Relatório Final - COS887

TÓPICOS ESPECIAIS EM OTIMIZAÇÃO E I.A.

Satyrus III

Pedro Maciel Xavier

DRE : 116023847

23 de setembro de 2019

Sumário

Sumário **2**

1 Introdução **3**
1.1 Cronograma 4
1.2 Informações técnicas 4
1.3 Implementação 5
1.4 Uso 6

2 Conceitos Teóricos **8**
2.1 Mapeamento 8
2.2 Tipos de Restrições 10

3 Tipos **13**
3.1 Números 13
3.2 Matrizes 13
3.3 Variáveis 14

4 Sintaxe do *SATish* **15**
4.1 Comentários 16
4.2 Diretivas 16
4.3 Atribuição 20
4.4 Definição de Restrições 21

5 Exemplo: Modelando a Coloração de Grafos **24**

6 Glossário **26**

Referências Bibliográficas **29**

Capítulo 1

Introdução

Pretendo fazer deste relatório uma primeira proposta de documentação para o **Satyrus III**. O desenvolvimento do projeto já se encontra na fase final e está suficientemente consolidado. Dessa forma, não veremos grandes mudanças no conteúdo desse documento.

O trabalho começou com um estudo preliminar, partindo do **Satyrus II**[1] e das suas principais referências[3][4]. Logo teve início o desenvolvimento, em conjunto com estudo mais aprofundado não só dos artigos sobre o **Satyrus** propriamente, como também acerca das redes de Hopfield[9] em um âmbito mais geral.

Os termos em negrito, em geral, tem sua essência resumida no glossário, ao fim do documento.

1.1 Cronograma

Quanto ao andamento do projeto, as etapas já concluídas foram:

1. Estudo:
    a) **Satyrus**[3][4], **Satyrus II**[1], **XOR-Satyrus**[2].
    b) Documentação do **`Ply`**[8] (**`Python Lex & Yacc`**).
    c) Documentação da **`D-Wave`**[5].
2. Desenvolvimento:
    a) Interpretador para o **SATish**.
    b) Tipagem do **Satyrus**.
    c) Geração da equação de energia.
    d) Simplificação das fórmulas.

As próximas atividades serão:

1. Estudo:
    a) Redução da ordem das redes de Hopfield.
    b) Topologia do grafo de conectividade da **`D-Wave`**.
2. Desenvolvimento:
    a) Interface com os *solvers*.
    b) Testes unitários.

1.2 Informações técnicas

Assim como no caso do **Satyrus II**, esta nova implementação foi desenvolvida em **`Python`**, mas em sua versão mais recente, a **`3.7`**. Dentre as principais bibliotecas utilizadas encontra-se a **`ply`**, onde estão implementações do **Lex** e do **Yacc** e a **`decimal`**, que proporciona tipos numéricos de precisão arbitrária.

Apesar de ter tido o código do **Satyrus II** como referência fundamental, o **Satyrus III** foi feito sem incorporar código anterior, visando mudanças em sua estrutura, sintaxe e funcionamento.

1.3 Implementação

Nesta terceira instância da plataforma foram feitas algumas escolhas de implementação que acho interessante ressaltar.

Expressões

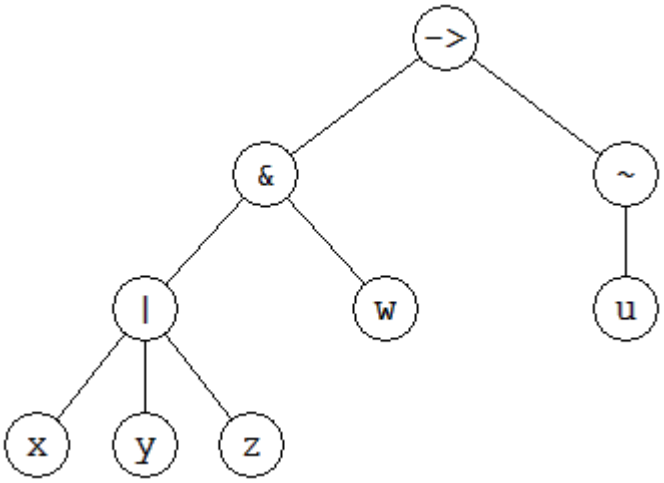
Um dos pontos principais da remodelagem foi a introdução do tipo **`Expr`** que representa uma expressão lógica ou aritmética (algumas vezes **pseudo-booleana**). Na versão anterior as manipulações eram feitas através de operações em *strings*. Isso agora é feito com uma estrutura em árvore.

Uma expressão tem dois atributos principais: **`head`** e **`tail`**. O **`head`** guarda o símbolo que representa um determinado operador, como **`&`**, **`->`** ou **`+`**. Já o **`tail`** é uma lista que pode conter diversos operandos, dependendo do operador.

Por exemplo, a expressão

$$(x \vee y \vee z) \wedge w \implies \neg u$$

é representada internamente como:



Quando mostrado na tela, este objeto da classe **`Expr`** dá origem a expressão:

```
(x | y | z) & w -> ~u
```

É visível que as expressões não armazenam os parênteses. Estes são gerados automaticamente quando uma saída na forma de *string* é solicitada.

Compilação e Execução

Durante o processo de geração da equação de energia a partir do decorrem, ao todo, três etapas:

1. Análise léxica e sintática

Essa primeira etapa fica a cargo do `|`lex— e do `|`yacc— e procura apontar erros no código, informando onde o mesmo se encontra e terminando a compilação.

2. Análise Semântica

Nesse momento, verifica-se a corretude das definições, isto é, se não há uso de variáveis não inicializadas; se as dimensões das matrizes são valores inteiros; etc. Assim como na etapa anterior, qualquer falha interrompe o processo e alerta o usuário.

3. Geração das Equações de Energia

Por fim, se todas as restrições forem interpretadas corretamente, teremos uma série de expressões lógicas resultantes que serão transformadas para a **C.N.F** e que terão suas respectivas penalidades aplicadas.

Em seguida, as fórmulas são traduzidas em equações de energia segundo o mapeamento proposto e então teremos uma expressão ainda estruturada em árvore, que será transformada e apresentada conforme as especificações do sistema resolvedor.

1.4 Uso

É possível usar o **Satyrus** sem instalação prévia se o **`Python 3.7`** estiver presente no sistema. Esse comportamento é o padrão nas distribuições mais recentes do **`Linux`** e do **`OSx`**.

Após escrever o código de especificação das restrições em um arquivo **.sat**, basta executar o seguinte comando:

bash:

```
.../Satyrus3$ ./satyrus problem.sat
```

power shell:

```
...\Satyrus3> python satyrus.py problem.sat
```

Ao término da compilação, o processo continua dependendo do que for especificado na diretiva **#out**.

Capítulo 2

Conceitos Teóricos

2.1 Mapeamento

O mapeamento a seguir apresenta uma forma de transformar expressões lógicas em fórmulas aritméticas, mais especificamente, em **equações de energia**.

Seja a um literal positivo (átomo) e p, q fórmulas lógicas quaisquer:

$$\begin{aligned}
\mathbb{H}(T) &= 1 \\
\mathbb{H}(F) &= 0 \\
\mathbb{H}(a) &= a \\
\mathbb{H}(p) &= 1 - \mathbb{H}(p) \\
\mathbb{H}(p \wedge q) &= \mathbb{H}(p) \times \mathbb{H}(q) \\
\mathbb{H}(p \vee q) &= \mathbb{H}(p) + \mathbb{H}(q) - \mathbb{H}(p \wedge q) \\
\mathbb{H}(p \oplus q) &= \mathbb{H}(p) + \mathbb{H}(q) - 2 \times \mathbb{H}(p \wedge q)
\end{aligned}$$

Tendo isso em mente, podemos repensar fórmulas de primeira ordem em termos da lógica proposicional.

Por exemplo, a fórmula

$$\forall x \quad \neg x, x \in \Omega$$

pode ser vista como

$$\bigwedge_{x \in \Omega} \neg x$$

De um modo geral, se $\#\Omega = n$,

$$\begin{aligned} \forall x_i \quad & \Psi(x_i) \equiv \bigwedge_{i=1}^{n} \Psi(x_i) \\ \exists x_i \quad & \Psi(x_i) \equiv \bigvee_{i=1}^{n} \Psi(x_i) \\ !\exists x_i \quad & \Psi(x_i) \equiv \bigoplus_{i=1}^{n} \Psi(x_i) \end{aligned}$$

Aplicando o mapeamento acima, teríamos:

$$\begin{aligned} \bigwedge_{i=1}^{n} x_i &\equiv \prod_{i=1}^{n} x_i \\ \bigvee_{i=1}^{n} x_i &\equiv \sum_{k=1}^{n} (-1)^{k+1} \sum_{I \in S(n,k)} \prod_{i \in I} x_i \\ \bigoplus_{i=1}^{n} x_i &\equiv \sum_{k=1}^{n} k \cdot (-1)^{k+1} \sum_{I \in S(n,k)} \prod_{i \in I} x_i \end{aligned}$$

Onde:

$$S(n,k) := \begin{cases} \{\{i\} : 1 \leq i \leq n\} & \text{se } k = 1 \\ \{I \cup \{j\} : \max I < j \leq n, I \in S(n, k-1)\} & \text{se } k \geq 2 \end{cases}$$

No entanto, nos dois casos de quantificação existencial ($\exists, !\exists$) o número de termos cresce exponencialmente com a quantidade de variáveis (2^n).

Por este motivo, evitamos arcar com o custo computacional da formulação acima e tomamos $\exists$ por:

$$\bigvee_{i=1}^{n} x_i \triangleq \sum_{i=1}^{n} x_i \qquad (\circledast)$$

O caso da existência com unicidade ($!\exists$) é abordado de maneira alternativa e foi estudado com maior profundamente em [2].

Por fim, para obter uma **equação de energia** cujo estado mínimo é a solução do problema de **SAT**, negamos a fórmula e aplicamos o mapeamento. Desse jeito, minimizar o valor da função resultante equivale a encontrar uma interpretação que satisfaça a fórmula original.

Seja Φ uma disjunção de fórmulas e Ψ uma conjunção:

$$\mathbb{E}(\Phi) = \prod_{i=1}^{m} \mathbb{H}(\Phi_i)$$

$$\mathbb{E}^*(\Psi) = \sum_{j=1}^{n} \mathbb{H}(\Psi_j)$$

Por conta da definição em ($\circledast$), escrevemos $\mathbb{E}^*(\Psi)$ em vez de $\mathbb{E}(\Psi)$, para simbolizar a adaptação realizada.

2.2 Tipos de Restrições

Na modelagem do **Satyrus** encontramos dois tipos de restrições que serão apresentadas a seguir. Combinando-as somos capazes de especificar problemas usando sentenças lógicas. Um problema, quando apresentado através de sua formulação lógica, se aproxima da forma usual de interpretar o seu significado. Isso nem sempre acontece com a formulação típica no contexto de otimização.

$$\min_{x \in \Omega} f(x)$$

$$\sum_{i} x_i \cdot b_i \geq 0$$

$$x_i \geq c$$

Apesar de serem compactas e muito úteis, especificações dessa forma nem sempre advêm de simples traduções a partir do contexto semântico onde reside o enunciado do problema, tampouco é fácil remontar ao significado do problema original.

Integridade

As restrições de integridade limitam o espaço de solução $\Omega \subseteq \mathbb{Z}_2^n$ e, em geral, identificam o problema em questão.

Por exemplo, como aparece no caso da coloração de grafos, dizer que dois vértices vizinhos i, j não podem ser pintados com a mesma cor k, ou seja,

$$vizinhos(i, j) \implies \neg(colore(i, k) \wedge colore(j, k))$$

representa uma restrição de integridade.

Otimalidade

As restrições de otimalidade são aquelas que, dentre as soluções plausíveis, determinam quais respostas oferecem o menor custo ao sistema, ou o melhor desempenho.

No cenário da coloração de grafos, a otimalidade aparece no tocante ao número de cores utilizadas, que queremos minimizar.

Um problema enunciado sem nenhuma restrição de otimalidade é simplesmente uma instância comum de **SAT**, onde só interessa saber se existe alguma solução no espaço de busca.

Penalidades

Durante a modelagem, as restrições são organizadas em diferentes níveis de penalidade. Isso é feito da seguinte forma: Define-se p_0 (penalidade base) e ε (fator de "desempate") e, em seguida, calculamos as penalidades de cada nível com base na seguinte relação:

$$p_k = p_{k-1} \times (n_{k-1} + 1) + \varepsilon$$

Onde n_k é o número de cláusulas presentes no nível k.

Essa construção se fundamenta no fato de que violar uma única restrição em um determinado nível custa ε mais caro do que violar todas as cláusulas dos níveis anteriores em conjunto.

Por definição, as restrições de otimalidade residem no nível `0`, restando às de integridade serem alocadas nos níveis superiores.

Capítulo 3

Tipos

3.1 Números

Internamente, foi definido o tipo **`Number`** que representa tanto as constantes reais usuais quanto os valores booleanos (**`0`**, **`1`**) Podem ser especificados em diversos formatos:

- **`12`**
- **`-1.032`**
- **`2E-7`**

A precisão das operações numéricas pode ser definida pelo usuário através da diretiva **`#prec`**.

3.2 Matrizes

As matrizes, modeladas sob a classe **`Array`**, são tabelas esparsas que guardam somente as posições declaradas na inicialização. São implementadas usando dicionários do **`Python`** (**`dict`**). É através delas que são definidas as variáveis do problema de **SAT**, como será discutido na próxima seção.

3.3 Variáveis

Instanciar uma matriz induz a criação de uma série de variáveis que serão utilizadas pelo sistema resolvedor selecionado. Toda vez que alguma entrada de uma matriz não for especificada, ela será convertida em um literal, cujo nome é dado pelo nome da variável, seguido da posição na matriz.

Por exemplo, Se não especificamos $A_{1,3}$ então a posição **`A[1][3]`** guardará o valor **`A_1_3`**.

Capítulo 4

Sintaxe do *SATish*

A linguagem **SATish** vem tomando forma com o passar do tempo, como pode ser visto em [3] e [1]. Com a presente modelagem do sistema também pode ser vista uma nova apresentação da sintaxe que busca simplificar a escrita e torná-la mais próxima de outras linguagens conhecidas, como `Python` e `C`, mas sem perder a identidade das versões anteriores.

Em linhas gerais, o **SATish** é uma linguagem declarativa, mas que pode conter traços do paradigma imperativo.

4.1 Comentários

São dois os tipos de comentário em **SATish**: o de linha simples, presente após a primeira ocorrência de `%` em uma linha; e o de múltiplas linhas, compreendido entre `%{` e `}%`, como nos exemplos abaixo:

Uma linha

```
% Um comentario

% Outro comentario
```

Múltiplas Linhas

```
%{
  Muitos
  comentarios
  pedem
  muitas
  linhas
  }%
```

4.2 Diretivas

As ”diretivas”são declarações especiais iniciadas em **#** que permitem interagir com os parâmetros do modelo, assim como incorporar matrizes e restrições definidas em outros arquivos **.sat** que podem ser .

Precisão Decimal

É possível definir a precisão (em casas decimais) do tipo numérico do **Satyrus**, que é **16** por padrão.

```
#prec : 32;
```

Vale ressaltar que, independente da precisão arbitrada pelo programador, os números especificados são armazenados conforme foram declarados e que esta

configuração só se torna evidente quanto ao número de casas decimais quando alguma operação aritmética é realizada.

Diretório

Definir o diretório é importante na hora de usar a diretiva **`#load`**, que lê constantes, matrizes e restrições definidas em outro arquivo.

```
#dir : "C:/Users/";
```

Também podem ser usados caminhos relativos, pois essa diretiva opera de forma similar ao comando **`cd`**, disponível para **`power shell`** e **`bash`**.

Penalidade Base (p_0)

Conforme introduzido nos conceitos teóricos do projeto, é possível definir a penalidade base da seguinte forma:

```
#p0 : 1;
```

Está sendo analisada a possibilidade (e a real necessidade[1]) de introduzir funções auxiliares para definir a penalidade a partir dos dados do problema. Isso seria feito, por exemplo, através de funções como máximo e mínimo aplicadas a matrizes.

```
#p0 : max x;

x[6] = {
        (1) :  5.7, (2) : 12.3, (3) : 25.7,
        (4) : 12.8, (5) :  9.4, (6) :  0.1
        };
```

É interessante notar que a penalidade é sempre definida por último, em um momento imediatamente anterior à compilação da equação de energia. Isso faz com que uma definição de p_0 dependente de outros dados do corpo do programa seja avaliada corretamente sem que nos preocupemos com a ordem em que isso é feito.

[1] Presentes nos processos de modelagem em [1], as funções `max` e `min` podem ser muito úteis em problemas maiores.

Fator de ”Desempate”(ε)

O fator de ”desempate”é, naturalmente, tal que $\varepsilon \ll p_0$, e recomenda-se que sua especificação seja feita através de notação científica, como no exemplo:

```
#eps : 1E-7;
```

É também preciso que a definição da precisão numérica (**#prec**) considere os efeitos de ε nos cálculos. Assim, um sinal de advertência será mostrado ao usuário caso $-\log_{10}\varepsilon \geq |prec|$ ou $\varepsilon > p_0$.

Execução de *script* externo

É possível ler matrizes, constantes e restrições de um outro arquivo **.sat**. As diretivas **#dir** e **#load** contidas em arquivos externos serão consideradas e as demais, porém, serão desprezadas.

```
#load : "script.sat";
```

Quando uma diretiva do tipo **#dir** aparece durante a execução do **#load**, o diretório é alterado momentaneamente para suprir outras eventuais instruções de **#dir** e **#load**, mas retorna ao estado anterior ao final do processo. Isso é feito com uma pilha.

Escolha do resolvedor

A escolha do sistema que receberá as equações de energia como entrada é realizada pela opção **#out**. As opções passadas em **[parametros]** são repassadas ao programa alvo.

```
#out : "cvxpy" [parametros];
```

As opções, a princípio, serão:

cvxpy *Convex Optimization Python Library*[6]
Apesar de inicialmente desenvolvida para problemas convexos, agora conta também com as ferramentas da biblioteca **GLPK**[7], cujo arcabouço engloba problemas de programação inteira.

dwave *D-Wave Quantum Annealing System*[5] [2,3]
Através do **API** da plataforma *Leap*, o problema pode ser enviado para solução no computador quântico *D-Wave 2000Q*. Seu funcionamento requer conexão com a internet e um **API Token** para autenticação.

neal D-Wave Ocean Library Annealing Simulator.[5][3]
O *solver* **neal** realiza o **Simulated Annealing** para um problema formulado segundo o **Modelo de Ising** e pode ser muito útil para realizar testes sem precisar do acesso ao computador quântico real. Essa ferramenta foi distribuída pela própria **D-Wave** e possui interface idêntica a da plataforma *Leap*.

mosel Xpress-Mosel.[4]
Conhecido *solver* de uso geral, desenvolvido pela **AMPL**.

text Escreve a **equação de energia** em forma de texto e salva em um arquivo especificado.

[2] Ainda não temos acesso a essa plataforma, pois atualmente só é possível obter um **Token** com uma conta advinda da Europa ou da América do Norte.

[3] A tradução do problema no **Modelo de Ising** ainda precisa de uma redução na ordem da rede de Hopfield, o que está sendo estudado.

[4] Essa opção foi incluída por já estar presente no **Satyrus II** e em seus testes, o que é importante como critério de comparação e validação. No entanto, há preferência pelo uso de *software open-source* e, principalmente, com integração imediata pelo **Python**.

4.3 Atribuição

Toda atribuição demanda um identificador que deve iniciar com letras ou _ (*underscore*) mas que pode conter números. Isso é representado pela expressão regular **"[a-zA-Z_][a-zA-Z0-9_]*"**.

Escalares

A declaração de escalares ocorre de forma simples:

```
m = 3;
n = 5;

x = y;
```

A uma variável pode ser atribuído um valor numérico literal ou o valor previamente armazenado em outra constante. Isto é particularmente útil na hora de definir as dimensões de uma matriz e também para a inicialização de suas entradas.

Matrizes

A sintaxe para definição de matrizes é outra nuance do **SATish** que procurei tornar semelhante ao **C**. Possui pequenas diferenças por se tratar de uma estrutura esparsa, o que torna a sintaxe parecida com a definição de dicionários do **Python**. Em linhas gerais, a declaração de uma matriz demanda, além do nome, o tamanho referente a cada dimensão. A inicialização das entradas é opcional. Vejamos os exemplos:

```
x[3] = {(1) : 0, (2) : 1, (3) : 1};

y[m] = {(1) : 1, (m) : 1};

A[3][3] = {
    (1,1) : 4, (1,2) : 3, (1,3) : 2,
    (2,1) : 1, (2,2) : 0, (2,3) :-1,
    (3,1) :-2, (3,2) :-3, (3,3) :-4
};
```

```
z[m][n];
```

Neste último, temos uma matriz de incógnitas z de dimensão $m \times n$.

4.4 Definição de Restrições

O ponto principal no código **Satyrus** é, de fato, a definição de restrições e, por isso, será descrita cada parte da declaração de forma separada.

```
(int) C[2]:

@ {i = [1 : m]}
@ {j = [1 : n]}

u[i][j] -> v[i][j];
```

Tipo de Restrição

Antes do nome que identifica a restrição, temos que informar o seu tipo. Existem duas opções, que são especificadas entre parênteses. Através dos seguintes identificadores as restrições serão agrupadas e as respectivas penalidades serão aplicadas:

- `(int)` Integridade
- `(opt)` Otimalidade

Nome e nível de Penalidade

Assim como os demais nomes que podem ser escolhidos na linguagem, estes devem começar com letras ou _ mas podem conter números. O identificador é seguido do nível de penalidade, que aparece entre colchetes. Aqui também se pode informar o nível através de uma constante definida anteriormente.

```
(int) A[2]:
(int) B[3]:
```

Quando o nível não é informado, o mesmo é definido como `0`, por padrão. Este é o usual para restrições de otimalidade.

```
(opt) C:
```

Variáveis Quantificadas e suas condições

Cada declaração de quantificação contém quatro partes: o quantificador, a variável, o domínio e as condições, sendo esta última opcional.

A quantificação de variáveis é dada pelos símbolos quantificadores:

- `@` universal (∀)
- `$` existencial (∃)
- `!$` existencial com unicidade (!∃) [5]

Após declarar o nome da variável, o domínio é especificado entre colchetes, sendo obrigatório determinar o início e o fim na forma `[início:fim]`. Uma sintaxe alternativa permite dizer também o passo (incremento). Para isso escreve-se `[início:fim:passo]`. Todos os parâmetros devem ser números inteiros. Como a própria notação de colchetes sugere, os intervalos descritos são fechados, isto é, `i = [1:n]` equivale a $1 \leq i \leq n$.

Também é possível declarar múltiplas condições, que devem depender de variáveis e constantes especificadas anteriormente, como no exemplo abaixo:

```
@ {i = [1:5]}
$ {j = [m:n]}

@ {k = [1:n], k!=i, k!=j}
```

Expressões Lógicas

As expressões lógicas são composta por literais, operadores e índices que estendem a sintaxe de lógica binária (bit-a-bit) da linguagem `C`. Vejamos:

- `&` E (Conjunção, ∧)

[5] Apesar de compreendido na linguagem, esta funcionalidade está sob estudo[2] e ainda não foi implementada internamente.

- `|` OU (Disjunção, $\vee$)
- `^` OU-Exclusivo (Disjunção Exclusiva, $\oplus$)
- `~` NÃO (Negação, $\neg$)

Além de alguns símbolos adicionais:

- `->`,`<-` SE ... ENTÃO (Implicação, $\Longrightarrow$, $\Longleftarrow$)
- `<->` SE E SOMENTE SE (Equivalência, $\Longleftrightarrow$)

Alguns exemplos:

A fórmula $(x \wedge y) \implies (z \vee w)$, escrita em **SATish** fica:

```
(x & y) -> (z | w)
```

Já para a expressão $x_i \oplus y_j \iff z_k$ escreve-se:

```
(x[i] ^ y[j]) <-> z[k]
```

Conclusão

Por fim, uma restrição de integridade hipotética da forma

$$\forall i \, \exists j \; x_i \wedge y_j \implies z_{i,j} : 1 \leq i \leq m, 1 \leq j \leq n, j \neq i$$

no segundo nível de penalidade seria escrita como:

```
(int) A[2]:

@ {i = [1:m]}
$ {j = [1:n], j != i}

(x[i] & y[j]) -> z[i][j];
```

Capítulo 5

Exemplo:
Modelando a Coloração de Grafos

```
#eps : 0.0001;

#p0  : 1;

%{ ---------------------- }
 * Satyrus III - Exemplo *
 *                        *
 *  Coloracao de Grafos   *
 { ---------------------- }%

m = 4; % numero de nos

viz[m][m] = { % Matriz de Adjacencias
    (1,1) : 0, (1,2) : 1, (1,3) : 0, (1,4) : 1,
    (2,1) : 1, (2,2) : 0, (2,3) : 1, (2,4) : 0,
    (3,1) : 0, (3,2) : 1, (3,3) : 0, (3,4) : 1,
    (4,1) : 1, (4,2) : 0, (4,3) : 1, (4,4) : 0
};

cor[m]; % Vetor de Cores

vc[m][m]; % Matriz de Solucao (mapa de cores)
```

```
%{ ---------- *
 * Restricoes *
 * ---------- }%

(int) A[1]: % Todo vertice deve ser pintado

@ {i = [1:m]} % para todo vertice `i`
$ {j = [1:m]} % existe uma cor `j`

vc[i][j];

(int) B[2]: % Vizinhos de cores diferentes

@ {i = [1:m]}          % para todo vertice `i`
@ {j = [1:m], j != i} % para todo outro vertice `j`
@ {k = [1:m]}          % para toda cor `k`

viz[i][j] -> ~(vc[i][k] & vc[j][k]);

(int) C[2]: % Pintar cada vertice com uma unica cor

@ {i = [1:m]}          % para todo `i`
@ {j = [1:m]}          % para toda cor `j`
@ {k = [1:m], k != j} % para toda cor `k`

~(vc[i][j] & vc[i][k]);

(int) D[1]: % Cores utilizadas

@ {i = [1:m]} % para todo vertice `i`
@ {k = [1:m]} % para toda cor `k`

vc[i][k] -> cor[k];

(opt) E: % Minimizar o numero de cores

$ {k = [1:m]} cor[k];
```

Capítulo 6

Glossário

C.N.F.

Conjunctive Normal Form, ou Forma Normal Conjuntiva, é a representação de uma fórmula lógica como uma conjunção de cláusulas, onde cada cláusula é uma disjunção de literais. Por exemplos, seja Ψ uma fórmula e $p_{i,j}$ seus literais, de modo que:

$$\Psi = \bigvee_{i}^{m} \bigwedge_{j}^{n} p_{i,j} = (p_{1,1} \wedge \cdots \wedge p_{1,n}) \vee \cdots \vee (p_{m,1} \wedge \cdots \wedge p_{m,n})$$

Dizemos que Ψ está na **C.N.F**.

Fórmula Pseudo-booleana

Uma expressão desse tipo contém não somente os conectivos lógicos usuais ($\wedge, \vee, \oplus, \neg, \implies, \impliedby, \iff$), mas também constantes multiplicativas que muitas vezes atuam como penalidades. Isso significa que uma interpretação sobre uma fórmula dessa natureza não necessariamente será valiada como `0` ou `1`, mas pode assumir um valor real qualquer.

Lex

Ferramenta responsável por transformar uma *string* de código (texto) em uma sequência de *tokens*, segundo a gramática especificada.

Modelo de Ising

Modelo matemático criado para descrever interações ferromagnéticas, cuja

forma característica é dada pelo **Hamiltoniano**

$$H = \sum_i h_i s_i + \sum_{i<j} J_{i,j} s_i s_j$$

Onde $s \in \{|\uparrow\rangle, |\downarrow\rangle\}$, h_i são os termos lineares e $J_{i,j}$ são os termos quadráticos.

Em um sistema físico, os termos quadráticos ($J_{i,j}$) são entendidos como a interação entre pares de partículas; os termos lineares (h_i) como as interações destas com o campo magnético externo; e s representa o *spin*.

Quantum Annealing

Em Português, algo como "Têmpera Quântica", é um processo físico no qual um sistema quântico, inicialmente em seu nível mais baixo de energia, evolui gradativamente para um estado diferente do inicial permanecendo no estrato energético fundamental.

No contexto de computação quântica, é compreendido como um método de computação adiabática, que ataca problemas de otimização através das propriedades quânticas de superposição e tunelamento. Consiste em definir um estado quântico simples que represente todas as variáveis do problema em termos lineares e realizar a transformação para gerar um estado que pode conter interações de até segunda ordem (matematicamente, polinômios de grau 2). A forma de segunda ordem é escolhida como sendo aquela que representa a solução do problema e deve se encontrar, ao término do processo, no estado de menor energia. As interações são geralmente representadas pelo **Modelo de Ising**.

Apesar de ambos se tratarem de métodos de otimização e da similaridade dos nomes, o **Quantum Annealing** e o **Simulated Annealing** não apresentam demais relações no seu funcionamento.

SAT

O Problema de satisfatibilidade booleana consiste em determinar se existe um interpretação que satisfaz uma fórmula booleana, isto é, uma atribuição para as variáveis que torna uma expressão verdadeira.

SATish
Linguagem desenvolvida para o **Satyrus**, que engloba os conceitos discutidos na seção ”Sintaxe”.

Simulated Annealing
Conhecido também como ”Têmpera Simulada”, é um método de otimização inspirado em processos de resfriamento. Consiste em vasculhar o espaço de busca indo de um ponto (ou estado) a outro segundo um critério regido pela ”temperatura” T do sistema. Isto é, escolhe-se um estado na vizinhança do estado atual e, se este configurar uma energia mais baixa, escolhe-se o novo como atual solução. Caso contrário, ainda podemos decidir por uma solução aparentemente pior, dependendo do valor de T. Quanto maior for T, maior será a probabilidade de caminhar para um estado de maior energia. O processo é repetido iterativamente, com um decréscimo na ”temperatura” a cada etapa.

Diferentemente de outros métodos conhecidos, o **Simulated Annealing** eventualmente prefere caminhar por soluções piores com o intuito de escapar de mínimos locais.

Yacc
Acrônimo para ”*Yet another compiler compiler*”, programa que atua em conjunto com o **Lex**, e a partir de regras estabelecidas pelo programador, recebe as sequências de *tokens* como entrada e constrói árvores sintáticas. É também responsável por apontar erros de sintaxe.

Referências Bibliográficas

[1] MONTEIRO, B. F. SATyrus2: **Compilando Especificações de Raciocínio Lógico.** Dissertação (Engenharia de Sistemas e Computação) – PESC/-COPPE, UFRJ. Rio de Janeiro, 2010.

[2] Novello. C. F. **XOR via SATyrus** Dissertação (Engenharia de Sistemas e Computação) – PESC/COPPE, UFRJ. Rio de Janeiro, 2012.

[3] P.M.V. Lima, M.M.M. Morveli-Espinoza, G.C. Pereira & F.M.G. França. SATyrus: **A SAT-based Neuro-Symbolic Architecture for Constraint Processing.** December 2004, DOI: 10.1109/ICHIS.2005.97, Procs of 5th International Conference on Hybrid Intelligent Systems (HIS 2005), 6.9 November 2005, Rio de Janeiro, Brazil.

[4] P.M.V. Lima, G.C. Pereira, M.M.M. Morveli-Espinoza & F.M.G. França (2005) **Mapping and Combining Combinatorial Problems into Energy Landscapes via Pseudo-Boolean Constraints.** In: M. De Gregorio, V. Di Maio, M. Frucci & C. Musio (eds) Brain, Vision, and Artificial Intelligence. BVAI 2005. Lecture Notes in Computer Science, vol 3704. Springer, Berlin, Heidelberg

[5] D-Wave Systems Inc., **D-Wave System Documentation**, https://docs.dwavesys.com/docs/ (2019)

[6] Steven Diamond & Stephen Boyd, **CVXPY: A Python-Embedded Modeling Language for Convex Optimization**, Journal of Machine Learning Research, 2016, vol. 17, num. 83, p. 1-5

[7] Andrew Makhorin, **GLPK (GNU Linear Programming Kit)**, 2012. `https://www.gnu.org/software/glpk`

[8] David Beazley, **PLY (Python Lex-Yacc)**, 2018.
https://www.dabeaz.com/ply/

[9] J.J. Hopfield & D. W. Tank, **"Neural"Computation of Decisions in Optimization Problems**, Biological Cybernetics, 52, p. 141-152, February 1985