Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Aprendizado por Refoço Relacional:
uma análise de eficiência

Thiago Yukio Sikusawa

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisora: Prof.ª Leliane Nunes de Barros

São Paulo
2024

Esta seção é opcional e fica numa página separada;
ela pode ser usada para uma dedicatória ou epígrafe.

Agradecimentos

Do. Or do not. There is no try.
— Mestre Yoda

Texto texto. Texto opcional.

Resumo

Thiago Yukio Sikusawa. **Aprendizado por Refoço Relacional:: *uma análise de eficiência***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

Abstract

Thiago Yukio Sikusawa. **Relational Reinforcement Learning: *an analysis of its efficiency***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de abreviaturas

MDP	Processo de Markov de Decisão (*Markov Decision Process*)
DP	Dynamic Programming (*Dynamic Programming*)
MC	Monte Carlo (*Monte Carlo*)
TD	Temporal Difference (*Temporal Difference*)
RRL	Aprendizado por reforço relacional (*Relational Reinforcement Learning*)

Lista de símbolos

ω	Frequência angular
ψ	Função de análise *wavelet*
Ψ	Transformada de Fourier de ψ

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Apêndices

Anexos

Capítulo 1

Introdução

A essência de **problemas de tomada de decisão sequencial** é aprender interagindo com o ambiente, o que é uma ideia bem natural, já que é assim que todos seres humanos aprendem. O objetivo é conseguir mapear toda situação a alguma ação para ser executada, de tal modo que alguma forma de recompensa seja maximizada.

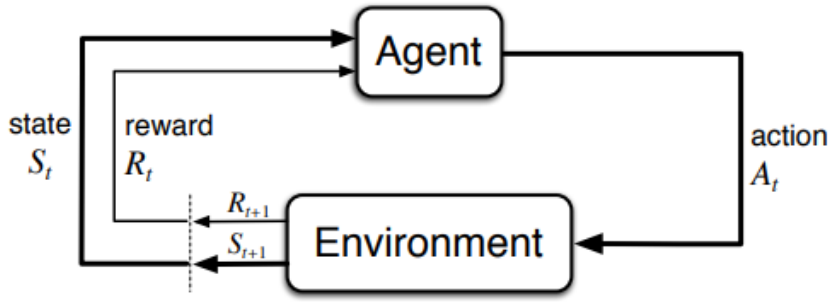


Figura 1.1: *A interação entre o agente e o ambiente.. Fonte: Reinforcement Learning: An Introduction, p. 54 (Sutton e Barto, 2015)*

Em qualquer problema de tomada de decisão sequencial há dois elementos principais: o **agente**, que realiza ações com o objetivo de maximizar alguma recompensa, e o **ambiente**, que é tudo que interage e que é interagido pelo agente.

A Figura 1.1 demonstra a relação entre o agente e o ambiente: Em um passo no tempo t, o ambiente encontra-se no estado S_t. O agente vê esse estado, e reponde executando uma ação A_t. Essa ação afeta o ambiente, resultando na mudança do estado do ambiente de S_t para S_{t+1}, e, além disso, o ambiente devolve uma recompensa R_{t+1} para o agente, indicando para o agente a qualidade de sua escolha de ação.

Um fator importante nessa relação é o modo como o ambiente escolhe S_{t+1} e R_{t+1} quando o ambiente está no estado S_t e o agente faz a ação A_t. Esse comportamento do ambiente é chamado de **dinâmica do ambiente**, e pode ser descrito completamente com uma **função de transição probabilística** p, tal que $p(S_{t+1}, R_{t+1} \mid S_t, A_t)$ é a probabilidade

de que o ambiente devolverá o estado S_{t+1} e a recompensa R_{t+1} quando o ambiente estiver no estado S_t e o agente fizer a ação A_t.

O objetivo final do agente é maximizar a longo prazo o total de recompensa acumulada por um dado horizonte de interações. Por causa disso, ter conhecimento total da dinâmica do ambiente, ou seja, da função p, é uma grande vantagem para cumprir esse objetivo. De fato, problemas de tomada de decisão sequencial em que conhecemos toda a função p são chamados de **problemas de planejamento probabilístico**. Em contrapartida, problemas de tomada de decisão sequencial em que a função p não é conhecida são chamados de **problemas de aprendizagem por reforço**.

1.1 Modelo do ambiente: enumerativo ou fatorado

Uma parte importante de um problema de tomada de decisão sequencial é como os seus estados serão representados, pois o estado do ambiente é um dos fatores que pode determinar qual ação o agente executará. Assim, a forma como representamos os estados do ambiente determina que tipo de informação o agente receberá do ambiente para decidir a sua próxima ação.

A forma mais simples de representar os estados do ambiente é chamado de **modelo enumerativo**, o qual simplesmente representa cada estado com um número diferente. Esse modelo resulta em informação mínima sendo enviada para o agente, quem só saberá distinguir se dois estados são iguais ou diferentes. Um exemplo de interação entre o agente e um ambiente com estados representado com o modelo enumerato é:

> **Ambiente:** Você está no estado 22.
> **Agente:** Eu faço ação 3.
> **Ambiente:** Você recebeu recompensa −4. Você está no estado 10.
> **Agente:** Eu faço ação 7.
> **Ambiente:** Você recebeu recompensa +2. Você está no estado 29.
> **Agente:** ...

Uma forma mais expressiva de representar os estados do ambiente é chamado de **modelo fatorado**. Nesse modelo, cada estado é representado como um conjunto de informações sobre o próprio estado. Com esse modelo, o agente pode usar as informações presentes na representação dos estados para ajudar na escolha de sua próxima ação.

Neste trabalho, veremos dois tipos de modelos fatorado. O primeiro é chamado de **modelo fatorado proposicional**, o qual representa cada estado do ambiente como um vetor de atributos para cada propriedade presente no ambiente. O segundo é chamado de **modelo fatorado relacional**. Este separa o ambiente em diversos objetos, e cada objeto tem diversas propriedades e pode ter relações com outros objetos. Assim, o modelo fatorado relacional representa cada estado com o conjunto de propriedades dos objetos, e das relações entre os objetos.

1.2 Proposta deste trabalho

Neste trabalho, estamos interessados em comparar a eficiência de algoritmos de problemas de aprendizagem por reforço que usam o modelo fatorado proposicional de representação dos estados, com os que usam o modelo fatorado relacional. Mais detalhadamente, queremos comparar tanto a quantidade de treinamento necessário assim como a quantidade de tempo usado pelos algoritmos para treinar o agente.

Veremos que algoritmos que usam o modelo fatorado relacional possuem potencial de aprenderem com bem menos treinamento do agente necessário comparado com algoritmos que usam o modelo fatorado proposicional. Porém, o primeiro também apresenta uma maior inconsistência nos resultados obtidos, e também precisam ser rodados por mais tempo do que o segundo.

No capítulo 2, apresentaremos conceitos básicos de problemas de planejamento probabilísticos, e também como resolvê-los. Nos capítulos 3 e 4, introduziremos algoritmos para resolver problemas de aprendizagem por reforço. No capítulo 5 introduziremos uma classe de algoritmos que resolvem problemas de aprendizagem por reforço com estados representados pelo modelo fatorado relacional, assim como o domínio do Mundo dos Blocos, o qual será usado nos experimentos. Nos capítulos 6 e 7, apresentaremos dois algoritmos da classe que introduzimos no capítulo 5, e veremos como eles resolvem problemas no domínio do Mundo dos Blocos com experimentos. Finalmente, no capítulo 8 Compararemos e analisaremos todos os resultados dos experimentos para concluír o trabalho.

Capítulo 2

Processo de Decisão Markoviano - MDP

Neste capítulo faremos a formalização de Processos de Decisão Markovianos em que conhecemos a função probabilística de transição de estados e mostraremos como avaliar políticas e como encontrar políticas ótimas.

É possível descrever qualquer problema de tomada de decisão sequencial como um **Processo de Decisão Markoviano** (Puterman, 1994) (do inglês *Markov Decision Process*, abreviado como **MDP**). Um MDP é definido como uma tupla $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$, em que:

- $\mathcal{S}$ é o conjunto de todos os estados possíveis;
- $\mathcal{A}$ é o conjunto de todas as ações possíveis;
- $\mathcal{R} \subseteq \mathbb{R}$ é o conjunto de todas as recompensas possíveis;
- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$ é a função probabilística de transição. Assim, dado um estado $s \in \mathcal{S}$ e uma ação $a \in \mathcal{A}$, o valor de $p(s', r \mid s, a)$ é a probabilidade de, ao fazer a ação a quando no estado s, receber uma recompensa $r \in \mathcal{R}$ e terminar no estado $s' \in \mathcal{S}$

Mais detalhadamente, para cada passo no tempo $t \in \mathbb{N}$, podemos dizer que o ambiente está no estado $S_t \in \mathcal{S}$, e que baseado nisso o agente escolhe uma ação $A_t \in \mathcal{A}$. Assim, no próximo passo no tempo o agente recebe uma recompensa $R_{t+1} \in \mathcal{R}$ e encontra-se em um novo estado $S_{t+1} \in \mathcal{S}$. Dessa forma, o MDP e o agente criam uma trajetória: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$.

O objetivo de todo agente modelado como um MDP é maximizar a recompensa esperada acumulada em um intervalo de tempo. Dividiremos esse período de tempo discretamente, assim, podemos descrever o problema em um passo no tempo $t \in \mathbb{N}$ específico.

Seja $\mathcal{S}$ o conjunto de todos os estados possíveis no ambiente, e seja $\mathcal{A}$ o conjunto de todas as ações que o agente pode realizar. Caso tenhamos um problema em que as possíveis ações dependem de qual estado o ambiente está, para todo $s \in \mathcal{S}$ defina $\mathcal{A}(s) \subseteq \mathcal{A}$ como o conjunto de ações que o agente pode tomar no estado s.

Em um MDP *finito*, temos que $\mathcal{S}$, $\mathcal{A}$ e $\mathcal{R}$ têm tamanhos finitos, e neste caso, para cada $t \in \mathbb{N}$, segue que R_t e S_t são variáveis aleatórias com com distribuição de probabilidade discreta dependente apenas do estado e da ação precedente. Além disso, a função p é o que determina toda a *dinâmica* da MDP. Veja que como p é uma distribuição de probabilidade dependente apenas de s e r, temos que:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \text{ para cada } s \in \mathcal{S} \text{ e } a \in \mathcal{A}(s).$$

Veja que a probabilidade de cada par de valores de S_t e R_t é determinado completamente por S_{t-1} e A_{t-1}, e que se soubermos isso não é necessário saber nada sobre os passos no tempo anteriores a $t-1$. Por causa disso, é importante que todos os estados do ambiente incluam todas as informações das interações passadas entre o agente e o ambiente, para que haja diferença nas interações futuras. Se esse for o caso, dizemos que os estados têm a **propriedade de Markov**. A partir desse ponto, assumiremos que todos os problemas têm estados que respeitam a propriedade de Markov.

2.1 Recompensa e retorno

Em todo problema de tomada de decisão sequencial, a cada passo no tempo $t \in \mathbb{N}$, o agente recebe uma recompensa imediata $R_t \in \mathbb{R}$. Queremos que o agente consiga maximizar a soma total de recompensa que ele recebe, o que significa que ele não deveria focar em uma recompensa imediata, mas sim na recompensa acumulada a longo prazo.

Formalmente, para todo passo no tempo $t \in \mathbb{N}$, queremos maximizar o valor esperado do **retorno** (Sutton e Barto, 2015), denotado G_t, que é definido como alguma função da sequência de recompensas $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$. Um exemplo simples de definir o retorno é:

$$G_t := R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{2.1}$$

em que $T \in \mathbb{N}$ é algum último passo no tempo do problema. Tal definição faz sentido se houver alguma noção natural de "último passo no tempo", isto é, se a interação entre o agente e o ambiente pode ser separado naturalmente em subsequências, chamados de **episódios** (Sutton e Barto, 2015). Cada episódio termina em um estado especial chamado de **estado terminal** (Sutton e Barto, 2015).

Problemas com episódios são chamados de **problemas episódicos** (Sutton e Barto, 2015), e nesses tipos de problemas pode ser importante distinguir o conjunto de todos os estados não terminais, denotados por $\mathcal{S}$, do conjunto de todos os estados juntos com os estados terminais, denotado por $\mathcal{S}^+$.

Por outro lado, em múltiplos problemas a interação entre o agente e o ambiente não tem nenhuma forma de ser naturalmente separado em episódios, e ao invés disso só continua sem limite. Tais problemas são chamados de **problemas contínuos** (Sutton e Barto, 2015). Nesses casos, definir G_t como na Definição (2.1) não seria ideal, pois teríamos que

o último passo no tempo é $T = \infty$, assim, a soma pode facilmente divergir. Para resolver isso, introduzimos uma constante $\gamma \in \mathbb{R}$ tal que $0 \leq \gamma \leq 1$, chamada de **fator de desconto** (Sutton e Barto, 2015). Assim, podemos definir o retorno como:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{2.2}$$

O fator de desconto determina o quão importante as recompensas do futuro são para o valor do retorno, pois para cada $k \in \mathbb{N}$, temos que uma recompensa k passos no tempo no futuro vale γ^{k-1} vezes menos do que valeria se fosse uma recompensa imediata. Veja que se $\gamma < 1$, então desde que a sequência $\{R_k\}$ seja limitada, garantimos que a soma da Definição (2.2) converge.

Note que dado um problema episódico, é possível representar o retorno da Definição (2.1) usando a Definição (2.2). Basta fazer o fator de desconto $\gamma = 1$, e definir que para todo $k > T$ tem-se que $R_k = 0$. Assim, a partir desse ponto, usaremos a Definição (2.2) de retorno para ambos problemas episódicos e contínuos.

2.2 Política e função valor

Uma parte bem importante de múltiplos algoritmos de planejamento probabilístico e de aprendizado por reforço é uma **função valor** (Sutton e Barto, 2015), que são funções que determinam quão bom é um agente encontrar-se em um dado estado, ou quão bom é fazer uma dada ação em um estado. Veja que a noção de "quão bom" depende das recompensas futuras, que determinam o valor esperado do retorno. Obviamente essas recompensas dependem de que ações o agente escolherá, logo a função valor depende do comportamento do agente. Este comportamento do agente é chamado de **política** (Sutton e Barto, 2015).

Formalmente, uma política é um mapeamento dos estados para as probabilidades do agente selecionar cada ação. Assim, seja π a política que um agente está seguindo, então em cada passo de tempo t, definimos que $\pi(a|s)$ é a probabilidade que $A_t = a$ dado que $S_t = s$, para cada $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$. Uma política é dita **determinística** se para todo estado $s \in \mathcal{S}$ tem-se que $\pi(a|s) = 1$ para apenas um $a \in \mathcal{A}(s)$, e nesse caso, denotamos que $\pi(s) = a$.

Definimos a **função valor-estado** (Sutton e Barto, 2015) em um estado $s \in \mathcal{S}$ sob a política π, denotada como $v^\pi(s)$, como o valor esperado do retorno quando começando no estado s e sempre seguindo a política π. Em MDPs podemos definir como:

$$v^\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \forall s \in \mathcal{S},$$

em que $\mathbb{E}_\pi[\cdot]$ denota o valor esperado de uma variável aleatória dado que o agente segue a política π, e t é qualquer passo no tempo.

Similarmente, defina a **função valor-ação** (Sutton e Barto, 2015) quando tomando

uma ação $a \in \mathcal{A}(s)$ em um estado $s \in \mathcal{S}$ sob a política π, denotada $q^\pi(s,a)$, como o valor esperado do retorno quando começando no estado s, escolhendo a ação a, e depois sempre seguindo a política π. Ou seja:

$$q^\pi(s,a) := \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right], \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s).$$

Uma propriedade fundamental das funções valor é que elas satisfazem uma relação recursiva, devido ao fato de que:

$$\begin{aligned} G_t &:= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \sum_{k=0}^{\infty} \gamma^{k+1} R_{t+k+2} = R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \\ &= R_{t+1} + \gamma G_{t+1}, \end{aligned} \tag{2.3}$$

assim, para cada política π e qualquer estado $s \in \mathcal{S}$, temos que:

$$\begin{aligned} v^\pi(s) &:= \mathbb{E}_\pi[G_t \mid S_t = s] \overset{(2.3)}{=} \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']) \\ &= \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v^\pi(s')), \end{aligned} \tag{2.4}$$

e similarmente, para todo $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$, temos que:

$$\begin{aligned} q^\pi(s,a) &:= \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \overset{(2.3)}{=} \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \sum_{a' \in \mathcal{A}(s')} \pi(a'|s')(r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_t = s', A_t = a']) \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \sum_{a' \in \mathcal{A}(s')} \pi(a'|s')(r + \gamma q^\pi(s', a')). \end{aligned} \tag{2.5}$$

A equação (2.4) é chamado de **Equação de Bellman para** v^π, e similarmente, a equação (2.5) é chamado de **Equação de Bellman para** q^π.

Dado uma política π, é de se esperar que as funções valor v^π e q^π tenham alguma relação juntas, e de fato, algumas propriedades importantes delas são:

- $q^\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma v^\pi(S_{t+1}) \mid S_t = s, A_t = a]$, para todo $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$.
- $v^\pi(s) = q^\pi(s, \pi(s))$ para todo $s \in \mathcal{S}$, se π for determinístico.

2.3 Política ótima e função valor ótima

O objetivo final de um problema de planejamento probabilístico e de aprendizado por reforço é encontrar alguma política que obtêm bastante recompensa a longo prazo. Em MDPs, conseguimos definir uma ordenação parcial entre políticas, em que dizemos que uma política π é melhor ou igual a uma política π', denotado como $\pi \geq \pi'$, se para todo estado o valor esperado do retorno em π é maior ou igual do que em π'. Em outras palavras, $\pi \geq \pi'$ se, e somente se, $v^{\pi}(s) \geq v^{\pi'}(s)$ para todo $s \in \mathcal{S}$.

Sempre existirá pelo menos uma política que é melhor ou igual a qualquer outra política. Tal política é chamada de **política ótima** (Sutton e Barto, 2015), e mesmo possivelmente existindo mais do que uma, denotamos todas políticas ótimas com π^*. Todas compartilham a mesma função valor-estado, chamada de **função valor-estado ótima** (Sutton e Barto, 2015), denotada como v^*, e é definida como:

$$v^*(s) := \max_{\pi} v^{\pi}(s),$$

para todo $s \in \mathcal{S}$.

Políticas ótimas também compartilham a mesma função valor-ação, chamada de **função valor-ação ótima** (Sutton e Barto, 2015), denotado como q^*, e definida como:

$$q^*(s, a) := \max_{\pi} q^{\pi}(s, a),$$

para todo $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$.

Note que como v^* e q^* são funções valor, elas devem satisfazerem a Equação de Bellman, porém, podemos usar o fato de que elas representam políticas ótimas para obter as **Equações de Bellman de otimalidade** (Sutton e Barto, 2015). Intuitivamente, as seguintes equações dizem que o valor de um estado com uma política ótima deve sempre selecionar a ação que maximiza o valor esperado:

$$\begin{aligned} v^*(s) &= \max_{a \in \mathcal{A}(s)} q^*(s, a) \\ &= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v^*(s')), \end{aligned} \tag{2.6}$$

e equivalentemente para q^*:

$$q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') \mid S_t = s, A_t = a]$$
$$= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma \max_{a'} q^*(s', a')). \tag{2.7}$$

Dado uma política qualquer π, se as funções valor de π satisfazerem as Equações de Bellman de otimalidade, então é possível concluir que π é uma política ótima.

2.4 Soluções para MDPs conhecidos

O método Progrmação Dinâmica (abreviado como DP, do inglês *Dynamic Programming*) (Sutton e Barto, 2015) refere-se a algoritmos que computam políticas ótimas, assumindo que tenhamos um modelo perfeito das dinâmicas do ambiente no MDP (ou seja, que conhecemos o valor de $p(s', r \mid s, a)$ para todo $s, s' \in \mathcal{S}$, $a \in \mathcal{A}(s)$, e $r \in \mathcal{R}$). Porque precisamos conhecer a função p, o método DP é usado para resolver problemas de planejamento probabilístico, mas não problemas de aprenizado por reforço.

A partir desse capítulo, assumiremos que os ambientes dos problemas possuem um MDP finito, ou seja, que os conjuntos $\mathcal{S}$, $\mathcal{A}$, e $\mathcal{R}$ tenham tamanho finito. A ideia principal de DP é utilizar as funções valor para organizar e conseguir procurar políticas boas. Para isso, vamos primeiro resolver dois problemas menores:

1. Dado uma política π, descobrir a função valor-estado v^π.
2. Dado a função valor-estado v^π, encontrar uma política melhor do que π.

Assim, a estratégia será alternar entre os dois problemas para que possamos encontrar políticas melhores até chegar em uma política ótima.

2.5 Avaliação de política (predição)

O processo de computar a função valor-estado v^π dado uma política π é chamado de **avaliação de política** (Sutton e Barto, 2015). Fazer isso é um problema bem comum quando lidando com planejamento probabilístico, e é normalmente referenciado como o **problema de predição** (Sutton e Barto, 2015). Recorde que por (2.4) temos que:

$$v^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v^\pi(s')),$$

para todo $s \in \mathcal{S}$. A existência e unicidade da função v^π é garantida desde que $\gamma < 1$. Veja que se soubermos a função p, a equação acima é um sistema linear de $|\mathcal{S}|$ variáveis e $|\mathcal{S}|$ equações, então já é possível computar analiticamente a função v^π.

Porém, ao invés disso consideraremos um método iterativo. Considere uma sequência de aproximações da função valor-estado: $v_0, v_1, v_2, \ldots$, cada um mapeando $\mathcal{S}$ para $\mathbb{R}$. A

aproximação inicial v_0 é escolhida arbitrariamente (exceto que todos os estados terminais são mapeados para 0), e para obter a próxima aproximação, usamos a Equação de Bellman (2.4) iterativamente:

$$v_{k+1}(s) := \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v_k(s')), \tag{2.8}$$

para todo $s \in \mathcal{S}$ e $k \in \mathbb{N}$. A equação de Bellman nos garante que $v_k = v^\pi$ é um ponto fixo, e de fato, pode ser provado que a sequência $\{v_k\}$ converge para v^π quando $k \longrightarrow \infty$ quando $\gamma < 1$. Esse algoritmo é chamado de **avaliação de política iterativa** (Sutton e Barto, 2015).

2.6 Aperfeiçoamento de política (controle)

Dado uma política π e sua função valor-estado v^π, queremos encontrar uma política π' tal que $\pi' \geq \pi$. De novo, fazer isso é bem comum em problemas de planejamento probabilístico, e este problema é normalmente chamado de **problema de controle** (Sutton e Barto, 2015). Por enquanto vamos focar apenas em políticas determinísticas.

Dado uma política π e um estado s, queremos descobrir se seria vantajoso mudar a política para que ela escolha alguma ação $a \neq \pi(s)$. Como já sabemos o valor de $v^\pi(s)$, uma estratégia é compará-lo com o valor de $q^\pi(s, a)$. Lembrando que:

$$\begin{aligned} q^\pi(s, a) &= \mathbb{E}[R_{t+1} + \gamma v^\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v^\pi(s')), \end{aligned} \tag{2.9}$$

então é possível computar $q^\pi(s, a)$ com a função v^π.

Se acontecer que $q^\pi(s, a) \geq v^\pi(s)$, pode ser esperado que sempre será vantajoso escolher a ação a quando no estado s. Isso de fato é verdade, e é um caso particular do *teorema do aperfeiçoamento de política* (Sutton e Barto, 2015):

> Sejam π e π' duas políticas determinísticas. Se para todo $s \in \mathcal{S}$ tem-se que $q^\pi(s, \pi'(s)) \geq v^\pi(s)$, então $v^{\pi'}(s) \geq v^\pi(s)$ para todo $s \in \mathcal{S}$, ou seja, $\pi' \geq \pi$. Além disso, se para algum estado $s \in \mathcal{S}$ tem-se que $q^\pi(s, \pi'(s)) > v^\pi(s)$, então $v^{\pi'}(s) > v^\pi(s)$.

Usando o teorema acima, é possível construir uma política *gulosa* π' que seja melhor ou igual a política original π. Basta definir que:

$$\pi'(s) := \operatorname*{argmax}_a q^\pi(s, a), \tag{2.10}$$

em que argmax_a é uma função que retorna uma ação que maximiza a expressão que segue

(os empates são decidios arbitrariamente). Não é difícil verificar que π' satisfaz as condições do teorema do aperfeiçoamento de política, então sabemos que $\pi' \geq \pi$. Esse processo de construir uma política nova, deixando-a gulosa a respeito da função valor da política original, é chamado de **aperfeiçoamento de política** (Sutton e Barto, 2015).

Suponha que ao construir a política gulosa π', ela seja tão bom como a original, mas não melhor, ou seja, que $v^{\pi'} = v^{\pi}$. Então de (2.10), isso significa que:

$$\begin{aligned} v^{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v^{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s' \in S} \sum_{r \in R} p(s', r|s, a)(r + \gamma v^{\pi'}(s')), \end{aligned}$$

mas veja que isso é a Equação de Bellman de otimalidade (2.6), logo, $v^{\pi'} = v^*$, e tanto π como π' são políticas ótimas. Em outras palavras, aperfeiçoamento de política sempre construirá uma política estritamente melhor, a não ser que a política já seja ótima.

2.7 Iteração de política

Como discutimos antes, podemos usar as técnicas de avaliação de política e de aperfeiçoamento de política para gerar cada vez políticas melhores, eventualmente convergindo para π^*:

$$\pi_0 \xrightarrow{\text{Av}} v^{\pi_0} \xrightarrow{\text{Ap}} \pi_1 \xrightarrow{\text{Av}} v^{\pi_1} \xrightarrow{\text{Ap}} \pi_2 \xrightarrow{\text{Av}} \cdots \xrightarrow{\text{Ap}} \pi^* \xrightarrow{\text{Av}} v^*$$

em que $\xrightarrow{\text{Av}}$ denota avaliação de política, e $\xrightarrow{\text{Ap}}$ denota aperfeiçoamento de política.

Essa estratégia de encontrar uma política ótima é chamada de **iteração de política** (Sutton e Barto, 2015). Segue o pseudocódigo desse algoritmo:

Programa 2.1 Iteração de política.

1 *Entrada*: *MDP* $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, $\gamma \in (0, 1]$, **e** $\theta > 0$ (*um número real positivo pequeno, determinando a acurácia da estimação*)
2 1. *Inicialização*:
3 *Escolha* $V(s) \in \mathbb{R}$ **e** $\pi(s) \in \mathcal{A}(s)$, **para** *todo* $s \in \mathcal{S}$ *arbitrariamente*
4 *Faça* $V(s) = 0$ **para** *todo* $s \in \mathcal{S}$ *que seja terminal*
5 2. *Avaliação* **de** *política*:
6 *Faça*:
7 $\Delta \leftarrow 0$
8 **Para** *cada estado* $s \in \mathcal{S}$ *faça*:
9 $v \leftarrow V(s)$
10 $V(S) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))(r + \gamma V(s'))$
11 $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
12 **enquanto** $\Delta > \theta$
13 3. *Aperfeiçoamento* **de** *política*:
14 $politica\_estavel \leftarrow true$
15 **Para** *cada* $s \in \mathcal{S}$ *faça*:
16 $acao\_velha \leftarrow \pi(s)$
17 $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$
18 **Se** $acao\_velha \neq \pi(s)$ *então* $politica\_estavel \leftarrow false$
19 **Se** $politica\_estavel$ *então pare* **e** *retorne* $V \approx v^*$, **e** $\pi \approx \pi^*$; **se** *não, volte* **para** 2.

No Programa 2.1, a linha 1 recebe de entrada um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, e os valores de γ e θ do usuário. As linhas 2 a 4 inicializam uma política π arbitrária, e uma estimação V de V^π arbitrária (garantindo que $V(s) = 0$ para todo $s \in \mathcal{S}$ terminal). As linhas 9 e 10 são uma iteração de avaliação de política visto na Definição (2.8) e na Equação (2.9). A linha 11 calcula Δ, a maior mudança em V após a iteração de avaliação de política. Se essa mudança for maior que θ, então o algoritmo faz mais uma iteração de avaliação de política. A linha 17 faz o aperfeiçoamento de política visto na Definição (2.10). As linhas 14, 16 e 17 determinam se o aprefeiçoamento de política resultou em uma política difeente. Se a política mudou no passo de aprefeiçoamento de política, a linha 19 faz o algoritmo voltar ao passo de avaliação de política. Caso contrário, encontramos uma política ótima, então paramos o programa.

2.8 Iteração de valor

Uma desvantagem da iteração de política é que cada uma de suas iteração requer que seja feito avaliação de política, que pode consumir muito tempo pois precisamos esperar até que a função v^π convirja. A questão então torna-se se é necessário esperar até essa convergência, ou se é possível parar mais cedo.

De fato, há múltiplas formas que a avaliação de política pode ser reduzida sem perder as condições de convergência para a política ótima, e um caso particular importante é quando paramos a avaliação de política após atualizar cada estado uma única vez. Tal algoritmo é chamado de **iteração de valor** (Sutton e Barto, 2015), e usando-o é possível

combinar os passos de avaliação e aprimoramento de política usando a seguinte regra para gerar a próxima função valor:

$$\begin{aligned} v_{k+1}(s) &:= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v_k(s')), \end{aligned} \tag{2.11}$$

para todo estado $s \in \mathcal{S}$. Para uma função valor arbitrária v_0, pode ser provado que a sequência $\{v_k\}$ converge para v^*sob as mesmas condições que garantem a existência de v^*.

Uma forma intuitiva de pensar sobre iteração de valor é comparando a regra (2.11) com a equação de Bellman de otimalidade (2.6). Veja que as duas são idênticas, e como a função valor ótima v^* é um ponto fixo da equação de Bellman de otimalidade, no algoritmo de iteração de valor aplicamos a regra (2.11) até que a função valor esteja se convergindo, pois assim ela deve estar próxima de v^*:

Programa 2.2 Iteração de valor.

```
1   Entrada: MDP M = ⟨S, A, R, p⟩, γ ∈ (0, 1], e θ > 0 (um número real positivo pequeno,
        determinando a acurácia da estimação)
2   Inicialização:
3       Escolha V(s) ∈ ℝ, para todo s ∈ S arbitrariamente
4       Faça V(s) = 0 para todo s ∈ S que seja terminal
5   Faça:
6       Δ ← 0
7       Para cada estado s ∈ S faça:
8           v ← V(s)
9           V(S) ← max_a Σ_{s',r} p(s', r|s, a)(r + γV(s'))
10          Δ ← max(Δ, |v − V(s)|)
11  enquanto Δ > θ
12  Devolva uma política π ≈ π* tal que π(s) = argmax_a Σ_{s',r} p(s', r|s, a)(r + γV(s'))
```

No programa 2.2, a linha 1 recebe de entrada um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, e os valores de γ e θ do usuário. As linhas 2 a 4 inicializam uma estimação V de v^* arbitrariamente (garantindo que $V(s) = 0$ para todo $s \in \mathcal{S}$ terminal). As linhas 8 e 9 fazem uma iteração de iteração de valor, que vimos na Definição (2.11). A linha 10 calcula Δ, a maior mudança em V após uma fazer iteração de valor uma vez. Se essa mudança for maior do que Δ, o algoritmo volta para a linha 5. Caso contrário, usamos V como uma aproximação de v^*, e devolvemos π, computado usando a Definição (2.10) com a Equação (2.9).

Capítulo 3

Aprendizado por Reforço: métodos básicos

3.1 Método Monte Carlo

Como discutimos antes, em problemas de planejamento probabilístico é necessário conhecermos completamente a função p, enquanto em problemas reais tal informação normalmente não é conhecida. Nesses casos, precisamos de algum método que interagia diretamente com o ambiente para obter experiências, e assim, podemos usar essas experiências para calcular uma política ótima. Esse é o princípio fundamental do aprendizado por reforço.

O método Monte Carlo (abreviado para MC) (Sutton e Barto, 2015) é uma forma de resolver problemas de aprendizado por reforço usando as médias dos retornos experienciados. Como precisamos saber o valor dos retornos, vamos usar os métodos Monte Carlo apenas com problemas episódicos, assim, atualizaremos as estimativas da função valor e atualizamos a política no final de cada episódio.

3.2 Avaliação de política com o método MC

Como no aprendizado por reforço não temos conhecimento da função p, é mais útil estimar a função valor-ação do que a função valor-estado, pois se soubermos apenas v^*, para determinar qual é a melhor ação em um dado estado seria necessário usar a função p para determinar a ação que tem a melhor combinação de recompensa e estado um passo no futuro, assim como foi feito no capítulo 2. Se ao invés disso estimarmos a função valor-ação q^*, dado um estado s qualquer, a melhor ação a será simplesmente uma ação tal que $q^*(s, a)$ seja máxima.

A questão agora é, dado uma política π, como estimamos a função q^π? Lembre que por definição o valor de $q^\pi(s, a)$ é o valor esperado do retorno quando começamos no estado s e escolhemos a ação a, e depois seguimos a política π. Uma forma óbvia de estimar isso é pegar uma amostra de retornos observados após escolher a ação a quando no estado

s, e simplesmente calcular a média desses retornos. Quanto mais amostras de retornos tivermos, mais próximo a média será do valor esperado. Essa é a ideia fundamental em todos os métodos MC.

Mais detalhadamente, para estimar o valor de $q^\pi(s, a)$, precisamos de um conjunto de episódios que seguem a política π e que passaram pelo estado s e nesse estado escolheram a ação a. Chamaremos cada ocorrência disso como uma **visita** ao par estado-ação (s, a). Note que é possível que um mesmo par (s, a) seja visitado múltiplas vezes em um mesmo episódio, assim, chamaremos a primeira vez que um par estado-ação é visitado em um episódio de **primeira visita** a (s, a). O *método MC primeira-visita* (Sutton e Barto, 2015) estima $q^\pi(s, a)$ com a média dos retornos usando apenas as primeiras visitas a (s, a) em cada episódio, enquanto o *método MC toda-visita* (Sutton e Barto, 2015) usa a média de todas as visitas a (s, a).

Programa 3.1 MC primeira-visita, estimar q^π.

```
1   Entrada: MDP M = ⟨S, A, R, p⟩, γ ∈ (0, 1], e uma política π que será avaliada.
2   Inicialização:
3       Q(s, a) ∈ ℝ arbitrariamente, para todo s ∈ S e a ∈ A(s)
4       Retornos(s, a) ← uma lista vazia, para cada s ∈ S e a ∈ A(()s)
5   Loop:
6       Gere um episódio seguindo π: S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T
7       G ← 0
8       Para cada t = T − 1, T − 2, ..., 0 faça:
9           G ← γG + R_{t+1}
10          Se (S_t, A_t) ∉ {(S_0, A_0), (S_1, A_1), ..., (S_{t-1}, A_{t-1})} faça:
11              Adicione G em Retornos(s, a)
12              Q(S_t, A_t) ← média(Retornos(S_t, A_t))
```

O Programa 3.1 descreve um pseudo código do método MC primeira-visita de avaliação de política. Na linha 1 o programa recebe um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p\rangle$, o valor de γ e a política π do usuário. Na linha 3 inicializamos uma estimação Q de q^π arbitrariamente, e para cada $(s, a) \in \mathcal{S} \times \mathcal{A}$, inicializamos uma lista vazia na linha 4. Assim, para cada episódio gerado pela política π na linha 6, conseguimos computar o valor de G_t na linha 9 para cada t usando a Equação (2.3). A linha 10 verifica se (S_t, A_t) ocorreu pela primeira vez no passo no tempo t, e se esse for o caso, adicionamos os valor de G_t na lista que criamos correspondente ao par estado e ação (S_t, A_t) na linha 4. Assim, podemos atualizar a estimação $Q(S_t, A_t)$ para ser a média dos valores guardados nessa lista na linha 12.

Uma possível complicação é que pode haver diversos pares estado-ação que nunca são visitados em nenhum dos episódios. Por exemplo, se π for uma política determinística, então em um dado estado s, a ação que será escolhida sempre será $\pi(s)$ e nenhuma outra ação possível será explorada. Assim, se (s, a) nunca for visitado então a estimativa de $q^\pi(s, a)$ nunca melhorará, o que é um problema bem grande, pois assim não saberemos se a é uma ação melhor do que $\pi(s)$.

É possível evitar esse problema especificando de modo aleatório para cada episódio

um par estado-ação que será o ponto de partida, e que todo par estado-ação tem uma probabilidade positiva de ser escolhida como o início de um episódio. Isso garante que todos os pares estado-ação serão visitados infinitas vezes dado infinitos episódios. Chamamos a suposição de que usar essa estratégia é possível em um problema de aprendizado por reforço de **começo de exploração** (do inglês *exploring starts*) (Sutton e Barto, 2015).

A suposição de termos começo de exploração é útil, mas na prática não é possível aplicá-la em todos os problemas, em particular se não estivermos usando um simulador e o agente está interagindo com um ambiente real. Por enquanto assumiremos que temos começo de exploração, e na seção 3.3 discutiremos outras alternativas.

3.3 Iteração de política com Monte Carlo

Vamos considerar agora como usar o método Monte Carlo pode ser usado para aproximar políticas ótimas. A ideia é usar uma estratégia similar à iteração de política com DP (seção 2.7):

$$\pi_0 \xrightarrow{\text{Av}} q^{\pi_0} \xrightarrow{\text{Ap}} \pi_1 \xrightarrow{\text{Av}} q^{\pi_1} \xrightarrow{\text{Ap}} \pi_2 \xrightarrow{\text{Av}} \cdots \xrightarrow{\text{Ap}} \pi^* \xrightarrow{\text{Av}} q^*,$$

ou seja, dada uma política inicial aleatória π, fazemos avaliação de política para encontrar a função valor-ação q^π, e depois fazemos aperfeiçoamento de política usando q^π para gerar uma política π' tal que $\pi' \geq \pi$. Assim podemos repetir esse processo até chegarmos em uma política ótima π^*.

Já vimos na seção 3.1 como fazer a estimativa da avaliação de política para estimar a função valor-ação, então precisamos de alguma forma para fazer aprimoramento de política dado a função valor-ação. Notavelmente, como estamos trabalhando com a função valor-ação ao invés da função valor-estado, é mais fácil gerar uma política gulosa, pois dado um estado s, a melhor ação a será uma tal que o valor de $q(s, a)$ seja máxima, ou seja:

$$\pi(s) := \operatorname*{argmax}_a q(s, a), \tag{3.1}$$

então aprimoramento de política pode ser feito construindo cada π_{k+1} como uma política gulosa com respeito a q^{π_k}.

Assumindo que a política π_k é gulosa, para todo $k \in \mathbb{N}$, deve seguir por construção que $q^{\pi_k}(s, \pi_k(s)) \geq v^{\pi_k}(s)$, assim, para todo $s \in \mathcal{S}$:

$$\begin{aligned} q^{\pi_k}(s, \pi_{k+1}(s)) &= q^{\pi_k}(s, \operatorname*{argmax}_a q^{\pi_k}(s, a)) \\ &= \max_a q^{\pi_k}(s, a) \\ &\geq q^{\pi_k}(s, \pi_k(s)) \\ &\geq v^{\pi_k}(s), \end{aligned}$$

então é possível aplicar o *teorema do aperfeiçoamento de política* (que vimos na seção 2.6) para dizer que π_{k+1} sempre será melhor do que π_k, a não ser que π_k já seja ótima, e nesse caso, π_{k+1} também será ótima.

Agora sabemos como fazer avaliação e aperfeiçoamento de política usando Monte Carlo, mas uma questão ainda é quantos episódios são necessários para cada etapa de avaliação de política? Similarmente à iteração de valor em planejamento probabilístico, um caso importante é quando usamos um único episódio novo para estimar a função valor-ação. Dessa forma, o algoritmo para gerar uma política ótima usando Monte Carlo é começar com uma política e função valor-ação arbitrária, e a cada episódio novo gerado, fazer um passo de avaliação de política, seguido de um passo de aprimoramento de política. Segue o pseudo-código desse algoritmo assumindo que temos começo de exploração (Sutton e Barto, 2015):

Programa 3.2 Monte Carlo com começo de exploração, para estimar $\pi \approx \pi^*$.

```
1   Entrada: MDP M = ⟨S, A, R, p⟩, γ ∈ (0, 1].
2   Inicialização:
3       π(s) ∈ A(s) arbitrariamente, para todo s ∈ S
4       Q(s, a) ∈ ℝ arbitrariamente, para todo s ∈ S e a ∈ A(s)
5       Retornos(s, a) ← uma lista vazia, para cada s ∈ S e a ∈ A(s)
6   Loop:
7       Escolha S_0 ∈ S e A_0 ∈ A(S_0) aleatoriamente, tal que todos os pares ocorram com
            probabilidade > 0
8       Gere um episódio seguindo π: S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T
9       G ← 0
10      Para cada t = T − 1, T − 2, ..., 0 faça:
11          G ← γG + R_{t+1}
12          Se (S_t, A_t) ∉ {(S_0, A_0), (S_1, A_1), ..., (S_{t-1}, A_{t-1})} faça:
13              Adicione G em Retornos(s, a)
14              Q(S_t, A_t) ← média(Retornos(S_t, A_t))
15              π(S_t) ← argmax_a Q(S_t, a)
```

No Programa 3.2, na linha 1 o programa recebe um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$ e o valor de γ do usuário. Na linha 3 inicializamos uma política π arbitrária; na linha 4 inicializamos uma estimação de q^π arbitrária; e na linha 5 inicializamos uma lista vazia para cada par estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}(s)$. Na linha 7 escolhemos um par estado e ação $(S_0, A_0) \in \mathcal{S} \times \mathcal{A}(S_0)$ como estado e ação inicial, e geramos o resto do episódio na linha 8. Assim, calculamos o valor de G_t para cada t na linha 11 usando a Equação (2.3). A linha 12 verifica se (S_t, A_t) ocorreu pela primeira vez no passo no tempo t, e se esse for o caso, adicionamos os valor de G_t na lista que criamos correspondente ao par estado e ação (S_t, A_t) na linha 5. Assim, podemos atualizar a estimação $Q(S_t, A_t)$ para ser a média dos valores guardados nessa lista na linha 14, e na linha 15 atualizamos a política π assim como vimos na Definição (3.1).

3.4 Monte Carlo sem começo de exploração

Em muitos problemas não podemos usar a suposição de começo de exploração, então precisamos de outra forma para garantir que todos pares estado-ação sejam visitados. Vimos que usar políticas determinísticas resulta em ações que nunca serão tomadas em um dado estado, então precisamos considerar políticas estocásticas.

Uma estratégia é começar usando políticas **suaves** (Sutton e Barto, 2015), significando que $\pi(a|s) > 0$ para todo $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$, e gradualmente mudando para uma política ótima determinística. Nessa seção, utilizaremos as chamadas políticas ε-gulosas, em que a maioria das vezes é usada a ação gulosa (ou seja, que maximiza a função valor-ação), mas com probabilidade ε, é selecionada uma ação disponível aleatoriamente. Mais detalhadamente, quando em um estado $s \in \mathcal{S}$, toda ação não gulosa tem probabilidade $\frac{\varepsilon}{|\mathcal{A}(s)|}$ de ser selecionada, e a ação gulosa é selecionada com probabilidade $1-\varepsilon+\frac{\varepsilon}{|\mathcal{A}(s)|}$. Políticas ε-gulosas são exemplos de políticas ε-suaves, que são definidas como qualquer política π em que $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ para todo $s \in \mathcal{S}$ e $a \in \mathcal{A}(s)$.

No Programa 3.3 apresentamos um pseudo código do algoritmo Monte Carlo modificado para trabalhar com políticas ε-suaves (Sutton e Barto, 2015):

Programa 3.3 Monte Carlo com políticas ε-suaves, para estimar $\pi \approx \pi^*$.

```
1  Parâmetros: MDP M = ⟨S, A, R, p⟩, γ ∈ (0, 1], e ε > 0 pequeno.
2  Inicialização:
3      π ← uma política ε−suave arbitrária
4      Q(s, a) ∈ ℝ arbitrariamente, para todo s ∈ S e a ∈ A(s)
5      Retornos(s, a) ← uma lista vazia, para cada s ∈ S e a ∈ A(s)
6  Loop:
7      Inicialize um estado inicial S_0
8      Gere um episódio seguindo π: S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T−1}, A_{T−1}, R_T
9      G ← 0
10     Para cada t = T − 1, T − 2, ..., 0 faça:
11         G ← γG + R_{t+1}
12         Se (S_t, A_t) ∉ {(S_0, A_0), (S_1, A_1), ..., (S_{t−1}, A_{t−1})} faça:
13             Adicione G em Retornos(s, a)
14             Q(S_t, A_t) ← média(Retornos(S_t, A_t))
15             A' ← argmax_a Q(S_t, a)
16             Para cada a ∈ A(A_t) faça:
17                 Se a = A' faça:
18                     π(a|S_t) ← 1 − ε + ε/|A(S_t)|
19                 Caso contrário:
20                     π(a|S_t) ← ε/|A(S_t)|
```

No Programa 3.3, na linha 1 o programa recebe um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$ e os valores de γ e ε do usuário. Na linha 3 inicializamos uma política ε-suave arbitrariamente; na linha 4 inicializamos uma estimação de q^π arbitrária; e na linha 5 inicializamos uma lista vazia para cada par estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}(s)$. Na linha 7 inicializamos um estado inicial

S_0, e geramos o resto do episódio na linha 8. Assim, calculamos o valor de G_t para cada t na linha 11 usando a Equação (2.3). A linha 12 verifica se (S_t, A_t) ocorreu pela primeira vez no passo no tempo t, e se esse for o caso, adicionamos os valor de G_t na lista que criamos correspondente ao par estado e ação (S_t, A_t) na linha 5. Assim, podemos atualizar a estimação $Q(S_t, A_t)$ para ser a média dos valores guardados nessa lista na linha 14. As linhas 15 a 20 são responsáveis em atualizar a política π de tal forma que ela continue sendo gulosa em relação a Q e ε-suave.

3.5 Método Temporal-Difference

O método Temporal-Difference (abreviado para TD) (Sutton e Barto, 2015) pode ser pensado como uma combinação dos métodos Monte Carlo com o método Dynamic Programming. Assim como DP, o método TD atualiza as estimativas da função valor baseado nas estimativas em outros estados, e não é necessário esperar até obtermos o retorno. Ao mesmo tempo, assim como MC, o método TD aprende interagindo diretamente com o ambiente, sem a necessidade de conhecer as suas dinâmicas.

Começaremos vendo como usar TD para resolver o problema de *predição*, ou seja, como estimar a função valor q^π para uma dada política π. A forma como resolvemos o problema de *controle* com TD é bem semelhante a como fazemos com DP e com MC. A principal diferença entre esses três métodos é como cada um resolve o problema de predição.

3.6 Predição com TD

Seja π a política que estamos usando, e seja Q a estimativa de q^π atual. Assuma que em um episódio, para todo passo no tempo t, já sabemos o valor do retorno G_t. Então, uma forma de medir o quão errado a estimativa Q está é analisando o valor da expressão $G_t - Q(S_t, A_t)$ para todo passo no tempo t. Se a estimativa Q for próxima de q^π, então pela definição de função valor, é esperado que $Q(S_t, A_t)$ e G_t sejam valores próximos, ou seja, que $G_t - Q(S_t, A_t)$ seja perto de zero. Segue disso que se $G_t - Q(S_t, A_t)$ não for perto de zero, então a estimativa atual Q não é uma boa estimativa de q^π, e mais detalhadamente, que $Q(S_t, A_t)$ deveria ser uma valor mais perto de G_t. Assim, uma ideia de como atualizar Q para que ela fique mais perto de q^π é usando a regra (Sutton e Barto, 2015):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t)),$$

para todo passo no tempo t, em que $\alpha \in \mathbb{R}$ é uma constante tal que $0 < \alpha \leq 1$, representando o quão próximo de G_t queremos atualizar o valor de $Q(S_t, A_t)$.

O maior problema com a estratégia acima é o mesmo problema com o método Monte Carlo, e é o fato de que é necessário saber o valor do retorno G_t. Obviamente podemos fazer as mesmas suposições que o método MC, usando a estratégia apenas com problemas episódicos, e nesse caso a estratégia acima é um método chamado de **MC constante-α** (Sutton e Barto, 2015). Porém, o ideal seria remover a necessidade de conhecermos G_t para que seja possível usar a estratégia em problemas não episódicos.

Uma ideia é ao invés de usar G_t, podemos usar uma estimativa de G_t. A forma como vamos estimar G_t será usando o fato de que, por (2.3), sabemos que $G_t = R_t + \gamma G_{t+1}$, assim, quando estivermos no passo no tempo $t+1$, estimaremos o G_t com a expressão $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$. Ou seja, a regra que usaremos será (Sutton e Barto, 2015):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)). \tag{3.2}$$

Essa regra será utilizada para atualizar Q depois de cada passo no tempo. Se S_{t+1} for um estado terminal, definimos $Q(S_{t+1}, A_{t+1})$ como sendo zero. Veja que essa regra sempre usa o conjunto de cinco elementos $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, e é por causa disso que esse método é chamado de método **sarsa** (Sutton e Barto, 2015).

3.7 Algoritmo Sarsa

Agora que sabemos fazer o problema de predição usando o método sarsa, resta apenas fazer o problema de controle. Na realidade, a estratégia que usaremos será idêntica a que usamos no método Monte Carlo, que é possível pois estamos estimando a função valor-ação q^π. Isso é, usaremos políticas ε-gulosas e ε-suaves.

Um fato importante é que por causa de como a regra (3.2) é feita, existe um resultado bem conhecido da teoria da aproximação estocástico que nos garante a convergência a uma política ótima. Em específico, a cada passo no episódio podemos usar um valor diferente de ε, ou seja, em cada passo no tempo t definimos algum $0 \leq \varepsilon_t \leq 1$ tal que nesse mesmo passo a política será ε_t-suave. Dessa forma, desde que:

$$\sum_{t=0}^{\infty} \varepsilon_t = \infty, \text{ e } \sum_{t=0}^{\infty} \varepsilon_t^2 < \infty, \tag{3.3}$$

então com probabilidade 1, a política convergirá a uma política ótima (um exemplo de como fazer isso é definindo $\varepsilon_t = \frac{1}{t+1}$ para todo passo no tempo t)

Segue um algoritmo para conseguir estimar q^* usando o método sarsa (Sutton e Barto, 2015):

Programa 3.4 Algoritmo Sarsa, para estimar $Q \approx q^*$.

```
1   Parâmetros: MDP M = ⟨S, A, R, p⟩, γ ∈ (0, 1], e α ∈ (0, 1], e ε > 0 pequeno.
2   Inicialização:
3       Q(s, a) ∈ ℝ arbitrariamente, para todo s ∈ S e a ∈ A(s)
4       Faça Q(s, ·) = 0, para todo s terminal.
5   Para cada episódio, faça:
6       Inicialize um estado inicial S
7       Escolha ação A a partir de S usando a política ε−suave gerado com Q
8       Para cada passo no tempo, faça:
9           Faça a ação A, e observe a recompensa R e o próximo estado S′
10          Escolha ação A′ a partir de S′ usando a política ε−suave gerado com Q
11          Q(S, A) ← Q(S, A) + α(R + γQ(S′, A′) − Q(S, A))
12          S ← S′
13          A ← A′
14          Atualize ε seguindo as condições (3.3)
15      até que S seja terminal
```

No Programa 3.4, na linha 1 recebemos um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, e os valores de γ, α e ε do usuário. Na linha 3 e 4 inicializamos uma estimação Q de q^*, garantindo que $Q(s, a) = 0$ para todo $s \in \mathcal{S}$ terminal. Assim, para cada episódio, inicializamos um estado inicial S na linha 6, e geramos uma ação A a partir do estado S escolhida pela política gulosa ε-suave gerada com Q na linha 7. Assim, para cada passo no tempo do episódio, executamos a ação A no ambiente e observamos a recompensa R e o próximo estado S' na linha 9, e escolhemos uma ação A' a partir do estado S' escolhida pela política gulosa ε-suave gerada com Q na linha 10. Com isso, podemos atualizar Q na linha 11 usando a Regra de Atualização (3.2) que vimos na seção 3.6. As linhas 12 e 13 preparam S e A para o próximo passo no tempo do episódio, e a linha 14 atualiza ε seguindo as Condições (3.3) que vimos na seção 3.7. A linha 15 apenas verifica se o episódio terminou.

3.8 Algoritmo Q-Learning

A ideia do algoritmo Sarsa é começar com uma estimativa Q inicial, usar Q para criar uma política gulosa π (assim como vimos em (3.1)), e usar a regra (3.2) para que a estimativa Q aproxime de q^π. Porém, o objetivo final é descobrir qual é a política ótima π^*, assim, a ideia principal de **Q-Learning** (Sutton e Barto, 2015) é fazer com que a estimativa Q aproxime-se diretamente a q^*, ao invés de aproximar a q^π. Isso pode ser feito modificando a Regra de Atualização (3.2), para que ela seja assim:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)). \tag{3.4}$$

Note que a única diferença entre (3.2) e (3.4) é que estimamos o valor de G_t com $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$, em vez de usar $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$. Intuitivamente, como queremos estimar q^*, e não q^π, não faz sentido usarmos A_{t+1} para estimar G_t, pois a ação A_{t+1} é

específico para a política π. Logo, como π^* é uma política ótima, estimaremos G_t com a ação que uma política ótima escolheria, o que seria uma ação $a \in \mathcal{A}(S_{t+1})$ que maximiza o valor de $q^*(S_{t+1}, a)$ (e lembre que estamos usando Q como uma estimativa de q^*).

Com essa modificação, o algoritmo Q-learning é o seguinte (Sutton e Barto, 2015):

Programa 3.5 Algoritmo Q-learning, para estimar $Q \approx q^*$.

1 *Parâmetros*: *MDP* $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, $\gamma \in (0, 1]$, **e** $\alpha \in (0, 1]$, **e** $\varepsilon > 0$ *pequeno.*
2 *Inicialização*:
3 $Q(s, a) \in \mathbb{R}$ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(s)$
4 *Faça* $Q(s, \cdot) = 0$, **para** *todo s terminal.*
5 **Para** *cada episódio, faça*:
6 *Inicialize um estado inicial S*
7 **Para** *cada passo no tempo, faça*:
8 *Escolha ação A a partir* **de** *S usando a política ε−suave gerado com Q*
9 *Faça a ação A*, **e** *observe a recompensa R* **e** *o próximo estado S′*
10 $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$
11 $S \leftarrow S'$
12 *Atualize ε seguindo as condições* (3.3)
13 *até que S seja terminal*

No Programa 3.4, na linha 1 recebemos um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, e os valores de γ, α e ε do usuário. Na linha 3 e 4 inicializamos uma estimação Q de q^*, garantindo que $Q(s, a) = 0$ para todo $s \in \mathcal{S}$ terminal. Assim, para cada episódio, inicializamos um estado inicial S na linha 6. Para cada passo no tempo do episódio, escolhemos uma ação A a partir do estado S escolhida pela política gulosa ε-suave gerada com Q na linha 8, e executamos a ação A no ambiente e observamos a recompensa R e o próximo estado S' na linha 9. Com isso, podemos atualizar Q na linha 10 usando a Regra de Atualização (3.4) que vimos na seção 3.8. A linha 11 prepara S para o próximo passo no tempo do episódio, e a linha 12 atualiza ε seguindo as Condições (3.3) que vimos na seção 3.7. A linha 13 apenas verifica se o episódio terminou.

Capítulo 4

Sobre aprendizado por reforço relacional

A partir desse ponto, começaremos a falar sobre **aprendizado por reforço relacional** (abreviado para **RRL**, de *relational reinforcement learning*), e a principal referência para esse capítulo e todos a seguir é a tese de doutorado *Relational Reinforcement Learning* (Driessens, 2004).

4.1 Representação do estado e ação

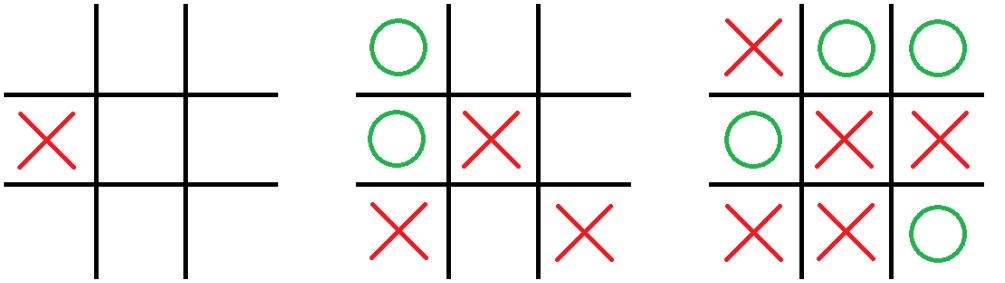
Já discutimos bastante sobre PMD, e um dos elementos importantes de um PMD é o conjunto de estados $\mathcal{S}$ e ações $\mathcal{A}$. Até agora não falamos exatamente como deveríamos definir $\mathcal{S}$ e $\mathcal{A}$ para um dado problema, além de que deve respeitar a propriedade de Markov.

Vamos ver a seguir dois métodos de representar os elementos de $\mathcal{S}$ e de $\mathcal{A}$:

4.1.1 Representação proposicional

O método de representação proposicional (Driessens, 2004) corresponde a representar cada estado como um conjunto de propriedades, com cada propriedade sendo atribuída algum valor.

Por exemplo, se estivermos jogando *jogo da velha*, podemos representar um estado desse jogo da seguinte forma: existem 9 espaços que podem estar ou vazio, ou com um "X", ou com um "O", assim, um estado desse jogo pode ser representado por um vetor de tamanho 9, tal que cada elemento desse vetor é alguém no conjunto $\{v, X, O\}$ (em que v significa que o espaço está vazio). Assim, o vetor pode mostrar o que tem em cada espaço se vermos da esquerda para direita, e de cima para baixo.



(a) *[v, v, v, X, v, v, v, v, v].* **(b)** *[O, v, v, O, X, v, X, v, X].* **(c)** *[X, O, O, O, X, X, X, X, O].*

Figura 4.1: *Exemplo de estados de jogo da velha, e os vetores correspondentes.*

Dessa forma, é possível ver que precisaremos lidar com até 3^9 estados se quisermos usar aprendizado por reforço no jogo da velha (mas na prática são menos estados, pois há múltiplos vetores que representam estados impossíveis de serem encontrados em um jogo normal).

Sobre as ações em $\mathcal{A}$, o único fator importante é que tenha um elemento em $\mathcal{A}$ para cada ação que um agente pode fazer. Por exemplo, no jogo da velha, para um dos jogadores as ações podem ser representadas como $\mathcal{A} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, uma ação para cada espaço no tabuleiro.

Alguns problemas da representação proposicional é o fato de que é difícil representar relações entre múltiplos objetos de um jogo. Ou seja, é difícil de capturar regularidades entre diferentes elementos do vetor, quando vemos estados diferentes. Por exemplo, dado um tabuleiro do jogo da velha em um certo estado, e considere um segundo estado dado pela rotação de 180° desse tabuleiro. Esses dois estados apresentariam múltiplas similaridades, mas a representação proposicional não enxergaria essas semelhanças, e consideraria os dois estados completamente diferentes.

4.1.2 Representação relacional

No método de representação relacional (Driessens, 2004), cada par estado e ação são descritos por um conjunto de *fatos relacionais*.

Um fato relacional tem três partes importantes:

- Um *funtor*, que normalmente descreve o tipo de relação que o fato descreve;
- Uma *aridade*, que é um número inteiro que diz quantos objetos participam dessa relação; e
- *parâmetros*, sendo os objetos que participam dessa relação.

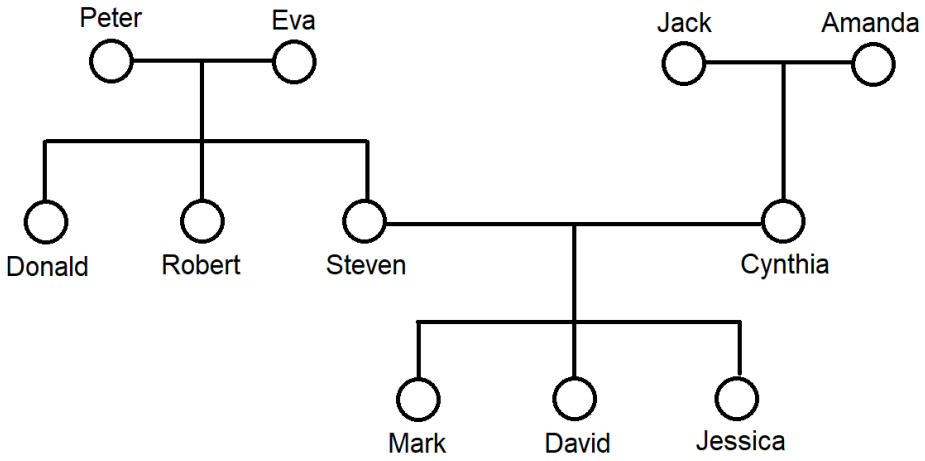


Figura 4.2: *Um exemplo de uma árvore genealógica.*

Por exemplo, na Figura 4.2 temos uma árvore genealógica. Alguns fatos relacionais dessa árvore seriam:

- *male(peter)*, indicando que Peter é um homem. Nesse caso temos que o funtor é "male", a aridade é 1, e o parâmetro é "peter".
- *parent(cynthia, david)*, indicando que Cynthia é um(a) pai/mãe de David. Nesse caso o funtor é "parent", a aridade é 2, e os parâmetros são "cynthia" e "david".
- *grandma(amanda, jessica)*, indicando que Amando é uma avó de Jessica. Nesse caso, o funtor é "grandma", a aridade é 2, e os parâmetros são "amanda" e "jessica".

Note que como o número de pessoas em uma árvore genealógica arbitrária não é conhecido, seria difícil descrever árvores genealógicas com um vetor de tamanho fixo usando a representação proposicional. Assim, a representação relacional tem a vantagem de poder usar uma quantidade arbitrária de fatos relacionais.

4.2 Q-Learning Relacional

Q-Learing relacional (Driessens, 2004) é bem similar ao Q-Learning regular (Seção 4.3), com uma das principais diferenças sendo que o primeiro usa a representação relacional dos estados e ação, enquanto o segundo usa a representação proposicional.

Note que em Q-Learning regular precisamos de uma estimativa de q^*, ou seja, para cada par estado ação $(s, a) \in \mathcal{S} \times \mathcal{A}$ precisamos guardar uma estimativa do valor de $q^*(s, a)$. Por causa disso, a representação proposicional é ideal para Q-Learning regular, pois os estados são representados como vetores finitos, em que cada elemento também tem uma quantidade finita de valores possíveis. Assim, é fácil organizar os dados no algoritmo Q-Learning.

O principal obstáculo de usar a representação relacional dos estados e ação é o fato de que a quantidade de fatos relacionais em cada estado é variante, e também que o tamanho de $\mathcal{S}$ cresce bem mais rápido quando precisamos lidar com uma quantidade grande de objetos e suas relações uma com as outras. Isso significa que será necessário de algoritmos

feitos para lidarem com a representação relacional dos estados e ações, e esse é o tópico que discutiremos nas seções 6 e 7.

De modo geral, todo algoritmo de da classe Q-Learning Relacional tem o seguinte formato:

Programa 4.1 Formato de algoritmos da classe Q-Learning Relacional.

```
1   Entrada: MDP M = ⟨S, A, R, p⟩
2            γ ∈ (0, 1]
3            α ∈ (0, 1]
4            ε > 0 pequeno
5
6   Inicialize uma estimação Q̂ de q*
7   Para cada episódio, faça:
8       Inicialize um estado inicial S
9       Para cada passo no tempo, faça:
10          Escolha ação A a partir de S usando a política ε−suave gerado com Q̂
11          Faça a ação A e observe a recompensa R e o próximo estado S′
12          q ← Q̂(S, A)
13          Q′ ← {Q̂(S′, a) | a ∈ A(S′)}
14          q′ ← q + α(R + γ max Q′ − q)
15          Atualize a estimação Q̂ usando q′
16      Até que S seja terminal
```

No Programa 4.1, nas linhas 1 a 4 o programa recebe um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$ e os valores de γ, α e ε do usuário. Na linha 6 inicializamos $\hat{Q}$, que é alguma forma de estimarmos q^*. Assim, para cada episódio de treinamento inicializamos um estado inicial na linha 8, e para cada passo no episódio, escolhemos uma ação A usando a política ε-suave gerado com $\hat{Q}$ (assim como vimos na seção 3.3), executamos a ação no ambiente e observamos a recompensa R e o próximo estado S' nas linhas 10 e 11. Nas linhas 12 a 14 calculamos a estimação de $q^*(S, A)$ usando a Regra de Atualização (3.4) que vimos na seção 3.8.

Assim, para definir um algoritmo da classe Q-Learning Relacional, precisamos resolver três problemas, correspondentes a três partes do Programa 4.1 que não estão definidas:

1. Como inicializar $\hat{Q}$, o que ocorre na linha 6 do algoritmo.
2. Como fazer a estimação $\hat{Q}(s, a)$ para algum par estado e ação (s, a), o que ocorre nas linhas 10, 12 e 13 do algoritmo (esse é o problema de *predição*).
3. Como melhorar a estimação $\hat{Q}$, o que ocorre na linha 15 do algoritmo (esse é o problema de *controle*).

4.3 Domínio do Mundo dos Blocos

Usaremos os problemas no domínio do **Mundo dos Blocos** para analisar os algoritmos relacionais nas próximas seções.

Neste problema, temos uma quantidade constante de blocos, e um chão grande o suficiente para todos os blocos estarem em cima. Cada bloco pode ou estar diretamente acima do chão, ou pode estar empilhado acima de outro bloco. Não consideraremos estados em que um bloco está empilhado acima de dois ou mais blocos.

Uma ação nesse problema consiste em mover um bloco que esteja *livre* (definido como um bloco que não tenha nenhum outro bloco empilhado em cima dele), e movê-lo para o chão, o em cima de outro bloco livre.

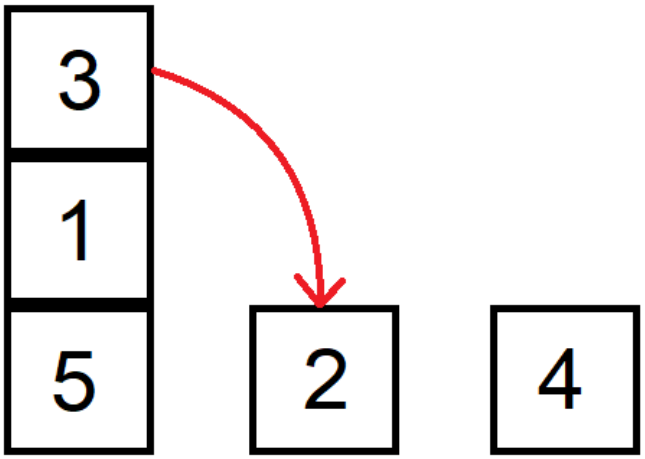


Figura 4.3: *Exemplo de estado e ação no Mundo dos Blocos.*

Na Figura 4.3, temos um exemplo de estado no domínio do Mundo dos Blocos, e a ação que queremos fazer é mover o bloco 3 para acima do bloco 2.

4.3.1 Representação relacional e proposicional

Usando representação relacional, o estado e ação dessa figura pode ser descrita com os seguintes fatos relacionais (Driessens, 2004):

- *on(1, 5)*;
- *on(2, floor)*;
- *on(3, 1)*;
- *on(4, floor)*;
- *on(5, floor)*;
- *clear(2)*;
- *clear(3)*;
- *clear(4)*;
- *move(3, 2)*,

em que:

- *on(X, Y)* significa que o bloco X está acima de Y. Note que Y pode ser um bloco ou o chão.
- *clear(X)* significa que o bloco X é um bloco livre.
- *move(X, Y)* significa que a ação que queremos fazer é mover o bloco X para acima de Y. Note que Y pode ser um bloco ou o chão.

Agora, se usarmos a representação proposicional, como o número de blocos em um problema do domínio do Mundo dos Blocos é constante, digamos n blocos, podemos usar uma matriz $n \times n$ para representar o estado (e a matriz pode ser representada com um vetor de tamanho n^2). Por exemplo, o estado na Figura 4.3 pode ser representado pela matriz:

$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ 3 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 \\ 5 & 2 & 4 & -1 & -1 \end{pmatrix},$$

em que:

- Os elementos da matriz com valores de 1 a 5 representam espaços que os blocos ocupam.
- Os elementos da matriz com valor -1 representam espaços que não tem nenhum bloco.
- Se um bloco X estiver diretamente acima do chão, então na matriz esse bloco estará na última linha.
- Se um bloco X estiver empilhado diretamente acima do bloco Y, então na matriz, X e Y estarão na mesma coluna, com X uma linha acima de Y.

Note que existem múltiplas matrizes que conseguem representar o mesmo estado. Por exemplo, o estado da Figura 4.3 também pode ser representado pelas matrizes:

$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & 3 \\ -1 & -1 & -1 & -1 & 1 \\ -1 & -1 & 4 & 2 & 5 \end{pmatrix} \text{ e } \begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 & -1 \\ 2 & 5 & -1 & 4 & -1 \end{pmatrix}.$$

O algoritmo Q-Learning regular ainda funciona com essa redundância, mas causará o algoritmo a convergir mais lentamente.

As ações na representação proposicional podem ser representados por pares ordenados do formato (X, Y), em que X é o bloco que queremos mover, e Y é o local onde queremos mover X. No exemplo na Figura 4.3, a ação seria $(3, 2)$.

4.3.2 Objetivos e recompensas

Sobre o objetivo de um problema do domínio do Mundo dos Blocos, vamos ver três objetivos diferentes (Driessens, 2004):

1. O objetivo *empilhe todos os blocos* é quando queremos mover os blocos até que todos os blocos estejam empilhados em uma única pilha.

2. O objetivo *desempilhe todos os blocos* é quando queremos mover os blocos até que nenhum bloco esteja empilhado acima de outro bloco, ou seja, todos os blocos devem estar diretamente acima do chão.

3. O objetivo *empilhe dois blocos específicos* é quando especificamos dois blocos X e Y, e queremos mover os blocos até que o bloco X esteja empilhado diretamente acima de Y. Note que se usarmos esse objetivo, o estado precisa de alguma forma incluir qual bloco é X e qual é Y. Na representação relacional, usaremos o fato relacional *goal(X, Y)*, e na representação proposicional basta estender o vetor para que seja de tamanho $n^2 + 2$ (em que n é o número de blocos no problema), e usar esses dois elementos extras para guardar quais blocos X e Y são.

Vamos definir as recompensas bem simplesmente: se a ação causou o estado a mudar a um estado em que o objetivo escolhido está cumprido, damos recompensa igual a 1.0, caso contrário, damos uma recompensa de -0.1 (estamos dando uma recompensa negativa pra incentivar o agente a resolver o problema de uma forma rápida).

Para evitar episódios muito longos, em que agente não consegue chegar no objetivo, também vamos limitar a quantidade de ações em um episódio para 100 ações. Se o problema não for resolvido depois de 100 ações, o episódio terminará, e começaremos um episódio novo.

4.3.3 Inicialização do estado inicial

Dado um dos três objetivos que descrevemos na seção 4.3.2, o estado inicial de um problema do domínio do Mundo dos Blocos com n blocos é escolhido aleatoriamente da seguinte forma:

1. Primeiro, entre todas as formas de empilhar n blocos iguais em que a ordem das pilhas não importa, escolhemos uma com probabilidade uniforme. Por exemplo, se tivermos 6 blocos existem 11 possíveis escolhas, mostradas na Figura 4.4.

2. Depois, rotulamos os blocos com os números de 1 a n, sem repetir rótulos em blocos diferentes. Por exemplo, se tivermos 6 blocos então existem $6! = 720$ formas de rotular os blocos.

3. Finalmente, se o estado inicial que criamos depois dos dois primeiros passos já estiver com o objetivo cumprido, descartamos o estado e repetimos esse processo até chegarmos em um estado inicial com o objetivo não cumprido.

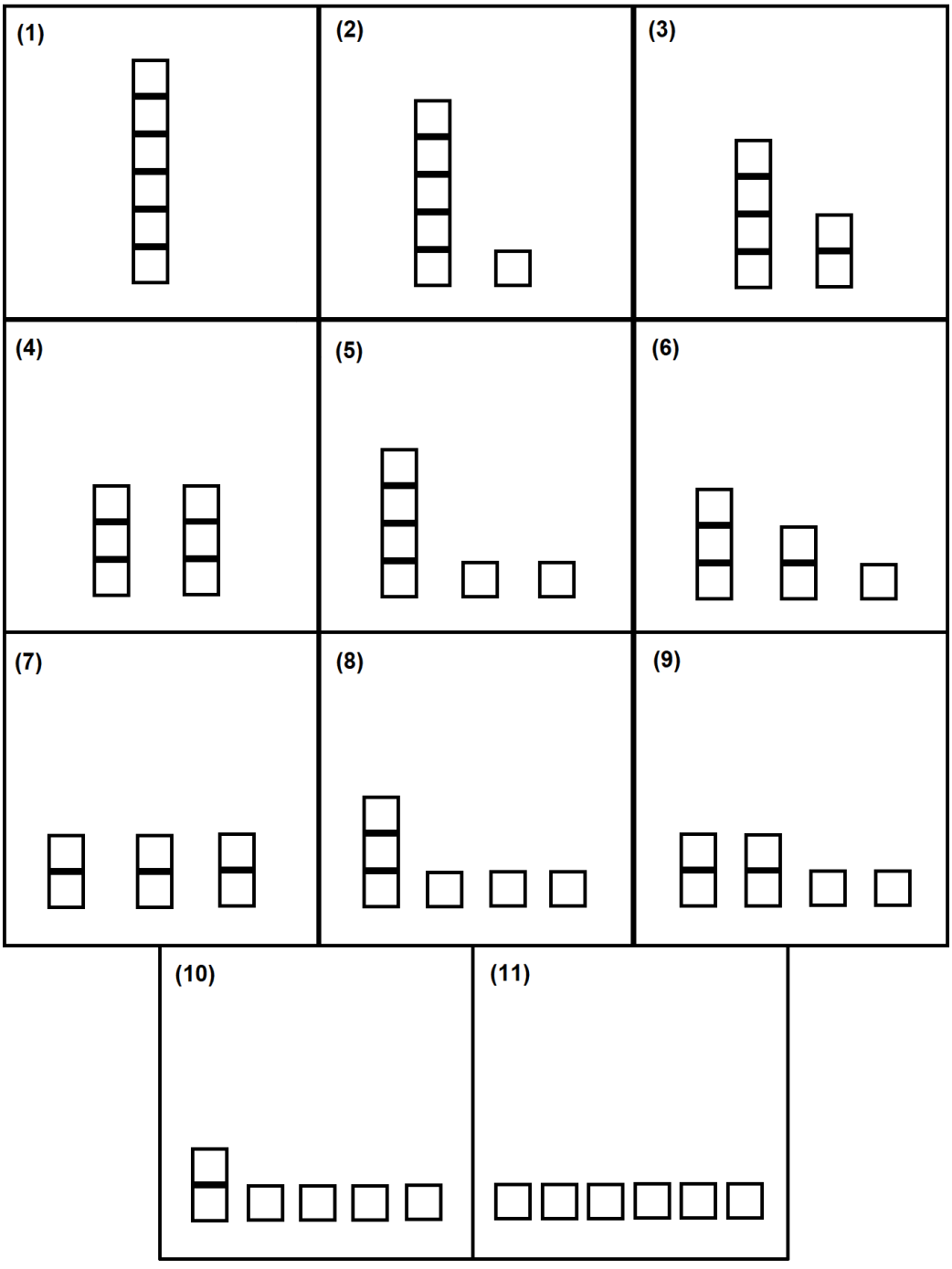


Figura 4.4: *Todas as* 11 *formas de empilhar* 6 *blocos iguais em que ordem das pilhas não importam.*

4.4 Q-Learning regular no Mundo dos Blocos

Para termos um ponto de referência e comparar aprendizado por reforço relacional com não relacional, vamos primeiro resolver os três objetivos no domínio do Mundo dos Blocos com Q-Learnign regular.

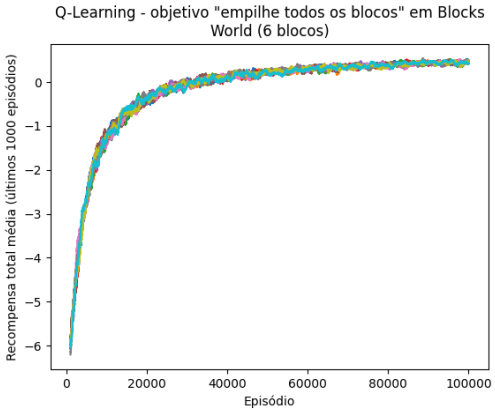
Todos os experimentos nessa seção rodaram o Programa 3.5 com parâmetros $\gamma = 0.95$, e $\alpha = 1$, e $\varepsilon = 0.1$, por um total de 100000 episódios. Para conseguir uma análise estatística, vamos repetir cada experimento (ou seja, repetir cada objetivo) 10 vezes.

Nos gráficos de recompensa total por episódio abaixo, o eixo X começará no episódio 1000 em vez do episódio 1. Isso é porque, para facilitar a leitura do gráfico, o ponto no eixo Y para cada episódio será a média das recompensas totais obtidas no episódio no eixo X junto com os 999 episódios anteriores.

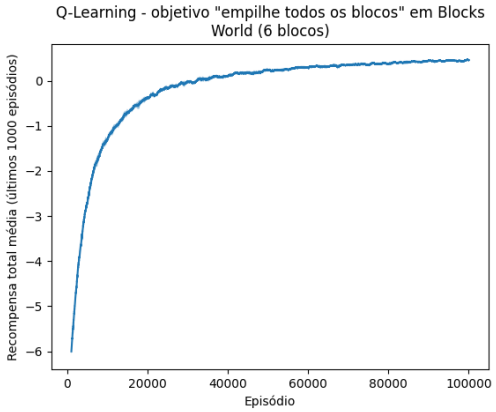
Também mostraremos gráficos demonstrando quanto tempo o algoritmo demorou até terminar a execução do episódio no eixo X. Ou seja, para cada episódio e, o ponto no eixo Y marcado será a quantidade de tempo em segundos que demorou para o algoritmo executar os espisódios 1 até o episódio e.

4.4.1 Objetivo empilhe todos os blocos

Usando o objetivo *empilhe todos os blocos* com 6 blocos, o algoritmo Q-Learning gerou os resultados mostrados nas Figuras 4.5a e 4.5b.



(a) *Gráfico das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 4.5a.*

Figura 4.5: *Objetivo empilhe todos os blocos, gráficos de recompensa com Q-Learning.*

Cada uma das 10 repetições que fizemos desse experimento geraram resultados bem semelhantes, então pode ser difícil ver, mas no gráfico 4.5a tem 10 linhas, e cada cor representa o resultado de uma repetição diferente.

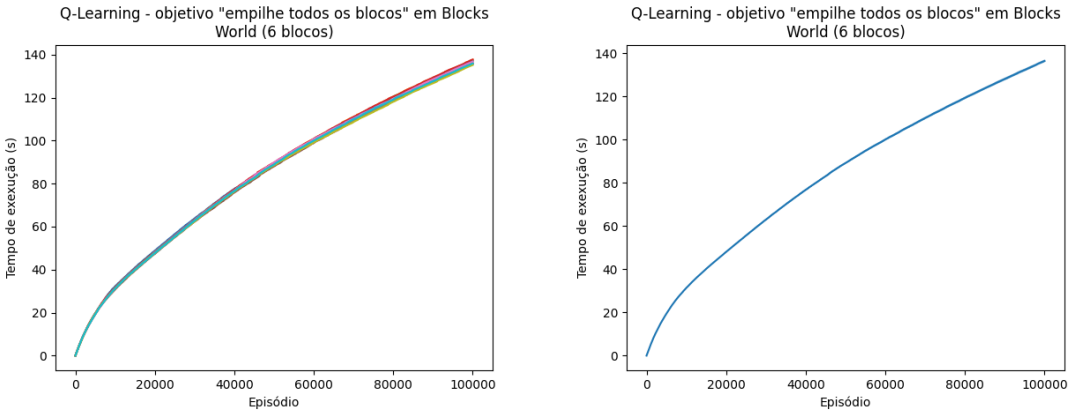
O gráfico 4.5b tem uma linha de cor azul escuro, representando a média das 10 repetições mostradas no gráfico 4.5a. Como cada repetição teve uma trajetória semelhante pode ser difícil de perceber, mas o gráfico 4.5b também mostra o intervalo de confiança de 95% em cada um dos episódios, representado em cor azul claro.

Nesta monografia, todos os gráfico que mostraremos para ver os resultados dos experimentos terão esse mesmo formato: o primeiro gráfico mostrara o resultado para cada repetição do experimento, e o segundo gráfico mostra a média e o intervalo de confiança de 95% das repetições no primeiro gráfico.

Por causa de como definimos as recompensas no Mundo dos Blocos na seção 4.3, a recompensa máxima de um episódio é 1.0, e se não for possível resolver o objetivo em um único passo, cada passo necessário custa −0.1 de recompensa. Por causa disso, quando estamos lidando com 6 blocos, podemos dizer que um agente está com um desempenho bom se a recompensa total de um episódio for próxima de 0.0.

Vendo o gráfico 4.5b, após100000 episódios de experiência, a média de recompensa por episódio é cerca de 0.46, o que pode indicar que, em média, o agente está precisando de 6 ou 7 ações para resolver o objetivo. Também sabemos que Q-Learning no objetivo *empilhe todos os blocos* é bem consistence, pois a variância das repetições é baixa, como visto no gráfico 4.5b.

Além de analisar o desempenho dos agentes treinados com os algoritmos, também vamos analisar a quantidade de tempo real que precisamos deixar o algoritmo rodando. Esses resultados são mostrados nas Figuras 4.6a e 4.6b.



(a) *Gráficos das* 10 *repetições do experimento.* **(b)** *Média e intervalo de confiança do gráfico 4.6a.*

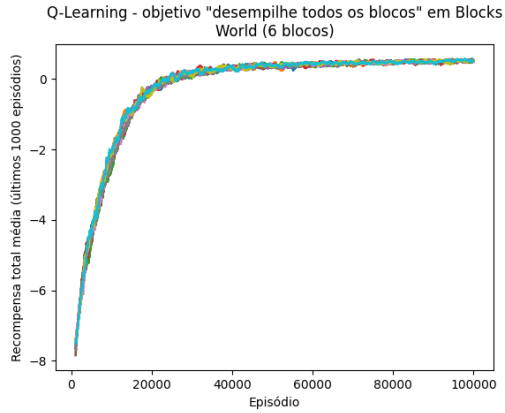
Figura 4.6: *Objetivo empilhe todos os blocos, gráficos de tempo com Q-Learning.*

É possível ver novamente a consistência do algoritmo Q-Learning no objetivo *empilhe todos os blocos*, com o intervalo de confiança no gráfico 4.6b tão pequeno que mal pode ser visto. Segundo o gráfico, pode ser visto que o tempo de execução para treinar o agente por todos os 100000 espisódios é cerca de 2 minutos e 16 segundos.

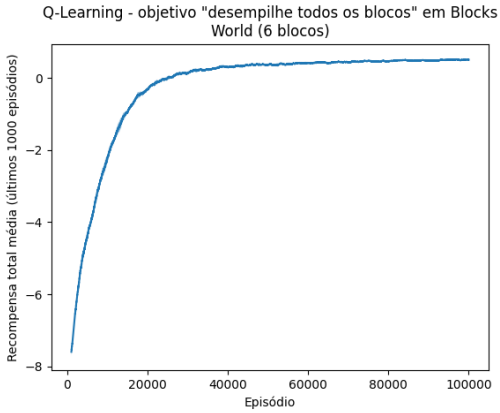
Sobre os gráficos nas Figuras 4.5a e 4.6a, ambos mostram estatísticas de cada uma das 10 repetições do experimento, e linhas da mesma cor entre os dois gráficos representam uma mesma execução do experimento. Nesse caso não conseguimos extrair mais informações sabendo isso, por causa da baixa variância, mas vamos usar esse fato para experimentos posteriores.

4.4.2 Objetivo desempilhe todos os blocos

Usando o objetivo *desempilhe todos os blocos* com 6 blocos, o algoritmo Q-Learning gerou os resultados mostrados nas Figuras 4.7a e 4.7b.



(a) *Gráficos das* 10 *repetições do experimento.*



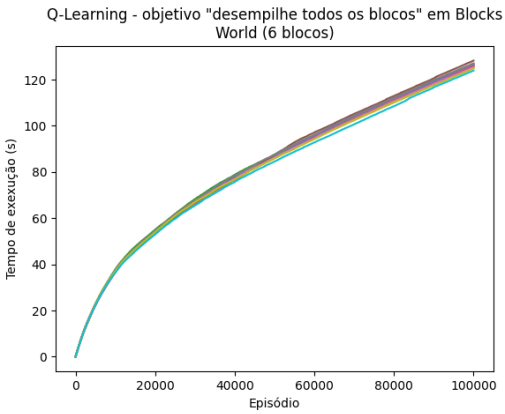
(b) *Média e intervalo de confiança do gráfico 4.7a.*

Figura 4.7: *Objetivo desempilhe todos os blocos, gráficos de recompensa com Q-Learning.*

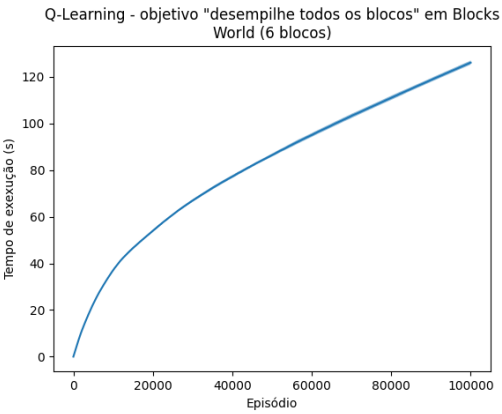
Assim como com o objetivo *empilhe todos os blocos*, pode ser visto nas Figuras 4.7a e 4.7b que Q-Learning é bem consistente quando lidando com o objetivo *desempilhe todos os blocos.*

Além disso, o desempenho do agente depois de 100000 episódios é um pouco melhor com o objetivo *desempilhe todos os blocos* comparado com o objetivo *empilhe todos os blocos.* Após os 100000 episódios, o agente conseguiu uma recompensa total média de cerca de 0.51. Isso indica que cada episódio o agente faz cerca de 6 ações até resolver o objetivo.

Sobre a análise de tempo de execução, os gráficos que mostram essa informação estão nas Figuras 4.8a e 4.8b.



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 4.8a.*

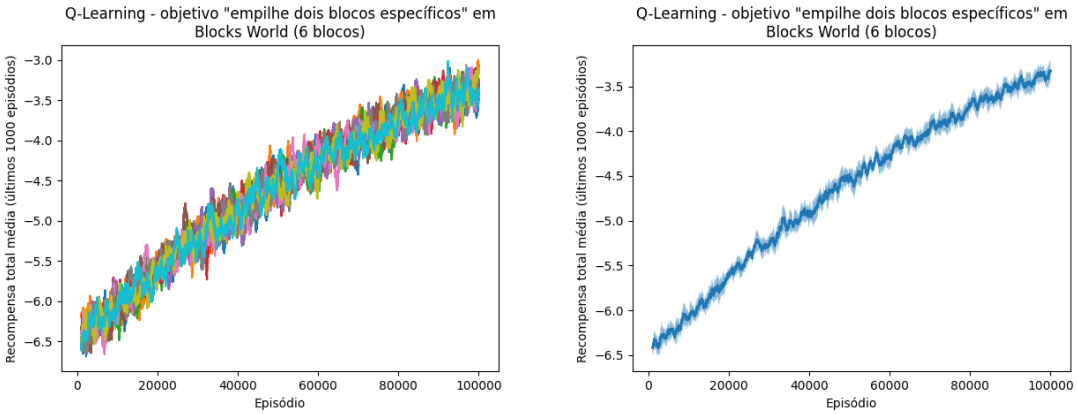
Figura 4.8: *Objetivo desempilhe todos os blocos, gráficos de tempo com Q-Learning.*

É possível ver que o algoritmo Q-Learning executou os 100000 episódios um pouco mais rapidamente com o objetivo *desempilhe todos os blocos* (cerca de 2 minutos e 6 segundos) do que com o objetivo *empilhe todos os blocos.* Porém, considerando o desempenho melhor,

isso não é inesperado, pois estamos usando menos ações por episódio, logo, cada episódio na média dura menos tempo.

4.4.3 Objetivo empilhe dois blocos específicos

Usando o objetivo *empilhe dois blocos específicos* com 6 blocos, o algoritmo Q-Learning gerou os resultados mostrados nas Figuras 4.9a e 4.9b.



(a) *Gráficos das* 10 *repetições do experimento.* **(b)** *Média e intervalo de confiança do gráfico 4.9a.*

Figura 4.9: *Objetivo empilhe dois blocos específicos, gráficos de recompensa com Q-Learning.*

É fácil ver que há diversas diferenças entre o desempenho de um agente treinado usando Q-Learning com o objetivo *empilhe dois blocos específicos* comparado com os outros dois. Primeiro, as repetições do experimento demonstram um nível mais significativo de variância, visível no gráfico da Figura 4.9b. Segundo, o agente não conseguiu aprender muito bem a resolver esse objetivo comparado com os outros dois. Após os 100000 episódios, a recompensa total média é apenas cerca de −3.32.

O desempenho final do agente foi bem pior com o objetivo *empilhe dois blocos específicos* comparado com os outros dois objetivos. De fato, mesmo após os 100000 episódios de treinamento há diversos episódios que o agente não consegue cumprir o objetivo no limite de 100 ações. A Figura 4.10 é um histograma da repetição do experimento que obteve o melhor desempenho final (a média da recompensa total dos últimos 1000 episódios de treinamento dessa repetição foi −3.07). O histograma mostra a quantidade de episódios que o agente conseguiu cumprir o objetivo. Cada barra cobre um intervalo de 1000 episódios, e a altuda da barra corresponde a quantidade desses 1000 episódios que o agente cumpriu o objetivo antes do limite de 100 ações. A última barra do histograma tem uma altura de 767, portanto pode ser visto que o agente ainda não consegue cumprir o objetivo em mais de 20% das configurações iniciais em menos de 100 ações.

Isso tudo sugere que o algoritmo Q-Learning tem uma dificuldade bem maior em resolver o objetivo *empilhe dois blocos específicos* comparado com os outros dois objetivos. Um possível motivo por isso são o que apontamos no final da seção 4.1.1: usando a representação proposicional, não conseguimos capturar as relações entre diferentes elementos dos vetores, o que aumenta significantemente a quantidade de experiência necessária para certos problemas. Nesse caso, a adição de dois blocos específicos no vetor

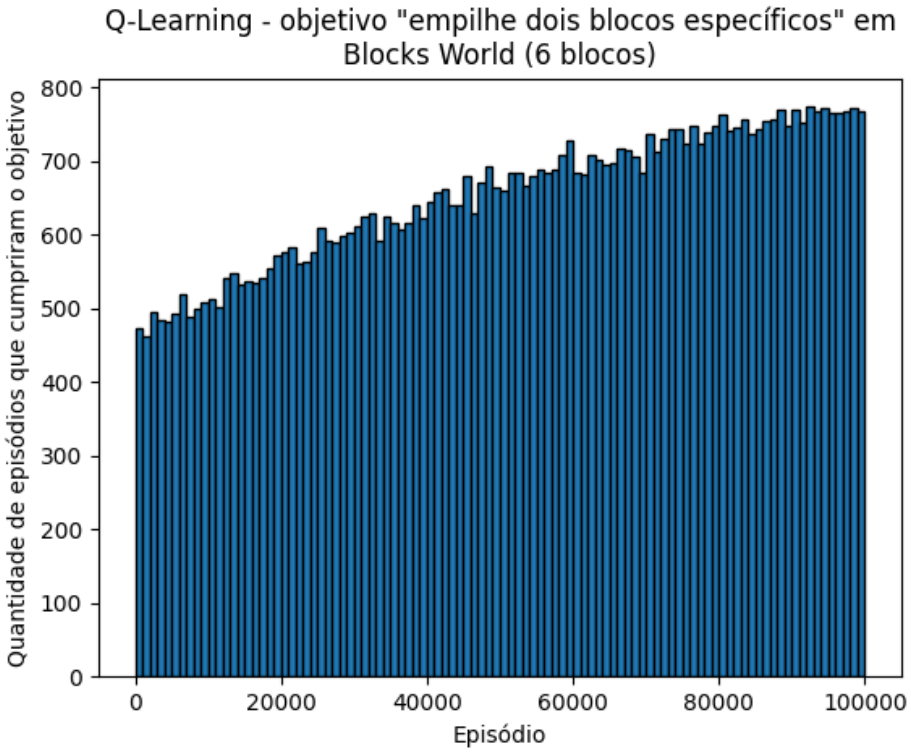


Figura 4.10: *Objetivo empilhe dois blocos específicos, histograma da repetição com melhor resultado final entre os experimentos do objetivo empilhe dois blocos específicos com Q-Learning.*

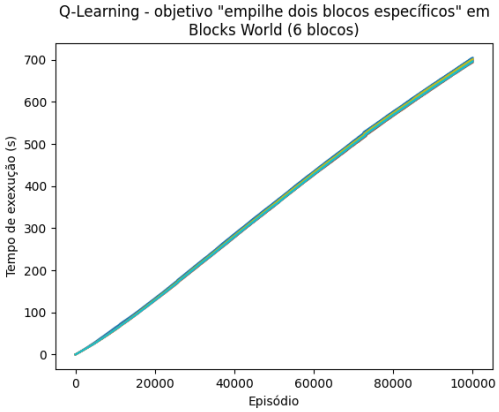
que determinam quais dois blocos o objetivo *empilhe dois blocos específicos* quer empilhar é um possível culpado, pois o algoritmo vai precisar aprender a resolver o problema para cada possível par de blocos que podem ser o objetivo em *empilhe dois blocos específicos*, e o algoritmo tratará cada par como sendo casos completamente distintos, sem nenhuma correlação uma com a outra.

Agora, sobre a análise de tempo de execução, os gráficos que apresentam essa informação estão nas Figuras 4.11a e 4.11b.

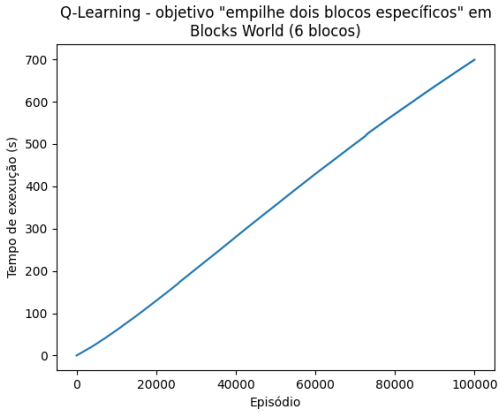
Não é difícil ver que o algoritmo Q-Learning demora mais com esse objetivo comparado com os outros dois, em média precisando de cerca de 11 minutos e 39 segundos de tempo de execução para terminar os 100000 episódios. Mas, novamente, isso não é inesperado, pois o fato de que o desempenho nesse objetivo é bem pior implica que há múltiplos episódios em que o agente precisou executar mais ações comparado com os outros dois objetivos, o que consome mais tempo.

4.4.4 Tabela com resumo dos resultados dos experimentos

Os resultados finais após treinar os agentes por 100000 episódios usando o algoritmo Q-Learning são mostrados nas seguintes tabelas:



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 4.11a.*

Figura 4.11: *Objetivo empilhe dois blocos específicos, gráficos de tempo com Q-Learning.*

Média de recompensa acumulada dos últimos 1000 episódios com Q-Learning			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Média	0.46	0.51	−3.32
Mínimo	0.42	0.48	−3.46
Máximo	0.49	0.55	−3.07
Desvio padrão	0.03	0.02	0.13
Intervalo de confiança de 95%	0.44 a 0.48	0.50 a 0.52	−3.42 a −3.23

Tempo para treinar o agente por 100000 episódios com Q-Learning			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Média	136.41s	126.07s	699.18s
Mínimo	135.16s	123.91s	694.04s
Máximo	137.66s	128.25s	705.08s
Desvio padrão	0.89s	1.22s	3.28s
Intervalo de confiança de 95%	135.78s a 137.05s	125.20s a 126.95s	696.83s a 701.53s

Capítulo 5

O Algoritmo RRL-TG

Na área de aprendizado de máquina, uma técnica utilizada são as árvores de decisões. O primeiro algoritmo que veremos para RRL usa uma variação dessas árvores, chamadas de **árvores de decisões lógicas de primeira ordem** (abreviada para **FOLDT**, do inglês *first-order logical decision tree*) (Blockeel e Raedt, 1998). O algoritmo é chamado **RRL-TG**.

5.1 Árvore de decisão lógica de primeira ordem

Uma FOLDT é uma árvore binária de decisão que recebe um conjunto de fatos relacionais, e retorna algum valor dependendo desse conjunto. No caso de RRL-TG, esse conjunto de fatos relacionais será a representação relacional do par estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}$ do problema, e queremos que a árvore retorne uma estimação do valor de $q^*(s, a)$.

A parte mais importante de uma FOLDT são os conteúdos em seus nós internos e em suas folhas:

- Cada nó interno contém algum fato relacional, possivelmente contendo variáveis nos parâmetros.
- Cada folha contém algum valor de retorno.

Um fato relacional ter uma variável nos parâmetros significa que há múltiplas possibilidades para esse fato. Por exemplo, em *move(2, X)*, temos que X é uma variável (por convenção, uma variável sempre começa com uma letra maiúscula), então *move(2, X)* pode significar *move(2, 5)*, ou *move(2, 2)*, ou *move(2, floor)*, etc.

Além disso, há uma restrição que se uma variável é usada pela primeira vez em um nó interno, essa mesma variável não pode ser usada na subárvore direita desse mesmo nó interno.

Com isso, uma FOLDT determina qual valor retornar usando o seguinte algoritmo (Blockeel e Raedt, 1998):

Programa 5.1 Algoritmo FOLDT.

```
1   Entrada: Uma FOLDT T, e um conjunto de fatos relacionais B
2   node ← raiz(T)
3   fatos ← {}
4   Enquanto node não for uma folha, faça:
5       Se B satisfaz fatos ∪ fato(node) faça:
6           fatos ← fatos ∪ fato(node)
7           node ← esquerda(node)
8       Caso contrário:
9           node ← direita(node)
10  Retone valor(node)
```

em que $raiz(T)$ é o nó raiz da FOLDT T; $fato(node)$ é o fato relacional no nó interno $node$; $esquerda(node)$ e $direita(node)$ são o nó à esquerda e à direita de $node$, respectivamente; e $valor(node)$ é o valor guardado na folha $node$.

Uma parte importante desse algoritmo é entender a condição na linha 5. Note que $fatos$ é um conjunto de fatos relacionais com variáveis de alguns nós internos de T. Assim, dado um conjunto de fatos relacionais sem variáveis B, dizemos que B satisfaz $fatos$ se existe alguma substituição das variáveis em $fatos$, digamos $fatos'$, tal que $B \supseteq fatos'$.

Por exemplo, se $B = \{on(1, floor), on(2, 1), on(3, floor), clear(2), clear(3), move(2, 3)\}$ e $fatos = \{clear(X), move(X, Y)\}$, então B satisfaz $fatos$, pois se fizermos a substituição de X para 2 e de Y para 3, então $fatos$ transforma-se em $fatos' = \{clear(2), move(2, 3)\}$, e temos que $B \supseteq fatos'$.

Assim, no Programa 5.1, na linha 1 o programa recebe uma FOLDT T e um conjunto de fatos relacionais sem variáveis B do usuário. Na linha 2 inicializamos *node* com a raiz da FOLDT T, e na linha 3 inicializamos *fatos* como um conjunto vazio, que será usado para guardar os fatos relacionais de alguns nós internos. Com tudo isso, seguimos o seguinte processo:

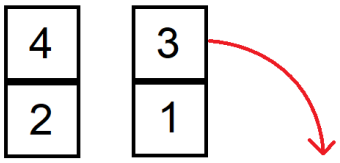
1. Se *node* for uma folha, paramos o algoritmo e devolvemos o valor guardado nessa folha (linhas 4 e 10 do algoritmo).

2. Caso contrário, *node* é um nó interno. Seja f o fato relacional nesse nó interno.

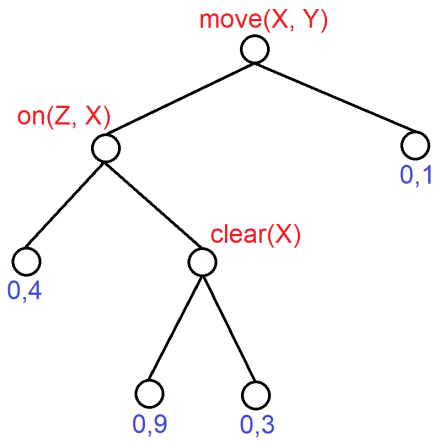
3. Se B satisfazer $fatos \cup f$, então atualizamos *fatos* para que inclua f, e fazemos *node* ser o nó a sua esquerda (linhas 5 a 7 do algoritmo)

4. Se B não satisfazer $fatos \cup f$, então não mudamos *fatos*, e fazemos *node* ser o nó a sua direita (linhas 8 e 9 do algoritmo).

5. Volte para o passo (1) (ou seja, volte para a linha 4 do algoritmo).



(a) *Exemplo estado e ação no Mundo dos Blocos.*



(b) *Exemplo de FOLDT.*

Figura 5.1: *Exemplo de uso de uma FOLDT para o Mundo dos Blocos.*

Um exemplo de como uma FOLDT pode ser usado em um problema do Mundo dos Blocos está mostrado na Figura 5.1. Usando a representação relacional usado na seção 4.3, o estado e ação mostrado na Figura 5.1a seria representado pelo seguinte conjunto de fatos relacionais: $B = \{on(1, floor), on(2, floor), on(3, 1), on(4, 2), clear(3), clear(4), move(3, floor)\}$.

É fácil ver que B satisfaz $\{move(X, Y)\}$, com a única possível substituição sendo X com 3, e Y com $floor$. Note também que B não satisfaz $\{move(X, Y), on(Z, X)\}$, pois não tem nenhum bloco acima de 3. Além disso, B satisfaz $\{move(X, Y), clear(X)\}$, o que pode ser visto fazendo a mesma substituição de X com 3, e Y com $floor$. Portanto, no exemplo da Figura 5.1, se esse par estado e ação for $(s, a) \in \mathcal{S} \times \mathcal{A}$, então o valor estimado de $q^*(s, a)$ seria 0.9.

O algoritmo RRL-TG resolve o problema de *predição* usando uma FOLDT para estimar o valor de $q^*(s, a)$ para um dado par estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}$. Assim, seja $\hat{q}^T(s, a)$ o valor retornado pelo Programa 5.1 com FOLDT T e fatos relacionais (s, a). Se o ambiente estiver no estado s e o agente executar a ação a, resultando em uma recompensa r e um próximo estado s', similarmente a Regra de Atualização (3.4) em Q-Learning, podemos estimar $q^*(s, a)$ da seguinte forma:

$$q^T(s, a) := \hat{q}^T(s, a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}(s')} \hat{q}^T(s', a') - \hat{q}^T(s, a)). \tag{6.1}$$

A Definição (6.1) é o equivalente de q' no Programa 4.1 na linha 14.

5.2 Candidatos para fato relacional em nós internos

Tudo que falta agora é descobrir como construír uma FOLDT que estime q^* bem, e é isso que o algoritmo RRL-TG faz.

O algoritmo começa com uma FOLDT inicial. Pode ser que seja uma árvore com apenas

um nó (ou seja, apenas uma folha), mas é possível começar com uma FOLDT que já tem nós internos com fatos relacionais pré-determinados. Cada folha começa com um valor de retorno 0.

Quando expandirmos a FOLDT inicial, precisamos de alguma forma de determinar qual fato relacional e quais variáveis será usado para um nó interno que é criado. Para fazer isso, antes do algoritmo começar, o usuário precisa especificar quais são os possíveis fatos relacionais que um nó interno pode ter. Isso é feito com declarações chamadas de **rmode** (Dzeroski *et al.*, 2001).

Um rmode tem o seguinte formato: *rmode(N: fato_relacional)*. Tal declaração significa que *fato_relacional* pode ser usado em um nó interno, mas no máximo N vezes em um caminho na FOLDT começando da raiz. Se N for omitido, então não tem limite quantas vezes *fato_relacional* pode ser usado.

Para determinar quais variáveis podem ser usadas, cada variável em *fato_relacional* é atribuído pelo menos um dos seguintes modificadores:

- O modificador "+" indica que a variável usada pode ser uma que já apareceu no caminho do nó interno novo até a raiz.
- O modificador "–" indica que a variável usada pode ser uma variável nova, ou seja, que não aparece no caminho do nó interno novo até a raiz.
- O modificador "#" indica que no lugar da variável pode uma das constantes que é pré-definida pelo usuário.

É possível combinar múltiplos modificadores para uma mesma variável. Por exemplo, se uma variável tiver o modificador "+–", então pode ser usado tanto já usadas quanto variáveis novas.

Para demonstrar como os rmodes agem na prática, considere que definimos apenas dois rmodes:

- *rmode(5: clear(+-X))*;
- *rmode(5: on(+X, -#Y))* (e a única constante que Y pode ser é definida para ser *floor*),

então, vendo a FOLDT na Figura 5.1b, se quisermos expandi-la transformando a folha com valor 0.9 em um nó interno, então o primeiro rmode define os seguintes candidatos para fatos relacionais:

- *clear(X)*;
- *clear(Y)*;
- *clear(W)*,

em que W é uma variável nova. E o segundo rmode define os seguintes candidatos para fatos relacionais:

- *on(X, W)*;
- *on(Y, W)*;

- *on(X, floor)*;
- *on(Y, floor)*.

Note que a variável Z nunca aparece entre os candidatos, pois a folha com valor 0.9 está na subárvore direita da primeira vez que ela aparece (no nó interno com fato relacional *on(Z, X)*).

5.3 Seleção de fato relacional para nó interno

Agora que conseguimos determinar quais são os candidatos para fato relacional para um nó interno novo, o que falta é determinar quando queremos transformar uma folha em um nó interno, e determinar qual fato relacional será escolhido entre os candidatos (Driessens, 2004).

Para fazer isso, cada folha da FOLDT atual guardará informações de diversas estatísticas. Mais especificamente, vamos coletar todos os pares estado e ação (s, a) que o agente percorre dunrate o algoritmo RRL-TG, e para cada candidato para fato relacional r na folha f, vamos guardar:

1. A quantidade de pares estado e ação (s, a) que, se passado como B nos parâmetros do Programa 5.1, chegaria na folha f (chamaremos esse valor de n^f).
2. Entre os pares estado e ação (s, a) que foram contados na estatística *(1)*, a quantidade que também satisfaz r (chamaremos esse valor de $n_p^{f,r}$).
3. Entre os pares estado e ação (s, a) que foram contados na estatística *(1)*, a quantidade que não satisfaz r (chamaremos esse valor de $n_n^{f,r}$).
4. A soma das estimações de $q^*(s, a)$ (calculadas com a Definição (6.1)) para cada uma dos $n_p^{f,r}$ pares estado e ação que contaram para a estatística *(1)* (definiremos $q^f := (q_1^f, q_2^f, \dots, q_{n^f}^f)$ como o vetor dos valores somados para essa estatística).
5. A soma das estimações de $q^*(s, a)$ (calculadas com a Definição (6.1)) para cada uma dos $n_p^{f,r}$ pares estado e ação que contaram para a estatística *(2)* (definiremos $q_p^{f,r} := (q_{p,1}^{f,r}, q_{p,2}^{f,r}, \dots, q_{p,n_p^{f,r}}^{f,r})$ como o vetor dos valores somados para essa estatística).
6. A soma das estimações de $q^*(s, a)$ (calculadas com a Definição (6.1)) para cada uma dos $n_n^{f,r}$ pares estado e ação que contaram para a estatística *(3)* (definiremos $q_n^{f,r} := (q_{n,1}^{f,r}, q_{n,2}^{f,r}, \dots, q_{n,n_n^{f,r}}^{f,r})$ como o vetor dos valores somados para essa estatística).
7. A soma dos quadrados das estimações de $q^*(s, a)$ (calculadas com a Definição (6.1)) para cada um dos n^f pares estado e ação que contaram para a estatística *(1)*.
8. A soma dos quadrados das estimações de $q^*(s, a)$ (calculadas com a Definição (6.1)) para cada um dos $n_p^{f,r}$ pares estado e ação que contaram para a estatística *(2)*.
9. A soma dos quadrados das estimações de $q^*(s, a)$ (calculadas com a Definição (6.1)) para cada um dos $n_n^{f,r}$ pares estado e ação que contaram para a estatística *(3)*.

Uma vantagem das estatísticas acima é o fato de que é fácil de computá-las incrementalmente, ou seja, toda vez que passamos por um par (s, a) novo, é fácil atualizar o valor em tempo constante.

Queremos guardas essas estatísticas em cada folha pois o critério que usaremos para determinar quando queremos expandir uma das folhas da FOLDT é a variância das estimações de q^* que cada folha retorna. Mais detalhadamente, se a divisão de uma folha f resultar em uma variância estatísticamente menor com um candidato para fato relacional r, então vamos transformar f em um nó interno com fato relacional r. Formalmente, para cada folha f na FOLDT e cada candidato r para fato relacional em f, queremos comparar os valores de:

$$\frac{n_p^{f,r}}{n^f}(\sigma_p^{f,r})^2 + \frac{n_n^{f,r}}{n^f}(\sigma_n^{f,r})^2 \text{ vs. } (\sigma_{total}^f)^2, \tag{6.2}$$

em que $\sigma_p^{f,r}$ e $\sigma_n^{f,r}$ são os desvios padrões de $q_p^{f,r}$ e de $q_n^{f,r}$, respectivamente, e σ_{total}^f é o desvio padrão de q^f.

Para determinar quando a diferença na variância é estatísticamente significativa, usaremos o Teste F com nível de significância $\beta \in (0, 0.5]$, um valor pré-definido pelo usuário. De modo geral, quanto maior for β, mais fácil e frequente as divisões das folhas na FOLDT ocorrerão.

Por definição, se $x = (x_1, x_2, \ldots, x_n)$ for um vetor de n números reais, então a média de x é:

$$\overline{x} := \frac{\sum_{i=1}^n x_i}{n},$$

e a variância é definida como:

$$\begin{aligned}
\sigma^2 &:= \frac{\sum_{i=1}^n (x_i - \overline{x})^2}{n} \\
&= \frac{\sum_{i=1}^n (x_i^2 - 2x_i\overline{x} + \overline{x}^2)}{n} \\
&= \frac{\sum_{i=1}^n x_i^2 - 2\overline{x}\sum_{i=1}^n x_i + n\overline{x}^2}{n} \\
&= \frac{\sum_{i=1}^n x_i^2 - 2\overline{x} \cdot n\overline{x} + n\overline{x}^2}{n} \\
&= \frac{\sum_{i=1}^n x_i^2 - n\overline{x}^2}{n}.
\end{aligned}$$

Sabendo isso, podemos transformar a comparação em (6.2) para:

$$\frac{n_p^{f,r}}{n^f}\frac{\sum_{i=1}^{n_p^{f,r}}(q_{p,i}^{f,r})^2 - n_p^{f,r}\overline{q_p^{f,r}}}{n_p^{f,r}} + \frac{n_n^{f,r}}{n^f}\frac{\sum_{i=1}^{n_n^{f,r}}(q_{n,i}^{f,r})^2 - n_n^{f,r}\overline{q_n^{f,r}}}{n_n^{f,r}} \text{ vs. } \frac{\sum_{i=1}^{n^f}(q_i^f)^2 - n^f\overline{q^f}}{n^f}. \tag{6.3}$$

Por causa de como o Teste F é calculado, podemos multiplicar ambos lados da comparação em (6.3) por n^f, e o resultado do Teste F continuará sendo o mesmo. Fazendo isso, o valor da expressão à direita fica:

$$\begin{aligned}\sum_{i=1}^{n^f}(q_i^f)^2 - n^f\overline{q^f} &= \sum_{i=1}^{n^f}(q_i^f)^2 - n^f\left(\frac{\sum_{i=1}^{n^f} q_i^f}{n^f}\right)^2 \\ &= \sum_{i=1}^{n^f}(q_i^f)^2 - \frac{1}{n^f}\left(\sum_{i=1}^{n^f} q_i^f\right)^2,\end{aligned}$$

e, simetricamente, o valor da expressão à esquerda fica:

$$\begin{aligned}&\sum_{i=1}^{n_p^{f,r}}(q_{p,i}^{f,r})^2 - n_p^{f,r}\overline{q_p^{f,r}} + \sum_{i=1}^{n_n^{f,r}}(q_{n,i}^{f,r})^2 - n_n^{f,r}\overline{q_n^{f,r}} \\ &= \sum_{i=1}^{n_p^{f,r}}(q_{p,i}^{f,r})^2 - \frac{1}{n_p^{f,r}}\left(\sum_{i=1}^{n_p^{f,r}} q_{p,i}^{f,r}\right)^2 + \sum_{i=1}^{n_n^{f,r}}(q_{n,i}^{f,r})^2 - \frac{1}{n_n^{f,r}}\left(\sum_{i=1}^{n_n^{f,r}} q_{n,i}^{f,r}\right)^2,\end{aligned}$$

portanto, a comparação em (6.3) fica:

$$\sum_{i=1}^{n_p^{f,r}}(q_{p,i}^{f,r})^2 - \frac{1}{n_p^{f,r}}\left(\sum_{i=1}^{n_p^{f,r}} q_{p,i}^{f,r}\right)^2 + \sum_{i=1}^{n_n^{f,r}}(q_{n,i}^{f,r})^2 - \frac{1}{n_n^{f,r}}\left(\sum_{i=1}^{n_n^{f,r}} q_{n,i}^{f,r}\right)^2 \text{ vs. } \sum_{i=1}^{n^f}(q_i^f)^2 - \frac{1}{n^f}\left(\sum_{i=1}^{n^f} q_i^f\right)^2. \tag{6.4}$$

A parte importante de (6.4) é o fato de que conseguimos computar ambas expressões em tempo constante, pois estamos guardando as estatísticas em cada folha da FOLDT.

Para evitar que uma folha seja dividida com poucos exemplos, também pré-definimos uma quantidade mínimo de exemplos m que uma folha precisa ter para que possamos transformá-la em um nó interno. A quantidade de exemplos de uma folha f é definido como a soma $n_p^{f,r} + n_n^{f,r}$ para algum candidato para fato relacional r (veja que a soma é a mesma para qualquer r).

Uma parte que não discutimos ainda são as estatísticas das folhas novas que aparecem depois de dividir uma folha. As duas folhas novas não precisam começar com estatísticas

zeradas, pois é possível aproveitar partes da estatística da folha original f. Mais especificamente, se e e d forem as folhas novas na esquerda e direita, respectivamente, então:

- As folhas e e d podem herder a estatística *(1)* com a estatística *(2)* e *(3)* de f, respectivamente;
- As folhas e e d podem herder a estatística *(4)* com a estatística *(5)* e *(6)* de f, respectivamente;
- As folhas e e d podem herder a estatística *(7)* com a estatística *(8)* e *(9)* de f, respectivamente.

Porém, as estatísticas *(2), (3), (5), (6), (8)* e *(9)* das duas folhas novas precisam começar do zero.

Além disso, se a adição das informações de um novo par estado e ação não resultar na divisão de nenhuma folha da FOLDT, ainda podemos atualizar os valores das folhas atuais. Como estamos guardando a quantidade de pares estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}$ que chegam em uma folha f (estatística (1)), e a soma das estimações de $q^*(s, a)$ dos pares estado e ação contados na primeira estatística (estatística (4)), podemos usar essas informações para atualizar o valor devolvido pela folha para ser a média dos valores no vetor q^f.

Todo esse processo discutido nessa seção é como o algoritmo RRL-TG resolve o problema de *controle*.

5.4 Algoritmo RRL-TG

Com tudo isso, podemos finalmente apresentar o algoritmo RRL-TG (Driessens, 2004):

Programa 5.2 Algoritmo RRL-TG, para estimar $Q \approx q^*$.

```
1   Entrada: MDP M = ⟨S, A, R, p⟩
2            γ ∈ (0, 1]
3            α ∈ (0, 1]
4            ε > 0 pequeno
5            T, uma FOLDT inicial
6            β, o nível de significância do Teste F para dividir uma folha
7            m, número mínimo de exemplos para uma folha poder ser dividida
8
9   Inicialize os rmodes
10  Para cada episódio, faça:
11      Inicialize um estado inicial S
12      Para cada passo no tempo, faça:
13          Escolha ação A a partir de S usando a política ε−suave gerado com a FOLDT T
14          Faça a ação A e observe a recompensa R e o próximo estado S′
15          q ← FOLDT(T, (S, A))
16          Q′ ← {FOLDT(T, (S′, a)) | a ∈ A(S′)}
17          q′ ← q + α(R + γ max Q′ − q)
18          Seja f a folha que o Programa 5.1 chega com FOLDT T e fatos relacionais
                (S, A)
19          Atualize as estatísticas na folha f com (S, A) e q′
20          Se número de exemplos em f for ≥ m e o Teste F com nível de significância β
                indicar que a folha pode ser dividida:
21              Gere um nó interno com um candidato para fato relacional que melhor
                    sucediu o Teste F
22              Gere duas folhas, com estatísticas herdados das estatísticas em f
23          Caso contrário:
24              Atualize o valor retornado por f em T para \overline{q^f}
25          S ← S′
26      Até que S seja terminal
```

em que $FOLDT(T, B)$ é o que a função no Programa 5.1 retorna com FOLDT T e fatos relacionais B.

Assim, no Programa 5.2, nas linhas 1 a 7 o programa recebe um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$ e os valores de γ, de α, de ε, de β, de m, e uma FOLDT inicial T. Na linha 9 declaramos os *rmodes* definidos pelo usuário assim como discutimos na seção 5.2. Dessa forma, para cada episódio, inicializamos um estado inicial S (seguindo o método que discutimos na seção 4.3.3) na linha 11, e para cada passo no tempo, escolhemos uma ação A seguindo a política gulosa ε-suave gerado com a estimação de q^* (que é feita usando a FOLDT T) na linha 13, e executamos a ação A no ambiente para observar a recompensa R e o próximo estado S' na linha 14. Nas linhas 15 a 17 computamos a estimação de $q^*(S, A)$ usando a regra de atualização do Q-Learning (3.4).

O restante do algoritmo é o problema de *controle* que vimos na seção 5.3. Encontramos a folha f que a função $FOLDT(T, (S, A))$ termina na linha 18, e atualizamos as estatísticas na folha f na linha 19. Se a quantidade de estatísticas em f superar m e o Teste F sucedir

com nível de significância β para algum candidato para fato relacional r, então dividimos a folha f, transformando-a em um nó interno e escolhemos um r que melhor sucediu o Teste F como fato relacional, e fazemos as novas folhas herderem estatísticas de f (linhas 20 a 22 do programa). Se f tiver menos do que m estatísticas ou o Teste F não sucedir com nível de significância β, então atualizamos o valor devolvido pela folha f para ser a média dos valores no vetor q^f, o que pode ser calculado com as estatísticas em f (linhas 23 e 24 do programa). Finalmente, atualizamos S para ser S' para o próximo passo no tempo do episódio na linha 25, e verificamos se o episódio acabou na linha 26.

Dessa forma, o algoritmo RRL-TG é um algoritmo da classe Q-Learning Relacional, e resolve os três problemas que apontamos na seção 4.2 da seguinte forma:

1. A inicialização de $\hat{Q}$ é feita com a FOLDT inicial T que o usuário define.
2. O problema de *predição*, ou seja, a estimação de $\hat{Q}(s, a)$ para cada $(s, a) \in \mathcal{S} \times \mathcal{A}$, é feito com o Programa 5.1, usando T como FOLDT e (s, a) como fatos relacionais.
3. O problema de *controle* é feito guardando estatísticas nas folhas da FOLDT T, e usando o Teste F para determinar se vamos dividir uma folha ou atualizar o valor que ela devolve, assim como vimos na seção 5.3.

5.5 RRL-TG no Mundo dos Blocos

Para cada um dos experimentos a seguir, rodamos o Programa 5.2 com os parâmetros $\gamma = 0.95$, e $\alpha = 1$, e $\varepsilon = 0.1$, e $m = 100$. Cada experimento foi repetido 10 vezes para fazer as análises estatísticas.

Cada experimento rodou o algoritmo RRL-TG até 500 episódios. Similarmente com o que discutimos na seção 4.4, os gráficos de recompensa total por episódio mostram a média da recompensa total obtida no episódio junto com os 49 episódios anteriores, para facilitar a leitura do gráfico.

Vimos na seção 4.3 que na forma como fazemos a representação relacional do problema do Mundo dos Blocos, usamos fatos relacionais do formato *on(X, Y)*, *clear(X)* e *move(X, Y)*. Também, no objetivo *empilhe dois blocos específicos* usamos um fato relacional do formato *goal(X, Y)* para representar o objetivo. Além desses fatos relacionais, nos experimentos do RRL-TG incluiremos fatos relacionais do formato *above(X, Y)*, significando que os blocos X e Y estão na mesma pilha, mas X está em um ponto mais alto da pilha do que Y; e também fatos relacionais do formato *equal(X, Y)*, significando que as variáveis X e Y referenciam o mesmo objeto.

Com isso, os seguintes rmodes foram usadom em todos os experimentos abaixo:

- *rmode(6: clear(+X))*;
- *rmode(6: on(+-X, +-#Y))*;
- *rmode(6: above(+-X, +-Y))*;
- *rmode(6: equal(+X, +#Y))*;
- *rmode(1: move(+-X, +-#Y))*;

em que a única constante é *floor*.

5.5.1 Objetivo empilhe todos os blocos

Para o objetivo *empilhe todos os blocos*, usaremos a FOLDT mostrada na Figura 5.2 como árvore inicial do Programa 5.2. Ou seja, as informações que a FOLDT inicial distingue são quais dois objetos participam da ação *move(X, Y)*, e se a ação envolve mover X para o chão ou para outro bloco. Também, usaremos um nível de significância $\beta = 0.4$.

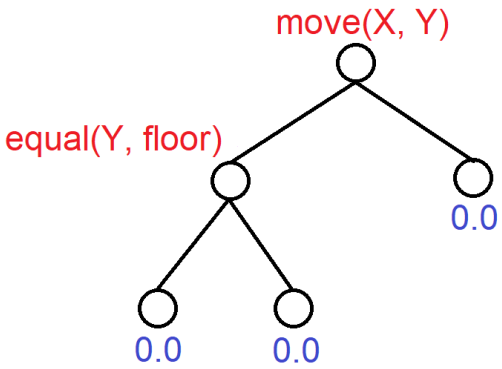
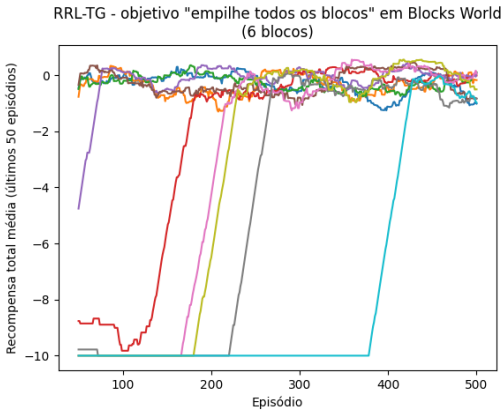
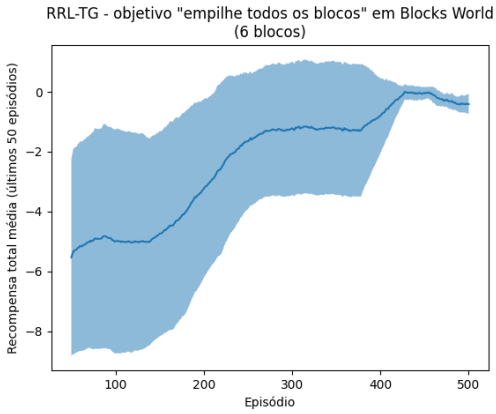


Figura 5.2: *FOLDT inicial para o problema empilhe todos os blocos.*

Usando o algoritmo RRL-TG no objetivo *empilhe todos os blocos* com 6 blocos, os seguintes resultados obtidos são mostrados nos gráficos das figuras 5.3a e 5.3b.



(a) *Gráficos das 10 repetições do experimento.*

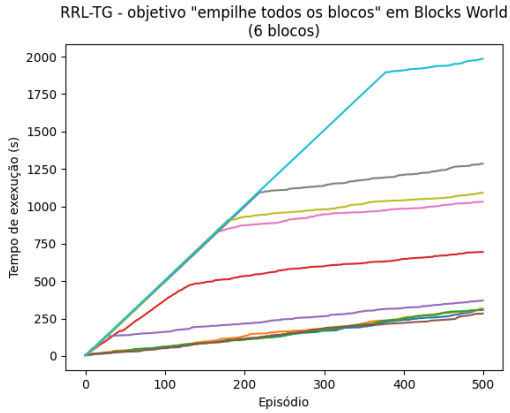


(b) *Média e intervalo de confiança do gráfico 5.3a.*

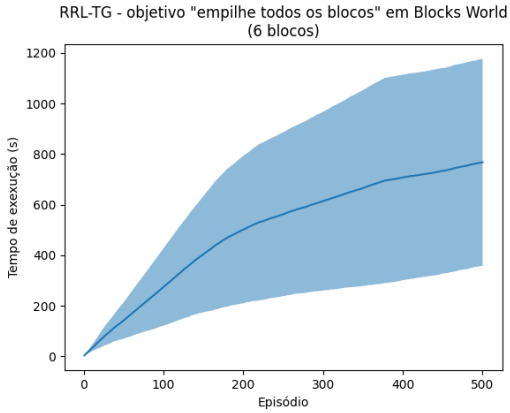
Figura 5.3: *Objetivo empilhe todos os blocos, gráficos de recompensa com RRL-TG.*

Observando o gráfico na Figura 5.3a, é possível perceber que o agente consegue um desempenho final bom, mas a quantidade de episódios necessários até chegar nesse nível de desempenho varia bastante entre repetições do exmperimento. Em algumas repetições o agente começa quase imediatamente com recompensa total média perto de 0.0, enquanto em outras o agente precisou treinar por mais do que 300 episódios até começar a melhorar. Essa variação pode ser vista no gráfico da Figura 5.3b, porém, essa variação reduz significantemente depois do episódio 400. A média entre as repetições da recompensa total média dos últimos 50 episódios é −0.41.

Sobre a análise de tempo de execução do algoritmo, os gráficos que mostram essa informação estão nas Figuras 5.4a e 5.4b.



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 5.4a.*

Figura 5.4: *Objetivo empilhe todos os blocos, gráficos de tempo com RRL-TG.*

Comparando os gráficos nas Figuras 5.3a e 5.4a, de modo geral é possível ver que as repetições que tiveram um desempenho bom desde o começo também têm um tempo de execução menor. Como a preformance tem uma variação grande, o tempo de execução também tem essa variação. De fato, a repetição com omenor tempo de execução demorou cerca de 4 minutos e 43 segundos, enquanto a repetição com o maior tempo de execução demorou cerca de 33 minutos e 6 segundos.

5.5.2 Objetivo desempilhe todos os blocos

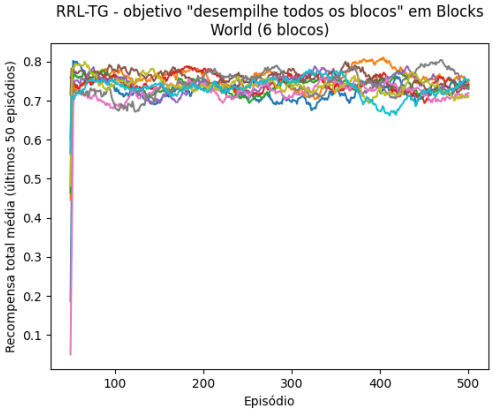
Para o objetivo *desempilhe todos os blocos*, a FOLDT inicial que usamos como árvore inicial do Programa 5.2 doi a mesma que usamos para o objetivo *empilhe todos os blocos*, ou seja, a FOLDT na Figura 5.2. Também usaremos um nível de significância $\beta = 0.4$.

Executando o algoritmo RRL-TG com o objetivo *desempilhe todos os blocos* e com 6 blocos, os resultados obtidos são mostrados nas Figuras 5.5a e 5.5b.

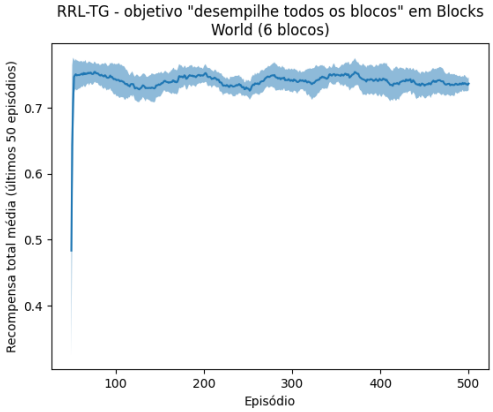
Vendo o gráfico na Figura 5.5a, é fácil ver que o algoritmo RRL-TG teve bastante facilidade com o objetivo *desempilhe todos os blocos*, com todas as repetições do experimento começando com um desempenho bom. A média entre as repetições da recompensa total média dos últimos 50 episódios de treinamento é cerca de 0.74, e a variação do desempenho entre repetições é baixa, assim como pode ser visto no gráfico da Figura 5.5b.

Agora, a análise de tempo de execução pode ser visto nas Figuras 5.6a e 5.6b.

Novamente, olhando o gráfico na Figura 5.6a, podemos ver uma consistência na quantidade de tempo necessário para treinar o agente por 500 episódios, um resultado da consistência do desempenho do agente entre repetições. Em média, a execução dos 500 episódio durou cerca de 1 minuto e 22 segundos.

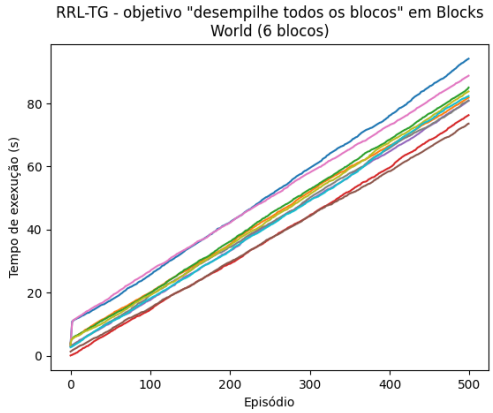


(a) *Gráficos das* 10 *repetições do experimento.*

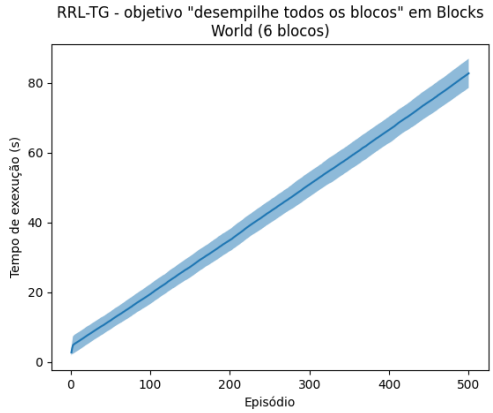


(b) *Média e intervalo de confiança do gráfico 5.5a.*

Figura 5.5: *Objetivo desempilhe todos os blocos, gráficos de recompensa com RRL-TG.*



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 5.6a.*

Figura 5.6: *Objetivo desempilhe todos os blocos, gráficos de tempo com RRL-TG.*

5.5.3 Objetivo empilhe dois blocos específicos

Para o objetivo *empilhe dois blocos específicos*, usaremos a FOLDT mostrada na Figura 5.7 como árvore inicial do Programa 5.2. Ou seja, as informações que a FOLDT inicial distingue são quais dois objetos fazem parte do objetivo *empilhe dois blocos específicos* (por causa do fato relacional *goal(X, Y)* na raiz); quais objetos participam da ação *move(Z, W)*; e se a ação envolve mover Z para o chão ou para outro bloco. Além disso, usaremos um nível de significância $\beta = 0.001$.

Rodando o algoritmo RRL-TG para o problema *empilhe dois blocos específicos* com 6 blocos, os resultados que foram obtidos são mostrados nas Figuras 5.8a e 5.8b.

Os resultados vistos na Figura 5.8a demonstram que, há uma alta variação nos desempenhos de cada repetição, mas em geral, a maioria das repetições convergeram para uma recompensa total média alta. Esssa análise é visível no intervalo de confiança do gráfico da Figura 5.8b, em que o tamanho do intervalo é grande, mas a média tende a um valor próximo de 0.0. De fato, a média das recompensa total média dos últimos 50 episódios entre as repetições foi cerca de -0.13.

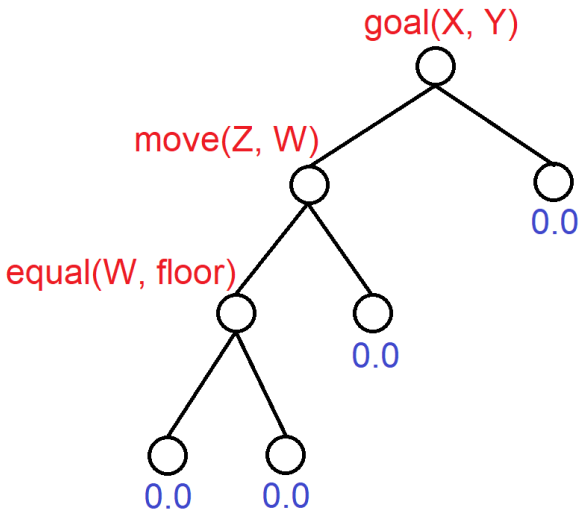
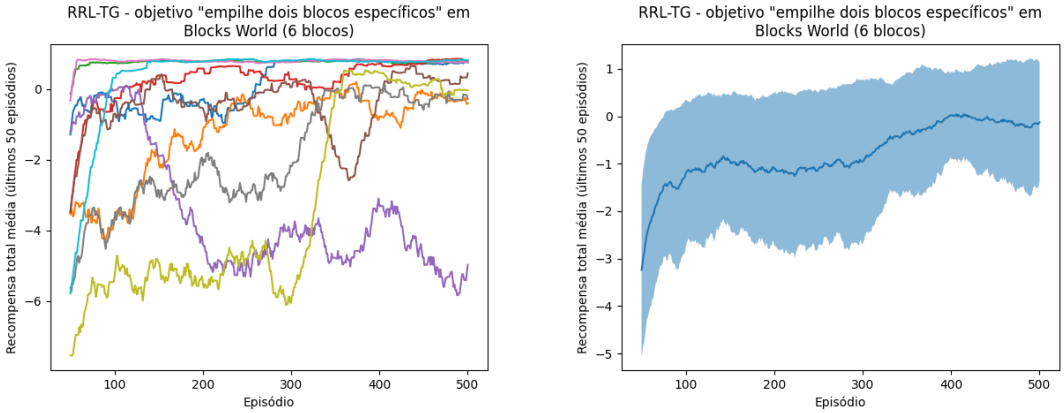
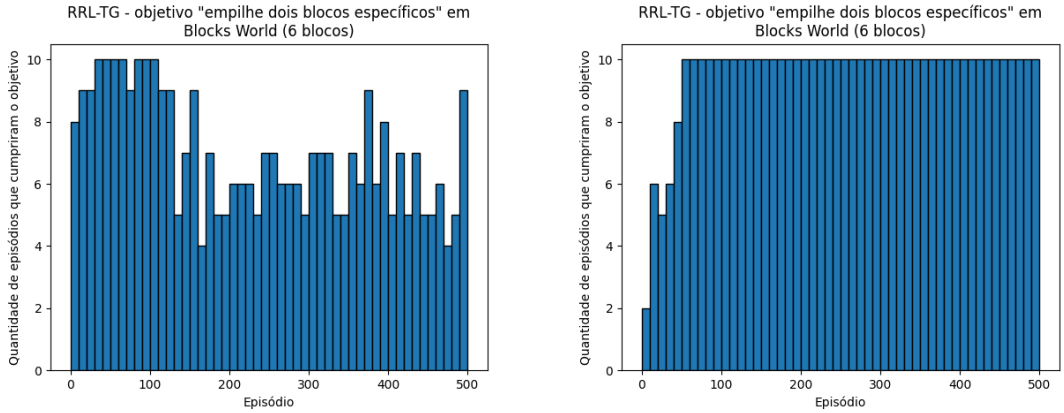


Figura 5.7: *FOLDT inicial para o problema empilhe dois blocos específicos.*



(a) *Gráficos das* 10 *repetições do experimento.* **(b)** *Média e intervalo de confiança do gráfico 5.8a.*

Figura 5.8: *Objetivo empilhe dois blocos específicos, gráficos de recompensa com RRL-TG.*



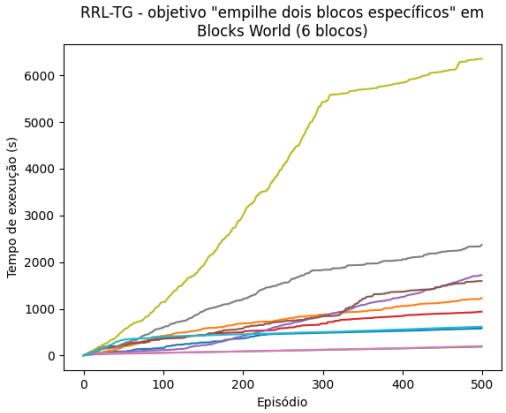
(a) *Repetição com o pior desempenho final.* **(b)** *Repetição com o melhor desempenho final.*

Figura 5.9: *Objetivo empilhe dois blocos específicos, histogramas das repetições com o pior e o melhor desempenho final com RRL-TG.*

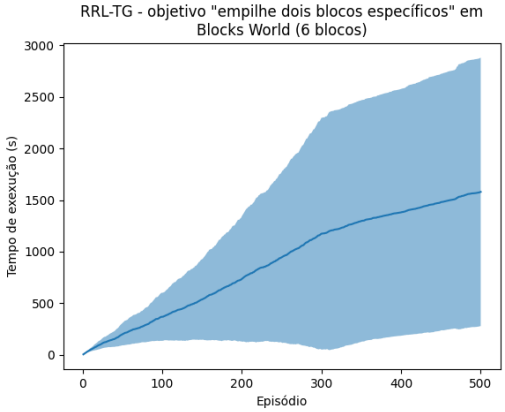
A variação alta indica que há uma inconsistência no desempenho final após os 500 episódios de treinamento. Isso pode ser visto nos histogramas das Figuras 5.9a e 5.9b. Nesses histogramas, cada barra cobre um intervalo de 10 episódios, e a sua altura indica a quantidade desses 10 episódios que o agente cumpriu o objetivo antes do limite de 100 ações. O histograma na Figura 5.9a é da repetição do experimento que obteve o pior desempenho final (a média da recompensa total dos últimos 50 episódios foi cerca de −4.98), e o histograma na Figura 5.9b é da repetição que obteve o melhor desempenho final (a média da recompensa final dos últimos 50 episódios foi cerca de 0.83).

Além disso, note que no histograma da Figura 5.9a, nos primeiros 100 episódios o agente conseguia cumprir o objetivo com uma consistência relativamente alta, mas depois essa consistência diminui. Isso implica que o algoritmo RRL-TG tem uma chance de piorar o desempenho do agente. Uma possível explicação por isso é o fato que estamos fazendo abstrações nos pares estado e ação em $\mathcal{S} \times \mathcal{A}$. Isso significa que estamos agrupando diversos pares estado e ação usando a representação relacional, e por causa disso o algoritmo precisa fazer diversas suposições sobre esses agrupamentos. Assim, a perda de desempenho pode acontecer quando o algoritmo faz alguma suposição incorreta.

Sobre a quantidade de tempo que o algoritmo demorou para executar, essa informação está representado nos gráficos das Figuras 5.10a e 5.10b.



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 5.10a.*

Figura 5.10: *Objetivo empilhe dois blocos específicos, gráficos de tempo com RRL-TG.*

Assim como o gráfico de recompensa, o gráfico de tempo de execução apresenta uma alta variação que nunca diminui. A média de tempo que as 10 repetições demoraram foi cerca de 26 minutos e 19 segundos. Entre essas 10 repetições, o menor tempo de execução foi cerca de 3 minutos e 10 segundos, enquanto o maior tempo de execução foi cerca de 1 hora, 45 minutos e 57 segundos.

5.5.4 Tabela com resumo dos resultados dos experimentos

Os resultados finais após treinar os agentes por 500 episódios usando o algoritmo RRL-TG são mostrados nas seguintes tabelas:

Média de recompensa acumulada dos últimos 50 episódios com RRL-TG			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Média	−0.41	0.74	−0.13
Mínimo	−1.01	0.71	−4.98
Máximo	0.11	0.75	0.83
Desvio padrão	0.47	0.01	1.77
Intervalo de confiança de 95%	−0.75 a −0.07	0.73 a 0.75	−1.39 a 1.14

Tempo para treinar o agente por 500 episódios com RRL-TG			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Média	767.83s	82.77s	1579.96s
Mínimo	283.42s	73.60s	190.16s
Máximo	1986.40s	94.19s	6357.22s
Desvio padrão	572.35s	5.84s	1818.51s
Intervalo de confiança de 95%	358.39s a 1177.26s	78.59s a 86.95s	279.08s a 2880.84s

Capítulo 6

O algoritmo RRL-RIB

O algoritmo RRL-RIB usa uma estratégia chamado de *aprendizado baseada em instâncias*, que é conhecido por ser simples e com bom desempenho. A estratégia de aprendizado baseada em instâncias é guardar um conjunto de exemplos durante o aprendizado. Assim, para estimarmos um valor para um exemplo novo, podemos compará-lo com os exemplos nesse conjunto. Para fazer essa comparação, algum tipo de medida de distância entre exemplos precisa ser definida.

6.1 Distância relacional

Se estivéssemos usando a representação proposicional, a definição de uma distância seria bem mais simples, pois dados dois vetores de números reais com a mesma quantidade de elementos, poderíamos simplesmente usar a distância euclidiana. Porém, como estamos usando a representação relacional, precisaremos de uma forma mais sofisticada de determinar a distância entre dois pares estado e ação de um problema. Tal distância é chamada de *distância relacional* (Driessens, 2004).

Nessa seção, focaremos especificamente na definição de uma distância relacional em um problema do domínio do Mundo dos Blocos. Assumindo que o objetivo seja *empilhe dois blocos específicos*, dado dois pares estado e ação (S_1, A_1) e (S_2, A_2), definiremos a distância relacional entre os dois usando o seguinte algoritmo:

1. Em (S_1, A_1), haverá um fato relacional *move(X1, Y1)* e também um fato relacional *goal(Z1, W1)*. Para cada bloco em {*X1, Y1, Z1, W1*}, dê um rótulo com um caractere **distinto**.

2. Cada bloco em (S_1, A_1) que não recebeu um rótulo no passo *(1)* receberá um rótulo com um mesmo caractere, distinto dos caracteres usados no passo *(1)*

3. Em (S_2, A_2), haverá um fato relacional *move(X2, Y2)* e um fato relacional *goal(Z2, W2)*. Tente dar rótulos **distintos** para os blocos em {*X2, Y2, Z2, W2*} de tal forma que combine com os rótulos dados no passo *(1)*. Ou seja, tente rotular os blocos de tal forma que *X1* tenha o mesmo rótulo de *X2*, e que *Y1* tenha o mesmo rótulo de *Y2*,

e assim por diante. Se não for possível, cada erro aumentará a distância relacional por uma constante real k_1.

4. Cada bloco em (S_2, A_2) que não recebeu um rótulo no passo *(3)* receberá o mesmo rótulo usado para os blocos no passo *(2)*.

5. Represente cada pilha em (S_1, A_1) e em (S_2, A_2) como uma *string*, seguindo os rótulos dados para cada bloco, de baixo para o topo da pilha. Contrua P_1 e P_2, definidos como conjuntos das representações das pilhas como *string* em (S_1, A_1) e em (S_2, A_2), respectivamente.

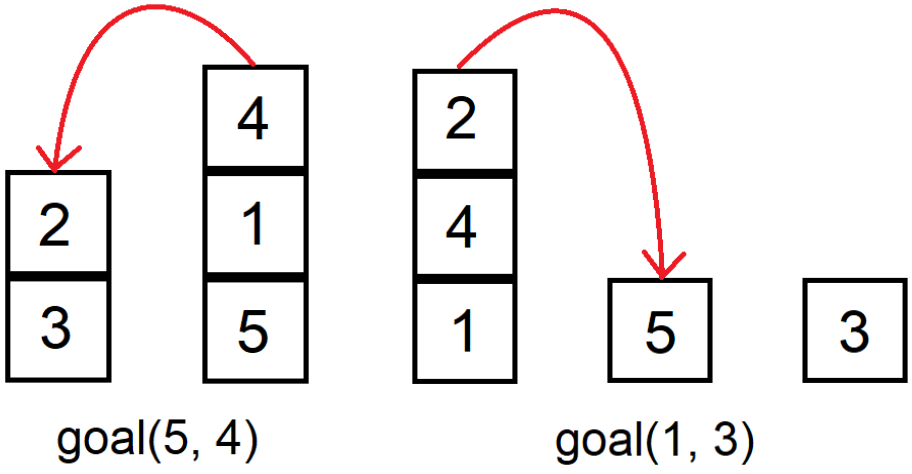
6. Para cada par $(p_1, p_2) \in P_1 \times P_2$, compute a **distância de edição** (Wagner e Fischer, 1974) entre as *strings* p_1 e p_2.

7. Pareie os elementos em P_1 com os elementos em P_2 de tal forma que a soma das distâncias de edição de cada par seja minimizada. A distância relacional aumentará por k_2 vezes essa soma das distâncias de edição, em que k_2 é uma constante real. Também, se não for possível parear todas as pilhas (o que só acontecerá se (S_1, A_1) e (S_2, A_2) tiver quantidade de pilhas diferentes), cada pilha não pareada aumentará a distância relacional por uma constante real k_3.

8. Retorne a distância relacional acumulada até agora.

Caso o objetivo seja *empilhe todos os blocos* ou *desempilhe todos os blocos*, as únicas mudanças no algoritmo acima são os passos (1) e (3), em que podemos ignorar as variáveis *Z1*, *W1*, *Z2* e *W2* (pois não haverá um fato relaiconal *goal(Z1, W1)* e *goal(Z2, W2)*). Assim, basta encontrar rótulos para os blocos em {*X1, Y1*} no passo (1), e encontrar rótulos para os blocos em {*X2, Y2*} no passo (3).

Considere os exemplos de pares estados e ação (S_1, A_1) e (S_2, A_2) demonstrados nas Figuras 6.1a e 6.1b, respectivamente.



(a) *Exemplo de par estado e ação* (S_1, A_1). **(b)** *Exemplo de par estado e ação* (S_2, A_2).

Figura 6.1: *Dois pares estado e ação em problemas no domínio do Mundo dos Blocos com objetivo empilhe dois blocos específicos.*

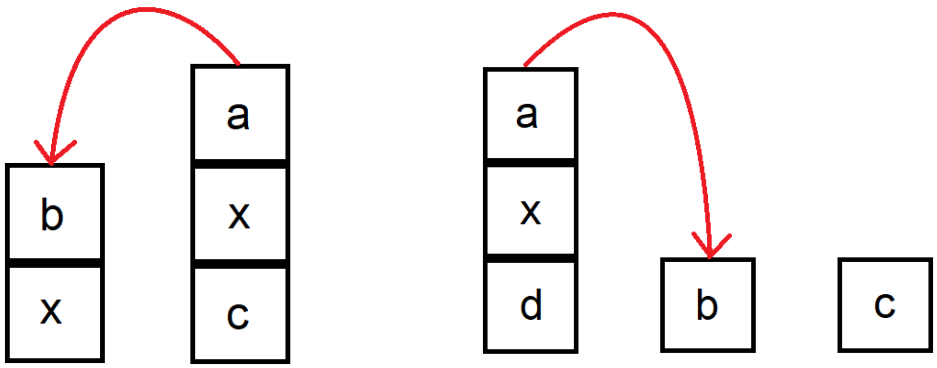
Primeiro, considere que definimos as constantes $k_1 = 2$, e $k_2 = 1$, e $k_3 = 2$. Para computar a distância relacional entre (S_1, A_1) e (S_2, A_2), primeiro vamos rotular os blocos em (S_1, A_1). Nesse par estado e ação, temos os fatos relacionais *move(4, 2)* e *goal(5, 4)*, portanto, no passo (1) do algoritmo, segue que $X1 = 4$, e $Y1 = 2$, e $Z1 = 5$ e $W1 = 4$. Logo, precisamos dar rótulos distintos para os blocos em {4, 2, 5}. Assim, vamos rotular o bloco 4 com o caractere "a", o bloco 2 com o caractere "b", e o bloco 5 com o caractere "c".

Os blocos ainda não rotulados em (S_1, A_1) são os blocos 1 e 3, então vamos rotular ambos blocos com o caractere "x".

Agora, no par estado e ação (S_2, A_2), há os fatos relacionais *move(2, 5)* e *goal(1, 3)*. Portanto, no passo (3) do algoritmo, segue que $X2 = 2$, e $Y2 = 5$, e $Z2 = 1$ e $W2 = 3$. Logo, precisamos dar rótulos distintos para os blocos em {2, 5, 1, 3}. Como cada um desses quatro blocos precisam de um rótulo diferente, é impossível combinar perfeitamente com os rótulos dados para (S_1, A_1) no passo (1) do algoritmo. Mas se em (S_2, A_2) rotularmos o bloco 2 com o caractere "a", o bloco 5 com o caractere "b", o bloco 3 com o caractere "c", e o bloco 1 com o caractere "d", então os blocos $X1$ e $X2$ têm o mesmo rótulo "a", os blocos $Y1$ e $Y2$ têm o mesmo rótulo "b", e os blocos $W1$ e $W2$ têm o mesmo rótulo "c". O único par que não combina é $Z1$ (que recebeu o rótulo "c") e $Z2$ (que recebeu o rótulo "d"), portanto a distância relacional é aumentada por $1 \cdot k_1 = 2$.

O único bloco ainda não rotulado em (S_2, A_2) é o bloco 4, então vamos rotular o bloco 4 com o caractere "x".

Após os passos (1) até (4) do algoritmo, os blocos em (S_1, A_1) e (S_2, A_2) ficaram com os rótulos mostrados nas Figuras 6.2a e 6.2b, respectivamente.



(a) *Rótulos dos blocos no par estado e ação* (S_1, A_1). **(b)** *Rótulos dos blocos no par estado e ação* (S_2, A_2).

Figura 6.2: *Rótulos dados para os blocos nos dois pares estado e ação.*

O próximo passo é transformar cada pilha em uma *string* usando os rótulos. Assim, as pilhas em (S_1, A_1) geram as *strings* no conjunto {*xb*, *cxa*}, enquanto as pilhas em (S_2, A_2) geram as *strings* no conjunto {*dxa*, *b*, *c*}.

Seja edit$(\cdot, \cdot)$ a função que retorna a distância de edição entre duas *strings*. Então podemos computar que:

- edit(*xb*, *dxa*) = 3

- edit(xb, b) = 1
- edit(xb, c) = 3
- edit(cxa, dxa) = 2
- edit(cxa, b) = 4
- edit(cxa, c) = 2

Assim, podemos parear a pilha xb com b, e a pilha cxa com dxa para minimizar a soma das distâncias de edição, que será edit(xb, b) + edit(cxa, dxa) = 3. Isso aumenta a distância relacional por $3 \cdot k_2 = 3$. Além disso, a pilha c em (S_2, A_2) não foi pareada, o que aumenta a distância relacional por $1 \cdot k_3 = 2$.

Assim, a distância relacional entre (S_1, A_1) e (S_2, A_2) será a soma dos aumentos que ocorreram durante o algoritmo. Nesse caso, será $1 \cdot k_1 + 3 \cdot k_2 + 1 \cdot k_3 = 7$.

6.2 Estimação de valor-ação

Com uma distância relacional definida, para cada $i, j \in \mathcal{S} \times \mathcal{A}$, podemos definir dist$(i, j)$ como a distância relacional entre os pares estado e ação i e j.

A ideia principal do algoritmo RRL-RIB é, enquanto o agente interage com o ambiente, vamos guardar um conjunto E de pares estado e ação que o agente percorre. Dessa forma, dado um par estado e ação $i \in \mathcal{S} \times \mathcal{A}$, podemos usar E para aproximar o valor de $q(i)$ da seguinte forma (Driessens, 2004):

$$\hat{q}^E(i) := \frac{\sum_{j \in E} \frac{q^E(j)}{\text{dist}(i,j)}}{\sum_{j \in E} \frac{1}{\text{dist}(i,j)}}, \tag{7.1}$$

em que $q^E(j)$ é a estimação de $q^*(j)$ quando o agente passou pelo par estado e ação j. Ou seja, se $j = (s, a)$, e fazer a ação a quando o agente estava no estado s resultou em recompensa r e próximo estado s', então, similarmente a Regra de Atualização (3.4) que usamos em Q-Learning, estimamos que:

$$q^E(s, a) := \hat{q}^E(s, a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}(s')} \hat{q}^E(s', a') - \hat{q}^E(s, a)). \tag{7.2}$$

A Definição (7.2) é o equivalente a q' no Programa 4.1 na linha 14.

Para cada par estado e ação $j \in \mathcal{S} \times \mathcal{A}$ que guardamos em E, vamos também guardar o valor de $q^E(j)$ que computamos com a Definição (7.2). Note que no começo do algoritmo, vamos ter que E é o conjunto vazio. Nesse caso, definimos que $\hat{q}^E(i) = 0$ para qualquer $i \in \mathcal{S} \times \mathcal{A}$ se E for vazio.

Intuitivamente, a Definição (7.1) é simplesmente uma média ponderada entre os $q^E(j)$

para todo $j \in E$. O peso de $q^E(j)$ nessa média ponderada será $\frac{1}{\text{dist}(i,j)}$, ou seja, quando menor for a distância relacional entre i e j, maior é o peso de $q^E(j)$. A heurística utilizada pela Definição (7.1) é a ideia que se dois pares estado e ação $i, j \in \mathcal{S} \times \mathcal{A}$ forem semelhantes, o esperado é que os valores de $q^*(i)$ e $q^*(j)$ também serão semelhantes.

Há um problema com a Definição (7.1) caso existir algum $j \in E$ tal que $\text{dist}(i, j) = 0$, pois isso resultará em uma divisão por zero. Para evitar isso, vamos redefinir a definição para que seja (Driessens, 2004):

$$\hat{q}^E(i) := \frac{\sum_{j \in E} \frac{q^E(j)}{\text{dist}(i,j)+\delta}}{\sum_{j \in E} \frac{1}{\text{dist}(i,j)+\delta}}, \tag{7.3}$$

em que $\delta \in \mathbb{R}$ é uma constante real pequena. Nos experimentos na seção 6.6, usamos $\delta = 0.01$.

É usando o conjunto E junto com a definição (7.3) que o algoritmo RRL-RIB resolve o problema de *predição*.

6.3 Limitação do influxo

Quanto mais exemplos de pares estado e ação em E, melhor será a estimação $\hat{q}$. Porém, na prática, se E tiver muitos exemplos então o algoritmo RRL-RIB ficará muito lento para ser viável. Por causa disso, quando o agente percorre um par estado e ação $i \in \mathcal{S} \times \mathcal{A}$, precisamos determinar se compensa adicionar esse exemplo para E. Há dois critérios que vamos utilizar, e vamos incluír i em E se pelo menos um dos critérios for satisfeito.

6.3.1 Limite local

O primeiro critério usa as estimações $\hat{q}^E(i)$ e $q^E(i)$, vimos na Definição (7.3) e Definição (7.2), respectivamente. Se a diferença entre os valores de $\hat{q}^E(i)$ e $q^E(i)$ for grande, isso significa que os exemplos em E não são o suficiente para uma boa estimação de $q^E(i)$. Assim, a adição de i em E ajudaria em melhorar essa estimação, e de outros pares estado e ação semelhantes a i.

O que resta é definir quão grande a diferença entre $\hat{q}^E(i)$ e $q^E(i)$ precisa ser para adicionarmos i em E. É difícil de determinar um valor constante para ser esse limite para a inclusão de i, pois se pares estado e ação semelhantes a i naturalmente resultarem em valores retornados por q^* com variação muito alta, poderemos estar incluindo exemplos em E desnecessariamente.

Para resolver esse problema vamos definir um limite proporcional ao desvio padrão de exemplos próximos a i em E. Mais formalmente, seja E^i_{local} um conjunto dos ℓ pares estado e ação em E que minimizam a distância relacional com i, em que ℓ é uma constante inteira pré-definida pelo usuário. Assim, a média local de i com respeito a E é definido como:

$$\overline{E^i_{local}} := \frac{\sum_{j \in E^i_{local}} q^E(j)}{\ell},$$

e o desvio padrão local de i com respeito a E é definido como:

$$\sigma^E_{local}(i) := \sqrt{\frac{\sum_{j \in E^i_{local}} (q^E(j) - \overline{E^i_{local}})^2}{\ell}}.$$

Assim, vamos determinar que um novo par estado e ação $i \in \mathcal{S} \times \mathcal{A}$ será incluído em E se (Driessens, 2004):

$$|\hat{q}^E(i) - q^E(i)| > \sigma^E_{local}(i) \cdot F_l, \tag{7.4}$$

em que $F_l \in \mathbb{R}$ é uma constante real adequado pré-definida pelo usuário.

6.3.2 Limite global

O segundo critério segue da ideia de que a variância de q^* entre todos os pares estado e ação não é constante, ou seja, pode haver $i, j \in \mathcal{S} \times \mathcal{A}$ tais que a variância do valor de q^* para pares estado e ação próximos de i é alta, mas a variância do valor de q^* para pares estado e ação próximas de j é baixa (lembrando que o conceito de "próximo" para pares estado e ação em $\mathcal{S} \times \mathcal{A}$ pode ser definido com a distância relacional). Intuitivamente faz sentido querermos guardar mais exemplos de pares estado e ação próximas de i do que as próximas de j, mas o Critério (7.4) faz o oposto, pois a alta variância de pares estado e ação próximos de i faz com que $\sigma^E_{local}(i)$ tenha um valor alto.

Por causa disso, o segundo critério determina que se um par estado e ação $i \in \mathcal{S} \times \mathcal{A}$ estiver em uma região em $\mathcal{S} \times \mathcal{A}$ com variância grande em comparação com outras, então vamos incluir i em E. Mais formalmente, a média das estimações dos exemplos em E pode ser definido como:

$$\overline{E} := \frac{\sum_{j \in E} q^E(j)}{|E|},$$

em que $|E|$ é o número de elementos em E. Assim, podemos definir o desvio padrão global em E como:

$$\sigma^E_{global} := \sqrt{\frac{\sum_{j \in E} (q^E(j) - \overline{E})^2}{|E|}}.$$

Dessa forma, podemos determinar que um novo par estado e ação $i \in \mathcal{S} \times \mathcal{A}$ será incluído em E se (Driessens, 2004):

$$\sigma^E_{local}(i) > \frac{\sigma^E_{global}}{F_g}, \tag{7.5}$$

em que $F_g \in \mathbb{R}$ é uma constante real adequada pré-definida pelo usuário.

6.4 Exclusão de exemplos

Como já discutimos antes, se E acumular muitos exemplos o algoritmo RRL-RIB ficará muito lento para ser útil na prática. Mesmo as limitações que vimos na seção 6.3 pode não ser o suficiente, então é necessário que o usuário define um limite $m \in \mathbb{N}$ para o tamanho do conjunto E. Por causa disso, se o Critério (7.4) ou o Critério (7.5) determinar que um novo par estado e ação $i \in \mathcal{S} \times \mathcal{A}$ vai entrar em E, e isso causar o tamanho de E a ultrapassar o limite m, precisamos decidir qual exemplo em E vamos excluir.

A ideia que usaremos é o fato de que, para estimar $q^*(j)$ para algum $j \in \mathcal{S} \times \mathcal{A}$, pela Definição (7.3), fazemos uma média ponderada com a distância relacional entre j e pares estado e ação em E. Notavelmente, os pares estado e ação em E mais próximos de j têm um peso maior nessa média ponderada. Portanto, se algum $i \in E$ estiver próximo de outros pares estado e ação em E que têm um erro de estimação alta, uma possível explicação é que i está amplificando esse erro na média ponderada, logo, i é um bom candidato para excluírmos de E.

Formalmente, para cada $i \in E$, vamos calcular uma pontuação chamada *EP-score* (abreviada do inglês *Error Proximity score*). Essa pontuação é definida da seguinte forma (Driessens, 2004):

$$EP\text{-}score(i) := \sum_{\substack{j \in E \\ j \neq i}} \frac{|q^E(j) - \hat{q}^E(j)|}{\text{dist}(i,j) + \delta}, \tag{7.6}$$

em que δ é o mesmo usado na Definição (7.3), para evitar divisão por zero. Assim, um par estado e ação $i \in E$ que maximizar $EP\text{-}score(i)$ é o que será excluído de E.

Esse método de exclusão de exemplos, junto com a limitação de inclusão de exemplos que vimos na seção 6.3, é como o algoritmo RRL-RIB resolve o problema de *controle*.

6.5 O algoritmo RRL-RIB

Com tudo que vimos até agora, podemos apresentar o algoritmo RRL-RIB (Driessens, 2004):

Programa 6.1 Algoritmo RRL-RIB, para estimar $Q \approx q^*$.

```
1   Entrada: MDP M = ⟨S, A, R, p⟩
2            γ ∈ (0, 1]
3            α ∈ (0, 1]
4            ε > 0 pequeno
5            ℓ, m ∈ ℕ
6            F_l, F_g ∈ ℝ
7
8   E ← {}
9   Para cada episódio, faça:
10      Inicialize um estado inicial S
11      Para cada passo no tempo, faça:
12          Escolha ação A a partir de S usando a política ε−suave gerado com E
13          Faça a ação A e observe a recompensa R e o próximo estado S′
14          q ← q̂^E((S, A))
15          Q′ ← {q̂^E((S′, a)) | a ∈ A(S′)}
16          q′ ← q + α(R + γ max Q′ − q)
17          Se |q − q′| > σ^E_local((S, A)) · F_l ou σ^E_local((S, A)) > σ^E_global / F_g, faça:
18              E ← E ∪ {(S, A)}
19              Se |E| > m, faça:
20                  i ← argmax_{j∈E}{EP-score(j)}
21                  E ← E \ {i}
22          S ← S′
23      Até que S seja terminal
```

No Programa 6.1, nas linhas 1 a 6 o programa recebe um MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$ e os valores de γ, de α, de ε, de l, de m, de F_l, e de F_g. Inicializamos E como um conjunto vaizo na linha 8. Assim, para cada episódio, inicializamos um estado inicial S na linha 10. Para cada passo no tempo do episódio, escolhemos uma ação A a partir do estado S usando a política gulosa ε-suave gerada com o conjunto E e a estimação na Definição (7.3) na linha 12 do programa, e executamos a ação A no ambiente, observando a recompensa R e o próximo estado S' na linha 13. Nas linhas 14 a 16 computamos $\hat{q}^E((S, A))$ e $q^E((S, A))$ usando a Definição (7.3) e a Definição (7.2), respectivamente.

O restante do algoritmo é a parte de *controle* do RRL-RIB. Primeiro, verificamos se o Critério (7.4) ou o Critério (7.5) é satisfeito na linha 18. Se for o caso, então adicionamos o exemplo (S, A) em E (junto com o $q^E((S, A))$ computado antes) na linha 18. Depois, verificamos se o tamanho de E ultrapassou o limite m na linha 19. Se for o caso, computamos o *EP-score* para todos os exemplos em E, e escolhemos um exemplo i que o maximize na linha 20. Assim, excluímos i de E na linha 21. Na linha 22 trocamos o valor de S para S', preparando para o próximo passo no tempo do episódio. Finalmente, na linha 23 verificamos se o episódio terminou.

Dessa forma, o algoritmo RRL-RIB é um algoritmo da classe Q-Learning Relacional, e ele resolve os três problemas que discutimos na seção 4.2 da seguinte forma:

1. A inicialização de $\hat{Q}$ é feita quando inicializamos E como um conjunto vazio.

2. O problema de *predição*, ou seja, a estimação $\hat{Q}(s, a)$ para qualquer $(s, a) \in \mathcal{S} \times \mathcal{A}$, é feito usando E e computando $\hat{q}^E((s, a))$ usando a Definição (7.3).

3. O problema de *controle* é resolvido com o que vimos nas seções 6.3 e 6.4. Para cada $(s, a) \in \mathcal{S} \times \mathcal{A}$ que o agente percorre, verificamos se ele satisfaz o Critério (7.4) ou o Critério (7.5) para determinar se o incluímos em E. Depois, se o tamanho de E ultrapassar o limite passado pelo usuário, computamos o *EP-score* que vimos na Definição (7.6) para todo exemplo em E para determinar qual é o melhor exemplo para excluir de E.

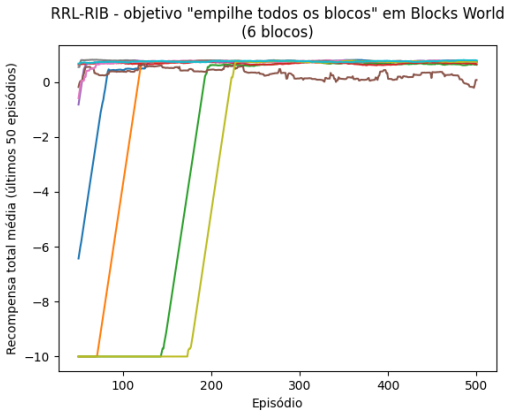
6.6 RRL-RIB no Mundo dos Blocos

Para cada um dos experimentos a seguir, rodamos o Programa 6.1 com os parâmetros $\gamma = 0.95$, e $\alpha = 1.0$, e $\varepsilon = 0.1$, e $\ell = 16$, e $m = 64$. Os parâmetros F_l e F_g serão definidos no começo das seções de cada objetivo.

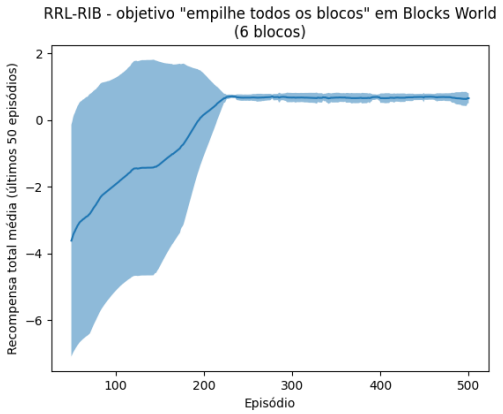
Cada experimento rodou o algoritmo RRL-RIB por 500 episódios. Assim como discutimos na seção 4.4, os gráficos de recompensa total por episódio mostram a média da recompensa total obtida no episódio junto com os 49 episódios anteriores, para facilitar a leitura do gráfico.

6.6.1 Objetivo empilhe todos os blocos

Para o objetivo *empilhe todos os blocos*, usaremos os parâmetros $F_l = 1.2$ e $F_g = 0.8$ para o Programa 6.1. Assim, os resultados obtidos são mostrados nos gráficos das Figuras 6.3a e 6.3b.



(a) *Gráficos das 10 repetições do experimento.*



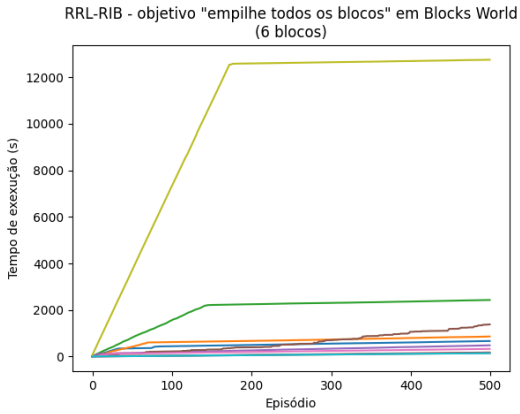
(b) *Média e intervalo de confiança do gráfico 6.3a.*

Figura 6.3: *Objetivo empilhe todos os blocos, gráficos de recompensa com RRL-RIB.*

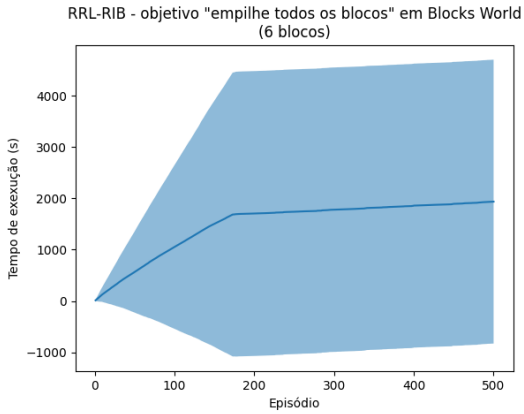
Observando o gráfico na Figura 6.3a, pode ser visto que em todas as repetições do experimento o agente consegue um bom desempenho final. A maior diferença entre repetições do experimento é a quantidade de episódios que o agente precisou ser treinado até ter um desempenho bom. Isso pode ser visto no gráfico na Figura 6.3b, em que nos primeiros episódios há uma variância alta no desempenho do agente, mas depois de

aproximadamente 100 episódios essa variância reduz significantemente, e a média da recompensa total demonstra um bom desempenho do agente. A média entre as repetições da recompensa total média dos últimos 50 episódios é 0.65.

Sobre a análise de tempo de execução do algoritmo, os gráficos que mostram essa informação estão nas Figuras 6.4a e 6.4b.



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 6.4a.*

Figura 6.4: *Objetivo empilhe todos os blocos, gráficos de tempo com RRL-RIB.*

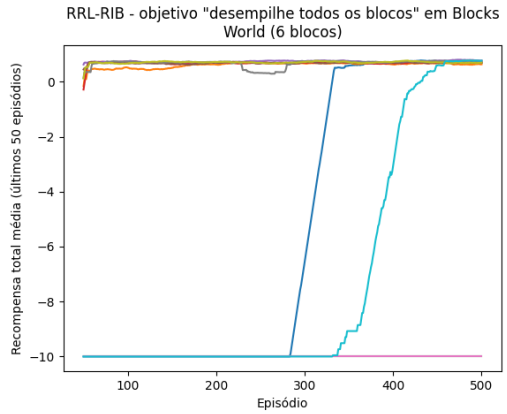
Vendo o gráfico na Figura 6.4a, de modo geral, treinar um agente usando o algoritmo RRL-RIB por 500 episódios demora menos do que 30 minutos. Porém, há um "outlier" que demorou cerca de 3 horas 32 minutos e 32 segundos, o que foi a repetição do experimento com o maior tempo de execução. Por causa disso, a variância encontrada no gráfico 6.4b é bem alto. Em média, o tempo de execução de treinar o agente por 500 episódios usando o algoritmo RRL-RIB é cerca de 32 minutos e 16 segundos. A repetição do experimento que demorou menos tempo treinou o agente por apenas 2 minutos e 23 segundos.

6.6.2 Objetivo desempilhe todos os blocos

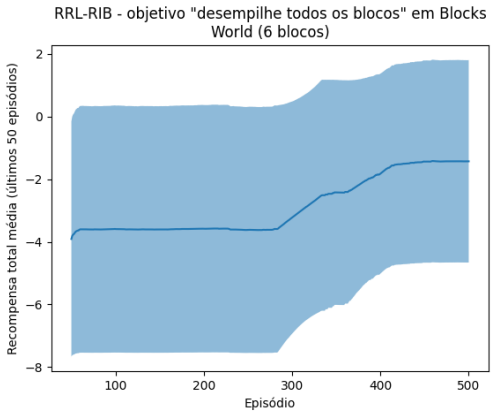
Para o objetivo *desempilhe todos os blocos*, usaremos os parâmetros $F_l = 1.2$ e $F_g = 0.8$ para o Programa 6.1. Assim, os resultados obtidos são mostrados nos gráficos das Figuras 6.5a e 6.5b.

Similarmente ao objetivo *empilhe todos os blocos*, de modo geral cada repetição do experimento obteve um desempenho final bom, com a maior diferença entre repetições sendo a quantidade de episódios de treinamento necessário até que o agente obter esse desempenho bom.

Porém, pode ser visto no gráfico na Figura 6.6a que há uma repetição do experimento em que o agente não teve uma melhora no desempenho, resultando na média da recompensa total dos últimos 50 episódios dessa repetição do experimento sendo −10.0, o menor possível. O motivo por isso é que o agente nunca passou por um par estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}$ que resultou no objetivo ser cumprido, logo, o E no algoritmo RRL-RIB dessa repetição não tinha nenhuma informação sobre como resolver o objetivo. Esse problema deveria ser resolvido usando uma política ε-suave (o que ocorre na linha 12 do Programa 6.1), mas 500



(a) *Gráficos das* 10 *repetições do experimento.*



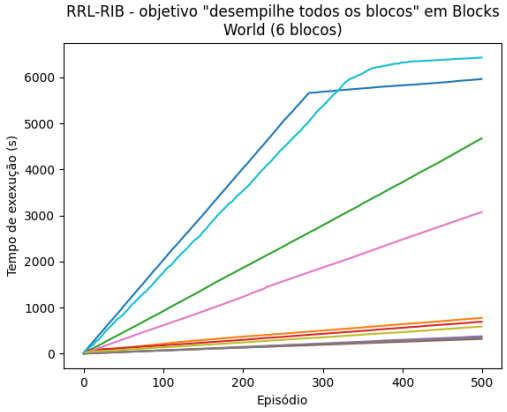
(b) *Média e intervalo de confiança do gráfico 6.5a.*

Figura 6.5: *Objetivo desempilhe todos os blocos, gráficos de recompensa com RRL-RIB.*

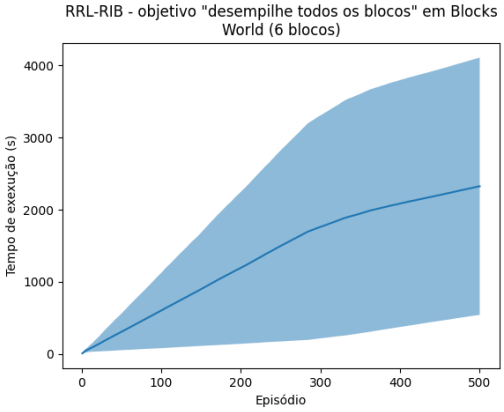
episódios não foi o suficiente para o agente explorar os pares estado e ação necessários para obter um bom desempenho.

Por causa disso, a variância no gráfico na Figura 6.5b é alta em todo o gráfico, e a média entre as 10 repetições de recompensa acumulada dos últimos 50 episódios é −1.43, o que foi altamente influenciado por essa repetição com desempenho final ruim.

Sobre a análise de tempo de execução do algoritmo, os gráficos que mostram essa informação estão nas Figuras 6.6a e 6.6b.



(a) *Gráficos das* 10 *repetições do experimento.*



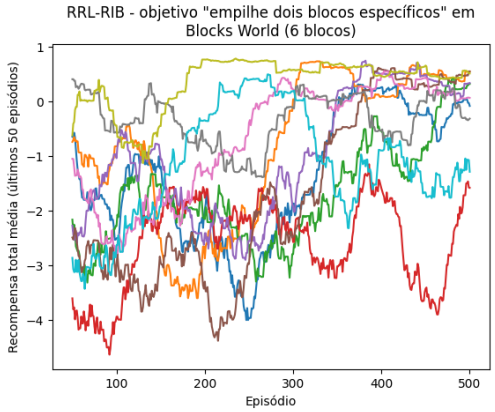
(b) *Média e intervalo de confiança do gráfico 6.6a.*

Figura 6.6: *Objetivo desempilhe todos os blocos, gráficos de tempo com RRL-RIB.*

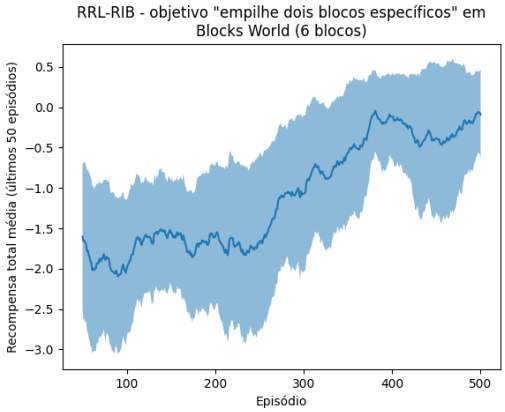
Os tempos de execução das repetições do experimento apresentam uma variância alta, o que pode ser visto no gráfico na Figura 6.6b. Em média, o tempo de execução de treinar o agente por 500 episódios usando o algoritmo RRL-RIB é cerca de 38 minutos e 45 segundos. A repetição que teve o menor tempo de execução demorou 5 minutos e 22 segundos, e a repetição que teve o maior tempo de execução demorou 1 hora 47 minutos e 10 segundos.

6.6.3 Objetivo empilhe dois blocos específicos

Para o objetivo *empilhe dois blocos específicos*, usaremos os parâmetros $F_l = 0.5$ e $F_g = 8.0$ para o Programa 6.1. Assim, os resultados obtidos são mostrados nos gráficos das Figuras 6.7a e 6.7b.



(a) *Gráficos das* 10 *repetições do experimento.*

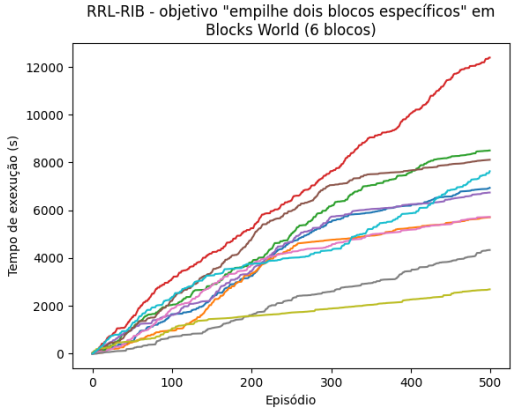


(b) *Média e intervalo de confiança do gráfico 6.7a.*

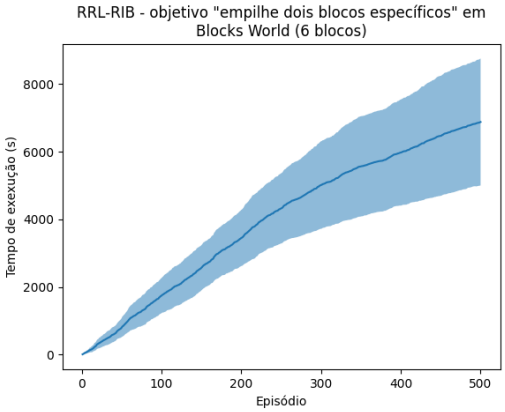
Figura 6.7: *Objetivo empilhe dois blocos específicos, gráficos de recompensa com RRL-RIB.*

É difícil de interpretar o gráfico na Figura 6.7a, pois cada repetição do experimento teve uma trajetória de desempenho do agente bem diferente um do outro. Porém, podemos usar o gráfico na Figura 6.7b para ver que, em geral, o desempenho do agente melhora quanto mais episódios o agente é treinado usando RRL-RIB, com a média entre as 10 repetições da média da recompensa acumulada dos últimos 50 episódios sendo −0.09.

Sobre a análise de tempo de execução do algoritmo, os gráficos que mostram essa informação estão nas Figuras 6.8a e 6.8b.



(a) *Gráficos das* 10 *repetições do experimento.*



(b) *Média e intervalo de confiança do gráfico 6.8a.*

Figura 6.8: *Objetivo empilhe dois blocos específicos, gráficos de tempo com RRL-RIB.*

As grandes diferenças na trajetória de desempenho do agente entre repetições diferentes do experimento é refletido na alta variância de tempo de execução vista no gráfico na Figura 6.8b. Em média, o tempo de execução de treinar o agente por 500 episódios usando

RRL-RIB foi 1 hora 54 minutos e 38 segundos. A repetição do experimento com o menor tempo de execução demorou 44 minutos e 52 segundos, enquanto a repetição com o maior tempo de execução demorou 3 horas 26 minutos e 32 segundos.

6.6.4 Tabela com resumo dos resultados dos experimentos

Os resultados finais após treinar os agentes por 500 episódios usando o algoritmo RRL-RIB são mostrados nas seguintes tabelas:

Média de recompensa acumulada dos últimos 50 episódios com RRL-RIB			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Média	0.65	−1.43	−0.09
Mínimo	0.07	−10.00	−1.58
Máximo	0.78	0.77	0.55
Desvio padrão	0.21	4.52	0.77
Intervalo de confiança de 95%	0.50 a 0.80	−4.66 a 1.80	−0.64 a 0.46

Tempo para treinar o agente por 500 episódios com RRL-RIB			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Média	1936.30s	2325.79s	6878.94s
Mínimo	143.95s	322.53s	2692.66s
Máximo	12753.27s	6430.51s	12392.01s
Desvio padrão	3866.63s	2493.40s	2621.49s
Intervalo de confiança de 95%	−829.72s a 4702.32s	542.12s a 4109.46s	5003.64s a 8754.25s

Capítulo 7

Conclusão

Agora temos resultados de experimentos para resolver três objetivos diferentes no domínio do Mundo dos Blocos (*empilhe todos os blocos*, *desempilhe todos os blocos*, e *empilhe dois blocos específicos*) usando três algoritmos diferentes (Q-Learning, RRL-TG, e RRL-RIB). Vamos comparar os resultados obtidos para analisar a eficiência de cada algoritmo, tanto em relação ao desempenho do agente quanto em relação ao tempo de execução.

7.1 Desempenho do agente

O primeiro ponto que pode ser observado é o desempenho final do agente com cada algoritmo. De modo geral, os algoritmos de aprendizado por reforço relacional (RRL-TG e RRL-RIB) têm o potencial de resultarem em agentes com desempenho tão bom ou melhor do que agentes treinado com Q-Learning. Esse fato pode ser verificado observando o desempenho final da melhor repetição do experimento de cada algoritmo:

Valor da média de recompensa acumulada dos últimos k episódios da repetição do experimento que a maximizou.			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Q-Learning (k = 1000)	0.49	0.55	−3.07
RRL-TG (k = 50)	0.11	0.75	0.83
RRL-RIB (k = 50)	0.78	0.77	0.55

Podemos ver que em quase todos os casos o melhor desempenho final usando algoritmos de aprendizado por reforço relacional é melhor do que quando usamos Q-Learning. Mesmo no caso de usarmos RRL-TG no objetivo *empilhe todos os blocos*, cujo melhor desempenho final do agente foi pior do que com Q-Learning, o algoritmo obteve um bom desempenho final.

O segundo ponto que podemos observar é a eficácia dos episódios para o aprendizado, isto é, quão rápido o agente consegue aprender em relação à quantidade de episódios. Para visualizar esse fator, para cada objetivo vamos combinar os gráficos de desempenho de

cada algoritmo em um únco gráfico. Diferente dos experimentos que fizemos com RRL-TG e RRL-RIB, os experimentos com Q-Learning treinaram o agente por 100000 episódios em vez de por 500 episódios, portanto, para analisar a eficácia dos episódios para o aprendizado, mostraremos nos gráficos combinados apenas os primeiros 500 episódios dos experimentos que usaram Q-Learning. Os resultados podem ser observados nas Figuras 7.1, 7.2 e 7.3.

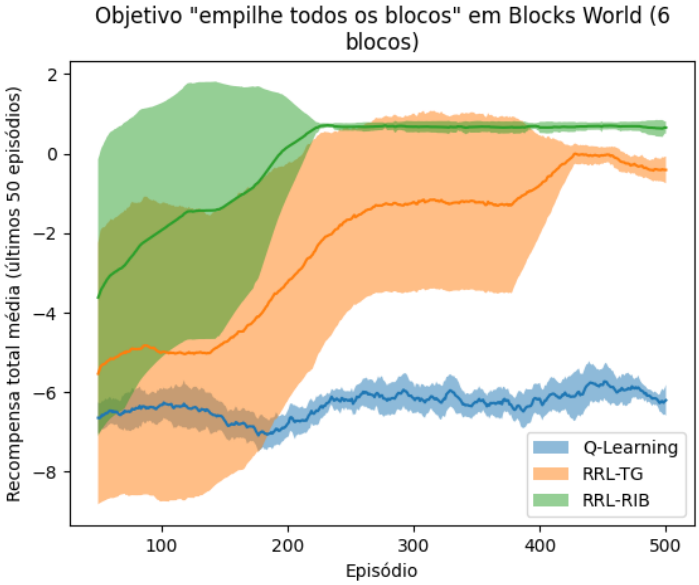


Figura 7.1: *Comparação da média entre as* 10 *repetições do experimento da média da recompensa acumulada dos últimos* 50 *episódios e intervalo de confiança de* 95% *entre os três algoritmos, com o objetivo empilhe todos os blocos.*

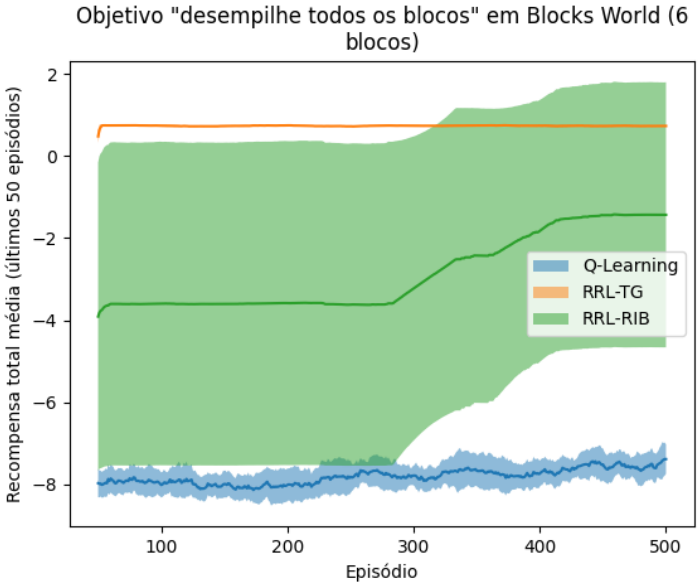


Figura 7.2: *Comparação da média entre as* 10 *repetições do experimento da média da recompensa acumulada dos últimos* 50 *episódios e intervalo de confiança de* 95% *entre os três algoritmos, com o objetivo desempilhe todos os blocos.*

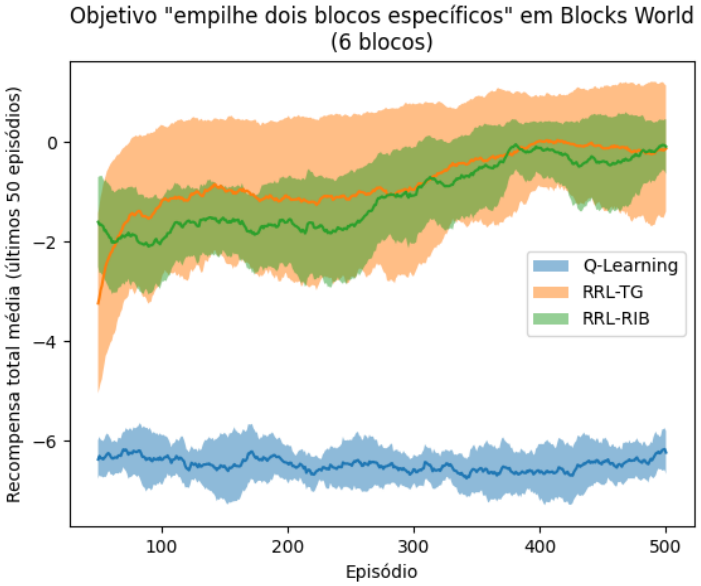


Figura 7.3: *Comparação da média entre as* 10 *repetições do experimento da média da recompensa acumulada dos últimos* 50 *episódios e intervalo de confiança de* 95% *entre os três algoritmos, com o objetivo empilhe dois blocos específicos.*

Podemos observar nesses gráficos que, em todos os objetivos, o agente treinado com algoritmos de RRL consegue aprender bem mais com poucos episódios comparado com um agente treinado com Q-Learning. Assim, combinando isso com o ponto anterior, concluímos que, de modo geral, algoritmos de RRL precisam de bem menos episódios para alcançar o superar o desempenho do agente com Q-Learning.

Outra observação que pode ser feita com esses gráficos é a variância do desempenho do agente. De modo geral, usar algoritmos de RRL resultou em uma variância do desempenho maior do que quando usamos Q-Learning. Uma possível explicação por isso é o fato de que algoritmos de RRL fazem abstrações em pares estado e ação $(s, a) \in \mathcal{S} \times \mathcal{A}$, logo, vários pares estado e ação são agrupados nessa abstração, o que pode deixar o aprendizado mais instável.

7.2 Tempo de execução

Em relação à eficiência dos algoritmos em relação ao tempo de execução, isso pode ser analisado vendo, em média, quanto tempo cada algoritmo demora para treinar um agente por um certo número de episódios. Essa informação pode ser observada na seguinte tabela:

Média entre as 10 repetições do experimento do tempo de execução para treinar um agente por n episódios.			
	Empilhe todos os blocos	Desempilhe todos os blocos	Empilhe dois blocos específicos
Q-Learning (n = 100000)	136.41s	126.07s	699.18s
RRL-TG (n = 500)	767.83s	82.77s	1579.96s
RRL-RIB (n = 500)	1936.30s	2325.79s	6878.94s

Como podemos ver, de modo geral os algoritmos de RRL precisam de bem mais tempo por episódio para treinar um agente, comparado com Q-Learning, principalmente quando consideramos o fato de que treinamos um agente com Q-Learning por 200 vezes mais episódios do que com os algoritmos de RRL.

Existe uma variância maior no tempo de execução de algoritmos de RRL comparado com Q-Learning. Isso pode ser explicado pela grande variância no desempenho do agente quando treinado por algoritmos de RRL, pois quanto melhor for o desempenho do agente, menos ações são necessárias para terminar um episódio, o que consequentemente faz os espisódios necessitarem de menos tempo de execução até o seu término. Assim, uma alta variância no desempenho do agente implica uma alta variância no tempo de execução.

Referências

[Blockeel e Raedt 1998] Hendrik Blockeel e Luc De Raedt. "Top-down induction of first-order logical decision trees". *Elsevier, Artificial Intelligence 101* (mar. de 1998), pp. 287–292 (citado na pg. 39).

[Driessens 2004] Kurt Driessens. "Relational Reinforcement Learning". Tese de dout. Leuven, Bélgica: Universidade Católica de Lovaina, mai. de 2004 (citado nas pgs. 25–27, 29, 31, 43, 46, 55, 58–61).

[Dzeroski *et al.* 2001] Saso Dzeroski, Luc De Raedt e Kurt Driessens. "Relational reinforcement learning". *Machine Learning* (2001), pp. 7–52 (citado na pg. 42).

[Puterman 1994] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994 (citado na pg. 5).

[Sutton e Barto 2015] Richard S. Sutton e Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2ª ed. The MIT Press, 2015 (citado nas pgs. 1, 6, 7, 9–13, 15–23).

[Wagner e Fischer 1974] Robert A. Wagner e Michael J. Fischer. "The string-to-string correction problem". *Journal of the ACM* (jan. de 1974), pp. 168–173 (citado na pg. 56).

Índice remissivo