CENTRO UNIVERSITÁRIO FEI

THIAGO FREITAS DOS SANTOS

HEURÍSTICAS E SOLUÇÃO AUTOMÁTICA EFICIENTE PARA PROBLEMAS ESPACIAIS: HoASP(MDP) e quebra-cabeça espacial

São Bernardo do Campo
2019

THIAGO FREITAS DOS SANTOS

HEURÍSTICAS E SOLUÇÃO AUTOMÁTICA EFICIENTE PARA PROBLEMAS ESPACIAIS: HoASP(MDP) e quebra-cabeça espacial

Dissertação de Mestrado apresentada ao Centro Universitário FEI para obtenção do título de Mestre em Engenharia Elétrica. Orientado pelo Prof. Dr. Paulo Eduardo Santos.

São Bernardo do Campo

2019

Santos, Thiago Freitas dos.
Heurísticas e solução automática eficiente para problemas espaciais: HoASP (MDP) e quebra-cabeça espacial / Thiago Freitas dos Santos. São Bernardo do Campo, 2019.
101 f. : il.

Dissertação - Centro Universitário FEI.
Orientador: Prof. Dr. Paulo Eduardo Santos.

1. Heurística. 2. Aprendizado por Reforço. 3. Answer Set Programming. 4. Processo de Decisão de Markov. 5. Quebra-Cabeças Espaciais. I. Santos, Paulo Eduardo, orient. II. Título.

Elaborada pelo sistema de geração automática de ficha catalográfica da FEI com os dados fornecidos pelo(a) autor(a).

centro universitário FEI

APRESENTAÇÃO DE DISSERTAÇÃO
ATA DA BANCA EXAMINADORA

Programa de Pós-Graduação Stricto Sensu em Engenharia Elétrica

Mestrado

PGE-10

Aluno: Thiago Freitas dos Santos **Matrícula:** 117125-5

Título do Trabalho: Heurísticas e solução automática eficiente para problemas espaciais: HoASP (MDP) e quebra-cabeça espacial.

Área de Concentração: Inteligência Artificial Aplicada à Automação e Robótica

Orientador: Prof. Dr. Paulo Eduardo Santos

Data da realização da defesa: 28/02/2019

ORIGINAL ASSINADA

Avaliação da Banca Examinadora:

São Bernardo do Campo, / / .

MEMBROS DA BANCA EXAMINADORA	
Prof. Dr. Paulo Eduardo Santos	Ass.:
Prof. Dr. Reinaldo Augusto da Costa Bianchi	Ass.:
Prof. Dr. Marcelo Finger	Ass.:

A Banca Julgadora acima-assinada atribuiu ao aluno o seguinte resultado:

APROVADO ☒ REPROVADO ☐

VERSÃO FINAL DA DISSERTAÇÃO

APROVO A VERSÃO FINAL DA DISSERTAÇÃO EM QUE FORAM INCLUÍDAS AS RECOMENDAÇÕES DA BANCA EXAMINADORA

Aprovação do Coordenador do Programa de Pós-graduação

Prof. Dr. Carlos Eduardo Thomaz

AGRADECIMENTOS

Agradeço principalmente à minha família pelo suporte não apenas durante todo o projeto de mestrado, mas também durante toda minha vida.

Agradeço também ao meu orientador Professor Doutor Paulo Eduardo Santos por acreditar no meu trabalho e ajudar no processo de desenvolvimento desse mestrado.

Agradeço ao Centro Universitário FEI e aos professores que foram essenciais para a minha formação.

Agradecimento em especial ao Doutor Leonardo Anjoletto Ferreira, Professor Doutor Reinaldo Bianchi e Professor Doutor Pedro Cabalar por auxiliarem neste trabalho.

Por fim, agradeço à FAPESP, processo nº 2017/07833-9, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), pela bolsa de estudos, permitindo que me dedicasse exclusivamente para concluir essa dissertação.

“Não há ensino sem pesquisa e pesquisa sem ensino [...] Enquanto ensino continuo buscando, reprocurando. Ensino porque busco, porque indaguei, porque indago e me indago. Pesquiso para constatar, constatando, intervenho, intervindo educo e me educo. Pesquiso para conhecer o que ainda não conheço e comunicar ou anunciar a novidade.”
(FREIRE, P., 1996)

RESUMO

Quebra-Cabeças Espaciais formados por objetos rígidos, cordas flexíveis e buracos oferecem domínios interessantes que lidam com elementos espaciais comuns no cenário onde as experiências humanas diárias acontecem. Por essa razão, esses quebra-cabeças são a base de estudos dessa dissertação, já que encontrar uma solução envolvendo problemas espaciais é um dos itens essenciais para um agente computacional inteligente interagir em um ambiente humano. Portanto, este trabalho tem como objetivo o estudo de uma solução automática para um conjunto de problemas espaciais, especificamente os quebra-cabeças de Fisherman's Folly e Rope Ladder. Esse objetivo é atingido por meio da adaptação do algoritmo oASP(MDP), em que há a utilização de heurísticas com o intuito de acelerar o processo de aprendizado e desenvolver o algoritmo Heurística oASP(MDP) (HoASP(MDP)). No HoASP(MDP), Answer Set Programming (ASP) é utilizado para representar o domínio como um Processo de Decisão de Markov (MDP), enquanto um algoritmo de Aprendizado por Reforço (AR) (Q-Learning) é usado para encontrar políticas ótimas desse MDP. A metodologia deste trabalho, para obtenção de heurísticas, baseia-se em uma estratégia incremental, onde começa-se a resolver os quebra-cabeças a partir de um conjunto muito restritivo de limitações, relaxando-as gradualmente enquanto as soluções de cada versão relaxada são utilizadas como heurísticas para quebra-cabeças mais complexos. Experimentos foram realizados em diferentes versões dos quebra-cabeças, como determinístico, não-determinístico e não-estacionário. Os resultados mostram que a abordagem proposta (HoASP(MDP)) pode acelerar o processo de aprendizado, apresentando assim uma vantagem quando comparado com as versões dos algoritmos oASP(MDP) e Q-Learning que não utilizam heurísticas.

Palavras-chave: Heurística. Aprendizado por reforço. Answer Set Programming. Processo de Decisão de Markov. Quebra-cabeças espaciais.

ABSTRACT

Spatial puzzles composed of rigid objects, flexible strings and holes offer interesting domains for reasoning about spatial entities that are common in the human daily-life's activities. For this reason, these puzzles are the base of study in this dissertation, since finding a solution that deals with spatial problems is one of the essential items for a computational agent to interact in a human environment. Thus, the goal of this work is to investigate the automated solution of this kind of puzzles, specifically the Fisherman's Folly and the Rope Ladder puzzles, adapting an algorithm that combines Answer Set Programming (ASP) with Markov Decision Process (MDP), algorithm oASP(MDP). The aim of this adaptation is to use heuristics to accelerate the learning process and to build the Heuristic oASP(MDP) (HoASP(MDP)) algorithm. In HoASP(MDP), ASP is applied to represent the domain as an MDP, while a Reinforcement Learning algorithm (Q-Learning) is used to find the optimal policies. In this work, the heuristics were obtained from the solution of relaxed versions of the puzzles, using an incremental approach, in which the agent starts to solve the puzzles with a set of restrictive limitations, relaxing them gradually while using the solutions for the relaxed versions as heuristics for the more complex puzzles. Experiments were performed on deterministic, non-deterministic and non-stationary versions of the puzzles. Results show that the proposed approach (HoASP(MDP)) can accelerate the learning process, presenting an advantage when compared to the non-heuristic versions of oASP(MDP) and Q-Learning.

Keywords: Heuristic. Reinforcement learning. Answer Set Programming. Markov Decision Process. Spatial puzzles.

LISTA DE ILUSTRAÇÕES

Ilustração 1 – Estados iniciais dos Quebra-Cabeças Espaciais estudados nesse trabalho . 16
Ilustração 2 – Estados iniciais e estados objetivos dos Quebra-Cabeças Espaciais estudados nesse trabalho 20
Ilustração 3 – Sequência de ações possíveis para resolver o Quebra-Cabeça de Fisherman's Folly 21
Ilustração 4 – Um programa em Answer Set Programming (ASP) 25
Ilustração 5 – Solução para o programa ASP da figura 4 25
Ilustração 6 – Um programa em ASP 27
Ilustração 7 – O programa, representado na figura 6, depois do processo de *grounding* . 27
Ilustração 8 – Diagrama apresentando a interação por um episódio do processo de aprendizado entre o Agente HoASP(MDP) e o Ambiente com Oráculo (Planejador PROLOG) 37
Ilustração 9 – Número de Passos e Valores de Retorno para o Fisherman's Folly Original 48
Ilustração 10 – Resultados de Estados para o Fisherman's Folly Original 49
Ilustração 11 – Testes Estatísticos para o Fisherman's Folly Original 50
Ilustração 12 – Número de Passos e Valores de Retorno para o Fisherman's Folly Simplificado 52
Ilustração 13 – Resultados de Estados para o Fisherman's Folly Simplificado 53
Ilustração 14 – Número de Passos e Valores de Retorno para o Fisherman's Folly Original Não-Determinístico 55
Ilustração 15 – Resultados de Estados para o Fisherman's Folly Não-Determinístico . . . 56
Ilustração 16 – Testes Estatísticos para o Fisherman's Folly Original Não-Determinístico 57
Ilustração 17 – Número de Passos e Valores de Retorno para o Fisherman's Folly Simplificado Não-Determinístico 59
Ilustração 18 – Resultados de Estados para o Quebra-Cabeça Fisherman's Folly Simplificado Não-Determinístico 60
Ilustração 19 – Número de Passos e Valores de Retorno para o Fisherman's Folly Esfera Não-Estacionáiro 62
Ilustração 20 – Resultados de Estados para o Fisherman's Folly Esfera Não-Estacionário . 63
Ilustração 21 – Número de Passos e Valores de Retorno para o Fisherman's Folly Disco Não-Estacionário 65
Ilustração 22 – Resultados de Estados para o Fisherman's Folly Disco Não-Estacionário . 66
Ilustração 23 – Testes Estatísticos para o Fisherman's Folly Disco Não-Estacionário . . . 67
Ilustração 24 – Número de Passos e Valores de Retorno para o Rope Ladder 69
Ilustração 25 – Resultados de Estados para o Rope Ladder 70
Ilustração 26 – Testes Estatísticos para o Rope Ladder 71

Ilustração 27 – Código utilizado para execução de um experimento. Os resultados apresentados nessa dissertação realizam 30 experimentos 91
Ilustração 28 – Parte do código utilizado para execução de uma determinada ação 92
Ilustração 29 – Parte do código utilizado atualização dos valores entre Python e ASP . . . 93
Ilustração 30 – Código utilizado para definir os parâmetros dos experimentos 94
Ilustração 31 – Código utilizado para atualizar o valor do estado/ação na Tabela Q 95
Ilustração 32 – Código utilizado para atualizar a descrição da tabela Q com informações do ASP 95
Ilustração 33 – Código utilizado para criar o arquivo ASP com as Ações Proibidas 96
Ilustração 34 – Código utilizado para criar o arquivo ASP com os pares Estado/Ações proibidos 96
Ilustração 35 – Código utilizado para criar o arquivo ASP com os Estados Objetivos . . . 97

LISTA DE TABELAS

Tabela 1 – Número aproximado de interações necessárias para o agente começar a aprender a política ótima para os quebra-cabeças estacionários 72
Tabela 2 – Número aproximado de interações necessárias para o agente começar a aprender a política ótima para os quebra-cabeças não estacionários 72
Tabela 3 – Número de Estados Visitados pelo agente durante a interação com o ambiente para os quebra-cabeças estacionários 72
Tabela 4 – Número de Estados Visitados pelo agente durante a interação com o ambiente para os quebra-cabeças não estacionários 73
Tabela 5 – Número de pares Estado/Ação presentes na Tabela Q para os quebra-cabeças estacionários 73
Tabela 6 – Número de pares Estado/Ação presentes na Tabela Q para os quebra-cabeças não estacionários 73
Tabela 7 – Tempo de execução médio, em minutos, para um experimento completo (6000 episódios) 101

LISTA DE ALGORITMOS

Algoritmo 1 – oASP(MDP) 33
Algoritmo 2 – HoASP(MDP) 36

LISTA DE ABREVIATURAS

AM	Aprendizado de Máquina.
AR	Aprendizado por Reforço.
ASP	Answer Set Programming.
FEI	Centro Universitário FEI.
HARL	Aprendizado por Reforço Acelerado por Heurística.
MDP	Processo de Decisão de Markov.
NMR	Raciocínio Não-Monotônico.
oASP(MDP)	Online Answer Set Programming para Processo de Decisão de Markov.
RNA	Rede Neural Artificial.
TD	Aprendizado de Diferença Temporal.

LISTA DE SÍMBOLOS

α Taxa de aprendizado do algoritmo Q-Learning.

A Conjunto de ações.

a_t Ação no instante t.

β Potência de influência da heurística.

ϵ Fator de influência na estratégia de escolha de ação do algoritmo Q-Learning.

S Conjunto de estados.

s_t Estado no instante t.

γ Fator de desconto do algoritmo Q-Learning.

R Função de Recompensa.

T Função de transição.

ξ Fator de influência da heurística.

SUMÁRIO

1 INTRODUÇÃO 15
2 REVISÃO TEÓRICA 19
2.1 O DOMÍNIO 19
2.1.1 Fisherman's Folly Original (FFO) 22
2.1.2 Fisherman's Folly Simplificado (FFS) 22
2.1.3 Fisherman's Folly Não-Determinístico (FFND) 22
2.1.4 Fisherman's Folly Simplificado Não-Determinístico (FFSND) 22
2.1.5 Fisherman's Folly Esfera Não-Estacionário (FFENE) 23
2.1.6 Fisherman's Folly Disco Não-Estacionário (FFDNE) 23
2.1.7 Rope Ladder 23
2.2 ANSWER SET PROGRAMING (ASP) 24
2.3 APRENDIZADO POR REFORÇO (AR) 27
2.4 ONLINE ANSWER SET PROGRAMING PARA PROCESSO DE DECISÃO DE MARKOV (oASP(MDP)) 31
3 HEURÍSTICA E oASP(MDP) 35
3.1 HoASP(MDP) PARA RESOLVER QUEBRA-CABEÇAS ESPACIAIS 38
4 EXPERIMENTOS 42
4.1 FISHERMAN'S FOLLY ORIGINAL (FFO) 42
4.2 FISHERMAN'S FOLLY SIMPLIFICADO (FFS) 43
4.3 FISHERMAN'S FOLLY NÃO-DETERMINÍSTICO (FFND) 43
4.4 FISHERMAN'S FOLLY SIMPLIFICADO NÃO-DETERMINÍSTICO (FFSND) 43
4.5 FISHERMAN'S FOLLY ESFERA NÃO-ESTACIONÁRIO (FFENE) 44
4.6 FISHERMAN'S FOLLY DISCO NÃO-ESTACIONÁRIO (FFDNE) 44
4.7 ROPE LADDER 45
5 RESULTADOS E DISCUSSÃO 46
5.1 FISHERMAN'S FOLLY ORIGINAL (FFO) 46
5.2 FISHERMAN'S FOLLY SIMPLIFICADO (FFS) 51
5.3 FISHERMAN'S FOLLY NÃO-DETERMINÍSTICO (FFND) 54
5.4 FISHERMAN'S FOLLY SIMPLIFICADO NÃO-DETERMINÍSTICO (FFSND) 58
5.5 FISHERMAN'S FOLLY ESFERA NÃO-ESTACIONÁRIO (FFENE) 61
5.6 FISHERMAN'S FOLLY DISCO NÃO-ESTACIONÁRIO (FFDNE) 64
5.7 ROPE LADDER 68
6 REVISÃO BIBLIOGRÁFICA 75
7 CONCLUSÃO 82
7.1 TRABALHOS FUTUROS 82
REFERÊNCIAS 84
ÍNDICE 88

APÊNDICE A – CÓDIGOS UTILIZADOS PARA OS EXPERIMENTOS 90
APÊNDICE B – CONFIGURAÇÃO PARA EXECUÇÃO DOS EXPERIMENTOS 98
APÊNDICE C – TEMPO DE EXECUÇÃO DOS EXPERIMENTOS . . . 100

1 INTRODUÇÃO

A capacidade de aprender uma sequência de ações para solucionar tarefas complexas, a partir de interações com o domínio, é uma habilidade essencial para qualquer agente inteligente presente no mundo físico. Isso é particularmente importante quando se lida com domínios que contém elementos espaciais, como objetos rígidos, flexíveis e até mesmo com buracos, em que as ações e seus efeitos não são triviais. Por essa razão, o principal desafio considerado neste trabalho é o aprendizado por meio de interações de um conjunto de ações que são necessárias para solucionar problemas espaciais que contam com esses elementos. Portanto, para lidar com esse desafio é considerado especificamente os quebra-cabeças espaciais compostos de objetos rígidos, cordas flexíveis e buracos. Já que esses são elementos de interesse presentes não apenas no mundo físico em que os humanos estão inseridos, mas também em áreas importantes de aplicação, como cirurgia robótica e manutenção de maquinário, em que elementos com características distintas (ou até mesmo contrastantes) devem ser manipulados com cuidado para que um certo objetivo seja atingido (podendo ser a remoção de um tumor, ou a reparação de uma peça quebrada em uma determinada máquina).

Os quebra-cabeças de Fisherman's Folly e Rope Ladder, mostrados nas figuras 1a e 1b respectivamente, são os domínios de estudo considerados por este trabalho. Sendo que basicamente são formados por: Postes, Corda, Anel, um par de Esferas (que são atravessadas pela Corda) e um par de Discos (fixados nas extremidades da Corda). Com o intuito de facilitar o entendimento desses quebra-cabeças, uma descrição do estado inicial do Fisherman's Folly se torna relevante, por exemplo, o estado inicial do domínio pode ser aquele em que o Poste atravessa o buraco do Anel; a Corda atravessa o buraco do Poste; cada Disco e Esfera está de um lado do Poste (figura 1a). Para solucionar esse quebra-cabeça é necessário executar uma sequência de ações que tem como objetivo liberar o Anel das relações que este tem com os outros elementos. Vale lembrar que essas ações devem ser não destrutivas, excluindo por exemplo cortar a Corda ou quebrar o Poste. Com a execução de ações e mudanças nos estados dos quebra-cabeças, o relacionamento entre os elementos que formam o quebra-cabeça também muda com o tempo, considerando que o relacionamento depende da interação em um determinado momento, pois a execução de uma ação, por exemplo passar o Disco pelo Buraco do Poste, altera a forma como os elementos (Disco e Poste) estão relacionados. Além disso, esses domínios têm na formalização dos estados outro elemento desafiador, já que o número de estados visitados pode crescer arbitrariamente, dado que as relações entre os elementos aparecem e desaparecerem inúmeras vezes (por exemplo enrolar e desenrolar a Corda pelo Buraco do Poste).

Considerando as características dos quebra-cabeças descritas acima, a escolha desses domínios dá-se pelo fato de representarem bons exemplos para testes, já que mantém a complexidade presente em problemas que lidam com elementos espaciais comuns do mundo físico, além de permitir a representação de estados como um Processo de Decisão de Markov (MDP).

Figura 1 – Estados iniciais dos Quebra-Cabeças Espaciais estudados nesse trabalho

(a) Quebra-cabeça de Fisherman's Folly

(b) Quebra-cabeça Rope Ladder

Fonte: Autor

Sendo assim, vale destacar como pode ser complexo, devido à necessidade de formalização dos quebra-cabeças e do processo de interação entre um agente e tais quebra-cabeças, o desenvolvimento de um solucionador automático que seja capaz de lidar com essas características (ressaltando também quebra-cabeças com diferentes componentes e relações) e ainda aprender uma sequência de ações que deve ser executada para solucionar um quebra-cabeça específico.

Trabalhos anteriores lidaram com soluções automáticas para esse tipo de domínio, mas partindo de uma perspectiva em lógica formal (CABALAR; SANTOS, 2011, 2016; SANTOS; CABALAR, 2016), em que as ações eram claramente representadas, permitindo que um planejador fosse construído a fim de encontrar a solução para um conjunto de quebra-cabeças espaciais (CABALAR; SANTOS, 2011). Porém, essas abordagens anteriores não lidavam com o aprendizado de tarefas e não podiam ser aplicados em versões dos quebra-cabeças mais complexas, que permitissem mudanças no domínio[1] (não-estacionário) e com não-determinismo. Sendo assim, essas soluções não foram usadas como base para comparação com o algoritmo proposto nessa dissertação, já que contavam com um escopo de aplicação mais restrito (apenas em domínios determinísticos e no quebra-cabeça Fisherman's Folly).

Portanto, para resolver os problemas deixados em abertos por trabalhos anteriores, essa dissertação teve como foco a aplicação da combinação de Answer Set Programming (ASP) e MDP, o algoritmo oASP(MDP) (seção 2.4), para resolver os quebra-cabeças de Fisherman's Folly e Rope Ladder (seção 2.1). ASP é utilizado para representar o domínio como um MDP, enquanto um algoritmo de Aprendizado por Reforço (AR), especificamente Q-Learning, é utili-

[1] São domínios em que mudanças no ambiente, enquanto o agente está interagindo, podem acontecer. É a contrapartida ao domínio estacionário, aquele em que o ambiente se mantém o mesmo durante toda a interação.

zado para encontrar uma política ótima para esse MDP. Com isso, a contribuição principal dessa dissertação é o desenvolvimento do algoritmo Heurística oASP(MDP) (HoASP(MDP)), que é adaptação do algoritmo oASP(MDP) incluindo o uso de heurísticas para acelerar o processo de aprendizado. As heurísticas foram obtidas por meio da solução de versões simplificadas dos quebra-cabeças (capítulo 3) e facilitam o processo de aprendizado, servindo como um guia para o algoritmo de AR no momento da escolha da ação que deve ser executada pelo agente, sendo que essa abordagem foi definida por Bianchi, Ribeiro e Costa (2008) como Aprendizado por Reforço Acelerado por Heurística (HARL). As heurísticas obtidas foram aplicadas não só para as versões determinísticas e não-determinísticas desses domínios, como também para versões em que a Corda pôde ser enrolada através dos Buracos.

O primeiro dos conceitos que formam o algoritmo Online Answer Set Programming para Processo de Decisão de Markov (oASP(MDP)) é o MDP, um formalismo utilizado para descrever problemas de tomada de decisão sequencial, representados por um tupla <S, A, T, R> que contém informações sobre os estados (S), ações (A), função de transição (T) e recompensa (R) de um domínio. É importante pontuar uma característica do MDP que é a capacidade de representar experiências passadas de uma maneira compacta, mas sem perder informações relevantes (SUTTON; BARTO, 2018). Depois que um domínio é formalizado como um MDP, não sendo necessário descrever previamente as funções T e R, é possível encontrar uma solução para este domínio por meio do uso de uma técnica de Aprendizado de Máquina (AM) conhecida como AR. AR é um modelo em que a solução de um problema é encontrada através de interações entre um agente (capaz de executar uma ação) e o ambiente (descrição do domínio, local onde as ações são executadas). O algoritmo específico que foi utilizado por este trabalho é o Q-Learning, que tem como objetivo aproximar uma função valor-ação Q(s,a), que é usada para escolher uma ação $a \in A$ que deverá ser executada quando o agente estiver em um estado $s \in S$. Sendo assim, Q(s,a) acaba sendo a função que determina qual é a sequência ótima de ações a ser executada pelo agente (SUTTON; BARTO, 2018).

O segundo conceito utilizado pelo algoritmo oASP(MDP) é a linguagem de programação declarativa ASP. ASP é baseada em programação lógica não-monotônica (EITER; IANNI; KRENNWALLNER, 2009) e tem como objetivo auxiliar na representação de um domínio que conta com características dinâmicas, além de ser efetiva para solução de problemas NP-Completos, como o problema do Caixeiro Viajante (HOFFMAN; PADBERG; RINALDI, s.d.). Sendo assim, ASP permite que um algoritmo de AR possa aprender a política ótima inclusive quando há mudanças no ambiente enquanto as interações acontecem. A regra de escolha é uma outra ferramenta da linguagem, usando essas regras é possível gerar diferentes saídas dada uma mesma entrada, ou seja, dado que o agente está em um estado $s0$ e executa uma ação a é possível, com ASP, descrever diversos estados futuros como consequência dessa execução, por exemplo os estados $s1$ e $s2$. Além dessa capacidade, programas ASP também podem ser usados para representar as restrições do domínio, como ações e estados proibidos.

Depois da apresentação dos conceitos que formam o algoritmo oASP(MDP), é possível descrever como este algoritmo funciona combinando ASP e MDP. O oASP(MDP) permite a construção e atualização do conjunto de estados visitados S e das ações A que podem ser executadas de uma maneira *online*, enquanto as interações acontecem. A grande vantagem de se utilizar ASP nesse contexto é que mesmo que os conjuntos S e A sofram mudanças enquanto o agente interage com o ambiente, um algoritmo de AR ainda é capaz de aproximar a função valor-ação Q(s, a), bastando que inicialmente o algoritmo receba o conjunto de ações A, um método que aproxima a função Q(s, a) e o número interações que devem ser realizadas. Ferreira et al. (2018) mostram como o algoritmo oASP(MDP) apresenta uma melhor performance quando comparado com os algoritmos padrões de AR como Q-Learning e SARSA, em outros domínios, como o domínio do mundo de grades.

Com o intuito de avaliar a abordagem proposta neste trabalho, o algoritmo HoASP(MDP), experimentos (capítulo 4) foram realizados nas versões determinísticas, não-determinísticas e não-estacionárias dos quebra-cabeças especiais. Com isso, os resultados e a discussão (capítulo 5) mostraram que essa abordagem apresenta uma melhor performance que as versões sem heurísticas dos algoritmos oASP(MDP) e Q-Learning.

Os próximos capítulos deste trabalho estão organizados da seguinte maneira: o capítulo 2 faz uma apresentação dos conceitos que serão utilizados para o desenvolvimento deste trabalho. Já o capítulo 3 apresenta o método desenvolvido por este trabalho. O capítulo 4 descreve os experimentos e o capítulo 5 apresenta os resultados obtidos e a discussão desses resultados. O capítulo 6 faz uma revisão bibliográfica dos trabalhos desenvolvidos na área, tanto relacionados ao tema geral quanto relacionados ao algoritmo oASP(MDP). Por fim, o capítulo 7 conclui o trabalho e apresenta direções futuras para este trabalho.

2 REVISÃO TEÓRICA

Este capítulo apresenta uma descrição teórica dos conceitos que foram utilizados neste trabalho. Começando pela descrição dos quebra-cabeças explorados, então partindo para a linguagem lógica (ASP) usada para representar um MDP e a técnica de AR usada para encontrar uma solução ótima para esse MDP. Por fim, o algoritmo oASP(MDP) é descrito.

2.1 O DOMÍNIO

Como este trabalho procura uma solução aplicável a problemas espaciais, dois domínios que apresentam as características espaciais que se quer estudar (como objetos com buracos ou objetivos flexíveis) são os quebra-cabeças de Fisherman's Folly (figura 2a) e Rope Ladder (figura 2c).

Os dois quebra-cabeças considerados por este trabalho são formados pelo seguinte conjunto de elementos: Poste com buraco; uma Corda; um Anel; um par de Esferas, que são atravessadas pela corda; e um par de Discos, cada um está fixado em uma extremidade diferente da Corda. Com relação à dinâmica e às possibilidades de movimentações dos elementos, há a possibilidade de mover as Esferas pela Corda. Além disso, a Corda passa pelo Buraco do Poste, fazendo com que cada Esfera e cada Disco fique de um lado do Poste. Um outro ponto importante a se notar é que as Esferas são maiores que os Buracos dos Postes, impossibilitando assim a passagem destas através deste buraco. Já os Discos podem atravessar o Buraco do Poste, porém não podem atravessar o Buraco do Anel. Outro elemento que pode passar pelo Buraco do Poste é o Anel.

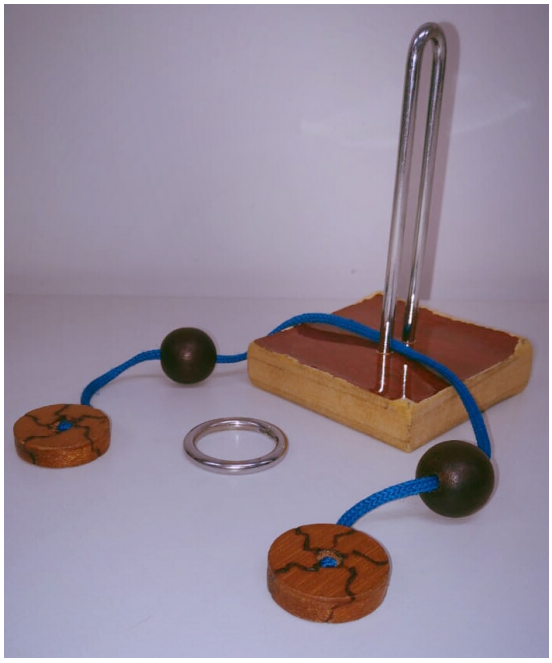
O objetivo desses quebra-cabeças é liberar o Anel do conjunto de elementos (figura 2b e figura 2d). Para que se possa atingir este objetivo, um agente (uma pessoa, um robô, etc), que participe da interação com esses quebra-cabeças, deve executar uma sequência de ações. Nesse contexto, uma ação é a manipulação de dois objetos e a face de um buraco, por exemplo passar o Disco através do Buraco do Poste da direita para à esquerda. Essas interações entre os elementos dos quebra-cabeças podem levar à situações complexas, como a possibilidade de enrolar, múltiplas vezes, a Corda no Buraco do Anel ou no Buraco do Poste. Esse fenômeno aumenta o Espaço de Estados e consequentemente o tempo necessário para encontrar a solução (o impacto dessa possibilidade de enrolar a Corda pelos buracos pode ser visto no capítulo 5). A proposta apresentada neste trabalho também lida com essa situação.

Apesar desses dois quebra-cabeças possuírem similaridades, existem também diferenças fundamentais entre eles: o Rope Ladder é mais complexo que o Fisherman's Folly, já que o primeiro tem dois Postes (conectados por uma barra no topo) com Buracos; o Rope Ladder não tem uma base, o que permite a passagem dos Postes pelo Buraco do Anel em qualquer direção (direita ou esquerda); por fim, o Rope Ladder apresenta uma relação mais complexa entre a

Figura 2 – Estados iniciais e estados objetivos dos Quebra-Cabeças Espaciais estudados nesse trabalho

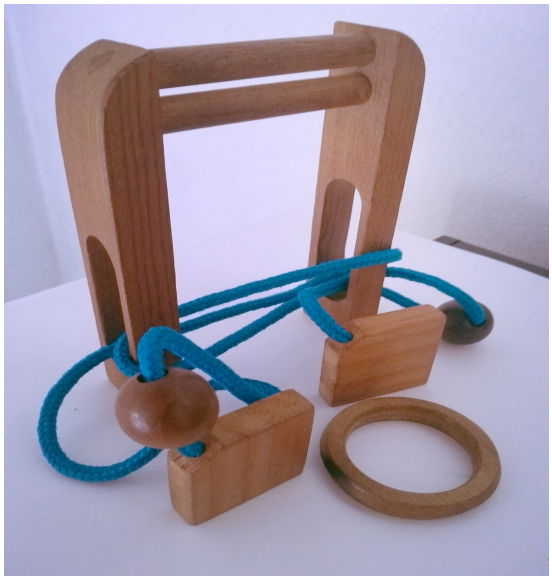
(a) Quebra-cabeça de Fisherman's Folly

(b) Estado objetivo do Fisherman's Folly



(c) Quebra-cabeça Rope Ladder

(d) Estado objetivo do Rope Ladder

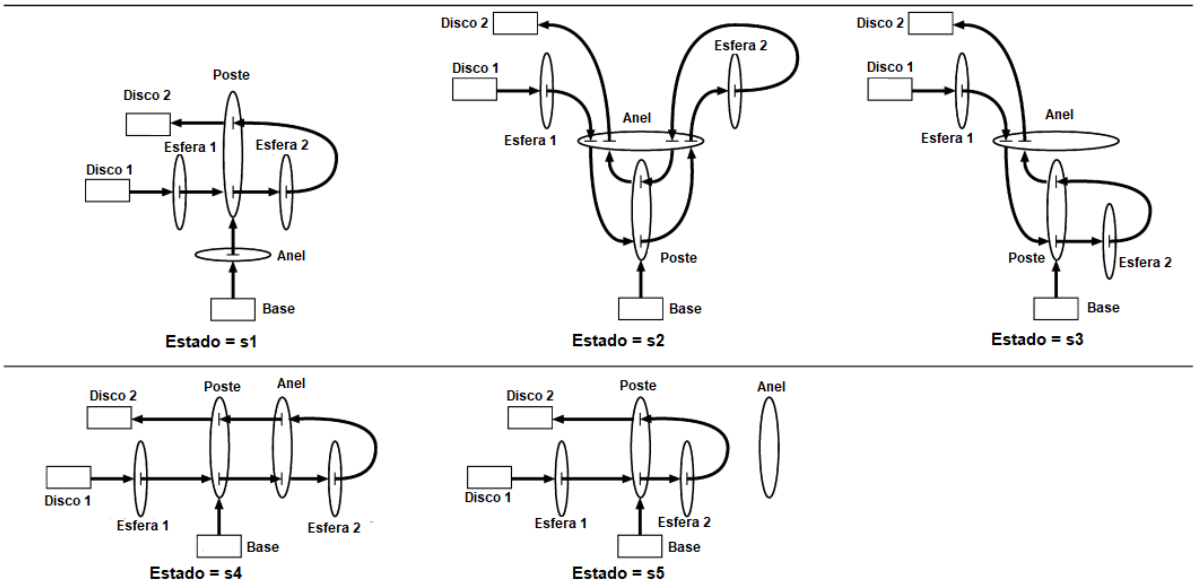


Fonte: Autor

Corda e os Postes, já que a Corda atravessa os Buracos dos Postes em quatro diferentes pontos (figura 2c).

Os elementos que formam esses quebra-cabeças podem ser separados em três diferentes categorias como apresentado por Cabalar e Santos (2011). A primeira categoria é a Regular, que representa objetos simples, como discos; a segunda categoria é a de elementos que contém Buracos, como anéis; e por fim a categoria de elementos Longos, como cordas.

Figura 3 – Sequência de ações possíveis para resolver o Quebra-Cabeça de Fisherman's Folly



Fonte: Cabalar e Santos (2011)

a) categoria 1 (objetos regulares):
   - Disco1;
   - Disco2;
   - Base;

b) categoria 2 (objetos com buracos):
   - Anel;
   - Esfera1;
   - Esfera2;
   - Buraco do Poste1;
   - Buraco do Poste2;

c) categoria 3 (objetos longos):
   - Corda;
   - Poste1;
   - Poste2.

Já que a partir do estado inicial (figura 2a e 2c) uma sequência de ações deve ser executada com o objetivo de liberar o Anel do quebra-cabeça, a figura 3 apresenta uma possível sequência de ações que pode ser aplicada no quebra-cabeça de Fisherman's Folly para esse fim. Considerando $s0$ como o estado inicial, a aplicação da ação "Passar Disco2 pelo Poste" é realizada e então o agente avança para o estado $s1$. Depois disso, a execução da ação "Tirar Anel do Poste" é realizada, e o agente passa para o estado $s2$. Agora em $s2$, a ação "Passar Esfera2 pelo Anel" é executada, levando o agente para o estado $s3$. A próxima ação escolhida é "Passar Anel

pelo Buraco do Poste", como resultado chega-se ao estado $s4$. Finalmente, quando o estado $s4$ é visitado, a ação "Passar Esfera2 pelo Anel" é executada e o estado objetivo $s5$ é alcançado, concluindo assim a tarefa de liberar o Anel do quebra-cabeça de Fisherman's Folly (figura 2b).

Para verificar como o agente autônomo construído nessa dissertação pôde ser usado para resolver esses quebra-cabeças, foram realizados experimentos (capítulo 4) com sete configurações diferentes do Fisherman's Folly e Rope Ladder, apresentadas a seguir.

2.1.1 Fisherman's Folly Original (FFO)

Essa é a configuração original do quebra-cabeça de Fisherman's Folly (figura 2a), sem nenhuma alteração nos elementos e nem nas relações entre eles.

2.1.2 Fisherman's Folly Simplificado (FFS)

Nessa configuração (FFS), os elementos que formam o quebra-cabeça são os mesmos daqueles que formam o Fisherman's Folly Original (FFO), a diferença é que na configuração simplificada o agente não pode enrolar a Corda através do Buraco do Poste (único elemento com essa restrição).

2.1.3 Fisherman's Folly Não-Determinístico (FFND)

Nessa configuração (FFND), os elementos e as relações são os mesmos que formam o quebra-cabeça Fisherman's Folly Original (FFO), mas agora considerando o não-determinismo na execução de ações. A diferença dessa configuração para as configurações anteriores dos quebra-cabeças é que quando um agente escolhe uma ação para ser executada no ambiente, há uma chance de 80% dessa ação alcançar o resultado esperado, uma chance de 10% da ação ser executada com os mesmos elementos, mas em uma direção oposta e uma chance de 10% do agente executar nenhuma ação (ficando no mesmo estado). As ações ótimas que levam à solução deste quebra-cabeça são as mesmas do Fisherman's Folly Original (FFO)

2.1.4 Fisherman's Folly Simplificado Não-Determinístico (FFSND)

Essa é a configuração simplificada do Fisherman's Folly Não-Determinístico (FFND), sendo que nesse quebra-cabeça não é possível enrolar a Corda através do Buraco do Poste (único elemento com essa restrição).

2.1.5 Fisherman's Folly Esfera Não-Estacionário (FFENE)

Nesta configuração (FFENE) é permitido que aconteça uma mudança no quebra-cabeça em um determinado momento (a ser escolhido pelo projetista e conforme os requisitos dos experimentos, detalhes na seção 4.5). Sendo que antes da mudança ocorrer o quebra-cabeça permite que a Esfera1 e a Esfera2 passem através do Buraco do Poste. Depois da mudança, o quebra-cabeça passa a ter a mesma configuração e simplificações do quebra-cabeça Fisherman's Folly Simplificado (FFS).

2.1.6 Fisherman's Folly Disco Não-Estacionário (FFDNE)

Nesta configuração (FFDNE) também é permitido que aconteça uma mudança no quebra-cabeça em um determinado momento. Sendo que antes da mudança ocorrer o quebra-cabeça FFDNE permite que o Disco1 e o Disco2 possam atravessar o Buraco do Anel, mas não possam atravessar o Buraco do Poste. Após a mudança, o quebra-cabeça FFDNE passa a ter a mesma configuração e simplificações que o Fisherman's Folly Simplificado (FFS).

2.1.7 Rope Ladder

Essa é a configuração original do quebra-cabeça Rope Ladder (figura 2c), sem nenhuma alteração nos elementos e nas relações entre eles.

Para além das definições dos quebra-cabeças estudados nessa dissertação, é possível também apresentar como pode-se separar essas configurações em quatro categorias distintas:

a) categoria 1 (quebra-cabeças originais):
   - Fisherman's Folly Original (FFO);
   - Rope Ladder;

b) categoria 2 (quebra-cabeças simplificados)
   - Fisherman's Folly Simplificado (FFS);

c) categoria 3 (quebra-cabeças não-determinísticos)
   - Fisherman's Folly Não-Determinístico (FFND);
   - Fisherman's Folly Simplificado Não-Determinístico (FFSND);

d) categoria 4 (quebra-cabeças não-estacionários)
   - Fisherman's Folly Esfera Não-Estacionário (FFENE);
   - Fisherman's Folly Disco Não-Estacionário (FFDNE).

Por fim, como o objetivo deste trabalho é construir um agente autônomo capaz de lidar com esses quebra-cabeças, foi necessário encontrar um modo que permitisse a interação entre o agente (o algoritmo desenvolvido por essa dissertação) e o quebra-cabeça. Com esse intuito,

esse trabalho utilizou um Oráculo (uma simulação do quebra-cabeça) desenvolvido por Cabalar e Santos (2011), que é um planejador escrito na linguagem PROLOG. A necessidade de um Oráculo surge porque não se tem um agente físico capaz de executar ações neste quebra-cabeça no mundo real. A interação entre o agente e o ambiente (representado pelo Oráculo) acontece através de perguntas e respostas, já que o simulador permite realizar perguntas sobre um estado futuro quando uma determinada ação é executada. Por exemplo, o agente pergunta ao planejador PROLOG qual é o resultado de executar uma ação em um determinado estado, então o Oráculo tem a possibilidade de responder com duas informações diferentes: ou o estado futuro (resultado da execução da ação) ou que aquela ação não pode ser executada naquele instante. Apesar desse planejador PROLOG poder ser usado para solucionar o quebra-cabeça de Fisherman's Folly utilizando Aprofundamento Iterativo como a estratégia de busca, esse planejador não foi capaz de solucionar quebra-cabeças mais complexos (como Rope Ladder), Não-Determinísticos ou Não-Estacionários.

2.2 ANSWER SET PROGRAMING (ASP)

Como pode ser visto no trabalho de Eiter, Ianni e Krennwallner (2009), Answer Set Programming (ASP) é uma linguagem que faz uso do paradigma de programação declarativa, diferenciando-se de outros paradigmas por não ser necessário conhecer como resolver um problema, bastando apenas a descrição de tal. ASP é baseada em programação lógica com raciocínio não-monotônico, sendo usada com sucesso para solução de problemas NP-Completo, como o problema do Caixeiro Viajante (HOFFMAN; PADBERG; RINALDI, s.d.). ASP também é usada para representar formalismos que lidam com problemas de tomada de decisão sequencial (FREITAS DOS SANTOS et al., 2018), além de ser construída para modelar e resolver problemas que lidam com raciocínio de senso comum, como raciocínio espacial não-monotônico (WAŁĘGA; SCHULTZ; BHATT, 2017).

A escolha do ASP é justificada pelas características apresentadas por Eiter, Ianni e Krennwallner (2009), em que as seguintes vantagens da linguagem são apontadas: (1) possibilidade de descrição de quais soluções são preferidas em comparação com outras, definindo algum critério de qualidade para realizar tal classificação, possibilitando assim trabalhar com problemas ou domínios que requerem manipulação de preferências; (2) a habilidade do ASP de lidar com falta de informações adequadamente; (3) o uso de regras de escolha, permitindo assim o mapeamento entre uma entrada e diferentes saídas.

Uma importante característica do ASP, apontada por Lifschitz (2008), é que essa linguagem depende de uma semântica declarativa baseada na definição de modelos estáveis, que é definido por Gelfond e Lifschitz (1988) como "um possível conjunto de crenças de um agente, considerando as premissas do programa ASP". Quando se utiliza esta linguagem, os problemas de busca são reduzidos a computar modelos estáveis (GELFOND, 2008) e a realização desta busca é feita por um programa solucionador *answer set* responsável pela geração dos modelos

Figura 4 – Um programa em ASP

```
bird(tweety).
bird(woody).
penguin(tweety).
flies(X) :- bird(X), not ab(X).
ab(X) :- penguin(X).
```

Fonte: Cabalar (2017)

Figura 5 – Solução para o programa ASP da figura 4

```
bird(tweety) bird(woody) penguin(tweety)
ab(tweety) flies(woody)
```

Fonte: Cabalar (2017)

estáveis. Um exemplo de programa em ASP pode ser visto na figura 4, e a sua solução (*answer set*) é apresentada na figura 5, nota-se que neste caso há apenas uma solução para este programa, gerando assim, apenas um *answer set*.

Quando se escreve um programa em ASP são definidas uma série de cláusulas que determinam como este programa será executado e o que deverá ser seguido. Lifschitz (2008) apresenta a forma dessas cláusulas, sendo do tipo:

$$\texttt{A :- L1, L2,..., Ln} \tag{1}$$

em que A é um átomo, a cabeça da cláusula, enquanto a conjunção de literais, L1, L2,..., Ln são os átomos que formam o corpo. Além desses elementos, há também os símbolos que formam uma cláusula, tal qual ":-" que é a implicação lógica, podendo ser lida como um condicional "se" e o símbolo "," representando a conjunção. Portanto, a cláusula 1 pode ser representada como $\texttt{L1} \wedge \cdots \wedge \texttt{Ln} \rightarrow \texttt{A}$.

ASP também conta com dois conceitos que tratam de valores que podem ser assumidos e de tipos de negação. O primeiro é a possibilidade de se ter um terceiro valor de *desconhecido*, completando as duas opções padrões, *verdadeiro* e *falso*. O segundo conceito trata da possibilidade de dois tipos de negação, negação forte, ou clássica, e negação fraca, que representa negação por falha. Para representar esses conceitos na linguagem, diferentes símbolos são utilizados, o símbolo "-" (traço) representa a negação forte, e o símbolo "not" representa a negação fraca.

Para melhor entendimento, é possível explorar um exemplo apresentado por Lifschitz (2008), em que um ônibus apenas deve atravessar uma linha de trem quando nenhum trem está vindo. Portanto, aplicando a negação por falha na equação (2):

$$\texttt{atravessar :- not trem.} \tag{2}$$

não se pode ter certeza de que esta condição é atendida, já que essa regra permite que o ônibus atravesse a linha mesmo quando não se tem informações sobre a aproximação ou não do trem, podendo levar a uma consequência desastrosa. A forma mais segura de representar a regra de que só se deve atravessar a linha quando não houver trem se aproximando é por meio do uso da negação forte, mostrada na equação (3), porque esse tipo de negação garante que o ônibus só atravessará quando houver informação concreta de que não há trem se aproximando.

```
atravessar :- -trem.
```
(3)

Uma outra característica do ASP é a possibilidade de gerar diferentes saídas para a mesma entrada (FERREIRA et al., 2018). Isso quer dizer que, por exemplo, dado um estado qualquer s e uma ação qualquer a, é possível descrever, utilizando um programa ASP, os estados $s1$ e $s2$ como possíveis saídas da execução da ação a no estado s, essa regra pode ser representada em ASP utilizando a formula:

```
1{s1, s2}1 :- a, s.
```
(4)

essa regra é lida da seguinte maneira, dado que o agente está em $s \in S$ (conjunto de estados possíveis de um determinado domínio) e a ação $a \in A$ (conjunto de ações que podem ser executadas em um determinado domínio) é executada, deve-se escolher um e apenas um estado futuro (que é consequência de executar a ação a no estado s), entre os estados $s1$ e $s2$.

Com isso em mente, este trabalho parte da suposição de que para cada estado $s \in S$ existe um conjunto de regras em ASP que descrevem as consequências de se executar uma ação $a \in A$. Sendo assim, para cada ação do conjunto A deve existir a descrição da consequência, informando os estados possíveis de serem alcançados, como na cláusula apresentada na formula 4.

Um programa ASP também permite a representação de restrições, como mostrado por Lifschitz (2008) e Eiter, Ianni e Krennwallner (2009), que é uma cláusula sem a cabeça. Esse tipo de restrição é utilizado, principalmente, para excluir possíveis soluções na execução de um programa. Por exemplo, considerando o domínio de colorir diferentes vértices de um grafo, tem-se o requerimento que vértices adjacentes não devem ser coloridos com a mesma cor, para representar essa condição de pintura, a restrição 5 pode ser utilizada.

```
:- borda(X, Y ), vermelho(X), vermelho(Y).
```
(5)

No contexto deste trabalho, as restrições estão relacionadas com a descrição de estados impossíveis (estados dos quebra-cabeças que não podem ser fisicamente alcançados), ações impossíveis (que não podem ser executadas fisicamente) e pares estado/ações impossíveis (ações que não podem ser fisicamente executadas em um estado específico).

Por fim, para encontrar as soluções (*answer sets*) de um programa lógico, ASP deve realizar o processo de *grounding* e *solving*. O processo de *grounding* é a substituição das variáveis de um programa ASP por um programa que não possui variáveis ou funções, é a substituição

Figura 6 – Um programa em ASP

```
p(a). p(b). p(c).
q(X) :- p(X).
2 {r(X) : p(X)}.
```

Fonte: Lifschitz (2008)

Figura 7 – O programa, representado na figura 6, depois do processo de *grounding*

```
p(a). p(b). p(c).
q(a) :- p(a).
q(b) :- p(b).
q(c) :- p(c).
2 {r(a),r(b),r(c)}.
```

Fonte: Lifschitz (2008)

de todas as variáveis por todas as instâncias possíveis dessas mesmas variáveis. A figura 6 mostra um exemplo de um programa ASP antes do processo de *grounding* e a figura 7 mostra o programa após o processo de *grouding*.

Após o *grouding* é possível executar o processo de *solving*, que é responsável por encontrar os *answer sets* do programa representado na figura 7.

Com a apresentação do ASP e seu entendimento teórico pode-se definir que o objetivo deste trabalho na sua utilização é representar um conjunto de estados S de um MDP. Então, com essa representação, encontrar todo *answer set* para cada estado que o agente tem permissão para visitar, em outras palavras, toda transição permitida para o par estado-ação (s, a). Além disso, ASP também é útil para lidar com quebra-cabeças que sofrem mudanças com o tempo (não-estacionários), já que essa linguagem pode revisar conhecimento previamente obtido e assim descrever as regras de transição de um domínio após as mudanças encontradas. Finalmente, após a obtenção da representação dos estados e das funções de transição, aplicar Aprendizado Por Reforço (AR), Q-Learning especificamente, para encontrar uma sequência de ações ótimas que resolva o MDP que descreve o domínio dos Quebra-Cabeças de Fisherman's Folly e Rope Ladder.

2.3 APRENDIZADO POR REFORÇO (AR)

Há alguns problemas que são difíceis de solucionar utilizando apenas regras pré-definidas, como regressão, classificação e problemas de tomada de decisão em ambientes desconhecidos.

Por isso diferentes métodos de Aprendizado de Máquina (AM) vêm se tornando cada vez mais comuns para solução de diversos problemas do mundo real.

Há na literatura uma separação comum em três diferentes categorias que descrevem como um sistema aplica os conceitos de AM, sendo estas: Aprendizado Supervisionado, Aprendizado não Supervisionado e Aprendizado por Reforço.

Como descrito por Hastie, Tibshirani e Friedman (2009), o Aprendizado Supervisionado é uma técnica em que tanto os dados de entrada quanto a saída esperada são fornecidos para o algoritmo de aprendizagem. Isso ocorre no momento em que o algoritmo é executado e o processo iniciado. É uma abordagem utilizada para solução de diversos problemas, como por exemplo, classificar números em uma imagem. Neste caso, os dados para treinamento podem ser imagens que representam um determinado número e como saída esperada o valor numérico que é representado por essa imagem. Uma das técnicas mais utilizadas dentro deste paradigma é a de Rede Neural Artificial (RNA) aplicada em tarefas de regressão e classificação (HASTIE; TIBSHIRANI; FRIEDMAN, 2009).

Uma outra abordagem de aprendizado é a de Aprendizado não Supervisionado, que é definido por Barlow (1989) como um tipo de aprendizado em que a saída esperada não é fornecida para o algoritmo, apenas os dados de entrada são fornecidos. Essa abordagem é utilizada em problemas, por exemplo, em que precisa-se descobrir agrupamentos em um conjunto de dados (o projetista não sabe o que forma esse agrupamento previamente). O exemplo de classificação de números também pode ser visto como um problema de Aprendizado não Supervisionado, já que bastaria retirar o fornecimento dos dados de saída esperado e obter os grupos que podem ser formados pelas imagens fornecidas. Um dos métodos que faz uso desta abordagem é o K-Means (JAIN, 2010).

Por fim tem-se o Aprendizado por Reforço (AR) que, segundo Sutton e Barto (2018), pode ser definido como o problema de aprendizagem por meio de interações com o ambiente a fim de atingir um determinado objetivo. Sendo assim, para que haja essa interação, o sistema de AR é dividido em duas entidades distintas, o Agente e o Ambiente. Neste contexto, o Agente é a parte que aprende e toma as decisões (executa ações), enquanto que o local (ou mundo) em que o Agente executa essas ações é o Ambiente, que pode ser definido como o local em que as interações acontecem. Essas interações ocorrem seguindo uma sequência de passos:

a) primeiro, o Agente escolhe uma ação para ser executada no Ambiente. Vale notar que a execução da ação em um determinado instante "t" não tem consequência apenas em "t", mas também pode ter consequências nos estados futuros e, consequentemente, nas recompensas futuras;
b) em seguida, o Ambiente responde para o Agente com o estado resultado da ação e a recompensa de se ter executado aquela determinada ação;
c) por fim, o Agente atualiza o valor de executar essa ação baseando-se na recompensa recebida, usando uma fórmula que é específica para cada algoritmo de AR.

Como AR lida com problemas de tomada de decisão sequencial, se faz necessário a definição de um formalismo para representação desses problemas, sendo assim, Sutton e Barto (2018) definem o AR sobre o Processo de Decisão de Markov (MDP), que pode ser representado pela tupla <S, A, T, R>, em que:

a) S é o conjunto de estados que podem ser observados no domínio;
b) A é o conjunto de ações que podem ser executadas por um Agente;
c) T representado pela equação (6), é a função de transição que provê a probabilidade de o Agente, estando em um estado $s \in S$ e executando uma ação $a \in A$, alcançar o estado futuro $s' \in S$;
d) R representado pela equação (7), é a função de recompensa que provê a recompensa para o Agente, quando este se encontra em um estado $s \in S$ e executa uma ação $a \in A$, alcançando o estado futuro $s' \in S$.

$$\text{Transição} = S \times A \times S \mapsto [0,1] \tag{6}$$

$$\text{Recompensa} = S \times A \times S \mapsto Re \tag{7}$$

Além dessa tupla, um MDP assume a Propriedade de Markov de primeira-ordem, definindo que um Agente, presente em um estado $s \in S$, tem toda informação necessária para decidir que ação deve ser executada, sem contar com a necessidade de consultar qualquer tipo de histórico para isso. Em outras palavras, cada estado do Ambiente deve resumir experiências passadas de uma forma compacta, mantendo as informações que ainda são relevantes para o processo de tomada de decisão por parte do Agente (SUTTON; BARTO, 2018).

Outro ponto de destaque de AR é a escolha entre exploitar e explorar[2] já que, na busca pela melhor recompensa, o Agente deve escolher as ações que já foram executadas antes e que resultaram em algum tipo de sucesso, medido através da recompensa (SUTTON; BARTO, 2018). Mas para encontrar essas ações bem-sucedidas, o Agente deve explorar o ambiente, e assim testar diferentes ações não vistas anteriormente. Pode-se ver assim como o uso individual de qualquer uma das duas opções leva à falha, portanto a melhor abordagem, segundo Sutton e Barto (2018), é uma combinação das duas, podendo começar com uma taxa de exploração alta e ir diminuindo gradualmente com o passar das interações, permitindo assim que o Agente possa encontrar novas ações e também utilizar aquelas que já foram testadas.

Observando todas essas características do AR é possível notar o porquê da sua escolha como técnica de aprendizado neste trabalho, já que AR apresenta uma maneira de aprender uma política ótima por meio de interações com os quebra-cabeças, movimentando os elementos, criando e modificando as relações entre estes elementos, sem a necessidade de nenhum conhecimento anterior sobre as funções de Transição e Recompensa. Neste trabalho o algoritmo de

[2]Exploitar é escolher a melhor decisão dada uma informação já conhecida, neste caso, escolher a ação com o maior valor determinado pela função Q(s, a), enquanto que explorar é escolher uma ação qualquer para ser executada naquele estado, não considerando o valor da função Q(s,a).

AR que foi usado é o Q-Learning, que aprende uma política ótima enquanto um determinado agente interage com o ambiente. Essa interação acontece com o agente executando uma ação $a \in A$, no instante t e presente no estado $s \in S$. Após essa execução, o agente observa o estado futuro $s' \in S$ no instante t_{+1} e recebe uma recompensa r. Ao fim de cada interação, Q-Learning é capaz de atualizar a função valor-ação Q(s,a), utilizando para isso a equação (8). Os termos adicionais dessa fórmula como o "α" e o "γ" representam a taxa de aprendizado e o fator de desconto, respectivamente.

$$Q_{t+1}(s_t,a_t) \leftarrow Q_t(s_t,a_t) + \alpha \cdot (r + \gamma \cdot \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t)). \tag{8}$$

Essas interações são repetidas inúmeras vezes (definida pelo projetista), sendo que ao fim de todas as interações, é esperado que um agente equipado com Q-Learning encontre uma política ótima para o problema.

Um ponto a se destacar sobre o processo de aprendizado do Q-Learning é que no começo das interações o agente não possui nenhum conhecimento sobre qual ação escolher, sendo uma escolha totalmente aleatória inicialmente. Sutton e Barto (2018) mostram como o conhecimento que o Agente pode ter no início da tarefa (antes de começar a interagir com o Ambiente) pode diminuir o tempo necessário para o Agente aprender tal tarefa. Portanto, uma outra possibilidade quando usa-se AR é a aplicação de heurísticas para acelerar o processo de aproximação da função valor-ação Q(s,a), fornecendo o conhecimento que foi obtido antes do processo de aprendizado começar. Uma abordagem que aplica o conceito de conhecimento prévio para acelerar o processo de aprendizado é conhecida como Aprendizado por Reforço Acelerado por Heurística (HARL) e foi introduzida por Bianchi, Ribeiro e Costa (2008). Essa aceleração é possível porque a heurística é capaz de guiar o processo de aprendizado, agindo no momento da escolha da ação pelo algoritmo de AR. Com isso, considerando que se usa a estratégia ϵ-greedy[3] para seleção de ação, a única diferença entre a aplicação tradicional de Q-Learning e HARL é no momento da escolha da ação, já que essa regra, no algoritmo HARL, passa a ser descrita como:

$$\pi(s) = \begin{cases} argmax_a[\hat{Q}(s,a) + \xi H(s,a)^{\beta}], & \text{se } q \leq p. \\ a_{aleatória}, & \text{caso contrário,} \end{cases} \tag{9}$$

em que $H(s,a)$ é a função heurística que guia a escolha da ação; ξ e β são parâmetros que controlam a influência de $H(s,a)$ no processo de escolha; q é um valor aleatório entre 0 e 1; p é um valor entre 0 e 1 responsável por definir o equilíbrio entre exploitar/explorar no algoritmo Q-Learning; $a_{aleatória}$ é uma ação que foi escolhida de forma aleatória quando q é maior que p.

Apesar do conhecimento prévio (heurísticas) ajudar na aceleração do aprendizado, é a interação com o Ambiente que leva o Agente a aprender a política ótima, sendo este um processo

[3] A estratégia em que o agente tem uma probabilidade (1 - ϵ) de escolher uma ação que maximize o valor da política e uma probabilidade (ϵ) de escolher uma ação de maneira aleatória.

essencial para que o Agente possa ajustar seu comportamento e explorar certas características específicas da tarefa[4], que não puderam ser mapeadas apenas com as heurísticas conhecidas.

Considerando as definições de AR e MDP apresentadas acima, é possível definir a importância desses conceitos nessa dissertação, que é formalizar os estados dos quebra-cabeças como um MDP e então aplicar um algoritmo de AR para encontrar uma solução para estes quebra-cabeças. Para interagir com os domínios dos quebra-cabeças, um planejador em PROLOG é usado como Oráculo, permitindo que o agente Q-Learning possa perguntar ao Oráculo qual a consequência de executar uma ação em um determinado estado. Além disso, apesar do Q-Learning não precisar conhecer informações sobre as funções de Transição e Recompensa, ainda precisa saber sobre os conjuntos de ações possíveis A e de estados S. Essa necessidade justifica a combinação com ASP, já que o último pode ser usado para representar um MDP permitindo a construção do conjunto de estados visitados S e do conjunto de ações A de uma maneira *online*, mesmo quando há mudanças no domínio. Na próxima seção é apresentado o algoritmo oASP(MDP) que combina ASP e MDP.

2.4 ONLINE ANSWER SET PROGRAMING PARA PROCESSO DE DECISÃO DE MARKOV (oASP(MDP))

Ferreira et al. (2018) definem o algoritmo Online ASP for MDP (oASP(MDP)) como a combinação dos conceitos descritos acima, obtendo vantagem do ASP com programação lógica, do MDP com o formalismo para descrever um processo de tomada de decisão, e de AR com o algoritmo Q-Learning. Desse modo, ASP é usado para descrever o MDP, fazendo com que seja possível encontrar os *answer sets* através da exploração do espaço de estados, enquanto que o Q-Learning é usado para aproximar a função valor-ação Q(s, a) desse MDP.

Um dos primeiros pontos de observação quando se considera esta nova proposta de algoritmo é como modelar um MDP utilizando um programa lógico. Essa modelagem é feita através da definição de regras de escolha que descrevem uma transição t(s, a, s'), com a fórmula lógica em ASP a seguir:

```
1{s'}1 :- a, s                    (10)
```

Segundo Ferreira et al. (2018), essa descrição pode ser aplicada para toda ação e todo estado no domínio. Portanto, descrevendo toda transição para cada ação e para cada estado que foi visitado, em um determinado instante, como um programa lógico, é possível usar ASP para obter um conjunto de estados e ações observáveis e, através da interação entre o agente Q-Learning e o ambiente, definir a função valor-ação Q(s,a).

ASP é uma linguagem que acrescenta ao algoritmo de AR (Q-Learning) a capacidade de realizar revisões nos conjuntos de estados e ações quando uma mudança é observada no domínio. Essa revisão tem como consequência a alteração da formalização do MDP que foi utilizado para descrição inicial do domínio. Todo esse processo, de alteração e revisão, faz com

[4]Por exemplo no domínio de Fisherman's Folly, aprender quais são as ações ótimas que levam ao objetivo.

que ocorra uma parada no cálculo da função Q(s, a), permitindo assim que os *answer sets* do programa lógico, que descrevem o novo MDP modificado, sejam encontrados e imediatamente passam a ser utilizados como uma nova descrição do MDP. Assim, não há necessidade alguma de reinicialização completa do processo de aprendizado e nem do MDP, como ocorre na versão tradicional do Q-Learning. O algoritmo oASP(MDP) tem como objetivo utilizar uma combinação entre ASP e MDP para então, a partir de um método de AR, aproximar a função valor-ação Q(s, a) de um MDP, podendo este ser não-estacionário (FERREIRA et al., 2018).

No algoritmo 1 tem-se a representação em pseudocódigo do algoritmo oASP(MDP). O algoritmo começa recebendo três parâmetros de entrada (linha 2, algoritmo 1): o primeiro é o conjunto de ações que o agente pode executar inicialmente no ambiente, é importante notar que esse conjunto de ações pode sofrer mudanças enquanto as interações acontecem, já que oASP(MDP) permite a realização de revisão do conhecimento, passando a não levar em consideração ações que não podem ser executadas fisicamente no ambiente (como passar o Disco1 pelo Anel). Essas ações impossíveis são aprendidas enquanto há a interação com o ambiente; o segundo parâmetro é um algoritmo M de AR que pode aproximar a função valor-ação Q(s, a); e por último o oASP(MDP) recebe a quantidade n de episódios que deve ser executado. Depois de definidas essas entradas, o próximo passo do algoritmo é inicializar o conjunto de estados visitados S como um conjunto vazio (linha 4, algoritmo 1), já que a construção de S é feita passo a passo conforme o agente interage com o ambiente. A novidade deste algoritmo, comparando com os algoritmos de AR tradicionais, é que M ainda é capaz de aproximar a função valor-ação Q(s, a), mesmo com a construção incremental dos conjuntos de estados e de ações para cada estado, e retornar a tabela Q, definindo que a política ótima é seguida se as ações com os maiores valores são executadas (FERREIRA et al., 2018).

Logo após a sua inicialização, o algoritmo entra em um laço de repetição (linha 5, algoritmo 1), que será executado um número n de vezes (n é um valor informado ao algoritmo através do parâmetro de entrada). Em cada repetição deste laço, o algoritmo observa o estado atual s, e então, poderá tomar duas decisões distintas, dependendo do estado atual s estar presente ou não no conjunto S. Portanto, se $s \notin S$ (linha 8, algoritmo 1), então: s é adicionado ao conjunto S, uma ação aleatória é escolhida e executada, retornando um estado futuro s'. A partir disso é possível descrever as transições para cada ação em cada estado como um programa lógico através das regras de escolha (como a formula 10), considerando que um solucionador ASP é usado para encontrar todo *answer set* para cada estado s visitado neste episódio. Finalmente, após a definição das regras de escolha ocorre a atualização da descrição de Q(s, a) (linha 13, algoritmo 1). Por outro lado, se $s \in S$ (linha 15, algoritmo 1), então um outro fluxo é seguido pelo oASP(MDP), já que neste caso, a escolha da ação é feita pelo algoritmo M (linha 16, algoritmo 1) e não há atualização das regras de escolha, pois o estado s já está presente em S.

Enfim, independente de qual fluxo de execução for seguido pelo oASP(MDP), a função valor-ação Q(s, a) é atualizada (linha 20, algoritmo 1) n vezes, de acordo com o algoritmo de AR M. Outra atualização que ocorre ao mesmo tempo da atualização de Q(s, a) é a do estado

Algoritmo 1 – oASP(MDP)

1 **Algoritmo:** oASP(MDP)
2 **Entrada:** O conjunto de ações $\mathcal{A}$, um método de aproximação da função valor-ação M e o número de episódios n.
3 **Saída:** A função aproximada $Q(s,a)$.
4 Inicialize o conjunto de estados observados $\mathcal{S}_o = \emptyset$
5 **enquanto** *número de episódios for menor que* n **faça**
6 **repita**
7 Observe o estado atual s
8 **se** $s \notin \mathcal{S}_o$ **então**
9 Adicione s ao conjunto de estados $\mathcal{S}_o$.
10 Escolha e execute uma ação aleatória $a \in \mathcal{A}$.
11 Observe o estado futuro s'.
12 Atualize o estado s adicionando ao programa lógico a transição observada através da regra de escolha correspondente.
13 Atualize a descrição de $Q(s,a)$ encontrando cada answer set para cada estado s adicionado a $\mathcal{S}_o$ neste episódio.
14 **fim**
15 **senão**
16 Escolha uma ação $a \in \mathcal{A}$ definida por M.
17 Execute a ação escolhida a.
18 Observe o estado futuro s'.
19 **fim**
20 Atualize o valor de $Q(s,a)$ a partir da definição encontrada por M.
21 Atualize o estado atual $s \leftarrow s'$.
22 **até** *fim do episódio*
23 **fim**

Fonte: Ferreira et al. (2018)

atual (linha 21, algoritmo 1), que passa a ser o estado (s') observado quando executada uma ação a no estado s.

Com relação às mudanças no conjunto de ações A, se a ação que o agente tentou executar for fisicamente impossível de ser executada (por exemplo, tentar passar qualquer Esfera pelo Buraco do Poste), o ambiente informa isso ao agente, que então não considera mais essa, não a levando em consideração em momentos futuros. Além da remoção de ações impossíveis para todos os estados, oASP(MDP) também é capaz de excluir ações especificamente por estado.

Outra vantagem do oASP(MDP) é sua capacidade de lidar com mudanças nos conjuntos de estados, tornando esse algoritmo viável para diferentes tipos de problemas que sejam não-estacionários. Diferente da versão tradicional do Q-Learning, que precisa reinicializar seus valores quando uma mudança é detectada no ambiente, o oASP(MDP) é capaz de usar o conhecimento obtido antes da mudança ocorrer para tomar decisão no domínio após a mudança.

A próxima seção apresenta como o algoritmo oASP(MDP) foi aplicado nos quebra-cabeças espaciais. Além disso, a contribuição principal dessa dissertação também é apre-

sentada, que é a adaptação do oASP(MDP) para utilizar heurísticas, o algoritmo Heurística oASP(MDP).

3 HEURÍSTICA E oASP(MDP)

Este capítulo apresenta a principal contribuição deste trabalho, a adaptação do algoritmo oASP(MDP) para utilizar heurísticas, o algoritmo Heurística oASP(MDP) (HoASP(MDP)) (algoritmo 2).

No algoritmo HoASP(MDP) as heurísticas são levadas em consideração no momento da escolha da ação (linha 16, algoritmo 2). Para um melhor entendimento sobre essa mudança, uma aplicação do HoASP(MDP) no domínio do quebra-cabeça de Fisherman's Folly é apresentada a seguir.

HoASP(MDP) é inicializado recebendo o conjunto de Ações A, um algoritmo M de AR e o número de episódios n que deve ser executado.

Depois do recebimento dos parâmetros de entrada, o agente inicializa o conjunto de estados visitados S como um conjunto vazio (linha 4, algoritmo 2), que é o ponto inicial do episódio. O agente verifica se o estado inicial $s0$ (Figura 2a) está no conjunto S (linha 8, algoritmo 2), como o conjunto S está vazio no começo do processo de aprendizado, $s0 \notin S$, o agente executa uma ação aleatória, por exemplo, passar o Disco através do Buraco do Poste da direita para esquerda. Depois dessa execução, o agente está no estado resultante $s1$, $s0$ pode ser adicionado ao conjunto S e o agente recebe a recompensa como resposta do ambiente. Após a primeira interação o agente passa a ter conhecimento sobre uma transição no domínio, esse conhecimento é expressado como uma regra de escolha em ASP (linha 12, algoritmo 2) com a fórmula: 1{s1}1 :- s0, a(AçãoExecutada), em que "AçãoExecutada" é a ação "passar o Disco através do Buraco do Poste da direita para esquerda" executada pelo agente. Seguindo a obtenção das regras de escolha, o agente pode encontrar todos *answer sets* para o estado visitado e então atualizar a descrição (os pares estado/ação possíveis) da função valor-ação Q(s,a) para este estado.

O resultado da ação executada em $s0$ leva o agente para o estado $s1$, portanto todos os passos descritos acima são executados da mesma maneira. Porém uma nova situação pode acontecer, assumindo que a ação que foi escolhida aleatoriamente tenha sido a ação passar o Disco através do Buraco do Poste da esquerda para direita, desfazendo a ação executada previamente no estado $s0$, levando o agente de volta para o estado inicial ($s0$). Agora que o agente está em $s0$, e $s0 \in S$, outro fluxo de execução do HoASP(MDP) deve ser seguido. Ao invés de escolher uma ação aleatória, o agente usará o algoritmo de AR (linha 16, algoritmo 2) a fim de escolher uma ação para ser executada, e é exatamente neste instante que a influência da heurística é levada em consideração. Primeiro há o mapeamento entre um estado na tarefa que fornece a heurística (Fisherman's Folly Simplificado (FFS), subseção 2.1.2) para um estado na tarefa original (aquela que é o foco da solução, Fisherman's Folly Original (FFO), subseção 2.1.1) e então o agente escolhe, seguindo a equação 9, qual ação deve ser executada.

No fim de cada interação, a recompensa recebida pelo agente é usada para atualizar o valor da função valor-ação Q(s,a) para o estado visitado e a ação executada. Essas interações,

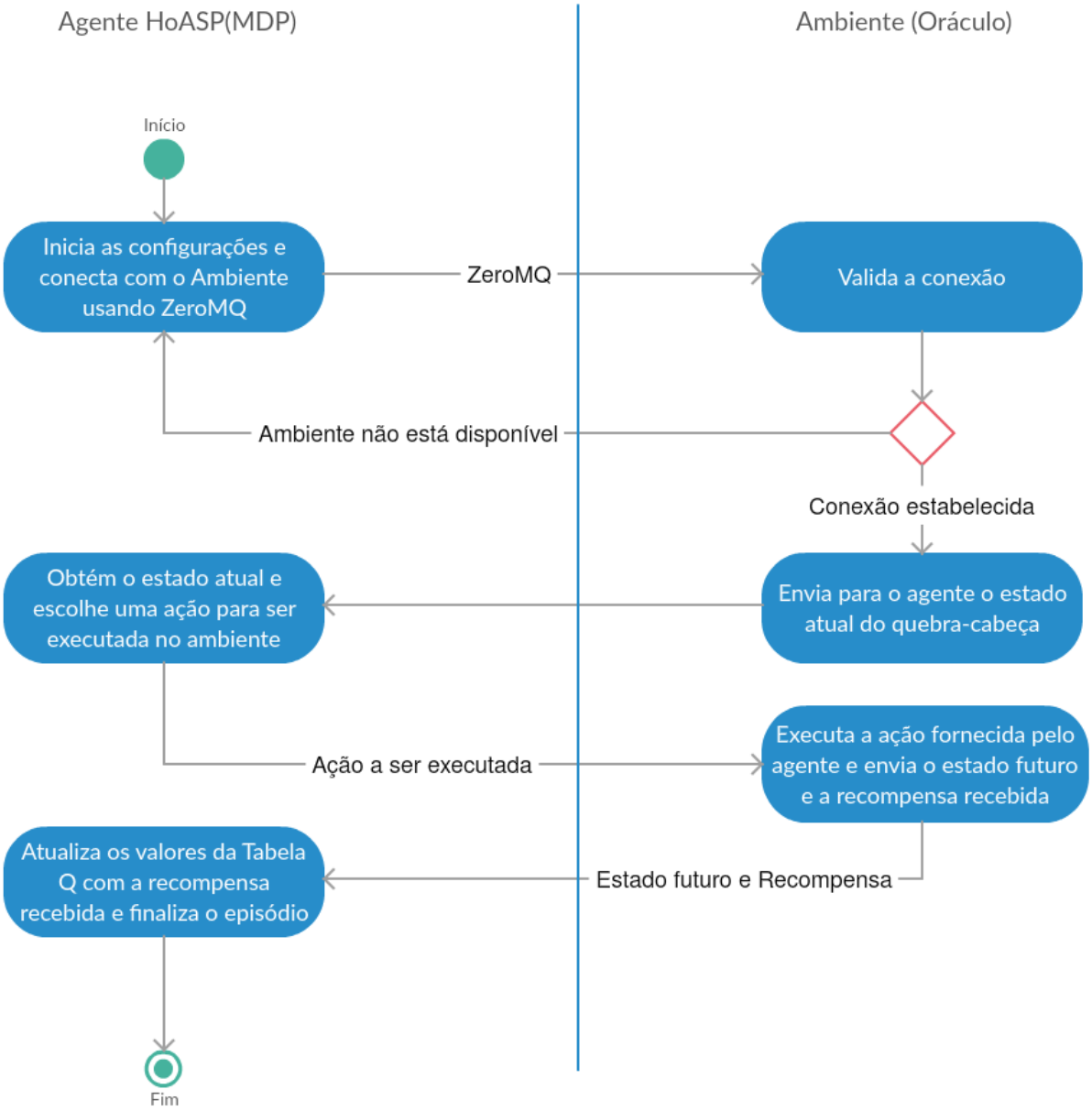
Algoritmo 2 – HoASP(MDP)

1 **Algoritmo:** HoASP(MDP)
2 **Entrada:** O conjunto de ações $\mathcal{A}$, um método de aproximação da função valor-ação M e o número de episódios n.
3 **Saída:** A função aproximada $Q(s,a)$.
4 Inicialize o conjunto de estados observados $\mathcal{S}_o = \emptyset$
5 **enquanto** *número de episódios for menor que* n **faça**
6 **repita**
7 Observe o estado atual s
8 **se** $s \notin \mathcal{S}_o$ **então**
9 Adicione s ao conjunto de estados $\mathcal{S}_o$.
10 Escolha e execute uma ação aleatória $a \in \mathcal{A}$.
11 Observe o estado futuro s'.
12 Atualize o estado s adicionando ao programa lógico a transição observada através da regra de escolha correspondente.
13 Atualize a descrição de $Q(s,a)$ encontrando cada answer set para cada estado s adicionado a $\mathcal{S}_o$ neste episódio.
14 **fim**
15 **senão**
16 **Escolha uma ação $a \in \mathcal{A}$ definida por M, utilizando a regra de escolha com heurística (equação 9).**
17 Execute a ação escolhida a.
18 Observe o estado futuro s'.
19 **fim**
20 Atualize o valor de $Q(s,a)$ a partir da definição encontrada por M.
21 Atualize o estado atual $s \leftarrow s'$.
22 **até** *fim do episódio*
23 **fim**

Fonte: Autor

que também podem ser vistas no diagrama apresentado na Figura 8, são repetidas n vezes (onde n é o número de episódios), sempre atualizando o valor da função valor-ação Q(s,a) com a recompensa proveniente do ambiente. Ao fim de todas as interações, o algoritmo HoASP(MDP) é capaz de retornar a tabela Q. Essa tabela representa os valores para cada ação executada em cada estado visitado pelo agente, sendo que uma das propriedades mais importantes dessa tabela é que se o agente sempre executar as ações com os maiores valores, então o agente está executando a política ótima que foi aprendida (SUTTON; BARTO, 2018). Portanto, vale pontuar que a saída do HoASP(MDP) é uma política, não um plano. Um plano é uma sequência de ações que precisam ser seguidas para que um objetivo seja alcançado (sendo crucial a execução correta do plano, um replanejamento é necessário caso algo não esperado aconteça, como o agente ir para um estado diferente do planejado inicialmente); por outro lado, uma política é o mapeamento entre estado/ação, em que há diversos pares estado/ação que podem ser executados para que o agente atinja um objetivo (nesse caso não há necessidade de encontrar uma outra política caso o agente se encontre em um estado diferente do estado esperado).

Figura 8 – Diagrama apresentando a interação por um episódio do processo de aprendizado entre o Agente HoASP(MDP) e o Ambiente com Oráculo (Planejador PROLOG)



Fonte: Autor

No contexto desse trabalho, cada estado visitado $s \in S$ tem um arquivo ASP com todas as regras de transição (para cada ação executada, há a consequência de executar essa ação em um determinado estado), sendo que esse arquivo ASP é atualizado a cada nova transição descoberta pelo agente para o estado s. Para os quebra-cabeças determinísticos e não-estacionários, a fórmula abaixo representa uma regra aprendida em ASP pelo agente HoASP(MDP):

$$1 \ \{s1\} \ 1 \ :- \ a, \ s0;$$

em que $s1$ é o estado futuro quando o agente executa a ação a no estado $s0$.

Já para os quebra-cabeças não-determinísticos, como uma ação pode ter três diferentes consequências, cada regra de transição pode ser descrita como:

$$1\ \{s1,\ s2,\ s3\}\ 1\ :-\ a,\ s0.$$

com os estados $s1$, $s2$ e $s3$ como possíveis estados futuros quando o agente executa a ação a no estado $s0$.

Cada estado pode ter no máximo 20 ou 28 das regras acima, esse número depende do quebra-cabeça considerado, 20 regras caso o quebra-cabeça seja o Fisherman's Folly e 28 regras caso o quebra-cabeça seja o Rope Ladder. A quantidade de regras está diretamente relacionada com o número de possíveis ações que podem ser executadas nos quebra-cabeças.

Além dessas regras de transição, cada estado tem um outro arquivo ASP que contém as restrições representando as ações impossíveis para aquele estado em específico, essas regras são definidas como:

$$:-\ a,\ s0.$$

em que a é uma ação presente no conjunto A que é impossível de ser executada fisicamente no quebra-cabeça quando o agente está no estado $s0$.

Portanto, cada estado visitado pelo agente tem dois arquivos ASP, um com todas as regras de transição descobertas nas interações e o outro com as ações que são impossíveis de serem executadas naquele estado.

Por fim, o último conjunto de regras ASP, que é aprendido pelo HoASP(MDP), é o que contém restrições, representando todas as ações que são impossíveis em todos os estados, com a seguinte formula:

$$:-\ a.$$

a é uma ação no conjunto A que não pode ser executada em nenhum estado do quebra-cabeça, como tentar atravessar a Esfera1 pelo Buraco do Poste.

3.1 HoASP(MDP) PARA RESOLVER QUEBRA-CABEÇAS ESPACIAIS

Definidos os principais pontos acerca do algoritmo HoASP(MDP) para solução de problemas de tomada de decisão sequencial, essa seção descreverá como o HoASP(MDP) foi adaptado especificamente para solucionar quebra-cabeças espaciais (de crescente complexidade, não-determinísticos e não-estacionários) descritos nessa dissertação. Ao todo, sete quebra-cabeças foram estudados neste trabalho, que foram separadas em quatro categorias distintas (detalhes na seção 2.1): Quebra-Cabeças Originais, aqueles sem alterações nos elementos e nas relações; Quebra-Cabeças Simplificados, aqueles que contam com algum tipo de simplificação com o intuito de facilitar o processo de encontrar uma solução; Quebra-Cabeças Não-Determinísticos, aqueles em que a execução de uma ação pode ter diferentes consequências; e Quebra-Cabeças Não-Estacionários, aqueles que sofrem mudanças nos elementos ou relações conforme as interações acontecem.

Para resolver esses quebra-cabeças e acelerar o processo de aprendizado, nessa dissertação duas diferentes abordagens são utilizadas para a obtenção das heurísticas de domínio, dependendo de qual quebra-cabeça espacial é considerado.

Nas versões determinísticas e não-determinísticas do Fisherman's Folly, uma heurística admissível é obtida por meio da solução de uma versão simplificada do quebra-cabeça, aplicando o algoritmo oASP(MDP) original para encontrar essa solução. A tabela Q é a solução obtida pelo algoritmo, que é usada como heurística para acelerar o processo de aprendizado. Sendo que especificamente para o quebra-cabeça Fisherman's Folly Original (FFO) (subseção 2.1.1), o quebra-cabeça Fisherman's Folly Simplificado (FFS) (subseção 2.1.2) foi utilizado para obtenção da heurística. Já para o Fisherman's Folly Não-Deterministico (FFND) (subseção 2.1.3), o quebra-cabeça Fisherman's Folly Simplificado Não-Determinístico (FFSND) (subseção 2.1.4) foi o domínio utilizado para obtenção da heurística. Os quebra-cabeças simplificados mantiveram a mesma configuração e relacionamento entre objetos que os quebra-cabeças que utilizaram a heurística. Essa simplificação está relacionada com o funcionamento do quebra-cabeça espacial, adicionando uma restrição na quantidade de vezes que o agente pode enrolar a Corda pelo Buraco do Poste. Na simplificação, a Corda não pode ser enrolada através do Buraco do Poste, enquanto que nas versões FFO e FFND, a Corda pôde ser enrolada, até duas vezes, através do Buraco do Poste.

No quebra-cabeça Rope Ladder uma simplificação também foi utilizada para obtenção da heurística, mas com diferenças importantes a destacar considerando que uma outra estratégia foi utilizada para obtenção destas heurísticas. No Rope Ladder utilizado como base para obtenção de heurísticas, a Corda está, inicialmente, atravessando os dois Buracos do Poste apenas uma vez, que é o suficiente para simplificar o quebra-cabeça e permitir que a solução ótima seja encontrada. Outro ponto sobre a adaptação é a necessidade de criar uma função de mapeamento entre um estado no Rope Ladder com a simplificação para o estado no Rope Ladder Original. Como o conjunto de ações A é o mesmo em ambas as versões do quebra-cabeça, a função de mapeamento foi responsável por equiparar as ações que levaram o agente a um determinado estado na versão com a simplificação com as mesmas ações que levaram o agente para um outro estado na versão Original.

As ações em HoASP(MDP) são definidas pela tupla $< EP, EB, FP >$, em que:

a) EP é um *Elemento que Passa*, um elemento do quebra-cabeça que atravessa um buraco;

b) EB é um *Elemento com Buraco*, um elemento do quebra-cabeça que tem um buraco;[5]

c) FP é a *Face de Passagem*, a direção que o elemento EP vai passar através do elemento EB. Existem duas possíveis faces: positiva (+) e negativa (-). Se a passagem pelo buraco é feita da esquerda para direita a face é positiva, ao contrário a face é negativa.

[5]Por simplicidade, o buraco de um objeto é identificado pelo nome do próprio objeto.

Nos quebra-cabeças espaciais investigados, cada ação é a manipulação de um EP que passa através de um EB em direção à FP, sendo assim, a tupla $< EP, EB, FP >$ pode ser lida da seguinte maneira "EP está atravessando EB em direção à face FP". Essa formalização da ação é geral o suficiente para descrever as ações em todos os quebra-cabeças (e configurações) consideradas nesse trabalho. Como uma ação é a combinação de um EP, um EB e uma FP, o agente não pode escolher uma ação tendo o mesmo elemento em EP e EB. Outro ponto de consideração é que o agente, no início do processo de aprendizado, não tem conhecimento algum sobre quais elementos EP podem passar através dos elementos EB, por conta da restrição de tamanho dos elementos (por exemplo, passar a Esfera1 através do Buraco do Poste). O agente aprende essas restrições conforme interage com o ambiente e recebe as recompensas indicando quais ações são impossíveis de serem executadas fisicamente. Conforme o número de interações aumenta, o agente HoASP(MDP) aprende sobre os efeitos das ações e então passa a não considerar ações impossíveis no conjunto A, levando-o a executar, ao longo termo, menos ações impossíveis.

A tupla $< EP, EB, FP >$ pode ser definida especificamente para cada um dos quebra-cabeças. Começando com o Fisherman's Folly, tem-se 20 ações no conjunto A, que conta com a seguinte configuração: 6 elementos EP {Esfera1, Esfera2, Poste, Disco1, Disco2, Anel}; 2 elementos EB {Buraco do Poste, Anel}; e 2 FP {Positiva (+), Negativa (-)}.

Considerando o Rope Ladder, tem-se 28 possíveis ações a serem escolhidas com a seguinte configuração: 7 EP {Esfera1, Esfera2, Poste1, Poste2, Disco1, Disco2, Anel}; 3 EB {Buraco do Poste1, Buraco do Poste2, Anel}; e também 2 FP {Positiva (+), Negativa (-)}.

Além dessa formalização das ações, se faz necessária uma formalização dos estados dos quebra-cabeças espaciais. Sendo assim, esse trabalho segue uma abordagem similar à abordagem descrita por Santos e Cabalar (2016), que usa duas listas (uma com os elementos que são atravessados pelo Poste e outra para os elementos atravessados pela Corda) para representar todos os estados possíveis do quebra-cabeça. Essa formalização é escolhida por conta de como esses estados podem ser descritos, já que basta definir os elementos que são atravessados pela Corda e pelo Poste para representar um estado qualquer do quebra-cabeça. Por exemplo, para o estado inicial do Fisherman's Folly apresentando na Figura 2a, as listas que representam esse estado são definidas da seguinte maneira:

a) lista(corda) = [+Esfera1, +Poste, +Esfera2]
b) lista(poste) = [+Anel]

em que a lista(corda) e a lista(poste) contêm os elementos que são atravessados (elementos com buracos, EB) pela Corda e pelo Poste, respectivamente. Para facilitar a representação dos estados, cada elemento EB é descrito na lista pelo nome do elemento que contém o buraco e com a face que o buraco foi atravessado. Por exemplo, na lista dos elementos atravessados pelo Poste, tem-se +Anel, que representa o Buraco do Anel e a face que o Poste atravessou (+).

Finalmente, após a definição do algoritmo HoASP(MDP) e sua aplicação em Quebra-Cabeças Espaciais, o próximo capítulo apresentará os experimentos realizados para avaliar essa abordagem.

4 EXPERIMENTOS

O objetivo deste capítulo é avaliar como diferentes algoritmos de AR lidam com dois tipos de Quebra-Cabeças Espaciais (Fisherman's Folly e Rope Ladder) em diferentes configurações (Determinístico, Não-Determinístico e Não-Estacionário). Quatro diferentes algoritmos de AR foram aplicados nos diferentes quebra-cabeças. O algoritmo Q-Learning (SUTTON; BARTO, 2018), o oASP(MDP) original (FERREIRA et al., 2018), Q-Learning Acelerado por Heurística (HAQL) (BIANCHI; RIBEIRO; COSTA, 2008) e o HoASP(MDP), proposto nesta dissertação.

Com o intuito de realizar a comparação entre esses algoritmos, os parâmetros de AR utilizados em todos os experimentos foram fixados com os seguintes valores: fator de desconto de 0.9; ϵ (equilíbrio entre exploitar e explorar) ficou fixado até o episódio 4000 em 0.1, depois disso foi caindo com uma taxa de 0.01 a cada 250 episódios, até o valor de 0.03; a taxa de aprendizado foi de 0.2; o valor de controle da heurística foi de 0.25. Com relação às recompensas, se tem: -100 para ações impossíveis (aquelas que levam ao mesmo estado ou que são fisicamente impossíveis de serem executadas), 1000 para ações que levam ao estado objetivo e -1 para cada ação executada que não seja impossível e nem leve ao estado objetivo. O número de passos máximo em cada episódio é 500 e o número de episódios de cada experimento é 6000. Com o intuito de produzir resultados que sejam estatisticamente relevantes, 30 testes foram executados para cada experimento.

Ao todo, 7 diferentes configurações dos quebra-cabeças foram consideradas neste trabalho (detalhes na seção 2.1). Em todas essas configurações a Corda é sempre finita, sendo assim existe um limite de vezes que a corda pode atravessar o Buraco do Anel (6 vezes no Fisherman's Folly e no Rope Ladder) e o Buraco do Poste (6 vezes no Fisherman's Folly e 8 vezes no Rope Ladder) sem enrolar.

As seções a seguir descrevem cada um dos experimentos para cada configuração estudada.

4.1 FISHERMAN'S FOLLY ORIGINAL (FFO)

Essa é a configuração inicial do quebra-cabeça de Fisherman's Folly (figura 2a) apresentado na subseção 2.1.1. Todos os quatro algoritmos de AR estudados nesse trabalho foram aplicados nessa configuração com o intuito de mostrar como o HoASP(MDP) é a melhor abordagem para lidar com esse quebra-cabeça. Considerando os algoritmos que usam heurísticas, a tabela Q do Fisherman's Folly Simplificado (FFS) foi usada para guiar o processo de aprendizado.

4.2 FISHERMAN'S FOLLY SIMPLIFICADO (FFS)

Já que nessa configuração do quebra-cabeça não é permitido que o agente enrole a Corda através do Buraco do Poste (subseção 2.1.2), um dos objetivos desse experimento é verificar se o espaço de estado seria menor que o da configuração original (FFO) e, consequentemente, a quantidade de interações necessárias para solucionar o quebra-cabeça. Como essa é a configuração utilizada para obter as heurísticas, apenas os algoritmos sem aceleração por heurística foram aplicados nesse domínio, o oASP(MDP) original e o Q-Learning. A solução (tabela Q) dessa configuração foi utilizada como heurística para a configuração original do Fisherman's Folly (FFO).

4.3 FISHERMAN'S FOLLY NÃO-DETERMINÍSTICO (FFND)

Essa é a versão do quebra-cabeça apresentada na subseção 2.1.3 que conta com os mesmos elementos que a versão original do Fisherman's Folly (FFO), mas agora com não-determinismo na execução de ações. Apesar da política ótima para solução desse quebra-cabeça ser a mesma política do quebra-cabeça FFO, o processo de aprendizado da versão não-determinística fornece resultados diferentes que justificam seu estudo. Esse quebra-cabeça segue a regra de não-determinismo definida na subseção 2.1.3, em que há 80% de chance de uma determinada ação ser de fato executada, recebendo a recompensa esperada, já no caso dos 20% restantes, há 10% de chance da ação ser executada com a face oposta, recebendo uma recompensa negativa pequena (-1) e 10% de chance de não executar a ação e receber uma recompensa de -100. Todos os quatro algoritmos estudados nessa dissertação foram aplicados nesse domínio. Assim como no Fisherman's Folly Original (FFO), a solução do Fisherman's Folly Simplificado Não-Determinístico (FFSND) foi utilizada como heurística para aceleração do processo de aprendizado, já que este é o quebra-cabeça que conta com a mesma configuração do FFND.

4.4 FISHERMAN'S FOLLY SIMPLIFICADO NÃO-DETERMINÍSTICO (FFSND)

Nessa configuração (FFSND), que foi apresentada na subseção 2.1.4, o objetivo é mostrar como a simplificação no quebra-cabeça acelera o processo de aprendizado, através da diminuição do espaço de estados, mesmo nos casos não-determinísticos. Além disso, o quebra-cabeça FFSND fornece heurísticas para o quebra-cabeça FFND. Por conta desses objetivos, apenas a versão original do oASP(MDP) e Q-Learning foram aplicadas nessa configuração.

4.5 FISHERMAN'S FOLLY ESFERA NÃO-ESTACIONÁRIO (FFENE)

Nessa configuração (FFENE), que foi apresentada na subseção 2.1.5, e no começo do aprendizado (até o episódio 2000), o ambiente permite que a Esfera1 e a Esfera2 atravessem o Buraco do Poste. Depois do episódio 2000, o ambiente volta para a configuração do Fisherman's Folly Simplificado (FFS). Como há uma mudança no ambiente enquanto as interações acontecem, a política ótima também sofre uma mudança, até o episódio 2000 para solucionar o quebra-cabeça era necessário um conjunto de 3 ações[6], depois da mudança, a política ótima virou a mesma que a do Fisherman's Folly Simplificado (FFS), com 5 ações para a solução. Apenas os algoritmos oASP(MDP) original e Q-Learning foram executados, não houve a aplicação de nenhuma heurística nesse quebra-cabeça, já que esse domínio permite mudanças enquanto as interações ocorrem e não faz sentido aplicar as heurísticas em dois problemas diferentes[7]. Além disso, o objetivo de estudar os quebra-cabeças não-estacionários é avaliar como os algoritmos se comportam nesse ambiente (que permite mudança enquanto a interação acontece), mostrando como oASP(MDP) pode oferecer uma vantagem quando comparado com a versão tradicional do Q-Learning.

4.6 FISHERMAN'S FOLLY DISCO NÃO-ESTACIONÁRIO (FFDNE)

Nessa configuração (FFDNE), que foi apresentada na subseção 2.1.6, e no começo do aprendizado (até o episódio 2000), o ambiente permite que o Disco1 e o Disco2 possam atravessar o Anel, mas não possam atravessar o Buraco do Poste. A mudança acontece depois do episódio 2000, quando o quebra-cabeça volta a ser o Fisherman's Folly Simplificado (FFS) (em que os Discos não podem atravessar o Anel, mas podem atravessar o Buraco do Poste). As heurísticas também não foram aplicadas nesse quebra-cabeça pois há uma mudança no ambiente enquanto as interações acontecem, não fazendo sentido aplicar a mesma heurística para dois problemas diferentes. Apenas as versões originais do oASP(MDP) e Q-Learning foram executadas, já que não se utilizou de heurísticas para solucionar esse quebra-cabeça. Apesar do trabalho de Ferreira et al. (2017) apresentar a aplicação do algoritmo *offline* ASP(MDP)[8] para domínios não-estacionários e o quebra-cabeça anterior também ser não-estacionário, com essa configuração (FFDNE) o objetivo é avaliar qual é o comportamento do oASP(MDP), comparado com Q-Learning tradicional, quando se tem ambientes com mudanças em que uma política ótima aprendida anteriormente pode ser uma desvantagem para uma tarefa nova.

[6] O autor dessa dissertação contava com essa informação prévia por ter conhecimento sobre o quebra-cabeça.

[7] As mesmas heurísticas não aceleram o processo de aprendizado de dois quebra-cabeças diferentes. As heurísticas acelerariam um dos quebra-cabeças e seria uma desvantagem para o outro quebra-cabeça (antes ou depois da mudança ocorrer no ambiente).

[8] Versão anterior do algoritmo oASP(MDP), contando com um processo offline.

4.7 ROPE LADDER

Em termos de complexidade do quebra-cabeça, esse é um dos quebra-cabeças mais difíceis de serem resolvidos, já que há um maior número de ações para serem executadas e um maior número de elementos conectados de uma maneira mais desafiadora, sendo que nesse quebra-cabeça há sete elementos distintos, com dois Postes ligados por uma barra (subseção 2.1.7). Além disso, a Corda atravessa os Buracos dos Postes em quatro diferentes pontos. A política ótima para solução do Rope Ladder conta com a execução de 12 ações. Todos os quatro algoritmos de AR estudados nessa dissertação foram aplicados nesse domínio. Considerando os algoritmos que usam heurísticas, a tabela Q de uma versão simplificada (em que a Corda atravessa os Buracos dos Postes apenas uma vez) do Rope Ladder foi usada como guia para o processo de aprendizado. Uma restrição que foi aplicada para esse experimento é que o agente não pode atravessar o Disco1 através do Buraco do Poste2 e o Disco2 não pode atravessar o Buraco do Poste1. Outra restrição, presente nas configurações para obtenção de heurística e do Rope Ladder original, é que não é possível enrolar a Corda através do Buraco do Poste. Essas duas restrições têm o objetivo de excluir estados que não contribuem e são irrelevantes para a solução desse tipo de quebra-cabeça, servindo então para diminuir o tempo necessário para o aprendizado.

Portanto, para avaliar e comparar os algoritmos de AR (oASP(MDP), Q-Learning, HAQL e HoASP(MDP)) estudados, considerando os objetivos descritos acima de cada experimento, quatro diferentes medidas foram utilizadas. A primeira é o número de passos para completar o quebra-cabeça. Também foi avaliado o acúmulo dos valores de retorno. A terceira medida foi o número de estados visitados. Por último, foi obtido o número de par estado/ação presentes na tabela Q. Os resultados e a discussão da execução dos algoritmos de AR nos quebra-cabeças são apresentados no próximo capítulo.

5 RESULTADOS E DISCUSSÃO

Este capítulo apresenta os resultados e a discussão dos experimentos descritos no capítulo anterior, principalmente mostrando como Q-Learning e oASP(MDP) podem ser comparados com suas versões aceleradas por heurísticas, HAQL e HoASP(MDP).

Já que se tem as 7 configurações dos experimentos separados por seções (no capítulo 4 de Experimentos), manteve-se essa separação neste capítulo.

5.1 FISHERMAN'S FOLLY ORIGINAL (FFO)

Para o experimento com o quebra-cabeça Fisherman's Folly Original (FFO) (seção 4.1) todos os quatro algoritmos estudados (Q-Learning, oASP(MDP), HAQL e HoASP(MDP)) foram aplicados. A figura 9a mostra o gráfico com o Número de Passos para solucionar o quebra-cabeça, é possível observar que no começo do processo de aprendizado os algoritmos que são acelerados por heurísticas apresentam uma melhor performance do que suas versões não aceleradas, o teste T de Student na figura 11a mostra que essa diferença entre as curvas de aprendizado dos algoritmos é estatisticamente significativa. Portanto, é possível concluir que o uso da solução da versão simplificada do quebra-cabeça como heurística acelera o processo de aprendizado, isso acontece porque a solução do quebra-cabeça Fisherman's Folly Simplificado (FFS) é a solução para o FFO.

A figura 9b apresenta o gráfico para os Valores de Retorno, que também mostra uma performance melhor (valores de retorno mais altos) nos algoritmos acelerados por heurísticas. Isso ocorre porque a heurística serve como guia indicando quais ações devem ser executadas, não executando ações que possam ter recompensas negativas. Desse modo, o agente equipado com heurística precisa de menos interações para aprender a política ótima, que apresenta maiores valores de recompensa. Como cada episódio do treinamento conta com 500 interações e uma recompensa de -100 para cada ação que é impossível de ser executada fisicamente, inicialmente os valores de retorno são muito baixos, por volta de -35.000 a -40.000. Esses valores muito baixos ocorrem em todos os quebra-cabeças estudados.

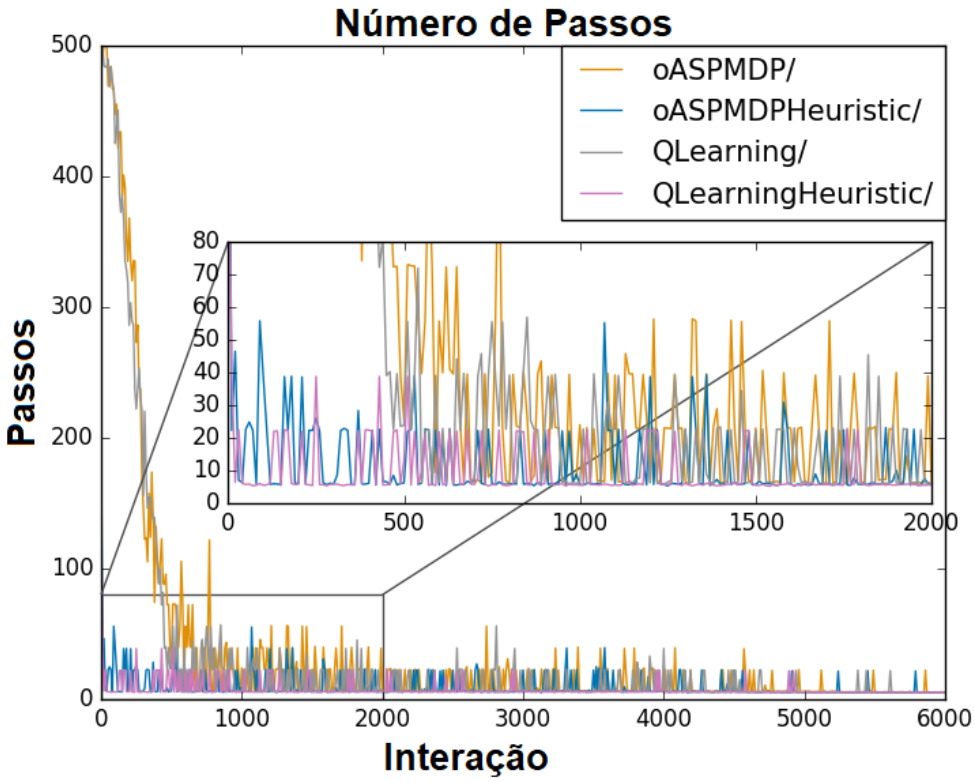
Já a figura 10a mostra o número de Estados Visitados pelo agente. É possível observar que o oASP(MDP) visita mais estados que o Q-Learning por conta da remoção das ações impossíveis, sendo esta remoção uma característica do algoritmo oASP(MDP). Além disso, os algoritmos que usam heurísticas exploram menos do espaço de estados que as versões não aceleradas por heurísticas. Isso ocorre porque com a heurística guiando o processo de aprendizado, o agente já conta com alguma informação sobre quais estados devem ser visitados, diminuindo assim o espaço de estados que precisa ser efetivamente visitado.

Por último, há a apresentação do número de pares Estado/Ações na figura 10b, o Q-Learning é o algoritmo com o maior número de pares Estado/Ações em sua tabela Q. Este último gráfico tem diferença quando se compara com o gráfico de Estados Visitados (figura 10a) porque

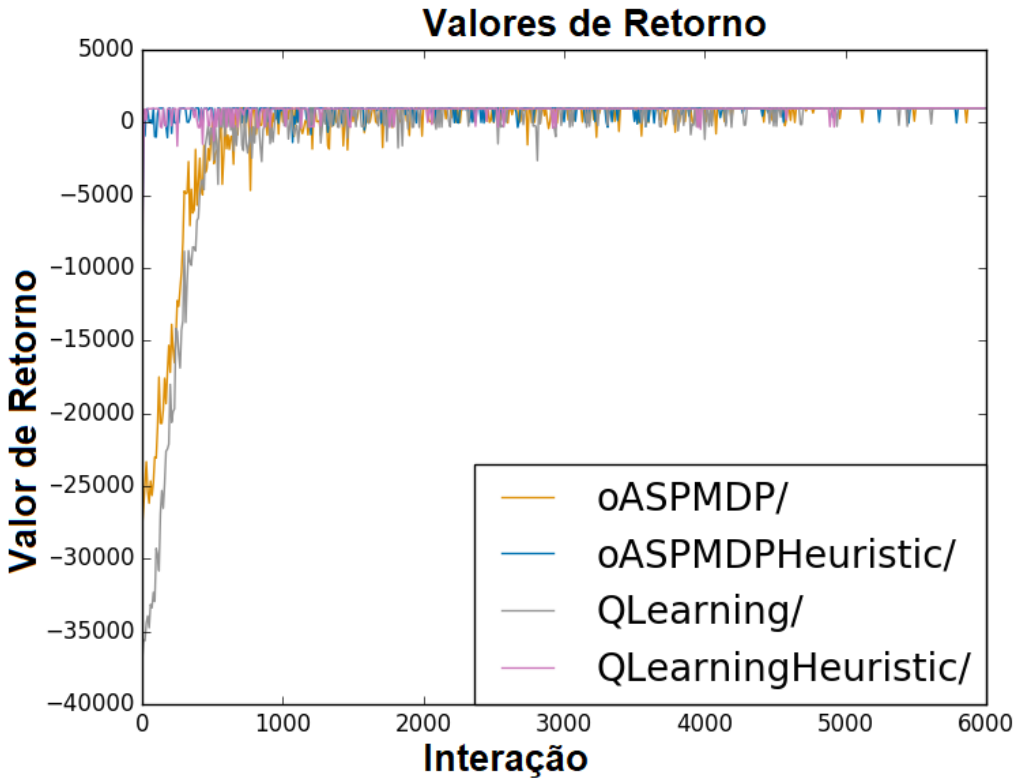
para o Q-Learning todas as ações já estão na tabela Q assim que o estado é visitado, diferente do oASP(MDP), que não adiciona na tabela Q as ações que são impossíveis de serem executadas fisicamente. Sendo assim, Q-Learning tem um número maior de pares Estado/Ações porque o agente não remove as ações impossíveis da tabela Q (Q-Learning apenas atribui recompensas negativas para essas ações), fazendo com que a combinação de pares seja maior do que quando há a remoção das ações, como no caso do oASP(MDP).

Figura 9 – Número de Passos e Valores de Retorno para o Fisherman's Folly Original

(a) Número de Passos para resolver o quebra-cabeça



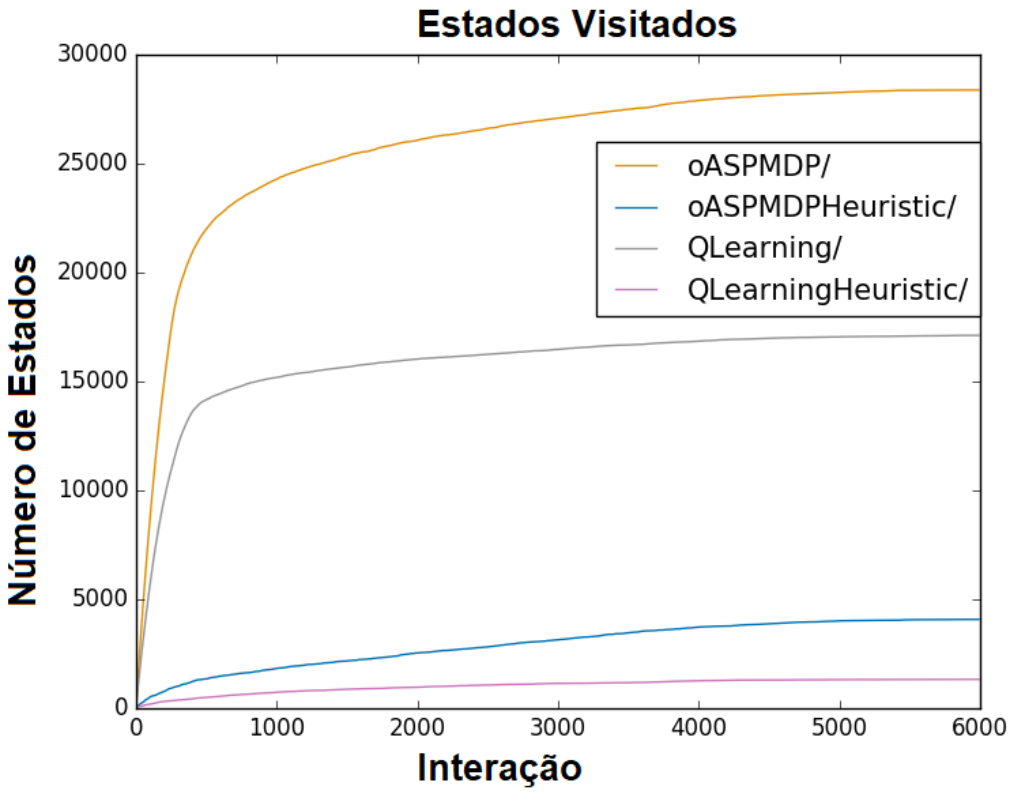
(b) Valor de Retorno recebido por episódio



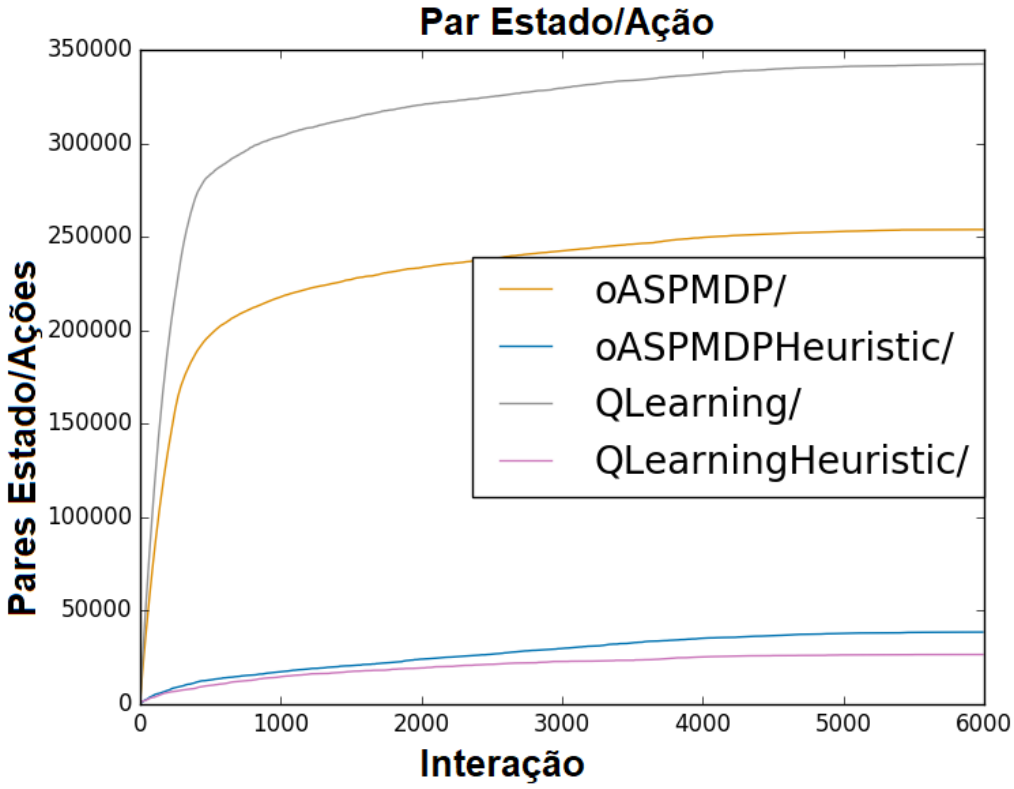
Fonte: Autor

Figura 10 – Resultados de Estados para o Fisherman's Folly Original

(a) Número de Estados Visitados pelo agente



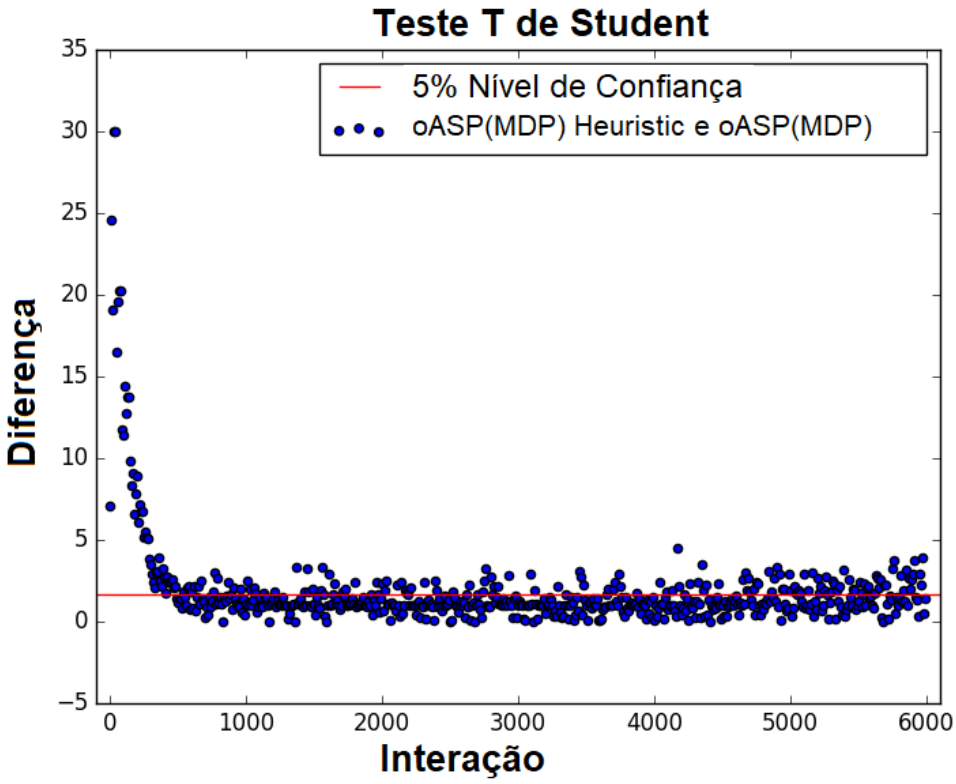
(b) Número de pares Estado/Ações



Fonte: Autor

Figura 11 – Testes Estatísticos para o Fisherman's Folly Original

(a) Teste T de Student comparando o oASP(MDP) original com o HoASP(MDP)



Fonte: Autor

5.2 FISHERMAN'S FOLLY SIMPLIFICADO (FFS)

Como descrito no capítulo de Experimentos (seção 4.2), para o quebra-cabeça Fisherman's Folly Simplificado (FFS), apenas dois algoritmos foram avaliados: oASP(MDP) e o Q-Learning, já que é a solução desse domínio que fornece as heurísticas para o quebra-cabeça original (FFO).

A figura 12a mostra o gráfico que representa o Número de Passos para resolver o quebra-cabeça, é possível observar como as curvas de aprendizado para ambos os algoritmos são parecidas, esse tipo de comportamento é esperado porque o oASP(MDP) usa o Q-Learning como o algoritmo de aprendizado.

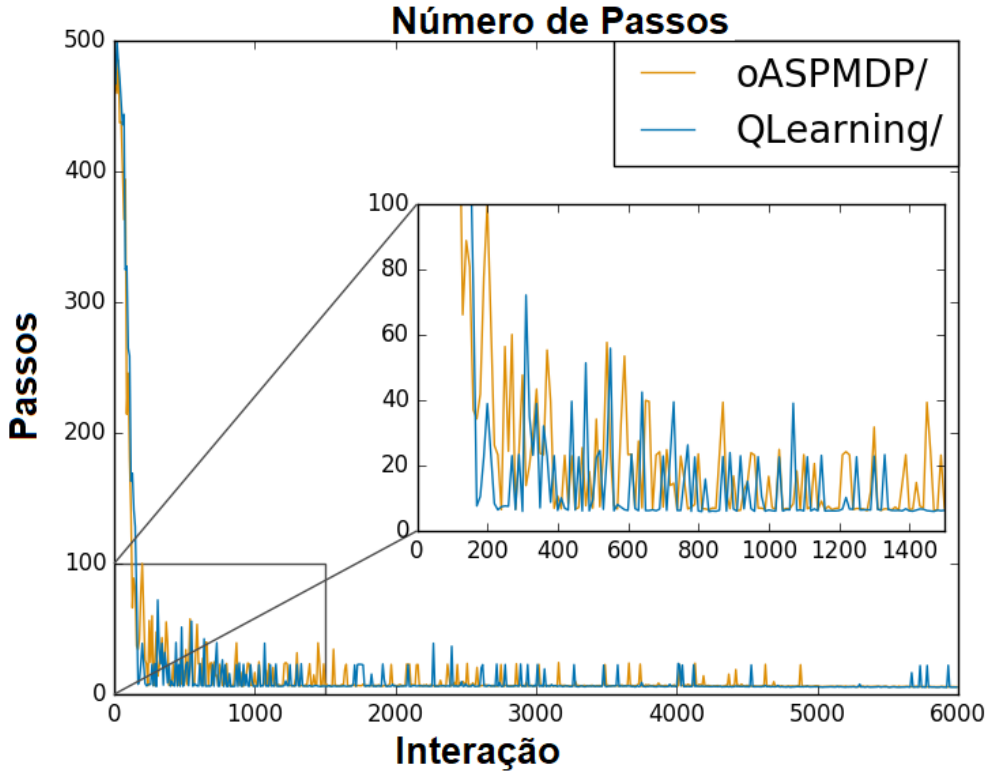
Já a figura 12b apresenta o gráfico dos Valores de Retorno, mostrando como as curvas para ambos os algoritmos (Q-Learning e oASP(MDP)) são similares novamente. Apesar dessa similaridade, há uma diferença em como os algoritmos se comportam, já que o oASP(MDP) é capaz de remover ações que são impossíveis (aquelas ações que não podem ser executadas no quebra-cabeça e recebem valores absolutos altos como recompensa negativa), diferente do Q-Learning que mantém essas ações impossíveis na tabela Q e pode até executá-las nos estados visitados, recebendo recompensas negativas.

A figura 13a mostra o número de Estados Visitados, sendo que o agente oASP(MDP) visita mais estados que o Q-Learning. Como o oASP(MDP) não leva em consideração as ações impossíveis, o agente acaba por executar mais ações que levam a diferentes estados, explorando assim mais do espaço de estados. Já o Q-Learning pode executar ações que são impossíveis e, consequentemente, acaba por explorar menos o ambiente, destacando que ações impossíveis não levam a novos estados. Outro ponto de destaque é que observando o número de interações necessárias para aprender a política ótima (figura 12a) e também o número de Estados Visitados (figura 13a) é possível concluir que a restrição de não enrolar a Corda através do Buraco do Poste diminui a complexidade para encontrar a solução do quebra-cabeça. Para verificar isso deve-se comparar essa versão do quebra-cabeça com a versão Original (FFO) (figura 9a e figura 10a da seção 5.1, em que é possível enrolar a Corda através do Buraco do Poste até duas vezes). Com relação ao espaço de estados, com o quebra-cabeça simplificado FFS, o oASP(MDP) visita cerca de 6000 estados, enquanto que com o quebra-cabeça FFO, o agente oASP(MDP) visita cerca de 25000 estados.

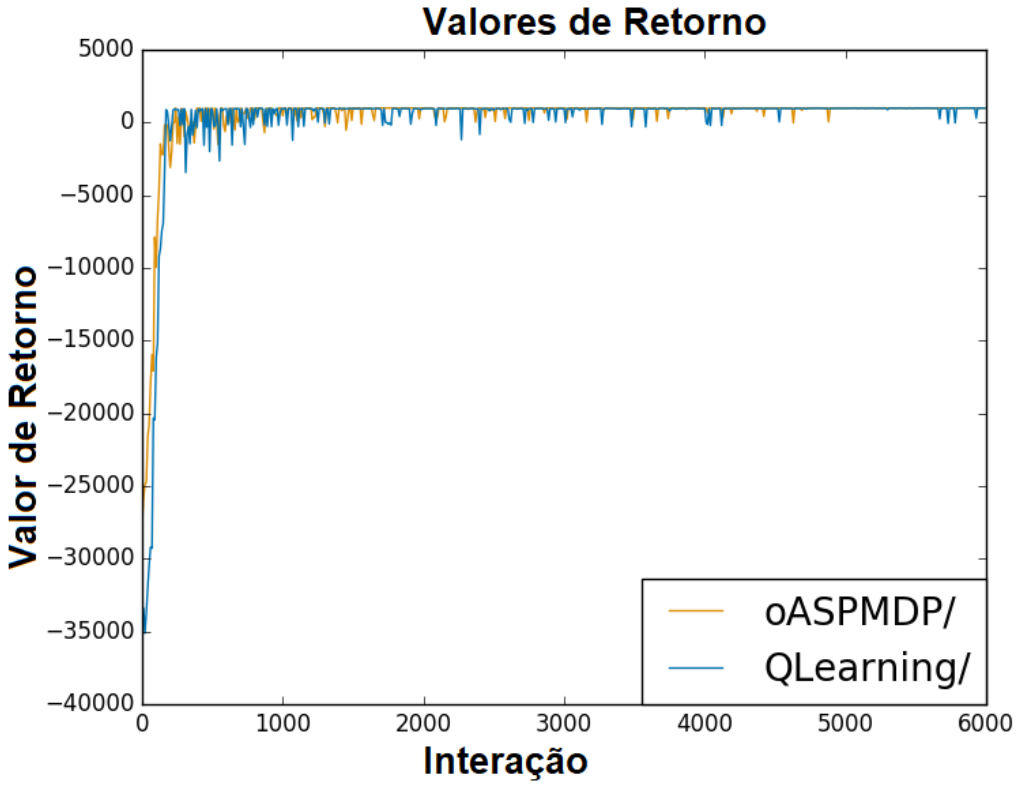
Por último, a figura 13b mostra o número de pares Estado/Ações presentes na tabela Q. O Q-Learning tem mais pares por não remover ações que são impossíveis da sua tabela Q.

Figura 12 – Número de Passos e Valores de Retorno para o Fisherman's Folly Simplificado

(a) Número de Passos para resolver o quebra-cabeça



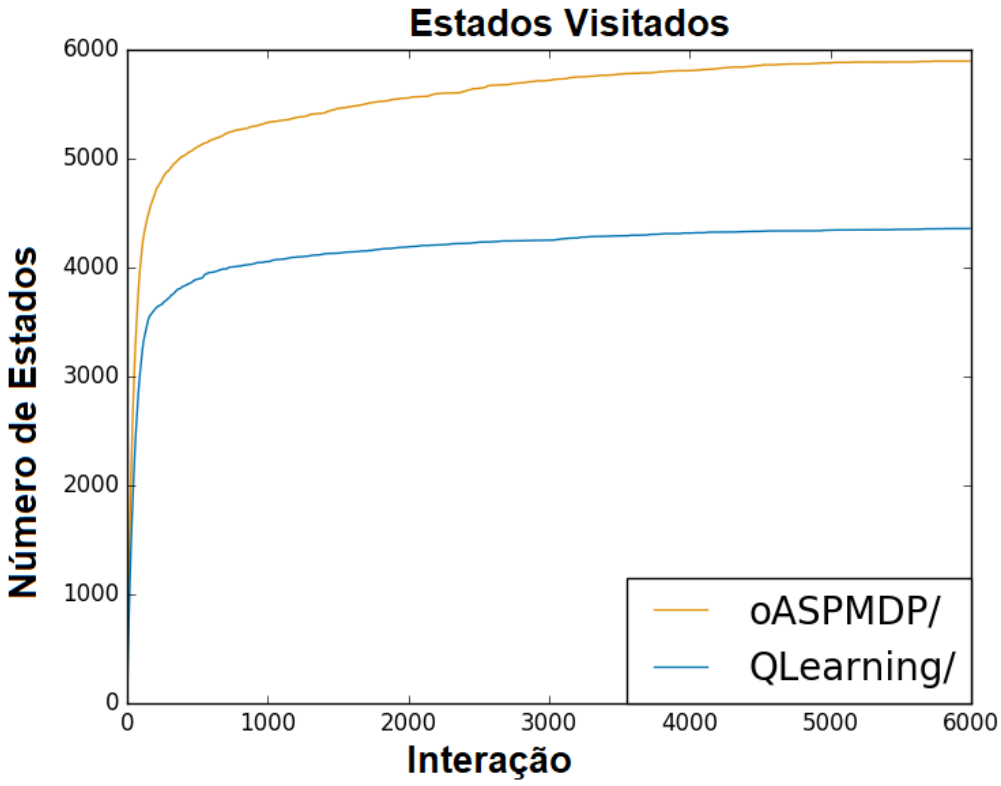
(b) Valor de Retorno recebido por episódio



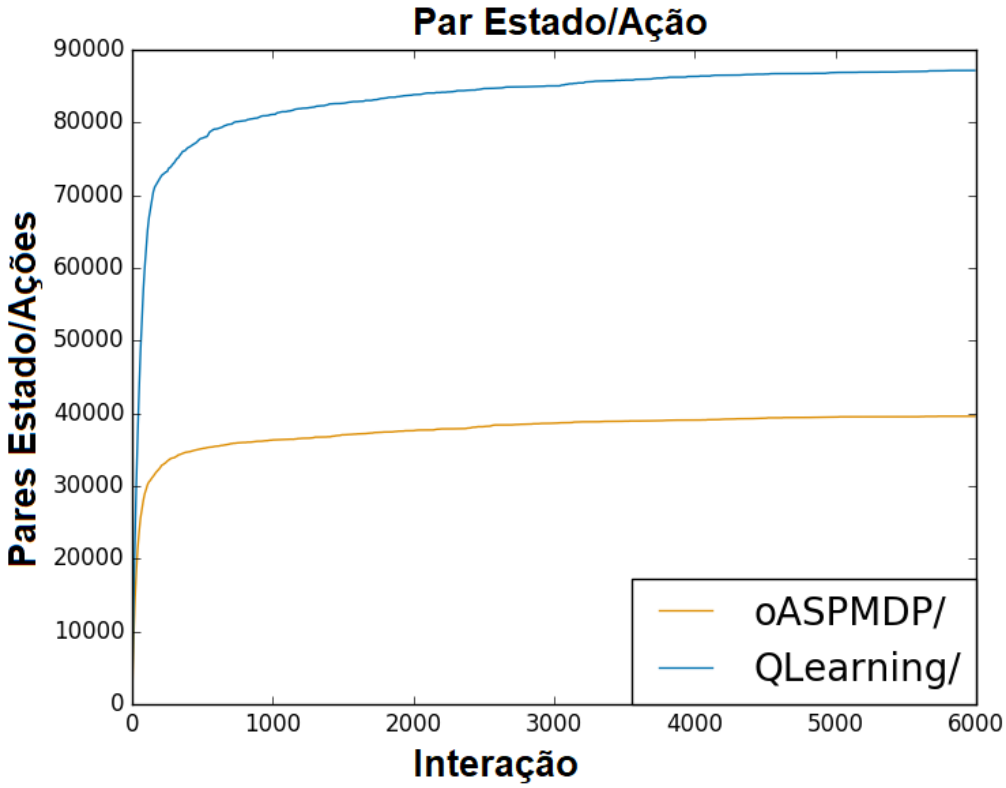
Fonte: Autor

Figura 13 – Resultados de Estados para o Fisherman's Folly Simplificado

(a) Número de Estados Visitados pelo agente



(b) Número de pares Estado/Ações



Fonte: Autor

5.3 FISHERMAN'S FOLLY NÃO-DETERMINÍSTICO (FFND)

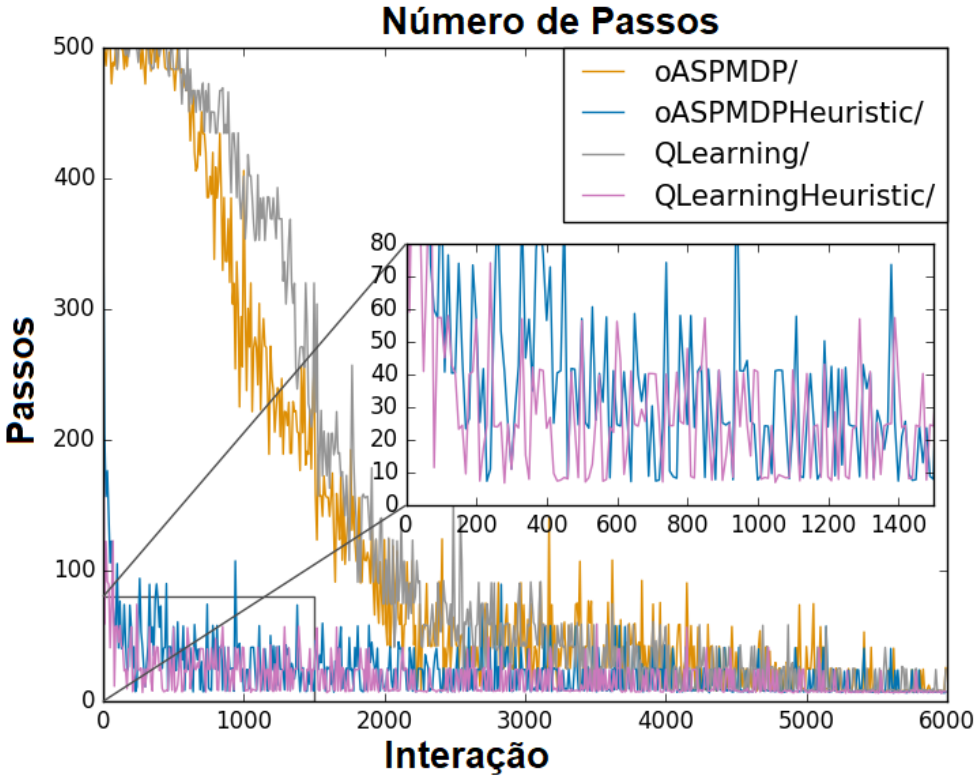
Para os experimentos com o quebra-cabeça Fisherman's Folly Não-Determinístico (FFND) (seção 4.3) todos os quatro algoritmos (Q-Learning, oASP(MDP), HAQL, HoASP(MDP)) foram executados. O resultado sobre o Número de Passos para solucionar o quebra-cabeça está na figura 14a. Nesta figura é possível observar que no começo do processo de aprendizado os algoritmos acelerados por heurísticas apresentam uma melhor performance que os algoritmos não acelerados, o teste T de Student na figura 16a mostra que essa é uma diferença com relevância estatística.

A figura 14b mostra os Valores de Retorno, novamente os algoritmos acelerados por heurísticas apresentam valores melhores no começo do processo. Um ponto de consideração é que o oASP(MDP) tem maiores valores de retorno do que Q-Learning no começo do processo, mas conforme o número de interações aumenta, essa diferença cai até os dois algoritmos estarem no mesmo nível.

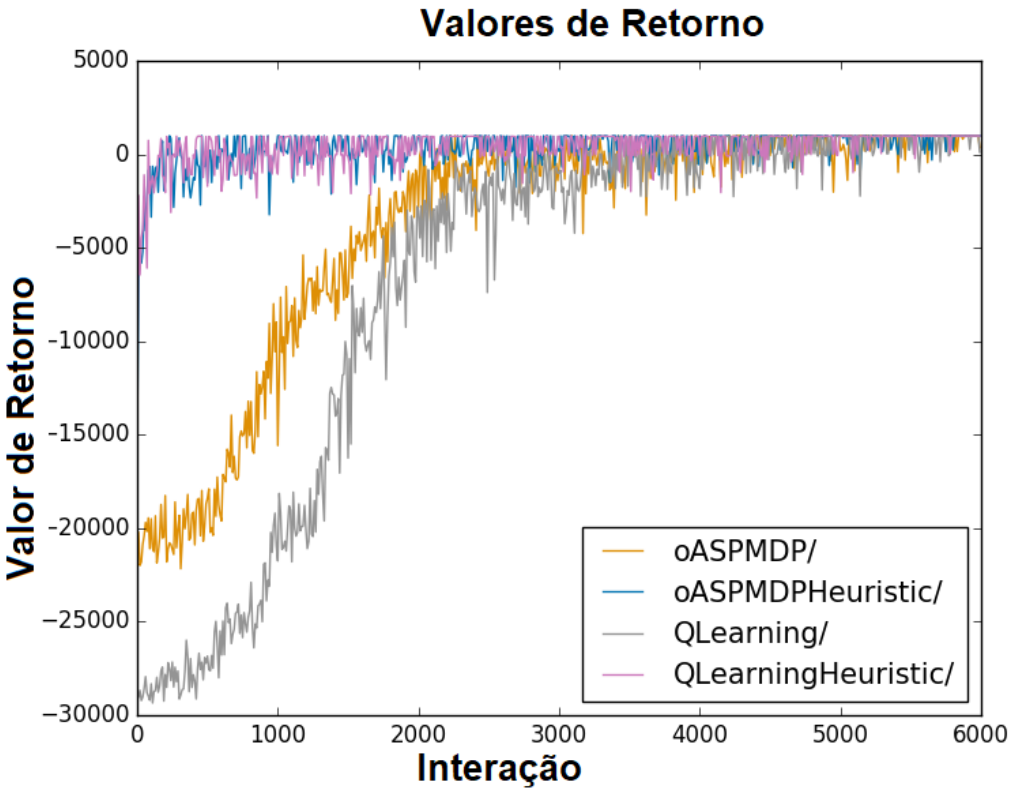
O gráfico na figura 15a mostra o número de Estados Visitados, apresentando como oASP(MDP) explora mais dos estados válidos que os outros algoritmos, enquanto a figura 15b mostra que Q-Learning tem mais pares Estado/Ações na tabela Q. Como nessa configuração de quebra-cabeça é possível enrolar (até duas vezes por estado) a Corda através do Buraco do Poste, o agente oASP(MDP) explora ainda mais do espaço de estados que todas as outras configurações dos quebra-cabeças (figura 15a), visitando por volta de 80000 estados. Esse comportamento faz com que o algoritmo leve mais tempo para aprender a política ótima (Figure 14a). Apesar desse problema ser Não-Determinístico, os algoritmos acelerados por heurísticas ainda foram capazes de acelerar o processo de aprendizado, já que a influência da heurística só é considerada no momento da escolha da ação. Portanto, mesmo que o resultado da execução da ação seja o caso do não-determinismo, a recompensa negativa não é capaz de ser acumulada com valores suficientes para contrapor a influência da heurística quando esta indica uma determinada ação.

Figura 14 – Número de Passos e Valores de Retorno para o Fisherman's Folly Original Não-Determinístico

(a) Número de Passos para resolver o quebra-cabeça



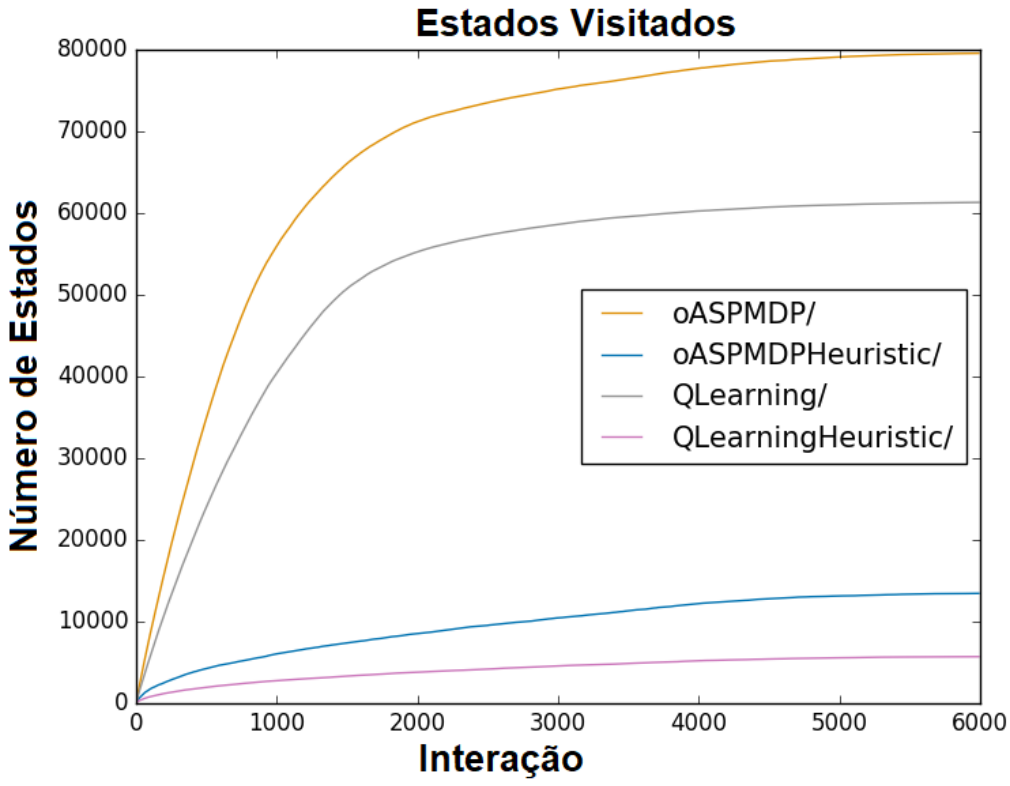
(b) Valor de Retorno recebido por episódio



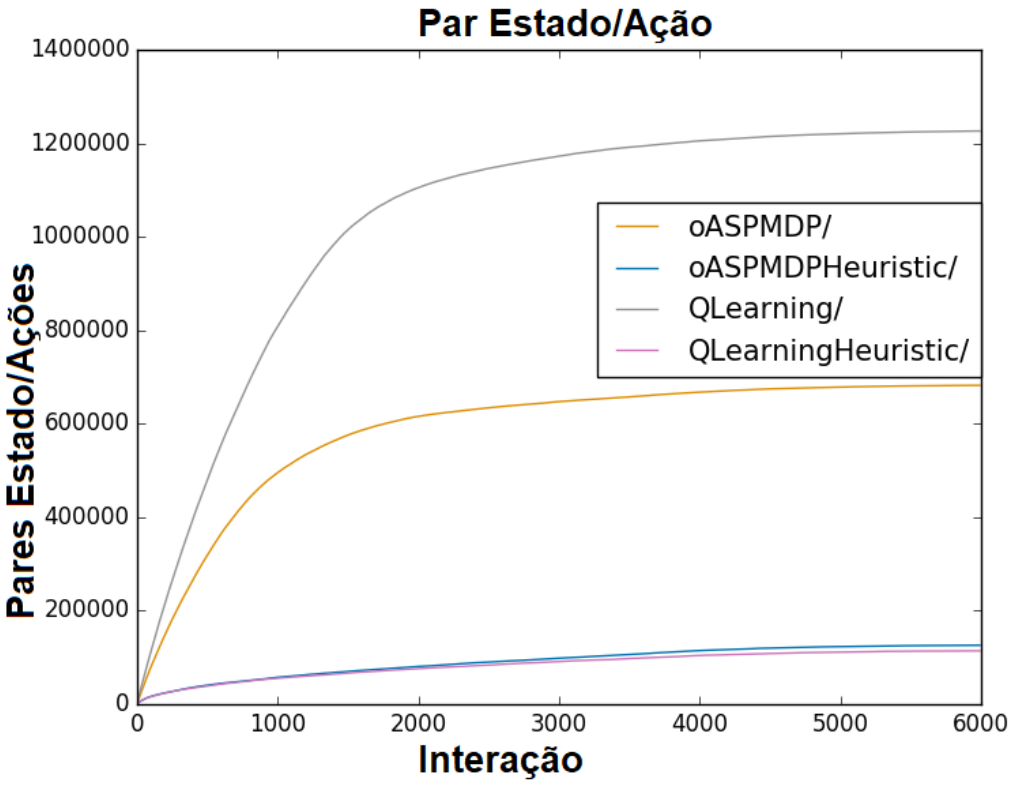
Fonte: Autor

Figura 15 – Resultados de Estados para o Fisherman's Folly Não-Determinístico

(a) Número de Estados Visitados pelo agente



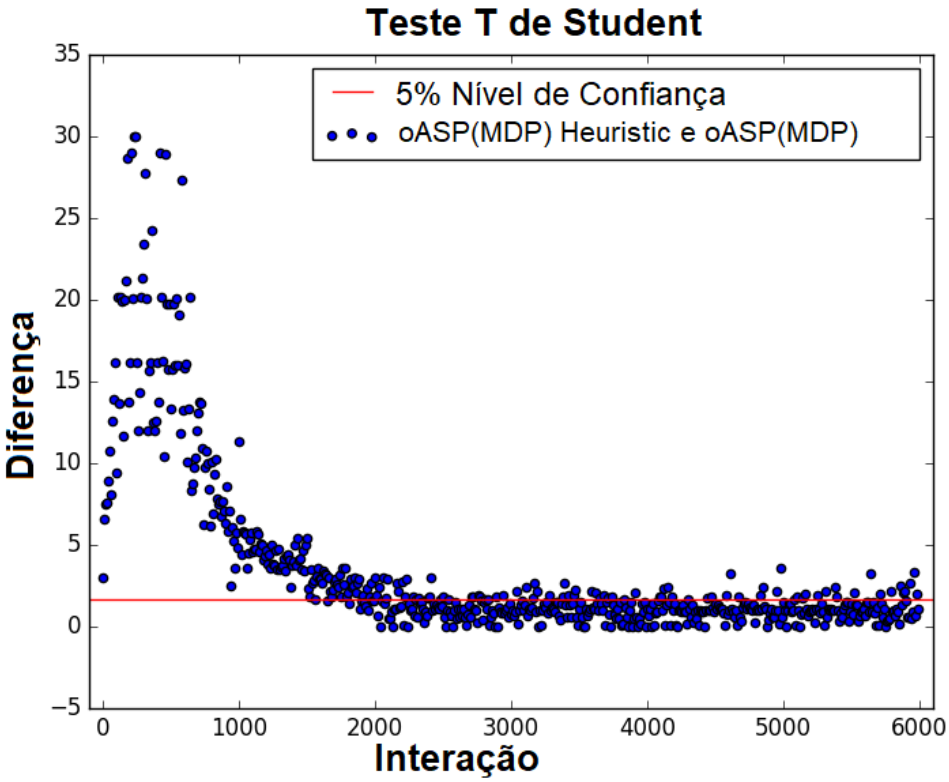
(b) Número de pares Estado/Ações



Fonte: Autor

Figura 16 – Testes Estatísticos para o Fisherman's Folly Original Não-Determinístico

(a) Teste T de Student comparando o oASP(MDP) original com o HoASP(MDP)



Fonte: Autor

5.4 FISHERMAN'S FOLLY SIMPLIFICADO NÃO-DETERMINÍSTICO (FFSND)

Ainda considerando que as ações têm efeitos não-determinísticos, os resultados do quarto quebra-cabeça estão relacionados com os experimentos para o Fisherman's Folly Simplificado Não-Determinístico (seção 4.4), em que não é possível enrolar a corda através do Buraco do Poste. Nessa configuração, apenas os algoritmos Q-Learning e oASP(MDP) foram executados, pois a solução desse quebra-cabeça é usada como heurística para o Fisherman's Folly Não-Determinístico (FFND) (seção 4.3).

A figura 17a apresenta o gráfico para o Número de Passos para solução do quebra-cabeça e também mostra como as curvas para os dois algoritmos são similares (as curvas não têm uma diferença significativa estatisticamente). Já a figura 17b apresenta os Valores de Retorno, também mostrando como as duas curvas não apresentam diferenças relevantes.

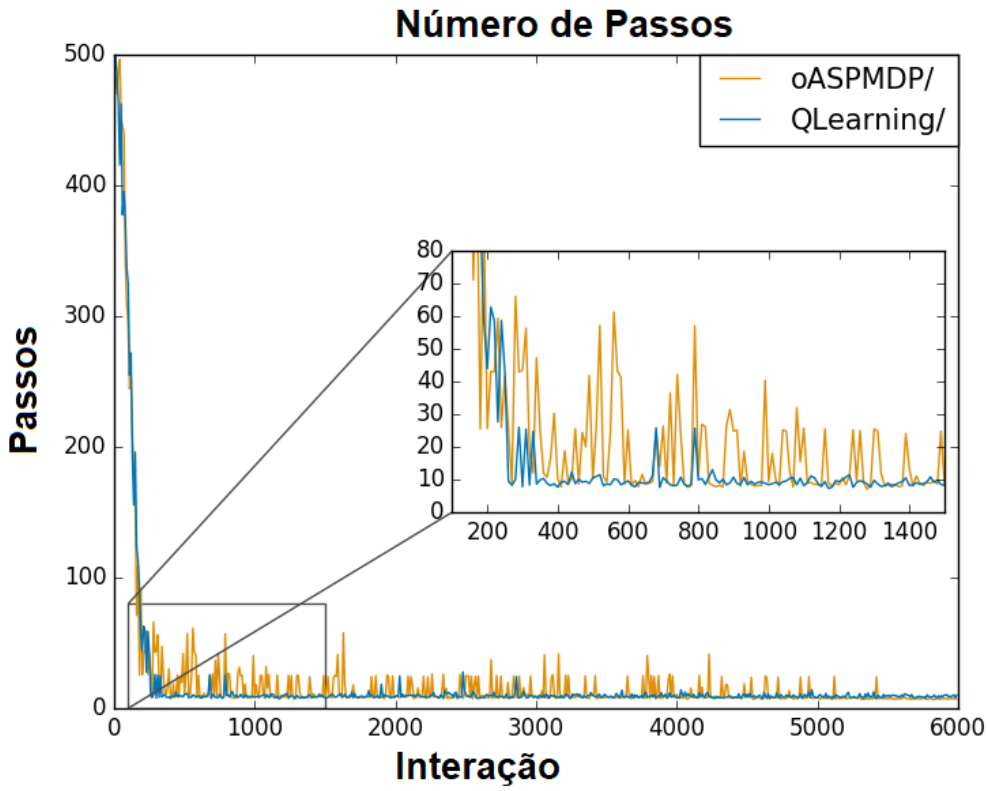
Para o Número de Estados Visitados, tem-se a figura 18a mostrando que oASP(MDP) explora mais do espaço de estados, já que executa mais ações que levam a estados novos.

Finalmente, a figura 18b apresenta o número de pares Estado/Ações presentes na tabela Q, o algoritmo Q-Learning tem o maior número de pares. Isso ocorre porque o Q-Learning mantém ações impossíveis em sua tabela Q.

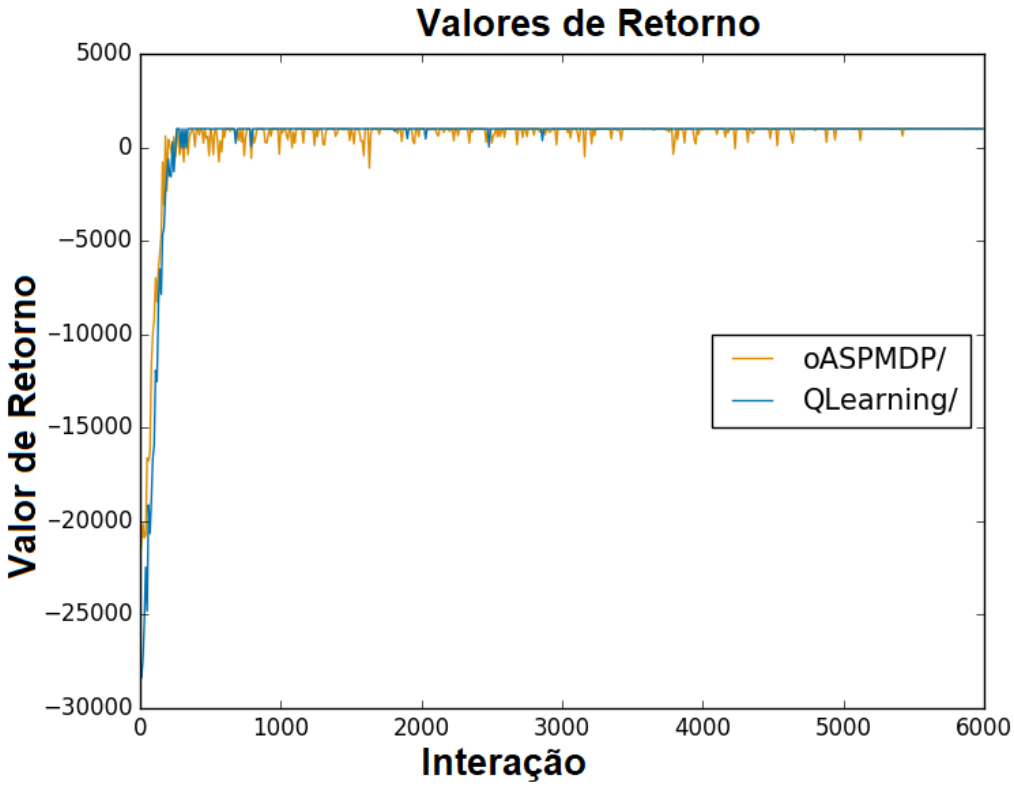
Apesar de essa ser uma versão simplificada do Fisherman's Folly Não-Determinístico, os resultados são similares aos resultados do quebra-cabeça Fisherman's Folly Simplificado (FFS), isso acontece porque quando se utiliza a simplificação do quebra-cabeça, o processo de aprendizado ocorre de uma maneira muito rápida, portanto o não-determinismo acaba por não ter influência relevante no processo de aprendizado como um todo. Em outras palavras, quando se remove a possibilidade de enrolar a Corda através do Buraco do Poste, o espaço de estados do quebra-cabeça é pequeno o suficiente para levar ao aprendizado da política ótima de uma forma rápida (cerca de apenas 200 interações), sem grande exploração dos estados e ações possíveis, o que acaba por diminuir a influência do não-determinismo para solução do problema.

Figura 17 – Número de Passos e Valores de Retorno para o Fisherman's Folly Simplificado Não-Determinístico

(a) Número de Passos para resolver o quebra-cabeça



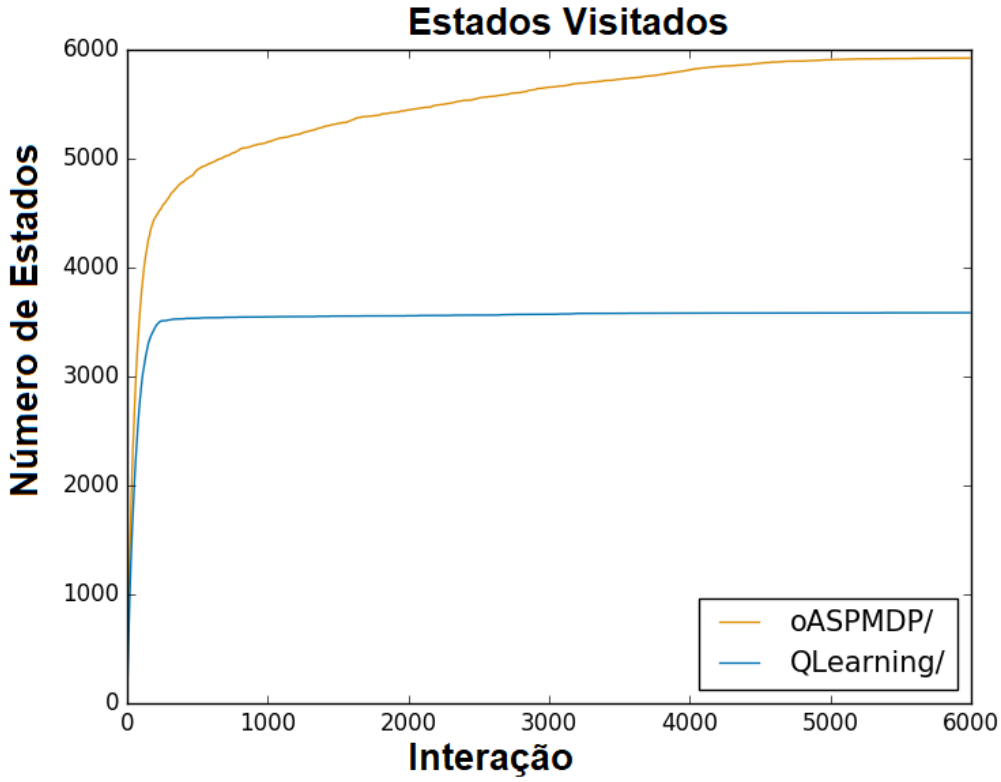
(b) Valor de Retorno recebido por episódio



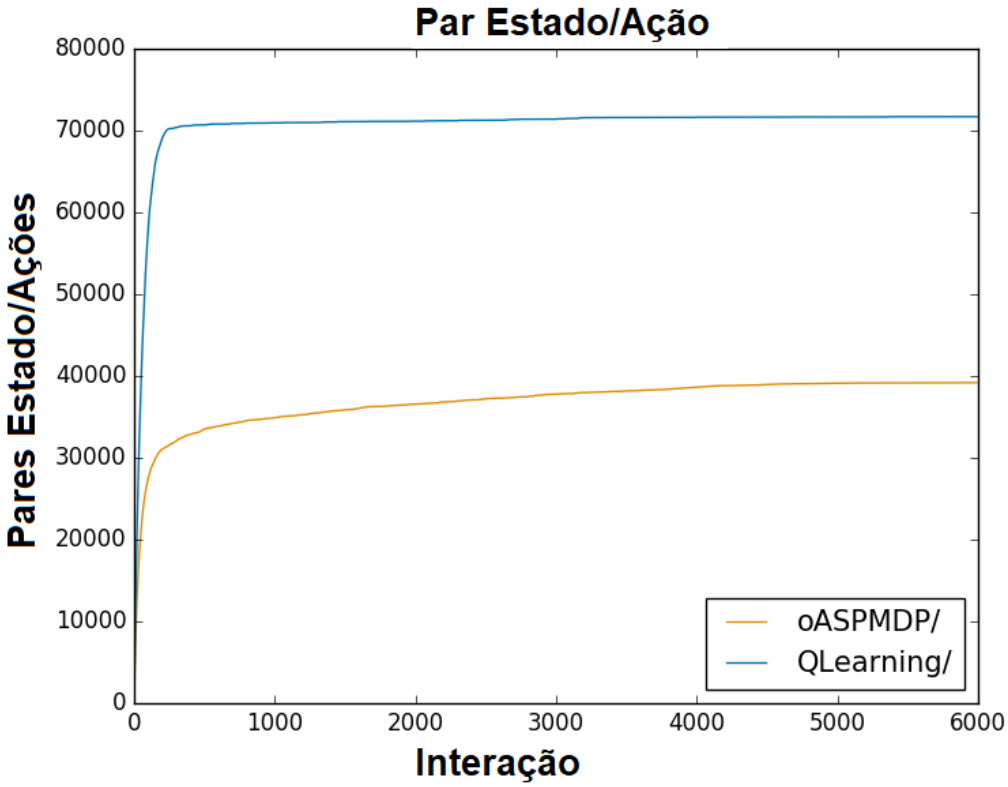
Fonte: Autor

Figura 18 – Resultados de Estados para o Quebra-Cabeça Fisherman's Folly Simplificado Não-Determinístico

(a) Número de Estados Visitados pelo agente



(b) Número de pares Estado/Ações



Fonte: Autor

5.5 FISHERMAN'S FOLLY ESFERA NÃO-ESTACIONÁRIO (FFENE)

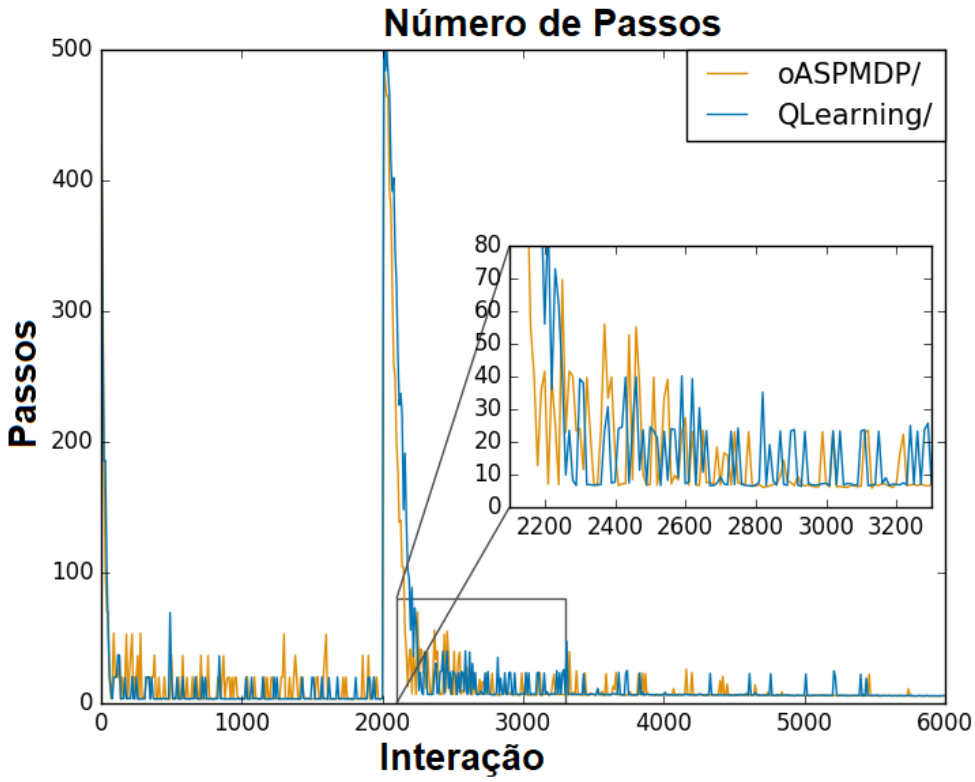
Com a possibilidade de adicionar mudanças no domínio enquanto a interação acontece, os resultados dos experimentos para o quebra-cabeça Fisherman's Folly Esfera Não-Estacionário (FFENE) (seção 4.5) são apresentados. Como descrito na seção 4.5, no FFENE apenas dois algoritmos (Q-Learning e oASP(MDP)) foram executados.

Na figura 19a é possível observar que ambos os algoritmos têm uma performance muito similar, não apresentando diferenças significativas, mesmo após a mudança ocorrida no ambiente. Isso ocorre porque a política ótima encontrada no problema antes da mudança não faz parte da política ótima depois da mudança. Portanto, na prática, o agente oASP(MDP) deve aprender uma nova política para o novo ambiente, do mesmo modo que o Q-Learning (que reinicializa todos os parâmetros quando uma mudança é detectada) também tem que aprender, sem o uso de conhecimento da política ótima anterior. A vantagem do oASP(MDP) é que esse agente aprende quais ações são impossíveis e então o agente não precisa executar essas ações novamente, diferente do que acontece com Q-Learning, que não reutiliza esse conhecimento sobre ações impossíveis.

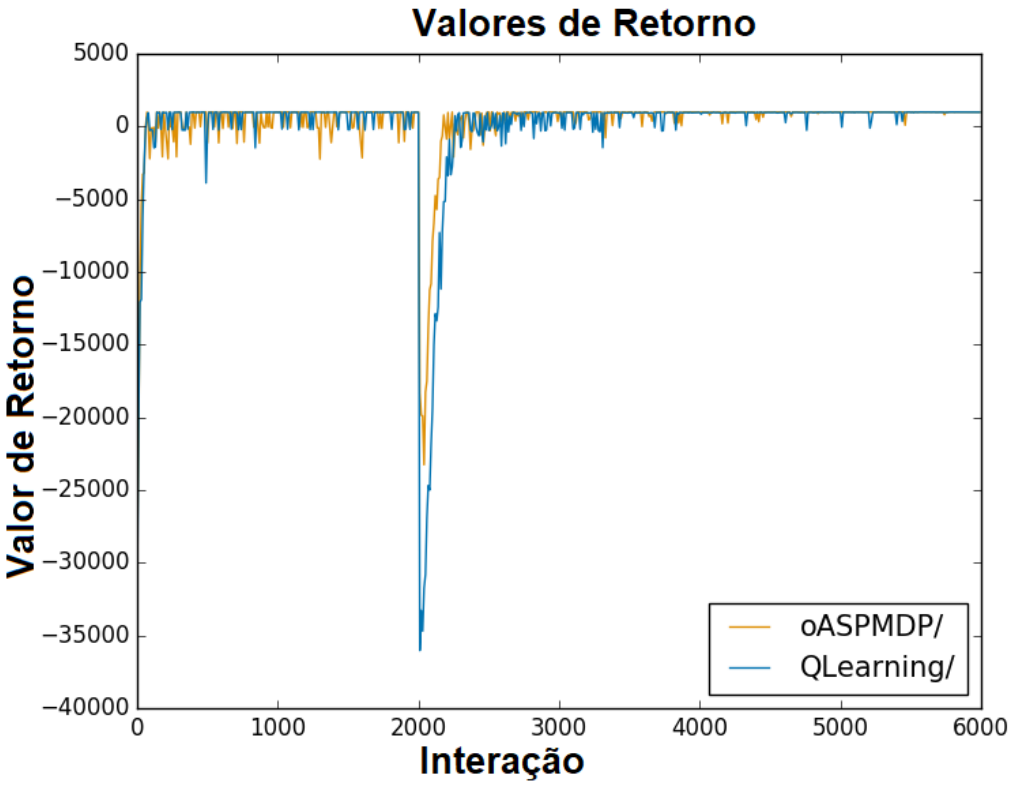
O gráfico para os Valores de Retorno (figura 19b) também apresenta similaridade entre oASP(MDP) e Q-Learning. As mudanças relevantes começam a ser vistas nos gráficos referentes aos Estados Visitados (figura 20a) e pares Estado/Ações (figura 20b). O oASP(MDP) visita mais estados e não reinicializa os valores da tabela Q, com isso o número de estados visitados não sofre uma queda no seu valor (figura 20a) o que também acontece para o caso dos pares Estado/Ações (figura 20b). Em contraste, o Q-Learning reinicializa todos os parâmetros obtidos antes da mudança (no episódio 2000) e portanto apresenta uma queda em seus valores (seguido de uma subida nesses números).

Figura 19 – Número de Passos e Valores de Retorno para o Fisherman's Folly Esfera Não-Estacionáiro

(a) Número de Passos para resolver o quebra-cabeça



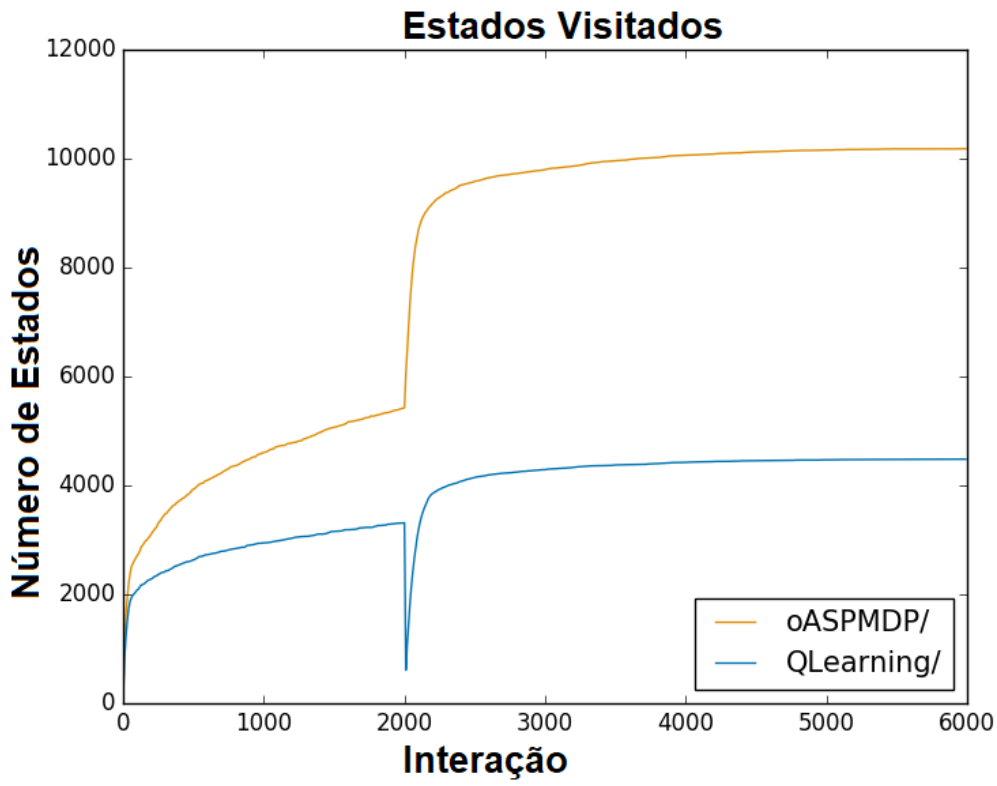
(b) Valor de Retorno recebido por episódio



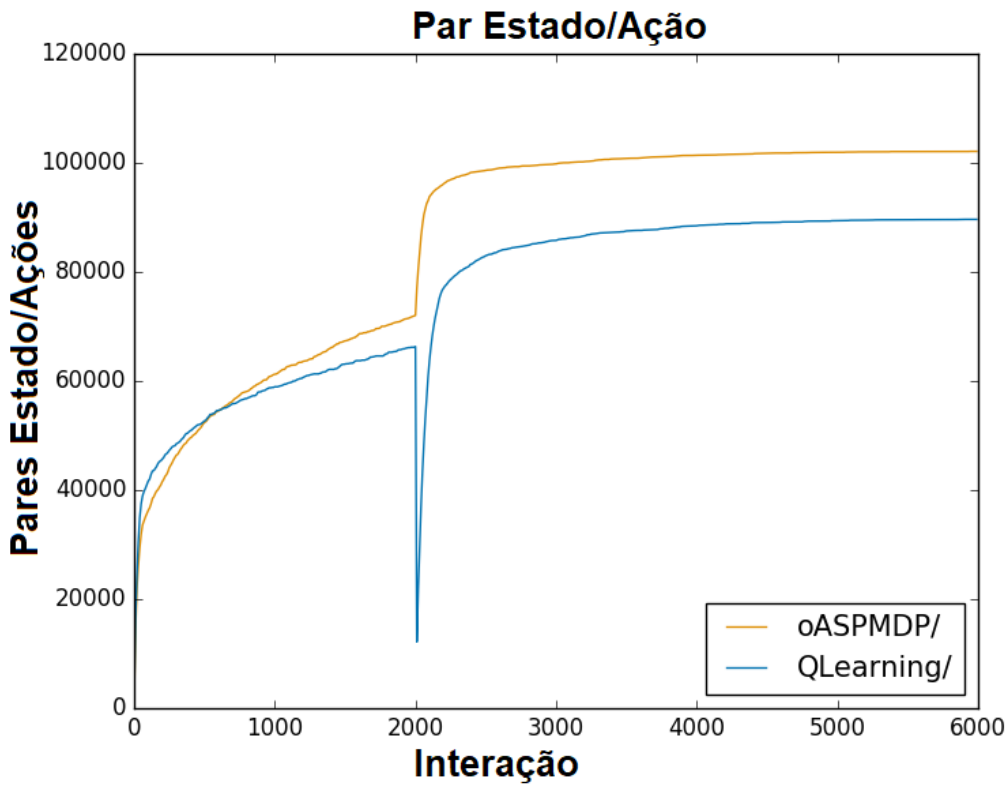
Fonte: Autor

Figura 20 – Resultados de Estados para o Fisherman's Folly Esfera Não-Estacionário

(a) Número de Estados Visitados pelo agente



(b) Número de pares Estado/Ações



Fonte: Autor

5.6 FISHERMAN'S FOLLY DISCO NÃO-ESTACIONÁRIO (FFDNE)

Partindo agora para outro caso Não-Estacionário, os resultados para os experimentos do quebra-cabeça Fisherman's Folly Disco Não-Estacionário (FFDNE), em que apenas dois algoritmos (Q-Learning e oASP(MDP)) foram executados (seção 4.6).

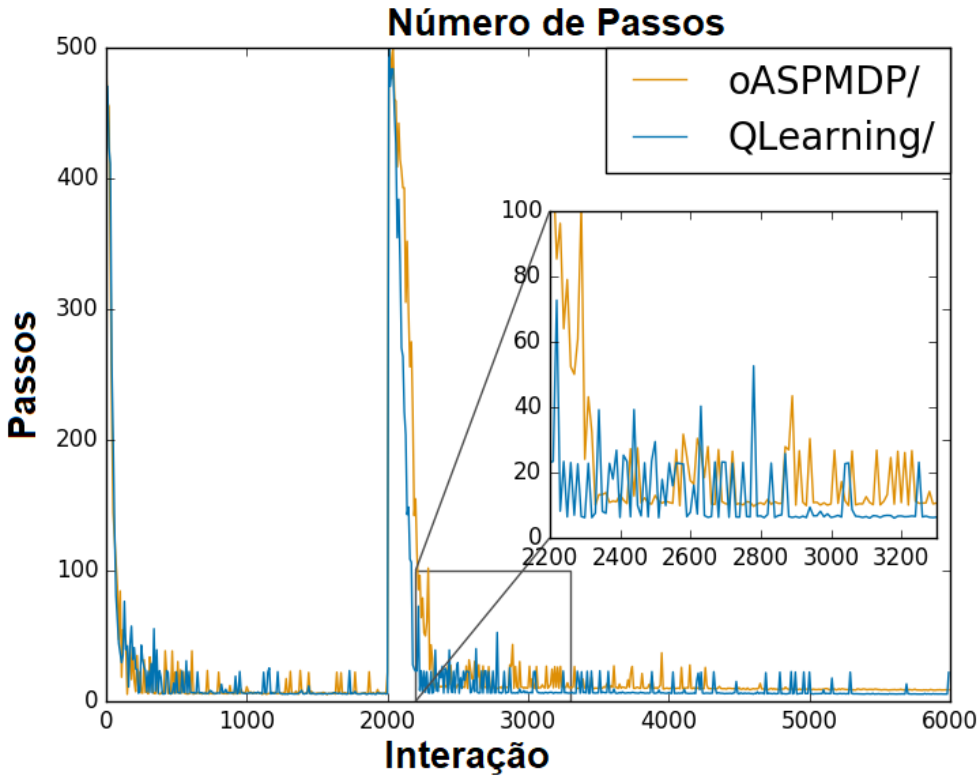
A figura 21a mostra que os dois algoritmos apresentam performance similar até o momento da mudança no ambiente (no episódio 2000). Depois dessa mudança, o Q-Learning passa a ter uma performance melhor que o oASP(MDP), já que o último converge apenas para uma política sub-ótima. A diferença na performance é estatisticamente relevante como mostrado pelo teste T de Student na figura 23a. Nessa configuração, o modo como a mudança ocorre no ambiente acaba por impactar a performance do agente oASP(MDP). Pois este agente aprendeu uma política ótima para o domínio antes da mudança que é uma desvantagem no domínio depois da mudança (após a interação 2000), considerando que o oASP(MDP) obteve apenas uma política sub-ótima, diferente do Q-Learning (figura 21a). Em outras palavras, primeiro o agente oASP(MDP) aprende uma política ótima para resolver o domínio antes da mudança; segundo, depois que a mudança é detectada, a tabela Q do oASP(MDP) não é reinicializada, portanto a política ótima aprendida anteriormente influencia negativamente o comportamento do agente no novo ambiente, sendo que essa política ótima anterior não é parte da política ótima do novo ambiente. Como o agente oASP(MDP) não recebe recompensas negativas suficientes para contrapor essas ações aprendidas, o agente acaba executando ações que não são ótimas (depois da mudança) e, consequentemente, o algoritmo apresenta uma convergência para uma política sub-ótima na segunda parte do ambiente.

A figura 21b mostra os Valores de Retorno, nesse caso não há diferenças significativas. Apesar da diferença no Número de Passos (figura 21a) para solucionar o quebra-cabeça depois da mudança ocorrer, essa diferença não pode ser observada nos Valores de Retorno (Figure 21b), isso ocorre porque a recompensa por cada ação tomada é pequena (-1) quando comparada com a recompensa para quando o objetivo é alcançado (+1000).

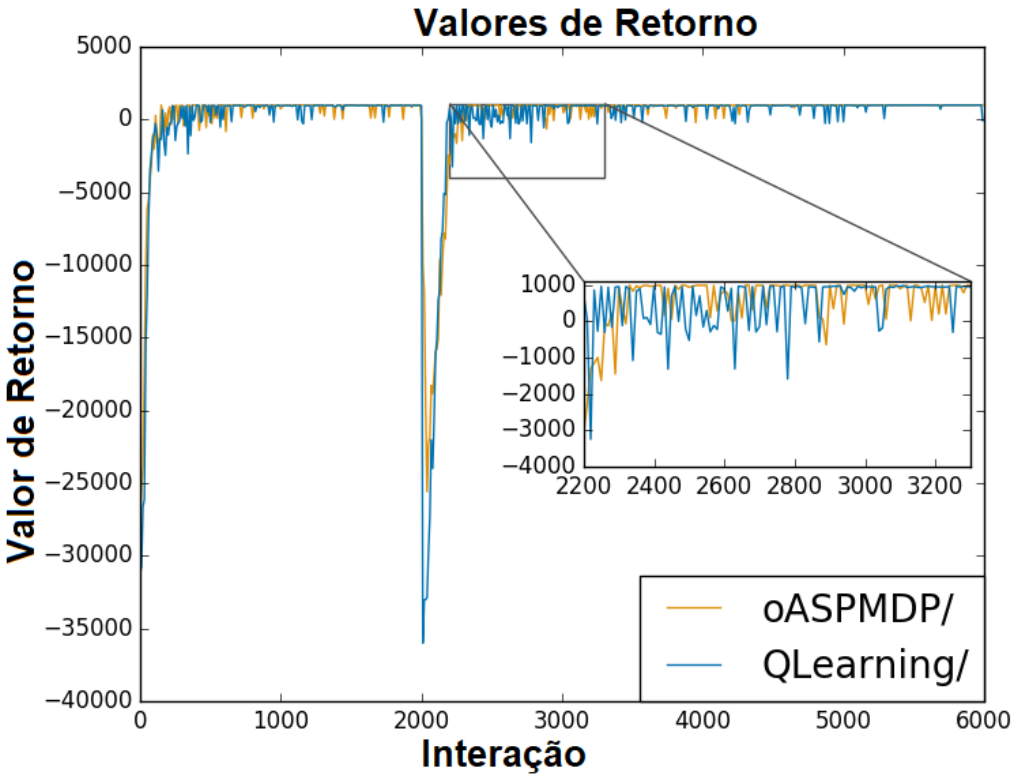
O comportamento relacionado com o número de Estados Visitados é mostrado na figura 22a, apresentando como o oASP(MDP) mantém sua tabela Q, mesmo após a mudança no ambiente, enquanto Q-Learning precisa reinicializar esses valores. A figura 22b mostra que o Q-Learning ainda tem um maior número de pares Estado/Ações, mesmo que esse algoritmo precise reinicializar sua tabela Q. O número de pares Estado/Ações apenas aumentam para o oASP(MDP) porque a tabela Q não é reinicializada.

Figura 21 – Número de Passos e Valores de Retorno para o Fisherman's Folly Disco Não-Estacionário

(a) Número de Passos para resolver o quebra-cabeça



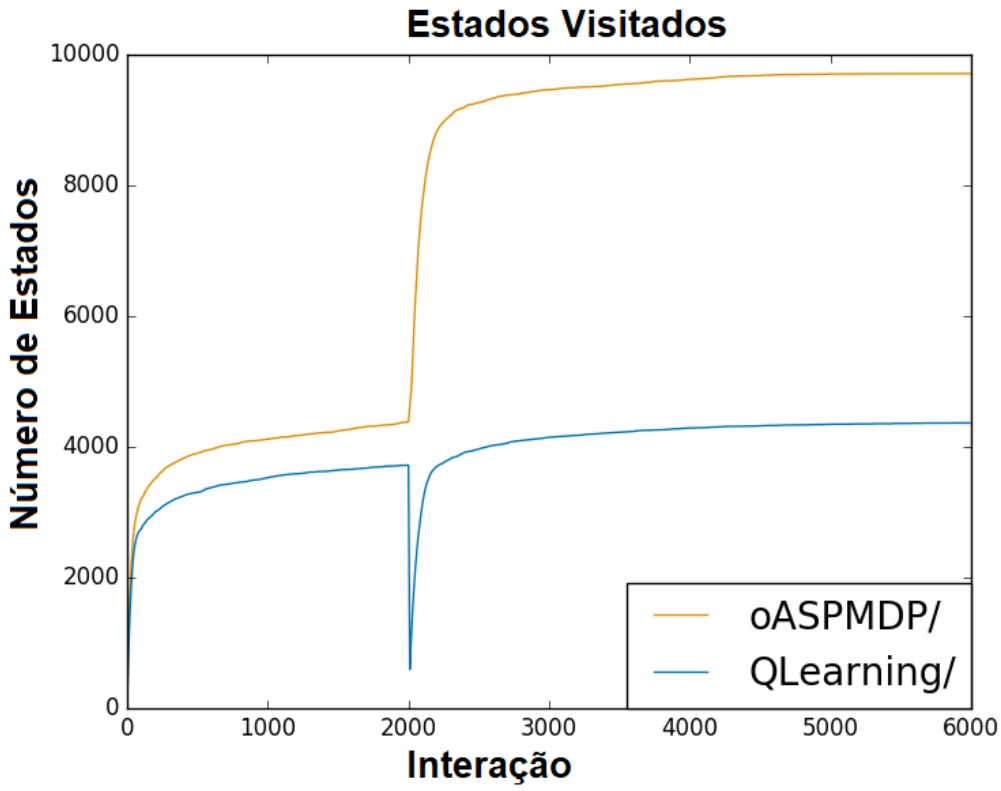
(b) Valor de Retorno recebido por episódio



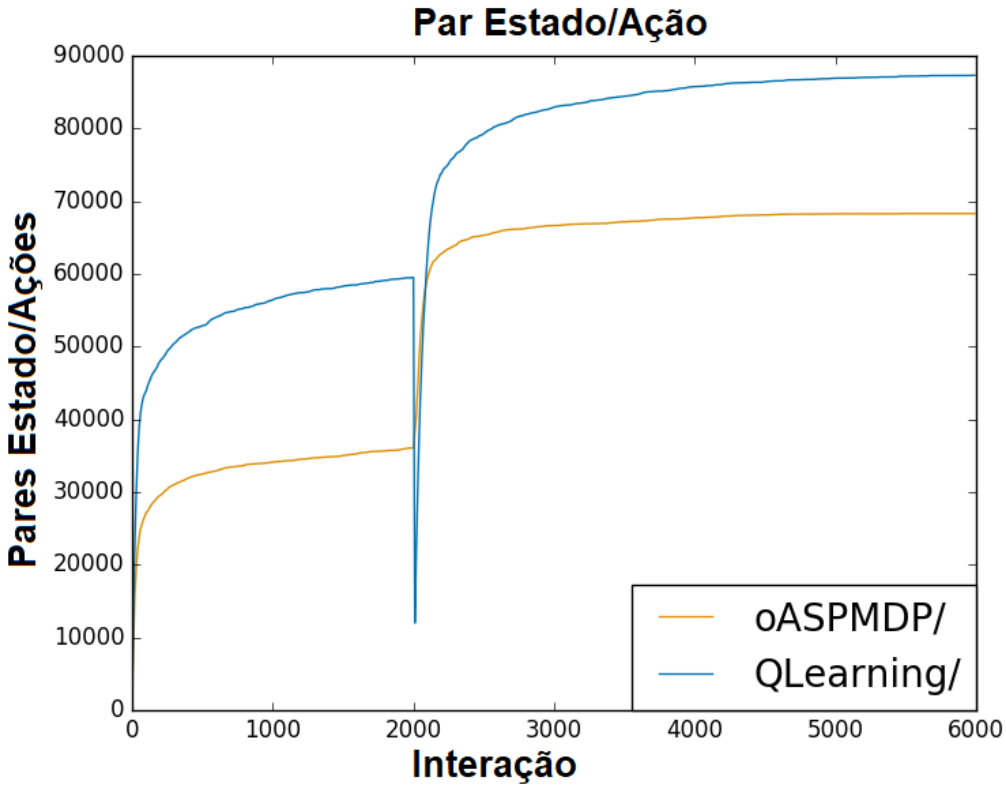
Fonte: Autor

Figura 22 – Resultados de Estados para o Fisherman's Folly Disco Não-Estacionário

(a) Número de Estados Visitados pelo agente



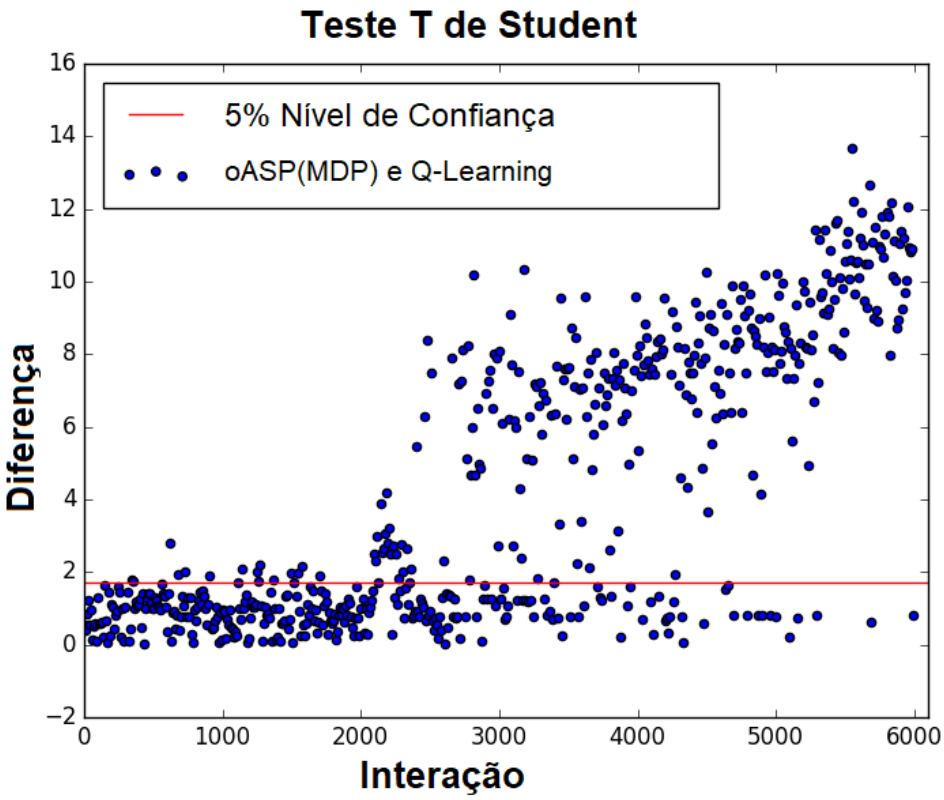
(b) Número de pares Estado/Ações



Fonte: Autor

Figura 23 – Testes Estatísticos para o Fisherman's Folly Disco Não-Estacionário

(a) Teste T de Student comparando o oASP(MDP) com Q-Learning



Fonte: Autor

5.7 ROPE LADDER

Por fim, os resultados para os experimentos do quebra-cabeça mais complexo, Rope Ladder (seção 4.7), são apresentados. Todos os quatro algoritmos estudados (Q-Learning, oASP(MDP), HAQL e HoASP(MDP)) foram executados.

A figura 24a mostra as curvas de aprendizado relacionadas ao Número de Passos para solucionar o quebra-cabeça, é possível ver que os algoritmos acelerados por heurísticas aprendem a política ótima de forma mais rápida quando comparados com os algoritmos não acelerados, a figura 26a apresenta o teste T de Student mostrando como essa diferença é relevante estatisticamente. Além disso, o algoritmo HoASP(MDP) apresenta uma melhor performance, por um pequeno número de interações (por volta de 300 interações), do que o Q-Learning Acelerado por Heurística, o teste T de Student que sustenta essa hipótese pode ser visto na figura 26b.

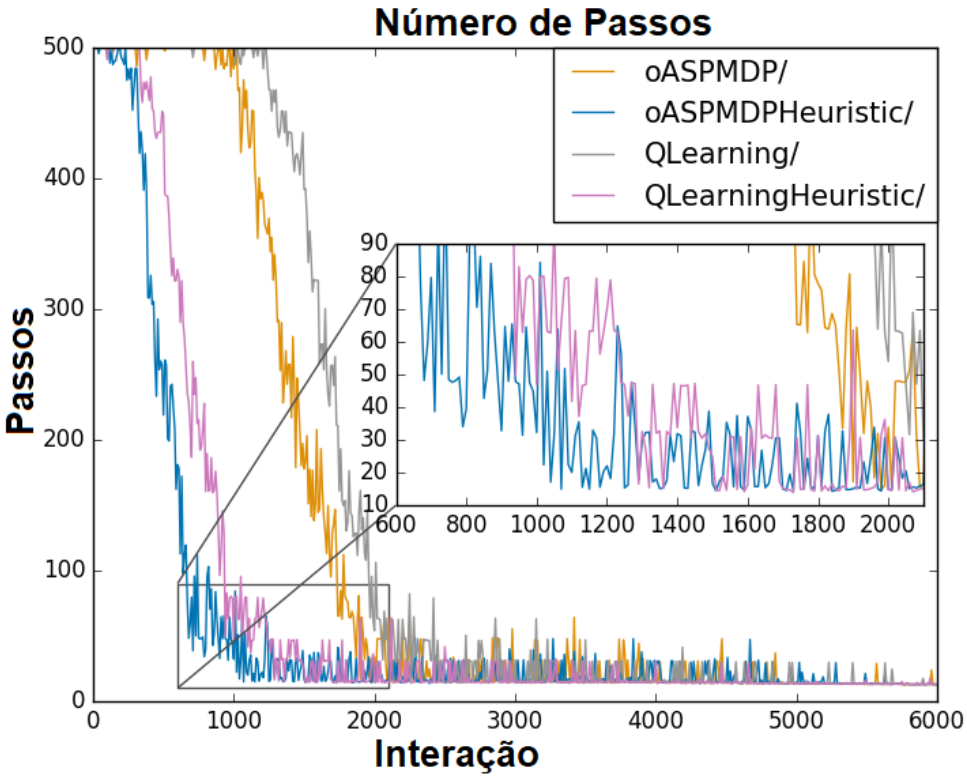
A figura 24b apresenta os Valores de Retorno por algoritmo, é possível observar como esses valores mudam com o tempo, já que no começo da interação, os algoritmos não acelerados por heurísticas apresentam um valor maior de retorno que os algoritmos acelerados por heurísticas, mas essa relação muda conforme o número de interações aumenta, porque os algoritmos acelerados por heurísticas começam a receber valores maiores de retorno com o tempo.

Uma das observações mais importantes que pode ser feita sobre esse domínio é em relação ao gráfico de Valores de Retorno (figura 24b) e o número de Estados Visitados (Figure 25a). É possível observar que, no começo do processo de aprendizado (na parte ampliada do gráfico), os algoritmos que não usam heurísticas têm um maior valor de retorno que os algoritmos que usam heurísticas (já que esses algoritmos exploram mais estados novos, levando os algoritmos acelerados por heurísticas a receber recompensas piores por conta da exploração de estados desconhecidos e da execução de ações desconhecidas nesses estados), isso acontece porque o número de Estados Visitados não cresce na mesma proporção que os Valores de Retorno para os algoritmos sem heurísticas. Em outras palavras, os algoritmos sem heurísticas visitam o mesmo estado com mais frequência, eles demoram mais para explorar o espaço de estados, o que os leva a executarem menos ações impossíveis, já que eles receberam recompensas indicando quais ações são melhores nesses estados conhecidos. Os algoritmos se comportam dessa maneira apenas no começo das interações, conforme o número de interações aumenta, os algoritmos acelerados por heurísticas exploram mais do espaço de estados e acabam recebendo recompensas mais altas (já que acabam chegando mais rápido no estado objetivo). Os Valores de Retorno aumentam para os algoritmos com heurísticas com o tempo, enquanto os algoritmos sem heurísticas precisam de um número maior de interações para chegar ao mesmo resultado.

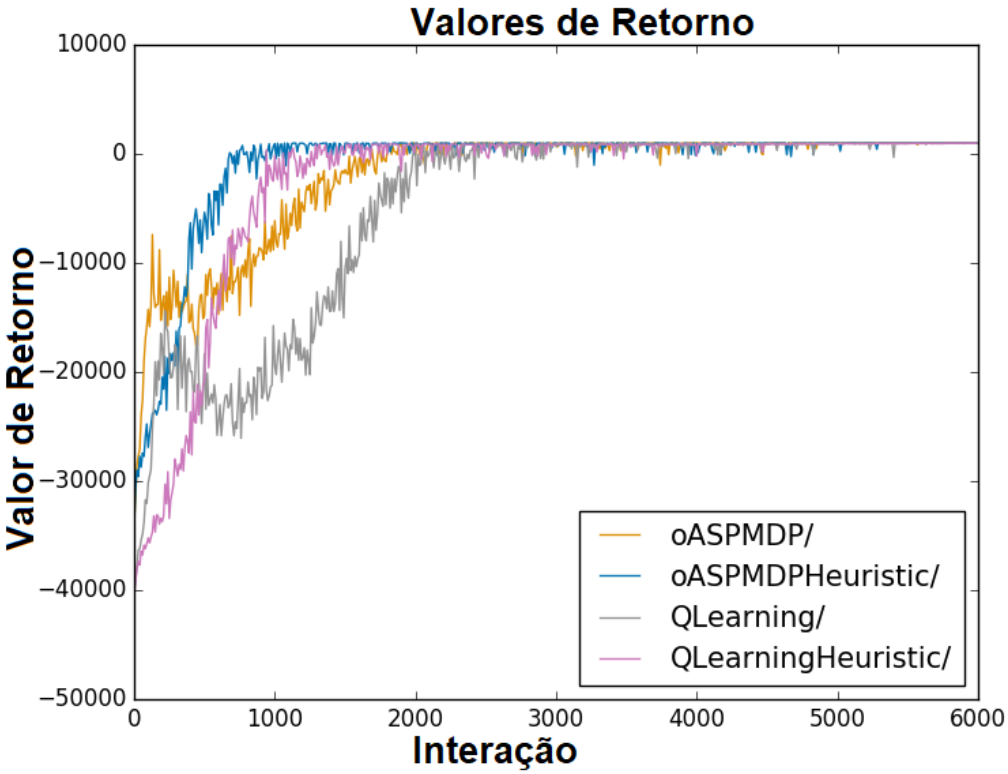
Por último, o gráfico mostrando o número de pares Estado/Ações pode ser visto na figura 25b, apresentando como Q-Learning tem um maior número de pares que os outros algoritmos, pois mantém em sua tabela Q todas as ações para cada estado visitado do domínio.

Figura 24 – Número de Passos e Valores de Retorno para o Rope Ladder

(a) Número de Passos para resolver o quebra-cabeça



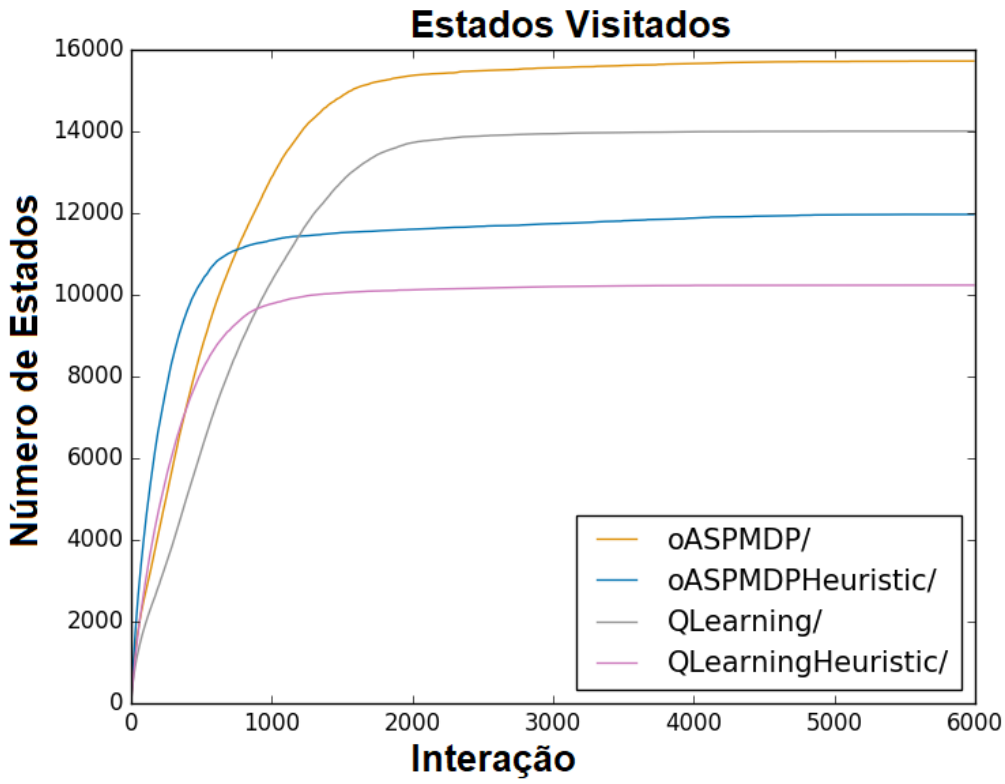
(b) Valor de Retorno recebido por episódio



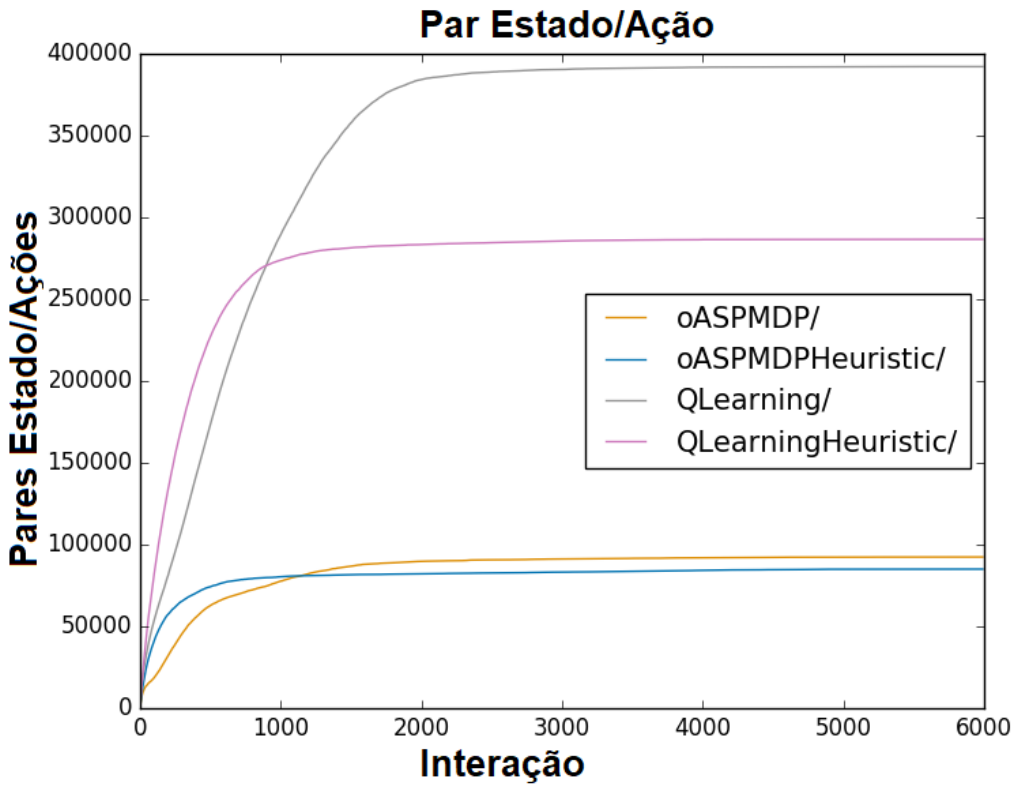
Fonte: Autor

Figura 25 – Resultados de Estados para o Rope Ladder

(a) Número de Estados Visitados pelo agente



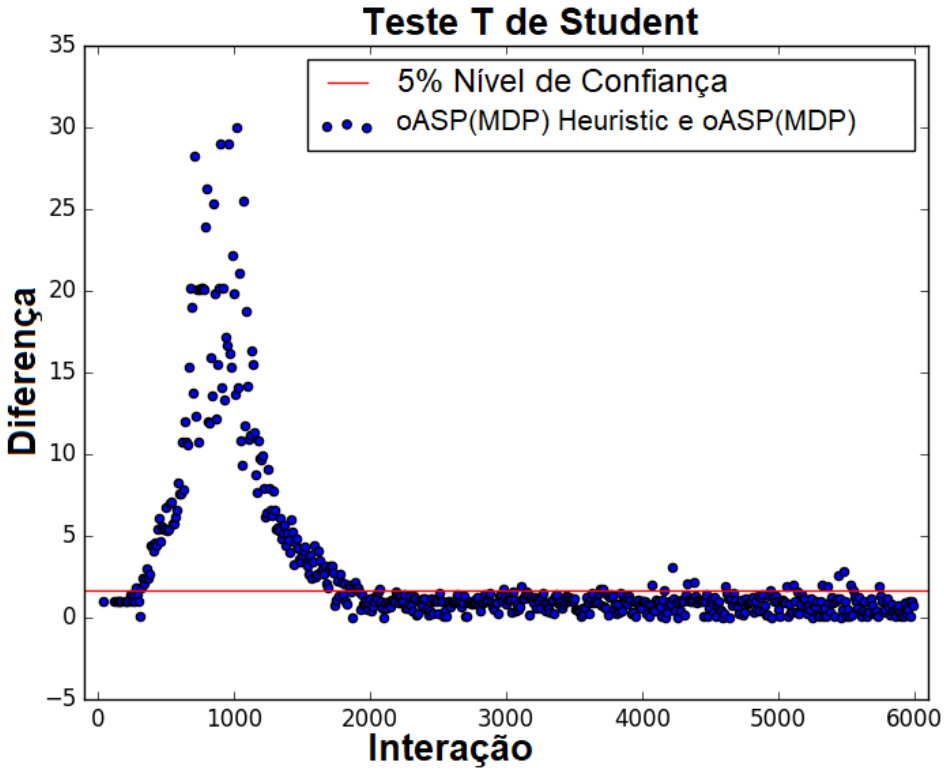
(b) Número de pares Estado/Ações



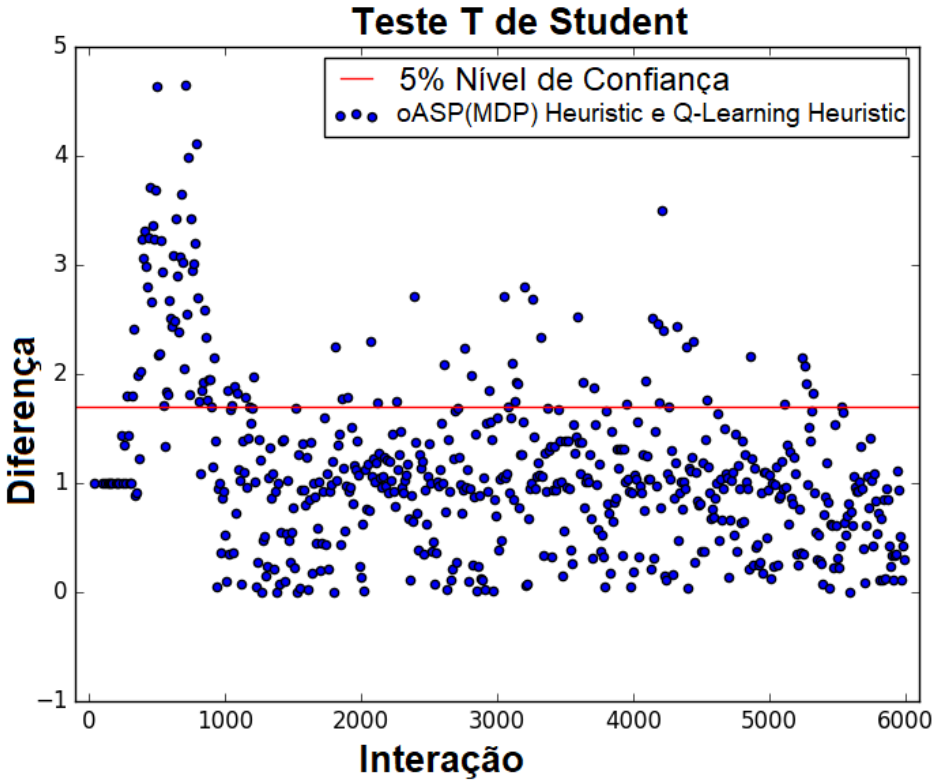
Fonte: Autor

Figura 26 – Testes Estatísticos para o Rope Ladder

(a) Teste T de Student comparando oASP(MDP) original com HoASP(MDP)



(b) Teste T de Student comparando HoASP(MDP) com HAQL



Fonte: Autor

Após a apresentação em gráficos dos resultados, é possível também fazer um resumo e mostrar esses valores de uma forma tabular. Sendo assim, cada tabela mostrada a seguir apresenta uma das métricas utilizada para avaliação dos algoritmos. Primeiro, começando com o "Número de Interação", depois mostrando os valores para o “Número de Estados Visitados” pelo agente, e por fim, a tabela para o "Número de Pares Estado/Ação"que o agente tem no fim do treinamento. A métrica com os “Valores de Retorno” não foi apresentada porque a tabela com valores de “Número de Interação” já resumem as informações que estariam presente na tabela de retorno.

Tabela 1 – Número aproximado de interações necessárias para o agente começar a aprender a política ótima para os quebra-cabeças estacionários

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	HoASP(MDP)	Q-Learning	Q-Learning Heurística
FFO	541	13	476	2
FFS	226	X	185	X
FFND	2197	179	3388	91
FFSND	217	X	252	X
Rope Ladder	1898	1022	2134	1242

Fonte: Autor

Tabela 2 – Número aproximado de interações necessárias para o agente começar a aprender a política ótima para os quebra-cabeças não estacionários

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	oASP(MDP) Pós Mudança	Q-Learning	Q-Learning Pós Mudança
FFENE	62	2210	66	2263
FFDNE	119	2230	142	2213

Fonte: Autor

Tabela 3 – Número de Estados Visitados pelo agente durante a interação com o ambiente para os quebra-cabeças estacionários

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	HoASP(MDP)	Q-Learning	Q-Learning Heurística
FFO	28379	4080	17117	1328
FFS	5895	X	4358	X
FFND	79565	13455	61315	5705
FFSND	5923	X	3585	X
Rope Ladder	15721	11967	14005	10235

Fonte: Autor

Tabela 4 – Número de Estados Visitados pelo agente durante a interação com o ambiente para os quebra-cabeças não estacionários

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	oASP(MDP) Pós Mudança	Q-Learning	Q-Learning Pós Mudança
FFENE	5426	10182	3315	4481
FFDNE	4384	9710	3719	4366

Fonte: Autor

Tabela 5 – Número de pares Estado/Ação presentes na Tabela Q para os quebra-cabeças estacionários

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	HoASP(MDP)	Q-Learning	Q-Learning Heurística
FFO	253855	38570	342358	26573
FFS	39600	X	87157	X
FFND	682128	126103	1226319	114118
FFSND	39186	X	71703	X
Rope Ladder	92426	85005	392163	28660

Fonte: Autor

Tabela 6 – Número de pares Estado/Ação presentes na Tabela Q para os quebra-cabeças não estacionários

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	oASP(MDP) Pós Mudança	Q-Learning	Q-Learning Pós Mudança
FFENE	72029	102111	66313	89639
FFDNE	36101	68303	59509	87322

Fonte: Autor

Para além da discussão de cada uma das configurações dos quebra-cabeças, há algumas observações gerais que podem ser feitas sobre o gráfico com o Número de Passos para todas as configurações (figuras 12a, 9a, 14a, 21a e 24a). Quando o processo de aprendizado está estável (depois do algoritmo ter encontrado a política ótima), o agente ainda tem uma taxa de exploração, isso faz com que as curvas de aprendizado pareçam estar estáveis em um certo número de passos (acima da política ótima). Esse comportamento acontece porque o agente, em um dos 30 experimentos, não encontra a solução ótima (por conta da exploração) e acaba executando todas as 500 ações naquele episódio, não encontrando o estado objetivo. Sendo assim, o valor médio de Número de Passos calculado para aquele episódio em todos os testes acaba ficando acima do valor ótimo.

Outra observação é que as heurísticas não foram utilizadas para os domínios que contam com mudança (não-estacionários), já que os problemas, antes e depois da mudança, eram muito diferentes. O que se procurou com esses experimentos foi avaliar o comportamento do oASP(MDP) e do Q-Learning nesses casos, já que foi mostrado (nos resultados para os quebra-cabeças FFO, FFND e Rope Ladder) como o uso de heurísticas tem um impacto positivo na solução automática desse conjunto de quebra-cabeças.

Após a finalização dos experimentos e apresentação dos resultados e discussão é possível destacar alguns pontos que foram aprendidos no processo de desenvolvimento. As heurísticas mostraram impactar positivamente na aceleração do processo de aprendizado para os quebra-cabeças estudados, evidenciando como as soluções dos quebra-cabeças simplificados podem ser aproveitadas para os quebra-cabeças completos (sendo estes determinísticos ou não). Apesar da vantagem apresentada com uso de heurística, uma desvantagem no seu uso pode ser no processo de desenvolvimento da função que faz o mapeamento entre a tarefa utilizada para obter a heurística e a tarefa que se deseja resolver, pois nos quebra-cabeças estudados essas funções de mapeamento ficaram atreladas aos domínios específicos, não sendo possível abstrair o conhecimento obtido no Fisherman's Folly para o Rope Ladder, por exemplo.

Outro ponto de destaque é como a possibilidade de enrolar a Corda através do Buraco do Poste aumenta o espaço de estado do domínio e a quantidade de interações necessárias para aprender uma política ótima. O aprendizado da política ótima nos quebra-cabeças simplificados (sem possibilidade de enrolar a Corda) é feito de forma mais rápida (necessitando por volta de apenas 200 interações), fazendo com que esses quebra-cabeças sejam relevantes para obtenção de conhecimento prévio que auxilie na solução de problemas mais complexos.

Os experimentos com os quebra-cabeças não-estacionários mostraram que oASP(MDP) não tem uma performance melhor que o Q-Learning em todos os casos, podendo, inclusive, não encontrar a política ótima em alguns domínios (como no Fisherman's Folly Disco Não-Estacionário, seção 4.6). Porém também foi possível observar que, devido à capacidade de não considerar ações impossíveis, o oASP(MDP) pôde obter melhores valores de retorno (como no Rope Ladder, seção 5.7).

Todos os experimentos foram executados em um processador Intel Core I7-8700 3.2GHz com 16GB de RAM, no sistema operacional Ubuntu 16.04 LTS. Os algoritmos foram implementados usando a linguagem de programação Python 3.5, que usa a biblioteca ZeroMQ para providenciar troca de mensagens entre o agente e o ambiente. SWI-PROLOG foi usado para execução do Oráculo e CLINGO foi a engine ASP. Os códigos fontes para os experimentos estão disponíveis em: `https://bitbucket.org/thiagomestrado/journalarticle/src`.

6 REVISÃO BIBLIOGRÁFICA

Neste capítulo estão relacionados outros trabalhos que exploram a combinação de programação em lógica com MDP para resolver tarefas de tomada de decisão sequencial, já que esse é o mesmo tipo de tarefa estudada neste projeto. Além disso, o estudo de Aprendizado por Reforço (AR) com heurística para acelerar o processo de aprendizado também foi explorado. Por fim, são descritas pesquisas que fazem o estudo da aplicação de uma abordagem em lógica formal nos domínios de Quebra-Cabeças Espaciais. Considerando os trabalhos que estão relacionados com o tema deste projeto, pode-se observar duas frentes principais que se destacam, uma é o domínio em que é proposta uma solução (cordas, buracos e objetos flexíveis, representados pelos quebra-cabeças de Fisherman's Folly e Rope Ladder) e a outra é a aplicação do algoritmo HoASP(MDP).

Em seu trabalho, Gelfond e Rushton (2010) apresentam P-Log, uma linguagem declarativa que faz a combinação de Answer Set Programming com probabilidade. Esta linguagem pode ser usada para permitir a representação de conhecimento de bom senso envolvendo lógica e probabilidade, além de fornecer também uma melhora no grau de tolerância à elaboração, devido à generalização de sua base lógica. Apesar desta abordagem ser capaz de calcular probabilidades de transição em ambientes não-determinísticos, ela não pode ser usada para gerar uma política que seja a melhor solução, diferente do algoritmo HoASP(MDP).

Uma outra pesquisa que usa uma abordagem similar à deste trabalho é a realizada por Nickles (2012), que desenvolveu um sistema que faz a integração entre Aprendizado por Reforço e ASP, sendo este último utilizado para representação, processamento e restrição de estados e ações. Esse sistema desenvolvido é o QASP, nomeado assim devido ao uso do método Q-Learning para aprendizado da função valor-ação Q(s, a), apesar do nome QASP remeter ao Q-Learning, a abordagem desenvolvida por Nickles (2012) também pode ser adaptada facilmente para o SARSA, um outro algoritmo capaz de aprender a função valor-ação Q(s, a). O ambiente que este sistema é executado pode ser tanto determinístico quanto não-determinístico, com a limitação de não permitir mudanças na descrição do domínio (não lida com problemas não-estacionários). Essa limitação se dá devido ao modo como QASP funciona, já que este utiliza ASP depois que uma ação é executada e a atualização da função valor-ação Q(s,a) ocorre, diferente do HoASP(MDP) que gera os *answer set* antes da ação ser executada e antes do processo de aproximação da função valor-ação Q(s,a), portanto mudanças no domínio não impedem seu funcionamento.

Diferente do uso com MDPs, mas mostrando a capacidade de resolver tarefas de planejamento com ASP, Erdem et al. (2013) apresentam um trabalho em que ASP é utilizada para encontrar planos ótimos para múltiplos times de robôs que interagem de uma maneira colaborativa. Como um dos passos dessa abordagem é encontrar um plano ótimo global que é formado por planos ótimos de cada time, ASP pôde ser usado para encontrar as respostas para esses

problemas de planejamento. Essa abordagem difere-se do HoASP(MDP) por conta de não lidar com aprendizado, além de não fazer uso de heurísticas para acelerar o processo de AR.

Considerando um domínio de perguntas e respostas, utilizando-se de um conjunto de dados reais, o trabalho de Rettinger, Hagemann e Nickles (2014) desenvolve um sistema híbrido, combinando inferências estatísticas e dedutivas por meio do uso do *framework* de Aprendizado por Reforço Relacional Interativo (IRRL), que faz uso do sistema QASP desenvolvido por Nickles (2012) para a tarefa de aprendizado. Porém, um problema dessa abordagem é a restrição que a disponibilidade (ou a falta) de dados de treinamento necessários para o processo de aprendizado. Essa abordagem conta com as mesmas limitações apresentadas acima sobre QASP.

Outro trabalho que faz uso do ASP para tarefas de planejamento é a pesquisa de Yang et al. (2014), em que o sistema desenvolvido permite que os robôs aprendam custos de ações individuais por meio da interação com o ambiente. ASP é utilizada para representar partes do sistema, como o conhecimento sobre informações que não mudam com o tempo, conhecimento sobre informações que mudam com o tempo independente da ação do robô, conhecimento sobre informações que são diretamente afetadas por ações do robô e por último conhecimento sobre regras que formalizam as ações do robô. O experimento demonstrou como o robô pôde se adaptar mesmo quando há uma mudança no ambiente. Diferente do HoASP(MDP) que usa Aprendizado por Reforço, esse trabalho faz o uso de "Exponentially Weighted Moving Average" para realizar a atualização de estimativas das ações. O trabalho desenvolvido em Khandelwal et al. (2014) é uma extensão de Yang et al. (2014) que faz o uso da linguagem de ação BC (LEE; LIFSCHITZ; YANG, 2013), ao invés de ASP. Essas linguagens são interessantes pelo fato de serem tolerantes à elaboração. A abordagem de Khandelwal et al. (2014) lida com planejamento e aprende os custos das ações, já o HoASP(MDP) não aplica planejamento e ainda assim é capaz de aproximar a função valor-ação Q(s,a) que indica qual é a política ótima a ser seguida.

Já Shiqi Zhang, Sridharan e Wyatt (2015) apresentam uma combinação de Answer Set Prolog com POMDP, tendo como objetivo o uso desse *framework* por um robô que está em um ambiente com observação parcial e executando ações não-determinísticas. O uso do POMDP se dá pelo fato de haver incerteza sobre o processamento dos sinais de entrada dos sensores. Essa abordagem difere-se do HoASP(MDP) usado nessa dissertação por conta da linguagem declarativa que é utilizada para descrever o formalismo (dependendo de uma fase *a priori* de planejamento para prover uma estimativa para o POMDP). Além disso, faz uso do POMDP como o formalismo para descrever o domínio e suas incertezas, ao invés de Answer Set Programming (ASP) e MDP para modelar e representar o problema que pode trabalhar com mudanças no ambiente.

O trabalho de Leonetti, Iocchi e Stone (2016) explora a combinação de ASP e AR, apresentando o framework DARLING, que usa ASP para representar modelos e permitir planejamento e raciocínio, enquanto que AR é usado para tornar o agente adaptativo ao ambiente. DARLING foi testado com um robô de serviço em um escritório, isso permitiu aos autores do

trabalho mostrar que quando DARLING é usado, o robô é capaz de aprender as tarefas de uma maneira mais rápida e melhorar sua performance com o tempo. Um aspecto importante que diferencia o DARLING do HoASP(MDP) é a dependência de uma fase de planejamento no primeiro (que não é necessária no segundo). Trabalho futuro deve explorar essa diferença.

Com o uso de Aprendizado Profundo em muitos trabalhos recentes é de se esperar algo também em combinação com Aprendizado por Reforço, e isso foi feito por Garnelo, Arulkumaran e Shanahan (2016) por meio do desenvolvimento de uma arquitetura que pode ser separada em dois grandes blocos, um que é formado por uma Rede Neural (essa rede tem como saída uma representação simbólica que é passada para o outro bloco da arquitetura) e pelo bloco de Seleção de Ações (que recebe como entrada a saída da Rede Neural). Esta combinação foi adotada a fim de solucionar problemas do Aprendizado Profundo, como a necessidade de uma quantidade extensa de dados para treinamento e lentidão na tarefa de aprendizado. Sendo assim, Aprendizado Profundo é utilizado para encontrar a descrição de um conjunto de estados que são descritos como regras para um programa lógico probabilístico. Com essas informações, o agente equipado com Aprendizado Por Reforço pode interagir com o ambiente e aprender a política ótima. Essa abordagem se diferencia do HoASP(MDP) no modo como gera e representa os estados e ações, usando Aprendizado Profundo para essa tarefa (que envolve mais tempo de processamento e mais poder computacional), ao invés de ASP.

Um outro trabalho que segue o conceito de Aprendizado Profundo (contando com suas limitações e particularidades citadas acima) é o trabalho de Mnih et al. (2015), apresentando o uso de Redes Neurais Profundas de Convolução para aproximar a função valor-ação Q(s,a) a fim de aumentar a capacidade de generalização dos algoritmos de Aprendizado por Reforço. Um ponto de desvantagem sobre essa abordagem a se destacar é que as camadas de convolução não parecem ter grande peso no momento de tomada de decisão do agente, atuando apenas como uma maneira de reduzir a dimensionalidade dos estados.

Zamani et al. (2017) também apresentam uma abordagem relacionada com o uso de Aprendizado Profundo, onde é desenvolvida uma combinação de uma Rede-Q Profunda (Deep-Q Network) com Representação Simbólica. Esta representação simbólica serve tanto para entrada como para saída do algoritmo profundo, fazendo com que este seja na prática um sistema de abordagem híbrida. O domínio utilizado para experimentos dessa abordagem foi o de dar instrução para uma máquina em linguagem natural sobre um ambiente que conta com relação espacial entre objetos. Para isso, há a passagem da Representação Simbólica do ambiente para o agente do Aprendizado por Reforço, que é formada por predicados que tem como objetivo descrever ou uma relação espacial entre dois objetos ou o estado de um determinado objeto, sendo que para se formar um estado do ambiente é necessário que ocorra uma combinação desses predicados. A diferença em relação ao HoASP(MDP) está na fase de geração de estados, já que no algoritmo HoASP(MDP), essa fase é de responsabilidade do ambiente, não sendo necessário o uso de um sistema intermediário que seja capaz de mapear um conjunto de predicados para um estado.

Continuando com os estudos relacionados com Aprendizado Profundo, o trabalho de Garcez, Dutra e Alonso (2018) apresenta uma extensão do que foi realizado por Garnelo, Arulkumaran e Shanahan (2016), o algoritmo Aprendizado por Reforço Simbólico com Senso Comum (SRL+CS), onde há a inclusão de raciocínio de senso comum, oferecendo um melhor balanço entre generalização e especialização quando comparado com Garnelo, Arulkumaran e Shanahan (2016). A representação dos estados é feita através de símbolos, por exemplo os objetos de uma imagem são representados com um símbolo e sua localização na imagem. Com isso é possível realizar uma representação mais abstrata do domínio, incluindo a capacidade de criar estados totalmente novos a partir de uma determinada entrada. Apesar da aplicação em um domínio muito simples, esse trabalho apresentou um resultado melhor do que implementações tradicionais de Q-Learning e da implementação de Garnelo, Arulkumaran e Shanahan (2016).

Já o trabalho de Yang et al. (2018) apresenta um *framework* que integra planejamento simbólico com Aprendizado por Reforço Hierárquico (HRL) (diferente do HoASP(MDP), que utiliza Q-Learning), lidando com tomada de decisão em um ambiente não-determinístico. Os planos simbólicos são utilizados para guiar o aprendizado do agente, após isso, a experiência que foi aprendida é alimentada de volta para o planejador simbólico com o intuito de melhorar o plano fornecido para o algoritmo de HRL. Essa abordagem é definida como Planejamento, Execução, Observação e Aprendizado por Reforço (PEORL). A linguagem BC é utilizada para representar conhecimento de senso comum. Resultados mostraram que há um retorno maior dos valores acumulados de recompensa, quando comparada com versões tradicionais de AR.

Outro trabalho que também estudou a solução do quebra-cabeça de Fisherman's Folly, formalizando-o como um Processo de Decisão de Markov (MDP), é o apresentado por Freitas dos Santos et al. (2018), sendo que essa dissertação estende este trabalho considerando novos quebra-cabeças, além de explorar novos conjuntos de parâmetros para os algoritmos de AR.

Considerando que esse trabalho adaptou o algoritmo oASP(MDP) para aplicar heurísticas (HoASP(MDP)) com o intuito de acelerar o processo de aprendizado, vale destacar alguns trabalhos que utilizaram dessa abordagem. Primeiramente apresentado por (BIANCHI; RIBEIRO; COSTA, 2008), o Aprendizado por Reforço Acelerado por Heurística (HARL) faz uso de heurísticas para acelerar e guiar o processo de aprendizado. A função heurística é diretamente ligada com a política, visto que a primeira serve para indicar qual ação deve ser tomada em um determinado estado, desse modo, após inúmeras interações, o algoritmo de AR é capaz de aprender a política ótima que soluciona um problema. Essa implementação mostrou melhores resultados do que a aplicação tradicional de Q-Learning, podendo ser aplicada também por outros algoritmos de AR, como SARSA.

O trabalho de Celiberto Jr. et al. (2011) apresenta a combinação de três diferentes técnicas para acelerar o processo de aprendizado de um algoritmo de AR, o Raciocínio Baseado em Casos, Heurística e Redes Neurais Artificiais, criando o algoritmo L3, que funciona da seguinte maneira: primeiro, o algoritmo de AR aprende uma política ótima para determinada tarefa, gravando a política ótima aprendida como um caso (para ser utilizado posteriormente);

segundo, uma rede neural é aplicada para realizar um mapeamento das ações de um domínio para outro; finalmente, os casos obtidos na primeira etapa são utilizados como heurísticas para acelerar a o processo de aprendizado. É importante ressaltar que o processo de aprendizado é feito em uma tarefa relacionada, porém diferente daquela utilizada para obtenção dos casos. A abordagem de Celiberto Jr. et al. (2011) foi aplicada em dois domínios diferentes, onde a política do primeiro domínio é utilizada como heurística pelo segundo domínio. O maior desafio dessa abordagem é determinar a equivalência entre ações de um domínio para outro (criação da função de mapeamento).

Martins e Bianchi (2014) apresentam um estudo comparativo de três diferentes algoritmos de AR com suas versões aceleradas por heurística, mostrando como a aceleração por heurística é benéfica no contexto de AR. Outra conclusão apresentada no trabalho de Martins e Bianchi (2014) é que o uso de uma boa função heurística é útil como um guia para fase de seleção das ações, principalmente quando o agente tem pouco ou nenhum conhecimento (se destacando especialmente no começo do processo de aprendizado), importante observar que conforme o número de interações aumenta, a importância e influência da heurística diminuem, fazendo com que no fim os algoritmos acelerados por heurística e suas contrapartidas (sem heurística) tenham o mesmo comportamento.

Também explorando a combinação entre heurística e AR, o trabalho de Morozs, Clarke e Grace (2015) aplicou o algoritmo HAQL no domínio de Compartilhamento Dinâmico de Espectro Secundário em sistemas de celular[9]. Os experimentos realizados mostram como essa abordagem alcança um controle alto dos padrões de compartilhamento de uma maneira totalmente autônoma. Como os outros trabalhos apresentados anteriormente, os resultados também mostram a superioridade dessa abordagem quando comparado com AR tradicional.

Bianchi et al. (2018) apresentam uma classe de algoritmos que utiliza Raciocínio Baseado em Casos (CBR) como heurística em uma configuração que há aprendizado por transferência para acelerar o processo de aprendizado. A aplicação dessa abordagem foi realizada em dois domínios diferentes de robôs. Primeiro, o algoritmo de AR é aplicado em uma tarefa que é a fonte da heurística. Após a estabilização do aprendizado, uma base de casos é construída. Segundo, esses casos aprendidos são transferidos e utilizados na tarefa desejada, nessa fase, essa adaptação dos casos é feita seguindo os conceitos de CBR.

Fengyun Zhang, Duan e Wang (2017) apresentam um método melhorado de AR, utilizando Redes Neurais com Funções de Ativação de Base Radial (Rede RBF) e Q-Learning acelerado com heurística. Essa aplicação foi realizada no domínio de busca de rotas contando com um grande espaço de estados. Os experimentos e resultados também mostraram como o uso de heurística acelera o processo de aprendizado.

Apesar do HoASP(MDP) utilizar a abordagem de heurística para acelerar o processo de aprendizado e fornecer conhecimento obtido previamente para um algoritmo de AR, há outras

[9]Domínio que lida com o compartilhamento de espectros disponíveis no contexto de redes de rádio. Um melhor compartilhamento desse espectro resulta em uma melhora em chamadas de voz e transmissão de dados, além de prover uma boa qualidade de serviço para os usuários da rede.

abordagens que procuram esses mesmos efeitos (KOGA; FREIRE, V.; COSTA, 2015; TAYLOR, A. et al., 2014). O trabalho de Topin et al. (2015) introduz um *framework* para descoberta de opção e transferência de aprendizado em domínios complexos que são representados como MDP orientado a objeto (OO-MDPs). Um ponto de destaque dessa abordagem é a utilização de um método capaz de realizar o mapeamento entre domínios orientado a objeto que contam com um espaço de estados diferente. O uso de heurística pode ser tão geral que é também utilizado nesse *framework* como um modo de aumentar a eficiência do processo de mapeamento. Utilizando essa abordagem, o trabalho de Bonini et al. (2018) aplica uma maneira probabilística para obtenção do aprendizado de opções em um contexto de domínio orientado a objeto por meio do encapsulamento de um comportamento aprendido para uma forma mais genérica, providenciando assim uma melhor escolha de tarefas para os agentes e acelerando o processo de aprendizado (com experimentos executados no domínio de Mundo de Grades e no Mundo de Táxi).

Para além dos trabalhos que foram apresentados aqui, Leno Da Silva, Matthew E. Taylor e Costa (2018) apresentam uma gama de pesquisas que lidam com diferentes técnicas que são capazes de reutilizar conhecimento e prover aceleração do processo de aprendizado no contexto de AR, incluindo domínios com Multi Agentes. A escolha pelo uso da abordagem HARL nessa dissertação é porque um dos objetivos do trabalho é explorar como as heurísticas podem guiar o processo de aprendizado neste tipo de domínio (quebra-cabeças espaciais), que mantém as propriedades dos elementos de diferentes configurações do quebra-cabeças da mesma forma (com adição ou remoção de elementos) e com o mesmo comportamento.

Por fim, segue uma descrição de alguns trabalhos que abordam o domínio em que o algoritmo HoASP(MDP) foi aplicado, contando com uma abordagem mais formal sobre representação do conhecimento.

Cabalar e Santos (2011) resolvem os problemas relacionados ao quebra-cabeça de Fisherman's Folly que não foram considerados pelo trabalho anterior (CABALAR; SANTOS, 2006). Esses problemas estão relacionados com o tratamento de fluentes (propriedades do domínio que podem ter os seus valores alterados por meio da aplicação de uma ação) e ações. Para resolver essas limitações, Cabalar e Santos (2011) apresentam uma formalização dos quebra-cabeças espaciais explorados nessa dissertação (Fisherman's Folly e Rope Ladder), aplicando uma combinação de Cálculo de Situações (uma ferramenta de sintaxe para formalizar o domínio) e Lógica de Equilíbrio (usada para representar a semântica). Além disso Cabalar e Santos (2011) também desenvolveram um planejador PROLOG, utilizado como um Oráculo no *framework* de AR. Esse planejador PROLOG também pode ser usado para solucionar o quebra-cabeça de Fisherman's Folly, porém há um ponto negativo nesse planejador, já que ele não pode ser usado para resolver quebra-cabeças mais complexos (como Rope Ladder), pois o espaço de estados é muito grande nesses casos, o que faz com que leve muito tempo para encontrar a solução ou não encontra a solução. Diferente do que é abordado por esse trabalho (HoASP(MDP)), em que uma combinação de ASP e AR é utilizada para realizar a busca nesse

espaço de estados, aprendendo as ações ótimas conforme as interações acontecem. Outras vantagens do HoASP(MDP) incluem a possibilidade de aprendizado de uma solução e a aplicação em quebra-cabeças Não-Determinísticos e Não-Estacionários.

Com o intuito de solucionar alguns problemas de representação deixados em aberto por Cabalar e Santos (2011), o trabalho desenvolvido por Santos e Cabalar (2013) apresenta a formalização que leva em consideração a possibilidade de formação de laços com cordas e de um objeto com buraco atravessar outro objeto com buraco. Já o trabalho de Santos e Cabalar (2016) apresenta uma descrição e formalização de buracos como cordas e de buracos passando através de outros buracos, podendo esses buracos ser criados conforme o agente interage com o ambiente (no caso com o quebra-cabeça). Além disso, Santos e Cabalar (2016) formalizam duas outras ações que são aplicadas sobre cordas que podem criar ou destruir os buracos: passar uma parte da corda por um buraco e puxar de uma corda para desfazer um laço. Apesar desse formalismo funcionar, essa dissertação faz uso de ASP porque com essa linguagem é possível descrever cada estado com um programa ASP que descreve as consequências de executar cada ação permitida em um estado. Diferente do trabalho de Santos e Cabalar (2016), que foca em uma formalização mais lógica desses quebra-cabeças (com uma implementação complexa), essa dissertação foca em uma abordagem em processos de decisão de Markov para o problema, procurando entender como AR pode trabalhar combinado com lógica (especificamente ASP) e sendo acelerado com o uso de heurísticas.

7 CONCLUSÃO

Este trabalho apresentou a adaptação do algoritmo oASP(MDP), que combina Answer Set Programming (ASP) com MDP, para utilizar heurísticas com o intuito de acelerar o processo de aprendizado. Essa adaptação deu origem ao algoritmo Heurística oASP(MDP) (HoASP(MDP)) que foi aplicado a um conjunto de quebra-cabeças espaciais com diferentes configurações.

Dois quebra-cabeças espaciais (Fisherman's Folly e Rope Ladder) foram utilizados como base para os diferentes experimentos realizados. Esses quebra-cabeças são formados por cordas flexíveis, objetos rígidos e buracos, em que esses objetos são entrelaçados entre si formando um estado do quebra-cabeça. Quando comparado com o Fisherman's Folly, o Rope Ladder tem uma relação mais desafiadora entre os objetos que o formam.

Sete experimentos foram realizados em diferentes configurações determinísticas, não-determinísticas e não-estacionárias dos quebra-cabeças espaciais. Sendo assim, foi possível avaliar como quatro algoritmos, que utilizam do framework de Aprendizado por Reforço (AR), lidam com as características dos domínios explorados. Os quatro algoritmos considerados são: Q-Learning, Q-Learning Acelerado por Heurística, oASP(MDP) e Heurística oASP(MDP), proposto nessa dissertação. Todos esses algoritmos foram capazes de solucionar um conjunto de quebra-cabeças espaciais formados por cordas flexíveis, objetos rígidos e buracos.

As heurísticas foram obtidas por meio da solução de configurações relaxadas dos domínios, aplicando alguma restrição com o intuito de facilitar o processo de aprendizado e tornar o quebra-cabeça viável para obtenção das heurísticas.

Resultados mostraram que as heurísticas fornecem informações que possibilitaram guiar e acelerar o processo de aprendizado, sendo que os algoritmos acelerados por heurísticas apresentam uma melhor performance do que os algoritmos que não usam heurísticas. Além disso, os algoritmos que utilizam oASP(MDP) demostraram uma capacidade de maior exploração do espaço de estados, já que executam menos ações que são impossíveis fisicamente.

Portanto, é possível concluir que a principal contribuição dessa dissertação é o desenvolvimento do algoritmo HoASP(MDP). Sendo que os resultados mostraram que entre os quatro algoritmos avaliados, o HoASP(MDP) conta com a melhor performance em termos de interações necessárias para se encontrar a política ótima. Como o HoASP(MDP) apenas necessita da tabela Q da solução de um problema como heurística, esse algoritmo é geral o suficiente para também ser aplicado em outros domínios além dos quebra-cabeças espaciais.

7.1 TRABALHOS FUTUROS

Para explorar mais o uso do ASP no contexto de AR, um dos temas para trabalho futuro é a aplicação dessa linguagem para extrair regras mais gerais de domínios (restrições, por exemplo) e então aplicar essas regras para permitir raciocínio sobre o domínio, possivelmente

guiando o processo de aprendizado. Com essa abordagem, seria possível construir uma arquitetura mais genérica para reutilização de conhecimento previamente obtido.

Uma outra possibilidade de trabalho futuro é incluir uma etapa de planejamento, em que as consequências das ações possam ser inferidas e então usá-la no processo de escolha da ação.

Por fim, considerar outros domínios, que lidam com os elementos estudados nos quebra-cabeças espaciais, a fim de verificar como a abordagem proposta neste trabalho (HoASP(MDP)) lida com outras aplicações reais, de interesses da indústria, tais como cirurgia robótica e manutenção de maquinário.

REFERÊNCIAS

BARLOW, Horace B. Unsupervised learning. **Neural computation**, MIT Press, v. 1, n. 3, p. 295–311, 1989.

BIANCHI, Reinaldo AC; RIBEIRO, Carlos HC; COSTA, Anna HR. Accelerating autonomous learning by using heuristic selection of actions. **Journal of Heuristics**, Springer, v. 14, n. 2, p. 135–168, 2008.

BIANCHI, Reinaldo AC et al. Heuristically Accelerated Reinforcement Learning by Means of Case-Based Reasoning and Transfer Learning. **Journal of Intelligent & Robotic Systems**, Springer, v. 91, p. 301–312, 2 2018.

BONINI, R et al. A Framework to Discover and Reuse Object-Oriented Options in Reinforcement Learning. In: BRAZILIAN CONFERENCE ON INTELLIGENT SYSTEMS (BRACIS), 7., 2018, São Paulo. **Proceedings**... [S.l.]: IEEE, 2018. p. 109–114.

CABALAR, Pedro. **Relational Representation and Reasoning**. Corunna: Corunna University, 2017. (Notas de aula, Chapter 3). Acesso em: 12 Jan 2019. Disponível em: <http://www.dc.fi.udc.es/~cabalar/kr/index.html>.

CABALAR, Pedro; SANTOS, Paulo E. A qualitative spatial representation of string loops as holes. **Artificial Intelligence**, Elsevier Scienc, v. 238, n. 1, p. 1–10, 2016.

_____. Formalising the Fisherman's Folly puzzle. **Artificial Intelligence**, Elsevier Science, v. 175, n. 1, p. 346–377, 2011.

_____. Strings and Holes: An Exercise on Spatial Reasoning. In: ADVANCES in Artificial Intelligence - IBERAMIA-SBIA, 2006, Ribeirão Preto. **Proceedings**... Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 419–429.

CELIBERTO JR., Luiz A. et al. Using cases as heuristics in reinforcement learning: a transfer learning application. In: 1. INTERNATIONAL Joint Conference on Artificial Intelligencee - IJCAI, 2011, Barcelona. **Proceedings**... [S.l.]: AAAI Press, 2011. v. 22. (IJCAI'11, 1), p. 1211–1217.

EITER, Thomas; IANNI, Giovambattista; KRENNWALLNER, Thomas. Answer set programming: A primer. In: REASONING Web. Semantic Technologies for Information Systems, 2009, Brixen-Bressanone. **Proceedings**... Berlin, Heidelberg": Springer Berlin Heidelberg, 2009. p. 40–110.

ERDEM, Esra et al. Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. **Theory and Practice of Logic Programming**, Cambridge University Press, v. 13, n. 4-5, p. 831–846, 2013.

FERREIRA, Leonardo A et al. A Method for the Online Construction of the Set of States of a Markov Decision Process Using Answer Set Programming. In: INTERNATIONAL CONFERENCE ON INDUSTRIAL ENGINEERING AND OTHER APPLICATIONS OF APPLIED INTELLIGENT SYSTEMS, 2018, Montreal. **Proceedings**... Cham: Springer International Publishing, 2018. p. 3–15.

FERREIRA, Leonardo A et al. Answer set programming for non-stationary Markov decision processes. **Applied Intelligence**, Springer, v. 47, n. 4, p. 993–1007, 2017.

FREIRE, Paulo. **Pedagogia da autonomia:** saberes necessários à prática docente. São Paulo: Paz e Terra, 1996. p. 25.

FREITAS DOS SANTOS, Thiago et al. Solving a Spatial Puzzle Using Answer Set Programming Integrated with Markov Decision Process. In: BRAZILIAN CONFERENCE ON INTELLIGENT SYSTEMS (BRACIS), 7., 2018, São Paulo. **Proceedings**... [S.l.]: IEEE, 2018. p. 528–533.

GARCEZ, Artur S. d'Avila; DUTRA, Aimore Resende Riquetti; ALONSO, Eduardo. Towards Symbolic Reinforcement Learning with Common Sense. **CoRR**, abs/1804.08597, 2018. Acesso em: 20 Dez 2018. Disponível em: <http://arxiv.org/abs/1804.08597>.

GARNELO, Marta; ARULKUMARAN, Kai; SHANAHAN, Murray. Towards deep symbolic reinforcement learning. In: DEEP REINFORCEMENT LEARNING WORKSHOP ON NEURAL INFORMATION PROCESSING SYSTEMS, 2016, Barcelona. **Proceedings**... [S.l.: s.n.], 2016.

GELFOND, Michael. Answer sets. **Foundations of Artificial Intelligence**, Elsevier, v. 3, p. 285–316, 2008.

GELFOND, Michael; LIFSCHITZ, Vladimir. The stable model semantics for logic programming. In: LOGIC PROGRAMMING, INTERNATIONAL CONFERENCE AND SYMPOSIUM, 1988, Londres. . **Proceedings**... Cambridge: MIT Press, 1988. v. 2, p. 1070–1080.

GELFOND, Michael; RUSHTON, Nelson. Causal and probabilistic reasoning in p-log. In: HEURISTICS, PROBABILITIES AND CAUSALITY. A TRIBUTE TO JUDEA PEARL, 2010. **Proceedings**... [S.l.]: College Publications, 2010. p. 337–359.

HASTIE, Trevor; TIBSHIRANI, Robert; FRIEDMAN, Jerome. **The Elements of Statistical Learning:** Data Mining, Inference, and Prediction. New York: Springer, 2009. p. 9–41.

HOFFMAN, Karla L.; PADBERG, Manfred; RINALDI, Giovanni. Traveling Salesman Problem. In: GASS, Saul I.; FU, Michael C. **Encyclopedia of Operations Research and Management Science**. 2ed. Boston: Springer US, 2013. [S.l.: s.n.]. p. 1573–1578.

JAIN, Anil K. Data Clustering: 50 Years Beyond K-means. **Pattern Recognition Letters**, Elsevier Science, v. 31, n. 8, p. 651–666, 2010.

KHANDELWAL, Piyush et al. Planning in Action Language BC while Learning Action Costs for Mobile Robots. In: INTERNATIONAL CONFERENCE ON AUTOMATED PLANNING AND SCHEDULING, 2014, Portsmouth. **Proceedings**... [S.l.]: AAAI Press, 2014.

KOGA, Marcelo L; FREIRE, Valdinei; COSTA, Anna HR. Stochastic abstract policies: Generalizing knowledge to improve reinforcement learning. **IEEE Transactions on Cybernetics**, IEEE, v. 45, n. 1, p. 77–88, 2015.

LEE, Joohyung; LIFSCHITZ, Vladimir; YANG, Fangkai. Action Language BC: Preliminary Report. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2013, Beijing. **Proceedings**... [S.l.]: AAAI Press, 2013. p. 983–989.

LENO DA SILVA, Felipe; TAYLOR, Matthew E.; COSTA, Anna HR. Autonomously Reusing Knowledge in Multiagent Reinforcement Learning. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, IJCAI-18, 2018, Estocolmo. **Proceedings**... [S.l.]: International Joint Conferences on Artificial Intelligence Organization, jul. 2018. p. 5487–5493.

LEONETTI, Matteo; IOCCHI, Luca; STONE, Peter. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. **Artificial Intelligence**, v. 241, p. 103–130, 2016.

LIFSCHITZ, Vladimir. What Is Answer Set Programming?. In: 2008. NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2008, Chicago. **Proceedings**... [S.l.]: AAAI Press, 2008. v. 8. (AAAI'08, 2008), p. 1594–1597.

MARTINS, Murilo Fernandes; BIANCHI, Reinaldo AC. Heuristically-accelerated reinforcement learning: A comparative analysis of performance. In: TOWARDS AUTONOMOUS ROBOTIC SYSTEMS, 2014, Oxford. **Proceedings**... Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 15–27.

MNIH, Volodymyr et al. Human-level control through deep reinforcement learning. **Nature**, Nature Research, v. 518, n. 7540, p. 529–533, 2015.

MOROZS, N.; CLARKE, T.; GRACE, D. Heuristically Accelerated Reinforcement Learning for Dynamic Secondary Spectrum Sharing. **IEEE Access**, v. 3, p. 2771–2783, 2015.

NICKLES, Matthias. A system for the use of answer set programming in reinforcement learning. In: LOGICS IN ARTIFICIAL INTELLIGENCE, 2012, Toulouse. **Proceedings**... Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 488–491.

RETTINGER, Achim; HAGEMANN, Alexander; NICKLES, Matthias. Learning an Optimal Sequence of Questions for the Disambiguation of Queries over Structured Data. In: WORKSHOPS AT THE TWENTY-EIGHTH AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2014, Quebec. **Proceedings**... [S.l.]: AAAI Press, 2014.

SANTOS, Paulo E; CABALAR, Pedro. An investigation of actions, change, space within a hole-loop dichotomy. In: SYMP. ON LOGICAL FORMALIZATIONS OF COMMONSENSE REASONING, 2013, Ayia Napa. **Proceedings**... [S.l.]: Elsevier Science, 2013.

_____. Framing holes within a loop hierarchy. **Spatial Cognition & Computation**, Taylor & Francis, v. 16, n. 1, p. 54–95, 2016.

SUTTON, Richard S; BARTO, Andrew G. **Reinforcement learning an introduction**. 2. ed. Cambridge, Mass.: MIT, 2018. Disponível em: <http://incompleteideas.net/book/bookdraft2018jan1.pdf>. Acesso em: 16 fev. 2018.

TAYLOR, Adam et al. Accelerating learning in multi-objective systems through transfer learning. In: CITESEER. IJCNN. [S.l.: s.n.], 2014. p. 2298–2305.

TOPIN, Nicholay et al. Portable Option Discovery for Automated Learning Transfer in Object-Oriented Markov Decision Processes. In: IJCAI. [S.l.: s.n.], 2015. p. 3856–3864.

WAŁĘGA, Przemysław Andrzej; SCHULTZ, Carl; BHATT, Mehul. Non-monotonic spatial reasoning with answer set programming modulo theories. **Theory and Practice of Logic Programming**, Cambridge University, v. 17, n. 2, p. 205–225, 2017.

YANG, Fangkai et al. PEORL: Integrating Symbolic Planning and Hierarchical Reinforcement Learning for Robust Decision-Making. **CoRR**, abs/1804.07779, 2018. Acesso em: 20 Dez 2018. Disponível em: <http://arxiv.org/abs/1804.07779>.

YANG, Fangkai et al. Planning in Answer Set Programming while Learning Action Costs for Mobile Robots. In: SYMPOSIUM ON KNOWLEDGE REPRESENTATION AND REASONING IN ROBOTICS (AAAI-SSS), 2014. **Proceedings**... Palo Alto: AAAI Press, 2014.

ZAMANI, Mohammad Ali et al. Deep Reinforcement Learning using Symbolic Representation for Performing Spoken Language Instructions. In: BEHAVIOR ADAPTATION, INTERACTION AND LEARNING FOR ASSISTIVE ROBOTICS (BAILAR) ON ROBOT AND HUMAN INTERACTIVE COMMUNICATION (RO-MAN) WORKSHOP, 2017, Lisboa.**Proceedings**... [S.l.]: IEEE, 2017.

ZHANG, Fengyun; DUAN, Shukai; WANG, Lidan. Route searching based on neural networks and heuristic reinforcement learning. **Cognitive neurodynamics**, Springer, v. 11, n. 3, p. 245–258, 2017.

ZHANG, Shiqi; SRIDHARAN, Mohan; WYATT, Jeremy L. Mixed logical inference and probabilistic planning for robots in unreliable worlds. **IEEE Transactions on Robotics**, IEEE, v. 31, n. 3, p. 699–713, 2015.

ÍNDICE

ASP, 16, 19
asp, 17

HARL, 17

MDP, 15, 19, 82
ML, 17

RL, 16, 17, 19
RNA, 28

APÊNDICE A – CÓDIGOS UTILIZADOS PARA OS EXPERIMENTOS

Figura 27 – Código utilizado para execução de um experimento. Os resultados apresentados nessa dissertação realizam 30 experimentos

```
def experiment():
    for trial in range(trials):
        print("Trial: %s" % int(trial+1))
        pTable = dict()
        qTable = dict()
        trialStartTime = time.time()

        # variables to be saved
        qTablesList = list()
        steps = list()
        returns = list()
        diffQ = list()
        oldQTable = qTable.copy()
        numberOfStateActionPairs = list()

        #Variables that needs to be reinitialized
        updatedEpsilon = exp.initialEpsilon
        forbiddenStates = []
        forbiddenActions = []
        forbiddenActionState = []
        goalState = []
        actionsByState = dict()

        currentEpisode = 0
        for currentEpisode in range(episodes):
            print("Episode: %s" % int(currentEpisode))
            qTable, pTable, step, reward = episode(qTable, pTable, updatedEpsilon, forbiddenStates, forbiddenActions,
                                                   forbiddenActionState, goalState, actionsByState)
            if((currentEpisode+1) % qDifferenceWindow == 0):
                differenceBetweenQs = mdp.rmsd(oldQTable, qTable)
                oldQTable = qTable.copy()
                diffQ.append(differenceBetweenQs)
            steps.append(step)
            returns.append(reward)
            qTablesList.append(len(qTable)) #To get the number of states in the Q-Table in the current time step.
            numberOfStateActionPairs.append(getNumberOfStateActionPair(qTable))

            if(currentEpisode >= 4000):
                updatedEpsilon = updateEpsilon(currentEpisode, updatedEpsilon)
            print("Number of States: " + str(len(qTable)))
            print("Steps: " + str(step))

        timeToExecute = time.time() - trialStartTime
            print("Steps: " + str(step))

        timeToExecute = time.time() - trialStartTime
        saveData(qTablesList, diffQ, steps, returns, qTable, timeToExecute, numberOfStateActionPairs, trial+1)
```

Fonte: Autor

Figura 28 – Parte do código utilizado para execução de uma determinada ação

```
# choose action
actionToPerform = mdp.chooseAction(qTable, actions, currentState, choice, forbiddenActions,
                                   forbiddenActionState, updatedEpsilon, currentHeuristicState,
                                   actionsByState)
# perform action in the environment
online.sendMessage(currentState, actionToPerform, socket)
# receive information from environment
newState, reward, end, forbiddenActionPerformed = online.receiveMessage(socket)
# UPDATE Q AND P FUNCTIONS AND ASP FILES
changes = asp.updateASPfiles(qTable, pTable,
        currentState, actionToPerform, newState,
        reward, minimumReward, maximumReward,
        goalState, forbiddenStates, forbiddenActionState, forbiddenActionPerformed,
        forbiddenActions, actions, actionsByState)
somethingChanged = somethingChanged or changes

if(newState in forbiddenStates):
    reward = minimumReward
    end = True
    step = maxSteps - 1

if(actionToPerform in qTable[currentState]):
    qTable[currentState][actionToPerform] = mdp.updateQvalue(qTable, currentState,
                                                             actionToPerform, reward, newState)
heuristicAction = int(actionToPerform)
if(currentHeuristicState in heuristicStateMapping and
   heuristicAction in heuristicStateMapping[currentHeuristicState]):
    currentHeuristicState = heuristicStateMapping[currentHeuristicState][heuristicAction]
else:
    currentHeuristicState = ""

# variables to send
step += 1
returnReward += reward

tSteps += 1
if step == maxSteps:
    end = True
    if(reward != maximumReward):
        reward = minimumReward
```

Fonte: Autor

Figura 29 – Parte do código utilizado atualização dos valores entre Python e ASP

```
def updateASPfiles(qTable, pTable, currentState, action, newState,
        reward, minimumReward, maximumReward,
        goalStates, forbiddenStates, forbiddenActionState, forbiddenActionPerformed,
        forbiddenActions, allActions, actionsByState):
    somethingChanged = False
    removeForbiddenAction = False

    # forbidden state-action pairs. criteria: transition to the same state
    if DETERMINISTIC:
        if (currentState == newState or reward == minimumReward) and [currentState, action]
                                                                    not in forbiddenActionState:
            forbiddenActionState.append([currentState, action])
            if RUN_ASP:
                updateRestrictStates = False
                actionToDelete = []
                actionToDelete.append(action)
                actionsByState[currentState] = set(actionsByState[currentState]) - set(actionToDelete)
                if(len(actionsByState[currentState]) <= 0):
                    forbiddenStates.append(currentState)
                    updateRestrictStates = True
                if(len(actionsByState[newState]) <= 0):
                    forbiddenStates.append(newState)
                    updateRestrictStates = True
                if(updateRestrictStates): #If there is a state where all actions were removed
                    py2asp.restrictStates(forbiddenStates)

                py2asp.restrictSApairs(forbiddenActionState, currentState)

            somethingChanged = True
            removeForbiddenAction = True

    # forbidden actions. criteria: Received from the environment
    if(forbiddenActionPerformed):
        if action not in forbiddenActions:
            forbiddenActions.append(action)
            if RUN_ASP:
                py2asp.restrictActions(forbiddenActions)

            somethingChanged = True
        removeForbiddenAction = True
    # save goal state.criteria: maximum reward
    if reward == maximumReward and newState not in goalStates:
        goalStates.append(newState)
        if RUN_ASP:
            py2asp.saveGoalStates(goalStates)
        somethingChanged = True
    # update Q and P functions
    if currentState not in pTable:
        pTable[currentState] = dict()
        somethingChanged = True
    if currentState not in qTable:
        somethingChanged = True
        if(RUN_ASP):#This update is done faster then calling ASP every time. But it's the same result
            qTable[currentState] = dict()
            for actionAux in actionsByState[currentState]:
                qTable[currentState][actionAux] = random()
        else:
            qTable[currentState] = dict()
            for actionAux in allActions:
                qTable[currentState][actionAux] = random() #0
    if action not in pTable[currentState]:
        pTable[currentState][action] = list()
        somethingChanged = True
    if(not RUN_ASP):
        if action not in qTable[currentState]:
            qTable[currentState][action] = random() #0
            somethingChanged = True
    if newState not in pTable[currentState][action]:
        pTable[currentState][action].append(newState)
        somethingChanged = True
    return somethingChanged
```

Fonte: Autor

Figura 30 – Código utilizado para definir os parâmetros dos experimentos

```
# experiments variables
nenvs = 3
episodes = 6000
trials   = 25
maxSteps = 500

# RL vars
learningRate = 0.2
discount = 0.9 #Favor future actions, rather than recent ones.
initialEpsilon = 0.10
stepToUpdateEpsilon = 250

# Configuration variables
DEBUG = False
RUN_ASP = True
DETERMINISTIC = True
USE_HEURISTIC = True

# Heuristic Variables
heuristicValues = {} #List of heuristic values
heuristicStateMapping = dict()
if(USE_HEURISTIC):
    heuristicFile = open("heuristic", "r")
    stringHeuristicValues = heuristicFile.read()
    heuristicFile.close()
    heuristicValues = ast.literal_eval(stringHeuristicValues)

    import pickle
    import pickle
    from os import popen
    heuristicStateMapping = pickle.load(open("stateMappingHeuristic.p

port = 5001
qDifferenceWindow = 10
graphicValuesWindow = 1
```

Fonte: Autor

Figura 31 – Código utilizado para atualizar o valor do estado/ação na Tabela Q

```
def updateQvalue(qTable, state, action, reward, newState):

    stateActionQ = qTable[state][action]

    if newState not in qTable:
        newStateNewActionQ = 0
    elif not qTable[newState]:
        newStateNewActionQ = 0
    else:
        newStateNewActionQ = qTable[newState][maxAToUpdate(qTable, newState)]

    updatedQ = stateActionQ + learningRate * (reward + discount * newStateNewActionQ - stateActionQ)

    return updatedQ
```

Fonte: Autor

Figura 32 – Código utilizado para atualizar a descrição da tabela Q com informações do ASP

```
def updateQwithASP(currentQTable, newQTable):
    for newState in newQTable:
        if newState in set(currentQTable):
            actionsToDelete = []
            for action in currentQTable[newState]:
                if (action not in set(newQTable[newState])):
                    actionsToDelete.append(action)

            for action in newQTable[newState]:
                if (action not in set(currentQTable[newState])):
                    currentQTable[newState][action] = newQTable[newState][action]
                #If the action is already on the QTable, then the algorithm does nothing

            for actionToDelete in actionsToDelete:
                del currentQTable[newState][actionToDelete]

        else:
            currentQTable[newState] = newQTable[newState]

    return currentQTable
```

Fonte: Autor

Figura 33 – Código utilizado para criar o arquivo ASP com as Ações Proibidas

```
def restrictActions(actions):
    code = ''
    for a in actions:
        code += ':- a(' + str(a) + ').\n'

    save('restrictActions.lp', code)
```

Fonte: Autor

Figura 34 – Código utilizado para criar o arquivo ASP com os pares Estado/Ações proibidos

```
def restrictSApairs(pairs, state):
    code = ''

    for pair in pairs:
        s = pair[0]
        if s == state:
            a = pair[1]
            code += ':- a(' + str(a) + ').\n'

    name = 's' + s + 'r.lp'
    save(name, code)
```

Fonte: Autor

Figura 35 – Código utilizado para criar o arquivo ASP com os Estados Objetivos

```
def saveGoalStates(states):
    code = ''
    for s in states:
        code += ':- s(' + s + ').\n'
    save('goalStates.lp', code)
```

Fonte: Autor

APÊNDICE B – CONFIGURAÇÃO PARA EXECUÇÃO DOS EXPERIMENTOS

Considerando a reprodutividade dos experimentos realizados nesse trabalho, será apresentado a seguir um guia de como configurar os programas necessários e executar os experimentos.

Em primeiro lugar deve-se utilizar o sistema operacional Linux, a distribuição específica usada por esse trabalho foi o Ubuntu 16.04 LTS.

Com o sistema operacional funcionando, deve-se instalar:

a) Python 3.5 - `https://www.python.org/downloads/release/python-350/`;

b) Biblioteca ZeroMQ - `http://zeromq.org/`;

c) SWI-PROLOG - `http://www.swi-prolog.org/Download.html`;

d) CLINGO - `https://potassco.org/clingo/`.

Após a instalação das dependências acima, deve-se baixar o código fonte do experimento desejado diretamente do repositório (Bitbucket) desse trabalho, disponível em: `https://bitbucket.org/thiagomestrado/journalarticle/src`.

Com o código fonte baixo, vá até a pasta "Agente" e execute com Python o arquivo "agent.py". Essa execução faz com que o agente seja iniciado e espere por uma conexão com o ambiente. Portanto, agora vá até a pasta "Ambiente" e execute com Python o arquivo "nomeDoQuebraCabeça.py", em que "nomeDoQuebraCabeça" deve ser substituído pelo nome do quebra-cabeça específico que está sendo executado. Pronto, agora é só esperar o fim do experimento e observar os valores obtidos.

É possível alterar os parâmetros dos experimentos, para isso deve-se ir até o arquivo "exp.py" na pasta "Agente" e fazer as alterações desejadas.

APÊNDICE C – TEMPO DE EXECUÇÃO DOS EXPERIMENTOS

Tabela 7 – Tempo de execução médio, em minutos, para um experimento completo (6000 episódios)

Quebra-Cabeça	Algoritmos			
	oASP(MDP)	HoASP(MDP)	Q-Learning	Q-Learning Heurística
FFO	42.78	6.34	20.64	2.33
FFS	9.06	X	5.09	X
FFND	311.94	32.33	91.01	20.22
FFSND	6.20	X	4.16	X
FFENE	37.53	X	14.67	X
FFDNE	58.99	X	36.55	X
Rope Ladder	72.38	63.47	48.94	26.25

Fonte: Autor