**Planejamento hierárquico
sob incerteza Knightiana**

Ricardo Guimarães Herrmann

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO
DE
MESTRE EM CIÊNCIAS

Área de Concentração: Ciência da Computação
Orientadora: Prof^a Dr^a Leliane Nunes de Barros

São Paulo, maio de 2008

Planejamento hierárquico sob incerteza Knightiana

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Ricardo Guimarães Herrmann
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof^a Dr^a Leliane Nunes de Barros (orientadora) - IME-USP.
- Prof. Dr. Marcelo Finger - IME-USP.
- Prof. Dr. Fábio Gagliardi Cozman - EP-USP.

Agradecimentos

Agradeço principalmente à minha orientadora, Dra. Leliane Barros. A experiência que eu adquiri durante o período vai me acompanhar durante toda minha vida e carreira. Dr. José Pérez Alcázar, Dr. Marcelo Finger e Dr. Fábio Cozman, por participarem das bancas de qualificação e defesa e pelos comentários que ajudaram a melhor definir o escopo deste trabalho. O incentivo dos professores (e ótimos coaches) Dr. Cássio Polpo e Aritanan Gruber, que inicialmente me mostraram a abrangência da área de estudo que é Ciência da Computação.

Dedico esta dissertação à minha esposa Michelle, pela paciência e suporte nestes últimos anos. Também agradeço aos meus pais, Jorge e Solange, pela formação moral e intelectual que tiveram condições de me dar.

Agradeço também meus empregadores durante o período (Agência Estado, MRO Software e IBM) e respectivos gerentes (Sandro, Fábio e Diego), que não pouparam esforços para estender prazos sempre que necessário, nos períodos difíceis.

É impossível deixar de mencionar também meus colegas David Camargo, Domingos Soares, Rafael Alves, Renato Lucindo e Maryana Alegro, que também embarcaram nesta jornada de conhecimento, por todos estes anos que renderam milhares de conversas interessantes sobre este trabalho.

“If I have seen further, it is by standing on the shoulders of giants.”
– Sir Isaac Newton

Resumo

Esta dissertação tem como objetivo estudar a combinação de duas técnicas de planejamento em inteligência artificial: planejamento hierárquico e planejamento sob incerteza Knightiana. Cada uma delas possui vantagens distintas, mas que podem ser combinadas, permitindo um ganho de eficiência para o planejamento sob incerteza e maior robustez a planos gerados por planejadores hierárquicos.

Primeiramente, estudamos um meio de efetuar uma transformação, de modo sistemático, que permite habilitar algoritmos de planejamento determinístico com busca progressiva no espaço de estados a tratar problemas com ações não-determinísticas, sem considerar a distribuição de probabilidades de efeitos das ações (incerteza Knightiana).

Em seguida, esta transformação é aplicada a um algoritmo de planejamento hierárquico que efetua decomposição a partir das tarefas sem predecessoras, de modo progressivo. O planejador obtido é competitivo com planejadores que representam o estado-da-arte em planejamento sob incerteza, devido à informação adicional que pode ser fornecida ao planejador, na forma de métodos de decomposição de tarefas.

Palavras-chave: planejamento em IA, planejamento hierárquico, planejamento não-determinístico

Abstract

This dissertation’s objective is to study the combination of two artificial intelligence planning techniques, namely: hierarchical planning and planning under Knightian uncertainty. Each one of these has distinct advantages, but they can be combined, allowing the planning under uncertainty a performance gain and giving the hierarchical planning the ability to produce more robust plans.

First, we study a way of performing a transformation, in a systematic way, that enables forward-chaining deterministic planning algorithms to deal with non-deterministic actions, that doesn’t take into account the probability distribution of actions’ effects (Knightian uncertainty).

Afterwards, this transformation is applied to a hierarchical planning algorithm that progressively performs decomposition starting from tasks without predecessors. The obtained planner is competitive with state-of-the-art non-deterministic planners, thanks to the additional information that can be given to the planner, in the form of task decomposition methods.

Keywords: AI planning, hierarchical planning, non-deterministic planning

Sumário

Lista de Abreviaturas	xi
Lista de Símbolos	xiii
Lista de Figuras	xvii
1 Introdução	1
1.1 Objetivos e organização da dissertação	4
2 Planejamento	7
2.1 Modelo conceitual para planejamento	7
2.2 Planejamento clássico	9
2.2.1 Representação de estados e ações	11
2.2.2 Algoritmos para planejamento clássico	15
2.2.3 Outras técnicas de planejamento clássico	17
3 Planejamento Hierárquico	21
3.1 Planejamento com redes hierárquicas de tarefas	23
3.2 Planejamento com redes simples de tarefas	26
3.3 Extensões	31
3.3.1 Extensões adicionais	32
3.3.2 Planejadores hierárquicos	32
3.3.3 Vantagens e desvantagens	33

4	Planejamento sob Incerteza	35
4.1	Fontes de incerteza em planejamento	35
4.1.1	Não-determinismo nos efeitos de ações	35
4.1.2	Observabilidade parcial	37
4.2	Técnicas de planejamento sob incerteza	37
4.3	Modelo formal	39
4.4	Planejamento forte, forte cíclico e fraco	40
4.4.1	Planejamento forte	41
4.4.2	Planejamento forte cíclico	42
4.4.3	Planejamento fraco	42
4.5	Planejamento baseado em verificação de modelos	43
4.5.1	Verificação de modelos	45
4.5.2	Domínios de planejamento	46
4.5.3	Verificação de modelos simbólicos	46
4.5.4	Planejadores baseados em verificação de modelos	47
4.5.5	Limitações dessa abordagem	47
5	Planejamento Hierárquico sob Incerteza	49
5.1	ND-transformação de planejadores progressivos	50
5.1.1	Propriedades teóricas	53
5.2	ND-transformação de planejadores HTN progressivos	54
5.2.1	Algoritmo para planejamento forte cíclico	54
5.2.2	Algoritmo para planejamento forte	60
5.3	Utilização de BDDs	61
5.3.1	Representação de problemas de planejamento com BDDs	62
5.4	Metas de decomposição sob incerteza	64
5.4.1	Planos estendidos	64
5.4.2	Planos estendidos para metas de decomposição	66

6	Análise Empírica	69
6.1	Metodologia para análise do planejador	69
6.2	O domínio Predador-Presas	70
6.3	Estratégias hierárquicas para Predador-Presas	72
6.4	Um exemplo de uso do ND-HSHOP no domínio Predador-Presas	74
6.5	Análise de desempenho do ND-HSHOP	75
6.6	Análise do uso de hierarquias	76
6.7	Sobre o uso da linguagem Haskell	78
7	Conclusão	81
7.1	Trabalhos futuros	83
A	Apêndice: Implementação	85
A.1	Haskell	85
A.2	Linguagem para planejamento	88
A.3	Algoritmo FS_{SHOP}^3	89
A.3.1	Módulos utilizados	89
A.3.2	Modelo	90
A.3.3	Algoritmo	91
A.3.4	Funções auxiliares	96
B	Apêndice: Um Estudo Sobre a Lógica CTL	99
B.1	Lógica de Tempo Ramificado (<i>Computation Tree Logic</i>) (CTL)	99
B.1.1	Lógica modal proposicional	99
B.1.2	Árvores de computação	100
B.1.3	A lógica de tempo ramificado CTL	100
B.2	Planejamento com CTL	101
	Referências Bibliográficas	103

Lista de Abreviaturas

IA	Inteligência Artificial
VM	Verificação de Modelos
STN	Rede Simples de Tarefas (<i>Simple Task Network</i>)
HTN	Rede Hierárquica de Tarefas (<i>Hierarchical Task Network</i>)
TFD	Decomposição Progressiva em Ordem Total (<i>Total-order Forward Decomposition</i>)
PFD	Decomposição Progressiva em Ordem Parcial (<i>Partial-order Forward Decomposition</i>)
FCP	Planejamento Progressivo (<i>Forward Chaining Planning</i>)
ND-FCP	<i>Non-Deterministic Forward Chaining Planning</i>
MBP	<i>Model Based Planner</i>
SHOP	<i>Simple Hierarchical Ordered Planner</i>
STRIPS	<i>Stanford Research Institute Planning System</i>
UMCP	<i>Universal Method Composition Planner</i>
NOAH	<i>Nets Of Action Hierarchies</i>
SOUP	<i>Semantics Of User Programs</i>
CTL	Lógica de Tempo Ramificado (<i>Computation Tree Logic</i>)
LTL	Logica de Tempo Linear (<i>Linear Temporal Logic</i>)
TAL	Lógicas de Ação Temporal (<i>Temporal Action Logics</i>)

BDD	Diagrama de Decisão Binária (<i>Binary Decision Diagram</i>)
OBDD	Diagrama de Decisão Binária Ordenado (<i>Ordered Binary Decision Diagram</i>)
MDP	Processo de Decisão de Markov (<i>Markov Decision Process</i>)
FS³	Separação de Espaço de Estados Progressiva (<i>Forward State-Space Splitting</i>)
GHC	<i>Glasgow Haskell Compiler</i>
ESA	Agência Espacial Européia

Lista de Símbolos

Σ	sistema de transição de estados
Σ'	sistema de transição de estados com restrições (estático e determinístico)
Σ_π	estrutura de execução de uma política
ϵ	evento nulo
η	função de observação
γ	função de transição de estados
$\gamma(s, a)$	progressão, conjunto de estados resultantes da aplicação de a a s
$\gamma^{-1}(s, a)$	regressão, conjunto de estados que levam a s com a aplicação de a
π	plano, política
σ	substituição lógica
$\cdot$	composição de seqüências
2^S	conjunto potência de S
$\mathcal{A}$	conjunto de ações
$\mathcal{C}$	conjunto de símbolos constantes
$\mathcal{D}$	domínio de planejamento, conjunto de operadores
$\mathcal{E}$	conjunto de eventos
$\mathcal{F}$	conjunto de símbolos de tarefas primitivas
$\mathcal{K}$	estrutura de Kripke
$\mathcal{M}$	conjunto de métodos
$\mathcal{N}$	conjunto de rótulos para redes de tarefas
$\mathcal{O}$	conjunto de observações
$\mathcal{P}$	conjunto de proposições de um domínio de planejamento

$\mathcal{P}$	conjunto de símbolos de predicados
$\mathcal{S}$	conjunto de estados
$\mathcal{T}$	conjunto de símbolos de tarefas compostas
$\mathcal{V}$	conjunto de símbolos de variáveis
E	conjunto de arestas
O	conjunto de observações, sub-conjunto de $\mathcal{S}$
P	problema de planejamento
S	conjunto de estados, sub-conjunto de $\mathcal{S}$
S_0	conjunto de estados iniciais
S_g	conjunto de estados meta
U	conjunto de nós
no-op	ação nula
a	ação de $\mathcal{A}$
e	evento de $\mathcal{E}$
k	índice
l	literal
m	método
n	rótulo de $\mathcal{N}$
r	termo
s	estado de $\mathcal{S}$
s_0	estado inicial
t_u	tarefa associada a um nó de uma rede de tarefas
u, v	nós de uma rede de tarefas
w	rede de tarefas
x	variável
$\langle \dots \rangle$	tupla
$(\dots)$	lista
$\{ \dots \}$	conjunto
$\cup$	união de conjuntos

$\cap$ interseção de conjuntos

$\setminus$ subtração de conjuntos

Lista de Figuras

2.1	Exemplo de sistema de transição de estados.	8
2.2	Modelo conceitual para planejamento.	10
3.1	Exemplo de decomposição de tarefa composta.	23
3.2	Exemplo de decomposição de tarefas em ordem total.	29
3.3	Processo de compilação de planejadores dependentes de domínio em JSHOP2.	34
4.1	Exemplo de sistema de transição de estados não-determinístico.	39
5.1	Exemplo de poda do espaço de busca através de ações compostas. . . .	50
5.2	Exemplo de solução cíclica forte na decomposição com ações não-determinísticas.	56
5.3	BDDs resultantes de diferentes ordens de variáveis para a mesma fórmula. . . .	62
5.4	Falta de expressividade de políticas no planejamento hierárquico não-determinístico.	67
5.5	Tabelas das funções <i>act</i> e <i>ctxt</i> de Π para o exemplo 5.4.	68
6.1	Exemplo de estado de um problema com 2 presas em uma grade 5×5 . .	71
6.2	Número de estados possíveis em problemas Predador-Presas, com número de presas variando de 1 a 10, para $n = \{2, 3, 4\}$	72
6.3	Número de estados possíveis em problemas Predador-Presas, com tamanho da grade variando de 1 a 100, para $p = \{1, 2, 3, 4\}$	73
6.4	Problema de Predador-Presas com uma presa em grade 2×2	74

6.5	Gráfico com número de estados das políticas geradas por ND-HSHOP, para grades $n \times n$, com $n = \{2, 3, \dots, 70\}$	76
6.6	Gráfico de tempo de execução de ND-HSHOP em escala logarítmica. . .	77
6.7	Comparação de tempo entre MBP, ND-HSHOP e HMBP, em problemas com uma presa. Tempo em escala logarítmica.	78
6.8	Comparação de tempo entre MBP e ND-HSHOP, em problemas com uma presa. Tempo em escala linear.	79
A.1	Diagrama de classes UML mostrando as relações entre formalismos utilizados.	87

Lista de Algoritmos

1	PLANEJAMENTOPROGRESSIVO $(s_0, S_g, \mathcal{D})$	16
2	PLANEJAMENTOREGRESSIVO $(s_0, S_g, \mathcal{D})$	17
3	DECOMPOSIÇÃOORDEMPARCIAL $(s, w, \mathcal{D}, \mathcal{M})$	30
4	VERIFICAFORTE $(\pi, S_{abertos}, S_g, S_0)$	42
5	VERIFICAFORTECÍCLICO $(\pi, S_{abertos}, S_g, S_0)$	43
6	VERIFICAFRACO $(\pi, S_{abertos}, S_g, S_0)$	44
7	FCP $(s_0, g, \mathcal{D}, \alpha)$	51
8	ND-FCP $(S_0, g, \mathcal{D}', \alpha')$	53
9	YoYo $(\mathcal{D}, S_0, S_g, w, \mathcal{M})$	55
10	YoYoAUX $(\mathcal{D}, X, S_g, \mathcal{M}, \pi, x_0)$	55
11	PODASITUAÇÕES (X, S_g, π)	57
12	EFETUADECOMPOSIÇÃO $(S, w, \mathcal{D}, \mathcal{M})$	58
13	ENCONTRASUCESORES (F, X)	59
14	VERIFICAPOLÍTICA_FORTE (π, X, S_g, x_0)	61

Capítulo 1

Introdução

*“First, we shape our structures.
Then, our structures shape us.”
– Winston Churchill*

A Inteligência Artificial (IA) é a sub-área da Ciência da Computação interessada na construção de agentes de software inteligentes [RN03]. Uma das principais características de um agente inteligente é a capacidade de planejar suas ações para: realizar tarefas, satisfazer determinados objetivos ou ainda, de uma maneira mais geral, resolver problemas. Assim, planejamento em IA [GNT04], ou *planejamento automatizado*, consiste na construção de planos de ações que, ao serem executados, permitem que o agente alcance seus objetivos (ou *metas*).

Há diversas formas de planejamento para fins específicos, como planejamento de projeto [Bel89], de sensoriamento [Kri95], navegação robótica [LaV06, pt. II] e programação automatizada [McL06], entre outras. Apesar de existirem soluções específicas para alguns desses problemas, o planejamento em IA concentra-se em soluções *independentes de domínio*, isto é, no desenvolvimento de sistemas de planejamento que possam ser reutilizados em diferentes domínios de aplicação, com o intuito de reduzir o esforço necessário à criação de novos planejadores [GNT04] e possibilitar a construção de agentes autônomos.

Uma *meta* de planejamento simples pode ser uma descrição de um conjunto de propriedades do ambiente (ou *estado do mundo*) que devem ser verdadeiras. Uma des-

criação de estado do mundo é completa se ela especifica *todas* as propriedades relevantes do ambiente. Por exemplo, para um agente carteiro, a propriedade g dada por “todas as correspondências devem estar sob a mesa da secretária” define um conjunto de estados do mundo, ou seja, todos os estados em que g é verdadeira. Dizemos assim que g é uma *sub-meta* do agente de planejamento e S_g é o conjunto de estados em que g é verdadeira. S_g é chamado de conjunto de estados meta.

No instante em que um agente inteligente decide planejar para alguma sub-meta, ele observa o ambiente a fim de determinar qual é o estado do mundo em que ele se encontra. Este estado é chamado de *estado inicial*; é a partir deste estado que o plano deverá ser executado. Dependendo da capacidade de observação do agente, dizemos que este possui *observabilidade total* ou *observabilidade parcial* do ambiente.

Após a execução de uma ação (ou plano) o agente encontra-se em um novo estado do mundo. Planejar envolve a antecipação das conseqüências das ações, isto é, raciocinar sobre os estados que o agente pode alcançar com suas ações, antes mesmo de executá-las. Uma ação pode ser *determinística* quando todas as suas conseqüências (*efeitos* da ação) são totalmente previsíveis e, dado o estado do mundo, é possível prever qual será o próximo estado. Quando mais que um estado pode ser previsto, dizemos que a ação é *não-determinística*. Neste caso, o agente deve ser capaz de planejar sob *incerteza*.

Uma outra fonte de incerteza, além da observabilidade parcial e uso de ações não-determinísticas, é a presença de outros agentes atuando no mesmo ambiente, do que decorrem *ações exógenas*, ações que fogem ao controle do agente, mas que também alteram o estado do mundo.

O uso e escolha de linguagens para a descrição de ações, estados e planos é fundamental no desenvolvimento de sistemas de planejamento. Planos podem ser de diferentes tipos: os mais simples podem ser representados como seqüências de ações, totalmente ou parcialmente ordenadas. Um plano condicional possui ramificações definidas por condições que deverão ser testadas somente durante sua execução. Um plano pode ser uma *política*: uma função que mapeia estados do mundo em ações, representando planos na forma de um agente reativo específico para o problema. Existem ainda *planos estendidos*, que são políticas que mapeiam estados e contextos de execução (históricos) a ações.

O *planejamento clássico* [GNT04, cap. 2] faz suposições restritivas sobre: o modelo do mundo (ou modelo do ambiente); o tipo de plano desejado; o tipo de meta; a capacidade do agente em termos de observação do estado do mundo e interação com o ambiente. No planejamento clássico, o ambiente pode ser modelado como um sistema de transição de estados determinístico, completamente observável, sem ações exógenas, com tempo implícito e metas especificadas através de estados desejados (*metas de alcançabilidade*), com soluções que possam ser dadas por seqüências de ações. Mesmo sob todas estas restrições, o planejamento clássico é um problema NP-difícil [ENS95] e, ainda assim, problemas de interesse prático, em sua maioria, não podem ser resolvidos sob tais suposições.

Uma variação de planejamento clássico é o *planejamento hierárquico* [Yan90] [EHN94b], em que o controle de busca é efetuado pela decomposição de *tarefas de alto nível* em tarefas mais específicas, até o nível de ações primitivas (isto é, ações que podem ser executadas diretamente pelo sistema de controle). Os diferentes *métodos de decomposição* de uma tarefa em sub-tarefas são especificados pelo projetista do domínio. Os métodos de decomposição permitem maior controle sobre a busca de soluções, o que contribui para o fato de que o planejamento hierárquico seja a técnica mais utilizada em aplicações do mundo real. A utilização deste conhecimento adicional sobre o domínio permite derivar planejadores que obtêm soluções em tempo polinomial em relação ao tamanho do problema para casos que, através do planejamento clássico, só poderiam ser solucionados em tempo exponencial [EHN94a].

Uma outra característica de aplicações reais é a *incerteza* sobre o estado do mundo e os efeitos das ações. Tradicionalmente, a incerteza é modelada como um jogo entre duas entidades [BG79] [Per07]: (i) o *agente*, que planeja suas ações para atingir suas metas e (ii) a *natureza*, cujas intenções são desconhecidas e cujas ações (consideradas *eventos exógenos*) podem interferir com as ações do agente, modificando seus efeitos. Do ponto de vista do agente, ações podem falhar ou ter efeitos diferentes em situações específicas (*e.g.*, no lançamento de um dado, onde o resultado é determinado pela natureza) em que ocorrem eventos exógenos, sobre os quais o agente não tem controle ou até mesmo conhecimento da existência. A incerteza do agente deve ser levada em consideração para que os planos sintetizados sejam válidos (*i.e.*, corretos).

Existem extensões do planejamento clássico para lidar com domínios de planeja-

mento *não-determinístico*, em que uma ação pode levar o agente a diferentes estados do mundo. Quando é possível identificar distribuições de probabilidades nos efeitos das ações e associar recompensas aos estados do mundo, a técnica predominantemente utilizada é a representação de problemas através de um Processo de Decisão de Markov (*Markov Decision Process*) (MDP) [How60]. Nesse caso, o objetivo do planejador é obter uma política que maximiza a recompensa obtida pela visitação dos estados, com base nas probabilidades dos efeitos das ações [BG01]. Quando as distribuições de probabilidade não são conhecidas, a técnica mais utilizada é a de *verificação de modelos* [Rov01]. Soluções podem ser políticas de diferentes níveis de garantias de alcance de metas, caracterizadas, em nível decrescente de garantia, como *fortes*, *fortes cíclicas* ou *fracas*.

Uma outra abordagem promissora para resolver problemas de planejamento é a de *planejamento progressivo hierárquico*, originalmente proposta para fornecer heurísticas eficientes de controle de busca para domínios com efeitos determinísticos [NAI⁺03]. Kuter et al. [KN04] propõem uma forma de generalizar essa abordagem, de modo a tratar o caso não-determinístico, tornando possível reutilizar uma classe de outras técnicas de planejamento progressivo. Esta técnica baseia-se na propriedade de que planejadores que utilizam busca progressiva no espaço de estados sempre têm conhecimento completo do estado corrente do mundo, enquanto o emprego de planejamento hierárquico possibilita uma redução do espaço de busca. Entretanto, não existe uma implementação disponível do planejador descrito por Kuter et al. [KN04, KNPT05]. As técnicas de planejamento hierárquico e planejamento sob incerteza Knightiana apresentam suas características próprias, bem como fazem diferentes suposições sobre os problemas de planejamento. É, portanto, de grande utilidade integrar essas diferentes técnicas de planejamento, como é a proposta dos algoritmos ND-SHOP [KN04] e YoYo [KNPT05].

1.1 Objetivos e organização da dissertação

O objetivo deste trabalho é implementar algoritmos para resolver problemas que envolvam planejamento sob incerteza Knightiana e planejamento hierárquico com busca progressiva. O sistema desenvolvido será comparado em termos de desempenho com algoritmos clássicos de planejamento sob incerteza, que não fazem decomposição hie-

rárquica. Assim, será possível mostrar como o conhecimento sobre tarefas hierárquicas em um domínio não-determinístico pode resultar em algoritmos mais eficientes de planejamento.

Esta dissertação está estruturada nos seguintes capítulos:

- No Capítulo 2 é definido um modelo geral utilizado em planejamento, bem como as suposições restritivas do planejamento clássico. Além disso, são definidas formas de representação simbólica de estados e ações.
- No Capítulo 3 são apresentadas duas formas de planejamento hierárquico: uma mais geral e expressiva, mas com maior complexidade, e outra, simplificada e especializada para lidar com busca progressiva. Além disso, enumeram-se algumas possíveis extensões, discutindo suas vantagens e desvantagens.
- No Capítulo 4 introduz-se o conceito de não-determinismo e suas conseqüências no processo de planejamento. Além disso, é dada uma breve revisão sobre a verificação de modelos aplicada a sistemas de planejamento.
- No Capítulo 5 é feita a união das técnicas listadas anteriormente, mostrando como é possível obter um planejador hierárquico não-determinístico baseado em busca progressiva. Também são descritas duas extensões para o planejamento hierárquico sob incerteza: a primeira visa permitir que se faça uso de técnicas utilizadas em verificação de modelos simbólicos para conseguir maior eficiência na busca. A segunda permite um meio de encontrar planos para alguns problemas que não aceitam políticas como soluções válidas.
- No Capítulo 6 são apresentados os resultados de testes com diversos problemas não-determinísticos, assim como a metodologia de testes utilizada nesta dissertação.
- Finalmente, no Capítulo 7 são exibidas as conclusões tomadas a partir dos experimentos analisados e uma lista de possíveis trabalhos futuros.
- Notas sobre a implementação dos algoritmos, junto com uma parte comentada do código fonte, podem ser encontradas no Apêndice A. Também, no Apêndice B, é

resumido o formalismo utilizado pela Lógica de Tempo Ramificado (*Computation Tree Logic*) (CTL).

Capítulo 2

Planejamento

Planejamento, em sua forma mais geral, envolve diversos aspectos do mundo, como, por exemplo: tempo, recursos, otimização, entre outros. Além disso, há diversas maneiras de representar as ações do agente e seus efeitos. O planejamento clássico trabalha com um modelo simplificado e bem fundamentado do mundo, que serve como base para técnicas mais sofisticadas de planejamento. Este capítulo apresenta o planejamento clássico, além de uma breve revisão sobre os principais algoritmos utilizados.

2.1 Modelo conceitual para planejamento

Definição 2.1.1. O formalismo mais comum para ambientes de planejamento é um *sistema de transição de estados* [GNT04, cap. 1], uma 4-tupla $\Sigma = \langle \mathcal{S}, \mathcal{A}, \mathcal{E}, \gamma \rangle$, onde:

- $\mathcal{S} = \{s_1, s_2, \dots\}$ é um conjunto finito ou recursivamente enumerável de estados;
- $\mathcal{A} = \{a_1, a_2, \dots\}$ é um conjunto finito ou recursivamente enumerável de ações;
- $\mathcal{E} = \{e_1, e_2, \dots\}$ é um conjunto finito ou recursivamente enumerável de eventos;
- $\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{E} \rightarrow 2^{\mathcal{S}}$ é uma função de transição de estados. $\diamond$

Estados representam configurações específicas de ambientes. Ações e eventos representam os acontecimentos que levam um ambiente a mudar de estado. A diferença entre ações e eventos é que eventos não estão sob controle do agente e podem ocorrer independentemente das ações, que são controladas pelo agente planejador.

Figura 2.1: Exemplo de sistema de transição de estados.

Um sistema de transição de estados Σ pode ser representado como um *grafo direcionado*, onde nós representam estados e transições são arestas anotadas com pares ação-evento (Figura 2.1(a)). A função γ pode ser explicitamente representada através de uma tabela (Figura 2.1(b)), mas, em geral, é dada através de representações simbólicas mais compactas [GNT04, cap. 2]. Também utiliza-se a ação nula $\text{no-op} \in \mathcal{A}$ e o evento nulo $\epsilon \in \mathcal{E}$ para denotar transições causadas exclusivamente por eventos ou ações, respectivamente (onde a omissão de uma ação ou evento na notação de transições implica a ocorrência de um ou outro).

Definição 2.1.2. Uma ação a é *aplicável* a um estado s , ou $a \in \mathcal{A}(s)$ se e só se $\gamma(s, a) \neq \emptyset$. Por definição, $\mathcal{A}(s) \subseteq \mathcal{A}$. Se a é aplicada no estado s , isto leva o ambiente a um estado $s' \in \gamma(s, a, e)$. $\diamond$

É conveniente visualizar o problema de planejamento através de uma interação entre três componentes (Figura 2.2):

1. Um *sistema de transição de estados* Σ , que evolui de acordo com a sua função de transição de estados γ , reagindo aos eventos e ações às quais é submetido;
2. Um *controlador*, que recebe, como entrada, o estado corrente s do sistema (atra-

vés de observações) e provê, como saída, uma ação a , de acordo com algum *plano*;

3. Um *planejador*, que tem como entrada uma descrição do sistema Σ , uma situação inicial e uma meta. Sintetiza um *plano* para ser fornecido ao controlador, com o intuito de atingir o objetivo. Em alguns casos, o planejador necessita de realimentação por parte do controlador, na forma de informação sobre o *estado da execução* do plano. Planejadores que dependem de realimentação (Figura 2.2, seta tracejada) são classificados como planejadores de *malha fechada*. Caso contrário, são ditos que trabalham em *malha aberta*.

O controlador trabalha *online* com Σ . Já o planejador não se comunica diretamente com Σ e trabalha *offline* se não recebe realimentação. Este último concentra-se não no estado atual do mundo em tempo de planejamento, mas em quais estados o ambiente modelado possa vir a estar enquanto o controlador executa o plano.

O modelo do mundo geralmente não considera todas as características do sistema físico a ser controlado, tanto por questões de eficiência quanto de viabilidade. Portanto, é comum supor que o controlador seja robusto o suficiente para lidar com as diferenças entre Σ e o mundo real.

Outro elemento importante pode ser adicionado ao modelo: a informação que o controlador possui sobre o estado atual do mundo pode ser incompleta. O conhecimento parcial do estado pode ser modelado a partir de um conjunto $\mathcal{O}$ de observações, como uma função de observação $\eta : \mathcal{S} \rightarrow 2^{\mathcal{O}}$, que associa, a cada estado s , um conjunto $\eta(s) = \{o_1, o_2, \dots\} \subseteq \mathcal{O}$ de possíveis observações. As percepções do controlador tornam-se então a observação $O = \eta(s)$.

2.2 Planejamento clássico

O formalismo clássico para planejamento baseia-se em um modelo restrito, a fim de manter uma semântica bem definida, e utiliza o mesmo conceito de estados e operadores que o *Stanford Research Institute Planning System* (STRIPS) [FN71], considerado um dos sistemas pioneiros em planejamento automatizado.

Definição 2.2.1. Um *sistema de transição de estados restrito*, supondo determinismo

Figura 2.2: Modelo conceitual para planejamento.

nos efeitos das ações, é uma 3-tupla $\Sigma' = \langle \mathcal{S}, \mathcal{A}, \gamma \rangle$, onde:

- $\mathcal{S} = \{s_1, s_2, \dots\}$ é o conjunto finito ou recursivamente enumerável de estados;
- $\mathcal{A} = \{a_1, a_2, \dots\}$ é o conjunto finito ou recursivamente enumerável de ações;
- $\gamma : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ é a função (parcial) de transição de estados. $\diamond$

Definição 2.2.2. Um *problema de planejamento clássico* é a 3-tupla $\mathcal{P} = \langle \Sigma', s_0, S_g \rangle$, onde:

- Σ' um sistema de transição de estados restrito, ou *domínio de planejamento*;
- s_0 é um *estado inicial*;
- S_g é uma *meta* (ou *objetivo*) de alcançabilidade, formada por *estados meta*. $\diamond$

Dado um problema, um *planejador clássico* deve devolver um *plano*: uma sequência de n ações $\pi = (a_1, \dots, a_n)$ que, se executadas a partir de s_0 , levam a um estado meta, ou seja, $\gamma(s_0, a_1) = s_1, \gamma(s_1, a_2) = s_2, \dots, \gamma(s_{n-1}, a_n) = s_n$, onde $s_n \in S_g$. O planejamento clássico parte das seguintes suposições restritivas [GNT04]:

- Σ' é **finito**, ou seja, é composto de um número finito de estados;

- Σ' é **completamente observável**, o que significa que é sempre possível saber com exatidão qual o estado atual do mundo;
- Σ' é **determinístico**, onde ações levam o sistema a um único estado possível;
- Σ' é **estático**, o que implica que não ocorrem eventos exógenos, externos ao agente, isto é, somente o agente manipula o mundo;
- **Metas restritas**, especificadas por um conjunto de estados meta;
- **Planos sequenciais**, representados simplesmente como uma sequência de ações a serem executadas na ordem indicada (parcial ou total);
- **Tempo implícito**, supondo tempo discreto (episódico), onde apenas uma ação ocorre por vez e não há informação sobre sua duração;
- **Planejamento *offline***, caracterizando a independência do planejador em relação ao controlador, sem realimentação sobre a execução do plano (malha aberta). Por exemplo, na Figura 2.2, o estado de execução (indicado pela seta tracejada) não seria comunicado ao planejador.

2.2.1 Representação de estados e ações

Problemas de planejamento reais normalmente envolvem uma grande quantidade de estados e ações, o que torna sua enumeração explícita inviável, devido à explosão combinatória causada pela representação dos estados individuais dos objetos envolvidos nos problemas. Portanto, é necessário utilizar meios simbólicos de representação e manipulação do mundo. Ghallab et al. [GNT04] apresentam três representações tradicionais de estados e ações:

Representação em teoria dos conjuntos. Cada estado do mundo é um conjunto de proposições. Cada ação é uma expressão sintática especificando quais proposições devem fazer parte do estado para que a ação seja aplicável, junto com as proposições que a ação irá adicionar e/ou remover para gerar um novo estado do mundo;

Representação clássica. Estados e ações são como os descritos pela representação em teoria dos conjuntos, com a diferença de que literais de uma linguagem lógica de primeira ordem e conectivos lógicos são utilizados no lugar das proposições. É a representação mais utilizada pelos planejadores tradicionais;

Representação com variáveis de estado. Cada estado é representado por uma tupla de n variáveis de estado $\langle x_1, \dots, x_n \rangle$. Uma ação é representada por uma função parcial, que mapeia certa tupla em uma outra tupla de valores das n variáveis de estado. Esta abordagem é útil na representação de domínios nos quais um estado é um conjunto de atributos que recebem valores de domínios finitos e cujos valores variam no tempo.

As duas primeiras formas de representação são discutidas a seguir. O planejador implementado durante esta dissertação faz uso destes dois formalismos. A terceira representação (variáveis de estado) não foi utilizada e, portanto, por brevidade na apresentação, não será discutida. É possível consultar o trabalho de Ghallab et al. [GNT04], cap. 2 para maiores detalhes.

Representação em teoria dos conjuntos

É possível representar estados e ações utilizando a teoria dos conjuntos. Partimos da *hipótese do mundo fechado*, ou seja, o que não é representado explicitamente é interpretado como falso, o que torna possível representar estados como conjuntos de proposições.

Definição 2.2.3. Seja $L = \{p_1, \dots, p_n\}$ um conjunto finito de *símbolos de proposições*. Um *domínio de planejamento em teoria dos conjuntos* sobre L é um sistema de transição de estados restrito $\Sigma = \langle \mathcal{S}, \mathcal{A}, \gamma \rangle$, tal que:

- $\mathcal{S} \subseteq 2^L$, isto é, cada estado $s \in \mathcal{S}$ é composto por um sub-conjunto de L , onde dizemos que p é verdadeira se e só se $p \in s$, e se $p \notin s$ então p é falsa no estado do mundo representado por s (hipótese do mundo fechado);
- Cada ação $a \in \mathcal{A}$ é uma 3-tupla de sub-conjuntos de L , denotada por:

$$a = \langle \text{precond}(a), \text{efeitos}^-(a), \text{efeitos}^+(a) \rangle.$$

O conjunto $\text{precond}(a)$ é chamado de conjunto de *pré-condições* de a e os conjuntos $\text{efeitos}^-(a)$ e $\text{efeitos}^+(a)$ são chamados de *efeitos* de a . Estes dois conjuntos devem ser disjuntos, isto é, $\text{efeitos}^-(a) \cap \text{efeitos}^+(a) = \emptyset$. A ação a é *aplicável* a um estado s se $\text{precond}(a) \subseteq s$;

- A função de transição de estados é $\gamma(s, a) = (s \setminus \text{efeitos}^-(a)) \cup \text{efeitos}^+(a)$, se $a \in \mathcal{A}$ é aplicável a $s \in \mathcal{S}$. Caso contrário, $\gamma(s, a)$ é indefinida;
- $\mathcal{S}$ tem a propriedade de que se $s \in \mathcal{S}$, então, para cada ação a aplicável a s , $\gamma(s, a) \in \mathcal{S}$, isto é, a aplicação de uma ação a um estado sempre leva a um outro estado válido. $\diamond$

Definição 2.2.4. Um *problema de planejamento em teoria dos conjuntos* é uma 3-tupla $P = \langle \Sigma, s_0, g \rangle$, onde:

- s_0 , o estado inicial, é um membro de $\mathcal{S}$;
- $g \subseteq L$ é um conjunto de proposições que estados devem satisfazer para que sejam estados meta. O conjunto de estados meta é $S_g = \{s \in \mathcal{S} | g \subseteq s\}$.

$\diamond$

Definição 2.2.5. Um *plano* é uma sequência de ações $\pi = (a_1, \dots, a_k)$, onde $k \geq 0$. O *comprimento* do plano é $|\pi| = k$, o número de ações. Se $\pi_1 = (a_1, \dots, a_k)$ e $\pi_2 = (a'_1, \dots, a'_j)$ são planos, então sua *concatenação* é o plano $\pi_1 \cdot \pi_2 = (a_1, \dots, a_k, a'_1, \dots, a'_j)$.

$\diamond$

Representação clássica

A representação clássica generaliza a representação em teoria dos conjuntos utilizando notação proveniente de uma linguagem de primeira ordem $\mathcal{L}$, na qual existe um número finito de símbolos de predicados e constantes mas não símbolos de funções. Portanto, todo termo de $\mathcal{L}$ é ou um símbolo de variável ou um símbolo de constante.

Entretanto, apesar do uso de uma linguagem lógica de primeira ordem, um estado não é representado por um conjunto de fórmulas, e sim, apenas, por um conjunto

de literais totalmente instanciados, verdadeiros sob alguma interpretação. Além de permitir representações mais compactas, a representação clássica de estados e ações promove o desacoplamento entre domínios e problemas de planejamento, separando-os em sua estrutura de ações e suas constantes, respectivamente.

Definição 2.2.6. A representação clássica de um estado é um conjunto de átomos *totalmente instanciados* (ou *ground*), isto é, com termos compostos exclusivamente de símbolos de constantes. Assim como na representação em teoria dos conjuntos, um átomo p é verdadeiro em s se e só se $p \in s$. Se g é um conjunto de literais (átomos simples e átomos negados), dizemos que s *satisfaz* g , ou $s \models g$, quando existe uma substituição σ tal que todo literal positivo de $\sigma(g)$ está em s e nenhum literal negativo de $\sigma(g)$ está em s . $\diamond$

O planejamento clássico costuma utilizar a *suposição de mundo fechado*: se um átomo não aparece na especificação de um estado, então o valor-verdade do literal correspondente é considerado falso, ao invés de indefinido. Adicionalmente, o valor-verdade de átomos pode variar de estado para estado. Alguns predicados podem ser considerados como uma função do conjunto de estados e são chamados de *fluentes*, ou *relações flexíveis*. Outros são invariantes, fornecidos como meio de estabelecer *relações rígidas* entre objetos.

Definição 2.2.7. No planejamento clássico, um *operador de planejamento* é uma 3-tupla $o = \langle \text{nome}(o), \text{precond}(o), \text{efeitos}(o) \rangle$, cujos elementos são os seguintes:

- $\text{nome}(o)$, o *nome* do operador, é uma expressão sintática da forma $n(x_1, \dots, x_k)$, onde n é um símbolo de operador, $x_1, \dots, x_k$ são os símbolos variáveis que aparecem que qualquer parte de o e n é único (não existem dois operadores com o mesmo nome);
- $\text{precond}(o)$ e $\text{efeitos}(o)$ são as pré-condições e efeitos de o , respectivamente, e são generalizações de uma ação representada em teoria dos conjuntos: utilizam conjuntos de literais ao invés de conjuntos de proposições. $\diamond$

Exemplo 2.2.8. O *mundo dos blocos* [CL03] é o domínio tradicionalmente mais utilizado na pesquisa em planejamento. Sua forma mais simples consiste em um conjunto

$\text{carregar}(x)$	precond: $\text{sobre}(x, \text{mesa}), \text{livre}(x)$ efeitos: $\neg\text{sobre}(x, \text{mesa}), \neg\text{livre}(x), \text{segurando}(x)$
$\text{deixar}(x)$	precond: $\text{segurando}(x)$ efeitos: $\neg\text{segurando}(x), \text{sobre}(x, \text{mesa}), \text{livre}(x)$
$\text{desempilhar}(x, y)$	precond: $\text{sobre}(x, y), \text{livre}(x)$ efeitos: $\neg\text{sobre}(x, y), \neg\text{livre}(x), \text{segurando}(x), \text{livre}(y)$
$\text{empilhar}(x, y)$	precond: $\text{segurando}(x), \text{livre}(y)$ efeitos: $\neg\text{segurando}(x), \neg\text{livre}(y), \text{sobre}(x, y), \text{livre}(x)$

Tabela 2.1: Operadores clássicos para o domínio do mundo dos blocos.

de blocos, unicamente identificados, dispostos sobre uma *mesa* ou empilhados sobre algum outro bloco. Uma garra robótica pode pegar um bloco e movê-lo para outra localização, denotada pelo objeto sobre o qual o bloco deverá estar localizado. Um bloco é dito livre se nenhum outro estiver no topo deste. Um estado é caracterizado pela localização de todos os blocos (*i.e.*, a configuração das pilhas). A tarefa do planejador é então encontrar uma seqüência de ações que transforma um dado estado inicial em um dado estado meta. Encontrar uma solução ótima é um problema NP-difícil [GN92]. Uma descrição clássica dos operadores do mundo dos blocos é dada na Tabela 2.1. $\diamond$

2.2.2 Algoritmos para planejamento clássico

Busca no espaço de estados

Os algoritmos mais simples para planejamento clássico efetuam busca no espaço de estados [GNT04, cap. 5]. Nestes algoritmos, o espaço de busca é um subconjunto do espaço de estados, onde cada nó corresponde a um estado do mundo e cada aresta indica uma possível transição. Os algoritmos de planejamento no espaço de estados podem ser *progressivos* ou *regressivos*.

O procedimento de busca progressiva começa pelo estado inicial e aplica, não-deterministicamente ¹, a função de transição de estados, produzindo sub-problemas. Estes sub-problemas buscam *soluções parciais*, pois são parte da solução final devolvida como resposta para o problema original. A busca termina quando um destes sub-problemas

¹No Algoritmo 1, a escolha não-determinística representa os pontos de retrocesso (*backtracking*) da busca, onde novas escolhas são feitas quando a busca *falha* durante o processo.

alcança um estado meta, ou *falha* se não houver nenhum plano possível para os sub-problemas.

Uma característica importante do planejamento progressivo (Algoritmo 1) é que permite ao planejador ter conhecimento sobre o estado completo do mundo a qualquer instante. Isto deve-se ao fato de que este opera a partir de um estado inicial, completamente especificado, e aplica sempre ações totalmente instanciadas aos estados, resultando em mais especificações completas de estados. É possível tirar proveito desta propriedade de diversas maneiras, sendo que esta dissertação explora algumas destas possibilidades.

Algoritmo 1: PLANEJAMENTOPROGRESSIVO ($s_0, S_g, \mathcal{D}$)	
Entrada: Estado inicial s_0 , Meta S_g , Domínio $\mathcal{D}$	
Saída: Plano π	
1	início
2	$\pi \leftarrow \emptyset; s \leftarrow s_0;$
3	repita
4	se $s \in S_g$ então devolva π ;
5	$A \leftarrow \{a \mid a \text{ é uma ação (instância total de um operador em } \mathcal{D}) \text{ e } a \in \mathcal{A}(s)\};$
6	se $A = \emptyset$ então devolva <i>falha</i> ;
7	não-deterministicamente escolha $a \in A$;
8	$\pi \leftarrow \pi.a;$
9	$s \leftarrow \gamma(s, a);$
10	fim
11	fim

O planejamento no espaço de estados também pode ser efetuado por uma busca regressiva (Algoritmo 2). O procedimento começa pelos estados meta e aplica a função inversa da transição, produzindo sub-metas. A busca termina quando uma destas sub-metas é satisfeita pelo estado inicial.

Busca no espaço de planos

Outro meio de encontrar planos para problemas de planejamento clássico é realizar *planejamento no espaço de planos* [GNT04, cap. 5]. Neste caso, o espaço de busca é o de *planos parcialmente especificados*, sendo que os nós representam planos parciais e as

Algoritmo 2: PLANEJAMENTOREGRESSIVO ($s_0, S_g, \mathcal{D}$)	
Entrada: Estado inicial s_0 , Meta S_g , Domínio $\mathcal{D}$	
Saída: Plano π	
1	início
2	$\pi \leftarrow \emptyset$;
3	repita
4	se $s_0 \in S_g$ então devolva π ;
5	$A \leftarrow \{a \mid a \text{ é uma ação (instância total de um operador em } \mathcal{D}) \text{ e } a \text{ é relevante para } S_g\}$;
6	se $A = \emptyset$ então devolva <i>falha</i> ;
7	não-deterministicamente escolha $a \in A$;
8	$\pi \leftarrow a.\pi$;
9	$S_g \leftarrow \gamma^{-1}(S_g, a)$;
10	fim
11	fim

arestas representam *operações de refinamento* de planos. Uma operação de refinamento adiciona restrições ao plano, e pode ter uma dentre as seguintes formas: *introdução de ações*, *vínculos causais*, *restrições de ordem entre ações* ou *atribuição de variáveis*. Todas estas formas de refinamento evitam adicionar ao plano restrições que não sejam estritamente necessárias, o que caracteriza o *princípio do comprometimento mínimo* [Wel94]. A busca ocorre de forma progressiva, iniciando-se a partir de um nó inicial, representando o plano vazio, e prossegue com o refinamento, até que um nó contendo um plano que alcance as metas de planejamento seja encontrado. Não há como efetuar uma busca regressiva no espaço de planos, pois o nó inicial seria exatamente o próprio plano que se está buscando.

2.2.3 Outras técnicas de planejamento clássico

Grafo de planejamento

Graphplan [BF97] é outro método de planejamento clássico, que utiliza como espaço de busca um *grafo de planejamento*. Graphplan devolve uma seqüência de conjuntos de ações, por exemplo:

$$\pi = (\{a_1, a_2\}, \{a_3, a_4\}, \{a_5, \dots\}, \dots),$$

que representa todas as seqüências começando em a_1 e a_2 em qualquer ordem, seguidas por a_3 e a_4 , também em qualquer ordem, e assim por diante. Diferentemente do planejamento no espaço de planos, o Graphplan estabelece fortes compromissos: ações são consideradas totalmente instanciadas (*ground*) e devem ocorrer em passos específicos. Esta técnica de planejamento depende da *análise de alcançabilidade* e do *refinamento disjuntivo* [Kam97]. A análise de alcançabilidade serve para determinar se um estado é alcançável a partir de algum outro estado s_0 . O refinamento disjuntivo consiste em lidar com uma ou mais falhas através de uma disjunção de soluções. Como, em geral, falhas não são independentes e suas soluções podem interferir, relações de dependência são colocadas como restrições a serem respeitadas em um estágio posterior.

A estrutura utilizada é um grafo em camadas alternadas de proposições e ações, onde cada camada de proposições representa, através de sua combinação, os possíveis estados alcançáveis a partir do estado inicial, pela aplicação de ações das camadas anteriores. Arestas são adicionadas ligando uma proposição a uma ação quando uma proposição é pré-condição desta. Do mesmo modo, ações ligam-se a proposições para denotar efeitos. Além disso, são adicionadas *restrições de exclusão mútua* entre pares de ações ou de proposições, a fim de representar escolhas de ações que não podem ocorrer simultaneamente em uma camada. Esta forma de planejamento apresenta duas fases distintas: uma de expansão do grafo de planejamento, adicionando um par de camadas de ações e proposições, e outra de tentativa de extração de planos, respeitando as restrições de exclusão mútua. Estas fases alternam-se iterativamente até que algum plano seja encontrado.

Planejamento como satisfazibilidade

A idéia por detrás do planejamento como satisfazibilidade é mapear um problema de planejamento a um outro problema bem conhecido, para o qual já existem algoritmos eficientes. Um plano é então extraído da solução deste outro problema. Mais especificamente, a idéia é reduzir um problema de planejamento a um *problema de satisfazibilidade proposicional* (também conhecido como SAT [DLL62]), ou seja, que procura dizer se uma fórmula proposicional é satisfazível. Avanços no desempenho de algoritmos para SAT permitem lidar com problemas mais complexos sem que seja necessário introduzir melhorias em algoritmos de planejamento. Esta técnica foi ori-

ginalmente proposta pelo planejador SatPlan [\[KS92\]](#), mas existem diferentes formas de codificação de problemas de planejamento como problemas SAT: um exemplo é sua utilização na fase de extração de planos de Graphplan, como faz o planejador Blackbox [\[KS99\]](#).

Capítulo 3

Planejamento Hierárquico

O *planejamento hierárquico* [GNT04, cap. 11], ou Rede Hierárquica de Tarefas (*Hierarchical Task Network*) (HTN) [EHN96], fornece uma forma particular de especificação de regras de controle de busca, aproveitando-se da natureza hierárquica inerente a muitos problemas práticos. Dentre as mais diversas áreas de aplicação, podemos citar seu uso em: *web* semântica [SPW⁺04], otimização de navegação de robôs [BHH03], geração automatizada de cursos baseada em tarefas pedagógicas [UII05], planejamento de operações militares [Wil88] e planejamento semi-automático para montagem dos foguetes Ariane-V da Agência Espacial Européia (ESA) [PV93]. Seu sucesso deve-se parcialmente ao fato de que *métodos* HTN fornecem um meio conveniente de representar “receitas” de soluções de problemas, correspondentes ao modo como especialistas de domínio humanos raciocinam ao planejar.

Assim como no planejamento clássico, cada estado do mundo é representado por um conjunto de átomos, e cada ação corresponde a uma transição de estados determinística. Também como no clássico, a entrada para o planejamento hierárquico inclui um conjunto de operadores. Adicionalmente, o projetista do domínio deve especificar diferentes *métodos* de decomposição de *tarefas*, junto com suas pré-condições para que sejam aplicáveis. Um método corresponde a um meio de *decompor* alguma tarefa, ou seja, expandí-la em um conjunto de sub-tarefas (tarefas menores). Um exemplo de decomposição pode ser visto na Figura 3.1, onde existe um método que decompõe a tarefa “construir casa” no conjunto de tarefas apresentado. Conjuntos de tarefas são representados através de *redes de tarefas*, que também expressam restrições sobre sua

ordem de execução, podendo ser tanto uma ordem total quanto uma ordem parcial; esta última permite maior expressividade, mas, em contrapartida, aumenta a complexidade do problema de planejamento. Uma rede de tarefas é composta por *ações primitivas* e *tarefas compostas* e assemelha-se a um plano parcialmente ordenado, por apresentar um conjunto de passos (tarefas) e restrições, mas difere delas por possuir ações abstratas (tarefas compostas), representando outras redes de tarefas. Diferentemente do planejamento clássico, no planejamento HTN, o objetivo é, ao invés de alcançar um estado meta, realizar um conjunto de tarefas, decompondo-as em sub-tarefas, através de métodos, até que a rede de tarefas possua apenas ações primitivas.

A função do planejador hierárquico é selecionar métodos para cada tarefa composta, decompondo-as de modo com que restrições sejam respeitadas. Esse processo é recursivo e termina quando a rede é composta apenas por ações primitivas, isto é, ações que não podem mais ser decompostas. Cada método aplicável a uma tarefa, substituindo-a por uma rede de tarefas, representa uma ramificação na árvore de busca do problema. Um planejador HTN irá aplicar um operador somente se orientado por um método a fazê-lo; portanto, cada método é parte da especificação de uma função de ramificação. O uso de hierarquias reduz drasticamente o espaço de busca, pois, ao invés de escolher ações individualmente, o planejador hierárquico utiliza caminhos inteiros como escolhas de possíveis componentes do plano, o que basicamente permite obter maior eficiência devido ao maior controle sobre o feixe de busca.

Há diferentes formalismos para planejamento hierárquico [Yan90] [EHN94b], pois as contribuições partiram de mais de um planejador, onde pode-se citar: *Nets Of Action Hierarchies* (NOAH) [Sac75], utilizando a linguagem *Semantics Of User Programs* (SOUP), com busca no espaço de planos e planos em ordem parcial; NONLIN [Tat77] com vínculos causais; SIPE-2 [Wil90], permitindo ordens arbitrárias de ações; O-Plan [CT91], permitindo maior modularidade e flexibilidade; *Universal Method Composition Planner* (UMCP) [EHN94c], com um sólido formalismo. Existe também uma abordagem híbrida [KMS98], para domínios parcialmente hierárquicos. Além disso, o poder de expressão do planejamento hierárquico é maior que o do planejamento clássico, permitindo que todo problema clássico seja representado por um hierárquico, onde o contrário não se aplica [EHN94a].

Existem também, além da decomposição hierárquica, outras diferentes abordagens

Figura 3.1: Exemplo de decomposição de tarefa composta.

para se efetuar o controle sobre o espaço de busca, incluindo poda de ações aplicáveis, através do uso de fórmulas em lógica temporal, expressando propriedades sobre caminhos possíveis. Dentre os planejadores baseados nestas técnicas, pode-se citar: TLPlan [BK00], utilizando Logica de Tempo Linear (*Linear Temporal Logic*) (LTL) [Pnu77], e TALplanner [DK99], utilizando Lógicas de Ação Temporal (*Temporal Action Logics*) (TAL) [DGKK98]. No Apêndice B é apresentado um meio de controle de busca que utiliza lógica CTL.

3.1 Planejamento com redes hierárquicas de tarefas

O formalismo para HTN desta dissertação é o mesmo utilizado por Erol et al. [EHN96]. O uso de *críticas* [EHNT95] é deixado de lado, pois certos tipos de críticas podem comprometer a corretude e a completude do algoritmo.

A linguagem $\mathcal{L}$ para planejamento HTN é uma linguagem de primeira ordem com algumas extensões, similar a NONLIN [Tat77]. O vocabulário de $\mathcal{L}$ é a 6-tupla $\langle \mathcal{V}, \mathcal{C}, \mathcal{P}, \mathcal{F}, \mathcal{T}, \mathcal{N} \rangle$, onde:

- $\mathcal{V}$ é um conjunto infinito de símbolos de variáveis;

- $\mathcal{C}$ é um conjunto finito de símbolos de constantes;
- $\mathcal{P}$ é um conjunto finito de símbolos de predicados;
- $\mathcal{F}$ é um conjunto finito de símbolos de *tarefas primitivas* (denotando ações);
- $\mathcal{T}$ é um conjunto finito de símbolos de *tarefas compostas*;
- $\mathcal{N}$ é um conjunto infinito de símbolos utilizados para rotular tarefas.

Todos os conjuntos de símbolos de $\mathcal{L}$ são mutuamente disjuntos.

Como no planejamento clássico, um estado é uma lista de átomos totalmente instanciados. Os átomos pertencentes à lista são verdadeiros no estado e os que não aparecem são falsos neste estado.

Um novo tipo de símbolo é um *símbolo de tarefa*. Todo símbolo de operador é um símbolo de tarefa primitiva, e existem alguns símbolos de tarefas adicionais chamados de *símbolos de tarefas compostas*. Uma *tarefa* é uma expressão da forma $t(r_1, \dots, r_k)$ tal que t é um símbolo de tarefa, e $r_1, \dots, r_k$ são termos. Se t é um símbolo de operador, então a tarefa é *primitiva*; caso contrário, a tarefa é *não-primitiva*. A tarefa é *totalmente instanciada* se todos os termos são totalmente instanciados; caso contrário, ela é *parcialmente instanciada*. Uma ação $a = (\text{nome}(a), \text{pre}(a), \text{efeitos}(a))$ *executa* uma tarefa primitiva totalmente instanciada t em um estado s se $\text{nome}(a) = t$ e a é aplicável a s .

Definição 3.1.1. Tipos de tarefas:

- Uma *tarefa primitiva* é uma construção sintática da forma executar $[f(r_1, \dots, r_k)]$, onde $f \in \mathcal{F}$ e $r_1, \dots, r_k$ são termos;
- Uma *tarefa meta* é uma construção sintática da forma obter $[l]$, onde l é um literal;
- Uma *tarefa composta* é uma construção sintática da forma realizar $[t(r_1, \dots, r_k)]$, onde $t \in \mathcal{T}$ e $r_1, \dots, r_n$ são termos. $\diamond$

Tarefas meta e tarefas compostas são ambas também denominadas *tarefas não-primitivas*.

Definição 3.1.2. Uma *rede de tarefas* é uma construção sintática da forma $[(n_1 : \alpha_1) \dots (n_m : \alpha_m), \phi]$, onde:

- Cada α_i é uma tarefa;
- $n_i \in \mathcal{N}$ é um *rótulo* para α_i (para distinguir diferentes ocorrências de α_i na rede);
- ϕ é uma fórmula booleana construída a partir de:
 - **restrições de substituição de variáveis**, tais como $(v = v')$ e $(v = c)$, onde $v, v' \in \mathcal{V}$ e $c \in \mathcal{C}$;
 - **restrições de ordem**, tais como $(n \prec n')$, onde $n, n' \in \mathcal{N}$, significando que a tarefa rotulada n deve preceder a tarefa rotulada n' ; Algumas restrições de ordem são também implícitas pela hierarquia de tarefas;
 - **restrições sobre estados**, tais como (n, l) , (l, n') e (n, l, n') , onde l é um literal e $v, v' \in \mathcal{V}$, significando que l deve ser verdadeiro no estado imediatamente posterior a n , imediatamente anterior a n' e em todos os estados entre n e n' , respectivamente. $\diamond$

Um vértice de um grafo direcionado com grau de entrada zero é chamado de *fonte* e um vértice com grau de saída zero é chamado de *sorvedouro*. Relações de precedência $(n \prec n')$ entre tarefas compostas implicam em um conjunto de restrições de ordem, dos sorvedouros para as fontes:

$$\{(n_s \prec n'_f) \mid n_s \in \text{sorvedouros}(n), n'_f \in \text{fontes}(n')\}.$$

Para que a decomposição de redes de tarefas possa proceder em qualquer ordem, faz-se necessário um mecanismo de manutenção de histórico. Este deve permitir a representação de restrições que o algoritmo de planejamento ainda não tenha analisado. A manutenção de históricos em HTN é feita através da representação de restrições na rede de tarefas.

Uma rede de tarefas contendo apenas tarefas primitivas é denominada *rede de tarefas primitiva*. Um *plano* é uma sequência π de tarefas primitivas totalmente instanciadas.

Definição 3.1.3. Um *operador* tem a forma:

$$[\text{operador } f(v_1, \dots, v_k)(\text{pre} : l_1, \dots, l_m)(\text{efeitos} : l'_1, \dots, l'_n)],$$

onde:

- f é um símbolo de tarefa primitiva;
- $l_1, \dots, l_m$ são literais descrevendo as condições para que f seja executável;
- $l'_1, \dots, l'_n$ são literais descrevendo os efeitos de f ;
- $v_1, \dots, v_k$ são os símbolos de variáveis que aparecem nos literais. ◇

Definição 3.1.4. Um *método* HTN é uma construção da forma (α, d) , onde α é uma tarefa não-primitiva e d é uma rede de tarefas. Significa que um modo de realizar a tarefa α é completar a rede de tarefas d , ou seja, realizar todas as sub-tarefas da rede de tarefas sem violar sua fórmula de restrição. ◇

Para cada tarefa meta existe, implicitamente, um método da forma

$$(\text{obter}[l], [(n : \text{executar}[f]), (l, n)]),$$

contendo apenas uma tarefa primitiva fictícia f , sem efeitos, e uma restrição de que o literal l seja verdadeiro no estado imediatamente anterior a executar $[f]$.

A descrição de métodos HTN omite a especificação de pré-condições, pois estas são representadas através de restrições do tipo (l, n) .

3.2 Planejamento com redes simples de tarefas

Esta seção descreve uma linguagem para uma versão simplificada, um caso restrito de planejamento HTN, denominada Rede Simples de Tarefas (*Simple Task Network*) (STN) [GNT04]. No planejamento STN, associa-se dois tipos de restrições a um método: restrições de ordem e de pré-condições. Representa-se restrições de ordem diretamente na rede de tarefas através das arestas de um dígrafo. Diferente de HTN, não se mantém histórico destas pré-condições na rede de tarefas: ao invés, pré-condições

são *forçadas* através da construção das redes de tarefas, de modo a satisfazê-las. Este é o propósito do conjunto de arestas $\{(v, u') \mid u' \in \text{succ}_1(u)\}$ na definição 3.2.1. O único tipo de procedimento de planejamento viável para uso desta técnica é a decomposição progressiva [GNT04].

As definições de termos, literais, operadores, ações e planos são as mesmas do planejamento clássico. A definição de $\gamma(s, a)$, o resultado da aplicação de uma ação a a um estado s , é também a mesma do planejamento clássico. No entanto, a linguagem inclui também tarefas, métodos e redes de tarefas, de um modo simplificado.

Definição 3.2.1. Uma *rede simples de tarefas* é um *grafo direcionado* (ou *dígrafo*) acíclico $w = (U, E)$ onde U é o conjunto de nós, E é o conjunto de arestas e cada nó $u \in U$ contém uma tarefa t_u . w é *totalmente instanciada* se todas as tarefas $\{t_u \mid u \in U\}$ são totalmente instanciadas; caso contrário, w é *parcialmente instanciada*. w é *primitiva* se todas as tarefas $\{t_u \mid u \in U\}$ são primitivas; caso contrário, w é *não-primitiva* (ou *composta*). $\diamond$

As arestas de w definem uma ordem parcial de U . Formalmente, $u \prec v$ se, e somente se, existe um caminho de u para v . Se a ordem parcial possui apenas uma linearização possível, ou seja, se é também uma ordem total, dizemos que w é *totalmente ordenada*. Neste caso, algumas vezes dispensa-se a notação de grafo para w e, ao invés, escreve-se w como a seqüência de tarefas $w = \langle t_1, \dots, t_k \rangle$, onde t_1 é a tarefa no primeiro nó de U , t_2 é a tarefa no segundo nó de U e assim por diante. Se w é totalmente instanciada, primitiva, totalmente ordenada e existem ações $a_1, \dots, a_k$ cujos nomes são $t_1, \dots, t_k$, então w corresponde ao plano $\pi_w = \langle a_1, \dots, a_k \rangle$.

Definição 3.2.2. Um *método STN* é uma 4-tupla da forma:

$$m = \langle \text{nome}(m), \text{tarefa}(m), \text{pre}(m), \text{rede}(m) \rangle,$$

onde:

- $\text{nome}(m)$ é o nome do método, uma expressão do tipo $m(x_1, \dots, x_k)$, onde $x_1, \dots, x_k$ são todos os símbolos de variáveis que ocorrem em m ;
- $\text{tarefa}(m)$ é uma tarefa não-primitiva;

- $\text{pre}(m)$ é um conjunto de literais especificando as *pré-condições* do método;
- $\text{rede}(m)$ é uma rede de tarefas contendo as *sub-tarefas* de m . $\diamond$

Na definição (3.2.2), $\text{nome}(m)$ tem o mesmo propósito do nome de um operador de planejamento clássico; permite referência desambígua a instâncias de substituição do método, sem que seja necessário descrever explicitamente as pré-condições e efeitos. $\text{tarefa}(m)$ diz a que tipo de tarefa m pode ser aplicado, $\text{pre}(m)$ especifica que condições o estado atual deve satisfazer para que m seja aplicável, e $\text{rede}(m)$ especifica as sub-tarefas a realizar para que complete $\text{tarefa}(m)$.

Um método m é *totalmente ordenado* se $\text{rede}(m)$ é totalmente ordenada. Neste caso, ao invés de utilizar um dígrafo, é possível utilizar uma lista $\text{sub-tarefas}(m) = \langle t_1, \dots, t_k \rangle$ como representação compacta de:

$$\text{rede}(m) = (\{u_1, \dots, u_k\}, \{(u_1, u_2), (u_2, u_3), \dots, (u_{k-1}, u_k)\}).$$

Definição 3.2.3. Uma instância de método m é *aplicável* a um estado s se $\text{pre}^+(m) \subseteq s$ e $\text{pre}^-(m) \cap s = \emptyset$. $\diamond$

Um planejador STN efetua uma busca progressiva no espaço de estados, guiada pela decomposição da rede de tarefas inicial (Figura 3.2). A decomposição aplica métodos a tarefas sem predecessores, até o nível de tarefas primitivas, e procura extrair planos válidos (Algoritmo 3).

Quando um método m é utilizado para decompor uma tarefa t , tal que t é parte de um nó u de uma rede de tarefas w , obtemos a nova rede de tarefas $\delta(w, u, m, \sigma)$ (Def. 3.2.4). Intuitivamente, u é removido de w , uma cópia w_u de $\text{sub-tarefas}(m)$ é inserida em w no lugar de u , e toda restrição de ordem que se aplicava a u passa a valer para todo nó de w_u .

Definição 3.2.4. Seja $w = (U, E)$ uma rede de tarefas, u um nó de w sem predecessores, e m um método relevante para t_u sob alguma substituição σ . Seja $\text{succ}(u)$ o conjunto de todos os sucessores imediatos de u , isto é, $\text{succ}(u) = \{u' \in U \mid (u, u') \in E\}$, e $\text{succ}_1(u)$ o conjunto de sucessores imediatos para os quais u é o único predecessor. Seja (U', E') o resultado da remoção de u e todas as arestas que contenham u e seja

Figura 3.2: Exemplo de decomposição de tarefas em ordem total.

(U_m, E_m) uma cópia de $\text{rede}(m)$. Se (U_m, E_m) não é vazio, então o resultado da decomposição de u em w sob σ é o conjunto de redes de tarefas:

$$\delta(w, u, m, \sigma) = \{(\sigma(U' \cup U_m), \sigma(E_v)) \mid v \in \text{sub-tarefas}(m)\},$$

onde

$$E_v = E_m \cup (U_m \times \text{succ}(u)) \cup \{(v, u') \mid u' \in \text{succ}_1(u)\}.$$

Caso contrário, $\delta(w, u, m, \sigma) = \{(\sigma(U'), \sigma(E'))\}$. ◇

Devido a essa estratégia progressiva, o planejador sempre mantém controle do estado atual do mundo (em tempo de planejamento) e pode se beneficiar desta informação (para extração de heurísticas, calcular custos de planos, entre demais usos [Ilg05]). Também é possível ver STN como uma busca regressiva, onde as tarefas correspondem às metas e a decomposição age como regressão.

O algoritmo para planejamento hierárquico em ordem parcial (Algoritmo 3) baseia-se nos três casos em que um plano $\pi_w = (a_1, \dots, a_k)$ é uma solução para um problema

Algoritmo 3: DECOMPOSIÇÃOORDEMPARCIAL ($s, w, \mathcal{D}, \mathcal{M}$)	
Entrada: Estado inicial s , Rede de tarefas w , Domínio $\mathcal{D}$, Conjunto de métodos $\mathcal{M}$	
Saída: Plano π	
1	início
2	se $w = \emptyset$ então devolva plano vazio;
3	não-deterministicamente escolha $u \in w$, sem predecessores em w ;
4	se t_u é uma tarefa primitiva então
5	$A \leftarrow \{(a, \sigma) \mid a \text{ é uma ação de } \mathcal{D}, \sigma \text{ é uma substituição tal que } \text{nome}(a) = \sigma(t_u) \text{ e } a \text{ é aplicável a } s\}$;
6	se $A = \emptyset$ então devolva falha;
7	não-deterministicamente escolha $(a, \sigma) \in A$;
8	$\pi \leftarrow \text{DECOMPOSIÇÃOORDEMPARCIAL}(\gamma(s, a), \sigma(w \setminus \{u\}), \mathcal{D}, \mathcal{M})$;
9	se $\pi = \text{falha}$ então devolva falha;
10	senão devolva $a.\pi$;
11	senão
12	$M \leftarrow \{(m, \sigma) \mid m \text{ é um método totalmente instanciado de } \mathcal{M}, \sigma \text{ é uma substituição tal que } \text{nome}(m) = \sigma(t_u) \text{ e } m \text{ é aplicável a } s\}$;
13	se $M = \emptyset$ então devolva falha;
14	não-deterministicamente escolha $(m, \sigma) \in M$;
15	não-deterministicamente escolha $w' \in \delta(w, u, m, \sigma)$;
16	devolva $\text{DECOMPOSIÇÃOORDEMPARCIAL}(s, w', \mathcal{D}, \mathcal{M})$;
17	fim

$\mathcal{P} = (s_0, w, \mathcal{D}, \mathcal{M})$:

1. Se w é vazia, então π é uma solução para $\mathcal{P}$ se π é vazio, ou seja, $k = 0$;
2. Se existe um nó de tarefa primitiva $u \in w$ que não tem predecessores em w , então π é uma solução para $\mathcal{P}$ se a_1 é aplicável a t_u em s_0 e o plano $\pi' = (a_2, \dots, a_k)$ é uma solução para o problema $\mathcal{P}' = (\gamma(s_0, a_1), w \setminus \{u\}, \mathcal{D}, \mathcal{M})$. Intuitivamente, $\mathcal{P}'$ é o problema produzido pela execução da primeira ação de π e remoção da tarefa correspondente de w ;
3. Existe um nó de tarefa não-primitiva $u \in w$ que não tem predecessores em w . Supondo que existe uma instância m de algum método em $\mathcal{M}$ tal que m é relevante para t_u e aplicável em s_0 , então π é uma solução para $\mathcal{P}$ se existe

uma rede de tarefas $w' \in \delta(w, u, m, \sigma)$ tal que π é uma solução para $\mathcal{P}' = (s_0, w', \mathcal{D}, \mathcal{M})$.

Se π é uma solução para $\mathcal{P} = (s_0, w, \mathcal{D}, \mathcal{M})$, então para cada nó de tarefa $u \in U$ existe uma *árvore de decomposição* cujas folhas são as ações de π .

3.3 Extensões

É relativamente fácil incorporar extensões ao planejamento hierárquico, visto que os procedimentos de decomposição STN planejam progressivamente a partir do estado inicial. Têm conhecimento em tempo de planejamento do estado atual do mundo a cada passo do planejamento. Isto torna fácil executar procedimentos arbitrários envolvendo o estado atual [GNT04, cap. 11], incluindo computações numéricas complexas, inferência axiomática e chamadas a pacotes de software específicos de domínio (tal como sistemas de bancos de dados e raciocinadores probabilísticos). Dentre os problemas envolvidos na incorporação de técnicas adicionais, pode-se citar:

- **Símbolos de Funções.** Se deixamos a linguagem $\mathcal{L}$ de planejamento HTN conter símbolos de funções, então os argumentos de um átomo ou tarefa não são mais restritos a ser símbolos constantes e símbolos variáveis. Ao invés, estes podem ser termos arbitrariamente complexos, fazendo com que o planejamento torne-se um problema indecidível.
- **Axiomas.** Para incorporar inferência axiomática, é necessário utilizar um provador de teoremas como uma sub-rotina do procedimento de planejamento. A abordagem mais simples é restringir os axiomas a cláusulas de Horn e utilizar um provador de teoremas baseado nestas.
- **Procedimentos Anexados.** Pode-se modificar o algoritmo de avaliação de pré-condições (que, no caso de se ter também inferência axiomática incorporada, seria o provador de teoremas em cláusulas de Horn) para reconhecer que certos termos ou símbolos de predicados devem ser avaliados através do uso de procedimentos anexados ao invés do uso do mecanismo normal de prova de teoremas.

3.3.1 Extensões adicionais

- **Efeitos de Alto Nível.** Alguns planejadores HTN permitem ao usuário declarar na descrição do domínio de planejamento que várias tarefas não-primitivas, ou os métodos para estas tarefas, alcançarão vários efeitos.
- **Pré-condições Externas.** Supondo que para decompor alguma tarefa t , decide-se usar algum método m . Além disso, supondo que existe alguma condição c tal que, não importando qual sub-plano seja produzido abaixo de m , pelo menos uma ação no sub-plano irá requerer c como pré-condição e o sub-plano não irá conter nenhuma ação que alcança c . Neste caso, para que o sub-plano apareça como parte de um plano solução π , a pré-condição c deve de algum modo ser alcançada em algum outro lugar em π . Portanto, dizemos que a pré-condição c é *externa* ao método m .
- **Tempo.** É possível generalizar processos de decomposição de tarefas para fazer certos tipos de planejamento temporal, por exemplo, lidar com ações que têm durações de tempo e podem se sobrepor entre si.
- **Grafos de Planejamento.** Esta extensão é diferente das demais por ser uma extensão do algoritmo de planejamento mas não da representação HTN: é possível modificar o algoritmo de decomposição para fazer uso de grafos de planejamento. O algoritmo modificado gera uma árvore de decomposição e um grafo de planejamento. O tamanho do grafo de planejamento é reduzido através da geração de somente aquelas ações que fazem par com tarefas na árvore de decomposição, e o tamanho da árvore de decomposição é reduzido através da decomposição de tarefas somente se o grafo de planejamento pode verificar todos os seus predecessores. [MK98]

3.3.2 Planejadores hierárquicos

A família *Simple Hierarchical Ordered Planner* (SHOP) [NCLMA99] de planejadores permite planejamento hierárquico utilizando o caso restrito de HTN denominado STN, em ordem total (SHOP) e em ordem parcial (SHOP2) [NAI⁺03]. Dentre suas propriedades interessantes está o fato de que a busca é feita progressivamente, de modo

que o planejador conheça sempre o estado atual. Isto torna possível um maior poder de expressão: axiomas, manipulação simbólica e numérica, execução de procedimentos externos durante o planejamento.

JSHOP2 [IN04] é uma re-implementação em Java de seu antecessor, SHOP2, escrito em Common Lisp. A principal diferença entre eles é uma fase adicional em JSHOP2, que, a partir da descrição de um domínio, gera código Java de um planejador especializado, dependente de domínio. Este planejador obtido é tira proveito de otimizações feitas pelo próprio compilador Java, tornando-o mais eficiente que sua versão interpretada, pois utiliza *arrays* de tamanho fixo como estruturas de dados para representar operadores e métodos. Resultados experimentais sugerem que o ganho obtido pela técnica de compilação é de aproximadamente até uma ordem de magnitude [IN04].

Os planejadores SHOP têm sido utilizados em aplicações diversas [NAI⁺05], como planejamento de evacuação, combate a incêndio em florestas, integração de sistemas de software, controle de veículos autônomos e seleção de materiais para manufatura, além de diversas aplicações à *web* semântica [WPS⁺03] [WSH⁺03] [SPW⁺04] [AKN05].

Esta fase intermediária de compilação pode ser vista como um processo de *especialização de programas*, através de sua *avaliação parcial* [Jon92]. Se um programa recebe mais de uma entrada e algumas destas variam com menos frequência que outras, então a especialização do programa em respeito a estas entradas produz uma versão mais específica e geralmente mais eficiente. No caso do JSHOP2, o domínio de planejamento é tratado como a entrada menos variável e os planejadores específicos de domínio são suas especializações.

O modelo de decomposição de tarefas de JSHOP2 é a base para a função EFETUA-DECOMPOSIÇÃO (Algoritmo 12) do algoritmo do planejador YoYo (Seção 5.2.1).

3.3.3 Vantagens e desvantagens

Comparados aos planejadores clássicos, a principal vantagem da abordagem HTN é sua sofisticada representação de conhecimento e raciocínio sob ações compostas. Eles podem representar e solucionar uma variedade de problemas de planejamento; com um bom conjunto de métodos de decomposição, eles podem solucionar problemas de planejamento clássico com maior eficiência. A principal desvantagem de planejadores

Figura 3.3: Processo de compilação de planejadores dependentes de domínio em JSHOP2.

HTN é que autores de domínios necessitam especificar um conjunto de métodos. O planejamento clássico necessita apenas do conjunto de operadores de planejamento.

Comparando HTNs com regras de controle [BK00], é difícil dizer que tipo de técnica de controle é mais efetiva. Planejadores hierárquicos têm sido mais utilizados em aplicações práticas, mas isto deve-se parcialmente ao fato de serem mais antigos. HTNs dão a um planejador informação sobre quais opções considerar, e regras de controle sobre quais opções *não* considerar. Os dois tipos de informação são úteis em diferentes situações e combiná-las pode ser vantajoso [SBTM02].

Capítulo 4

Planejamento sob Incerteza

Agentes raramente têm acesso à informação completa sobre o ambiente. Assim, um agente inteligente deve ser capaz de agir sob *incerteza*. A *decisão racional* depende tanto da importância relativa de vários objetivos a serem alcançados quanto da chance de que estes serão alcançados [RN03].

O *planejamento sob incerteza* relaxa as suposições que o planejamento clássico (Seção 2.2) faz sobre *efeitos das ações*, *observabilidade* e *tipos de metas*:

- Quanto ao efeito das ações, um problema de planejamento pode ser *determinístico* (como supõe o planejamento clássico), *probabilístico*, ou *não-determinístico* (também chamado de *possibilístico* [vR01] ou sob *incerteza Knightiana* [Kni21] [TCdB07]);
- A observabilidade pode ser, além de *total*, também *parcial*, onde apenas parte do estado do mundo pode ser conhecido pelo agente, ou até *nula*, quando nenhuma informação sobre o ambiente pode ser obtida;
- Metas podem ser de *alcançabilidade*, *estendidas* ou de *maximização de utilidade* (ou *otimização*).

4.1 Fontes de incerteza em planejamento

4.1.1 Não-determinismo nos efeitos de ações

Determinismo é uma visão simplificada do mundo que supõe que este evolui ao

longo de um único caminho, totalmente previsível dado o conhecimento de seu histórico [RN03]. Em alguns casos, modelar um sistema pressupondo determinismo pode ser uma abstração útil. Segundo a visão determinista, é necessário apenas estender propriamente um modelo para que se possa prever com antecedência os eventos que podem ocorrer no ambiente modelado. Por exemplo, com um modelo perfeito, o lançamento de um dado seria totalmente determinado.

Entretanto, a tentativa de tratar todo sistema como sendo determinístico é uma suposição irrealista e impraticável. A suposição de ações *não-determinísticas* assume uma postura mais realista. É mais útil modelar todos os possíveis resultados do lançamento de um dado e raciocinar sobre múltiplas possibilidades, sendo que nenhuma delas é certa. Pode ser melhor modelar o fato de que um componente pode parar ou falhar e planejar para tratamento de exceções ou mecanismos de recuperação.

O planejamento em domínios não-determinísticos (*i.e.*, domínios com ações não-determinísticas) apresenta uma característica que o difere do planejamento clássico: um plano pode resultar em múltiplos caminhos diferentes de execução. Como consequência, planos não devem ser especificados como uma sequência de ações, mas como *políticas*, que mapeiam estados em ações.

O não-determinismo pode ser também representado pela associação de probabilidades aos possíveis resultados de ações. Isto permite modelar o fato de que alguns resultados têm maior chance de ocorrer que outros. O planejamento sob incerteza Knightiana representa uma abordagem *qualitativa*, enquanto uma abordagem *quantitativa* leva em consideração distribuições de probabilidades nos efeitos de ações.

Em aplicações mais complexas, pode ser muito difícil obter as distribuições de probabilidades corretas, tanto por falta de dados estatísticos quanto pelo fato de que, em alguns casos, a meta é, garantir que certas condições ocorram. Nesta dissertação, o interesse é encontrar políticas para domínios com incerteza nos efeitos das ações, sem conhecer as distribuições de probabilidade. Dessa forma, procura-se encontrar soluções com diferentes garantias de alcance de metas.

4.1.2 Observabilidade parcial

Em várias aplicações, o estado de um sistema é apenas parcialmente observável em tempo de execução e, como consequência, diferentes estados do sistema são indistinguíveis do ponto de vista do controlador.

Muitas aplicações apresentam uma ou mais das seguintes situações: (i) algumas variáveis do sistema podem nunca ser observáveis pelo controlador; (ii) algumas variáveis podem ser observáveis somente em alguns estados ou após alguma ação de sensoriamento ser executada.

A principal consequência da observabilidade parcial é que as observações do agente não correspondem a um único estado. No planejamento sob incerteza Knightiana, a função de observação devolve um conjunto de estados, o que torna o espaço de busca o conjunto potência de $\mathcal{S}$, denominado conjunto de *estados de crença* [GNT04]. Assim, o problema de planejamento pode ser reformulado como planejamento no espaço de crenças sob observabilidade total e planos gerados são políticas que mapeiam estados de crença a ações. No caso do planejamento probabilístico, o espaço de crenças é representado através de distribuições de probabilidades sobre o conjunto de estados. Observações devolvem distribuições de probabilidades sobre conjuntos de estados, o que torna o espaço de busca potencialmente infinito. Esta dissertação trata apenas de problemas de planejamento sob observabilidade total.

4.2 Técnicas de planejamento sob incerteza

Russel e Norvig [RN03] classificam domínios quanto à sua quantidade de não-determinismo em quatro tipos de planejamento:

Não-determinismo Limitado. Neste caso específico de não-determinismo as ações podem ter efeitos imprevisíveis, mas os efeitos possíveis podem ser listados nos axiomas de descrição de ações. Existem dois tipos de planejamento sob não-determinismo limitado:

Planejamento Conformante. Também conhecido como planejamento sem sensores [CR00], este tipo de planejamento visa a construção de planos sequenciais que possam ser executados sem percepção. Estas soluções são

fortes sob observabilidade nula, ou seja, garantem atingir um estado meta a despeito do não-determinismo do domínio, para todos os estados iniciais determinados e todos os efeitos incertos de ações. Nenhuma ação deve ser executada em estados que não satisfaçam as pré-condições e qualquer estado que resulte da execução do plano garante ser um estado meta.

Planejamento Contingente. Lida com o não-determinismo limitado construindo um plano condicional com diferentes ramificações para as diferentes contingências que possam vir a ser necessárias durante a execução do plano. Também conhecido como condicional, permite superar a incerteza até certo ponto, mas apenas se ações de sensoriamento puderem obter a informação necessária, e somente se não houver muitas contingências diferentes.

Não-determinismo Ilimitado. Quando o conjunto de pré-condições ou efeitos possíveis é desconhecido ou grande demais para ser enumerado completamente, o não-determinismo é classificado como ilimitado. Existem duas técnicas para lidar com esse tipo de não-determinismo:

Monitoramento e Replanejamento. Neste caso específico, o agente pode usar qualquer das técnicas de planejamento precedentes para construir um plano, mas também utiliza o monitoramento de execução para julgar se o plano falha durante sua execução e precisa ser revisto. Presumivelmente, tais planos funcionarão a maior parte do tempo, mas surgirão problemas quando os acontecimentos contrariarem a teoria do agente, obrigando-o a buscar um novo plano;

Planejamento Contínuo. Projetado para persistir ao longo de inúmeras execuções, o planejamento contínuo pode lidar com o abandono e criação de novas metas, respeitando prioridades entre elas. Em alguns casos metas devem ser deixadas de lado até que uma nova meta — de maior prioridade e algumas vezes conflitante — seja atingida.

O assunto tratado nesta dissertação é classificado como planejamento contingente.

Figura 4.1: Exemplo de sistema de transição de estados não-determinístico.

4.3 Modelo formal

Definição 4.3.1. Um domínio de planejamento não-determinístico baseia-se em um *sistema de transição de estados não-determinístico* (Figura 4.1), uma 3-tupla $\Sigma'' = \langle \mathcal{S}, \mathcal{A}, \gamma \rangle$, onde:

- $\mathcal{S} = \{s_1, s_2, \dots\}$ é o conjunto finito ou recursivamente enumerável de estados;
- $\mathcal{A} = \{a_1, a_2, \dots\}$ é o conjunto finito ou recursivamente enumerável de ações;
- $\gamma : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ é a função de transição de estados. $\diamond$

Quando o ambiente de planejamento evolui deterministicamente, a cada ação executada, o agente sabe exatamente qual é o próximo estado do ambiente e, portanto, pode escolher com absoluta certeza qual será a próxima ação a ser executada [Per07]. Por outro lado, em ambientes não-determinísticos, o agente deve estar preparado para lidar com situações em que as coisas não acontecem conforme era esperado.

Planos sequenciais, como os utilizados no planejamento clássico, não são suficientemente expressivos para representar contingências. Assim, utiliza-se a noção de *política* π , que diz qual é a melhor ação, dado um determinado estado, definida como uma função parcial de estados a ações, $\pi : \mathcal{S} \mapsto \mathcal{A}$, podendo ser representada como um conjunto $\pi = \{(s_i, a_i)\}_{i=1}^k$, onde $k \geq 0$ é o tamanho da política. O conjunto de estados para os quais π tem alguma ação associada é denotado pelo conjunto $S_\pi = \{s \mid (s, a) \in \pi\}$.

Intuitivamente, uma política define um padrão de comportamento para o agente [Per07]. Estes comportamentos podem apresentar diferentes qualidades de garantia de alcance de metas (Seção 4.4).

Definição 4.3.2. A *estrutura de execução* Σ_π de uma política π é um grafo direcionado (ou *dígrafo*) no qual nós representam estados alcançáveis através de ações de π e arestas são possíveis transições de estados causadas por ações de π . Existe um caminho em Σ_π de s_1 até s_2 se e só se s_1 é π -ancestral de s_2 (s_2 é π -descendente de s_1). Se um nó n qualquer de Σ_π não possui descendentes, é dito que n é um *nó terminal*. $\diamond$

4.4 Planejamento forte, forte cíclico e fraco

Políticas, como soluções de problemas de planejamento, podem ser caracterizadas nas seguintes classes [CPRT03] [Per06]:

Soluções Fortes. São políticas que atingem a meta, garantidamente, a despeito do não-determinismo, ou seja, todos os caminhos no dígrafo Σ_π são finitos e atingem a meta. Formalmente, uma política π é uma solução *forte* para $\mathcal{P}$ se e só se Σ_π é acíclico e todo nó terminal em Σ_π está em S_g ;

Soluções Fortes Cíclicas. São políticas que atingem a meta, supondo que os ciclos sejam eventualmente interrompidos. Formalmente, uma política π é uma solução *forte cíclica* para $\mathcal{P}$ se e só se, para cada nó em Σ_π , existe um caminho até um nó terminal em Σ_π e todo nó terminal em Σ_π está em S_g ;

Soluções Fracas. São políticas que podem atingir a meta, mas não garantidamente. Formalmente, uma política π é uma solução *fraca* para $\mathcal{P}$ se e só se, para cada estado em S_0 , existe pelo menos um caminho que leva a um estado em S_g que é nó terminal em Σ_π .

Toda solução forte é também uma solução forte cíclica e toda solução forte cíclica é também uma solução fraca [CPRT03]. Soluções estritamente fracas e soluções fortes correspondem a dois casos extremos de satisfação de metas de alcançabilidade simples. Intuitivamente, soluções estritamente fracas correspondem a planos otimistas e soluções fortes correspondem a planos seguros. Entretanto, na prática, há casos em que soluções estritamente fracas não são aceitáveis e soluções fortes não existem. Nestes casos, soluções estritamente fortes cíclicas podem ser uma alternativa viável.

Exemplo 4.4.1. Utilizando a Figura 4.1 como exemplo e adotando s_1 como estado inicial e s_4 como estado meta, podemos classificar a política $\pi_1 = \{(s_1, \text{move}(l_1, l_4))\}$ como uma política forte cíclica: a aplicação da ação $\text{move}(l_1, l_4)$ no estado inicial s_1 pode levar o agente diretamente à meta s_4 ou de volta a s_1 , onde novamente esta ação será aplicada, sob a suposição de que eventualmente este ciclo de tentativa e erro se interrompa. Uma solução forte para o problema consiste na política $\pi_2 = \{(s_1, \text{move}(l_1, l_2)), (s_2, \text{move}(l_2, l_3)), (s_3, \text{move}(l_3, l_4)), (s_5, \text{move}(l_5, l_4))\}$. Esta política π_2 não é cíclica e sempre leva ao estado meta s_4 . Já uma solução fraca pode ser dada por $\pi_3 = \pi_2 \setminus \{(s_5, \text{move}(l_5, l_4))\}$, que não especifica nenhuma ação para o caso em que o agente possa ter acidentalmente alcançado s_5 . $\diamond$

Nos algoritmos seguintes, $S_{abertos}$ representa um conjunto de estados que ainda não foram considerados pelo planejador, permitindo com que possam operar sobre políticas parciais, ainda em construção.

4.4.1 Planejamento forte

Definição 4.4.2. A *pré-imagem forte* de um conjunto S de estados é definida como:

$$\text{PRÉIMAGEMFORTE}(S) = \{(s, a) \mid \forall s \in \mathcal{S}. \forall a \in \mathcal{A}. \gamma(s, a) \neq \emptyset \wedge \gamma(s, a) \subseteq S\}.$$

$\diamond$

Intuitivamente, a pré-imagem forte de um conjunto de estados S é o conjunto de todo estado s , tal que todos os seus sucessores (estados resultantes da aplicação de a em s) encontram-se em S .

No planejamento forte, uma política π deve apresentar um caminho acíclico até

um estado meta para todo estado alcançável a partir dos estados iniciais. Portanto, a estrutura de execução Σ_π deve ser acíclica. O algoritmo que verifica se uma política é forte (Algoritmo 4) efetua uma busca regressiva, começando a partir dos estados meta e estados em aberto, aplicando a função de pré-imagem forte (Definição 4.4.2) até alcançar os estados iniciais. A cada iteração são removidos pares de estado-ação de π que já foram verificados e pertencem à pré-imagem forte dos estados meta. Ao final da busca, se ainda existe algum par de estado-ação na política π , então a política induz um ciclo na estrutura de execução Σ_π e, portanto, não é uma solução forte.

Algoritmo 4: VERIFICAFORTE($\pi, S_{abertos}, S_g, S_0$)

Entrada: Política π , Conjunto de estados em aberto $S_{abertos}$, Conjunto de estados meta S_g , Conjunto de estados iniciais S_0

```

1 início
2    $S' \leftarrow \emptyset; S \leftarrow S_g \cup S_{abertos};$ 
3   enquanto  $S' \neq S$  faça
4      $S' \leftarrow S;$ 
5      $S \leftarrow S \cup \{s' \mid (s', a) \in \pi \text{ e } \gamma(s', a) \subseteq S\};$ 
6      $\pi \leftarrow \pi \setminus \{(s, a) \mid s \in S \text{ e } (s, a) \in \pi\};$ 
7   se  $S_0 \subseteq S$  e  $\pi = \emptyset$  então devolva verdadeiro;
8   devolva falso;
9 fim
```

4.4.2 Planejamento forte cíclico

O algoritmo que determina se uma política é forte cíclica (Algoritmo 5) efetua uma busca regressiva em Σ_π . Consiste em certificar-se de que todo par estado-ação na política parcial π é verificado pela busca. Ao final da busca, se existe algum par de estado-ação não removido de π , então π viola a suposição do planejamento forte cíclico de que ciclos sejam eventualmente interrompidos, ou seja, existe um ciclo induzido por π a partir do qual não há possibilidade de alcançar estados meta ou em aberto.

4.4.3 Planejamento fraco

Definição 4.4.3. A *pré-imagem fraca* de um conjunto S de estados é definida como:

$$\text{PRÉIMAGEMFRACA}(S) = \{(s, a) \mid \forall s \in S. \forall a \in \mathcal{A}. \gamma(s, a) \neq \emptyset \wedge \gamma(s, a) \cap S \neq \emptyset\}.$$

Algoritmo 5: VERIFICAFORTECÍCLICO($\pi, S_{abertos}, S_g, S_0$)

Entrada: Política π , Conjunto de estados em aberto $S_{abertos}$, Conjunto de estados meta S_g , Conjunto de estados iniciais S_0

```

1 início
2    $S' \leftarrow \emptyset; S \leftarrow S_g \cup S_{abertos};$ 
3   enquanto  $S' \neq S$  faça
4      $S' \leftarrow S;$ 
5      $S \leftarrow S \cup \{s' \mid (s', a) \in \pi \text{ e } S \cap \gamma(s', a) \neq \emptyset\};$ 
6      $\pi \leftarrow \pi \setminus \{(s, a) \mid s \in S \text{ e } (s, a) \in \pi\};$ 
7   se  $S_0 \subseteq S$  e  $\pi = \emptyset$  então devolva verdadeiro;
8   devolva falso;
9 fim

```

◇

Intuitivamente, a pré-imagem fraca de um conjunto de estados S é o conjunto de todo estado s , tal que pelo menos um de seus sucessores (resultantes da aplicação de a) encontre-se em S .

O algoritmo para verificar se uma política é fraca (Algoritmo 6) efetua uma busca por Σ_π por caminhos acíclicos, que, a partir de cada estado inicial $s \in S_0$ de $\mathcal{P}$, levam a um estado meta. Para isso, o algoritmo utiliza uma busca regressiva, a partir dos estados meta e dos estados em $S_{abertos}$, até os estados iniciais S_0 . Esta busca regressiva é baseada na função de pré-imagem fraca (Definição 4.4.3). A busca termina quando todos os estados que podem ser alcançados através da pré-imagem do objetivo e de $S_{abertos}$ estão em S . A este ponto, S contém todos os estados a partir dos quais existe um caminho acíclico até um estado meta. Depois disso, simplesmente verifica se os estados iniciais estão todos em S . Em caso afirmativo, a política parcial π não viola os requisitos do planejamento fraco.

4.5 Planejamento baseado em verificação de modelos

Planejamento baseado em verificação de modelos [GNT04] é uma abordagem para planejamento sob incerteza que lida com não-determinismo, observabilidade parcial e metas estendidas. A idéia principal é resolver problemas de planejamento na teoria

Algoritmo 6: VERIFICAFRACO($\pi, S_{abertos}, S_g, S_0$)	
Entrada: Política π , Conjunto de estados em aberto $S_{abertos}$, Conjunto de estados meta S_g , Conjunto de estados iniciais S_0	
1 início	
2	$S'' \leftarrow \emptyset; S \leftarrow S_g \cup S_{abertos};$
3	enquanto $S'' \neq S$ faça
4	$S'' \leftarrow S;$
5	$S' \leftarrow \{s' \mid (s', a) \in \pi \text{ e } \gamma(s', a) \cap S \neq \emptyset\};$
6	$\pi \leftarrow \pi \setminus \{(s, a) \mid s \in S \text{ e } (s, a) \in \pi\};$
7	$S \leftarrow S \cup S'';$
8	se $S_0 \subseteq S$ então devolva <i>verdadeiro</i> ;
9	devolva <i>falso</i> ;
10 fim	

dos modelos. Dados um sistema de transição de estados e uma fórmula temporal, planejamento baseado em verificação de modelos gera planos que controlam a evolução do sistema tal que todos os comportamentos do modelo façam com que a fórmula seja verdadeira. É baseado nas seguintes convenções:

- Um domínio de planejamento é um sistema de transição de estados não-determinístico, como na Definição 4.3.1.
- Fórmulas em lógica temporal expressam metas de alcançabilidade, ou seja, um conjunto desejado de estados meta, assim como metas temporais com condições em todos os caminhos de execução do plano.
- Planos resultam em comportamentos condicionais e iterativos (similares a programas com comandos condicionais e laços) e, em geral, são estritamente mais expressivos que planos que simplesmente mapeiam estados a ações a serem executadas.
- Dados um sistema de transição de estados e uma fórmula temporal, planejamento baseado em verificação de modelos gera planos que controlam a evolução do sistema tal que todos os comportamentos do modelo façam com que a fórmula seja verdadeira. A validação de planos pode ser formulada como um problema de verificação de modelos.

- Algoritmos de planejamento podem usar técnicas de verificação de modelos simbólicos. Em particular, conjuntos de estados são representados por fórmulas proposicionais, e busca no espaço de estados é executada através de transformações lógicas sobre estas fórmulas. Os algoritmos podem ser implementados através do uso de técnicas simbólicas, que se beneficiam de representações através de um Diagrama de Decisão Binária (*Binary Decision Diagram*) (BDD) [Bry92], permitindo a manipulação e armazenamento de estados e ações de forma compacta.

A principal vantagem do planejamento baseado em verificação de modelos é a habilidade de planejar sob incerteza de modo formal, dando garantias fortes sobre a validade dos planos gerados.

4.5.1 Verificação de modelos

Verificação de Modelos (VM) [MOSS99] é uma técnica de verificação formal de sistemas que visa provar propriedades com base na exploração exaustiva dos estados do sistema. O problema de verificação de modelos consiste em determinar se uma propriedade se mantém no sistema, devolvendo *sucesso* em caso afirmativo, ou, caso contrário, como resposta negativa, um *contra-exemplo* (e.g., um estado do modelo $\mathcal{K}$ onde a propriedade φ é violada). Mais formalmente, dados um modelo $\mathcal{K}$, descrevendo a dinâmica de um sistema, e uma fórmula φ , descrevendo uma propriedade, verificação de modelos consiste em resolver o problema $\mathcal{K} \stackrel{?}{\models} \varphi$.

Definição 4.5.1. Seja $\mathbb{P} \neq \emptyset$ um conjunto finito de proposições atômicas. Uma *estrutura de Kripke* sobre P é uma 3-tupla $\mathcal{K} = \langle \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$, onde:

- $\mathcal{S} \neq \emptyset$ é um conjunto finito de estados;
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ é uma *relação de acessibilidade* (ou *transição* de estados), onde $\forall s \in \mathcal{S}. \exists s' \in \mathcal{S}. (s, s') \in \mathcal{T}$;
- $\mathcal{L} : \mathcal{S} \mapsto 2^{\mathbb{P}}$ é uma *função de interpretação de estados*. $\diamond$

Intuitivamente, as proposições em $\mathbb{P}$ representam propriedades de estados e $\mathcal{L}(s)$ denota o conjunto de proposições que descrevem um estado s . Supõe-se que $\mathbb{P}$ contém as proposições $\top$ e $\perp$ e que, para todo $s \in \mathcal{S}$, temos $\top \in \mathcal{L}(s)$ e $\perp \notin \mathcal{L}(s)$.

Seja $\mathcal{K}$ uma estrutura de Kripke. Dizemos que $\mathcal{K}$ é *total* se, para todo $s \in S$, existe um $s' \in S$ tal que $(s, s') \in \mathcal{T}$; caso contrário, dizemos que $\mathcal{K}$ é *parcial*.

Estruturas de Kripke foram originalmente propostas como um modelo semântico para lógicas modais [Kri63], cujas fórmulas utilizam a modalidade $\Box$ para expressar *necessidade* e a modalidade dual, $\Diamond$, para expressar *possibilidade*. No contexto de verificação de modelos, entretanto, onde estruturas de Kripke são interpretadas como *modelos temporais* [Eme90], os estados numa estrutura de Kripke denotam instantes no tempo (*i.e.*, estados do mundo) e, portanto, as modalidades são utilizadas para expressar aspectos temporais: $\Box$ e $\Diamond$ têm, respectivamente, a semântica de *sempre* e *eventualmente*.

4.5.2 Domínios de planejamento

Definição 4.5.2. Um *domínio de planejamento qualitativo* (ou *estrutura de Kripke com ações*) com assinatura (P, A) é definido por uma tupla $\mathcal{D} = \langle S, \mathcal{T}, \mathcal{L} \rangle$, onde:

- $S \neq \emptyset$ é um conjunto finito de estados;
- $\mathcal{T} \subseteq S \times A \times S$ é uma *função de transição de estados*;
- $\mathcal{L} : S \mapsto 2^{\mathbb{P}}$ é uma *função de rotulação de estados*. $\diamond$

Um domínio de planejamento qualitativo difere de uma estrutura de Kripke (Definição 4.5.1) quanto à presença de ações rotulando transições entre estados, o que possibilita sua seleção por parte do planejador.

4.5.3 Verificação de modelos simbólicos

Modelos realistas necessitam de um enorme número de estados. Uma abordagem simbólica [JEK⁺90] para verificação de modelos utiliza fórmulas proposicionais para representar, de modo compacto, modelos de estados finitos e transformações sobre fórmulas. A exploração do espaço de estados utiliza então estas formas compactas para ganhar eficiência.

Diferente dos algoritmos tradicionais, que verificam estados individualmente, os algoritmos de verificação de modelos simbólicos efetuam busca regressiva sobre con-

juntos de estados e transições. Um tratado sobre planejamento em domínios não-determinísticos, utilizando verificação de modelos simbólicos, pode ser encontrado no trabalho de Roveri [Rov01].

4.5.4 Planejadores baseados em verificação de modelos

Model Based Planner (MBP) [BCP⁺01] é um planejador baseado em VM, considerado o estado da arte em planejamento sob incerteza Knightiana [BPR02]. MBP utiliza técnicas de verificação de modelos simbólicos para efetuar uma busca regressiva baseada em funções de pré-imagem de conjuntos de estados e manipula conjuntos de estados e ações na forma de BDDs, através da biblioteca CUDD [Som98], também considerada estado da arte em seu domínio de aplicação.

4.5.5 Limitações dessa abordagem

Como mostra Pereira [Per07], a lógica CTL [EC82], mais detalhada no Apêndice B, não é adequada para formalizar algoritmos para síntese de políticas para metas de alcançabilidade estendidas.

Definição 4.5.3 ([Per07]). Uma *meta de alcançabilidade estendida* é um par de fórmulas $\langle \varphi_1, \varphi_2 \rangle$, onde φ_1 especifica uma condição a ser preservada durante a execução de uma política e φ_2 é uma condição a ser alcançada ao final da execução dessa política.

Este tipo de meta pode ser utilizada como uma forma de controle de busca, onde φ_1 faz com que a visitação de estados não promissores possa ser evitada. TLPlan [BK00] e TALPlanner [DK99] utilizam outros tipos de lógicas temporais para especificar restrições sobre caminhos visitados.

Kuter et al. [KNRG07] sugerem que a utilização de conhecimento adicional sobre o domínio pode não ser apenas desejável, mas essencial à resolução de problemas de grande porte sob incerteza. Entretanto, definir φ_1 pode ser extremamente difícil em certos domínios onde é preferível especificar os estados desejados ao invés de estados a evitar. O planejamento hierárquico (Capítulo 3) fornece meios de especificar “caminhos” desejáveis na estrutura do domínio de planejamento qualitativo. O Capítulo 5 mostra como utilizar hierarquias de tarefas no tratamento de problemas de planejamento sob incerteza Knightiana.

Capítulo 5

Planejamento Hierárquico sob Incerteza

Em domínios de planejamento determinísticos, muito trabalho tem sido feito com o objetivo de melhorar a eficiência de planejadores, procurando prevení-los de visitar estados não promissores. A maior parte desses trabalhos concentra-se em planejadores que utilizam busca heurística progressiva no espaço de estados, tais como HSP [BG99], FF [HN01], TLPlan [BK00], TALplanner [DK99], e SHOP2 [NAI+03]. Estes planejadores conhecem sempre o estado atual do mundo, o que facilita o uso de algumas técnicas eficientes de poda do espaço de busca, especialmente em planejadores que permitem usar formas de poda específicas de domínio (isto é, heurísticas).

Kuter e Nau [KN04] apresentam um meio de transportar estas melhorias de eficiência para planejamento sob incerteza Knightiana. Dentre suas contribuições, pode-se citar uma técnica geral de *ND-transformação* de planejadores progressivos determinísticos em planejadores progressivos não-determinísticos. Este processo de transformação preserva a corretude e a completude do algoritmo original [KN04]. Um exemplo de uso desta técnica é descrito por Kuter et al. [KNPT05], que modifica o planejador hierárquico SHOP2, preparando-o para lidar com planejamento não-determinístico.

Neste capítulo, será mostrado passo a passo o processo de transformação de planejadores determinísticos em planejadores não-determinísticos. Chamaremos esse processo de ND-transformação (“*non-determinization*” [KN04]).

Figura 5.1: Exemplo de poda do espaço de busca através de ações compostas.

5.1 ND-transformação de planejadores progressivos

Kuter e Nau [KN04] descrevem uma técnica geral para produzir, a partir de planejadores progressivos para domínios determinísticos, suas versões não-determinísticas (*i.e.*, que tratam problemas com ações que apresentam efeitos não-determinísticos). Tal transformação deve passar pelas seguintes etapas:

1. Modificar o algoritmo de planejamento progressivo heurístico Planejamento Progressivo (*Forward Chaining Planning*) (FCP) (Algoritmo 7) para sintetizar políticas ao invés de planos sequenciais;
2. Modificar o FCP para a construção de políticas considerando os conjuntos de estados alcançados a partir de S_0 e para os quais não existem ações associadas, considerando apenas soluções fortes cíclicas;
3. Modificar o algoritmo para possibilitar a escolha de diferentes garantias de sucesso, isto é, além de buscar soluções fortes cíclicas, buscar também soluções fortes ou fracas.

A base para a técnica é composta de dois algoritmos de planejamento: FCP (Algoritmo 7), para domínios determinísticos, e um procedimento correspondente, com algu-

Algoritmo 7: FCP ($s_0, g, \mathcal{D}, \alpha$)**Entrada:** Estado inicial s_0 , Meta g , Domínio $\mathcal{D}$, Função de seleção de ações α **Saída:** Plano π

```

1 início
2    $\pi \leftarrow \emptyset; s \leftarrow s_0;$ 
3   repita
4     se  $s$  satisfaz  $g$  então devolva  $\pi;$ 
5      $A \leftarrow \{(s, a) \mid a \text{ é uma instância total de um operador em } \mathcal{D}, a \in \mathcal{A}(s) \text{ e}$ 
       $a \in \alpha(s)\};$ 
6     se  $A = \emptyset$  então devolva falha;
7     não-deterministicamente escolha  $(s, a) \in A;$ 
8      $\pi \leftarrow \pi \cup \{(s, a)\};$ 
9      $s \leftarrow \gamma(s, a);$ 
10  fim
11 fim

```

mas modificações, Planejamento Progressivo Não-Determinístico (*Non-Deterministic Forward Chaining Planning* (ND-FCP)) (Algoritmo 8).

O propósito da utilização de procedimentos abstratos é permitir mostrar a ND-transformação de forma mais clara, isto é, mostrar como planejadores progressivos podem ser modificados, de forma sistemática, para lidar com ações não-determinísticas.

Formalmente, um planejador Λ é uma *instância* de FCP se existe uma substituição de certas partes do procedimento FCP de modo que este se comporte como Λ . Se um planejador Λ pode ser descrito como uma instância do procedimento FCP (um *algoritmo progressivo determinístico*), a instância correspondente ND- Λ de ND-FCP (*algoritmo progressivo não-determinístico*) será a ND-transformação cíclica forte de Λ . ND-SHOP2 [KN04] é resultante da ND-transformação de SHOP2 [NAI+03].

Em instâncias de FCP, a relação de satisfação de metas e a função α variam, de acordo com o algoritmo de planejamento Λ usado como base. A relação *ssatisfaz* depende de Λ e pode ir além da simples verificação $s \in S_g$. A *função de seleção de ações* α é utilizada para podar o espaço de busca: devolve um sub-conjunto das ações $\mathcal{A}(s)$ aplicáveis a s , que são denominadas ações que *satisfazem* $\alpha(s)$. Alguns exemplos de instâncias de FCP podem ser:

- No caso mais simples, planejamento STRIPS, $\alpha(s)$ apenas devolve as ações a aplicáveis a s e a relação de satisfação $s \in S_g$ se mantém, onde g é substituído por S_g ;
- Já em SHOP2 [NAI⁺03], a busca é controlada pelo conjunto de métodos utilizados para decompor sub-tarefas, e $\alpha(s)$ nos procedimentos denota o conjunto de ações que podem ser produzidas por estes métodos. Em casos com metas puramente de decomposição, qualquer s , se alcançado através das decomposições, satisfaz g . Em outras palavras, se a rede de tarefas w alcançada junto a s é vazia, então s satisfaz g ;
- Outro planejador que pode ser utilizado é TLPlan [BK00], que tem a busca controlada por uma fórmula ϕ especificada em LTL [Pnu77], uma lógica temporal de tempo linear. Para cada estado s , $\alpha(s)$ é o conjunto de todas as ações a aplicáveis a s , tal que o estado sucessor $\gamma(s, a)$ satisfaça ϕ .

Um ponto importante a notar é que a função α pode vir a carregar consigo um estado interno. No caso do planejamento hierárquico, a manutenção da hierarquia de decomposição corrente é de responsabilidade de α , que pode mudar durante a escolha não-determinística. Por se tratar de procedimentos abstratos, FCP e ND-FCP mantêm este fato implícito.

FCP (Algoritmo 7) começa com um estado inicial s_0 e efetua uma busca progressiva (Algoritmo 1), explorando outros estados, sucessivamente, através da escolha e aplicação de ações. A busca termina quando um estado que satisfaz a meta g é alcançado. Ao invés de examinar todas as ações aplicáveis a s , FCP considera somente o sub-conjunto de ações que satisfazem $\alpha(s)$.

Como FCP, ND-FCP (Algoritmo 8) também planeja progressivamente a partir dos estados pertencentes ao conjunto de estados iniciais S_0 , mas o critério de sucesso é mais complexo. Em FCP, um plano π é uma solução se seu estado final satisfaz a condição meta g . Para que uma política em ND-FCP seja uma solução, *todo* estado na estrutura de execução de π deve ter pelo menos um π -descendente que satisfaz a meta. Estados que já têm um caminho em Σ_π que leva a um estado meta são denotados estados *resolvidos*. Do contrário, são chamados de estados *em aberto*. A busca termina

quando todos os estados alcançáveis são resolvidos, se existir uma política solução. Caso contrário, devolve *falha*.

Algoritmo 8: ND-FCP ($S_0, g, \mathcal{D}', \alpha'$)

Entrada: Conjunto inicial S_0 , Meta g , Domínio $\mathcal{D}'$, Função de seleção de ações α'

Saída: Plano π

```

1 início
2    $\pi \leftarrow \emptyset; S \leftarrow S_0; S_{resolvidos} \leftarrow \emptyset;$ 
3   repita
4     se  $S = \emptyset$  então devolva  $\pi;$ 
5     selecione um estado  $s \in S$  e remova-o de  $S;$ 
6     se  $s$  satisfaz  $g$  então insira  $s$  em  $S_{resolvidos};$ 
7     senão se  $s \notin S_\pi$  então
8        $A \leftarrow \{(s, a) \mid a \text{ é uma instância total de um operador em } \mathcal{D}',$ 
9          $a \in \mathcal{A}(s) \text{ e } a \in \alpha'(s)\};$ 
10      se  $A = \emptyset$  então devolva falha;
11      não-deterministicamente escolha  $(s, a) \in A;$ 
12       $\pi \leftarrow \pi \cup \{(s, a)\};$ 
13       $S \leftarrow S \cup \gamma(s, a);$ 
14    senão se  $s$  não tem  $\pi$ -descendentes em  $(S \cup S_{resolvidos}) \setminus S_\pi$  então
15      devolva falha ;
16  fim
17 fim

```

5.1.1 Propriedades teóricas

Definição 5.1.1. Um domínio de planejamento é *fortemente conexo* se, para dois estados s e s' quaisquer, existe uma seqüência de ações que alcança s' se executada a partir de s . $\diamond$

Em suma, domínios fortemente conexos não possuem ações irreversíveis. Qualquer ND-transformação de um domínio deste tipo será também fortemente conexa.

Um teorema de Kuter e Nau [KN04] prova que se o número de possíveis sucessores de cada estado é limitado por uma constante, então ND- Λ encontra políticas em tempo polinomial em relação ao tamanho da política solução.

Teorema 5.1.2 ([KN04]). *Seja Λ uma instância de FCP. Suponha que Λ encontre planos seqüenciais em tempo $O(p(|\pi|))$ em domínios de planejamento fortemente conexos, onde $|\pi|$ é o tamanho do plano solução e p é uma função monotônica. Então, $ND\text{-}\Lambda$ encontra políticas em tempo $O(p(|\Sigma_{\pi'}|))$, onde $|\Sigma_{\pi'}|$ é o tamanho da estrutura de execução da política devolvida por $ND\text{-}\Lambda$.* $\diamond$

O resultado de complexidade é válido também para domínios de planejamento que não são fortemente conexos.

Teorema 5.1.3 ([KN04]). *Seja Λ uma instância de FCP e suponha que o tempo de execução de Λ seja $O(p(|\pi|))$, onde π é o tamanho do plano solução devolvido por Λ , e p seja uma função monotônica. Então, $ND\text{-}\Lambda$ encontra políticas em tempo médio $O(p(n) + \frac{xdn}{y})$, onde $n = |\Sigma_{\pi'}|$ é o tamanho da estrutura de execução da política solução π' devolvida por $ND\text{-}\Lambda$, x é o número máximo de pares estado-ação que são adicionados a qualquer política depois que $ND\text{-}\Lambda$ visita um estado beco-sem-saída, y é o número máximo de ações aplicáveis a qualquer estado, e em todo estado s , $0 \leq d \leq y$ é o número máximo de ações aplicáveis a s que levam a estados beco-sem-saída.* $\diamond$

5.2 ND-transformação de planejadores HTN progressivos

YoYo [KNPT05] é um algoritmo de planejamento hierárquico progressivo que lida com ações não-determinísticas. Sua vantagem sobre $ND\text{-}SHOP2$ [KN04] é que combina o planejamento hierárquico com técnicas de verificação de modelos simbólicos. YoYo pode utilizar BDDs [Bry92] para representar um conjunto de estados. Esta forma de representação busca obter maior eficiência, pois permite tratar transições de conjuntos de estados a seus sucessores de uma só vez. O uso de BDDs em YoYo é discutido na Seção 5.3.

5.2.1 Algoritmo para planejamento forte cíclico

A entrada para YoYo (Algoritmo 9) consiste no problema de planejamento $\mathcal{P} = \langle \mathcal{D}, S_0, S_g \rangle$ em um domínio não-determinístico $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \gamma \rangle$, uma rede de tarefas inicial w e um conjunto $\mathcal{M}$ de métodos para o domínio $\mathcal{D}$. O algoritmo explora tuplas da forma (S, w) , denominadas *situações*, que são resolvidas pela decomposição da rede de tarefas w nos estados de S .

Algoritmo 9: YoYo($\mathcal{D}, S_0, S_g, w, \mathcal{M}$)

Entrada: Domínio de planejamento $\mathcal{D}$, Conjunto de estados iniciais S_0 ,
Conjunto de estados meta S_g , Rede de tarefas w , Conjunto de
métodos $\mathcal{M}$

Saída: Plano π

```

1 início
2   devolva YoYoAux( $\mathcal{D}, \{(S_0, w)\}, S_g, \mathcal{M}, \emptyset, \{(S_0, w)\}$ );
3 fim

```

Algoritmo 10: YoYoAux($\mathcal{D}, X, S_g, \mathcal{M}, \pi, x_0$)

Entrada: Domínio de planejamento $\mathcal{D}$, Conjunto de situações X , Conjunto de
estados meta S_g , Conjunto de métodos $\mathcal{M}$, Política parcial π ,
Situação inicial x_0

Saída: Plano π

```

1 início
2    $X \leftarrow \text{PODASITUACÕES}(X, S_g, \pi)$ ;
3   se existe uma situação  $(S, w = \emptyset) \in X$  tal que  $S \not\subseteq S_g$  então devolva falha;
4   se VERIFICAPOLÍTICA( $\pi, X, S_g, x_0$ ) então devolva falha;
5   se  $X = \emptyset$  então devolva  $\pi$ ;
6   escolha uma situação  $(S, w)$  de  $X$  e remova-a;
7    $F \leftarrow \text{EFETUADECOMPOSIÇÃO}(S, w, \mathcal{D}, \mathcal{M})$ ;
8   se  $F = \emptyset$  então devolva falha;
9    $X' \leftarrow \text{ENCONTRASUCESSORES}(F, X)$ ;
10   $\pi' \leftarrow \pi \cup \{(s, a) \mid (S', a, w') \in F \text{ e } s \in S'\}$ ;
11   $\pi \leftarrow \text{YoYoAux}(\mathcal{D}, X', S_g, \mathcal{M}, \pi', x_0)$ ;
12  se  $\pi = \text{falha}$  então devolva falha;
13  devolva  $\pi$ ;
14 fim

```

Começando a partir da situação inicial (S_0, w) , YoYo recursivamente gera sucessivos conjuntos de situações até que uma solução para o problema de planejamento dado seja encontrada (Figura 5.2). A cada iteração do processo de planejamento YoYo primeiro examina o conjunto X de situações à procura de ciclos e estados meta, utilizando PODASITUACÕES. Para cada situação $(S, w) \in X$, PODASITUACÕES verifica o conjunto de estados S e remove qualquer estado que já apareça na política (já possuindo, portanto, uma ação planejada para tal) ou no conjunto de estados meta (e, portanto,

Algoritmo 11: PODASITUAÇÕES(X, S_g, π)**Entrada:** Conjunto de situações X , Conjunto de estados meta S_g , Política parcial π **Saída:** Conjunto simplificado de situações X'

```

1 início
2    $X' \leftarrow \emptyset$ ;
3   para toda situação  $(S, w) \in X$  faça
4      $S' \leftarrow S \setminus (S_g \cup S_\pi)$ ;
5     se  $S' = \emptyset$  então  $X' \leftarrow X' \cup \{(S', w)\}$ ;
6   devolva  $X'$ ;
7 fim

```

a discussão é deixada para a próxima seção.

Se a política parcial π não adequa-se aos requisitos de uma solução para o problema de entrada, então YoYo retorna do feixe de busca corrente com *falha*. Caso contrário, e se não há mais situações abertas a serem exploradas, π é uma solução para o problema de planejamento. Isto deve-se ao fato de que π não viola os requisitos do problema de entrada, pois passou pela verificação de VERIFICAPOLÍTICA no passo anterior, e não tem mais tarefas para decompor.

Caso ainda existam situações abertas a explorar, então YoYo seleciona uma delas, (S, w) e procura planejar uma ação para cada estado em S . EFETUADECOMPOSIÇÃO, basicamente um planejador STN, é responsável por esta operação, como segue: em uma situação (S, w) , seja t uma tarefa que não tem predecessores em w . Se t é uma tarefa primitiva, então t pode ser executada diretamente no ambiente. Seja a uma ação correspondente a t , e a possa ser aplicada a cada estado de S , ou seja, $S \subseteq S_a$. Ressalta-se o fato de que aplicar uma ação a a um conjunto de estados S não gera nenhuma nova situação aberta: S deve ser um sub-conjunto de S_a , pois, do contrário, existe pelo menos um estado em S ao qual nenhuma ação é aplicável, e este é um ponto de falha no planejamento.

Se t não é primitiva, então sucessivamente aplica-se métodos às tarefas não-primitivas em w , até que uma ação seja gerada. Supondo a escolha da aplicação de um método m a t , surgem duas possíveis situações:

Algoritmo 12: EFETUADECOMPOSIÇÃO($S, w, \mathcal{D}, \mathcal{M}$)	
Entrada: Conjunto de estados S , Rede de tarefas w , Domínio de planejamento $\mathcal{D}$, Conjunto de métodos $\mathcal{M}$	
Saída: Conjunto de controle F	
1	início
2	$F \leftarrow \emptyset; X \leftarrow \{(S, w)\};$
3	repita
4	se $X = \emptyset$ então devolva F ;
5	escolha uma tupla $(S, w) \in X$ e remova-a;
6	escolha uma tarefa t que não tenha predecessores em w ;
7	se t <i>é uma tarefa primitiva</i> então
8	$A \leftarrow \{a \mid a \in \mathcal{A} \text{ é uma ação para } t, \text{ e } S \subseteq S_a\};$
9	se $A = \emptyset$ então devolva $\emptyset$;
10	escolha uma ação a de A ;
11	$F \leftarrow F \cup \{(S, a, w \setminus \{t\})\};$
12	senão
13	$M \leftarrow \{m \mid m \text{ é um método em } \mathcal{M} \text{ para } t, \text{ e } S \cap S_m \neq \emptyset\};$
14	se $M = \emptyset$ então devolva $\emptyset$;
15	escolha um método m de M ;
16	$X \leftarrow X \cup \{(S \cap S_m, (w \setminus \{t\}) \cup w')\};$
17	se $S \setminus S_m \neq \emptyset$ então $X \leftarrow X \cup \{(S \setminus S_m, w)\};$
18	fim
19	fim

1. A situação que surge da decomposição de t por m nos estados $S \cap S_m$, nos quais m é aplicável;
2. A situação que especifica os estados nos quais m não é aplicável, isto é, as situações $(S \setminus S_m, w)$.

No primeiro caso, procede-se com a decomposição de sub-tarefas de t como especifica m . No segundo caso, por outro lado, outros métodos para t devem ser utilizados. Se não existem outros métodos para t em situações como $(S \setminus S_m, w)$, então EFETUADECOMPOSIÇÃO devolve o conjunto vazio, fazendo YoYo reportar falha.

EFETUADECOMPOSIÇÃO devolve um conjunto F da forma $\{(S_i, a_i, w_i)\}_{i=0}^k$. Se $F = \emptyset$, então o processo de decomposição falha, já que existe um estado $s \in S$ tal que

não é possível escolher uma ação para s utilizando os métodos provenientes do domínio de planejamento. Se $F \neq \emptyset$ então a rotina gerou uma ação a_i para cada estado em S , ou seja, $S = \bigcup_i S_i$, e uma rede de tarefas w_i a ser completada após a aplicação desta ação.

Supondo que EFETUADECOMPOSIÇÃO tenha devolvido um conjunto F não-vazio de tuplas da forma (S', a, w') , YoYo deve proceder computando as situações sucessoras a explorar, utilizando EFETUADECOMPOSIÇÃO como segue: para cada tupla $(S', a, w') \in F$, primeiro gera-se o conjunto de estados possíveis da aplicação de a a S' a partir da função:

$$\text{SUCC}(S', a) = \{s'' \mid s' \in S' \text{ e } (s', a, s'') \in \gamma\}, \quad (5.1)$$

onde γ é a relação de transição de estados do domínio de planejamento. A situação seguinte à aplicação de ações é definida como $(\text{SUCC}(S', a), w')$.

Algoritmo 13: ENCONTRASUCESSORES(F, X)
<p>Entrada: Conjunto F de ações e redes de tarefas para estados, Conjunto de situações X</p> <p>Saída: Conjunto de próximas situações X'</p> <p>1 início</p> <p>2 $X' \leftarrow X \cup \{(\text{SUCC}(S, a), w) \mid (S, a, w) \in F\};$</p> <p>3 $X' \leftarrow \{(\text{COMPÕE}(w, X'), w) \mid (S, w) \in X'\};$</p> <p>4 devolva X';</p> <p>5 fim</p>

Uma vez que YoYo tenha gerado o conjunto de todas as situações sucessoras a explorar, são compostas as novas situações, provenientes das redes de tarefas especificadas. Formalmente,

$$\text{COMPÕE}(w, X) = \{s \mid (S, w) \in X \text{ e } s \in S\}. \quad (5.2)$$

A composição de um conjunto de situações é um passo de otimização do processo de planejamento. A progressão de situações abertas pode criar um conjunto de situações nas quais mais de uma situação pode especificar a mesma rede de tarefas.

A composição de tais situações não é requerida para a corretude, mas tem a vantagem de permitir utilizar representações mais compactas, pelo fato de que estes estados geralmente possuem características comuns.

A poda causada pelo uso de planejamento hierárquico pode em alguns casos ignorar planos que seriam válidos para problemas com metas de alcançabilidade simples. Isso pode fazer com que, dependendo dos métodos de decomposição utilizados, planos com garantias mais fortes não sejam encontrados. Por exemplo, MBP pode achar uma solução forte para um problema em que ND-SHOP só encontra soluções fortes cíclicas. O contrário não se aplica.

A tese de [Kut06] apresenta uma generalização das técnicas de ND-transformação e representações simbólicas, chamada de Separação de Espaço de Estados Progressiva (*Forward State-Space Splitting*) (FS^3). O algoritmo FS^3_{SHOP} , uma instância de FS^3 que utiliza planejamento STN, é similar ao empregado pelo planejador YOYO e sua implementação é apresentada, a título de demonstração, no Apêndice A.

5.2.2 Algoritmo para planejamento forte

No planejamento forte, uma política deve induzir uma estrutura de execução até um estado meta, a partir de todo estado alcançável pela aplicação de ações da política, começando pelos estados iniciais. Além disso, não deve haver ciclos nessa estrutura de execução. Esta condição pode ser verificada através de VERIFICAPOLÍTICA_FORTE, construída a partir da função de pré-imagem regressiva forte de Cimatti et al. [CPRT03]. Iniciando a partir do conjunto de estados nas situações abertas e os estados meta, calcula o conjunto de estados na política a partir da qual um estado aberto ou um estado meta é alcançável. Durante o processo, remove os pares estado-ação para estes estados calculados pela pré-imagem regressiva forte. Ao final do processo, se resta algum par estado-ação na política, isto significa que a política induz um ciclo na estrutura de execução e, portanto, não pode ser uma solução forte para o problema de planejamento.

VERIFICAPOLÍTICA_FORTE utiliza uma função denominada ESTADOS, que devolve o conjunto de todos os estados que aparecem em um dado conjunto de situações. Mais formalmente:

$$\text{ESTADOS}(X) = \{s \mid (S, w) \in X \text{ e } s \in S\}. \quad (5.3)$$

Algoritmo 14: VERIFICAPOLÍTICA_FORTE(π, X, S_g, x_0)

```

1 início
2    $S' \leftarrow \emptyset$ ;
3    $S_0 \leftarrow \text{ESTADOS}(x_0)$ ;
4    $S \leftarrow S_g \cup \text{ESTADOS}(X)$ ;
5   enquanto  $S' \neq S$  faça
6      $S' \leftarrow S$ ;
7      $S \leftarrow S \cup \{s' \mid (s', a) \in \pi \text{ e } \gamma(s', a) \subseteq S\}$ ;
8      $\pi \leftarrow \pi \setminus \{(s, a) \mid s \in S \text{ e } (s, a) \in \pi\}$ ;
9   se  $S_0 \subseteq S$  e  $\pi = \emptyset$  então devolva falso;
10  devolva verdadeiro;
11 fim

```

5.3 Utilização de BDDs

Um Diagrama de Decisão Binária (*Binary Decision Diagram*) (BDD) [Bry92] representa uma fórmula booleana como um grafo direcionado acíclico (dígrafo), com raiz (apenas um nó sem predecessores), onde nós internos (não-terminais) correspondem a variáveis da função e nós terminais são rotulados com os valores $\perp$ ou $\top$. Cada nó interno v é rotulado por uma variável $var(v)$ e possui arestas ligando-o a dois outros nós: $lo(v)$, que representa o caso em que a variável é falsa, e $hi(v)$ caso contrário. Para uma dada valoração, o valor-verdade da função é determinado através de um caminho a partir da raiz até um nó terminal, seguindo as arestas indicadas pelos valores das variáveis. O valor da função é então dado pelo valor do nó terminal.

É possível redefinir o algoritmo YoYo em termos de representações de conjuntos de estados na forma de BDDs para tirar proveito da representação simbólica e compacta que resulta do uso de BDDs [KNPT05]. É importante notar que, em alguns casos, a representação por BDDs pode acabar sendo tão verbosa quanto a enumeração explícita de conjuntos, devido à ordem das variáveis. Entretanto, pacotes (ou bibliotecas) que lidam com BDDs geralmente cuidam transparentemente da reordenação de variáveis. A Figura 5.3 apresenta duas BDDs resultantes da fórmula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow$

Figura 5.3: BDDs resultantes de diferentes ordens de variáveis para a mesma fórmula.

b_3) [Rov01].

5.3.1 Representação de problemas de planejamento com BDDs

Em planejamento, um vetor $\bar{s}$ de proposições representa o estado atual do ambiente. Um *estado* é uma valoração de $\{\perp, \top\}$ para cada proposição em $\bar{s}$. Baseado nesta formulação, um conjunto de estados S corresponde à fórmula $S(\bar{s})$, tal que:

$$S(\bar{s}) = \bigvee_{s \in S} s(\bar{s}).$$

Outro vetor $\bar{s}'$ de variáveis proposicionais é utilizado para representar os estados sucessores do ambiente. Similarmente, utiliza-se um vetor $\bar{a}$ de variáveis de ações para representar um conjunto de ações. Uma política π pode ser representada pela fórmula $\pi(\bar{s}, \bar{a})$ nas variáveis $\bar{s}$ e $\bar{a}$. Denota-se um conjunto de estados S com uma fórmula $S(\bar{s})$ no vetor de estados $\bar{s}$. Representa-se uma situação como um par da forma $(S(\bar{s}), w)$,

onde w é uma rede de tarefas.

A situação inicial pode ser representada por $x_0 = \{(I(\bar{s}), w)\}$, onde $I(\bar{s})$ representa o conjunto de estados iniciais e w é a rede de tarefas inicial. Similarmente, o conjunto de estados meta é representado pela fórmula $S_g(\bar{s})$ e as relações de transição de estados por $R(\bar{s}, \bar{a}, \bar{s}')$.

As operações de desigualdade entre conjuntos, diferença de conjuntos e relação de sub-conjuntos constituem as primitivas básicas utilizadas em condicionais nos algoritmos da seção 5.2.1. Estas operações podem ser codificadas em termos de manipulações lógicas nas fórmulas descritas anteriormente.

O resultado da aplicação de uma ação a ao conjunto de estados S pode ser representado pela fórmula: $\exists \bar{s}' : S(\bar{s}) \wedge R(\bar{s}, \bar{a}, \bar{s}') [\bar{s}', \bar{s}]$, onde $[\bar{s}', \bar{s}]$ é chamada de operação de *deslocamento progressivo*. Esta fórmula representa a função $\text{SUCC}(S, a)$.

A função ESTADOS (5.3) pode ser representada pelo operador de união de conjuntos sobre as situações de interesse. Em particular, buscar o conjunto de todos os estados de $X = \{x_1, x_2, \dots, x_n\}$ corresponde à fórmula $S_1(\bar{s}) \vee S_2(\bar{s}) \vee \dots \vee S_n(\bar{s})$, onde $x_i = (S_i, w_i)$. PODASITUAÇÕES (Algoritmo 11) pode ser representada como: $S(\bar{s}) \wedge \neg (S_g(\bar{s}) \vee \exists \bar{a} : \pi(\bar{s}, \bar{a}))$.

As funções VERIFICAPOLÍTICA para planejamento forte (Algoritmo 4) e forte cíclico (Algoritmo 5) são baseadas em duas primitivas para o cálculo pré-imagens fracas e fortes de um conjunto particular de estados. Estes cálculos correspondem a $\exists \bar{a} \exists \bar{s}' . \pi(\bar{s}, \bar{a}) \wedge S(\bar{s}')$ e $\exists \bar{a} \exists \bar{s}' . \pi(\bar{s}, \bar{a}) \implies S(\bar{s}')$, respectivamente.

ENCONTRASUCESSORES (Algoritmo 13) corresponde às fórmulas $S(\bar{s}) \implies S_a(\bar{s})$ e $S(\bar{s}) \wedge S_m(\bar{s})$, onde $S(\bar{s})$ representa o conjunto de estados nos quais a verificação ocorre e $S_a(\bar{s})$ e $S_m(\bar{s})$ representam o conjunto de todos os estados nos quais a ação a e o método m são aplicáveis, respectivamente.

A atualização de uma política π é efetuada por uma disjunção com um conjunto de novos pares estado-ação π' , através da fórmula $\pi(\bar{s}, \bar{a}) \vee \pi'(\bar{s}, \bar{a})$.

5.4 Metas de decomposição sob incerteza

Todos os algoritmos utilizados até o momento partem da suposição de que apenas ações primitivas são importantes para o plano gerado, desconsiderando o processo de decomposição, que é visto aqui apenas como uma heurística dependente de domínio, utilizada com a finalidade de se obter maior eficiência em domínios não-determinísticos. Se apenas metas de alcançabilidade forem utilizadas, então é possível desconsiderar a relação entre decomposição e contextos de execução, como se existisse apenas um único contexto (tornando sua representação desnecessária), o que possibilita a utilização de políticas simples para a representação de planos.

O planejamento sob incerteza Knightiana tradicionalmente lida com metas de alcançabilidade, enquanto o planejamento hierárquico trabalha com metas de decomposição. As metas de decomposição têm como objetivo selecionar “caminhos” no sistema de transição de estados e são satisfeitas contanto que as pré-condições das ações primitivas sejam sempre aplicáveis. O estado final alcançado pelo plano é irrelevante para as metas de decomposição.

O fato de existir, no planejamento sob incerteza Knightiana, a possibilidade de se chegar a um estado do plano através de caminhos diferentes leva à necessidade de um mecanismo de manutenção de histórico de estados visitados durante a seleção de ações, para que metas de decomposição possam também ser respeitadas. Políticas determinísticas são funções do tipo $\pi : \mathcal{S} \mapsto \mathcal{A}$, ou seja, especificam apenas uma ação por estado. Políticas não-determinísticas, da forma $\pi' : \mathcal{S} \mapsto 2^{\mathcal{A}}$, devolvem um conjunto de ações que podem ser executadas. Entretanto, ambos os tipos de políticas consideram apenas o estado atual na escolha de ações. Isso exclui a possibilidade de utilizá-las para especificar ações baseadas em históricos.

5.4.1 Planos estendidos

Em domínios não-determinísticos, pode ser necessário que o agente busque alcançar um determinado estado e, se não for possível fazê-lo, garantir que algum estado seguro seja mantido. Por exemplo, pode-se requerer que um robô tente chegar a um determinado local, com a garantia de sempre evitar salas perigosas durante o trajeto, ou que o robô mantenha-se movendo alternadamente entre uma localidade l_1 e, se possível,

uma localidade l_2 . Para tanto, especifica-se que o robô deve passar sempre pelo local l_1 , enquanto deve passar pelo local l_2 somente se possível: as garantias dos requisitos para os dois locais são diferentes.

Metas estendidas [GNT04] deste tipo podem ser representadas de diferentes maneiras. Uma abordagem é representar metas com funções de utilidade, através de custos e recompensas sobre estados e ações. Outra é envolver condições temporais, estabelecendo propriedades a ser mantidas ao invés de alcançadas. Neste caso, o planejamento consiste em gerar planos que permitam que o agente satisfaça a fórmula temporal. Planejamento sob incerteza com metas estendidas é um problema desafiador em ambos os casos, pois estas adicionam ainda mais complexidade a um problema já por si complexo.

Em geral, um plano para uma meta estendida pode ser definido em termos de uma *função de ação* que, dado um estado e um contexto de execução, codificando o estado interno do agente, especifica a ação a ser executada, e em termos de uma *função de contexto* que, dependendo do efeito da ação, especifica o próximo contexto de execução.

Definição 5.4.1. Um plano estendido, sobre um domínio de planejamento $\mathcal{D}$, é uma tupla $\Pi = \langle \mathcal{C}, c_0, act, ctxt \rangle$, onde:

- $\mathcal{C}$ é um conjunto de *contextos de execução*;
- $c_0 \in \mathcal{C}$ é o *contexto inicial*;
- $act : \mathcal{S} \times \mathcal{C} \mapsto \mathcal{A}$ é a *função de ação*;
- $ctxt : \mathcal{S} \times \mathcal{C} \times \mathcal{S} \mapsto \mathcal{C}$ é a *função de contexto*. ◇

Se estamos num estado s e o contexto de execução é c , então $act(s, c)$ devolve a ação que deve ser executada pelo agente, enquanto $ctxt(s, c, s')$ associa a cada estado s' atingido o novo contexto de execução. Isto permite associar diferentes ações a um mesmo estado, dependendo do contexto considerado. As funções act e $ctxt$ são parciais, já que alguns estados e contextos nunca são atingidos durante a execução do plano.

Dizemos que um plano Π é *executável* se, sempre que $act(s, c) = a$ e $ctxt(s, c, s') = c'$, temos $s' \in \mathcal{T}(s, a)$. Dizemos que um plano é *completo* se, sempre que $act(s, c) = a$ e $s' \in \mathcal{T}(s, a)$, existe um contexto c' tal que $ctxt(s, c, s') = c'$ e $act(s', c')$ está definida.

A execução de um plano estendido resulta em mudanças de estados e contextos. Assim, podemos descrevê-la em termos de transições entre *pares estado-contexto*. Formalmente, dado um domínio $\mathcal{D}$ e um plano Π , uma transição de Π em $\mathcal{D}$ é uma tupla $(s, c) \xrightarrow{a_0} (s', c')$ tal que $act(s, c) = a$, $s' \in \mathcal{T}(s, a)$ e $ctxt(s, c, s') = c'$. Uma *trajetória* de um plano Π , a partir de s_0 , é uma seqüência infinita $(s_0, c_0) \xrightarrow{a_0} (s_1, c_1) \xrightarrow{a_1} (s_2, c_2) \xrightarrow{a_2} (s_3, c_3) \dots$, em que $(s_i, c_i) \xrightarrow{a_i} (s_{i+1}, c_{i+1})$ são transições.

Devido ao não-determinismo, um plano pode ter várias trajetórias. O conjunto dessas trajetórias pode ser representado pela estrutura de Kripke induzida pelo plano, ou seja, a estrutura de Kripke cujo conjunto de estados é o conjunto de pares estado-contexto e cujo conjunto de transições corresponde às transições das trajetórias. Tal estrutura de Kripke é também denominada *estrutura de execução* do plano Π .

Definição 5.4.2. A *estrutura de execução* de um plano Π num domínio $\mathcal{D}$, a partir do estado s_0 , é a estrutura de Kripke $\mathcal{K}_\Pi = \langle \mathcal{S}_\Pi, \mathcal{T}_\Pi, \mathcal{L}_\Pi \rangle$, onde:

- $\mathcal{S}_\Pi = \{(s, c) : act(s, c) \text{ é definida }\}$;
- $\mathcal{T}_\Pi = \{((s, c), (s', c')) : (s, c) \xrightarrow{a} (s', c') \text{ para alguma ação } a\}$;
- $\mathcal{L}_\Pi(s, c) = \{p : p \in \mathcal{L}(s)\}$. ◇

Definimos quando uma meta φ é satisfeita em (s, c) , denotado por $(\mathcal{K}_\Pi, (s, c)) \models \varphi$, usando a semântica padrão para fórmulas CTL (Apêndice B) sobre a estrutura de Kripke $\mathcal{K}_\Pi$.

Definição 5.4.3. Sejam $\mathcal{D}$ um domínio de planejamento e φ uma meta em $\mathcal{D}$. Sejam Π um plano e $\mathcal{K}_\Pi$ a estrutura de execução desse plano. O plano Π *satisfaz* a meta φ , a partir do estado inicial s_0 , denotado por $(\Pi, s_0) \models \varphi$, se $(\mathcal{K}_\Pi, (s_0, c_0)) \models \varphi$. O plano Π *satisfaz* a meta φ , a partir do conjunto de possíveis estados iniciais S_0 , denotado por $(\Pi, S_0) \models \varphi$, se $(\mathcal{K}_\Pi, (s, c_0)) \models \varphi$, para cada $s \in S_0$. ◇

5.4.2 Planos estendidos para metas de decomposição

É possível utilizar o formalismo de planos estendidos (Seção 5.4.1), que consideram também um contexto de execução na escolha de ações. *O planejamento hierárquico com*

metas de decomposição e sob incerteza sobre o estado corrente necessita de soluções como planos estendidos. A consideração sobre contextos de execução é necessária para representar as diferentes redes de tarefas que podem levar a um mesmo estado e que especificam diferentes ações para seus estados sucessores.

Figura 5.4: Falta de expressividade de políticas no planejamento hierárquico não-determinístico.

Contextos devem ser criados para estados de planos onde tarefas se “cruzam”, como mostra o exemplo (Figura 5.4(a)). Neste exemplo, há uma rede de tarefas onde o método m_1 é aplicável no estado s_1 e m_2 em s_2 , onde $s_1, s_2 \in S_0$. m_1 consiste na lista de ações primitivas (a_1, a_4) e m_2 consiste em (a_3, a_2) . Para que ambos os métodos tenham sucesso, entretanto, diferentes ações devem ser geradas para s_3 , dependendo da lista de tarefas especificada pelo método utilizado. Utilizar uma política não-determinística não resolve o problema, pois em tempo de execução isto pode resultar no caminho $s_1 \xrightarrow{a_1} s_3 \xrightarrow{a_2} s_4$, que viola a meta de decomposição. Neste caso, faz-se necessário o planejamento no sistema de transição de estados induzido pela introdução de contextos. Somente a divisão do estado s_3 em seus dois contextos possíveis c_1 e c_2 (Figura 5.4(b)) permite especificar diferentes ações a tomar. Durante a execução do plano agora só são possíveis as trajetórias $(s_1, c_d) \xrightarrow{a_1} (s_3, c_1) \xrightarrow{a_4} (s_5, c_d) \xrightarrow{\text{no-op}} \dots$ e $(s_2, c_d) \xrightarrow{a_3} (s_3, c_2) \xrightarrow{a_2} (s_4, c_d) \xrightarrow{\text{no-op}} \dots$. Somente s_3 necessita de contextos diferentes, portanto os demais estados podem receber um contexto padrão c_d . O plano estendido solução Π é composto do conjunto de contextos $\mathcal{C} = \{c_d, c_1, c_2\}$, o contexto inicial c_d e as funções act e txt ,

<i>act</i>			<i>ctxt</i>			
<i>s</i>	<i>c</i>	<i>a</i>	<i>s</i>	<i>c</i>	<i>s'</i>	<i>c'</i>
<i>s</i> ₁	<i>c</i> _d	<i>a</i> ₁	<i>s</i> ₁	<i>c</i> _d	<i>s</i> ₃	<i>c</i> ₁
<i>s</i> ₂	<i>c</i> _d	<i>a</i> ₃	<i>s</i> ₂	<i>c</i> _d	<i>s</i> ₃	<i>c</i> ₂
<i>s</i> ₃	<i>c</i> ₁	<i>a</i> ₄	<i>s</i> ₃	<i>c</i> ₁	<i>s</i> ₅	<i>c</i> _d
<i>s</i> ₃	<i>c</i> ₂	<i>a</i> ₂	<i>s</i> ₃	<i>c</i> ₂	<i>s</i> ₄	<i>c</i> _d

Figura 5.5: Tabelas das funções *act* e *ctxt* de Π para o exemplo 5.4.

exibidas nas tabelas da Figura 5.4.2.

Outra característica das metas de decomposição, causada pela ausência de um conjunto de estados meta, é a impossibilidade do uso de algoritmos baseados em pré-imagem, pois não há um conjunto de onde se possa iniciar a busca regressiva.

Capítulo 6

Análise Empírica

Este capítulo apresenta os resultados da utilização do planejador ND-HSHOP, um sistema de planejamento hierárquico para domínios não-determinísticos. O ND-HSHOP foi obtido pela aplicação da ND-transformação (Seção 5.1) ao HSHOP, uma re-implementação feita neste trabalho da funcionalidade básica do planejador SHOP [NCLMA99] na linguagem de programação Haskell [JH02]. Os nomes HSHOP e ND-HSHOP foram adotados no intuito de seguir a mesma nomenclatura utilizada pela família de planejadores SHOP. Notas adicionais sobre a implementação podem ser encontradas no Apêndice A.

6.1 Metodologia para análise do planejador

Um dos objetivos dessa análise é mostrar a quantidade e tamanho dos problemas que podem ser resolvidos com uma implementação do planejador hierárquico não-determinístico. Além disso, outro objetivo importante dos testes realizados sobre o planejador implementado é analisar o ganho de desempenho do planejamento hierárquico não-determinístico quando comparado ao planejamento não-determinístico, isto é, deseja-se analisar como o uso de conhecimento hierárquico sobre as ações do domínio pode melhorar o planejamento em domínios não-determinísticos. Para isso, o ND-HSHOP foi comparado ao planejador MBP (descrito na Seção 4.5.4), um planejador baseado em técnicas de Verificação de Modelos com BDDs e que não usa conhecimento hierárquico sobre as ações do domínio.

Adicionalmente, foi implementado, também em Haskell, um planejador baseado

em pré-imagem como o MBP, mas sem fazer uso de BDDs, para que pudéssemos estabelecer uma comparação justa entre planejadores sem e com hierarquias de ações. Este planejador, denominado HMBP, foi também incluso nos testes. A Tabela 6.1 mostra as diferentes características dos planejadores que foram usados nessa análise.

Planejador	Descrição
MBP	Planejador não-determinístico baseado em Verificação de Modelos e uso de BDDs para testes eficientes de satisfação de condições, implementado em C++.
HMBP	Planejador não-determinístico baseado em Verificação de Modelos sem o uso de BDDs, implementado em Haskell.
ND-HSHOP	Planejador não-determinístico que usa decomposição hierárquica de ações compostas, sem uso de BDDs, implementado em Haskell.

Tabela 6.1: Características dos planejadores utilizados nos testes.

Por se tratar de planejadores com características bem diferentes, a análise realizada foi feita através da medição do tempo absoluto gasto durante o planejamento e do tamanho da política encontrada pelos diferentes sistemas, para os problemas selecionados. Os valores de tempo medidos foram obtidos a partir da média de 3 execuções. Testes que levaram tempo maior que uma hora ou esgotaram a memória disponível foram descartados.

Os experimentos foram executados em um *laptop* Lenovo T60p com CPU Intel T2600 *dual-core* com *clock* de 2.16 GHz, 2 GB de RAM. Como o MBP é distribuído apenas na versão compilada para a plataforma Linux, os testes com os três planejadores foram realizados na distribuição Ubuntu Linux 7.10. O compilador utilizado em ND-HSHOP foi o *Glasgow Haskell Compiler* (GHC) 6.8.2, com a opção de otimização `-O` habilitada e com 1 GB de espaço disponível para o *heap* durante a execução (*flags +RTS -k1024m*).

6.2 O domínio Predador-Presas

Selecionamos para os testes o domínio de planejamento chamado de Predador-Presas [KS95], o mesmo utilizado por Kuter et al. [KNPT05] na avaliação do planejador YoYo. A motivação para o teste com este domínio é que métodos de planejamento hierárquico podem podar o espaço de busca, utilizando estratégias descritas por tarefas

compostas [KNPT05].

Figura 6.1: Exemplo de estado de um problema com 2 presas em uma grade 5×5 .

Um problema no domínio *Predador-Presas* [KS95] consiste em gerar um plano para um único agente predador, que deve agarrar várias presas movendo-se em uma grade quadrada de tamanho $n \times n$. O predador e as presas deslocam-se para regiões adjacentes através de movimentos para o norte, sul, leste ou oeste. As presas apresentam uma deficiência de velocidade em relação ao predador e, portanto, podem também *descansar*, o que faz com que elas não se desloquem. O predador também possui a ação *agarrar*, que é executável desde que este ocupe a mesma região de uma presa. O objetivo é planejar ações para o predador de modo a agarrar todas as presas. As presas são tratadas como diferentes agentes manipulando o mesmo ambiente, o que faz com que suas ações sejam tratadas como eventos exógenos. O movimento destas (incluindo a ação *descansar*) é responsável pelo não-determinismo do domínio. A descrição de um problema é dada pelas posições do predador e de cada presa e, adicionalmente, para cada presa, o predicado *agarrada*(x), que indica que a presa x já foi agarrada pelo predador (e, portanto, também não se movimenta mais).

Apesar de simples, este domínio apresenta um alto grau de dificuldade para o planejador, por possuir um grande fator de ramificação. O exemplo ilustrado pela Figura 6.1 mostra um estado de um problema envolvendo uma grade 5×5 e 2 presas, em que existem 3 possíveis ações para o predador e 5 para cada uma das presas, o que resulta em $3 \times 5 \times 5 = 75$ possíveis estados sucessores. Em geral, o tamanho do espaço de estados de problemas nesse domínio pode ser calculado através da equação $|P_n^p| =$

Figura 6.2: Número de estados possíveis em problemas Predador-Presas, com número de presas variando de 1 a 10, para $n = \{2, 3, 4\}$.

$\sum_{i=1}^{p+1} (n^2)^i$, onde P_n^p denota um problema de Predador-Presas em uma grade $n \times n$ com p presas iniciais. Com as curvas de $|P_n^p|$ plotadas em escala semi-log (Figura 6.2), onde linhas retas no gráfico equivalem a curvas exponenciais, pode-se observar que uma das características desse domínio é que o número de estados cresce exponencialmente em relação ao número de presas. Em compensação, o aumento do tamanho n da grade resulta em um aumento polinomial, como é mostrado na Figura 6.3, em escala log-log.

6.3 Estratégias hierárquicas para Predador-Presas

Problemas no domínio Predador-Presas podem ser resolvidos sem o uso de conhecimento hierárquico sobre ações compostas, utilizando apenas as ações primitivas descritas na Seção 6.2. Entretanto, o uso de hierarquias de tarefas nos permite adotar duas estratégias neste domínio:

1. O predador pode escolher perseguir uma presa x específica, alcançá-la e agarrá-la. Assim que a presa x for alcançada e agarrada, o predador irá escolher outra

Figura 6.3: Número de estados possíveis em problemas Predador-Presas, com tamanho da grade variando de 1 a 100, para $p = \{1, 2, 3, 4\}$.

presa para perseguir, até que todas tenham sido agarradas;

2. Outro meio de aproveitar a decomposição de tarefas para podar o espaço de busca é fazer com que o predador siga a presa x primeiro no sentido horizontal (leste-oeste) e depois no vertical (norte-sul). Isto já reduz pela metade o número de ações aplicáveis a considerar. Não há diferença de qualidade entre planos obtidos pelo uso ou não deste método, pois a distância Manhattan imposta pela grade faz com que qualquer caminho percorrido sem movimentos na direção oposta à presa tenha o mesmo número de passos.

O conjunto de métodos adotados para o domínio Predador-Presas é apresentado na Tabela 6.2. Na lista de pré-condições dos métodos, x e y representam as coordenadas do predador, x' e y' representam as coordenadas da presa selecionada para ser perseguida e $preys$ representa o número de presas restantes. A Tabela 6.3 apresenta uma descrição informal dos métodos.

Tarefas	Métodos		
	Nome	Pré-condição	Lista de tarefas
<i>catch_preys</i>	done	$preys = 0$	$\emptyset$
	more	$preys > 0$	$(catch_prey, catch_preys)$
<i>catch_pre</i>	chase	$\top$	$(chase_prey, catch)$
<i>chase_pre</i>	same	$x = x' \wedge y = y'$	$\emptyset$
	diff	$x \neq x' \vee y \neq y'$	$(chase_horizontal)$
<i>chase_horizontal</i>	chase_e	$x < x'$	$(east, chase_prey)$
	chase_w	$x > x'$	$(west, chase_prey)$
	horz_ok	$x = x'$	$(chase_vertical)$
<i>chase_vertical</i>	chase_n	$y < y'$	$(north, chase_prey)$
	chase_s	$y > y'$	$(south, chase_prey)$
	vert_ok	$y = y'$	$\emptyset$

Tabela 6.2: Métodos para decompor as ações do domínio Predador-Presas.

6.4 Um exemplo de uso do ND-HSHOP no domínio Predador-Presas

(a) Sistema de transição de estados para P_2^1

(b) Estrutura de execução da política obtida por ND-HSHOP.

Figura 6.4: Problema de Predador-Presas com uma presa em grade 2×2 .

O sistema de transição de estados de um teste em Predador-Presas com uma presa em grade 2×2 é ilustrado na Figura 6.4(a). O círculo maior denota a posição do predador e o menor a posição da presa. A Figura 6.4(b) mostra a estrutura de execução da política obtida por ND-HSHOP para este problema. Como esperado, sempre que a presa está em uma posição diagonal em relação ao predador, este move-se na horizontal,

Método	Descrição
done	Não há mais nenhuma presa a ser agarrada, portanto não há mais nada a fazer.
more	Ainda existe pelo menos uma presa não agarrada, então agarrá-la e depois agarrar as restantes.
chase	Perseguir presa e agarrá-la.
same	Predador e presa na mesma posição, perseguição termina.
diff	Predador e presa em posições diferentes, então persegui-la primeiro no sentido horizontal.
chase_e	Presa a leste do predador, então mover-se a leste e continuar perseguindo-a.
chase_w	Presa a oeste do predador, então mover-se a oeste e continuar perseguindo-a.
horz_ok	Predador e presa no mesmo eixo horizontal, persegui-la na vertical.
chase_n	Presa ao norte do predador, então mover-se ao norte e continuar perseguindo-a.
chase_s	Presa ao sul do predador, então mover-se ao sul e continuar perseguindo-a.
vert_ok	Predador e presa no mesmo eixo vertical, perseguição termina.

Tabela 6.3: Descrição informal dos métodos utilizados em Predador-Presas.

guiado pelo método *diff* (Tabela 6.2), que decompõe a tarefa *chase_prey* na tarefa *chase_horizontal*.

6.5 Análise de desempenho do ND-HSHOP

Nos experimentos com o domínio de Predador-Presas, foram utilizadas grades de tamanho $n \in \{2, 3, \dots, 70\}$, em problemas de 1 a 4 presas. Os tamanhos das políticas obtidas (Figura 6.5) foram plotados em escala log-log, onde linhas retas no gráfico equivalem a curvas polinomiais. O gráfico de tempo de execução (Figura 6.6), em segundos, também é mostrado em escala log-log.

Os resultados mostram que ND-HSHOP consegue resolver um número grande de problemas envolvendo uma presa, gerando planos de até 550.000 passos para cerca de 70 problemas, variando-se o tamanho da grade. O número de problemas resolvidos decai sensivelmente para números de presas 2, 3 e 4 (Figura 6.5). Isso pode ser tam-

Figura 6.5: Gráfico com número de estados das políticas geradas por ND-HSHOP, para grades $n \times n$, com $n = \{2, 3, \dots, 70\}$.

bém verificado no gráfico de tempo da Figura 6.6, que mostra como o tempo cresce rapidamente para problemas com mais presas. Na verdade, os problemas não puderam ser resolvidos devido ao grande uso de memória. Este fato é justificado pela própria característica do domínio Predador-Presas de que o número de estados cresce exponencialmente com o número de presas (Figura 6.2). Cada problema com grade 3×3 , com presas variando de 1 a 4, levou tempo aproximadamente 10 vezes maior que o problema com uma presa a menos. Apesar do uso do método de perseguir uma presa por vez, o fator de ramificação é muito grande, pois é preciso levar em conta o movimento das demais presas.

6.6 Análise do uso de hierarquias

Para compararmos o ND-HSHOP com o MBP e o HMBP, isto é, com planejadores não-determinísticos que não usam hierarquias, foram resolvidos problemas com apenas 1 presa, para grades de tamanho n variando de 1 a 70. Os testes terminaram quando toda a memória disponível foi consumida. As Figuras 6.7 e 6.8 mostram que o ND-

Figura 6.6: Gráfico de tempo de execução de ND-HSHOP em escala logarítmica.

HSHOP conseguiu resolver 2 vezes mais problemas que o MBP e aproximadamente 6 vezes mais problemas que o HMBP.

T(s)/Probl.	5	10	15	20	25	30	35	40	50	60	70
MBP	0,104	2,104	13,36	57,76	235,1	1136	7453	—	—	—	—
ND-HSHOP	0,024	0,076	0,820	1,384	3,476	8,388	19,20	45,12	172,2	613,4	2119
HMBP	0,812	228,0	—	—	—	—	—	—	—	—	—

Tabela 6.4: Comparação de tempo entre MBP, ND-HSHOP e HMBP, em problemas com uma presa, variando o tamanho da grade.

A desvantagem de MBP em relação ao ND-HSHOP nos testes se deve ao fato de que MBP faz busca regressiva em largura. Devido a essa estratégia de busca, estados não alcançáveis a partir do estado inicial também foram considerados por MBP e incluídos nas políticas, aumentando seu tamanho desnecessariamente e tornando necessária a computação adicional de ações para estes estados. Já o ND-HSHOP considera apenas aqueles estados possivelmente alcançáveis durante a execução do plano, minimizando o tamanho da política obtida. *Isso só é possível através do uso da decomposição hierárquica de ações compostas.*

Figura 6.7: Comparação de tempo entre MBP, ND-HSHOP e HMBP, em problemas com uma presa. Tempo em escala logarítmica.

Apesar de considerar mais estados que o necessário os caminhos de um estado qualquer até um estado meta em uma política obtida por MBP são minimais e, por isso, esperava-se que esse planejador apresentasse um desempenho mais próximo do ND-HSHOP. No entanto, a busca em profundidade com uso de hierarquias no ND-HSHOP faz com que o primeiro caminho encontrado seja selecionado (sendo não necessariamente o menor caminho). Os testes, entretanto, mostram que, apesar de não garantir uma solução ótima, ND-HSHOP é capaz de resolver um grande número de problemas com políticas envolvendo centenas de milhares de ações.

6.7 Sobre o uso da linguagem Haskell

A utilização de estruturas de dados nativas para representar estados de problemas permitiu um uso mais eficiente da memória. Por se tratarem de estruturas de dados persistentes (imutáveis), é possível compartilhar partes que não mudam entre diferentes representações de estados, o que por si só já é uma forma de representação compacta de estados. Esse tipo de representação não é tão eficiente quanto o uso de BDDs pois não

Figura 6.8: Comparação de tempo entre MBP e ND-HSHOP, em problemas com uma presa. Tempo em escala linear.

permite a aplicação de um conjunto de ações a um conjunto de estados (Seção 5.3.1), mas foi o que tornou possível tratar problemas grandes, que exigiriam uma quantidade proibitiva de memória caso contrário.

Capítulo 7

Conclusão

A recente tendência do uso de planejamento automatizado em aplicações práticas, que dependem de planos seguros e de grande porte, vem cada vez mais tornando inviável o uso do planejamento clássico em problemas reais.

Problemas práticos de planejamento envolvem a tomada de decisões sob incerteza, o que envolve maior esforço computacional e, por isso, vem cada vez mais motivando o uso de conhecimento específico de domínio para compensar o aumento do espaço de busca.

O planejamento hierárquico é a forma mais popular de uso de conhecimento específico em planejamento [GNT04] pois, além de podar o espaço de busca, fornece métodos de decomposição de tarefas compostas (ações de alto nível) em tarefas menores, de uma maneira correspondente ao modo com que humanos raciocinam sobre problemas complexos.

Nesta dissertação resolvemos problemas de planejamento não-determinístico, também conhecido como planejamento sob incerteza Knightiana [TCdB07], em que ações são modeladas como transições de um estado a um conjunto de possíveis estados sucessores, sem informação sobre custos ou distribuições de probabilidades e com observabilidade total. Examinamos um conjunto de algoritmos [KN04] [KNPT05] [Kut06] para lidar com planejamento sob incerteza Knightiana, a partir de algoritmos de planejamento clássico já existentes, baseados em busca progressiva no espaço de estados e em decomposição hierárquica de ações [KN04].

A primeira técnica examinada foi um método de generalizar um algoritmo de planejamento clássico baseado em busca progressiva no espaço de estados, habilitando-o a trabalhar com domínios não-determinísticos, chamada de ND-transformação de planejadores progressivos (Seção 5.1).

Em seguida, a ND-transformação foi aplicada a uma versão simplificada, re-escrita, do planejador hierárquico SHOP [NCLMA99], levando à obtenção de ND-HSHOP (Seção 5.2). O planejador obtido foi comparado com MBP [BCP⁺01], um planejador regressivo baseado em verificação de modelos simbólicos, através de testes de desempenho em alguns domínios prototípicos. Apesar da busca regressiva baseada em pré-imagem ser considerada mais eficiente que a busca progressiva para planejamento sob incerteza Knightiana, o uso de HTNs possibilitou obter maior desempenho ao custo de uma maior verbosidade na descrição do domínio (Tabela 6.2).

Os resultados mostram que ND-HSHOP consegue resolver um número grande de problemas envolvendo uma presa, gerando planos de até 550.000 passos para cerca de 70 problemas, variando-se o tamanho da grade. ND-HSHOP conseguiu resolver 2 vezes mais problemas que o MBP e aproximadamente 6 vezes mais problemas que o HMBP.

A desvantagem de MBP em relação ao ND-HSHOP nos testes se deve ao fato de que MBP faz busca regressiva em largura. Devido a essa estratégia de busca, estados não alcançáveis a partir do estado inicial também foram considerados por MBP e incluídos nas políticas, aumentando seu tamanho desnecessariamente e tornando necessária a computação adicional de ações para estes estados. Já o ND-HSHOP considera apenas aqueles estados possivelmente alcançáveis durante a execução do plano, minimizando o tamanho da política obtida. *Isso só é possível através do uso da decomposição hierárquica de ações compostas.*

Com relação ao uso da linguagem Haskell, um resultado interessante da análise empírica foi que, apesar de MBP fazer uso de BDDs para obter eficiência através de representações compactas de estados e ações, a abordagem baseada em estruturas de dados nativas de ND-HSHOP permitiu que este se mostrasse competitivo em termos do uso de memória, mesmo sem o uso de BDDs.

7.1 Trabalhos futuros

Uma contribuição do trabalho (Seção 5.4.2) foi mostrar que, para alguns problemas em específico, políticas não são estruturas suficientemente expressivas para representar alguns planos para problemas de planejamento hierárquico não-determinístico, mesmo sob a ausência de ações não-determinísticas, envolvendo apenas incerteza sobre o estado corrente. Este caso pretende ser tratado em trabalhos posteriores.

Para trabalhos futuros, além de extensões temporais ou de escalonamento facilitadas pelo fato do planejamento ser efetuado de maneira progressiva, outro possível passo seria estendê-lo para lidar com efeitos não especificados no domínio, que não pertencem à descrição do ambiente dada ao planejador (ou seja, atribuídos ao não-determinismo ilimitado), através do uso de técnicas de *replanejamento*. Esta última, em contrapartida, também obteria benefícios de uma especificação mais verbosa do domínio, onde o conhecimento de falhas comuns pode ser adicionado a este, reduzindo a necessidade do constante replanejamento por já gerar planos com ações contingentes para o caso esperado.

Assim como é possível efetuar a ND-transformação de planejadores progressivos determinísticos, também podemos transformá-los em planejadores aptos a lidar com observabilidade parcial [KNRG07]. Essa abordagem, denominada “condicionalização”, utiliza a representação de problemas através de variáveis de estado e modela ações sensoriais como a observação do valor de uma dentre as variáveis desconhecidas no estado corrente. Assim como mostram Kuter e Nau [KN04], a transformação é sistemática e especificada através de um procedimento abstrato, neste caso chamado de Cond-FCP. Aparentemente, as duas técnicas são complementares e podem ser unificadas, e isto seria uma importante extensão deste trabalho, pois habilitaria o planejador a lidar com mais uma dimensão de incerteza.

Existe também uma outra alternativa para planejamento com HTNs sob observabilidade parcial [BK04], que utiliza estados de crença para representar a informação incompleta sobre o estado do mundo. Além disso, obtém realimentação sobre a execução do plano, em um regime de malha fechada. A incorporação destas características ao planejador implementado possibilitaria tratar problemas que vão além da suposição de observabilidade total.

São também apresentados por Kuter [Kut06] meios de utilizar regras de controle para melhorar a eficiência de algoritmos de planejamento baseados em MDPs com busca progressiva, o que sugere que seja possível adaptar as técnicas ao trabalho de Trevizan et al. [TCdB07], sobre um modelo para planejamento com probabilidades imprecisas, e possibilitar o uso de ações não-determinísticas e probabilísticas em um mesmo domínio hierárquico.

Apêndice A

Apêndice: Implementação

A.1 Haskell

Para a implementação da técnica apresentada de planejamento hierárquico sob incerteza, foi utilizada a linguagem Haskell [HHJW07], uma linguagem de programação funcional pura, com tipagem estática, funções de alta ordem e avaliação preguiçosa. A programação funcional, fundamentada no λ -cálculo [Bar88], permite um estilo declarativo de programação [Hug89]. As características da linguagem são vantajosas para uma certa classe de problemas, que lidam mais com manipulações simbólicas do que entrada/saída. Um trabalho de [Erw04] mostra que Haskell pode ser tão adequado quanto Prolog [SS94] para resolver problemas de lógica.

A tipagem estática permite, junto com a inferência automática de tipos, verificação da utilização correta de funções e estruturas de dados em tempo de compilação. Programas errados são detectados prematuramente, o que resulta em um ganho de produtividade. A verificação estática de tipos também evita que estes testes precisem ser efetuados em tempo de execução, o que resulta em melhor desempenho. Testes podem ser efetuados de forma mais simples, pois a ausência de efeitos colaterais nas funções garante que resultados de funções dependam exclusivamente dos argumentos da função.

Outra vantagem proporcionada por esse paradigma de programação é que estruturas de dados puramente funcionais exibem a propriedade de *persistência* [Oka98], ou seja, são estruturas de dados imutáveis. Operações de modificação destas devolvem

novas cópias das estruturas, com a alteração desejada. Na prática, entretanto, é possível compartilhar as partes da estrutura original que não necessitam ser alteradas. Isso facilita a implementação de busca com retrocesso (*backtracking*), sem que seja necessário manter controle das estruturas de dados explicitamente. Os ramos da computação não-determinística recebem as cópias modificadas dessas estruturas.

A avaliação preguiçosa permite que partes de programas possam se comunicar facilmente com a ajuda de estruturas de dados potencialmente infinitas, facilitando a implementação de estratégias *generate-and-test*. Neste molde, um planejador pode ser representado como uma função que recebe um problema de planejamento (domínio, constantes do problema, estados iniciais e estados meta) e devolve uma lista, possivelmente infinita, de planos solução. Quando o primeiro plano da lista é solicitado (*e.g.*, para impressão na tela) então a primeira ação a ser impressa vai disparar a avaliação de toda a computação estritamente necessária para a geração do plano. Isto ocorre pois é necessário efetuar a busca completa até os estados meta para garantir que esta primeira ação seja correta. Se apenas um plano é suficiente e o resto da lista de planos é descartada, então não haverá nenhuma computação adicional.

É possível tirar vantagem da avaliação preguiçosa também para facilitar a criação de instâncias de operadores somente quando necessárias. Um operador `mover( $x, y, z$ )`, com pré-condições (`livre( $x$ )`, `livre( $y$ )`, `sobre( $x, y$ )`), especificado por alguma tarefa primitiva, terá as variáveis x , y e z instanciadas no momento de aplicação da ação. As instâncias aplicáveis serão derivadas através da unificação lógica do operador com os átomos especificados pelo estado atual do mundo.

Bibliotecas existentes para manipulação de BDDs em Haskell vão desde interfaces para bibliotecas externas em outras linguagens [Gam01] até implementações totalmente nativas [Chr06]. Implementações tradicionais efetuam a construção de toda a estrutura do BDD sem levar em consideração quais partes serão realmente necessárias. A utilização de avaliação preguiçosa embutida na linguagem de programação permite efetuar a construção de BDDs construção sob demanda de modo mais fácil.

Para a leitura de domínios e problemas, foi utilizada a biblioteca PARSEC [Lei01], que permite implementar *parsers* diretamente em Haskell. O objetivo foi ler uma versão simplificada da linguagem de JSHOP2, contendo especificação de operadores, tarefas

Figura A.1: Diagrama de classes UML mostrando as relações entre formalismos utilizados.

e métodos. Isto descarta funcionalidades adicionais como, por exemplo, especificação de axiomas, símbolos de funções e procedimentos anexados.

A representação por funções de mais altas ordens permite que seja utilizada uma forma de especialização de programas [Jon92], através de sua avaliação parcial. Se um programa recebe mais de uma entrada e algumas destas variam com menos frequência que outras, então a especialização do programa, em respeito a estas entradas, produz uma versão mais específica do programa. No caso de planejamento, é possível criar funções para planejadores específicos de domínio apenas fornecendo o argumento do domínio para a função de planejamento. Deste modo, foi possível tirar proveito de otimizações feitas pelo compilador e produzir planejadores específicos de domínio, especializados e mais eficientes.

A.2 Linguagem para planejamento

O modelo para planejamento hierárquico não-determinístico (Figura [A.1](#)) pode ser visto como uma extensão do planejamento clássico, que por sua vez baseia-se em uma lógica de primeira ordem, livre de funções e quantificadores. A diferença é que ações não-determinísticas são modeladas como operadores não-determinísticos, que podem possuir mais de um efeito. Operadores são associados a tarefas primitivas, ligando-os ao modelo de planejamento hierárquico. Domínios também devem especificar métodos de decomposição de tarefas compostas. Métodos aproveitam o mecanismo de verificação de pré-condições, assim como operadores. Um problema consiste na decomposição de uma tarefa composta, sem perda de generalidade.

A.3 Algoritmo FS_{SHOP}^3

Esta seção foi escrita utilizando *programação literal* [Knu84] com a ferramenta `lhs2TeX` [HL07], portanto condensando tanto o código quanto a documentação.

Alguns parâmetros necessitam ser passados ao compilador (no caso, o *Glasgow Haskell Compiler*, GHC 6.6.1), para especificar o uso de certas extensões não presentes no padrão da linguagem Haskell 98:

```
{-# OPTIONS_GHC -fglasgow-exts -fno-monomorphism-restriction #-}
```

As declarações explícitas de tipos das funções (linhas utilizando a notação `::`) abaixo não são necessárias, pois a linguagem Haskell possui inferência de tipos em tempo de compilação, mas estão presentes por motivos de clareza na apresentação e também acabam por agir como um “contrato” com o compilador, que tenta verificar se os tipos são satisfeitos.

A.3.1 Módulos utilizados

O código é estruturado em sistema de módulos hierárquicos, sob a categoria de algoritmos de planejamento.

```
module AI.Planning.Algorithm.FSSHOP3 where
```

O código utiliza o formalismo de *mônadas* [Wad92] (provenientes da teoria das categorias) em sua construção: *Reader* é utilizada para propagar as funções do domínio implicitamente, e *List* permite especificar os pontos de retrocesso através da representação de funções que podem potencialmente devolver mais de uma resposta.

```
import Control.Monad      (Monad (return), sequence)
import Control.Monad.Reader (MonadReader, asks)
import Control.Monad.List   (MonadPlus, guard, msum)
```

O operador de catamorfismo com avaliação estrita *foldl'* é utilizado para tornar mais eficientes algumas funções. Algumas funções auxiliares para funções parciais (*Maybe*) também são utilizadas:

```

import Data.List    (foldl')
import Data.Maybe  (catMaybes)

```

Também são utilizadas estruturas de dados para representação de mapas e conjuntos puramente funcionais. Estas estruturas apresentam a propriedade de persistência e, portanto, são adequadas para o processo de busca com retrocesso:

```

import Data.Set    (Set)
import Data.Map    (Map)
import qualified Data.Set  as S
import qualified Data.Map  as M

```

Entre outras, as funções fazem uso das operações de união de conjuntos e de mapas, denotadas, respectivamente, como $\cup_S$, de *Data.Set*, e $\cup_M$, de *Data.Map*.

A.3.2 Modelo

Tarefas são representadas através do tipo de dado algébrico *Task*, polimórfico em relação ao tipo de estados e ações. O construtor *Primitive* referencia uma ação primitiva. *Compound* possui um *String* com o nome da tarefa composta e uma lista de métodos de decomposição. O construtor *End* é utilizado para especificar fim prematuro da uma lista de tarefas.

```

data Task s a = Primitive a
               | Compound String [Method s a]
               | End

```

Métodos também são representados como um tipo de dado algébrico, mas com um único construtor, que possui uma lista de sub-tarefas e uma função *isRelevant*, responsável pela verificação de pré-condições do método.

```

data Method s a = Method { isRelevant :: s → Bool
                          , subtasks   :: [Task s a] }

```

Um domínio é caracterizado pelas funções de aplicabilidade e de aplicação de ações, respectivamente *applyFn* e *applicableFn*, encapsuladas em um tipo *Domain*.

data $\text{Domain } s \ a = \text{Domain} \{ \text{applyFn} \quad :: s \rightarrow a \rightarrow \text{Set } s$
 $\quad \quad \quad , \text{applicableFn} :: s \rightarrow a \rightarrow \text{Bool} \}$

Situation é um tipo sinônimo representando situações (Seção 5.2.1), também polimórficas nos tipos de estados e informação de controle de busca. *Control* é outro sinônimo de tipo representando a informação de controle para próximos estados, de acordo com EFETUADECOMPOSIÇÃO (Algoritmo 12). Especifica que ação deve ser aplicada a cada conjunto de estados e a informação de controle que deve ser passada aos estados sucessores destes.

type $\text{Situation } s \ x = (\text{Set } s, x)$
type $\text{Control } s \ a \ x = [(\text{Set } s, a, x)]$

No caso do planejamento hierárquico em ordem total, a informação de controle é uma lista de tarefas. Isto é representado através das especializações de tipos:

type $\text{SituationH } s \ a = \text{Situation } s \ [\text{Task } s \ a]$
type $\text{ControlH } s \ a = \text{Control } s \ a \ [\text{Task } s \ a]$

Políticas neste caso podem ser representadas através de simples mapas associativos e, portanto, são apenas um sinônimo de tipo.

type $\text{Policy } s \ a = \text{Map } s \ a$

A.3.3 Algoritmo

O ponto de entrada do algoritmo é a função fs_{shop}^3 , que recebe uma lista de situações em aberto, um conjunto de estados meta, uma política parcial e um conjunto de estados iniciais. Retorna uma lista das soluções possíveis, na forma de políticas cíclicas fortes.

$fs_{\text{shop}}^3 :: (\text{MonadReader } (\text{Domain } s \ a) \ m, \text{MonadPlus } m, \text{Ord } s, \text{Ord } (\text{Set } s),$
 $\quad \quad \quad \text{Eq } [\text{Task } s \ a])$
 $\Rightarrow [\text{SituationH } s \ a] \rightarrow \text{Set } s \rightarrow \text{Policy } s \ a \rightarrow \text{Set } s \rightarrow m (\text{Policy } s \ a)$

$$\begin{aligned}
fs_{shop}^3 \text{ open } gs \ \pi \ s_0 = & \text{do let } open' = \text{simplify } open \ gs \ \pi \\
& ok \leftarrow \text{good } \pi \ (\text{statesOf } open) \ gs \ s_0 \\
& \text{guard } ok \\
& fs_{shop}^3 \text{ ' } open' \ gs \ \pi \ s_0
\end{aligned}$$

A função auxiliar $fs_{shop}^3 \text{ '}$ simplesmente devolve π quando a lista de situações é vazia. Enquanto há uma situação sit , esta é decomposta em seu controle f . Caso f seja vazio, a função devolve falha. Caso contrário, a partir daí, são computadas as situações $open'$ resultantes de f . As ações a para cada estado s são acumuladas na política π' e $fs_{shop}^3 \text{ '}$ é chamada recursivamente, para processar as situações remanescentes.

$$\begin{aligned}
fs_{shop}^3 \text{ ' } & :: (\text{MonadReader } (\text{Domain } s \ a) \ m, \text{MonadPlus } m, \text{Ord } s, \text{Ord } (\text{Set } s), \\
& \quad \text{Eq } [\text{Task } s \ a]) \\
& \Rightarrow [\text{SituationH } s \ a] \rightarrow \text{Set } s \rightarrow \text{Policy } s \ a \rightarrow \text{Set } s \rightarrow m \ (\text{Policy } s \ a) \\
fs_{shop}^3 \text{ ' } [] & \quad - \ \pi \ - = \text{return } \pi \\
fs_{shop}^3 \text{ ' } (sit : sits) \ gs \ \pi \ s_0 = & \text{do } f \leftarrow \text{decompose } sit \\
& \text{guard } \$ \neg (\text{null } f) \\
& open' \leftarrow \text{successors } f \ sits \\
& \text{let } part = [(s, a) \mid (ss, a, _) \leftarrow f, s \leftarrow \text{toList}_S \ ss] \\
& \quad \pi' = \pi \cup_M \text{fromList}_M \ part \\
& fs_{shop}^3 \text{ ' } open' \ gs \ \pi' \ s_0
\end{aligned}$$

PODASITUAÇÕES (Algoritmo 11) é implementado através da função *simplify*:

$$\begin{aligned}
\text{simplify} & :: (\text{MonadPlus } m, \text{Ord } s) \\
& \Rightarrow m \ (\text{Situation } s \ x) \rightarrow \text{Set } s \rightarrow \text{Policy } s \ a \rightarrow m \ (\text{Situation } s \ x) \\
\text{simplify } open \ gs \ \pi = & \text{do let } notExpansible = gs \cup_S \text{statesOfM } \pi \\
& (ss, x) \leftarrow open \\
& \text{let } ss' = ss \setminus_S notExpansible \\
& \text{guard } \$ \neg (\text{null}_S \ ss') \\
& \text{return } (ss', x)
\end{aligned}$$

A função *decompose* efetua o mesmo papel de EFETUADECOMPOSIÇÃO (Algoritmo 12). Possui três casos para os construtores do tipo de dados algébrico *Task s a*, além de tratar o casos de listas de situações vazias.

```

decompose      :: (MonadReader (Domain s a) m, MonadPlus m, Ord s)
                => SituationH s a → m (ControlH s a)
decompose sit = decompose' [sit] []

```

```

decompose' :: (MonadReader (Domain s a) m, MonadPlus m, Ord s)
            => [SituationH s a] → ControlH s a → m (ControlH s a)
decompose' [] f = return f
decompose' ((-, []): -) = return []
decompose' ((-, End: -): -) f = return f
decompose' ((ss, ((Primitive a): ts)): sits) f = msum $ map decomp [a]
  where decomp a = do ok ← applToAll a ss
                    guard ok
                    let f' = (ss, a, ts): f
                    decompose' sits f'
decompose' ((ss, x@((Compound _ ms): ts)): sits) f = msum $ map decomp ms
  where decomp m = do let (i, o) = splitRelv m ss
                    guard $ ¬ (nullS i)
                    let sits' = if nullS o then sits else (o, x): sits
                    sits'' = (i, subtasks m ++ ts): sits'
                    decompose' sits'' f

```

ENCONTRASUCESSORES (Algoritmo 13) é implementado através da função *successors*:

```

successors :: (MonadReader (Domain s a) m, Eq [Task s a], Ord s, Ord (Set s))
            => ControlH s a → [SituationH s a] → m [SituationH s a]
successors f open = do open' ← sequence $ [compSucc ss a x | (ss, a, x) ← f]
                    let open'' = open' ++ open
                    return [(compose x open'', x) | (-, x) ← open'']

```

A função *compSucc* devolve uma situação contendo um conjunto de estados sucessores, obtidos pela aplicação da ação *a* a todos os estados de *ss*.

```

compSucc      :: (MonadReader (Domain s a) m, Ord s, Ord (Set s))
                => Set s → a → [Task s a] → m (SituationH s a)
compSucc ss a x = do ss' ← next ss a
                    return (ss', x)

```

Os estados de uma situação para os quais um certo método é relevante podem ser identificados através da função *splitRelv*, que através de um método *m* e um conjunto de estados, separa-os em um par de conjuntos de estados, onde o método é relevante para os primeiros e irrelevante para os segundos.

```

splitRelv    :: Ord s => Method s a -> Set s -> (Set s, Set s)
splitRelv m = foldS spl (∅S, ∅S)
  where spl s (i, o) | isRelevant m s = (insertS s i, o)
                    | otherwise      = (i, insertS s o)

```

A função *compose* é a implementação da definição de COMPÕE (5.2). Identifica todos os estados do conjunto de situações, *open*, que estão sujeitos ao controle *w*.

```

compose      :: (Eq x, Ord s) => x -> [Situation s x] -> Set s
compose x open = fromListS [s | (ss, x') ← open, x' ≡ x, s ← toListS ss]

```

A função *next* devolve um conjunto de estados sucessores, obtidos pela aplicação da ação *a* a todos os estados de *ss*.

```

next        :: (MonadReader (Domain s a) m, Ord s, Ord (Set s))
              => Set s -> a -> m (Set s)
next ss a = do γ ← asks applyFn
              let sss = mapS (λs -> γ s a) ss
              return $ foldS unionS ∅S sss

```

A função *applToAll* verifica se uma dada ação *a* é aplicável a todos os estados de um conjunto *ss*.

```

applToAll    :: MonadReader (Domain s a) m => a -> Set s -> m Bool
applToAll a ss = do applicable ← asks applicableFn
                  let fn s b = b ∧ applicable s a
                  return $ foldS fn True ss

```

Como esta implementação em particular busca sempre por planos fortes cíclicos, *good* é apenas uma chamada a *isStrongCyclic*, com os mesmos argumentos (implícitos, devido ao uso de estilo *pointfree*).

```

good :: (MonadReader (Domain s a) m, Eq (Set s), Ord s)
      => Policy s a -> Set s -> Set s -> Set s -> m Bool
good = isStrongCyclic

```

Verificar se uma política π é ou pode ser estendida a uma política cíclica forte depende do conjunto de estados em aberto ss , do conjunto de metas gs e dos estados iniciais ss_0 .

```

isStrongCyclic :: (MonadReader (Domain s a) m, Eq (Set s), Ord s)
               => Policy s a -> Set s -> Set s -> Set s -> m Bool
isStrongCyclic  $\pi$  ss gs ss0 = do let initial = (ss  $\cup_S$  gs,  $\pi$ )
                                (ss',  $\pi'$ ) <- fixedPointM cmpFst testSC initial
                                let ok = ss0  $\subseteq_S$  ss'  $\wedge$  nullM  $\pi'$ 
                                return ok
testSC :: (MonadReader (Domain s a) m, Ord s)
       => (Set s, Policy s a) -> m (Set s, Policy s a)
testSC (ss,  $\pi$ ) = do sc <- weakPreImgPi ss  $\pi$ 
                    let ss' = ss  $\cup_S$  sc
                         $\pi'$  = deleteMany ss'  $\pi$ 
                    return (ss',  $\pi'$ )

```

A função de pré-imagem fraca de uma política, *weakPreImgPi*, é dada por:

```

weakPreImgPi :: (MonadReader (Domain s a) m, Ord s)
             => Set s -> Policy s a -> m (Set s)
weakPreImgPi ss  $\pi$  = do let fn (s, a) = leadsToStates s a ss
                        abc <- sequence  $\circ$  map fn $ toListM  $\pi$ 
                        return  $\circ$  fromListS $ catMaybes abc

```

leadsToStates verifica se existe algum estado sucessor de s através da aplicação de a que se encontre em um conjunto de estados ss

```

leadsToStates :: (MonadReader (Domain s a) m, Ord s)
              => s -> a -> Set s -> m (Maybe s)
leadsToStates s a ss = do  $\gamma$  <- asks applyFn
                        if or $ map ( $\in_S$  ss) (toListS $  $\gamma$  s a)

```

```

then return (Just s)
else return Nothing

```

statesOfM denota o conjunto de estados presentes em uma política, que, por ser representada por um *Map*, torna possível especificar a equivalência dessas funções.

```

statesOfM :: Policy s a → Set s
statesOfM = keysSetM
statesOf   :: (Ord s) ⇒ [SituationH s a] → Set s
statesOf open = unionsS $ map fst open

```

A.3.4 Funções auxiliares

A remoção de um conjunto de entradas de um mapa pode ser feita através da função *deleteMany*, que beneficia-se do use de uma versão de *foldl* com avaliação estrita, *foldl'*. Desse modo, pode ser evitada a construção de estruturas contendo versões intermediárias do conjunto, que nunca serão utilizadas.

```

deleteMany :: (Ord a) ⇒ Set a → Map a b → Map a b
deleteMany s m = foldl' (flip deleteM) m (toListS s)

```

O cálculo do ponto fixo de uma função *f* dentro de uma mônada *m* arbitrária é efetuado através da aplicação sucessiva de *f* seguida de uma função de comparação *cmp*, também arbitrária, entre dois valores consecutivos. Caso o resultado de *cmp* seja verdadeiro, o elemento mais recente é devolvido.

```

fixedPointM :: Monad m ⇒ (a → a → Bool) → (a → m a) → a → m a
fixedPointM cmp f x = do y ← f x
                    if cmp x y
                    then return y
                    else fixedPointM cmp f y

```

Uma função de comparação utilizada anteriormente é *cmpFst*, que compara o primeiro elemento de dois pares utilizando o operador padrão de comparação.


```

cmpFst      :: Eq a => (a, b) -> (a, c) -> Bool
cmpFst x y = fst x == fst y

```

Detalhes de implementação

Para que seja possível exibir uma rede de tarefas, *Task s a* deve ser uma instância do *typeclass Show*:

```

instance Show a => Show (Task s a) where
  show (Primitive a)    = "Primitive " ++ show a
  show (Compound n _) = n
  show End              = "END"

```

Tarefas devem ser também instâncias de *Eq* e *Ord*, para que possam ser utilizadas por estruturas que fazem comparação de elementos, como *Data.Set*:

```

instance Eq a => Eq (Task s a) where
  (Primitive a1)      == (Primitive a2)    = a1 == a2
  (Compound n1 _)    == (Compound n2 _)    = n1 == n2
  _                   == _                  = False

instance Ord a => Ord (Task s a) where
  (Primitive a1)      'compare' (Primitive a2)    = a1 'compare' a2
  (Compound n1 _)    'compare' (Compound n2 _)    = n1 'compare' n2
  Primitive{ }       'compare' Compound{ }       = LT
  Compound{ }        'compare' Primitive{ }       = GT
  End                'compare' _                  = LT
  _                  'compare' End                = GT

```

Métodos também são instâncias de *Eq* e *Ord* pelos mesmos motivos que tarefas.

```

instance (Eq s, Eq a) => Eq (Method s a) where
  (Method _ ts1) == (Method _ ts2) = ts1 == ts2

instance (Ord s, Ord a) => Ord (Method s a) where
  (Method _ ts1) 'compare' (Method _ ts2) = ts1 'compare' ts2

```


Apêndice B

Apêndice: Um Estudo Sobre a Lógica CTL

B.1 Lógica de Tempo Ramificado (*Computation Tree Logic*) (CTL)

B.1.1 Lógica modal proposicional

A linguagem da lógica modal proposicional é constituída de um conjunto de proposições atômicas $\mathbb{P} \supset \{\top, \perp\}$, o conectivo $\rightarrow$ (ou qualquer outro conjunto funcionalmente completo de conectivos proposicionais clássicos) e da modalidade $\Box$. O conjunto de fórmulas da lógica modal proposicional, denotado por $\mathcal{L}_m$, é definido indutivamente, assim como faz Pereira [Per06]:

$$\varphi \stackrel{\text{def}}{=} p \in \mathbb{P} \mid \varphi_1 \rightarrow \varphi_2 \mid \Box\varphi.$$

Os demais conectivos da lógica proposicional e a modalidade dual $\Diamond$ são definidos como:

- $\neg p \stackrel{\text{def}}{=} \varphi \rightarrow \perp$
- $\varphi_1 \vee \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \rightarrow \perp) \rightarrow \varphi_2$
- $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \rightarrow (\varphi_2 \rightarrow \perp)) \rightarrow \perp$
- $\Diamond\varphi \stackrel{\text{def}}{=} \neg\Box\neg\varphi$

Definição B.1.1. Um *modelo de Kripke* é um par $\langle \mathcal{K}, \models \rangle$, onde $\mathcal{K} = \langle \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$ é uma estrutura de Kripke e $\models$ é uma relação entre um estado $s \in \mathcal{S}$ e uma *fórmula modal* $\varphi \in \mathcal{K}_m$, tal que:

- $(\mathcal{K}, s) \models p$ se e só $p \in \mathcal{L}(s)$;

- $(\mathcal{K}, s) \models \varphi_1 \rightarrow \varphi_2$ se e só se $(\mathcal{K}, s) \not\models \varphi_1$ ou $(\mathcal{K}, s) \models \varphi_2$;
- $(\mathcal{K}, s) \models \Box \varphi$ se e só se $(\mathcal{K}, s') \models \varphi$ para todo $s' \in S$ tal que $(s, s') \in \mathcal{T}$. $\diamond$

A relação $\models$ é denominada *relação de satisfação* e a notação $(\mathcal{K}, s) \models \varphi$ significa que o estado s , da estrutura de Kripke $\mathcal{K}$, satisfaz a fórmula φ . Quando a estrutura de $\mathcal{K}$ está clara no contexto, podemos escrever apenas $s \models \varphi$.

B.1.2 Árvores de computação

A *árvore de computação* obtida pelo desdobramento de uma estrutura de Kripke $\mathcal{K}$ em uma árvore infinita, usando um estado s como raiz (*i.e.*, o estado inicial), é denotada por $\Upsilon_{\mathcal{K}}^s$. Também usamos $\Upsilon_{\mathcal{K}}^s$ para denotar o conjunto de todos os caminhos que partem de um estado s numa estrutura de Kripke $\mathcal{K}$.

Um *caminho* numa árvore de computação $\Upsilon_{\mathcal{K}}^{s_0}$ é uma seqüência infinita de estados $\pi = \langle s_0, s_1, \dots \rangle$, tal que para cada par de estados consecutivos $(s_i, s_{i+1}) \in \pi$ é um elemento da relação de transição $\mathcal{T}$. Dado um caminho $\pi = \langle s_0, s_1, \dots \rangle$, usamos π_i para denotar um estado s_i , de π e $\pi_{[i \dots]}$ para denotar um sufixo $\langle s_i, s_{i+1}, \dots \rangle$ de π .

B.1.3 A lógica de tempo ramificado CTL

Lógica de Tempo Ramificado (*Computation Tree Logic*) (CTL) [EC82] é uma lógica proposicional de tempo ramificado, cujas fórmulas permitem especificar propriedades sobre caminhos em uma árvore de computação. Na lógica CTL, os operadores temporais da lógica LTL devem ser imediatamente precedidos por um quantificador de caminhos ($\forall$ ou $\exists$).

Os operadores temporais em CTL são os seguintes:

- $\forall \bigcirc$ (em *todos* os *próximos* estados)
- $\forall \Box$ (em *todos* os estados futuros, *invariantemente*)
- $\forall \Diamond$ (em *todos* os estados futuros, *finalmente*)
- $\forall \cup$ (em *todos* os estados futuros, *até*)

- $\exists\bigcirc$ (em *algum* dos *próximos* estados)
- $\exists\Box$ (em *algum* dos estados futuros, *invariantemente*)
- $\exists\Diamond$ (em *algum* dos estados futuros, *finalmente*)
- $\exists\cup$ (em *algum* dos estados futuros, *até*)

O conjunto de fórmulas CTL é definido indutivamente como:

$$\varphi \stackrel{\text{def}}{=} p \in \mathbb{P} \mid \varphi_1 \rightarrow \varphi_2 \mid \forall\bigcirc\varphi \mid \forall(\varphi_1 \cup \varphi_2) \mid \exists(\varphi_1 \cup \varphi_2).$$

Os demais conectivos lógicos são definidos em termos de $\perp$ e $\rightarrow$, da maneira usual. Os demais operadores temporais são definidos conforme a seguir:

- $\forall\Box\varphi \stackrel{\text{def}}{=} \neg\exists(\top \cup \neg\varphi)$
- $\exists\bigcirc\varphi \stackrel{\text{def}}{=} \neg\forall\bigcirc\neg\varphi$
- $\exists\Diamond\varphi \stackrel{\text{def}}{=} \exists(\top \cup \varphi)$
- $\forall\Diamond\varphi \stackrel{\text{def}}{=} \forall(\top \cup \varphi)$
- $\exists\Box\varphi \stackrel{\text{def}}{=} \neg\forall(\top \cup \neg\varphi)$

Definição B.1.2. Sejam $\mathcal{K}$ uma estrutura de Kripke, s um estado dessa estrutura e φ uma fórmula CTL. A *semântica* das fórmulas CTL é definida como segue:

- $s \models p$ se e só se $p \in \mathcal{L}(s)$;
- $s \models \varphi_1 \rightarrow \varphi_2$ se e só se $s \not\models \varphi_1$ ou $s \models \varphi_2$;
- $s \models \forall\bigcirc\varphi$ se e só se, para todo caminho $\pi \in \Upsilon_{\mathcal{K}}^s$, $\pi_1 \models \varphi$;
- $s \models \forall\Diamond\varphi$ se e só se, para *todo* caminho $\pi \in \Upsilon_{\mathcal{K}}^s$, existe $j \geq 0$ tal que $\pi_j \models \varphi_2$ e, para todo $i < j$, $\pi_i \models \varphi_1$;
- $s \models \exists\bigcirc\varphi$ se e só se, para *algum* caminho $\pi \in \Upsilon_{\mathcal{K}}^s$, existe $j \geq 0$ tal que $\pi_j \models \varphi_2$ e, para todo $i < j$, $\pi_i \models \varphi_1$. $\diamond$

B.2 Planejamento com CTL

CTL é utilizada no planejamento através de verificação de modelos para especificar propriedades sobre a evolução do estado do ambiente, mas apenas na *validação* de

planos, e não para sua *síntese*. α -CTL [Per06] é uma extensão que busca prover o formalismo necessário para a síntese de planos sem recorrer a interceptação de passos do provador de teoremas CTL.

Referências Bibliográficas

- [AKN05] Tsz-Chiu Au, Ugur Kuter, e Dana S. Nau. Web service composition with volatile information. Em *International Semantic Web Conference*, páginas 52–66, 2005. [33](#)
- [Bar88] H. P. Barendregt. Introduction to lambda calculus. Em *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988. [85](#)
- [BCP⁺01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, e P. Traverso. MBP: a Model Based Planner, 2001. [47](#), [82](#)
- [Bel89] Colin E. Bell. Maintaining project networks in automated artificial intelligence planning. *Management Science*, 35(10):1192–1214, Outubro de 1989. [1](#)
- [BF97] Avrim Blum e Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997. [17](#)
- [BG79] David A. Blackwell e Meyer Abraham Girshick. *Theory of Games and Statistical Decisions*. Dover, New York, 1979. [3](#)
- [BG99] Blai Bonet e Hector Geffner. Planning as heuristic search: New results. Em *European Conference on Planning (ECP)*, páginas 360–372, 1999. [49](#)
- [BG01] Blai Bonet e Hector Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001. [4](#)

- [BHH03] Thorsten Belker, Martin Hammel, e Joachim Hertzberg. Learning to optimize mobile robot navigation based on HTN plans. Em *International Conference on Robotics and Automation (ICRA)*, páginas 4136–4141, Setembro de 2003. [21](#)
- [BK00] Fahiem Bacchus e Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000. [23](#), [34](#), [47](#), [49](#), [52](#)
- [BK04] A. Bouguerra e L. Karlsson. Hierarchical task planning under uncertainty, 2004. Em 3rd Italian Workshop on Planning and Scheduling (AI*IA). [83](#)
- [BPR02] Piergiorgio Bertoli, Marco Pistore, e Marco Roveri. Planning as model checking: Aips’02 hands-on tutorial, Abril de 2002. [47](#)
- [Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *Association for Computing Machinery Surveys*, 24(3):293–318, 1992. [45](#), [54](#), [61](#)
- [Chr06] Jan Christiansen. *A purely functional implementation of ROBDDs in Haskell*. Tese de Doutorado, Christian-Albrechts-Universität zu Kiel, 2006. [86](#)
- [CL03] Stephen A. Cook e Yongmei Liu. A complete axiomatization for blocks world. *Journal of Logic and Computation*, 13(4):581–594, 2003. [14](#)
- [CPRT03] Alessandro Cimatti, Marco Pistore, Marco Roveri, e Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003. [40](#), [41](#), [60](#)
- [CR00] Alessandro Cimatti e Marco Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research (JAIR)*, 13:305–338, 2000. [37](#)
- [CT91] Ken Currie e Austin Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991. [22](#)

- [DGKK98] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, e Jonas Kvarnström. TAL: Temporal Action Logics language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2:273–306, 1998. [23](#)
- [DK99] Patrick Doherty e Jonas Kvarnström. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. Em *TIME*, páginas 47–54, 1999. [23](#), [47](#), [49](#)
- [DLL62] Martin Davis, George Logemann, e Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. [18](#)
- [EC82] E. Allen Emerson e Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982. [47](#), [100](#)
- [EHN94a] Kutluhan Erol, James A. Hendler, e Dana S. Nau. HTN planning: Complexity and expressivity. Em *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, páginas 1123–1128, Seattle, Washington, USA, 1994. AAAI Press / The MIT Press. [3](#), [22](#)
- [EHN94b] Kutluhan Erol, James A. Hendler, e Dana S. Nau. Semantics for HTN planning. Relatório Técnico CS-TR-3239, University of Maryland, Março de 1994. [3](#), [22](#)
- [EHN94c] Kutluhan Erol, James A. Hendler, e Dana S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. Em *AIPS*, páginas 249–254, 1994. [22](#)
- [EHN96] Kutluhan Erol, James A. Hendler, e Dana S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996. [21](#), [23](#)
- [EHNT95] Kutluhan Erol, James A. Hendler, Dana S. Nau, e Reiko Tsuneto. A critical look at critics in HTN planning. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 1592–1598, 1995. [23](#)

- [Eme90] E. Allen Emerson. Temporal and modal logic. Em *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, páginas 995–1072. 1990. [46](#)
- [ENS95] Kutluhan Erol, Dana S. Nau, e V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995. [3](#)
- [Erw04] Martin Erwig. Escape from Zurg: an exercise in logic programming. *Journal of Functional Programming*, 14(3):253–261, 2004. [85](#)
- [FN71] Richard Fikes e Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971. [9](#)
- [Gam01] Peter Gammie. A Haskell binding to Long’s BDD library, 2001. [86](#)
- [GN92] Naresh Gupta e Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992. [15](#)
- [GNT04] Malik Ghallab, Dana S. Nau, e Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufman, San Francisco, CA, Maio de 2004. [1](#), [3](#), [7](#), [8](#), [10](#), [11](#), [12](#), [15](#), [16](#), [21](#), [26](#), [27](#), [31](#), [37](#), [43](#), [65](#), [81](#)
- [HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, e Philip Wadler. A history of Haskell: Being lazy with class. Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), Abril de 2007. [85](#)
- [HL07] Ralf Hinze e Andres Löh. Guide2lhs2tex, Janeiro de 2007. Institut für Informatik III, Universität Bonn Römerstraße, Bonn, Alemanha. [89](#)
- [HN01] Jörg Hoffmann e Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 14:253–302, 2001. [49](#)
- [How60] R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960. [4](#)

- [Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989. [85](#)
- [Ilg05] Okhtay Ilghami. Documentation for JSHOP2, Fevereiro de 2005. Technical Report CS-TR-4694, Department. of Computer Science, University of Maryland. [29](#)
- [IN04] Okhtay Ilghami e Dana S. Nau. A general approach to synthesize problem-specific planners, Outubro de 2004. [33](#)
- [JEK⁺90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, e L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. Em *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, páginas 1–33, Washington, D.C., 1990. IEEE Computer Society Press. [46](#)
- [JH02] Simon Peyton Jones e John Hughes. Haskell 98 language and libraries: The revised report, Dezembro de 2002. [69](#)
- [Jon92] Neil D. Jones. Partial evaluation and the generation of program generators. Aceito em *Communications of The ACM*, 1993, 1992. [33](#), [87](#)
- [Kam97] Subbarao Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2):67–97, 1997. [18](#)
- [KMS98] Subbarao Kambhampati, Amol Dattatraya Mali, e Biplav Srivastava. Hybrid planning for partially hierarchical domains. Em *AAAI/IAAI*, páginas 882–888, 1998. [22](#)
- [KN04] Ugur Kuter e Dana S. Nau. Forward-chaining planning in nondeterministic domains. Em *AAAI*, páginas 513–518, 2004. [4](#), [49](#), [50](#), [51](#), [53](#), [54](#), [81](#), [83](#)
- [Kni21] Frank H. Knight. *Risk, Uncertainty and Profit*. Hart, Schaffner, and Marx, 1921. [35](#)
- [KNPT05] Ugur Kuter, Dana S. Nau, Marco Pistore, e Paolo Traverso. A hierarchical task-network planner based on symbolic model checking. Em *ICAPS*, páginas 300–309, 2005. [4](#), [49](#), [54](#), [61](#), [70](#), [71](#), [81](#)

- [KNRG07] Ugur Kuter, Dana Nau, Elnatan Reisner, e Robert Goldman. Conditionalization: Adapting forward-chaining planners to partially observable environments. Em *17th International Conference on Automated Planning & Scheduling (ICAPS'07), 3rd Workshop on Planning and Execution for Real-World Systems: Principles and Practices for Planning in Execution*, Providence, Rhode Island, Estados Unidos, 22–26 de Setembro de 2007. [47](#), [83](#)
- [Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. [89](#)
- [Kri63] Saul Aaron Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963. [46](#)
- [Kri95] Steen Kristensen. Sensor planning with bayesian decision theory. Em *Reasoning with Uncertainty in Robotics*, páginas 353–367, Amsterdam, Holanda, Dezembro de 1995. Springer. RUR'95 Proceedings. [1](#)
- [KS92] Henry A. Kautz e Bart Selman. Planning as satisfiability. Em *European Conference on Artificial Intelligence (ECAI)*, páginas 359–363, 1992. [19](#)
- [KS95] Sven Koenig e Reid G. Simmons. Real-time search in non-deterministic domains. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 1660–1669, 1995. [70](#), [71](#)
- [KS99] Henry A. Kautz e Bart Selman. Unifying SAT-based and graph-based planning. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 318–325, 1999. [19](#)
- [Kut06] Ugur Kuter. *Planning under Uncertainty: Moving Forward*. Tese de Doutorado, University of Maryland, College Park, 2006. [60](#), [81](#), [84](#)
- [LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. [1](#)
- [Lei01] Daan Leijen. Parsec, a fast combinator parser, 2001. [86](#)

- [McL06] Laurianne McLaughlin. Automated programming: The next wave of developer power tools. *IEEE Softw.*, 23(3):91–93, 2006. [1](#)
- [MK98] Amol Dattatraya Mali e Subbarao Kambhampati. Encoding HTN planning in propositional logic. Em *Artificial Intelligence Planning Systems*, páginas 190–198, 1998. [32](#)
- [MOSS99] Markus Müller-Olm, David A. Schmidt, e Bernhard Steffen. Model-checking: A tutorial introduction. Em *SAS*, páginas 330–354, 1999. [45](#)
- [NAI⁺03] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, e Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003. [4](#), [32](#), [49](#), [51](#), [52](#)
- [NAI⁺05] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Héctor Muñoz-Avila, J. William Murdock, Dan Wu, e Fusun Yaman. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2):34–41, 2005. [33](#)
- [NCLMA99] Dana S. Nau, Yue Cao, Amnon Lotem, e Héctor Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 968–975, 1999. [32](#), [69](#), [82](#)
- [Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, Reino Unido, 1998. [85](#)
- [Per06] Sílvio do Lago Pereira. Planejamento não-determinístico baseado em verificação de modelos. qualificação de doutorado, Universidade de São Paulo, Instituto de Matemática e Estatística, Agosto de 2006. [40](#), [99](#), [102](#)
- [Per07] Sílvio do Lago Pereira. *Planejamento sob incerteza para metas de alcançabilidade estendidas*. Tese de Doutorado, Universidade de São Paulo, Instituto de Matemática e Estatística, Novembro de 2007. [3](#), [39](#), [40](#), [47](#)
- [Pnu77] Amir Pnueli. The temporal logic of programs. Em *18th IEEE Symposium on the Foundations of Computer Science (FOCS)*, páginas 46–57, 1977. [23](#), [52](#)

- [PV93] Y. Parrod e S. Valera. OPTIMUM-AIV: A planning tool for spacecraft AIV. *Preparing for the Future, ESA's Technology Programme Quarterly*, 3(3):7–9, Setembro de 1993. European Space Agency. [21](#)
- [RN03] Stuart Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edição, 2003. [1](#), [35](#), [36](#), [37](#)
- [Rov01] Marco Roveri. Planning in non-deterministic domains via symbolic model checking. *Periodico dell'Associazione Italiana per l'Intelligenza Artificiale*, 4:13–21, 2001. [4](#), [47](#), [62](#)
- [Sac75] Earl D. Sacerdoti. The nonlinear nature of plans. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 206–214, 1975. [22](#)
- [SBTM02] Tran Cao Son, Chitta Baral, Nam Tran, e Sheila McIlraithT. Domain-dependent knowledge in answer set planning. 2002. [34](#)
- [Som98] F. Somenzi. Cudd: CU decision diagram package release, 1998. University of Colorado, Boulder. [47](#)
- [SPW⁺04] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler, e Dana S. Nau. HTN planning for web service composition using SHOP2. *Journal of Web Semantics*, 1(4):377–396, 2004. [21](#), [33](#)
- [SS94] Leon Sterling e Ehud Shapiro. *The Art of Prolog*. The MIT Press, 2^a edição, Março de 1994. [85](#)
- [Tat77] Austin Tate. Generating project networks. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 888–893, 1977. [22](#), [23](#)
- [TCdB07] Felipe W. Trevizan, Fabio G. Cozman, e Leliane N. de Barros. Planning under risk and knightian uncertainty. Em *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007. [35](#), [81](#), [84](#)
- [Ull05] Carsten Ullrich. Course generation based on HTN planning. Em *LWA*, páginas 74–79, 2005. [21](#)

- [vR01] Femke van Raamsdonk. Book review - logic in computer science: Modeling and reasoning about systems by michael r. a. huth and mark d. ryan, cambridge university press. *Theory and Practice of Logic Programming*, 1(1):123–125, 2001. [35](#)
- [Wad92] Philip Wadler. The essence of functional programming. Em *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 1–14, New York, NY, USA, 1992. ACM. [89](#)
- [Wel94] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994. [17](#)
- [Wil88] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, Setembro de 1988. [21](#)
- [Wil90] David E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6:232–246, 1990. [22](#)
- [WPS⁺03] Dan Wu, Bijan Parsia, Evren Sirin, James A. Hendler, e Dana S. Nau. Automating DAML-S web services composition using SHOP2. Em *International Semantic Web Conference*, páginas 195–210, 2003. [33](#)
- [WSH⁺03] Dan Wu, Evren Sirin, James A. Hendler, Dana S. Nau, e Bijan Parsia. Automatic web services composition using SHOP2. Em *WWW (Posters)*, 2003. [33](#)
- [Yan90] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6:12–24, 1990. [3](#), [22](#)

Índice Remissivo

- π -ancestrabilidade, [40](#)
- π -descendência, [40](#)
- árvores de computação, [100](#)
- agentes, [1](#)
- análise de alcançabilidade, [18](#)
- aplicabilidade
 - de ações, [8](#), [13](#)
- backtracking, *veja* retrocesso, [86](#)
- busca
 - progressiva, [15](#)
 - regressiva, [15](#)
- contra-exemplo, [45](#)
- controlador, [8](#)
- decomposição de tarefas, [21](#)
- determinismo, [35](#)
- domínio de planejamento, [10](#)
 - qualitativo, [46](#)
- domínios
 - determinísticos, [35](#)
 - não-determinísticos, [35](#)
 - probabilísticos, [35](#)
- efeitos, [13](#)
- estado, [7](#)
 - inicial, [10](#)
 - meta, [10](#)
- estados de crença, [37](#)
- estrutura de Kripke, [45](#)
- exclusão mútua, [18](#)
- fluentes, [14](#)
- grafo de planejamento, [17](#)
- Haskell, [85](#)
- HTN, [21](#)
- incerteza, [3](#), [35](#)
 - Knightiana, [35](#)
- lógica
 - CTL, [100](#)
 - de primeira ordem, [88](#)
 - LTL, [100](#)
 - modal, [99](#)
 - proposicional, [99](#)
- método de decomposição, [21](#)
- métodos
 - STN, [27](#)
- malha

- aberta, 9
- fechada, 9
- metas, 10
 - de alcançabilidade, 35
 - de decomposição, 22
 - de otimização, 35
 - estendidas, 35, 65
 - maximização de utilidade, 35
- modelo de Kripke, 99
- modelos temporais, 46
- mundo
 - fechado, 12
- mundo dos blocos, 14
- mundo fechado, 14
- não-determinismo, 36
 - ilimitado, 38
 - limitado, 37
- ND-transformação, 49
- objetivo, *veja* metas
- observabilidade
 - nula, 35
 - parcial, 35
 - total, 35
- operadores, 14
 - não-determinísticos, 88
 - temporais, 100
- persistência, 85
- planejador, 1, 9
- planejamento, 1
 - clássico, 3, 9
 - condicional, 38
 - conformante, 37
 - contínuo, 38
 - contingente, 38
 - específico de domínio, 1
 - hierárquico, 3, 21
 - independente de domínio, 1
 - não-determinístico, 35
 - no espaço de estados, 15
 - no espaço de planos, 16
 - possibilístico, 35
 - probabilístico, 35
 - sem sensores, 37
- plano, 10
- planos, 13
- pré-condições, 13
 - de métodos, 21, 26, 28
- problema de planejamento
 - clássico, 10
- redes de tarefas, 21, 27
- refinamento disjuntivo, 18
- relações
 - flexíveis, 14
 - rígidas, 14
- replanejamento, 38, 83
- retrocesso, 15
- SAT, 18
- satisfazibilidade, *veja* SAT
- sistema de transição de estados, 7, 8
- STN, 26
- STRIPS, 9
- tarefas, 21

compostas, [22](#), [24](#)

não-primitivas, [24](#)

primitivas, [22](#), [24](#)

vínculos causais, [17](#)

verificação de modelos, [4](#), [45](#)

simbólicos, [46](#)