DAVID DA SILVA ALEIXO

APRENDENDO ESTRATÉGIAS PROGRAMÁTICAS ATRAVÉS DE CARACTERÍSTICAS COMPORTAMENTAIS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

Orientador: Levi Henrique Santana de Lelis

VIÇOSA - MINAS GERAIS
2023

Agradecimentos

Primeiramente, gostaria de declarar que este estudo foi parcialmente financiado pela Fundação de Amparo à Pesquisa de Minas Gerais (FAPEMIG) e pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES). Por isso, expresso a minha gratidão a essas instituições.

Eu gostaria de agradecer muito ao meu orientador e amigo, Levi Lelis, pela paciência e pela moral que me deu durante todos esses anos. Espero que nossa parceria dure por muito mais tempo.

Gostaria de demonstrar minha gratidão também à maioria dos professores e amigos que fiz durante minha vida acadêmica.

Além deles, não posso deixar de me sentir grato pelos amigos do grupo de pesquisa, em especial, ao Julian que, além de me receber com muito carinho desde o primeiro momento que nos conhecemos, também foi meu padrinho neste projeto; ao Leandro, que foi meu parceiro neste projeto; e ao Rubens, que me deu valiosos conselhos sobre o projeto e a vida acadêmica.

Por fim, gostaria de agradecer também a todos os amigos e familiares que estiveram comigo por toda esta jornada. E, para finalizar, gostaria de citar uma estrofe da música avuá feita por Kamu em colaboração com Emicida, Rael, Coruja BC1, Drik Barbosa e Fióti:

"Turbulência lapida o piloto
Pra chegar não é só por talento
Pra que alguns se sentissem no topo
Alguém tinha que ser fundamento"

É Noix.

Resumo

Aleixo, David da Silva, M.Sc., Universidade Federal de Viçosa, fevereiro de 2023. **Aprendendo estratégias programáticas através de características comportamentais**. Orientador: Levi Henrique Santana de Lelis.

Nos últimos anos, a pesquisa em síntese de estratégias programáticas tem despertado bastante interesse dos pesquisadores por gerar programas que podem ser interpretadas por humanos, sendo até mesmo possível modificá-los manualmente. Entretanto, sintetizar estratégias programáticas é uma tarefa desafiadora devido à necessidade de realizar uma busca não diferenciável em um grande espaço de programas. Deste modo, utiliza-se atualmente uma abordagem de *self-play* para guiar a busca. No entanto, essa abordagem geralmente fornece um fraco sinal para guiar a busca, uma vez que o *self-play* só é capaz de avaliar o desempenho de um programa em relação a outros programas. Assim, enquanto pequenas mudanças em um programa podem não melhorar o seu desempenho, tais mudanças podem representar passos na direção de um programa mais eficiente. Nessa dissertação, apresentaremos duas abordagens que utilizam características comportamentais para guiar a busca. A primeira utiliza as características comportamentais provenientes de um oráculo através do processo de clonagem comportamental para aprender um *sketch* de uma estratégia programática. Em nossos experimentos, observamos que até mesmos oráculos fracos podem prover informações úteis, ajudando na síntese. Testamos nossa proposta de aprendizado de *sketch* juntamente com os algoritmos de busca *Simulated Annealing* e UCT como sintetizadores. Com isso, tentamos sintetizar as melhores respostas aproximadas para uma estratégia tradicional do *Can't Stop* e para o vencedor da competição de 2020 do MicroRTS. Nosso método foi capaz de sintetizar estratégias programáticas mais fortes do que os oráculos originais e derrotaram as estratégias alvo. Em nossa segunda abordagem, propomos um algoritmo de busca hierárquica que busca concorrentemente no espaço de programas e em um espaço de características comportamentais. Nossa hipótese é que um algoritmo de *self-play* em conjunto com uma função baseada em características pode gerar um forte sinal para auxiliar a síntese. Enquanto ambas as funções são usadas para guiar a pesquisa no espaço do programa, a função de *self-play* é usada para guiar a pesquisa no espaço de características, com o objetivo de permitir a seleção de características com maior probabilidade de levar a programas vitoriosos. Testamos nossa hipótese

no MicroRTS, um jogo de estratégia em tempo real. Nossos resultados mostram que a busca hierárquica sintetiza estratégias mais fortes do que métodos que buscam apenas no espaço do programa. Além disso, as estratégias que nosso método sintetiza obtiveram a maior taxa de vitórias em um torneio simulado com vários agentes *baselines*, incluindo os melhores agentes das duas últimas competições do MicroRTS.

Palavras-chave: Inteligência Artificial, Síntese de Programas, Algoritmo de Busca, jogos.

Abstract

Aleixo, David da Silva, M.Sc., Universidade Federal de Viçosa, February, 2023. **Learning Programmatic Strategies Though Behavioral Features**. Adviser: Levi Henrique Santana de Lelis.

In recent years, research on the synthesis of programmatic strategies has aroused a lot of interest among researchers as it generates programs that can be interpreted by humans, and it is even possible to modify them manually. However, synthesizing programmatic strategies is a challenging task due to the need to perform a non-differentiable search on a large programs' space. Thus, a self-play approach is currently used to guide the search. Nevertheless, this approach generally provides a weak signal to guide the search, since self-play is only able to evaluate the performance of a program in relation to other programs. Thereby, while small changes to a program may not improve its performance, such changes may represent steps towards a more efficient program. In this dissertation, we will present two approaches that use behavioral features to guide the search. The first uses the behavioral features derived from an oracle through the process of behavioral cloning to learn a sketch of a programmatic strategy. In our experiments, we observed that even weak oracles can provide useful information, helping with the synthesis. We tested our sketch learning proposal with Simulated Annealing and UCT search algorithms as synthesizers. With this, we tried to synthesize the best approximate answers for a traditional Can't Stop strategy and for the winner of the 2020's MicroRTS competition. Our method was able to synthesize stronger programmatic strategies than the original oracles and to defeate the target strategies. In our second approach, we propose a hierarchical search algorithm that searches concurrently in the programs' space and in a space of behavioral features. Our hypothesis is that a self-play algorithm in conjunction with a feature-based function can generate a strong signal to aid synthesis. While both functions are used to guide the search in the program space, the self-play function is used to guide the search in the features' space, with the aim of allowing the selection of features that are more likely to lead to winning programs. We tested our hypothesis on MicroRTS, a real-time strategy game. Our results show that the hierarchical search synthesizes stronger strategies than methods that search only in the program space. Furthermore, the strategies that our method synthesizes

achieved the highest win rate in a simulated tournament with multiple baseline agents, including the best agents from the last two MicroRTS competitions.

Keywords: Artificial Intelligence, Program Synthesis, Search Algorithm, games.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

1.1 O problema e sua importância

A síntese de programas tem como objetivo sintetizar programas de modo a satisfazer uma determinada especificação (GULWANI et al., 2017), podendo ser empregada em diversos cenários. SINGH (2014) desenvolveu um sistema onde o sintetizador tenta ensinar programação como um professor, ajudando os alunos a escrever programas, fornecendo pareceres ao seu usuário. AHMED; GULWANI; KARKARE(2013) apresenta um algoritmo capaz de resolver problemas de programação e cria problemas similares. CHEUNG; SOLAR-LEZAMA; MADDEN (2012) desenvolveu um sistema de recomendação que combina síntese de programas com algoritmos de aprendizado de máquina.

Os métodos baseados em aprendizagem de máquina, sobretudo em redes neurais profundas, têm sido empregados com muito êxito nas soluções de vários problemas. Entretanto, essas soluções apresentam uma baixa interpretabilidade e transparência, caracterizando-se como uma solução baixa preta (CARVALHO; PEREIRA; CARDOSO, 2019). Tendo em vista que, em muitos casos - como problemas judiciais, problemas relacionados à saúde e decisões críticas - é necessário, além de uma resposta precisa, uma explicação satisfatória do porquê de tal resposta ter sido escolhida.

Em vista disso, a área de síntese de programas vem ganhando interesse dos pesquisadores por ser capaz de gerar programas interpretáveis, modificáveis e que podem ser usados, inclusive, para ensinar humanos. No entanto, a síntese destes programas é uma tarefa desafiadora, pois é necessário realizar uma busca em grandes espaços não diferenciáveis de programa. Algoritmos de busca são empregados na tarefa de navegar por estes espaços que são definidos por uma *domain-specific language* (DSL, em português “linguagem específica de domínio")(DEURSEN; KLINT; VISSER, 2000). Entretanto, casos em que o problema é de difícil resolução ou o tamanho do espaço de programa induzido pela gramática é grande, os algoritmos de busca provavelmente terão dificuldade para encontrar uma solução em tempo hábil.

Na síntese de programas de computador representando estratégias para jogos, atualmente se utiliza algoritmos *self-play* de modo a gerar um sinal capaz de guiar a busca, como o *Iterated-Best Response* (IBR), apresentado por (MARIÑO et al., 2021). Porém, muitas vezes, esse sinal tende a ser fraco. Isso ocorre devido a ele somente avaliar a utilidade de uma estratégia jogando-a contra outra estratégia. Assim, o *self-play* não consegue capturar pequenas mudanças que, apesar de não apresentarem uma melhora instantânea, podem ser passos importantes na direção de um programa vitorioso.

Nesta dissertação, apresentaremos duas abordagens que usam características comportamentais para ajudar a guiar a síntese de estratégias programáticas. Em nossa primeira proposta, mostraremos que o uso da clonagem comportamental para aprender *sketches* pode contribuir para se acelerar o processo de síntese. Investigamos a utilização dessa abordagem de aprendizado de *sketch* com sintetizadores empregando *Simulated Annealing* (SA) (KIRKPATRICK; GELATT; VECCHI, 1983) e UCT (KOCSIS; SZEPESVÁRI, 2006) como algoritmos de busca, e os avaliamos tentando calcular uma melhor resposta para uma determinada estratégia alvo em jogos de soma zero para dois jogadores. Especificamente, avaliamos nossos métodos no jogo de tabuleiro *Can't Stop* (KELLER, 1986) e no jogo de estratégia em tempo real MicroRTS (ONTAÑÓN, 2020). Mostramos que nossos métodos podem ser eficazes inclusive ao clonar o comportamento de jogadores fracos, como por exemplo um jogador que escolhe suas ações aleatoriamente para o jogo *Can't Stop*. Isso acontece, pois uma estratégia fraca pode ser capaz de transmitir informações que são úteis para a síntese de estratégias programáticas, como a estrutura do programa necessária para decidir quando parar de jogar no *Can't Stop* ou quando construir uma estrutura específica no MicroRTS.

Em seguida, avaliamos nossos métodos SA e UCT baseados em *sketch*, os quais denominamos de Sketch-SA e Sketch-UTC, respectivamente, tentado sintetizar as melhores respostas aproximadas para uma estratégia programática descrita em (GLENN; ALOI, 2009) para *Can't Stop* e para COAC, o vencedor da competição de 2020 do MicroRTS (ONTAÑÓN, 2020), em quatro mapas. Nosso Sketch-SA sintetizou estratégias fortes em todas as configurações testadas. Como destaque, sintetizou uma estratégia que derrota o COAC em grandes mapas de MicroRTS; sendo que nenhuma das estratégias sintetizadas pelas *baselines* foi capaz de derrotar o COAC em mapas grandes.

Em nossa segunda abordagem, apresentamos um algoritmo para sintetizar estratégias programáticas que buscam em um espaço hierárquico de dois níveis, chamado *Hierarchical Synthesis Search* (HSS). Nosso objetivo com essa busca hierárquica é melhorar o sinal de busca dos algoritmos de *self-play*, como o IBR, amplamente utilizados em jogos de soma zero para dois jogadores.

O HSS busca concorrentemente em um espaço de programa definido por uma DSL e no espaço de características comportamentais do jogo que são dependentes do domínio. Em cada iteração da busca, é selecionado, no espaço de características, um conjunto de características que será usado juntamente com um algoritmo **self-play** para guiar a busca no espaço de programas. Todas as avaliações realizadas pelo algoritmo de **self-play** durante a busca no espaço de programas são utilizadas para informar a busca no espaço de características, ou seja, a busca no espaço de características tenta aprender a combinação de valores de características que leva a programas que otimizam a função de **self-play**. A hipotese é que essa hierarquia de dois níveis permite uma busca mais informada no espaço de programas do que a busca com apenas o sinal dos algoritmos **self-play**.

Assim sendo, modelamos o problema de busca no espaço de características como um problema combinatorial **multi-armed bandit** (CMAB) (GAI; KRISHNAMACHARI; JAIN, 2010) e implementamos uma versão de duas fases do **Naïve Sampling** (NS) para aproximar uma solução para ele (ONTAÑÓN, 2017a). Para realizar a busca no espaço de programas empregamos nossa proposta anterior Sketch-SA. Por fim, avaliamos nossa implementação de HSS no MicroRTS.

Nossos resultados mostraram que o HSS é capaz de superar consideravelmente outros sistemas que buscam apenas no espaço de programas, quando aplicado juntamente com o IBR, o *Fictitious play* (BROWN, 1951) ou outro algoritmo *self-play*. Também simulamos um torneio de MicroRTS, onde avaliamos os programas sintetizados em HSS, bem como várias outras *baselines*, incluindo as estratégias programáticas, escritas por programadores humanos, que venceram as duas últimas competições de MicroRTS (ONTAÑÓN, 2017b). Sendo que, o HSS com IBR e FP ficaram em primeiro e segundo lugar respectivamente em nosso torneio simulado.

2 Referencial Teórico

2.1 Clonagem Comportamental

Dado um oráculo para gerar uma lista de pares estado-ação, um agente de *Behavioral Cloning* (BC, em português "Clonagem Comportamental") aprenderá uma política a partir dos dados gerados pelo oráculo. Sendo $L = \{(s_i, a_i)\}_{i=1}^{m}$ o conjunto de dados de pares estado-ação produzidos pelo oráculo, um agente BC tenta aprender uma política π que mapeia cada s_i em L para a ação a_i que o oráculo realizaria.

Observa-se que a Clonagem Comportamental tem sido utilizada com sucesso em distintas aplicações. SAMMUT et al. (1992) experimentou dados humanos registrados com um programa de simulação de vôo como oráculo e empregou uma árvore de decisão para aprender uma política baseada nos dados do oráculo. ROSS; GORDON; BAGNELL (2011) apontaram uma fragilidade no método BC clássico: a política aprendida tinha dificuldades quando o agente enfrenta um estado que nunca tinha visto durante o estágio de aprendizado. Foi então proposto por eles o algoritmo chamado DAgger, o qual adiciona interativamente estados não vistos ao conjunto de dados original. Dessa forma, o oráculo rotula todos os estados recém-coletados e a etapa de aprendizado por imitação é repetida.

TORABI; WARNELL; STONE (2018a) propõem uma variação da configuração BC chamada *Behavioral Cloning from Observation* (BCO, em português "Clonagem Comportamental por Observação"), na qual o agente aprende uma política apenas a partir de observações. Nesse caso, o conjunto de dados utilizado é $L = \{(s_i)\}_{i=1}^{m}$. Dessa forma, foi proposta uma variação chamada BCO(α), a qual é um algoritmo de duas etapas: primeiro, tem-se a implementação proposta do BCO - executada para aprender uma política π -; e, segundo, tem-se a execução da política π no ambiente para coletar um novo conjunto de dados $L_\pi = \{(s_j, a_j)\}_{j=1}^{n}$. Posteriormente, a política aprendida L_π é atualizada usando a implementação BC clássica com o novo conjunto de dados L_π, onde os pares ação-observação recém coletados são imitados.

2.2 *Combinatorial Multi-armed Bandits*

2.2.1 Definição

O *Combinatorial Multi-armed Bandit* (CMAB) é uma variação do problema *Multi-armed Bandit* (MAB), definido da seguinte forma:

- Um conjunto de n variáveis $X = \{X_1, X_2, \cdots, X_n\}$, de modo que cada variável X_j

pode-se assumir K_j valores, sendo eles $\mathcal{X}_j = \{v_j^1, v_j^2, \cdots, v_j^{k_j}\}$. Assim, temos $\mathcal{X} = \{(v_1, \cdots, v_n) \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_n\}$ como o conjunto de possibilidades de combinações de valores assumidos por X, onde cada uma destas combinações $V \in \mathcal{X}$ é chamada de macro-arm.

- $\mu : \mathcal{X} \to \mathbb{R}$ é uma função de utilidade, que recebe uma macro-arm e retorna um valor de utilidade deste macro-arm.

Em um problema CMAB, tem-se como objetivo procurar o macro-arm que maximize o valor de utilidade esperado.

2.2.2 *Two-Phase Naïve Sampling* (NS)

A *Naïve Sampling* (NS) foi projetada de modo a lidar com problemas nos quais não é possível avaliar sistematicamente todos macro-arms de forma exaustiva através de uma busca (ONTAÑÓN, 2017a). NS faz parte de uma família de estratégias de amostragem que assume que as variáveis de macro-arm são independentes mesmo que na pratica não sejam, por isso, o nome "*naïve*", o qual significa "ingênuo" em inglês. Portanto, podemos aproximar μ de um macro-arm A como a soma das utilidades dos valores de cada variável $a_i \in A$, denotado $\mu(a_i)$, isto é, $\mu(A) \approx \sum_{a_i \in A} \mu'(a_i)$, uma vez que cada uma depende apenas do valor de uma das variáveis do CMAB. Assim, podemos dividir o problema de n variáveis em $n + 1$ problemas MAB:

- n MABs locais, um para cada variável $X_j \in X$, sendo que X_j é uma variável que pode assumir K_j valores pertencentes a $\mathcal{X}_j$.
- Um MAB global denominado MAB_g, que considera cada macro-arm avaliado durante a busca como um braço do MAB_g. Inicialmente o MAB_g se encontra vazio.

A cada iteração, o NS usa uma política π_0 para determinar se adiciona um macro-arm ao MAB_g por meio dos MABs (explorar) locais ou se avalia um macro-arm existente no MAB_g (explorar MAB global).

1. Caso a exploração seja selecionada: um macro-arm é escolhido usando a política π_l para escolher o valor para cada $X_i \in X$ de forma independente, sendo o mesmo adicionado no MAB global.
2. Caso a exploitação seja selecionada: um macro-arm é selecionado usando uma estratégia π_g sobre os macro-arms já presentes no MAB global.

Por conseguinte, as políticas π_0, π_l e π_g são utilizadas como ϵ-guloso e empregamos o NS em sua versão bifásica, que consiste em, após k iterações, substituir o conjunto de

políticas (ϵ_0^1, ϵ_l^1, and ϵ_g^1) que apresenta uma natureza mais exploradora por outro conjunto de políticas (ϵ_0^2, ϵ_l^2, and ϵ_g^2) que possui uma natureza mais exploitadora. Ademais, usamos um fator de decaimento γ, onde $0 < \gamma < 1$, para multiplicar todos os μ de modo que os macro-braços avaliados recentemente tenham um maior impacto na busca do que os avaliados a mais tempo.

2.3 *Simulated Annealing*

O *Simulated Annealing* (SA) é um algoritmo de busca local que foi proposto inicialmente por (Kirkpatrick; Gelatt; Vecchi, 1983), que consiste em procurar o mínimo/máximo de uma função de utilidade Ψ. Para isso, ele combina o algoritmo *Hill Climbing* (HC, em português “subida da encosta") com passeios aleatórios de modo a conseguir escapar de mínimos/máximos locais. O SA usa a temperatura para controlar o quão gulosa é a busca, ao invés de sempre escolher o estado vizinho que minimiza/maximiza Ψ, como é feito no algoritmo HC. Com esse fim, o SA emprega uma função de aceitação que usa a temperatura atual para decidir se a amostra vizinha será aceita ou não, de modo que, quando a temperatura está alta, a busca aceita soluções piores que a atual tomando as decisões não gulosa. À medida que a temperatura abaixa, o número de decisões não gulosa diminui de modo a tornar-se similar ao HC.

Segue abaixo o pseudocódigo:

```
1 def simulated_annealing():
2     s = amostragem_aleatoria()
3     for i in range(MAX_ITE):
4         atual_temperatura = atualiza_temperatura(i)
5         s' = escolhe_vizinho(s)
6         if funcao_de_aceite(s, s', atual_temperatura, Ψ):
7             s = s'
8     return s
```

Na linha 2, o SA começa gerando um estado aleatório s e então faz um loop até atingir uma determinada condição de parada, a qual pode ser, por exemplo, o número de iterações ou o tempo de execução (linha 3). O SA atualiza a atual temperatura de acordo com uma determinada função (linha 4) e seleciona um novo estado s' da vizinhança de s. Já na linha 6, o SA aceita ou rejeita de acordo com uma função de aceitação. Caso aceite, o valor de s' é atribuído ao s (linha 7) e o processo é repetido. Se rejeitar, SA repete o procedimento gerando outro vizinho.

A forma como o escalonamento da temperatura é implementada é geralmente específica ao problema e é amplamente estudada, em (HAJEK, 1988) e (NOURANI;

ANDRESEN, 1998). No tocante à função de aceitação, tem-se que a expressão abaixo desenvolvida por (KIRKPATRICK; GELATT; VECCHI, 1983) ainda é muito utilizada na literatura:

$$\min\left(1, \exp\left(\frac{\beta \cdot (\Psi(s') - \Psi(s))}{T_i}\right)\right).$$

Sendo T_i a temperatura na iteração i, dada pela equação $T_i = \frac{T_1}{(1+\alpha \cdot i)}$, onde T_1 é a temperatura inicial e Ψ uma função de utilidade. Nota-se que se $\Psi(p') \geq \Psi(p)$, então s' terá a probabilidade de ser aceito pelo SA igual a 1.0. Caso contrário, a probabilidade de aceitação dependerá do T_i e β, onde β é um parâmetro de entrada que ajusta a ganância, sendo que valores maiores de β tornaram o algoritmo mais guloso e. Além disso, os valores T_i irão diminuir com o passar das iterações fazendo com que o algoritmo se torne mais guloso.

2.4 *Monte-Carlo Tree Search*

O *Monte-Carlo Tree Search* (MCTS, em português "Árvore de Busca Monte-Carlo"), desenvolvido por (COULOM, 2006), combina avaliações de Monte-Carlo - onde realiza simulações de acordo com uma política pre-determinada e calcula o resultado médio retornado pelo encontrado - e busca em árvore. Depois de um número suficiente de simulações, o MCTS armazenará informações sobre o valor de utilidade de uma ação aplicada a um estado. Um valor de utilidade baixo significa que a aplicação dessa ação naquele estado provavelmente produzirá uma pontuação baixa por uma função de avaliação Ψ.

O algoritmo é separado em quatro rotinas MCTS:

1. Seleção: A etapa de seleção começa na raiz da árvore e escolhe o filho que maximiza uma fórmula de seletividade. COULOM (2006) indica que a fórmula de seletividade ideal deve alocar mais simulações e, portanto, ser pesquisada mais profundamente, para nós promissores. Se o filho selecionado for um nó folha, ele passa para a rotina de expansão. Caso contrário, ele chama a rotina de seleção novamente até que um nó folha seja alcançado.

2. Expansão: Dado o nó folha recuperado no estágio de seleção, ele expande um filho aleatório (para reduzir o viés) adicionando esse nó à árvore.

3. Simulação: Com o nó recém-expandido da fase de expansão, uma simulação é aplicada a partir desse nó até o final do jogo. A simulação segue uma política aplicada a este nó até que um nó terminal seja alcançado. Então, uma função Ψ é usada para avaliar este estado, retornando o valor de utilidade Ψ daquele estado.

4. Retropropagação: A etapa de retropropagação atualiza os valores X_i de todos os nós visitados na etapa de seleção com o valor Ψ da etapa de simulação.

A Figura 1 mostra uma visão geral do *pipeline* iterativo do algoritmo MCTS. Este é um processo iterativo, assim, quanto mais tempo for dado ao algoritmo, mais precisa será a estimativa em relação a qual ação escolher em um estado. Passado um determinado limite de tempo imposto pelo usuário ou qualquer outro critério de parada, o algoritmo então retornará a ação referente a qual o nó filho teve o maior número de visitas durante o processo iterativo.

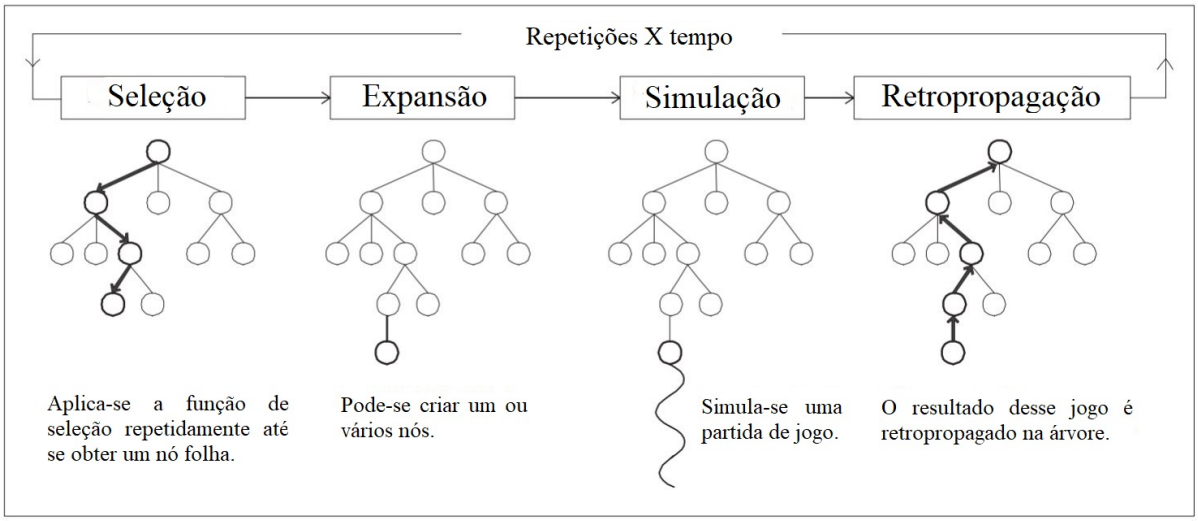


Figura 1 – Visão geral Monte-Carlo Tree Search Algorithm

Source: (CHASLOT et al., 2008)

KOCSIS; SZEPESVÁRI (2006) desenvolveu um algoritmo que desenvolve uma árvore de pesquisa enquanto explora o espaço baseado em MCTS chamado *Upper Confidence Bound Applied to Trees* (UCT). A utilidade de um estado s baseado nesta heurística é chamada de UCB1 (*Upper Confidence Bounds*), a qual foi proposta por (AUER; CESA-BIANCHI; FISCHER, 2002) e é calculada da seguinte forma:

$$\text{UCB1}(s) = \bar{X}_i + C\sqrt{\frac{\ln\left(N(s)\right)}{n_i(s)}}$$

Aqui, $\bar{X}_i$ é o valor Ψ médio do i-ésimo filho de s; $N(s)$ é o número de vezes que o nó foi visitado nas etapas de seleção anteriores; $n_i(s)$ é o número de vezes que s foi visitado e o i-ésimo filho foi selecionado; e C é uma constante de exploração. O primeiro termo da equação é um termo de exploitação, pois favorece o filho com maior valor médio de avaliação; o segundo termo é um termo de exploração. A fórmula UCB1 é empregada no estágio de seleção como a fórmula de seletividade no algoritmo UCT.

2.5 *Self-Play*

Seja uma estratégia σ_i para um jogador i e uma estratégia σ_{-i} para um jogador $-i$, dizemos que é σ_i a melhor resposta (em inglês "*best response*", BR) para σ_{-i} caso não exista nenhuma estratégia σ'_i que consiga uma recompensa superior a ela jogando contra σ_{-i}. Com isso, podemos definir o conceito de estratégia dominante, no qual σ_i é uma estratégia dominante caso ela seja a melhor resposta para qualquer estratégia que o adversário use (PETROSYAN; ZENKEVICH, 2016).

Nessa dissertação, consideramos dois algoritmos de aprendizado *self-play*, sendo eles: a *Iterated-Best Response* (IBR) (LANCTOT et al., 2017) e o *Fictitious Play* (FP) (BROWN, 1951).

2.5.1 *Poachers and Rangers*

Antes de introduzimos o IBR e FP, iremos apresentar um jogo chamado *Poachers and Rangers*, que nos ajudará a entender melhor os algoritmos.

Poachers and Rangers é um jogo simples jogado por dois jogadores, sendo que um joga como *rangers* sendo o seu objetivo proteger os portões de um parque nacional evitando que os *poachers* entrem no parque e o outro jogador como *poachers* tendo como objetivo entrar no parque por um portão desprotegido. As partidas têm um número arbitrário de portões e os jogadores jogam de forma alternada até os *rangers* encontrarem a estratégia dominante na qual todos os portões estarão protegidos. Veja o exemplo 1 para entender melhor o funcionamento de uma partida. de uma partida com 3 portões enumerados de 1 a 3:

Example 1 *Consideraremos uma partida com 3 portões enumerados de 1 a 3. A partida começa com os poachers atacando o portão 1, chamaremos esta jogada de [ataque[1]]. Nesse momento, os poachers estão vencendo, pois estão atacando um portão desprotegido. Agora é a vez dos rangers jogarem e eles defendem o portão 1, chamaremos esta jogada de [defender[1]], como nenhum dos poachers estão atacando um portão desprotegido, os rangers estão ganhando a partida. Novamente é a vez dos poachers jogarem e eles fazem o seguinte movimento [ataque[2],[ataque[3]] e, em resposta a isso, os rangers jogam [defender[1],defender[2],defender[3]], defendendo todos os portões e terminando jogo.*

2.5.2 *Iterated-Best Response*

O método IBR é um algoritmo simples em que as estratégias dos jogadores são atualizadas de forma sequencial e iterativa. Ele funciona do seguinte modo: o IBR inicia com uma estratégia arbitrária σ_i e tenta calcular a melhor resposta σ_{-i} para ela. Caso consiga, o algoritmo passa então a tentar achar uma melhor resposta para σ_{-i} e este

processo alternado se repetirá até encontrar o perfil de equilíbrio Nash que consiste em atingir um ponto em que tanto σ_i quanto σ_{-i} são as melhores respostas uma para a outra. O exemplo 3 demonstra o funcionamento do IBR.

Example 2 *Consideramos novamente uma partida de Poachers and Rangers com 3 portões. o IBR começa procurando uma jogada para poachers que é uma melhor resposta para os rangers, assim escolhendo [ataque[3]]. Em seguida, o método irá procurar uma melhor resposta para a jogada dos poachers, selecionando [defender[2],defender[3]]. Esse processo se repete com os poachers escolhendo [ataque[2],[ataque[1]], a qual é uma melhor resposta para a estratégia atual dos rangers. Por fim, os rangers escolhem [defender[1],defender[2],defender[3]].*

Entretanto, IBR pode não progredir em direção a um perfil de equilíbrio de Nash uma vez que existe a possibilidade que, em sua execução, ocorra um ciclo de melhores respostas, como, por exemplo, cenários em que A é a melhor resposta de B, B é a melhor resposta de C e C é a melhor resposta de A.

Example 3 *Consideramos novamente uma partida de Poachers and Rangers com 3 portões. Os poachers começam escolhendo [ataque[2]]. Em resposta a isso, os rangers jogarão [defender[2]], fazendo com que os poachers escolham [ataque[1]], assim levando os rangers a jogarem [defender[1]]. Com isso, os poachers escolhem novamente [ataque[2]] e todo essa sequência de escolhas se repete criando um ciclo que nunca irá convergir para um perfil de equilíbrio Nash.*

2.5.3 *Fictitious Play*

Introduzido por (BROWN, 1951), o FP foi inicialmente apresentado como um algoritmo para calcular o valor de um jogo de soma zero, tendo sido estudado em profundidade por (ROBINSON, 1951), o que lhe deu o nome do processo de aprendizagem de Brown-Robinson. Ele funciona do seguinte modo:

Da mesma forma que o IBR, o FP começa com uma estratégia arbitrária $sigma_i$ para a qual uma melhor resposta é calculada. No entanto, em contraste com o IBR, o FP mantém dois conjuntos, $Sigma_i$ e $Sigma_{-i}$, com todas as melhores respostas calculadas para cada jogador. Em cada iteração do PF calcula-se a melhor resposta para uma estratégia mista cujo suporte é formado por todas as estratégias em $Sigma_i$ (ou $Sigma_{-i}$). A distribuição das estratégias em $Sigma_i$ e $Sigma_{-i}$ converge para um perfil de equilíbrio de Nash de estratégias mistas para o jogo, ilustramos este processo através do O exemplo 4. Note que, o FP não apresenta os ciclos de melhores respostas como o IBR, pois ele responde a todas as estratégias vistas até o momento no processo, porém isso faz com que ele fique mais caro computacionalmente.

Example 4 *Utilizaremos uma partida de Poachers and Rangers com 3 portões e os conjuntos de estratégias $Sigma_p$ e $Sigma_r$ dos poachers e rangers respectivamente. Como o conjunto $Sigma_r$ ainda está vazio, os poachers escolhem [ataque[2]] como seu primeiro movimento e FP adiciona essa estratégia em $Sigma_p$. Os rangers jogam [defender[2]] como resposta a $Sigma_p$ e [defender[2]] é adicionado em $Sigma_r$. Em seguida, é a vez dos poachers jogarem e eles escolhem [ataque[1]], o qual também é adicionado em $Sigma_p$. Depois disso, começa o turno dos poachers, note que se eles escolherem a estratégia [defender[1]], conseguem responder ao último movimento dos poachers. Porém no FP é necessário responder a todas as estratégias do $Sigma_p$, isto é, é preciso uma estratégia que responda a [ataque[1]] e [ataque[2]] simultaneamente. Logo os rangers escolhem a estratégia [defender[1],defender[2],defender[3]] e o jogo termina.*

2.6 Definição do Problema

Seja G um jogo de sequencial de soma zero para dois jogadores definido por um conjunto de estados S, um par de jogadores $\{i, -i\}$, um estado inicial $s_{init} \in S$, uma função $A_i(s)$ que recebe um estado s e retorna um conjunto de ações que podem ser realizadas pelo jogador i no estado s, e uma função $U_i(s)$ que retorna a utilidade do jogador i; como G é um jogo de soma zero, temos que $U_i(s) = -U_{-i}(s)$. Uma estratégia programática é um programa de computador que codifica uma estratégia σ, tendo que uma estratégia para jogador i é uma função $\sigma_i : S \to A_i$ mapeando um estado s para uma ação a. Denotamos $U(s, \sigma_i, \sigma_{-i})$ como o valor do jogo para s dado que i segue a estratégia σ_i e $-i$ a σ_{-i}.

Para definir nosso espaço de programas, usamos uma linguagem específica de domínio (DSL) (DEURSEN; KLINT; VISSER, 2000) D, sendo $[\![D]\!]$ o conjunto infinito de programas que podem ser escritos com D.

Nessa dissertação, abordaremos o problema de se sintetizar uma melhor resposta σ_i para uma dada estratégia σ_{-i} em $[\![D]\!]$, isso é, maximizar a função de utilidade do jogador i contra o jogador $-i$, ainda em outros termos, resolver $\max_{\sigma_i \in [\![D]\!]} U(s_{init}, \sigma_i, \sigma_{-i})$.

2.6.1 *Domain-Specific Languages*

Uma DSL é definida como uma gramática livre de contexto (V, Σ, R, I), onde V, Σ, e R são conjuntos de símbolos não terminais, terminais e relações que definem as regras de produção da gramática, respectivamente, e I é o símbolo inicial da gramática. A Figura 2 mostra uma DSL onde $V = \{I, C, B\}$, $\Sigma = \{c_1, c_2, b_1, b_2$ if, then$\}$, sendo que R são as relações (por exemplo, $C \to c_1$).

A DSL da Figura 2 permite programas com um único comando (c_1 ou c_2) e programas com ramificação. Representamos programas como árvores de sintaxe abstrata (em

$$I \rightarrow C \mid \text{if}(B) \text{ then } C$$
$$C \rightarrow c_1 \mid c_2$$
$$B \rightarrow b_1 \mid b_2$$

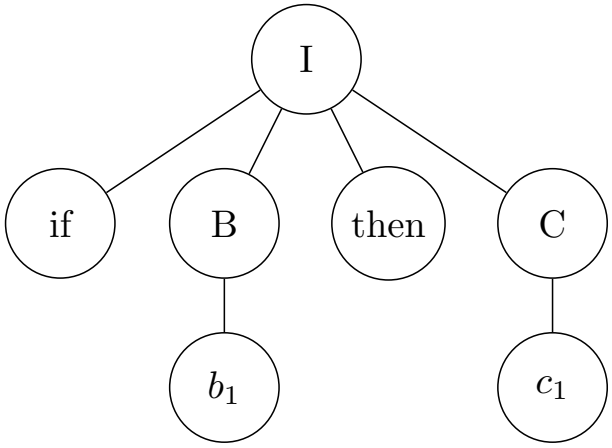


Figura 2 – DSL (esquerda) e AST para "if b_1 then c_1"(direita).

inglês "*abstract syntax trees*", AST), onde a raiz da árvore é I, os nós internos são não terminais e os nós folha são terminais. A Figura 2 mostra um exemplo de AST, onde as folhas são símbolos terminais e os nós internos são não terminais.

3 Trabalhos Relacionados

Nossas propostas Sketch-SA e HSS estão relacionadas ao aprendizado por reforço programático interpretável (PIRL) (VERMA et al., 2018), no qual se tenta sintetizar um programa que codifica uma política para resolver processos de decisão de Markov (BASTANI; PU; SOLAR-LEZAMA, 2018; VERMA et al., 2019), em particular aqueles que usam alguma forma de clonagem comportamental, como a aprendizagem por imitação (SCHAAL, 1999). BASTANI; PU; SOLAR-LEZAMA (2018) apresenta o algoritmo VIPER que usa uma variante do algoritmo de aprendizado por imitação DAgger (ROSS; GORDON; BAGNELL, 2011) para refinar uma política neural de alto desempenho em uma árvore de decisão interpretável. O VIPER usa uma rede neural profunda como um oráculo para gerar dados rotulados para consultas do tipo DAgger, a fim de treinar a árvore de decisão mencionada anteriormente.

VERMA et al.(2018) usa DAgger e uma política neural para ajudar na síntese de políticas programáticas. As ações que a política neural escolhe em um conjunto de estados são usadas em um procedimento de otimização Bayesiana para encontrar valores de constantes adequadas para as políticas programáticas. VERMA et al. (2019) empregam uma abordagem semelhante, mas o modelo neural é treinado para que não seja “muito diferente"das políticas sintetizadas, com o objetivo de facilitar a tarefa de otimização.

Nosso trabalho difere do PIRL porque focamos em jogos, o outro trabalho foca em problemas de agente único. Além disso, não assumimos que o oráculo está disponível para consultas como nos métodos do tipo DAgger, como também não assumimos que o oráculo é uma rede neural tal qual (VERMA et al., 2019)) e (BASTANI; PU; SOLAR-LEZAMA, 2018; VERMA et al., 2019). Por exemplo, em nossos experimentos, usamos um conjunto de dados de um oráculo humano. Também não exigimos que a estratégia clonada tenha um alto desempenho; somos capazes de aprender *sketches* eficazes mesmo com estratégias fracas e HSS não utiliza um oráculo. Ademais, os trabalhos do PIRL dependem da aprendizagem por imitação para sintetizar políticas programáticas, a qual consiste em treinar uma política neural que é então utilizada como um oráculo para guiar a busca.

MARIÑO et al. (2021) introduziram o Lasi, um método que emprega a clonagem comportamental para simplificar a linguagem utilizada para a síntese. O Lasi retira da linguagem as instruções que não são necessárias para clonar uma estratégia. Ele pode ser usado com nossos métodos de aprendizagem, simplificando a linguagem para só então realizar a busca. Além disso, não é tão geral quanto nossos métodos, porque não pode ser aplicado a domínios nos quais todos os símbolos da linguagem são necessários, como o Can’t Stop.

Há na literatura outros que sintetizaram programas para servir como funções de avaliação (BENBASSAT; SIPPER, 2011), mas não para servir como estratégias completas. Outros exploraram a síntese de estratégias para jogos cooperativos (CANAAN et al., 2018) e problemas de agente único (BUTLER; TORLAK; POPOVIĆ, 2017; FREITAS; SOUZA; BERNARDINO, 2018). Esses métodos podem se beneficiar potencialmente de nossos métodos de aprendizado de *sketch* através de características comportamentais.

QIU; ZHU (2022) argumentaram que abordagens baseadas em aprendizagem por imitação podem levar a políticas fracas devido a uma lacuna de representação, ou seja, políticas programáticas podem não ser capazes de imitar políticas neurais, pois um programa que imita o oráculo pode não existir no espaço programático. QIU; ZHU introduziram um sistema para aprender políticas programáticas sem oráculos usando uma aproximação diferenciável da linguagem usada para codificar as políticas. O sistema de QIU; ZHU é limitado a linguagens com estruturas *if-then-else*, tendo em vista que sua aproximação não suporta loops. Em contraste com o sistema de QIU; ZHU, o HSS e Sketch-SA suporta estruturas e loops *if-then-else*. Além de que, em contraste com os algoritmos PIRL anteriores, o HSS não usa um oráculo para guiar a pesquisa.

No mais, algoritmos *self-play* foram empregados para aprender estratégias com aprendizado por reforço (HEINRICH; LANCTOT; SILVER, 2015; LANCTOT et al., 2017). Nestes trabalhos, a estratégia é codificada em uma rede neural e aprendida com gradiente ascendente. Deve-se considerar o cenário onde as estratégias são codificadas em programas de computador, de modo que o espaço procurado seja não diferenciável e métodos baseados em gradiente não possam ser usados.

Algoritmos de busca hierárquica foram aplicados no contexto de busca em jogos (ONTAÑÓN; BURO, 2015), de busca de caminhos baseados em mapas (BOTEA; MüLLER; SCHAEFFER, 2004) e de busca heurística (AGUAS; JIMÉNEZ; JONSSON, 2018). Até onde sabemos, o HSS é o primeiro algoritmo de busca hierárquica para sintetizar estratégias ou políticas.

4 Formulação do Problema

4.1 *Simulated Annealing* para Síntese de Estratégias Programáticas

Com o intuito de sintetizar estratégias, usamos SA para aproximar uma melhor resposta programática a uma estratégia alvo σ_{-i}, ou seja, SA aproxima uma solução para $\arg\max_{\sigma_i \in [\![D]\!]} U(s_{init}, \sigma_i, \sigma_{-i})$. Para isso, o SA inicia com um programa gerado aleatoriamente da seguinte maneira: começamos com o símbolo inicial I e o substituímos por uma regra de produção escolhida de forma randômica; então, repetidamente, substituímos um símbolo não terminal no programa gerado por uma regra de produção aleatória e válida; enfim, paramos quando o programa contém apenas terminais. Uma vez definido o programa inicial p, SA gera um vizinho p' de p modificando uma subárvore na AST de p. Escolhemos aleatoriamente um símbolo não terminal n na AST (todos os símbolos não terminais podem ser escolhidos com igual probabilidade) e substituímos a subárvore com raiz em n com uma subárvore que é gerada com o mesmo procedimento usado para gerar o programa inicial. Por fim, caso o SA aceite p', ele será atribuído ao p; caso contrário, se rejeitado, o SA repete o procedimento, gerando outro vizinho de p.

4.2 UCT para Síntese de Estratégias Programáticas

A UCT desenvolve uma árvore de pesquisa enquanto explora o espaço. Cada nó na árvore representa um programa, que pode ser completo ou incompleto. Dizemos que um programa está completo se todas as folhas em seu AST são terminais. A raiz da árvore UCT representa o programa incompleto do símbolo inicial I da DSL. Os filhos de um nó n na árvore UCT são os programas que podem ser gerados pela aplicação de uma regra de produção ao símbolo não terminal mais à esquerda do programa que n representa.

As quatro etapas da UCT são repetidas várias vezes e ele retorna o programa com maior valor de Ψ entre todos os programas avaliados durante a pesquisa, assim atinge um limite de tempo especificado. A UCT armazena os valores Ψ de programas que foram avaliados em iterações anteriores do algoritmo. A árvore UCT pode crescer para incluir programas completos, ou seja, nós sem filhos. Se a etapa de seleção terminar em um programa completo, ela não realizará nenhuma expansão e retornará o valor Ψ armazenado de n na etapa de retropropagação.

Fizemos a escolha de usar uma única execução de SA como política de simulação da UCT. Ao executar SA como política de simulação para um programa incompleto p,

os vizinhos de um programa só podem ser obtidos alterando as subárvores do AST que estão enraizadas em um nó folha não terminal.

4.3 Aprendendo *Sketches* com Clonagem Comportamental

Foi considerado o cenário em que o sintetizador recebe como entrada um conjunto de dados de pares estado-ação $L = \{(s_j, a_j)\}_{j=1}^{m}$ com ações escolhidas pelo oráculo σ_o para os estados de uma ou mais partidas do jogo. Usamos esse conjunto de dados para aprender um *sketch* para acelerar a síntese de uma estratégia programática.

SA e UCT podem ser empregados para clonar o comportamento de σ_o substituindo Ψ por uma função de avaliação $C(L, p)$ que recebe o conjunto de dados L e um programa p, e retorna uma pontuação de quanto p clona σ_o. Observe, no entanto, que a clonagem do comportamento de σ_o pode resultar em estratégias fracas. Isso ocorre porque σ_o pode não ser representado em $[\![D]\!]$, ou o conjunto de dados L é limitado e é preciso realizar consultas do tipo DAgger (ROSS; GORDON; BAGNELL, 2011) para aumentá-lo. Porém, σ_o pode não estar disponível para tais consultas, por exemplo, no caso de σ_o ser um jogador humano indisponível ou de σ_o ser uma estratégia fraca e clonar exatamente seu comportamento resultaria em uma estratégia fraca. Ao invés de aprendermos uma estratégia diretamente com a clonagem comportamental, nós a utilizamos para aprender um *sketch* que auxilia o processo de síntese de uma estratégia forte.

Tem-se que os métodos de aprendizado de *sketch* podem ser mais eficazes do que aqueles que otimizam Ψ diretamente por dois motivos. Primeiro, Ψ pode ser computacionalmente mais caro do que C. Usar C para aprender partes da estratégia programática tenderá a ser mais eficiente do que aprender toda a estratégia com Ψ. Segundo, a função C pode oferecer um sinal mais denso para pesquisa (por exemplo, o vizinho p' de p pode não derrotar σ_{-i}, mas pode ter um valor de C melhor, o que pode ser útil para orientar a pesquisa).

4.3.1 Aprendendo *Sketches* com UCT

Executamos a UCT com a função de avaliação $C(L, p)$ por várias iterações e, sempre que encontramos um programa completo p com um valor C maior que a melhor solução atual, avaliamo-lo com Ψ. Chamamos essa etapa de busca de *sketch*. Uma vez atingido um limite de tempo, usamos o programa encontrado na busca de *sketch* com o maior valor Ψ para inicializar uma segunda busca UCT, que chamamos de busca de melhor resposta (em inglês "*best response*", BR). Seja p o programa com o maior valor Ψ encontrado na busca de *sketch*, o programa é definido por uma sequência de regras de produção que substituem o símbolo não terminal mais à esquerda na sequência de programas parciais, começando com o símbolo inicial da DSL. Iniciamos a árvore UCT da

busca BR com um ramo que representa as regras de produção de p. Em seguida, realizamos uma etapa de retropropagação no ramo adicionado com $\Psi(p)$, que foi calculado na busca de *sketch*. Ao adicionar o ramo que leva p à árvore da busca BR, estamos direcionando-o para explorar programas que compartilham a estrutura de p. Isso ocorre porque os nós ao longo do ramo adicionado provavelmente terão valores $\bar{X}$ mais altos do que outros ramos, especialmente nas primeiras iterações da pesquisa. O ramo adicionado à árvore UCT da busca BR funciona como um *sketch* conforme definido na literatura (SOLAR-LEZAMA, 2009), pois representa um programa com "buracos"que são preenchidos pela busca BR. O aprendizado de *sketch* fornece um conjunto de *sketches* com níveis variados de detalhes (nós mais profundos na ramificação representam *sketches* com mais informações) que a pesquisa BR explora enquanto otimiza para Ψ.

4.3.2 Aprendendo *Sketches* com *Simulated Annealing*

Como com UCT, executamos SA para clonar σ_o usando $C(L, p)$ como função de avaliação. Durante a busca, toda vez que encontramos uma solução com melhor valor C, também a avaliamos com Ψ. Uma vez que atingimos um limite de tempo, SA retorna o programa p com o maior valor Ψ. Também chamamos essa etapa de busca de *sketch*. Em seguida, usamos p como o programa inicial de outra busca SA que otimiza diretamente para Ψ, o que também denominamos de busca BR. Consideramos que o programa p permite que a busca BR comece em uma parte mais promissora do espaço do programa, uma vez que p pode ter uma estrutura semelhante à estrutura de um programa que se aproxima de uma melhor resposta para σ_{-i}. Enquanto o ramo adicionado à árvore UCT da busca BR pode ser visto como um conjunto de *sketches* que são explorados de acordo com a priorização definida pela UCT, a conexão entre o uso do programa p para inicializar a busca SA-BR e os *sketches* não é tão clara. Vemos o programa p como um rascunho suave, porque fornece uma estrutura inicial para o sintetizador, mas não especifica explicitamente um conjunto de lacunas, uma vez que SA pode alterar qualquer subárvore do AST de p. Algumas subárvores são mais propensas a serem substituídas do que outras devido à função de aceitação do SA, ou seja, SA prefere alterar subárvores que resultarão em um aumento no valor de Ψ.

4.4 Função de Pontuação com Clonagem Comportamental

Usamos funções dependentes de domínio $C(L, p)$ e as descrevemos na seção empírica. Consideramos funções de pontuação que usam tanto o estado quanto as ações no conjunto de dados $L = \{(s_j, a_j)\}_{j=1}^{m}$ e funções que usam apenas os estados em L, como em abordagens recentes sobre aprendizado por imitação a partir de observações (TORABI; WARNELL; STONE, 2018b).

Algorithm 1 HSS

Require: Jogo G, DSL D, Utilidade U, estratégia σ_{-i}.
Ensure: Aproximar melhor resposta σ_i para σ_{-i}.
1: Inicializar biblioteca vazia L de programas.
2: Initialize dictionary T mapeando uma característica para um valor.
3: $p_b \leftarrow$ **None**
4: **while not** limite de tempo atingido **do**
5: $V \leftarrow$`Busca-Características`(T)
6: $p \leftarrow argmin_{p \in L} \sum_i^{|\mathcal{F}|} |F(p)[i] - V[i]|$
7: $p, P \leftarrow$`Busca Local`$(G, D, p, V, U, \sigma_{-i})$
8: Atualiza $T[F(p')]$ com $U(p')$ para todos $p' \in P$
9: $L \leftarrow L \cup P$
10: **if** $U(p) > U(p_b)$ **then**
11: $p_b \leftarrow p$
12: **return** p_b

4.5 *Hierarchical Synthesis Search* (HSS)

HSS realiza sua busca pesquisando em dois espaços: o espaço dos programas e o espaço das características comportamentais do jogo. Seja $\mathcal{F}$ um conjunto de características para um conjunto de estados do jogo, cada característica $f \in \mathcal{F}$ define uma função mapeando um conjunto de estados para um inteiro. Por exemplo, uma característica pode contar o número máximo de peças de um determinado tipo que um jogador controla durante uma partida de um jogo de tabuleiro. Cada iteração da busca do HSS no espaço de características define um vetor V de valores de características, que é utilizado para guiar a busca no espaço de programa. A busca no espaço do programa tenta principalmente encontrar um programa p que maximize $U(s_{init}, p, \sigma_{-i})$. No caso de empates entre os programas p e p' em termos de valores de U, o HSS irá escolher o mais similar ao V. Uma vez que a busca no espaço do programa atinge um limite de tempo e retorna um programa p que se aproxima da melhor resposta σ_{-i}, usamos o valor de $U(s_{init}, p, \sigma_{-i})$ para atualizar as estatísticas sobre os valores das características de V, de forma que a busca no espaço de características possa aprender a combinação de valores de características que são mais prováveis de maximizar $U(s_{init}, p, \sigma_i)$.

4.5.1 Pseudocódigo de HSS

O algoritmo 1 mostra o pseudocódigo do HSS. O HSS é invocado para que se aproxime uma melhor resposta para uma estratégia de destino σ_{-i} em cada iteração de um algoritmo de aprendizado como IBR ou FP. O HSS recebe o jogo G, uma DSL D, uma função de utilidade U e uma estratégia σ_{-i}. O HSS retorna uma melhor resposta aproximada para σ_{-i} codificada em um programa p. Para isso, o HSS usa duas estruturas de dados: um conjunto L de programas encontrados na busca e um dicionário T que coleta estatísticas sobre as características avaliadas na busca.

O conjunto L é uma biblioteca de programas que são usados para inicializar a busca local no espaço de programas. Dado que a busca no espaço de características retorna um vetor V (linha 5), o HSS busca o programa $p \in L$ cujo vetor de características $F(p)$ é mais similar a V em termos de diferença absoluta dos valores de características (linha 6). O programa p é usado para inicializar a busca local no espaço do programa (linha 7). Ao inicializar a pesquisa com p, permitimos que a pesquisa comece em um local do espaço do programa onde é mais provável encontrar um programa que corresponda aos valores de recursos em V.

A busca local retorna um programa p que se aproxima de uma melhor resposta para σ_{-i}; ele também retorna o conjunto de todos os programas P encontrados na pesquisa (linha 7). Atualizamos as estatísticas de que a pesquisa no espaço de características usa para cada programa em P (linha 8) e adicionamos todos os programas de P à biblioteca de programas (linha 9). Quando o HSS atinge um limite de tempo, ele retorna o programa com o melhor valor U encontrado na pesquisa (linhas 10–12). .

4.5.2 Usando Biblioteca de Programas para Transferir Aprendizagem

Se a biblioteca L de programas que o HSS acumula for reutilizada em outras chamadas do algoritmo, pode-se transferir parte do conhecimento gerado na pesquisa entre as chamadas do HSS. Inicializamos a biblioteca L com um conjunto vazio apenas na primeira chamada do HSS. Chamadas posteriores reutilizarão a biblioteca L acumulada para iniciar a busca IBR ou FP.

A biblioteca de programas permite que a busca local comece em um local do espaço de programa que a busca no espaço de características considera mais promissora. Se a pesquisa no espaço de características descobre que a característica f_j leva a valores altos de U, por exemplo, então a pesquisa no espaço do programa pode começar em uma região onde f_j é mais provável de ocorrer. A biblioteca de programas permite que o HSS "pule" efetivamente pelo espaço do programa de acordo com a orientação de pesquisa do espaço de características.

5 Resultados e discussões

Nós testamos nossa hipótese de que a utilização de características comportamentais pode ajudar a acelerar o processo de síntese de encontrar uma melhor resposta para um dado um alvo σ_{-i} ou um conjunto de alvos Σ_{-i} que denotaremos apenas como σ_{-i} para simplificar.

- Em nosso primeiro cenário, tentaremos sintetizar uma estratégia programática contra um alvo σ_{-i} determinado. Para isso, utilizaremos as características comportamentais providas de um oráculo para ajudar a guiar a busca.
- No segundo cenário, tentaremos utilizar características comportamentais para auxiliar algoritmos *self-play* durante a busca de uma melhor resposta em seu processo iterativo. Empregaremos os algoritmos *self-play* para encontrar um perfil que aproxima um equilíbrio Nash no espaço de estratégias programáticas, o qual basicamente é um par de programas que melhor respondem um ao outro, ou seja, as estratégias σ_i e σ_{-i} que resolvem $\max_{\sigma_i \in [\![D]\!]} \min_{\sigma_{-i} \in [\![D]\!]} U(s_{init}, \sigma_i, \sigma_{-i})$. Em nosso trabalho, consideramos DSLs que geram programas de estratégias puras. Assim, assumimos a existência de um equilíbrio Nash de estratégia pura.

5.1 Domínios

5.1.1 *Can't Stop*

O jogo de tabuleiro *Can't Stop* contém 11 colunas, enumeradas de 2 a 12. A coluna 2 tem 3 linhas e o número de linhas é incrementado em 2 a cada coluna até a coluna 7, que tem 13 linhas. O número de linhas decresce em 2, começando na coluna 8 até a coluna 12, que também possui 3 linhas. O objetivo do jogo é conquistar 3 colunas primeiro. A Figura 3 apresenta um exemplo de um tabuleiro *Can't Stop*.

O jogo funciona do seguinte modo: a cada rodada do jogo, o jogador tem 3 fichas neutras e rola 4 dados de seis faces. O jogador pode colocar uma ficha neutra em qualquer coluna dada pela formação de um par de dados combinados de forma arbitrária. Assim, esta ficha será colocada imediatamente acima do seu marcador permanente daquela coluna. Caso não haja nenhum, será colocada na primeira linha. Em seguida, o jogador deve escolher se irá para sua jogada fazendo com que as fichas neutras se tornem fichas permanentes, de modo a preservar suas posições no tabuleiro. Caso o jogador queira continuar sua rodada, ele pode jogar os dados novamente e usar as fichas neutras restantes ou avançar nas colunas onde já possui fichas neutras posicionadas. Porém, caso o jogador

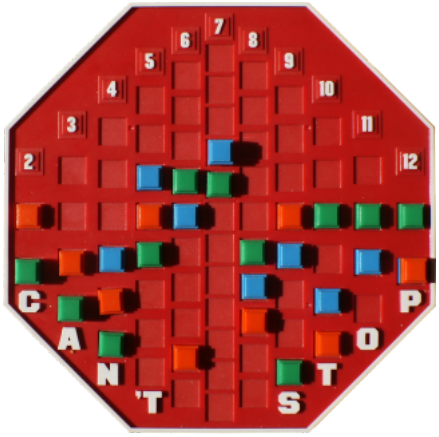


Figura 3 – Tabureiro do *Can't Stop*.

Source: <https://en.wikipedia.org/w/index.php?curid=4276555> (Acessado em 11/2022)

não tiver fichas neutros e as combinações de dados resultarem apenas em números de coluna para os quais o jogador não tem ficha neutra, o jogador perde as fichas neutras posicionadas e passa sua vez para o outro jogador. Este processo se repete até que haja um vencedor.

Desenvolvemos uma DSL para sintetizar programas para as decisões sim-não e de colunas. A DSL inclui operadores como funções `map`, `sum`, `argmax` e `lambda`. A DSL também inclui um conjunto de funções específicas de domínio, como uma função para contar o número de linhas que um jogador avançou em um turno. Descrevemos a DSL em detalhes no apêndice. A tarefa de síntese é gerar um programa que resolva simultaneamente o sim-não e as decisões da coluna.

5.1.2 MicroRTS

O MicroRTS é uma implementação minimalista de um jogo de estratégia em tempo real (em inglês "*real-time strategy*") amplamente utilizado para avaliar sistemas inteligentes (ONTANÓN, 2013). Sendo ele um jogo de soma zero jogado por dois jogadores, no qual cada jogador controla um conjunto de unidades e, a cada 100 milissegundos, o jogador tem que atribuir qual ação cada uma destas unidades irá realizar (todas as ações são determinísticas e não há informações ocultas no MicroRTS); o seu objetivo é destruir todas as unidades do adversário. Nele existem vários mapas e em cada mapa pode haver uma estratégia dominante diferente. Além disso, o MicroRTS possui os seguintes tipos de unidades:

- *Worker*: são responsáveis por construir *Barracks* e *Bases*, além de coletarem recursos e poderem ser usados em combate, mas são menos eficientes do que as unidades de combate.

- *Ranged*: unidade de combate capaz de atacar de uma certa distância.
- *Heavy*: unidade de combate lenta, porém resistente.
- *Light*: unidade de combate rápida.
- *Barracks*: é responsável por treinar unidades de combate.
- *Base*: é responsável por guardar recursos e treinar Worker.

A Figura 4 mostra um exemplo de um estado de jogo do MicroRTS: as unidades de círculo cinza representam as unidades *Worker*, as unidades de círculo ciano são as unidades *Ranged* e as unidades de círculo amarelo representam as unidades *Heavy*. Os quadrados verde-claros representam os recursos a serem coletados, os quadrados brancos são as *Bases* e os quadrados cinzas são os *Barracks*.

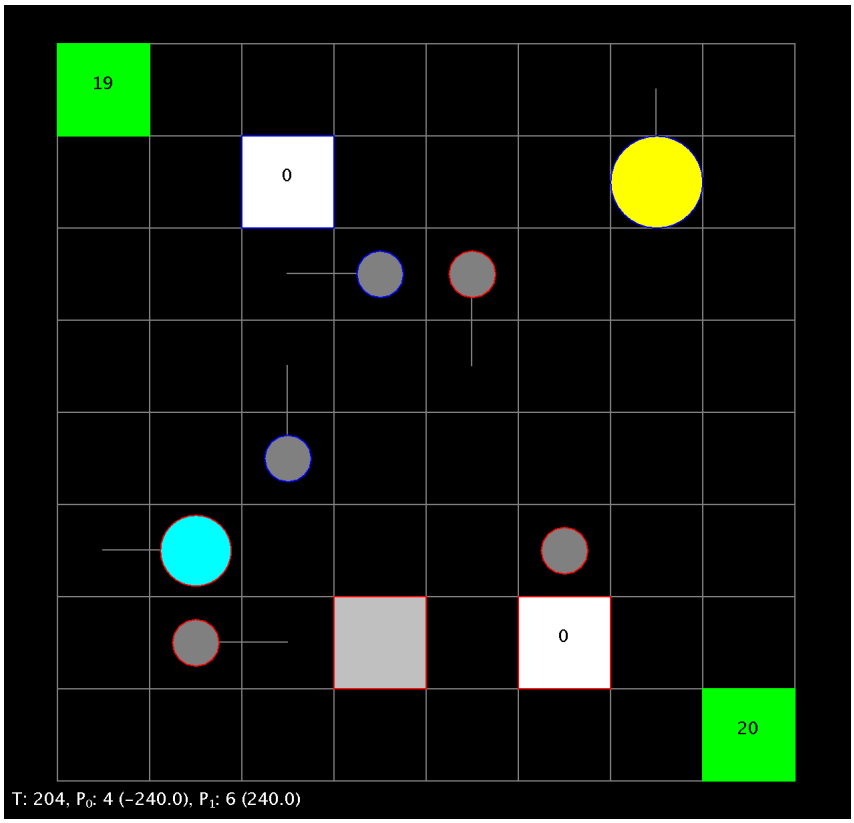
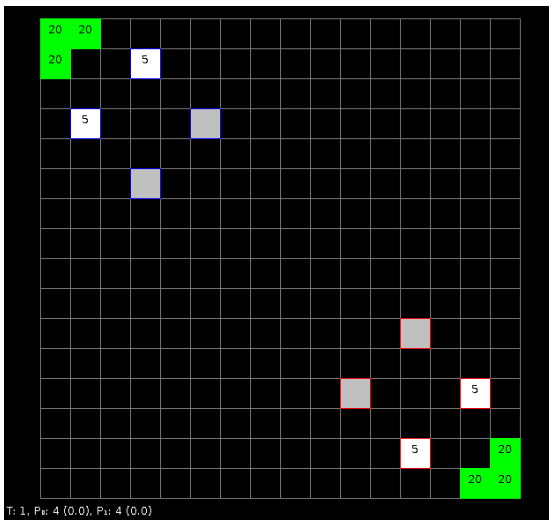


Figura 4 – Exemplo de partida do *MicroRTS*.

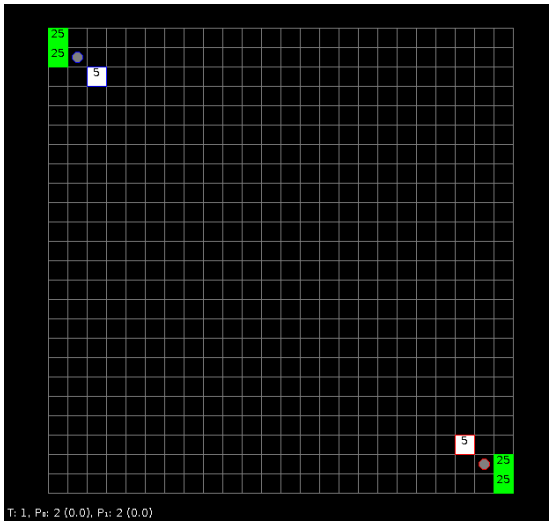
Source: <https://sites.google.com/site/micrortsaicompetition/microrts> (Acessado em 11/2022)

De modo geral, mapas maiores geralmente resultam em partidas mais longas e outras particularidades do mapa, como a distância das unidades do jogador dos recursos, que podem afetar significativamente as estratégias dos jogadores. Usamos quatro mapas de tamanhos diferentes, onde os nomes entre parênteses são os nomes dos mapas na base de código do MicroRTS[1]: 16×16 (TwoBasesBarracks), 24×24 (BasesWorkers), 32×32 (Ba-

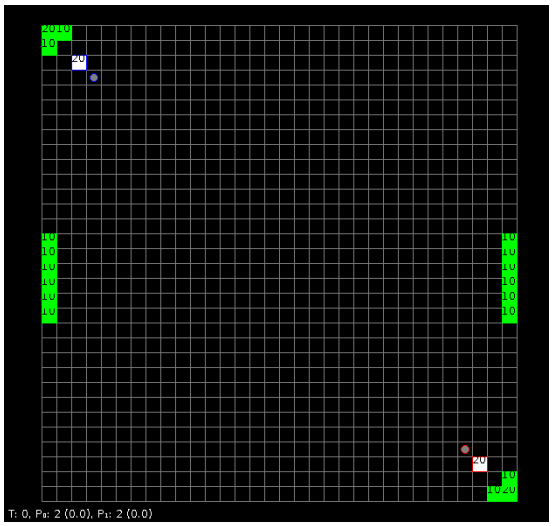
[1] <https://github.com/santiontanon/microrts>



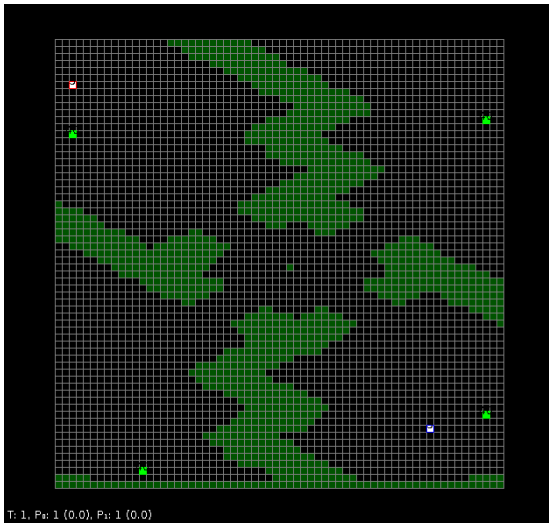
16×16 (TwoBasesBarracks)



24×24 (BasesWorkers)



32×32 (BasesWorkers)



64×64 (BloodBath-B)

Figura 5 – Mapas usados nos experimentos.

sesWorkers) e 64×64 (BloodBath-B). O mapa menor e os maiores são das Competições do MicroRTS. A Figura 5 mostra os mapas MicroRTS empregados em nossos experimentos.

Uma DSL semelhante à apresentada por (MARIÑO et al., 2021) foi implementada, de modo que a DSL inclui loops, condicionais e um conjunto de funções específicas de domínio que atribuem ações às unidades (por exemplo, construir uma unidade *Barracks*) e um conjunto de funções Booleanas. Descrevemos a DSL no apêndice.

5.2 Síntese de *Sketches* com Clonagem de Comportamento

5.2.1 Configurações Usadas

5.2.2 Configurações *Can't Stop*

GLENN; ALOI (2009) usou um algoritmo genético para melhorar uma estratégia programática existente para *Can't Stop* (KELLER, 1986). Empregamos a estratégia de GLENN; ALOI como σ_{-i} e o chamamos de GA. O GA decide parar de jogar em um turno do jogo sempre que a soma das pontuações dos marcadores neutros exceder um limite. Ele define um programa para calcular tal pontuação (decisão sim-não) e outro programa para decidir em qual coluna avançar a seguir (decisão da coluna). Nossa DSL foi projetada de modo a ser capaz de sintetizar esses programas.

A função Ψ para *Can't Stop* é o número médio de vitórias de p contra σ_{-i} em 1.000 partidas. Executamos a busca de Sketch-SA por 1 hora e a busca de Sketch-UCT por 10 horas, pois a UCT é mais lenta do que o SA na exploração do espaço. Posteriormente rodamos o busca BR para cada um dos algoritmos até completar dois dias.

5.2.3 Configurações MicroRTS

Usamos o vencedor da Competição de 2020 MicroRTS, COAC - estratégia programática escrita por programadores humanos, como σ_{-i} (ONTAÑÓN, 2020).

A função Ψ para MicroRTS pode assumir os seguintes valores: 1,0 caso ele vença σ_{-i}, 0,5 em caso de empate e 0,0 caso perca. Cada mapa tem dois locais de início, logo teremos testes independentes que sintetizarão para um lado e outros testes para o outro. O MicroRTS não requer uma definição explícita para dividir o tempo entre a busca de *sketch* e a busca BR. Isso ocorre porque tanto Ψ quanto $C(L, p)$ são calculados fazendo p jogar uma partida contra σ_{-i}. A transição entre a busca de *sketch* e a busca BR ocorre naturalmente se definirmos a função de avaliação dos algoritmos de busca como a função com empates sendo quebrados de acordo com $C(L, p)$. No início da síntese, o valor Ψ será zero para todos os programas avaliados na busca.

5.2.4 Funções de Pontuação Baseadas em Ações

Nós consideramos uma função de pontuação que se baseia nas ações escolhidas por p em todos os estados (s_j, a_j) em relação às escolhidas pelo oráculo, dada pela seguinte fórmula $\sum_{(s_j,a_j)\in L} 1[a_j = p(s_j)]/|L|$, onde 1 é a função indicadora. Denotamos a apredizagem de *sketch* através do SA e UCT utilizando essa função de pontuação como Sketch-SA(A) e Sketch UCT(A) respectivamente.

5.2.5 Funções de Pontuação Baseadas em Observações

5.2.5.1 *Can't stop*

Usamos uma função de pontuação baseada em observação que mede a porcentagem de marcadores permanentes no estado final do jogo de uma partida que se sobrepõe aos marcadores permanentes obtidos por um programa p no estado final do jogo se p o tivesse jogado. Essa pontuação é calculada iterando cada estado s_j de uma partida em L e aplicando os efeitos das ações $p(s_j)$ a um tabuleiro inicialmente vazio do jogo. Uma vez que um estado de fim de jogo s_f é alcançado, calculamos a porcentagem de marcadores permanentes sobrepostos entre s_f e o estado de fim de jogo em L. Por exemplo, se o jogador no estado de final de uma partida em L conquistou as colunas 2, 3 e 7 e teve um marcador na coluna 12; e o programa p conquistou as colunas 2, 3 e teve um marcador na coluna 8; então a pontuação é $(3+5)/(3+5+13+1+1) = 0,34$. Aqui, $(3+5)$ é o número de posições na interseção dos estados de final de jogo e $(3+5+13+1+1)$ é a união das posições. Se p e σ_o retornam às mesmas ações para todos os estados em L, nesse caso a pontuação é $1,0$. Se L for composto por varias partidas, retornamos à pontuação média de todas as partidas. Sendo assim, denotamos SA e UCT usando esta função de pontuação como Sketch-SA(O) e Sketch-UCT(O).

5.2.5.2 MicroRTS

Usamos uma função de pontuação baseada em observação que calcula uma diferença absoluta normalizada entre (i) o número de unidades e recursos que a estratégia p treina e coleta em uma partida de p contra σ_i e (ii) o número de unidades e recursos que a estratégia σ_o treina e coleta em uma correspondência no conjunto de dados L. Sejam n_u e n'_u o número de unidades do tipo u que p e σ_o treinaram em suas partidas, respectivamente, a pontuação referente às unidades do tipo u é dada por $1 - \frac{|n_u - n'_u|}{\max(n_u, n'_u)}$. Por exemplo, se o número de unidades do tipo *Ranged* que a estratégia p treinou for 4 e se o número de unidades do tipo *Ranged* que a estratégia σ_o treinou for 10, a pontuação para unidades *Ranged* será $1 - 6/10 = 0,4$. Assim, o valor retornado é a pontuação média de todos os tipos de unidades e recursos. Já a pontuação é 1,0 se p e σ_o treinam o mesmo número de unidades de cada tipo e coletam o mesmo número de recursos. Sendo assim, denotamos SA e UCT usando esta função como Sketch-SA(O) e Sketch-UCT(O).

5.2.6 Estratégias Clonadas

Usamos estratégias fracas e fortes σ_o para gerar os conjuntos de dados L. L é composto de pares estado-ação de partidas nas quais σ_o joga o jogo com ele mesmo ou com outra estratégia, sobre o qual especificaremos abaixo.

5.2.6.1 *Can't Stop*

Consideramos 3 conjuntos de dados L, cada um gerado com um σ_o diferente. O primeiro σ_o escolhe aleatoriamente uma das ações disponíveis em cada estado do jogo. O conjunto de dados é composto por 3 partidas σ_o contra σ_o. Selecionamos o conjunto de dados do vencedor das partidas. Essa estratégia é capaz de vencer aproximadamente 2,8% das partidas jogadas contra GA, chamamos essa estratégia de "Aleatória". Usamos um conjunto de dados composto por 3 partidas, jogando a estratégia GA contra ela mesma, novamente pegamos o conjunto vencedor e um conjunto de dados composto por 3 partidas que um humano jogou com GA; o humano venceu todas as partidas.

5.2.6.2 MicroRTS

Também consideramos 3 conjuntos de dados L para MicroRTS. O primeiro L é composto por 2 partidas (uma em cada local inicial do mapa) de *Ranged Rush* (RR), que é uma estratégia programática simples (STANESCU et al., 2016) contra o COAC. Também usamos um conjunto de dados composto por 2 partidas de A3N (MORAES et al., 2018) contra o COAC. O A3N considera ações de baixo nível de unidades (por exemplo, mover um quadrado para a direita) enquanto planejam suas ações. Escolhemos o A3N porque achamos que seria difícil para o sintetizador clonar seu comportamento, pois as estratégias derivadas com A3N dificilmente estariam no espaço das estratégias definidas pela DSL (a DSL não permite um controle fino das unidades, como faz o A3N). Tanto RR quanto A3N não conseguiram vencer nenhuma partida contra o COAC, nosso σ_{-1}, em todos os mapas avaliados. Também consideramos um conjunto de dados composto por estados de 2 partidas COAC contra ele mesmo.

5.2.7 Resultados Empíricos: *Can't Stop*

A Figura 6 mostra os resultados do *Can't Stop*. Cada gráfico mostra a taxa de vitórias contra σ_{-1} (eixo y) da melhor estratégia sintetizada ao longo do tempo (eixo x) dos seguintes métodos: algoritmos de busca (seja SA ou UCT) sem clonagem de características comportamentais para aprendizado de *sketch* (*baseline*) e algoritmos de busca que aprendem *sketch*: Sketch-SA(A), Sketch-UCT(A), Sketch-SA(O) ou Sketch-UCT(O). Nos gráficos, contabilizamos o tempo utilizado nas buscas de *sketch*. Executamos 30 execuções independentes de cada método. As linhas representam os resultados médios, enquanto as áreas sombreadas representam o desvio padrão das execuções.

Os métodos que aprendem *sketches* são muito mais rápidos do que suas *baselines*. Na maioria dos casos, Sketch-SA e Sketch-UCT atingem taxas de vitória que suas contrapartes de *baseline* não atingiram dentro do limite de tempo. Os resultados também mostram que os métodos SA têm melhor desempenho do que seus equivalentes UCT.

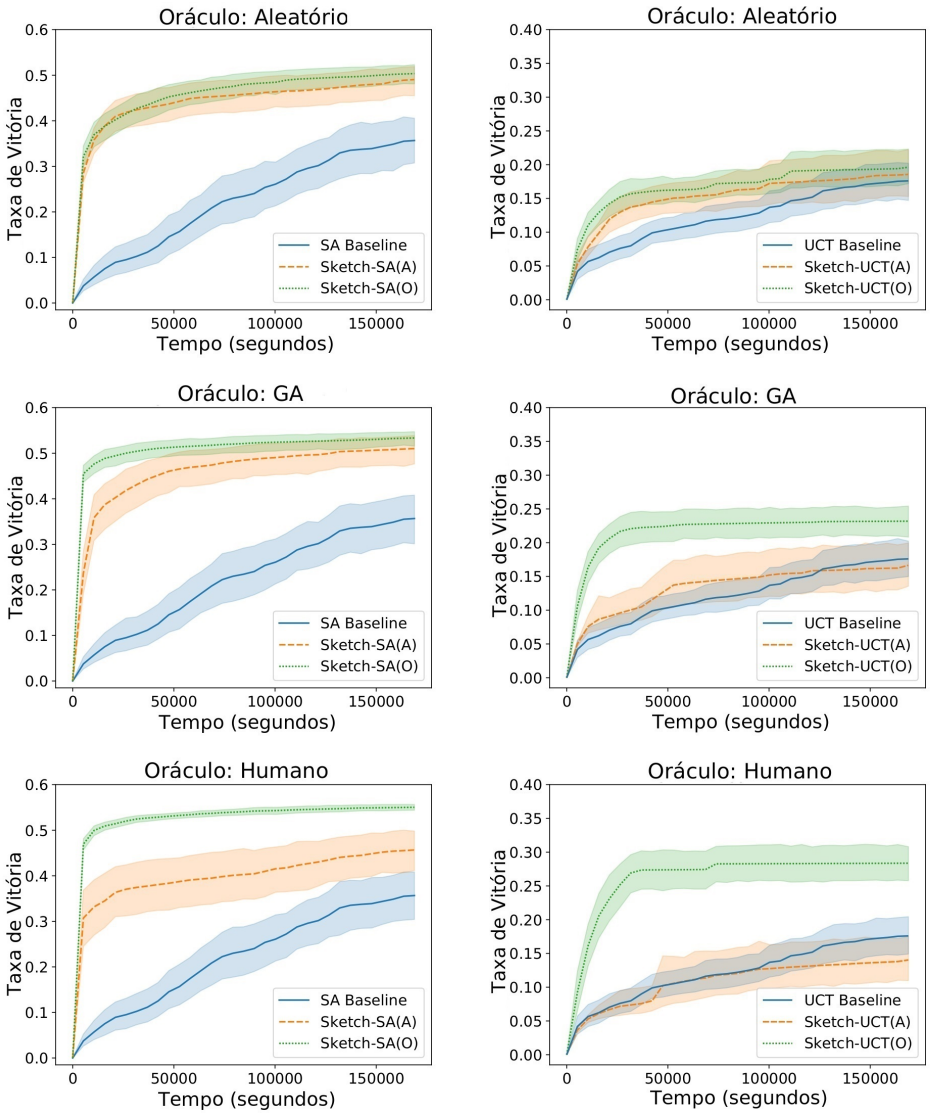


Figura 6 – Taxa de vitórias das variantes SA (esquerda) e UCT (direita).

Apenas o Sketch-SA sintetiza estratégias que derrotam σ_i em mais de 50% das partidas. Em particular, o Sketch-SA(O) é o método mais eficaz para aproximar uma melhor resposta para σ_i. Conjecturamos que o SA sintetiza estratégias mais fortes do que a UCT, visto que explora o espaço mais rapidamente do que a UCT. A complexidade de tempo da etapa de seleção da UCT é quadrática em relação à duração do programa. Isso ocorre porque, a cada etapa de seleção, a busca percorre todas as regras de produção do programa incompleto atual para cada regra de produção aplicada a ele. Por outro lado, o SA pode sintetizar um grande número de instruções com uma única operação de vizinhança.

Embora o Sketch-SA(O) tenha um desempenho melhor clonando o comportamento do jogador humano, o método funciona surpreendentemente bem quando aprende *sketches* clonando o comportamento de uma estratégia aleatória. Um dos aspectos-chave para jogar *Can't Stop* é decidir quando parar de jogar para que os marcadores neutros se tornem

marcadores permanentes. O programa deve ter uma estrutura específica para calcular a pontuação que leva a uma ação de parada, semelhante a $sum(map(\lambda.f, neutrals))$. Aqui, neutros é uma lista com os marcadores neutros e f é uma função de pontuação para marcadores individuais.

Os operadores *sum* e *map* retornam a soma das pontuações de todos os marcadores. Apesar de a estrutura desse programa não ser trivial, a estratégia aleatória tem 50% de chance de escolher a ação de parada e seus efeitos são refletidos nos estados em L (ou seja, marcadores neutros tornam-se marcadores permanentes). Os sintetizadores descobrem o *sketch* como o programa acima, porque esses programas colocam marcadores permanentes no quadro, como faz a estratégia aleatória.

A busca BR modifica o *sketch* para maximizar a utilidade do jogador, mas a maior parte da estrutura do programa é mantida. Os métodos baseados em *sketch* têm pior desempenho com a função baseada em ação. Isso ocorre porque a função baseada em observação captura os efeitos até mesmo de ações raras. Um bom jogador de *Can't Stop* escolhe continuar jogando na maioria dos estados, mas em estados cruciais ele escolhe parar. Um jogador que nunca para tem uma pontuação alta baseada em ação porque a ação de parar é rara. Como resultado, quando utilizamos funções de pontuação baseadas em ações, os métodos baseados em *sketches* geralmente falham em aprender a estrutura do programa necessária para decidir corretamente quando parar.

5.2.8 Resultados Empíricos: MicroRTS

A Figura 7 mostra os resultados das variantes SA no MicroRTS. Como no *Can't Stop*, os métodos baseados em *sketch* são superiores às *baselines*, com Sketch-SA(O) alcançando taxas de vitória próximas a $1,0$, mesmo ao aprender o *sketch* de A3N, que é uma estratégia incapaz de derrotar σ_{-i} . Observa-se também uma distância entre as funções baseadas em ação e observação, a qual parece aumentar com o tamanho do mapa. O Sketch-SA(A) não sintetizou estratégias que derrotaram σ_{-i} para o L gerado com A3N e COAC no mapa 64×64. A explicação para o baixo desempenho de Sketch-SA(A) é semelhante à de *Can't Stop*: algumas ações são raras, mas desempenham um papel importante no jogo (por exemplo, pode-se treinar unidades *Ranged* após construir uma *Barracks*, o que pode acontecer apenas uma vez em uma partida). Os gráficos dos mapas de tamanho 16×16 e 24×24 na Figura 7 mostram um tempo de execução reduzido (aproximadamente 6 horas para o primeiro e 24 horas para o segundo) para que possamos visualizar melhor as curvas. Cada linha representa a taxa média de vitórias e as áreas sombreadas o desvio padrão de 20 execuções independentes de cada método, sendo 10 para cada como jogador 1 e 10 como jogador 0.

Por sua vez, a Figura 8 mostra os resultados das variantes UCT no MicroRTS, os quais são semelhantes aos do *Can't Stop*. As variantes UCT têm desempenho pior

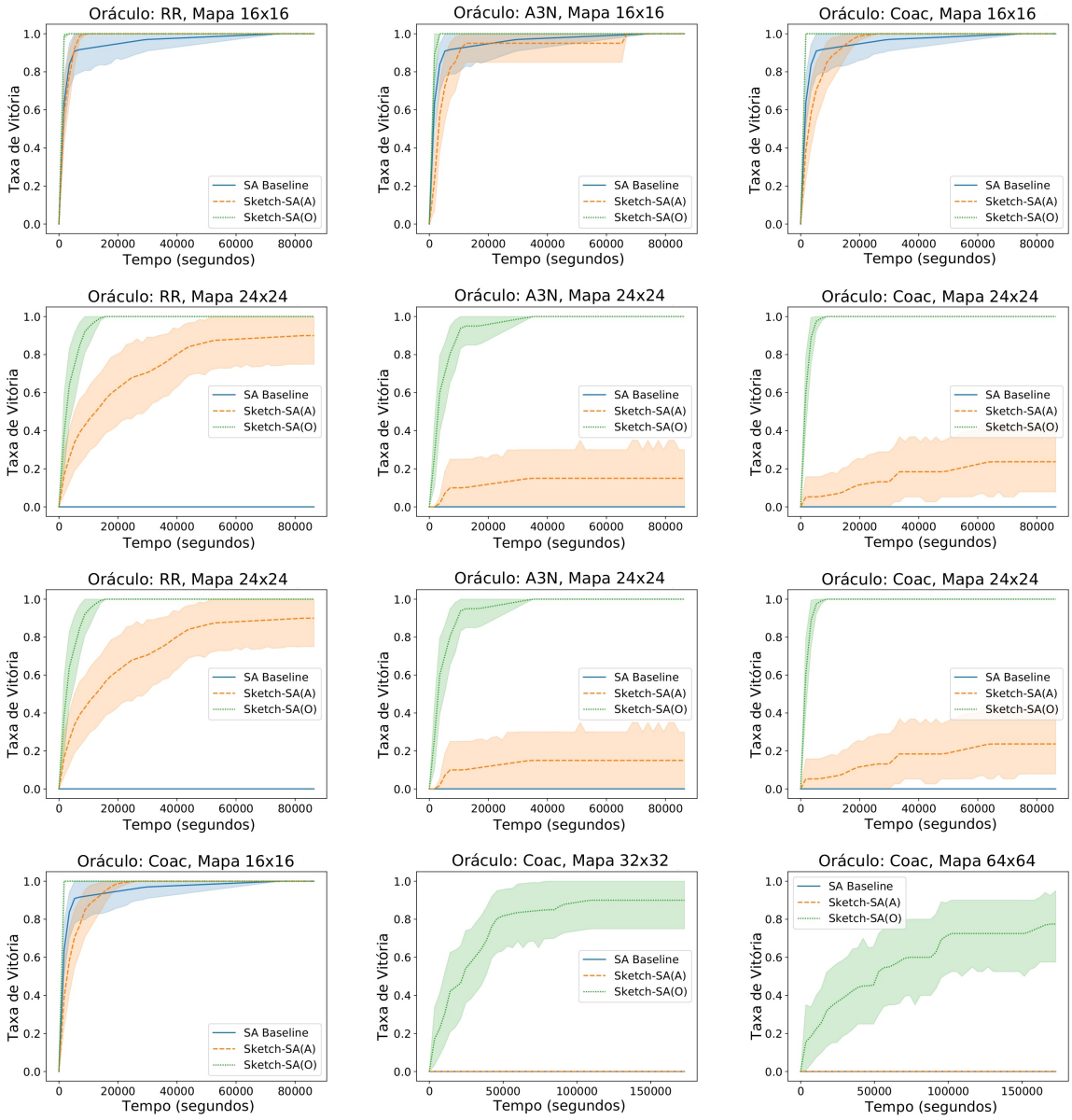


Figura 7 – Taxa de vitória das estratégias sintetizadas por variantes SA.

do que suas contrapartes SA. Em particular, nenhum método UCT, incluindo a UCT de *baseline*, é capaz de sintetizar uma estratégia que derrota σ_{-i} nos mapas maiores de 32×32 e 64×64. Além disso, as abordagens que aprendem *sketch* funcionam melhor do que suas contrapartes nos mapas menores de 16×16 e 24×24.

5.2.9 Correlação entre Taxa de Vitória e Pontuação de Clonagem

A Figura 9 mostra a relação entre a taxa de vitórias (eixo y esquerdo) e a taxa de clonagem de observação (eixo y direito) para a melhor estratégia programática encontrada durante as buscas Sketch-SA no jogo *Can't Stop*. A estratégia clonada é a Aleatória. O pico no valor da taxa de clonagem no início da pesquisa representa o final da pesquisa de *sketch*, que otimiza a taxa de clonagem e a busca BR começa onde o valor da pontuação

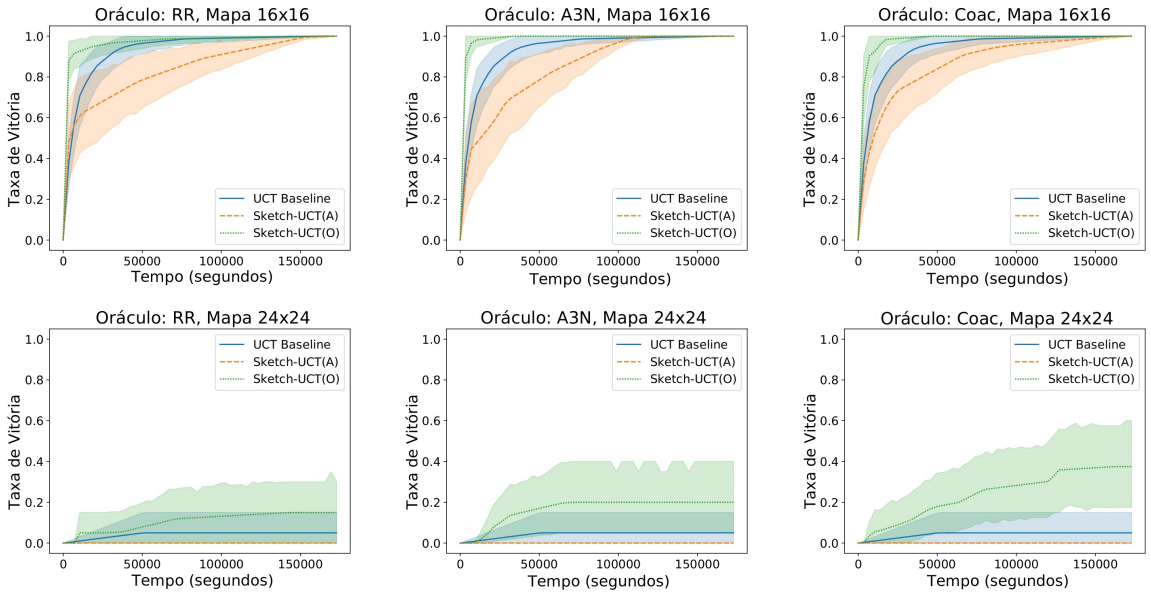


Figura 8 – Taxa de vitória das estratégias sintetizadas por variantes UCT.

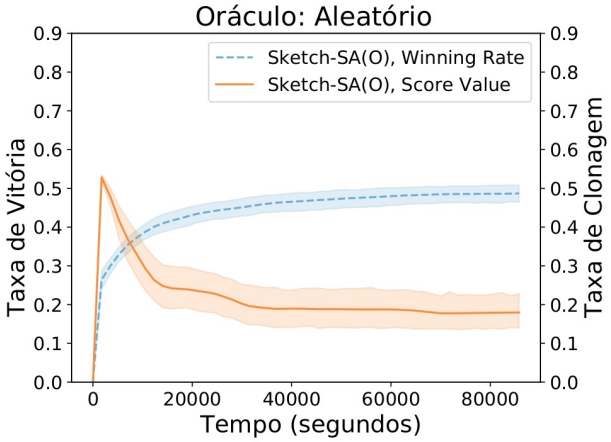


Figura 9 – Taxa de vitória e taxa de clonagem do melhor programa encontrado durante a síntese.

cai.

Ao realizar a busca de *sketch*, onde a função de avaliação é C, pode-se observar que enquanto o valor C cresce, o valor Ψ também cresce. É provável que o algoritmo comece com um programa que nem mesmo seja compilável e, portanto, com uma taxa de vitória e valor de pontuação 0. Assim, sempre que a busca de *sketch* consegue encontrar um vizinho que seja um programa compilável e passar a imitar melhor o oráculo, este programa poderá agora pontuar possivelmente uma taxa de vitórias superior a 0. Por isso, podemos observar um aumento tanto no valor da pontuação quanto na taxa de vitórias contra σ_{-1}.

No entanto, na busca BR, onde a função de avaliação é Ψ, o valor Ψ aumenta enquanto o valor C diminui. A queda em C é esperada, pois a estratégia se tornaria

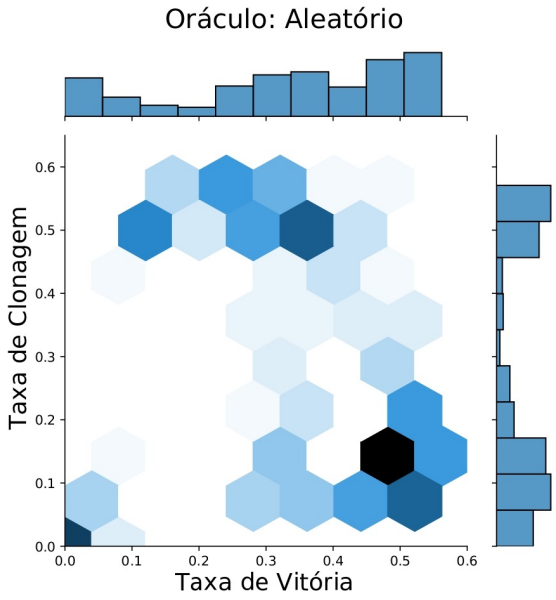


Figura 10 – Distribuição de programas de acordo com sua taxa de vitória e sua taxa de clonagem.

cada vez mais semelhante à estratégia aleatória, que é uma estratégia fraca, se o valor C continuasse a crescer. A Figura 10 apresenta uma história semelhante, mostrando a distribuição de um conjunto de estratégias programáticas de acordo com sua taxa de clonagem (eixo y) e taxa de vitórias contra a estratégia de GA (eixo x). As estratégias com taxas de vitória mais altas não são semelhantes à estratégia Aleatória (veja as cores mais escuras no canto inferior direito). Apesar da falta de correlação entre a taxa de vitórias e a taxa de clonagem, a clonagem do comportamento do Aleatório ainda pode ajudar na busca, permitindo que ele encontre *sketches* úteis do programa.

5.2.10 Resultados com Apenas *Sketches*

Tipo de Clonagem	L	WR - Sketch only	WR - Sketch & BR
A	Human	0.352(199)	0.491(117)
	GA	0.424(070)	0.516(092)
	Random	0.182(058)	0.507(085)
O	Human	0.212(216)	0.555(017)
	GA	0.368(088)	0.543(024)
	Random	0.244(114)	0.518(058)

Tabela 1 – Comparação da taxa vencedora contra GA entre a abordagem que faz apenas a Busca de *Sketch* e Busca de *Sketch* junto com Busca-BR (*Sketch* & BR) usando diferentes conjuntos de dados L para clonagem.

A Tabela 1 mostra os resultados do uso apenas da pesquisa de *sketch* com a função de avaliação de clonagem comportamental $C(L, p)$ no domínio *Can't Stop* usando

SA. Esses resultados são apresentados em tabela porque mostraremos apenas o último programa retornado pela pesquisa contra a estratégia de GA.

Quando comparada aos resultados alcançados pela busca de *sketch* junto com a busca BR, a taxa de vitória das abordagens que não utilizam a busca BR é inferior em ambos os tipos de pesquisa de *sketch* (baseada em ação (A) e em observação (O)). Isso provavelmente acontece porque o sinal que o algoritmo recebe da busca de *sketch* é apenas um sinal que representa se o algoritmo está clonando a ação escolhida pelo oráculo. Portanto, o algoritmo de busca irá otimizar a busca para este sinal e não para a taxa vencedora contra a estratégia de GA, ou seja, o sinal dado por Ψ. A busca BR com a função de avaliação Ψ otimiza para programas que produzem um sinal Ψ melhor e, ao final da execução, o programa sintetizado se aproximará melhor de uma melhor resposta à estratégia GA.

Um resultado interessante é que, com essa abordagem, melhores programas foram sintetizados ao clonar o comportamento do conjunto de dados gerado pela estratégia de GA. Acreditamos que, como a DSL que usamos permite que a estratégia de GA seja sintetizada perfeitamente, ele pode sintetizar uma estratégia que clona com precisão sua estratégia se houver tempo suficiente, em oposição à estratégia do humano, que é improvável de ser representável em nossa DSL.

5.2.11 Amostra de Estratégia Programática

Nesta seção discutimos algumas estratégias programáticas sintetizadas pelo Sketch-SA(O) que foram capazes de derrotar a estratégia σ_{-1} em cada domínio.

5.2.11.1 Programas Sintetizados para MicroRTS

Abaixo é mostrada uma estratégia Sketch-SA(O) sintetizada para o mapa 24×24. Editamos levemente a estratégia para facilitar a leitura. Esta estratégia atinge a taxa de vitórias de 1,0 contra o COAC.

```
1 def Sketch-SA-O-24x24(state s):
2     for u in s:
3         if not u.isWorker():
4             u.moveToUnit(Ally, LessHealthy)
5         u.train(Ranged)
6         for u in s:
7             u.attackIfInRange()
8         u.build(Barracks)
9     for u in s:
10        u.harvest(4)
11        u.attack(LessHealthy)
```

A estratégia recebe um estado s e atribui uma ação a cada unidade. Se uma unidade não receber uma ação, ela não executará uma ação na próxima rodada do jogo. Uma vez que a estratégia atribui uma ação a uma unidade u, a ação não pode ser substituída por outra ação. Por exemplo, a estratégia não altera a ação atribuída às unidades na linha 4, mesmo que tentemos atribuir a elas uma ação diferente posteriormente no programa. Esta estratégia treina unidades de Ranged (linha 5) uma vez que a *Barracks* é construída (linha 8); uma única *Barracks* é construída porque todos os recursos são gastos treinando unidades *Ranged* assim que a *Barracks* estiver disponível. As unidades *Ranged* se agrupam (linha 4) e atacam as unidades inimigas dentro de seu alcance de ataque (linha 7). Se não houver unidades inimigas dentro de seu alcance, elas atacam as unidades inimigas que estão perto de serem removidas do jogo (linha 11). A estratégia designa 4 *Workers* para coletar recursos (linha 10). A seguir, apresentamos os programas Sketch-SA(O) sintetizados para os mapas utilizados em nossos experimentos. Os programas apresentados abaixo não são simplificados para melhorar a legibilidade.

```
1 def Sketch-SA-O-16x16(state s):
2     for u in s:
3         u.train(Ranged, Right, 4)
4         u.harvest(9)
5         u.attack(Closest)
6         u.train(Worker, Up, 1)
7
8 def Sketch-SA-O-24x24(state s):
9     for u in s:
10        if not u.isBuilder():
11            u.moveToUnit(Ally, LessHealthy)
12        u.train(Ranged,Left,15)
13        for u in s:
14            u.idle()
15        u.build(Barracks, EnemyDir, 100)
16    for u in s:
17        u.harvest(4)
18        u.attack(LessHealthy)
19
20
21 def Sketch-SA-O-32x32(state s):
22    for u in s:
23        for u in s:
24            for u in s:
```

```
25              u.harvest(2)
26            for u in s:
27              u.train(Ranged,EnemyDir,6)
28        u.train(Worker,Left,7)
29        for u in s:
30            u.build(Barracks,Right,1)
31        u.idle()
32        u.harvest(5)
33        u.attack(Strongest)
34
35def Sketch-SA-O-64x64(state s):
36    for u in s:
37        u.attack(Weakest)
38        for u in s:
39            u.train(Worker, Right, 6)
40            u.harvest(7)
41            for u in s:
42              u.build(Barracks, Left, 10)
43            if HaveUnitsAttacking(4):
44               u.attack(Farthest)
45            else:
46              u.idle()
47            u.train(Ranged, Down, 50)
```

Referências

AGUAS, J. S.; JIMÉNEZ, S.; JONSSON, A. Computing hierarchical finite state controllers with classical planning. **Journal of Artificial Intelligence Research**, v. 62, p. 755–797, 2018.

AHMED, U. Z.; GULWANI, S.; KARKARE, A. Automatically generating problems and solutions for natural deduction. In: **Twenty-Third International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2013.

AUER, P.; CESA-BIANCHI, N.; FISCHER, P. Finite-time analysis of the multiarmed bandit problem. **Machine learning**, Springer, v. 47, n. 2-3, p. 235–256, 2002.

BASTANI, O.; PU, Y.; SOLAR-LEZAMA, A. Verifiable reinforcement learning via policy extraction. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2018. p. 2499–2509.

BENBASSAT, A.; SIPPER, M. Evolving board-game players with genetic programming. In: **Genetic and Evolutionary Computation Conference**. [S.l.: s.n.], 2011. p. 739–742.

BOTEA, A.; MüLLER, M.; SCHAEFFER, J. Near optimal hierarchical path-finding. **Journal of Game Development**, v. 1, p. 7–28, 2004.

BROWN, G. Iterative solution of games by fictitious play, 1951. **Activity Analysis of Production and Allocation (TC Koopmans, Ed.)**, p. 374–376, 1951.

BUTLER, E.; TORLAK, E.; POPOVIĆ, Z. Synthesizing interpretable strategies for solving puzzle games. In: **Proceedings of the 12th International Conference on the Foundations of Digital Games**. [S.l.: s.n.], 2017. p. 1–10.

CANAAN, R. et al. Evolving agents for the hanabi 2018 cig competition. In: IEEE. **2018 IEEE Conference on Computational Intelligence and Games**. [S.l.], 2018. p. 1–8.

CARVALHO, D. V.; PEREIRA, E. M.; CARDOSO, J. S. Machine learning interpretability: A survey on methods and metrics. **Electronics**, MDPI, v. 8, n. 8, p. 832, 2019.

CHASLOT, G. et al. Monte-carlo tree search: A new framework for game ai. In: . [S.l.: s.n.], 2008.

CHEUNG, A.; SOLAR-LEZAMA, A.; MADDEN, S. Using program synthesis for social recommendations. In: **Proceedings of the 21st ACM international conference on Information and knowledge management**. [S.l.: s.n.], 2012. p. 1732–1736.

COULOM, R. Efficient selectivity and backup operators in monte-carlo tree search. In: SPRINGER. **International conference on computers and games**. [S.l.], 2006. p. 72–83.

DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 35, n. 6, p. 26–36, 2000.

FREITAS, J. M. D.; SOUZA, F. R. de; BERNARDINO, H. S. Evolving controllers for mario ai using grammar-based genetic programming. In: IEEE. **2018 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.], 2018. p. 1–8.

GAI, Y.; KRISHNAMACHARI, B.; JAIN, R. Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation. In: IEEE. **New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on**. [S.l.], 2010. p. 1–9.

GLENN, J. R.; ALOI, C. J. A generalized heuristic for can't stop. In: **FLAIRS Conference**. [S.l.: s.n.], 2009.

GULWANI, S. et al. Program synthesis. **Foundations and Trends® in Programming Languages**, Now Publishers, Inc., v. 4, n. 1-2, p. 1–119, 2017.

HAJEK, B. Cooling schedules for optimal annealing. **Mathematics of operations research**, INFORMS, v. 13, n. 2, p. 311–329, 1988.

HEINRICH, J.; LANCTOT, M.; SILVER, D. Fictitious self-play in extensive-form games. In: PMLR. **International conference on machine learning**. [S.l.], 2015. p. 805–813.

KELLER, M. **Can't Stop? Try the Rule of 28**. 1986. World Game Review 6.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. **science**, American association for the advancement of science, v. 220, n. 4598, p. 671–680, 1983.

Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. Optimization by Simulated Annealing. **Science**, v. 220, n. 4598, p. 671–680, maio 1983.

KOCSIS, L.; SZEPESVÁRI, C. Bandit based monte-carlo planning. In: SPRINGER. **European conference on machine learning**. [S.l.], 2006. p. 282–293.

LANCTOT, M. et al. A unified game-theoretic approach to multiagent reinforcement learning. In: **Proceedings of the International Conference on Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 4193–4206.

MARIÑO, J. R. H. et al. Programmatic strategies for real-time strategy games. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 35, n. 1, p. 381–389, 2021.

MORAES, R. O. et al. Action abstractions for combinatorial multi-armed bandit tree search. In: AAAI. **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.], 2018. p. 74–80.

NOURANI, Y.; ANDRESEN, B. A comparison of simulated annealing cooling strategies. **Journal of Physics A: Mathematical and General**, IOP Publishing, v. 31, n. 41, p. 8373, 1998.

ONTANÓN, S. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: **Ninth Artificial Intelligence and Interactive Digital Entertainment Conference**. [S.l.: s.n.], 2013.

ONTAÑÓN, S. Combinatorial multi-armed bandits for real-time strategy games. **Journal of Artificial Intelligence Research**, v. 58, p. 665–702, 2017.

ONTAÑÓN, S. **MicroRTS Competition**. 2017. <https://sites.google.com/site/microrttsaicompetition/>. [Online; accessed 15-August-2022].

ONTAÑÓN, S. **Results of the 2020 MicroRTS Competition**. 2020. <https://sites.google.com/site/micrortsaicompetition/competition-results/2020-cog-results>. Accessed: 2021-09-30.

ONTAÑÓN, S.; BURO, M. Adversarial hierarchical-task network planning for complex real-time games. In: **Proceedings of the International Conference on Artificial Intelligence**. [S.l.]: AAAI Press, 2015. p. 1652–1658. ISBN 9781577357384.

PETROSYAN, N.; ZENKEVICH, N. **Game theory**. [S.l.]: World Scientific, 2016. v. 3.

QIU, W.; ZHU, H. Programmatic reinforcement learning without oracles. In: **International Conference on Learning Representations**. [s.n.], 2022. Disponível em: <https://openreview.net/forum?id=6Tk2noBdvxt>.

ROBINSON, N. An iterative method of solving a game. **Annals of Mathematics**, v. 54, n. 2, p. 296–301, 1951.

ROSS, S.; GORDON, G.; BAGNELL, D. A reduction of imitation learning and structured prediction to no-regret online learning. In: **Proceedings of the International Conference on Artificial Intelligence and Statistics**. [S.l.]: PMLR, 2011. (Proceedings of Machine Learning Research, v. 15), p. 627–635.

SAMMUT, C. et al. Learning to fly. In: **Machine Learning Proceedings 1992**. [S.l.]: Elsevier, 1992. p. 385–393.

SCHAAL, S. Is imitation learning the route to humanoid robots? **Trends in Cognitive Sciences**, v. 3, p. 233–242, 1999.

SINGH, R. **Accessible programming using program synthesis.** Tese (Doutorado) — Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2014.

SOLAR-LEZAMA, A. The sketching approach to program synthesis. In: SPRINGER. **Asian Symposium on Programming Languages and Systems**. [S.l.], 2009. p. 4–13.

STANESCU, M. et al. Evaluating real-time strategy game states using convolutional neural networks. In: IEEE. **Proceedings IEEE Conference on Computational Intelligence and Games**. [S.l.], 2016. p. 1–7.

TORABI, F.; WARNELL, G.; STONE, P. Behavioral cloning from observation. **arXiv preprint arXiv:1805.01954**, 2018.

TORABI, F.; WARNELL, G.; STONE, P. Behavioral cloning from observation. In: **Proceedings of the International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2018.

VERMA, A. et al. Imitation-projected programmatic reinforcement learning. In: **Advances in Neural Information Processing Systems**. [S.l.]: Curran Associates, Inc., 2019. v. 32, p. 1–12.

VERMA, A. et al. Programmatically interpretable reinforcement learning. In: **Proceedings of the International Conference on Machine Learning**. [S.l.: s.n.], 2018. p. 5052–5061.

Apêndices

APÊNDICE A – Domain-Specific Languages usadas para o Can't Stop e o MicroRTS

A.1 Domain-Specific Language para Can't Stop

A DSL que desenvolvemos para Can't Stop é descrita pela seguinte gramática livre de contexto.

$$
\begin{aligned}
S &\rightarrow \quad AA \\
A &\rightarrow \quad \text{if}(B < B) \text{ then } A \text{ else } A \quad | \quad \text{argmax}(L) \quad | \quad \text{sum}(L) \\
&\rightarrow \quad E \otimes E \\
B &\rightarrow \quad N \quad | \quad E \otimes E \\
E &\rightarrow \quad E \otimes E \quad | \quad N \quad | \quad \text{sum}(L) \quad | \quad L_2 \quad | \quad F_2 \\
L &\rightarrow \quad \text{map}(\lambda F_1, L) \quad | \quad L_1 \quad | \quad \text{locallist} \\
\lambda F_1 &\rightarrow \quad \text{sum}(L) \quad | \quad \text{map}(\lambda F_1, L) \quad | \quad E \otimes E \\
F_2 &\rightarrow \quad \text{NumberAdvancedThisRound} \\
&\rightarrow \quad \text{NumberAdvancedByAction} \\
&\rightarrow \quad \text{PositionsPlayerHasSecuredInColumn} \\
&\rightarrow \quad \text{PositionsOpponentHasSecuredInColumn} \\
&\rightarrow \quad \text{DifficultyScore} \\
&\rightarrow \quad \text{IsNewNeutral} \\
L_1 &\rightarrow \quad \text{neutrals} \quad | \quad \text{actions} \\
L_2 &\rightarrow \quad \text{progress\_value} \quad | \quad \text{move\_value} \\
N &\rightarrow \quad 0 \quad | \quad 1 \\
\otimes &\rightarrow \quad + \quad | \quad - \quad | \quad *
\end{aligned}
$$

Essa DSL permite que nosso sistema sintetize ações para ambas as decisões em Can't Stop: decisão sim-não e decisão de coluna. F1 representa as funções lambda usadas como parâmetros para o operador do `map` e F2 é o conjunto de funções específicas do domínio que fornecem informações sobre um estado do jogo. A seguir, descrevemos essas funções:

- *NumberAdvancedThisRound*:Calcula o número de células que o jogador avançou na rodada atual
- *NumberAdvancedByAction*: Calcula o número de células que o jogador avançará se executar a ação passada como parâmetro
- *PositionsPlayerHasSecuredInColumn*: Calcula quantas células o jogador garantiu em uma coluna passada como parâmetro
- *PositionsOpponentHasSecuredInColumn*: Calcula quantas células o oponente garantiu em uma coluna passada como parâmetro
- *DifficultyScore*: Calcula uma pontuação de dificuldade (GLENN; ALOI, 2009) de um estado, onde o valor retornado depende da posição das fichas neutras; um estado é considerado mais difícil se os tokens neutros estiverem em colunas ímpares ou se todos os tokens estiverem em colunas com números menores que 7 ou maiores que 7
- *IsNewNeutral*: Retorna 1 se a ação for usar uma ficha neutra, retorna 0 caso contrário

A seguir, descrevemos as listas de L, L_1 e L_2 utilizadas em nosso DSL:

- *locallist*: O conjunto de ações é representado como uma lista de listas. Quando usado em um operador `map`, chamamos as listas dentro de uma lista de ação de "locallist", que é representada na DSL como l_1.
- *neutrals*: Lista que representa em quais colunas as fichas neutras estão localizados no estado passado como parâmetro
- *actions*: Lista que representa as ações disponíveis para o estado passado como parâmetro
- *progress_value*: Lista que representa os pesos das colunas utilizadas na estratégia de Glenn e Aloi. Em sua estratégia, esse conjunto de pesos é usado na decisão sim-não. Em nosso DSL, esta lista pode ser utilizada por ambas as ações. progresso_value = $[7, 7, 3, 2, 2, 1, 2, 2, 3, 7, 7]$
- *move_value*: Lista que representa os pesos das colunas utilizadas na estratégia de Glenn e Aloi na decisão da coluna. Em nosso DSL, esta lista pode ser usada em ambas as decisões. mover_valor = $[7, 0, 2, 0, 4, 3, 4, 0, 2, 0, 7]$

A.2 Domain-Specific Language para MicroRTS

Nosso DSL para MicroRTS é inspirado no de MARIÑO et al. (2021). A DSL é descrita pela seguinte gramática livre de contexto.

$S \rightarrow$ SS | for S | if(B) then S | if(B) then S else S | C | λ

$B \rightarrow$ HasNumberOfUnits(T, N) | OpponentHasNumberOfUnits(T, N)

$\rightarrow$ HasLessNumberOfUnits(T, N) | HaveQtdUnitsAttacking(N)

$\rightarrow$ HasUnitWithinDistanceFromOpponent(N)

$\rightarrow$ HasNumberOfWorkersHarvesting(N) | is_Type(T) | IsBuilder

$\rightarrow$ CanAttack | HasUnitThatKillsInOneAttack

$\rightarrow$ OpponentHasUnitThatKillsUnitInOneAttack

$\rightarrow$ HasUnitInOpponentRange | OpponentHasUnitInPlayerRange

$\rightarrow$ CanHarvest

$C \rightarrow$ $Build(T, D, N)$ | $Train(T, D, N)$ | $moveToUnit(T_p, O_p)$

$\rightarrow$ $Attack(O_p)$ | $Harvest(N)$ | $Idle$ | $MoveAway$

$T \rightarrow$ Base | Barracks | Ranged | Heavy | Light | Worker

$N \rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 15

$\rightarrow$ 20 | 25 | 50 | 100

$D \rightarrow$ EnemyDir | Up | Down | Right | Left

$O_p \rightarrow$ Strongest | Weakest | Closest | Farthest

$\rightarrow$ LessHealthy | MostHealthy | Random

$T_p \rightarrow$ Ally | Enemy

Este DSL permite loops aninhados e condicionais. Ele contém várias funções booleanas (B) e funções orientadas a comandos (C) que fornecem informações sobre o estado atual do jogo ou comandos para as unidades aliadas.

A seguir, descrevemos as funções booleanas usadas em nossa DSL.

- *HasNumberOfUnits(T,N)*: Verifica se o jogador aliado possui N unidades do tipo T
- *OpponentHasNumberOfUnits(T,N)*:Verifica se o jogador adversário possui N unidades do tipo T
- *HasLessNumberOfUnits(T,N)*: Verifica se o jogador aliado tem menos de N unidades do tipo T
- *HaveQtdUnitsAttacking(N)*:Verifica se o jogador aliado tem N unidades atacando o oponente

- *HasUnitWithinDistanceFromOpponent(N)*:Verifica se o jogador aliado tem uma unidade dentro de uma distância N de uma unidade do oponente
- *HasNumberOfWorkersHarvesting(N)*:Verifica se o jogador aliado possui N unidades do tipo Worker colhendo recursos
- *is_Type(T)*: Verifica se uma unidade do tipo T
- *IsBuilder*: Verifique se uma unidade é do tipo Worker
- *CanAttack*: Verifica se uma unidade pode atacar
- *HasUnitThatKillsInOneAttack*: Verifica se o jogador tem uma unidade que mata uma unidade do oponente com um ataque
- *OpponentHasUnitThatKillsUnitInOneAttack*: Checks if the opponent player has a unit that kills an ally's unit with one attack action
- *HasUnitInOpponentRange*: Verifica se o jogador adversário tem uma unidade que mata a unidade com um ataque
- *OpponentHasUnitInPlayerRange*: Verifica se uma unidade do jogador oponente está dentro do alcance de ataque de uma unidade aliada
- *CanHarvest*: Verifica se uma unidade pode coletar recursos

A seguir, descrevemos as funções orientadas a comandos usadas em nossa DSL:

- *Build(T,D,N)*: Treina N unidades do tipo T em uma célula localizada na direção D
- *Train(T,D,N)*: Treina N unidades do tipo T em uma célula localizada na direção D da estrutura responsável por treiná-las
- *moveToUnit(T_p, O_p)*: Comanda uma unidade para se mover em direção ao jogador T_p seguindo um critério O_p
- *Attack(O_p)*: Comanda uma unidade para atacar unidades do jogador adversário seguindo um critério O_p
- *Harvest(N)*:Envia N unidades de Worker para colher recursos
- *Idle*:Comanda uma unidade para ficar ociosa e atacar se uma unidade adversária entrar em seu alcance de ataque
- *MoveAway*:Comanda uma unidade para se mover na direção oposta da base do jogador

T representa os tipos que uma unidade pode assumir. N é um conjunto de inteiros. D representa as direções disponíveis usadas nas funções de ação. O_p é um conjunto de critérios para selecionar uma unidade adversária com base em seu estado atual. T_p representa o conjunto de jogadores. Veja os diferentes tipos, números inteiros, direções, critérios e jogadores que usamos na gramática livre de contexto acima.