Jhonatan Alves

Desenvolvimento de um Sistema de Planejamento Automático baseado na redução ao problema SAT

FLORIANÓPOLIS – SC, Brasil

Julho de 2013

Jhonatan Alves

Desenvolvimento de um Sistema de Planejamento Automático baseado na redução ao problema SAT

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientadora:
Profª. Dr. Jerusa Marchi

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

FLORIANÓPOLIS – SC, Brasil

Julho de 2013

Trabalho de Conclusão de Curso apresentado para a disciplina de Projetos II do curso de Ciências da Computação da Universidade Federal de Santa Catarina - UFSC, sob o título "*Desenvolvimento de um Sistema de Planejamento Automático baseado na redução ao problema SAT*", apresentado por Jhonatan Alves em 17 de junho de 2013, Florianópolis - SC, com banca avaliadora constituída pelos seguintes professores:

Profª. Dr. Jerusa Marchi
Universidade Federal de Santa Catarina

Prof. Dr. Mauro Roisenberg
Universidade Federal de Santa Catarina

Prof. Dr. Olinto José Varela Furtado
Universidade Federal de Santa Catarina

Prof. Dr. Ricardo Azambuja Silveira
Universidade Federal de Santa Catarina

Dedicatória

Dedico este trabalho aos meus pais, Leomar Argentil Alves e Marlei Angelita Borges Alves, pelo apoio que me foi dado ao longo destes anos de graduação, pelo carinho, amor sem par e por me fazerem acreditar na realização dos meus sonhos.

Agradecimentos

Gostaria de agradecer, primeiramente, a professora Jerusa Marchi pela sua excelente orientação ao longo deste trabalho, pelo seu incentivo e, também, pela sua amizade.

Aos professores membros da banca avaliadora, pelo apoio e sugestões que contribuíram para um trabalho de qualidade.

Ao Anderson Camargo, que infelizmente não está mais entre nós, por ter sido meu melhor amigo, por sempre ter me escutado e me ajudado nas tomadas de decisões que me fizeram seguir sempre o caminho correto.

A todos, muito obrigado.

"*O que você ganha por alcançar os seus objetivos não é tão importante quanto o que você se torna ao atingí-los*".

Henry David Thoreau

Resumo

Planejamento Automático é a área da Inteligência Artificial que se preocupa com a representação do conhecimento e com o desenvolvimento de mecanismos de manipulações eficientes, que possibilitem a um agente autônomo raciocinar e decidir sobre um conjunto de ações que, quando executadas em uma determinada sequência, permitem ao agente cumprir com a sua tarefa. Neste trabalho, apresenta-se uma visão geral da área de Planejamento Automático, focando na técnica de planejamento chamada de SATPLAN (planejamento SAT). Nesta técnica, o problema de planejamento é reduzido a uma fórmula lógica, cujos símbolos proposicionais representam estados do mundo e ações do agente. O plano que solucionar o problema passa a ser, então, uma atribuição de valores verdade para os quais a fórmula lógica é válida. O objetivo deste trabalho é o desenvolvimento de um sistema de planejamento automático que utilize SATPLAN como técnica de busca de planos para instancias de problemas de planejamento.

Palavras-chave: planejamento automático, satisfatibilidade booleana, redução.

Abstract

Automated planning is an Artificial Intelligence area that deals with knowledge representation and with the development of mechanisms for efficient manipulation that enable an autonomous agent to reason and decide over a set of actions, when executed in a particular sequence, allow the agent to comply with his tasks. In this work, an overview of automated planning is presented, focusing on the SATPLAN planning technique. In this technique, the planning problem is reduced to a logical formula, whose propositional symbols represent states of the world and the agent's actions. The plan that solves the problem becomes an atribution of truth values for which the logical formula is satisfiable. The objective of this work is to develop an automated planning system that uses SATPLAN as a technique of plan search for automated planning instances.

Keywords: automated planning, boolean satisfiability, reduction.

Sumário

1 Introdução p. 1

1.1 Objetivo . . . . p. 2

1.1.1 Objetivo geral . . . . p. 2

1.1.2 Objetivos específicos . . . . p. 2

1.2 Organização do trabalho . . . . p. 3

2 Fundamentação Teórica p. 4

2.1 Lógica Proposicional . . . . p. 4

2.1.1 Conceito de Lógica Proposicional . . . . p. 6

2.1.2 Sintaxe e Semântica da Lógica Proposicional . . . . p. 7

2.1.3 Equivalência Lógica . . . . p. 10

2.1.4 Formas Normais Canônicas . . . . p. 11

2.2 Complexidade Computacional . . . . p. 13

2.2.1 Alfabeto, Cadeias e Linguagens . . . . p. 13

2.2.2 Problemas de Decisão . . . . p. 14

2.2.3 A Máquina de Turing . . . . p. 15

2.2.4 Reduções . . . . p. 20

2.2.5 Classes de Problemas P e NP . . . . p. 20

2.2.6 O Problema da Satisfatibilidade Booleana . . . . p. 22

2.3 Planejamento em Inteligência Artificial . . . . p. 28

2.3.1 Histórico . . . . p. 29

2.3.2 Definição Conceitual de Planejamento Automático . . . . p. 31

2.3.3 Definição Formal de Planejamento Automático . . . . p. 31

2.3.4 Cálculo Situacional . . . . p. 34

2.3.5 A linguagem STRIPS . . . . p. 36

2.3.6 Planejamento SAT . . . . p. 39

3 Desenvolvimento de um Sistema de Planejamento Automático p. 47

3.1 O bloco Compilador . . . . p. 48

3.1.1 Análise Léxica . . . . p. 48

3.1.2 Análise Sintática . . . . p. 50

3.1.3 Análise Semântica . . . . p. 53

3.2 O bloco Codificador . . . . p. 54

3.3 O bloco Resolvedor SAT . . . . p. 56

3.4 O bloco Decodificador . . . . p. 56

4 Resultados p. 57

4.1 Testes sobre o Mundo das Células . . . . p. 57

4.1.1 Testes sobre uma extensão do Mundo das Células . . . . p. 58

4.2 Testes sobre o Mundo dos Blocos . . . . p. 60

5 Conclusão p. 62

Referências p. 65

1 Introdução

O ato de planejar a solução de um problema é algo extremamente importante, pois, fazendo-o é possível realizar tarefas complexas que envolvam situações que desconhecemos ou que envolvam ambientes de alto risco, atividades que necessitam de um grande número de pessoas trabalhando em conjunto ou até mesmo quando os recursos usados para resolver um problema são limitados.

A capacidade de planejar é uma característica inerente a seres inteligentes, pois, para planejar é necessário raciocinar, aprender, errar, etc. O planejamento nos ajuda a prever certos comportamentos de um sistema, evitar situações indesejáveis, cortar custos, entre outros problemas.

Dentre as áreas de destaque da Inteligência Artificial está a área de *Planejamento Automático* que é responsável por pesquisar técnicas que permitam a um agente resolver de modo automático problemas de planejamento. Um problema de planejamento automático é um modelo de estados com transições determinísticas cuja solução é dada por um *plano* elaborado por um agente. Quando o plano é executado o problema é resolvido [1].

Durante alguns anos a área de planejamento automático careceu de boas propostas para o seu crescimento, no entanto em meados dos anos 90 os pesquisadores Kautz e Selman [2] propuseram uma técnica inovadora. Esta técnica consiste em reduzir o problema do planejamento automático ao problema da *satisfatibilidade booleana* e extrair um plano da fórmula lógica proposicional encontrada caso ela seja satisfazível. Esta proposta se mostrou muito eficiente e rápida principalmente pelo fato de que os algoritmos usados para tratar o problema da satisfatibilidade booleana como, por exemplo, *GSat* [3] e

WalkSat [4] foram aprimorados e tornaram-se algoritmos velozes e robustos.

Alguns anos mais tarde o domínio dos métodos de pesquisa ramificou-se e nasceram aqueles que se baseiam em redes de Petri [5], funções heurísticas [6], grafos [7], raciocínio temporal [8], etc.

Na década de 80 a área de planejamento automático começou a ganhar mais espaço entre as pesquisas em IA. Competições internacionais como *International Planning Competition* [9], *Artificial Intelligence Planning System* [10], *International Competition on Knowledge Engineering For Planning and Scheduling* [11] entre outras surgiram com o propósito de incentivar mais pesquisa nesta área e uma série de problemas complexos passaram a ser resolvidos por técnicas dessa área como, por exemplo, problemas de controle de redes de satélites, movimentação de robôs, sistemas de manufatura, entre outros [1].

Neste cenário o presente trabalho foca no estudo da técnica de redução de instancias de problemas de planejamento automático ao problema SAT.

1.1 Objetivo

1.1.1 Objetivo geral

O objetivo deste trabalho é o desenvolvimento de um sistema de planejamento automático que busque planos para problemas de planejamento reduzindo-os ao problema da satisfatibilidade booleana.

1.1.2 Objetivos específicos

Como objetivos específicos tem-se:

- Estudar a área de planejamento automático.
- Estudar o problema da satisfatibilidade booleana (SAT).
- Investigar técnicas que reduzam problemas de planejamento ao problema SAT.

- Estudar e escolher um resolvedor SAT para decidir satisfatibilidade dos problemas reduzidos.
- Calcular um plano para o problema quando a sua redução representa uma fórmula lógica satisfazível.

1.2 Organização do trabalho

Este trabalho está organizado do seguinte modo:

- Capítulo 2 apresenta uma revisão bibliográfica dos tópicos aos quais este trabalho está relacionado. A saber: Lógica Proposicional, Complexidade Computacional e Planejamento em Inteligência Artificial.
- Capítulo 3 apresenta o desenvolvimento do sistema de planejamento proposto.
- Capítulo 4 apresenta os testes realizados sobre o sistema de planejamento desenvolvido e resultados obtidos.
- Capítulo 5 apresenta as conclusões e considerações obtidas ao longo do desenvolvimento deste trabalho.

2 Fundamentação Teórica

Este capítulo apresenta um resumo introdutório dos conceitos teóricos referentes aos temas contemplados neste trabalho. Inicialmente apresenta-se a definição de Lógica Proposicional, focando no problema da satisfatibilidade booleana. Em seguida, conceitos relativos à reduções e complexidade computacional são apresentados. E por fim fala-se do problema do planejamento automático e estratégias para solucioná-lo.

2.1 Lógica Proposicional

A lógica nasceu há mais de 23 séculos e o seu desenvolvimento ocorreu em distintos lugares como China, Índia, Grécia entre outros. Mas foi na Macedônia que o filósofo Aristóteles (384 - 322 a.C) definiu a lógica como hoje a conhecemos e reuniu seus principais escritos em uma obra intitulada *Organon*(do grego *instrumento*) [12]. A lógica Aristotélica era fundamentada na linguagem natural e era considerada como uma ferramenta para auxiliar o pensamento.

Aristóteles preocupou-se em estudar as estruturas do raciocínio de tal forma a aplicá-las na investigação da verdade. Os passos necessários para uma investigação eram metodológicos e seguiam as seguintes etapas [13]:

- Obter conhecimento sobre os fenômenos de interesse através de observações.
- Obter os princípios que explicavam as ocorrências de tais fenômenos através de indução.

- Deduzir as causas dos fenômenos através dos princípios obtidos.

Se estas etapas fossem bem elaboradas então, segundo Aristóteles, qualquer conclusão obtida seria considerada válida.

O coração da lógica Aristotélica era o *silogismo*: argumento composto por premissas que inferem uma conclusão válida. O silogismo era a marca dos raciocínios típicos de Aristóteles. O argumento:

"Todo homem é mortal. Sócrates é homem. Logo, Sócrates é mortal".

É um exemplo clássico de silogismo. Os estudos realizados por Aristóteles foram muito significativos para o estudo da lógica porém a lógica Aristotélica sofria algumas limitações por possuir ambiguidade semântica e também pelo fato da linguagem natural não permitir uma transformação fácil da lógica em um cálculo manejável.

O grego Crísipo de Soles (280-206 a.C.) foi responsável pela origem da *lógica proposicional*. Crísipo estudou o uso dos conectivos *"e"*, *"ou"* e *"se"* em sentenças declarativas. O valor (verdadeiro ou falso) de uma sentença de lógica proposicional dependia do valor das frases ligadas pelos conectivos [14].

No século $XVII$, o matemático alemão Leibniz introduziu, em sua obra *Dissertatio de arte combinatória* [15], o conceito de uma lógica simbólica de caráter matemático baseada no modelo de cálculo algébrico de sua época. Leibniz também propôs a construção de uma linguagem universal que seria capaz de expressar de forma transparente o pensamento humano. Embora os conceitos propostos por Leibniz não pudessem, em sua maioria, ser concretizados os seus feitos foram de grande utilidade para a lógica contemporânea [16].

No século XIX, George Boole (1815-1864) deu continuidade a obra de Leibniz e definiu formalmente, para o domínio da lógica proposicional, o cálculo algébrico, chamado de álgebra booleana ou álgebra de boole, em sua obra *Mathematical Analysis of Logic* [17]. A álgebra booleana é constituída por um conjunto de operadores e axiomas. As

proposições que compõem uma fórmula desta álgebra podem assumir um dentre os dois possíveis valores: verdadeiro (T) ou falso (F). São os postulados, propriedades e teoremas fundamentais criados por Boole que fundamentam toda a base da *eletrônica digital* que, por sua vez, contribuiu para o surgimento dos computadores [18].

No final do século XIX, os estudos realizados por Giuseppe Peano (1858-1932) tentaram mostrar que toda a matemática poderia ser expressa por cálculos de lógica. Gottlobe Frege (1848-1925) introduziu a idéia de quantificadores e a formação de regras de inferência primitiva [14].

A lógica passou a ser base formal para diversas áreas da Matemática e outros campos da ciência. Em IA, a lógica, se tornou ferramenta para a representação e processamento de conhecimento, prova automática de teoremas e sistemas especialistas, principalmente por possuir uma teoria semântica bem definida o que a torna interessante para ser usada nesta área [19].

2.1.1 Conceito de Lógica Proposicional

A lógica proposicional é um formalismo matemático composto por uma linguagem formal e um conjunto de regras de inferência que, sobre um argumento, determinam a sua validade [20]. Um argumento é formado por um conjunto de premissas que estão relacionadas entre si e acompanhadas de uma conclusão. Um argumento é dado como válido se e somente se a sua conclusão é inferida pelas suas premissas.

Os elementos de um argumento são representados por proposições que, por sua vez, são elementos básicos de uma fórmula lógica. Proposições são representadas por símbolos, em geral por símbolos de alfabetos, como o latino, por exemplo.

Toda proposição, obrigatoriamente, tem como características [21]:

- Ser declarativa.
- Possuir um entre dois valores lógicos: verdadeiro ou falso. Esta regra é conhecida

como *princípio do terceiro excluído* [21].

- Ter sujeito e predicado.

Por exemplo, as proposições:

- p = *2 é um número ímpar.*
- q = *O Brasil é um país sul-americano.*

Possuem valores falso e verdadeiro, respectivamente e têm como sujeitos *2* e *Brasil*.

O argumento *2 multiplicado por 5 é igual a 10?* não pode ser considerado uma proposição já que não é declarativo.

Uma linguagem formal consiste em:

- Um conjunto de variáveis atômicas.
- Um conjunto de operadores lógicos.

A área de estudo que se concentra na atribuição de valores verdade às variáveis lógicas afim de determinar quando uma fórmula é verdadeira chama-se Teoria de Modelos [22] [23]. Já a Teoria das Provas estuda a aplicação das regras de inferência em fórmulas lógicas e o valor dos resultados obtidos [22].

2.1.2 Sintaxe e Semântica da Lógica Proposicional

Em Lógica Proposicional são permitidos os seguintes símbolos:

- conectivos lógicos: $\wedge$, $\vee$, $\neg$, $\rightarrow$ e $\leftrightarrow$.
- constantes lógicas: verdade (V) e falso (F).
- símbolos de pontuação: ( ).

- proposições.

Uma conjunção de duas proposições p e q refere-se a proposição $p \wedge q$ cujo valor lógico é verdadeiro somente quando p e q também o são. Uma disjunção refere-se a proposição $p \vee q$ cujo valor lógico é verdadeiro se pelo menos uma das duas proposições for verdadeira. A atribuição de valores verdade às proposições constitui a semântica daquela proposição.

Uma implicação é uma proposição do tipo $p \rightarrow q$ que significa que o evento p deve ocorrer para que q também ocorra. Esse tipo de fórmula é denominada condicional e lê-se "se p então q". Um condicional possui valor lógico falso somente quando a proposição antecedente é verdadeira e a consequente é falsa. Por exemplo:

1. Se ele praticar exercícios regularmente, ele terá uma vida saudável.
2. Se ele não praticar exercícios regularmente, ele terá problemas de saúde.
3. Se ele praticar exercícios regularmente, ele terá uma boa autoestima.
4. Ele não tem autoestima.

Realizando o processo de formalização sobre estes argumentos obtém-se as seguintes proposições:

p: "ele pratica exercícios regularmente".

q: "ele terá uma vida saudável".

r: "ele terá problemas de saúde".

s: "ele terá uma boa autoestima".

Deste modo, as seguintes fórmulas lógicas podem ser escritas a partir das proposições obtidas:

1. $p \rightarrow q$

2. $\neg p \rightarrow r$

3. $p \rightarrow s$

4. $\neg s$

A tabela 2.1 sumariza a semântica dos conectivos lógicos de conjunção e disjunção.

p	q	p ∧ q	p ∨ q
F	F	F	F
F	V	F	V
V	F	F	V
V	V	V	V

Tabela 2.1: Tabela Verdade para as operações de conjunção e disjunção

A Tabela Verdade 2.2 se refere a operação de negação.

p	¬p
F	V
V	F

Tabela 2.2: Tabela Verdade para a operação de negação

Tautologias são proposições que para qualquer que sejam os valores lógicos de seus operadores sempre assumem um valor verdadeiro. Uma tautologia é chamada de verdade universal [22]. A Tabela Verdade 2.3 se refere a Lei do Terceiro Excluído [21].

p	¬p	p ∨ ¬p
F	V	V
V	F	V

Tabela 2.3: Tabela Verdade para $p \vee \neg p$

Ao passo que tautologias são sempre verdadeiras uma contradição sempre assume valor falso independente do valor de seus operadores. A Tabela Verdade 2.4 se refere a contradição $p \wedge \neg p$.

Se uma fórmula lógica possuir alguma atribuição de valores verdade que a torne *verdadeira*, então ela é dita *satisfazível*. Por exemplo, a atribuição de valores verdade à uma fórmula lógica segundo a Tabela 2.3 sempre irá satisfazer tal fórmula.

p	¬p	p ∧ ¬p
F	V	F
V	F	F

Tabela 2.4: Tabela Verdade para p ∧ ¬p

2.1.3 Equivalência Lógica

Duas fórmulas p e q são ditas equivalentes se suas Tabelas Verdades também o forem. A equivalência entre p e q é representada por $p \equiv q$ ou pelo conectivo bicondicional $p \leftrightarrow q$ e lê-se “p se e somente se q”.

A tabela 2.5 reúne um conjunto de proposições equivalentes.

Nome	Equivalência
Comutatividade	$(p \wedge q) \leftrightarrow (q \wedge p)$ $(p \vee q) \leftrightarrow (q \vee p)$
Associatividade	$p \wedge (q \wedge r) \leftrightarrow (p \wedge q) \wedge r$ $p \vee (q \vee r) \leftrightarrow (p \vee q) \vee r$
Distributividade	$p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$
Leis de De Morgan	$\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$ $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$
Meio Excluído	p $\vee \neg p \leftrightarrow 1$
Contradição	p $\wedge \neg p \leftrightarrow 0$
Lei da Dupla Negação	$\neg(\neg p) \leftrightarrow p$
Lei da Contrapositiva	p $\rightarrow q \leftrightarrow \neg q \rightarrow \neg p$
Condicional como disjunção	$p \rightarrow q \leftrightarrow \neg p \vee q$
Negação do condicional	$\neg(p \rightarrow q) \leftrightarrow (p \wedge \neg q)$
Leis de Idempotência	$p \wedge p \leftrightarrow p$ $p \vee p \leftrightarrow p$
Leis de Absorção	$p \wedge (p \vee q) \leftrightarrow p$ $p \vee (p \wedge q) \leftrightarrow p$
Leis de Dominância	$p \vee 1 \leftrightarrow 1$ $p \wedge 0 \leftrightarrow 0$
Leis de Exportação	$p \rightarrow (q \rightarrow r) \leftrightarrow (p \wedge q) \rightarrow r$
Leis de Identidade	$p \wedge 1 \leftrightarrow p$ $p \vee 0 \leftrightarrow p$

Tabela 2.5: Tabela Verdade de Relações de Equivalência

2.1.4 Formas Normais Canônicas

Formas normais são fórmulas lógicas proposicionais que possuem um formato padrão de como seus literais devem ser alocados. As duas principais formas normais são: Forma Normal Conjuntiva (FNC) e Forma Normal Disjuntiva (FND).

Forma Normal Conjuntiva

Um fórmula lógica está em uma forma normal conjuntiva se ela é formada por conjunções de cláusulas e as cláusulas, por sua vez, são formadas por disjunções de literais [24]:

$$(\alpha_{11} \vee \ldots \vee \alpha_{1k_1}) \wedge \ldots \wedge (\alpha_{n1} \vee \ldots \vee \alpha_{nk_n})$$

São exemplos de FNCs:

- $p \wedge q$ onde p *e* q são cláusulas.
- $(p \vee q) \wedge (r \vee t)$ onde $(p \vee q)$ e $(r \vee t)$ são as cláusulas e p, q, r e s são literais.

Toda fórmula lógica possui uma FNC equivalente. Os passos que transformam uma fórmula lógica em uma FNC estão descrito como segue:

1. Remover símbolos bicondicionais

   $(\alpha \leftrightarrow \beta) \leftrightarrow ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha))$

2. Remover símbolos de implicação

   $(\alpha \rightarrow \beta) \leftrightarrow (\neg\alpha \vee \beta)$

3. Aplicar as leis De Demorgan e Dupla Negação

   $\neg(\alpha \wedge \neg\beta) \leftrightarrow (\neg\alpha \vee \beta)$

4. Aplicar a Distributividade da conjunção sobre a disjunção (ver Tabela 2.5)

Seja $\neg(p \leftrightarrow \neg q)$ uma fórmula lógica proposicional. A sua transformação para a FNC equivalente é dada como segue:

1. $\neg((p \rightarrow \neg q) \wedge (\neg q \rightarrow p))$
2. $\neg((\neg p \vee \neg q) \wedge (q \vee p))$
3. $(p \wedge q) \vee (\neg q \wedge \neg p)$
4. $(p \wedge q) \vee (\neg q \wedge \neg p)$
5. $((p \wedge q) \vee \neg q) \wedge ((p \wedge q) \vee \neg p)$
6. $(p \vee \neg q) \wedge (q \vee \neg p)$

Para que uma FNC seja satisfazível é necessário que pelo menos uma proposição de cada cláusula seja assinada como verdadeira.

Forma Normal Disjuntiva

Um fórmula lógica esta em sua forma normal disjuntiva se ela é formada por disjunções de cláusulas conjuntivas[24]:

$$(\alpha_{11} \wedge \ldots \wedge \alpha_{1k_1}) \vee \ldots \vee (\alpha_{n1} \wedge \ldots \wedge \alpha_{nk_n})$$

São exemplos de FNDs:

- $p \vee q$ onde p *e* q são cláusulas.
- $(p \wedge q) \vee r$ onde $(p \wedge q)$ e r são cláusulas e p e q são literais.

Toda fórmula lógica possui uma FND equivalente. Os passos para se obter a FND equivalente são iguais aos passos para se obter a FNC equivalente com exceção do último passo que, neste caso, se da por distribuir a disjunção sobre a conjunção. Para que uma FND seja satisfazível é necessário que todos os literais de pelo menos uma cláusula sejam assinados como verdadeiros.

2.2 Complexidade Computacional

Esta seção apresenta conceitos de Teoria de Linguagens e Complexidade de Algoritmos. Tais conceitos são importantes para a compreensão do funcionamento das máquinas de Turing e da reduções de problemas.

2.2.1 Alfabeto, Cadeias e Linguagens

No contexto da Teoria da Computação, uma linguagem nada mais é, do que um conjunto de sentenças sobre um alfabeto. Por sua vez, um alfabeto é definido como um conjunto finito e não-vazio de símbolos. Um símbolo é uma entidade abstrata sem definição formal. Um alfabeto é representado pelo símbolo Σ. São exemplos de alfabetos: $\Sigma = \{a, b\}$, $\Sigma = \{0, 1, x, y, \#, \%\}$

Uma cadeia de tamanho n sobre um determinado alfabeto é uma sequência formada pelos símbolos do alfabeto em questão, e que, tem comprimento n[25]. O comprimento da cadeia n é denotado por $|n|$. Dado $\Sigma = \{0, 1\}$, são exemplos de cadeias com tamanhos diferentes, $\Sigma' = 0, 11, 0011, 010101$, com tamanhos iguais a 1, 2, 4, e 6 respectivamente.

A operação *estrela de Kleene* define todas as potências sobre uma linguagem L e é denotada por L^*. Ou seja:

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

Se $L = \Sigma$ então L^* é o conjunto de todas as cadeias possíveis sobre o alfabeto. Para

o alfabeto do exemplo anterior tem-se $\Sigma^* = \{\varepsilon, 0, 1, 00, 01,$
$10, 11, 110, \ldots\}$. Nesse contexto ε refere-se a cadeia de comprimento 0 que também é uma sentença sobre qualquer alfabeto, onde $|\varepsilon| = 0$.

Se p e q são cadeias, então, uma operação de concatenação sobre elas é representada por pq. A concatenação é uma operação que consiste em unir os símbolos de duas cadeias formando uma *nova cadeia*[1] em que os símbolos da primeira são seguidos pelos símbolos da segunda. Se $p = ab$ e $q = cde$ então $pq = abcde$. A concatenação sucessiva da cadeia n é expressa por n^x, tal que, x é o número de vezes em que os símbolos de n são concatenados sucessivamente.

2.2.2 Problemas de Decisão

Um problema de decisão é um problema formulado de modo a admitir apenas uma dentre duas possíveis respostas: "SIM" ou "NÃO". Se X é um problema de decisão, X_1 é o conjunto de instancias para as quais a resposta é *sim* e X_2 é o conjunto de instancias para as quais a resposta é *não*, então, $X = X_1 \cup X_2$.

Por exemplo, *"y é um número primo?"* é um problema de decisão. O algoritmo que resolve este problema para uma instancia com $y = 3$ responde *sim*.

Uma instancia que responde *sim* é chamada de instancia positiva, caso contrário, é chamada de negativa [25]. Nem todo problema possui um algoritmo correspondente que o decida. Tais problemas são chamados de *indecidíveis*.

Um problema indecidível clássico é o problema da *parada* que consiste em determinar se uma máquina de Turing pára ao computar uma cadeia, ou seja, se em algum momento ela a aceita ou a rejeita (termina a sua execução). A prova da indecidibilidade de um problema é feita através da redução do problema da parada a ele [26].

[1] Dizemos que a operação de concatenação não é, necessariamente, fechada sob uma linguagem L, pois, a nova cadeia formada pode não pertencer a L.

2.2.3 A Máquina de Turing

A máquina de Turing é um modelo abstrato de um computador proposto por Alan Turing em 1936 [27]. Este modelo implementa apenas os aspectos lógicos do funcionamento de um computador porém reproduz com exatidão o significado do termo computação. Turing demonstrou que sua máquina é capaz de realizar qualquer cálculo imaginável [19]. Church formulou uma hipótese sobre o poder de computação desta máquina, afirmando [28]:

> *"A capacidade de computação representada pela Máquina de Turing é o limite máximo que pode ser atingido por qualquer dispositivo de computação".*

Ou seja, qualquer mecanismo que computa um algoritmo tem capacidade máxima de processamento igual a de uma máquina de Turing. Em sua Tese, Church, disserta que a máquina é capaz de processar as mesmas informações que um computador real processaria.

Apesar de a Tese de Church ser amplamente aceita na comunidade científica ela não pode ser provada formalmente, primeiramente, porque a noção de algoritmo é intuitiva, logo não é matematicamente precisa, e em segundo lugar porque a hipótese de Church não agrega um resultado matemático, já que o que ele formulou não foi um teorema. Entretanto, a Tese é passível de ser "desmentida" se alguém propuser um novo modelo de computação que seja comprovadamente apto a realizar computações que uma máquina de Turing não é capaz de fazer [29].

Uma máquina de Turing M é dita um decididor se para toda cadeia $w \in \Sigma^*$ a máquina pára, isto é, M aceita ou rejeita w. Denotamos $L(M)$ o conjunto de cadeias aceitas por uma máquina de Turing M.

Todo problema computacional pode ser reescrito como um problema de reconhecimento de linguagem, tal que, cada instancia do problema é mapeado para uma cadeia. Usamos M para decidir se uma cadeia qualquer pertence a uma linguagem de interesse.

As linguagens que são reconhecidas por uma máquina de Turing formam o conjunto das *linguagens recursivas* [30]. Porém nem toda máquina decide uma linguagem. Algumas máquinas processam indefinidamente uma cadeia e não representam a idéia de algoritmo. A estas linguagens nos referimos como linguagens indecidíveis.

Uma máquina de Turing tem como memória uma fita dividida em células que se estende infinitamente à direita e que é delimitada à esquerda pelo símbolo $\triangleright$. Possui um cabeçote que se movimenta para a direita ou esquerda sobre a fita a fim de ler e gravar um símbolo nela. A função de transição δ determina para que direção o cabeçote deve se movimentar, qual símbolo escrever e qual será o novo estado da máquina, ou seja, define o comportamento da máquina de um passo para outro.

A máquina é alimentada com uma cadeia w que é gravada na posição mais a esquerda da fita logo após o símbolo $\triangleright$. As demais posições são marcadas com o símbolo $\sqcup$ o qual informa ao cabeçote que elas estão em branco. As células em branco podem ser preenchidas pela máqina durante a computação de w se assim for necessário. Quando o processo de computação começa o cabeçote é posicionado sobre o primeiro símbolo de w, e se movimenta sobre uma célula por vez.

A figura 2.1 mostra o esquema de uma máquina de Turing.

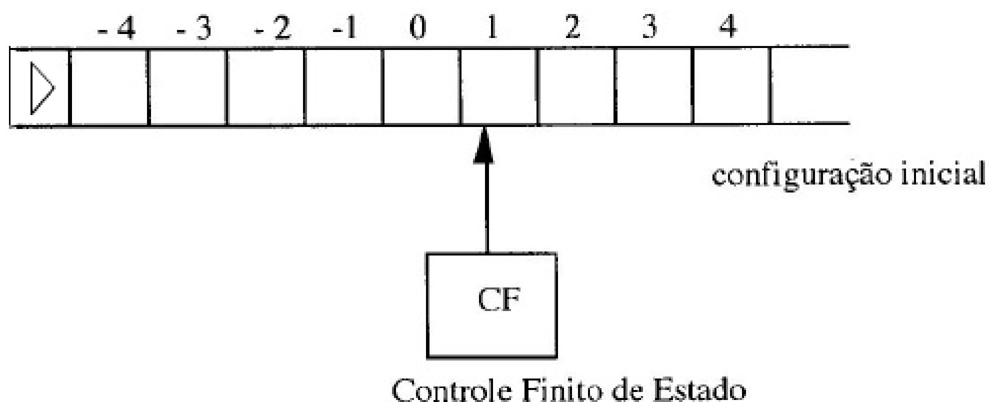

Figura 2.1: Esquema de uma máquina de Turing M [25].

Uma máquina de Turing M é definida como uma sétupla [25]:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{aceita}, q_{rejeita})$$

Onde, Q é o conjunto finito de estados. Σ é o alfabeto de entrada sendo que $\sqcup \notin \Sigma$. $\Sigma \subset \Gamma$ e $\Gamma \cup \sqcup$ é o alfabeto da fita. A função de transição é definida como $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ onde $\times$ refere-se ao produto cartesiano e L e R referem-se *esquerda* (Left) e *direita* (Right), respectivamente. A produção de novas configurações é descrita como $\delta(q_i, w_i) = (q_j, w_j, x)$ onde $x \in \{L, R\}$.

O estado $q_0 \in Q$ referece-se ao estado inicial da máquina, $q_{aceita} \in Q$ e $q_{rejeita} \in Q$ são, respectivamente, os estados de aceitação e rejeição. Os estados de aceitação e rejeição devem ser disjuntos.

O processo de computação sobre a cadeia w segue as regras descritas em δ e sempre que o cabeçote lê o símbolo $\triangleright$ ele se move imediatamente à direita. A computação de w só pára quando a máquina entra em algum estado de aceitação ou rejeição, caso contrário, a computação permanece ativa indeterminadamente.

Uma configuração é a combinação entre o conteúdo corrente da fita, o estado em que a máquina se encontra e a localização do cabeçote. Esquematicamente, uma configuração pode ser escrita como uqv onde u e v representam o conteúdo da fita e q o estado atual da máquina. Uma configuração sempre produz uma outra configuração até que a máquina aceite a cadeia de entrada [25]. Por exemplo, se $u = ab$, $v = cde$ e q_2 é o estado atual da máquina, então a configuração atual seria: abq_2cde. O cabeçote estaria sobre a célula que contém o símbolo c.

A computação de uma entrada $w = w_1w_2 \ldots w_n \in \Sigma^*$ segue da seguinte forma: no início tem-se a seguinte configuração $q_0w_1w_2 \ldots w_n$ (esta é a configuração inicial para qualquer cadeia) que produzirá uma nova configuração de acordo com δ como, por exemplo, a configuração $w_4q_1w_2 \ldots w_n$. Esta nova configuração é dada pela seguinte função de

transição $\delta(q_0, w_1) = (q_1, w_4, R)$ que significa que quando a máquina está no estado q_0 e lê o símbolo w_1 ela passa para o estado q_1 sobrescrevendo w_1 por w_4 e o cabeçote se move para a célula à direita daquela em que se encontrava.

Seja L a linguagem constituída pelas cadeias de 0's cujo comprimento é uma potência de 2 denotada por $A = \{0^{2^n} | n \geq 0\}$. Seja $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ a máquina de Turing que reconhece L sendo $Q = \{q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$, $\Sigma = \{0\}$ e $\Gamma = \{0, x, \sqcup\}$. Sendo q_1 o estado inicial, q_{accept} o estado de aceitação e, por fim, q_{reject} o estado de rejeição.

O diagrama da figura 2.2 representa os estados de M_1. os vértices representam os estados e as arestas representam as trocas de configurações entre os estados q_i e q_j. Uma configuração que está rotulada como $a \rightarrow b, c$ significando que a é sobrescrito por b e o cabeçote se move para a direção c.

A descrição em alto nível para M_1 indicando os passos que a máquina executa seriam:

1. Avance até o final da fita (ou seja, até encontrar $\sqcup$) trocando um 0 por x e outro não, sucessivamente.
2. Se a cadeia de entrada era composta por apenas um único 0 então aceite.
3. Se não, se a cadeia de entrada continha um comprimento ímpar de 0's então rejeite.
4. Retorne o cabeçote para o inicio da fita.
5. Retorne ao estágio 1.

Um outro modelo computacional, não menos importante, que o que fora até aqui apresentado é a máquina de Turing não-determinística [25].

A diferença entre uma máquina de Turing determinística e uma não-determinística está no fato de que em qualquer ponto de uma computação a máquina não-determinística pode realizar mais de uma ação. A função de transição para este tipo de máquina tem a forma $\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$, onde $P(A)$ refere-se ao conjunto potência de A.

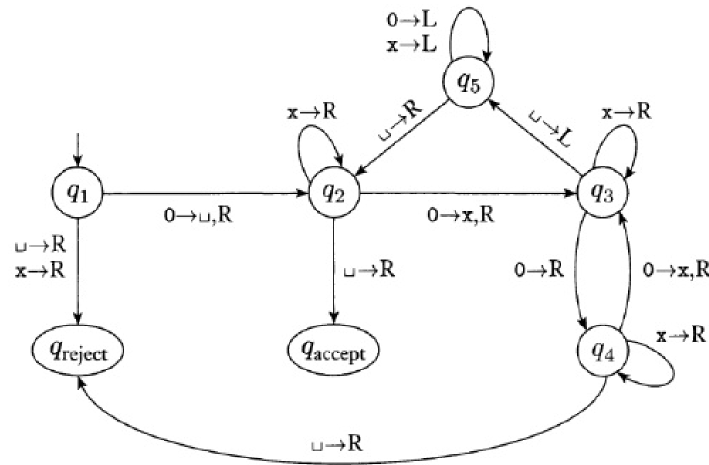

Figura 2.2: Diagrama de estados para M_1 [25].

A computação na máquina não-determinística gera uma árvore cujos os ramos representam as diferentes possibilidades de ações para a máquina. Se algum ramo tomado leva a máquina a um estado de aceitação, dizemos então, que a Máquina de Turing não-determinística aceita a entrada. Uma entrada só é rejeitada se todos os ramos tomados levam a um estado de rejeição. Encontrar um estado de aceitação ou rejeição significa realizar uma busca em profundidade na árvore.

A figura 2.3 mostra a diferença entre a computação em uma máquina de Turing determinística e uma máquina de Turing não-determinística, onde $f(n)$ corresponde ao número de trocas de configurações que uma máquina de Turing determinística levaria para decidir uma cadeia de tamanho n.

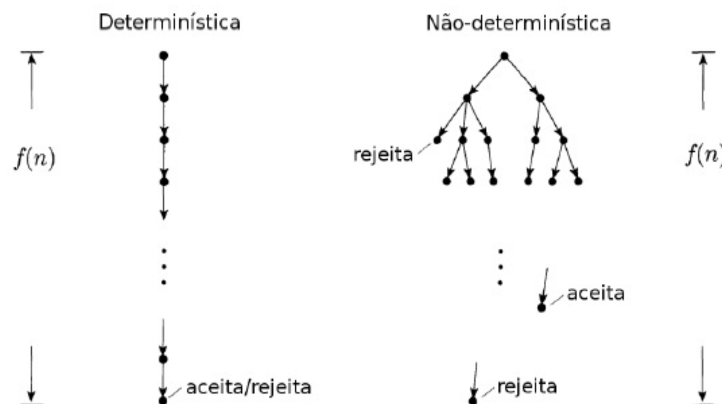

Figura 2.3: Troca de configurações entre máquinas de Turing determinísticas e máquina de Turing não-determinísticas [25].

2.2.4 Reduções

Uma redução implica converter um problema do domínio A em um problema do domínio B, se for conhecido um algoritmo para B. A solução encontrada para B é, então, solução para A. Se A for um problema indecidível, B também o será.

Uma redução por mapeamento consiste em uma função computável f, não necessariamente sobrejetora, que mapeia cadeias de A em cadeias de B, descrita como $A \leq_f B$, é dada de acordo com a seguinte regra:

$$w \in A \leftrightarrow f(w) \in B$$

Se f existe então diz-se que A é redutível por mapeamento a B.

Se A e B puderem ser reduzidos um ao outro em tempo polinomial então são ditos problemas polinomialmente equivalentes. Um problema de decisão A é redutível a um problema de decisão B se existe uma máquina de Turing que, ao computar uma cadeia $w \in A$ produz um problema $v \in B$ com resposta equivalente a w.

A redução de um problema pode ser um processo difícil a ponto de ser mais prático resolver o problema original do que se tentar achar a função de redução. Uma redução difícil pode implicar, mas não necessariamente, uma solução difícil para o problema reduzido [25]. Uma redução para um problema A a um problema B consiste em encontrar um algoritmo X que solucione A e que, por sua vez, usa uma subrotina S afim de resolver B, tal que, se S é polinomial então X também o é [31].

2.2.5 Classes de Problemas P e NP

O fato de um algoritmo resolver um problema não significa que na prática ele seja aceitável, pois, recursos como tempo e espaço podem ser altamente requeridos pelo algoritmo em tempo de execução que pode vir a ser um processo custoso. A quantidade de recursos consumidos pelo algoritmo é relativa à *complexidade* do problema [30].

A quantidade de tempo e espaço estão, respectivamente, relacionados à velocidade e à memória consumidos por um algoritmo ótimo, ou seja, aquele que, dentre um conjunto de algoritmos que resolvem o problema possui a menor complexidade. A complexidade de um problema expressa a quantidade de trabalho despendido por um algoritmo na busca por uma solução. Quanto maior for a complexidade do problema mais trabalho o algoritmo exercerá para terminar a sua execução [30].

Os problemas estão divididos em classes de acordo com a sua complexidade. As classes *P*, *NP* e *NP-Completo*[2] dividem os problemas de acordo com o tempo que seus algoritmos tomam para encontrar uma solução.

A classe *P* é constituída pelo conjunto de todos os problemas de decisão aceitos por máquinas de Turing [32]. Estes problemas são ditos tratáveis já que são resolvidos em tempo $O(n^k)$ onde n é o tamanho da instancia do problema para alguma constante k. Ou seja, são resolvidos em tempo polinomial.

A classe *NP* é constituída pelo conjunto de todos os problemas de decisão que são *verificados* em tempo polinomial por uma máquina de Turing determinística. Encontrar uma solução para um problema desta classe demandaria grande esforço computacional, pois, a busca tomaria muito tempo e ocuparia uma quantidade considerável de memória. Essas características classificam estes problemas como problemas *intratáveis* [25].

Uma linguagem L é verificável se a ela está associada uma máquina de Turing determinística M que aceita sentenças de L em um tempo limitado por uma função polinomial.

Uma grande questão em aberto dentro da Teoria da Computação é se $P = NP$. Saber se essas duas classes são equivalentes significaria que um problema que é polinomialmente verificável também é polinomialmente decidível. Sem sucesso, muitos investigadores tentaram responder a esta questão. Existem grandes indícios de que $P \neq NP$ [30]. A prova

[2] Existem algumas classes que reúnem problemas de acordo com a quantidade de memória consumida pelos algoritmos que os resolvem. São exemplos as classes: PSpace, EXPSpace e PSPACE-Completo. As classes PSpace e EXSpace referem-se aos problemas de decisão que ocupam uma quantidade polinomial e exponencial de memória, respectivamente. A classe PSPACE-Completo é formada pelo conjunto de problemas mais difíceis em PSpace.

pela desigualdade também é alvo de pesquisas. Provar que $P \neq NP$ acarretaria dizer que não é possível encontrar um algoritmo rápido que substitua a busca por força-bruta [25].

A classe *NP-Completo* refere-se ao subconjunto dos problemas mais difíceis que pertencem a NP [30]. Se for achado um meio de resolvê-los em tempo polinomial qualquer problema em NP também o será. Isso porque um problema em NP pode ser reduzido a um problema *NP-Completo* em tempo polinomial.

O conjunto *NP-Completo* foi descoberto por Stephen Cook e Leonid Levin por volta de 1970 e tal descoberta foi considerada um avanço sobre a questão P versus NP [25].

O primeiro problema provado ser *NP-Completo* foi o problema da *satisfatibilidade booleana* através do *Teorema de Cook*. A prova da *NP-Completude* de um problema é feita através da redução do problema da satisfatibilidade a ele [30].

2.2.6 O Problema da Satisfatibilidade Booleana

O problema da satisfatibilidade booleana (SAT), é um problema de decisão NP-Completo que consiste em determinar a existência de alguma atribuição de valores verdade que satisfaça uma fórmula lógica na forma normal conjuntiva. A busca pelo conjunto de valores verdade consiste, basicamente, em determinar todas as possíveis combinações de valores verdade e testá-las sobre a fórmula lógica. Diferentemente da busca, um teste é executado em tempo polinomial [30].

Em 1971 Cook identificou o problema da satisfatibilidade booleana como o primeiro problema NP-Completo provando que todo problema pertencente a classe NP pode ser reduzido em tempo polinomial ao problema SAT [33], ou seja, qualquer máquina de turing determinística pode ser transformada em uma fórmula em lógica proposicional.

Algumas variações do problema SAT estão relacionadas ao número de proposições que formam as cláusulas da fórmula lógica. Cláusulas compostas por duas proposições classificam a instancia SAT como 2-SAT. Cláusulas compostas por três proposições classificam a instancia SAT como 3-SAT.

Problemas 2-SAT podem ser resolvidos em tempo polinomial por uma máquina de Turing não-determinística com uma quantidade de memória $O(log\ n)$, ao passo que problemas 3-SAT são resolvidos em tempo $O(2^n)$.

O estudo sobre SAT se tornou popular em áreas como IA, lógica, engenharias entre outras. Existe uma série de problemas reais onde o problema SAT se aplica, como problemas de design VLSI, planejamento automático, criptografia, circuitos integrados etc. Devido a grande importância do problema SAT, algumas competições foram criadas com o intuito de incentivar desenvolvimento de novos resolvedores SAT como a *SAT Competition*[3], *SAT-Race*[4] e *SAT Challenge*[5].

Solucionadores SAT

Um solucionador SAT é um algoritmo de decisão para o problema da satisfatibilidade booleana. Este algoritmo recebe uma instancia SAT como entrada e realiza uma busca por um conjunto de valores verdade que satisfaça a instancia. Em virtude de SAT ser um problema NP-Completo a busca pelo conjunto verdade demanda grande esforço computacional.

Na literatura existem diferentes propostas de como a busca pelo conjunto verdade deve ser realizada afim de diminuir ao máximo a demanda computacional. Este trabalho apresenta as heurísticas de busca dos resolvedores DPLL [34], GSat [3] e MiniSat [35].

DPLL é um algoritmo de busca, completo, baseado em *backtracking* para encontrar um conjunto de valores verdade que satisfaça uma fórmula lógica em sua forma normal conjuntiva. Este algoritmo é a reformulação do algoritmo de Davis-Putnam [36].

Este algoritmo faz uso de duas técnicas: *propagação unitária* e *eliminação de literais puros*. A primeira técnica afirma que se uma cláusula é unitária, ou seja, contém apenas um literal, ela pode ser satisfeita atribuindo a ela o valor verdade V. A segunda técnica

[3]http://www.satcompetition.org/
[4]http://baldur.iti.uka.edu/

afirma que toda cláusula que contém um literal puro, ou seja, aquele que não possui complemento na fórmula lógica, pode ser satisfeita pela atribuição de um valor verdade V a ele. Estas duas técnicas tornam o espaço de busca do algoritmo DPLL menor do que o espaço de busca do algoritmo Davis-Putnam.

A execução do algoritmo DPLL consiste, inicialmente, em eliminar da fórmula lógica cláusulas unitárias e cláusulas que contenham literais puros. Em seguida é escolhido aleatoriamente um literal e lhe é atribuído um valor verdade. Caso esta atribuição satisfaça a fórmula lógica o algoritmo pára a sua execução e retorna que a fórmula é satisfazível. Caso esta atribuição torne falso todos os literais de uma cláusula então o algoritmo detecta um *conflito*. Se nada se pode afirmar sobre a satisfatibilidade da fórmula lógica até a última atribuição de um valor verdade a um literal, então, um novo literal é escolhido e um valor verdade lhe é atribuído.

Ao detectar um conflito, o algoritmo retorna seu ponto de execução sobre o último literal que recebeu apenas uma única atribuição de valor verdade. Supondo que x_0 seja tal literal então lhe é atribuído seu valor complementar e todas as atribuições realizadas após x_0 são desfeitas.

O pseudo algoritmo de DPLL está descrito em *Algorithm 1.*

Por exemplo, seja $H = (x_1 \vee x_2) \wedge (x_3) \wedge (x_5 \vee \neg x_1) \wedge (x_1 \vee \neg x_2)$ uma fórmula lógica na FNC. O primeiro passo do algoritmo elimina as cláusulas unitárias, neste caso a cláusula (x_3). O segundo passo elimina as cláusulas com literais puros, neste caso a cláusula $(x_5 \vee \neg x_1)$. Após esses passos serem efetuados a fórmula lógica passa a ser $H = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$.

Em seguida o literal x_1 é escolhido arbritariamente e lhe é atribuído o valor F. Com essa designação nada se pode afirmar sobre a fórmula H. Então, o literal x_2 é escolhido e lhe é atribuído o valor F o que provoca conflito na cláusula $(x_1 \vee x_2)$. Assim, o algoritmo retorna ao literal x_2 e lhe atribui o valor V. Após esta atribuição todas as cláusulas são satisfeitas e o algoritmo retorna que H é satisfazível.

Algorithm 1 Pseudo Algoritmo DPLL

```
 1: procedure DPLL(Fórmula H)
 2:     Elimine todas as cláusulas unitárias e
 3:     todas as cláusulas com literais puros em H
 4:     while true do
 5:
 6:         if status = satifazível then
 7:             return satifazível
 8:         end if
 9:
10:         if status = insatifazível then
11:             return insatifazível
12:         end if
13:
14:         if status = desconhecido then
15:             escolher um literal x
16:             atribuir V ou F a x
17:             definir novo status
18:         end if
19:
20:         if status = conflito then
21:             encontrar o último literal x que não possui
22:             duas atribuições
23:             if x não existe then
24:                 status = insatifazível
25:             else
26:                 atribuir a x o valor de seu complemento
27:                 definir novo status
28:             end if
29:         end if
30:     end while
31: end procedure
```

GSat é um algoritmo guloso que, inicialmente, gera um conjunto aleatório de valores verdade para uma fórmula lógica H na FNC e em seguida tenta satisfazer o maior número de cláusulas através da troca do valor verdade de uma de suas proposições. O número de trocas é realizado até que se encontre um conjunto de valores verdade que satisfaça a fórmula H ou até que o número máximo de trocas seja atingido.

O pseudo algoritmo de GSat está descrito em *Algorithm 2*.

Por exemplo, seja $H = (x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_3)$ uma fórmula lógica na FNC. As operações realizadas para encontrar o conjunto de valores verdade que

Algorithm 2 `Pseudo Algoritmo GSat`

```
1: procedure GSat(H, MAX-TROCAS, MAX-TENTATIVAS)
2:    for i = 1 até MAX-TENTATIVAS do
3:       σ = conjunto aleatório de valores verdade
4:       for j = 1 até MAX-TROCAS do
5:          if σ satisfaz H then
6:             retorne σ
7:          end if
8:          X = conjunto de proposições que satisfazem o
9:          maior número de cláusulas em H quando seus valores
10:         verdade são alterados
11:         ρ = proposição aleatória de X
12:         σ = σ com o valor de ρ alterado
13:      end for
14:   end for
15:   return insatisfazível
16: end procedure
```

a satisfaz estão listadas na tabela 2.6.

Operação	Interpretação	Cláusulas SAT	Candidatos	fliping
	x_1 x_2 x_3	x_1 x_2 x_3		
1	0 1 0	3 3 2	x_1 x_2	x_1
2	1 1 0	1 4 4	x_2 x_3	x_2
3	1 0 0	1 3 4	x_3	x_3
	1 0 1			

Tabela 2.6: Conjunto de operações calculadas pelo GSat.

Inicialmente a interpretação 0 1 0 é gerada aleatoriamente para as proposições x_1, x_2, e x_3, respectivamente. GSat calcula, então, o número de cláusulas satisfazíveis ao realizar a troca de valor verdade de cada proposição em H. As proposições x_1 e x_2 são ditas *candidatas* já que ambas satisfazem três cláusulas, enquanto x_3 satisfaz apenas duas cláusulas. GSat escolhe aleatoriamente a proposição x_1 como a proposição a sofrer oficialmente a troca de valor verdade e calcula a nova interpretação.

Como a nova interpretação 1 1 0 não satisfaz H é necessário reiniciar o processo de busca de qual proposição satisfaz o maior número de cláusulas. Este processo se repetiu até que o conjunto 1 0 1 fosse encontrado, ou seja, o conjunto que satisfaz H.

MiniSat é um resolvedor SAT que faz uso das técnicas de *aprendizagem de cláusula* e *backtraking dirigido a conflito* para encontrar o conjunto de valores verdade baseado no algoritmo DPLL. MiniSat é um algoritmo veloz e extremamente eficiente, vencedor de alguns prêmios, entre eles o de resolvedor mais eficiente na categoria industrial da *SAT Competition.*

A busca se dá expandindo uma árvore de possibilidades fixando um valor verdade a cada proposição. Se em algum momento da busca um conflito é detectado, ou seja, o valor assinado a uma proposição torna a fórmula falsa, o algoritmo realiza backtraking, assina um novo valor a proposição conflitante e reexpande a árvore.

No momento em que um conflito é detectado, a cláusula que contém a proposição conflitante é analisada para que a causa do conflito seja compreendida. O algoritmo gera uma cláusula especial, cláusula de aprendizagem, que representa o motivo conflito. A cláusula de aprendizagem é armazenada em uma base de dados que é utilizada pelo MiniSat a medida que a árvore de busca é expandida para que assinaturas conflitantes sejam detectadas e, assim, evitadas. A análise de conflito indica, também, para qual nível da árvore deve-se realizar backtraking. O backtraking dirigido a conflito adiciona um ganho de desempenho ao MiniSat.

O Formato DIMACS

DIMACS é um formato que simplifica a representação de problemas escritos em lógica proposicional. É amplamente utilizando pela comunidade científica na realização de testes para o problema da satisfatibilidade booleana. Uma fórmula lógica descrita em DIMACS possui a seguinte estrutura:

COMENTÁRIOS p FORMATO PROPOSIÇÕES CLÁUSULAS

Onde, COMENTÁRIOS é um conjunto de comentários sobre o problema. Cada

linha de comentário deve iniciar com a letra c. A letra p indica o início da descrição do problema. FORMATO indica o formato da fórmula lógica. A quantidade de proposições e cláusulas da fórmula lógica são determinadas por PROPOSIÇÕES e CLÁUSULAS, respectivamente.

Cada proposição presente em uma cláusula é representada por um número inteiro i, tal que, $1 \leq i \leq$ PROPOSIÇÕES. Uma proposição semanticamente falsa é representada por -i. Cláusulas são separadas umas das outras pelo número 0.

Considere a fórmula lógica H da seção 2.2.6.3 como exemplo de codificação para o formato DIMACS:

```
c um exemplo
p cnf 3 4
-2  1 0
-2 -3 0
-1  3 0
-3  1 0
```

A resposta para um problema satisfazível é uma lista de números inteiros que representam suas proposições. Estes números expressam o valor semântico das proposições assinadas como verdadeiras ou falsas no momento em que o problema foi dado como satisfazível. Para H, teria-se como retorno a lista: 1 -2 3.

2.3 Planejamento em Inteligência Artificial

Esta seção apresenta o problema do planejamento em Inteligência Artificial. Primeiramente, é descrito de forma breve o histórico desta área. Em seguida são apresentadas a definição conceitual e formal do problema do planejamento. Logo após, fala-se da representação do conhecimento deste problema através do uso do Cálculo de Situações e da linguagem STRIPS. E por fim, o problema do planejamento tratado como o problema da satisfatibilidade booleana é abordado.

2.3.1 Histórico

Em meados da década de 50 pesquisadores da área de Computação enfrentavam a difícil tarefa de formalizar problemas para que estes fossem resolvidos por computadores. Os algoritmos que atravessavam o espaço de estados em busca de soluções eram excessivamente complexos e muitas vezes específicos para cada problema.

Na década de 60 as pesquisas focaram a criação de solucionadores gerais, ou seja, solucionadores que fossem capazes de resolver problemas de qualquer domínio como, por exemplo, GPS [37] e QA3 [38]. A maioria dos algoritmos desta época recebiam como entrada problemas descritos em *Cálculo Situacional* [38] e *Cálculo de Eventos* [39].

O desenvolvimento e aperfeiçoamento de solucionadores gerais representou um marco para a área de Inteligência Artificial, porém a idéia de que qualquer problema fosse passível de ser resolvido pelos solucionadores gerais nunca se concretizou. No começo dos anos 70 os pesquisadores Fikes e Nilsson da Universidade de Stanford desenvolveram a linguagem STRIPS [40] para a modelagem de domínios de problemas. STRIPS deu origem aos estudos de planejamento em IA marcando esta época como a *Éra Clássica do Planejamento Automático* [1].

Entre os planejadores automáticos de destaque dos anos 70 estão ABSTRIPS [41] e NOAH [42]. Os primeiros planejadores sofriam de vários problemas, entre eles o processo de busca, que era exaustiva no espaço de estados. Nos anos 80, houve uma redução significativa nas pesquisas relacionadas ao planejamento devido às limitações enfrentadas para a geração automática de soluções e a falta de boas propostas para tais problemas. A maioria dos planejadores propostos até metade da década de 90 tiveram como maior problema o desempenho de seus sistemas.

Somente em 95 com o surgimento do planejador GRAPHPLAN [7] que a comunidade de IA voltou a investir pesado nas pesquisas em planejamento automático. O planejador GRAPHPLAN realiza a busca por um plano em uma estrutura chamada *grafo de planejamento*. Uma série de estudos mostra que GRAPHPLAN era o sistema de planejamento

mais rápido, até então. Este planejador marcou uma nova era na área de planejamento em IA conhecida como a *Éra Neoclássica do Planejamento Automático* [1].

Em 96 Henry Kautz e Bart Selman lançaram o planejador SATPLAN [2] que compilava problemas de planejamento em fórmulas proposicionais afim de encontrar um conjunto de valores verdade que satisfizesse a fórmula. O alto desempenho do planejador SATPLAN permitiu a cientistas trabalharem com problemas de domínios de alto grau de complexidade e, também, proporcionou o desenvolvimento de novos planejadores que se basearam nas técnicas exploradas pelo SATPLAN, a exemplo dos planejadores MEDIC [43] e BLACKBOX [44].

Novas técnicas de planejamento automático surgiram no ano de 98, como o *planejamento por busca heurística* [45] que usava uma função de custo na busca por uma solução tornando o processo de busca muito rápido. No mesmo ano foi anunciada a linguagem PDDL [46] para a representação do conhecimento de problemas de planejamento. PDDL passou a ser a linguagem padrão para a descrição de problemas planejamento e obrigatória para planejadores que participassem de competições como a *International Planning Competition* [5], *Artificial Intelligence Planning System*[6] e *International Conference on Automated Planning and Scheduling*[7].

Estas competições fizeram com que surgissem novos planejadores cada vez mais eficientes como em planejador FF [47] baseado em buscas heurísticas no ano de 2000 e no ano de 2004 o planejador SATPLAN'04 [48] (evolução do SATPLAN) que obteve excelentes resultados em velocidade de resposta e ganhou a competição.

Atualmente, as principais pesquisas em planejamento automático estão relacionadas ao escalonamento de recursos, onde os planejadores consistem em encontrar um plano que determine uma sequência de ações, o tempo de duração de cada ação e alocar recursos que uma ação possa necessitar para que ela seja concluída.

[5]http://ipc.informatik.uni-freiburg.de/
[6]http://www.cs.cmu.edu/ aips98/
[7]http://www.icaps-conference.org/

2.3.2 Definição Conceitual de Planejamento Automático

Uma das características eminentes de seres inteligentes é a capacidade de raciocinar sobre problemas de diferentes circunstâncias e decidir quais passos deverão ser tomados para que tais problemas sejam solucionados. Este processo de deliberar ações e a ordem em que elas ocorrem é tipicamente chamado de *planejamento*.

Com o intuito de reproduzir a capacidade de planejar objetivos, a IA passou a dedicar-se ao estudo de técnicas de representação do conhecimento e a construção de planejadores para solucionar problemas de planejamento. Este campo da IA passou a ser chamado de *planejamento automático*. Um problema de planejamento automático é definido como o processo que determina mecanicamente uma sequência finita de ações linearmente ordenadas a serem aplicadas a um conjunto finito de estados. A *ordem* em que as ações são executadas formam o *plano*, ou *solução*, que alcançará o objetivo do problema [1].

Um problema de planejamento automático abstrai fatores como, por exemplo, tempo, custo, recursos e pessoas envolvidas, que, não deveriam ser ignorados em áreas como o planejamento financeiro, tático, estratégico etc. Tais fatores são excluídos porque dificultam a modelagem do domínio de um problema de planejamento automático [1].

Modelar o domínio de um problema de planejamento é uma tarefa complexa que depende do conhecimento que temos sobre o problema e da representação de tal conhecimento. A modelagem se torna mais complexa a medida que o número de variáveis de um problema cresce. Muitas instancias de problemas de planejamento possuem uma explosão combinatória de variáveis. Alguns destes problemas pertencem a *NP-Completo*. Determinar se existe um plano para uma instancia de planejamento automático é um problema *PSPACE-Completo* [49].

2.3.3 Definição Formal de Planejamento Automático

Formalmente, um problema de planejamento é uma tripla $\rho = \langle \Delta, s_0, G \rangle$ onde [1]:

- Δ é o sistema de transição de estados.
- s_0 é o estado inicial.
- G é o conjunto de objetivos.

Um sistema de transição de estados restrito, por sua vez, também é uma tripla $\Delta =< S, A, \gamma >$ tal que:

- $S = \{s_1, s_2, \ldots, s_n\}$ é o conjunto finito de estados.
- $A = \{a_1, a_2, \ldots, a_n\}$ é o conjunto finito de ações.
- $\gamma : S \times A \rightarrow S$ é a função de transição de estados.

A tripa ρ é passada como parâmetro para o planejador que buscará por um plano, ou seja, uma sequência linear de ações que quando aplicadas sobre o estado inicial s_0 satisfaz G. As ações do plano modificam os estados do mundo a medida que são executadas, isto é, $s_1 = \gamma(s_0, a_1)$, $s_2 = \gamma(s_1, a_2)$, ..., $G = \gamma(s_{n-1}, a_n)$, se e somente se $\gamma(s_j, a_i) \neq \emptyset$ para qualquer ação a_i e estado s_j.

Um sistema de transição de estados Δ pode ser representado por um grafo orientado, simplesmente mapeando os estados e as ações para os vértices e arestas correspondentes, respectivamente. O plano encontrado para esta abordagem corresponde a um caminho no grafo que percorre o estado inicial ao estado meta. Este é o modo mais simples de se realizar planejamento, porém é ineficiente já que o grafo que representa Δ pode ser muito grande mesmo para problemas pequenos, tal que, o tamanho do espaço de busca é exponencial.

O uso de heurísticas é desejável em casos de problemas onde o número de estados e ações sejam numerosos. Deste modo, o sistema de transição de estados tende a ser o mais compacto possível. Alguns planejadores implementam heurísticas como o *HSP* [45] e *FF* [47].

Problemas de planejamento automático devem compartilhar das seguintes características [1]:

- Ser *Completamente observável*, ou seja, sempre é possível saber com precisão qual é o estado corrente do mundo.
- Ser *Finito* na quantidade de estados.
- Ter *Ações determinísticas e sequências.*
- Ser *Estático*, ou seja, eventos externos ao agente não o influenciam durante a busca pelo plano.
- Ter *Tempo discreto entre ações*, ou seja, apenas uma ação ocorre por vez e o tempo de sua duração é irrelevante.
- Ter *Objetivos restritos*, ou seja, só é permitido lidar com objetivos que foram especificados explicitamente em G.

A representação de conhecimento é um quesito de grande importância na descrição de um problema de planejamento automático. A má representação de conhecimento certamente fará com que o planejador encontre um plano de qualidade inferior ou impossibilite que um plano seja encontrado. A descrição deve ser feita através de alguma linguagem que o planejador possa compreender. Comumente a modelagem de domínio é realizada através da *Teoria de Conjuntos*, *Variáveis de Estados* e *Representação Clássica*, esta última é a mais utilizada [1].

As linguagens mais utilizadas na Representação Clássica são as linguagens *STRIPS* [40] e *PDDL* [46]. Porém, é possível representar o conhecimento sobre problemas de planejamento através de outras técnicas como, por exemplo, o Cálculo Situacional [38] que é descrito na próxima seção devido a sua importância histórica para o desenvolvimento de novas linguagens de descrição de problemas de planejamento e conceitos herdados pelo planejamento tratado como um problema SAT.

2.3.4 Cálculo Situacional

O Cálculo Situacional é um dialeto da *lógica de primeira ordem* capaz de expressar dinamicamente as mudanças do mundo através do tempo [50]. Neste contexto, o mundo consiste em uma sequência de *situações* que, por sua vez, denotam o estado do mundo após a ocorrência de uma ação. Por exemplo, *empilhar(r,x,y)*, denota uma ação onde o robô r *empilha* o objeto x sobre o objeto y.

Fluentes são predicados que tem sua semântica modificada com o avanço do tempo. Seus valores são alterados pela ocorrência de ações. O último argumento de um fluente sempre será uma situação. Por exemplo, *segurando(r,y,s)* denota um fluente onde o robô r está segurando o objeto y na situação s.

O Cálculo Situacional emprega duas funções especiais: a função *do(a,s)* denota a nova situação s' do mundo pela aplicação da ação a na situação s e a função *poss(a,s)* que denota que é possível executar a ação a na situação s.

A solução para o problema do planejamento automático tratado sobre o domínio do Cálculo de Situações consiste em encontrar uma sequência de ações que quando executadas a partir de uma situação inicial s_0 resultam em uma nova situação que satisfaz o conjunto G de objetivos, ou seja [50]:

$$\mathcal{A} \wedge \mathcal{I} \models \mathrm{G}(\mathrm{do}(\mathrm{a},s_0)) \wedge L(\mathrm{do}(\mathrm{a},s_0))$$

Tal que, $\mathcal{A}$ é o conjunto de axiomas que descrevem as ações, $\mathcal{I}$ é o conjunto de axiomas que descrevem a situação inicial do problema, do(a,s_0) abrevia $\mathrm{do}(a_n,\mathrm{do}(a_{n-1},\ldots,\mathrm{do}(a_1, s_0)\ldots))$ e $L(\mathrm{do}(\mathrm{a},s_0))$ abrevia $\bigwedge_{i=1}^{n} \mathrm{poss}(a_i,\mathrm{do}(\langle a_1,\ldots,a_{i-1}\rangle,s_0))$.

Axiomas de Precondições e Efeitos

Uma ação pode ser aplicada a uma situação apenas quando suas precondições são verdadeiras [50]. Axiomas de precondições descrevem ações em função de suas precondições,

por exemplo:

$$\text{empilhar(r,x,y,s)} \Rightarrow \text{segurando(x,s)} \wedge \text{livre(y,s)}$$

Significa que o robô r poderá realizar a ação *empilhar* somente se na situação s ele estiver segurando o bloco x e se não houver outro bloco sobre o bloco y.

A aplicação de uma ação altera a semântica de certos fluentes e resulta em uma nova situação do mundo. Uma fórmula é chamada de *axioma de efeito positivo* quando esta descreve a mudança do valor de um fluente para verdadeiro após a ocorrência de uma ação. Caso a fórmula descreva a mudança do valor de um fluente para falso ela é chamada de *axioma de efeito negativo*.

Para cada fluente *F(x,s)* podemos reunir em uma única fórmula todos os seus axiomas de efeitos positivos e reunir em uma outra única fórmula todos os seus axiomas de efeitos negativos, como segue:

$$\Pi_F(x, a, s) \Rightarrow F(x, do(a, s)) \tag{2.1}$$

$$N_F(x, a, s) \Rightarrow \neg F(x, do(a, s)) \tag{2.2}$$

Para exemplificar o uso das fórmulas (2.1) e (2.2) considere os seguintes axiomas de efeitos positivos:

$$\text{frágil(x)} \Rightarrow \text{quebrado(x,do(derrubar(r,x),s))}$$

$$\text{proximoDe(b,x,s)} \Rightarrow \text{quebrado(x,do(explodir(b),s))}$$

No primeiro caso se o objeto x for frágil e o robô r o derrubar então x se quebrará. No segundo caso se o objeto b explodir e x estiver próximo a ele então x se quebrará.

Reescrevendo estas fórmulas de acordo com (2.1) tem-se:

$$\exists \mathrm{r}(\mathrm{a} = \mathrm{derrubar(r,x)} \wedge \mathrm{frágil(x)}\ ) \vee \exists \mathrm{b}(\mathrm{a} = \mathrm{explodir(b)} \wedge \mathrm{proximoDe(b,x,s)})$$

$$\Rightarrow \mathrm{quebrado(x,do(a,s))}$$

Axiomas de Estado-Sucessor

As fórmulas (2.1) e (2.2) podem ser combinadas em uma única fórmula chamada de *axioma de estado-sucessor* da seguinte forma [50]:

$$\mathrm{F(x,do(a,s))} \Leftrightarrow \Pi_F(x,a,s) \vee (\mathrm{F(x,s)} \wedge \neg N_F(x,a,s)) \tag{2.3}$$

Esta fórmula diz que um fluente F é verdadeiro na situação *s'* se e somente se a ação *a* torna F verdadeiro ou se F na situação *s* já é verdadeiro e *a* não torna F falso. Esta fórmula será revista na seção 2.3.7 que fala sobre planejamento tratado como um problema SAT.

2.3.5 A linguagem STRIPS

STRIPS é uma linguagem formal derivada da lógica de primeira ordem usada para decompor problemas de planejamento automático em condições lógicas [40]. Formalmente, um problema descrito em STRIPS é uma tripla $\langle A, O, G \rangle$ onde [50]:

- A é um conjunto de literais.
- O é um conjunto de operadores.
- G é um conjunto de objetivos.

As ações do mundo são representadas por elementos básicos denominados *operadores*. Um operador O_i é aplicável somente aos estados que satisfazem os seus pré-requisitos.

Um operador é uma quádrupla

$$O_i : nome((a_1, a_2, \ldots, a_n), PRE, ADD, DEL)$$

onde:

- *nome* é o nome do operador O_i.
- $(a_1, a_2, \ldots, a_n)$ é uma lista de argumentos.
- PRE, ADD, DEL são as lista de pré-condições, efeitos e eliminações, respectivamente.

Uma lista de pré-condições é composta por um conjunto de literais que devem ser válidos no estado em que O_i se aplica. Uma lista de efeitos é composta pelos literais que denotam as mudanças positivas em um estado mediante aplicação de O_i. E a lista eliminação é composta por aqueles literais que se tornam falsos após a aplicação de O_i.

Um estado s_i é representado por uma conjunção de n literais positivos, isto é:

$$s_i : l_1 \wedge l_2 \wedge l_3 \wedge \ldots \wedge l_n$$

Literais não mencionados em s_i não fazem parte deste e são dados como negativos, ou seja:

$$\forall_{lj} \notin s_i \rightarrow \neg l_j$$

Os literais que não pertencem a alguma das três listas do operador O_i não são afetados por ele. Estes literais permanecem com seus antigos valores inalterados quando o sistema realiza a transição de um estado s_i para um estado s_j. Sendo assim, existe apenas a necessidade de descrever a um operador somente àqueles literais que estão relacionados a ele [20].

Problema Exemplo: O Mundo das Células

O problema do Mundo das Células consiste em movimentar um robô sobre um conjunto de células adjacentes. Inicialmente, o robô está posicionado sobre uma célula x e o seu objetivo é chegar a célula y.

Utilizando STRIPS para modelar o domínio do problema, definiu-se o literal EM(x) para indicar sobre qual célula o robô se encontra e o operador MOVER(x,y) para indicar a única ação permitida ao robô. O operador MOVER é definido da seguinte forma:

- MOVER((x,y),
  PRE: EM(x),
  ADD: EM(y),
  DEL: EM(x))

Este operador descreve que para o robô se movimentar da célula x para a célula y é necessário que ele se encontre sobre a célula x. Caso este pré-requisito seja satisfeito, o robô poderá se movimentar para a célula y. Após a aplicação do operador MOVER um novo estado é gerado, onde EM(y) é dado com verdadeiro e EM(x) passa a ser falso.

Uma configuração deste problema consiste em um conjunto de três células adjacentes (A,B,C), onde o estado inicial s_0 e o conjunto de objetivos G são descritos da seguinte forma:

s_0: EM(A)

G: EM(C)

A figura 2.4 ilustra esta configuração do problema.

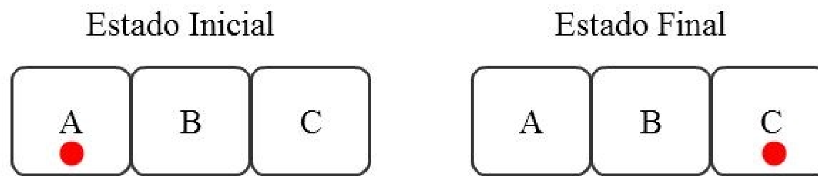

Figura 2.4: Configuração do problema do Mundo das Células com três células. O ponto vermelho indica a posição do robô.

O robô precisa passar por todas as células, iniciando seu trajeto pela célula A, em seguida pela célula B e, finalmente, passando pela célula C onde termina seu caminho.

A busca pelo plano inicia-se aplicando ao estado s_0 o operador MOVER(A,B). A aplicação deste operador modifica a configuração do mundo e produz um novo estado, o estado s_1.

$$s_1\text{: EM(B)}$$

No estado s_1 o robô se encontra sobre a célula B, então o operador MOVER(B,C) tem seus requisitos satisfeitos e ao aplicá-lo sobre o estado s_1 um novo estado, o estado s_2, é gerado.

$$S_2\text{: EM(C)}$$

Neste momento o objetivo foi alcançado e o planejador pára a sua busca respondendo com o *plano* encontrado. O plano é composto pela seguinte sequência de operadores:

1. MOVER(A,B)
2. MOVER(B,C)

2.3.6 Planejamento SAT

Este seção descreve o problema do planejamento automático tratado sob o ponto de vista do problema da satisfatibilidade booleana. Serão apresentados o conceito de

planejamento SAT, o processo de redução de um problema ao outro e, ao final da seção, o problema exemplo do Mundo da Células da seção anterior será usado para exemplificar o processo de redução do problema do planejamento ao problema SAT.

Definição Conceitual de Planejamento SAT

Partindo do princípio de que todo problema NP-Completo pode ser reduzido ao problema da satisfatibilidade [30], os pesquisadores Kautz e Selman propuseram em [51] a formalização da redução do problema do planejamento automático a SAT.

A redução de uma instancia Π de planejamento automático é realizada pela função θ que mapeia Π a uma sentença lógica ω em FNC, tal que, $\omega \equiv \theta(\Pi, i)$ onde $i = 0, \ldots, T_{max}$ representa o i'ésimo instante de tempo para se obter um conjunto σ de valores verdade que satisfaça ω [52].

O valor de i é incrementado à medida que a função de redução obtém um ω insatisfazível. Para que este processo não se repita infinitamente se faz necessário um valor que determine o número máximo de passos, ou seja, T_{max}. Caso este valor seja atingido e o planejador não tenha apresentado um plano o problema é dado como insatisfazível [53]. A fórmula lógica ω é a entrada para algum algoritmo SAT que buscará pelo conjunto σ. Caso o conjunto σ seja encontrado, então, o planejador SAT realizará a tradução do plano extraído de σ para a linguagem na qual o problema de planejamento foi descrito, a fim de descrever o plano como uma sequência de ações.

Este plano é formado pelo conjunto $\{A_0, \ldots, A_n\}$, tal que, para cada i existe uma ação A_i que representa a i'ésima ação do plano e $n \in [0, T_{max})$. Se existirem diferentes conjuntos σ existirão, então, diferentes planos para Π. Os passos que um planejador SAT realiza ao reduzir uma instancia de planejamento estão descritos no Algorithm 3

O valor de T_{max} pode ser definido de duas maneiras:

- Valor arbitrário [54].

Algorithm 3 `Pseudo Algoritmo para redução de problemas de planejamento ao problema da satisfatibilidade booleana`

1: **procedure** PLANNINGTOSAT(`Fórmula lógica` ω)
2:     **for** $i = 0, \ldots, T_{max}$ **do**
3:         $\omega = \theta(\Pi, i)$
4:         **if** σ satisfaz ω **then**
5:             extrair um plano ρ de σ
6:             **return** ρ
7:         **end if**
8:     **end for**
9: **end procedure**

- Valor obtido através de um cálculo de estimativa [53].

Neste trabalho o valor de T_{max} é definido arbitrariamente pelo usuário. A estrutura com suporte matemático será implementada como trabalho futuro.

Estrutura de um planejador SAT

A estrutura típica de um planejador SAT está ilustrada pela figura 2.5.

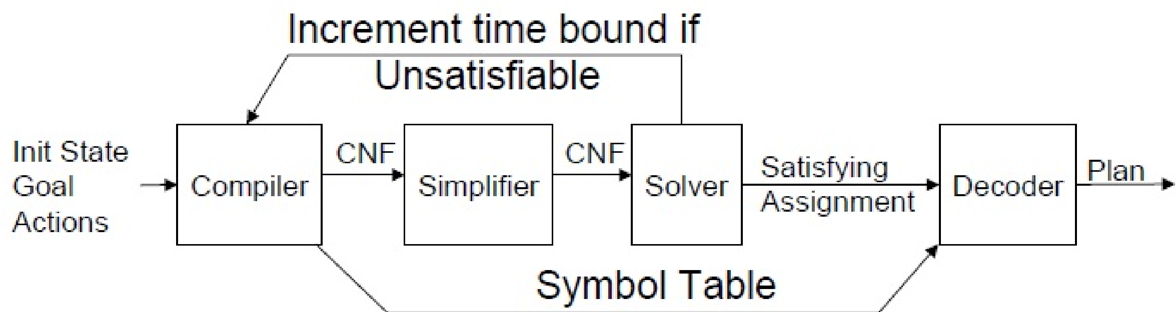

Figura 2.5: Estrutura de um planejador SAT [55].

Os blocos de um planejador SAT são:

- Compiler: recebe o problema descrito em alguma linguagem de representação de conhecimento para problemas de planejamento e reduz o problema a uma fórmula lógica em FNC.

- Symbol table: armazena informações sobre o problema durante a fase de redução do problema para que elas sejam usadas durante a tradução do plano encontrado.
- Simplifier: responsável por reduzir o tamanho da fórmula lógica.
- Solver: algoritmo SAT responsável por encontrar um conjunto de valores verdade que satisfaça a fórmula lógica que recebe.
- Decoder: traduz o plano encontrado, quando a fórmula lógica é satisfazível, em uma sequência de ações.

Redução do problema de planejamento a SAT

A redução de um problema de planejamento ao problema SAT consiste em calcular uma conjunção de axiomas do seguinte modo [56]:

$$\mathrm{F} \equiv s_0 \wedge \left(\bigwedge_{1 \leq i < n} AES_i \wedge AP_i \wedge AE_i \right) \wedge G_n \tag{2.4}$$

Onde:

- s_0 é o axioma de estado inicial.
- AES é o conjunto de *axiomas de estados-sucessores*.
- AP é o conjunto de *axiomas de precondições*.
- AE é o conjunto de *axiomas de exclusão*.
- G_n é o axioma de objetivos.

Um índice i indexa ações e literais a cada instante de tempo usado para se realizar a redução. Para os literais o índice i indica a qual estado do mundo o literal pertence e assume um valor no intervalo [0,n]. Para as ações o índice i indica a qual estado do

mundo a ação é aplicada e assume um valor no intervalo [0,n-1]. Por exemplo, o literal EM(A,i) indica que este literal é verdadeiro no estado s_i.

Seja L o conjunto de literais do problema, as codificações de s_0 e de G são calculadas do seguinte modo:

$$s_0 : \bigwedge \{l_0 \mid l_0 \in s_0\} \wedge \bigwedge \{\neg l_0 \mid l_0 \in \text{L - } s_0\} \quad (2.5)$$

$$G_n : \bigwedge \{l_n \mid l_n \in s_n\} \wedge \bigwedge \{\neg l_n \mid l_n \in \text{L - } s_n\} \quad (2.6)$$

A codificação do estado inicial e objetivos do problema do Mundo das Células é dada como segue:

$$s_0\text{: EM(A,0)} \wedge \neg\text{EM(B,0)} \wedge \neg\text{EM(C,0)}$$

$$G_n\text{: } \neg\text{EM(A,n)} \wedge \neg\text{EM(B,n)} \wedge \text{EM(C,n)}$$

Axiomas de exclusão determinam que, para um instante i, a ocorrência de uma ação elimina a ocorrência das demais ações, ou seja, a ocorrência de ações é um evento mutuamente exclusivo. Para cada ação a e b, sendo a diferente de b, tem-se:

$$\neg a_i \vee \neg b_i \quad (2.7)$$

Por exemplo, o seguinte axioma:

$$\neg\text{MOVER(B,A,}i) \vee \neg\text{MOVER(B,C,}i)$$

Significa que para o instante i o robô só pode se mover da célula B para a célula A ou para a célula C. Para que uma ação ocorra em um instante i é necessário descrever seu

axioma de precondições, por exemplo:

$$\text{MOVER(A,B,}i\text{)} \Rightarrow \text{EM(A,}i\text{)}$$

Para que o robô se mova da célula A para a célula B é precondição que *EM(A,i)* seja verdadeiro no estado s_i. A fórmula (2.3) referente ao axioma de estado-sucessor é neste contexto reescrita da seguinte forma:

$$L(x, i+1) \Leftrightarrow ActionCausesL \vee (L(x, i) \wedge \neg ActionCausesNotL) \quad (2.8)$$

Onde *ActionCausesL* refere-se a disjunção entre as ações que possuem em sua lista de adição (ADD) o literal L e *ActionCausesNotF* refere-se a disjunção entre as ações que possuem em sua lista de deleção (DEL) o literal L. Para o instante $i = 0$ a codificação do literal EM(B,1) segundo a fórmula (2.8) é dado como:

$$\text{EM(B,1)} \Leftrightarrow (\text{MOVER(C,B,0)} \vee \text{MOVER(A,B,0)}) \vee$$

$$(\text{EM(B,0)} \wedge \neg\text{MOVER(B,A,0)} \wedge \neg\text{MOVER(B,C,0)})$$

O *axioma de estado sucessor* indica que um literal l será verdadeiro no intente *i+1* se alguma ação que tenha um efeito positivo sobre l seja aplicada no estado s_i, ou se l já é verdadeiro em s_i e nenhuma ação que tenha efeito negativo sobre l seja aplicada em s_i de modo que l permanecerá verdadeira em *i+1*.

Codificação total do problema do Mundo das Células

Para que este exemplo seja satisfazível são necessárias duas iterações. Com apenas uma iteração o objetivo não é alcançado já que, neste caso, o robô se encontraria sobre a célula B.

Os axiomas que formam a fórmula lógica ω estão explicitamente separados para fa-

cilitar o entendimento sobre a obtenção desta fórmula. A conjunção entre os axiomas está destacada em vermelho. Em negrito estão declarados as proposições assinadas como verdadeiras:

axioma de estado inicial:

$\mathbf{EM(A,0)} \wedge \neg\mathbf{EM(B,0)} \wedge \neg\mathbf{EM(C,0)} \wedge$

axiomas de estados sucessores:

$\mathrm{EM(A,1)} \leftrightarrow (\mathrm{MOVER(B,A,0)}) \vee (\mathbf{EM(A,0)} \wedge \neg\mathrm{MOVER(A,B,0)}) \wedge$

$\mathbf{EM(B,1)} \leftrightarrow (\mathrm{MOVER(C,B,0)} \vee \mathbf{MOVER(A,B,0)}) \vee (\mathrm{EM(B,0)} \wedge$

$\neg\mathbf{MOVER(B,A,0)} \wedge \neg\mathbf{MOVER(B,C,0)}) \wedge$

$\mathrm{EM(C,1)} \leftrightarrow (\mathrm{MOVER(B,C,0)}) \vee (\mathrm{EM(C,0)} \wedge \neg\mathbf{MOVER(C,B,0)}) \wedge$

$\mathrm{EM(A,2)} \leftrightarrow (\mathrm{MOVER(B,A,1)}) \vee (\mathrm{EM(A,1)} \wedge \neg\mathbf{MOVER(A,B,1)}) \wedge$

$\mathrm{EM(B,2)} \leftrightarrow (\mathrm{MOVER(C,B,1)} \vee \mathrm{MOVER(A,B,1)}) \vee (\mathbf{EM(B,1)} \wedge$

$\neg\mathbf{MOVER(B,A,1)} \wedge \neg\mathrm{MOVER(B,C,1)}) \wedge$

$\mathbf{EM(C,2)} \leftrightarrow \mathbf{MOVER(B,C,1)} \vee (\mathrm{EM(C,1)} \wedge \neg\mathbf{MOVER(C,B,1)}) \wedge$

axiomas de precondições:

$\mathbf{MOVER(A,B,0)} \rightarrow \mathbf{EM(A,0)} \wedge$

$\mathrm{MOVER(B,A,0)} \rightarrow \mathrm{EM(B,0)} \wedge$

$\mathrm{MOVER(B,C,0)} \rightarrow \mathrm{EM(B,0)} \wedge$

$\mathrm{MOVER(C,B,0)} \rightarrow \mathrm{EM(C,0)} \wedge$

$\mathrm{MOVER(A,B,1)} \rightarrow \mathrm{EM(A,1)} \wedge$

$\mathrm{MOVER(B,A,1)} \rightarrow \mathbf{EM(B,1)} \wedge$

$\mathbf{MOVER(B,C,1)} \rightarrow \mathbf{EM(B,1)} \wedge$

$\mathrm{MOVER(C,B,1)} \rightarrow \mathrm{EM(C,1)} \wedge$

axiomas de exclusão:

(¬MOVER(A,B,0) ∨ ¬**MOVER(B,A,0)** ∨ ¬**MOVER(B,C,0)** ∨

¬**MOVER(C,B,0)**) ∧

(¬**MOVER(A,B,1)** ∨ ¬**MOVER(B,A,1)** ∨ ¬MOVER(B,C,1) ∨

¬**MOVER(C,B,1)**) ∧

axioma de objetivos:

¬**EM(A,2)** ∧ ¬**EM(B,2)** ∧ **EM(C,2)**

O planejador irá verificar quais ações foram assinadas como verdadeiras para cada instante i afim de extrair o plano. Nesta caso o plano é o conjunto $\{A_0$: MOVER(A,B,0), A_1:MOVER(B,C,1)$\}$.

3 Desenvolvimento de um Sistema de Planejamento Automático

Este capítulo apresenta o desenvolvimento de um sistema de planejador automático baseado na redução do problema de planejamento ao problema SAT. Conforme visto na seção 2.3.6 esta reduçao consiste na transformação da instancia que descreve o domínio do problema a uma fórmula lógica proposicional na FNC. Se a fórmula resultante for satisfazível, tem-se então um plano que resolve a instancia do problema.

A figura 3.1 apresenta um diagrama em blocos do sistema de planejamento automático proposto.

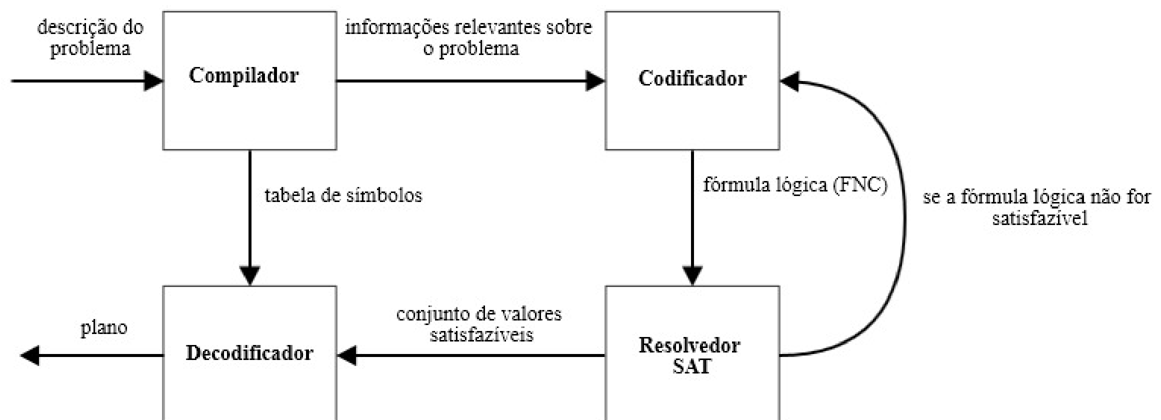

Figura 3.1: Diagrama em blocos do sistema proposto.

O desenvolvimento do sistema consistiu, basicamente, em dois passos: a construção de um Compilador e a construção do Planejador automático que é formado pelos blocos *Codificador*, *Resolvedor SAT* e *Decodificador*. A execução inicia-se pelo bloco *Compilador*

que é responsável por analisar e extrair informações do domínio do problema definido pelo usuário. Estas informações são repassadas ao bloco *Codificador* que codifica as informações recebidas em uma fórmula lógica booleana segundo as regras de tradução da seção 3.2.6. A fórmula lógica é, então, enviada ao bloco *Resolvedor SAT* que, por sua vez, busca por uma assinatura de valores verdade que satisfaça a fórmula lógica. Caso a fórmula lógica seja satisfazível o bloco *Decodificador* a traduz em uma sequência linear de ações, o *plano*.

Nas seções seguintes serão descritos em detalhes cada um dos componentes do sistema.

3.1 O bloco Compilador

O desenvolvimento do bloco *Compilador* se deu através do gerador de analisadores léxicos e sintáticos JavaCC. Ao receber uma especificação de uma gramática, JavaCC a converte em um programa escrito na linguagem Java que sobre um fluxo de caracteres realiza ações léxicas, sinstáticas e semânticas. A seguir estas ações são descritas em maiores detalhes.

3.1.1 Análise Léxica

A análise léxica é responsável por ler o fluxo de entrada de caracteres a fim de identificá-los como símbolos válidos da linguagem. Um símbolo válido pode ser um um identificador de variável, um operador ou uma palavra reservada, por exemplo [57].

Padrões são definidos por meio de expreções regulares, ou seja, mecanismos que identificam cadeias de caracteres de interesse. As seguintes expressões regulares foram definidas para o reconhecimento de palavras reservadas da linguagem e identificadores de literais, ações e argumentos.

$$
\begin{aligned}
\langle Problem\rangle \quad &::= [\text{“}p\text{”}, \text{“}P\text{”}][\text{“}r\text{”}, \text{“}R\text{”}][\text{“}o\text{”}, \text{“}O\text{”}][\text{“}b\text{”}, \text{“}B\text{”}][\text{“}l\text{”}, \text{“}L\text{”}]\\
&\quad [\text{“}e\text{”}, \text{“}E\text{”}][\text{“}m\text{”}, \text{“}M\text{”}]\\
\langle Domain\rangle \quad &::= [\text{“}d\text{”}, \text{“}D\text{”}][\text{“}o\text{”}, \text{“}O\text{”}][\text{“}m\text{”}, \text{“}M\text{”}][\text{“}a\text{”}, \text{“}A\text{”}][\text{“}i\text{”}, \text{“}I\text{”}]\\
&\quad [\text{“}n\text{”}, \text{“}N\text{”}]\\
\langle Literals\rangle \quad &::= [\text{“}l\text{”}, \text{“}L\text{”}][\text{“}i\text{”}, \text{“}I\text{”}][\text{“}t\text{”}, \text{“}T\text{”}][\text{“}e\text{”}, \text{“}E\text{”}][\text{“}r\text{”}, \text{“}R\text{”}]\\
&\quad [\text{“}a\text{”}, \text{“}A\text{”}][\text{“}l\text{”}, \text{“}L\text{”}][\text{“}s\text{”}, \text{“}S\text{”}]\\
\langle Objects\rangle \quad &::= [\text{“}o\text{”}, \text{“}O\text{”}][\text{“}b\text{”}, \text{“}B\text{”}][\text{“}j\text{”}, \text{“}J\text{”}][\text{“}e\text{”}, \text{“}E\text{”}][\text{“}c\text{”}, \text{“}C\text{”}]\\
&\quad [\text{“}t\text{”}, \text{“}T\text{”}][\text{“}s\text{”}, \text{“}S\text{”}]\\
\langle Instance\rangle \quad &::= [\text{“}i\text{”}, \text{“}I\text{”}][\text{“}n\text{”}, \text{“}N\text{”}][\text{“}s\text{”}, \text{“}S\text{”}][\text{“}t\text{”}, \text{“}T\text{”}][\text{“}a\text{”}, \text{“}A\text{”}]\\
&\quad [\text{“}n\text{”}, \text{“}N\text{”}][\text{“}c\text{”}, \text{“}C\text{”}][\text{“}e\text{”}, \text{“}E\text{”}]\\
\langle Initial\rangle \quad &::= [\text{“}i\text{”}, \text{“}I\text{”}][\text{“}n\text{”}, \text{“}N\text{”}][\text{“}i\text{”}, \text{“}I\text{”}][\text{“}t\text{”}, \text{“}T\text{”}][\text{“}i\text{”}, \text{“}I\text{”}]\\
&\quad [\text{“}a\text{”}, \text{“}A\text{”}][\text{“}l\text{”}, \text{“}L\text{”}]\\
\langle Goal\rangle \quad &::= [\text{“}g\text{”}, \text{“}G\text{”}][\text{“}o\text{”}, \text{“}O\text{”}][\text{“}a\text{”}, \text{“}A\text{”}][\text{“}l\text{”}, \text{“}L\text{”}]\\
\langle Actions\rangle \quad &::= [\text{“}a\text{”}, \text{“}A\text{”}][\text{“}c\text{”}, \text{“}C\text{”}][\text{“}t\text{”}, \text{“}T\text{”}][\text{“}i\text{”}, \text{“}I\text{”}][\text{“}o\text{”}, \text{“}O\text{”}]\\
&\quad [\text{“}n\text{”}, \text{“}N\text{”}][\text{“}s\text{”}, \text{“}S\text{”}]\\
\langle Edges\rangle \quad &::= [\text{“}e\text{”}, \text{“}E\text{”}][\text{“}d\text{”}, \text{“}D\text{”}][\text{“}g\text{”}, \text{“}G\text{”}][\text{“}e\text{”}, \text{“}E\text{”}][\text{“}s\text{”}, \text{“}S\text{”}]\\
\langle Pre\rangle \quad &::= [\text{“}p\text{”}, \text{“}P\text{”}][\text{“}r\text{”}, \text{“}R\text{”}][\text{“}e\text{”}, \text{“}E\text{”}]\\
\langle Add\rangle \quad &::= [\text{“}a\text{”}, \text{“}A\text{”}][\text{“}d\text{”}, \text{“}D\text{”}][\text{“}d\text{”}, \text{“}D\text{”}]\\
\langle Del\rangle \quad &::= [\text{“}d\text{”}, \text{“}D\text{”}][\text{“}e\text{”}, \text{“}E\text{”}][\text{“}l\text{”}, \text{“}L\text{”}]\\
\langle Letter\rangle \quad &::= [\text{“}a - \text{“}z\text{”}, \text{“}A - \text{“}Z\text{”}]\\
\langle Digit\rangle \quad &::= [\text{“}0 - \text{“}9\text{”}]\\
\langle Id\rangle \quad &::= \langle Letter\rangle(\langle Letter\rangle|\langle Digit\rangle)*
\end{aligned}
$$

As expressões regulares *Problem*, *Domain*, *Literals*, *Objects*, *Edges*, *Actions*, *Pre*, *Add*, *Del*, *Initial* e *Goal* determinam palavras reservadas. Estas expressões reconhecem quaisquer combinações dos caracteres que lhes foram atribuídos. A expressão regular *Id* faz uso da expressão *Letter* para determinar identificadores.

JavaCC oferece alguns tipos de tratamentos especiais para expressões regulares, um

deles é o *SKIP*. Uma expressão regular declarada como SKIP ao ser reconhecida durante a análise léxica é, simplesmente, ignorada. A seguinte expressão regular é usada para ignorar espaços em branco, quebras de linha e tabulação.

$$\langle SKIP \rangle \quad ::= \quad (\text{``}\backslash n\text{''} \mid \text{``}\backslash r\text{''} \mid \text{``}\backslash t\text{''} \mid \text{`` ''})^{+}$$

Caso a análise léxica finalize com sucesso o fluxo de execução segue para a fase de análise sintática.

3.1.2 Análise Sintática

A análise sintática é responsável por determinar se o fluxo de tokens provenientes do analisador léxico forma uma sentença válida da linguagem segundo a gramática. Uma gramática é um dispositivo formal composto por um conjunto de regras que derivam tais sentenças.

As técnicas de análise sintática consistem em construir uma árvore que represente a derivação da sentença de entrada, de acordo com as produções da gramática. As técnicas de derivação são divididas em: *top-down*, onde a árvore de derivação é construída da raíz em direção as folhas, ou *bottom-up*, onde a árvore de derivação é construída a partir das folhas em direção a raíz [57].

O analisador sintático gerado pela JavaCC pertence a família dos *analisadores sintáticos descendentes recursivos*. Estes analisadores implementam o método de análise *top-down* e são aplicáveis somente a gramáticas k fatoradas, ou seja, gramáticas livres de recursão à esquerda e não-ambíguas. A variável k indica o número de símbolos necessários para que as produções usadas na derivação sejam escolhidas deterministicamente.

A seguinte gramática especifica a linguagem na qual um problema de planejamento deve ser escrito para que ele seja analisado pelo sistema proposto. Esta gramática foi baseada na linguagem STRIPS.

Símbolos terminais estão destacados em negrito quando são derivados por alguma ex-

pressão regular citada anteriormente ou estão entre aspas quando o símbolo esperado deve ser exatamente o valor que está entre as aspas. Símbolos não-terminais estão representado entre $\langle$ e $\rangle$.

$\langle Start \rangle$ ::= $\langle Domain \rangle$ $\langle Instance \rangle$

$\langle Domain \rangle$ ::= **Problem Domain Id**"{" $\langle Objects \rangle$ $\langle Literals \rangle$
$\langle Actions \rangle$ "}"

$\langle Objects \rangle$ ::= **Objects** "{" $\langle Object \rangle^{+}$ "}"

$\langle Object \rangle$ ::= **Id** "(" **Id** ")"

$\langle Literals \rangle$ ::= **Literals** "{" $\langle Literal \rangle^{+}$ "}"

$\langle Literal \rangle$ ::= **Id** "(" $\langle Argument \rangle$ ")"

$\langle Actions \rangle$ ::= **Actions** "{" $\langle Action \rangle^{+}$ "}"

$\langle Action \rangle$ ::= **Id** "(" "(" $\langle Argument \rangle$ ")" "{" *Pre* " : "$\langle List \rangle$
Add " : "$\langle List \rangle$ *Del* " : "$\langle List \rangle$ "}" ")"

$\langle Argument \rangle$ ::= **Id** " : " **Id** ("," $\langle Argument \rangle$)*

$\langle List \rangle$ ::= $\langle Literal \rangle$ (" $\wedge$ " $\langle Literal \rangle$)*

$\langle Instance \rangle$ ::= **Problem Instance**"{"$\langle Objects \rangle$ $\langle Initial \rangle$
$\langle Goal \rangle$ $\langle Edges \rangle^{*}$"}"

$\langle Initial \rangle$ ::= **Initial** $\langle Statement \rangle$

$\langle Goal \rangle$ ::= **Goal** $\langle Statement \rangle$

$\langle Statement \rangle$ ::= ($(\neg)^{?}$ $\langle Literal \rangle$)$^{+}$

$\langle Edges \rangle$ ::= **Edges** $\langle Edge \rangle$

$\langle Edge \rangle$ ::= "(" $< Object >$ "," $< Object >$ ")"(" $\wedge$ "$\langle Edge \rangle$)*

A gramática prevê que um problema seja decomposto em duas partes: a primeira, refere-se a descrição do domínio, e a segunda refere-se a descrição de uma instancia do problema.

O domínio do problema consiste em descrever quais tipos de *objetos*, *literais* e *ações* compõem o mundo. Um objeto representa uma entidade do mundo e é constituído de um

tipo e um nome. Um literal é uma variável que denota algum significado no contexto do problema e é constituída de um nome e uma lista de argumentos. Uma ação representa um acontecimento sobre o mundo e é constituída de uma lista de argumentos, pré-condições, adições e eliminações.

A instancia do problema consiste em *instanciar objetos*, declarar o *estado inicial*, *objetivos* e *possíveis restrições*. Uma restrição é um par ordenado formado por objetos, onde cada elemento do par é interpretado como um vértice de um *grafo de restrições*. O grafo de restrições determina que ações só são possíveis de serem aplicadas sobre objetos cuja relação está explicitamente representada no grafo.

O problema exemplo da seção 2.3.5 segundo esta gramática é descrito da seguinte forma:

$$
\begin{array}{ll}
Problem & domain\ MundoDasCelulas\ \{ \\
Objects & \{Celula(x)\} \\
Literals & \{Em(a : Celula)\} \\
Actions & \{Mover((i : Celula, j : Celula)\ Pre:\ em(i) \\
 & Add:\ em(j)\ Del:\ em(i))\}\ \} \\
Problem & Instance\ \{ \\
Objects & \{Celula(A)\ Celula(B)\ Celula(C)\} \\
Initial & \{Em(celula(A)) \wedge \neg Celula(B) \wedge \neg Celula(C)\} \\
Goal & \{\neg Celula(A) \wedge \neg Celula(B) \wedge Celula(C)\} \\
Edges & \{(Celula(A), Celula(B)) \wedge (Celula(B), Celula(C))\}\}
\end{array}
$$

No domínio do problema foi declarado um objeto do tipo *Celula*, um literal *Em* e uma ação *Mover*. Na instancia do problema foram instanciadas três células (A,B,C), declarou-se que há uma relação entre a Celula(A) e Celula(B) e um relação entre a Celula(B) e Celula(C). Deste modo evita-se que o planejador realize alguma ação proibitiva, neste caso mover da Celula(A) diretamente para a Celula(C).

3.1.3 Análise Semântica

A análise semântica é responsável por verificar possíveis erros semânticos na descrição do problema. O método de análise semântica implementado consiste na técnica de tradução dirigida pela sintaxe. Esta técnica determina que ações semânticas sejam associadas às regras da gramática. A medida que as ações semânticas são aplicadas, os objetos, literais e ações declarados são armazenados em uma tabela se símbolos. Esta tabela guarda todas as informações relevantes para que o problema descrito pelo usuário seja reduzido a um problema SAT.

As ações semânticas implementadas possuem responsabilidades específicas no domínio e na descrição da instancia do problema. No domínio do problema, as ações semânticas são responsáveis por determinar se:

- Objetos constituintes de um argumento possuem tipos válidos.
- Listas de literais são formadas por literais existentes.
- Identificadores de novos objetos, ações e literais são diferentes dos já declarados.

Na instancia do problema, as ações semânticas são responsáveis por determinar se:

- Objetos, ações e literais instanciados são válidos, ou seja, foram previamente declarados no domínio do problema.
- Se existe duplicação de objetos, ações ou literais.
- Argumentos de ações e literais possuem número correto de objetos.
- Listas de literais das ações possuem número correto de literais ou a ordem dos literais nas listas está de acordo com a ordem declarada nas listas de literais do domínio.
- Objetos usados na criação de pares ordenados foram previamente instanciados.

Caso a análise semântica verifique que alguma destas condições não seja respeitada, um erro é gerado, apresentado ao usuário e a análise é interrompida. Caso contrário, a análise semântica é concluída com sucesso e a execução do sistema segue para o bloco *Codificador* que recebe as informações obtidas nesta fase de compilação.

3.2 O bloco Codificador

O bloco *Codificador* é formado por um conjunto de blocos ilustrados pelo diagrama da figura 3.2.

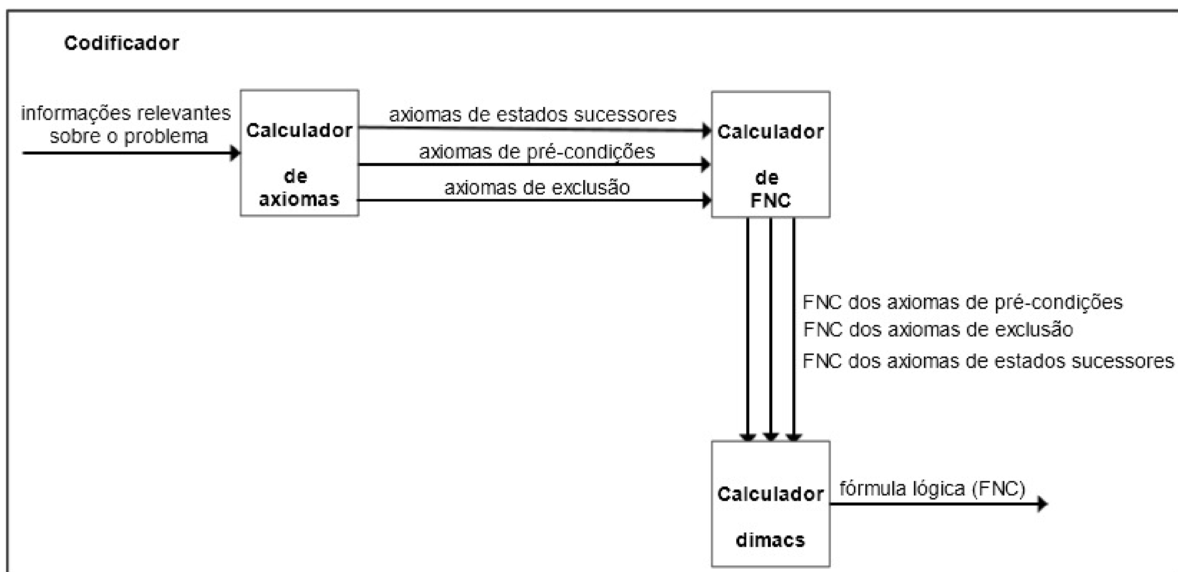

Figura 3.2: Conjunto de blocos que formam o bloco Codificador.

O bloco *Calculador de axiomas* implementa métodos que calculam os axiomas necessárias para a redução do problema do planejamento ao problema SAT. A redução se resume a aplicar os métodos *getAxiomasDeEstadosSucessores*, *getAxiomasDePreCondicoes* e *getAxiomaDeExclusao*. Estes métodos refletem os conceitos descritos na seção 2.3.6.

Cada método repassa o axioma que calculou ao bloco *Calculador de FNC*. Este bloco transforma os axiomas que recebe em uma nova fórmula equivalente na forma normal con-

juntiva. Para isso, foi implementado o método *getFormulaFNC* que realiza a eliminação de conectivos bicondicional e condicional, aplicação da dupla negação e distribuição da disjunção sobre a conjunção.

As novas fórmulas são, então, reescritas de acordo com o formato *dimacs* (ver seção 2.2.6) no bloco *Calculador dimacs* através do método *getFormulaDimacs*. Foi adotada a seguinte convenção: ações são iniciadas com o número 2 e literais com o número 1. Tal convenção permite identificar facilmente as ações durante o processo de decodificação.

Cada termo é salvo em uma tabela hash cuja chave é o número a ele associado. Dessa forma, para que a fórmula seja decodificada basta apenas ler os números contidos na fórmula e buscar os termos na tabela hash usando-os como chave de busca. Sempre que houver a necessidade de se realizar novas iterações a fim de que a fórmula final seja satisfazível, novos números serão associados aos literais e ações representando-os em cada instante de tempo *t'*.

A fórmula deve estar descrita no formato dimacs para que o algoritmo SAT do bloco *Resolvedor SAT* possa determinar a satisfatibilidade da fórmula reduzida, pois algoritmos deste domínio (em sua maioria) aceitam apenas entradas descritas neste formato. Todas as fórmulas dimacs calculadas são combinadas em uma única fórmula concatenando-as umas as outras. Esta última fórmula representa o problema do planejamento reduzido a SAT. A fórmula reduzida é passada ao bloco *Resolvedor SAT* que testará a sua satisfatibilidade.

Caso não seja encontrada uma assinatura de valores verdade que satisfaça a fórmula, todo o processo de redução se repete para um novo tempo *t'*. A fórmula resultante do tempo *t'* é concatenada a fórmula resultante do tempo t e repassada ao bloco *Resolvedor SAT*. Este processo se repete até que o SAT Solver responda com satisfazível ou enquanto o valor máximo de repetições estipulado pelo usuário não seja alcançado.

3.3 O bloco Resolvedor SAT

O bloco Resolvedor SAT é responsável por encontrar uma assinatura de valores verdade que satisfaça a fórmula lógica que recebe. Para isso, o algoritmo usado é o MiniSat cujo funcionamento está descrito na seção 2.2.6.

Como o algoritmo MiniSat está escrito na linguagem C dois passos foram necessários para que o sistema e o MiniSat pudessem se comunicar. O primeiro passo consistiu em compilar o algoritmo para que uma versão executável fosse gerada, o *MiniSat.exe*. E a segunda parte consistiu em criar um processo adicional, com auxílio da classe *Runtime* do Java, para que o arquivo *MiniSat.exe* fosse executado.

Assim que o processo retorna, o sistema verifica a sua resposta e, então, decide se o fluxo de execução retorna ao bloco *Codificador* ou segue para o bloco *Decodificador* enviando-lhe o conjunto de valores verdade que satisfaz a fórmula lógica.

3.4 O bloco Decodificador

O bloco *Decodificador* é responsável por traduzir o conjunto de assinaturas de valores verdade que recebe em um plano. A tradução é dada em dois passos. O primeiro passo consiste em ler todas as proposições buscando por aquelas que são assinadas como verdadeiras e iniciadas com o número 2. Quando uma proposição que satisfaz esses requisitos é encontrada ela é usada como chave de busca na tabela hash. O valor retornado da tabela hash é adicionado em uma lista de ações.

O segundo passo consiste em ordenar a lista de ações em ordem crescente de tempo de suas criações. Para isso, foi criado o método *getAcoesOrdenadas* que implementa os conceitos do algoritmo de ordenação *QuickSort* [58]. O retorno deste método, o *plano*, é a solução do problema de planejamento. A solução é enviada ao controle principal que a apresenta ao usuário e finaliza o fluxo de execução do sistema.

4 Resultados

Este capítulo apresenta os testes realizados sobre o sistema desenvolvido. Dois problemas de domínios diferentes foram usados para a aplicação dos testes. O domínio do primeiro problema é referente ao Mundo das Células apresentado na seção 2.3.5, e o domínio do segundo problema é referente ao Mundo dos Blocos. Os resultados obtidos são apresentados e discutidos nas seções seguintes.

Os testes apresentados neste capítulo foram realizados em um computador com processador Intel(R) Pentium(R) P6100 de 2.0 GHz, 2 GB de memória RAM e sistema operacional Windows 7 de 32 bits.

4.1 Testes sobre o Mundo das Células

Sobre o domínio do Mundo das Células foram realizados no total 10 testes que variaram o número de células do problema. A tabela 4.1 mostra os resultados destes testes e está dividida em duas colunas principais: a coluna *configurações* que indica o número de células e objetivo de um problema e a coluna *resultados* que indica o número de proposições e cláusulas do problema e o tempo que este levou para ser resolvido.

O tempo de resolução de um problema é a soma do tempo que este leva para ser reduzido, do tempo que o SAT solver leva para decidir a satisfatibilidade do problema e do tempo em que o conjunto de valores verdade que satisfaz o problema leva para ser decodificado em um plano. Todas as instancias tem como estado inicial: *Em(A)*.

Os resultados mostram que a medida que células são inseridas o número de pro-

Configurações		Resultados		
nº de células	objetivo	nº proposições	nº de cláusulas	tempo
2	Em(B)	6	14	115 ms
3	Em(C)	17	40	195 ms
4	Em(D)	34	80	366 ms
5	Em(E)	57	134	457 ms
10	Em(J)	262	614	1.78 s
25	Em(Z2)	1475	3522	1 min 19 s
50	Em(Z22)	2600	12.650	19 min 53 s
75	Em(Z47)	5.775	28.359	5 h 20 min
100	Em(Z69)	10.200	50.300	2 dias 1 h

Tabela 4.1: Resultado dos testes realizados sobre o problema do Mundo das Células.

posições e cláusulas cresce exponencialmente. Observou-se que os tempos de respostas para as primeiras instancias diferenciavam-se em pequenos intervalos de milessegundos, como mostram os 5 primeiros testes, onde os valores variam de 115 ms a 1.78 s. Os tempos de respostas começaram a apresentar um comportamento exponencial a partir de 50 células, que é de 5 h e 20 min. O tempo máximo de resposta obtido é de 2 dias e 1h para a instancia com 100 células.

4.1.1 Testes sobre uma extensão do Mundo das Células

O problema do Mundo das Células foi estendido modificando-se as posições das células que, nesta extensão, estão posicionadas em formato de um Grid. Deste modo, as instancias possuem grafos de restrições mais elaborados. A figura 4.1 ilustra o grafo de restrições de dois problemas distintos, onde os vértices estão nomeados com o nome das células que as compõem. Os dois problemas tem como estado inicial: *Em(A)*.

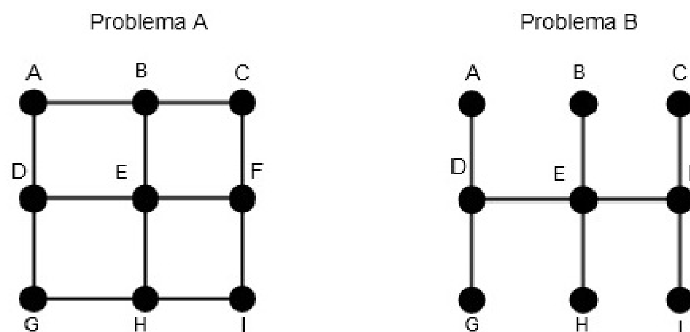

Figura 4.1: Grafo de restrições para os problemas A e B.

A tabela 4.2 mostra os resultados obtidos para estes dois testes.

Configurações		Resultados		
Problema	objetivo	nº proposições	nº de cláusulas	tempo
A	Em(I)	108	238	2 s 13 ms
B	Em(I)	87	200	2 s 52 ms

Tabela 4.2: Resultado dos testes realizados sobre o problema entendido do Mundo das Células.

A solução para o problema A apresentada pelo planejador se deu pela seguinte sequência de ações:

- instante 0: mover(A,B)
- instante 1: mover(B,C)
- instante 2: mover(C,F)
- instante 3: mover(F,I)

De fato há mais de um plano que resolve este problema, porém o algoritmo MiniSat pára a sua busca ao encontrar o primeiro conjunto σ de valores verdade que satisfaz a fórmula lógica.

Para a instancia B, como é possível ver em seu grafo, existe apenas um plano:

- instante 0: mover(A,D)
- instante 1: mover(D,E)
- instante 2: mover(E,F)
- instante 3: mover(F,I)

4.2 Testes sobre o Mundo dos Blocos

O problema do Mundo dos Blocos consiste em um robô e um conjunto de blocos localizados sobre uma superfície plana. Estão definidos para este domínio os literais *sobre(x,y)* o qual indica que um bloco x está sobre o bloco y, *livre(x)* indica que não existe outro bloco sobre x e *mesa(x)* indica que x está em cima da mesa - a superfície plana. Ao robô são permitidas as ações *empilhar* e *desempilhar* blocos, que são descritas da seguinte forma:

- EMPILHAR((x,y),
  PRE: livre(x) ∧ livre(y) ∧ mesa(x)
  ADD: sobre(x,y)
  DEL: livre(y) ∧ mesa(x))

- DESEMPILHAR((x,y),
  PRE: livre(x) ∧ sobre(x,y)
  ADD: mesa(x) ∧ livre(y)
  DEL: sobre(x,y))

A figura 4.2 ilustra o estado inicial e final do problema testado.

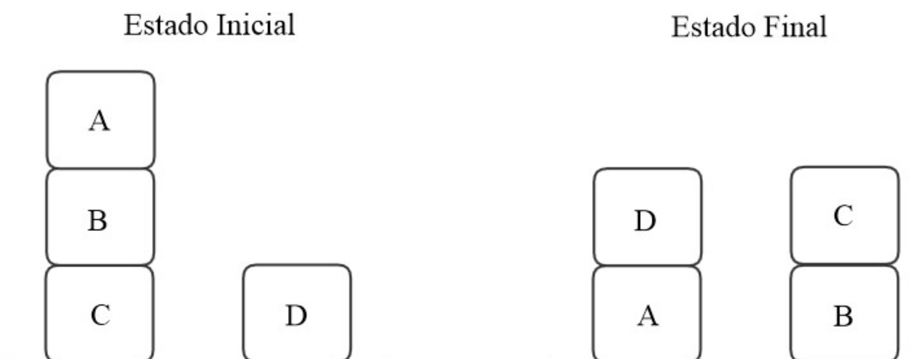

Figura 4.2: Diagrama em blocos do sistema proposto.

A tabela 4.3 mostra o resultado para este teste.

Configurações	Resultados		
nº blocos	nº proposições	nº de cláusulas	tempo
4	196	651	2 s 35 ms

Tabela 4.3: Resultado do teste realizado sobre o Mundo dos Blocos.

Em relação aos outros testes para uma mesma quantidade de objetos, o problema do Mundo dos Blocos apresenta uma fórmula lógica muito maior. Isso se deve ao fato de que para este domínio o número de ações e literais é superior ao domínio do Mundo das Células.

5 *Conclusão*

O presente trabalho apresentou um pesquisa sobre a área de Planejamento Automático tendo como foco a técnica que trata problemas de planejamento como um problema SAT. Como visto no Capítulo 2, o problema de planejamento automático é o problema de se determinar computacionalmente um conjunto de ações que quando executadas em uma sequência linear satisfazem um objetivo.

Desde o início dos anos setenta, pesquisadores da comunidade de IA tem estudado métodos que viabilizassem o desenvolvimento de planejadores automáticos mais eficientes. Porém, a área de Planejamento sempre careceu de boas propostas, principalmente por ser um problema NP-Completo, pela falta de algoritmos de busca robustos e pela dificuldade de se definir uma linguagem expressiva o suficiente para a representação de conhecimento.

No ano de 96 os pesquisadores Kautz e Selman apresentaram a técnica que busca por planos através da redução de instancias de planejamento ao problema da satisfatibilidade booleana - SATPLAN. Esta técnica se mostrou muito eficiente, pois, os resolvedores SAT usados naquela época apresentavam soluções em um tempo menor do que os métodos de busca tradicionais. Neste contexto, este trabalho apresentou como proposta o desenvolvimento de um planejador automático para solucionar problemas de planejamento através da técnica SATPLAN.

O sistema foi dividido em dois grandes blocos: o bloco *Compilador* responsável por receber o problema descrito na linguagem da seção 3.1.2, compilar e extrair informações do problema, e o bloco *Planejador* que busca por um plano que solucione o problema de entrada reduzindo-o ao problema SAT. Para verificar o desempenho do sistema, testes

foram realizados e os resultados obtidos foram apresentados e discutidos.

Foram realizados 9 testes sobre o problema do Mundo das Células que variaram o número de células do problema. Dentre os resultados obtidos, a instancia com 2 células apresentou como tempo de resposta apenas 115 ms, ao passo que a instancia com 100 células apresentou como tempo de resposta 2 dias e 1 h. Este conjunto de testes mostra a configuração real de um problema NP-Completo: a medida que o problema cresce o tempo de resposta de um algoritmo que o resolva cresce exponencialmente.

Este trabalho não apresenta inovação científica na área de Planejamento Automático, apenas realiza uma revisão bibliográfica sobre esta área. A compreensão profunda do problema e o estudo sobre uma técnica de solução (SAT) permitiu a aquisição do conhecimento necessário para que estudos e pesquisas mais avançados possam ser feitos.

Como possíveis trabalhos futuros, pode-se apontar:

- Estudar outras técnicas de Planejamento Automático a fim de compreender diferentes abordagens para a busca de planos, detectar as capacidades e limitações destas técnicas de modo a utilizar estas abordagens para a construção de um novo planejador. Entre estes modelos encontram-se o planejamento hierárquico, não-linear, probabilístico, temporal entre outros.

- Estudar técnicas de escalonamento para problemas de planejamento onde o consumo de recursos é um fator crítico. Estas técnicas vizam a busca por um plano que gerencie e aloque recursos ao longo do tempo de modo que ações sejam realizadas.

- Implementar o modelo matemático proposto em [53] de modo que o valor de T_{max} seja estimado com maior precisão. Assim, evitar-se-á que o valor máximo de iterações que o planejador SAT deve efetuar a fim de testar a satisfatibilidade da fórmula lógica reduzida seja de responsabilidade do usuário.

- Extender a gramática da seção 3.1.2 de modo a suportar o modelo de linguagem PDDL. Para isso, será estudado as diferentes versões desta linguagem (atualmente

PDDL encontra-se na versão 3.1), suas variações como PDDL+, NDDL, entre outras.

Por fim, conclui-se este trabalho com o conhecimento de que os trabalhos futuros provêm amplos desafios a serem enfrentados. Espera-se que a revisão bibliográfica sobre planejamento SAT, aqui apresentada, possa ajudar outros trabalhos relacionados a esta técnica a prosseguir com suas pesquisas, já que a literatura pouco consta de forma concreta o processo de redução de problemas de planejamento ao problema SAT.

Referências

[1] GHALLAB, M.; NAU, D.; TRAVERSO, P. *Automated Planning: Theory and Practice*. [S.l.]: Elsevier/Morgan Kaufmann Publishers, 2004. (The Morgan Kaufmann Series in Artificial Intelligence Series). ISBN 9781558608566.

[2] KAUTZ, H.; SELMAN, B. Planning as satisfiability. In: *IN ECAI-92*. [S.l.]: Wiley, 1992. p. 359–363.

[3] SELMAN, B.; LEVESQUE, H.; MITCHELL, D. A new method for solving hard satisfiability problems. In: *AAAI*. [S.l.: s.n.], 1992. p. 440–446.

[4] SELMAN, B.; KAUTZ, H.; COHEN, B. Local search strategies for satisfiability testing. In: *DIMACS: Series in Discrete Mathematics and theoretical Computer Science*. [S.l.: s.n.], 1995. p. 521–532.

[5] HICKMOTT, S. et al. *Planning via Petri net unfolding*. [S.l.], 2006.

[6] RUSSELL, S.; NORVIG, P. *Inteligência artificial*. [S.l.]: Campus, 2004. ISBN 9788535211771.

[7] BLUM, A. L.; FURST, M. L. Fast planning through planning graph analysis. *ARTIFICIAL INTELLIGENCE*, v. 90, n. 1, p. 1636–1642, 1995.

[8] ALLEN, J. F. *Planning as Temporal Reasoning*. 1991.

[9] EDELKAMP, S. International planning competition. 2004.

[10] MCDERMOTT, D. The 1998 ai planning systems competition. *AI Magazine*, 2000.

[11] BARTAK, R.; MCCLUSKEY, T. The first competition on knowledge engineering for planning and scheduling. *AI Magazine*, 2006.

[12] ARISTóTELES. *Organon V: Tópicos*. 1. ed. [S.l.]: Guimarães Editores, 1987. ISBN 9789726652427.

[13] CHAGAS, E. M. P. de F. Apresentando alguns aspectos históricos do desenvolvimento da lógica clássica, ciência das idéias e dos processos da mente. 2004.

[14] D'OTTAVIANO, H. d. A. F. Ítala M. L. Sobre a história da lógica, a lógica clássica e o surgimento das lógicas não-clássicas. 2003.

[15] LEIBNIZ, G. W. *DUTENS: Gothofridi Guillemi Leibnitii opera omnia*. [S.l.: s.n.], 1768.

[16] BURRIS, S.; SANKPPANAVAR, H. P. *A course in universal algebra*. 1. ed. [S.l.]: Springer, 1981.

[17] BOOLE, G. *The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning*. [S.l.]: Cambridge University Press, 2009. (Cambridge Library Collection - Mathematics). ISBN 9781108001014.

[18] CAPUANO, F.; IDOETA, I. *Elementos de eletrônica digital*. 39. ed. [S.l.]: Livros Erica, 2007. ISBN 9788571940192.

[19] BITTENCOURT, G. *Inteligência artificial: ferramentas e teorias*. 3. ed. [S.l.]: Editora da UFSC, 2006. (Série didática). ISBN 9788532801388.

[20] RICH, E.; KNIGHT, K. *Inteligencia Artificial*. 2. ed. [S.l.]: MARKON Books, 1994. ISBN 9788448118587.

[21] LEZZI, G. *Fundamentos de Matemática Elementar*. 7. ed. [S.l.]: Atual Editora, 1996.

[22] GALLIER, J. H. *Logic For Computer Science Foundations of Automatic Theorem Proving*. [S.l.: s.n.], 2003.

[23] CONIGLIO, M. E. Um curso de teoria de modelos. 1999.

[24] FITTING, M. *First Logic and Automated Theorem Proving*. [S.l.]: Springer, 1990.

[25] SIPSER, M. *Introdução à teoria da computação*. [S.l.]: Thomson Learning, 2007. ISBN 9788522104994.

[26] FURTADO, O. J. V. *Linguagens Formais e Compiladores*. [S.l.: s.n.].

[27] HODGES, A.; HOFSTADTER, D. *Alan Turing: The Enigma*. [S.l.]: Princeton University Press, 2012. ISBN 9780691155647.

[28] MENEZES, P. F. B. *Linguagens Formais e Autômatos: Volume 3 da Série Livros Didáticos Informática UFRGS*. 4. ed. [S.l.]: Sagra Luzzato, 2002. ISBN 9788577807994.

[29] LEWIS, H.; PAPADIMITRIOU, C. *Elementos de teoria da computação*. 2. ed. [S.l.]: Bookman, 2004. ISBN 9788573075342.

[30] COMPLEXIDADE de Algoritmos: Série Livros Didáticos Informática UFRGS - Vol. 13. [S.l.: s.n.].

[31] GAREY, M.; JOHNSON, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. [S.l.]: W. H. Freeman, 1979. (A Series of Books in the Mathematical Sciences). ISBN 0716710455.

[32] LEEUWEN, J. *Handbook of Theoretical Computer Science: Algorithms and complexity. Volume A*. [S.l.]: Elsevier Science & Technology, 1990. (Algorithms and Complexity). ISBN 9780444880710.

[33] CREIGNOU, N. *The Class of Problems that are Linearly Equivalent to Satisfiability or a Uniform Method for Proving NP-Completeness*. 1995.

[34] DAVIS, M.; LOGEMANN, G.; LOVELAND, D. *A MacHine Program for Theorem-Proving*. [S.l.]: BiblioBazaar, 2011. ISBN 9781179051789.

[35] EEN, N.; SÖRENSSON, N. *A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition*. 2005.

[36] DAVIS, M.; PUTNAM, H. A computing procedure for quantification theory. *J. ACM*, ACM, New York, NY, USA, v. 7, n. 3, p. 201–215, jul. 1960.

[37] ERNST, G. *GPS: A Case Study in Generality and Problem Solving*. [S.l.]: Academic Press, 1979.

[38] GREEN, C. Application of theorem proving to problem solving. In: . [S.l.]: Morgan Kaufmann, 1969. p. 219–239.

[39] KOWALSKI, R. A.; SERGOT, M. A. A logic-based calculus of events. 1969.

[40] FIKES, R.; NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, v. 2, n. 3/4, p. 189–208, 1971.

[41] KNOBLOCK, C. A. An analysis of abstrips. In: *In Proc. 1st Intl. Conf. AI Planning Systems*. [S.l.]: Morgan Kaufmann, 1992. p. 126–135.

[42] SACERDOTI, E. D. The nonlinear nature of plans. In: *Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1975. (IJCAI'75), p. 206–214.

[43] ERNST, M. D.; MILLSTEIN, T. D.; WELD, D. S. Automatic sat-compilation of planning problems. In: *IJCAI-97*. [S.l.]: Morgan Kaufmann, 1997. p. 1169–1176.

[44] KAUTZ, H.; SELMAN, B. Blackbox: A new approach to the application of theorem proving to problem solving. In: . [S.l.: s.n.], 1998. p. 58–60.

[45] BONET, B.; GEFFNER, H. Planning as heuristic search: New results. In: *IN PROCEEDINGS OF ECP-99*. [S.l.]: Springer, 1999. p. 360–372.

[46] MCDERMOTT, D. et al. Pddl - the planning domain definition language. n. TR-98-003, 1998.

[47] HOFFMANN, J.; NEBEL, B. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, v. 14, p. 2001, 2001.

[48] KAUTZ, H. et al. Satplan: Planning as satisfability. 2004.

[49] BYLANDER, T. The computational complexity of propositional strips planning. *Artificial Intelligence*, v. 69, p. 165–204, 1994.

[50] BRACHMAN, R.; LEVESQUE, H. *Knowledge Representation and Reasoning*. [S.l.]: Elsevier/Morgan Kaufmann Publishers, 2004. (Morgan Kaufmann). ISBN 9781558609327.

[51] KAUTZ, H.; SELMAN, B. Planning as satisfability. 1992.

[52] KAUTZ, H.; MCALLESTER, D.; SELMAN, B. Encoding plans in propositional logic. In: . [S.l.]: Morgan Kaufmann, 1996. p. 374–384.

[53] RINTANEN, J. *Introduction to Automated Planning*. [S.l.: s.n.], 2006.

[54] LIU, D. H. *A Survey of Planning in Intelligent Agents: From Externally Motivated to Internally Motivated Systems*. [S.l.: s.n.], 2008.

[55] WANG, Y. Compilation of planning to sat.

[56] NAREYEK, E. et al. Constraints and ai planning. *IEEE Intelligent Systems*, v. 20, p. 72, 2005.

[57] AHO, A.; ULLMAN, J.; SETHI, R. *Compiladores: Princípios, técnicas e ferramentas*. [S.l.]: LTC. ISBN 9788521610571.

[58] AZEREDO, P. A. *Métodos de Classificação de Dados e Análise de suas Complexidades*. [S.l.]: Editora Campus, 1995. ISBN 8535200045.