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

Aprendizagem por Refoço Relacional
uma análise de sua 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. **Aprendizagem por Refoço Relacional: *uma análise de sua 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 text text text text text.

Keywords: Keyword1. Keyword2. Keyword3.

Lista de abreviaturas

PDM	Processo de Decisão de Markov (*Markov Decision Process*)
DP	Dynamic Programming (*Dynamic Programming*)
MC	Monte Carlos (*Monte Carlos*)
TD	Temporal Difference (*Temporal Difference*)
RRL	Aprendizagem 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

Capítulo 1

Sobre aprendizagem por reforço

Os primeiros 5 capítulos dessa monografia são baseados principalmente no livro *Reinforcement Learning: An Introduction* (Sutton e Barto, 2015).

1.1 O que é aprendizagem por reforço

A essência de aprendizagem por reforço 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 todas situações a alguma ação para ser tomada, de tal forma que alguma forma de recompensa seja maximizada.

É importante notar que aprendizagem por reforço é diferente de aprendizagem supervisionada, pois este requer que todo dado de treinamento tenha um rótulo, determinando qual é a resposta correta. Essa abordagem não é ideal para problemas que envolvem interagir com um ambiente, pois muitas vezes não sabemos qual é a ação ideal para uma dada situação.

Além disso, aprendizagem por reforço é diferente de aprendizagem não supervisionada, que tenta encontrar alguma estrutura escondida em um conjunto de dados sem rótulos. O motivo por isso é porque aprendizagem por reforço não tenta encontrar uma estrutura escondida, ao invés disso, ele tenta maximizar um valor de recompensa.

1.2 Elementos em aprendizagem por reforço

Em qualquer problema de aprendizagem por reforço 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.

Além disso, existem mais quatro subelementos: uma **política**, uma **função de recompensa**, uma **função de valor**, e opcionalmente um **modelo**:

Uma **política** é o que define o comportamento do agente, mapeando cada estado possível do ambiente para uma ação possível que o agente pode efetuar. De modo geral,

uma política pode ser estocástico, especificando uma probabilidade para cada ação que pode ser tomada.

A **função de recompensa** é o que define o objetivo do problema de aprendizagem por reforço. A cada ação tomada pelo agente, o ambiente envia para o agente um valor numérico chamado simplesmente de **recompensa**, e o objetivo final do agente é maximizar a longo prazo o total de recompensa acumulado. Assim, a função de recompensa determina o que é um evento bom e o que é um evento ruim. De modo geral, a função de recompensa pode ser uma função estocástico do estado do ambiente junto com a ação do agente.

Enquanto a função de recompensa representa que ações são boas a curto prazo, a **função de valor** representa que ações são boas a longo prazo, normalmente sendo uma função que mapeia cada estado do ambiente ao total de recompensa esperado no futuro.

Finalmente, um problema de aprendizagem por reforço pode ter um **modelo** do ambiente, que imita as dinâmicas do ambiente, permitindo que inferências sejam feitas sobre como o ambiente agirá após alguma ação do agente. Note que a existência de um modelo não é necessária, pois existem métodos que usam um modelo, e métodos que não usam.

Capítulo 2

Interface agente-ambiente

2.1 Processo de Decisão de Markov

Nessa seção faremos a formalização matemática de um problema de aprendizagem por reforço.

O objetivo de todo problema de aprendizagem por reforço é maximizar a recompensa 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.

Defina $\mathcal{S}$ como o conjunto de todos os estados possíveis no ambiente, e defina $\mathcal{A}$ como o conjunto de todas as ações que o agente pode fazer. 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 quando no estado s. Além disso, defina $\mathcal{R} \subseteq \mathbb{R}$ como o conjunto de possíveis recompensas que o agente pode receber.

Juntando tudo isso, é possível descrever qualquer problema de aprendizagem por reforço como um **Processo de Decisão de Markov** (abreviado como **PDM**), em que os estados possíveis da PDM são todos os estados em $\mathcal{S}$, e as probabilidades do próximo estado e da recompensa obtida são determinadas pela ação em $\mathcal{A}$ que o agente escolheu. Em outras palavras, 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 $A_t \in \mathcal{A}(S_t)$. 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 PDM e o agente criam uma trajetória: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$.

Em um PDM *finito*, temos que $\mathcal{S}$, $\mathcal{A}$ e $\mathcal{R}$ têm tamanhos finitos, e nesse 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 no estado e na ação precedente. Por causa disso, podemos definir:

$$p(s', r|s, a) := \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\},$$

para todo $s, s' \in \mathcal{S}$, $r \in \mathcal{R}$ e $a \in \mathcal{A}(s)$. Ou seja, a função $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ acima

determina toda a *dinâmica* da PDM. 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.

2.2 Recompensa e retorno

Em todo problema de aprendizagem por reforço, a cada passo no tempo $t \in \mathbb{N}$, o agente recebe uma recompensa $R_t \in \mathbb{R}$. 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 acumulativa a longo prazo.

Formalmente, para todo passo no tempo $t \in \mathbb{N}$, queremos maximizar o valor esperado do **retorno**, 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**. Cada episódio termina em um estado especial chamado de **estado terminal**.

Problemas com episódios são chamados de **problemas episódicos**, 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**. Nesses casos, definir G_t como em (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 **taxa de desconto**. Assim, podemos definir o retorno como:

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

A taxa 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 (2.2) converge

Note que dado um problema episódico, é possível representar o retorno (2.1) usando a definição (2.2). Basta fazer a taxa 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.

2.3 Política e função de valor

Uma parte bem importante de múltiplos algoritmos de aprendizagem por reforço é uma **função de valor**, 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 de valor depende do comportamento do agente. Este comportamento do agente é chamado de **política**.

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** 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 PDMs podemos definir como:

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

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** 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:

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

Uma propriedade fundamental das funções de valor usado em vários algoritmos de aprendizagem por reforço 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 de valores 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.

2.4 Política ótima e função de valor ótima

O objetivo final de um problema de aprendizagem por reforço é encontrar alguma política que obtêm bastante recompensa a longo prazo. Em PDMs, 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**, 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**, 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**, 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 de valores, 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**. 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_*:

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

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

Capítulo 3

Dynamic Programming

O método Dynamic Programming (abreviado como DP) refere-se a algoritmos que computam políticas ótimas, assumindo que tenhamos um modelo perfeito das dinâmicas do ambiente no PDM (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 desse modelo perfeito, e também por causa do grande custo computacional, na prática métodos DP têm utilidade limitada, mas a sua teoria é importante como fundamento para outros métodos mas utilizados na prática.

A partir desse capítulo, assumiremos que os ambientes dos problemas possuem um PDM 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 de 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.

3.1 Avaliação de política (previsão)

O processo de computar a função valor-estado v_π dado uma política π é chamado de **avaliação de política**. Fazer isso é um problema bem comum quando lidando com aprendizagem por reforço, e é normalmente referenciado como o **problema de previsão**. 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: $v_0, v_1, v_2, ...$, 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:

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

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**.

3.2 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 aprendizagem por reforço, e este problema é normalmente chamado de **problema de controle**. 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:

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

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*:

> 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:

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

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 de valor da política original, é chamado de **aperfeiçoamento de política**.

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 (3.1), isso significa que:

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

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.

3.3 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**. Segue o pseudocódigo desse algoritmo:

Programa 3.1 Iteração de política.

Parâmetros: $\gamma \in (0, 1]$, **e** $\theta > 0$ (*um número real positivo pequeno, determinando a acurácia da estimação*)
1. *Inicialização*:
    Escolha $V(s) \in \mathbb{R}$ **e** $\pi(s) \in \mathcal{A}(s)$, **para** *todo* $s \in \mathcal{S}$ *arbitrariamente*
    Faça $V(s) = 0$ **para** *todo* $s \in \mathcal{S}$ *que seja terminal*
2. *Avaliação* **de** *política*:
    Faça:
        $\Delta \leftarrow 0$
        Para *cada estado* $s \in \mathcal{S}$ *faça*:
            $v \leftarrow V(s)$
            $V(S) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))(r + \gamma V(s'))$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

cont ⟶

$\longrightarrow$ *cont*

enquanto $\Delta > \theta$

3. *Aperfeiçoamento* **de** *política*:

$politica\_estavel \leftarrow true$

Para *cada* $s \in \mathcal{S}$ *faça*:

$acao\_velha \leftarrow \pi(s)$

$\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$

Se $acao\_velha \neq \pi(s)$ *então* $politica\_estavel \leftarrow false$

Se $politica\_estavel$ *então pare* **e** *retorne* $V \approx v_*$, **e** $\pi \approx \pi_*$; **se** *não, volte* **para** 2.

3.4 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**, e usando-o é possível combinar os passos de avaliação e aprimoramento de política usando a seguinte regra para gerar a próxima função de valor:

$$\begin{aligned} v_{k+1}(s) &:= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a)(r + \gamma v_k(s')), \end{aligned} \tag{3.2}$$

para todo estado $s \in \mathcal{S}$. Para uma função de 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 (3.2) com a equação de Bellman de otimalidade (2.6). Veja que as duas são idênticas, e como a função de valor ótima v_* é um ponto fixo da equação de Bellman de otimalidade, no algoritmo de iteração de valor aplicamos a regra (3.2) até que a função de valor esteja se convergindo, pois assim ela deve estar próxima de v_*:

Programa 3.2 Iteração de valor.

Parâmetros: $\gamma \in (0, 1]$, **e** $\theta > 0$ (*um número real positivo pequeno, determinando a acurácia da estimação*)

Inicialização:

Escolha $V(s) \in \mathbb{R}$, **para** *todo* $s \in \mathcal{S}$ *arbitrariamente*

cont $\longrightarrow$

$\longrightarrow$ *cont*

Faça $V(s) = 0$ **para** *todo* $s \in \mathcal{S}$ *que seja terminal*

Faça:

$\Delta \leftarrow 0$

Para *cada estado* $s \in \mathcal{S}$ *faça*:

$v \leftarrow V(s)$

$V(S) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

enquanto $\Delta > \theta$

Devolva *uma política* $\pi \approx \pi_*$ *tal que* $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$

Capítulo 4

Método Monte Carlos

Como discutimos antes, um problema com Dynamic Programming é que é necessário conhecermos completamente a função p, quando em problemas reais tal informação normalmente não é conhecida. Nesses casos, precisamos de algum método que interage diretamente com o ambiente para obter experiência, assim, podemos usar essa experiência para calcular uma política ótima.

O método Monte Carlos (abreviado para MC) são formas de resolver problemas de aprendizagem por reforço usando as médias dos retornos experienciados até o momento. Como precisamos saber o valor dos retornos, vamos usar os métodos Monte Carlos apenas com problemas episódicos, assim, atualizaremos as estimativas da função de valor e mudaremos a política no final de cada episódio.

4.1 Estimação da função valor-ação

Como 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 método DP. Se ao invés disso soubermos 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 começando no estado s e escolher a ação a, e depois seguir 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 principal em todos os métodos MC.

Mais detalhadamente, caso queremos 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* 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* usa a média de todas as visitas a (s, a).

Programa 4.1 MC primeira-visita, estimar $Q \approx q_*$.

Parâmetros: $\gamma \in (0, 1]$, **e** *uma política* π *que será avaliada.*
Inicialização:
    $Q(s, a) \in \mathbb{R}$ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(s)$
    $Retornos(s, a) \leftarrow$ *uma lista vazia*, **para** *cada* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(()s)$
Loop:
    Gere um episódio seguindo π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Para *cada* $t = T - 1, T - 2, \ldots, 0$ *faça*:
        $G \leftarrow \gamma G + R_{t+1}$
        Se $(S_t, A_t) \notin \{(S_0, A_0), (S_1, A_1), \ldots, (S_{t-1}, A_{t-1})\}$ *faça*:
            Adicione G *em* $Retornos(s, a)$
            $Q(S_t, A_t) \leftarrow$ média$(Retornos(S_t, A_t))$

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 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 começo 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 aprendizagem por reforço de **começo de exploração**.

A suposição de termos começo de exploração é útil, mas na prática é difícil conseguir aplicá-la. Por enquanto assumiremos que temos começo de exploração, e no futuro discutiremos outras alternativas.

4.2 Iteração de política com Monte Carlos

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

$$\pi_0 \xrightarrow{\text{Av}} q_{\pi_0} \xrightarrow{\text{Ap}} \pi_1 \xrightarrow{\text{Av}} q_{\pi_1} \xrightarrow{\text{Ap}} \pi_2 \xrightarrow{\text{Av}} \cdots \xrightarrow{\text{Ap}} \pi_* \xrightarrow{\text{Av}} q_*$$

Já vimos na seção 4.1 como fazer a avaliação de política para estimar a função valoração, 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:

$$\pi(s) = \underset{a}{\operatorname{argmax}}\ q(s, a), \tag{4.1}$$

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 $s \in \mathcal{S}$:

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \underset{a}{\operatorname{argmax}}\ q_{\pi_k}(s, a)) \\ &= \max_a q_{\pi_k}(s, a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s), \end{aligned}$$

então é possível aplicar o *terorema do aperfeiçoamento de política* 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 Carlos, mas uma questão ainda é quantos episódios são necessários para cada etapa de avaliação de política? Similarmente à iteração de valor com DP, 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 Carlos 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:

Programa 4.2 Monte Carlos com começo de exploração, para estimar $\pi \approx \pi_*$.

Parâmetros: $\gamma \in (0, 1]$.
Inicialização:
    $\pi(s) \in \mathcal{A}(s)$ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$
    $Q(s, a) \in \mathbb{R}$ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(s)$
    $Retornos(s, a) \leftarrow$ *uma lista vazia*, **para** *cada* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(()s)$
Loop:
    Escolha $S_0 \in \mathcal{S}$ **e** $A_0 \in \mathcal{A}(S_0)$ *aleatoriamente, tal que todos os pares ocorram com probabilidade* > 0
    Gere um episódio seguindo π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Para *cada* $t = T - 1, T - 2, \ldots, 0$ *faça*:

cont ⟶

$\longrightarrow$ *cont*

$G \leftarrow \gamma G + R_{t+1}$

Se $(S_t, A_t) \notin \{(S_0, A_0), (S_1, A_1), \ldots, (S_{t-1}, A_{t-1})\}$ *faça*:

Adicione G em Retornos(s, a)

$Q(S_t, A_t) \leftarrow$ média$(Retornos(S_t, A_t))$

$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$

4.3 Monte Carlos 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**, 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 $a \in \mathcal{A}(s)$.

Segue o algoritmo Monte Carlos modificado para trabalhar com políticas ε-suaves:

Programa 4.3 Monte Carlos com políticas ε-suaves, para estimar $\pi \approx \pi_*$.

Parâmetros: $\gamma \in (0, 1]$, **e** $\varepsilon > 0$ *pequeno.*

Inicialização:

$\pi \leftarrow$ *uma política* ε*−suave arbitrária*

$Q(s, a) \in \mathbb{R}$ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(s)$

$Retornos(s, a) \leftarrow$ *uma lista vazia*, **para** *cada* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(()s)$

Loop:

Escolha $S_0 \in \mathcal{S}$ **e** $A_0 \in \mathcal{A}(S_0)$ *aleatoriamente, tal que todos os pares ocorram com probabilidade* > 0

Gere um episódio seguindo π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Para *cada* $t = T - 1, T - 2, \ldots, 0$ *faça*:

$G \leftarrow \gamma G + R_{t+1}$

Se $(S_t, A_t) \notin \{(S_0, A_0), (S_1, A_1), \ldots, (S_{t-1}, A_{t-1})\}$ *faça*:

Adicione G em Retornos(s, a)

$Q(S_t, A_t) \leftarrow$ média$(Retornos(S_t, A_t))$

$A' \leftarrow \text{argmax}_a Q(S_t, a)$

cont $\longrightarrow$

$\longrightarrow$ *cont*

Para *cada* $a \in \mathcal{A}(A_t)$ *faça*:
    Se $a = A'$ *faça*:
        $\pi(a|S_t) \leftarrow 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)|$
    Caso *contrário*:
    $\pi(a|S_t) \leftarrow \varepsilon/|\mathcal{A}(S_t)|$

Capítulo 5

Método Temporal-Difference

O método Temporal-Difference (abreviado para TD) pode ser pensado como uma combinação dos métodos Monte Carlos 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 *previsão*, 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 previsão.

5.1 Previsão 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 de 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:

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

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 Carlos, 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**. 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:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)). \tag{5.1}$$

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**.

5.2 Algoritmo Sarsa

Agora que sabemos fazer o problema de previsão 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 Carlos, 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 (5.1) 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:

$$\sum_{t=0}^{\infty} \varepsilon_t = \infty, \text{ e } \sum_{t=0}^{\infty} \varepsilon_t^2 < \infty, \tag{5.2}$$

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:

Programa 5.1 Algoritmo Sarsa, para estimar $Q \approx q_*$.

1 *Parâmetros*: $\gamma \in (0, 1]$, **e** $\alpha \in (0, 1]$, **e** $\varepsilon > 0$ *pequeno.*
2 *Inicialização*:

cont ⟶

$\longrightarrow$ *cont*

Q(*s*, *a*) ∈ ℝ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(s)$
Faça $Q(s, \cdot) = 0$, **para** *todo s terminal.*
Para *cada episódio, faça*:
Inicialize um estado inicial S
Escolha ação A a partir **de** *S usando a política ε−suave gerado com Q*
Para *cada passo no tempo, faça*:
Faça a ação A, **e** *observe a recompensa R* **e** *o próximo estado S′*
Escolha ação A′ a partir **de** *S′ usando a política ε−suave gerado com Q*
$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
$S \leftarrow S'$
$A \leftarrow A'$
Atualize ε seguindo as condições (5.2)
até que S seja terminal

5.3 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 (4.1)), e usar a regra (5.1) 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** 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 (5.1), para que ela seja assim:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)). \tag{5.3}$$

Note que a única diferença entre (5.1) e (5.3) 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:

Programa 5.2 Algoritmo Q-learning, para estimar $Q \approx q_*$.

Parâmetros: $\gamma \in (0, 1]$, **e** $\alpha \in (0, 1]$, **e** $\varepsilon > 0$ *pequeno.*
Inicialização:
Q(*s*, *a*) ∈ ℝ *arbitrariamente*, **para** *todo* $s \in \mathcal{S}$ **e** $a \in \mathcal{A}(s)$
Faça $Q(s, \cdot) = 0$, **para** *todo s terminal.*
Para *cada episódio, faça*:
Inicialize um estado inicial S

cont $\longrightarrow$

$\longrightarrow$ *cont*

Para *cada passo no tempo, faça*:
    Escolha ação A a partir **de** *S usando a política ε−suave gerado com Q*
    Faça a ação A, **e** *observe a recompensa R* **e** *o próximo estado S′*
    $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$
    $S \leftarrow S'$
    Atualize ε seguindo as condições (5.2)
até que S seja terminal

Capítulo 6

Sobre aprendizagem por reforço relacional

A partir desse ponto, começaremos a falar sobre **aprendizagem 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* (Raedt e Bruynooghe, 2004).

6.1 Representação do estado e ação

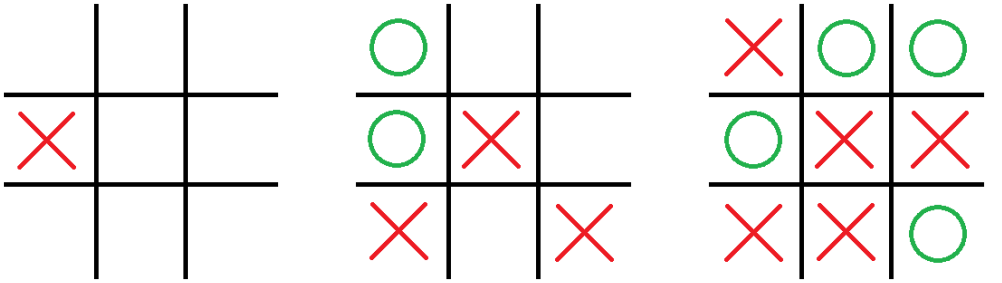
Já discutimos bastante sobre PMD, e um dos elementos importantes de um PMD é o conjunto de estados $\mathcal{S}$ e ações $\mathcal{A}$. Até agora não falamos exatamente como deveríamos definir $\mathcal{S}$ e $\mathcal{A}$ para um dado problema, além de que deve respeitar a propriedade de Markov.

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

6.1.1 Representação proposicional

Esse método 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 6.1: *Exemplo de estados de jogo da velha, e os vetores correspondentes*

Dessa forma, é possível ver que precisaremos lidar com no máximo 3^9 estados se quisermos usar aprendizagem 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, em jogo da velha, para um dos jogadores as ações podem ser representados 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 de 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.

6.1.2 Representação relacional

Nesse método de representação, cada par estado e ação são descritos como 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.

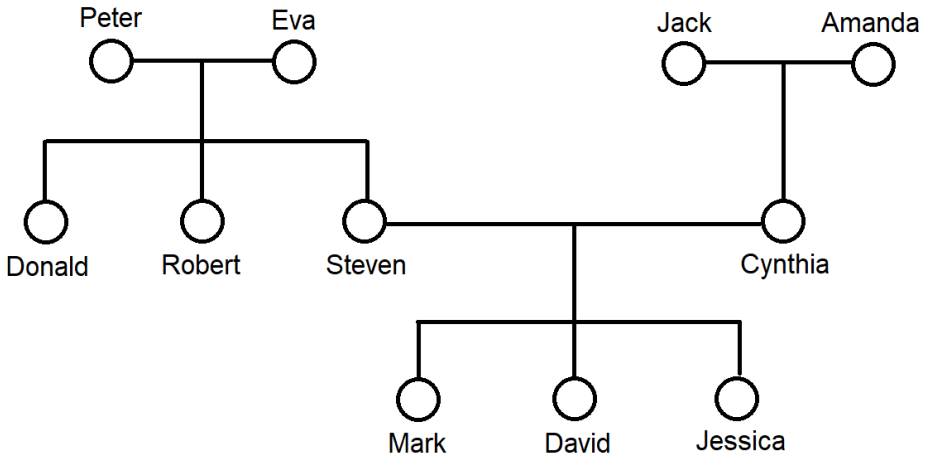


Figura 6.2: *Uma árvore genealógica*

Por exemplo, na Figura 6.2 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.

6.2 Q-Learning Relacional

Q-Learing relacional é bem similar ao Q-Learning regular (Seção 5.3), 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 7 em diante.

6.3 Blocks World

O problema do **Blocks World** é um que usaremos para analisar os algoritmos relacionais nas próximas seções.

Nesse 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.

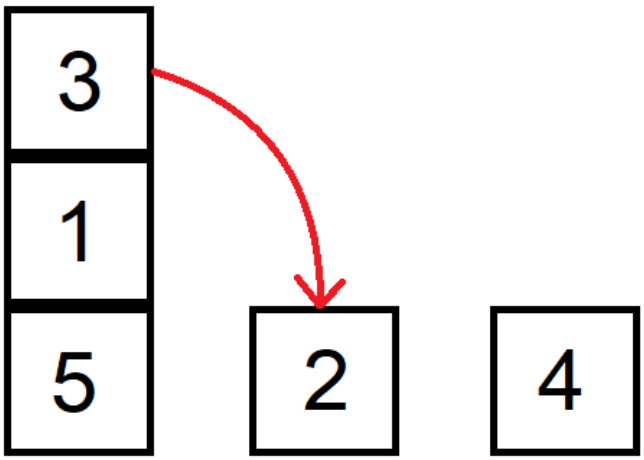


Figura 6.3: *Exemplo de estado e ação em Blocks World*

Na Figura 6.3, temos um exemplo de estado no problema do Blocks World, e a ação que queremos fazer é mover o bloco 3 para acima do bloco 2.

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

- *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 de Blocks World 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 6.3 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 6.3 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 6.3, a ação seria $(3, 2)$.

Sobre o objetivo do Blocks World, vamos ver três objetivos diferentes:

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

2. O objetivo *unstack* 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 *block on block* 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).

Além disso, é possível que o agente escolha uma ação inválida (por exemplo, mover um bloco X para acima de um bloco Y, quando Y não é um bloco livre; ou tentar mover um bloco X para acima do bloco X). Para evitar que essas ações sejam escolhidas muito frequentementes, toda vez que o agente escolher uma ação inválida não alteraremos o estado, e retornaremos uma recompensa de -1.0.

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.

6.4 Q-Learning regular em Blocks World

Para termos um ponto de referência para comparar aprendizagem por reforço relacional com não relacional, vamos primeiro resolver os três objetivos no problema do Blocks World com Q-Learnign regular.

Todos os experimentos nessa seção rodaram o Programa 5.2 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.

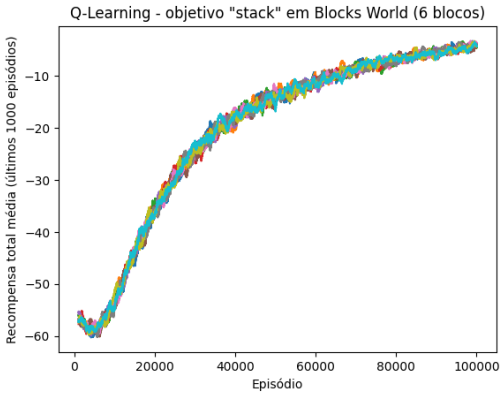
Nos gráficos de recompensa total por episódio abaixo, o eixo X começará no episódio 1000 em vez do episódio 1. Isso é porque, para facilitar a leitura do gráfico, o ponto no

eixo Y para cada episódio será a média das recompensas totais obtidas no episódio no eixo X junto com os 999 episódios anteriores.

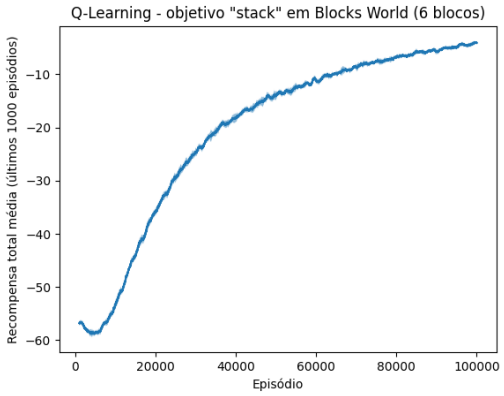
Também mostraremos gráficos demonstrando quanto tempo o algoritmo demorou até terminar a execução do episódio no eixo X. Ou seja, para cada episódio *e*, o ponto no eixo Y marcado será a quantidade de tempo em segundos que demorou para o algoritmo executar os espisódios 1 até o episódio *e*.

6.4.1 Objetivo “stack”

Usando o objetivo *stack* com 6 blocos, o algoritmo Q-Learning gerou os resultados mostrados nas figuras 6.4a e 6.4b.



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



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

Figura 6.4: *Objetivo stack, 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 6.4a tem 10 linhas, e cada cor representa o resultado de uma repetição diferente.

O gráfico 6.4b tem uma linha de cor azul escuro, representando a média das 10 repetições mostradas no gráfico 6.4a. Como cada repetição teve uma trajetória semelhante pode ser difícil de perceber, mas o gráfico 6.4b também mostra o intervalo de confiança de 95% em cada um dos episódios, representados em cor azul claro.

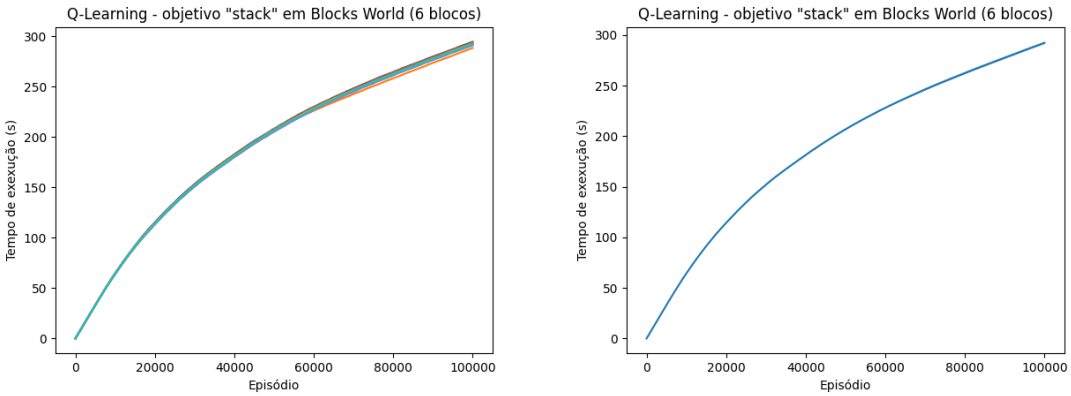
Nessa monografia, todos os gráfico que mostraremos para ver os resultados dos experimentos terão esse mesmo formato: o primeiro gráfico mostrara o resultado para cada repetição do experimento, e o segundo gráfico mostra a média e o intervalo de confiança de 95% das repetições no primeiro gráfico.

Por causa de como definimos as recompensas em Blocks World na seção 6.3, 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 uma performance boa se a recompensa total de um episódio for próxima de 0.0.

Vendo o gráfico 6.4b, mesmo com 100000 episódios de experiência, a média de recompensa por episódio é cerca de −4.14, o que pode indicar que, se assumirmos que uma ação

inválida nunca é escolhida, o agente está precisando de cerca de 51 ações para resolver o objetivo. Também sabemos que Q-Learning no objetivo *stack* é bem consistence, pois a variância das repetições é baixa, como visto no gráfico 6.4b.

Além de analisar a performance dos algoritmos, também vamos analisar a quantidade de tempo real que precisamos deixar o algoritmo rodando. Esses resultados são mostrados nas figuras 6.5a e 6.5b.



(a) *Gráficos das* 10 *repetições do experimento* **(b)** *Média e intervalo de confiança do gráfico 6.5a*

Figura 6.5: *Objetivo stack, gráficos de tempo com Q-Learning*

É possível ver novamente a consistência do algoritmo Q-Learning no objetivo *stack*, com o intervalo de confiança no gráfico 6.5b tão pequeno que mal pode ser visto. Segundo o gráfico, pode ser visto que o tempo de execução para todos os 100000 espisódios é cerca de 4 minutos e 52 segundos.

Sobre os gráficos nas figuras 6.4a e 6.5a, 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.

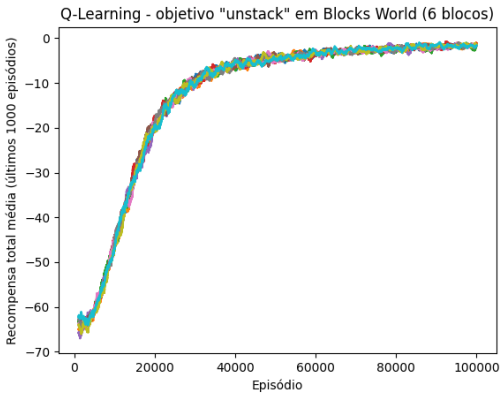
6.4.2 Objetivo “unstack”

Usando o objetivo *unstack* com 6 blocos, o algoritmo Q-Learning gerou os resultados mostrados nas figuras 6.6a e 6.6b.

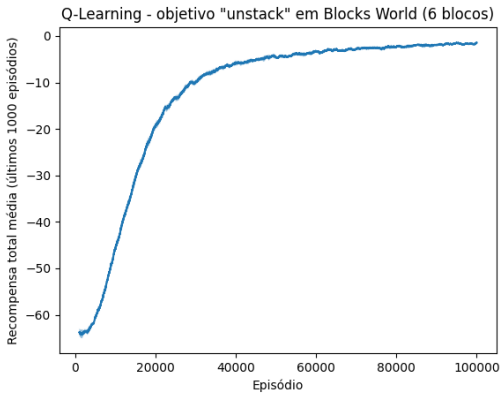
Assim como com o objetivo *stack*, pode ser visto nas figuras 6.6a e 6.6b que Q-Learning é bem consistente quando lidando com o objetivo *unstack*.

Além disso, a performance do agente depois de 100000 episódios é melhor com o objetivo *unstack* comparado com o objetivo *stack*. Após os 100000 episódios, o agente conseguiu uma recompensa total média de cerca de −1.45. Assumindo que o agente não escolhe ação inválida, isso indica que cada episódio o agente faz cerca de 25 ações até resolver o objetivo.

Sobre a análise de tempo de execução, os gráficos que mostram essa informação estão nas figuras 6.7a e 6.7b.

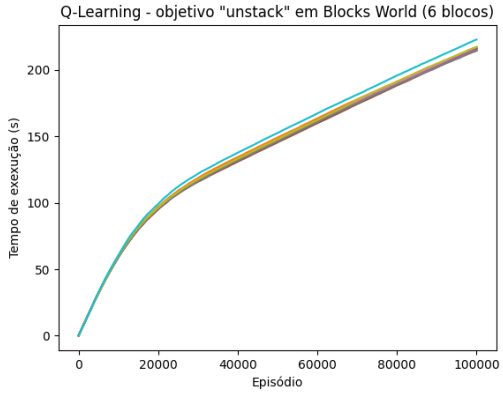


(a) *Gráficos das* 10 *repetições do experimento*

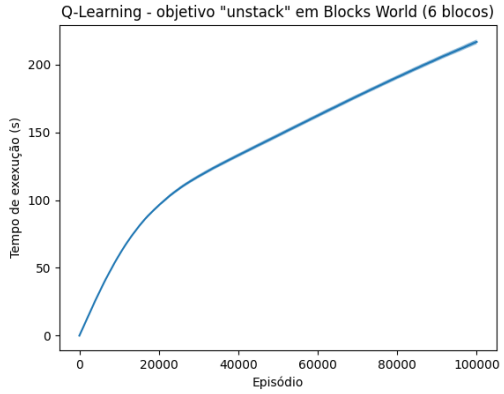


(b) *Média e intervalo de confiança do gráfico 6.6a*

Figura 6.6: *Objetivo unstack, gráficos de recompensa com Q-Learning*



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



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

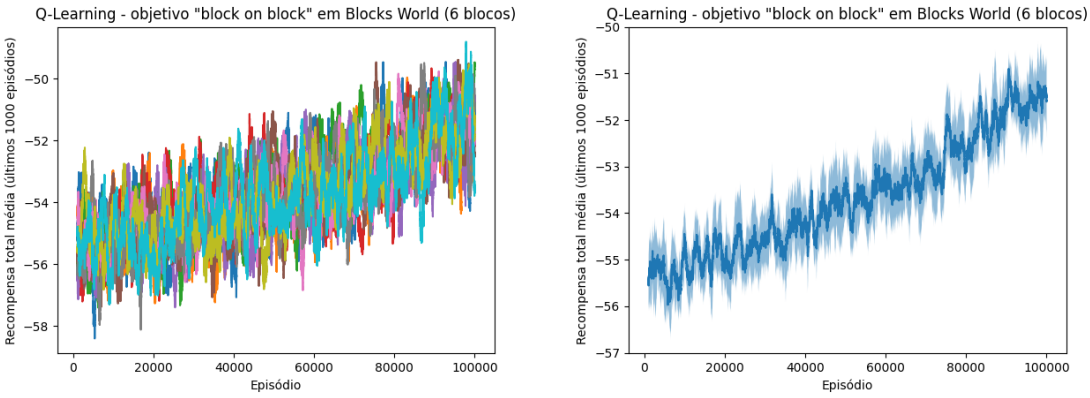
Figura 6.7: *Objetivo unstack, gráficos de tempo com Q-Learning*

É possível ver que o algoritmo Q-Learning executou os 100000 episódios mais rapidamente com o objetivo *unstack* (cerca de 3 minutos e 36 segundos) do que com o objetivo *stack*. Porém, considerando a performance melhor, isso não é inesperado, pois estamos usando menos ações por episódio, logo, cada episódio na média dura menos tempo.

6.4.3 Objetivo “block on block”

Usando o objetivo *block on block* com 6 blocos, o algoritmo Q-Learning gerou os resultados mostrados nas figuras 6.8a e 6.8b.

É fácil ver que há diversas diferenças entre a performance do Q-Learning com o objetivo *block on block* comparado com os outros dois. Primeiro, as repetições do experimento demonstram um nível significativo de variância, visível no gráfico da Figura 6.8b. 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 −51.60. Mesmo chegando no limite de 100 ações por episódio, se o agente evitar todas as ações inválidas a recompensa total seria −10.0. Portanto, é possível concluír que mesmo com toda



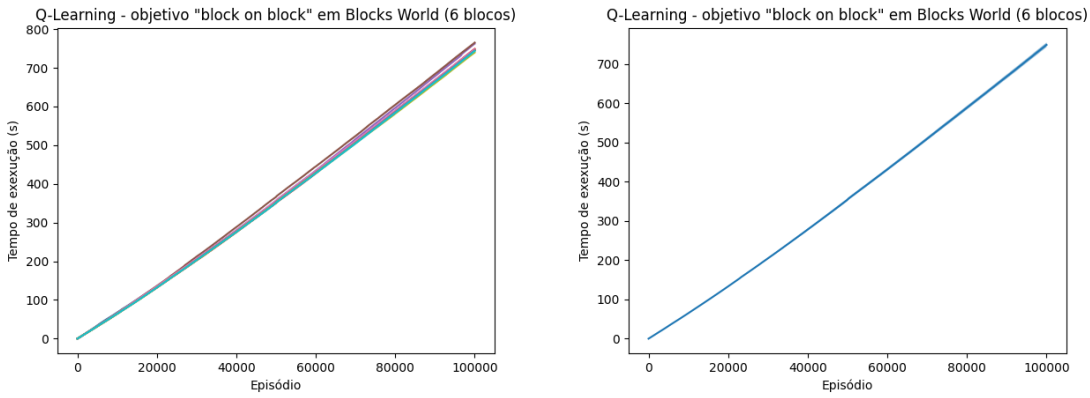
(a) *Gráficos das* 10 *repetições do experimento* **(b)** *Média e intervalo de confiança do gráfico 6.8a*

Figura 6.8: *Objetivo block on block, gráficos de recompensa com Q-Learning*

essa experiência, o agente ainda não conseguiu distinguir ações válidas das inválidas.

Isso tudo sugere que o algoritmo Q-Learning tem uma dificuldade bem maior em resolver o objetivo *block on block* comparado com os outros dois objetivos. Um possível motivo por isso são o que apontamos no final da seção 6.1.1: 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 que determinam quais dois blocos o objetivo *block on block* 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 *block on block*, 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 6.9a e 6.9b.



(a) *Gráficos das* 10 *repetições do experimento* **(b)** *Média e intervalo de confiança do gráfico 6.9a*

Figura 6.9: *Objetivo block on block, gráficos de tempo com Q-Learning*

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 12 minutos e 29 segundos de tempo de

execução para terminar os 100000 episódios. Mas, novamente, isso não é inesperado, pois o fato de que a performance nesse objetivo é bem pior implica que há múltiplos episódios em que o agente chegou no limite de 100 ações por episódio, o que consome mais tempo do que resolver o objetivo antes de cheagar nesse limite.

Capítulo 7

RRL-TG

Na área de aprendizagem 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** de *first-order logical decision tree*) (Blockeel e Raedt, 1998). O algoritmo é chamado **RRL-TG**.

7.1 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:

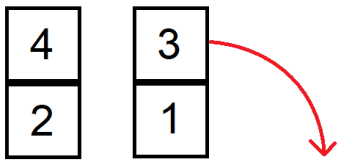
Programa 7.1 Algoritmo FOLDT.

```
1   Parâmetros: 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)
```

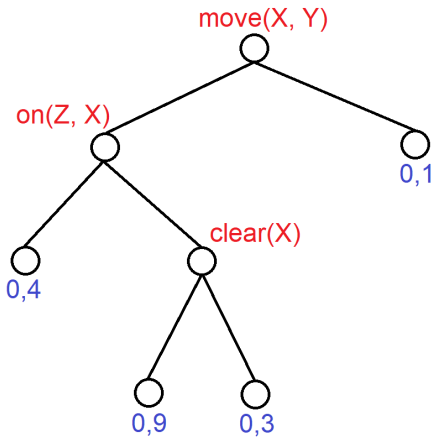
em que $raiz(T)$ é o nó raiz da FOLDT T; $fato(node)$ é o fato relacional no nó interno $node$; $esquerda(node)$ e $direita(node)$ são o nó à esquerda e à direita de $node$, respectivamente; e $valor(node)$ é o valor guardado na folha $node$.

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 $B \supseteq fatos'$.



(a) *Exemplo estado e ação em Blocks World*



(b) *Exemplo de FOLDT*

Figura 7.1: *Exemplo de uso de uma FOLDT para Blocks World*

Um exemplo de como uma FOLDT pode ser usado em um problema de Blocks World está mostrado na Figura 7.1. Usando a representação relacional usado na seção 6.3, o estado e ação mostrado na Figura 7.1a 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 7.1, 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.

7.2 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**.

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 7.1b, 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)*).

7.3 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.

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 7.1, 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 $q_*(s, a)$ 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 $q_*(s, a)$ 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 $q_*(s, a)$ 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 $q_*(s, a)$ para cada um dos n^f pares estado e ação que contaram para a estatística *(1)*.

8. A soma dos quadrados das estimações de $q_*(s, a)$ 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 $q_*(s, a)$ 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:

$$\frac{n_p^{f,r}}{n^f}(\sigma_p^{f,r})^2 + \frac{n_n^{f,r}}{n^f}(\sigma_n^{f,r})^2 \text{ vs. } \sigma_{total}^2, \tag{7.1}$$

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}^2 é o desvio padrão de $q^{f,r}$.

Para determinar quando a diferença na variância é estatísticamente significativa, usaremos o Teste F com nível de significância 0.001 (o que parece bem baixo, mas é suportado pela grande quantidade de exemplos que o algoritmo gera).

Por definição, se $x = (x_1, x_2, \dots, x_n)$ 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 (7.1) para:

$$\frac{n_p^{f,r}}{n^f}\frac{\sum_{i=1}^{n_p^{f,r}}(q_{p,i}^{f,r})^2 - n_p^{f,r}\overline{q_p^{f,r}}}{n_p^{f,r}} + \frac{n_n^{f,r}}{n^f}\frac{\sum_{i=1}^{n_n^{f,r}}(q_{n,i}^{f,r})^2 - n_n^{f,r}\overline{q_n^{f,r}}}{n_n^{f,r}} \text{ vs. } \frac{\sum_{i=1}^{n^f}(q_i^f)^2 - n^f\overline{q^f}}{n^f}. \tag{7.2}$$

Por causa de como o Teste F é calculado, podemos multiplicar ambos lados da comparação em (7.2), 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 (7.2) fica:

$$\sum_{i=1}^{n_p^{f,r}} (q_{p,i}^{f,r})^2 - \frac{1}{n_p^{f,r}} \left(\sum_{i=1}^{n_p^{f,r}} q_{p,i}^{f,r} \right)^2 + \sum_{i=1}^{n_n^{f,r}} (q_{n,i}^{f,r})^2 - \frac{1}{n_n^{f,r}} \left(\sum_{i=1}^{n_n^{f,r}} q_{n,i}^{f,r} \right)^2 \text{ vs. } \sum_{i=1}^{n^f} (q_i^f)^2 - \frac{1}{n^f} \left(\sum_{i=1}^{n^f} q_i^f \right)^2 . \tag{7.3}$$

A parte importante de (7.3) 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 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).

7.4 Algoritmo RRL-TG

Com tudo isso, podemos finalmente apresentar o algoritmo RRL-TG:

Programa 7.2 Algoritmo RRL-TG, para estimar $Q \approx q_*$.

```
1   Parâmetros:
2       γ ∈ (0, 1]
3       α ∈ (0, 1]
4       ε > 0 pequeno
5       T, uma FOLDT inicial
6       m, número mínimo de exemplos para uma folha poder ser dividida
7
8   Inicialize os rmodes
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 a FOLDT T
13          Faça a ação A e observe a recompensa R e o próximo estado S′
14          q ← o que Programa 7.1 retorna com FOLDT T e fatos relacionais (S, A)
15          Q′ ← lista vazia
16          Para cada a ∈ A(S′) faça:
17              Q′ ← Q′ ∪ { o que Programa 7.1 retorna com FOLDT T e fatos
                    relacionais (S′, a) }
18          q′ ← q + α(R + γ max Q′ − q)
19          Seja f a folha que o Programa 7.1 chega com FOLDT T e fatos relacionais
                (S, A)
20          Atualize as estatísticas na folha f com (S, A) e q′
21          Se número de exemplos em f for ≥ m e o Teste F indicar que a folha pode ser
                dividida:
```

cont ⟶

$\longrightarrow$ *cont*

22 *Gere um nó interno com o candidato* **para** *fato relacional que melhor sucediu o Teste f*

23 *Gere duas folhas, com estatísticas herdados das estatísticas em f*

24 **Caso** *contrário*:

25 *Atualize o valor retornado por f em T* **para** $\overline{q^f}$

26 $S \leftarrow S'$

27 *Até que S seja terminal*

Uma parte que não discutimos ainda é a linha 23 do algoritmo acima. Essa linha simplesmente indica que 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.

7.5 RRL-TG em Blocks World

Para cada um dos experimentos a seguir, rodamos o Programa 7.2 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 6.4, 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 6.3 que na forma como fazemos a representação relacional do problema do Blocks World, usamos fatos relacionais do formato *on(X, Y)*, *clear(X)* e *move(X, Y)*. Também, no objetivo *block on block* também 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.

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

- *rmode(clear(+X))*;
- *rmode(on(+-X, +-#Y))*;

- *rmode(move(+-X, +-#Y))*;
- *rmode(above(+-X, +-Y))*,

em que a única constante é *floor*.

Note que nenhum os rmodes que usamos tem um limite para a quantidade que pode ser usada na FOLDT.

7.5.1 Objetivo “stack”

Para o objetivo *stack*, usaremos a FOLDT mostrada na Figura 7.2 como árvore inicial do Programa 7.2.

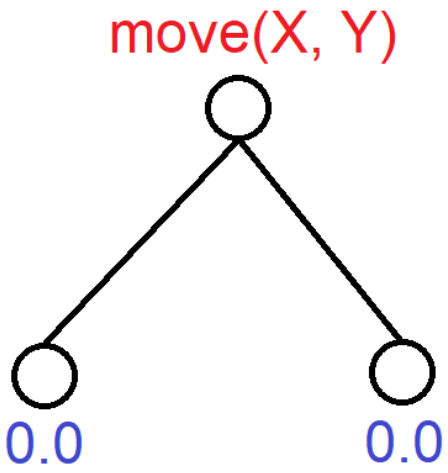
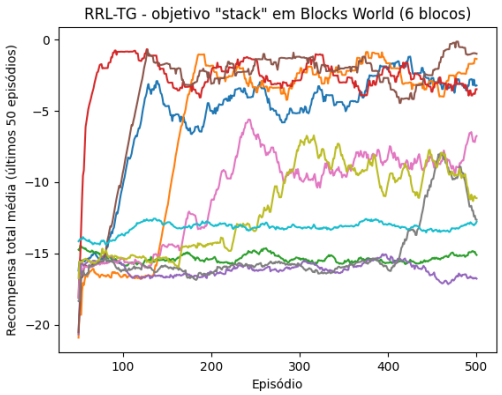
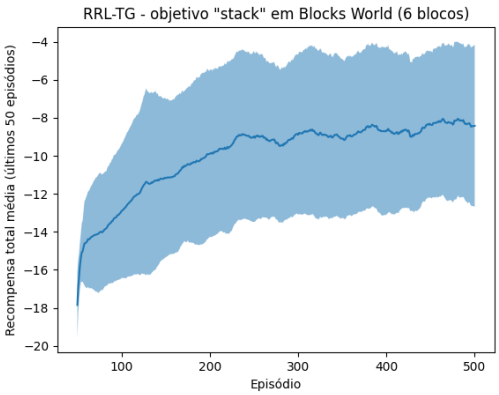


Figura 7.2: *FOLDT inicial para o problema stack*

Usando o algoritmo RRL-TG no objetivo *stack* com 6 blocos, os seguintes resultados obtidos são mostrados nos gráficos das figuras 7.3a e 7.3b.



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



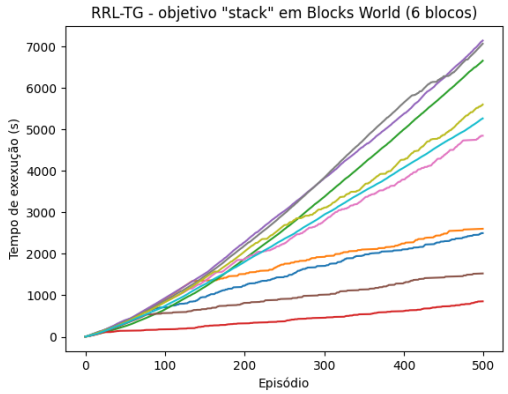
(b) *Média e intervalo de confiança do gráfico 7.3a*

Figura 7.3: *Objetivo stack, gráficos de recompensa com RRL-TG*

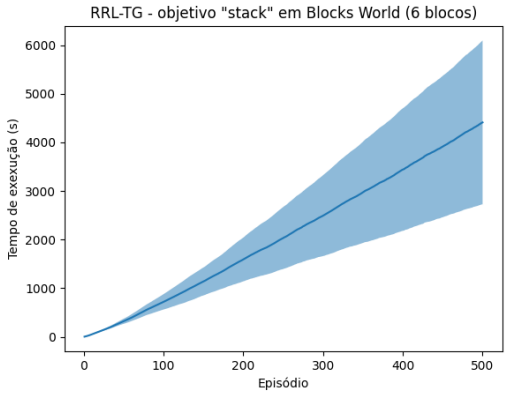
Observando o gráfico na Figura 7.3a, é possível perceber uma grande variação na performance do algoritmo. Essa variância pode ser vista claramente na Figura 7.3b, em que, após 500 episódios, a média da recompensa total das 10 repetições foi cerca de −8.42, mas o intervalo de confiança de 95% é varia de cerca de −12.67 até −4.17.

Em geral, comparado com Q-Learning a performance final é pior, porém há uma diferença bem grande na quantiadde de episódios que cada algoritmo foi treinado. Os resultados demosntram que o algoritmo RRL-TG precisou de bem menos episódios para aprender comparado com Q-Learning.

Sobre a análise de tempo de execução do algoritmo, os gráficos que mostram essa informação estão nas Figuras 7.4a e 7.4b.



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



(b) *Média e intervalo de confiança do gráfico 7.4a*

Figura 7.4: *Objetivo stack, gráficos de tempo com RRL-TG*

Comparando os gráficos nas Figuras 7.3a e 7.4a, de modo geral é possível ver que as repetições que tiveram uma performance melhor também têm um tempo de execução menor. Como a preformance tem uma variação grande, o tempo de execução tembém tem essa variação. De fato, a repetição com o menor tempo de execução demorou cerca de 14 minutos e 15 segundos, enquanto a repetição com o maior tempo de execução demorou cerca de 1 hora, 59 minutos e 9 segundos.

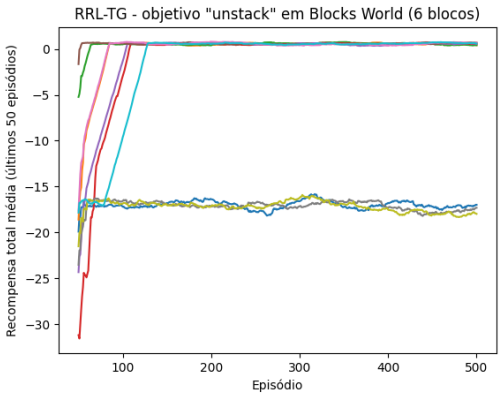
Assim, comparado com Q-Learning, o tempo de execução do RRL-TG é claramente maior, principalmente considerando o fato de que treinamos o agente com Q-Learning em 200 vezes mais episódios do que com RRL-TG. Porém, a magnitude de quão maior o tempo de execução é tem uma grande variação.

7.5.2 Objetivo “unstack”

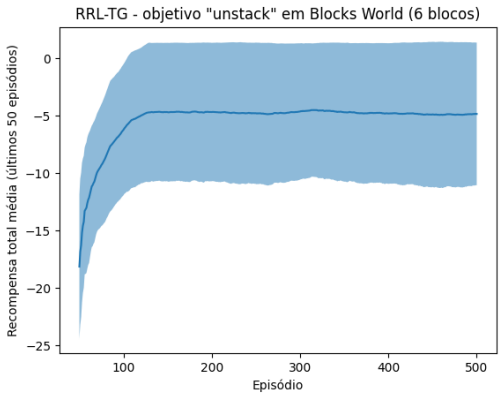
Para o objetivo *unstack*, a FOLDT inicial que usamos como árvore inicial do Programa 7.2 doi a mesma que usamos para o objetivo *stack*, ou seja, a FOLDT na Figura 7.2.

Executando o algoritmo RRL-TG com o objetivo *unstack* e com 6 blocos, os resultados obtidos são mostrados nas Figuras 7.5a e 7.5b.

Vendo o gráfico na Figura 7.5a, é fácil separar cada repetição desse experimento em dois grupos: os que terminaram com recompensa total médio perto de 0.0, e os que terminaram



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



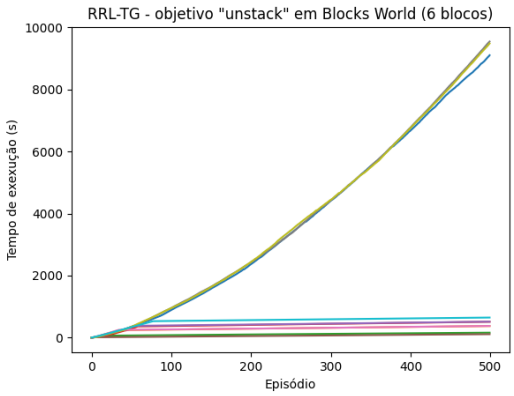
(b) *Média e intervalo de confiança do gráfico 7.5a*

Figura 7.5: *Objetivo unstack, gráficos de recompensa com RRL-TG*

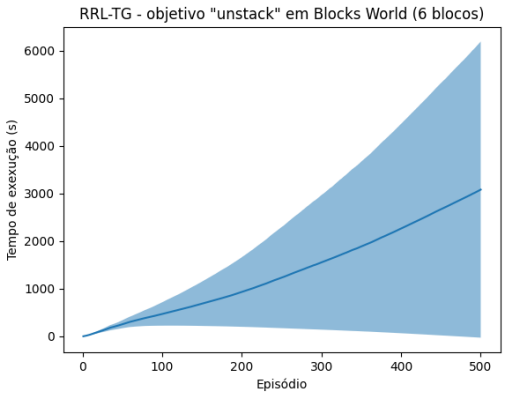
com recompensa total médio perto de −17.0. Por causa disso, o intervalo de confiança no gráfico da Figura 7.5b pode enganar a distribuição real, pois faz parecer que qualquer valor no intervalo pode acontecer, quando na realidade são os dois extremos do intervalo que aconteceram nas repetições.

Comparado com Q-Learning, a performance do RRL-TG nesse problema pode ser melhor ou pior, dependendo de qual dos dois grups uma execução acaba sendo. Se for do grupo com recompensa total média final perto de 0.0, então a performance final é semelhante ao agente treinado com Q-Learning, as a quantidade de episódios usados foi bem menor. Se for do grupo com recompensa total média final perto de −17.0, então a performance final do Q-Learning é claramente melhor.

Agora, a análise de tempo de execução pode ser visto nas Figuras 7.6a e 7.6b.



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



(b) *Média e intervalo de confiança do gráfico 7.6a*

Figura 7.6: *Objetivo unstack, gráficos de tempo com RRL-TG*

Novamente, olhando o gráfico na Figura 7.6a, é fácil dividir cada repetição desse experimento nos mesmos grupos que fizemos com o gráfico na Figura 7.5a. Isso significa que há uma grande diferença entre os tempos de execução entre os dois grupos. De fato, entre as 10 repetições, o menor tempo de execução foi cerca de 1 minuto e 52 segundos, enquanto

o maior tempo de execução foi cerca de 2 horas, 39 minutos e 4 segundos. Por causa disso, de novo o intervalo de confiança no gráfico da Figura 7.6b é meio enganador, pois é apenas os extremos do intervalo que foram observados nas repetições que foram feitas.

Comparado com Q-Learnign, nesse objetivo o algoritmo RRL-TG pode ser melhor ou pior em relação ao tempo de execução, dependendo de qual dos dois grupos uma execução do algoritmo RRL-TG nesse objetivo acaba sendo.

7.5.3 Objetivo "block on block"

Para o objetivo *block on block*, usaremos a FOLDT mostrada na Figura 7.7 como árvore inicial do Programa 7.2.

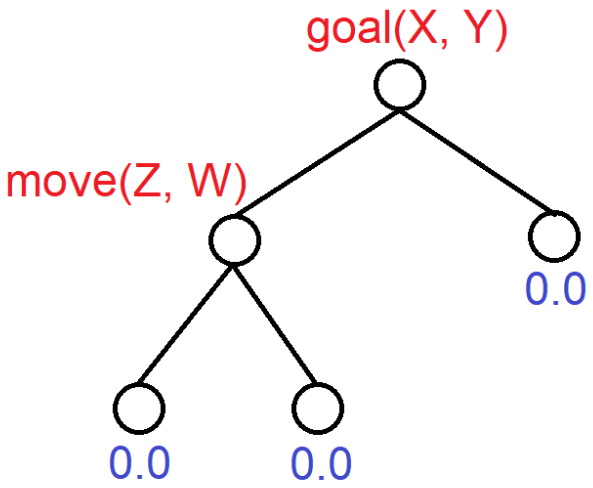
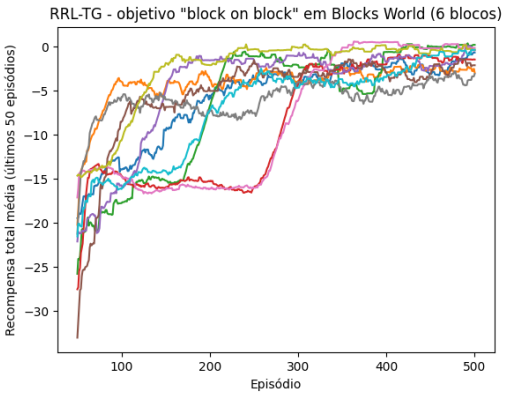
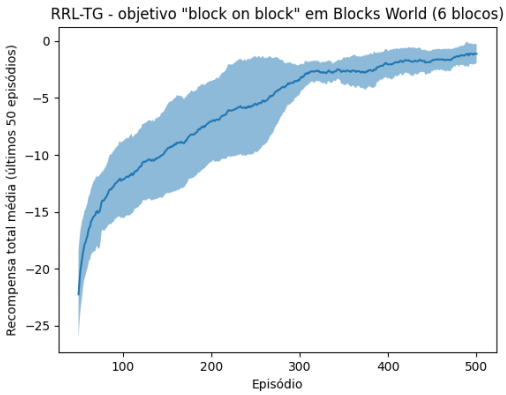


Figura 7.7: *FOLDT inicial para o problema block on block*

Rodando o algoritmo RRL-TG para o problema *block on block* com 6 blocos, os resultados foram obtidos são mostrados nas Figuras 7.8a e 7.8b.



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



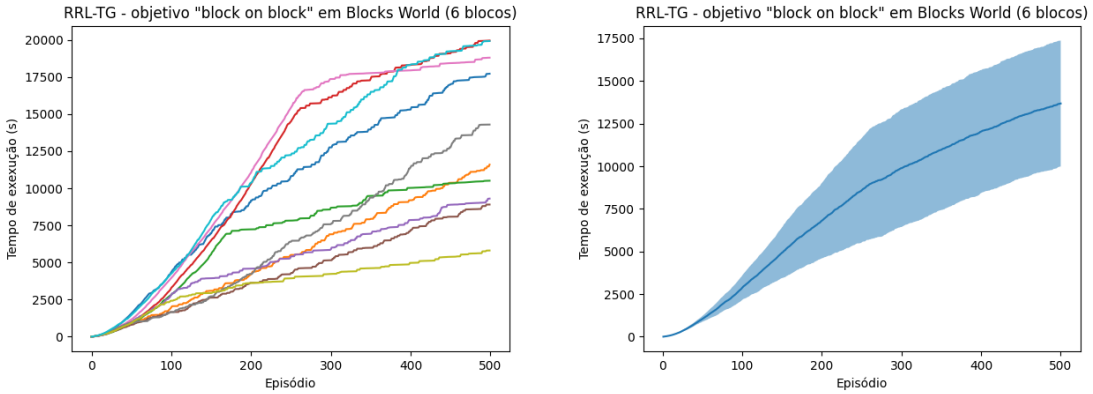
(b) *Média e intervalo de confiança do gráfico 7.8a*

Figura 7.8: *Objetivo block on block, gráficos de recompensa com RRL-TG*

Os resultados vistos na Figura 7.8a demonstram que, no começo, há uma alta variação nas performances de cada repetição, mas após cerca de 300 episódios, todas repetições convergeram para uma recompensa total média alta. Esssa análise é visível no intervalo de confiança do gráfico da Figura 7.8b, em que o tamanho do intervalo reduz depois de cerca do episódio 300.

Comparado com o algoritmo Q-Learning, a performance do RRL-TG nesse objetivo é claramente melhor, necessitando de bem menos episódios para alcançar performance melhor.

Sobre a quantidade de tempo que o algoritmo demorou para executar, essa informação está representado nos gráficos das Figuras 7.9a e 7.9b.



(a) *Gráficos das* 10 *repetições do experimento* **(b)** *Média e intervalo de confiança do gráfico 7.9a*

Figura 7.9: *Objetivo block on block, gráficos de tempo com RRL-TG*

Diferente do 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 3 horas, 48 minutos e 7 segundos. Entre essas 10 repetições, o menor tempo de execução foi cerca de 1 hora, 36 minutos e 51 segundos, enquanto o maior tempo de execução foi cerca de 5 horas, 32 minutos e 36 segundos.

Comparado com Q-Learning, nesse objetivo o algoritmo RRL-TG tem uma óbvia desvantagem no tempo de execução por episódio, principalmente considerando que treinamos o agente com RRL-TG em bem menos episódios do que com Q-Learning.

Capítulo 8

RRL-RIB

O algoritmo RRL-RIB usa uma estratégia chamado de *aprendizagem baseada em instâncias*, que é conhecido por ser simples e com boa performance. A estratégia de aprendizagem baseada em instâncias guarda um conjunto de exemplos, e compara os exemplos nesse conjunto com um novo exemplo para estimar o valor que queremos. Para fazer essa comparação, algum tipo de medida de distância entre exemplos precisa ser definida.

8.1 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 relacinoal*.

Nessa seção, focaremos especificamente na definição de uma distância relacional no problema de Blocks World. 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, dependendo do objetivo, também um fato relacional *goal(Z1, W1)*. Para cada bloco em {*X1, Y1*} (e se existir *goal(Z1, W1)*, 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 possivelment um fato relacional *goal(Z2, W2)*). Tente dar rótulos para os blocos em {*X2, Y2*} (e se existir *goal(Z2, W2)*, 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** 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) ou (S_2, A_2) tiver mais pilhas do que o outro), cada pilha não pareada aumentará a distância relacional por uma constante real k_3.
8. Retorne a distância relacional acumulada até agora.

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. 37).

[Raedt e Bruynooghe 2004] Luc De Raedt e Maurice Bruynooghe. "Relational Reinforcement Learning". Tese de dout. Leuven, Bélgica: Universidade Católica de Lovaina, mai. de 2004 (citado na pg. 25).

[Sutton e Barto 2015] Richard S. Sutton e Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2ª ed. The MIT Press, 2015 (citado na pg. 1).

Índice remissivo