. Buro, M. Lanctot, M. (2007). Adversarial planning through strategy simulation. 2007 IEEE Symposium on Computational Intelligence and Games, 80--87. IEEE.

Schank, R. C. (1983). Dynamic Memory: A Theory of Reminding and Learning in Computers and People. Cambridge University Press, New York, NY, USA. ISBN 0-521-24858-2.

Shantia, A. Begue, E. Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in starcraft. The 2011 International Joint Conference on Neural Networks, 1794--1801. IEEE.

Silver, D. Schrittwieser, J. Simonyan, K. Antonoglou, I. Huang, A. Guez, A. Hubert, T. Baker, L. Lai, M. Bolton, A. others (2017). Mastering the game of go without human knowledge. Nature, 550(7676):354.

Tavares, A. R. Azpúrua, H. Chaimowicz, L. (2014). Evolving Swarm Intelligence for Task Allocation in a Real Time Strategy Game. 2014 Brazilian Symposium on Computer Games and Digital Entertainment, 99--108.

Tavares, A. R. Vieira, D. K. S. Negrisoli, T. Chaimowicz, L. (2019). Algorithm Selection in Adversarial Settings: From Experiments to Tournaments in StarCraft. IEEE Transactions on Games, 11(3):238--247. ISSN 2475-1510.

Team, T. A. S. (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II | DeepMind.

Vinyals, O. Ewalds, T. Bartunov, S. Georgiev, P. Vezhnevets, A. S. Yeo, M. Makhzani, A. Küttler, H. Agapiou, J. Schrittwieser, J. Quan, J. Gaffney, S. Petersen, S. Simonyan, K. Schaul, T. Hasselt, H. v. Silver, D. Lillicrap, T. P. Calderone, K. Keet, P. Brunasso, A. Lawrence, D. Ekermo, A. Repp, J. Tsing, R. (2017). StarCraft II: A New Challenge for Reinforcement Learning. CoRR, abs/1708.04782.

Weber, B. Ontañón, S. (2010). Using Automated Replay Annotation for Case-Based Planning in Games.

Weber B.G Mateas M Jhala A 2011 AAAI Fall Symposium (2011). Building human-level AI for real-time strategy games. AAAI Fall Symp. Tech. Rep. AAAI Fall Symposium - Technical Report, FS-11-01:329--336. ISSN 9781577355458. OCLC: 776642966.

Yannakakis, G. N. Togelius, J. (2015). A Panorama of Artificial and Computational Intelligence in Games. IEEE Transactions on Computational Intelligence and AI in Games, 7(4):317--335. ISSN 1943-068X.

Apêndice A

Exemplo de execução do UFMGBOT

Esse apêndice foi criado com o objetivo de mostrar um exemplo de execução do UFMGBOT para colocar em perspectiva como os subsistemas se comportam e como se comunicam entre si na ordem em que são executadas. A execução do algoritmo pode ser resumido na imagem A.1. E como exemplo utilizaremos uma partida do UFMGBot como Protoss contra o bot chamado de IRON jogando como Terran.

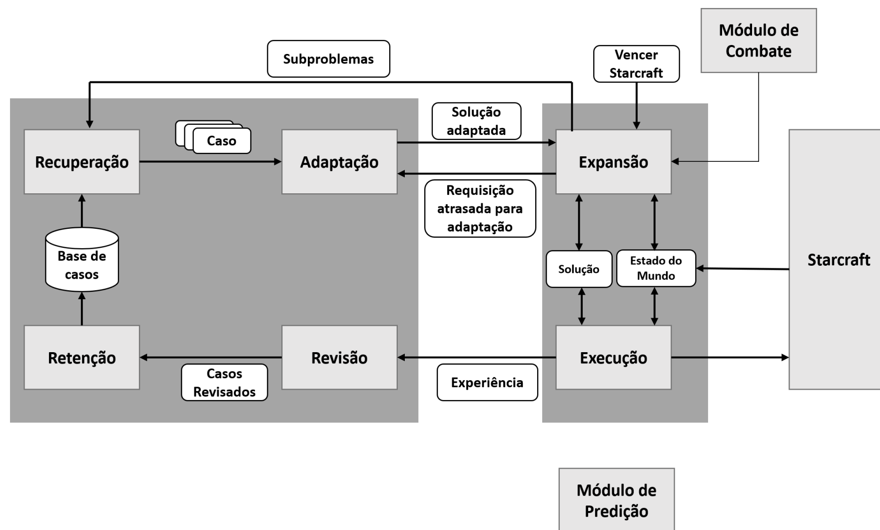


Figura A.1: Ciclo de execução do UFMGBot

O primeiro passo na execução do algoritmo é a criação das duas base de dados: uma contendo os casos e outra contendo os mapas de influência e as ações executadas. Para isso foi utilizado a base que contém replays de partidas chamado de STARDATA. É realizado a análise apenas de partidas de Protoss contra Terran pois é o confronto a ser realizado no momento. Para a criação da base de casos é analisado cada partida individualmente usando o algoritmo descrito na Seção 4.2.2:

$$P.S = S_t$$
$$P.G = grafo(A_t, A_{t+1}, ..., A + t + n)$$
$$P.O = categoria.O$$
$$P.O.parametros = computar_parametros(S_{t+n})$$

No fim se obtém vários grafos como mostrado na Figura A.2. E cada nó contém um trecho de plano associado a um episódio que indica o estado do jogo e o resultado da execução daquele trecho onde ele foi previamente executado, como mostrado na Figura A.3.

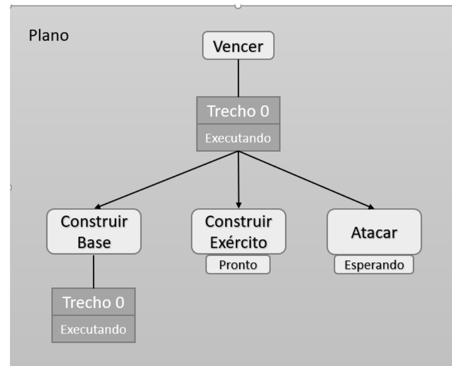


Figura A.2: Representação de um plano expandido.

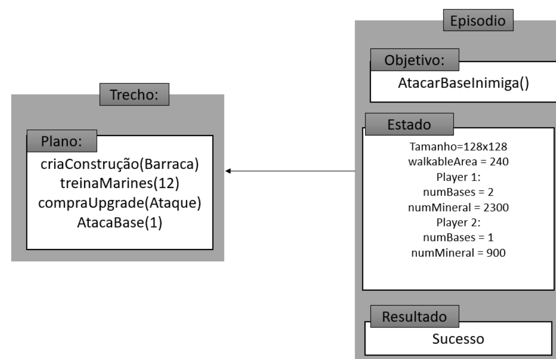
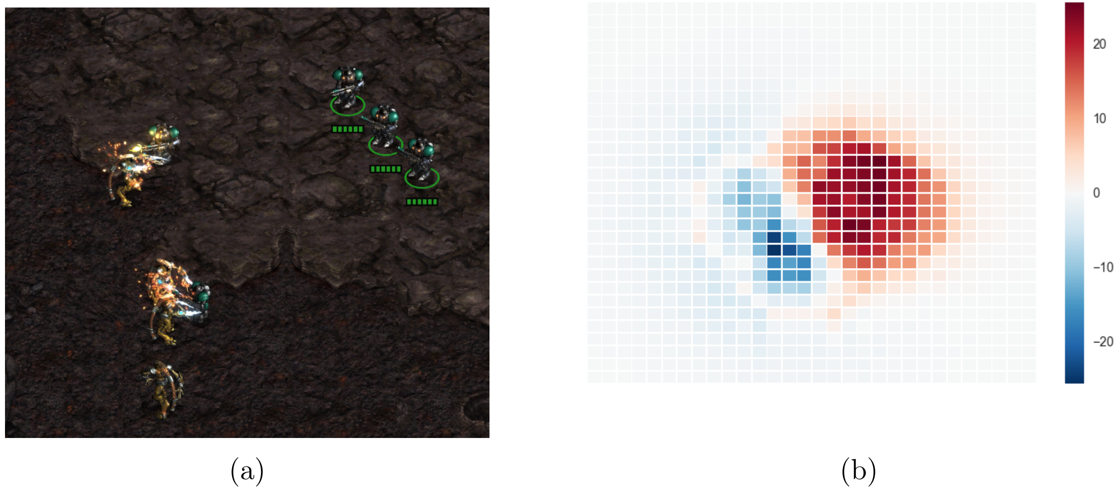


Figura A.3: Exemplo de um episódio e um trecho de plano.

Para se obter a base de dados com os mapas de influência também são analisadas as partidas da base de dados STARDATA e criados os mapas de acordo com as fórmulas apresentadas na seção 4.3.1, um exemplo de cena e o mapa de influência correspondente gerado pode ser visto na Figura A.4. Juntamente com os mapas de influência das cenas, também são armazenadas as ações que ocorreram nos 40 próximos frames.

Com as bases obtidas, agora é hora de analisar a execução do UFMGBot em uma partida. Primeiro, um plano é recuperado da base de casos que vai determinar a ação do UFMGBot nos próximos instantes, para fazer isso são usados os algoritmos mostrados na Seção 4.2.3 onde episódios associados a trechos de plano são comparados para obter o plano que mais se assemelha a situação do jogo. Para essa partida, o plano obtido dizia que deveria ser construído por dois *Gateways*, que são construções que permitem a criação de unidades de combates, e a partir desses *Gateways* construir unidades de combate do tipo *Dragoon*. A Figura A.5 mostra os dois *Gateways* construídos durante a partida. Ao terminar a execução do plano, o resultado é avaliado e se ocorreu tudo bem, novos



(a) (b)

Figura A.4: Uma batalha entre Marines e Zealots e o mapa de influência correspondente.

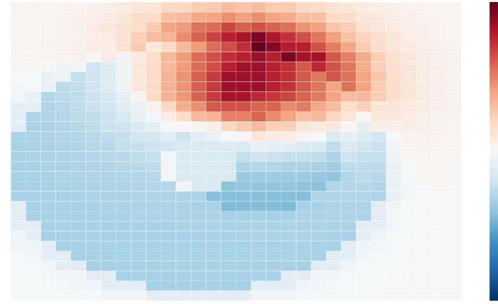
episódios serão criados e vinculados aos trechos de plano indicando o estado no qual esse trecho acabou de ser executado e o seu resultado.

Figura A.5: Construção dos 2 Gateways na partida

Depois é buscado na base um novo plano de execução, nesse exemplo foi obtido um plano para enviar um *Probe* vasculhar o mapa em busca da base inimiga, a construção de um *Templar Archives*. Essa construção permite a criação de *Dark Templars* que são unidades permanentemente camufladas. E também o plano obtido indica atacar a base inimiga. Após um tempo de jogo é realizado o ataque coma tropa de *Dragoons* a base inimiga. Ao encontrar a base inimiga o módulo de combate usando mapas de influência é acionado

A Figura A.6 mostra o encontro com a base a inimiga sendo defendida por um *Siege Tank* e um *Bunker* com três *Marines*. O bunker é uma construção que permite unidades entrarem dentro e aumentar o seu alcance de ataque. A Figura também mostra o mapa de influência gerado para o combate atual. As ações obtidas associadas ao mapa de

(a)



(b)

Figura A.6: Uma batalha entre Dragoons, Tanks e um bunker e o mapa de influência correspondente.

influência foram de atacar e recuar continuamente. Após executar as ações a unidade do tipo *Dark Templar* se junta a batalha e um novo mapa de influência é obtido e suas ações recuperadas, resultando em um ataque com sucesso com o *Dark Templar* conseguindo destruir o *Siege Tank*, resultando na vitória.