Planejamento probabilístico como busca num espaço de transição de estados

Daniel Javier Casani Delgado

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Mestrado em Ciências da Computação
Área de Concentração: Planejamento em Inteligência Artificial
Orientador: Profª. Drª. Leliane Nunes de Barros

São Paulo, Fevereiro de 2013

Planejamento probabilístico como busca num espaço de transição de estados

Esta dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa realizada por Daniel Javier Casani Delgado em 04/02/2013. O original encontra-se disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Profª. Drª. Leliane Nunes de Barros (orientadora) - IME-USP
- Prof. Dr. Silvio do Lago Pereira - FATEC-SP
- Prof. Dr. Valdinei Freire da Silva - EACH-USP

Agradecimentos

Agradeço a Deus, à minha família no Peru e à minha namorada pelo apoio ao longo de todos estes anos. À minha orientadora pela paciência e ajuda na elaboração deste trabalho.

Resumo

Um dos modelos mais usados para descrever problemas de planejamento probabilístico, i.e., planejamento de ações com efeitos probabilísticos, é o processo de decisão markoviano (*Markov Decision Process* - MDP). Soluções tradicionais são baseadas em programação dinâmica, sendo as mais eficientes aquelas baseadas em programação dinâmica em tempo real (*Real-Time Dynamic Programming* - RTDP), por explorarem somente os estados alcançáveis a partir de um dado estado inicial. Por outro lado, existem soluções eficientes baseadas em métodos de busca heurística em um grafo AND/OR, sendo que os nós AND representam os efeitos probabilísticos das ações e os nós OR representam as escolhas de ações alternativas. Tais soluções também exploram somente estados alcançáveis a partir de um estado inicial porém, guardam um subgrafo solução parcial e usam programação dinâmica para a atualização do custo dos nós desse subgrafo. No entanto, problemas com grandes espaços de estados limitam o uso prático desses métodos. MDPs fatorados permitem explorar a estrutura do problema, representando MDPs muito grandes de maneira compacta e assim, favorecer a escalabilidade das soluções. Neste trabalho, apresentamos uma análise comparativa das diferentes soluções para MDPs, com ênfase naquelas que fazem busca heurística e as comparamos com soluções baseadas em programação dinâmica assíncrona, consideradas o estado da arte das soluções de MPDs. Além disso, propomos um novo algoritmo de busca heurística para MDPs fatorados baseado no algoritmo ILAO* e o testamos nos problemas da competição de planejamento probabilístico IPPC-2011.

Palavras-chave: Planejamento probabilístico, Programação Dinâmica em Tempo Real, Busca em grafos AND/OR com transições probabilísticas.

Abstract

One of the most widely used models to describe probabilistic planning problems, i.e., planning of actions with probabilistic effects, is the Markov Decision Process - MDP. The traditional solutions are based on dynamic programming, whereas the most efficient solutions are based on Real-Time Dynamic Programming - RTDP, which explore only the reachable states from a given initial state. Moreover, there are efficient solutions based on search methods in a AND/OR graph, where AND nodes represent the probabilistic effects of an action and OR nodes represent the choices of alternative actions. These solutions also explore only reachable states but maintain the parcial subgraph solution, using dynamic programming for updating the cost of nodes of these subgraph. However, problems with large state spaces limit the practical use of these methods. Factored representation of MDPs allow to explore the structure of the problem, and can represent very large MDPs compactly and thus improve the scalability of the solutions. In this dissertation, we present a comparative analysis of different solutions for MDPs, with emphasis on heuristic search methods. We compare the solutions which are based on asynchronous dynamic programming which are also considered the state of the art. We also propose a new factored algorithm based on the search algorithm ILAO*. It is also tested by using the problems of the International Probabilistic Planning Competition IPPC-2011.

Keywords: Probabilistic planning, Real-Time Dynamic Programming, Search in graphs AND/OR with probabilistic transitions.

Sumário

Lista de Abreviaturas

MDP	Markov Decision Process
SSP	Stochastic Shortest Path
LAO*	Loop-And-Or
ILAO*	Improved LAO*
RLAO*	Reverse LAO*
BLAO*	Bidirectional LAO*
MBLAO*	Multi-thread Bidirectional LAO*
IBLAO*	Iterative Bounding LAO*
sLAO*	Symbolic LAO*
Fact-ILAO*	Factored ILAO*
RTDP	Real-Time Dynamic Programming
LRTDP	Labeled Real-Time Dynamic Programming
BRTDP	Bounded Real-Time Dynamic Programming
FRTDP	Focused Real-Time Dynamic Programming
sRTDP	Symbolic Real-Time Dynamic Programming
Fact-LRTDP	Factored LRTDP
ADD	Algebraic Decision Diagram
BDD	Binary Decision Diagram
DBN	Dynamic Bayesian Network
SPUDD	Stochastic Planning using Decision Diagrams
APRICODD	Approximate Policy Construction using Decision Diagrams

Lista de Símbolos

V	Conjunto de vértices
$\mathbb{A}$	Conjunto de arcos
N	Conjunto de nós
T	Conjunto de nós terminais
$\mathcal{E}$	Conjunto de hiperarcos
$\mathcal{H}$	Hipergrafo
$\bar{\mathcal{H}}$	Sub hipergrafo
F	Função de transição de estado
$\hat{h}$	Estimativa heurística
S	Conjunto de estados
A	Conjunto de ações
$A(s)$	Conjunto de ações aplicáveis no estado s
C	Função de custo
R	Função de recompensa
P	Função de transição probabilística
γ	Fator de desconto
π	Política
π^*	Política ótima
V	Função valor
V^*	Função valor ótima
$\mathcal{G}$	Conjunto de estados meta
ϵ	Erro
G	Hipergrafo implícito
G'	Hipergrafo solução parcial
$\pi_{G'}$	Política da solução parcial

Lista de Figuras

Capítulo 1

Introdução

1.1 Considerações Preliminares

Um agente pode ser definido como uma entidade que percebe seu ambiente por meio de sensores e interage com ele através de atuadores (Russell e Norvig, 2010). As capacidades de aprendizado, raciocínio, percepção e planejamento são algumas das características desejadas para um agente inteligente. Um agente de planejamento está interessado na síntese automática de estratégias de ações (planos), com base numa descrição formal de ações, observações e objetivos (Geffner, 2002).

Uma das abordagens mais estudadas de planejamento automatizado é chamada de *planejamento clássico*, que faz um conjunto de suposições restritivas sobre o ambiente do agente, entre elas: o agente é único e possue conhecimento completo do ambiente; os efeitos das ações são determinísticos; com definição de metas restritas e tempo implícito (Ghallab *et al.*, 2004). A busca é um dos métodos mais frequentemente usados na resolução de problemas em Inteligência Artificial, em particular, para problemas de planejamento clássico. Dado um espaço de estados de um sistema dinâmico (por exemplo, um agente e sua interação com um ambiente), o problema de planejamento se reduz a encontrar um caminho entre dois estados. Nesta visão, o espaço de estados é um grafo em que os nós representam os estados e os arcos representam as transições de estados através de ações que levam o agente de um estado a outro. Nesse grafo, cada nó é visto como um nó OR, ou seja, um nó que vai para outro nó através de ações alternativas.

O grafo dirigido ilustrado na Figura 1.1 a) representa um espaço de estados de um problema de planejamento clássico em que cada estado é definido pelas variáveis booleanas p e q. Por exemplo, no estado representado por $\{p,\neg q\}$, p tem o valor verdadeiro e q o valor falso. Note que cada arco dirigido é rotulado por uma ação distinta, o que carateriza ações determinísticas. A Figura 1.1 b) representa o mesmo espaço, porém rotula os nós como OR, característica essa geralmente implícita nos grafos de busca usados para representar sistemas determinísticos de transição de estados. O problema de busca é definido por: i) um estado inicial, ii) um conjunto de ações que o agente pode executar (especificadas em termos de uma função de transição de estados ou explicitamente pelo grafo representando o espaço de estados) e iii) um conjunto de estados meta. A solução para esse problema é uma sequência de ações que leve o agente do estado inicial até um estado meta ou falha, se tal sequência não existir. Uma solução eficiente para busca em grafos OR é o algoritmo A* (Hart *et al.*, 1968) que usa informação extraída do próprio problema (ou de caraterísticas gerais da tarefa) para construir heurísticas e tornar o processo de busca mais eficiente. O uso de heurísticas admissíveis permite focar a exploração nos estados relevantes do problema e desconsiderar outros, e permite alcançar soluções ótimas (Haslum e Geffner, 2000).

Para lidar com aplicações reais, as restrições do planejamento clássico devem ser relaxadas. O *planejamento sob incerteza* modifica as suposições do planejamento clássico com relação ao determinismo dos efeitos das ações e/ou observabilidade total. Quando o efeito de uma ação é dado por

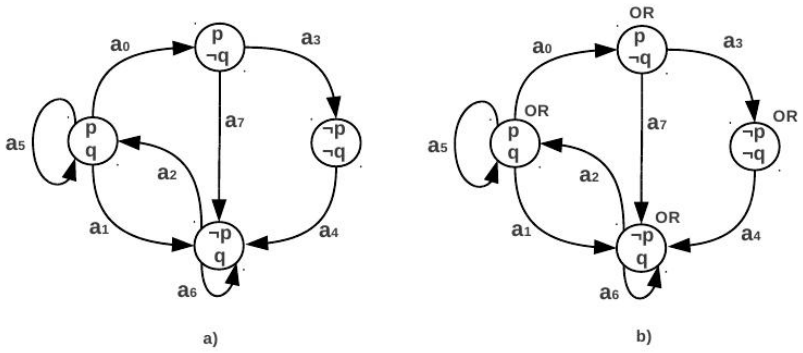

Figura 1.1: *a) Um espaço de estados representado por um grafo ordinário. b) O grafo da figura a) indicando explicitamente que os nós são do tipo OR (nós em que o agente pode escolher dentre várias ações para a mudança de estados).*

um conjunto de estados, dizemos que esse é um problema de ***planejamento não-determinístico***. Quando os efeitos de uma ação são dados em termos de uma distribuição de probabilidades sobre um conjunto de estados, dizemos que esse é um problema de ***planejamento probabilístico***.

Para resolver problemas de planejamento não-determinístico, o espaço de estados deve ser visto como um grafo AND/OR. Em um nó OR, o agente deve escolher uma dentre várias ações alternativas e um nó AND representa os efeitos não-determinísticos da escolha de uma ação. A busca em um grafo AND/OR pode ser considerada um arcabouço mais geral para problemas de busca em espaço de estados: em problemas de planejamento com ações determinísticas, todos os nós do grafo são OR, e as transições entre estados são representadas pelos arcos que levam somente para um estado sucessor. Dessa maneira, essa representação é mais geral e representa grafos ordinários como um caso particular.

Um grafo AND/OR (Nilsson, 1980) também pode ser definido e representado por um hipergrafo. Em um hipergrafo, os k-hiperarcos conectam um estado a um conjunto de k-estados. No planejamento não-determinístico, podemos interpretar que um k-hiperarco representa uma ação com incerteza nos efeitos. A ação transforma um estado em um de seus k possíveis estados sucessores. A Figura 1.2 ilustra várias representações de um mesmo espaço de busca de planejamento não-determinístico. Por exemplo, na Figura 1.2 a) todos os nós do grafo são nós OR e usamos o semicírculo unindo os arcos rotulados por a_0 para indicar uma transição de estados não-determinística com dois possíveis nós sucessores. Na Figura 1.2 b) introduzimos um nó AND para representar explicitamente os possíveis nós sucessores de aplicar a_0 no estado $\{p,q\}$. Note que esses nós não correspondem a estados propriamente ditos, mas são usados para indicar o conjunto de possíveis nós sucessores resultante da execução de uma ação. Na Figura 1.2 c) essas transições não-determinísticas são formalizadas através de um hipergrafo. As três representações são equivalentes e qualquer uma pode ser transformada na outra.

A solução de um grafo AND/OR ou hipergrafo é um subgrafo dirigido que começa no estado inicial e acaba nos estados meta, com ramificações dadas pelos nós AND (ou pelos hiperarcos). O algoritmo AO* (Hansen e Zilberstein, 1998) permite encontrar um grafo solução para resolver o problema de busca ótima num grafo AND/OR ou num hipergrafo. Ele intercala uma fase de expansão (para frente) com uma fase de revisão de custos (para trás). AO* usa uma função heurística admissível para estimar o custo de alcançar os estados meta a partir de um estado qualquer.

Até aqui falamos de planejamento não-determinístico, isto é, em que as ações possuem efeitos não determinísticos, sem informações sobre a probabilidade de ocorrer os diferentes estados. No planejamento probabilístico, o arcabouço formal adotado para representar e resolver problemas é o Processo de Decisão Markoviano (Markov Decision Process - MDP). Sob este arcabouço, o prob-

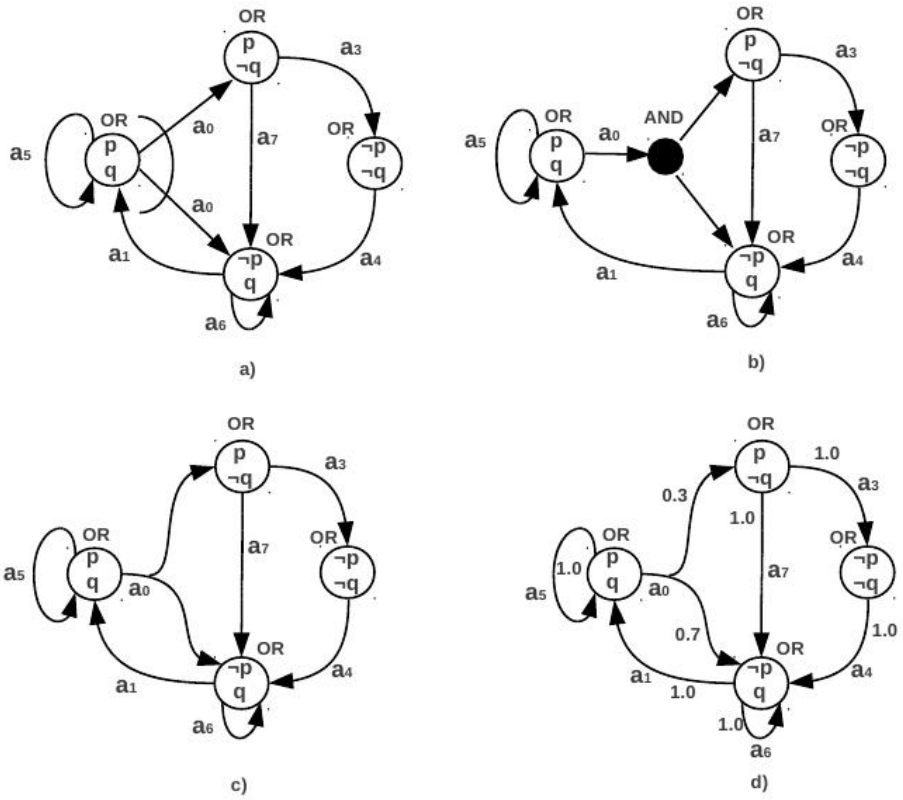

Figura 1.2: *Diferentes representações de um espaço de transição de estados (não-determinístico ou probabilístico) a) Os estados são representados por nós OR e as transições não-determinísticas de uma mesma ação, por arcos unidos por um semi-círculo. b) Representação alternativa introduzindo um nó ficticio AND (círculo preto) ao invés do semi-círculo. Note que o nó AND não representa um estado do mundo. c) A representação da transição não-determinística através de um hiperarco. d) A ação a_0 aplicada em {p,q} determina uma distribuição de probabilidades sobre os nós {p,¬q} e {¬p,q} (0.3 e 0.7 respectivamente).*

lema de planejamento probabilístico pode ser visto como um processo estocástico de decisão que se deseja otimizar. O sistema evolui ao longo do tempo e a cada estágio do processo, um agente observa o estado atual do sistema e escolhe uma ação. A ação escolhida produz dois resultados: o agente recebe uma recompensa (ou custo) e o sistema faz uma transição para outro estado (ou permanece no mesmo estado). O próximo estado é determinado por uma distribuição de probabilidades definida para a ação escolhida no estado atual. Em um processo de decisão markoviano, embora os efeitos das ações sejam incertos, o agente tem percepção completa do estado do ambiente após uma ação ser executada. A solução usual de um MDP é uma política, isto é, uma função que determina a ação que deve ser tomada em cada estado do ambiente. A computação de uma política ótima maximiza a recompensa esperada ou custo esperado (Puterman, 1994).

Iteração de valor (*IV*) (Bellman, 1957) e *Iteração de política* (*IP*) (Howard, 1960) são algoritmos clássicos para resolver MDPs e são baseados em ***programação dinâmica síncrona***. O algoritmo RTDP (*Real-Time Dynamic Programming*) (Barto *et al.*, 1995) é um algoritmo de ***programação dinâmica assíncrona*** que utiliza o conhecimento do estado inicial e heurísticas admissíveis para visitar os estados alcançáveis a partir do estado inicial e acelerar ainda mais o processo de convergência para uma política ótima. Este algoritmo deu origem a uma família de algoritmos relacionados. Por exemplo: o LRTDP (Bonet e Geffner, 2003) etiqueta estados que já convergiram para evitar processá-los novamente e permitir a visita de outros; o BRTDP (McMahan *et al.*, 2005) faz uso de limites superiores e inferiores sobre a função valor para dirigir a escolha dos próximos estados a serem visitados para estados que estão mais longe da convergência.

Para lidar com problemas que possuem grandes espaços de estados, podemos usar representações que explorem a estrutura interna do problema. *MDPs fatorados* permitem representar grandes MDPs de maneira compacta (Guestrin *et al.*, 2003). Nessa abordagem, os estados são descritos por um conjunto de variáveis de estados. O uso de técnicas de agregação de estados evita a enumeração explícita, tratando conjuntos de estados ao invés de estados individuais. Estruturas de dados para representar e compactar eficientemente o espaço de estados tem sido usadas em soluções de programação dinâmica síncrona (SPUDD (Hoey *et al.*, 1999), APRICODD (St-Aubin *et al.*, 2001), etc.) e assíncronas (sRTDP (Feng *et al.*, 2003) e Fact-LRTDP (Gamarra *et al.*, 2012)). Esses trabalhos fazem uso de ADDs *Algebraic Decision Diagrams* (Bahar *et al.*, 1993), BDDs *Binary Decision Diagrams* (Bryant *et al.*, 1986) e DBNs *Dynamic Bayesian Network* (Dean e Kanazawa, 1990).

Uma outra abordagem para resolver MDPs é generalizar as técnicas de busca em espaço de estados AND/OR (ou hipergrafo). Neste trabalho, resolvemos problemas de planejamento em ambientes não-determinísticos e/ou probabilísticos representados como uma busca num hipergrafo (Hansen e Zilberstein, 1998). Assim, um MDP também pode ser modelado como um problema de busca num hipergrafo em que os estados do MDP são os nós do hipergrafo e cada par estado-ação do MDP é representado por um k-hiperarco. A Figura 1.2 d) mostra um hipergrafo para um MDP. No estado $\{p,q\}$ o agente pode escolher a ação a_0 ou a_5. Caso ele escolha a ação a_5, o agente se mantén no mesmo estado. Caso sua escolha seja a ação a_0, o agente vai para o estado $\{p,\neg q\}$ com probabilidade 0.3 e para o estado $\{\neg p,q\}$ com probabilidade 0.7.

Como num MDP um estado pode ser visitado mais de uma vez sob uma política, o tratamento de ciclos na busca em um hipergrafo é necessário na geração de políticas robustas. Por isso, o algoritmo AO* (Martelli e Montanari, 1978) não pode ser usado diretamente para resolver MDPs porque assume uma solução acíclica. Em (Hansen e Zilberstein, 2001) o algoritmo AO* foi estendido para resolver MDPs modelados como problemas de caminho mínimo estocástico (*Stochastic Shortest Path Problem*) (Bertsekas e Tsitsiklis, 1991). Assim, o LAO* (Loop AO*) generaliza o AO* para encontrar soluções com ciclos para MDPs.

Diversas extensões para LAO* foram propostas. ILAO* (Hansen e Zilberstein, 2001) modifica o algoritmo de atualização de custos do LAO* (Iteração de Valor ou Iteração de Política) e faz a atualização baseada numa busca em profundidade, evitando desse modo atualizar várias vezes os mesmos estados e atingir a convergência para uma política ótima para as soluções parciais gulosas. RLAO* (Dai e Goldsmith, 2006) faz uma busca regressiva no espaço de estados, isto é, começa no estado meta e faz a busca pelo estado inicial. BLAO* (Bhuma e Goldsmith, 2003) faz uma busca bidirecional usando dois processos de busca simultaneamente: um orientado a encontrar um estado meta e outro para alcançar o estado inicial. O algoritmo MBLAO* (Dai e Goldsmith, 2007), estende o algoritmo anterior introduzindo caminhos (*threads*) no processo de busca, fazendo várias buscas em paralelo, para frente e para trás. IBLAO* (Warnquist *et al.*, 2010) é um outro algoritmo baseado no LAO* que faz uso de duas funções heurísticas para delimitar o custo ótimo. A busca é guiada pelo limite inferior como no LAO* e o limite superior serve para podar ramos do grafo de busca. O algoritmo sLAO* (Feng e Hansen, 2002) explora abstração de estados e busca heurística. Os conjuntos são representados utilizando ADDs. Usa análise de alcançabilidade para expandir a solução atual e uma versão do algoritmo SPUDD para atualizar os custos.

Tanto pela simplicidade conceitual quanto pelos resultados experimentais, o algoritmo ILAO* apresenta o melhor desempenho dentre as diversas estensões do algoritmo LAO* (Dai *et al.*, 2011).

1.2 Motivação

Os algoritmos para resolver MDPs baseados em programação dinâmica síncrona (Iteração de Valor e Iteração de Política), precisam atualizar todos os estados do espaço de estados em cada iteração. Apesar de sua complexidade computacional ser polinomial no número de estados, esse número cresce exponencialmente com o número de variáveis de estado (Papadimitriou e Tsitsiklis, 1987). Portanto, a construção de soluções eficientes para MDPs é um tópico de grande importância.

O algoritmo RTDP e suas extensões resolvem MDPs restringindo a programação dinâmica aos estados alcançáveis a partir do estado inicial (programação dinâmica assíncrona), sendo uma melhor alternativa do que Iteração de Valor ou Iteração de Política. O algoritmo LAO* e suas extensões, também só visitam estados alcançáveis, porém mantém o subgrafo solução parcial.

Os resultados experimentais apontam que, em alguns casos, os algoritmos baseados no ILAO* possuem menores tempos de convergência e fazem menor número de atualizações da função valor quando comparado ao algoritmo RTDP e suas extensões (Dai e Goldsmith, 2007). Porém, essas comparações empíricas são feitas basicamente para o domínio *Racetrack* (Barto *et al.*, 1995). Essa restrição impede garantir que os resultados obtidos sejam válidos em outros domínios com diferentes caraterísticas. Por exemplo, domínios de planejamento mais realísticos, que envolvam grandes fatores de ramificação, grandes espaços de estados, maior número de ações, eventos exógenos, etc. Por outro lado, MDPs com grandes espaços de estados precisam de soluções escaláveis e para isso a representação fatorada pode ser a melhor abordagem.

1.3 Objetivos

O principal objetivo desse trabalho é comparar o algoritmo de busca heurística para MDPs, ILAO* (Hansen e Zilberstein, 2001), com o estado da arte das soluções baseadas em programação dinâmica de tempo real para MDPs, o LRTDP (Bonet e Geffner, 2003), (Kolobov *et al.*, 2012). Além disso, propomos uma nova versão fatorada do algoritmo ILAO* e fazemos um estudo comparativo do desempenho desses algoritmos com problemas que envolvem grandes espaços de estados.

1.4 Contribuições

As principais contribuições deste trabalho são listadas abaixo:

- Revisão e levantamento bibliográfico dos algoritmos baseados em busca heurística e aqueles baseados em programação dinâmica assíncrona e que compõem o estado da arte em métodos de solução para planejamento probabilístico.
- Proposta de um novo algoritmo de busca heurística para MDPs fatorados.
- Implementação de uma versão fatorada e enumerativa do algoritmo ILAO*.
- Análise empírica do algoritmo proposto no ambiente de simulação da Competição Internacional de Planejamento Probabilístico IPPC-2011.

1.5 Organização do Trabalho

O presente trabalho estuda os métodos de busca em IA e sua generalização para resolver MDPs fatorados. Por tal motivo, no Capítulo 2, apresentamos os conceitos fundamentais da resolução de problemas usando busca em grafos ordinários, grafos AND/OR e hipergrafos. Descrevemos como essas técnicas resolvem problemas de planejamento clássico e não-determinístico. No Capítulo 3, é apresentada a formalização dos Processos de Decisão Markovianos e fazemos uma revisão dos

métodos de resolução baseados em programação dinâmica com ênfase nos que resolvem MDPs de maneira assíncrona (algoritmos baseados no RTDP). Além disso, mostramos como um MDP pode ser representado por um hipergrafo e resolvido usando algoritmos de busca heurística para MDPs, especialmente o algoritmo ILAO*. No Capítulo 4 revisamos a representação fatorada de MPDs e os métodos de resolução mais conhecidos na literatura. No Capítulo 5 propomos o algoritmo ILAO* Fatorado, suas caraterísticas e implementação. No Capítulo 6 apresentamos as características do ambiente da competição e discutimos os resultados das implementações realizadas. Finalmente, no Capítulo 7 são apresentadas as conclusões e as perspectivas de trabalhos futuros.

Capítulo 2

Fundamentos de Busca

O termo busca está relacionado ao processo de encontrar soluções para problemas num espaço de possíveis soluções. Por fornecerem um arcabouço geral para resolução de problemas, os métodos de busca são bastante empregados na construção de agentes em Inteligência Artificial. Um agente inteligente que pretende resolver um problema, pode aplicar um processo de busca para encontrar a solução desejada. O caso mais geral de busca assume que o espaço de possíveis soluções (também chamado de *espaço de estados*) é dado por um modelo baseado em estados do ambiente e transições de estados, e o que se deseja é encontrar um caminho entre um estado inicial e um estado meta conhecidos (eventualmente se deseja encontrar um caminho mínimo entre dois estados em termos de passos ou custo). Existem duas principais formulações para problemas de busca:

- o espaço de estados é dado por um grafo em que os vértices são os estados e os arcos as transições de estados e
- o espaço de estados é dado por um estado inicial e uma *função geradora de estados sucessores* $F(s)$, sendo essa a única opção para representar espaços de estados muito grandes.

Neste capítulo, definiremos diferentes tipos de espaços de estados e algoritmos de busca para eles. Veremos como resolver problemas de busca em IA, usando uma função geradora de estados (isto é, sem termos como entrada um grafo representando explicitamente o espaço de estados completo).

2.1 Tipos de espaços de estados

O espaço de estados de um problema de busca pode ser modelado por um grafo ordinário (grafo OR) ou um grafo do tipo AND/OR.

2.1.1 Grafos ordinários

Um grafo dirigido acíclico (*Directed Acyclic Graph* - DAG) é um par $G=(V,\mathbb{A})$ composto de um conjunto V (não necessariamente finito) de vértices (ou nós) e um conjunto $\mathbb{A}$ contendo pares de vértices ordenados. Cada elemento de $\mathbb{A}$ é chamado de *arco*. Um arco (v_i,v_j) com $v_i,v_j \in V$ é um par ordenado. Se um arco está dirigido do vértice v_i para o vértice v_j, então o vértice v_j é sucessor ou filho do vértice v_i e o vértice v_i é antecessor ou pai de v_j. O grau de entrada de um vértice v é o número de arcos que incidem nele. O grau de saída de um vértice w é o número de arcos que saem de w.

Uma árvore dirigida é um caso particular de um DAG, em que cada vértice tem somente um antecessor. O vértice sem antecessor chama-se vértice *raiz*. Um vértice sem sucessores denomina-se vértice *folha*. A *profundidade* do vértice raiz é zero. A profundidade de qualquer outro vértice da árvore é definida como a profundidade do seu antecessor mais 1. Algumas árvores tem a propriedade que todos os seus vértices, exceto os vértices folha, tem o mesmo número b de sucessores. Assim, b chama-se *fator de ramificação* da árvore. Também podemos definir o fator de ramificação como

a média do número de vértices sucessores. Esse fator permite expressar algumas medidas de complexidade do DAG. Uma sequência de vértices $\langle v_1,v_2,...,v_k\rangle$ onde cada v_{i+1} é sucessor de v_i, para

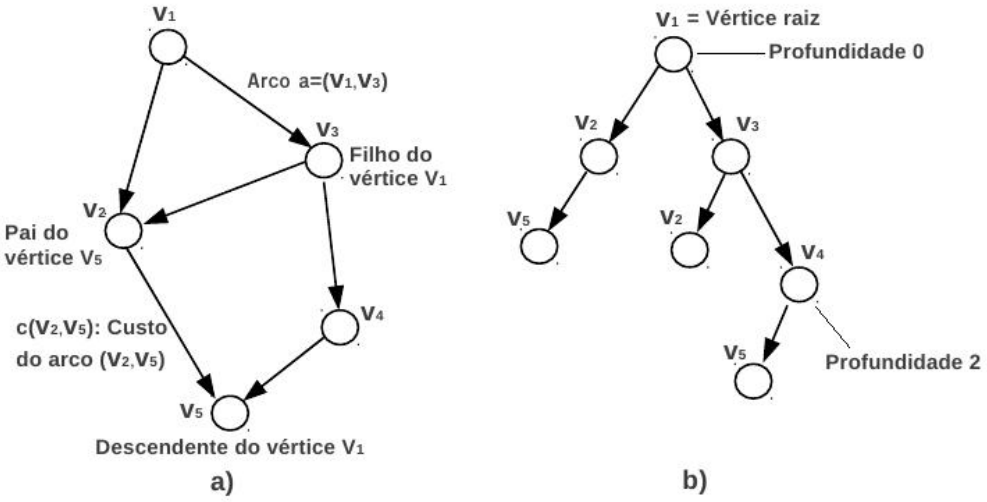

Figura 2.1: *a) Grafo dirigido. b) Árvore de busca do grafo. Adaptado de (Nilsson, 1980).*

i=1,...,*k-1*, chama-se *caminho* de comprimento *k-1* do vértice v_1 até o vértice v_k. Se existir um caminho do vértice v_i até o vértice v_j, então o vértice v_j é acessível a partir do vértice v_i. Também dizemos que o vértice v_j é descendente do vértice v_i e o vértice v_i é ancestral do v_j. Assim, o comprimento do caminho que une a raiz com v_j é sua profundidade na árvore.

Às vezes é conveniente atribuir custos aos arcos. Usaremos $c(v_i,v_j)$ (ou $c(a)$ quando a for um rótulo atribuido ao arco dirigido (v_i,v_j)). Um grafo G é chamado *ponderado* se seus arcos tiverem valores reais. Geralmente, assumimos que os valores são positivos. O custo do caminho entre dois vértices será a soma dos custos de todos os arcos que conectam os vértices do caminho.

Um grafo dirigido $G=(V,\mathbb{A})$ é ilustrado na Figura 2.1(a) contendo o conjunto de vértices $V=\{v_1,v_2,v_3,v_4,v_5\}$ e o conjunto de arcos $\mathbb{A}=\{(v_1,v_2), (v_1,v_3),(v_3,v_2),(v_3,v_4),(v_2,v_5),(v_4,v_5)\}$. O vértice v_3 tem grau de entrada 1 e grau de saída 2. Um caminho do vértice v_1 até o vértice v_5 é dado por $\langle v_1,v_3,v_4,v_5\rangle$ e seu comprimento é 3. O conjunto de todos os caminhos de um grafo dado a partir do vértice v_1 é denominado *árvore de expansão* ou *árvore de busca* do grafo (árvore com raiz em v_1). A Figura 2.1(b) mostra uma árvore de expansão com vértice raiz v_1 e com vértices folha v_2 e v_5. Dado um vértice arbitrário v_0 definido como vértice inicial e um conjunto de vértices $\mathbb{O}\subset V$ que correspondem aos vértices definidos como meta, o problema de busca em grafos é definido como encontrar um caminho entre v_0 e algum vértice $v_i\in\mathbb{O}$. Formalmente, o problema corresponde a tupla $\mathcal{PG} = (V,\mathbb{A},v_0,\mathbb{O})$.

Para alguns problemas, é necessário encontrar o caminho que tenha o custo mínimo entre dois vértices. Esse caminho será conhecido como *caminho ótimo*. O algoritmo de Dijkstra com $O(|V|+|\mathbb{A}|\log|V|)$ (Dijkstra, 1959) e o algoritmo de Bellman-Ford com $O(|V||\mathbb{A}|)$ (Bellman, 1958), (Ford, 1956), resolvem esse problema.

Em grafos explícitos, isto é, grafos cujos vértices e arcos estão completamente representados na memória do computador, podemos encontrar um caminho até algum vértice v_i, a partir de qualquer outro vértice do grafo. No entanto, para problemas de caminho mínimo em que o grafo não pode ser representado completamente na memória, mas em que é conhecida a função geradora de estados sucessores, é possível ainda encontrar caminhos mínimos usando técnicas de busca da Inteligência Artificial como por exemplo o algoritmo A* (Seção 2.2.2).

Como foi discutido na introdução, um grafo ordinário pode ser usado para representar um espaço de estados. Cada vértice deste grafo representa um estado do mundo em que o agente deve fazer escolhas alternativas para executar uma transição de estados que representam ações. Assim, podemos caraterizar os vértices do grafo como OR e os arcos como ações. Neste texto, usaremos o termo nó para nos referir aos vértices de um grafo.

2.1.2 Grafos AND/OR

Os grafos AND/OR foram descritos inicialmente em (Martelli e Montanari, 1978) e em (Nilsson, 1980). Originalmente foram usados para modelar esquemas de *redução de problemas* (Bullers *et al.*, 1980). Existem três representações convencionais para grafos AND/OR: (1) como um grafo dirigido com nós etiquetados como OR ou AND (Figura 2.2 a)); (2) como nós OR com marcações nos arcos do tipo AND (Figura 2.2 b)); e (3) como hipergrafos (Figura 2.2 c)).

Representação de grafos AND/OR com grafos ordinários

Um grafo AND/OR com os nós etiquetados com OR ou AND é um DAG $G=(N,\mathbb{A})$, tal que os nós $n\in N$ são do tipo OR ou AND. Os nós definidos como terminais (nós folha) são do tipo OR. O significado destes nós depende muito do domínio que estamos modelando. Por exemplo, em um esquema de redução de problemas, um nó OR representa um problema que pode ser resolvido por qualquer um dos seus nós sucessores, e um nó AND representa um problema que é resolvido somente quando todos os seus nós sucessores são resolvidos. Assim, os arcos representam relações de dependência entre problemas. Por outro lado, em um problema de planejamento não-determinístico, em que os nós representam os estados do mundo e os arcos as transições (ações), os nós OR permitem ao agente escolher ações e analisar diferentes caminhos de escolhas mutuamente excludentes. Os nós AND indicam os possíveis nós resultantes de aplicar uma ação. Neste caso, esses nós AND não correspondem a estados do mundo, mas, são nós fictícios ou artificiais introduzidos pela modelagem dos efeitos não-determinísticos das ações. Podemos usar maneiras alternativas para representar um grafo AND/OR ao invés de utilizar um nó fictício, por exemplo, as Figuras 2.2 (b) e (c) ilustram as outras representações de grafos AND/OR.

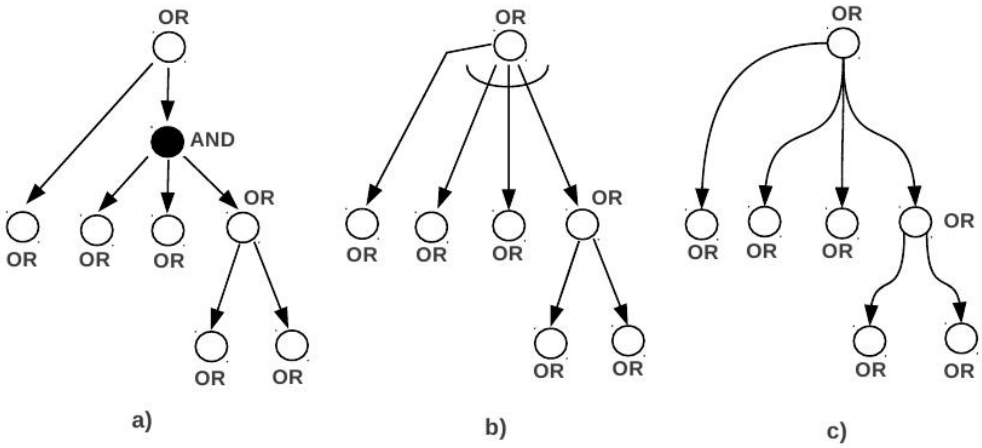

Figura 2.2: *Diferentes representações de um espaço de transição de estados a) Um grafo AND/OR com os estados representados por nós OR e um nó fictício AND. b) Um grafo AND/OR com arco AND identificado pela linha curva. c) Representação do espaço de estados por um hipergrafo, em que os estados são representados por nós OR e os efeitos das ações pelos hiperarcos (AND).*

Representação de grafos AND/OR com hipergrafos

Um hipergrafo é uma generalização do conceito de grafo ordinário. Em um hipergrafo, os hiperarcos conectam conjuntos de nós. O conceito foi amplamente estudado por Claude Berge na área de otimização combinatória (Berge, 1980) e usado como ferramenta de representação em diferentes áreas de pesquisa, por exemplo, no projeto de bancos de dados relacionais (Fagin, 1983), no problema da satisfatibilidade booleana (SAT) (Gallo *et al.*, 1996), na análise de sistemas de tráfego (Nguyen *et al.*, 2001), na modelagem de sistemas nebulosos e de inteligência artificial (Ausiello e Giaccio, 1997). Neste trabalho mostraremos que os hipergrafos podem ser usados para representar grafos AND/OR e MDPs Capítulo 3.

Definição 2.1 (Hipergrafo). Um hipergrafo é definido pelo par $\mathcal{H}=(N,\mathcal{E})$ em que $N=\{n_1,n_2,...,n_{|N|}\}$ é o conjunto de nós e $\mathcal{E}=\{e_1,e_2,...,e_{|\mathcal{E}|}\}$ é o conjunto de hiperarcos, tal que $e_i \subseteq N$ $\forall i=1,2,...,|\mathcal{E}|$.

Definição 2.2 (Hiperarco dirigido). Um hiperarco dirigido $e \in \mathcal{E}$ é um par ordenado $e=(T(e),H(e))$ de subconjuntos de nós, $T(e)$ é o *conjunto de partida* de e e $H(e)$ é o *conjunto de chegada* de e (Kristensen e Nielsen, 2006). Quando $|T(e)|=1$ e $|H(e)|=1$, $\forall e=(T(e),H(e))$, isto é, os hiperarcos unem dois nós, o hipergrafo coincide com o grafo ordinário. A Figura 2.3 representa um hipergrafo com hiperarcos dirigidos. Por exemplo, o hiperarco $e_1=(T(e_1),H(e_1))$ conecta o conjunto de nós $T(e_1)=\{n_1,n_2\}$ ao conjunto de nós $H(e_1)=\{n_4,n_5,n_6\}$.

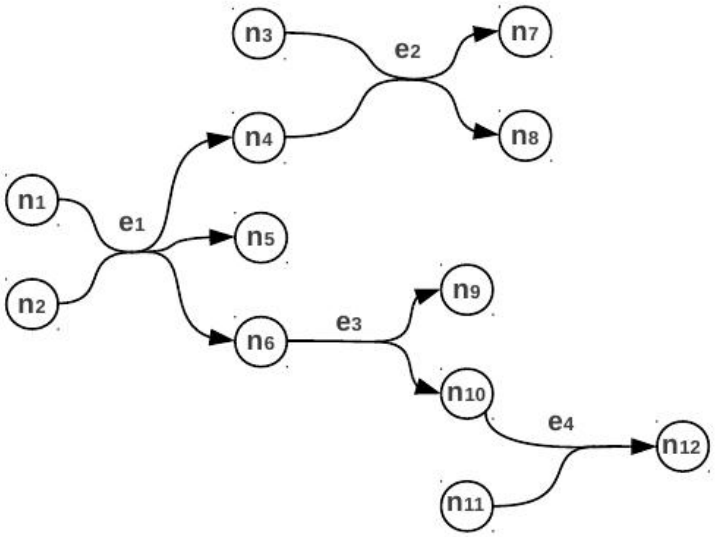

Figura 2.3: *O hiperarco* $e_1=(\mathrm{T}(e_1),\mathrm{H}(e_1))$ *é composto pelos subconjuntos* $\mathrm{T}(e_1)=\{n_1,n_2\}$ *e* $\mathrm{H}(e_1)=\{n_4,n_5,n_6\}$.

Um hiperarco $e=(T(e),H(e))$ é *para trás* se $|H(e)|=1$. Um hiperarco $e=(T(e),H(e))$ é *para frente* se $|T(e)|=1$. Um *B-hipergrafo* é um hipergrafo dirigido em que os hiperarcos são para trás. Um *F-hipergrafo* é um hipergrafo dirigido em que os hiperarcos são para frente. Um *BF-hipergrafo* é um hipergrafo dirigido em que os hiperarcos são para trás ou para frente. Na Figura 2.4 mostramos os tipos de hiperarcos que podem ser parte de um hipergrafo dirigido. Na parte (a) da figura, um hiperarco para trás e_1 em que $|T(e_1)|=2$ e $|H(e_1)|=1$, isto é, o conjunto de partida possui dois nós e o conjunto de chegada um nó. Em (b), um hiperarco para frente e_2 em que $|T(e_2)|=1$ e $|H(e_2)|=3$. Neste trabalho, usaremos apenas os hiperarcos para frente.

Em um hipergrafo ponderado, w é uma função que atribui para cada hiperarco e_i um valor real $w(e_i)$. Dependendo do problema, o valor $w(e)$ representa custo, comprimento, etc.

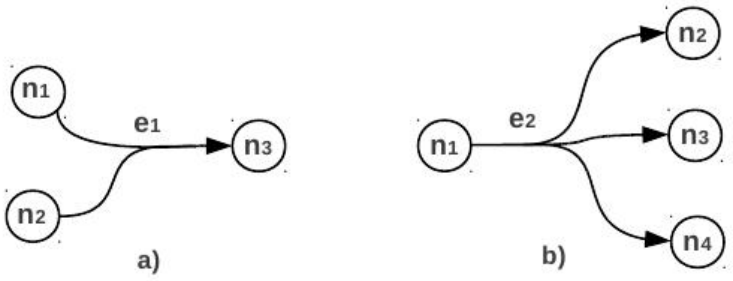

Figura 2.4: *a) Hiperarco para trás. b) Hiperarco para frente.*

Um hipergrafo $\bar{\mathcal{H}}=(\bar{N},\bar{\mathcal{E}})$ é um sub hipergrafo de $\mathcal{H}$, se $\bar{N}\subset N$ e $\bar{\mathcal{E}}\subset\mathcal{E}$. Um sub hipergrafo é próprio se alguma das inclusões é estrita.

Definição 2.3 (Hipercaminho). Um *hipercaminho* $\pi_{ot}=(N_\pi,\mathcal{E}_\pi)$ a partir de um nó origem o até um nó destino t, é um sub hipergrafo de $\mathcal{H}$ tal que, se $o=t$, então $\mathcal{E}_\pi=\emptyset$, caso contrário, $q\geq 1$ hiperarcos em $\mathcal{E}_\pi$ podem ser ordenados numa sequência $(e_1,...,e_q)$ tal que:

1. $t=H(e_q)$.
2. $T(e_i)\subseteq\{o\}\cup\{H(e_1),...,H(e_{i-1})\}$, $\forall e_i\in\mathcal{E}_\pi$.
3. Nenhum sub hipergrafo próprio de π_{ot} é um hipercaminho desde o até t.

A Figura 2.5 apresenta um grafo AND/OR representado por um hipergrafo (N,C) em que $N=\{n_0,n_1,n_2,n_3,n_4,n_5,n_6,n_7,n_8\}$ e conjunto de k-hiperarcos $C=\{(\{n_0\},\{n_1\}),(\{n_0\},\{n_4,n_5\}),(\{n_1\},\{n_2\}),(\{n_1\},\{n_3\}),(\{n_2\},\{n_3\}),(\{n_2\},\{n_4,n_5\}),(\{n_3\},\{n_5,n_6\}),(\{n_4\},\{n_5\}),(\{n_4\},\{n_8\}),(\{n_5\},\{n_6\}),(\{n_5\},\{n_7,n_8\}),(\{n_6\},\{n_7,n_8\})\}$. Sob esta representação, os nós representados explícitamente correspondem aos nós tipo OR e os nós AND são representados implicitamente pelos hiperarcos.

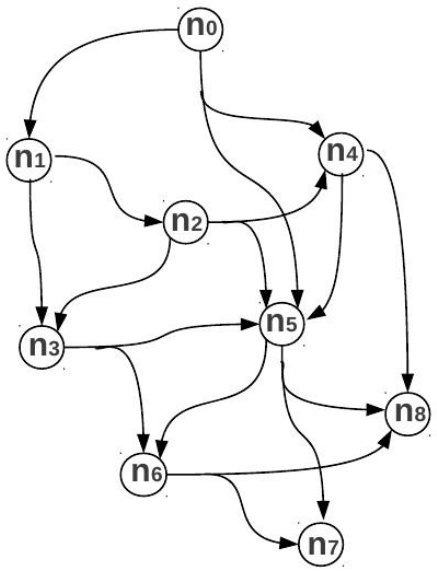

Figura 2.5: *Instância de um F-hipergrafo. Por exemplo, n_0 é nó de entrada do 1-hiperarco $(\{n_0\},\{n_1\})$ e do 2-hiperarco $(\{n_0\},\{n_4,n_5\})$ e os nós n_7 e n_8 são nós de saída do 2-hiperarco $(\{n_6\},\{n_7,n_8\})$.*

Neste trabalho, quando mencionarmos o termo hipergrafo estaremos nos referindo a um F-hipergrafo.

2.2 Busca em espaço de estados modelado como um grafo OR

O ambiente em que um agente inteligente está inserido pode ser representado por um grafo dirigido cujos nós representam os estados e os arcos representam as transições de estados causadas pela execução de ações. Esse tipo de grafo chama-se *grafo de transição de estados* ou *espaço de estados*. Se as ações do agente forem determinísticas, isto é, o efeito da execução de uma ação leva o agente para um estado determinado, o espaço de estados da busca pode ser modelado por um grafo ordinário do tipo OR. Se o espaço de estados for suficientemente pequeno, esse grafo pode representar todos os estados e todas as ações de maneira explícita, na memória do computador ou robô.

Porém, se o espaço de estados for muito grande é preciso representar o espaço de estados de maneira implícita, definindo um nó inicial, que modela o estado inicial do agente e uma função que representa as transições provocadas pelas ações do agente. Essa função é chamada de *função de transição de estados* ou *função geradora de estados sucessores*. Por exemplo, a função de transição de estados pode ser definida por um conjunto de *operadores*. Um operador define as condições que devem ser satisfeitas no estado em que se deseja executar uma ação e as propriedades que deverão valer no(s) estado(s) sucessor(es), através de uma lista de efeitos. Através de um conjunto de operadores, é possível induzir o espaço de estados. Além disso, é necessário definir uma *condição ou teste de meta* que determina se um estado é um estado meta.

Exemplo 2.1 Domínio do Aspirador de Pó (Aspirador I)

No domínio do aspirador de pó (Russell e Norvig, 2010), o agente é um aspirador de pó e sua função é limpar salas de um prédio. Numa versão simplificada, consideramos somente duas salas. Identificamos a sala esquerda como sala 1 e a sala direita como sala 2. Para esse problema temos que:

- *Os estados do problema são determinados pela localização do agente em uma das salas e pelas condições de sala limpa ou sala suja. Assim, temos 2 × 2^2=8 possíveis estados (Figura 2.6);*
- *O estado inicial pode ser qualquer um dos 8 estados da Figura 2.6;*
- *As ações Aspirar, Mover-para-Esquerda e Mover-para-Direita são definidas pelos operadores da Figura 2.7. Cada operador estabelece uma precondição, isto é, uma condição que deve ser satisfeita para que a ação possa ser aplicada. O efeito define o resultado da aplicação do operador; e*
- *O teste de meta verifica se todas as salas estão limpas.*

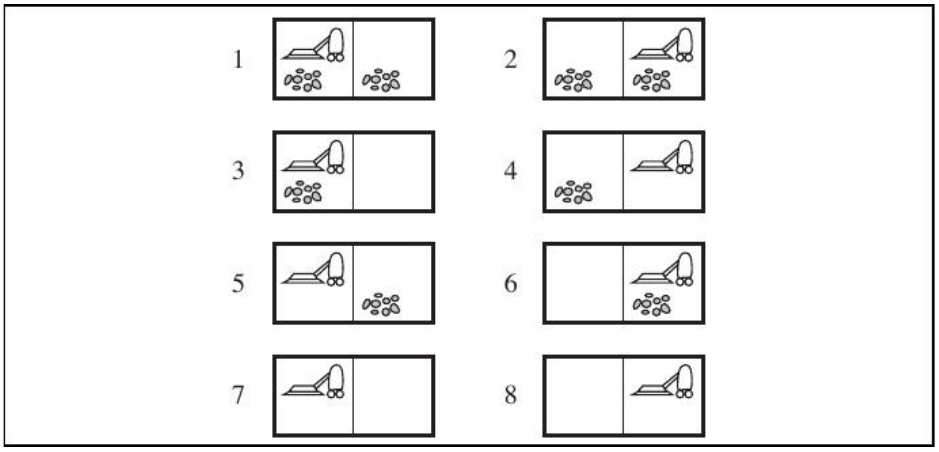

Figura 2.6: *Possíveis estados para o domínio do aspirador. Os números ao lado das figuras identificam os estados. Fonte: (Russell e Norvig, 2010).*

Formalmente, um problema de busca em um espaço de estados qualquer é definido pela tupla: $\mathcal{P} = (S,A,s_0,M)$ em que:

- S é um conjunto de estados. Cada estado é uma descrição da configuração do ambiente ou domínio de atuação do agente, em que o agente está inserido.
- A é um conjunto finito de ações determinísticas, que podem ser representadas por operadores (isto é, uma função de transição de estados). Por exemplo, para o problema do aspirador de pó, os operadores definidos na Figura 2.7 definem como as ações do agente modificam o estado e suas condições de aplicabilidade. Denotamos por $A(s) \subseteq A$, o conjunto de ações aplicáveis no estado $s \in S$. Note que o conjunto A pode ser usado para definir a função geradora de estados sucessores $F(s,a)$, que carateriza os estados sucessores de s quando a ação a é executada.
- $s_0 \in S$ é o estado inicial.
- $M \subseteq S$ é o conjunto de estados meta que devem ser atingidos pela busca. Os estados desse conjunto satisfazem uma propriedade desejada e são usados como condição de parada da busca. No caso do agente aspirador os estados 7 e 8 da Figura 2.6 são estados meta.

Também, podemos associar um valor numérico a cada ação para indicar o custo de executar uma ação em um estado. A solução do problema de busca $\mathcal{P}$ é definida como $\pi_s = (a_1,a_2,...,a_k)$, dada por uma sequência ordenada de ações $a_i \in A$ que transformam o estado s_0 em algum estado meta $s' \in M$, isto é, existe uma sequência de estados u_i tal que $u_0 = s_0$, $u_k = s'$ e u_i é o resultado de aplicar a_i em u_{i-1}.

```
;; X é a sala 1 ou a sala 2.
;; em(X): O agente está na sala X.
;; limpa(X): A sala X está limpa.

Ação Aspirar(X)
      precondições: Ø
      efeito: limpa(X)

Ação Mover-para-esquerda
      precondições: Ø
      efeito: em(sala 1)

Ação Mover-para-direita
      precondições: Ø
      efeito: em(sala 2)
```

Figura 2.7: *Operadores que representam as condições e efeitos das ações do domínio do aspirador.*

Uma função geradora de estados ou função de transição de estados pode ser definida em termos de um conjunto de operadores. O espaço de estados desse problema pode ser representado explicitamente pelo grafo da Figura 2.8, em que os rótulos nos arcos representam as ações S (Suck), L (Left), R (Right), isto é, Aspirar, Mover-para-esquerda e Mover-para-direita, respectivamente. Note que o grafo da Figura 2.8 é um grafo do tipo OR, ou seja, todos os nós do grafo representam estados do mundo.

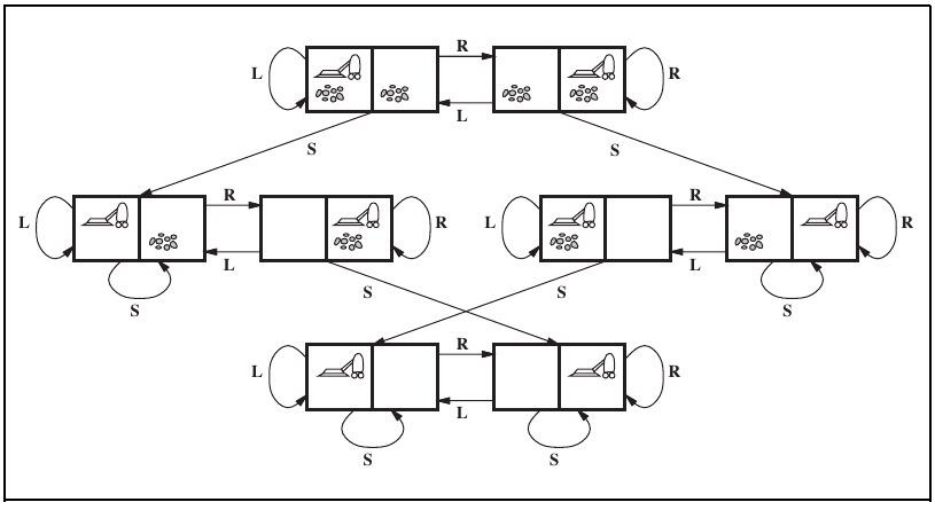

Figura 2.8: *Espaço de estados para o problema do aspirador de pó. Nessa figura, a ação* Aspirar *é representada por* ***S*** *(Suck), Mover-para-Esquerda é representada por* ***L*** *(Left) e Mover-para-Direita é representada por* ***R*** *(Right). Fonte: (Russell e Norvig, 2010).*

Exemplo 2.2 Domínio do N-Puzzle

O problema do n-puzzle (com $n=k^2$-1, $k\geq 2$) que consiste de uma grade $k\times k$ com k^2 posições, (k^2-1) pastilhas numeradas e um espaço em branco. Cada pastilha adjacente ao espaço em branco pode se deslizar para esse espaço. O objetivo é alcançar uma determinada configuração de pastilhas definida como objetivo. Os estados descrevem a localização de cada uma das (k^2-1) pastilhas e do espaço em branco. O estado inicial é qualquer configuração das pastilhas definida como inicial. As ações são os movimentos do espaço em branco para a Esquerda, Direita, Acima ou Abaixo. Dependendo da posição dele, alguns movimentos são possíveis e outros não. O estado meta é determinado por uma configuração de pastilhas arbitrária.

A Figura 2.9 ilustra uma instância do problema 15-puzzle para as configurações definidas como estado inicial e estado meta. Embora, o problema n-puzzle seja aparentemente simples, ele pertence à classe *NP-Completo*. O 15-puzzle tem acima de 10^{13} estados (Edelkamp e Schrodl, 2011). Esse é um exemplo de problema de busca em que o grafo de estados não pode ser completamente representado em memória.

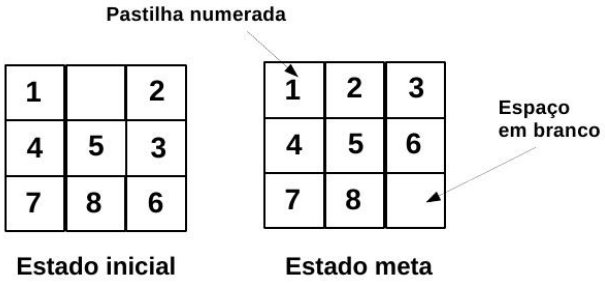

Figura 2.9: *Instância do problema 15-puzzle.*

Definição 2.4 (Correspondência entre um problema de busca em grafos e busca em um espaço de estados).

Um problema de busca em grafos dado por $\mathcal{PG}=(V,\mathbb{A},v_0,\mathbb{O})$ corresponde a um problema de busca em espaço de estados $\mathcal{P}=(S,A,s_0,M)$ sendo: $V\equiv S$, o conjunto de nós; $\mathbb{A}=\{(v_i,v_j)|\ v_i\equiv s_i$, $v_j\equiv s_j$, $\exists a\in A(s_i)$, s_j é o estado resultante da execução de a em s_i, com $v_i,v_j\in V$ e $s_i,s_j\in S\ \}$, o conjunto de arcos, $s_0\equiv v_0$; e $\mathbb{O}\equiv M$ o conjunto de nós meta.

Note que, apesar do problema $\mathcal{P}$ ser definido em termos de S, o conjunto completo de estados, com o uso das ações, os estados $s_j \in S$ podem ser gerados sob demanda. Assim, o problema pode ser redefinido como $\mathcal{P}=(A,s_0,M)$.

A solução de um problema de busca em espaço de estados que corresponde a um grafo G do tipo OR, envolve o uso de um algoritmo que explore o grafo de estados G começando pelo nó s_0 ainda que o grafo não seja dado explicitamente. De fato, dada uma função geradora de nós sucessores $F(s,a)$ (por exemplo, as ações definidas na Figura 2.7 para o domínio do Aspirador) podemos gerar o grafo, a partir do nó inicial s_0, durante a busca por uma solução. Se o algoritmo estiver visitando o nó s_i e aplicar uma ou várias ações, o algoritmo deve gerar todos os nós sucessores $s_j \in S$. Dizemos que s_j foi *gerado* e s_i foi *expandido*. O conjunto de todos os nós que foram visitados a partir de s_0 através da geração de nós sucessores chama-se *conjunto de nós alcançáveis*. Esse conjunto é dividido em *nós expandidos* e *nós gerados* (nós que foram gerados porém ainda não foram expandidos).

Durante a exploração, o algoritmo constrói uma *árvore de busca* (um grafo G' explícito que é um subgrafo de G). A *borda* da árvore (nós folha) corresponde ao conjunto de nós gerados. Dependendo da estratégia de busca, isto é, da ordem de visita dos nós da borda, a árvore de busca é expandida de maneira diferente (ou seja, o grafo explícito G' é diferente). A estratégia selecionada define diferentes medidas de complexidade de tempo, complexidade de espaço, completeza (se o algoritmo garante encontrar a solução, se uma existir) e otimalidade (se o algoritmo encontra a solução ótima).

No domínio do aspirador definimos, por exemplo, que o estado inicial descreve "a *sala 1* e a *sala 2* estão sujas e o agente está na *sala 2*". Dado o conjunto de operadores da Figura 2.7, um algoritmo de busca deve gerar a árvore de busca da Figura 2.10. Note que, somente estão representadas as ações aplicáveis nos estados, sendo que as ações que levam para estados já gerados (ciclos), não são consideradas. Por outro lado, na Figura 2.11 mostramos parte da árvore de busca construída para a instância do problema da Figura 2.9.

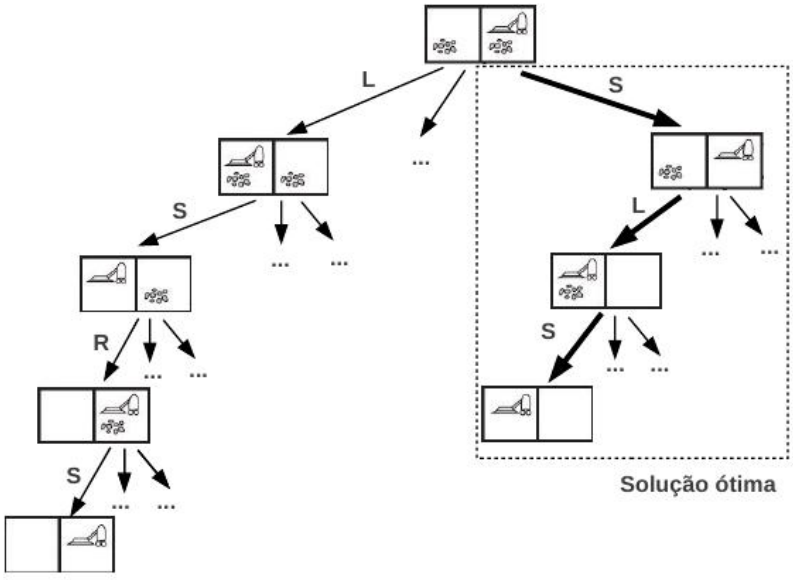

Figura 2.10: *Árvore de busca do domínio do aspirador, que evita expandir nós repetidos.*

2.2.1 Estratégias de busca não-informada

Os algoritmos de busca não-informada, ou busca cega, são aqueles que fazem uma exploração sistemática do espaço de estados (grafo G implícito) até encontrar uma solução ou acabar com falha, isto é, não existe uma solução para o problema dado. Neste tipo de busca existem duas estratégias

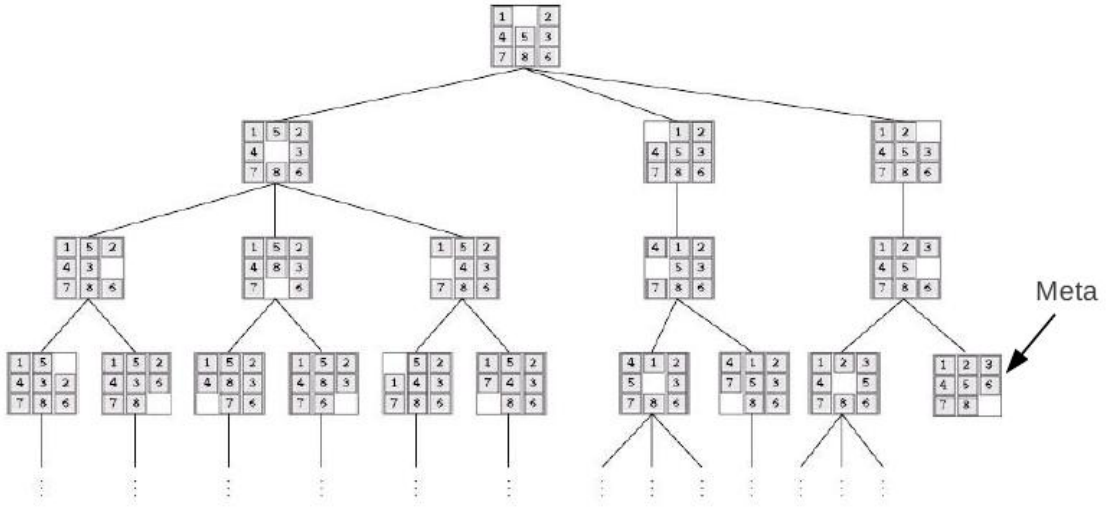

Figura 2.11: *Árvore de busca gerada para uma instância do problema do 8-puzzle. Cada nó está associado a um estado e os arcos representam ações. No ramo mais a direita, um caminho solução une o nó inicial ao nó meta. Fonte: (Ertel, 2011).*

principais de exploração, a *busca em largura* e a *busca em profundidade*.

A *busca em largura* constrói a árvore de busca (G') a partir do nó inicial, visitando todos os nós localizados na mesma profundidade antes que nós em profundidades maiores sejam visitados, isto é, os nós na profundidade n serão visitados antes que os nós na profundidade $n+1$. Desta maneira, cada nível da árvore (conjunto de nós na mesma profundidade) é completamente construído antes que qualquer nó do próximo nível seja adicionado à árvore. Se existir um ou vários nós meta, será encontrado o nó meta mais próximo, ou seja aquele na menor profundidade.

A *busca em profundidade* usa um critério diferente para escolher o nó a expandir. O algoritmo sempre expande o nó mais profundo na borda atual da árvore de busca G' até que uma solução seja encontrada ou até alcançar um nó sem possível expansão seja atingido. Neste caso, a busca é reiniciada no próximo nó mais profundo ainda não expandido.

O Algoritmo 1 descreve um procedimento geral de busca não informada. A lista ABERTOS contém nós que foram gerados porém, ainda não expandidos. A lista FECHADOS contém aqueles nós que foram gerados e expandidos. Os nós do grafo explícito G' são compostos pela união dos nós na lista ABERTOS e FECHADOS. A estratégia E define se a busca será do tipo busca em largura ou busca em profundidade. No primeiro caso, é necessário colocar os novos nós no final da lista ABERTOS (a estratégia E usa uma estrutura de dados fila). Para a busca em profundidade, os novos nós devem ser colocados no início da lista ABERTOS (a estratégia E usa uma estrutura de dados do tipo pilha). Chamamos a árvore de busca de G', que por sua vez é construída durante a busca, sem a necessidade de visitar todo o grafo implícito representando o espaço de estados completo.

Algoritmo 1: BuscaNãoInformada(s_0, $F(s,a)$, A, M, E, c). Fonte: (Nilsson, 1980)

Input:
s_0: Estado inicial.
$F(s,a)$: Função de transição de estados.
A: Conjunto de ações.
M: conjunto de nós meta.
E: Estratégia.
c: Função de custo.
Output: Devolve caminho solução ou FALHA

1 Inicializar o grafo de busca G' com o nó inicial n_0=s_0. Insira n_0 na lista ABERTOS;
`/* A lista ABERTOS pode ser uma fila ou uma pilha. */`
2 Criar a lista FECHADOS inicialmente vazia;
3 **if** ABERTOS estiver vazia **then**
4 | Devolver FALHA;
5 **end**
6 Selecionar um nó de ABERTOS (usando o critério E). Esse nó é chamado de n;
7 **if** n for um nó meta **then**
8 | Devolver a solução (Caminho definido pelos arcos que levam n até n_0 no grafo de busca G');
9 **end**
10 Inserir n na lista FECHADOS;
11 Expandir o nó n (usando as ações $a \in A$ aplicáveis em n), gerando o conjunto de nós sucessores M. Incorporar os nós de M como sucessores de n no grafo de busca G' somente se não estiverem em ABERTOS nem em FECHADOS (evitando assim nós repetidos);
12 Organizar ABERTOS de acordo com a estratégia E;
13 Ir para a linha 3;

Se o problema de busca envolver custo nas ações e a solução deve ser a de menor custo é preciso armazenar em cada nó uma função de custo $g(n)$ que calcula o custo acumulado de n_0 até n. A busca de custo uniforme pode ser facilmente implementada com o Algoritmo 2, usando E como uma fila de prioridades com relação à função $f(n)$ e fazendo $\hat{h}$=0.

As estratégias descritas usam uma *expansão para frente* ou *progressiva*, isto é, começando do nó inicial a busca avança para frente, usando as ações aplicáveis para gerar os nós sucessores, procurando o nó meta. No entanto, a busca também pode começar do nó meta e retroceder buscando o nó inicial, neste caso usamos um *expansão para trás* ou *regressiva*. Essas duas técnicas podem ser combinadas na *busca bidirecional*. A ideia é executar duas buscas simultâneas, a progressiva e a regressiva. Uma delas iniciando do nó inicial e a outra iniciando num nó meta. Quando as buscas se encontrarem, o processo acaba. Na implementação da busca bidirecional verificamos se um nó está na borda da árvore de busca recíproca. Se isso acontecer, um caminho solução foi encontrado.

2.2.2 Estratégias de busca heurística: A*

Os algoritmos de *busca informada* ou *busca heurística* utilizam informação específica da tarefa para fazer o processo de busca mais eficiente. A busca *primeiro o melhor (best-first)* (Russell e Norvig, 2010) seleciona o nó mais promissor para ser expandido. O que faz um nó mais promissor depende do problema a ser resolvido. Por exemplo, a estimativa da menor distância ou do menor custo de um nó até o nó meta. Essa estimativa é dada por uma *função de avaliação*. Quando essa função incorporar informação heurística extraída do problema chama-se *função de avaliação heurística* $\hat{f}$. O algoritmo A* (Hart *et al.*, 1968) permite lidar com essa função para encontrar uma solução ótima.

Algoritmo 2: A*(s_0, $F(s,a)$, A, M, E, $\hat{h}$, c). Fonte: (Nilsson, 1980)

Input:
s_0: Estado inicial.
$F(s,a)$: Função de transição de estados.
A: Conjunto de ações.
M: conjunto de nós meta.
E: Estratégia.
$\hat{h}$: Função heurística.
c: Função de custo.
Output: Devolve caminho solução ou FALHA

1 Inicializar o grafo de busca G' com o nó inicial n_0=s_0. Insira n_0 na lista ABERTOS;
`/* A lista ABERTOS é uma fila de prioridades. */`
2 Criar a lista FECHADOS inicialmente vazia;
3 **if** ABERTOS estiver vazia **then**
4 | Devolver FALHA;
5 **end**
6 Extrair o nó de ABERTOS cujo valor $\hat{f}$ é mínimo. Esse nó é chamado de n;
7 **if** n for um nó meta **then**
8 | Devolver a solução (Caminho definido pelos arcos que levam n até n_0 no grafo de busca G');
9 **end**
10 Inserir n na lista FECHADOS;
11 Expandir o nó n, gerando o conjunto de nós sucessores M. Incorporar os nós de M como sucessores de n no grafo de busca G';
12 **foreach** nó sucessor n' **do**
13 | Se não estiverem em ABERTOS nem em FECHADOS, usar a estimativa $\hat{h}(n')$ e calcular $\hat{f}(n')$=$g(n')$+$\hat{h}(n')$ em que $g(n')$=$g(n)$+$c(n,n')$;
14 | Se já estiver em ABERTOS ou em FECHADOS, direcionar seus arcos ao longo do caminho que atinge o menor valor $g(n)$;
15 | Se n' precisou modificação de arcos e foi encontrado na lista FECHADOS, mover n' para ABERTOS;
16 **end**
17 Organizar a lista ABERTOS em ordem crescente dos valores da função $\hat{f}$ (se alguns valores $\hat{f}$ forem iguais, ordenar em ordem decrescente da profundidade do nó);
18 Ir para a linha 3;

Seja n um nó qualquer na borda da árvore de busca, A* decompõe a função de avaliação heurística $\hat{f}$ como a soma de duas funções de avaliação g e $\hat{h}$, tal que $g(n)$ é definida como o custo do caminho a partir do nó inicial até o nó n e $\hat{h}(n)$ é o custo estimado do caminho de custo mínimo a partir do nó n até o nó meta. Desse modo, $\hat{f}(n)$= $g(n)$+$\hat{h}(n)$ define o custo estimado do caminho de custo mínimo a partir do nó inicial até um nó meta considerando o nó n como parte do caminho. A função $\hat{f}(n)$ é uma aproximação de $f(n)$, função que fornece o custo real do caminho mínimo que une o nó inicial e o nó meta.

Como procuramos o caminho de custo mínimo, selecionar o nó com o menor valor da função $\hat{h}(n)$ será a estratégia certa. No entanto, isso garante a otimalidade somente se $\hat{h}(n)$ for uma *heurística admissível* ou uma estimativa otimista, isto é, não superestima o custo real de atingir o meta, isto é, $\hat{h}(n) \leq h(n)$. Além disso, a condição de *consistência (ou monotonicidade)* sobre $\hat{h}(n)$ tem de ser satisfeita. Essa condição exige que para qualquer nó n e cada sucessor n' de n gerado pela aplicação de uma ação a, o custo estimado de atingir o nó meta a partir de n não pode ser maior

que alcançá-lo a partir n' mais o custo de aplicar a ação. Formalmente, $\hat{h}(n) \leq c(n,n') + \hat{h}(n')$ (Nilsson, 1980). O Algoritmo 2 mostra o algoritmo A*, construído sobre o Algoritmo 1. Para isso, foi modificada a estratégia de ordenação da lista ABERTOS usando uma fila de prioridade sobre os valores da estimativa heurística $\hat{h}$.

Dependendo da heurística utilizada no A* o tamanho de G' pode ser maior ou menor. Duas heurísticas $\hat{h}_1$ e $\hat{h}_2$ admissíveis, que portanto garantem encontrar a solução ótima, podem resultar com desempenho diferente, por exemplo, se o grafo de busca G' gerado por $\hat{h}_1$ for maior que o gerado por $\hat{h}_2$, dizemos que $\hat{h}_2$ é uma heurística melhor que $\hat{h}_1$.

2.3 Busca em espaço de estados modelado como um grafo AND/OR

Nos algoritmos anteriormente apresentados, assumimos que o agente atua em um ambiente completamente observável e com ações determinísticas. No entanto, existem problemas cuja dinâmica é não-determinística ou parcialmente observável. No primeiro caso, é possível gerar uma solução que leve em conta os possíveis estados resultantes da aplicação de uma sequência de ações. Esse plano é chamado de *plano contingente*. Quando realizamos a busca com ações determinísticas, a solução fornece um caminho que alcança um nó meta a partir de um nó inicial. Se as ações tiverem efeitos não-determinísticos, o plano pode produzir mais que um possível caminho que alcance os nós definidos como meta. Nesses ambientes, podemos distinguir três tipos de soluções:

- *Soluções fracas*, são planos que podem atingir um nó meta porém, sem garantias de sucesso.
- *Soluções fortes*, são planos que garantem alcançar um nó meta apesar do não-determinismo.
- *Soluções fortes cíclicas*, garantem alcançar um nó meta assumindo que a execução do plano pode entrar em um ciclo e que eventualmente poderá sair deste.

Podemos redefinir o exemplo 2.2 do problema do aspirador para introduzir não-determinismo definindo que a ação *Aspirar* tem o seguinte comportamento:

- *Quando uma sala suja for aspirada essa sala fica limpa e, às vezes, a outra sala também é limpa.*

Vamos chamar esse novo domínio de *Aspirador II*. Para esse problema, a aplicação de *Aspirar* no Estado 1, pode levar o agente para os estados 5 ou 7 (estados da Figura 2.6) e dizemos que o estado resultante é uma escolha da **Natureza**. A solução do problema precisa lidar com essas contingências construindo um plano solução que não seja simplesmente uma sequência de ações totalmente ordenadas mas sim, uma estrutura condicional (**se-então-senão**) que permita definir as ações dependendo das condições encontradas na execução do plano contingente. Por exemplo, a solução para o novo problema do aspirador deveria conter a seguinte estrutura:

[*Aspirar*, **se** estado = 5, **então** *Mover-para-Direita* e *Aspirar* **senão se** estado = 7, **então** []]

Para representar um ambiente não-determinístico (estados do mundo e as transições não determinísticas entre estados), podemos usar um grafo AND/OR (Seção 2.1.2). Um nó **OR** representa um estado em que o agente deve escolher uma ação e um nó **AND** representa uma ação e os possíveis resultados de sua execução. A Figura 2.12 ilustra o grafo AND/OR do domínio do *Aspirador II*. Os retângulos representam os nós OR e os círculos pretos representam os nós AND.

O problema de busca num grafo AND/OR é definido pela tupla $\mathcal{PBAO} = (N,\mathbb{A},n_0,T)$, em que:

- N é um conjunto de nós, sendo do tipo OR ou AND.
- $\mathbb{A}$ é um conjunto de arcos.

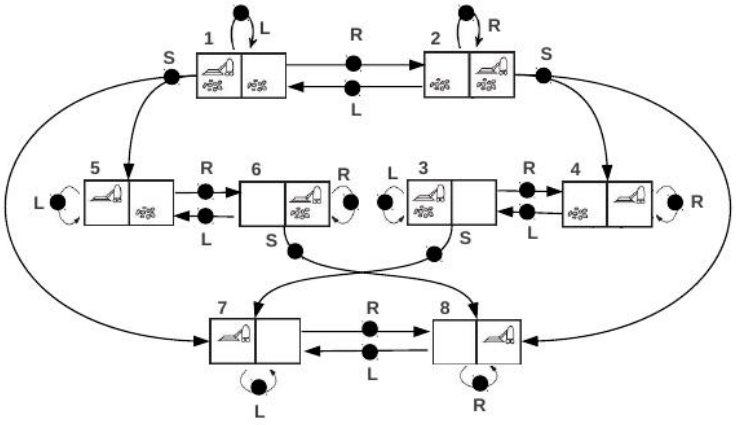

Figura 2.12: *O grafo AND/OR para o domínio do Aspirador II. A ação Aspirar é representada pelo nó AND S (Suck), Mover-para-Esquerda é representada pelo nó AND L (Left) e Mover-para-Direita é representada pelo nó AND R (Right). Fonte: Adaptado de (Russell e Norvig, 2010).*

- $n_0 \in N$ é o nó inicial.
- $T \subset N$ é o conjunto de nós terminais (ou nós meta).

A solução do problema da busca num grafo AND/OR generaliza a noção de solução de busca num espaço de estados representado por um grafo composto unicamente por nós OR (Seção 2.1.2), definida como um caminho em ambientes determinísticos, para transformar-se em ambientes não-determinísticos em uma árvore solução, se não houver nós repetidos (e no caso mais geral, um grafo solução, se houver nós repetidos). Ou seja, dado um grafo AND/OR implícito G (representado pelo estado inicial s_0 e uma função $F(s,a)$), construímos o grafo AND/OR explícito G. A solução que precisamos encontrar adotará a forma de um subgrafo do grafo AND/OR. Para isso, como descrito na Seção 2.1.2, o algoritmo de busca recebe um grafo AND/OR implícito G (através do estado inicial e a função de transição de estados) e constrói um grafo explícito G' que contém o subgrafo solução.

A Figura 2.13 mostra parte do grafo de busca AND/OR gerado a partir do nó 1. As linhas em negrito representam uma solução do problema, isto é, o *subgrafo solução*. Essa solução do problema do *Aspirador II*, começando no estado 1, inclui aplicar a ação Aspirar no nó 1. Se o nó sucessor for o nó 7, o agente está num nó meta, se o nó resultante for o nó 5, o agente deve aplicar a ação Direita, chegando ao nó 6. Nesse nó, deve aplicar a ação Aspirar, atingindo o nó 8 (meta).

Note que o subgrafo solução permite alcançar os nós meta a partir de um nó definido como inicial, considerando tanto nós OR quanto nós AND como parte da solução. Assim, um subgrafo solução de um grafo AND/OR G inclui n_0 e satisfaz as seguintes propriedades:

- Se n for um nó OR e estiver incluído no grafo solução, então um e somente um dos seus arcos, e seus nós sucessores através desse arco, serão incluídos no grafo solução.
- Se n for um nó AND e estiver incluído no grafo solução, então todos seus arcos, e os nós sucessores através desses arcos, serão incluídos no grafo solução.
- Os nós folha do grafo solução correspondem ao conjunto de nós meta.

Nas próximas seções apresentamos dois algoritmos de busca em grafos AND/OR: AO e AO*, sendo que ambos fazem a suposição de grafos AND/OR acíclicos. No Capítulo 3 apresentamos o algoritmo LAO* que estende AO* para encontrar uma solução num grafo cíclico.

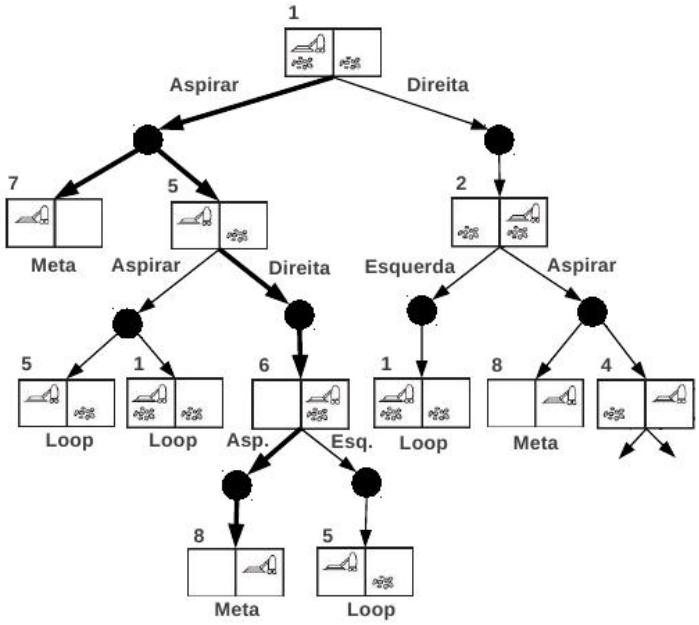

Figura 2.13: *Parte do grafo de busca do domínio do aspirador não-determinístico. As linhas em negrito mostram uma solução do problema. Fonte: Adaptado de (Russell e Norvig, 2010).*

2.3.1 Busca não-informada em grafos AND/OR

Na Seção 2.1.2 definimos que um grafo AND/OR pode ser representado usando um grafo ordinário ou um hipergrafo. Nesta seção, vamos estudar o problema de busca num grafo AND/OR G especificado implicitamente por um estado inicial s_0 e uma função geradora de estados sucessores, isto é, uma função de transição de estados não-determinística. A aplicação dessa função em qualquer nó n gera um número finito de nós sucessores e uma etiqueta específica quando os nós são do tipo OR ou do tipo AND (Martelli e Montanari, 1973) da seguinte forma:

- n_0 é um nó OR.
- Os nós sucessores de um nó OR, são nós do tipo AND.
- Os nós sucessores de um nó AND, são nós do tipo OR.

Um grafo AND/OR também pode ser explorado usando métodos de busca não informada para grafos ordinários, por exemplo, busca em largura, busca em profundidade ou busca pelo melhor primeiro (*best-first*). O Algoritmo 3 permite encontrar o subgrafo solução num grafo AND/OR G acíclico. Esse algoritmo é uma adaptação da busca em largura (Seção 2.2.1). O algoritmo recebe um grafo AND/OR G implícito especificado pelo nó inicial e pela função geradora de estados sucessores e constrói o grafo explícito G' a partir do nó inicial. Duas estruturas de dados do tipo fila são criadas para receber os novos nós gerados (ABERTOS) e para manter os nós já expandidos (FECHADOS).

Em cada iteração do algoritmo, um nó da fila ABERTOS é selecionado para expansão. Esse nó é adicionado à fila FECHADOS e removido da fila ABERTOS. A função de estados sucessores gera os nós sucessores e os novos nós são adicionados na fila ABERTOS. Cada nó sucessor, é avalidado através do Algoritmo 4, etiquetando-o como "Resolvido" ou "Não Resolvido". Para cada nó, atualizamos sua etiqueta dependendo do tipo de nó (OR, AND ou meta) e da relação com seus nós sucessores. A seguir, fazemos um processo de etiquetagem ascendente começando do nó acabado de revisar até chegar ao nó inicial. Se o nó inicial for etiquetado como "Resolvido" então o grafo solução foi encontrado e o algoritmo acaba, caso contrário, o algoritmo faz outra iteração tomando o próximo nó da lista ABERTOS. Como o algoritmo não incorpora informação para determinar qual

nó é o mais promissor, ele seleciona os nós na ordem de geração usando uma estratégia de fila simples.

Algoritmo 3: AO(s_0, $F(s,a)$, A, T). Fonte: (Pearl, 1984)

Input:
s_0: Estado inicial.
$F(s,a)$: Função de transição de estados.
A: Conjunto de ações.
T: Conjunto de nós terminais.
Output: Devolve um subgrafo solução ou FALHA

```
1  O grafo de busca G' consiste inicialmente do nó inicial n0=s0. Insira n0 na fila ABERTOS;
2  Criar a lista FECHADOS inicialmente vazia;
3  if ABERTOS estiver vazia then
4  |  Devolver FALHA;
5  end
6  Extrair um nó da lista ABERTOS. Esse nó é chamado de n;
7  Inserir n na lista FECHADOS;
8  Expandir o nó n, gerando o conjunto de nós sucessores M. Incorporar os nós de M como
   sucessores de n no grafo de busca G' e na fila ABERTOS somente se não estiverem em
   ABERTOS, nem em FECHADOS;
9  foreach nó sucessor n' do
10 |  ETIQUETAR(n');
11 end
12 if nó inicial n0 for etiquetado como "Resolvido" then
13 |  Devolver o subgrafo solução;
14 else
15 |  Ir para a linha 3;
16 end
```

Algoritmo 4: ETIQUETAR(n)

Input: n: Nó

```
1  if n for nó terminal then
2  |  Etiquetar n como "Resolvido";
3  end
4  else if n for nó OR then
5  |  if algum dos seus nós sucessores já foi etiquetado como "Resolvido" then
6  |  |  Etiquetar n como "Resolvido";
7  |  else
8  |  |  Etiquetar n como "Não resolvido";
9  |  end
10 end
11 else if n for nó AND then
12 |  if todos os seus nós sucessores já foi etiquetado como "Resolvido" then
13 |  |  Etiquetar n como "Resolvido";
14 |  else
15 |  |  Etiquetar n como "Não resolvido";
16 |  end
17 end
18 if n foi etiquetado como "Resolvido" then
19 |  Propagar o processo de etiquetagem a partir do n até n0;
20 end
```

Modificamos a definição do problema de busca num grafo AND/OR para definir o problema de busca num hipergrafo. O problema de busca num hipergrafo é definido pela tupla $\mathcal{PBHG} = (N,\mathcal{E},n_0,T)$, em que:

- N é um conjunto de nós.
- $\mathcal{E}$ é um conjunto de hiperarcos.
- $n_0 \in N$ é o nó inicial.
- $T \subset N$ é o conjunto de nós terminais (ou nós meta).

Na seguinte seção, estudamos um algoritmo que permite resolver o problema de busca em um hipergrafo, fazendo uso de informação heurística para dirigir a busca.

2.3.2 Busca heurística em hipergrafos: Algoritmo AO*

O algoritmo AO* pode ser considerado uma generalização do algoritmo A* (Seção 2.2.2) para problemas de busca em um espaço de estados AND/OR acíclico, para isso representamos o problema como um hipergrafo em que os nós representam os estados e os hiperarcos representam as ações e seus possíveis estados sucessores. Assim, modificamos a definição de hiperarco (Seção 2.1.2) para indicar ações. Note que para fins didáticos, definimos o algoritmo AO representando o algoritmo de busca por um grafo AND/OR. No algoritmo AO*, assim como as suas extensões descritas nessa dissertação, usaremos a representação de hipergrafos.

Definição 2.5 (Hiperarco dirigido com ação a). Um hiperarco com ação a é uma tupla ordenada $e=(T(e),a,H(e))$, em que $T(e)$ é o *conjunto de partida* de e, $a \in A$ é uma ação e $H(e)$ é o *conjunto de chegada* de e.

Como o A*, AO* pode encontrar uma solução ótima sem avaliar todo o espaço de estados. A seguir, definiremos alguns conceitos necessários para a formulação do algoritmo. Como descrito anteriormente, o hipergrafo G é conhecido como hipergrafo *implícito*, isto é, G é especificado pelo nó inicial n_0 e pela função geradora de estados sucessores (operadores não-determinísticos). O hipergrafo *explícito* G' representa a parte do hipergrafo que é gerada pelo processo de busca. Ou seja, o processo de busca no hipergrafo G constrói (explicitamente) o hipergrafo G'. Quando um nó for expandido, os seus nós sucessores são adicionados ao hipergrafo explícito G'. Os caminhos neste hipergrafo acabam em *nós folha*, isto é, nós que não tem sucessores no hipergrafo G', podendo ser *não-terminais* (nós ainda não-expandidos uma vez que possuem sucessores em G) ou nós *terminais* (nós meta em G) .

Hipergrafo solução. Em uma busca em hipergrafos a solução adota a forma de um subgrafo acíclico chamado de *hipergrafo solução*. Podemos entender o hipergrafo solução como um subgrafo que conecta o nó inicial n_0 aos nós meta T. De maneira análoga ao caminho solução de um grafo ordinário ou num grafo AND/OR, podemos começar em n_0 e selecionar um hiperarco. Para cada nó sucessor do hiperarco selecionado (por exemplo, aquele com menor custo), escolhemos um hiperarco que parta desse nó e continuamos assim por diante, até que os nós sucessores produzidos pertençam ao conjunto de nós meta T (Martelli e Montanari, 1973).

Um hipergrafo solução é *completo* se cada hipercaminho (Definição 2.3) que começa em n_0 acaba num ou mais nós meta. Se algum hipercaminho acaba num nó ainda não-expandido, então temos um *hipergrafo solução parcial* (HGSP). Formalmente, seja n_i um nó qualquer de um hipergrafo G. Um *hipergrafo solução* $HGS(n_i)$ com nó inicial n_i é um sub-hipergrafo de G em que:

- n_i está em $HGS(n_i)$.

- Se n_j for um nó em G e n_j estiver em $HGS(n_i)$, então exatamente um dos seus k-hiperarcos para frente e todos seus k nós sucessores em G estarão em $HGS(n_i)$.
- Cada caminho em $HGS(n_i)$ acaba em um nó $n_t \in T$.

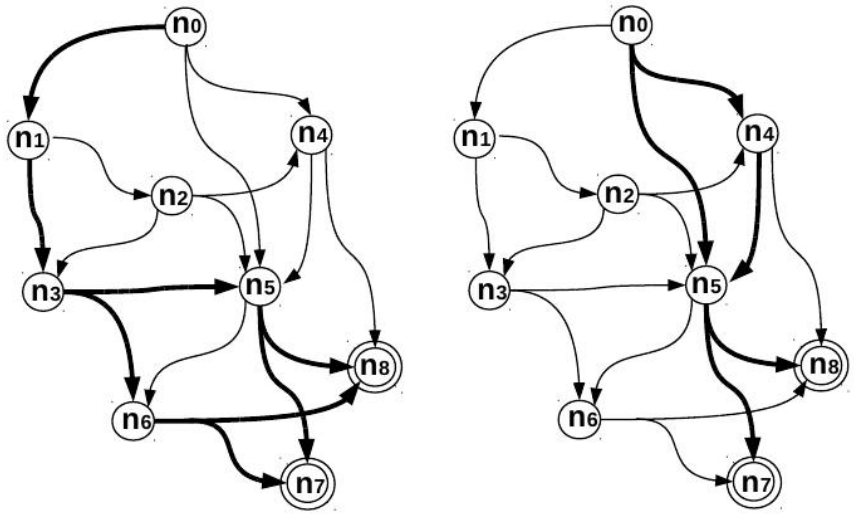

Figura 2.14: *Dois hipergrafos solução do hipergrafo apresentado na Figura 2.5.*

Na Figura 2.14 mostramos dois hipergrafos solução do hipergrafo da Figura 2.5. Em ambos os casos, os grafos satisfazem as condições definidas para o hipergrafo solução. De uma forma geral, dado um hipergrafo G e um nó n_i, chamamos de $HGS(n_i)$ o hipergrafo solução de G que começa em n_i e de $HGSP(n_j)$ o hipergrafo solução parcial que começa em n_j, sendo n_j um nó de $HGS(n_i)$ que também acaba num nó não-terminal de $HGS(n_i)$.

Custo de um hipergrafo solução. Na Seção 2.2 mencionamos que podemos atribuir custos aos arcos de um grafo ordinário. Nos hipergrafos também podemos associar custos para seus hiperarcos. Seja n_i um nó do hipergrafo e seja $(\{n_i\},a,\{n_{i_1},...,n_{i_k}\})$ um k-hiperarco que tem como nó de entrada n_i. Uma função de custo atribui um valor a cada k-hiperarco de G. Denotamos por $c(n_i,a)$ o custo do k-hiperarco associado com aplicar a ação a no nó n_i. Além disso, definimos outra função que atribui um custo para cada nó n_i (similar à função g) e representa o custo acumulado do hipergrafo solução que tem n_i como nó inicial e que seus nós folhas pertencem ao conjunto T. Formalmente, definimos que $f(n_i)$ é o custo do hipergrafo solução do nó n_i até os nós $n_t \in T$ (Jimenez e Torras, 2000):

- Se n_i for elemento de T, então $f(n_i) = 0$.
- Se n_i for nó de entrada do k-hiperarco $(\{n_i\},a,\{n_{i_1},...,n_{i_k}\})$ então $f(n_i) = c(n_i,a) + f(n_{i_1}) + f(n_{i_2}) + ... + f(n_{i_k})$.

Assim, o custo do hipergrafo solução a partir de n_i até T é composto do custo do k-hiperarco mais a soma dos custos dos subgrafos soluções dos nós sucessores de n_i através do k-hiperarco selecionado. *O hipergrafo solução de custo mínimo* de G' é o hipergrafo solução que tem nó inicial n_0 e cujos nós selecionam o k-hiperarco que produz o menor custo de alcançar os nós de T. Intuitivamente, este grafo é determinado escolhendo gulosamente, em cada nó n_i, o hiperarco de menor custo esperado. Formalmente, a seguinte equação recursiva define o custo mínimo do hipergrafo solução com início em n_i (Nilsson, 1980):

$$f^*(n_i) = \begin{cases} 0 & \text{, se } n_i \text{ for nó meta.} \\ \min_{a \in A(i)}[c(n_i,a) + \sum_{j=1}^{k} f^*(n_{i_j})] & \text{, em outro caso, (2.1)} \end{cases}$$

em que, $f^*(n_i)$ denota o custo da solução ótima do nó n_i e f^* é chamada de *função de avaliação ótima*. A Equação 2.1 induz que o hipergrafo solução ótimo seja acíclico. Assim, os nós meta podem ser atingidos a partir do nó inicial após executar um número limitado de ações.

O algoritmo AO* encontra o hipergrafo solução (HGS) de custo mínimo de um hipergrafo G. Como no caso de outros algoritmos de busca heurística, AO* encontra o hipergrafo solução sem considerar todos os nós alcançáveis. O algoritmo AO* utiliza uma função de avaliação heurística $\hat{h}$ como uma estimativa da função de custo ótimo f^*. A função $\hat{h}$ deve satisfazer algumas condições para que o algoritmo possa encontrar o hipergrafo solução ótimo. Assim, definimos $\hat{h}(n_i)$ como sendo a estimativa de custo do hipergrafo com início em n_i e que deve satisfazer as seguintes propriedades:

- $\hat{h}$ é admissível, isto é, para cada nó n_i, $\hat{h}(n_i)$ é uma estimativa que não superestima o custo do hipergrafo solução ótimo com raiz em n_i, isto é, $\hat{h}(n_i) \leq f^*(n_i)$.
- $\hat{h}$ é consistente, isto é, para cada k-hiperarco $(\{n_i\},a,\{n_{i_1},...,n_{i_k}\})$, $\hat{h}(n_i) \leq c(n_i,a) + \hat{h}(n_{i_1}) + \hat{h}(n_{i_2}) + ... + \hat{h}(n_{i_k})$.
- Se n_i for um nó meta, $\hat{h}(n_i) = 0$.

Definição 2.6 (Custo estimado de um hipergrafo solução com início em n_i)
O custo estimado de um hipergrafo solução com início em n_i e usando a ação a, $q(n_i,a)$, sendo que n_i é nó inicial do k-hiperarco $(\{n_i\},a,\{n_{i_1},...,n_{i_k}\})$, representa o custo estimado do hipergrafo solução ótimo a partir de n_i até os nós meta. A seguinte equação define esse custo:

$$q(n_i,a) = c(n_i,a)+q(n_{m_1})+q(n_{m_2})+...+q(n_{m_k}).$$

O algoritmo AO* pode ser definido através de três fases que se repetem em cada iteração do algoritmo:

- **Construção do hipergrafo solução parcial (HGSP)**: Nessa fase, o hipergrafo solução parcial é reconstruído usando as marcações, isto é, seguindo os hiperarcos de menor custo esperado. Os hiperarcos marcados foram escolhidos de maneira gulosa no processo de busca.
- **Expansão do hipergrafo solução parcial (HGSP)**: Nessa fase, sendo que as folhas de um hipergrafo solução parcial podem não ser nós meta, selecionamos e expandimos um desses nós folha e atribuímos um custo estimado para seus sucessores $\hat{h}$.
- **Atualização de custos**: Essa fase envolve a atualização de custos de maneira *botton-up*, marcação de hiperarcos e etiquetagem de nós como "Resolvido". Um nó n_i pode ser considerado resolvido: n_i é um nó meta ou quando todos seus sucessores são "Resolvido". AO* usa programação dinâmica (*backward induction* (Hansen e Zilberstein, 1998)) para a atualização de custos do conjunto de nós formado pelo nó acabado de expandir e seus ancestrais (conjunto Z), de modo que a variação do custo de qualquer nó n_i seja propagado em direção ao nó inicial n_0.

O algoritmo AO* (Algoritmo 5) recebe como entrada o hipergrafo G (isto é, o nó inicial n_0 e a função geradora de estados sucessores) e constrói passo a passo o hipergrafo explícito G'. No início do algoritmo, o hipergrafo G' contém somente o nó n_0 (Linha 1) e o custo do hipergrafo solução, cujo nó inicial é n_0, é determinado pela estimativa heurística $\hat{h}(n_0)$. O algoritmo AO* mantém um número de soluções parciais candidatas e cada nó é etiquetado com a melhor ação atual. Dessa maneira, o melhor hipergrafo solução parcial pode ser selecionado seguindo os hiperarcos marcados nos nós n_i, começando em n_0 (Linha 4).

Em cada iteração, um nó não-terminal n_i do melhor hipergrafo solução parcial é selecionado para expansão (Linha 5). Quando n_i for expandido, todos seus nós sucessores (que não pertençam a G') são adicionados a G'. Cada novo sucessor n_j é inicializado com uma estimativa do custo $\hat{h}(n_j)$

(Linha 6 até linha 9). Como sabemos que o custo da solução a partir do nó n_i depende do custo dos seus sucessores, o custo atual de n_i tem de ser atualizado toda vez que os custos dos nós sucessores mudarem. No algoritmo AO* apresentado, da linha 10 até a linha 23, mostramos essa atualização de custos. Note que os custos dos ancestrais de n_i são atualizados baseados nesse novo custo. Mais precisamente, os ancestrais que podem mudar seu custo estimado $\hat{h}$ são aqueles que tem o nó n_i como descendente através de um caminho de custo mínimo, isto é, o conjunto Z (Linha 21). A ação que fornece o valor mínimo do custo estimado é escolhida como a melhor ação para o nó n_i (Linha 16).

Dado este tipo de atualização de custos, o hipergrafo deve ser acíclico. O algoritmo continua até conseguir o hipergrafo solução completo e seus valores sejam atualizados e propagados até o nó inicial. Em resumo, a fase de expansão para frente do algoritmo AO* usa uma heurística admissível para guiar a busca e na fase de atualização de custos utiliza programação dinâmica para atualizar os custos. Além disso, o algoritmo introduz um processo de etiquetagem para diferenciar aqueles nós que não precisam ser avaliados ou expandidos novamente.

É importante mencionar que a escolha do próximo nó não-terminal do melhor grafo solução parcial a expandir não condiciona o desempenho do algoritmo e não afeta sua otimalidade. Com relação à complexidade computacional do algoritmo AO*, no pior dos casos, o grafo implícito G será completamente explicitado e serão expandidos uma quantidade de nós proporcional ao espaço de estados completo. No entanto, os resultados experimentais mostram que o algoritmo expande muito menos nós e a solução ótima é composta por um número ainda menor de nós.

Algoritmo 5: AO*(s_0, $F(s,a)$, A, T, $\hat{h}$, c), Fonte: (Nilsson, 1980)

Input:
s_0: Estado inicial.
$F(s,a)$: Função de transição de estados.
A: Conjunto de ações.
T: conjunto de nós terminais.
$\hat{h}$: Estimativa heurística.
c: Função de custo.
Output: Devolve o hipergrafo solução ótimo acíclico

```
1  O hipergrafo explícito inicial G' consiste inicialmente do nó inicial n0=s0;
2  Associar ao nó n0 o custo q(n0) = ĥ(n0). Se n0 for nó meta (n0∈T), etiquetar n0 como
   "Resolvido";
3  while n0 não esteja etiquetado como "Resolvido" do
4  |  Construir o hipergrafo solução parcial HGSP, seguindo os hiperarcos marcados de G', a
   |  partir do nó n0;
5  |  Selecionar um nó folha ni de HGSP e expandir o nó ni gerando os seus sucessores
   |  (usando F(s,a) ∀a∈A );
6  |  foreach nó sucessor nj que não esteja já em G' do
7  |  |  Associar o custo q(nj)=ĥ(nj);
8  |  |  Etiquetar como "Resolvido" aqueles nós sucessores que forem nós meta;
9  |  end
10 |  Criar o conjunto Z de nós contendo inicialmente o nó ni;
11 |  while Z não esteja vazio do
12 |  |  Remover de Z um nó nm tal que não tenha descendentes em G' que pertençam a Z;
13 |  |  foreach k-hiperarco tendo nó de entrada nm e nós de saída {nm1,nm2,...,nmk} do
14 |  |  |  Calcular qi(nm) = c(nm,a) + q(nm1)+q(nm2)+...+q(nmk);
15 |  |  end
16 |  |  Estabelecer q(nm) com o mínimo dos valores de qi(nm) e marcar nm com o
   |  |  k-hiperarco em que se encontrou o valor mínimo, removendo a etiqueta anterior;
17 |  |  if todos os nós sucessores através desse k-hiperarco estiverem etiquetados como
   |  |  "Resolvido" then
18 |  |  |  Etiquetar nm como "Resolvido";
19 |  |  end
20 |  |  if nm foi etiquetado como "Resolvido" ou o custo atual de nm for diferente do
   |  |  anterior then
21 |  |  |  Adicionar a Z todos aqueles ancestrais de nm tal que nm é um nó sucessor através
   |  |  |  de um k-hiperarco etiquetado;
22 |  |  end
23 |  end
24 end
```

2.3.3 Busca em grafos cíclicos AND/OR para ações não-determinísticas

Os problemas de busca com ações não-determinísticas (sem probabilidades nos efeitos das ações), muitas vezes, envolvem lidar com ciclos. No caso de grafos acíclicos, AO* executa a atualização de custos eficientemente, nunca atualizando mais de uma vez um nó. No entanto, quando existirem ciclos no hipergrafo explícito *G'*, o algoritmo pode desdobrar o hipergrafo G, gerando nós que não possuem um sub-hipergrafo solução a partir deles. Assim, AO* pode entrar num loop infinito, ou pode atualizar muitas vezes um nó numa única iteração.

A possibilidade de fazer busca num grafo AND/OR cíclico foi estudada por Chakrabarti, (1994). Esse trabalho apresenta dois novos algoritmos que encontram a melhor solução a partir do nó inicial num grafo AND/OR explícito.

- *Iterative-revise* é um algoritmo iterativo que constrói um conjunto de nós cujo custo do subgrafo solução foram determinados. O processo continua até o nó inicial obter seu valor ótimo. O algoritmo guarda os estados visitados na iteração atual e se o custo do subgrafo solução foi determinado previamente. O algoritmo também guarda o limite superior do custo de um nó. Armazenando essas informações, o algoritmo tenta determinar se um nó já foi visitado e se seu valor ótimo computado (igual ao limite superior e menor de um ϵ), desse modo evitando os problemas do AO* anteriormente mencionados.

- *REV** é um algoritmo que segue um esquema *bottom-up* que começa colocando todos os nós folha do grafo explícito numa lista *Abertos* e atribuindo-lhes um valor heurístico. Depois, iterativamente seleciona o nó com menor custo estimado de *Abertos*, atribui valores de custo para seus nós pais e percorre o grafo de maneira bottom-up, completando um nível antes de continuar com o próximo nível. Quando o algoritmo ficar preso na fase ascendente, ele seleciona um nó de *Abertos* e inicia outra fase ascendente de avaliação de nós. Desse modo, REV* continua até que o nó inicial seja avaliado.

A eficiência do algoritmo REV* foi melhorado por Jimenez e Torras (2000), introduzindo algumas modificações derivadas do algoritmo CF (Mahanti e Bagchi, 1985). O algoritmo INT, usa uma estratégia top-down baseada no CF, no entanto, o processo de revisão de custos bottom-up foi inspirado no REV*. O processo de revisão de custos do INT utiliza uma estratégia de atualização de custos que torna possível a busca num grafo AND/OR cíclico. O algoritmo CFC_{REV*} mantém na lista *Abertos*, um subconjunto dos nós folha. Com essa lista, o algoritmo decide que nós devem ser visitados na fase de atualização de custos.

Neste trabalho, estamos interessados em resolver problemas de busca em grafos cíclicos AND/OR, porém, aqueles em que os nós AND estão associados a uma distribuição de probabilidades de um MDP. Algoritmos propostos para resolver esse tipo de problemas são apresentados no próximo capítulo, ou seja, algoritmos que fazem busca em grafos AND/OR ou hipergrafos cíclicos.

Capítulo 3

Fundamentos de Processos de Decisão Markovianos

Os Processos de Decisão Markovianos (*Markov Decision Processes* - MDPs) (Puterman, 1994) são modelos usados para planejamento probabilístico, em que: as ações selecionadas por um agente causam mudanças no comportamento do sistema; o estado atual do sistema e a ação escolhida para ser executada geram uma recompensa (ou custo) e determinam uma distribuição de probabilidades sobre os possíveis próximos estados do sistema.

Um MDP representa a interação do agente com seu ambiente. O modelo define *estágios* nos quais o agente observa o estado atual do sistema e decide executar uma ação. O agente faz suas decisões locais com o objetivo de otimizar o valor esperado de uma função utilidade ao longo de um horizonte H de t estágios de decisões (com t=[1..∞[), isto é, a decisão t é feita no começo do estágio t, correspondendo ao intervalo de tempo entre a decisão t e a decisão t+1 (não incluindo esse ponto do tempo). Geralmente, o objetivo do agente envolve a maximização da recompensa esperada ou a minimização do custo esperado ao longo do processo (Puterman, 1994).

Exemplo 3.1 Robô explorador espacial (Leffler, 2009)

Nesse problema, o robô explorador deve decidir que ações executar e em que momentos se locomover entre os diversos lugares do ambiente, fazendo o melhor uso dos recursos disponíveis. Para navegar no ambiente desconhecido do planeta, o robô precisa saber como sua posição mudará após executar uma ação. Assim, essa dinâmica deve ser modelada e conhecida pelo robô para alcançar seus objetivos.

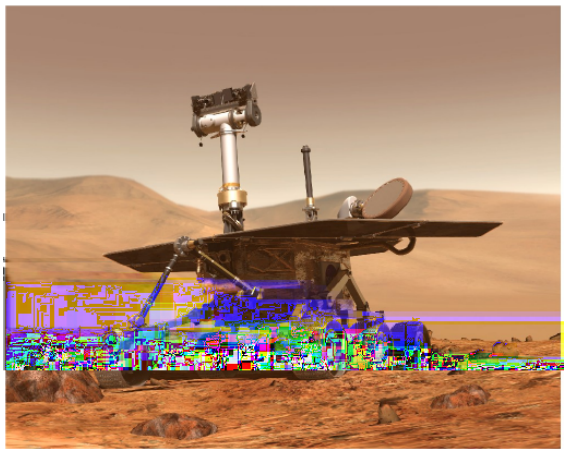

Figura 3.1: *O robô explorador. Fonte: Cortesia da NASA/JPL-Caltech.*

Para esse problema, podemos identificar os seguintes elementos:

- Cada cenário ou condição encontrada pelo robô no ambiente, representa um *estado* do sistema. Por exemplo, condições do terreno, clima e as condições do próprio robô (posição, reserva de energia, temperatura, etc.) definem um estado.
- As *transições* entre estados acontecem como resultado de uma sequência de *ações* que o robô executa nos *estágios de decisão*. O robô pode executar a ação "Coletar amostra" ou "Enviar leitura" em intervalos de tempo previamente definidos.
- Em qualquer estágio, o estado do sistema é *completamente conhecido* pelo robô.
- Os resultados das ações aplicadas pelo agente são probabilísticos. Dessa forma, quando uma ação for executada num estado o sistema pode, com diferentes *probabilidades de transição*, ser levado para vários estados. Sob este modelo, assumimos que essas probabilidades são conhecidas pelo agente.
- Quando o robô executar uma ação no ambiente recebe uma *recompensa*. No entanto, se a ação exigir a redução de algum recurso, a recompensa é negativa.
- O objetivo do robô é otimizar alguma *medida de utilidade*. Por exemplo, o robô pode maximizar a recompensa total obtida pela tomada de decisões ou tentar minimizar o consumo de algum recurso disponível (energia, tempo, distância, etc).

Os MDPs tem sido muito estudados em planejamento probabilístico e no aprendizado supervisionado. Na comunidade de planejamento, assume-se um conhecimento completo a priori do modelo MDP. Assim, foca seu esforço no desenvolvimento de métodos computacionalmente eficientes para calcular uma política ótima usando o modelo conhecido. O aprendizado supervisionado (Barto *et al.*, 1995) estuda o problema que enfrenta o agente quando não tem acesso ao modelo completo e tem que aprender baseado em experiências, isto é, interação com o ambiente. Uma definição formal do modelo MDP é descrita a seguir.

3.1 Planejamento probabilístico e MDPs

Vários problemas de planejamento probabilístico podem ser modelados como um MDP, consequentemente podemos representá-los e resolvê-los como um problema de otimização. O domínio de planejamento é descrito como um sistema estocástico, isto é, um sistema de transição de estados não-determinístico que atribui probabilidades às transições entre estados. Assim, a incerteza dos resultados de uma ação é modelada usando uma função de distribuição de probabilidades. Os objetivos são representados usando funções de utilidade, por exemplo, a função recompensa e a função custo. Essas funções devolvem valores numéricos associados com a escolha de uma ação ou ao alcançar um estado. As funções de utilidade expressam preferências ao longo do processo de decisão (Ghallab *et al.*, 2004). Assim, o problema de planejamento utiliza critérios de otimização na busca da política, dessa maneira, está orientada a minimizar ou maximizar a função de utilidade.

Os planos são representados como políticas, funções que prescrevem uma ação a cada estado. Formalmente, uma política é uma função π: $S \rightarrow A$, que define uma ação para cada estado. Denotamos $\pi(s)$ a ação prescrita pela política π quando estamos no estado s. Podemos classificar as políticas em um MDP definindo quanto tempo o agente vai segui-la. O horizonte de um MDP é o número de estágios que são necessários para chegar ao final do processo. Se usarmos um *horizonte finito*, o agente está limitado a agir só durante t estágios. No entanto, em um MDP de *horizonte infinito*, o número de estágios não está restrito e pode prolongar-se indefinidamente, porque o processo não está limitado no tempo. Uma outra possibilidade é considerar que o processo possui um horizonte finito porém desconhecido, isto é, o processo alcançará um estado considerado meta em

algum estágio (Delgado, 2010).

Quando uma política for *estacionária* a ação que será executada depende do estado atual e não depende dos estágios de decisão restantes. Numa política *não estacionária*, a ação selecionada no primeiro estágio depende dos t estágios seguintes, a ação considerada no segundo estágio depende dos seguintes *t-1* estágios, até o horizonte acabar. Em (Puterman, 1994), afirma-se que todo MDP de horizonte infinito possui uma política estacionária como solução. A qualidade de uma política pode ser avaliada usando o conceito de valor acumulado esperado. No caso de horizonte infinito, a avaliação é feita sobre o valor acumulado esperado e descontado.

3.2 Definição de MDP

Formalmente, um Processo de Decisão Markoviano é definido pela tupla $\mathcal{M}=\langle S,A,R,P,\gamma\rangle$ sendo:

- S é um conjunto finito e discreto de estados e representa todos os possíveis estados em que o ambiente pode se encontrar. O conjunto também é chamado de *espaço de estados*. Assumimos que cada estado captura a *propriedade de Markov*, a saber, o próximo estado do ambiente depende somente do estado atual e da ação selecionada.
- A é um conjunto das ações disponíveis para o agente, isto é, o conjunto de todas as ações que o agente pode executar.
- R: $S \times A \to \mathbb{R}$ é a função recompensa que define o valor da recompensa numérica obtida pelo agente quando escolher uma ação em um determinado estado.
- P: $S \times A \times S \to [0,1]$ é uma função de transição de estados que representa a probabilidade condicional de transição, isto é, define a probabilidade de chegar ao estado $s' \in S$ quando o agente está no estado $s \in S$ e executa a ação $a \in A$.
- γ define o fator de desconto, $0 \leq \gamma < 1$. Quando estivermos usando um horizonte infinito, determina o valor relativo das recompensas imediatas com relação às posteriores. As recompensas recebidas em i estágios no futuro são descontadas em um fator γ^i (Boutilier *et al.*, 1999).

3.2.1 MDP de horizonte infinito de recompensa descontada

Seja V_π:$S \to \mathbb{R}$ uma função utilidade, chamada de função valor do estado, ou simplesmente de função valor, que associa cada estado a um valor numérico representando a esperança ou expectativa de um estado $s \in S$, quando as ações prescritas pela política π são executadas a partir do estado s. Em problemas de horizonte infinito, o valor da política π iniciando no estado s, $V_\pi(s)$, é definido como a soma esperada das recompensas descontadas. Formalmente:

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right], \tag{3.1}$$

em que r_t é a recompensa imediata obtida no estágio t.

Definimos a função de avaliação $Q_\pi(s,a)$ tal que seu valor é a recompensa descontada esperada que pode ser obtida no estado s, aplicando a ação a e seguindo a política π nos estados futuros. Isto é, o valor de Q_π é o valor da recompensa imediata recebida ao executar a ação a no estado s mais o valor esperado (descontado por γ) de seguir uma política π começando desse ponto.

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_\pi(s'). \tag{3.2}$$

Uma política gulosa é aquela que escolhe a ação, em cada estado, que maximiza o valor esperado desse estado com relação a alguma função valor arbitrária. Se π for a política e V for a função valor associada a π, temos:

$$\pi_V(s) = \arg\max(Q_\pi(s,a)). \tag{3.3}$$

Definimos a política ótima π^* de um MDP como aquela que produz a maior utilidade esperada, isto é, aquela que produz o máximo valor esperado para cada estado tal que, $\forall s$, $\forall \pi^{'}$, $V_{\pi^*}(s) \geq V_{\pi'}(s)$. Seja V^* a função valor associada a uma política ótima, então:

$$V^*(s) = \max_{a \in A}\{R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V^*(s')\}. \tag{3.4}$$

A Equação 3.4 é chamada de *Princípio de Otimalidade de Bellman* para MDP (Bellman, 1957). Desta maneira, o problema de encontrar uma política ótima para um MDP é reduzido ao problema de resolver a equação de otimalidade.

3.2.2 MDP de horizonte finito

Neste modelo, a dinâmica do agente está limitada por um número finito de estágios. Consideramos MDPs com $\{0,..,T\}$ decisões e *T-1* estágios. Neste caso, o somatório da Equação 3.1 considera um número finito de estágios $T<\infty$, chamado de *horizonte*. Quando um MDP de horizonte finito é usado na modelagem de um problema, o fator de desconto γ é definido como 1. Desse modo, a Equação 3.1 é redefinida como:

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{T-1} r_t | s_0 = s \right], \forall t < T \quad e \quad V_\pi(s) = 0, t = T. \tag{3.5}$$

A função valor ótima para este modelo de MDP satisfaz a equação:

$$V^*(s,t) = \max_{a \in A}\{R(s,a) + \sum_{s' \in S} P(s'|s,a)V^*(s', t+1)\}. \tag{3.6}$$

A política ótima π^* associada à função valor ótima é markoviana (só depende do estado atual do sistema) e determinística, satisfazendo a seguinte equação:

$$\pi^*(s,t) = \arg\max_{a \in A}\{R(s,a) + \sum_{s' \in S} P(s'|s,a)V^*(s', t+1) \forall s \in S \quad e \quad t < T\}. \tag{3.7}$$

3.2.3 MDP de horizonte indefinido

Os dois tipos de MDPs apresentados anteriormente, podem ser considerados casos limites em relação ao número de estágios que o agente pode seguir. No entanto, podemos considerar um tipo de MDP em que o horizonte é finito mas desconhecido, ou seja, o agente deve parar quando atingir um estado meta. Enquanto não alcançar um estado meta, o agente recebe uma recompensa negativa, isto é, um custo. Desse modo, o custo total esperado de qualquer política ótima deve ser finito, usando fator de desconto ou não. Definir as condições sob as quais uma política constitui um política ótima para um MDP de horizonte indefinido resulta numa outra classe de MDP conhecido como *problema do caminho mínimo estocástico* (Seção 3.2.4).

Definição 2.2 (Política própria). Dado um MDP com um conjunto de estados meta $\mathcal{G} \subseteq S$ e $\forall s_g \in \mathcal{G}$ e $\forall a \in A$, $P(s_g|s_g,a)=1$ e $R(s_g,a)=0$. Seja π uma política, se a probabilidade do agente chegar num estado $s_g \in \mathcal{G}$ seguindo a política π e selecionando qualquer caminho que começa num estado $s \in S$ é igual a 1, então a política π é chamada de *política própria*. Caso contrário, é chamada de *política imprópria*.

Desse modo, para resolver um MDP de horizonte indefinido precisamos definir as condições para que alguma de suas políticas ótimas seja própria. Se todas as recompensas obtidas pelo agente forem negativas (custos), o agente incorrerá em um custo maior quando estiver mais longe do estado meta. Se o agente seguir uma política imprópria não alcançará o estado meta e o custo total esperado para alguns estados será infinito. Por outro lado, se a política seguida pelo agente for própria, então o custo será finito para todos os estados.

3.2.4 Problema do caminho mínimo estocástico

O problema do caminho mínimo estocástico (*Stochastic Shortest Path Problem* - SSP) (Bertsekas, 1995) pode ser considerado uma generalização do problema do caminho mínimo determinístico, no qual incluímos probabilidades nas transições de estados e funções de utilidade. No planejamento probabilístico, o SSP pode ser usado para formular e definir MDPs de horizonte indefinido. A definição formal do problema do caminho mínimo estocástico envolve:

- S, um conjunto finito dos possíveis estados do sistema, completamente observáveis.
- A, um conjunto finito das ações disponíveis para o agente.
- P: $S \times A \times S \rightarrow [0,1]$; uma função de transição probabilística que representa a probabilidade do sistema ir para o estado $s' \in S$, depois do agente executar a ação $a \in A$ no estado $s \in S$.
- C: $S \times A \rightarrow \mathbb{R}$; a função que associa um custo a cada estado-ação. Representa o custo de executar uma ação em um estado determinado.
- $s_0 \in S$; o estado inicial (ou um conjunto S_0 de estados iniciais).
- $\mathcal{G} \subseteq S$; um conjunto de estados meta, em que $\forall\ s_g \in \mathcal{G}$, $P(s_g,a,s_g) = 1.0$ e $C(s_g,a)=0\ \forall a \in A$. Esses estados s_g são chamados de *absorventes*.
- $\gamma = 1.0$, em que γ é o fator de desconto.

As diferenças com relação à definição do MDP da seção 3.2, estão no uso de uma função de custo ao invés de recompensa e o valor do fator de desconto. Desse modo, no SSP estamos interessados em minimizar o custo esperado (poderíamos definir também R=-C e voltar à formulação de maximização da recompensa esperada). A função valor ótima para SSP é markoviana e satisfaz:

$$V^*(s) = \min_{a \in A}\{C(s,a) + \sum_{s' \in S} P(s'|s,a)V^*(s')\}, \forall s \in S. \tag{3.8}$$

Assim, $V^*(s)$ representa o menor custo esperado que podemos obter a partir do estado s até um estado meta. Toda política ótima no SSP é própria e minimiza o custo esperado de chegar num estado meta a partir de cada estado $s \in S$. Há duas condições sobre a política ótima que permitem definir completamente a solução do SSP:

- Existe pelo menos uma política própria.
- Para cada política imprópria π e $\forall s \in S$, $V^*(s)=\infty$.

Assim, a solução do problema do caminho mínimo estocástico é uma política própria que alcance algum estado meta, minimizando o custo total esperado, a partir do estado s_0. Essa política ótima não necessariamente é única. Se assumirmos que o estado inicial do sistema s_0 é conhecido, será suficiente calcular uma *política parcial* relativa ao estado s_0. Uma política parcial π só prescreve ações $\pi(s)$ a serem selecionadas sobre o subconjunto $S_\pi \subseteq S$ de estados. Os estados alcançáveis pela política ótima são chamados de *estados relevantes* (Bonet e Geffner, 2003).

As três classes de MDPs revisadas parecem definir três modelos distintos entre si, no entanto, em (Mausam e Kolobov, 2012) mostra-se que essas classes são relacionadas e que o SSP é um modelo mais geral que as outras classes, sendo que MDPs de horizonte infinito de recompensa descontada (HINF) e MDPs de horizonte finito (HFIN) podem ser definidos como casos particulares do SSP. Um SSP com o conjunto de estados iniciais $S_0{=}S$, e estados meta são transformados em estados absorventes pode ser definido como um HINF (Seção 3.2.1). Dessa maneira, as soluções que apresentamos a seguir estão orientadas a resolver o SSP.

3.3 Métodos para resolver um MDP usando Programação Dinâmica Síncrona

Quando resolvemos um MDP precisamos encontrar uma política ótima ou ϵ-ótima que maximiza a recompensa esperada ou minimiza o custo esperado dos estados $s{\in}S$. Existem várias abordagens para resolver MDPs, as mais conhecidas são: 1) Aquelas baseadas em programação dinâmica síncrona, em que os valores de todos os estados são atualizados em cada iteração; 2) aquelas baseadas em programação dinâmica assíncrona, em que somente alguns estados são atualizados em cada iteração e 3) aquelas baseadas na busca em grafos AND/OR (ou hipergrafos). Nos últimos dois casos, resolvemos um MDP em que somente um estado inicial é conhecido.

Nesta seção exploraremos o primeiro grupo. As soluções baseadas em programação dinâmica assíncrona serão revisadas na Seção 3.4 e as soluções baseadas em busca em grafos AND/OR (hipergrafos) serão estudadas na Seção 3.6. Os algoritmos clássicos para resolver MDPs usam técnicas de programação dinâmica baseada no princípio de otimalidade de Bellman. Existem dois algoritmos principais que usam esta abordagem: *Iteração de Valor* e *Iteração de Política*. Ambos são algoritmos ótimos que fornecem políticas completas, isto é, calculam a solução para todo o espaço de estados.

3.3.1 O algoritmo Iteração de Valor

O algoritmo Iteração de Valor (IV) calcula valores para os estados fazendo atualizações sucessivas da função valor V_t (*Bellman backups*). Em cada iteração, a função V_{t+1} é calculada usando o valor estimado da iteração anterior, ou seja, usando a estimativa da função V_t. Começando numa V_0 arbitrária, as atualizações são realizadas sobre todos os estados de S. No estágio t, o algoritmo encontra uma ação a que minimiza $V_t(s)$ e a seleciona como parte da política. A Equação 3.4 é aplicada como um algoritmo de programação dinâmica, da seguinte forma:

$$V_{t+1}(s) = \min_{a \in A(s)} \{C(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_t(s')\}. \tag{3.9}$$

O Algoritmo 6 descreve esse procedimento de programação dinâmica. A sequência de funções V_t converge para a função ótima V^* depois de um número finito de iterações (Puterman, 1994). Geralmente, o algoritmo precisa um valor ϵ para indicar o erro máximo permitido entre duas iterações consecutivas para todos os estados de S e na prática este valor serve como parte do critério de parada do algoritmo.

$$\max_{s \in S} |V_{t+1}(s) - V_t(s)| < \epsilon(1-\gamma)/\gamma. \tag{3.10}$$

Como o algoritmo precisa atualizar o espaço de estados completo em cada iteração, sua complexidade computacional é dada por $\mathcal{O}(|S|{\times}|A|{\times}|S|)$.

Algoritmo 6: ITERAÇÃO-DE-VALOR(M, ϵ). Fonte: (Ghallab *et al.*, 2004).

Input:
M: MDP.
ϵ: erro.
Output: Devolve uma política ϵ-ótima do MDP

foreach $s \in S$ **do**
  $V_0(s) = C(s,a)$
end
$n = 0$;
repeat
  $n = n + 1$;
  foreach $s \in S$ **do**
    foreach $a \in A$ **do**
      $Q_n(s,a) = C(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s')$
    end
    $V_n(s) = \min_{a \in A} Q_n(s,a)$;
    $\pi_n(s) = \arg\min_{a \in A} Q_n(s,a)$;
  end
until $|V_n(s) - V_{n-1}(s)| < \epsilon(1-\gamma)/\gamma$; $\forall s \in S$;
return π_n;

3.3.2 O algoritmo Iteração de Política

O algoritmo Iteração de Política melhora a estimativa do valor esperado da política em cada iteração. Para isso, define-se uma política inicial arbitrária π_0 que é melhorada ao longo do processo. Cada nova política π_{t+1} está baseada na política π_t encontrada na iteração anterior. O processo iterativo pode ser visto como a repetição de dois passos:

- *Obtenção do valor.* A política atual π_t é avaliada, para cada estado $s \in S$; o valor V_{π_t} é computado usando essa política.
- *Refinamento da política.* A política atual é melhorada, geramos uma política gulosa π_{t+1} baseada em π_t tal que para cada estado s escolhemos uma ação a que minimize $Q_{\pi_t}(s,a)$, assim

$$\pi_{t+1}(s) = \arg \min_{a \in A(s)} \{C(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_{\pi_t}(s')\}. \tag{3.11}$$

O critério de parada do algoritmo é definido pela igualdade entre as políticas em duas iterações consecutivas, isto é, $\forall s \in S$, $\pi_{t+1}(s) = \pi_t(s)$. O processo completo descrito anteriormente é descrito pelo Algoritmo 7. O passo da obtenção de valor pode ser resolvido em tempo $\mathcal{O}(|S| \times |S| \times |A|)$, enquanto que o refinamento da política é feito em tempo $\mathcal{O}(|S| \times |S| \times |A|)$ (Littman *et al.*, 1995).

Algoritmo 7: ITERAÇÃO-DE-POLITICA(M). Fonte: (Mausam e Kolobov, 2012).

Input: M: MDP
Output: Devolve uma política ótima do MDP

Inicializar π_0 com uma política arbitrária;
$n = 0$;
repeat
  $n = n + 1$;
  Avaliação da política: Computar $V_{\pi_{n-1}}$;
  Melhoramento da política;
  foreach $s \in S$ **do**
    $\pi_n(s) = \pi_{n-1}(s)$;
    foreach $a \in A$ **do**
      $Q_{\pi_{n-1}}(s,a) = C(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_{\pi_{n-1}}(s')$
    end
    $V_n(s) = \min_{a \in A} Q_{\pi_{n-1}}(s,a)$;
    if $Q_{\pi_{n-1}}(s, \pi_{n-1}(s)) > V_n(s)$ **then**
      $\pi_n(s) = \arg\min_{a \in A} Q_{\pi_{n-1}}(s,a)$;
    end
  end
until $\pi_n == \pi_{n-1}$;
return π_n;

Os métodos mencionados anteriormente produzem políticas que minimizam o custo esperado de todo estado $s \in S$. No entanto, os algoritmos IV e IP não usam o conhecimento sobre o estado inicial e os estados meta.

3.4 Programação Dinâmica Assíncrona: O algoritmo RTDP

Um SSP com estado inicial s_0 e estados meta, pode ser resolvido de maneira mais eficiente. Barto propôs o algoritmo *Real-Time Dynamic Programming* - RTDP (Barto *et al.*, 1995) para evitar a avaliação exaustiva do espaço de estados. O algoritmo usa a informação sobre o estado inicial s_0 e focaliza a visita bem como a atualização dos estados que são alcançáveis a partir de s_0. O RTDP generaliza o algoritmo de busca heurística desenvolvido por Korf em (Korf, 1990) chamado *Learning Real-Time A** - LRTA*, permitindo que as transições entre estados sejam estocásticas.

O algoritmo RTDP realiza a simulação de uma política gulosa amostrando caminhos ou trajetórias no espaço de estados e fazendo atualizações de Bellman somente nos estados desses caminhos. Cada processo de amostragem é chamado de *sessão de simulação* ou *trial*. Cada trial seleciona repetidamente a ação gulosa a no estado atual s, executa a atualização de s e faz uma transição a algum sucessor de s alcançável através de a.

Dessa maneira, o algoritmo RTDP tenta encontrar uma política ótima somente para os estados relevantes do problema. Segundo (Bonet e Geffner, 2003), o algoritmo tem duas vantagens principais: não precisa avaliar o espaço de estados completo para encontrar uma política ótima e, sob certas condições, pode descobrir políticas parciais ótimas rapidamente. Essa última condição depende em grande parte da heurística admissível definida para o problema e da profundidade das sessões de simulação.

O Algoritmo RTDP (Algoritmo 8) inicializa V_l^t (valor estimado de V^*) a um valor de limite inferior admissível, isto é, $V_l^t(s) \leq V^*(s)\ \forall s \in S$; em seguida executa uma sequência de trials, em que

cada trial começa a partir do estado inicial s_0. O valor $V_l^t(s)$ dos estados encontrados ao longo da simulação são atualizados e uma ação gulosa é escolhida para cada estado visitado. Além disso, o algoritmo sorteia um estado, baseado na distribuição de probabilidades da função de transição de estados $P(s'|s,a)$, para determinar qual será o próximo estado a ser visitado. O trial finaliza quando o agente alcança o estado meta ou quando a profundidade limite é alcançada. O algoritmo mantém uma pilha *estadosVisitados* para guardar os estados visitados numa sessão e facilitar a atualização dos seus valores.

As propriedades do RTDP determinam que se os valores de todos os estados são inicializados com valores fornecidos por uma função heurística admissível (limite inferior admissível), então os valores dos estados atualizados também serão admissíveis. Por outro lado, se existir uma política própria, o RTDP não pode ficar preso num ciclo e deve acabar em algum estado meta depois de um número finito de passos. Além disso, para os estados relevantes, o algoritmo converge assintoticamente com valores ótimos e fornece uma política ótima. A velocidade de convergência está relacionada com a qualidade da heurística e ao tamanho do espaço de estados. Em geral, o algoritmo RTDP pode ter uma taxa de convergência lenta porque faz uma exploração estocástica do espaço de estados, seguindo os caminhos mais prováveis e portanto, desconsiderando outros.

Algoritmo 8: RTDP(s_0, $\mathcal{G}$, maxProf, V_{l_0}). Fonte: (Delgado, 2010).

Input:
s_0: Estado inicial.
$\mathcal{G}$: Estados meta.
maxProf: Profundidade máxima.
V_{l_0}: Limite inferior admissível.
Output: Devolve os valores estimados de V^*

```
1  t = 0;
2  V_l^t = V_{l_0};
3  s = s_0;
4  while convergência não alcançada e tempo limite não alcançado do
5  |  profundidade = 0;
6  |  estadosVisitados.LIMPAR();
7  |  while s∉G e s≠NULL e (profundidade<maxProf) do
8  |  |  profundidade = profundidade + 1;
9  |  |  t = t + 1;
10 |  |  visitados.EMPILHAR(s);
11 |  |  V_l^t(s) = min_{a∈A}{C(s,a) + γ Σ_{s'∈S} P(s'|s,a) V_l^{t-1}(s')};
12 |  |  a = argmin_{a∈A}{C(s,a) + γ Σ_{s'∈S} P(s'|s,a) V_l^{t-1}(s')};
13 |  |  s = s' ~ P(.|s,a);
14 |  end
15 |  while ¬estadosVisitados.PILHA-VAZIA() do
16 |  |  s = estadosVisitados.DESEMPILHAR();
17 |  |  V_l^t(s) = min_{a∈A}{C(s,a) + γ Σ_{s'∈S} P(s'|s,a) V_l^{t-1}(s')};
18 |  end
19 end
20 return V_l^t;
```

O algoritmo RTDP foi estendido e/ou melhorado por vários trabalhos gerando uma família de algoritmos relacionados. A seguir, explicamos rapidamente esses trabalhos:

- O algoritmo LRTDP (Labeled Real-Time Dynamic Programming) foi proposto por (Bonet e Geffner, 2003). Um esquema de deteção de convergência foi introduzido no algoritmo RTDP para melhorar seu desempenho. A ideia é rotular um estado s como convergido quando

o resíduo dele e de todos seus descendentes não excede um ϵ dado. O método CHECK-SOLVED realiza essa verificação. Em cada iteração, o método seleciona um estado s e verifica se o resíduo atual é maior que ϵ. Se acontecer isso, o estado s não pode ser rotulado ainda, no contrário, o estado é expandido para verificar seus sucessores. Se o resíduo de todos os descendentes e do estado s for menor que ϵ, todos são rotulados como resolvidos. No caso contrário, os estados são atualizados usando a atualização de Bellman.

O processo de rotulado começa do último estado visitado nas sessões. Dessa maneira, a convergência dos estados perto do estado meta é reconhecida antes da convergência dos estados perto do estado inicial. Portanto, os estados que já convergiram são rotulados para evitar sua futura visita. Isto reduz o tempo de processamento e permite que outros estados sejam visitados.

- O algoritmo FRTDP (Focused Real-Time Dynamic Programming) (Smith e Simmons, 2006) é uma modificação do RTDP que mantém dois limites sobre a função valor contruindo a política baseado no limite inferior. FRTDP guia o processo de busca focalizando-se nas partes do espaço de busca que contribuem com os melhores valores para as políticas. Define um *valor de prioridade* que estima o benefício de dirigir a busca para algum nó em particular. Desta maneira, a seleção baseada em prioridade processa as partes mais relevantes do espaço e permite evitar nós que já convergiram.

- O algoritmo BRTDP (Bounded Real-Time Dynamic Programming) (McMahan *et al.*, 2005) é um outro algoritmo que fornece boas políticas parciais avaliando somente parte do espaço de estados. Bounded RTDP mantém dois limites sobre a função de avaliação ótima. $v_u(s)$ e $v_l(s)$ denotam o limite superior e inferior respectivamente. BRTDP utiliza a diferença $v_u(y)$-$v_l(y)$ como medida de incerteza do valor para s, o algoritmo utiliza esse valor para estabelecer uma prioridade na escolha dos estados. Aqueles que tiverem maior diferença, terão mais oportunidade de ser visitados.

- O algoritmo LR^{2}TDP (Kolobov *et al.*, 2012) é baseado no LRTDP e foi usado pelo planejador GLUTTON, segundo lugar da IPPC-2011. Dado um MDP(H) com horizonte H podemos construir uma política usando MDPs de horizonte menor. Resolvendo a sequência de MDP(*1*), MDP(*2*),... o planejador obterá uma boa política para o MDP(H). A estratégia descrita chama-se *Reverse Iterative Deepening*. O algoritmo LR^{2}TDP reusa os valores computados na sequência MDP(*1*),MDP(*2*),...,MDP(*H-1*) para resolver o MDP(H).

3.5 UCT

UCT (Kocsis e Szepevári, 2006) é um algoritmo de planejamento baseado em técnicas de amostragem de Monte-Carlo projetado para resolver MDPs. Ao contrário que outras técnicas, UCT precisa fazer amostragem da função de transição e da função recompensa de um simulador não precisando conhecer as probabilidades de transição ou valores da recompensa das ações. De modo similar ao RTDP, UCT realiza um número de *rollouts* através do espaço de estados e atualizações da função valor nos estados que visita nessas trajetórias.

A escolha dos estados sucessores é estocástica, porém a escolha das ações é gulosa como no RTDP mas adicionando um termo que garante que as ações aplicáveis são testadas em todos os estados. Para cada estado, UCT mantem um contador do número de vezes que um estado foi visitado e um outro contador que mantem quantas vezes uma ação foi selecionada num estado dado. O algoritmo mantem um grafo parcial explícito que é expandido incrementalmente. Os rollouts começam da raiz e acabam num estado terminal ou em um estado que não pertence ao grafo. Quando um novo estado for gerado, este é adicionado no grafo explícito, a recompensa é amostrada simulando uma política base para d passos a partir do estado inicial. Os valores são propagados usando atualizações de

Monte-Carlo (Sutton e Barto, 1998).

O algoritmo UCT foi usado com sucesso na IPPC-2011. A maioria dos sistemas de planejamento da competição foram baseados no UCT. O planejador PROST (Keller e Eyerich, 2012), primeiro lugar da competição, usa o algoritmo UCT e introduz algumas otimizações para melhorar seu desempenho, entre elas, técnicas para reduzir o fator de ramificação e para limitar a profundidade da busca.

3.6 Busca heurística e MDPs

Na Seção 3.2.1 vimos que a solução de um MDP é uma política que indica a ação que devemos aplicar quando estivermos em um estado s qualquer. A política encontrada pelos métodos de programação dinâmica tem uma limitação prática, a quantidade de memória que precisam é proporcional ao tamanho do espaço de estados, uma vez que computam uma política completa (isto é, definem a ação ótima para cada estado de S). No entanto, dada a informação do estado inicial s_0, podemos orientar a solução do MDP para encontrar uma política parcial relativa a esse estado.

Definição 2.3 (Política parcial relativa a um estado s). Uma política parcial $\pi_s : S' \rightarrow A$, $S' \subseteq S$, é relativa ao estado s se qualquer estado s' alcançável usando π_s a partir de s pertence a S'.

Em outras palavras, π_s especificará uma ação para qualquer estado s' que possa ser alcançado a partir de s e usando essa política. Quando a política é parcial com relação ao estado inicial s_0 excluímos grandes partes do espaço de estados, precisando menos memória e recursos para computá-la. Além disso, a política ótima será, geralmente, ainda menor porque S' não inclui alguns estados que poderiam ser alcançados a partir de s_0. Desse modo, computar políticas parciais ótimas pode ser reduzido a resolver o SSP.

3.6.1 Modelando um MDP como um hipergrafo

Um SSP (Seção 3.2.4) possui um hipergrafo correspondente que representa a estrutura de conectividade do espaço de estados. Intuitivamente, os estados do SSP são representados pelos nós do hipergrafo e a função de transição de estados do SSP é representada pelos hiperarcos do hipergrafo. Ou seja, a aplicação de uma ação a em um estado s é representada por um k-hiperarco conectando um nó (associado ao estado s) com seus k nós sucessores.

Por exemplo, para o MDP definido pelo conjunto de estados $S=\{s_1,s_2,s_3\}$, conjunto de ações $A=\{a_1\}$, probabilidades de transição $P(s_1|s_1,a_1)=0.5$, $P(s_2|s_1,a_1)=0.5$, $P(s_1|s_2,a_1)=0.3$, $P(s_3|s_2,a_1)=0.7$ e $P(s_2|s_3,a_1)=1.0$, a Figura 3.2 ilustra sua representação através de um hipergrafo. Os arcos individuais de cada k-hiperarco são etiquetados com os valores da distribuição de probabilidades da ação.

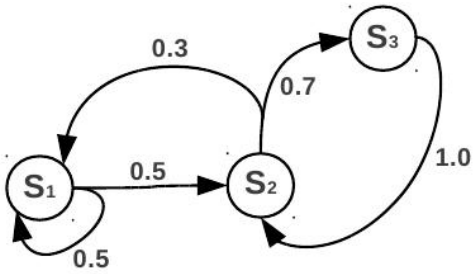

Figura 3.2: *Representação de um MDP usando um hipergrafo. Os círculos representam os nós do hipergrafo associados aos estados do MDP. Os hiperarcos representam as transições probabilísticas das ações do MDP. O valores etiquetados nos arcos indicam a distribuição de probabilidades da ação.*

O *hipergrafo de conectividade com raiz no estado s*, é um sub-hipergrafo do hipergrafo de conectividade do MDP que considera os nós *s'* alcançáveis a partir de *s*. Em particular, se a raiz for o estado inicial s_0, os estados incluídos no hipergrafo serão aqueles alcançáveis mediante sequências de ações que começam em s_0.

Em geral, a política de um MDP envolve a existência de ciclos. Por exemplo, depois de executar uma ação, pode existir uma probabilidade diferente de zero de manter-se no mesmo estado ou de ir para um estado já visitado. Dado que a solução de um MDP inclui ciclos, não podemos aplicar o algoritmo AO* apresentado na Seção 2.3.2 porque a fase de atualização de custos assume uma solução acíclica. É importante lembrar que nessa fase, o algoritmo AO* utiliza programação dinâmica. Os algoritmos que apresentaremos a seguir, generalizam essa fase para permitir a geração de soluções cíclicas para SSPs.

3.6.2 O algoritmo LAO*

O algoritmo LAO* (*Loop* AO*) (Hansen e Zilberstein, 2001) é uma extensão do algoritmo AO* que permite encontrar soluções cíclicas e que pode ser usado para resolver um SSP. LAO* é um algoritmo *offline*[1] de busca heurística que resolve o SSP. No algoritmo LAO* mantemos as três fases descritas para o algoritmo AO*, isto é, a *construção do hipergrafo solução parcial (HGSP)*, *expansão do hipergrafo solução parcial (HGSP)* e a *atualização de custos*. As primeiras duas fases são praticamente as mesmas, isto é, constrói o hipergrafo solução parcial HGSP com as ações gulosas e selecionamos um nó folha do HGSP e o expandimos para gerar os nós sucessores. Porém, a terceira fase apresenta uma variação importante. Enquanto o algoritmo AO* atualiza os nós do conjunto Z (composto pelo nó expandido e seus ancestrais) numa ordem topológica *bottom-up*, o LAO* atualiza os valores dos nós do conjunto Z usando *Iteração de Valor* (Seção 3.3.1) ou *Iteração de Política* (Seção 3.3.2). Dessa maneira, o algoritmo LAO* generaliza a fase de atualização de custos do AO* usando um algoritmo de programação dinâmica de horizonte infinito para MDPs, para lidar com a presença de ciclos.

LAO* constrói um hipergrafo explícito *G'* que inicialmente só contém o nó inicial. O hipergrafo *G'* é expandido usando políticas gulosas, até completar o hipergrafo solução (*HGS*). Quando um hiperarco está direcionado para um nó já visitado, esse nó sucessor não é adicionado a *G'*. Durante a construção do *HGSP*, a ação que minimiza o custo esperado é escolhida e etiquetada como a melhor ação do nó visitado (política gulosa). Cada caminho a partir do nó inicial até os nós folha é um caminho de menor custo esperado (estimado).

LAO* é descrito no Algoritmo 9, recebe como entrada o estado inicial s_0; um hipergrafo G (especificado implicitamente pelo estado inicial e pela função de transição probabilística); a função

[1] Um algoritmo offline computa uma solução antes de iniciar a execução. Por outro lado, no planejamento online, a solução é computada conforme a execução avança.

Algoritmo 9: LAO*(G, s_0, C, $\hat{h}$, ϵ). Fonte:(Hansen e Zilberstein, 2001)

Input:
G: Hipergrafo implícito dado pela função de transição probabilística e pelo estado inicial.
s_0: Estado inicial.
C: Função custo.
$\hat{h}$: Estimativa heurística do custo.
ϵ: erro.
Output: Devolve o hipergrafo solução ótimo ou ϵ-ótimo

1 O hipergrafo explícito inicial G' consiste inicialmente do nó inicial n_0=s_0;
2 **while** O hipergrafo solução parcial tiver algum nó não-terminal **do**
3 | */*Expandir a melhor solução parcial*/*;
4 | Selecionar algum nó não-terminal n_i do hipergrafo solução parcial;
5 | Expandir o nó n_i e adicionar os novos nós sucessores a G'. Para cada novo nó n_j adicionado a G' pela expansão de n_i, se n_j for nó meta então $V(n_j) = 0$, caso contrário $V(n_j) = \hat{h}(n_j)$;
6 | */*Atualização do custo dos nós e etiquetação das melhores hiperarcos (ações)*/*;
7 | Criar um conjunto Z que contém o nó expandido e todos seus ancestrais em G' seguindo os hiperarcos etiquetados, isto é, Z contém aqueles nós ancestrais que podem alcançar o nó expandido seguindo a melhor solução atual;
8 | Executar *Iteração de Política* ou *Iteração de Valor* sobre os nós do conjunto Z para atualizar os custos e determinar o melhor hiperarco para cada nó;
9 | Reconstruir G';
10 **end**
11 */*Teste de convergência*/*;
12 **if** *Iteração de Política* foi usada **then**
13 | Ir para a linha 21;
14 **end**
15 **else**
16 | Executar *Iteração de Valor* sobre os nós do hipergrafo solução parcial;
17 | Continuar até que alguma das seguintes condições seja alcançada;
18 | **i**) Se o erro for menor do que ϵ, para todos os nós n_i de G', então ir à linha 21;
19 | **ii**) Se o hipergrafo solução parcial tiver algum nó não-terminal não expandido, ir à linha 2;
20 **end**
21 **return** Devolver o hipergrafo solução ótimo ou ϵ-ótimo;

custo C e uma função heurística $\hat{h}$. Inicializa G' com n_0, que é o hipergrafo solução parcial inicial. As linhas 2-9 serão executadas enquanto existir algum nó não-terminal no hipergrafo solução parcial. Um nó não-terminal n_i desse grafo é selecionado e expandido, adicionando os novos nós sucessores a G'. Se o nó sucessor for meta, o valor de $V(n_i)$ (Equação 3.4) é igual a 0, caso contrário, $V(n_i)$ é definido pela função heurística $\hat{h}$. Na linha 7, criamos um conjunto de nós, que inclui o nó recém expandido e seus ancestrais em G' que fornecem o menor custo esperado. Sobre esse conjunto aplicamos *Iteração de Valor* ou *Iteração de Política* para atualizar os custos desses nós.

Quando o hipergrafo solução parcial não possui nós não-terminais, não podemos garantir que o hipergrafo encontrado é necessariamente ótimo, por tanto, é realizado um teste de convergência. Se *Iteração de Política* for usada na atualização de custos, os valores dos nós são exatos (valores ótimos), assim, o LAO* alcança convergência para todos esses nós e o hipergrafo solução é ótimo. Quando os custos são atualizados usando *Iteração de Valor*, o teste de convergência é usado para encontrar uma solução ϵ-ótima. O teste consiste em executar *Iteração de Valor* sobre os nós do hipergrafo solução atual. Como o algoritmo pode mudar a escolha da melhor ação de algum nó, o melhor hipergrafo solução pode mudar entre duas iterações consecutivas. Se o melhor hipergrafo solução mudar tal que inclua algum nó não-terminal, a execução do algoritmo continua na linha 2 posibilitando que esse nó seja expandido e o hipergrafo solução possa ser avaliado novamente.

Tanto no AO* quanto no LAO*, a borda do melhor hipergrafo solução parcial pode conter vários nós não-expandidos e a escolha de qual será o próximo nó a expandir é não-determinística (uma vez que todos os nós do hipergrafo solução parcial devem, necessariamente, ser expandidos). O algoritmo LAO* compartilha as propriedades de AO* com relação à condição de parada e do uso de heurísticas. Dada uma função de avaliação heurística admissível, todos os custos dos nós do hipergrafo explícito não superestimam os seus custos reais depois de cada iteração do algoritmo e LAO* converge para uma solução ótima ou ϵ-ótima, sem necessariamente avaliar todos os estados do problema. Dependendo do algoritmo de programação dinâmica usado na fase de atualização de custos, obtemos alguns fatos interessantes. Se *Iteração de Política* for usada:

- $V(n_i) \leq V^*(n_i)$ para cada nó n_i, depois de cada iteração do LAO*.
- $V(n_i) = V^*(n_i)$ para cada nó n_i do melhor hipergrafo solução quando LAO* termina.
- LAO* termina depois de um número finito de iterações.

Por outro lado, se *Iteração de Valor* for usada:

- $V(n_i) \leq V^*(n_i)$ para cada nó n_i em qualquer iteração do algoritmo.
- $V(n_i)$ converge para um valor ϵ perto de $V^*(n_i)$ para cada nó n_i do melhor hipergrafo solução, depois de um número finito de iterações.

Os resultados experimentais realizados em (Hansen e Zilberstein, 2001) mostram que uma implementação do LAO* usando o Algoritmo 9 pode ser muito ineficiente, uma vez que a fase de atualização de custos do algoritmo LAO* precisa atualizar o custo dos nós até a convergência a cada iteração. Ou seja, utilizando o algoritmo LAO*, essa fase (usando *Iteração de Política* ou *Iteração de Valor*), deve atualizar alguns nós muitas vezes até que a convergência seja alcançada, sendo isso computacionalmente custoso. Desse modo, algumas modificações foram introduzidas no algoritmo ILAO*, descrito a seguir.

3.6.3 O algoritmo ILAO*

O Algoritmo 10 define uma versão melhorada do LAO*, chamada de *Improved* LAO* (ILAO*) (Hansen e Zilberstein, 2001). A principal diferença entre LAO* e ILAO* é que não usa os algoritmos de programação dinâmica síncrona (Seção 3.3) para atualizar custos. Na *fase de construção*

do hipergrafo solução parcial (HGSP), ILAO* o constrói seguindo os hiperarcos marcados, isto é, hiperarcos que correspondem com escolhas gulosas. Na *fase de expansão do hipergrafo solução parcial*, o algoritmo ILAO* realiza uma busca em profundidade do melhor hipergrafo solução parcial para encontrar os nós não-terminais e fazer a expansão. O algoritmo ILAO* expande todos esses nós do melhor hipergrafo solução parcial. Além disso, ao invés de executar *Iteração de Política* ou *Iteração de Valor* na *fase de atualização de custos*, ILAO* atualiza todos os nós do melhor grafo solução somente uma vez. Porém, essa atualização é realizada numa ordem inversa à busca em profundidade executada no melhor hipergrafo solução parcial atual.

Algoritmo 10: ILAO*(G, s_0, C, $\hat{h}$). Fonte:(Hansen e Zilberstein, 2001)

Input:
G: Hipergrafo implícito dado pela função de transição probabilística e pelo estado inicial.
s_0: Estado inicial.
C: Função custo.
$\hat{h}$: Estimativa heurística do custo.
Output: Devolve o hipergrafo solução ótimo ou ϵ-ótimo

1 O hipergrafo explícito inicial G' consiste inicialmente do nó inicial n_0=s_0;
2 **while** O hipergrafo solução parcial tiver algum nó não-terminal **do**
3 /*Busca em profundidade e expansão*/;
4 Executar busca em profundidade no melhor hipergrafo solução parcial atual;
5 **foreach** nó n_i visitado em pós-ordem **do**
6 Se o nó n_i não estiver expandido, expanda-o e para cada nó sucessor n_j inicializar $V(n_j)$=$\hat{h}(n_j)$;
7 /*Atualização*/;
8 $V(n_i) = \min_{a \in A(n_i)} [C(n_i, a) + \sum_{n_j \in S} P(n_j|n_i, a)V(n_j)]$ e
9 etiquetar o melhor hiperarco (ação) para n_i;
10 **end**
11 Reconstruir G';
12 **end**
13 /*Teste de convergência*/;
14 Executar *Iteração de Valor* sobre os nós do melhor hipergrafo solução;
15 Continuar até que alguma das seguintes condições seja alcançada;
16 **i**) Se o erro for menor do que ϵ então ir à linha 18;
17 **ii**) Se o melhor hipergrafo solução atual tiver algum nó não-terminal não expandido, ir à linha 2;
18 **return** Devolver o hipergrafo solução ótimo ou ϵ-ótimo;

Cada nó encontrado pelo processo de busca (fase de expansão) é armazenado numa pilha. Depois que todas as expansões forem realizadas, na pilha estarão todos os nós do melhor hipergrafo solução parcial. Esses nós são os únicos que serão atualizados (fase de atualização). A ordem estabelecida pela pilha implica que os nós ancestrais serão atualizados depois que os nós sucessores. A seguir, detalhamos as três fases que permitem implementar o algoritmo ILAO*.

- Construção do hipergrafo solução parcial (HGSP), permite determinar quais nós do hipergrafo guloso são nós não-terminais folhas e não folhas (internos). Saber se ainda existem nós não-terminais folhas, serve como critério de parada do algoritmo. Começando no nó inicial n_0, calculamos o melhor hiperarco (ação) do nó atual e obtemos os nós sucessores através desse hiperarco. A seguir, o método verifica se os nós sucessores pertencem ao hipergrafo guloso. Se todos os sucessores já estão no hipergrafo guloso, eles são também processados e o nó

pai é considerado um nó interno. Caso contrário, o nó é considerado um nó folha e sua ação associada é definida como "noop".

- BUSCA EM PROFUNDIDADE E EXPANSÃO DO HIPERGRAFO SOLUÇÃO PARCIAL (HGSP), permite fazer a busca em profundidade do HGSP consultando os hiperarcos associados aos nós (ações). Quando um nó possui uma ação diferente de "noop", ele é expandido usando essa ação e se verifica que os nós sucessores estão no hipergrafo guloso. Os valores dos custos dos novos nós sucessores são inicializados com valores heurísticos. Quando um nó tiver associada a ação "noop", uma ação gulosa é computada, o nó é expandido e seus nós sucessores são adicionados ao hipergrafo guloso (se não estiverem previamente). Durante a busca, os nós são armazenados numa pilha, na ordem em que foram visitados.

- ATUALIZAÇÃO DA FUNÇÃO VALOR, recebe a pilha gerada pelo procedimento anterior e atualiza os valores dos nós. A ordem de atualização é em pós-ordem, já que todos os nós filhos são atualizados antes que os nós pais. Nessa pilha estarão todos os nós do hipergrafo solução parcial atual.

As iterações continuam até que o melhor hipergrafo solução não tenha nós não-terminais. Quando isso acontecer, o algoritmo de Iteração de Valor é executado sobre os nós do melhor hipergrafo solução encontrado, isto é, as iterações são realizadas até que o resíduo seja menor que um valor ϵ. Finalmente, o hipergrafo ótimo é devolvido como solução. No pior dos casos, $|S|$ atualizações são realizadas, porém, na prática o número é muito menor. Em cada iteração, a complexidade é limitada pelo número de nós presentes no melhor hipergrafo solução nessa iteração.

A Figura 3.3 ilustra a fase de busca em profundidade e expansão do hipergrafo solução parcial. A expansão é baseada nas ações definidas nós como as melhores. Os valores da função valor dos nós do HGSP e as ações associadas são também apresentadas.

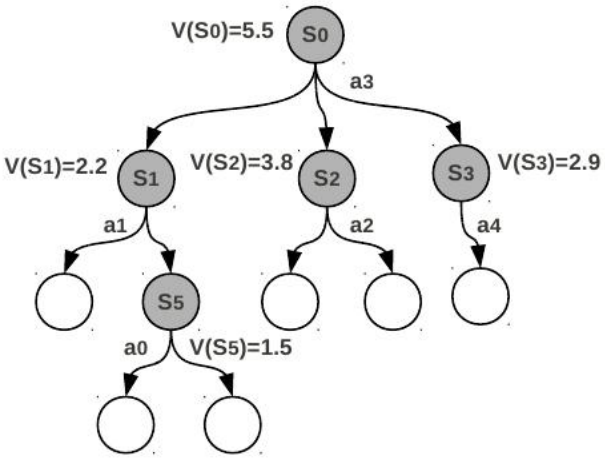

Figura 3.3: *Busca em profundidade e expansão do hipergrafo solução parcial (HGSP). Os nós cinza pertencem ao HGSP.*

Na Figura 3.4 mostramos a fase de atualização da função valor. Os únicos nós atualizados são os nós do HGSP. Os valores são atualizados de maneira bottom-up assim, o nó inicial é o último a atualizar. Dado que os novos valores podem mudar a ação definida anteriormente para um nó, a fase de construção do HGSP é necessária.

Finalmente, na Figura 3.5 mostramos a fase de construção do hipergrafo solução parcial. Os nós em cinza identificam os nós internos e os nós em branco identificam os nós folha. Na construção do HGSP novas ações podem ser escolhidas, por exemplo, as ações dos nós s_2 e s_3 mudaram para a_5 e a_0 respectivamente.

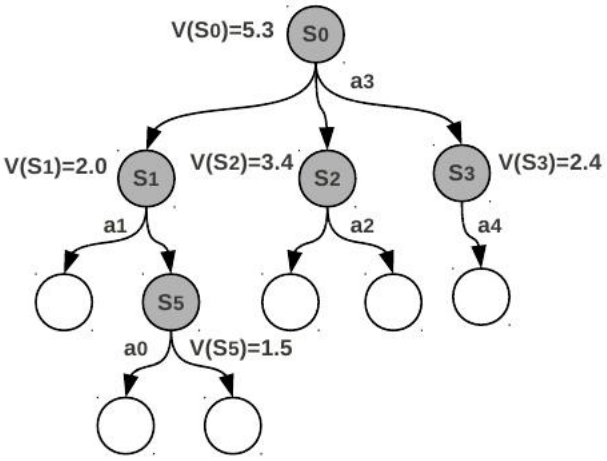

Figura 3.4: *Atualização da função valor do hipergrafo solução parcial (HGSP).*

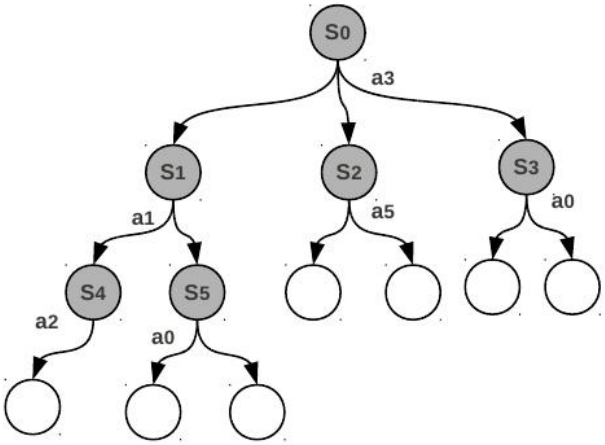

Figura 3.5: *Construção do hipergrafo solução parcial (HGSP).*

3.6.4 Outras extensões do algoritmo LAO*

RLAO*: (Reverse)LAO*

O algoritmo RLAO* (Dai e Goldsmith, 2006) é uma versão do LAO* em que as expansões são realizadas para trás, isto é, começando a partir do nó meta até atingir o nó inicial. A motivação do algoritmo está baseada na crença de que se um nó n_i estiver longe do nó meta, os nós sucessores de n_i também estarão longe da meta. Se a expansão começar do nó inicial, os valores do custo estimado das primeiras iterações serão inexatos. Isto acontece porque nas primeiras iterações, e se ainda não alcançarmos nenhum nó meta, atualizaremos os valores desses nós com valores heurísticos (estimativas). Desse modo, se considerarmos atualizações a partir do nó meta permitirá propagar valores mais exatos.

BLAO*: (Birectional)LAO*

O algoritmo BLAO* (Bhuma e Goldsmith, 2003) extende o algoritmo LAO* para fazer uma busca a partir do nó inicial e também a partir do nó meta em paralelo. O BLAO* recebe como entrada um hipergrafo implícito G e o conjunto de nós meta M. Ao final, BLAO* fornece um hipergrafo explícito G', cujos nós são alcançáveis a partir do nó inicial ou a partir do nó meta e uma política ótima que minimiza o valor esperado do nó inicial. Este algoritmo mantém duas buscas: uma para frente e outra para trás. Inicialmente, os valores da função valor do espaço de estados são atribuídos por valores heurísticos. Ambas buscas começam concorrentemente em cada iteração.

A busca para frente é praticamente a mesma que a busca realizada pelo algoritmo LAO*. Os nós não-expandidos são adicionados ao hipergrafo explícito G' mediante expansão. Numa expansão, um nó não-terminal é escolhido, uma ação gulosa e todos seus sucessores são incluídos em G'. Após a expansão, os valores da função valor do nó expandido e dos seus nós ancestrais são atualizados.

Simetricamente, a busca para trás inicia do nó meta e expande os nós mais promissores na direção do nó inicial. A atualização de custos é a mesma que na busca para frente. Depois de cada iteração, o teste de convergência é realizado para verificar se a diferença entre os valores da função valor de duas iterações consecutivas excede algum erro predefinido. Caso contrário, a política ótima é extraída do hipergrafo solução e o algoritmo acaba.

MBLAO*: (Multi-Thread Birectional)LAO*

A análise experimental do algoritmo BLAO* mostra que faz um menor número de atualizações porque os valores heurísticos dos nós perto do nó meta são frequentemente mais exatos do que aqueles nós perto do nó inicial. Em BLAO*, a busca para trás pode propagar valores mais exatos melhorando os valores heurísticos dos nós que estão mais longe da meta. Baseado na observação anterior (Dai e Goldsmith, 2007) introduz MBLAO* para tentar reduzir o número de atualizações. Nesse trabalho propõe-se iniciar vários "caminhos" (*threads*) na expansão do hipergrafo explícito.

Define-se o *caminho ótimo de um MDP* como o caminho mais provável a partir do nó inicial até o nó meta, se em cada passo seguimos uma política ótima e escolhemos um nó sucessor com a mais alta probabilidade da função de transição. Os pontos de início das buscas para trás são selecionados em nós que estão no caminho ótimo atual. Como a busca está interessada nos nós deste caminho, propagações de valores desde pontos medios deveriam ser mais úteis.

IBLAO*: (Iterative Bounding) LAO*

O algoritmo IBLAO* (Warnquist *et al.*, 2010) faz uso de dois limites sobre o valor do custo ótimo dos nós. O limite inferior permite guiar a busca, por outro lado, o limite superior possibilita podar alguns ramos do hipergrafo de busca. IBLAO* é um algoritmo que fornece soluções ϵ-ótimas, cujas políticas tem limites certos em qualquer momento da execução. Para o algoritmo ser executado em um ambiente online o valor ϵ é dinamicamente modificado, iniciando em um valor alto e posteriormente reduzido pelo refinamento das soluções encontradas.

Quando um nó n for visitado, seus limites inferior e superior $f_l(n)$ e $f_u(n)$, respectivamente, são calculados tal que a relação $f_l(n) \leq V_{\pi^*}(n) \leq f_u(n)$ se mantém ao longo da busca. O IBLAO* começa inicializando o hipergrafo explícito G' com o nó inicial n_0. O laço externo se executa até atingir uma condição de parada definida pelo usuário. O valor $\bar{\epsilon}$ é definido como um fator do limite do erro atual $\hat{\epsilon}(n_0)$ do nó inicial. O conjunto $borda(G'_{\pi_l})$ contém todos os nós extremos de G'_{π_l}, se $borda(G'_{\pi_l}) \neq \emptyset$, selecionamos um subconjunto de nós S_{expand}. O algoritmo escolhe aqueles nós cujas expansões poderiam ter maior impacto no erro estimado do nó inicial.

Anytime AO*

No planejamento online, a decisão do agente está focado na seleção da próxima ação a executar, ao invés de computar uma política para o MDP completo. Isso envolve a seleção de uma ação para o estado atual s. Essa seleção pode ser feita resolvendo um MDP de horizonte finito (Seção 3.2.2), com raiz no estado s e com horizonte definido H. Geralmente, essa escolha está restrita pelo tempo disponível do agente. Os algoritmos Anytime permitem encontrar uma solução válida para um problema inclusive se for interrompido antes do tempo acabar (Zhou e Hansen, 2007).

O algoritmo Anytime AO* (Bonet e Geffner, 2012) é uma modificação do algoritmo AO* (Seção 2.3.2) que permite resolver MDPs de horizonte finito. Introduz um comportamento anytime e garante encontrar uma solução ótima. A principal diferença com relação do AO* está na escolha do próximo nó a expandir. Ao invés de sempre selecionar um nó não terminal do melhor grafo solução parcial, Anytime AO* seleciona com probabilidade p um nó não terminal do grafo explícito que não pertence à solução parcial e portanto, com probabilidade *1-p* um nó não terminal do grafo solução parcial. Se for necessário expandir um nó da solução parcial e ele não estiver disponível, o algoritmo seleciona um nó do grafo explícito e vice-versa. O algoritmo acaba quando as expansões não são possíveis, isto é, não existem nós não terminais no grafo explícito ou quando o tempo acabar.

Anytime AO* é ótimo, se a heurística for admissível ou não, porque o algoritmo acaba quando o grafo explícito tem sido completamente gerado. No pior dos casos, a complexidade do algoritmo não é pior que AO*.

3.6.5 Comparações empíricas dos trabalhos correlatos

A avaliação de desempenho dos algoritmos anteriormente mencionados usa geralmente o domínio chamado de *Racetrack*. O Racetrack foi originalmente proposto em (Barto *et al.*, 1995) e usado em (Hansen e Zilberstein, 2001) para testar o desempenho do LAO* e do RTDP. Esse problema envolve uma simples simulação de uma corrida de carros sobre uma pista. A pista possui qualquer forma e comprimento, ela inclui uma linha de início num extremo e uma linha de chegada no outro. A pista é discretizada como uma grade, em que cada posição representa uma possível localização do carro. Começando na linha de início, o carro tenta se mover ao longo da pista e chegar até a linha de chegada. Desse maneira, a tarefa consiste em dirigir o carro desde um estado inicial até algum estado no conjunto de estados meta. Cada estado do sistema é definido por uma tupla (x,y,dx,dy) que representa a posição e velocidade do carro nas dimensões x e y.

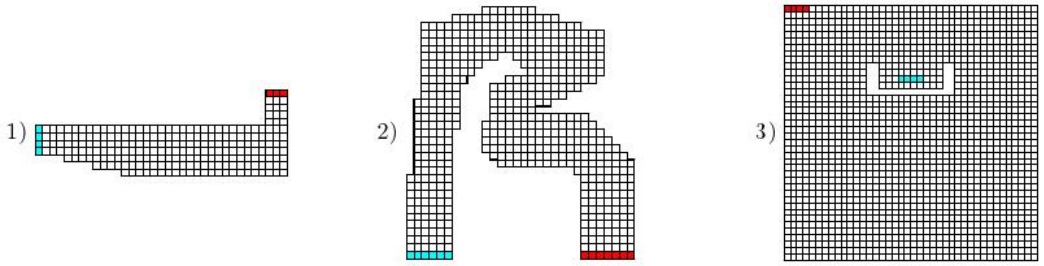

Figura 3.6: *Diferentes pistas do domínio Racetrack, indicando as posições iniciais em cor celeste e as posições objetivo em cor vermelha.*

A Figura 3.6 apresenta algumas pistas usadas nos testes experimentais deste domínio. Os resultados experimentais mostram que o RTDP encontra uma solução de melhor qualidade. Isto acontece porque está focalizado nas trajetórias com alta probabilidade, no entanto, o LAO* sistematicamente explora todas as trajetórias, incluindo as de baixa probabilidade. No entanto, o tempo de convergência do algoritmo LAO* para uma solução ϵ-ótima é muito mais rápido do que RTDP, porque RTDP tende a evitar atualizar os estados sobre trajetórias com baixa probabilidade, o que retarda sua convergência (Hansen e Zilberstein, 2001).

Quando foi avaliado o desempenho dos algoritmos RTDP, LAO* e BLAO* (Bhuma e Goldsmith, 2003), os resultados mostraram que BLAO* converge para uma solução ótima muito mais rápido e expande menos estados do que LAO* e RTDP. No caso dos algoritmos BLAO* e RLAO*, o resultado foi que BLAO* apresenta um melhor desempenho, isto pode ser explicado porque o grau de saída dos estados do hipergrafo inverso é maior do que os graus do hipergrafo original. Os resultados da comparação do algoritmo MBLAO*, o algoritmo BLAO* e o LRTDP mostram que MBLAO*

possui melhor desempenho, devido à introdução de vários *threads* no processo de busca.

O IBLAO* foi avaliado experimentalmente com relação a algoritmos que também usam dois valores para limitar o valor ótimo esperado, isto é, FRTDP e BRTDP, bem como com o algoritmo ILAO* modificado para atualizar o limite superior durante as atualizações de Bellman. Os resultados indicam que o desempenho do IBLAO* supera os outros algoritmos em quase todos as instâncias do domínio. Somente o BRTDP consegue ter um melhor desempenho em alguns casos. No entanto, o algoritmo ILAO* é considerado por alguns trabalhos a versão mais eficiente, devido a sua simplicidade conceitual (Dai *et al.*, 2011). Além disso, ele é frequentemente escolhido pelas comparações empíricas, desse modo, temos mais informação de seu desempenho. Neste trabalho escolhemos o algoritmo ILAO* e propomos uma versão fatorada para resolver MDPs fatorados.

Capítulo 4

Processos de Decisão Markovianos Fatorados

Na definição de MDP apresentada no capítulo anterior, a função recompensa (ou custo) e a função de transição probabilística são dadas em termos de um conjunto de estados explicitamente enumerados. Esses MDPs são chamados de *enumerativos*. O principal problema com este tipo de representação é que os problemas do mundo real possuem um espaço de estados muito grande sendo a enumeração explícita não factível, isto é, MDPs enumerados, podem exigir um consumo excessivo dos recursos computacionais (memória e tempo) tanto na representação do MDP, quanto na computação de uma solução (pode ser exponencial no número de variáveis de estado, gerando uma explosão do número de estados). No entanto, muitos problemas de MDP possuem uma estrutura interna com certas regularidades e propriedades que podem ser exploradas, tanto na representação de problemas como em suas soluções.

Assim, para melhorar a escalabilidade das soluções, é mais conveniente expressar propriedades de estados, ao invés de enumerar todos os estados explicitamente. Uma representação baseada em propriedades ou caraterísticas do estado do sistema é chamada de *fatorada*. Num MDP fatorado, o modelo de transição probabilístico pode ser representado compactamente usando *DBNs* (*Dynamic Bayesian Networks* (Dean e Kanazawa, 1990)) e a função de transição probabilística e recompensa podem ser representadas usando *ADDs* (*Algebraic Decision Diagrams* (Bahar *et al.*, 1993)).

4.1 Representação fatorada de estados

Num MDP fatorado, os estados são caraterizados por um conjunto de variáveis aleatórias $\mathbf{X}=\{X_1,X_2,...,X_n\}$ em que X_i assume valores no domínio $Dom(X_i)$. Um estado é representado por um vetor $\mathbf{x}=\{x_1,x_2,...,x_n\}$ que define um valor $x_i \in Dom(X_i)$ para cada variável X_i. Dessa forma, o vetor $\mathbf{x}$ denota uma instância particular das variáveis aleatórias $\mathbf{X}$ e corresponde a um único estado $s \in S$. O espaço de estados do MDP fatorado é representado pelo conjunto $S = Dom(X_1) \times ... \times Dom(X_n)$. A cardinalidade do espaço de estados S é exponencial no número de variáveis de estado, isto é, $|S|=2^n$ (Mausam e Kolobov, 2012).

4.2 Representação fatorada da função recompensa

No modelo fatorado, assumimos que a função recompensa também é fatorada em um conjunto de funções de recompensa local $R_1(\mathbf{x},a)$, $R_2(\mathbf{x},a)$,...,$R_\phi(\mathbf{x},a)$, em que o escopo de cada $R_j(\mathbf{x},a)$ é restrito a um subconjunto de variáveis de $\mathbf{X}$. Mais formalmente, a recompensa $R(\mathbf{x},a)$ é definida em (Guestrin, 2003) como:

$$R(\mathbf{x}, a) = \sum_{j=1}^{\phi} R_j(\mathbf{x}, a). \tag{4.1}$$

4.3 Representação fatorada da função de transição probabilística

Na representação fatorada de MDPs, podemos representar a aplicação de uma ação num estado que produz efeitos ou mudanças somente em algumas variáveis de estados, uma vez que, em geral, as ações afetam um subconjunto das variáveis. As DBNs constituem um elemento apropriado de representação deste tipo de dependências, possibilitando especificar uma distribuição probabilística local sobre cada variável e descrevendo o impacto da ação sobre as variáveis de estado. Usando DBNs, podemos especificar os efeitos de uma ação sobre um estado, condicionado às variáveis relevantes (Sanner, 2008).

Uma DBN é formada por uma rede de dependências entre as variáveis de um estado $\mathbf{x}$ e as variáveis do estado $\mathbf{x}'$ resultante da execução de uma ação, e uma tabela de probabilidade condicional para cada variável. Assim, as ações de um MDP são descritas por DBNs, em que estrutura da rede para uma determinada ação a precisa de dois conjuntos de variáveis de estado, $\mathbf{x}$ e $\mathbf{x}'$. Ou seja, $\mathbf{x}$= $\{x_1, x_2,..., x_n\}$, o estado antes da ação a ser aplicada e $\mathbf{x}'$= $\{x'_1, x'_2,..., x'_n\}$, o estado obtido após aplicar a ação a.

A rede de dependências de uma DBN também é chamada de *grafo de transição* da DBN que corresponde a um grafo dirigido acíclico com duas camadas: a primeira camada corresponde ao estado $\mathbf{x}$ e a outra camada corresponde ao estado $\mathbf{x}'$. Denotamos por $Pais(\mathbf{x}')$ os pais de $\mathbf{x}'$ no grafo, ou seja, os nós que possuem uma aresta dirigida para x'_i. Essas arestas podem ser direcionadas dos nós da primeira camada para a segunda camada e entre nós da segunda camada, sendo que as arestas indicam uma relação de influência direta entre variáveis.

A tabela de probabilidade condicional (*Conditional Probability Table - CPT*) para cada variável x'_i define uma distribuição de probabilidades condicional sobre x'_i, isto é, os efeitos de a sobre x_i. Basicamente, uma CPT é uma função de probabilidade condicional porém, o valor desta função depende somente das variáveis x_i que são pais de x'_i, ou seja, $P(x'_i|Pais(x'_i),a)$. Em resumo, a função de transição para os valores da variável x'_i e uma ação a, dada por uma CTP, a probabilidade da variável x'_i ter valor x_i dados os valores dos seus pais diretos x_i na DBN e é definida por:

$$P(\mathbf{x}'|\mathbf{x}, a) = \prod_{i=1}^{n} P(x'_i|Pais(x'_i), a). \tag{4.2}$$

Precisamos uma DBN para cada ação de $a \in A$. A Figura 4.1 mostra um exemplo simples de uma DBN para uma ação a qualquer. Na parte a) mostramos as duas camadas da DBN. Observamos que a variável de estado $X_1{}'$ depende das variáveis X_1 e X_2 e a $X_2{}'$ depende unicamente de X_2. Na parte b) mostramos as CPTs para $X_1{}'$ e para $X_2{}'$.

4.4 Diagrama de decisão binário e algébrico

Um diagrama de decisão binário (*Binary Decision Diagram - BDD*) (Akers, 1978), (Bryant *et al.*, 1986) é um grafo dirigido acíclico para representação e manipulação eficiente de funções booleanas. Um BDD representa uma função que mapeia n variáveis booleanas em um valor booleano, isto é, $\mathcal{B}^n \rightarrow \mathcal{B}$. BDDs generalizam a representação estruturada em árvores, permitindo que os nós tenham vários pais, possibilitando a combinação de grafos isomorfos e assim, reduzindo o tamanho da representação. Um BDD com variáveis ordenadas, chamada de OBDD, permite uma representação única (canônica) e compacta. BDDs representam um caso especial dos ADDs.

Um diagrama de decisão algébrico (*Algebraic Decision Diagram - ADD*) (Bahar *et al.*, 1993) representa uma função que mapeia n variáveis booleanas no conjunto de números reais, isto é, uma função da forma $\mathcal{B}^n \rightarrow \mathbb{R}$. Um ADD é definido como um grafo acíclico dirigido, cujos nós podem ser nós internos (nós de decisão) ou nós folhas. Cada nó interno v é rotulado com uma variável

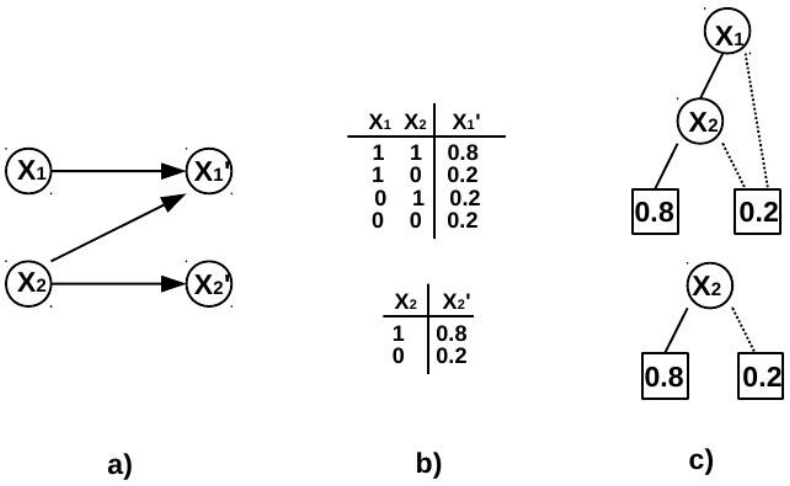

Figura 4.1: *a)Exemplo de DBN para uma ação a de um MDP. b) Tabelas de probabilidade condicional (CPTs) para as variáveis X_1' e X_2', considerando a ação a. c) Representação das CPTs usando ADDs.*

booleana $var(v) \in \mathbf{X}$ e duas arestas dirigidas a dois nós sucessores rotulados com $hi(v)$ e $lo(v)$. A aresta *hi* representa a atribuição 1 (ou verdadeiro) e a aresta *lo* representa a atribuição 0 (falso). Graficamente, a aresta $hi(v)$ é representada por uma linha contínua (ramo verdadeiro) e a aresta $lo(v)$ é representada por uma linha tracejada (ramo falso) (Mausam e Kolobov, 2012). Na Figura 4.2 mostramos uma representação gráfica de um ADD.

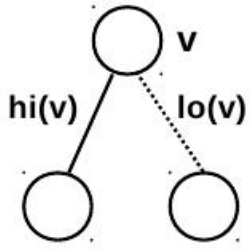

Figura 4.2: *Representação de um ADD. A etiqueta v identifica o nó, o ramo verdadeiro e o ramo falso do nó são identificados por* hi(v) *e* lo(v)*, respectivamente.*

O nó folha é etiquetado com $val(v) \in \mathbb{R}$. O nó de um ADD que não recebe arestas é chamado de *nó raiz*. Num *diagrama de decisão algébrico ordenado*, as variáveis booleanas dos caminhos do grafo, seguem uma ordem linear $X_1 \prec X_2 \prec,...,\prec X_n$. Dada qualquer atribuição de valores das variáveis X_i, percorremos o ADD a partir da raiz, escolhendo as arestas apropriadas (baseado nos valores) até chegarmos numa folha. O nó folha resultante representa o valor da função algébrica.

Mais formalmente, um ADD F, cujos nós folha tem um valor $C \in \mathbb{R}$, pode ser definido por uma gramática BNF (Backus Naur Form):

$$F ::= C | if(F^{var}) \quad then \quad F_{hi} \quad else \quad F_{lo} \tag{4.3}$$

Um ADD denota a função que representa da seguinte maneira (Bryant *et al.*, 1986):

1. A função de um nó folha é uma função constante $F()=C$, em que C é a etiqueta do nó folha.
2. A função de um nó interno etiquetado com uma variável booleana X_1 é dado por: $F(x_1, x_2,..., x_n) = x_1.F_{hi}(x_2,...,x_n) + \bar{x_1}.F_{lo}(x_2,...,x_n)$.

A Figura 4.4 mostra o ADD que representa a tabela da Figura 4.3. Dada qualquer atribuição de valores das variáveis X_1, X_2 e X_3, por exemplo, 1, 0 e 1, respectivamente, o valor pode ser computado começando na raiz X_1 e seguindo a aresta contínua, a aresta tracejada e a aresta contínua para chegar no valor 2.

X_1	X_2	X_3	**X**
0	0	0	0
0	0	1	2
0	1	0	2
0	1	1	2
1	0	0	0
1	0	1	2
1	1	0	5
1	1	1	5

Figura 4.3: *A tabela mostra uma função booleana sendo explicitamente enumerados os valores das variáveis.*

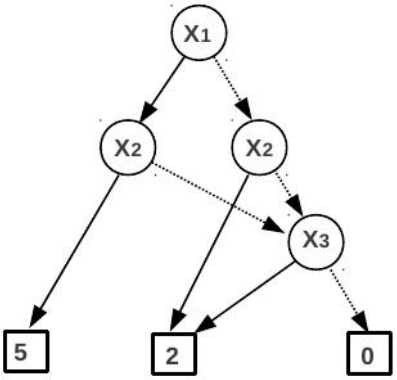

Figura 4.4: *Representação usando um ADD da função representada pela tabela Figura 4.3.*

Uma das principais vantagens de usar ADDs está em que a representação é muito mais compacta do que usar tabelas enumerativas, por exemplo, quando uma função tem uma estrutura tal que algumas variáveis são relevantes somente numa parte do espaço de estados. Esse fato é conhecido como *independência específica do contexto* (*Context Specific Independence* - *CSI*). Para nosso exemplo da Figura 4.4, observamos que se X_1 e X_2 são verdadeiras, então o valor de função é independente do valor de X_3. Notamos que o ADD não representa explicitamente (ou enumera) casos similares, mas sim os agrega num único valor.

Uma outra vantagem dos ADDs é que a complexidade das operações depende do número de nós presentes na estrutura, e não do tamanho do espaço de estados. Quando a ordem das variáveis de estado é definida a priori, as variáveis nunca são repetidas nos caminhos. Dada uma ordem das variáveis, um ADD tem uma única representação mínima. Uma outra vantagem de usar ADDs é a existência de algoritmos eficientes para a manipulação e composição de ADDs que permitem modificar direitamente a estrutura de dados. Se existir suficiente regularidade no modelo, ADDs podem ser muito compactos, permitindo que problemas com grandes espaços de estados possam ser representados e resolvidos.

Existem três conjuntos de operações que podemos usar para manipular ADDs: booleanas, aritméticas e abstração. Essas operações possibilitam a implementação de algoritmos que processam

ADDs. Vários algoritmos foram desenvolvidos para operar sobre ADDs, sendo os dois mais importantes (Mausam e Kolobov, 2012):

- **Reduzir:** Reconstrói um ADD ordenado tornando-o mais compacto sem modificar a avaliação da função. Para isso, essa operação procura por redundâncias na estrutura de dados e as remove. As redundâncias mais comuns são: folhas duplicadas, nós internos duplicados e nós internos redundantes. O algoritmo trabalha numa maneira bottom-up eliminando essas redundâncias e gerando um novo ADD ordenado.
- **Aplicar:** É um algoritmo de composição de funções sobre ADDs ordenados. Toma dois ADDs que representam funções e um operador de composição (soma, produto, máximo, etc.) e calcula o ADD resultante.

Por exemplo, a Figura 4.5 mostra as funções f e g de maneira tabular e usando ADDs. No exemplo, a função g depende unicamente da variável de estado X_2 e assim, pode ser representada de forma compacta por um ADD.

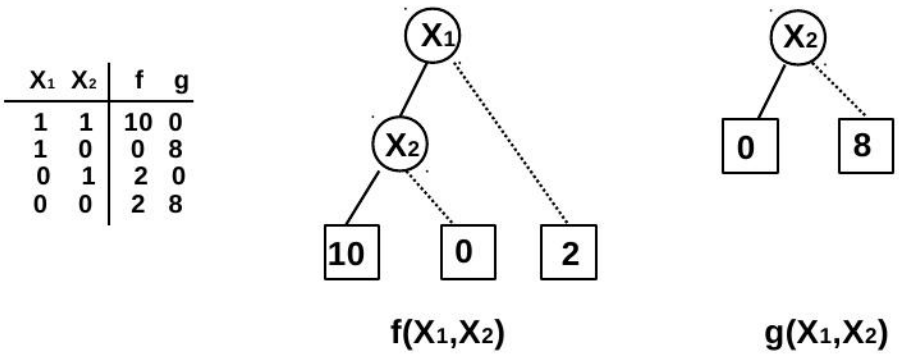

Figura 4.5: *Funções* f *e* g *e suas representações como ADDs. Fonte:(Delgado, 2010)*

Para ilustrar a operação $f + g$, mostramos a representação tabular, os ADDs que representam as funções e o ADD resultante da operação (Figura 4.6).

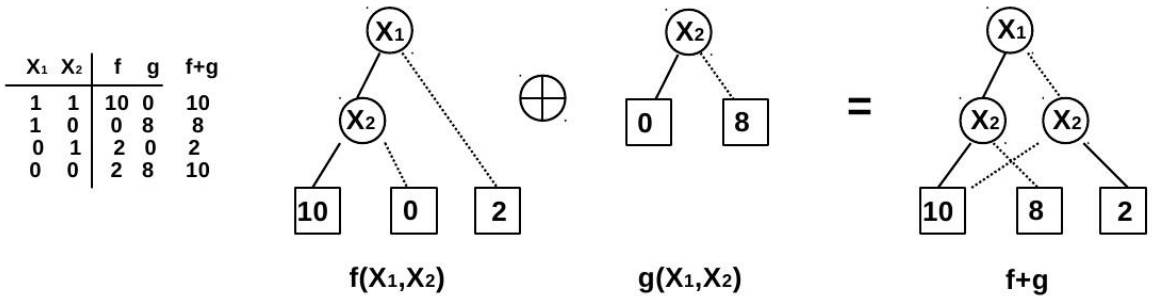

Figura 4.6: *Funções* f *e* g *e o resultante da operação* f+g. *Fonte:(Delgado, 2010)*

Na Figura 4.7 apresentamos o resultado de aplicar a operação unária MAX(f). As operações unárias produzem valores reais, no entanto as operações binárias (MAX(f,g)) devolvem ADDs (Figura 4.8).

Uma outra operação é a *restrição* de uma variável X_i para o valor verdadeiro ou falso. O operação consiste em selecionar aquelas linhas da tabela em que X_i é verdadeiro (1) ou falsa (0). Seja o ADD F, denotamos por $F|X_i = 1$ a operação para o valor verdadeiro e $F|X_i = 0$ a operação para o valor falso. A Figura 4.9 mostra a função Q e o ADD resultante da operação $Q|X_2 = 1$.

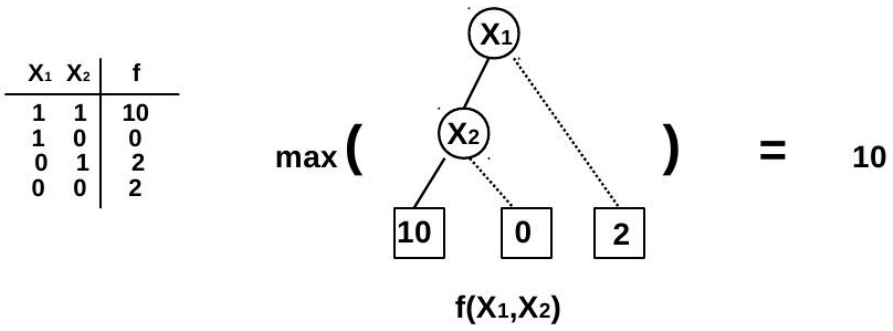

Figura 4.7: *Operação unária MAX sobre a função* f. *Fonte:(Delgado, 2010)*

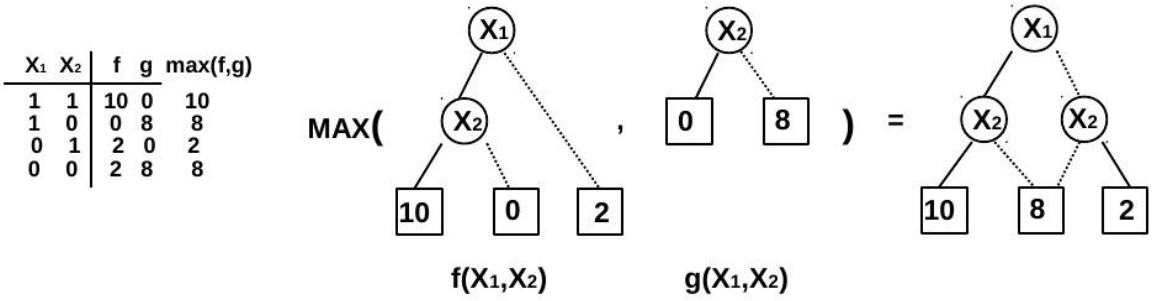

Figura 4.8: *Operação binária MAX sobre as funções* f *e* g. *Fonte:(Delgado, 2010)*

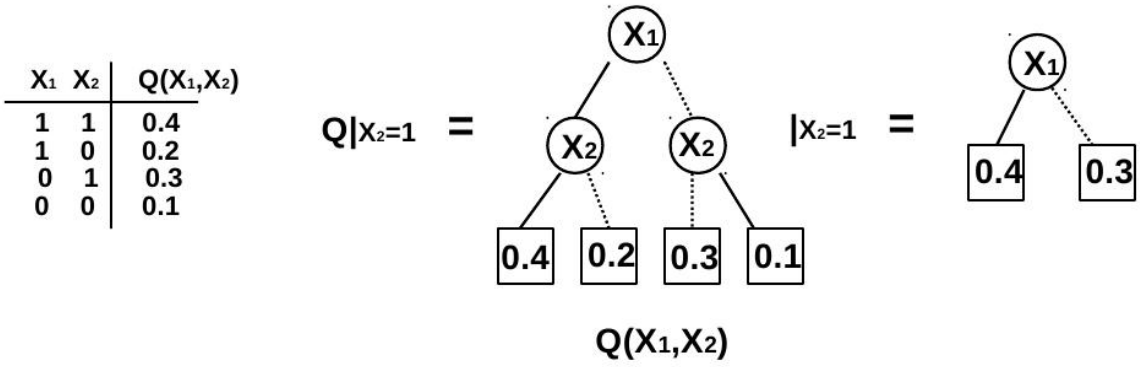

Figura 4.9: *A operação restrição aplicada sobre a função Q. Fonte:(Delgado, 2010)*

A operação *inversão* de um BDD $\mathbf{x}_{DD}$ gera um outro BDD $\overline{\mathbf{x}_{DD}}$ que representa os estados que não estão no BDD original. Isto é análogo à operação de complemento de um conjunto. A Figura 4.10 ilustra o resultado de aplicar a inversão no BDD que representa um conjunto de estados.

A intersecção de dois BDDs é uma outra operação que permite gerar um novo BDD que representa os estados na intersecção dos BDDs. Na Figura 4.11 mostramos um exemplo simples dessa operação.

4.5 Soluções para um MDP Fatorado

Existem vários métodos para encontrar soluções num MDP Fatorado, entre eles:
SPUDD (Hoey *et al.*, 1999), APRICODD (St-Aubin *et al.*, 2001), sLAO* (Feng e Hansen, 2002), sRTDP (Feng *et al.*, 2003). Esses algoritmos tentam agrupar estados similares para reduzir o tamanho do espaço de estados. Usando essa abordagem, os algoritmos executam operações sobre conjuntos de estados, evitando considerá-los individualmente. Usando procedimentos eficientes para ADDs e BDDs otimizam espaço e tempo de processamento. O algoritmo Fact-LRTDP (Gamarra *et al.*,

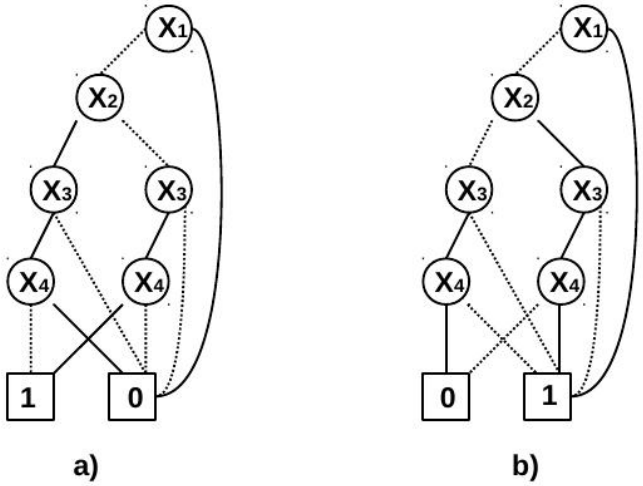

Figura 4.10: *Operação de inversão de uma BDD. a) O BDD inicial. b) O BDD resultado da operação.*

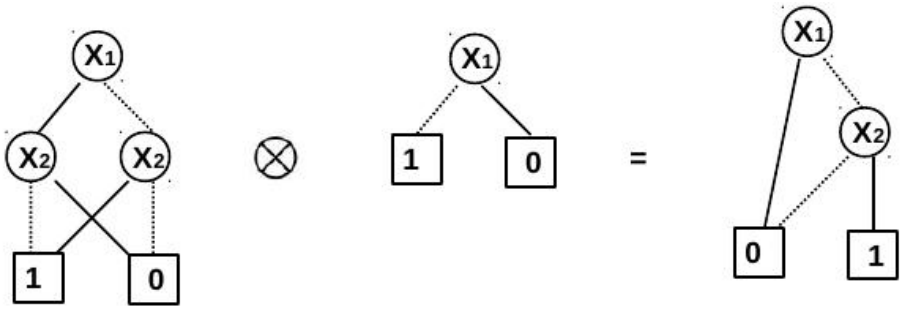

Figura 4.11: *Operação de intersecção de dois BDDs.*

2012) usa uma abordagem diferente quando atualiza a função valor dos estados.

4.5.1 SPUDD (Stochastic Planning using Decision Diagrams)

O algoritmo SPUDD (Hoey *et al.*, 1999) implementa uma versão do algoritmo Iteração de Valor usando ADDs para representar as funções valor, recompensa e as tabelas de probabilidade condicional (CPT). Isso permite capturar as regularidades da estrutura do domínio e evitar redundâncias quando alguns padrões forem encontrados. A atualização de Bellman usando operações sobre ADDs é expressada pela equação:

$$V_{DD}^{t}(\mathbf{x}) = \min_{a \in A}\{Q_{DD}^{t}(\mathbf{x}, a)\}. \tag{4.4}$$

$$V_{DD}^{t+1}(\mathbf{x}) = \min_{a \in A}\{C_{DD}(\mathbf{x}, a) \oplus \gamma \sum_{\mathbf{x}'} \bigotimes_{i=1}^{n} P_{DD}(x_i' | Pais(x_i'), a) V_{DD}^{t}(\mathbf{x}')\}. \tag{4.5}$$

em que os índices DD indicam funções representadas por ADDs. Nessa equação, a operações de marginalização, $\oplus$, $\otimes$ e min são usadas sobre ADDs. Em cada iteração, o ADD V_{DD}^{t+1} é atualizado com o valor mínimo dos $Q_{DD}^{t}(\mathbf{x},a)$.

O algoritmo SPUDD (Algoritmo 11) produz uma sequência de funções valor tal que a estrutura de V_{DD}^{t} é explorada para representar a estrutura de V_{DD}^{t+1}. O algoritmo inicialmente constrói os ADDs para cada CPT de todas as ações. O algoritmo é executado até atingir o número máximo de iterações ou até encontrar que o erro de Bellman for menor que o valor de tolerância (tol) . No laço principal, para cada ação, a função REGRESS é chamada e o V_{DD}^{t} é atualizado com o mínimo valor dos Q_{DD}^{t}.

Algoritmo 11: SPUDD(M, tol, maxIter). Fonte: (Delgado, 2010)

Input:
M: Representacao fatorada do MDP.
tol: Tolerância.
maxIter: Número máximo de iterações.
Output: Devolve a função valor V_{DD}^t.

```
1  ∀a∈A, criar os ADDs das CPTs ;
2  t = 0;
3  V_DD^0 = 0;
4  repeat
5  |   t = t + 1;
6  |   V_DD^t = ∞;
7  |   foreach a ∈ A do
8  |   |   Q_DD^t = REGRESS(V_DD^{t-1},a);
9  |   |   V_DD^t = min(V_DD^t, Q_DD^t);
10 |   end
11 |   Diff_DD^t = V_DD^t ⊖ V_DD^{t-1};
12 |   BE = max(max(Diff_DD^t),-min(Diff_DD^t));
13 |   if BE < tol then
14 |   |   break;
15 |   end
16 |   return V_DD^t;
17 until (t < maxIter) ;
```

A função auxiliar REGRESS (Algoritmo 12) substitui as variáveis de estado de V_{DD}^{t-1} pelas variáveis linhas correspondentes, obtidas depois de executar uma ação. Para gerar Q_{DD}^t, as CPTs de cada ação são multiplicadas e a variável X_i' é eliminada usando a operação de marginalização. A multiplicação pelo fator de desconto e a soma da recompensa completam o valor da atualização de Bellman (Delgado, 2010).

Algoritmo 12: REGRESS(V_{DD}, a). (Delgado, 2010)

Input:
V_{DD}: Função valor.
a: Ação.
Output: Q_{DD}

```
1  Q_DD = CONVERTER-PARA-LINHAS(V_DD);
2  foreach X_i' em Q_DD do
3  |   Q_DD = Q_DD ⊗ CPT(x_i',a);
4  |   Q_DD = ∑_{x_i'} Q_DD;
5  end
6  Q_DD = R_DD ⊕ (γ ⊗ Q_DD);
7  return Q_DD;
```

4.5.2 sRTDP (Symbolic RTDP)

O algoritmo sRTDP (Symbolic Real-Time Dynamic Programming) (Feng *et al.*, 2003) usa técnicas simbólicas de verificação de modelos para atualizar grupos de estados ao invés de estados individuais como calculado na Equação 4.5. sRTDP é uma solução fatorada assíncrona que constrói estados abstratos, isto é, um conjunto de estados que compartilham alguma propriedade. O algoritmo é baseado no algoritmo RTDP (Seção 3.4) e constrói um ADD para representar o estado

abstrato que contém o estado atual visitado pelo *trial*. Desse modo, as atualizações são realizadas nesses estados abstratos e não sobre estados individuais. Dado o estado abstrato E, a equação que atualiza os estados desse conjunto, é apresentada a seguir:

$$V_{DD}^{t+1,E}(\mathbf{x}) = \min_{a \in A}\{C_{DD}(\mathbf{x},a) \oplus \gamma \sum_{\mathbf{x}'} \bigotimes_{i=1}^{n} P_{DD}^{E \cup E'}(x_i'|Pais(x_i'),a) V_{DD}^{t,E}(\mathbf{x}').\} \tag{4.6}$$

Apesar dessa ser uma ideia interessante para melhorar a eficiência do algoritmo SPUDD, usando a informação do estado inicial, a computação do estado abstrato E e seus estados sucessores é muito custosa.

4.5.3 Fact-LRTDP

O algoritmo Fact-LRTDP (Gamarra *et al.*, 2012) é uma versão fatorada do algoritmo LRTDP (Bonet e Geffner, 2003). A principal diferença com relação ao sRTDP é a atualização do valor dos estados. Como mencionamos anteriormente, o sRTDP precisa atualizar todos os estados do estado abstrato E que contém o estado atual o que é computacionalmente custoso. A ideia do algoritmo Fact-LRTDP é atualizar somente o estado atual (representado por um BDD chamado de x_{DD}), definida por:

$$v^{t+1}(\mathbf{x}) = \min_{a \in A}\{Q_{DD}^{t(a)}(\mathbf{x},a)\}. \tag{4.7}$$

$$v^{t+1}(\mathbf{x}) = \min_{a \in A}\{C_{DD}(\mathbf{x},a) \oplus \gamma \sum_{\mathbf{x}'} \bigotimes_{i=1}^{n} P_{DD}(x_i'|Pais(x_i'),a) V_{DD}^{t}(\mathbf{x}')\}. \tag{4.8}$$

O algoritmo introduz uma versão fatorada do processo de etiquetagem CHECK-SOLVED (Seção 3.4) usando BDDs. Quando um estado é atualizado, seu valor é atualizado no ADD que representa a função valor atual. O novo valor de $\mathbf{x}$ obtido pela Equação 4.8 é inserido na função valor atual V_{DD}^{t} para obter uma nova função V_{DD}^{t+1}. As operações necessárias nessa inserção podem ser feitas eficientemente usando operações sobre ADDs. Na atualização da função valor, todas as variáveis de V_{DD}^{t} são convertidas em variáveis linhas ($X_i \rightarrow X_i$'). Em seguida, o valor $Q_{DD}^{t(a)}$ para cada $a \in A$ é calculado usando operações sobre ADDs.

A Figura 4.12 ilustra o processo de atualização de um estado no ADD da função valor. Na parte a) mostramos a função valor no tempo t; em b) o estado $\mathbf{x}_{DD}$=(T,T,F,F) é representado como um BDD. Na parte c), mostramos o ADD que representa o valor $\Delta v = (v^{t+1}(\mathbf{x}) - v^{t}(\mathbf{x})).\mathbf{x}_{DD}$, isto é, a variação Δv entre os valores de $\mathbf{x}$ em t e $t+1$; se $\Delta v > \epsilon$, uma nova folha é criada em V_{DD}^{t+1} com valor $v^{t+1}(\mathbf{x})$. Finalmente, em d) mostramos a função valor no tempo $t+1$.

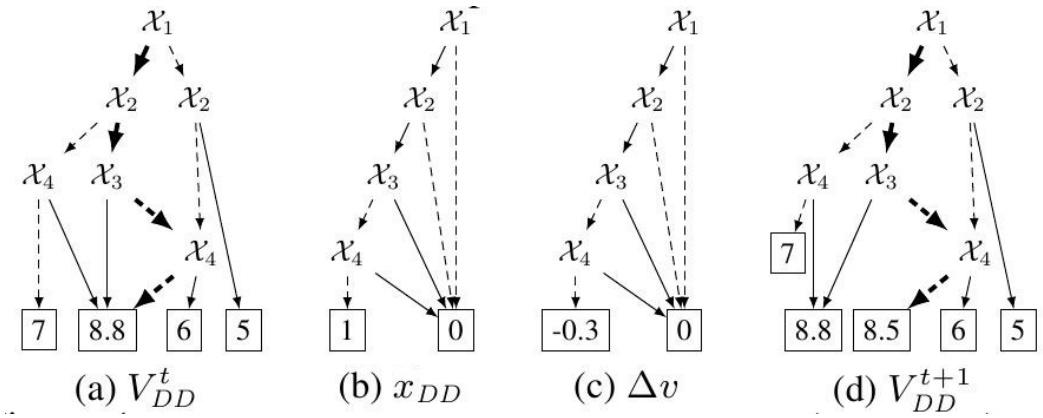

Figura 4.12: *Atualização de função valor para o estado $\mathbf{x}$. a) O ADD V_{DD}^{t}, b) O estado $\mathbf{x}_{DD}$, c) O valor de* Δv*, d) A função valor após atualização V_{DD}^{t+1}. Fonte: (Gamarra et al., 2012)*

4.5.4 sLAO* (Symbolic LAO*)

O algoritmo sLAO* (Feng e Hansen, 2002) é uma versão fatorada (simbólica) do algoritmo LAO* que combina análise de alcançabilidade e representações fatoradas baseadas em ADDs. Como no sRTDP, esse algoritmo manipula conjuntos de estados, ao invés de estados individuais. A ideia chave do sLAO* é limitar a computação somente ao subconjunto de estados alcançáveis a partir do estado inicial e seguindo a melhor política. Para isso, o algoritmo introduz o conceito de "máscara" de um ADD D. Dado um conjunto de estados S, definimos sua função caraterística $\mathcal{X}_{\mathcal{S}}$, tal que $s \in S \Leftrightarrow \mathcal{X}_S(s) = 1$.

Dado um ADD $\mathbf{D}$ e um conjunto de estados relevantes U, a máscara é computada multiplicando $\mathbf{D}$ e a função caraterística de U $\mathcal{X}_U$, gerando um novo ADD $\mathbf{D}_U$, em que os estados não relevantes são mapeados para zero e os estados relevantes são mapeados para valores desses estados em $\mathbf{D}$. sLAO* mantém um conjunto de estados expandidos G, a função valor parcial (somente para os estados de G) e a política parcial π.

O algoritmo sLAO* possui duas fases alternadas: (1) a fase de *expansão da solução parcial* e (2) a fase de atualização de custos. Na fase de expansão da solução, sLAO* realiza análise de alcançabilidade, a partir do conjunto de estados iniciais S_0 para encontrar o conjunto de estados alcançáveis F. A operação chave nesta fase é o cálculo da *Imagem* de um conjunto de estados. O operador imagem computa o conjunto dos estados sucessores do conjunto S após uma ação a ser executada, sendo P^a a função de transição probabilística para a ação a. Formalmente, é definida como: $Imagem_{X'}(S, P^a) = \exists \mathbf{x}\,[P(x'|\mathbf{x},a) \wedge \mathcal{X}_S(\mathbf{x})]$. Esse operador devolve uma função caraterística sobre $\mathbf{x}'$, que representa o conjunto de estados alcançáveis após uma ação ser tomada a partir do conjunto S. Uma outra informação útil usada no algoritmo é a política parcial π_G. Se uma ação a é associada a um conjunto de estados, que tem a como ação gulosa definida pela política parcial atual π_G, esse conjunto é denotado por S^a_π. Essa representação da política gera uma partição do conjunto de estados visitados pela política.

Na fase de *atualização de custos*, o algoritmo usa uma versão modificada do algoritmo SPUDD (Seção 4.5.1), para focar a computação somente no conjunto de estados visitados pela política atual, usando a operação de máscara para encontrar esses estados, e portanto, reduzir o espaço de estados. O algoritmo constrói o estado abstrato E e o atualiza. Sendo π_G uma política parcial, existem estados em E cujos estados sucessores não pertencem a G. Denotamos esse conjunto por E'. Com todas as funções representadas por ADDs, o sLAO* computa os novos valores usando a equação:

$$V_{DD}^{t+1,E}(\mathbf{x}) = \min_{a \in A}\{C_{DD}^{E}(\mathbf{x}, a) \oplus \gamma \sum_{\mathbf{x}'} \bigotimes_{i=1}^{n} P_{DD}^{E \cup E'}(x'_i | Pais(x'_i), a) V_{DD}^{\prime t, E'}(\mathbf{x}')\}. \tag{4.9}$$

Capítulo 5

ILAO* Fatorado

Neste capítulo propomos a versão fatorada do algoritmo ILAO* (Seção 3.6.3), chamada de *Fact-ILAO**. O algoritmo Fact-ILAO* usa as três fases usadas pelo algoritmo ILAO*, isto é, a *reconstrução do hipergrafo solução*, a *expansão do hipergrafo solução fazendo busca em profundidade* e a *atualização do custos em pós-ordem* (com relação aos nós visitados pela busca em profundidade), sendo que as três fases são realizadas de maneira fatorada. Para isso, Fact-ILAO* representa os estados através de variáveis de estado. Um estado s é representado por um BDD, denotado por x_{DD}, em que o caminho formado pelos valores das variáveis em s atinge o valor 1 e todos os outros caminhos atingem o valor 0. Fact-ILAO* representa o hipergrafo solução parcial G' através de conjuntos de nós (NI é o conjunto de nós internos de G', e NF é o conjunto de nós folha de G') e a política gulosa $\pi_{G'}$, que associa uma ação gulosa a cada estado $\mathbf{x} \in NI$ e NOOP a cada estado $\mathbf{x} \in NF$. Denotamos por NI_{DD} e NF_{DD} os estados internos e folhas de G' representados por BDDs. Além disso, Fact-ILAO* guarda um conjunto de todos os estados já gerados durante sua execução, chamado de H_{DD}, para reusar os valores desses estados no hipergrafo solução parcial reconstruído.

Fact-ILAO* (Algoritmo 13) recebe como entrada a representação fatorada de um MDP. Desse modo, a função valor e a função recompensa são representadas como ADDs. A função valor de todos os estados é inicializada com um valor heurístico. As CPTs (Seção 4.3) das variáveis de estado e ações, também são representadas por ADDs e o estado inicial é representado por um BDD. As linhas 2 e 3 do Algoritmo 13 inicializam o conjunto NI_{DD} com um conjunto vazio e NF_{DD} com $\mathbf{x_0}$. Desse modo, inicialmente, o hipergrafo solução parcial G' é composto apenas por $\mathbf{x_0}$. Enquanto o conjunto NF_{DD} for diferente de vazio, o laço principal é executado (linhas 4-11). Na linha 6 é feita a expansão em profundidade, em que todos os nós folha encontrados são expandidos, devolvendo uma pilha $STACK_{G'}$ dos nós visitados internos de G' (isto é NI_{DD} ordenados com relação à busca em profundidade). Na linha 8, é realizada a atualização do custo dos estados na ordem de extração da pilha. A ordem de atualização de estados garante que estados descendentes sejam atualizados antes que estados ancestrais. O método AtualizarCustos também devolve a política gulosa $\pi_{G'}$. Uma vez que os custos foram atualizados, a reconstrução do hipergrafo solução (linha 10) é feita para computar novas ações gulosas e determinar os novos conjuntos NI_{DD} e NF_{DD}. Para isso, esses conjuntos são recomputados usando da política $\pi_{G'}$ e operações sobre BDDs e ADDs. Quando o algoritmo sai do laço principal, o teste de convergência é realizado e um algoritmo de Iteração de Valor é executado (usando a versão fatorada SPUDD (Seção 4.5.1)) sobre o conjunto de estados NI_{DD} até que o erro seja menor que ϵ ou até $NF_{DD} \neq \emptyset$.

Algoritmo 13: FACT-ILAO*(M, $\mathbf{x_0}$, ϵ, $\hat{h}$)

Input:
M: Representação fatorada do MDP (DBNs, ADDs representando as CPTs e função custo).
$\mathbf{x_0}$: Estado inicial
Variável global: H_{DD}, inicializada com $\mathbf{x_0}$;
ϵ: Erro
$\hat{h}$: Estimativa heurística
Output: $\pi_{G'}$

1 $V_{DD} = \hat{h}$;
2 $\pi_{G'} = \pi_{G'}(\mathbf{x_0})$ = NOOP;
3 $NI_{DD} = \emptyset$;
4 $NF_{DD} = \mathbf{x_0}$;
5 **while** $NF_{DD} \neq \emptyset$ **do**
6 | */*Expansão de G' usando busca em profundidade a partir de* $\mathbf{x_0}$*, consultando* $\pi_{G'}$ **/*;
7 | $STACK_{G'}$ = EXPANSÃOEMPROFUNDIDADE($\mathbf{x_0}$, $\pi_{G'}$);
8 | */*Atualizar, em pós-ordem, os estados visitados pela busca em profundidade */*;
9 | $\pi_{G'}$ = ATUALIZARCUSTOS($STACK_{G'}$) (Ver Equação 4.8);
10 | */*Reconstrói o hipergrafo solução a partir de* $\mathbf{x_0}$ *para atualizar* $\pi_{G'}$ *com as ações gulosas*/*;
11 | (NI_{DD}, NF_{DD}, $\pi_{G'}$) = RECONSTRUIRSOLUÇÃO($\mathbf{x_0}$, $\pi_{G'}$);
12 **end**
13 */*Teste de convergência*/*;
14 Executar *Iteração de Valor* (SPUDD) sobre os nós NI_{DD};
15 Continuar até que alguma das seguintes condições seja alcançada;
16 **i**) Se o erro for menor do que ϵ, então ir à linha 18;
17 **ii**) Se $NF_{DD} \neq \emptyset$, ir à linha 5;
18 **return** $\pi_{G'}$;

As três fases do algoritmo Fact-ILAO*, realizadas de maneira fatorada, são descritas nas próximas seções em mais detalhes.

5.1 Reconstrução do hipergrafo solução parcial

O Algoritmo 14 reconstrói o melhor hipergrafo solução parcial G'. Como a fase de atualização de custos pode provocar mudança na ação considerada gulosa para um estado, é necessário reconstruir o hipergrafo solução. Os conjuntos NI_{DD} e NF_{DD} são inicializados como conjuntos vazios (Linhas 1-2). Além disso, o algoritmo usa um conjunto auxiliar AUX_{DD} para guardar os estados gerados durante a reconstrução de G', inicializado com $\mathbf{x_0}$ (Linhas 3-4). A política atual $\pi_{G'}$ é armazenada explicitamente para permitir a rápida consulta e atualização de valores dos estados. Para processar a reconstrução do hipergrafo solução parcial, utilizamos uma fila chamada de *abertos*. No procedimento RECONSTRUIRSOLUÇÃO, fazemos uso de dois procedimentos que fazem operações com BDDs e ADDs:

- COMPUTAR-SUCESSORES($\mathbf{x}$,a), gera um BDD que contém os estados sucessores do estado $\mathbf{x}$, dada uma ação a, chamado de suc_{DD}. O método usa as CPTs das variáveis de estados (definidas por P_{DD}). Quando essa probabilidade for maior que 0, o estado sucessor $\mathbf{x}$' atinge 1 em suc_{DD}, caso contrário será 0.

- *abertos*.ADICIONAR() adiciona um estado na fila *abertos*.
  Por outro lado, *abertos*.EXTRAIR() extrai um estado da fila *abertos*.

Durante a reconstrução do hipergrafo solução, a fila *abertos* armazena os estados que já estão no conjunto H_{DD} e que ainda não foram visitados. Enquanto a fila *abertos* tiver estados para processar, extraímos um estado $\mathbf{x}$ (linha 7), usamos a ação gulosa $\pi_{G'}(\mathbf{x})$ e com COMPUTAR-SUCESSORES($\mathbf{x}$,a) calculamos suc_{DD}, isto é, o BDD dos estados sucessores de $\mathbf{x}$ (linha 9).

Na linha 10 calculamos, com operações de complemento e intersecção entre BDDs, os estados sucessores que não pertencem à H_{DD} (isto é, nunca antes gerados). O estado $\mathbf{x}$ é adicionado no conjunto NI_{DD} somente se todos seus sucessores também estão no conjunto NI_{DD}. Se algum sucessor não está nesse conjunto, adicionamos o estado atual à NF_{DD} ($\mathbf{x}$ deverá ser propriamente expandido na fase de expansão do hipergrafo). Essa verificação é realizada através de operações com BDDs, da seguinte forma: (1) obtemos o conjunto de sucessores que não estão no conjunto H_{DD}, computando o complemento de H_{DD} e fazendo a intersecção com suc_{DD} (linha 15); se o BDD resultante dessa operação for diferente de vazio então, existe algum estado sucessor que não está em H_{DD} e o estado atual passa a ser um estado folha e é inserido no conjunto NF_{DD}; finalmente a política $\pi_{G'}$ para o estado $\mathbf{x}$ é atualizada com a ação NOOP (linhas 24-25). Caso o conjunto de sucessores já possua estados no conjunto AUX_{DD}, fazemos novamente a operação complemento sobre AUX_{DD} e realizamos a intersecção com o conjunto suc_{DD} (linha 15). Se o conjunto resultante for diferente de vazio, o adicionamos em *abertos* e realizamos a operação de união com o conjunto AUX_{DD} (linha 19).

Algoritmo 14: ReconstruirSolução($\mathbf{x_0}$, $\pi_{G'}$)

Input:
$\mathbf{x_0}$: Estado inicial.
$\pi_{G'}$: Política gulosa.
Variável global:
H_{DD}.
Output:
NI_{DD}, NF_{DD}: Conjuntos de estados.
$\pi_{G'}$: Política gulosa.

```
1  NI_DD = ∅;
2  NF_DD = ∅;
3  AUX_DD = x_0;
4  abertos = Fila vazia;
5  abertos.ADICIONAR(x_0);
6  while abertos ≠ FILA-VAZIA do
7      x = abertos.EXTRAIR();
8      if π_G'(x) ≠ NOOP then
9          suc_DD = COMPUTAR-SUCESSORES(x,π_G'(x));
10         SucNonH_DD = complement(H_DD) ∩ suc_DD;
11         /*Se todos os sucessores de x foram gerados anteriormente, então x é adicionado em
           NI_DD, senão x passa a ser folha e será expandido na fase de expansão em
           profundidade.*/;
12         if SucNonH_DD == ∅ then
13             /*Todos os sucessores de x foram visitados anteriormente.*/;
14             NI_DD = NI_DD ∪ x;
15             SucNonAUX_DD = complement(AUX_DD) ∩ suc_DD;
16             /*Adiciona os nós na fila abertos que ainda não foram visitados durante a
               reconstrucção.*/;
17             if SucNonAUX_DD ≠ ∅ then
18                 abertos.ADICIONAR(SucNonAUX_DD);
19                 AUX_DD = AUX_DD ∪ SucNonAUX_DD;
20             end
21         end
22         else
23             /*Existe um sucessor de x que não foi gerado anteriormente, então x é folha.*/;
24             NF_DD = NF_DD ∪ x;
25             π_G'(x) = NOOP;
26         end
27     end
28 end
29 return NI_DD, NF_DD, π_G';
```

5.2 Expansão do hipergrafo solução fazendo busca em profundidade

O Algoritmo 15 realiza a busca em profundidade no hipergrafo solução atual, consultando a política gulosa atual $\pi_{G'}$. Para isso, fazemos uso da pilha *abertos* para armazenar os estados a serem visitados e usamos a pilha $STACK_{G'}$ para guardar os estados que visitamos no percurso da busca em profundidade e cujo valor será atualizado posteriormente pela fase de atualização da função valor. Enquanto *abertos* conter estados (linha 5), extraímos um estado $\mathbf{x}$ fazendo uso do método *abertos*.DESEMPILHAR (análogo ao método *abertos*.EXTRAIR mas, neste caso,

usamos uma pilha) e empilhamos o estado na pilha $STACK_{G'}$ (Linha 7). Em seguida, consultamos a política $\pi_{G'}$ para determinar a ação associada ao estado atual $\mathbf{x}$. Se a ação for diferente de NOOP (linha 9), expandimos o estado através dessa ação e calculamos o conjunto de sucessores. Na linha 12, utilizamos as operações de complemento e intersecção de BDDs para obter o conjunto de sucessores sem repetição. Se o BDD resultante for diferente de vazio, indicando que ainda existem sucessores que devem ser processados, os empilhamos na pilha *abertos* e atualizamos o conjunto AUX_{DD} com a operação de união entre BDDs (linhas 15-16).

Algoritmo 15: ExpansãoEmProfundidade($\mathbf{x_0}$, $\pi_{G'}$)

Input:
$\mathbf{x_0}$: Estado inicial.
$\pi_{G'}$: Política gulosa
Output: $STACK_{G'}$: Pilha

```
1  AUX_DD = x_0;
2  abertos = Pilha vazia;
3  STACK_G' = Pilha vazia;
4  abertos.EMPILHAR(x_0);
5  while abertos ≠ ∅ do
6  |  x = abertos.DESEMPILHAR();
7  |  STACK_G'.EMPILHAR(x);
8  |  /*A política π_G' prescreve uma ação gulosa para o estado x.*/;
9  |  if π_G'(x) ≠ NOOP then
10 |  |  a = π_G'(x);
11 |  |  suc_DD = COMPUTAR-SUCESSORES(x,a);
12 |  |  sucNonAUX_DD = ¬AUX_DD ∩ suc_DD;
13 |  |  /*Existem estados sucessores de x não foram visitados anteriormente.*/;
14 |  |  if sucNonAUX_DD ≠ ∅ then
15 |  |  |  abertos.EMPILHAR(sucNonAUX_DD);
16 |  |  |  AUX_DD = AUX_DD ∪ sucNonAUX_DD;
17 |  |  end
18 |  end
19 |  else
20 |  |  a = ACAO-GULOSA(V^t_DD,x);
21 |  |  suc_DD = COMPUTAR-SUCESSORES(x,a);
22 |  |  /*Atualizar H_DD com os novos estados sucessores.*/;
23 |  |  sucNonH_DD = ¬H_DD ∩ suc_DD;
24 |  |  if sucNonH_DD ≠ ∅ then
25 |  |  |  H_DD = H_DD ∪ sucNonH_DD;
26 |  |  end
27 |  end
28 end
29 return STACK_G';
```

Se a ação definida pela política $\pi_{G'}$ para o estado $\mathbf{x}$ for NOOP (linha 19), computamos a ação gulosa (linha 20) e obtemos o BDD que representa o conjunto de sucessores através dessa ação (linha 21), suc_{DD}. Na linha 23, novamente com as operações de complemento e intersecção entre BDDs geramos o conjunto de novos sucessores. Se o BDD resultante conter estados, H_{DD} é atualizado através da operação de união (linha 25). Quando a busca em profundidade acabar, na pilha $STACK_{G'}$ estarão todos os estados internos do hipergrafo guloso atual na ordem em que foram visitados pela busca em profundidade. A ordem determinada pela pilha permitirá atualizar primeiro os estados filhos antes que os pais (pós-ordem), na fase de atualização de custos.

5.3 Atualização do custo em pós-ordem

Nessa fase, o algoritmo Fact-ILAO* visita cada estado do melhor hipergrafo solução atual do mesmo modo que o algoritmo ILAO* (Algoritmo 16). Fact-ILAO* não precisa fazer uma atualização de Bellman de todos os estados como no algoritmo SPUDD ou de estados abstratos como é feito em sRTDP e sLAO*, mas sim realiza uma atualização individual de estados como no algoritmo Fact-LRTDP (Seção 4.5.3) de maneira eficiente usando ADDs.

Esse procedimento, recebe a pilha $STACK_{G'}$ devolvida pela busca em profundidade durante a expansão e atualiza a função valor de cada estado (na ordem definida pela pilha). As atualizações continuam até a pilha ficar sem estados. Desse modo, todos os estados do hipergrafo solução atual são atualizados e, na próxima fase, um novo hipergrafo guloso será reconstruido. O Algoritmo 16 mostra o procedimento de atualização do Fact-ILAO*.

Algoritmo 16: AtualizarCustos($STACK_{G'}$)

Input: $STACK_{G'}$: Pilha
Output: $\pi_{G'}$

1 **while** $STACK_{G'} \neq \emptyset$ **do**
2 $\quad \mathbf{x} = STACK_{G'}$.DESEMPILHAR();
3 $\quad v^{t+1}(\mathbf{x}) = \min_{a \in A}\{C_{DD}(\mathbf{x},a) \oplus \gamma \sum_{\mathbf{x}'} \bigotimes_{i=1}^{n} P_{DD}(x'_i | Pais(x'_i), a) V^t_{DD}(\mathbf{x}')\}$;
4 $\quad \pi_{G'}(\mathbf{x}) = \operatorname{argmin}_{a \in A}\{C_{DD}(\mathbf{x},a) \oplus \gamma \sum_{\mathbf{x}'} \bigotimes_{i=1}^{n} P_{DD}(x'_i | Pais(x'_i), a) V^t_{DD}(\mathbf{x}')\}$;
5 $\quad$ Atualizar V^t_{DD} com $v^{t+1}(\mathbf{x})$ para obter V^{t+1}_{DD};
6 **end**
7 **return** $\pi_{G'}$;

5.4 Fact-ILAO* vs. sLAO*

A primeira diferença entre o Fact-ILAO* e sLAO* é que sLAO* é baseado no algoritmo LAO* e nosso algoritmo é baseado no ILAO*. Assim, as fases para computar o hipergrafo solução são diferentes. O algoritmo sLAO* executa duas fases e Fact-ILAO* executa três fases. Sendo ILAO* uma versão melhorada do LAO*, nossa versão aproveita as vantagens desse algoritmo, enquanto o sLAO* realiza a atualização de custos usando um algoritmo de programação dinâmica (modificação do algoritmo SPUDD). Além disso, sLAO* precisa construir estados abstratos e atualizar esses conjuntos em cada iteração, o que é uma operação computacionalmente muito custosa, limitando seu desempenho. Esse fato constitui uma diferença importante com nosso algoritmo. Fact-ILAO* atualiza os estados individualmente e usa a ideia de atualização em pós-ordem. A expansão da solução atual realizada no sLAO* usa análise de alcançabilidade para determinar que estados processar. Essa operação é análoga a expandir os nós da borda do hipergrafo solução parcial. Uma outra diferença importante no trabalho de Feng (Feng e Hansen, 2002) é que o problema é definido para um conjunto de estados iniciais S_0, enquanto ILAO* e Fact-ILAO* trabalham com um só estado inicial $\mathbf{x_0}$.

5.5 Fact-ILAO* vs. Fact-LRTDP

O algoritmo Fact-LRTDP é um algoritmo fatorado baseado em atualização de estados por amostragem (trial). Assim, ele não mantém um hipergrafo solução parcial mas somente o conjunto de todos os estados visitados e seus valores, o que corresponde ao conjunto H_{DD}. Fact-LRTDP usa o procedimento CHECK-SOLVED (Bonet e Geffner, 2003) para determinar estados que não precisam ser visitados novamente por já terem convergido para o valor ótimo enquanto Fact-ILAO*

continua atualizando os estados do hipergrafo solução, mesmo que eles tenham convergido. Com relação à atualização de custos, ambos os algoritmos coincidem em fazer a atualização de estados individualmente. Porém, a ordem dessas atualizações é diferente: Fact-LRTDP atualiza estados amostrados nos trials enquanto que o Fact-ILAO* atualiza os estados do hipergrafo solução parcial em pós-ordem com relação a busca em profundidade em G'.

Capítulo 6

Análise experimental

Nesse capítulo fazemos uma análise comparativa dos algoritmos Fact-ILAO* e ILAO* (que chamaremos de Enum-ILAO* para diferenciá-la da versão fatorada) com duas implementações do algoritmo LRTDP: o GLUTTON (Kolobov *et al.*, 2012) e Fact-LRTDP (Gamarra *et al.*, 2012).

6.1 Planejamento *online*

Apesar dos algoritmo descritos nos capítulos 3, 4 e 5 serem de tipo *offline*, isto é, o objetivo é encontrar a política ótima (com ϵ-convergência) antes da executá-la no ambiente real. A política obtida dificilmente pode ser modificada e qualquer mudança exige uma re-computação da política. Por outro lado, no planejamento *online* a política não precisa ser computada a priori, ser completa ou ótima. Somente precisamos focar na escolha da ação a ser executada no estado atual. No próximo estado devemos escolher novamente a melhor ação para executar e possivelmente usaremos informação de computações anteriores (Vianna *et al.*, 2012).

No ambiente real, as competições internacionais de planejamento probabilístico (IPPC-2011) propõem problemas de planejamento *online* em que são propostos cerca de 80 problemas para serem resolvidos em 24 horas. Para cada problema um planejador *online* deve executar 40 ações da política encontrada. Essa execução deve ser repetida por 30 vezes para se computar a média da recompensa acumulada das 40 ações executadas. Cada problema ou domínio da competição possui 10 instâncias que são numeradas de 1 a 10, cujo tamanho e/ou dificuldade cresce com o número da instância. A competição define um horizonte finito de 40 passos para todas as instâncias. Na competição de planejamento foi adotado um ambiente *online* baseado numa arquitetura cliente-servidor. O servidor envia para o planejador o estado inicial de uma instância de problema e o planejador envia para o servidor uma ação para esse estado. Logo, o servidor executa a ação recebida no estado atual e envia um estado sucessor (usando um simulador do ambiente real) para o planejador e assim por diante. Depois de 40 interações desse tipo, o servidor envia um novo estado inicial e o processo se repete.

6.2 Ambiente RDDL.sim e domínios de planejamento

Na Competição Internacional de Planejamento Probabilístico (IPPC-2011) (Sanner *et al.*, 2012) foram introduzidos novos domínios de planejamento para MDPs. Algumas das caraterísticas desses problemas envolvem horizonte finito, grandes fatores de ramificação (isto é, MDPs com função de transição muito densa), custo de ações não uniformes e, em alguns casos, ausência de estados meta (sendo o objetivo minimizar o custo acumulado em 40 ações executadas). Assim, os problemas apresentaram cenários de planejamento mais realísticos e representaram desafios para os planejadores conhecidos até então. A seguir, descrevemos algumas das caraterísticas da linguagem RDDL usada no IPPC-2011, para descrever problemas de planejamento probabilístico.

6.2.1 Linguagem RDDL

A Linguagem de Diagramas de Influência Dinâmica Relacional (*Relational Dynamic Influence Diagram Language - RDDL*) foi usada na modelagem dos domínios da IPPC-2011. É baseado nas linguagens PPDDL 1.0 e PDDL 3.0. No entanto, é diferente tanto na sintaxe quanto na semântica. Introduz a representação de observabilidade parcial e eventos exógenos e melhora a descrição de MDPs e POMDPs representando estados, ações e observações como variáveis parametrizadas (fluentes ou não-fluentes). As variáveis parametrizadas são modelos das variáveis *ground* que podem ser obtidas quando a instância de um problema define os possíveis objetos do domínio (Sanner, 2010).

RDDL tem sido desenvolvido para modelar problemas de planejamento que usam variáveis booleanas, multi-valoradas, inteiras ou contínuas, concorrência não-restrita, não-fluentes, independência probabilística entre efeitos complexos (necessário para representar eventos exógenos) e observabilidade parcial. Além disso, RDDL permite usar expressões lógicas, aritméticas, de comparação, condicionais na definição das funções de transição e recompensa.

6.2.2 Ambiente RDDL.sim

RDDL.sim é um ambiente de simulação adotado pela IPPC-2011. Trata-se de um projeto desenvolvido pela NIC-Australia para fornecer um ambiente geral de teste e comparação de algoritmos que resolvem os MDPs da competição. Algumas das caraterísticas do projeto são:

- Especificação de padrões léxicos e gramática BNF.
- Tradução de RDDL para um formato prefixo de fácil análise sintática.
- Tradutores de RDDL para PPDDL-SPUDD e Symbolic Perseus.
- Um simulador Java para RDDL.
- Interface cliente-servidor em Java.
- Ferramenta de visualização Java para facilitar a depuração e teste.

A Figura 6.1 mostra a janela gerada pelo visualizador do RDDL.sim para o domínio **Navigation**.

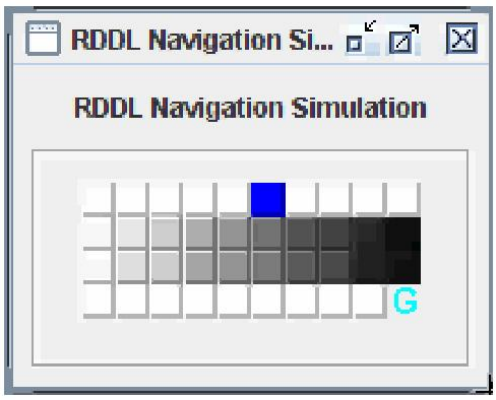

Figura 6.1: *Ilustração do domínio* Navigation *gerada pelo RDDL.sim. A letra* ***G*** *indica o estado objetivo. A variação de cor em tons de cinza representa a probabilidade do agente de desaparecer. Um tom oscuro indica uma grande probabilidade.*

Domínio	Propriedades	CTPs
SysAdmin	Altamente exógeno, transições complexas	Denso
Elevators	Altamente exógeno, concorrente	Denso
Traffic	Altamente exógenos, concorrente	Denso
Game of Life	Altamente combinatório	Denso
Navigation	Orientado a objetivo	Esparso
Recon	Poucos eventos exógenos	Esparso
Crossing Traffic	Orientado a objetivo	Esparso
Skill Teaching	Poucos eventos exógenos	Esparso

Tabela 6.1: *Domínios de planejamento probabilístico IPPC-2011 e suas caraterísticas principais.*

6.2.3 Domínios de teste

A competição IPPC-2011 definiu 8 domínios de planejamento. Cada um deles apresenta suas próprias caraterísticas relacionadas à concorrência, eventos exógenos, nível de explosão combinatória, etc. A Tabela 6.1 mostra um resumo das principais caraterísticas desses domínios. A seguir, detalhamos alguns dos domínios em que focamos nossos testes e análises empíricas.

- **SysAdmin:** Neste domínio, existem n computadores (c_1,...,c_n) conectados de acordo com alguma das seguintes topologias: anel unidirecional, anel bidirecional, anéis bidirecionais de pares de computadores e estrela. Em cada estado, um computador pode estar funcionando (ou não) e em cada estágio, um computador pode ser reiniciado, fazendo com que ele esteja funcionando no próximo estado. Um computador que não foi reiniciado possui uma probabilidade de estar funcionando no próximo estágio condicionada a seu estado atual e ao número de computadores atualmente funcionado. Uma política ótima tentará reiniciar o computador que tem o maior impacto na recompensa descontada esperada num horizonte infinito, sendo que a recompensa do estado é o número de computadores que estão funcionado.

- **Navigation:** Numa grade, um robô deve atingir uma posição meta. Cada posição tem associada uma probabilidade de o robô desaparecer. Assim, ele precisa escolher um caminho que o leve até a meta com a maior probabilidade possível.

- **Recon:** Numa grade, encontra-se o acampamento base do agente, algumas posições perigosas e objetos localizados em diferentes pontos. O agente possui três ferramentas, uma para detectar água, uma para detectar vida e uma outra para tirar fotos. Seus movimentos são determinísticos porém, a probabilidade de obter uma boa leitura dos sensores de água e vida é estocástica. Se o agente estiver numa posição perigosa ou numa posição adjacente de uma perigosa, existe uma probabilidade de danificar as suas ferramentas. Isso pode provocar que os objetos amostrados sejam contaminados, isto é, a ferramenta não detecta a presença de água ou vida no objeto. O agente pode retornar à base para consertar suas ferramentas. Tirar fotos de objetos onde vida foi detectada produz uma recompensa positiva. Por outro lado, uma recompensa negativa é dada quando vida não for detectada.

- **Elevators:** Certo número de elevadores pode estar no térreo ou no último andar de um prédio. Os passageiros chegam nos andares baseados na distribuição de Bernoulli, com diferentes probabilidades de chegada a cada andar. Os elevadores podem se mover dependendo de certas condições. Se a porta estiver fechada, o elevador mantém sua direção atual. Por outro lado, o elevador pode permanecer estacionário ou pode abrir sua porta e indicando a direção que tomará a seguir. O elevador somente pode mudar sua direção abrindo sua porta e indicando a direção oposta. Um plano ótimo tentará fornecer um elevador perto dos andares em que os passageiros possivelmente chegarão e coordenar eficientemente vários elevadores para subir e descer passageiros.

- **Crossing Traffic:** Numa grade, um robô deve alcançar uma posição objetivo e evitar os obstáculos que chegam aleatoriamente e na direção da esquerda. Quando um desses obstáculos atingir o robô, ele desaparece e não pode mover-se. O robô recebe -1 cada vez que não alcança a meta. O estado meta é absorvente com recompensa 0.

- **Skill Teaching:** O agente está tentando ensinar um conjunto de habilidades a estudantes através de dicas e perguntas de múltipla escolha. O agente pode introduzir num novo tópico dando ao estudante um teste de proficiência ou ajudando-o com o tópico anterior. A probabilidade do estudante de obter bons resultados depende do seu nível atual e da quantidade de ajuda que recebeu do agente. O objetivo é planejar as lições tal que o estudante aprenda o melhor possível e exista tempo para verificar seu conhecimento.

- **Game of Life:** Temos um autômato celular numa grade. A recompensa é obtida gerando padrões que mantem a maior quantidade de células vivas.

6.3 Sistemas de planejamento probabilístico usados na análise

Nesta seção, descrevemos os algoritmos que foram usados nos testes experimentais do presente trabalho. Mencionamos alguns dos detalhes de implementação e caraterísticas principais.

6.3.1 GLUTTON

O planejador GLUTTON, um dos ganhadores da IPPC-2011, realiza planejamento *offline* e é implementado em C++. O algoritmo principal usado pelo planejador chama-se LR^{2}TDP, apresentado na (Seção 3.4). Além disso, o planejador GLUTTON introduz algumas otimizações:

- Sub-amostragem da função de transição, isto é, realiza uma amostragem do conjunto de estados sucessores para reduzir o custo computacional da atualização de Bellman.

- Usa políticas específicas para cada tipo de efeitos das ações.

- Uso de políticas pre-determinadas, isto é, políticas escolhidas sob algum critério.

A implementação do Glutton usada nos experimentos foi a mesma usada na competição IPPC-11, cujos resultados estão disponíveis em *http://users.cecs.anu.edu.au/∼ssanner/IPPC-2011/*. Neste trabalho não implementamos esse algoritmo, mas sim, usamos seus resultados para comparar com os nossos. O GLUTTON não utiliza a biblioteca de ADDs do RDDL.sim.

6.3.2 Fact-LRTDP

O algoritmo Fact-LRTDP (Seção 4.5.3) foi implementado na linguagem Java e utilizando o ambiente (e a biblioteca de ADDs) do RDDL.sim. Recebe como entrada a representação fatorada de um MDP e o ADD que representa a função valor é inicializado com um valor heurístico admissível. Fact-LRTDP realiza os trails para atualizar o valor dos estados no tempo disponível para devolver uma ação ou até atingir um dado horizonte.

6.3.3 Enum-ILAO* e Fact-ILAO*

Dado que os algoritmos baseados no ILAO* computam uma política de horizonte infinito e o simulador RDDL.sim define um ambiente *online* de horizonte finito, adaptamos nossas implementações para operar nesse ambiente.

- **ILAO* Enumerativo:** Implementamos o algoritmo ILAO* representando explicitamente os estados e chamamos essa versão de *Enum-ILAO**. O algoritmo Enum-ILAO* foi implementado em Java e usa o ambiente da competição. Para ser rodado no ambiente *online*, modificamos

nossa implementação permitindo ao algoritmo rodar no tempo disponível para devolver uma ação. Assim, o teste de convergência não foi usado como critério de parada mas sim o tempo disponível. Dessa maneira, as três fases do algoritmo são repetidas até esse tempo acabar. Usamos listas de valores booleanos para representar explicitamente os estados. A função valor é representada usando uma tabela hash que mapeia estados em valores reais. A política gulosa é representada usando outra tabela hash que associa estados a nomes de ações. A computação de estados sucessores, dada uma ação e um estado, gera uma lista de estados. Os valores da função valor inicial são inicializados com o valor zero, isto é, a função heurística zero que é admissível.

- **ILAO* Fatorado:** Implementamos a nossa versão fatorada do ILAO* usando a linguagem Java e a biblioteca de ADDs do simulador. Como na versão enumerativa, modificamos o algoritmo *offline* para rodar no entorno *online*. As mudanças são praticamente as mesmas da versão *online* enumerativa. O algoritmo roda até o tempo disponível acabar e devolve uma ação. O teste de convergência também não foi usado como critério de parada. Como foi mencionado no Capítulo 5, fazemos uso de uma tabela hash para armazenar a política gulosa e assim representar o hipergrafo solução. A geração de estados sucessores cria um novo BDD que representa os estados sucessores. Para inicializar o ADD que representa a função valor, calculamos a diferença entre a recompensa máxima e a recompensa mínima fornecida por cada ação e selecionamos a maior destas diferenças. Dessa maneira, esse valor é usado como valor heurístico dos estados no inicio da execução.

6.4 Resultados Experimentais

Nesta seção, descreveremos os resultados das implementações dos algoritmos Enum-ILAO* e Fact-ILAO* e que foram testados usando os problemas da competição apresentados na Seção 6.2.3. Os parâmetros seguidos nos experimentos foram os mesmos definidos na competição. Com relação ao tempo disponível, distribuímos o tempo de uma simulação nos 40 passos do horizonte e quando o tempo de cada passo acabar, determinamos a melhor ação a executar para o estado inicial. Definimos uma distribuição exponencial do tempo, tal que nos primeiros passos o tempo fosse maior e nos últimos menor. A distribuição do tempo é definida como, $tempo = base^{horizonteRestante}/3$, em que $base = 1.04648$.

Usamos uma instância do *Amazon Elastic Compute Cloud* (*Amazon EC2*) com 4 núcleos virtuais sobre dois núcleos físicos e 7.5 GB de RAM para rodar os algoritmos e obter resultados comparáveis com os algoritmos dos competidores da IPPC-2011, isto é, os competidores também rodaram seus algoritmos usando essa mesma plataforma e configuração. Dada a complexidade das transições entre estados, definimos dois grupos de domínios: os domínios densos (SysAdmin, Traffic, Elevators e Game of Life) e os domínios esparsos (Navigation, Recon, Crossing Traffic e Skill Teaching).

6.4.1 Comparação entre Fact-ILAO* e Enum-ILAO*

Domínios densos. Para domínios com matrizes de transição muito densas Fact-ILAO* obteve um bom desempenho. No SysAdmin (Figura 6.2), os resultados mostram que nossa versão fatorada consegue resolver instâncias que a versão enumerativa não consegue. A saber, o Enum-ILAO* somente resolveu as primeiras duas instâncias sendo incapaz de resolver as maiores, enquanto que Fact-ILAO* obteve bons resultados para todas as instâncias maiores. No domínio do Traffic (Figura 6.3), observamos que na maioria das instâncias (1,2,4,9,10) Fact-ILAO* supera Enum-ILAO*. Nas instâncias 5, 6 e 7, os resultados são quase os mesmos.

No domínio Elevators (Figura 6.4), observamos que Enum-ILAO* supera o Fact-ILAO* em algumas instâncias, por exemplo, 2, 5 e 6. Nos outros casos, os resultados são muito similares entre

si. Finalmente, no domínio GAME OF LIFE (Figura 6.5) os resultados são muito próximos entre os dois algoritmos. Não podemos determinar uma clara superioridade de um sobre o outro. Nas instâncias 5 e 6 o Enum-ILAO* supera o Fact-ILAO* e nas instâncias 7 e 9, o Fact-ILAO* supera o Enum-ILAO*.

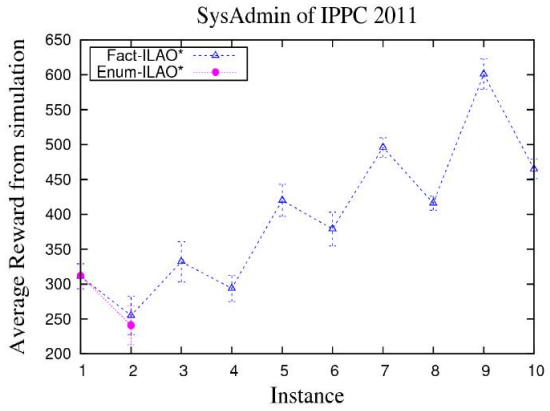

Figura 6.2: *Domínio do* SYSADMIN

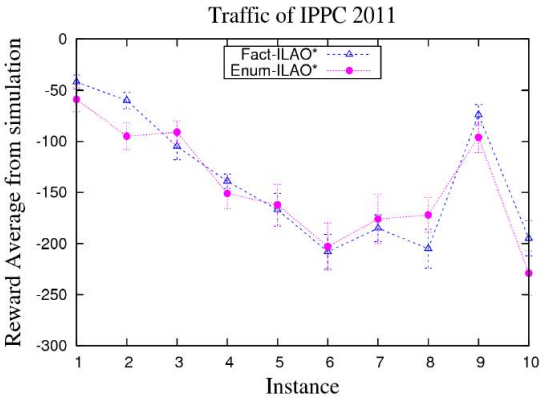

Figura 6.3: *Domínio do* TRAFFIC

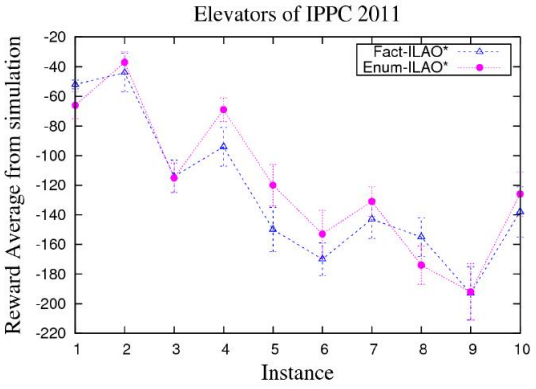

Figura 6.4: *Domínio do* ELEVATORS

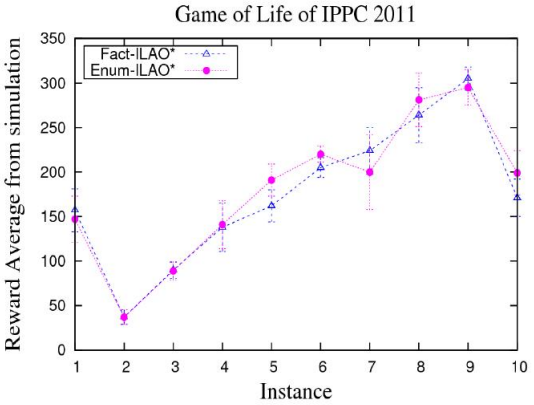

Figura 6.5: *Domínio do* GAME OF LIFE

Domínios esparsos. Nos domínios esparsos da competição, observamos que o Fact-ILAO* pode ser similar ou um pouco pior que Enum-ILAO*. No NAVIGATION (Figura 6.6) os resultados são muito próximos entre si. Somente na instância 4 o Fact-ILAO* supera significativamente o Enum-ILAO*. No domínio RECON (Figura 6.7) observamos que Enum-ILAO* supera o Fact-ILAO* na maioria dos casos. Na instância 8, o Fact-ILAO* teve muito melhor resultados que Enum-ILAO*, mas em geral Enum-ILAO* foi superior. Para o domínio do CROSSING TRAFFIC (Figura 6.8), resultados foram muito similares. Ambos os algoritmos obtevem melhores resultados nas instâncias 1 e 2 e depois não conseguiram obter uma boa recompensa nas instâncias maiores. A Figura 6.9 ilustra os resultados do domínio SKILL TEACHING. Observamos que o algoritmo Enum-ILAO* obteve melhores resultados em quase todas as instâncias. É importante notar que nossa versão enumerativa Enum-ILAO* supera tanto o Fact-ILAO* quanto Fact-LRTDP nas últimas instâncias.

Podemos concluir que o algoritmo Fact-ILAO* com sua representação fatorada pode obter soluções escaláveis. O fato de conseguir resolver instâncias do SYSADMIN que a versão enumerativa não consegue mostra que a representação fatorada é útil nesse tipo de problemas (com matrizes de transição densas e muita independência entre as variáveis de estado, tanto na função de transição de estados como na função de custo). Por outro lado, quando o domínio não exige uma representação compacta (domínios esparsos), a versão enumerativa é suficiente e, em alguns casos, superior que a versão fatorada.

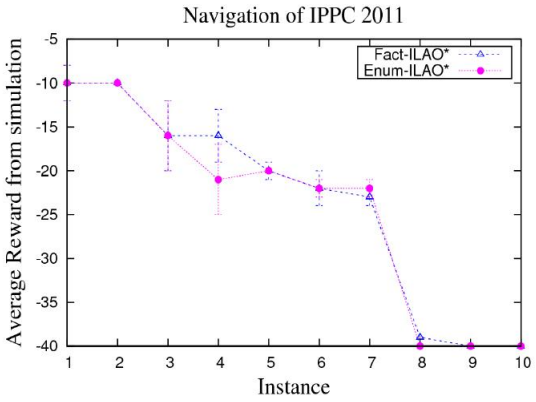

Figura 6.6: *Domínio do* NAVIGATION

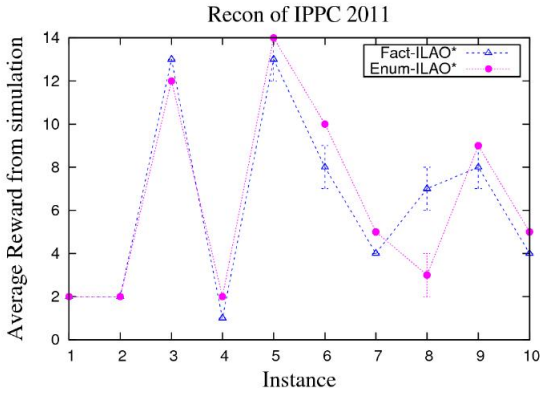

Figura 6.7: *Domínio do* RECON

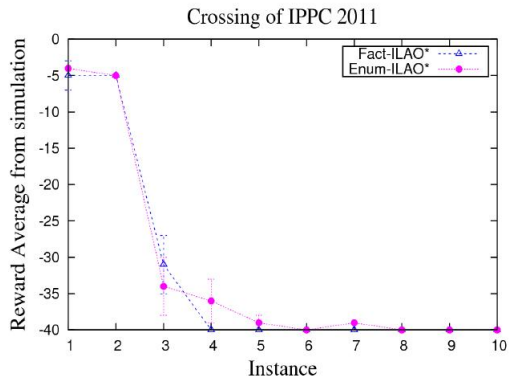

Figura 6.8: *Domínio do* CROSSING TRAFFIC

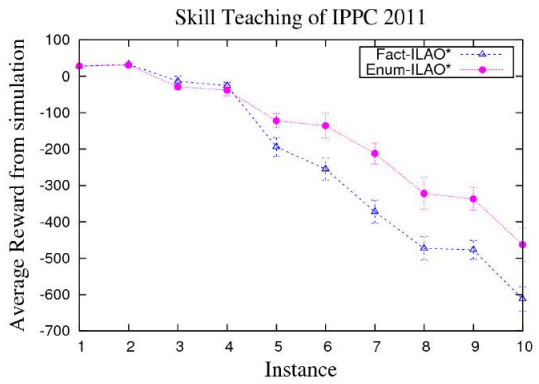

Figura 6.9: *Domínio do* SKILL TEACHING

6.4.2 Comparação entre Fact-ILAO* e Fact-LRTDP

Domínios densos. A Figura 6.10 mostra os resultados para o domínio do SYSADMIN. Notamos que os resultados são quase os mesmos, somente nas primeiras instâncias (1,2,3) Fact-LRTDP supera a recompensa obtida pelo Fact-ILAO* mas, em geral, ambos são competitivos nas instâncias

superiores. No domínio do TRAFFIC (Figura 6.11), vemos que o Fact-ILAO* supera o Fact-LRTDP em quase todas as instâncias. A maior diferença encontra-se na instância 7. Assim, nesse domínio afirmamos que o Fact-ILAO* possui um melhor desempenho. Para o domínio ELEVATORS, mostrado na Figura 6.12, vemos que em alguns casos Fact-ILAO* supera o Fact-LRTDP (instâncias 2,3,7,8 e 10) e em outros, o Fact-LRTDP obteve melhores resultados (instâncias 2,3,7,8 e 10). No entanto, não podemos afirmar uma superioridade absoluta de um algoritmo sobre outro. Finalmente, no GAME OF LIFE (Figura 6.13), vemos que Fact-LRTDP tem melhores resultados para as instâncias menores (1,2,3), sendo que para as outras instâncias os resultados são muito similares. O Fact-ILAO* supera Fact-LRTDP somente nas instâncias 4 e 7.

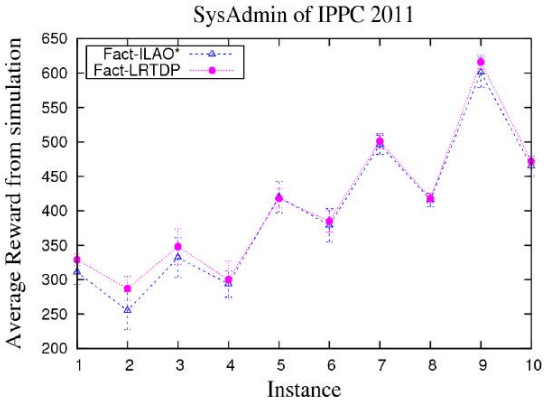

Figura 6.10: *Domínio do* SYSADMIN

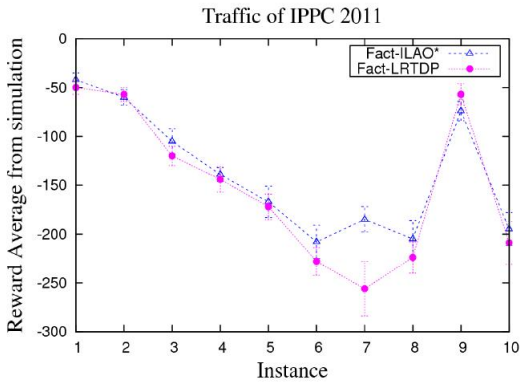

Figura 6.11: *Domínio do* TRAFFIC

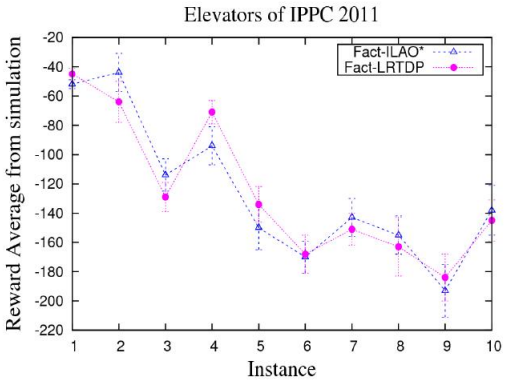

Figura 6.12: *Domínio do* ELEVATORS

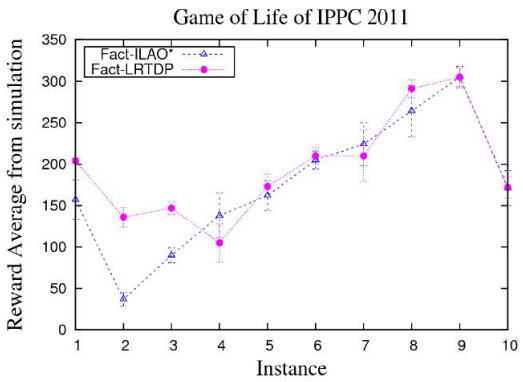

Figura 6.13: *Domínio do* GAME OF LIFE

Domínios esparsos. A Figura 6.14 apresenta os resultados do domínio NAVIGATION. Nesse caso, os resultados são praticamente os mesmos, não encontramos uma ampla diferença entre os dois algoritmos, exceto na instância 4, em que Fact-ILAO* supera o Fact-LRTDP. Nas outras instâncias, as diferenças não são significativas. No domínio RECON (Figura 6.15), o Fact-LRTDP supera o Fact-ILAO* em quase todas as instâncias. Somente na segunda instância Fact-ILAO* alcançou um melhor resultado. No entanto, os resultados obtidos pelo Fact-LRTDP não excedem muito os resultados do Fact-ILAO*. Na Figura 6.16, mostramos os resultados para o domínio CROSSING TRAFFIC. Observamos que os resultados são muito similares, exceto nas instâncias 3 e 4, em que Fact-LRTDP obteve melhores resultados do que Fact-ILAO* (nas instâncias de 5 a 10, os dois planejadores não encontram solução). Para concluir, no domínio SKILL TEACHING (Figura 6.17), observamos que não existe uma clara superioridade de um algoritmo sobre o outro. Os resultados coincidem na maioria das instâncias, sendo que Fact-LRTDP supera o Fact-ILAO* somente em duas instâncias, a saber, 6 e 9.

Em resumo, consideramos que ambas versões fatoradas possuem desempenhos similares, porém Fact-ILAO* se mostrou um pouco superior para os domínios densos. Sendo que ambos usam ADDs e BDDs e fazem a atualização da função valor para estados individuais, a única diferença importante entre eles é a maneira como são feitas as atualizações: Fact-ILAO* atualiza todos os estados sucessores da ação gulosa em cada iteração, no entanto o Fact-LRTDP sorteia apenas um estado sucessor para ser atualizado em cada trial. Sendo os domínios densos mais complexos e portanto é crucial fazer a atualização de estados relevantes, Fact-ILAO* mostrou vantagem sobre Fact-LRTDP para esses domínios.

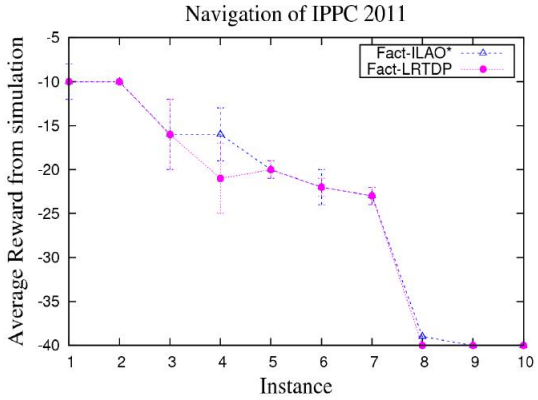

Figura 6.14: *Domínio do* NAVIGATION

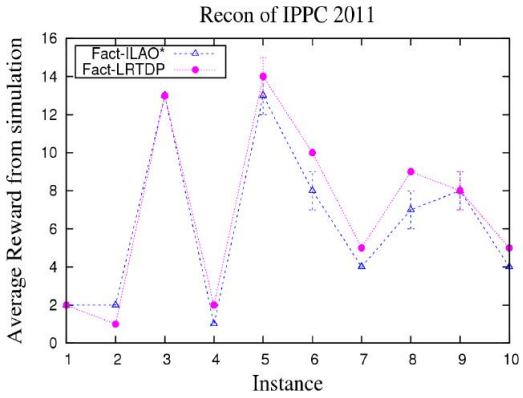

Figura 6.15: *Domínio do* RECON

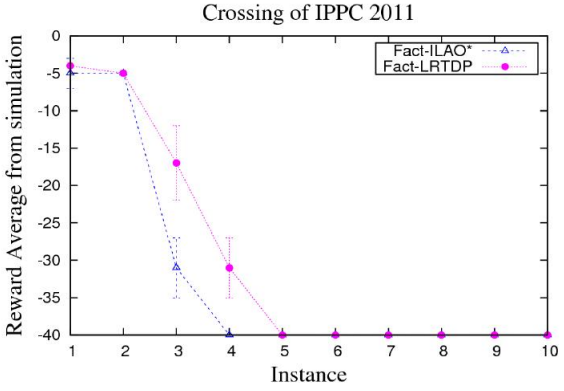

Figura 6.16: *Domínio do* CROSSING TRAFFIC

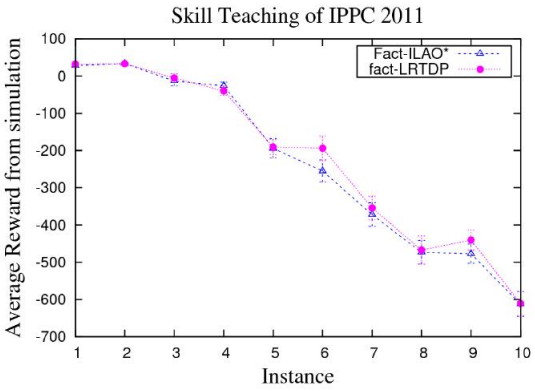

Figura 6.17: *Domínio do* SKILL TEACHING

6.4.3 Comparação entre Fact-ILAO*, Fact-LRTDP e o GLUTTON

Nessa seção, discutiremos os resultados obtidos pelo Fact-ILAO* quando comparado com os resultados do Fact-LRTDP e do GLUTTON. A Figura 6.18 apresenta os resultados do domínio SysAdmin, observamos que Fact-ILAO* somente é competitivo com o algoritmo GLUTTON (segundo lugar na IPPC-2011) nas instâncias maiores (6,7,8,9,10). Nas figuras 6.19, 6.20 e 6.21 mostramos os resultados do domínio Traffic, Elevators e Game of Life, respectivamente. Nesses três casos, o GLUTTON supera claramente os resultados Fact-ILAO* e do Fact-LRTDP. Em alguns casos isolados Fact-ILAO* obteve um desempenho aceitável, por exemplo, na instância 9 do Traffic, na instância 1 do Elevators e na instância 9 do Game of Life.

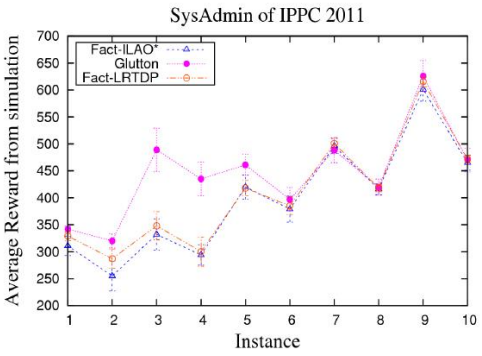

Figura 6.18: *Domínio do* SysAdmin

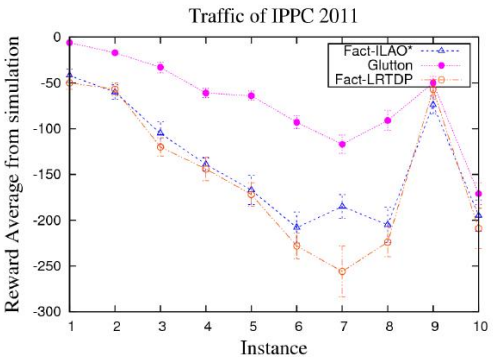

Figura 6.19: *Domínio do* Traffic

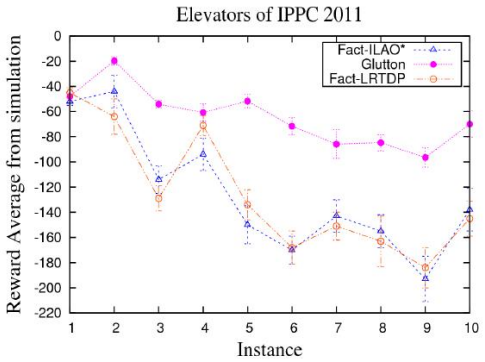

Figura 6.20: *Domínio do* Elevators

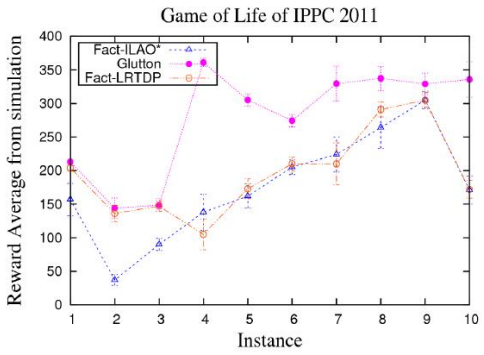

Figura 6.21: *Domínio do* Game of Life

Para os domínios esparsos, temos que o Fact-ILAO* apresenta um melhor desempenho no domínio NAVIGATION (Figura 6.22) tendo os mesmos resultados que o GLUTTON nas primeiras 7 instâncias. No caso da instância 4, a vantagem para Fact-ILAO* é maior. No entanto, o GLUTTON supera nosso algoritmo nas instâncias 8, 9 e 10. O Fact-LRTDP segue o comportamento de Fact-ILAO* sendo melhor que GLUTTON nesse domínio. A Figura 6.23 ilustra os resultados para o domínio RECON. Vemos que o Fact-ILAO* supera o GLUTTON na maioria de instâncias. No entanto, nas instâncias 7 e 8, ambos os algoritmos tem os mesmos resultados e somente na instância 10 o GLUTTON supera o Fact-ILAO*. No caso do domínio CROSSING TRAFFIC (Figura 6.24), o Fact-ILAO* obteve melhores resultados somente nas primeiras duas instâncias. Nas outras, GLUTTON supera claramente os outros dois algoritmos. No caso do SKILL TEACHING (Figura 6.25), o Fact-ILAO* e Fact-LRTDP foram competitivos somente nas primeiras instâncias (1 e 2), nas instâncias maiores o GLUTTON teve melhores resultados. Por serem domínios esparsos, o GLUTTON os resolve completamente ou explora seus espaços de estados mais intensamente.

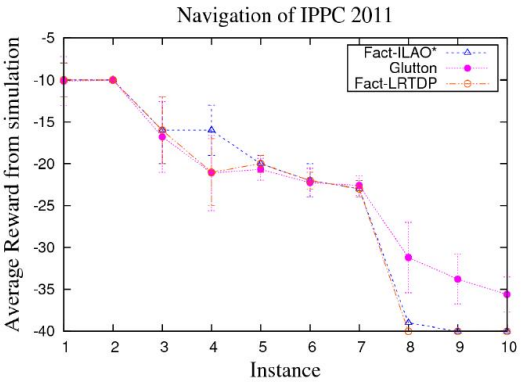

Figura 6.22: *Domínio do* NAVIGATION

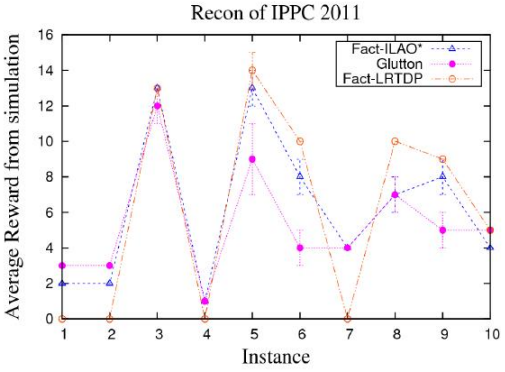

Figura 6.23: *Domínio do* RECON

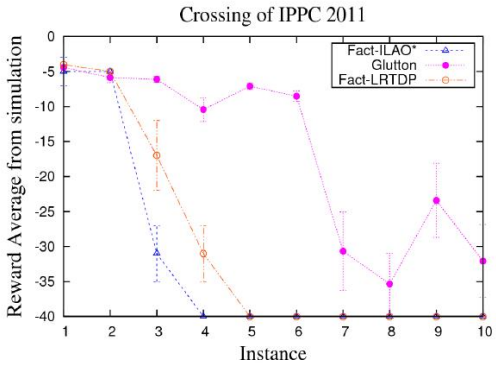

Figura 6.24: *Domínio do* CROSSING TRAFFIC

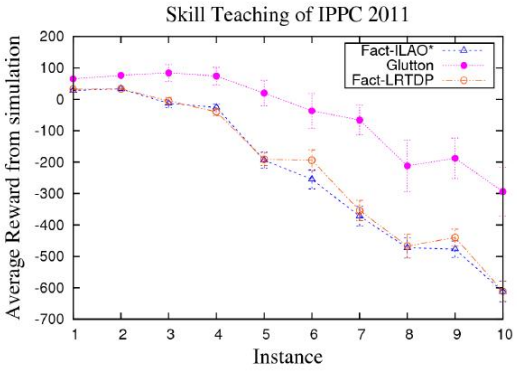

Figura 6.25: *Domínio do* SKILL TEACHING

É importante observar que, Fact-ILAO* e Fact-LRTDP não usam modificações ou otimizações que dependam do domínio para obter um melhor desempenho. Os resultados dessa seção mostram que a representação fatorada e operações sobre ADDs e BDDs podem obter um bom desempenho sem depender de outras melhorias ou truques usadas pelo sistema GLUTTON (Seção 6.3.1).

Capítulo 7

Conclusões

7.1 Considerações Finais

Resolver um problema de planejamento probabilístico modelado como um MDP, em que o espaço de estados cresce exponencialmente com o número de variáveis de estado, é uma tarefa difícil. Neste trabalho, propomos um novo algoritmo eficiente de planejamento probabilístico: uma versão fatorada do algoritmo de busca heurística ILAO*, chamada de Fact-ILAO*. Mostramos que Fact-ILAO* pode apresentar desempenho competitivo quando comparado com outras abordagens, particularmente os algoritmos baseados no RTDP. Os resultados empíricos mostram que o algoritmo Fact-ILAO* tem melhor desempenho em algumas instâncias dos domínios da competição IPPC-2011, sobretudo nas instâncias maiores de domínios densos, que possuem uma estrutura fatorada com independências entre as variáveis de estado.

Nossos resultados empíricos também mostraram que os algoritmos Fact-ILAO* e Fact-LRTDP, não apresentam uma superioridade absoluta entre si. Porém, Fact-ILAO* se mostra um pouco superior nos domínios densos. Ambos usam estimativas heurísticas admissíveis, o conhecimento do estado inicial e fazem atualizações individuais de estados. Esses algoritmos diferem basicamente, na ordem de exploração do espaço de estados e a ordem de atualização de estados, sendo essa a única explicação para o melhor desempenho em domínios densos.

7.2 Sugestões para Trabalhos Futuros

A seguir, mencionamos alguns tópicos que podem ser explorados como trabalhos futuros:

- Desenvolvimento de heurísticas: pesquisar novas heurísticas para definir a função valor inicial $V^0(s)$.
- Horizonte finito: modificar o Fact-ILAO* para trabalhar com horizonte finito. Para isso, uma solução similar ao LR^2TDP usada pelo algoritmo GLUTTON pode ser implementada.
- Testar outros domínios: analizar o desempenho dos algoritmos com outros domínios, pode fornecer novos resultados e novas perspectivas para o aprimoramento ou extensão do algoritmo proposto.
- Usar técnicas de otimizações usadas pelos planejadores da competição IPPC-2011, GLUTTON e PROST, em nossa versão fatorada do ILAO* para alcançar melhores resultados.

Referências Bibliográficas

Akers(1978) B. Akers. Binary Decision Diagramas. *IEEE Transactions*, 100(6):509–516. Citado na pág. 50

Ausiello e Giaccio(1997) G. Ausiello e R. Giaccio. On-line algorithms for satisfiability problems with uncertainty. *Theoretical Computer Science*, 171(1):3–24. Citado na pág. 10

Bahar *et al.***(1993)** R. I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo e F. Somenzi. Algebraic Decision Diagrams and their Applications. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, páginas 188–191. Citado na pág. 4, 49, 50

Barto *et al.***(1995)** Andrew G. Barto, Steve J. Bradtke e Satinder P. Singh. Learning to act using Real-Time Dynamic Programming. *Artificial Intelligence*, 72:81–138. Citado na pág. 3, 5, 30, 36, 47

Bellman(1958) R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1): 87–90. Citado na pág. 8

Bellman(1957) R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1º edição. Citado na pág. 3, 32

Berge(1980) C. Berge. *Hypergraphs: Combinatorics of finite sets*. Elsevier Science Publishing, New York, NY, 1º edição. Citado na pág. 10

Bertsekas(1995) D. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 1º edição. Citado na pág. 33

Bertsekas e Tsitsiklis(1991) D.P. Bertsekas e J.N. Tsitsiklis. An analysis of Stochastic Shortest Path Problem. *Mathematics of Operations Research*, 16(3):580–595. Citado na pág. 4

Bhuma e Goldsmith(2003) V.D.K. Bhuma e J. Goldsmith. Bidirectional LAO* Algorithm. *Proceeding of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 980–992. Citado na pág. 4, 45, 47

Bonet e Geffner(2012) B. Bonet e H. Geffner. Action Selection for MDP: Anytime AO* vs. UCT. *Proceeding of the Association for the Advancement of Artificial Intelligence (AAAI)*. Citado na pág. 47

Bonet e Geffner(2003) Blai Bonet e Hector Geffner. Labeled RTDP: Improving the convergence of Real-Time Dynamic Programming. *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, páginas 12–21. Citado na pág. 3, 5, 33, 36, 37, 57, 64

Boutilier *et al.***(1999)** C. Boutilier, S. Hanks e T. Dean. Decision-theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research (JAIR)*, 11:1–94. Citado na pág. 31

Bryant *et al.***(1986)** R. E. Bryant, R. L. Rudell e K. S. Brace. Graph-based algorithm for boolean function manipulation. *IEEE Transactions*, 100(8):677–691. Citado na pág. 4, 50, 51

Bullers *et al.*(**1980**) W. Bullers, S. Nof e A. Whinston. Artificial Intelligence in Manufacturing Planning and Control. *AIIE Transactions*, 12(4):351–363. Citado na pág. 9

Dai e Goldsmith(2006) P. Dai e J. Goldsmith. LAO*, RLAO*, or BLAO*. *In AAAI Workshop on Heuristic Search*, 1:59–64. Citado na pág. 4, 45

Dai e Goldsmith(2007) P. Dai e J. Goldsmith. Multi-threaded BLAO* Algorithm. *Proceeding of the Twenty International FLAIRS Conference*, páginas 56–62. Citado na pág. 4, 5, 46

Dai *et al.*(**2011**) P. Dai, Mausam, J. Goldsmith e D. Weld. Topological Value Iteration Algorithms. *Journal of Artificial Intelligence Research*, 42(1):181–209. Citado na pág. 4, 48

Dean e Kanazawa(1990) T. Dean e K. Kanazawa. A Model for Reasoning about Persistence and Causation. *Computational Intelligence*, 5(2):142–150. Citado na pág. 4, 49

Delgado(2010) K.V. Delgado. *Processos de decisão Markovianos fatorados com probabilidades imprecisas*. Tese de Doutorado, Instituto de Matemática e Estatística - USP, São Paulo, Brasil. Citado na pág. xvi, 31, 37, 53, 54, 56

Dijkstra(1959) E.W. Dijkstra. A Note on Two Problem in Connection with Graphs. *Numerische Mathematik*, 1:269–271. Citado na pág. 8

Edelkamp e Schrodl(2011) E. Edelkamp e S. Schrodl. *Heuristic Search - Theory and Applications*. Elsevier Inc, Waltham, MA, 1º edição. Citado na pág. 14

Ertel(2011) W. Ertel. *Introduction to Intelligence Artificial*. Springer-Verlag, London, 1º edição. Citado na pág. xvi, 16

Fagin(1983) R. Fagin. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *Journal of the ACM*, 30:514–550. Citado na pág. 10

Feng e Hansen(2002) Z. Feng e E.A. Hansen. Symbolic LAO* Search for Factored Markov Decision Processes. *Proceedings of the Eighteenth National Conference on Artificial Intelligence, Alberta, Canada*, páginas 455–560. Citado na pág. 4, 54, 58, 64

Feng *et al.*(**2003**) Zhengzhu Feng, Eric A. Hansen e Shlomo Zilberstein. Symbolic Generalization for On-line Planning. *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI), Acapulco, Mexico*, páginas 209–216. Citado na pág. 4, 54, 56

Ford(1956) L.R. Ford. Network Flow Theory. Relatório técnico, The RAND Corperation, Santa Monica, California. Citado na pág. 8

Gallo *et al.*(**1996**) G. Gallo, C. Gentile e D. Pretolani. Max Horn SAT and Directed Hypergraphs: Algorithmic enhancements and easy cases (Extended Abstract). Relatório técnico, Dipartamento di Informatica, Universidà di Pisa, Italy, Pisa, Italy. Citado na pág. 10

Gamarra *et al.*(**2012**) M. Gamarra, K. Delgado e L. de Barros. Factored Labeled Real-Time Dynamic Programming. *Association for the Advancement of Artificial Intelligence*. Citado na pág. xvi, 4, 54, 57, 67

Geffner(2002) H. Geffner. Perspectives on Artificial Intelligence Planning. Relatório técnico, Universitat Pompeu Fabra, Barcelona, Spain. Citado na pág. 1

Ghallab *et al.*(**2004**) M. Ghallab, D.S. Nau e P. Traverso. *Automated Planning: Theory and Practice*. Elsevier Inc, 2º edição. Citado na pág. 1, 30, 35

Guestrin *et al.*(**2003**) C. Guestrin, D. Coller, R. Parr e S. Venkataraman. Efficient Solution Algorithm for Factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468. Citado na pág. 4

Guestrin(2003) Carlos Ernesto Guestrin. *Planning Under Uncertainty in Complex Structured Environments*. Tese de Doutorado, Stanford University, United States. Citado na pág. 49

Hansen e Zilberstein(2001) E. Hansen e S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1):35–62. Citado na pág. 4, 5, 40, 41, 42, 43, 47

Hansen e Zilberstein(1998) Eric A. Hansen e Shlomo Zilberstein. Heuristic Search in Cyclic AND/OR Graphs. *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, páginas 412–418. Citado na pág. 2, 4, 25

Hart *et al.***(1968)** P.E. Hart, N.J Nilsson e B. Raphael. A formal basis for heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107. Citado na pág. 1, 17

Haslum e Geffner(2000) P. Haslum e H. Geffner. Admissible heuristic for optimal planning. *Proceedings of the International Conference on AI Planning Systems*, páginas 140–149. Citado na pág. 1

Hoey *et al.***(1999)** Jesse Hoey, Robert St-aubin, Alan Hu e Craig Boutilier. SPUDD: Stochastic Planning using Decision Diagrams. *Proceeding of the Fifteenth Conference on Uncertain in Artificial Intelligence (UIA)*, páginas 279–288. Citado na pág. 4, 54, 55

Howard(1960) Ronald A. Howard. *Dynamic Programming and Markov Process*. The MIT Press, Cambridge, MA, 1º edição. Citado na pág. 3

Jimenez e Torras(2000) P. Jimenez e C. Torras. An efficient algorithm for searching implicit AND/OR graphs with cycles. *Artificial Intelligence*, 124(1):1–30. Citado na pág. 24

Keller e Eyerich(2012) T. Keller e P. Eyerich. PROST: Probabilistic Planning Based on UCT. *Proceedings of the of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS)*. Citado na pág. 39

Kocsis e Szepevári(2006) L. Kocsis e C. Szepevári. Bandit Based Monte-Carlo Planning. *Proceedings of the Seventeenth European Conference on Machine Learning (ECML)*, páginas 282–293. Citado na pág. 38

Kolobov *et al.***(2012)** A. Kolobov, P. Dai, Mausam e D. Weld. Reserve Iterative Deepening for Finite-Horizon MDPs with Large Branching Factors. *Proceeding of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS)*. Citado na pág. 5, 38, 67

Korf(1990) R.E. Korf. Real Time Heuristic Search. *Artificial Intelligence*, 2(3):189–211. Citado na pág. 36

Kristensen e Nielsen(2006) A. R. Kristensen e L. R. Nielsen. Finding the K-best policies in a finite-horizon Markov Decision Process. *European Journal of Operational Research*, 175(2): 1146–1179. Citado na pág. 10

Leffler(2009) B. Leffler. Perception-Based Generalization in Model-Based Reinforcement Learning. Dissertação de Mestrado, The State University of New Jersey, United States. Citado na pág. 29

Littman *et al.***(1995)** M.L. Littman, T.L. Dean e L.P. Kaelbling. On the complexity of Solving Markov Decision Problem. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI)*, páginas 394–402. Citado na pág. 35

Mahanti e Bagchi(1985) A. Mahanti e A. Bagchi. AND/OR Graph Heuristic Search Method. *Journal of the ACM*, 32(1):28–51. Citado na pág. 28

Martelli e Montanari(1978) A. Martelli e U. Montanari. Optimizing Decision Trees Through Heuristically Guided Search. *Communications of the ACM*, 21(2):1025–1039. Citado na pág. 4, 9

Martelli e Montanari(1973) A. Martelli e U. Montanari. Additive AND/OR Graphs. *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 40–55. Citado na pág. 21, 23

Mausam e Kolobov(2012) Mausam e A. Kolobov. *Planning with Markov Decision Processes, An AI Perspective*. Athena Scientific, Belmont, MA, 1º edição. Citado na pág. 34, 36, 49, 51, 53

McMahan *et al.***(2005)** H. McMahan, M. Likhachev e G. Gordon. Bounded Real-Time Dynamic Programming: RTDP with monotone upper bounds and performance guarantees. *Proceedings of the Twenty-second International Conference on Machine Learning*, páginas 569–576. Citado na pág. 3, 38

Nguyen *et al.***(2001)** S. Nguyen, S. Pallottino e F. Malucelli. A Modeling Framework for Passenger Assigment on a Transport Network with Timetables. *Transportation Science*, 35(3):238–249. Citado na pág. 10

Nilsson(1980) N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, 2º edição. Citado na pág. xv, 2, 8, 9, 17, 18, 19, 24, 27

Papadimitriou e Tsitsiklis(1987) C. Papadimitriou e J. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441–450. Citado na pág. 5

Pearl(1984) J. Pearl. *Heuristics, Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1º edição. Citado na pág. 22

Puterman(1994) M. Puterman. *Markov Desicion Processes-Discrete Stochastic Dynamic Programming*. John Wiley and Sons, New York, NY, 2º edição. Citado na pág. 3, 29, 31, 34

Russell e Norvig(2010) S. J. Russell e P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice-Hall, 3º edição. New Jersey. Citado na pág. xv, xvi, 1, 12, 14, 17, 20, 21

Sanner(2010) S. Sanner. Relational Dynamic Influence Diagram Language (RDDL): Language Description. Relatório técnico, NICTA, Camberra, Australia. Citado na pág. 68

Sanner *et al.***(2012)** S. Sanner, A. Coles, A. Garcia Olaya, S. Jimenez, C. Linares Lopez e S. Yoon. A Survey of the Seventh International Planning Competition. *AI Magazine*. Citado na pág. 67

Sanner(2008) Scott Sanner. *First-Order Decision-Theoretic Planning in Structured Relational Enviroments*. Tese de Doutorado, Graduate Department of Computer Science, University od Toronto. Citado na pág. 50

Smith e Simmons(2006) T. Smith e R. Simmons. Focused Real-Time Dynamic Programming for MDPs: Squeezing More Out of a Heuristic. *Proceedings of the Twenty-first National Conference on Artificial Intelligence*. Citado na pág. 38

St-Aubin *et al.***(2001)** Robert St-Aubin, Jesse Hoey e Craig Boutilier. APRICODD: Approximate Policy Construction using Decision Diagrams. *Advances in Neural Information Processing System (NIPS), MIT Press*, páginas 1089–1095. Citado na pág. 4, 54

Sutton e Barto(1998) R. Sutton e A. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, 2º edição. Citado na pág. 39

Vianna *et al.***(2012)** L. Vianna, L. Barros e K. Valdivia. Online Simulation based Planning. *Association for the Advancement of Artificial Intelligence*. Citado na pág. 67

Warnquist *et al.*(**2010**) H. Warnquist, J. Kvarnstrom e P. Doherty. Iterative Bounding LAO*. *Proceeding of the 19th European Conference on Artificial Intelligence (ECAI)*, páginas 341–346. Citado na pág. 4, 46

Zhou e Hansen(2007) R. Zhou e E. Hansen. Anytime Heuristics Search. *Journal of Artificial Intelligence Research*, 28(1):267–297. Citado na pág. 46