Paulo Eduardo Rauber

Análise da Solução do Problema do Caminho Hamiltoniano Através de Redução para Problema da Satisfazibilidade Booleana

Florianópolis - SC, Brasil

17 de outubro 2011

Paulo Eduardo Rauber

Análise da Solução do Problema do Caminho Hamiltoniano Através de Redução para Problema da Satisfazibilidade Booleana

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

Orientadora:
Jerusa Marchi

Departamento de Informática e Estatística
Centro Tecnológico
Universidade Federal De Santa Catarina

Florianópolis - SC, Brasil

17 de outubro 2011

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

Profa. Dra. Jerusa Marchi
Orientadora

Prof. Dr. Eduardo Camponogara
Universidade Federal de Santa Catarina

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

Prof. Dr. Rosvelter Coelho da Costa
Universidade Federal de Santa Catarina

Agradecimentos

Gostaria de agradecer à minha mãe, Elsi Rauber. Sem seu apoio, a realização deste trabalho seria impossível.

Também agradeço às pessoas que me receberam em sua casa como um familiar durante esta graduação: Maria, Noivete e José Robaert.

Não menos importantes, agradeço aos amigos e colegas pelas experiências compartilhadas durante este período.

Agradeço à Jerusa Marchi, orientadora deste trabalho, pelo seu apoio, dedicação e colaboração.

Agradeço também ao Prof. Ricardo Silveira pela orientação durante a fase de projeto e aos demais membros da banca pelo interesse neste trabalho.

Resumo

O problema do caminho hamiltoniano e o problema da satisfazibilidade booleana são problemas NP-completos clássicos. Tais problemas são de grande importância para a teoria da computação.

Este trabalho tem como objetivo investigar a solução do problema do caminho hamiltoniano através de redução para o problema da satisfazibilidade booleana.

As reduções apresentadas no trabalho foram implementadas para que as fórmulas resultantes fossem submetidas a aplicações de determinação de satisfazibilidade. Essas aplicações e os algoritmos em que se baseiam também são detalhadas no texto.

O desempenho das reduções e das aplicações é analisado através de testes com grafos gerados aleatoriamente e com grafos do problema do passeio do cavalo. O desempenho de um método direto de determinação de caminhos hamiltonianos também é comparado com os resultados obtidos. Essa comparação demonstra que, apesar de interessante do ponto de vista teórico, a técnica implementada é inferior em desempenho a métodos diretos de determinação de caminhos hamiltonianos.

Abstract

The hamiltonian path problem and the boolean satisfiability problem are classic NP-complete problems. Such problems are very important to the theory of computation.

This work intends to investigate the solution of the hamiltonian path problem through reduction to the boolean satisfiability problem.

The reductions presented in this text were implemented so that the resulting formulae could be solved by boolean satisfiability applications. These applications and the algorithms in which they are based are also detailed in this text.

The performance of both reductions and applications is analysed through tests with random graphs and knight's tour graphs. The performance of a direct method to find hamiltonian paths is also compared with the results obtained. This comparison shows that the technique implemented, although theoretically interesting, is inferior in performance to finding hamiltonian paths through direct methods.

Sumário

1 Introdução

Um grafo é uma estrutura matemática que consiste em um conjunto de vértices conectados por arestas [1]. A importância do estudo de grafos reside no fato de tais estruturas serem muito utilizadas na representação de problemas tanto nas áreas de computação e engenharia quanto em outras áreas de conhecimento [1].

Há diversos problemas clássicos associados à teoria dos grafos. Dentre eles está o problema de determinar se existe um caminho que passa exatamente uma vez por cada vértice de um grafo. Esse problema é chamado de problema do caminho hamiltoniano [1].

O problema do *circuito* hamiltoniano foi provado NP-completo, junto com 21 outros, por Richard Karp [3]. Esse problema é uma pequena variação do problema do caminho hamiltoniano, que também é NP-completo [3]. Um problema é dito NP-completo quando está em NP e todos os problemas em NP podem ser reduzidos a ele em tempo polinomial [3].

Os problemas NP-completos são especialmente importantes para a teoria da computação. Isso porque vários problemas de importância prática são NP-completos e não é sabido se eles podem ser resolvidos em tempo polinomial em função do tamanho da entrada. Uma solução em tempo polinomial para um problema NP-completo resolveria um dos maiores problemas abertos da teoria da computação, pois P seria igual a NP. [3, 4]

O problema da satisfazibilidade booleana foi o primeiro problema a ser provado NP-completo [3]. Esse problema consiste em determinar se existe alguma atribuição de valores verdade às variáveis de uma fórmula booleana que a torne verdadeira [3]. As aplicações desse problema são variadas: prova automática de teoremas, verificação de hardware, agendamento e vários problemas combinatórios [5, 6]. A importância teórica e prática desse problema faz com que ele seja uma área de pesquisa bastante ativa [5].

[1] O termo hamiltoniano é uma referência ao matemático William Hamilton. Em 1859, Hamilton inventou um jogo equivalente a encontrar um circuito hamiltoniano em um grafo determinado pelos vértices e arestas de um dodecaedro [2].

No contexto de algoritmos, uma redução é um algoritmo que transforma um problema em outro de forma que uma solução para o segundo possa ser usada para resolver o primeiro [3]. O problema do planejamento, isto é, a descoberta de uma cadeia de ações para alcançar um objetivo, encontrou um bom desempenho quando reduzido ao problema de satisfazibilidade booleana [6]. Isso é devido principalmente ao desempenho de algoritmos de busca estocástica como o WalkSAT [6].

Inspirado por essa redução de sucesso, o objetivo deste trabalho é expor uma investigação sobre o desempenho da solução do problema do caminho hamiltoniano através de sua redução para o problema de satisfazibilidade booleana. Essa investigação foi feita através da implementação das reduções encontradas na literatura, aplicação dessas reduções em tipos significativos de grafos e solução das fórmulas resultantes através de aplicações de satisfazibilidade bem estabelecidas.

O texto está organizado da seguinte maneira: os capítulos 2 e 3 apresentam os conceitos básicos sobre grafos e lógica proposicional necessários para o entendimento deste texto. O capítulo 4, sobre complexidade computacional, tem o objetivo de esclarecer a importância dos problemas considerados neste trabalho e a maneira em que se relacionam. Introduzidos os fundamentos, no capítulo 5 são apresentadas duas reduções do problema do caminho hamiltoniano para o problema de satisfazibilidade encontradas na literatura. O capítulo 6 traz detalhes sobre a implementação das reduções e as aplicações de satisfazibilidade escolhidas para os testes. O capítulo 7 detalha os dois tipos de grafos escolhidos para testes, grafos aleatórios e grafos do problema do passeio do cavalo, e os resultados obtidos. Por fim, apresenta-se a conclusão do trabalho e a perspectiva de trabalhos futuros.

2 Grafos

Este capítulo introduz a estrutura matemática chamada *grafo*. A teoria dos grafos dispõe de conceitos que podem descrever diversos problemas e de uma grande coleção de teoremas sobre propriedades de grafos [1, 2].

Considerada a primeira publicação sobre grafos, "Solutio problematis ad geometriam situs pertinentis" foi escrita pelo matemático Leonard Euler em 1736 [2]. Gondran, Minoux e Vajda[2] listam publicações seminais na história dos grafos.

O conceito de caminho hamiltoniano e o problema do caminho hamiltoniano são apresentados neste capítulo. Um levantamento parcial de técnicas disponíveis para determinação de caminhos hamiltonianos também é exposto.

As definições apresentadas neste capítulo são baseadas em [2].

2.1 Definição

Um grafo G é um par (V, E). Nesse par, V é um conjunto cujos elementos são chamados de vértices e $E \subseteq V \times V$ é um conjunto cujos elementos são chamados de arestas.

Grafos podem ser representados graficamente através de um círculo para cada vértice e uma seta entre os vértices v_i e v_j para cada aresta $(v_i, v_j) \in E$. Como exemplo, a representação gráfica do grafo $G_1 = (V_1, E_1)$, $V_1 = \{v_1, v_2, v_3, v_4\}$, $E_1 = \{(v_1, v_2), (v_2, v_4), (v_2, v_3)\}$ é apresentada na figura 1.

Dados $v_i, v_j \in V_1$, o grafo G_1 poderia, por exemplo, representar:

- Um conjunto de cidades V_1, tal que v_i tem uma aresta para o v_j se e somente se existe uma estrada ligando v_i diretamente a v_j;
- Um conjunto de pessoas V_1, tal que v_i tem uma aresta para v_j se e somente se v_i

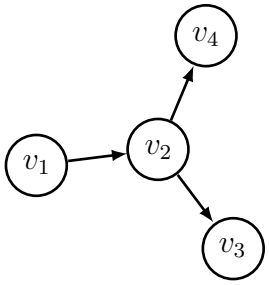

Figura 1: Grafo G_1.

conhece v_j pessoalmente;

- Um conjunto de estados de um jogo de xadrez V_1, tal que v_i tem uma aresta para v_j se e somente um movimento válido em v_i resulta em v_j;

O conceito de grau de um vértice é necessário para a compreensão da próxima seção. O grau de saída de um vértice v é o número de arestas com v como primeiro elemento. De forma semelhante, grau de entrada de um vértice é o número de arestas com v como segundo elemento. O grau de um vértice é a soma do grau de entrada com o grau de saída.

2.2 Caminho Hamiltoniano

Define-se um caminho de cardinalidade q em um grafo $G = (V, E)$ como qualquer sequência C da forma $C = (i_0, i_1), (i_1, i_2), ..., (i_{q-1}, i_q)$, com $(i_j, i_{j+1}) \in E$. Intuitivamente, um caminho é uma sequência de arestas que pode ser percorrida em ordem em um grafo. Um caminho onde $i_0 = i_q$ é chamado de circuito.

Um grafo não-orientado é um grafo cuja ordem de elementos em uma aresta não é importante. Um caminho em um grafo não-orientado, portanto, não precisa respeitar a "direção" das arestas [2].

Dado um grafo $G = (V, E)$, um caminho $C = (i_0, i_1), (i_1, i_2), ..., (i_{q-1}, i_q)$, com $(i_j, i_{j+1}) \in E$ e a sequência $S = i_0, i_1, ..., i_q$, diz-se que C é um caminho hamiltoniano se e somente todo vértice $v \in V$ aparece em S exatamente uma vez. Em termos simples, um caminho é dito hamiltoniano se e somente se todos os vértices do grafo são visitados por ele exatamente uma vez. A figura 2 exemplifica um grafo com um caminho hamiltoniano. O grafo G_1, já apresentado na figura 1, é um exemplo de grafo sem caminho hamiltoniano.

Um caminho hamiltoniano que também é um circuito é chamado de circuito hamilto-

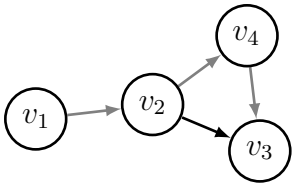

Figura 2: Um grafo com um caminho hamiltoniano destacado em cinza.

niano.

O termo hamiltoniano é uma referência ao matemático William Hamilton. Em 1859, Hamilton inventou um jogo equivalente a encontrar um circuito hamiltoniano em um grafo determinado pelos vértices e arestas de um dodecaedro [2].

No exemplo envolvendo cidades descrito na seção anterior, encontrar um caminho hamiltoniano seria equivalente à encontrar um caminho que passa exatamente uma vez por cada cidade.

Várias condições necessárias ou suficientes para que um grafo *não-orientado* possua um *circuito* hamiltoniano são expostas por Gondran, Minoux e Vajda[2], West[1] e Vandegriend[7]. São exemplos dessas condições que só dependem das definições já apresentadas:

- Condição necessária: Todo vértice tem grau maior ou igual a 2 [7]. Se um vértice tem grau 2, todas as arestas deste vértice estarão no circuito hamiltoniano [7].
- Condição necessária: Nenhum vértice está ligado por aresta a três vértices diferentes de grau 2 [7].
- Condição suficiente: O grafo tem mais de dois vértices, nenhum vértice tem aresta para si mesmo e todos os vértices tem grau maior ou igual a metade do número de vértices [7].

Esses teoremas podem ser úteis para o problema do caminho hamiltoniano em grafos orientados quando considerados dois fatos:

- Dados dois vértices v_i e v_f em um grafo orientado G, é possível construir um grafo não-orientado G' a partir de G que possui um caminho hamiltoniano se e somente se G possui um caminho hamiltoniano começando em v_i e terminando em v_f [3]. Além disso, um caminho encontrado em G' pode ser facilmente traduzido para um caminho em G [3].

- Para transformar o problema de encontrar um caminho hamiltoniano no problema de encontrar um circuito hamiltoniano, basta que seja adicionado ao grafo um novo vértice e adicionadas arestas partindo dele para todos os outros vértices e partindo de todos os outros vértices para ele [8].

2.2.1 Determinação de Caminhos Hamiltonianos

Determinar a existência de caminhos hamiltonianos em um grafo é especialmente interessante para a teoria da computação [3]. Isso porque o problema, juntamente com o circuito hamiltoniano [1, 7] e o problema para grafos não-orientados [3], pertence à classe de problemas NP-completos. A importância dessa classe será detalhada no capítulo 4.

Existem diversas técnicas para determinação de caminhos (ou circuitos) hamiltonianos. A busca exaustiva, apesar de simples, é impratícavel devido ao rápido crescimento do número de combinações possíveis [9].

O mais abrangente levantamento sobre técnicas de determinação de caminhos hamiltonianos encontrado na literatura foi realizado por Vandegriend[7] sobre *circuitos* hamiltonianos. Segundo Vandegriend[7], as técnicas são baseadas principalmente numa combinação de heurísticas de extensão de caminhos parciais. Essas heurísticas podem ser sumarizadas na seguinte lista:

- Transformação rotacional: alteração dos pontos extremos de um caminho que não altera os vértices envolvidos.
- Transformação rotacional orientada: usada em grafos orientados, divide um caminho em um caminho mais um circuito e então transforma ambos em apenas um circuito.
- Extensão de circuito: dado um circuito parcial, um vértice qualquer presente nele é escolhido. O vértice é então transformado no vértice final de um caminho através da remoção de uma aresta. Um vértice fora do circuito parcial para o qual o vértice escolhido tenha uma aresta é então inserido no novo caminho.
- Transformação rotacional de retrocesso (backtrack): envolve uso de transformações rotacionais para inserir em um circuito parcial um vértice ainda não presente.
- Extensão cruzada (*crossover*): pode ser considerada uma formalização ou extensão da transformação rotacional orientada.

- Extensão por contorno (*bypass*): usada quando um caminho não pode ser extendido pelas heurísticas anteriores. Consiste em encontrar um subcaminho em um caminho parcial que possa incluir mais vértices.

- Extensão de cadeia: uma formalização da combinação entre extensão cruzada e extensão por contorno.

- Caminhos de menor grau: envolve a criação de caminhos com vértices extremos de alto grau e vértices interiores de baixo grau, com o objetivo de facilitar a extensão do caminho.

- Menor grau primeiro: na extensão de um caminho, escolhem-se primeiro vértices de baixo grau, com o objetivo de retroceder mais próximo das folhas da árvore de busca. A vantagem dessa heurística de escolha também é detalhada por Russell e Norvig[6] para problemas de satisfação de restrições.

- Busca em múltiplos caminhos: uma lista de caminhos é expandida aleatoriamente até que tenham pontos extremos em comum.

- Poda (*pruning*): consiste em eliminar ramos de uma árvore de busca. A determinação de que ramos podem ser eliminados é bastante dependente do conjunto de heurísticas empregadas.

A transformação do problema do caminho hamiltoniano para problema de circuito hamiltoniano citada na seção anterior pode ser usada para que essas heurísticas sejam aproveitadas.

Definido o problema do caminho hamiltoniano e expostas algumas heurísticas usadas para resolvê-lo, o próximo capítulo introduz a lógica proposicional e o problema de satisfazibilidade booleana. A relação entre esse problema e o problema do caminho hamiltoniano será exposta no capítulo 4.

3 *Lógica Proposicional*

A lógica proposicional (também chamada de lógica booleana) é o estudo de proposições lógicas e suas combinações usando conectivos lógicos [10].

Este capítulo é baseado em [10] e apresenta os fundamentos de lógica proposicional necessários para o entendimento do problema de satisfazibilidade booleana e das reduções apresentadas.

3.1 Proposições

Uma proposição lógica é uma sentença declarativa em linguagem natural que tem um valor verdade bem definido. Um valor verdade é exclusivamente verdadeiro ou falso. São exemplos de proposições:

- Azul é uma cor.
- Carros voam.
- Todos os carros são azuis e voam.

Assim como Sipser[3], este texto representará o valor verdadeiro com o número 1 e falso com o número 0. Rosen et al.[10] usam T e F para representar verdadeiro e falso, respectivamente. A representação numérica facilitará o entendimento de uma das reduções apresentada no capítulo 5.

Proposições lógicas podem ser representadas por variáveis lógicas. A proposição "Azul é uma cor" seria associada à variável p através da denotação p : "Azul é uma cor", por exemplo.

3.2 Conectivos lógicos

Conectivos lógicos são operadores usados para construir proposições compostas a partir de outras proposições. Uma proposição que não é composta é chamada de proposição atômica. Proposições, compostas ou não, também são chamadas de expressões lógicas, expressões booleanas ou fórmulas booleanas.

Um exemplo de conectivo lógico é o operador de negação, denotado por $\neg$. Dada uma proposição p, o valor verdade de $\neg p$ é o oposto do valor de p.

Uma tabela verdade é uma forma de representar os valores de uma expressão lógica para todas as possíveis combinações de valores verdade das proposições atômicas que a compõe. A tabela 1 apresenta a tabela verdade da proposição $\neg p$.

p	$\neg p$
0	1
1	0

Tabela 1: Tabela verdade da proposição $\neg p$.

O operador de conjunção representa a operação intuitiva "e" da linguagem natural e é denotado por $\wedge$. A proposição $p \wedge q$ só é verdadeira quando as proposições p e q são verdadeiras (tabela 2).

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 2: Tabela verdade da proposição $p \wedge q$.

O operador de disjunção representa a operação de "ou" não exclusivo da linguagem natural e é denotado por $\vee$. A proposição $p \vee q$ só é verdadeira quando pelo menos a proposição p ou a proposição q é verdadeira (tabela 3).

O operador de condicional, denotado por $\rightarrow$, representa a relação de implicação. Uma proposição $p \rightarrow q$ é falsa somente quando p é verdadeira e q é falsa (tabela 4).

O operador $\leftrightarrow$, denominado bicondicional, representa uma relação de equivalência ou igualdade entre proposições. A proposição $p \leftrightarrow q$ é verdadeira somente quando p e q são iguais (tabela 5).

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 3: Tabela verdade da proposição $p \vee q$.

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Tabela 4: Tabela verdade da proposição $p \rightarrow q$.

No cálculo do valor verdade de proposições compostas é estabelecida uma ordem de precedência. Assim como em expressões aritméticas, é válido o uso de parênteses. Os valores de expressões entre os parênteses mais internos são os primeiros a serem calculados. Em expressões sem parênteses, calcula-se resultado das expressões na seguinte ordem: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$. Para maior clareza, o uso de parênteses é preferível à dependência das regras de precedência entre operadores, exceto na operação de negação [10].

Uma atribuição de valores verdade a uma proposição composta p que a torne verdadeira é chamada de modelo de p.

Uma implicação $p \rightarrow q$ verdadeira para qualquer atribuição de valores às proposições atômicas que compõe p e q é denotada como $p \implies q$. De forma semelhante, $p \iff q$ denota que a proposição $p \leftrightarrow q$ é sempre verdadeira.

Duas proposições são denominadas equivalentes quando seu valor verdade é sempre igual para todos os possíveis valores verdade das proposições atômicas que as compõe. A equivalência entre p e q seria denotada por $p \iff q$ ou $p \equiv q$. A tabela 6 exemplifica o procedimento de criação de uma tabela verdade parar provar a equivalêcia entre as

p	q	$p \leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

Tabela 5: Tabela verdade da proposição $p \leftrightarrow q$.

proposições $(p \vee q)$ e $\neg(\neg p \wedge \neg q)$.

p	q	$\neg p$	$\neg q$	$p \vee q$	$\neg p \wedge \neg q$	$\neg(\neg p \wedge \neg q)$
0	0	1	1	0	1	0
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	1	0	1

Tabela 6: Tabela que demonstra a equivalência $(p \vee q) \iff \neg(\neg p \wedge \neg q)$.

Algumas equivalências lógicas usadas frequentemente são chamadas de identidades lógicas. A tabela 7 mostra algumas dessas identidades.

Nome	Equivalência
Leis Comutativas	$(p \wedge q) \iff (q \wedge p)$ $(p \vee q) \iff (q \vee p)$
Leis Associativas	$p \wedge (q \wedge r) \iff (p \wedge q) \wedge r$ $p \vee (q \vee r) \iff (p \vee q) \vee r$
Leis Distributivas	$p \wedge (q \vee r) \iff (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \iff (p \vee q) \wedge (p \vee r)$
Leis de De Morgan	$\neg(p \wedge q) \iff (\neg p \vee \neg q)$; $\neg(p \vee q) \iff (\neg p \wedge \neg q)$
Meio excluído	$p \vee \neg p \iff 1$
Contradição	$p \wedge \neg p \iff 0$
Lei da Dupla Negação	$\neg(\neg p) \iff p$
Lei da Contrapositiva	$p \to q \iff \neg q \to \neg p$
Condicional como disjunção	$p \to q \iff \neg p \vee q$
Negação do condicional	$\neg(p \to q) \iff p \wedge \neg q$
Leis de Idempotência	$p \wedge p \iff p$ $p \vee p \iff p$
Leis de Absorção	$p \wedge (p \vee q) \iff p$ $p \vee (p \wedge q) \iff p$
Leis de Dominância	$p \vee 1 \iff 1$ $p \wedge 0 \iff 0$
Leis de Exportação	$p \to (q \to r) \iff (p \wedge q) \to r$
Leis de Identidade	$p \wedge 1 \iff p$ $p \vee 0 \iff p$

Tabela 7: Tabela de identidades lógicas. Fonte: [10]

Existem vários outros operadores lógicos que não são usados neste trabalho. No entanto, os operadores $\wedge$ e $\neg$, por exemplo, podem ser usados para construir proposições equivalentes a qualquer proposição composta [10].

Operadores iterados são definidos para as operações de conjunção e disjunção conforme

as fórmulas 3.1 e 3.2 [3].

$$(a_1 \wedge a_2 \wedge ... \wedge a_n) \iff \bigwedge_{i=1}^{n} a_i \tag{3.1}$$

$$(a_1 \vee a_2 \vee ... \vee a_n) \iff \bigvee_{i=1}^{n} a_i \tag{3.2}$$

3.3 Forma Normal Conjuntiva

Se p é uma proposição atômica, as proposições p e $\neg p$ são chamadas de literais.

Uma cláusula disjuntiva de tamanho n é uma cláusula composta pela operação de disjunção entre n literais a_i: $(a_1 \vee a_2 \vee ... \vee a_n)$.

Dada uma expressão proposicional, diz-se que ela está na forma normal conjuntiva (ou CNF, do inglês, *conjunctive normal form*) se é escrita como a conjunção de cláusulas disjuntivas conforme a fórmula 3.3. Ressalta-se que todos os $x_{w,y}$ presentes na fórmula são necessariamente literais.

$$(x_{1,1} \vee x_{1,2} \vee ... \vee x_{1,i}) \wedge (x_{2,1} \vee x_{2,2} \vee ... \vee x_{2,j}) \wedge ... \wedge (x_{c,1} \vee x_{c,2} \vee ...x_{c,k}) \tag{3.3}$$

Se todas as cláusulas de uma fórmula conjuntiva normal tiverem o mesmo tamanho t, diz-se que ela está em tCNF.

Todas as expressões proposicionais tem proposições equivalentes em forma conjuntiva normal [10]. Além disso, toda proposição em forma conjuntiva normal tem uma equivalente em 3CNF [5].

3.4 Satisfazibilidade

Algumas proposições compostas são sempre verdadeiras, independente do valor das proposições atômicas que as compõe. Essas proposições são chamadas de tautologias. Um exemplo de tautologia é a proposição $p \vee \neg p$. Em contrapartida, algumas proposições compostas são sempre falsas, independente do valor das proposições atômicas que as compõe. Essas proposições são chamadas de contradições ou proposições insatisfazíveis.

A proposição $p \wedge \neg p$ é insatisfazível.

Uma proposição p é dita satisfazível se e somente se existe pelo menos uma atribuição de valores verdade às proposições atômicas que a compõe que a torne verdadeira. Determinar se uma proposição é satisfazível é denominado problema da satisfazibilidade booleana [3].

Este problema, assim como o problema do caminho hamiltoniano, é NP-completo [3]. O capítulo 4 detalha a classe de problemas NP-completos. O termo booleana é uma referência a George Boole, desenvolvedor da álgebra booleana. A álgebra booleana é uma generalização da álgebra de proposições lógicas [10].

3.4.1 Determinação de Satisfazibilidade

As aplicações práticas do problema da satisfazibilidade booleana são variadas: prova automática de teoremas, verificação de hardware, agendamento e vários problemas combinatórios [5, 6]. Essa importância prática aliada à importância teórica dos problemas NP-completos faz com que a pesquisa de algoritmos de satisfazibilidade seja bastante ativa [5].

Esta seção apresenta três desses algoritmos. Dois deles são usados nas aplicações escolhidas para os testes, detalhadas no capítulo 6. O algoritmo de força bruta, que não é usado por essas aplicações, é útil para ilustrar a dificuldade de uma abordagem que não envolve heurísticas. Outros algoritmos podem ser encontrados em [5].

A descrição dos algoritmos é baseada em [6].

3.4.1.1 Algoritmo de Força Bruta

O algoritmo de força bruta é bastante simples. Dada uma fórmula booleana Φ com n variáveis (proposições atômicas), o algoritmo executa uma enumeração em profundidade de todas as atribuições possíveis. São possíveis 2^n atribuições diferentes de valores às variáveis. Dada uma atribuição, checar se ela torna a fórmula verdadeira é uma tarefa relativamente rápida, que consiste numa substituição de variáveis e aplicação de operadores.

A impraticabilidade desse algoritmo é clara: para determinar se uma fórmula com 30 variáveis é satisfazível precisariam ser testadas, no pior caso, 2^{30} atribuições, ou seja, mais de um bilhão de atribuições.

3.4.1.2 Algoritmo DPLL

O algoritmo DPLL (a sigla corresponde às iniciais do sobrenome dos autores: Davis, Putnam, Logemann, Loveland) consiste essencialmente, assim como o algoritmo de força bruta, numa enumeração de possíveis modelos. No entanto, ele implementa melhorias durante o processo de enumeração que o tornam um dos mais rápidos algoritmos de determinação de satisfazibilidade [6]. As melhorias dependem que a fórmula esteja em CNF. Conforme apresentado na seção 3.3, toda fórmula booleana pode ser transformada para CNF.

As melhorias implementadas são as seguintes:

- Terminação antecipada: O algoritmo pode determinar se a fórmula é satisfazível muito antes que todas as atribuições sejam feitas. Uma cláusula é verdadeira se um dos seus literais for verdadeiro, mesmo que os outros literais ainda não tenham atribuições. De maneira similar, a fórmula será insatisfazível se qualquer cláusula for falsa.

  A terminação antecipada elimina a necessidade de examinar sub-árvores inteiras do espaço de busca.

- Heurística de variáveis puras: Uma variável pura é uma variável que aparece sempre através do mesmo literal dentro das cláusulas. Isto é, uma variável é pura se aparece exclusivamente negada ou atomicamente na fórmula. Se uma fórmula for satisfazível, haverá sempre um modelo com uma variável pura atribuída de forma a fazer seus literais verdadeiros, já que isso nunca pode tornar uma cláusula falsa. Esse fato guia a atribuição de valores verdade para as variáveis puras.

  Nota-se que para determinar se uma variável é pura, o algoritmo pode ignorar cláusulas que já são verdadeiras por atribuição. Portanto, uma variável pode ser considerada pura quando aparece em cláusulas ainda não verdadeiras sempre através do mesmo literal.

  Como exemplo, dada a fórmula $\Phi = (a \vee \neg b) \wedge (\neg b \vee \neg c) \wedge (c \vee a)$ tem b e a como variáveis puras. A atribuição $b = 0$ tornará a cláusula em que c aparece no literal $\neg c$ verdadeira. A partir disso, c pode ser considerado puro pois apenas aparece numa cláusula não verdadeira no literal c da terceira cláusula [6].

- Heurística de cláusula unitária: uma cláusula é dita unitária quando só tem um literal. No contexto do DPLL, esse conceito é expandido de forma a incluir cláu-

sulas em que todos os outros literais já são falsos. Obviamente, cláusulas unitárias devem ter um literal verdadeiro. Esse fato guia a atribuição de valores verdade para cláusulas unitárias.

Nota-se que a atribuição de um valor para uma cláusula unitária também pode tornar outras cláusulas unitárias, criando uma cascata chamada de "propagação unitária".

Outras melhorias que podem ser adicionadas ao algoritmo DPLL básico são descritas por [11].

3.4.1.3 Algoritmo Walksat

O algoritmo Walksat é bastante diferente dos outros apresentados, pois usa uma busca local estocástica. Uma busca local, neste contexto, é uma busca que opera alterando o valor verdade de uma variável de cada vez, sem intenção de enumerar exaustivamente as atribuições. A busca é estocástica pois usa números pseudo-aleatórios para tomar decisões. A busca ocorre da seguinte forma:

- Um valor verdade aleatório é sorteado para cada variável.
- Uma cláusula não satifeita é escolhida. Uma probabilidade p, parâmetro do algoritmo, é usada para escolher entre duas ações: trocar a atribuição para uma variável qualquer da cláusula ou trocar a atribuição para uma variável que minimizaria o número de cláusulas insatisfeitas.

Esse processo é repetido até que um modelo seja encontrado ou que um limite de iterações seja alcançado. O algoritmo é incompleto, pois não é capaz de determinar em todo caso se uma fórmula é satisfazível.

Apesar da simplicidade desse algoritmo, ele tem bom desempenho em problemas reais [6]. Um exemplo de sucesso é a aplicação desse algoritmo em problemas de planejamento (descoberta de uma cadeia de ações para alcançar um objetivo) transformados em problema de satisfazibilidade booleana [6]. Esse sucesso é uma das motivações para este trabalho.

No próximo capítulo a teoria da complexidade computacional é introduzida. Ela garante a possibilidade de transformar uma instância do problema do caminho hamiltoniano em um problema de satisfazibilidade booleana, assim como feito com o problema do planejamento.

4 Complexidade Computacional

Este capítulo introduz conceitos da teoria da computação necessários para o estudo da complexidade computacional. A complexidade computacional esclarece a importância dos problemas centrais neste trabalho e como eles se relacionam. O capítulo foi baseado principalmente em Sipser[3].

4.1 Linguagem

Uma linguagem, para a teoria da computação, é um conjunto de *sentenças* sobre um *alfabeto*. Nesse contexto, um alfabeto é definido como um conjunto não-vazio de elementos chamados símbolos. Uma sentença de tamanho n sobre um alfabeto é qualquer sequência de tamanho n formada somente por símbolos desse alfabeto. Como exemplo, seja $\Sigma = \{a, b\}$ um alfabeto. São sentenças sobre Σ: a,b,ba,$aaaab$.

Dada uma sentença w, o tamanho da sequência que a compõe é denotado por $|w|$. Por convenção, uma sentença denotada por ε tem tamanho 0 e é uma sentença sobre qualquer alfabeto.

A concatenação é uma operação definida sobre duas sentenças x e y e denotada como xy. Consiste em anexar a sequência y ao fim da sequência x. Como exemplo, se $x = aba$ e $y = ba$, $xy = ababa$. A concatenação de n sentenças x pode ser representada com x^n. Como exemplo, $a^4 = aaaa$.

O papel dessa definição formal de linguagem ficará claro na próxima seção, que introduz máquinas de Turing.

4.2 Máquina de Turing

Uma máquina de Turing é um modelo matemático de dispositivo computacional proposto em 1936 por Alan Turing. A comumente aceita tese de Church-Turing afirma que

máquinas de Turing são capazes de executar qualquer processo entendido intuitivamente como um algoritmo. Apesar de não ser possível provar essa tese, já que não corresponde a uma proposição matemática, ela pode ser contradita no futuro com a implementação física de um novo modelo de computação [4]. Existem vários dispositivos equivalentes em poder computacional à máquina de Turing, entre eles: sistemas de Post, funções recursivas, cálculo lambda e algoritmos de Markov [12].

A importância principal da máquina de Turing é permitir um estudo matemático das propriedades da computação [3, 4]. Um dos resultados desse estudo é a descoberta que uma máquina de Turing não pode resolver determinados problemas computacionais. Esses problemas estão além dos limites da computação e são chamados de problemas *incomputáveis*. O significado formal de problema computacional ficará claro no decorrer desta seção.

Um modelo intuitivo do funcionamento da máquina de Turing será apresentado com o objetivo de facilitar o entendimento do modelo formal.

Uma máquina de Turing usa uma fita infinita como memória. Ela tem um cabeçote de leitura que pode ler e escrever símbolos nessa fita. Esse cabeçote pode mover-se apenas para a direita ou para esquerda, um passo de cada vez. A máquina recebe como entrada uma sentença que começa na posição mais a esquerda da fita. O cabeçote é posicionado sobre o primeiro símbolo dessa sentença. O resto da fita é preenchido com o símbolo $\sqcup$, que representa um espaço vazio. Internamente, a máquina tem estados que são alterados conforme regras pré-definidas, que levam em consideração o conteúdo da fita e o estado em que a máquina se encontra. Essas regras correspondem ao programa da máquina de Turing.

O objetivo de qualquer máquina de Turing é determinar se a sentença de entrada pertence ou não a uma determinada linguagem. Diz-se que uma máquina de Turing M reconhece a linguagem L quando atinge um estado especial, chamado de estado de aceitação, se e somente se a sentença de entrada pertence a L. O estado especial de rejeição é atingido somente (mas não necessariamente, como detalhado a seguir) quando a sentença de entrada não pertence a L. A máquina encerra seu processamento imediatamente quando alcança um desses estados especiais.

Diz-se que uma máquina de Turing decide uma linguagem se ela sempre aceita sentenças que pertencem à linguagem e sempre rejeita sentenças que não pertencem à linguagem. É possível que uma máquina de Turing entre entre em um ciclo de processamento interminável (chamado *loop*). Uma máquina que entra em *loop* para alguma sentença não

decide uma linguagem. É possível provar que máquinas de Turing não podem decidir determinadas linguagens. Essas linguagens são chamadas de linguagens indecidíveis.

Qualquer problema computacional pode ser redefinido como um problema de reconhecimento de linguagens. O problema de determinar se uma lista está ordenada, por exemplo, pode ser transformado no problema de reconhecer elementos da linguagem formada por representações de listas que estejam ordenadas. Sentenças podem ser usadas para representar qualquer estrutura matemática: números, grafos, listas, conjuntos etc. Por convenção, denota-se por $< x >$ a representação numa sentença da estrutura x.

Se um problema requer que a máquina de Turing produza uma saída, ao final do processamento a fita deve conter apenas a representação $< y >$ da saída esperada y. Voltando ao exemplo de listas, uma máquina de Turing seria capaz de receber uma representação de uma lista como entrada, realizar sua ordenação na fita e alcançar o estado de aceitação (reconhecendo, por consequência, a linguagem de listas ordenáveis).

Introduzido o modelo intuitivo do funcionamento da máquina, o modelo formal segundo Sipser[3] será apresentado.

Uma máquina de Turing M é definida como uma sétupla:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

O conjunto Q é o conjunto de estados. O conjunto Σ é o conjunto de símbolos, que não pode conter o símbolo $\sqcup$, e é chamado de alfabeto de entrada. O conjunto Γ, tal que $\Sigma \subset \Gamma$, é chamado de alfabeto da fita e sempre contém $\sqcup$.

A função $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ é chamada de função de transição. O operador $\times$ é o produto cartesiano usual. Dados dois conjuntos A e B, $A \times B = \{(a, b) | (a \in A) \wedge (b \in B)\}$.

O estado $q_0 \in Q$ é chamado de estado inicial. O estado $q_{\text{accept}} \in Q$ é o estado de aceitação. O estado $q_{\text{reject}} \in Q$ é chamado de estado de rejeição. O estado de aceitação não pode ser igual ao estado de rejeição.

O estado atual em que se encontra a máquina, o próximo símbolo a ser lido e o conteúdo da fita determinam unicamente o estágio do processamento e juntos são chamados de configuração. Uma configuração pode ser representada pelo conteúdo da fita antes do próximo símbolo a ser lido unido com o identificador do estado e o conteúdo depois do símbolo a ser lido. A configuração inicial de uma máquina de Turing com estado inicial

q_0 que recebe a sentença *aba* como entrada seria representada, por exemplo, por $q_0$aba.

A computação ocorre como segue. Dada uma sentença de entrada $w = w_1w_2...w_n$ com $w_i \in \Sigma$, a máquina assume a configuração $q_0w_1w_2...w_n$, onde q_0 é o estado inicial da máquina. A ação da máquina para essa configuração é dada pela tripla $\delta(q_0, w_1) = (q', w', x)$. Isso significa que a próxima configuração será formada pelo estado q', o símbolo w_1 será sobrescrito por w' e o próximo símbolo a ser lido da fita estará a esquerda ou a direita do símbolo atual w_1, conforme $x \in \{L, R\}$ (L representa esquerda, R representa direita). Tomando como exemplo $\delta(q_0, w_1) = (q', w', R)$, a configuração subsequente a $q_0w_1w_2...w_n$ seria $w'q'w_2...w_n$. Por definição, um movimento para a esquerda quando a máquina já está no primeiro símbolo da fita faz com que o primeiro símbolo seja o próximo a ser lido. Um movimento para a direita quando a máquina já está no último símbolo da configuração faz com que o símbolo $\sqcup$ seja lido.

De forma semelhante, essa troca de configurações acontece até que o estado de aceitação ou de rejeição seja alcançado, quando a máquina pára e informa se a sentença pertence ou não à linguagem.

A máquina de Turing $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, que reconhece a linguagem $A = \{0^{2^n} | n \geq 0\}$ (linguagem das sentenças compostas apenas por zeros cujos tamanhos são potências de 2), será usada como exemplo. Os componentes da sétupla M_1 são definidos como segue: $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$, $\Sigma = \{0\}$, $\Gamma = \{0, x, \sqcup\}$. Os estados de início, aceitação e rejeição são q_1, q_{accept} e q_{reject}, respectivamente.

A função de transição pode ser representada com um diagrama de estados conforme a figura 3.

Neste diagrama, cada vértice representa um estado. Cada aresta do estado q' para o estado q'' é rotulada com $a \to b, c$ e indica que ao ler o símbolo a na fita, no estado q', a máquina sobrescreve a com b, vai para o estado q'' e move-se para a direção c na fita. Arestas com rótulos da forma $a \to c$ são equivalentes a $a \to a, c$. O estado inicial é indicado com uma aresta sem origem.

Expressar algoritmos complexos através de diagramas de estados é bastante difícil. Por esse motivo, máquinas de Turing são comumente apresentadas com pseudocódigo. A máquina M_1 poderia ser descrita da seguinte forma (as numerações correspondem a estágios do algoritmo):

1. Troque um 0 por x avance para a direita. Deixe o próximo 0 intacto. Faça isso até o fim da fita.

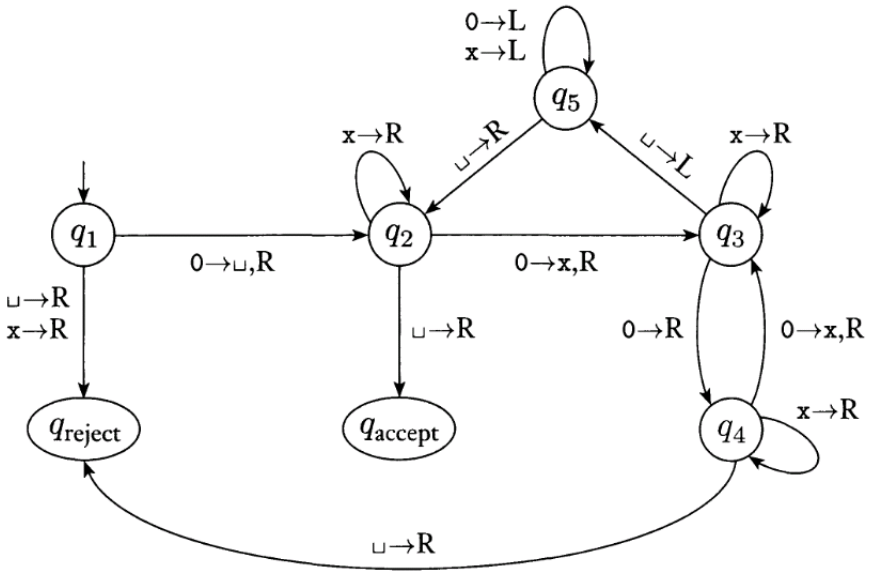

Figura 3: Diagrama de estados da máquina de Turing M_1. Fonte: [3].

2. Se no estágio 1 a fita continha apenas um 0, aceite.

3. Se no estágio 1 a fita continha mais do que um 0 e tinha um número ímpar de 0s, rejeite.

4. Retorne o cabeçote para a posição mais a esquerda.

5. Volte para o estágio 1.

Explicado o modelo usual de máquina de Turing, também chamado de máquina de Turing determinística, outro modelo equivalente em poder de reconhecimento também será útil para o estudo de complexidade computacional. Esse modelo é chamado de máquina de Turing não-determinística.

A máquina de Turing não-determinística é bastante semelhante a apresentada até agora. A diferença é que a partir de qualquer configuração a máquina pode ir para um conjunto de configurações, conforme a função de transição $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. O conjunto $\mathcal{P}(A)$ é chamado de conjunto potência de A. É o conjunto de todos os subconjuntos de A: $\mathcal{P}(A) = \{B | B \subset A\}$.

Intuitivamente, a computação de uma máquina de Turing não-determinística forma uma árvore (grafo sem circuitos) cujos vértices são configurações e arestas partem para as configurações que sucedem uma configuração diretamente. A figura 4 apresenta uma comparação entre a computação de uma máquina não-determinística com a de uma má-

quina determinística. A função f apresentada nas laterais da figura 4 será detalhada na próxima seção.

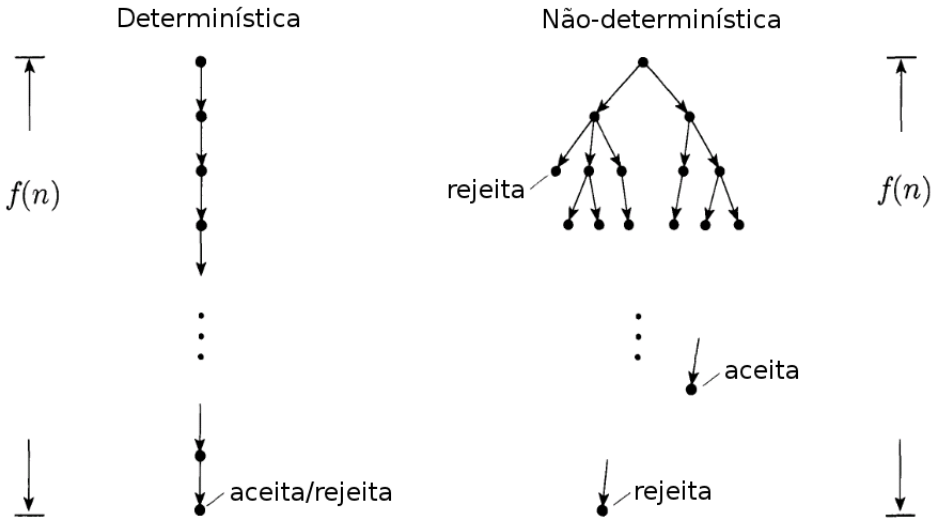

Figura 4: Sucessão de configurações em máquinas determinísticas e máquinas não-determinísticas. Fonte: [3].

Se algum ramo da árvore encontra um estado de aceitação, a máquina aceita a entrada. Se um ramo encontra um estado de rejeição a sentença não é imediatamente rejeitada. Uma sentença é rejeitada apenas se for rejeitada em todos os ramos da árvore. É importante que a máquina não-determinística explore a árvore de configurações em largura, para evitar que um ramo que entra em *loop* impeça outros ramos de serem explorados.

4.3 Classes de Complexidade

Dadas duas funções $f : \mathbb{N} \to \mathbb{R}^+$ e $g : \mathbb{N} \to \mathbb{R}^+$, diz-se que $f(n) = O(g(n))$ se existem inteiros positivos c e n_0 tal que, para qualquer $n \geq n_0$, $f(n) \leq cg(n)$. Intuitivamente, $f(n) = O(g(n))$ significa que a função f é aproximadamente igual a g se desprezadas diferenças até um fator constante [3].

Diz-se que uma máquina de Turing determinística decide uma sentença de tamanho n em tempo $f(n)$ se são necessárias $f(n)$ trocas de configuração para que ela aceite ou rejeite essa sentença.

Para uma máquina de Turing não-determinística, decidir uma sentença de tamanho n em tempo $f(n)$ significa encontrar o estado de aceitação ou rejeição em uma profundidade $f(n)$ na sua árvore de execução. A figura 4, apresentada na seção anterior, ilustra o

conceito de tempo nos dois tipos de máquina.

Uma linguagem é decidível em tempo $f(n)$ por uma máquina de Turing se e somente se a máquina decide qualquer sentença de tamanho n em tempo até $f(n)$.

A classe TIME($f(n)$) é a classe de todas as linguagens decidíveis por uma máquina de Turing determinística em tempo $O(f(n))$. De forma similar, a classe NTIME($f(n)$) é a classe de todas as linguagens decidíveis por uma máquina de Turing não-determinística em tempo $O(f(n))$.

4.3.1 Classe P

A classe P é a classe de linguagens que são decidíveis em tempo polinomial numa máquina de Turing determinística. Isto é:

$$\mathrm{P} = \bigcup_{k}^{\infty} \mathrm{TIME}(n^k)$$

4.3.2 Classe NP

A classe NP é a classe de linguagens que são decidíveis em tempo polinomial numa máquina de turing não-determinística. Isto é:

$$\mathrm{NP} = \bigcup_{k}^{\infty} \mathrm{NTIME}(n^k)$$

É fácil ver que $P \subseteq NP$, já que a máquina não-determinística pode se comportar de maneira determinística. Determinar se NP $\subseteq$ P, isto é, se P = NP é considerado um dos mais importantes problemas da teoria da computação [3, 4].

Uma definição alternativa de NP permite que problemas dessa classe sejam facilmente identificados [3]. Essa definição depende do conceito de verificador.

Um verificador para uma linguagem A é uma máquina de Turing determinística V conforme a fórmula 4.1. Uma linguagem que tem verificador que decide qualquer sentença em tempo polinomial está em NP.

$$A = \{w | V \text{ aceita } < w, c > \text{ para alguma sentença } c\} \tag{4.1}$$

Em outras palavras, se existe uma sentença c (chamada de certificado) que torna possível a decisão de qualquer $w \in A$ em tempo polinomial por V, A será decidível em tempo polinomial numa máquina determinística.

A justificativa intuitiva dessa afirmação é que a máquina não-determinística pode testar simultaneamente todos os certificados de um tamanho $i \in \mathbb{N}$ em tempo $f(i)$, onde f é uma função polinomial, até que algum seja aceito.

O conceito de verificador tornará fácil determinar que os dois problemas importantes para este trabalho estão em NP.

O problema de determinar se uma fórmula booleana é satisfazível pode ser transformado no problema de determinar se uma sentença pertence à linguagem SAT:

$$\text{SAT} = \{< \Phi > | \Phi \text{ é uma fórmula booleana satisfazível}\}$$

É fácil demonstrar que SAT $\in NP$, já que dada uma atribuição de valores verdade para as variáveis de Φ é possível testar em tempo polinomial se ela é um modelo. [3]. A representação de um modelo é, portanto, um certificado para essa linguagem.

O problema do caminho hamiltoniano pode ser transformado no problema de reconhecer elementos da linguagem $HAMPATH$:

$$\text{HAMPATH} = \{< G > | G \text{ é um grafo que possui pelo menos um caminho hamiltoniano}\}$$

Um certificado para $HAMPATH$ é representação da sequência de vértices que compõe o caminho [3].

4.3.2.1 NP-Completude

Uma linguagem A é polinomialmente redutível à outra linguagem B se existir uma máquina de Turing determinística que ao receber qualquer sentença a pára, em tempo polinomial, com a sentença b em sua fita de forma que $a \in A$ se e somente se $b \in B$.

Uma linguagem pertence à classe de problemas NP-completos se qualquer linguagem em NP puder ser reduzida em tempo polinomial a ela. A primeira linguagem a ser determinada NP-Completa, por Stephen Cook, foi SAT. A prova, bastante longa, pode ser encontrada em [3].

A partir da NP-completude de SAT é possível demonstrar que vários outros problemas são NP-completos. Isso pode ser feito demonstrando-se que existe uma redução em tempo polinomial de SAT para o problema que deseja-se provar NP-completo.

Um dos problemas demonstrados NP-Completos através de redução para *SAT* é o problema do caminho hamiltoniano [3]. Ou seja, é possível criar para qualquer fórmula booleana um grafo que possui um caminho hamiltoniano se e somente se a fórmula for satisfazível.

Não é conhecido um algoritmo em tempo polinomial que resolva sequer um problema NP-completo, apesar de muitos problemas de importância prática estarem nessa classe [13]. Uma lista abrangente de problemas problemas NP-completos pode ser encontrada em [13].

A importância dos problemas NP-completos fica clara pelo fato de que uma solução em tempo polinomial para qualquer um deles garantir uma solução em tempo polinomial para todos. Uma solução em tempo polinomial para um problema NP-completo provaria, inclusive, que P = NP.

O próximo capítulo apresenta reduções que criam uma fórmula booleana satisfazível a partir de um grafo se e somente se ele possui um caminho hamiltoniano. Elas são o inverso da redução aplicada para provar a NP-completude do problema do caminho hamiltoniano. A existência de reduções nessa direção é garantida pela NP-completude de *SAT*.

5 Reduções

Este capítulo descreve reduções do problema do caminho hamiltoniano para o problema de satisfazibilidade booleana, elementos centrais neste trabalho. Essas reduções permitem que técnicas de determinação de satisfazibilidade sejam usadas para determinar caminhos hamiltonianos em grafos. A pesquisa bibliográfica resultou na descoberta de duas reduções: uma proposta por Iwama e Miyazaki[14] e outra por Torn[15].

Dado um grafo com n vértices, a redução de Torn[15] gera fórmulas com n^2 variáveis. Em comparação, a redução de Iwama e Miyazaki[14] gera $n\lceil log_2 n \rceil$. Apesar dessa diferença, a análise das duas reduções pode ser justificada pelo trabalho de Hoos[16], que deixa claro que a estrutura das cláusulas é mais importante do que o número de vértices ou de cláusulas para o tempo de solução de uma fórmula.

Ainda concernente à estrutura das cláusulas, Russell e Norvig[6] apresentam o conceito de transição de fase, que identifica a razão entre o número de cláusulas e variáveis como um fator mais relevante para o tempo de solução do que seus tamanhos considerados individualmente.

5.1 Redução de Torn

Esta seção descreve a redução de Torn[15]. O grafo $G_z = (Z, E_z)$, $Z = \{z_0, z_1, z_2\}$ e $E_z = \{(z_0, z_1), (z_1, z_2)\}$ será usado como exemplo durante a explicação.

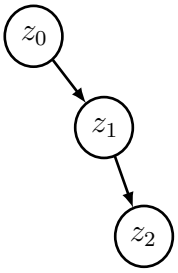

Figura 5: Grafo G_z.

Dado um grafo $G = (V, E)$ com $V = \{v_0, v_1, ..., v_{n-1}\}$, o objetivo da redução é construir uma fórmula booleana Φ satisfazível se e somente G tem um caminho hamiltoniano. Para isso, Φ terá n^2 variáveis chamadas $x_{i,j}$, com $0 \leq i, j < n$. Deseja-se que cada variável $x_{i,j}$ seja verdadeira em um modelo de Φ se e somente se a posição i no caminho for ocupada pelo vértice v_j.

Denotando por Φ_z a fórmula construída para o grafo de exemplo G_z, apenas as variáveis $x_{0,0}, x_{1,1}$ e $x_{2,2}$ devem ser verdadeiras em um modelo de Φ_z.

Será construída uma fórmula Φ que garante essa semântica para o caso geral. Ela será separada em cinco fórmulas menores para facilitar sua exposição, conforme a fórmula 5.1.

$$\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_4 \wedge \Phi_5 \tag{5.1}$$

A fórmula Φ_1 vai garantir que todos os vértices apareçam no caminho. Pode ser interpretada da seguinte forma: o vértice v_i está em alguma posição j.

$$\Phi_1 = \bigwedge_{i=0}^{n-1} \bigvee_{j=0}^{n-1} x_{j,i} \tag{5.2}$$

A fórmula Φ_1 gera n cláusulas de tamanho n.

Para o grafo G_z teria-se:

$$\Phi_{z_1} = (x_{0,0} \vee x_{1,0} \vee x_{2,0}) \wedge (x_{0,1} \vee x_{1,1} \vee x_{2,1}) \wedge (x_{0,2} \vee x_{1,2} \vee x_{2,2}) \tag{5.3}$$

A fórmula Φ_2 vai garantir que cada vértice apareça apenas uma vez no caminho. Dada a equivalência lógica $(\neg p \vee \neg q) \iff (p \rightarrow \neg q)$, essa fórmula representa que se uma posição j é ocupada por um vértice v_i, então nenhuma posição k subsequente a j pode ser ocupada pelo vértice v_i.

$$\Phi_2 = \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{n-1} \bigwedge_{k=j+1}^{n-1} (\neg x_{j,i} \vee \neg x_{k,i}) \tag{5.4}$$

Para o grafo G_z teria-se:

$$\begin{aligned}\Phi_{z_2} =&(\neg x_{0,0} \vee \neg x_{1,0}) \wedge (\neg x_{0,0} \vee \neg x_{2,0}) \wedge (\neg x_{1,0} \vee \neg x_{2,0}) \\ &(\neg x_{0,1} \vee \neg x_{1,1}) \wedge (\neg x_{0,1} \vee \neg x_{2,1}) \wedge (\neg x_{1,1} \vee \neg x_{2,1}) \\ &(\neg x_{0,2} \vee \neg x_{1,2}) \wedge (\neg x_{0,2} \vee \neg x_{2,2}) \wedge (\neg x_{1,2} \vee \neg x_{2,2})\end{aligned}$$

A fórmula Φ_2 gera $\frac{n^3-n^2}{2}$ cláusulas de tamanho 2.

A fórmula Φ_3 vai garantir toda posição seja ocupada por um nodo. Pode ser interpretada da seguinte forma: a posição i está associada a algum vértice v_j.

$$\Phi_3 = \bigwedge_{i=0}^{n-1} \bigvee_{j=0}^{n-1} x_{i,j} \tag{5.5}$$

Para o grafo G_z teria-se:

$$\Phi_{z_3} = (x_{0,0} \vee x_{0,1} \vee x_{0,2}) \wedge (x_{1,0} \vee x_{1,1} \vee x_{1,2}) \wedge (x_{2,0} \vee x_{2,1} \vee x_{2,2}) \tag{5.6}$$

A fórmula Φ_3 gera n cláusulas de tamanho n.

A fórmula Φ_4 vai garantir que dois nodos diferentes não ocupem a mesma posição. Representa que se uma posição i é ocupada por um vértice v_j, então não é ocupada por um vértice com índice k subsequente a j.

$$\Phi_4 = \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{n-1} \bigwedge_{k=j+1}^{n-1} (\neg x_{i,j} \vee \neg x_{i,k}) \tag{5.7}$$

Para o grafo G_z teria-se:

$$\begin{aligned}\Phi_{z_4} =&(\neg x_{0,0} \vee \neg x_{0,1}) \wedge (\neg x_{0,0} \vee \neg x_{0,2}) \wedge (\neg x_{0,1} \vee \neg x_{0,2}) \\ &(\neg x_{1,0} \vee \neg x_{1,1}) \wedge (\neg x_{1,0} \vee \neg x_{1,2}) \wedge (\neg x_{1,1} \vee \neg x_{1,2}) \\ &(\neg x_{2,0} \vee \neg x_{2,1}) \wedge (\neg x_{2,0} \vee \neg x_{2,2}) \wedge (\neg x_{2,1} \vee \neg x_{2,2})\end{aligned} \tag{5.8}$$

A fórmula Φ_4 gera $\frac{n^3-n^2}{2}$ cláusulas de tamanho 2.

A fórmula Φ_5 é a única dependente de arestas. Expressa que se um vértice v_j não tem uma aresta para um outro vértice v_k, então o vértice v_k não pode aparecer no caminho

diretamente após v_j.

$$\Phi_5 = \bigwedge_{i=0}^{n-2} \bigwedge_{j=0}^{n-1} \bigwedge_{k=0}^{n-1} f(i,j,k) \tag{5.9}$$

A função f é definida como:

$$f(i,j,k) = \begin{cases} (\neg x_{i,j} \vee \neg x_{i+1,k}) & \text{se } (v_j, v_k) \notin E \\ 1 & \text{se } (v_j, v_k) \in E \end{cases}$$

Para o grafo G_z teria-se:

$$\begin{aligned} \Phi_{z_5} =& (\neg x_{0,0} \vee \neg x_{1,0}) \wedge (\neg x_{0,0} \vee \neg x_{1,2}) \\ & (\neg x_{0,1} \vee \neg x_{1,0}) \wedge (\neg x_{0,1} \vee \neg x_{1,1}) \\ & (\neg x_{0,2} \vee \neg x_{1,0}) \wedge (\neg x_{0,2} \vee \neg x_{1,1}) \wedge (\neg x_{0,2} \vee \neg x_{1,2}) \\ & (\neg x_{1,0} \vee \neg x_{2,0}) \wedge (\neg x_{1,0} \vee \neg x_{2,2}) \\ & (\neg x_{1,1} \vee \neg x_{2,0}) \wedge (\neg x_{1,1} \vee \neg x_{2,1}) \\ & (\neg x_{1,2} \vee \neg x_{2,0}) \wedge (\neg x_{1,2} \vee \neg x_{2,1}) \wedge (\neg x_{1,2} \vee \neg x_{2,2}) \end{aligned} \tag{5.10}$$

No pior caso, Φ_5 gera $(n^3 - n^2)$ cláusulas de tamanho 2.

Voltando a $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \Phi_4 \wedge \Phi_5$, podemos determinar que seu número de cláusulas no pior caso é:

$$n + \frac{n^3 - n^2}{2} + n + \frac{n^3 - n^2}{2} + (n^3 - n^2) = 2n^3 - 2n^2 + 2n$$

Dessas cláusulas, $2n$ são de tamanho n e o restante é de tamanho 2.

Cada modelo para a fórmula Φ representa unicamente um caminho hamiltoniano no grafo que a gerou. É fácil obter o caminho a partir do modelo, já que uma variável $x_{i,j}$ será verdadeira somente quando o vértice v_j estiver na posição i do caminho hamiltoniano. Apesar de Torn[15] não apresentar uma prova da corretude dessa redução, ela é sustentada pelos testes realizados no capítulo 7.

5.2 Redução de Iwama e Miyazaki

Esta seção descreve a redução de Iwama e Miyazaki[14]. A explicação de Hoos[16] também contribuiu para sua elaboração. Durante toda esta seção será usado como exemplo o grafo $G_z = (Z, E_z)$, $Z = \{z_0, z_1, z_2\}$ e $E_z = \{(z_0, z_1), (z_1, z_2)\}$, já representado na figura 5 na seção anterior.

Seja um grafo com vértices ordenados $G = (V, E)$ e $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Uma forma de representar um caminho hamiltoniano em G é com uma sequência de posições $H = p_0, p_1, \ldots, p_{n-1}$, com $0 \leq p_i < n$. Nessa sequência, p_i indica qual é a posição do vértice v_i no caminho hamiltoniano. Para o grafo de exemplo G_z, tem-se $H_z = 0, 1, 2$.

Cada p_i em H pode ser representado com um número binário b_i de $k = \lceil \log_2 n \rceil$ bits. Em G_z, $k_z = \lceil log_2 3 \rceil = 2$, portanto cada vértice pode ter sua posição codificada com 2 bits: $H_z = (00)_2, (01)_2, (10)_2$.

O objetivo da redução é construir uma fórmula booleana Φ satisfazível se e somente se G tem um caminho hamiltoniano. Para isso, cada vértice v_i de G terá $k = \lceil \log_2 n \rceil$ variáveis booleanas em Φ. Denotaremos essas variáveis por $v_{i,0}, v_{i,1}, \ldots, v_{i,k-1}$. A idéia é que essas variáveis codifiquem a posição do vértice v_i num caminho hamiltoniano e que somente nesse caso Φ seja verdadeiro. No total, terá-se $n\lceil log_2 n \rceil$ variáveis em Φ.

Para o grafo G_z, as variáveis serão $z_{0,0}, z_{0,1}, z_{1,0}, z_{1,1}, z_{2,0}$ e $z_{2,1}$. Portanto, deseja-se que a única atribuição válida de valores para essas variáveis seja $0, 0, 0, 1, 1$ e 0, respectivamente.

O próximo passo da redução é criar cláusulas que restrinjam a atribuição de valores para as variáveis, de forma que um modelo de Φ codifique um caminho hamiltoniano válido.

Antes disso, será útil denotarmos por $C(i, p)$ uma cláusula que é falsa se e somente se as variáveis do vértice v_i codificam a posição p em binário. Isto é, dado $(b)_2 = (p)_{10}$ e b_j denotando a posição com o j-ésimo bit mais significativo de b, desejamos que $(\forall j)(v_{i,j} = b_j) \iff \neg C(i, p)$.

Por tabela verdade, $C(i, p) = \neg(a(i, 0, b_0) \wedge a(i, 1, b_1) \wedge \cdots \wedge a(i, k-1, b_{k-1}))$, com $(b)_2 = (p)_{10}$. Definimos a função a como:

$$a(i, j, b_j) = \begin{cases} v_{i,j} & \text{se } b_j = 1 \\ \neg v_{i,j} & \text{se } b_j = 0 \end{cases}$$

Em termos simples, b_j determina se a variável $v_{i,j}$ aparece negada na cláusula $C(i,p)$. Voltando a G_z, teria-se $C(0,2) = \neg(z_{0,0} \wedge \neg z_{0,1})$, com $b = 10$.

Agora serão apresentadas as cláusulas de Φ. Por clareza, separa-se Φ em três fórmulas:

$$\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$$

Cada uma delas irá tornar Φ falso para um conjunto específico de modelos inválidos.

A fórmula Φ_1 irá codificar cláusulas que invalidam a atribuição de valores maiores ou iguais a n para uma posição. Isso é necessário para casos em que n não é potência de 2, pois sobrarão números para codificar posições inválidas. Ela será uma conjunção de todas as $C(i,j)$, para todo $0 \leq i < n$ e $n \leq j \leq 2^k - 1$. Isto é:

$$\Phi_1 = \bigwedge_{i=0}^{n-1} \bigwedge_{j=n}^{2^k-1} C(i,j) \tag{5.11}$$

Será útil para os algoritmos utilizados na implementação que Φ esteja em CNF. Por isso, usaremos $C(i,p) = (\neg a(i,0,b_0) \vee \neg a(i,1,b_1) \vee \cdots \vee \neg a(i,k-1,b_{k-1}))$, equivalente ao apresentado anteriormente, apesar de um pouco menos intuitivo.

Uma atribuição de um valor maior que $n-1$ às variáveis de um vértice tornará Φ_1 insatisfazível. Por consequência, Φ será insatisfazível. No pior caso, sobrarão $n-2$ atribuições e Φ_1 terá $n^2 - 2n$ cláusulas de tamanho k.

Voltando ao exemplo G_z: com $k = 2$ podemos codificar quatro posições: $(00)_2$, $(01)_2$, $(10)_2$ e $(11)_2$. Dessas posições, a posição $(11)_2$ é inválida e deve-se invalidar sua atribuição a um vértice:

$$\begin{aligned}\Phi_{z_1} &= \neg(z_{0,0} \wedge z_{0,1}) \wedge \neg(z_{1,0} \wedge z_{1,1}) \wedge \neg(z_{2,0} \wedge z_{2,1}) \\ &= (\neg z_{0,0} \vee \neg z_{0,1}) \wedge (\neg z_{1,0} \vee \neg z_{1,1}) \wedge (\neg z_{2,0} \vee \neg z_{2,1})\end{aligned}$$

A fórmula Φ_2 irá codificar cláusulas que invalidam a atribuição de dois vértices diferentes para a mesma posição. Será uma conjunção de todas as $C(i,j) \vee C(l,j)$, para todo $0 \leq i < l < n$ e para todo $0 \leq j < n$. Isto é:

$$\Phi_2 = \bigwedge_{i=0}^{n-1} \bigwedge_{l=i+1}^{n-1} \bigwedge_{j=0}^{n-1} C(i,j) \vee C(l,j) \tag{5.12}$$

Tendo em vista que $(p \rightarrow \neg q) \iff (\neg p \vee \neg q)$, a fórmula expressa que se uma posição j está atribuída ao v_i a mesma posição não estará atribuída a v_l. É importante também notar a estrutura das cláusulas:

$$\begin{aligned} C(i,j) \vee C(l,j) \iff & (\neg a(i,0,b_0) \vee \neg a(i,1,b_1) \vee \cdots \vee \neg a(i,k-1,b_{k-1}) \\ & \vee \neg a(l,0,b_0) \vee \neg a(l,1,b_1) \vee \cdots \vee \neg a(l,k-1,b_{k-1})) \end{aligned} \tag{5.13}$$

Para todos os vértices, para todos os vértices subsequentes e para todas as posições é criada uma cláusula envolvendo todas as variáveis de um par de vértices. Portanto, Φ_2 tem $\frac{n^3-n^2}{2}$ cláusulas de tamanho $2k$.

Para o exemplo G_z temos:

$$\begin{aligned} \Phi_{z_2} = & \neg((\neg z_{0,0} \wedge \neg z_{0,1}) \wedge (\neg z_{1,0} \wedge \neg z_{1,1})) \wedge \neg((\neg z_{0,0} \wedge \neg z_{0,1}) \wedge (\neg z_{2,0} \wedge \neg z_{2,1})) \wedge \\ & \neg((\neg z_{1,0} \wedge \neg z_{1,1}) \wedge (\neg z_{2,0} \wedge \neg z_{2,1})) \\ & \wedge \\ & \neg((\neg z_{0,0} \wedge z_{0,1}) \wedge (\neg z_{1,0} \wedge z_{1,1})) \wedge \neg((\neg z_{0,0} \wedge z_{0,1}) \wedge (\neg z_{2,0} \wedge z_{2,1})) \wedge \\ & \neg((\neg z_{1,0} \wedge z_{1,1}) \wedge (\neg z_{2,0} \wedge z_{2,1})) \\ & \wedge \\ & \neg((z_{0,0} \wedge \neg z_{0,1}) \wedge (z_{1,0} \wedge \neg z_{1,1})) \wedge \neg((z_{0,0} \wedge \neg z_{0,1}) \wedge (z_{2,0} \wedge \neg z_{2,1})) \wedge \\ & \neg((z_{1,0} \wedge \neg z_{1,1}) \wedge (z_{2,0} \wedge \neg z_{2,1})) \end{aligned}$$

A fórmula Φ_3 irá codificar cláusulas que invalidam a atribuição de v_i para a posição anterior a v_j se $(v_i, v_j) \notin E$. Essa é a única fórmula dependente das arestas.

Define-se Φ_3 como a conjunção de $C(i,j) \vee C(l,j+1)$ para todo $0 \leq i,l < n$ e $0 \leq j < n-1$ tal que $(v_i, v_l) \notin E$. Ou seja:

$$\Phi_2 = \bigwedge_{i=0}^{n-1} \bigwedge_{l=0}^{n-1} \bigwedge_{j=0}^{n-2} f(C(i,j), C(l,j+1)) \tag{5.14}$$

$$f(C(i,j),C(l,j+1)) = \begin{cases} 1 & \text{se } (v_i, v_l) \in E \\ C(i,j) \vee C(l,j+1) & \text{se } (v_i, v_l) \notin E \end{cases}$$

A função f apenas serve para contornar uma limitação da notação. Em termos simples, Φ_3 garante que se não existe uma aresta ligando v_i a v_j, v_i não pode preceder v_j no caminho.

Em um grafo completo tem-se $\Phi_3 = 1$. O pior caso para o número de cláusulas acontece num grafo sem arestas, onde Φ_3 tem $n^3 - n^2$ cláusulas de tamanho $2k$.

No exemplo tem-se:

$$\begin{aligned}
\Phi_{z_3} =& \neg((\neg z_{0,0} \wedge \neg z_{0,1}) \wedge (\neg z_{0,0} \wedge z_{0,1})) \wedge \neg((\neg z_{0,0} \wedge z_{0,1}) \wedge (z_{0,0} \wedge \neg z_{0,1})) \wedge \\
& \neg((\neg z_{0,0} \wedge \neg z_{0,1}) \wedge (\neg z_{2,0} \wedge z_{2,1})) \wedge \neg((\neg z_{0,0} \wedge z_{0,1}) \wedge (z_{2,0} \wedge \neg z_{2,1})) \\
& \wedge \\
& \neg((\neg z_{1,0} \wedge \neg z_{1,1}) \wedge (\neg z_{0,0} \wedge z_{0,1})) \wedge \neg((\neg z_{1,0} \wedge z_{1,1}) \wedge (z_{0,0} \wedge \neg z_{0,1})) \wedge \\
& \neg((\neg z_{1,0} \wedge \neg z_{1,1}) \wedge (\neg z_{1,0} \wedge z_{1,1})) \wedge \neg((\neg z_{1,0} \wedge z_{1,1}) \wedge (z_{1,0} \wedge \neg z_{1,1})) \\
& \wedge \\
& \neg((\neg z_{2,0} \wedge \neg z_{2,1}) \wedge (\neg z_{0,0} \wedge z_{0,1})) \wedge \neg((\neg z_{2,0} \wedge z_{2,1}) \wedge (z_{0,0} \wedge \neg z_{0,1})) \wedge \\
& \neg((\neg z_{2,0} \wedge \neg z_{2,1}) \wedge (\neg z_{1,0} \wedge z_{1,1})) \wedge \neg((\neg z_{2,0} \wedge z_{2,1}) \wedge (z_{1,0} \wedge \neg z_{1,1})) \wedge \\
& \neg((\neg z_{2,0} \wedge \neg z_{2,1}) \wedge (\neg z_{2,0} \wedge z_{2,1})) \wedge \neg((\neg z_{2,0} \wedge z_{2,1}) \wedge (z_{2,0} \wedge \neg z_{2,1}))
\end{aligned}$$

Voltando a $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3$, o número de cláusulas na redução é, no pior caso:

$$(n^2 - 2n) + (\frac{n^3 - n^2}{2}) + (n^3 - n^2) = \frac{3n^3 - n^2 - 4n}{2}$$

Dessas cláusulas, $n^2 - 2n$ são de tamanho k e o restante é de tamanho $2k$. O número de cláusulas e tamanho das cláusulas apresentados foram calculados pelo autor deste trabalho.

Segundo Iwama e Miyazaki[14] e sustentado pelos testes realizados no capítulo 7, as cláusulas apresentadas são suficientes para que Φ seja satisfazível se e somente se G tem um caminho hamiltoniano. Além disso, qualquer modelo que torna Φ verdadeiro pode ser decodificado para determinar uma sequência de vértices que é um caminho hamiltoniano. A prova de corretude dessa redução foi omitida por Iwama e Miyazaki[14]

e não foi encontrada em outras fontes. A elaboração da prova foi considerada fora do escopo desse trabalho.

A implementação das duas reduções e as aplicações escolhidas para determinar a satisfazibilidade das fórmulas geradas são apresentadas no próximo capítulo.

6 Implementação das Reduções

Este capítulo detalha a implementação das reduções apresentadas no capítulo anterior. O objetivo dessa implementação é permitir uma investigação do desempenho de aplicações já bem estabelecidas para determinação de satisfazibilidade, que implementam algoritmos semelhantes aos expostos na seção 3.4, com fórmulas provindas das reduções.

As referidas aplicações de satisfazibilidade, das quais depende a arquitetura da implementação, também são apresentadas neste capítulo.

6.1 Aplicações para Determinação de Satisfazibilidade

As aplicações consideradas para determinação de satisfazibilidade participam da competição Sat-Race [17]. Dezenas de aplicações participam dessa competição [17]. Ela tem o objetivo de determinar as aplicações com melhor desempenho em três categorias: solução de fórmulas decorrentes de problemas reais, solução de fórmulas construídas e solução de fórmulas geradas aleatoriamente [17]. Além dessas categorias, existem três outras: aplicações com melhor desempenho para determinar satisfazibilidade, para determinar instatisfazibilidade e para ambos.

Foram escolhidas três aplicações com base em seus resultados na competição:

- A aplicação *precosat* [1] foi a aplicação com melhor desempenho na categoria satisfazibilidade e insatisfazibilidade para fórmulas de problemas reais na Sat-Race 2009 [17]. Sua implementação tem um núcleo em comum com uma aplicação de sucesso em versões anteriores da competição chamada *MiniSat* [18]. A aplicação *MiniSat*, por sua vez, implementa o algoritmo DPLL com algumas características adicionais, como aprendizado de cláusulas guiada por conflitos, reinícios dinâmicos de backtracking e outras [19]. O detalhamento das estruturas de dados e heurísticas adicionais usadas para atingir um bom desempenho pode ser encontrado em [19] e [18].

[1] Website: http://fmv.jku.at/precosat/.

- A aplicação *clasp* [2] foi a aplicação com melhor desempenho na categoria satisfazibilidade e insatisfazibilidade para fórmulas construídas especialmente para testar aplicações de satisfazibilidade. Essa aplicação usa uma combinação de técnicas de satisfação de restrições, técnicas empregadas em aplicações de sucesso como Minisat e SatElite e outras técnicas detalhadas em [18]. O trabalho de Russell e Norvig[6] oferece uma introdução àlgumas técnicas de satisfação de restrições.

- A aplicação *gnovelty+* foi a aplicação com segundo melhor desempenho na categoria de satisfazibilidade de fórmulas aleatórias. Implementa uma versão modificada do algoritmo WalkSAT, servindo como um exemplo de resolvedor de busca estocástica local. A aplicação de melhor desempenho na categoria, SatZilla, consiste em uma aplicação capaz de escolher dinâmicamente a melhor aplicação para resolver uma fórmula.

Mais detalhes sobre essas aplicações podem ser encontrados em [18].

O uso das aplicações da Sat-Race facilita a implementação das reduções. Isso porque todas as aplicações adotam o mesmo formato de entrada de fórmulas e de saída de modelos chamado *DIMACS* [20].

No padrão DIMACS, variáveis são representadas por números inteiros positivos. Uma cláusula é representada como uma lista de números inteiros: um número negativo na lista representa que a variável do seu módulo aparece negada na cláusula. Cada cláusula é declarada em uma linha terminada em 0. Precedendo as cláusulas no arquivo está uma linha declarando o número de variáveis e o número cláusulas.

A figura 6 apresenta um fragmento do arquivo gerado com a redução de Iwama e Miyazaki[14] para o grafo usado como exemplo no capítulo anterior [3].

```
p cnf 6 20
-1 -2 0
-3 -4 0
-5 -6 0
...
5 -3 -6 4 0
```

Figura 6: Fragmento de um arquivo DIMACS gerado para o grafo G.

[2] Website: http://www.cs.uni-potsdam.de/clasp/.
[3] Foram omitidas 16 cláusulas.

A saída das aplicações de satisfazibilidade, no caso de uma fórmula satisfazível é uma lista de números inteiros: uma variável com atribuição 0 aparece com sinal negativo, uma variável com atribuição 1 aparece com sinal positivo. Para o exemplo, teria-se a saída: "-1 -2 -3 4 5 -6".

6.2 Arquitetura da Implementação

A implementação foi separada em duas aplicações: *hampath_ encoder* e *hampath_ decoder*. Ambas reduções apresentadas no capitulo 5 foram implementadas e são escolhidas através de um parâmetro.

A aplicação *hampath_ encoder* recebe um arquivo que descreve um grafo e gera um arquivo de cláusulas. O arquivo de cláusulas segue o padrão *DIMACS* detalhado na seção anterior.

A aplicação *hampath_ decoder* é usada apenas no caso em que a fórmula é satisfazível. Recebe um modelo para as cláusulas em formato DIMACS e o grafo que as gerou. Sua saída é uma representação legível do caminho.

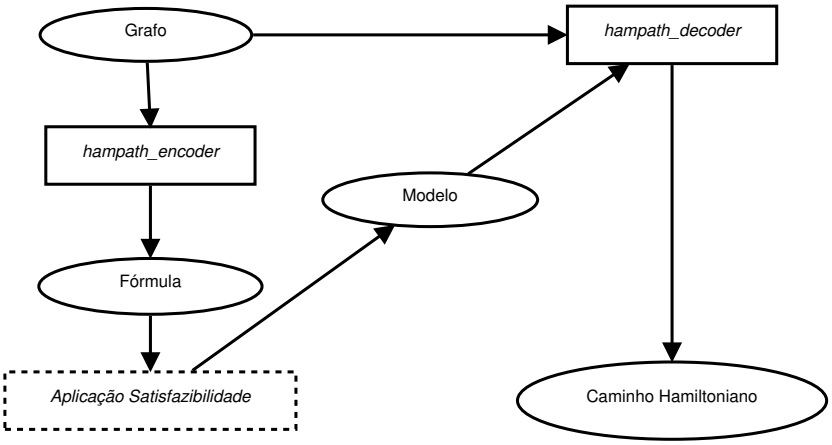

Figura 7: Arquitetura da Implementação.

A linguagem de programação escolhida para a implementação foi *C++*, também em uso por várias outras aplicações de satisfazibilidade como precosat, clasp e gnovelty+ [18]. No futuro, isso facilitará o uso dessas aplicações como bibliotecas, eliminando a necessidade dos arquivos intermediários.

Os grafos são declarados em arquivos XML. Os arquivos são lidos com a biblioteca TinyXml [21]. Como exemplo, seja $G = (V, E)$, $V = \{a, b, c\}$, $E = \{(a, b), (b, c)\}$. Sua declaração seria como apresentado na figura 8. A declaração dos vértices é opcional.

```
<?xml version="1.0" ?>
<graph>
        <vertex name="a" />
        <vertex name="b" />
        <vertex name="c" />

        <edge from="a" to="b" />
        <edge from="b" to="c" />
</graph>
```

Figura 8: Declaração em XML do grafo G.

A saída gerada pelo *hampath_ decoder* para esse grafo seria "a -> b -> c".

6.3 Projeto das Aplicações

As aplicações são implementadas usando orientação a objetos. A figura 9 apresenta os principais métodos das classes criadas.

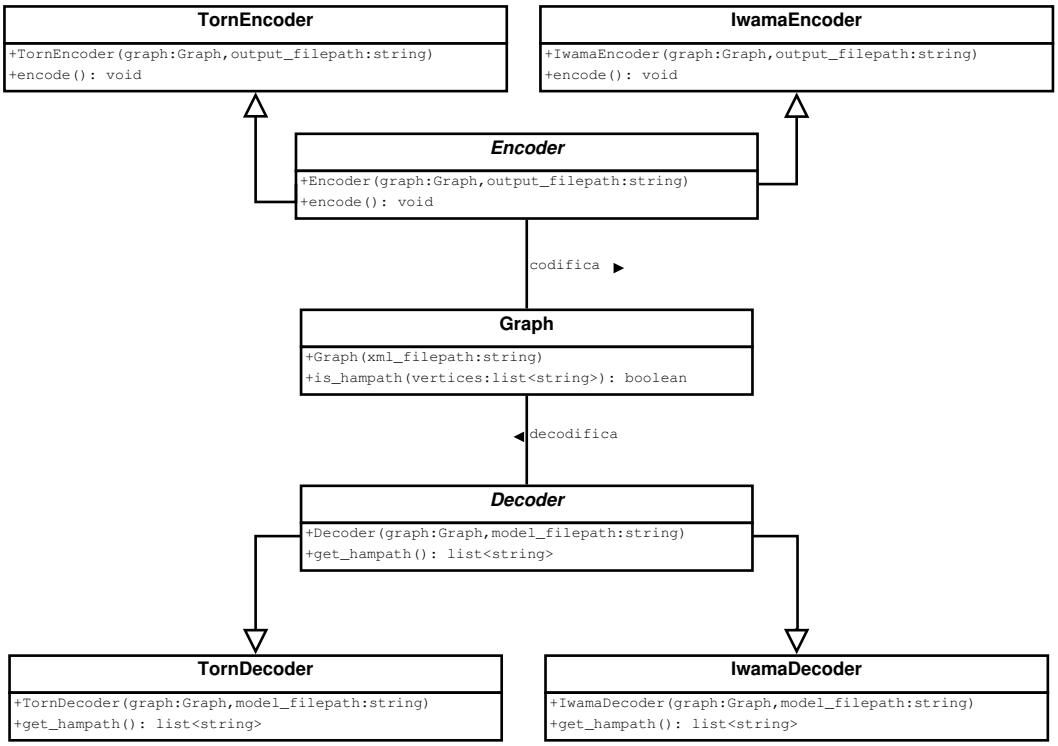

Figura 9: Diagrama de Classes simplificado.

A aplicação *hampath_ encoder* cria um objeto *Graph* a partir de um arquivo. A classe abstrata Encoder é herdada pelas classes IwamaEncoder e TornEncoder, que implementam a redução de Torn[15] e Iwama e Miyazaki[14], respectivamente. O Encoder escolhido

através de um parâmetro para a aplicação é criado a partir do objeto da classe *Graph* e do caminho para o arquivo de saída. O método *encode* de ambas as classes é usado para criar as cláusulas, conforme descrito no capítulo 5, no formato DIMACS.

Assim como a aplicação anterior, *hampath_ decoder* cria um objeto *Graph* a partir de um arquivo. Esse objeto é usado por uma das subclasses de *Decoder*, que também são escolhidas através de um parâmetro. O *Decoder* escolhido também recebe como parâmetro o caminho para um arquivo com um modelo em DIMACS. O método *get_ hampath* decodifica o caminho hamiltoniano a partir do modelo, conforme descrito no capítulo 5.

O método *is_ hampath* da classe *Graph* verifica a validade do caminho encontrado. É executado por padrão pelo *hampath_ decoder*, assegurando a corretude dos testes realizados e apresentados no capítulo 7.

O código fonte das aplicações e um arquivo com detalhes de execução está disponível no apêndice B.

O próximo capítulo descreve os resultados dos testes realizados com as aplicações desenvolvidas e com as aplicações de determinação de satisfazibilidade consideradas.

7 Resultados

Com o objetivo de conhecer o desempenho das aplicações desenvolvidas e da solução das fórmulas através das aplicações de satisfazibilidade apresentadas no capítulo anterior, foram realizados testes com dois tipos de grafos: grafos aleatórios e grafos do problema do passeio do cavalo. Os resultados desses testes são apresentados neste capítulo.

Por ser recente e apresentar estatísticas de desempenho, a análise da implementação de um algoritmo de determinação de caminhos hamiltonianos realizada por Narayanasamy[22] é usada como base para comparação com os resultados obtidos, apesar de não representar a melhor implementação disponível na literatura. O algoritmo em questão consiste fundamentalmente numa expansão baseada em agregação de circuitos parciais [22], portanto semelhante àlgumas heurísticas apresentadas no capítulo 2.

Vandegriend[7] disponibiliza o código-fonte de uma aplicação de determinação de caminhos hamiltonianos que implementa várias das heurísticas do capítulo 2. Um estudo dessa aplicação com o objetivo de enriquecer este capítulo será importante para trabalhos futuros.

Os testes apresentados neste capítulo foram realizados em um notebook com processador AMD Athlon(tm) X2 Dual-Core QL-64 de 2.1 GHz, 2 GB de memória RAM e sistema operacional baseado em GNU/Linux 3.0.

7.1 Grafos Aleatórios

Grafos aleatórios foram escolhidos para os testes por serem de fácil criação e usados para testar técnicas diretas de determinação de caminhos hamiltonianos por outros autores [7, 16, 22]. Vandegriend[7] cita propriedades interessantes desses grafos.

Um grafo aleatório $G = (V, E)$ é gerado como segue: dados o número de vértices desejado e uma probabilidade $p \in [0, 1]$, um número $x_i \in [0, 1]$ é sorteado para cada par de vértices $e_i \in V \times V$. Se e somente se $x_i \geq p$ terá-se $e_i \in E$. Grosso modo, p é a razão

esperada entre o número de arestas a partir de um vértice qualquer e o número total de vértices.

Diferentemente de Vandegriend[7] e Narayanasamy[22] e de acordo com Hoos[16], nos grafos aleatórios usados neste trabalho também é inserido um caminho passando por uma sequência de vértices aleatória, sem repetição e que contém todos os vértices. Isso garante a existência de um caminho hamiltoniano, mesmo dado $p = 0$.

Alguns grafos sem caminho hamiltoniano foram testados com sucesso, mas os testes não foram considerados abrangentes o suficiente para serem apresentados neste texto. Testes com diferentes tipos de grafos sem caminhos hamiltonianos serão considerados em trabalhos futuros.

A criação das fórmulas a partir de grafos tem seu pior caso, para ambas as reduções, com grafos gerados com $p = 0$. Isso acontece porque as cláusulas são usadas para restringir adjacências entre vértices. A tabela 8 apresenta o tempo de criação e tamanho em disco ocupado pelo arquivo DIMACS de fórmulas geradas por ambas as reduções para grafos aleatóritos com diferentes números de vértices e $p = 0$.

	Redução de Iwama e Miyazaki[14]		Redução de Torn[15]	
Vértices	Tempo (s)	Espaço (MB)	Tempo (s)	Espaço (MB)
50	1.5	9	1.08	3
100	15.755	89	8.08	27
150	55.827	359	28	97
200	145.435	898	69.609	235
250	260.013	1861	133.187	465

Tabela 8: Tempo de criação e espaço em disco ocupado por fórmulas.

A tabela 8 deixa claro que a redução de Torn[15] implementada tem desempenho muito melhor na codificação de fórmulas para o pior caso. Apesar do número de cláusulas nas fórmulas de Torn[15] ser maior, como pode ser observado no capítulo 5, a maioria das cláusulas tem tamanho 2 e, por consequência, o tamanho total do arquivo em DIMACS é menor.

As tabelas a seguir apresentam parte dos resultados dos testes com as aplicações apresentados na seção 6.1 para ambas as reduções e diferentes números de vértices.

Os tempos de solução e número de cláusulas apresentados são a média desses valores para execuções com três grafos diferentes. Esses tempos não incluem o tempo de geração das cláusulas. Os campos marcados com asterisco são de execuções que excederam cinco minutos e foram abortadas. Dois asteriscos indicam que a execução foi interrompida por

esgotar a memória RAM. Os resultados completos podem ser encontrados no apêndice A.

P=0.25						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.022	1062	40	0.005	1577	100
15	0.07	3657	60	0.015	5476	225
25	1.036	18027	125	0.095	25850	625
50	82.556	150101	300	1.678	213495	2500
100	*	1224394	700	33.008	1729829	10000
150	*	4177768	1200	8.746	5857043	22500
200	*	9902760	1600	53.384	13897963	40000
250	*	19360420	2000	58.301	27225413	62500

Tabela 9: Resultados para clasp com p = 0.25

Observa-se na tabela 9 que a aplicação clasp teve desempenho muito melhor com a redução de Torn, apesar do número maior de variáveis e cláusulas dessa redução. Esses resultados se repetem nos testes com outros valores de p. O consumo de memória RAM pelo clasp manteve-se sempre abaixo de 300 MB para todos os testes, mesmo para arquivos de cláusulas com mais de 800MB.

É possível notar uma anomalia entre 100 e 150 vértices para a aplicação clasp com a redução de Torn[15]: o tempo de solução é maior para 100 vértices do que para 150. Essa tendência também se confirmou em testes com diferentes parâmetros apresentados no apêndice A. Um estudo mais profundo da aplicação seria necessário para determinar a causa desse comportamento, já que nenhuma das outras duas exibe essa anomalia.

A aplicação precosat exibiu um comportamento de tempo bastante semelhante ao clasp (exceto no caso da anomalia citada anteriormente), apesar de ter um desempenho inferior. Ela alcança o limite de memória (próximo a 2GB) com 250 vértices para $p = 0.25$.

A aplicação gnovelty+ teve, no geral, desempenho inferior às outras duas com a redução de Torn. A partir de 100 vértices o uso de memória ultrapassou o limite. No entanto, por ser baseada no algoritmo de busca local estocástica WalkSAT, teve um desempenho melhor do que as outras com a redução de Iwama e Miyazaki. O número menor de variáveis dessa redução permite que uma atribuição seja encontrada com maior facilidade.

Com um objetivo similar ao deste trabalho, Hoos[16] analisa a dificuldade de determinar a satisfazibilidade de fórmulas geradas com a redução de Iwama e Miyazaki[14] com o algoritmo de busca estocástica local gsat. Mediante testes, sua conclusão é que os

problemas gerados pela redução de Iwama e Miyazaki[14] são especialmente difíceis para a aplicação que considera.

P=0.25						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.018	1062	40	0.107	1577	100
15	0.02	3657	60	0.022	5476	225
25	0.233	18027	125	0.106	25850	625
50	1.021	150101	300	1.712	213495	2500
100	17.066	1224394	700	39.787	1729829	10000
150	100.752	4177768	1200	**	5857043	22500
200	*	9902760	1600	**	13897963	40000
250	*	19360420	2000	**	27225413	62500

Tabela 10: Resultados para gnovelty+ com p = 0.25

Observa-se que as aplicações tem mais facilidade para resolver o problema em grafos com mais arestas. A tabela 11 é um exemplo deste comportamento, que pode ser observado em detalhe no apêndice A. Para grafos aleatórios, o número de arestas é claramente mais importante do que o número total cláusulas para o tempo de solução, como pode ser observado comparando-se as tabelas da aplicação clasp para $p = 0.25$ e $p = 0.75$.

P=0.75						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.012	702	40	0.007	1154	100
15	0.065	2281	60	0.012	4006	225
25	0.393	11059	125	0.062	19202	625
50	24.653	91824	300	0.832	154989	2500
100	*	738964	700	14.922	1248689	10000
150	*	2519249	1200	8.395	4204782	22500
200	*	5959973	1600	32.71	9964529	40000
250	*	11645985	2000	47.662	19497449	62500

Tabela 11: Resultados para clasp com p = 0.75

A aplicação para determinação de caminhos hamiltonianos de Narayanasamy[22], em comparação, é capaz de resolver casos difíceis (detalhados no seu próprio trabalho) de grafos aleatórios com até 800 vértices em 10 minutos usando uma máquina similar.

A próxima seção apresenta testes com outros tipos de grafos que confirmam o melhor desempenho das técnicas diretas.

7.2 Grafos do Problema do Passeio do Cavalo

Os grafos do problema do passeio do cavalo são baseados no problema de determinar se um cavalo pode atingir todas as posições de um tabuleiro de xadrez vazio sem passar mais de uma vez por cada posição. Cada vértice no grafo representa uma posição de um tabuleiro de n colunas e m linhas. Se a partir de uma posição x um cavalo puder, em um tabuleiro vazio, alcançar diretamente a posição y, haverá uma aresta entre os vértices x e y.

Esse tipo de grafo também é usado para testes em outras aplicações de determinação de caminho hamiltoniano [7, 9, 22]. Apesar desse problema para tabuleiros quadrados ter técnicas específicas eficientes [23, 24], ele gera instâncias de difícil determinação de caminhos hamiltonianos [7, 9, 22]. Vandegriend[7] faz uma análise bastante abrangente da dificuldade de resolver esse e vários outros tipos de grafos.

Para os testes, cada aplicação foi executada com fórmulas geradas a partir de grafos representando tabuleiros de 4×4 até 8×8. As tabelas a seguir mostram os resultados obtidos. Os asteriscos nas tabelas a seguir tem o mesmo significado que os apresentados na seção anterior. Três asteriscos indicam que uma aplicação não poderia terminar pois não existe um caminho hamiltoniano.

A aplicação clasp novamente se comportou de forma semelhante à aplicação precosat, mas com desempenho superior e menor uso de memória. Seu desempenho superou também a aplicação gnovelty+.

Aplicação: clasp. Problema do passeio do cavalo.						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
16 (4×4)	1.396	4800	64	0.117	6992	256
25 (5×5)	85.837	19771	125	0.3	27746	625
36 (6×6)	*	62188	216	2.387	85192	1296
49 (7×7)	*	159735	294	13.565	219074	2401
64 (8×8)	*	361872	384	124.523	495056	4096

Tabela 12: Resultados para clasp com grafos do problema do cavalo

A aplicação gnovelty+ desta vez teve desempenho melhor com a redução de Torn[15] para essa classe de grafos, ao contrário dos resultados com grafos aleatórios. O pequeno número de variáveis das instâncias testadas claramente favoreceu a estrutura das cláusulas de Torn[15].

Aplicação:gnovelty+. Problema do passeio do cavalo.						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
16 (4×4)	***	4800	64	***	6992	256
25 (5×5)	65.384	19771	125	1.014	27746	625
36 (6×6)	*	62188	216	57.893	85192	1296
49 (7×7)	*	159735	294	*	219074	2401
64 (8×8)	*	361872	384	*	495056	4096

Tabela 13: Resultados para gnovelty+ com grafos do problema do cavalo

Confirmando a dificuldade do problema afirmada por Vandegriend[7], Chalaturnyk[9] e Narayanasamy[22], instâncias de grafos do passeio do cavalo levam muito mais tempo para serem solucionadas do que instâncias de grafos gerados aleatoriamente com muito mais vértices.

A aplicação de Narayanasamy[22] de determinação de caminhos hamiltonianos, já citada na seção anterior, é capaz de resolver o problema para tabuleiros 20×20 em menos de dez minutos.

O rápido crescimento tanto do tamanho em disco ocupado pelas fórmulas quanto do tempo de solução em função do número de vértices deixa clara a superioridade em desempenho das técnicas diretas para determinação de caminhos hamiltonianos.

No entanto, alcançando os objetivos apresentados para este trabalho, os testes realizados fornecem uma noção geral do desempenho que pode ser esperado da técnica implementada. Como adicional, os resultados encontrados também podem auxiliar na consideração de reduções como intermediárias na solução de outros problemas NP-completos.

8 Conclusão

Vários problemas computacionais reais requerem muito tempo de processamento ou espaço de armazenamento para serem solucionados. O estudo da complexidade computacional, parte da teoria da computação, permite uma classificação matemática da dificuldade dos problemas. Vários problemas importantes são encontrados na classe de problemas NP-completos.

Dois problemas NP-completos clássicos foram detalhados neste trabalho: o problema do caminho hamiltoniano e problema da satisfazibilidade booleana. O objetivo do trabalho foi o estudo da solução do problema hamiltoniano através de redução (transformação) para problema de satisfazibilidade booleana. Duas reduções foram encontradas para essa tarefa: a redução de Torn[15] e a redução de Iwama e Miyazaki[14]. As reduções geram, a partir de um grafo, fórmulas booleanas que são satisfazíveis se e somente se o grafo possui um caminho hamiltoniano. Não se espera encontrar na literatura uma redução mais eficiente que essas, apesar deste trabalho não afirmar que elas não existam. Uma propriedade interessante dessas reduções é que restrições podem ser facilmente adicionadas ou removidas das fórmulas geradas. Isso pode ser útil para resolver variações do problema do caminho hamiltoniano.

A implementação dessas reduções permitiu que testes fossem realizados com aplicações de determinação de satisfazibilidade, com o objetivo de investigar o desempenho da abordagem apresentada. Essa implementação recebe como entrada um grafo e transforma-o em uma fórmula lógica no formato DIMACS (padrão de entrada para diversas aplicações de determinação de satisfazibilidade). Várias aplicações de satisfazibilidade podem ser usadas para determinar a satisfazibilidade da fórmula gerada. Neste trabalho, três aplicações representando diferentes algoritmos foram escolhidas para testes: *precosat*, *clasp* e *gnovelty+*. A escolha foi baseada no desempenho das aplicações na SAT-Race (competição entre aplicações de satisfazibilidade).

Dois tipos de grafos foram usados para testes: grafos aleatórios e grafos do problema do

problema do cavalo. Esses grafos também são considerados para testes em vários trabalhos sobre determinação direta (que não envolve redução) de caminhos hamiltonianos.

Realizados testes com os diferentes grafos, reduções e aplicações, o desempenho da solução através de redução foi considerado bastante inferior ao de técnicas diretas de determinação de caminhos hamiltonianos (representadas pelo trabalho de Narayanasamy[22]). No entanto, alcançar um desempenho equivalente ao das técnicas que exploram as especificidades do problema não era o objetivo principal do trabalho.

A aplicação *clasp* teve, no geral, o melhor desempenho, sendo capaz de resolver fórmulas geradas pela redução de Torn[15] para grafos aleatórios com 250 vértices e grafos do problema do cavalo com 64 vértices em pouco menos de cinco minutos.

A grande diversidade estrutural entre grafos torna difícil generalizar com segurança os resultados encontrados, já que o desempenho precisa ser empiricamente determinado para tipos individuais de grafos. Levando isso em consideração, testes com mais tipos de grafos seriam interessantes para trabalhos futuros, especialmente testes com grafos sem caminho hamiltoniano. A análise da técnica implementada também seria enriquecida através de uma comparação mais abrangente com outros algoritmos para determinação de caminhos hamiltonianos.

Uma investigação com o objetivo de encontrar a causa da dificuldade de solução das fórmulas pelos algoritmos de determinação de satisfazibilidade poderia esclarecer os resultados encontrados e auxiliar na busca por problemas que teriam solução eficiente através de redução para problema de satisfazibilidade.

Em um trabalho semelhante a este, Hoos[16] investigou o desempenho da solução do caminho hamiltoniano através da redução de Iwama e Miyazaki[14] com o algoritmo de satisfazibilidade *GSAT*. Seu trabalho conclui que as fórmulas resultantes da redução são especialmente difíceis para a aplicação que considera. O problema do planejamento [25] e o problema do *k-clique* [26] são outros exemplos de problemas NP-completos cujo desempenho da solução através de redução para problema de satisfazibilidade foi estudada.

Por fim, a idéia geral de reduzir um problema NP-completo a outro pode ser útil em outros cenários, devido ao imenso número de problemas nessa classe e a grande variedade de técnicas para resolvê-los. O estudo realizado neste trabalho contribui empiricamente para pesquisas nessa direção.

Referências

1 WEST, D. B. *Introduction To Graph Theory*. [S.l.]: Prentice-Hall, 1996.

2 GONDRAN, M.; MINOUX, M.; VAJDA, S. *Graphs and Algorithms*. [S.l.]: John Wiley & Sons, 1984.

3 SIPSER, M. *Introduction to the Theory of Computation*. 2. ed. [S.l.]: Course Technology, 2006.

4 LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elementos de teoria da Computação*. 2. ed. [S.l.]: Bookman, 2004.

5 BIERE, A. et al. (Ed.). *Handbook of Satisfiability*. [S.l.]: IOS Press, 2009. 980 p. (Frontiers in Artificial Intelligence and Applications, v. 185). ISSN 0922-6389. ISBN 978-1-58603-929-5.

6 RUSSELL, S. J.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 2. ed. [S.l.]: Prentice-Hall, 2003.

7 VANDEGRIEND, B. *Finding Hamiltonian Cycles: Algorithms, Graphs and Performance*. 1998. URL: http://webdocs.cs.ualberta.ca/ joe/Theses/vandegriend.ps. Acessado em 16 de agosto de 2011.

8 RUBIN, F. A search procedure for hamilton paths and circuits. *Journal of the ACM*, v. 21, 1974.

9 CHALATURNYK, A. *A Fast Algorithm For Finding Hamilton Cycles*. 2008. URL: ftp://www.combinatorialmath.ca/g&g/chalaturnykthesis.pdf. Acessado em 18 de julho de 2011.

10 ROSEN, K. H. et al. *Handbook of Discrete and Combinatorial Mathematics*. [S.l.]: CRC Press, 2000.

11 BESSIÈRE, C. *Principles and practice of constraint programming–CP 2007: 13th international conference, CP 2007, Providence, RI, USA, September 23-27, 2007 : proceedings*. [S.l.]: Springer, 2007. (Lecture notes in computer science). ISBN 9783540749691.

12 PAPADIMITRIOU, C. *Computational Complexity*. 2. ed. [S.l.]: Addison-Wesley, 1994.

13 GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN 0716710455.

14 IWAMA, K.; MIYAZAKI, S. Sat-variable complexity of hard combinatorial problems. *In Proceedings of the World Computer Congress of the IFIP*, Elsevier Science B.V., v. 1, p. 253–258, 1994.

15 TORN, E. *Example Reductions*. URL: http://www.cse.msu.edu/ torng/Classes/Archives/cse830.03fall/Lectures/example-reductions.pdf. Acessado em 26 de março de 2011.

16 HOOS, H. H. Solving hard combinatorial problems with gsat, a case study. In: *Lecture Notes In Artificial Intelligence*. [S.l.]: Springer Verlag, 1996. v. 1137, p. 107–119.

17 SATRACE 2010. URL: http://baldur.iti.uka.de/sat-race-2010/. Acessado em 27 de junho de 2011.

18 BERRE, D. L.; ROUSSEL, O.; SIMON, L. *SAT 2009 competitive events booklet: preliminary version*. URL: http://www.satcompetition.org/. Acessado em 02 de agosto de 2011.

19 MINISAT Page. URL: http://minisat.se/. Acessado em 27 de junho de 2011.

20 DIMACS Implementation Challenges. URL: http://dimacs.rutgers.edu/Challenges/. Acessado em 24 de junho de 2011.

21 THOMASON, L. *TinyXML Main Page*. URL: http://www.grinninglizard.com/tinyxml/. Acessado em 24 de junho de 2011.

22 NARAYANASAMY, P. *A New Heuristic for the Hamiltonian Circuit Problem*. 2009. URL: http://repository.lib.ncsu.edu/ir/bitstream/1840.16/166/1/etd.pdf. Acessado em 18 de julho de 2011.

23 CONRAD, A. et al. Solution of the knight's hamiltonian path problem on chessboards. *Discrete Applied Mathematics*, v. 50, n. 2, p. 125–134, 1994.

24 WEISSTEIN, E. W. *Knight's Tour*. URL: http://mathworld.wolfram.com/KnightsTour.html. Acessado em 11 de agosto de 2011.

25 KAUTZ, H.; SELMAN, B. *Planning as Satisfiability*. [S.l.]: John Wiley & Sons, Inc., 1992. 359–363 p.

26 LENHARDT, R. *Proof of Concept: Fast Solutions to NP-problems by Using SAT and Integer Programming Solvers*. 2010. URL: http://arxiv.org/abs/1011.5447.

APÊNDICE A – Tabelas Complementares sobre Desempenho das Aplicações

Tabelas de desempenho complementares às apresentadas no capítulo 7. Os tempos não incluem a geração das cláusulas.

Os campos marcados com asterisco são de execuções que excederam cinco minutos e foram abortadas. Dois asteriscos indicam que a execução foi interrompida por esgotar a memória RAM.

P=0						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.042	1239	40	0.007	1739	100
15	1.245	4334	60	0.015	6134	225
25	*	21499	125	0.057	29474	625
50	*	179599	300	0.605	242699	2500
100	*	1468099	700	5.886	1970399	10000
150	*	5000099	1200	24.766	6683099	22500
200	*	11871799	1600	80.669	15880799	40000
250	*	23220999	2000	**	31063499	62500

Tabela 14: Resultados para precosat com p = 0

P=0.25						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.050	1062	40	0.020	1562	100
15	0.525	3657	60	0.055	5457	225
25	4.229	18027	125	0.399	26002	625
50	*	150101	300	3.487	213201	2500
100	*	1224394	700	15.205	1726694	10000
150	*	4177768	1200	47.717	5860768	22500
200	*	9902760	1600	108.547	13911760	40000
250	*	19360420	2000	**	27202920	62500

Tabela 15: Resultados para precosat com p = 0.25

P=0.5						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.045	861	40	0.018	1361	100
15	0.296	2990	60	0.054	4790	225
25	1.433	14584	125	0.339	22562	625
50	4.938	121256	300	3.025	184356	2500
100	68.54	975409	700	12.769	1477709	10000
150	*	3351364	1200	41.507	5034364	22500
200	*	7920389	1600	89.783	11929389	40000
250	*	15495359	2000	**	23337859	62500

Tabela 16: Resultados para precosat com p = 0.5

P=0.75						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.017	702	40	0.058	1202	100
15	0.098	2281	60	0.039	4062	225
25	0.568	11059	125	0.292	19034	625
50	1.963	91824	300	2.666	154924	2500
100	25.53	738964	700	11.138	1241264	10000
150	141.62	2519249	1200	31.444	4202249	22500
200	*	5959973	1600	86.611	9968973	40000
250	*	11645985	2000	**	19488485	62500

Tabela 17: Resultados para precosat com p = 0.75

P=1						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.025	510	40	0.009	1010	100
15	0.11	1590	60	0.044	3390	225
25	0.861	7675	125	0.229	15650	625
50	2.39	61950	300	2.192	125050	2500
100	20.599	497800	700	8.82	1000100	10000
150	139.639	1692150	1200	27.671	3375150	22500
200	198.58	3991200	1600	109.652	8000200	40000
250	**	7782750	2000	**	15625250	62500

Tabela 18: Resultados para precosat com p = 1

P=0						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.011	1239	40	0.01	1739	100
15	0.198	4334	60	0.021	6134	225
25	*	21499	125	0.113	29474	625
50	*	179599	300	1.826	242699	2500
100	*	1468099	700	39.559	1970399	10000
150	*	5000099	1200	8.14	6683099	22500
200	*	11871799	1600	54.074	15880799	40000
250	*	23220999	2000	217.344	31063499	62500

Tabela 19: Resultados para clasp com p = 0

P=0.25						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.022	1062	40	0.005	1577	100
15	0.07	3657	60	0.015	5476	225
25	1.036	18027	125	0.095	25850	625
50	82.556	150101	300	1.678	213495	2500
100	*	1224394	700	33.008	1729829	10000
150	*	4177768	1200	8.746	5857043	22500
200	*	9902760	1600	53.384	13897963	40000
250	*	19360420	2000	58.301	27225413	62500

Tabela 20: Resultados para clasp com p = 0.25

P=0.5						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.00900	861	40	0.017	1361	100
15	0.044	2990	60	0.013	4790	225
25	0.679	14584	125	0.074	22562	625
50	48.683	121256	300	1.16	184356	2500
100	*	975409	700	24.405	1477709	10000
150	*	3351364	1200	8.462	5034364	22500
200	*	7920389	1600	48.998	11929389	40000
250	*	15495359	2000	103.657	23337859	62500

Tabela 21: Resultados para clasp com p = 0.5

P=0.75						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.012	702	40	0.007	1154	100
15	0.065	2281	60	0.012	4006	225
25	0.393	11059	125	0.062	19202	625
50	24.653	91824	300	0.832	154989	2500
100	*	738964	700	14.922	1248689	10000
150	*	2519249	1200	8.395	4204782	22500
200	*	5959973	1600	32.71	9964529	40000
250	*	11645985	2000	47.662	19497449	62500

Tabela 22: Resultados para clasp com p = 0.75

P=1						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.011	510	40	0.006	1010	100
15	0.027	1590	60	0.015	3390	225
25	0.21	7675	125	0.083	15650	625
50	6.768	61950	300	1.184	125050	2500
100	*	497800	700	23.636	1000100	10000
150	*	1692150	1200	66.31	3375150	22500
200	*	3991200	1600	35.74	8000200	40000
250	*	7782750	2000	39.522	15625250	62500

Tabela 23: Resultados para clasp com p = 1

P=0						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.021	1239	40	0.023	1739	100
15	0.031	4334	60	0.097	6134	225
25	24.313	21499	125	2.799	29474	625
50	*	179599	300	*	242699	2500
100	*	1468099	700	*	1970399	10000
150	*	5000099	1200	**	6683099	22500
200	*	11871799	1600	**	15880799	40000
250	*	23220999	2000	**	31063499	62500

Tabela 24: Resultados para gnovelty+ com p = 0

P=0.25						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.018	1062	40	0.107	1577	100
15	0.02	3657	60	0.022	5476	225
25	0.233	18027	125	0.106	25850	625
50	1.021	150101	300	1.712	213495	2500
100	17.066	1224394	700	39.787	1729829	10000
150	100.752	4177768	1200	**	5857043	22500
200	*	9902760	1600	**	13897963	40000
250	*	19360420	2000	**	27225413	62500

Tabela 25: Resultados para gnovelty+ com p = 0.25

P=0.5						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.017	861	40	0.013	1361	100
15	0.016	2990	60	0.02	4790	225
25	0.062	14584	125	0.088	22562	625
50	0.554	121256	300	1.431	184356	2500
100	10.711	975409	700	31.842	1477709	10000
150	29.641	3351364	1200	**	5034364	22500
200	*	7920389	1600	**	11929389	40000
250	*	15495359	2000	**	23337859	62500

Tabela 26: Resultados para gnovelty+ com p = 0.5

P=0.75						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.015	702	40	0.108	1154	100
15	0.025	2281	60	0.019	4006	225
25	0.043	11059	125	0.077	19202	625
50	0.42	91824	300	1.189	154989	2500
100	8.269	738964	700	25.957	1248689	10000
150	23.661	2519249	1200	**	4204782	22500
200	*	5959973	1600	**	9964529	40000
250	*	11645985	2000	**	19497449	62500

Tabela 27: Resultados para gnovelty+ com p = 0.75

P=1						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
10	0.016	510	40	0.01	1010	100
15	0.013	1590	60	0.017	3390	225
25	0.039	7675	125	0.065	15650	625
50	0.282	61950	300	0.992	125050	2500
100	2.702	497800	700	20.205	1000100	10000
150	16.174	1692150	1200	**	3375150	22500
200	50.527	3991200	1600	**	8000200	40000
250	**	7782750	2000	**	15625250	62500

Tabela 28: Resultados para gnovelty+ com p = 1

Aplicação: precosat. Problema do passeio do cavalo.						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
16 (4×4)	2.75	4800	64	0.507	6992	256
25 (5×5)	202.6	19771	125	0.755	27746	625
36 (6×6)	*	62188	216	10.285	85192	1296
49 (7×7)	*	159735	294	106.359	219074	2401
64 (8×8)	*	361872	384	*	495056	4096

Tabela 29: Resultados para precosat com grafos do problema do cavalo

Aplicação: clasp. Problema do passeio do cavalo.						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
16 (4×4)	1.396	4800	64	0.117	6992	256
25 (5×5)	85.837	19771	125	0.3	27746	625
36 (6×6)	*	62188	216	2.387	85192	1296
49 (7×7)	*	159735	294	13.565	219074	2401
64 (8×8)	*	361872	384	124.523	495056	4096

Tabela 30: Resultados para clasp com grafos do problema do cavalo

Aplicação: gnovelty+. Problema do passeio do cavalo.						
	Redução de Iwama e Miyazaki[14]			Redução de Torn[15]		
Vértices	Tempo (s)	Cláusulas	Variáveis	Tempo (s)	Cláusulas	Variáveis
16 (4×4)	*	4800	64	*	6992	256
25 (5×5)	65.384	19771	125	1.014	27746	625
36 (6×6)	*	62188	216	57.893	85192	1296
49 (7×7)	*	159735	294	*	219074	2401
64 (8×8)	*	361872	384	*	495056	4096

Tabela 31: Resultados para gnovelty+ com grafos do problema do cavalo

APÊNDICE B – Código fonte das aplicações desenvolvidas

```
Building:

Include the directory include/ in the compiler path.
Define HAMPATH_ENCODER before compiling the encoder.
Define HAMPATH_DECODER before compiling the decoder.
Define HAMPATH_DEBUG for debugging information.

Example input file for hampath_encoder:

<?xml version="1.0" ?>
<graph>
   <vertex name="a" />
   <vertex name="b" />
   <vertex name="c" />

   <edge from="a" to="b" />
   <edge from="b" to="c" />
</graph>

Declaring vertices is optional. Avoid special characters in the name field.

Example input file for hampath_decoder:

SAT
-1 -2 -3 4 5 -6

-----

Using:

Encoding graphs (graph_file) into DIMACS formulae (encoding_file). Iwama
and Torn are the reductions available:
\$ hampath_encoder (iwama|torn) (graph_file) (encoding_file)

Run your sat solver with the cnf file (encoding_file) as a parameter.
Example:
\$ solver (encoding_file) (model_file)

Decoding a model generated by your sat solver:
```

```
\$ hampath_decoder (iwama|torn) (graph_file) (model_file)
```

Arquivo B.1: README.txt

```
#ifdef HAMPATH_ENCODER

#include <iostream>
#include <stdexcept>

#include "reductions/iwama.h"
#include "reductions/torn.h"

using namespace hampath;

int main(int argc, char **argv) {
   try{
      if(argc != 4)
         throw std::runtime_error("Usage: hampath_encoder (iwama|torn) (graph_file) (
             encoding_file)");

      Graph graph(argv[2]);

      Encoder *encoder;
      if(std::string("torn") == argv[1])
         encoder = new TornEncoder(graph, argv[3]);
      else
         encoder = new IwamaEncoder(graph,argv[3]);

      encoder->encode();
      std::cout << "Success." << std::endl;

      delete encoder;
   }catch(std::exception& e){
      std::cout << e.what() << std::endl;
   }

   return 0;
}

#endif //HAMPATH_ENCODER
```

Arquivo B.2: main_encoder.cc

```
#ifdef HAMPATH_DECODER

#include <stdexcept>
#include <iostream>

#include "reductions/torn.h"
#include "reductions/iwama.h"

using namespace hampath;

int main(int argc, char **argv) {
```

```
    try{
        if(argc != 4)
            throw std::runtime_error("Usage: hampath_decoder (iwama|torn) (graph_file) (
                model_file)");

        Graph graph(argv[2]);

        Decoder* decoder;
        if(std::string("torn") == argv[1])
            decoder = new TornDecoder(graph,argv[3]);
        else
            decoder = new IwamaDecoder(graph,argv[3]);

        std::vector<std::string> hampath = decoder->get_hampath();
        if(hampath.size() != 0){
            if(!graph.is_hampath(hampath))
                throw std::runtime_error("Invalid hamiltonian path found.");

            std::cout << "Hamiltonian path: " << std::endl
                  << hampath[0];

            for(unsigned i = 1; i < hampath.size(); i++)
                std::cout << " -> " << hampath[i] ;

            std::cout << std::endl;
        }else{
            std::cout << "There isn't a Hamiltonian Path." << std::endl;
        }

    }catch(std::exception& e){
        std::cout << e.what() << std::endl;
    }

    return 0;
}

#endif //HAMPATH_DECODER
```

Arquivo B.3: main_decoder.cc

```
#ifndef DECODER_H_
#define DECODER_H_

#include <fstream>

#include "graph/graph.h"

namespace hampath{

class Decoder {
public:
    Decoder(Graph& graph, const char* filepath);
    virtual ~Decoder();
```

```
    virtual std::vector<std::string> get_hampath() = 0;
protected:
    Graph& graph;
    std::ifstream file;
};

};

#endif /* DECODER_H_ */
```

Arquivo B.4: decoder.h

```
#ifndef ENCODER_H_
#define ENCODER_H_

#include "graph/graph.h"

#include <fstream>

namespace hampath{

class Encoder {
public:
    Encoder(Graph& graph, const char* filepath);
    virtual ~Encoder();

    virtual void encode() = 0;
protected:
    Graph& graph;
    std::ofstream file;
};

};

#endif /* ENCODER_H_ */
```

Arquivo B.5: encoder.h

```
#ifndef DIMACSENCODING_H_
#define DIMACSENCODING_H_

#include <fstream>
#include <sstream>

#include "encoder.h"
#include "decoder.h"

namespace hampath{

class IwamaEncoder : public Encoder {
public:
    IwamaEncoder(Graph& graph, const char* filepath);
    virtual ~IwamaEncoder();
protected:
```

```
    void encode();

    void insert_clause(unsigned vertex_code, unsigned position_code);
    void insert_clause(unsigned v1_code, unsigned v1_position, unsigned v2_code, unsigned
        v2_position);

    unsigned power, n_clauses;
};

class IwamaDecoder : public Decoder {
public:
    IwamaDecoder(Graph& graph, const char* filepath);
    virtual ~IwamaDecoder();

    std::vector<std::string> get_hampath();
protected:
    unsigned power;
};

};

#endif /* DIMACSENCODING_H_ */
```

Arquivo B.6: iwama.h

```
#ifndef TORN_H_
#define TORN_H_

#include <sstream>

#include "encoder.h"
#include "decoder.h"

namespace hampath{

class TornEncoder : public Encoder {
public:
    TornEncoder(Graph& graph, const char* filepath);
    virtual ~TornEncoder();

    void encode();
protected:
    unsigned code_var(unsigned n, unsigned v) const;

    unsigned nVertices;
};

class TornDecoder : public Decoder{
public:
    TornDecoder(Graph& graph, const char* filepath);
    virtual ~TornDecoder();

    std::vector<std::string>  get_hampath();
};
```

```
};

#endif /* TORN_H_ */
```

Arquivo B.7: torn.h

```
#ifndef GRAPH_H_
#define GRAPH_H_

#include <graph/vertex.h>

#include <string>
#include <set>
#include <vector>

namespace hampath{

class Graph {
public:
    Graph(std::string filepath);
    virtual ~Graph();

    inline unsigned size() const { return vertices.size(); }
    bool is_hampath(const std::vector<std::string>& path) const;

    typedef std::set<Vertex*, VertexNameCompare> VertexSet;
    const VertexSet& get_vertices() const;

    std::string to_string() const;
protected:
    void read_xml(std::string filepath);

    Vertex* get_vertex(std::string name) const;
    Vertex* add_vertex(std::string name);
    void add_edge(std::string first, std::string second);

    VertexSet vertices;
};

}

#endif /* GRAPH_H_ */
```

Arquivo B.8: graph.h

```
#ifndef VERTEX_H_
#define VERTEX_H_

#include <set>
#include <string>

namespace hampath {
```

```
class Vertex {
public:
   Vertex(const std::string& name);
   virtual ~Vertex();

   void insert_successor(Vertex* successor);
   bool has_edge_to(const Vertex* destination) const;

   std::string to_string() const;
   inline std::string get_name() const { return name; }
protected:
   std::string name;

   typedef std::set<Vertex*> VertexSet;
   VertexSet successors;
};

struct VertexNameCompare{
   bool operator()(const Vertex* v1, const Vertex* v2) const{
      return v1->get_name() < v2->get_name();
   }
};

} /* namespace hampath */
#endif /* VERTEX_H_ */
```

Arquivo B.9: vertex.h

```
#include <stdexcept>

#include "reductions/encoder.h"

using namespace hampath;

Encoder::Encoder(Graph& graph, const char* filepath)
   : graph(graph), file(filepath, std::fstream::trunc){
   if(!file)
      throw std::runtime_error("Invalid output file");

   if(graph.size() < 2)
      throw std::runtime_error("Graph has less than 2 vertices, therefore it has a
          hamiltonian path.");

}

Encoder::~Encoder() {
}
```

Arquivo B.10: encoder.cpp

```
#include <stdexcept>

#include "reductions/decoder.h"
```

```
using namespace hampath;

Decoder::Decoder(Graph& graph, const char* filepath)
    : graph(graph), file(filepath){
    if(!file)
        throw std::runtime_error("Invalid input file");
}

Decoder::~Decoder() {
}
```

Arquivo B.11: decoder.cpp

```
#include "reductions/iwama.h"
#include <stdexcept>
#include <fstream>
#include <limits>

#ifdef HAMPATH_DEBUG
    #include <iostream>
#endif

using namespace hampath;

IwamaEncoder::IwamaEncoder(Graph& graph, const char* filepath)
    : Encoder(graph,filepath), n_clauses(0) {

    encode();
}

void IwamaEncoder::encode(){
    #ifdef HAMPATH_DEBUG
        std::cout << "Encoding." << std::endl;
    #endif //HAMPATH_DEBUG

    unsigned nVertices = graph.size();

    //Determines the number of variables necessary to represent the position of each
        vertex in the path
    unsigned max_representable = 2;
    power = 1;
    while(max_representable < nVertices) {
        max_representable *= 2;
        power++;
    }// power = ceil(log2(n))

    //Just a trick to allocate space in the file for the number of clauses (we don't have
        it yet)
    file << "p cnf " << nVertices*power << " ";
    unsigned nClausePosition = file.tellp();
    file << "          " << std::endl;

    //Reduction
```

```
#ifdef HAMPATH_DEBUG
   std::cout << "Encoding phi_1." << std::endl;
#endif //HAMPATH_DEBUG

//First step: Create clauses to invalidate positions between nVertices and
    max_representable
for(unsigned position_code = nVertices; position_code < max_representable;
    position_code++){
   for(unsigned vertex_code = 0; vertex_code < nVertices; vertex_code++){
      insert_clause(vertex_code, position_code);
   }
}

#ifdef HAMPATH_DEBUG
   std::cout << "Encoding phi_2." << std::endl;
#endif //HAMPATH_DEBUG

//Second step: Create clauses to invalidate assignment of the same position to two
    different vertices
for(unsigned position_code = 0; position_code < nVertices; position_code++){
   for(unsigned v1_code = 0; v1_code < nVertices; v1_code++){
      for(unsigned v2_code = v1_code + 1; v2_code < nVertices; v2_code++){
         insert_clause(v1_code, position_code, v2_code, position_code);
      }
   }
}

#ifdef HAMPATH_DEBUG
   std::cout << "Encoding phi_3." << std::endl;
#endif //HAMPATH_DEBUG

//Third step: Create clauses to invalidate assignment of consecutive positions to
    vertices without edges between them
const Graph::VertexSet& vertices = graph.get_vertices();
for(unsigned position_code = 0; position_code < nVertices - 1; position_code++){
   Graph::VertexSet::const_iterator v1_iterator = vertices.begin();

   for(unsigned v1_code = 0; v1_code < nVertices; v1_code++){
      Graph::VertexSet::const_iterator v2_iterator = vertices.begin();

      for(unsigned v2_code = 0; v2_code < nVertices; v2_code++){
         //if there is no edge between them and they are different
         if( ( ! (*v1_iterator)->has_edge_to(*v2_iterator) ) && ( v1_code != v2_code )
             )
            insert_clause(v1_code, position_code, v2_code, position_code + 1);

         v2_iterator++;
      }

      v1_iterator++;
   }

}
```

```
    file.seekp(nClausePosition);
    file << n_clauses;

    file.close();
}

void IwamaEncoder::insert_clause(unsigned vertex_code, unsigned position_code){
    int var_starting_index = vertex_code*power + 1;
    for(int i = 0; i < (int)power; i++){
        bool value = (( position_code >> (power - i - 1) ) & 1u);

        if(value){
            file << -(var_starting_index + i) << " ";
        }else{
            file << (var_starting_index + i) << " ";
        }
    }

    file << " 0" << std::endl;

    n_clauses++;
}

void IwamaEncoder::insert_clause(unsigned v1_code, unsigned v1_position, unsigned v2_code
    , unsigned v2_position){
    int var1_starting_index = v1_code * power + 1;
    int var2_starting_index = v2_code * power + 1;

    for(int i = 0; i < (int)power; i++){
        bool value1 = (( v1_position >> (power - i - 1) ) & 1u);
        bool value2 = (( v2_position >> (power - i - 1) ) & 1u);

        if(value1){
            file << -(var1_starting_index + i) << " ";
        }else{
            file << (var1_starting_index + i) << " ";
        }

        if(value2){
            file << -(var2_starting_index + i) << " ";
        }else{
            file << (var2_starting_index + i) << " ";
        }
    }

    file << " 0"<< std::endl;

    n_clauses++;
}

IwamaEncoder::~IwamaEncoder() {
}

//DimacsDecoding
```

```
IwamaDecoder::IwamaDecoder(Graph& graph, const char* filepath)
    : Decoder(graph,filepath){
}

std::vector<std::string>  IwamaDecoder::get_hampath(){
    std::string satisfiability;
    file >> satisfiability;
    if(satisfiability != "SAT")
        return std::vector<std::string>();

    unsigned n_vertices = graph.size();

    //Determines the number of variables necessary to represent the position of each
        vertex in the path
    unsigned max_representable = 2;
    power = 1;
    while(max_representable < n_vertices) {
        max_representable *= 2;
        power++;
    }// power = ceil(log2(n))

    std::vector<bool> model( (power * n_vertices) + 1);
    int number = 0;
    file >> number;
    while(number != 0){
        if(number > 0)
            model[number] = 1;
        else
            model[-number] = 0;

        file >> number;
    }

    std::vector<std::string>  hampath(n_vertices);

    unsigned index = 0;
    const Graph::VertexSet& vertices = graph.get_vertices();
    for(Graph::VertexSet::const_iterator v = vertices.begin(); v != vertices.end(); v++,
        index++){
        unsigned initial_variable = (power * index) + 1;

        unsigned position = 0;
        for(unsigned i = 0; i < power; i++){
            if(model[initial_variable + i]){
                position = ( (position << 1) | 1u);
            }else{
                position = (position << 1);
            }
        }

        if(position >= n_vertices)
            throw std::runtime_error("Invalid model.");
```

```
        hampath[position] = (*v)->get_name();
    }

    if(!graph.is_hampath(hampath))
        throw std::runtime_error("Invalid hampath found.");

    return hampath;
}

IwamaDecoder::~IwamaDecoder(){

}
```

Arquivo B.12: iwama.cpp

```
#include "reductions/torn.h"

using namespace hampath;

TornEncoder::TornEncoder(Graph& graph, const char* filepath)
    : Encoder(graph,filepath){
    nVertices = graph.size();
}

//Returns the code of the variable which will be true iff vertex v occupies the position
    n in the hamiltonian path.
unsigned TornEncoder::code_var(unsigned n, unsigned v) const{
    return nVertices*v + n + 1;
}

void TornEncoder::encode(){
    //Just a trick to allocate space in the file for the number of clauses (we don't have
        it yet)
    file << "p cnf " << nVertices * nVertices << " ";
    unsigned nClausePosition = file.tellp();
    file << "          " << std::endl;

    unsigned nClauses = 0;

    //Every node v must appear in the path in some position n
    for(unsigned v = 0; v < nVertices; v++){
        for(unsigned n = 0; n < nVertices; n++){
            file << code_var(n,v) << " ";
        }

        file << "0" << std::endl;
        nClauses++;
    }

    //Every node v must appear only once in the path
    for(unsigned v = 0; v < nVertices; v++){
        for(unsigned i = 0; i < nVertices; i++){
            for(unsigned j = i + 1; j < nVertices; j++){
                file << -1 * int(code_var(i,v)) << " " << -1 * int(code_var(j,v));
```

```
            file << " 0" << std::endl;
            nClauses++;
        }
    }
}

//Every position n has a node v assigned to it
for(unsigned n = 0; n < nVertices; n++){
    for(unsigned v = 0; v < nVertices; v++){
        file << code_var(n,v) << " ";
    }

    file << " 0" << std::endl;
    nClauses++;
}

//Two nodes v and w cannot be in the same position n
for(unsigned n = 0; n < nVertices; n++){
    for(unsigned v = 0; v < nVertices; v++){
        for(unsigned w = v + 1; w < nVertices; w++){
            file << -1 * int(code_var(n,v)) << " " << -1 * int(code_var(n,w));
            file << " 0" << std::endl;
            nClauses++;
        }
    }
}

//If v hasn't an edge to w, w cannot be the next node in the path
//This is the only set of clauses that differs between graphs with the same number of
    vertices
const Graph::VertexSet& vertices = graph.get_vertices();
int v_index = 0;
for(Graph::VertexSet::const_iterator v = vertices.begin(); v  != vertices.end(); v++,
    v_index++){
    //For all vertices
    int w_index = 0;
    for(Graph::VertexSet::const_iterator w = vertices.begin(); w  != vertices.end(); w
        ++, w_index++){
        //If v hasn't an edge to w
        if( !(*v)->has_edge_to(*w)){
            //For all the positions n except the last, position n+1 isn't occupied by w
            for(unsigned n = 0; n < nVertices - 1; n++){
                file << -1 * int(code_var(n ,v_index)) << " " << -1 * int(code_var(n+1 ,
                    w_index));
                file << " 0" << std::endl;

                nClauses++;
            }
        }
    }
}

file.seekp(nClausePosition);
file << nClauses;
```

```
    file.close();
}

TornEncoder::~TornEncoder() {
}

//TornDecoder

TornDecoder::TornDecoder(Graph& graph, const char* filepath)
    : Decoder(graph,filepath){

}

std::vector<std::string> TornDecoder::get_hampath(){
    std::string satisfiability;
    file >> satisfiability;
    if(satisfiability != "SAT")
        return std::vector<std::string>();

    unsigned nVertices = graph.size();
    std::vector<std::string> hampath(nVertices);

    const Graph::VertexSet& vertices = graph.get_vertices();
    Graph::VertexSet::const_iterator vertex = vertices.begin();
    int variable = 0;
    file >> variable;
    unsigned current_position = 0;
    //For all variables in the list
    while(variable != 0 && vertex != vertices.end()){
        //If the variable is true, put the vertex inside the current position.
        //The next positive number will be in the variables for the next vertex.
        if(variable > 0){
            hampath[current_position] = (*vertex)->get_name();
            vertex++;
        }

        //The next variable corresponds to the next position until the next set of
            variables.
        current_position = (current_position + 1) % nVertices;
        file >> variable;
    }

    return hampath;
}

TornDecoder::~TornDecoder(){
}
```

Arquivo B.13: torn.cpp

```
#include "graph/graph.h"
#include <stdexcept>
#include <fstream>
```

```
#include <sstream>
#include "tinyxml/tinyxml.h"

using namespace hampath;

#ifdef HAMPATH_DEBUG
   #include <iostream>
#endif

Graph::Graph(std::string filepath) {
   #ifdef HAMPATH_DEBUG
      std::cout << "Reading file." << std::endl;
   #endif

   read_xml(filepath);

   #ifdef HAMPATH_DEBUG
      std::cout << "The graph has " << vertex_map.size() << " vertices." << std::endl;
   #endif
}

Vertex* Graph::get_vertex(std::string name) const{
   //Creates a dummy vertex to search by name inside the set
   Vertex dummy_vertex(name);
   VertexSet::iterator i = vertices.find(&dummy_vertex);
   if(i == vertices.end())
      return 0;

   return *i;
}

Vertex* Graph::add_vertex(std::string name){
   Vertex* vertex = get_vertex(name);
   if(!vertex){
      vertex = new Vertex(name);
      vertices.insert(vertex);
   }

   return vertex;
}

void Graph::add_edge(std::string first, std::string second){
   if(first == second)
      throw std::runtime_error("Invalid edge.");

   Vertex *origin = add_vertex(first);
   Vertex *destination = add_vertex(second);

   origin->insert_successor(destination);
}

const Graph::VertexSet& Graph::get_vertices() const{
   return vertices;
}
```

```
void Graph::read_xml(std::string filepath){
   TiXmlDocument document( filepath.c_str() );
   if(!document.LoadFile())
      throw std::runtime_error(std::string("invalid input file: ").append(filepath));

   TiXmlHandle hDoc(&document);

   TiXmlElement *graphElement = hDoc.FirstChild("graph").Element();
   if(!graphElement)
      throw std::runtime_error("parsing error: tag <graph> not found");

   for(TiXmlElement *i = graphElement->FirstChildElement("vertex"); i != 0; i = i->
       NextSiblingElement("vertex")){
      add_vertex(i->Attribute("name"));
   }

   for(TiXmlElement *i = graphElement->FirstChildElement("edge"); i != 0; i = i->
       NextSiblingElement("edge")){
      add_edge(i->Attribute("from"), i->Attribute("to"));
   }
}

//Verifier.
bool Graph::is_hampath(const std::vector<std::string>& path) const{
   //The path has n vertices
   if(path.size() != size())
      return false;

   //They are all valid
   for(unsigned i = 0; i < path.size(); i++){
      if(get_vertex(path[i]) == 0)
         return false;
   }

   //They are all different. (std::set doesn't allow duplicates)
   std::set<std::string> vertex_set;
   vertex_set.insert(path.begin(), path.end());
   if(vertex_set.size() != size())
      return false;

   //The edges are valid
   Vertex* origin = get_vertex(path[0]);
   for(unsigned i = 1; i < path.size(); i++){
      Vertex* destination = get_vertex(path[i]);

      if(!origin->has_edge_to(destination))
         return false;

      origin = destination;
   }

   return true;
}
```

```
std::string Graph::to_string() const{
    std::stringstream buffer;

    for(std::set<Vertex*>::const_iterator i = vertices.begin(); i != vertices.end(); i++){
        buffer << (*i)->to_string() << std::endl;
    }

    return buffer.str();
}

Graph::~Graph() {
    for(VertexSet::iterator i = vertices.begin(); i != vertices.end(); i++){
        delete *i;
    }
}
```

Arquivo B.14: graph.cc

```
#include "graph/vertex.h"

#include <sstream>

namespace hampath {

Vertex::Vertex(const std::string& name)
    : name(name){
}

void Vertex::insert_successor(Vertex* successor){
    successors.insert(successor);
}

bool Vertex::has_edge_to(const Vertex* destination) const{
    return (successors.find((Vertex*)destination) != successors.end());
}

std::string Vertex::to_string() const{
    std::stringstream buffer;

    buffer << "Vertex: " << name << ". Edges: { ";

    for(VertexSet::iterator i = successors.begin(); i != successors.end(); i++){
        buffer << (*i)->get_name() << ", ";
    }

    buffer << "}.";

    return buffer.str();
}

Vertex::~Vertex() {
}
```

```
} /* namespace hampath */
```

Arquivo B.15: vertex.cpp