Projeto de Algoritmos Baseados em Florestas de Posets para o Problema de Otimização U-curve

Monografia apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
aprovação em MAC499 – Trabalho
de
Conclusão de Curso

Aluno: Gustavo Estrela de Matos

Orientador: Marcelo da Silva Reis

Centro de Toxinas, Resposta-imune e Sinalização Celular (CeTICS)

Laboratório Especial de Ciclo Celular, Instituto Butantan

São Paulo, 23 de Novembro de 2017

Resumo

A fazer.

Conteúdo

Capítulo 1

Introdução

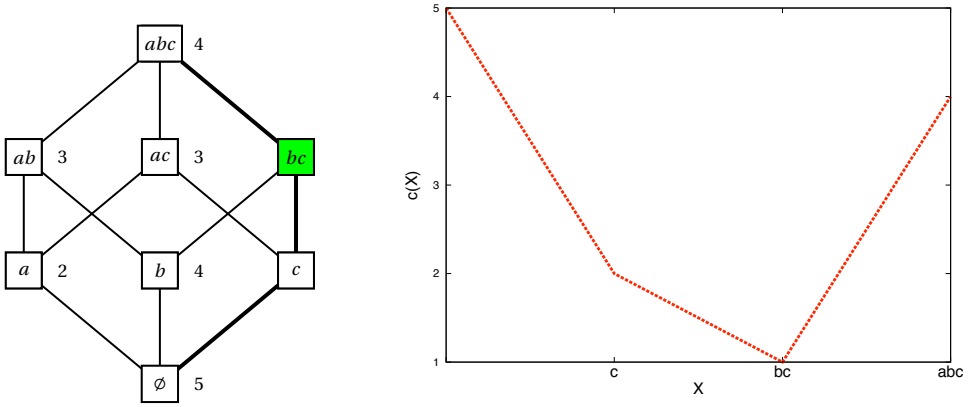
Seleção de características é uma técnica que pode ser utilizada em uma das etapas da construção de um modelo de aprendizado de máquina. Ela consiste em, dado o conjunto de características observadas nas amostras, escolher um subconjunto que seja ótimo de acordo com alguma métrica. Devemos considerar o uso de seleção de características quando a quantidade de características é muito grande, o que pode tornar o uso do modelo muito caro do ponto de vista computacional. Outra aplicação dessa técnica é em situações nas quais a quantidade de amostras é pequena comparada à complexidade do modelo original, em outras palavras, quando ocorre sobreajuste (do inglês, *overfitting*).

Mais formalmente, o problema de seleção de características consiste em um problema de otimização combinatória em que, dado um conjunto S de características, procuramos por um subconjunto $X \in \mathcal{P}(S)$ ótimo de acordo com uma função de custo $c : \mathcal{P}(S) \to \mathbb{R}_+$. É comum nas abordagens do problema explorar o fato de que o espaço de busca $\mathcal{P}(S)$ junto a relação $\subseteq$ define um reticulado Booleano [Rei12; AG+18]. No geral, a função de custo c deve ser capaz de medir quão informativas as características X são em respeito ao rótulo Y do problema de aprendizado; portanto c costuma depender da estimação da distribuição de probabilidade conjunta de (X, Y).

Quando ocorre a estimação da distribuição de probabilidade conjunta de (X, Y), o custo das cadeias do reticulado Booleano reproduzem um fenômeno conhecido em aprendizado de máquina, o das "curvas em U". Para entender intuitivamente esse fenômeno, devemos observar que conforme subimos uma cadeia do reticulado estamos aumentando o número de características sendo consideradas, portanto existem mais possíveis valores de X, permitindo descrever melhor os valores de Y; por outro lado, também precisaríamos de mais amostras para estimar bem $\mathbb{P}(X, Y)$, e, quando isso não é possível, erros de estimação fazem com que $c(X)$, isto é, o custo de X, aumente.

Podemos então considerar um caso particular do problema de seleção de características em que a função de custo descreve "curvas em U" em todas as cadeias do reticulado Booleano. Esse caso particular é conhecido como problema U-curve e existem na literatura algoritmos ótimos para esse problema como o `U-Curve Branch and Bound (UBB)`, `U-Curve-Search (UCS)` e `Poset Forest Search (PFS)` [RFB14; Rei12]. A solução do problema U-curve tem aplicações em problemas de aprendizado de máquina tais como como projeto de W-operadores [JCJB04] e preditores na estimação de Redes Gênicas Probabilísticas [Bar+07].

O problema U-Curve é NP-difícil [Rei12]; por conta deste fato, os algoritmos apresentados até então na literatura têm limitações tanto do ponto de vista de tempo de computação quanto do uso de memória. Dentre estes algoritmos, destacamos o PFS, que foi criado como um melhoramento do algoritmo UBB.



(a) (b)

Figura 1.1: Exemplo de instância do problema U-curve em que o conjunto de características é $S = \{a, b, c\}$. A figura 1.1(a) representa o diagrama de Hasse do reticulado Booleano $(\mathcal{P}(S), \subseteq)$, anotando ao lado de cada conjunto de características o seu custo. Os custos dos elementos da cadeia $\{\emptyset, c, bc, abc\}$, marcada em negrito, são apresentados na figura 1.1(b). O subconjunto $\{b, c\}$, marcado em verde, tem custo mínimo na cadeia em negrito e também no reticulado inteiro e, portanto é a solução ótima para esta instância. Imagem retirada de [Rei12] com permissão do autor.

O `UBB` é um algoritmo *branch-and-bound* que, a partir de uma enumeração, representa o espaço de busca como uma árvore e procura pelo mínimo global fazendo uma espécie de busca em profundidade que percorre as cadeias da árvore do espaço de busca, podando nós (e consequentemente seus descendentes) sempre que a função de custo cresce. Este algoritmo é unidirecional no sentido de que a busca em profundidade percorre as cadeias da árvore de baixo para cima, portanto se o custo dos elementos de uma cadeia nunca crescem então todos elementos desta serão visitados. A limitação deste algoritmo é evidente quando a função de custo usada é monótona não-crescente, pois isto implica que a condição de poda nunca será verdadeira, fazendo com que todo o espaço de busca seja percorrido.

O algoritmo `PFS` contorna esta limitação porque é bidirecional. Para fazer isto, ele precisa representar o espaço de busca de duas maneiras diferentes: uma que é similar ao que o `UBB` faz, para os percorrimentos de baixo para cima, e outra que deve ser uma representação equivalente a primeira para o reticulado Booleano dual $(\mathcal{P}(S), \supseteq)$, para os percorrimentos de cima para baixo; ambas representações são feitas com florestas de posets, em uma estrutura de dados capaz de armazenar raízes e adjacências dos nós. Uma iteração do `PFS` é constituída das seguintes etapas: escolha de uma direção de percorrimento; escolha de uma raíz na floresta escolhida; ramificação (percorrimento de uma cadeia); poda na floresta escolhida; e por último, atualização da floresta dual a escolhida para que ambas representem o mesmo espaço de busca.

Existem pontos do algoritmo `PFS` que ainda não foram explorados com o intuito de melhorar seu desempenho. Dentre eles, a escolha de raízes para etapa de ramificação, que é feita de maneira arbitrária atualmente; o uso de outras estruturas de dados para representação das florestas, como por exemplo diagramas de decisão binárias ordenados (*Ordered Binary Decision Diagrams* - OBDDs) [Bry86]; e também a paralelização do código, o que parece trazer ganhos no tempo de execução do algoritmo dado que, como as árvores do espaço de busca são disjuntas, a etapa de ramificação pode ser realizada de maneira paralela com pouca informação compartilhada entre threads.

1.1 Objetivos do Trabalho

Podemos dividir os objetivos deste trabalho em objetivos gerais e específicos.

Objetivos gerais:

1. Criar algoritmos para o problema U-curve que sejam mais eficientes em consumo de tempo e/ou de memória do que as presentes soluções;

2. Verificar a qualidade das soluções encontradas no desenvolvimento de modelos de Aprendizado Computacional.

Objetivos específicos:

- Estudar o algoritmo `Poset Forest Search (PFS)`;
- Modificar a etapa de ramificação do algoritmo `PFS` e avaliar as mudanças na dinâmica do algoritmo;
- Paralelizar o algoritmo `PFS`, com as modificações feitas na etapa de ramificação (se houver melhorias com tal mudança);
- Criar um novo algoritmo, de natureza paralela e facilmente combinável com outros algoritmos, para o problema U-Curve (o algoritmo `PUCS`);
- Avaliar o consumo de recursos computacionais dos algoritmos criados, comparando com os algoritmos já presentes na literatura como o `UBB`;
- Avaliar os conjuntos de características selecionados por cada algoritmo na seleção de modelos de aprendizado computacional, usando como exemplo conjuntos de dados do repositório UCI Machine Learning Repository.

1.2 Organização do Trabalho

Além deste capítulo, de introdução, este trabalho é organizado em outros cinco capítulos, que são os seguintes:

- Capítulo 2: faremos uma revisão dos conceitos fundamentais para desenvolvimento deste trabalho. Definiremos o problema de seleção de características e mostraremos uma função de custo baseada em entropia para o mesmo. Explicaremos o fenômeno das curvas em U em funções de custo utilizadas na seleção de características e definiremos o problema de otimização U-curve. Por fim, apresentaremos funções de custo usadas na criação de instâncias artificiais do problema U-Curve.

- Capítulo 3: descreveremos dois algoritmos ótimos para o problema U-curve, o `U-curve--Branch-and-Bound` (`UBB`) e um algoritmo semelhante e mais geral, o `Poset-Forest--Search` (`PFS`). Em seguida, apresentaremos modificações no `PFS` a fim de melhorar seu desempenho. Além disso, vamos mostrar um novo algoritmo baseado nos dois últimos que permite fácil paralelização. Faremos testes com estas modificações e novos algoritmos com instâncias artificias do problema U-curve.

- Capítulo 4: apresentaremos o `Parallel-U-Curve-Search`, um algoritmo de divisão em conquista paralelizado para o problema U-curve. Mostraremos seu funcionamento e testaremos seu desempenho em instâncias artificiais do problema U-curve.
- Capítulo 5 aplicaremos a seleção de características na seleção de modelos de aprendizado computacional. Os modelos criados serão treinados e validados com conjuntos de dados reais de aprendizado de máquina, disponíveis no repositório UCI Machine Learning Repository.
- Capítulo 6 revisaremos o conteúdo deste trabalho, mostrando as diferentes abordagens feitas para enfrentar o problema U-curve assim como os resultados obtidos. Além disso, mostraremos possíveis linhas que podem ser tomadas em trabalhos futuros que se relacionam com este.

Capítulo 2

Conceitos Fundamentais

2.1 O problema de seleção de características

A seleção de características é um problema de otimização combinatória em que procuramos o melhor subconjunto de um conjunto de características S. O espaço de busca desse problema é o conjunto potência de S, $\mathcal{P}(S)$, que é a coleção de todos os subconjuntos possíveis de S. A função de custo desse problema é uma função $c : \mathcal{P}(S) \to \mathbb{R}_+$.

Definição 2.1.1 (Problema de seleção de características)**.** *Seja S um conjunto de características, finito e não vazio, e c uma função de custo. Encontrar $X \in \mathcal{P}(S)$ tal que $c(X) \leq c(Y)$, $\forall Y \in \mathcal{P}(S)$.*

O espaço de busca do problema de seleção de características possui uma relação de ordem parcial definida pela relação $\subseteq$, portanto este conjunto é **parcialmente ordenado (poset)**.

Definição 2.1.2. *Uma **cadeia** do reticulado booleano é uma sequência X_1, X_2, ..., X_l tal que $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_l$.*

2.2 Funções de custo

A função de custo utilizada na solução do problema deve, de alguma forma, refletir a qualidade do conjunto de características avaliado. Por isso, diferentes aplicações de seleção de características podem ter diferentes funções de custo. No contexto de aprendizado de máquina, uma possível função de custo é a entropia condicional média (MCE), que já foi utilizada por exemplo na construção de W-operadores [DMJ06].

Definição 2.2.1. *Dado um problema de aprendizado em que Y é o conjunto de possíveis rótulos e $W = (w_1, ..., w_n)$, com $w_i \in A_i$, é o conjunto de variáveis. Seja $W' = (w_{I(1)}, w_{I(2)}, ..., w_{I(k)})$ um conjunto de variáveis (características) escolhidas, $\mathbf{X}$ uma vetor aleatório de tamanho k com $X_j \in A_{I(j)}$, e $log 0 = 0$. Então, a **entropia condicional** de Y dado $\mathbf{X} = \mathbf{x}$ é:*

$$H(Y|\mathbf{X} = \mathbf{x}) = -\sum_{y \in Y} \mathbb{P}(Y = y|\mathbf{X} = \mathbf{x}) log \mathbb{P}(Y = y|\mathbf{X} = \mathbf{x}).$$

Definição 2.2.2. *Sob o mesmo contexto definido em 2.2.1, definimos a **entropia condicional média** como:*

$$\mathbb{E}[H(Y|\mathbf{X})] = \sum_{\mathbf{x} \in \mathbf{X}} H(Y|\mathbf{X} = \mathbf{x}) \mathbb{P}(\mathbf{X} = \mathbf{x}).$$

A função H, em teoria da informação, mede o inverso da quantidade média de informação que uma variável tem. Esta função atinge valor máximo quando a distribuição de probabilidade da variável aleatória em questão é uniforme (todos valores que ela pode assumir são equiprováveis), e tem valores baixos quando essa distribuição é concentrada.

Problemas de aprendizado em que os rótulos tem uma distribuição concentrada são mais fáceis do que os problemas em que essa distribuição é menos concentrada. Tome como exemplo o problema de classificar o lançamento de uma moeda $\mathbf{x}$ em y (cara ou coroa); se toda moeda $\mathbf{x}$ é não viciada, então a distribuição de $\mathbb{P}(y|\mathbf{x})$ é pouco concentrada, por outro lado, quando a moeda é viciada, a distribuição de $\mathbb{P}(y|\mathbf{x})$ é concentrada e é mais fácil classificar este problema. Em termos mais formais, o erro do melhor classificador do problema mais fácil é menor do que o erro do melhor classificador do problema mais difícil.

Portanto, como a função H é capaz de medir a concentração da distribuição de Y dado $\mathbf{X} = \mathbf{x}$, e quanto maior esta concentração mais fácil é o modelo de aprendizado, podemos dizer que a função de custo $\mathbb{E}[H(Y|\mathbf{X})]$ pode representar a qualidade do modelo de classificação que usa o conjunto de características de $\mathbf{X}$.

Agora, como já entendemos o funcionamento da função de custo MCE e como ela se relaciona com a qualidade do conjunto de características avaliado, vamos entender o que acontece no modelo de aprendizado e na função de custo que usamos como exemplo quando percorremos uma cadeia do reticulado.

Uma cadeia do poset pode ser vista como uma sequência de possíveis escolhas de conjuntos de características ao qual a cada passo adicionamos uma característica. Isso significa que a cada passo dado a variável $\mathbf{x}$ ganha uma componente a mais. Quando estamos no início da cadeia, poucas variáveis do problema são consideradas, portanto há uma grande abstração dos dados dos objetos sendo classificados, e conforme subimos uma cadeia, diminuímos a abstração dos dados e isso faz com que a distribuição de Y dado $\mathbf{x}$ se concentre.

Essa concentração da distribuição da probabilidade indica que o custo dos subconjuntos deve diminuir conforme subimos por uma cadeia do reticulado, e este raciocínio nos leva a pensar que adicionar características sempre melhora a classificação; de fato, o valor de $\mathbb{E}[H(Y|\mathbf{X})]$ deve diminuir (até algum ponto de saturação) conforme aumentamos o número de variáveis do problema. Mas se isso é verdade, por que fazemos seleção de características? A inconsistência entre esse raciocínio e a motivação para seleção de característica é que essa linha de raciocínio negligenciou o fato de que problemas de classificação (supervisionada) dependem de uma amostra da distribuição de Y dado $\mathbf{X} = \mathbf{x}$, ou seja, não sabemos nem ao menos calcular $H(Y|\mathbf{X} = \mathbf{x})$, podemos apenas estimar o seu valor a partir da amostra.

A amostra da distribuição de Y dado $\mathbf{X} = \mathbf{x}$ é obtida do conjunto de treinamento do problema de aprendizado e quando o número de amostras não é grande o suficiente a qualidade do classificador é comprometida. Além disso, o número de amostras necessárias deve crescer conforme aumentamos a complexidade do modelo de aprendizado utilizado. Considerando que quando subimos uma cadeia do reticulado booleano estamos aumentando a complexidade do modelo, temos que, a partir de um certo ponto, a qualidade do classificador que utiliza tal conjunto de características deve piorar.

Portanto, é esperado que a função de custo descreva um formato de U nas cadeias do reticulado. No começo da cadeia, o custo deve diminuir por conta da maior granularidade dos dados de entrada, até algum ponto onde a limitação no número de amostras combinada com o aumento da complexidade do modelo causem erros de estimação que aumentam o erro do classificador criado em tal modelo.

No cálculo da entropia condicional média, o efeito do aumento da complexidade de $\mathbf{X}$ é a estimação ruim de $\mathbb{P}(Y = y|\mathbf{X} = \mathbf{x})$. Contorna-se este problema modificando a entropia

condicional média para penalizar a entropia de Y quando $\mathbf{x}$ foi observado poucas vezes. A função de custo utilizada é, então:

$$\hat{\mathbb{E}}[H(Y|\mathbf{X})] = \frac{N}{t} \sum_{\mathbf{x} \in \mathbf{X}} H(Y|\mathbf{X} = \mathbf{x}) \mathbb{P}(\mathbf{X} = \mathbf{x}). \tag{2.1}$$

2.3 O problema U-curve

A função de custo apresentada na seção 2.2 descrevem curvas que tem um formato em U (a menos de oscilações) nas cadeias do reticulado booleano, vamos definir esta propriedade agora.

Definição 2.3.1. *Uma cadeia é dita* ***maximal*** *se não existe outra cadeia no reticulado que contenha propriamente esta cadeia.*

Definição 2.3.2. *Uma função de custo c é dita* ***decomponível em curvas U*** *se para toda cadeia maximal $X_1, ..., X_l$, $c(X_j) \leq max\{c(X_i), c(X_k)\}$ sempre que $X_i \subseteq X_j \subseteq X_k$, $i, j, k \in \{1, ..., l\}$.*

Vamos considerar então o problema de seleção de características em que a função de custo utilizada é decomponível em curvas U. Este é o problema central deste trabalho.

Definição 2.3.3 (Problema U-Curve)**.** *Dados um conjunto finito e não-vazio S e uma função de custo c decomponível em curvas em U, encontrar um subconjunto $X \in \mathcal{P}(S)$ tal que $c(X) \leq c(Y)$, $\forall Y \in \mathcal{P}(S)$.*

O problema U-Curve é um caso particular do problema de seleção de características com uma propriedade que nos permite achar o mínimo global sem a necessidade de avaliar cada ponto do reticulado booleano. Isso é possível porque a propriedade U-Curve (da decomponibilidade da função de custo em curvas U) nos garante que o custo dos elementos de uma cadeia não podem cair uma vez que aumentaram. Sejam por exemplo dois elementos $A \subseteq B$ de $\mathcal{P}(S)$, então:

- se $c(B) > c(A)$, então $c(X) > c(A)$ para todo X do intervalo $[B, \mathcal{P}(S)]$;
- se $c(A) > c(B)$, então $c(X) > c(B)$ para todo X do intervalo $[\emptyset, A]$.

Desta maneira, quando um problema de seleção de características tem uma função de custo decomponível em curvas U a menos de algumas oscilações, é vantajoso aproximar a solução deste problema pela solução encontrada por um algoritmo de busca do problema U-Curve. Tal abordagem não é ótima, porém, como existem poucas oscilações da função de custo, é provável que a solução encontrada ainda seja próxima da melhor solução.

2.4 Funções de custo artificiais

Para testar e comparar o desempenho de algoritmos precisamos de muitas instâncias do problema U-Curve. Usar instâncias reais para este fim pode ser inviável devido a escassez de dados e, além disso, é possível isto cause uma análise viesada para os dados usados. Por isso é necessário usar funções de custo que nos permitam criar instâncias artificiais parecidas com problemas reais e que sejam tão gerais quanto possível, evitando que a avaliação dos algoritmos seja viesada.

Nesta seção apresentamos duas funções de custo artificiais, a primeira foi utilizada na maioria dos testes de desempenho dos algoritmos, servindo de base para avaliar tempo de execução,

otimalidade e número de chamadas da função custo. A segunda não foi utilizada em testes deste trabalho, mas foi objeto de estudos pois, diferente da primeira, é capaz de simular violações da hipótese de curva U, o que é esperado que ocorra mesmo que moderadamente em instâncias reais.

2.4.1 Soma de subconjuntos

Para se avaliar o desempenho dos algoritmos criados neste trabalho, utilizamos instâncias artificiais que são reduções do problema da soma de subconjuntos. Este problema consiste em, dado um conjunto finito de inteiros não-negativos S e um inteiro não-negativo t, descobrir se há um subconjunto de S que soma t. Podemos resolver este problema com a solução de uma instância do problema de seleção de características onde o conjunto de características é S' uma cópia de S e a função de custo é c:

$$c(X) = |t - \sum_{x \in X} x|, \text{ para todo } X \in \mathcal{P}(S'). \tag{2.2}$$

Assim como a função de custo MCE, a função de custo de somas de subconjuntos também apresenta formato de curva em U nas cadeias do reticulado Booleano [Rei12]. Para toda cadeia com elementos $A \subseteq B \subseteq C$ vale que $c(B) \leq max\{c(A), c(B)\}$, então de fato esta função é decomponível em curvas U. Vamos apresentar a prova desta propriedade, feita em [Rei12]. Começamos a demonstração definindo $D = B \setminus A$ e $E = C \setminus B$. Este problema tem dois casos disjuntos: $|t - \sum_{b \in B} b| > 0$ ou então $|t - \sum_{b \in B} b| \leq 0$.

- se $|t - \sum_{b \in B} b| > 0$, então:

$$\begin{aligned} c(B) &= |t - \sum_{b \in B} b| \\ &\leq |t - \sum_{b \in B} b + \sum_{d \in D} d| \quad \text{(pois } S \text{ contém apenas números positivos e } t - \sum_{b \in B} b > 0) \\ &= |t - \sum_{a \in B \setminus D} a| \\ &= |t - \sum_{a \in A} a| \\ &= c(A). \end{aligned}$$

  Portanto, $c(B) \leq c(A)$, logo $c(B) \leq max\{c(A), c(C)\}$.

- se $|t - \sum_{b \in B} b| \leq 0$, então:

$$\begin{aligned} c(B) &= |t - \sum_{b \in B} b| \\ &\leq |t - \sum_{b \in B} b - \sum_{e \in E} e| \quad \text{(pois } S \text{ contém apenas números positivos e } t - \sum_{b \in B} b \leq 0) \\ &= |t - \sum_{c \in B \cup E} c| \\ &= |t - \sum_{c \in C} c| \\ &= c(C). \end{aligned}$$

Portanto, $c(B) \leq c(C)$, logo $c(B) \leq max\{c(A), c(C)\}$.

Como acabamos de provar para os dois casos possíveis, temos que $c(B) \leq max\{c(A), c(C)\}$. Desta maneira, a função de custo de soma de subconjuntos é decomponível em curvas U. □

2.4.2 Violação da curva U

Como vimos na seção 2.2.1, funções de custo utilizadas em problemas de seleção de características podem descrever curvas em formato de U nas cadeias do reticulado Booleano, e isto nos permite aproximar a solução destes problemas assumindo que a função de custo é decomponível em curvas U. Entretanto, é possível que as funções de custo apresentem oscilações, ou seja, mínimos locais. Por isso, é interessante estudar a robustez dos algoritmos que resolvem o problema U-Curve quando a hipótese de curva em U não é verdadeira. Apesar de não analisarmos os algoritmos deste trabalho quanto sua robustez, achamos importante citar esta métrica de qualidade.

Uma maneira de gerar funções com oscilações e violações da curva em U é adicionar uma perturbação senoidal a uma função decomponível em curvas U. Seja a seguinte função:

$$c(X|X_0, \mathbf{W}, c_{max}) = c_{max}[1 - e^{-\frac{1}{2}(X-X_0)^T \mathbf{W}(X-X_0)}],$$

em que:

$X \in \mathcal{P}(S)$ é um conjunto de características;
$X_0 \in \mathcal{P}(S)$ é o conjunto de custo mínimo;
$c_{max} \in \mathbb{R}$ é uma constante que escala o custo máximo de um subconjunto;
$\mathbf{W}$ é uma matriz positiva-definida de pesos que dá forma a função.

A função de custo acima é decomponível em curvas em U. Podemos então adicionar à mesma um ruído senoidal, para esse fim generalizando a função com o acréscimo de um termo:

$$c(X|X_0, \mathbf{W}, c_{max}) = c_{max}[1 - e^{-\frac{1}{2}(X-X_0)^T \mathbf{W}(X-X_0)}] + A\cos(2\pi f \beta(X)), \tag{2.3}$$

onde $\beta(X) = \frac{1}{n}\sum_{i=1}^{n} X(i)$ e $A \in \mathbb{R}$ é uma constante que regula a amplitude do ruído, ou seja, a profundidade dos mínimos locais; e $f \in \mathbb{R}$ controla a frequência do ruído, isto é, qual a distância entre os mínimos locais gerados.

A função de custo da equação 2.3 foi utilizada por Atashpaz-Gargari e colegas para avaliar a robustez de um algoritmo para o problema U-curve [AG+18]. Esta função já encontra-se implementada no arcabouço featsel e será utilizada em experimentos computacionais futuros.

Capítulo 3

Melhoramentos no algoritmo `Poset-Forest-Search` (`PFS`)

Neste capítulo apresentamos o algoritmo `Poset-Forest-Search` (`PFS`), um algoritmo ótimo para o problema U-curve que foi criado para enfrentar a limitação do algoritmo `U-curve Branch and Bound` (`UBB`) ser unidirecional. Apesar do `PFS` ter solucionado tal problema com sucesso, este algoritmo apresenta pontos que ainda podem ser explorados para se criar uma modificação que tenha melhor desempenho computacional. Modificaremos então o `PFS`, explorando tais pontos e, além disso, vamos criar uma versão paralela do algoritmo.

Baseamos nosso trabalho no código-fonte do arcabouço featsel [Rei+17], que é software livre e está disponível no GitHub sob a licença de uso *GNU General Public License*. Todos os experimentos computacionais deste capítulo foram feitos utilizando algoritmos e funções de custo implementados no featsel, em uma servidora com 64 núcleos, 256 GB de memória RAM e sistema operacional Ubuntu server 14.04 LTS. Em todos os experimentos foi utilizada como função de custo a redução polinomial do problema da soma de subconjuntos (equação 2.2); instâncias artificiais foram geradas escolhendo aleatoriamente $n+1$ números inteiros, onde n é o número de características.

3.1 Descrição do algoritmo

3.1.1 O caso simples: o algoritmo `U-curve-Branch-and-Bound` (`UBB`)

O algoritmo `U-curve Branch and Bound` (`UBB`), que é uma versão simplificada do `PFS`, percorre o espaço de busca fazendo uma busca em profundidade em uma árvore que é subgrafo do diagrama de Hasse do reticulado Booleano $(\mathcal{P}(S), \subseteq)$. Esta árvore é definida por aplicações recursivas do seguinte lema:

Lema 3.1.1. *Sejam X e Y conjuntos, X não-vazio e x_i o i-ésimo elemento de X. Seja $X_0 \supseteq X_1 \supseteq \cdots \supseteq X_{|X|}$ uma cadeia tal que $X_0 = X$, $X_{|X|} = \emptyset$ e $X_i \cup \{x_i\} = X_{i-1}$ para todo $0 < i \leq |X|$. Vale que:*

$$\{Y\} \cup \bigcup_{i=1}^{|X|} \{W \cup Y \cup \{x_i\} : W \in \mathcal{P}(X_i)\} = \{W \cup Y : W \in \mathcal{P}(X)\}.$$

Demonstração. Faremos uma prova por indução no tamanho de X de maneira similar a Reis [Rei12].

- Suponha que $|X| = 1$. Então:

$$\begin{aligned}\{Y\} \cup \bigcup_{i=1}^{1} \{W \cup Y \cup \{x_i\} : W \in \mathcal{P}(X_i)\} &= \{Y\} \cup \{W \cup Y \cup \{x_1\} : W \in \mathcal{P}(X_1)\} \\ &= \{Y\} \cup \{Y \cup \{x_1\}\} \qquad \text{(Como } X_1 = \emptyset\text{)} \\ &= \{Y, Y \cup \{x_1\}\} \\ &= \{W \cup Y : W \in \mathcal{P}(X)\}.\end{aligned}$$

- Suponha que o lema é verdadeiro para todo X com $|X| < k$, então:

$$\{Y\} \cup \bigcup_{i=1}^{k} \{W \cup Y \cup \{x_i\} : W \in \mathcal{P}(X_i)\} =$$

$$\{Y\} \cup \bigcup_{i=2}^{k} \{W \cup Y \cup \{x_i\} : W \in \mathcal{P}(X_i)\} \cup \{W \cup Y \cup \{x_1\} : W \in \mathcal{P}(X_1)\}.$$

Seja $Z = Z_0 = X_1, Z_1 = X_2, \ldots Z_{|Z|} = X_{|X|}$, então $|Z| = k - 1$ e $z_1 = x_2, z_2 = x_3, \ldots, z_{|Z|} = x_{|X|}$, e:

$$\{Y\} \cup \bigcup_{i=2}^{k} \{W \cup Y \cup \{x_i\} : W \in \mathcal{P}(X_i)\} \cup \{W \cup Y \cup \{x_1\} : W \in \mathcal{P}(X_1)\} =$$

$$\{Y\} \cup \bigcup_{j=1}^{k-1} \{W \cup Y \cup \{z_j\} : W \in \mathcal{P}(Z_i)\} \cup \{W \cup Y \cup \{x_1\} : W \in \mathcal{P}(X_1)\} =$$

(Pela hipótese de indução)

$$\begin{aligned}&\{Y \cup W : W \in \mathcal{P}(Z)\} \cup \{W \cup Y \cup \{x_1\} : W \in \mathcal{P}(X_1)\} = \\ &\{Y \cup W : W \in \mathcal{P}(X_1)\} \cup \{W \cup Y \cup \{x_1\} : W \in \mathcal{P}(X_1)\} = \\ &\{Y \cup W : W \in \mathcal{P}(X_1 \cup x_1)\} = \\ &\{Y \cup W : W \in \mathcal{P}(X)\}.\end{aligned}$$

□

Para representar o espaço de busca como uma árvore, devemos aplicar o lema da seguinte forma. Vamos utilizar o conjunto X_Y para determinar para cada nó Y do reticulado quais são os nós alcançáveis por ele, de maneira que um nó Y pode alcançar todo nó do intervalo $[Y, X_Y \cup Y]$. Iniciamos a construção da árvore com a base da aplicação recursiva do lema, indicando que $X_{Y=\emptyset} = S$, pois $\emptyset$ deve ser a raiz da árvore e deve alcançar qualquer outro nó. Agora suponha que estamos em um nó Y, então definimos $X_0 = X_Y$, $X_{Y_i} \cup \{x_i\} = X_{Y_{i-1}}$, e $Y_i = Y \cup \{x_i\}$ para $x_i \in X_Y$; então para criar a sub-árvore com raiz Y basta adicionar os arcos (Y, Y_i) para cada i e aplicar o lema recursivamente para cada Y_i e X_{Y_i}. A figura 3.1 mostra uma árvore arbitrária gerada pela aplicação recursiva do lema.

A dinâmica do `UBB` é simples. Aplica-se o lema 3.1.1 para percorrer o espaço de busca enquanto o custo dos subconjuntos visitados se mantém ou diminui; quando o custo aumenta, podemos podar a sub-árvore que se inicia no nó onde o custo cresce. Por exemplo, em uma instância do problema U-curve com três características, se o custo do nó $Y = \{100\}$ é maior do que o custo de $\{000\}$ e, ao visitar Y, $X = \{011\}$ então podemos remover do espaço de busca todos os nós do intervalo $[100, 111]$. O funcionamento do `UBB` é apresentado no pseudo-código 1.

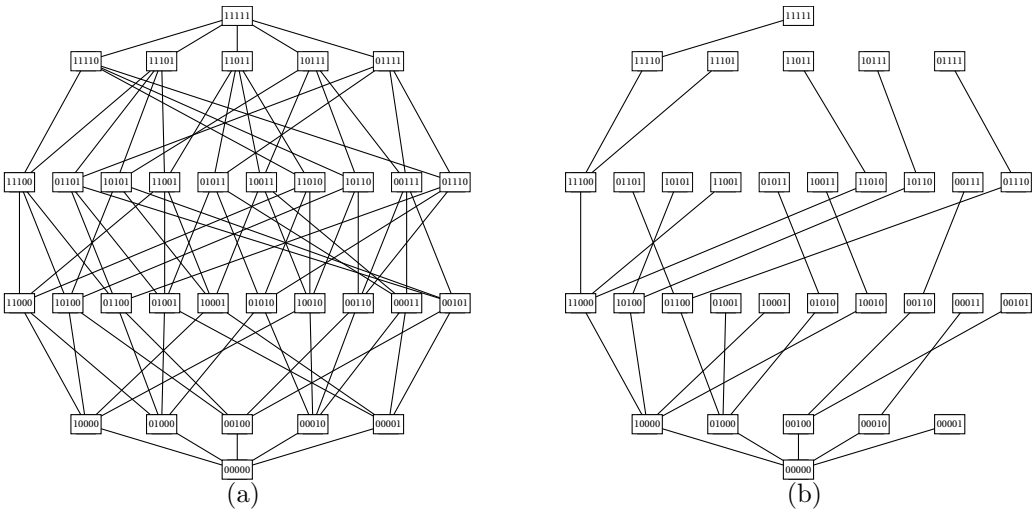


Figura 3.1: A figura 3.1(a) é o diagrama de Hasse do reticulado Booleano $(\mathcal{P}(S), \subseteq)$ e a figura 3.1(b) é uma árvore de busca definida pelo algoritmo `UBB`.

U-curve-Branch-and-Bound (S, c)
1: $\mathcal{M} \leftarrow$ Branch$(S, \emptyset, c, c(\emptyset))$
2: **return** $\{M \in \mathcal{M} : c(M)$é mínimo$\}$

Branch $(X, Y, c, cost_Y)$
1: $\mathcal{M} \leftarrow \{Y\}$
2: **while** $X \neq \emptyset$ **do**
3: remova um elemento x de X
4: $Y' \leftarrow Y \cup \{y\}$
5: $cost_{Y'} \leftarrow c(Y')$
6: **if** $cost_{Y'} \leq cost_Y$ **then**
7: $\mathcal{N} \leftarrow$ Branch$(X, Y', c, cost_{Y'})$
8: $\mathcal{M} \leftarrow \mathcal{N} \cup \mathcal{M}$
9: **end if**
10: **end while**
11: **return** $\mathcal{M}$

Algorithm 1: Pseudo-código do algoritmo `UBB`

Uma vez entendido como as podas acontecem, é fácil perceber que o `UBB` precisa percorrer uma cadeia inteira (não há podas) quando o custo nela não aumenta até o penúltimo nó da cadeia. Portanto, se a função de custo é, por exemplo, monótona não-crescente, então a condição de poda nunca será verdadeira em qualquer cadeia do reticulado, logo todo o espaço de busca será visitado, como em uma busca exaustiva. Esta é a maior limitação do algoritmo `UBB` e foi para enfrentá-la que o algoritmo `PFS` foi criado.

3.1.2 Princípios de funcionamento do algoritmo `PFS`

O algoritmo `Poset-Forest-Search` (`PFS`) é uma generalização do `UBB` capaz de percorrer o espaço de busca em duas direções, do menor para o maior elemento do reticulado (como o `UBB`) e também do maior elemento para o menor. Para fazer isso, o `PFS` inicia sua busca com

duas árvores complementares, uma gerada por aplicações recursivas do lema 3.1.1 e outra, que representa o reticulado Booleano dual $(\mathcal{P}(S), \supseteq)$, gerada por aplicações recursivas do lema a seguir.

Lema 3.1.2. *Sejam X e Y conjunto, X não-vazio e x_i o i-ésimo elemento de X. Seja $X_0 \supseteq X_1 \supseteq \cdots \supseteq X_{|X|}$ uma cadeia tal que $X_0 = X$, $X_{|X|} = \emptyset$ e $X_i \cup \{x_i\} = X_{i-1}$ para todo $0 < i \leq |X|$. Vale que:*

$$\{Y \cup X\} \cup \bigcup_{i=1}^{|X|} \{(X - (W \cup x_i)) \cup Y : W \in \mathcal{P}(X_i)\} = \{W \cup Y : W \in \mathcal{P}(X)\}.$$

Na figura 3.2 mostramos duas árvores do espaço de busca geradas pelo algoritmo. Note que estas árvores permitem o percorrimento do espaço de busca em duas direções. Seja Y um conjunto da árvore da figura 3.2(a) e X o conjunto usado para aplicação recursiva do lema 3.1.1, então os nós que compõe a sub-árvore com raiz Y é exatamente o intervalo $[Y, X \cup Y]$. Na árvore dual, apresentada na figura 3.2(a), os nós da sub-árvore com raiz Y é dado por $[Y \setminus X, Y]$.

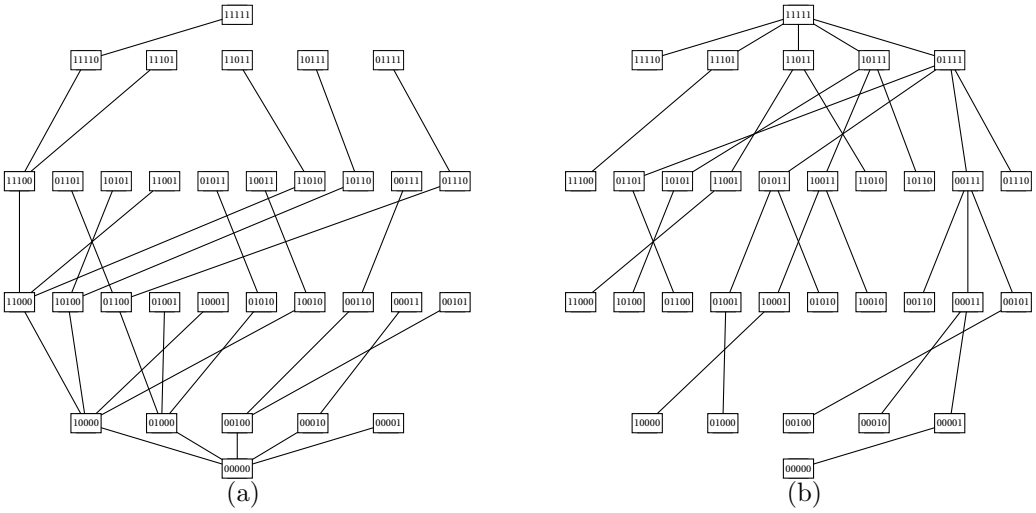


Figura 3.2: Exemplo de árvores do espaço de busca gerado pelo algoritmo `PFS`*. A figura 3.2(a) mostra a árvore gerada por aplicações recursivas do lema 3.1.1 enquanto a figura 3.2(b) mostra a árvore gerada por aplicações recursivas do lema 3.1.2.*

No algoritmo `UBB` o controle do espaço de busca pode ser facilmente implementado, pois este começa completo e para cada poda que ocorre basta não ramificar a sub-árvore em que a condição de poda for verdadeira, eliminando o intervalo $[Y, X \cup Y]$ do espaço de busca. No `PFS`, uma estratégia equivalente é capaz apenas de restringir nós da estrutura de dados que foi utilizada para o percorrimento de cadeias, ou seja, uma poda neste algoritmo implica na atualização das duas estruturas de dados que controlam o espaço de busca. Então, quando removemos um intervalo de uma árvore, precisamos remover este mesmo intervalo da árvore dual; como resultado, a árvore dual se torna uma **floresta**. Portanto, o `PFS` usa duas florestas para gerir o espaço de busca, uma com caminhos de conjuntos menores para maiores, $\mathcal{F}_A$ e outra dual, $\mathcal{F}_B$. Um exemplo de atualização de floresta dual após uma poda na floresta primal é apresentado na figura 3.3.

O `PFS` inicia escolhendo arbitrariamente uma direção de percorrimento e uma raiz da floresta correspondente a direção escolhida. O passo seguinte é a ramificação, que similar ao `UBB` percorre uma cadeia da árvore escolhida enquanto o custo dos subconjuntos não cresce e não

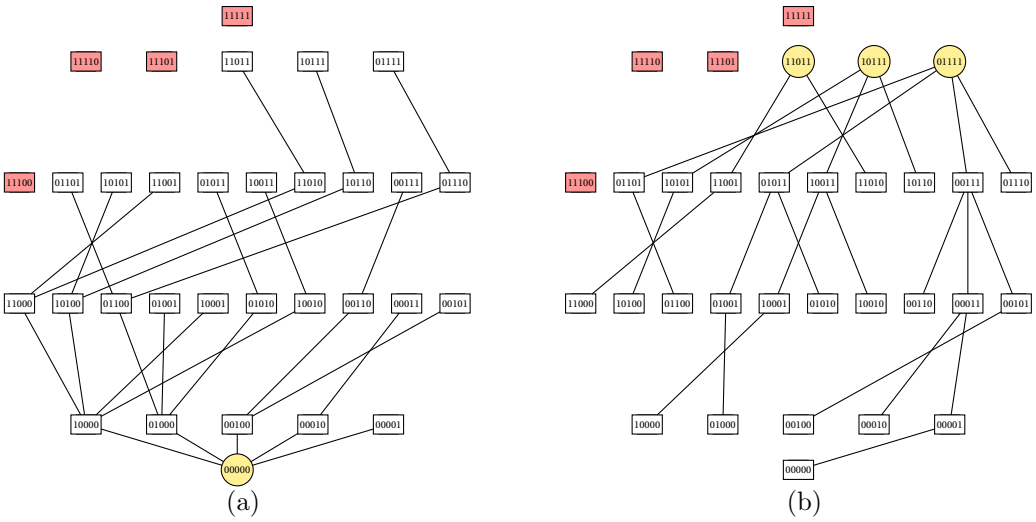


Figura 3.3: Exemplo de atualização de árvore após remoção de intervalo. Os nós em vermelho são removidos do espaço de busca enquanto os nós amarelos são raízes dos grafos. Removemos o intervalo $[11100, 11111]$ *da árvore da figura 3.3(a) e ao remover este mesmo intervalo da árvore dual, o grafo perde a raiz original* (11111) *e ganha as três raízes em amarelo.*

chegamos em uma folha da árvore. Ao fim da fase de ramificação, suponha que feita na floresta $\mathcal{F}_A$ ($\mathcal{F}_B$), teremos um intervalo $[Y, X \cup Y]$ ($[Y \setminus X, Y]$) que precisa ser removido da floresta dual. Esta atualização da floresta dual é feita de acordo com as seguintes regras.

Lema 3.1.3. *Sejam T e T' duas árvores tais que T é complementar a T'. Eliminar de T' os vértices contidos no intervalo $[Y, X \cup Y]$ é equivalente a remover de T' todos os vértices do caminho P de T com extremos Y e $X \cup Y$ e também todos os vértices de T' contidos em intervalos $[Y, B]$ tais que B contém propriamente Y e B é adjacente inferior a um vértice de P.*

Lema 3.1.4. *Sejam T e T' duas árvores tais que T é complementar a T'. Eliminar de T os vértices contidos no intervalo $[Y \setminus X, Y]$ é equivalente a eliminar de T todos os vértices do caminho P em T com extremos $Y \setminus X$ e Y e também todos os vértices de T contidos em intervalos $[A, Y]$ tais que A é contido propriamente em Y e A é adjacente superior a um vértice de P.*

As demonstrações dos lemas 3.1.3 e 3.1.4 estão disponíveis em Reis [Rei12].
O funcionamento do algoritmo deve seguir as seguintes regras:

a) *Inicialização das florestas:* as florestas $\mathcal{F}_A$ e $\mathcal{F}_B$ são iniciadas com as raízes $\emptyset$ e S respectivamente.

b) *Representação de árvores:* para todo nó do espaço de busca que não foi removido e não é raiz na floresta, a sub-árvore que se inicia neste nó está completa no espaço de buscas. Desta forma, para representar a floresta precisamos apenas armazenar as raízes da floresta e suas adjacências, pois sabemos que cada vértice adjacente é raiz de uma subárvore completa.

c) *Gerenciamento das florestas:* durante a etapa de ramificação, um vértice visitado é podado ou torna-se raiz na floresta. Com isso, temos que todas arestas de um caminho percorrido são removidas da floresta.

d) *Condição de poda (dual):* no percorrimento de uma cadeia, se o nó Y da floresta $\mathcal{F}_A$ ($\mathcal{F}_B$) tem custo maior que o último nó visitado nesta cadeia, então removemos da floresta $\mathcal{F}_A$ ($\mathcal{F}_B$) a subárvore que começa em Y, $[Y, X \cup Y]$ ($[Y \setminus X, Y]$). Se um nó Y visitado não tem conjuntos adjacentes no espaço de busca, então Y é removido do espaço de busca.

e) *Atualização de floresta:* ao podar o intervalo $[Y, X \cup Y]$ ($[Y \setminus X, Y]$) da floresta $\mathcal{F}_A$ ($\mathcal{F}_B$), devemos remover os mesmos subconjuntos da outra floresta de acordo com o lema 3.1.3 (3.1.4).

3.1.3 Pseudo-código e detalhes de implementação do `PFS`

Apresentamos nesta seção um pseudo-código para o algoritmo e ao mesmo tempo comentamos como algumas destas soluções foram implementadas em *C++* no arcabouço *featsel*.

Definição de árvores de busca

Os algoritmos que descrevemos neste capítulo dependem da aplicação recursiva dos lemas 3.1.1 e 3.1.2, e para cada ordem de características x_i escolhida na aplicação do lema obtemos uma árvore diferente para representar o espaço de busca (veja figura 3.4). A implementação de Reis possui uma enumeração sobre as características que nos permite fixar os resultados da aplicação recursiva destes lemas e também facilita o controle das árvores da floresta. Supomos então que o conjunto de características S é uma lista ordenada $\langle s_1, s_2, \ldots, s_n \rangle$ e que é possível obter o índice de um elemento. Assim, sempre que fazemos a decomposição do espaço em uma árvore (com ambos lemas) escolhemos as variáveis do conjunto X em ordem crescente de índice.

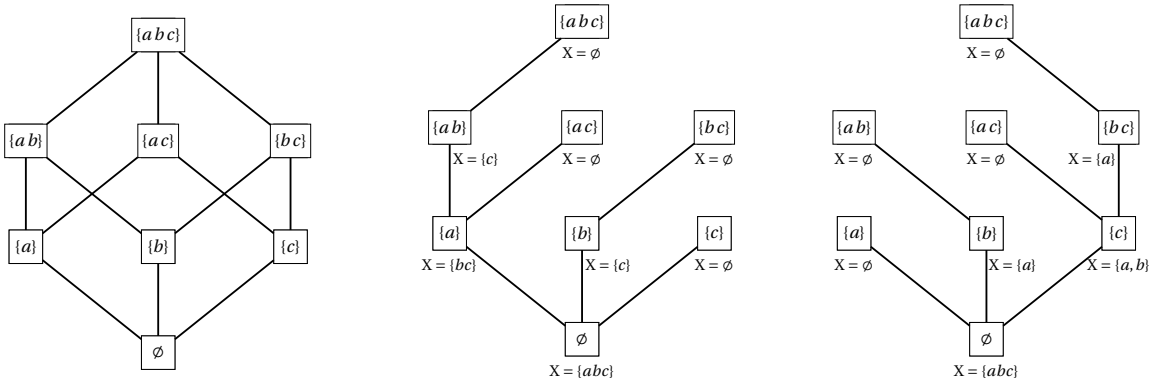


Figura 3.4: Exemplos de aplicações recursivas do lema 3.1.1 no reticulado Booleano da figura 3.4(a). O conjunto X indica para cada subconjunto Y qual é o conjunto de nós de sua sub-árvore: exatamente os nós do intervalo $[Y, X \cup Y]$. Na figura 3.4(b) a decomposição elimina de X os elementos na ordem $\langle a, b, c \rangle$ em todos os níveis de aplicação do lema, enquanto na figura 3.4(c) a ordem é $\langle c, b, a \rangle$.

Estrutura de dados

Vamos definir a estrutura de dados utilizada originalmente para representar nós das florestas. Um nó **N** é composto por quatro campos:

- `vertex`: armazena o conjunto de características que o nó representa.

- `adjacent`: armazena um conjunto de características que determina a adjacência do nó. Um nó $\mathbf{N}$ é adjacente aos conjuntos de características $\{\mathbf{N}[\texttt{vertex}] \cup \{x_i\} : x_i \in \mathbf{N}[\texttt{adjacent}]\}$.
- `leftmost`: um inteiro que define, pelo esquema de numeração, o conjunto de características X da decomposição em árvore do espaço de busca. Isto é, se $Y = \mathbf{N}[\texttt{vertex}]$ então o conjunto de nós da sub-árvore com raiz Y deve ser igual ao intervalo $[Y, X \cup Y]$, se $\mathbf{N}$ é da floresta $\mathcal{F}_A$, ou igual ao intervalo $[Y \setminus X, Y]$ se $\mathbf{N}$ é da floresta $\mathcal{F}_B$. Lembrando que assumimos uma ordem no conjunto de características ($\langle s_1, s_2, \ldots, s_n \rangle$), X é definido como $X = \bigcup_{i=\mathbf{N}[\texttt{leftmost}]}^{n} s_i$.
- `cost`: um número em ponto flutuante que armazena o custo de $\mathbf{N}[\texttt{vertex}]$.

Como descrevemos na seção anterior, uma das regras do algoritmo garante que para representar as florestas precisamos apenas armazenar as raízes das florestas e suas listas de adjacências. Desta maneira, as duas florestas $\mathcal{F}_A$ e $\mathcal{F}_B$ são representadas por conjuntos de nós. Estes nós são armazenados na estrutura de *map* da linguagem `C++`.

Pseudo-código

Vamos agora apresentar um pseudo-código do `PFS` em uma abordagem "top down", ou seja, começando pelo algoritmo principal (Algoritmo 2) e descrevendo funções conforme as mesmas são chamadas em um dado pseudo-código.

POSET-FOREST-SEARCH (S, c)
1: $\mathcal{M} \leftarrow \emptyset$
2: $\mathbf{T}[\texttt{vertex}] \leftarrow \emptyset$
3: $\mathbf{T}'[\texttt{vertex}] \leftarrow S$
4: $\mathbf{T}[\texttt{cost}] \leftarrow \mathbf{T}'[\texttt{cost}] \leftarrow \infty$
5: $\mathbf{T}[\texttt{adjacent}] \leftarrow \mathbf{T}'[\texttt{adjacent}] \leftarrow S$
6: $\mathbf{T}[\texttt{leftmost}] \leftarrow \mathbf{T}'[\texttt{leftmost}] \leftarrow 1$
7: $\mathcal{F}_A \leftarrow \{\mathbf{T}\}$
8: $\mathcal{F}_B \leftarrow \{\mathbf{T}'\}$
9: **while** $\mathcal{F}_A \neq \emptyset$ **do** ▷ equivalent to $\mathcal{F}_B \neq \emptyset$
10: *direction* $\leftarrow$ CHOOSE-DIRECTION
11: **if** *direction* **then**
12: $\langle \mathcal{N}, \mathcal{F}_A, \mathbf{N} \rangle \leftarrow$ LOWER-FOREST-BRANCH$(\mathcal{F}_A, \mathcal{F}_B, c)$
13: $\mathcal{F}_B \leftarrow$ UPPER-FOREST-PRUNNING$(\mathcal{F}_B, \mathbf{N})$
14: **else**
15: $\langle \mathcal{N}, \mathcal{F}_A, \mathbf{N} \rangle \leftarrow$ UPPER-FOREST-BRANCH$(\mathcal{F}_A, \mathcal{F}_B, c)$
16: $\mathcal{F}_A \leftarrow$ LOWER-FOREST-PRUNNING$(\mathcal{F}_A, \mathbf{N})$
17: **end if**
18: $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{N}$
19: **end while**
20: **return** $\{M \in \mathcal{M} : c(M)$ is minimum$\}$

Algorithm 2: Pseudo-código da rotina principal do algoritmo `PFS`.

Uma iteração qualquer do `PFS` começa com a inicialização das florestas do espaço de busca. A floresta $\mathcal{F}_A$, que permite percorrimentos de baixo para cima no reticulado, é inicializada com

a raiz $\emptyset$, enquanto que a floresta $\mathcal{F}_B$, de percorrimentos de cima para baixo, é inicializada com a raiz S. Assim que inicializadas as florestas, repetimos as etapas de escolha de direção de percorrimento, ramificação e poda das florestas até que ambas estejam vazias. Note que não importa qual floresta devemos verificar para terminar as iterações do algoritmo, já que ambas representam o mesmo espaço de busca, ou seja, quando uma floresta é vazia a outra também é.

A etapa de ramificação da floresta $\mathcal{F}_A$ é feita pela função Lower-Forest-Branch (Algoritmo 3) e consiste no percorrimento, a partir de uma raiz Y, de uma cadeia de sua sub-árvore. Nesta etapa, para manter válida a regra c) devemos remover todas as arestas do caminho percorrido e transformar todos os nós visitados e não podados em raízes da floresta. O percorrimento termina ao atingir um nó $\mathbf{N}$ que é folha ou que possui custo maior do que seu precedente no caminho percorrido; então o intervalo $[\mathbf{N}[\texttt{vertex}], \mathbf{N}[\texttt{vertex}] \cup \mathbf{N}[\texttt{adjacent}]]$ deve ser removido da floresta $\mathcal{F}_A$. Esta função retorna o nó $\mathbf{N}$, a floresta $\mathcal{F}_A$ atualizada, e o conjunto $\mathcal{M}$ de subconjuntos percorridos que são candidatos a mínimo.

Lower-Forest-Branch $(\mathcal{F}_A, \mathcal{F}_B, c)$
1: $\mathcal{M} \leftarrow \emptyset$
2: remova um nó $\mathbf{R}$ de $\mathcal{F}_A$
3: $\mathbf{R}[\texttt{cost}] \leftarrow$ Calc-Node-Cost$(\mathcal{F}_B, \mathbf{R})$
4: $\mathbf{M} \leftarrow \mathbf{N} \leftarrow \mathbf{R}$
5: **while** $\mathbf{N}[\texttt{cost}] \leq \mathbf{M}[\texttt{cost}]$ e $\mathbf{N}[\texttt{adjacent}] \neq \emptyset$ **do**
6: $\quad \mathbf{M} \leftarrow \mathbf{N}$
7: $\quad$ remova um elemento de s_i de $\mathbf{M}[\texttt{adjacent}]$
8: $\quad \mathcal{F}_A \leftarrow \mathcal{F}_A \cup \{\mathbf{M}\}$
9: $\quad$ crie o nó $\mathbf{N}$
10: $\quad \mathbf{N}[\texttt{vertex}] \leftarrow \mathbf{M}[\texttt{vertex}] \cup \{s_i\}$
11: $\quad \mathbf{N}[\texttt{leftmost}] \leftarrow i$
12: $\quad \mathbf{N}[\texttt{adjacent}] \leftarrow \bigcup_{i=\mathbf{N}[\texttt{leftmost}]}^{n} s_i$
13: $\quad \mathbf{N}[\texttt{cost}] \leftarrow$ Calc-Node-Cost$(\mathcal{F}_B, \mathbf{N})$
14: $\quad \mathcal{M} \leftarrow \mathcal{M} \cup \{\mathbf{N}\}$
15: **end while**
16: **return** $\langle \mathcal{M}, \mathcal{F}_A, \mathbf{N} \rangle$

Algorithm 3: Pseudo-código da função que faz o percorrimento da floresta $\mathcal{F}_A$.

Durante a execução do `PFS`, o custo de um subconjunto é calculado uma única vez; o recálculo é evitado com a função `Calc-Node-Cost` (Algoritmo 4).

Calc-Node-Cost $(\mathcal{F}, \mathbf{N})$
1: **if** $\mathbf{N}[\texttt{cost}] = \infty$ e existe $\mathbf{M} \in \mathcal{F}$ tal que $\mathbf{M}[\texttt{vertex}] = \mathbf{N}[\texttt{vertex}]$ e $\mathbf{M}[\texttt{cost}] \neq \infty$ **then**
2: $\quad$ **return** $\mathbf{M}[\texttt{cost}]$
3: **else**
4: $\quad$ **return** $c(\mathbf{R}[\texttt{vertex}])$
5: **end if**

Algorithm 4: Pseudo-código da função `Calc-Node-Cost`.

A ramificação dual, da floresta $\mathcal{F}_B$ é feita pela função Upper-Forest-Branch e é similar a função Lower-Forest-Branch ao percorrer uma cadeia da floresta de conjuntos maiores para conjuntos menores. A ramificação termina ao encontrar um nó $\mathbf{N}$ que

é folha da floresta ou tem custo maior que seu precedente no caminho; então o intervalo $[\mathbf{N}[\texttt{vertex}] \setminus \mathbf{N}[\texttt{adjacent}], \mathbf{N}[\texttt{vertex}]]$ deve ser removido da floresta $\mathcal{F}_B$ e todos os nós visitados, menos $\mathbf{N}$ tornam-se raízes. Esta função retorna o nó $\mathbf{N}$, a floresta $\mathcal{F}_B$ atualizada e o conjunto $\mathcal{M}$ de subconjuntos percorridos que são candidatos a mínimo.

A atualização da floresta $\mathcal{F}_B$ após o percorrimento e poda da floresta $\mathcal{F}_A$ é feito pela função Upper-Forest-Prunning (Algoritmo 5). Esta função recebe um nó $\mathbf{N}$ e deve remover os nós de $\mathcal{F}_B$ de acordo com a regra e). Para fazer isto, percorremos o caminho P que liga os nós de subconjuntos $\mathbf{N}[\texttt{vertex}]$ e $\mathbf{N}[\texttt{vertex}] \cup \mathbf{N}[\texttt{adjacent}]$ na floresta $\mathcal{F}_B$, removendo cada um dos nós deste caminho (se existirem, o que é verificado pela função descrita no algoritmo 6) e chamando a função Search-Lower-Children (Algoritmo 7) para verificar se os filhos destes nós devem ser podados ou não. Ao fim do caminho chamamos a função Search-Upper-Root para achar, na floresta $\mathcal{F}_B$, a raiz que alcança este caminho e atualizar a sua adjacência para garantir que uma ramificação não vai re-inserir os nós removidos. Esta função devolve a floresta $\mathcal{F}_B$ atualizada.

Uppper-Forest-Prunning $(\mathcal{F}_A, \mathcal{F}_B, \mathbf{N})$

1: $M \leftarrow \mathbf{N}[\texttt{vertex}]$
2: **while** $\mathbf{N}[\texttt{adjacent}] \neq \emptyset$ **do**
3: $\quad \mathbf{M} \leftarrow$ Get-Node$(\mathcal{F}_B, M)$
4: $\quad$ **if** $\mathbf{M} \neq NIL$ **then**
5: $\quad\quad$ remova $\mathbf{M}$ de $\mathcal{F}_B$
6: $\quad$ **end if**
7: $\quad \mathcal{F}_B \leftarrow$ Search-Lower-Children$(\mathcal{F}_B, \mathbf{M}, M, \mathbf{N}[\texttt{vertex}])$
8: $\quad$ remova o elemento s_i de $\mathbf{N}[\texttt{adjacent}]$ com maior i
9: $\quad M \leftarrow M \cup \{s_i\}$
10: **end while**
11: $\mathbf{M} \leftarrow$ Get-Node$(\mathcal{F}_B, M)$
12: **if** $\mathbf{M} = NIL$ **then**
13: $\quad$ Search-Upper-Root$(\mathcal{F}_B, M)$
14: **else**
15: $\quad$ remova $\mathbf{M}$ de $\mathcal{F}_B$
16: **end if**
17: $\mathcal{F}_B \leftarrow$ Search-Lower-Children$(\mathcal{F}_B, \mathbf{M}, M, \mathbf{N}[\texttt{vertex}])$
18: **return** $\mathcal{F}_B$

Algorithm 5: Pseudo-código da função que faz a atualização da floresta $\mathcal{F}_B$ depois de um percorrimento em $\mathcal{F}_A$.

Get-Node $(\mathcal{F}, N)$

1: **if** existe $\mathbf{N} \in \mathcal{F}$ tal que $\mathbf{N}[\texttt{vertex}] = N$ **then**
2: $\quad$ **return** $\mathbf{N}$
3: **else**
4: $\quad$ **return** NIL
5: **end if**

Algorithm 6: Pseudo-código da função `Get-Node`.

SEARCH-LOWER-CHILDREN $(\mathcal{F}_B, \mathbf{M}, M, Y)$

1: $i \leftarrow n$
2: **while** $i \geq 1$ e $s_i \in M$ **do**
3: $B \leftarrow M \setminus \{s_i\}$
4: **if** $B \supseteq Y$ **then**
5: $\mathbf{B} \leftarrow$ GET-NODE$(\mathcal{F}_B, M)$
6: **if** $\mathbf{B} \neq NIL$ **then**
7: remova $\mathbf{B}$ de $\mathcal{F}_B$
8: **end if**
9: **else**
10: crie o nó $\mathbf{B}$
11: $\mathbf{B}[\texttt{vertex}] \leftarrow B$
12: $\mathbf{B}[\texttt{leftmost}] \leftarrow i + 1$
13: $\mathbf{B}[\texttt{adjacent}] \leftarrow \bigcup_{i=\mathbf{B}[\texttt{leftmost}]}^{n} s_i$
14: $\mathbf{B}[\texttt{cost}] \leftarrow \infty$
15: $\mathcal{F}_B \leftarrow \mathcal{F}_B \cup \{\mathbf{B}\}$
16: **end if**
17: $i \leftarrow i - 1$
18: **return** $\mathcal{F}_B$
19: **end while**

Algorithm 7: Pseudo-código da função `Search-Lower-Children`.

A atualização dual, da floresta $\mathcal{F}_A$, após o percorrimento e poda da floresta $\mathcal{F}_B$ é feito pela função LOWER-FOREST-PRUNING e seu funcionamento é parecido com o de UPPER-FOREST-PRUNING. O caminho P de $\mathcal{F}_A$ que deve ser removido tem as pontas $\mathbf{N}[\texttt{vertex}] \setminus \mathbf{N}[\texttt{adjacent}]$ e $\mathbf{N}[\texttt{vertex}]$, e a função SEARCH-UPPER-CHILDREN é chamada para verificar se os filhos destes nós devem ser podados ou não. Ao fim do caminho chamamos a função SEARCH-LOWER-ROOT para achar, na floresta $\mathcal{F}_A$, a raiz que alcança este caminho e atualizar a sua adjacência para garantir que uma ramificação não vai re-inserir os nós removidos. Esta função devolve a floresta $\mathcal{F}_A$ atualizada.

A função SEARCH-LOWER-CHILDREN é chamada para cada nó M do caminho P, como descrito no lema 3.1.3. Esta função verifica para todos os filhos B de M se B contém propriamente o conjunto $\mathbf{N}[\texttt{vertex}]$; se contém, o nó de subconjunto B deve ser removido da floresta, caso contrário, um nó que representa este conjunto deve ser tornar raiz da floresta, pois o pai de B está sendo removido da floresta e isto significa que não haverá raiz que alcance B. Esta função devolve a floresta $\mathcal{F}_B$ atualizada. SEARCH-UPPER-CHILDREN é a função dual de SEARCH-LOWER-CHILDREN e faz um procedimento similar, para cada nó $\mathbf{M}$ do caminho P, como descrito no lema 3.1.4. Esta função verifica para todos os filhos A de M se A é contido propriamente pelo conjunto $\mathbf{N}[\texttt{vertex}]$; se é, o nó de subconjunto A deve ser removido da floresta, caso contrário, o um nó que representa o conjunto A deve se tornar raiz da floresta. Esta função devolve a floresta $\mathcal{F}_A$ atualizada.

Por fim, a função SEARCH-UPPER-ROOT (Algoritmo 8) recebe um subconjunto M e a floresta $\mathcal{F}_B$. O objetivo desta função é remover arestas da floresta que podem levar ao conjunto M. Para isto, a função deve percorrer um caminho de M até uma raiz da floresta (de conjuntos menores para maiores), e esta raiz deve ter sua lista de adjacência atualizada, removendo a aresta que a comunica com M. Para garantir que removemos todas as arestas, devemos criar raízes para todos nós deste caminho mas sem a adjacência que os comunica com M. O funcionamento de SEARCH-LOWER-ROOT é similar, mas deve ocorrer em outra direção (de

conjuntos maiores para menores).

SEARCH-UPPER-ROOT $(\mathcal{F}_B, M)$

1: $i \leftarrow n$
2: **while** $i \geq 1$ e $s_i \in M$ **do**
3: $\quad i \leftarrow i - 1$
4: **end while**
5: **while** $i \geq 1$ **do**
6: $\quad m \leftarrow s_i$
7: $\quad M \leftarrow M \cup \{m\}$
8: $\quad \mathbf{M} \leftarrow$ GET-NODE$(\mathcal{F}_B, M)$
9: $\quad$ **if** $\mathbf{M} \neq NIL$ **then**
10: $\quad\quad \mathbf{M}[\texttt{adjacent}] \leftarrow \mathbf{M}[\texttt{adjacent}] \setminus \{m\}$
11: $\quad\quad i \leftarrow 0$
12: $\quad$ **else**
13: $\quad\quad$ crie o nó $\mathbf{M}$
14: $\quad\quad$ **while** $i \geq 1$ e $s_i \in M$ **do**
15: $\quad\quad\quad i \leftarrow i - 1$
16: $\quad\quad$ **end while**
17: $\quad\quad \mathbf{M}[\texttt{leftmost}] \leftarrow i + 1$
18: $\quad\quad \mathbf{M}[\texttt{adjacent}] \leftarrow \cup_{j=\mathbf{M}[\texttt{leftmost}]}^{n} \{s_j\} \setminus \{m\}$
19: $\quad\quad \mathbf{M}[\texttt{cost}] \leftarrow \infty$
20: $\quad\quad$ adicione $\mathbf{M}$ em $\mathcal{F}_B$
21: $\quad$ **end if**
22: $\quad$ **return** $\mathcal{F}_B$
23: **end while**

Algorithm 8: Pseudo-código da função `Search-Upper-Root`.

3.2 Melhoramentos na escolha de raiz

Na implementação original do `PFS` foi utilizada a estrutura de *map* para armazenar as raízes das florestas do espaço de busca. Como chave das raízes, utiliza-se a *string* que representa o vetor característico da raiz. A estrutura de *map* em `C++`, geralmente implementada com árvores binárias de busca, mantém os elementos ordenados pelo valor de sua chave, portanto as raízes são armazenadas de maneira ordenada lexicograficamente.

A estratégia adotada na implementação original para a escolha de uma raiz da floresta consiste em escolher de maneira equiprovável o primeiro ou o último elemento do *map* de raízes. Esta estratégia é aleatória, porém viesada, uma vez que escolhe apenas os primeiros ou últimos elementos de uma ordenação. Como esta estratégia de ordenação é arbitrária, não existindo fundamentação teórica que a justifique, acreditamos que novas estratégias de escolhas de raízes podem trazer melhores resultados ao algoritmo `PFS`.

3.2.1 Escolha equiprovável

A primeira estratégia que testamos foi uma escolha também aleatória, mas com uma distribuição de probabilidade de escolhas igual para cada raiz da floresta. Como a estratégia da implementação original de Reis é viesada, julgamos necessário investigar se este viés não leva

o algoritmo a fazer ramificações ou podas ruins, que comprometessem o tempo de execução do algoritmo. Se o tempo de execução com a nova estratégia é menor, então confirmamos esta hipótese.

Para implementar esta nova escolha, utilizamos a mesma estrutura de dados utilizada por Reis com uma pequena modificação no código. Seja $\mathcal{F}$ a floresta de percorrimento escolhida e suponha que existe uma ordenação das raízes dessa floresta, $\langle r_1, r_2, ..., r_{|\mathcal{F}|}\rangle$, então sorteamos um número i entre 1 e $|\mathcal{F}|$ e escolhemos a raiz r_i para o percorrimento.

3.2.2 Escolha de maior árvore

A segunda estratégia que testamos para escolha de raízes é determinística e se baseia em escolher a raiz que tem maior sub-árvore completa. Acreditamos que este critério pode acelerar o algoritmo porque árvores maiores podem fazer podas maiores e, desta maneira, é possível que o algoritmo precise visitar menos nós do reticulado e portanto fazer menos operações, incluindo menos cálculos da função custo.

A sub-árvore completa que tem como raiz o nó $\mathbf{N}$ é composta pelos nós dos conjuntos do intervalo $[\mathbf{N}[\texttt{vertex}], \mathbf{N}[\texttt{vertex}] \cup \mathbf{N}[\texttt{adjacent}]]$ e portanto, o tamanho da árvore é dado por

$$\begin{aligned}|[\mathbf{N}[\texttt{vertex}], \mathbf{N}[\texttt{vertex}] \cup \mathbf{N}[\texttt{adjacent}]]| &= |\{\mathbf{N}[\texttt{vertex}] \cup W : W \in \mathcal{P}(\mathbf{N}[\texttt{adjacent}])\}| \\ &= |\mathcal{P}(\mathbf{N}[\texttt{adjacent}])| \\ &= |\mathcal{P}(\cup_{i=\mathbf{N}[\texttt{leftmost}]}^{n} s_i)| \\ &= 2^{n-\mathbf{N}[\texttt{leftmost}]}.\end{aligned}$$

Portanto, basta escolher a raiz com menor valor de `leftmost`. Agora note que, para as raízes da floresta $\mathcal{F}_A$, o valor de `leftmost` é igual ao maior índice de uma característica presente no conjunto, enquanto na floresta $\mathcal{F}_B$ é igual ao maior índice de uma característica não presente no conjunto.

Note que este número considera que a árvore está completa, o que não é necessariamente verdade para raízes da floresta; um cálculo exato deveria considerar também o conjunto `adjacent` da raiz. Entretanto, utilizamos esse cálculo como uma aproximação do tamanho da floresta por motivos de simplicidade, pois, como explicaremos a seguir, podemos ordenar as raízes de maneira rápida utilizando este cálculo aproximado; fazer esta ordenação com um cálculo que considera as adjacências pode não ser simples.

Se ordenamos as raízes de forma lexicográfica da direita para esquerda (da característica com maior índice para característica com menor índice). Então, a menor raiz é aquela que possui mais zeros da direita para esquerda, ou seja, o menor valor para o maior índice de uma característica presente e consequentemente o menor `leftmost` da floresta $\mathcal{F}_A$. Já a maior raiz é aquela que possui mais uns da direita para esquerda, ou seja, o menor valor para o maior índice de uma característica não presente e consequentemente o maior `leftmost` da floresta $\mathcal{F}_B$.

Portanto, para implementar esta modificação devemos mudar o tipo de ordenação feito pelo *map*, que por padrão é feito lexicograficamente da esquerda para direita (ordem alfabética), e escolher para floresta $\mathcal{F}_A$ o primeiro elemento do *map* e para a floresta $\mathcal{F}_B$ o último elemento.

3.2.3 Experimentos com instâncias artificiais

Testamos ambas abordagens em experimentos computacionais. Analisaremos agora o desempenho dos algoritmos tanto quanto ao tempo médio de execução quanto a quantidade média de nós computados.

Tabela 3.1: Comparação entre os algoritmos `PFS` e `PFS_RAND`. O tempo de execução do segundo é maior do que o primeiro enquanto a quantidade de chamadas da função custo é parecida em ambos.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	PFS	PFS_RAND	PFS	PFS_RAND
10	1024	0.013 ± 0.003	0.014 ± 0.003	590.8 ± 198.5	599.5 ± 177.5
11	2048	0.019 ± 0.004	0.022 ± 0.007	1114.8 ± 331.3	1090.1 ± 350.3
12	4096	0.029 ± 0.008	0.036 ± 0.013	1848.6 ± 600.8	1835.7 ± 683.0
13	8192	0.060 ± 0.018	0.090 ± 0.039	4314.4 ± 1496.4	4201.1 ± 1580.7
14	16384	0.100 ± 0.041	0.191 ± 0.110	7323.4 ± 3318.9	7333.8 ± 3161.0
15	32768	0.180 ± 0.076	0.453 ± 0.311	12958.1 ± 5654.0	12807.5 ± 5753.7
16	65536	0.406 ± 0.185	1.715 ± 1.400	27573.8 ± 12459.5	27036.9 ± 12687.5
17	131072	0.717 ± 0.397	5.416 ± 5.266	48176.2 ± 26938.3	47852.1 ± 26427.6
18	262144	1.325 ± 0.754	15.890 ± 17.726	84417.9 ± 48587.7	84025.0 ± 48882.4
19	524288	2.771 ± 1.603	69.600 ± 82.342	167659.1 ± 99686.7	164612.1 ± 102018.3

Tabela 3.2: Comparação entre os algoritmos `PFS` e `PFS_LEFTMOST`. O tempo de execução e também o número de chamadas da função custo é maior para o `PFS_LEFTMOST`.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	PFS	PFS_LEFTMOST	PFS	PFS_LEFTMOST
10	1024	0.013 ± 0.002	0.023 ± 0.004	606.1 ± 133.5	665.0 ± 165.8
11	2048	0.020 ± 0.004	0.042 ± 0.010	1122.1 ± 351.2	1316.6 ± 382.2
12	4096	0.032 ± 0.008	0.078 ± 0.024	2183.7 ± 733.2	2515.8 ± 871.3
13	8192	0.054 ± 0.017	0.160 ± 0.061	3887.7 ± 1389.9	4716.8 ± 1777.8
14	16384	0.107 ± 0.034	0.345 ± 0.133	7851.2 ± 2793.0	9506.8 ± 3673.9
15	32768	0.196 ± 0.085	0.672 ± 0.274	13780.3 ± 6049.9	17071.6 ± 7005.1
16	65536	0.348 ± 0.189	1.271 ± 0.661	24106.5 ± 13159.9	30055.6 ± 15363.6
17	131072	0.785 ± 0.361	3.137 ± 1.476	52369.0 ± 24751.2	67585.6 ± 30978.4
18	262144	1.445 ± 0.657	6.146 ± 3.032	92095.9 ± 42566.6	120635.7 ± 58039.0
19	524288	3.298 ± 1.883	13.881 ± 7.595	199151.0 ± 112167.8	256078.6 ± 135958.4
20	1048576	6.141 ± 3.728	28.733 ± 18.120	356259.2 ± 218253.1	465255.8 ± 276843.3
21	2097152	10.817 ± 7.714	52.374 ± 38.565	608984.4 ± 432151.7	801620.2 ± 561000.7

A tabela 3.1 mostra os resultados dos experimentos para os algoritmos `PFS` e sua variante que seleciona raízes de maneira aleatória e igualmente distribuída, o que chamamos de `PFS_RAND`. Podemos perceber na tabela que o número de chamadas de função custo tem média e variância parecidas em ambos algoritmos. O tempo médio de execução, por outro lado, é maior para o algoritmo `PFS_RAND`. Portanto, podemos concluir que a esta modificação do `PFS` não trouxe melhorias nos quesitos que estamos avaliando.

A tabela 3.2 mostra os resultados dos experimentos para os algoritmos `PFS` e sua variante que seleciona a raiz com maior potencial de poda, o que chamamos de `PFS_LEFTMOST`. A tabela mostra que o último algoritmo teve desempenho pior em tempo de execução e fez mais chamadas da função de custo, isto é, percorreu mais nós do que o `PFS`. Assim, concluímos que esta modificação não foi vantajosa ao algoritmo.

3.2.4 Comentários sobre experimentos

Os experimentos mostraram que a modificação `PFS_RAND` teve um número parecido de chamadas da função custo. Pelo fato destes números serem parecidos, acreditamos que o percorrimento do reticulado também seja parecido, o que implica que semanticamente esta modificação é equivalente ao algoritmo original. Entretanto, observamos que o `PFS_RAND` teve tempo de execução médio maior. Na busca por entender como o algoritmo pode ter ficado mais caro computacionalmente se sua dinâmica é essencialmente a mesma, investigamos a estrutura de `map` do `C++` e descobrimos que a forma com que implementamos a escolha aleatória de um elemento desta estrutura, avançando um iterador por uma quantidade aleatória de passos, adiciona tempo de complexidade linear (sobre o número de raízes) ao algoritmo, justificando o aumento de tempo de execução.

A outra modificação, `PFS_LEFTMOST`, não tem adições de complexidade computacional na escolha de raízes, visto que ela é feita da mesma forma no algoritmo original, apenas com diferente regra de ordenação. Esta modificação teve um número maior de cálculos da função de custo, o que significa que mais elementos do reticulado foram percorridos (menos foram podados) e também explica o maior tempo de execução. Podemos concluir que esta modificação não é semanticamente equivalente ao `PFS` porque implica em um percorrimento diferente do espaço de busca, evidenciado por mais cálculos da função custo.

3.3 Melhoramentos no controle de raízes

A quantidade de raízes em uma floresta pode ser muito grande no decorrer do algoritmo `PFS`. Um pior caso, exagerado, é considerar que os percorrimentos foram feitos sempre em uma das florestas e que a função de custo é monótona tal que não houveram podas por conta de aumento de custo; então, esta floresta poderá ter até $2^n - n + 1$ raízes. Desta forma, a estrutura de dados utilizada para gerenciar a floresta deve ser eficiente e capaz de armazenar muitas entradas. A estrurura *map* do `C++` é implementada através de uma árvore rubro-negra, uma árvore de busca binária que é balanceada mas não ordenada. Por isso, acreditamos que outras estruturas devem ser consideradas para gerenciar as raízes das florestas do `PFS`. Desta maneira, podemos encontrar novas estruturas mais eficientes para o problema.

Introduzimos então o uso de diagramas de decisão binária ordenados (Ordered Binary Decision Diagram (OBDD)). Esta estrutura representa uma função Booleana $f : \{0,1\}^k \to 0,1$ através de uma árvore. Para se encontrar o valor de um conjunto $X \in \{0,1\}^n$, percorremos um caminho da árvore ramificando de acordo com a presença ou não de uma variável, e o valor deste conjunto deve ser encontrado em uma folha da árvore que é o último vértice do caminho. Em nossa implementação precisamos modificar estes diagramas para que eles permitam armazenar nós ao invés de valores Booleanos. Além disso, a nossa OBDD remove nós redundantes, assim sempre que um vértice tem dois filhos que são folhas vazias, este vértice torna-se uma folha vazia. A figura 3.5 mostra um exemplo de OBDD que representa uma floresta do algoritmo `PFS`.

3.3.1 Escolha de raiz

Ao deixar de utilizar a estrutura de *map* como era feito na implementação original do `PFS`, precisamos definir uma regra para escolha de raízes. Decidimos utilizar a regra de escolha aleatória de raízes, dado que ela se mostrou semanticamente equivalente a implementação original (veja 3.2.3).

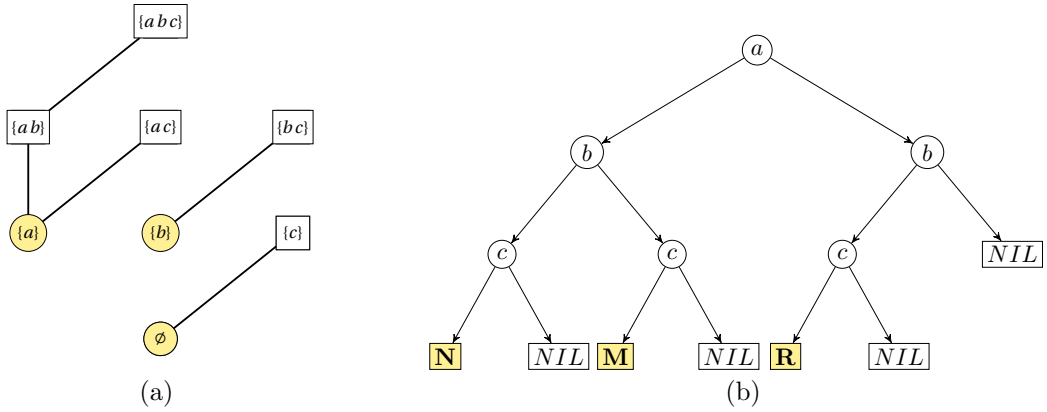


Figura 3.5: Exemplo de OBDD que representa uma floresta do `PFS`*. Esta OBDD contém os nós* **N**, **M** *e* **R**, *que representam respectivamente os subconjuntos* 000, 010 *e* 100. *As folhas* NIL *indicam que os subconjuntos de tal caminho na OBDD não são raízes na floresta, como por exemplo os subconjuntos* $11X$

Para escolher uma raiz em uma OBDD precisamos apenas descer um caminho da raiz até uma folha não vazia (que contenha um nó) guardando os elementos que pertencem ao subconjunto que o caminho representa. A nossa estrutura sempre contrai folhas redundantes para apenas uma, ou seja, não existem vértices na árvore que possuam dois filhos que são folhas vazias, portanto ao descer um caminho pela árvore, é sempre possível escolher a ramificação (ir a direita ou esquerda de um nó) que nos leve a uma folha com um nó (Algoritmo 9).

GET-OBDD-NODE (X)

```
1: v ← root
2: while v não é folha do
3:     if v.element ∈ X then
4:         v ← v.right_child
5:     else
6:         v ← v.left_child
7:     end if
8: end while
9: return v.node
```

Algorithm 9: Pseudo-código de uma função que recebe um conjunto $X \in \mathcal{P}S$ *e devolve o nó que representa este conjunto, se estiver na árvore.*

Desta maneira a escolha da raiz é feita por uma sequência de escolhas de ramificação, para cada vértice da OBDD visitado, que levam da raiz da OBDD até um nó que é folha. Para fazer isto de maneira aleatória e identicamente distribuída entre os nós das folhas, basta armazenar em cada vértice da árvore quantas raízes estão a sua direita e quantos estão a sua esquerda e então dar pesos de escolhas equivalentes a quantidade de raízes para cada lado de ramificação (Algoritmo 10).

3.3.2 Testes com instâncias artificiais

Testamos experimentalmente esta variação do `PFS` que usa OBDDs no controle das florestas. Vamos analisar o desempenho desta variação quanto ao tempo de execução e número de cha-

GET-RANDOM-OBDD-NODE ()

```
1: v ← root
2: while v não é folha do
3:     W ← v.right_child.weight + v.left_child.weight
4:     seja r um número aleatório entre 0 e W
5:     if r ≤ v.right_child.weight then
6:         v ← v.right_child
7:     else
8:         v ← v.right_child
9:     end if
10: end while
11: return v.node
```

Algorithm 10: Pseudo-código de uma função que devolve um nó aleatório da floresta de maneira identicamente provável.

Tabela 3.3: Comparação entre os algoritmos `PFS` e `OPFS`. O número de chamadas médio da função custo é parecido enquanto o tempo de execução é maior para o `OBDD`.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	PFS	OPFS	PFS	OPFS
10	1024	0.013 ± 0.003	0.018 ± 0.003	598.0 ± 192.8	635.5 ± 171.9
11	2048	0.020 ± 0.004	0.029 ± 0.007	1152.1 ± 314.7	1117.9 ± 336.4
12	4096	0.031 ± 0.010	0.049 ± 0.013	2024.1 ± 751.6	2048.2 ± 700.9
13	8192	0.057 ± 0.017	0.097 ± 0.033	3996.3 ± 1431.6	3973.4 ± 1462.6
14	16384	0.094 ± 0.038	0.171 ± 0.063	6634.8 ± 2944.0	6906.5 ± 2786.5
15	32768	0.182 ± 0.079	0.323 ± 0.156	13140.1 ± 6020.6	12711.2 ± 6319.7
16	65536	0.370 ± 0.169	0.660 ± 0.314	25658.2 ± 11606.7	25303.4 ± 12169.5
17	131072	0.819 ± 0.370	1.480 ± 0.665	53344.9 ± 24350.4	53217.2 ± 24154.5
18	262144	1.515 ± 0.905	2.736 ± 1.626	94677.6 ± 54496.3	94079.4 ± 55435.6
19	524288	2.612 ± 1.869	4.818 ± 3.355	156150.5 ± 107369.8	156021.8 ± 107516.8
20	1048576	6.085 ± 3.900	11.550 ± 7.661	344144.1 ± 212627.1	343229.2 ± 212624.4

madas da função de custo.

Como podemos ver na tabela 3.3, o número de chamadas da função de custo do algoritmo modificado é similar ao do `PFS`, o que indica que ambos são semanticamente equivalentes; o que já tinha sido constatado na variante do `PFS` que também escolhia raízes aleatoriamente, mas usando *map* para controle de floresta. Entretanto, a tabela também mostra que o tempo de execução da modificação é maior do que do algoritmo original. Desta maneira, o uso de OBDDs para o controle das florestas não foi vantajoso.

3.4 Paralelização do código

A dinâmica do algoritmo `Poset-Forest-Search` transforma o reticulado Booleano em uma floresta, ou seja, em um conjunto de árvores que são disjuntas. Por isso, o percorrimento destas árvores pode ser feito de maneira paralela sem interferências. Desta maneira, propomos uma paralelização do código de Reis utilizando a biblioteca *OpenMP*, que nos permite paralelizar blocos de instrução com anotações feitas ao compilador.

O primeiro ponto que devemos nos atentar ao paralelizar este código é que as árvores do

espaço de busca são disjuntas apenas se estamos considerando uma floresta. Ambas florestas representam o mesmo espaço de busca, portanto é possível que duas árvores de florestas diferentes se intersectem em alguns nós. Isto significa que fazer o percorrimento em ambas as direções pode causar recálculos, isto é, um nó pode ser visitado desnecessariamente por mais de uma thread por mais de uma vez; além disso, é possível que uma árvore que foi podada em uma direção ainda esteja sendo percorrida em outra direção.

Visando simplificar o trabalho, definimos que a paralelização ocorre em iterações do procedimento principal do `PFS`. Assim, a escolha de uma direção deve ser feita por uma thread principal e, assim cada thread deve escolher uma raiz para ser percorrida em tal direção. Desta maneira cada thread faz a ramificação e atualização da floresta dual e pausam em uma barreira, que sincroniza todas as threads para o finalização da iteração e começo da próxima.

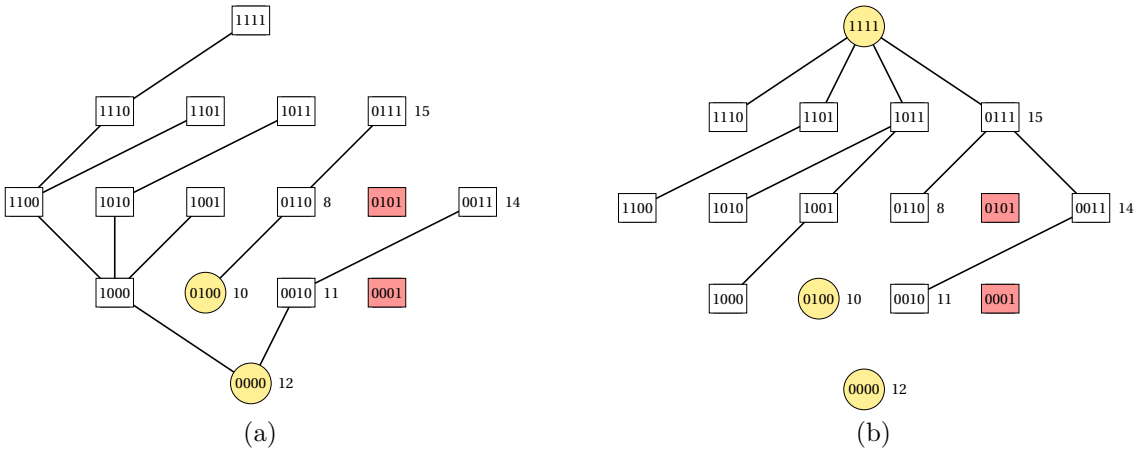


Figura 3.6: Exemplo de espaço de busca do `PFS`; a floresta da figura 3.6(a) é $\mathcal{F}_A$ enquanto a da figura 3.6(b) é $\mathcal{F}_B$. Estas florestas representam o espaço de busca após duas iterações do algoritmo, ambas feitas na direção de baixo para cima, ou seja, ramificando a floresta $\mathcal{F}_A$. Na primeira iteração, seleciona-se a raiz 0000 *para percorrimento e remove-se o nó* 0001*; na floresta dual, a atualização remove o nó* 0001 *e cria a raiz* 0000*. Na segunda iteração, escolhe-se novamente a raiz* 0000 *e a ramificação é feita para* 0100 *(que se torna raiz) e para* 0101*, que é removido do espaço; na floresta dual, remove-se o nó* 0101 *e o nó* 0100 *torna-se raiz.*

Entretanto, apesar desta simplificação tornar a fase de ramificação quase independente entre as linhas de processamento, a fase de atualização de floresta dual pode não ser. Como resultado, condições de corrida não tratadas podem causar inconsistências na atualização da floresta. As figuras 3.6–3.7 mostram um exemplo onde duas linhas de execução podem causar este tipo de inconsistência. A figura 3.6 mostra o espaço de busca do `PFS` no começo de uma iteração, enquanto as figuras 3.7(a)–3.7(b) e 3.7(c)–3.7(d) mostram respectivamente os resultados ao fim de uma iteração para duas diferentes escolhas de raízes. Na figura 3.7(b), que mostra a floresta $\mathcal{F}_B$ quando a raiz escolhida é 0100, o nó 0111 é removido do espaço de busca; na figura 3.7(d), que mostra a floresta $\mathcal{F}_B$ quando a raiz escolhida é 0000, o nó 0111 torna-se raiz. Assuma então que esta iteração do algoritmo é feita com duas linhas de processamento P_1 e P_2, que fazem o percorrimento, respectivamente, a partir de 0000 e 0100. Se a thread P_1 cria o nó 0111 antes de P_2 removê-lo, então não há inconsistências, entretanto se o contrário acontece, criamos uma raiz espúria na floresta $\mathcal{F}_B$.

Para contornar este problema, utilizamos uma outra estrutura de dados do `C++` capaz de armazenar um conjunto. Neste conjunto armazenamos todos os nós que foram removidos do espaço de busca em uma iteração e assim evitamos a criação de raízes espúrias. Note que

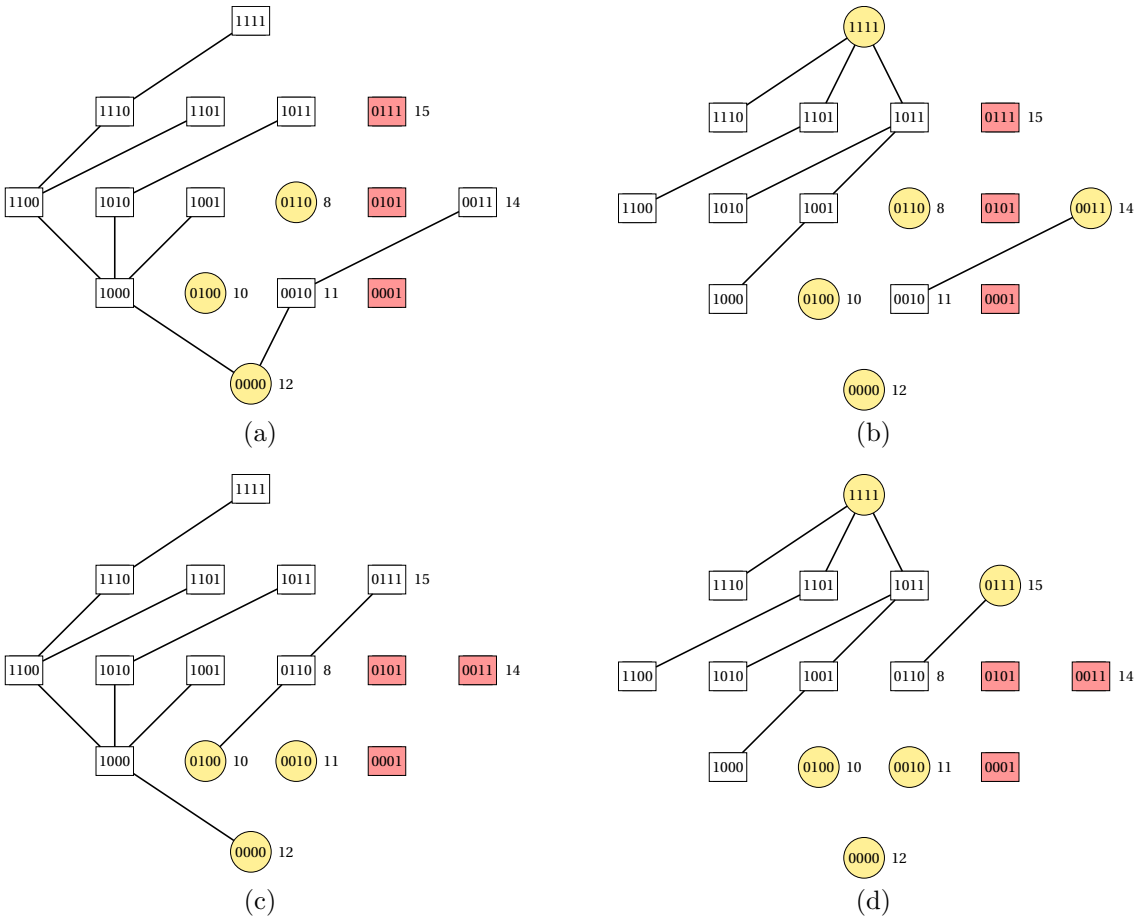


Figura 3.7: Continuação do exemplo da figura 3.6 para duas escolhas diferentes de raízes. As figuras 3.7(a) e 3.7(b) mostram o resultado da continuação da execução do algoritmo após escolha da raiz 0100 *de* $\mathcal{F}_A$ *em um percorrimento de baixo para cima que ramifica até o nó* 0111*, que é podado da floresta* $\mathcal{F}_A$*; na floresta dual, o nó* 0111 *é removido e os nós* 0110 *e* 0011 *tornam-se raízes. As figuras 3.7(c) e 3.7(d) mostram a continuação da execução quando a raiz* 0000 *de* $\mathcal{F}_A$ *é escolhida para percorrimento, que ramifica até o nó* 0011*, que é podado da floresta* $\mathcal{F}_A$*; na floresta dual, o nó* 0011 *é removido e o nó* 0010 *torna-se raiz.*

esse conjunto é esvaziado a cada iteração do `PFS`, portanto não deve ocupar grande espaço de memória; se este conjunto não fosse limpo a cada iteração ele poderia atingir tamanhos exponenciais sobre o número de características do problema, comprometendo a escalabilidade do algoritmo.

Voltando ao exemplo anterior, das figuras 3.6–3.7, com a criação do conjunto de nós deletados, se a linha de processamento P_1 deleta o nó 0111 antes da criação em P_2, então o nó já estará no conjunto de nós deletados quando P_2 tentar adicioná-lo na floresta e assim a criação desta raiz pode ser evitada. Se a ordem é contrária, então o nó é criado e depois deletado, o que pode ser desnecessário, porém consistente.

3.4.1 Testes com instâncias artificiais

Testamos experimentalmente esta versão paralela do `PFS`, que chamamos aqui de `PPFS`. Vamos agora analisar o desempenho deste algoritmo quanto ao tempo de execução e número de chamadas da função de custo.

Tabela 3.4: Comparação entre os algoritmos `PFS` e `PPFS`. O algoritmo `PPFS` apresenta um número similar de média de chamadas da função custo ao `PFS`, mas possui tempo de execução médio maior.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	PFS	PPFS	PFS	PPFS
10	1024	0.011 ± 0.001	0.146 ± 0.027	643.6 ± 133.0	626.5 ± 151.1
11	2048	0.017 ± 0.004	0.227 ± 0.062	1151.0 ± 359.7	1135.6 ± 381.1
12	4096	0.029 ± 0.007	0.385 ± 0.113	2173.1 ± 652.5	2139.2 ± 710.3
13	8192	0.049 ± 0.015	0.640 ± 0.237	3839.8 ± 1376.6	3743.5 ± 1532.9
14	16384	0.104 ± 0.037	1.337 ± 0.513	8175.9 ± 3037.4	8026.4 ± 3303.7
15	32768	0.163 ± 0.078	2.010 ± 1.096	12459.6 ± 6164.5	12062.2 ± 6897.2
16	65536	0.360 ± 0.163	4.483 ± 2.030	27027.3 ± 12397.0	26835.7 ± 12446.9
17	131072	0.664 ± 0.362	8.072 ± 4.370	48001.9 ± 26149.2	48093.4 ± 26233.2
18	262144	1.250 ± 0.690	14.341 ± 8.062	85880.9 ± 47950.8	86050.8 ± 49186.8
19	524288	2.936 ± 1.629	34.639 ± 19.074	198503.0 ± 108116.1	197832.5 ± 110659.6
20	1048576	5.024 ± 3.097	61.038 ± 38.250	321495.8 ± 198004.3	318507.0 ± 199354.3

Podemos observar na tabela 3.4 que os números de chamadas da função custo dos algoritmos `PFS` e `PPFS` são parecidos. Isto quer dizer que a nossa paralelização não trouxe grandes modificações semânticas ao algoritmo. Entretanto, o tempo médio de execução teve diferenças na versão paralela, que gastou mais tempo do que o algoritmo original em todas as instâncias. A tabela mostra que o `PPFS` foi pelo menos 10 vezes mais lento que a implementação serial para as instâncias vistas.

3.4.2 Comentários sobre experimentos

Acreditamos que os resultados pobres do algoritmo `PPFS` se deram por conta da complexidade da atualização da floresta dual. Apesar das ramificações serem feitas em paralelo com pouco ou nenhum entrelace de linhas de processamento, esta rotina consome tempo de execução pequeno comparado com a rotina de atualização de florestas. Com a paralelização, esta atualização tornou necessária uma nova estrutura de dados ao algoritmo para tratar condições de corrida que podiam adicionar inconsistências as florestas. Além disso, a atualização depende de adições, remoções e verificações nas florestas que não podem ser feitas em paralelo, criando muitas seções críticas no código, o que deve ter sido a principal causa da piora de tempo, pois estes blocos de código implicam em uma troca constante de contextos pelos processadores, que não podem processar blocos críticos ao mesmo tempo.

3.5 O algoritmo UBB-PFS

Tendo em vista que o principal problema da paralelização do `PFS` foi o fato de que há muito entrelace nas linhas de execução, propomos um novo algoritmo baseado no `UBB` e `PFS` que fosse também paralelo, mas com pouca ou nenhuma interferência entre as linhas de execução, chamamos este algoritmo de `UBB-PFS`.

3.5.1 Descrição

Este algoritmo tem duas etapas principais. A primeira etapa é idêntica ao `UBB`, e faz o percorrimento de uma árvore do espaço de busca como em uma busca em profundidade, podando os nós em que o custo cresce. Depois de um certo número de iterações, a primeira etapa é

finalizada e então se dá início a segunda etapa, que deve resolver cada sub-árvore do espaço de busca corrente utilizando o algoritmo `PFS`. A solução das sub-árvores deve ocorrer de maneira independente, possibilitando a paralelização mais fácil do código.

O pseudo-código do algoritmo 11 mostra como é feita a primeira etapa do algoritmo e também serve de base para a explicação da segunda etapa `UBB-PFS` e da corretude do algoritmo, pois possui os seguintes invariantes no final da linha 16:

a) Para todo nó do espaço de busca que não é raiz, existem três possibilidades: ou ele foi visitado e está em $\mathcal{M}$, ou podado, ou e está contido em uma sub-árvore tal que a raiz está na pilha $\mathcal{S}$. Todo nó que é raiz está em $\mathcal{M}$;

b) Todo nó na pilha $\mathcal{S}$ é raiz de uma sub-árvore completa.

UBB-PFS (S, c)
1: $\mathcal{M} \leftarrow \emptyset$
2: inicialize a pilha $\mathcal{S}$ vazia
3: empilhe $(\emptyset, S, c(\emptyset))$ em $\mathcal{S}$
4: **while** $\mathcal{S}$ não vazia **do**
5: desempilhe $(Y, X, cost_Y)$ de $\mathcal{S}$
6: $X_{Y_i} \leftarrow \emptyset$
7: **while** $X \neq \emptyset$ **do**
8: remova s_i de X
9: $Y_i \leftarrow Y \cup \{s_i\}$
10: $cost_{Y_i} \leftarrow c(Y_i)$
11: **if** $cost_{Y_i} \leq cost_Y$ **then**
12: empilhe $(Y_i, X_{Y_i}, cost_{Y_i})$ em $\mathcal{S}$
13: **end if**
14: $X_{Y_i} \leftarrow s_i$
15: **end while**
16: $\mathcal{M} \leftarrow \mathcal{M} \cup \{Y\}$
17: **if** FinishStepOne() **then**
18: $\mathcal{N} \leftarrow$ SolveTrees$(\mathcal{S}, S, c)$
19: esvazie $\mathcal{S}$
20: **end if**
21: **end while**
22: $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{N}$
23: **return** $\{M \in \mathcal{M} : c(M)$é mínimo$\}$

Algorithm 11: Pseudo-código do algoritmo `UBB-PFS`. A primeira etapa consiste no percorrimento feito nas linhas 4 - 21, de maneira idêntica ao `UBB`.

O invariante a) nos garante que o mínimo global já está em $\mathcal{M}$ ou é minimo dentro de uma das árvores que tem raiz em $\mathcal{S}$, portanto, quando estamos na linha do invariante, resolver o problema original é igual a achar o mínimo de cada uma das árvores e do conjunto $\mathcal{M}$. O invariante b) garante que toda árvore com raiz Y contém exatamente os subconjuntos do intervalo $[Y, X \cup Y]$.

Para resolver cada sub-árvore com raiz em $\mathcal{S}$, criamos o seguinte problema U-curve auxiliar. Seja $\langle S, c\rangle$ uma instância do problema U-curve e seja Y uma raiz de uma sub-árvore do problema e X um conjunto de características tal que os nós desta sub-árvore seja exatamente o intervalo

$[Y, X \cup Y]$. Considerando que este intervalo é igual ao conjunto $\{Y \cup W : W \in \mathcal{P}(X)\}$, então, encontrar o mínimo dentro deste conjunto é exatamente resolver a instância do problema U-curve $\langle S_X, c_Y \rangle$ tal que:

$$S_X = X$$
$$c_Y(X') = c(X' \cup Y), \;\; X' \in \mathcal{P}(S_X).$$

Cada uma dessas instâncias é resolvida utilizando a função `SolveTrees` (Algoritmo 12).

SolveTrees($\mathcal{S}, S, c$)
1: $\mathcal{N} \leftarrow \emptyset$
2: **while** $\mathcal{S}$ não vazia **do**
3: desempilhe $(Y, X, cost_Y)$
4: $S_X \leftarrow X$
5: seja $c_Y : X' \mapsto c(X' \cup Y)$
6: $N \leftarrow$ Poset-Forest-Search(S_X, c_Y)
7: $\mathcal{N} \leftarrow \mathcal{N} \cup N$
8: **end while**
9: **return** $\mathcal{N}$

Algorithm 12: Pseudo-código da função `SolveTrees`.

A função FinishStepOne determina qual é a iteração em que o algoritmo deve terminar a primeira etapa. Esta função Booleana controla, de certa forma, a maneira com que o trabalho é dividido entre etapa 1 e 2 do código. Se ela nunca for verdadeira, então o `UBB-PFS` deve se comportar exatamente como o `UBB`. Além disso, a segunda etapa deve ser rodada em linhas de processamento paralelos, portanto esta função também influencia a distribuição de trabalho e nível de paralelização.

3.5.2 Paralelização

A paralelização deste algoritmo também foi feita com a biblioteca *OpenMP*. Conseguimos paralelizar este código com apenas algumas anotações para o compilador, que indicam que a função SolveTrees deve criar tarefas (*tasks*) para cada chamada do algoritmo Poset-Forest-Search, e também que deve haver uma barreira que só permite o retorno da função SolveTrees quando todas as tarefas criadas foram resolvidas.

Para que a paralelização traga melhoras significativas é esperado que haja um número razoável de árvores na pilha $\mathcal{S}$. Um exemplo de número de raízes que é razoável para paralelização é a quantidade de processadores da máquina. A função FinishStepOne deve garantir que terminamos a primeira etapa do `UBB-PFS` com um número razoável de raízes na pilha. Note que o número de raízes é limitado pela própria topologia do problema; se a função é, por exemplo, monótona crescente, então a única raiz empilhada é a do conjunto vazio.

Em nossa implementação, definimos que a função FinishStepOne é verdadeira quando o número de raízes é maior que o número de cores da máquina ou quando o número de iterações é maior do que duas vezes a quantidade de características.

3.5.3 Experimentos com instâncias artificiais

Testamos o algoritmo `UBB-PFS` em experimentos computacionais. Analisaremos agora o desempenho dos algoritmos tanto quanto ao tempo médio de execução quanto a quantidade de nós computados.

Tabela 3.5: Comparação de tempo médio de execução entre os algoritmos `UBB`, `PFS` e `UBB-PFS`. Podemos observar que o `PFS` foi o mais lento enquanto o `UBB` foi o mais rápido e o `UBB-PFS` teve desempenho itermediário para estas instâncias. Este teste continua na tabela 3.6, que mostra apenas o `UBB` e o `UBB-PFS` já que o `PFS` mostrou ter desempenho pior.

Instância		Tempo de execução médio (s)		
$\|S\|$	$2^{\|S\|}$	UBB	PFS	UBB-PFS
10	1024	0.006 ± 0.001	0.011 ± 0.002	0.023 ± 0.004
11	2048	0.007 ± 0.001	0.017 ± 0.004	0.026 ± 0.004
12	4096	0.010 ± 0.003	0.029 ± 0.009	0.034 ± 0.006
13	8192	0.013 ± 0.006	0.047 ± 0.016	0.044 ± 0.011
14	16384	0.024 ± 0.013	0.094 ± 0.034	0.068 ± 0.023
15	32768	0.043 ± 0.026	0.186 ± 0.074	0.113 ± 0.042
16	65536	0.083 ± 0.060	0.339 ± 0.168	0.187 ± 0.082
17	131072	0.161 ± 0.122	0.650 ± 0.347	0.326 ± 0.175
18	262144	0.321 ± 0.233	1.482 ± 0.768	0.703 ± 0.380
19	524288	0.620 ± 0.447	2.711 ± 1.562	1.309 ± 0.729
20	1048576	1.312 ± 0.970	5.007 ± 3.302	2.478 ± 1.547
21	2097152	2.494 ± 1.893	11.125 ± 6.749	5.458 ± 3.294
22	4194304	4.589 ± 4.122	19.085 ± 15.147	8.832 ± 6.846
23	8388608	12.228 ± 7.922	40.323 ± 29.649	18.891 ± 12.786
24	16777216	24.273 ± 16.277	113.332 ± 76.688	67.178 ± 46.516

A tabela 3.5 mostra o tempo de execução dos algoritmos `PFS`, `UBB-PFS` e `UBB`. O `UBB-PFS` teve tempos de execuções médios menores do que o `PFS`, portanto precisamos investigar se ele é melhor também do que um outro algoritmo mais rápido que o `PFS`, o `UBB`. Este último mostrou ter melhor desempenho nesta tabela, entretanto, por conta da paralelização, é possível que o `UBB-PFS` se torne mais rápido para instâncias maiores. Investigamos isto com instâncias maiores na tabela 3.6 e constatamos que para estas instâncias o `UBB` ainda é mais rápido.

A tabela 3.7 mostra o número médio de nós computados pelos mesmos algoritmos. Podemos observar nesta tabela que o `UBB`, apesar de ser mais rápido, precisa computar mais nós que os outros algoritmos. O `PFS` é o algoritmo que precisa calcular menos nós, enquanto o `UBB-PFS` é intermediário, mas com desempenho mais próximo do `PFS`.

3.5.4 Comentários sobre os experimentos

Dado que o tempo de execução do `UBB-PFS` foi menor do que o tempo do `PFS` e que o número de nós computados foi menor do que o necessário pelo `UBB`, que é o mais rápido dos três, concluímos que existem instâncias em que o `UBB-PFS` é o algoritmo mais apropriado entre os três. Este tipo de instância é tal que o tempo de percorrimento é tão relevante quanto o tempo de computação da função custo, desta maneira o número maior de cálculos do `UBB` e o tempo maior de percorrimento do `PFS` tornariam o `UBB-PFS` a melhor opção, porque é intermediário no consumo de tempo e próximo do melhor dos três quanto ao número de cálculos da função custo.

Tabela 3.6: Comparação de tempo médio de execução entre os algoritmos `UBB` e `UBB-PFS`. O algoritmo `UBB-PFS` mostrou maior tempo médio de execução.

Instância		Tempo de execução médio (s)	
$\|S\|$	$2^{\|S\|}$	UBB	UBB-PFS
20	1048576	1.245 ± 0.861	5.106 ± 3.017
21	2097152	2.332 ± 1.900	9.178 ± 6.330
22	4194304	5.653 ± 3.904	17.405 ± 11.754
23	8388608	9.432 ± 8.938	28.746 ± 26.387
24	16777216	20.293 ± 16.147	77.392 ± 59.976
25	33554432	46.215 ± 35.837	144.196 ± 110.583

Tabela 3.7: Comparação sobre o número de chamadas de função custo entre os algoritmos `UBB`, `PFS` e `UBB-PFS`. Vemos nesta tabela que o número de nós computados pelo `UBB` é o maior enquanto o do `PFS` é o menor; o `UBB-PFS` tem desempenho intermediário, porém próximo ao do `PFS`.

Instância		Número médio de cálculos de custo		
$\|S\|$	$2^{\|S\|}$	UBB	PFS	UBB-PFS
10	1024	699.4 ± 361.3	611.3 ± 178.8	634.7 ± 209.8
11	2048	1217.2 ± 747.1	1145.4 ± 358.9	1178.8 ± 484.1
12	4096	2898.0 ± 1380.4	2103.1 ± 793.6	2363.0 ± 760.4
13	8192	4422.6 ± 3293.8	3650.9 ± 1371.9	3934.4 ± 1487.7
14	16384	10089.4 ± 6452.1	7536.3 ± 2926.1	8012.2 ± 3387.4
15	32768	19097.5 ± 12793.8	14546.5 ± 6081.0	15299.2 ± 6598.4
16	65536	37663.1 ± 28321.2	25744.0 ± 12795.4	27028.4 ± 13031.9
17	131072	73373.3 ± 55994.3	46808.9 ± 24533.5	49348.6 ± 24556.7
18	262144	150035.2 ± 108299.3	103166.6 ± 52464.7	105306.4 ± 53472.0
19	524288	292561.2 ± 210771.2	183125.7 ± 104965.4	189545.7 ± 102145.9
20	1048576	617049.5 ± 450468.2	323097.4 ± 213634.3	340694.2 ± 202389.6
21	2097152	1172641.6 ± 879148.5	691991.3 ± 413262.9	704790.2 ± 407143.8
22	4194304	2099973.2 ± 1863285.8	1133395.1 ± 874492.0	1156564.2 ± 862152.0
23	8388608	5435778.8 ± 3468245.3	2276694.5 ± 1621342.2	2345648.2 ± 1558258.5
24	16777216	10146842.9 ± 6673018.3	5527504.2 ± 3413432.3	5609052.7 ± 3337059.1

Capítulo 4

O algoritmo `Parallel-U-Curve-Search` (`PUCS`)

O algoritmo `Parallel U-Curve Search (PUCS)` foi desenvolvido para resolver o problema U-Curve particionando o espaço de busca em partes que podem ser resolvidas independentemente e de forma paralela. Além disso, a dinâmica desse algoritmo depende de parâmetros que determinam o tempo de execução e qualidade da solução obtida, permitindo ao usuário adequar o algoritmo aos recursos computacionais disponíveis.

4.1 Princípios

Seja S o conjunto de características do problema em questão. O primeiro passo do particionamento é escolher arbitrariamente S' um subconjunto de S; de maneira complementar, definimos $\overline{S'} = S \setminus S'$. Agora, sejam $X, Y \in \mathcal{P}(S)$ e $\sim$ a relação:

$$X \sim Y \iff (X \cap S') = (Y \cap S')$$

Esta relação é de equivalência, pois nela valem:

- reflexividade

$$X \sim X, \text{ pois } (X \cap S') = (X \cap S')$$

- simetria

$$\begin{aligned} X \sim Y &\iff \\ (X \cap S') = (Y \cap S') &\iff \\ (Y \cap S') = (X \cap S') &\iff \\ Y \sim X & \end{aligned}$$

- transitividade,

$$\begin{aligned} X \sim Y, Y \sim Z &\Rightarrow \\ (X \cap S') = (Y \cap S') = (Z \cap S') &\Rightarrow \\ (X \cap S') = (Z \cap S') &\Rightarrow \\ X \sim Z & \end{aligned}$$

Portanto, o conjunto das classes de equivalência definidas por $\sim$ é uma partição do espaço de busca original. Tome como exemplo o conjunto $S = \{a, b, c\}$; se $S' = a$, então existem duas classes de equivalência no particionamento do espaço de busca que definimos, formados pelos conjuntos $\{\emptyset, b, c, bc\}$ e $\{a, ab, ac, abc\}$.

Pela definição da relação $\sim$ temos que a presença de cada característica de S' em uma dada parte do reticulado não muda, isto é, ou ela está presente em todos subconjuntos da parte ou não está presente em nenhum, portanto, dizemos que estas variáveis são **fixas**. De modo análogo, as variáveis de $\overline{S'}$ são **livres**. Tanto variáveis fixas quanto livres podem definir reticulados Booleanos junto a relação de ordem parcial $\subseteq$.

O conjunto $\mathcal{P}(S')$ induz um reticulado Booleano em que cada elemento representa uma classe de equivalência do espaço de soluções do problema original, chamamos este de **reticulado externo**. Para cada classe de equivalência (nó do reticulado externo), o conjunto $\mathcal{P}(\overline{S'})$ induz um outro reticulado Booleano (**reticulado interno**) em que cada elemento representa um subconjunto de problema original. Seja $A \in \mathcal{P}(S')$ um elemento do reticulado externo, então cada $B \in \mathcal{P}(\overline{S'})$ do reticulado interno em A representa o conjunto $X = B \cup A$ do espaço de busca do problema original. A figura 4.1 apresenta um exemplo de particionamento feito pelo `PUCS` em um reticulado Booleano com cinco características.

Os reticulados internos e externo elucidam a estrutura recursiva do problema de seleção de características e sugerem que podemos construir uma solução ao problema original a partir de soluções de outros problemas, sobre os reticulados externo e internos, abordagem conhecida em computação como divisão e conquista. Seja $\langle S, c\rangle$ uma instância do problema de seleção de características, S' o conjunto de variáveis fixas, $\overline{S'}$ o conjunto de variáveis livres, e $A \in \mathcal{P}(S')$ um subconjunto que é nó do reticulado externo, então podemos definir um outro problema de seleção de características $\langle \overline{S'}, c_A\rangle$ em que

$$c_A(X) = c(X \cup A).$$

Resolver a instância $\langle \overline{S'}, c_A\rangle$ é essencialmente achar o mínimo do problema inicial restrito a classe de equivalência de A, dizemos também que estamos resolvendo a parte A. Se soubermos em qual classe o mínimo global reside, podemos resolver apenas tal parte e garantir que a solução encontrada é a solução do problema original.

4.2 Dinâmica

Com as estruturas de reticulado interno e externo, o `PUCS` resolve uma instância do problema U-Curve em duas etapas. Na primeira, o algoritmo percorre o reticulado externo, fazendo podas sempre que possível, e armazena cada parte que é candidata a conter o mínimo global do problema. Na segunda etapa, para cada parte candidata, resolve-se o problema U-Curve auxiliar que é equivalente ao problema original, mas restrito a parte de interesse; em seguida, escolhe-se como resposta o conjunto custo mínimo entre as soluções dos problemas parciais.

4.2.1 Condições de poda

As podas eliminam do reticulado externo intervalos da forma $[X, \mathcal{P}(S')]$ ou $[\emptyset, X]$ e são realizadas sempre que a hipótese de curva em U implica que todas as partes contidas nestes intervalos não contém o mínimo global. Para entender o critério de poda, vamos definir que a **ponta superior** de um reticulado Booleano $\mathcal{P}(A)$ é o próprio conjunto A e a **ponta inferior** deste reticulado é o conjunto vazio. Note que no reticulado interno de uma parte P a ponta inferior

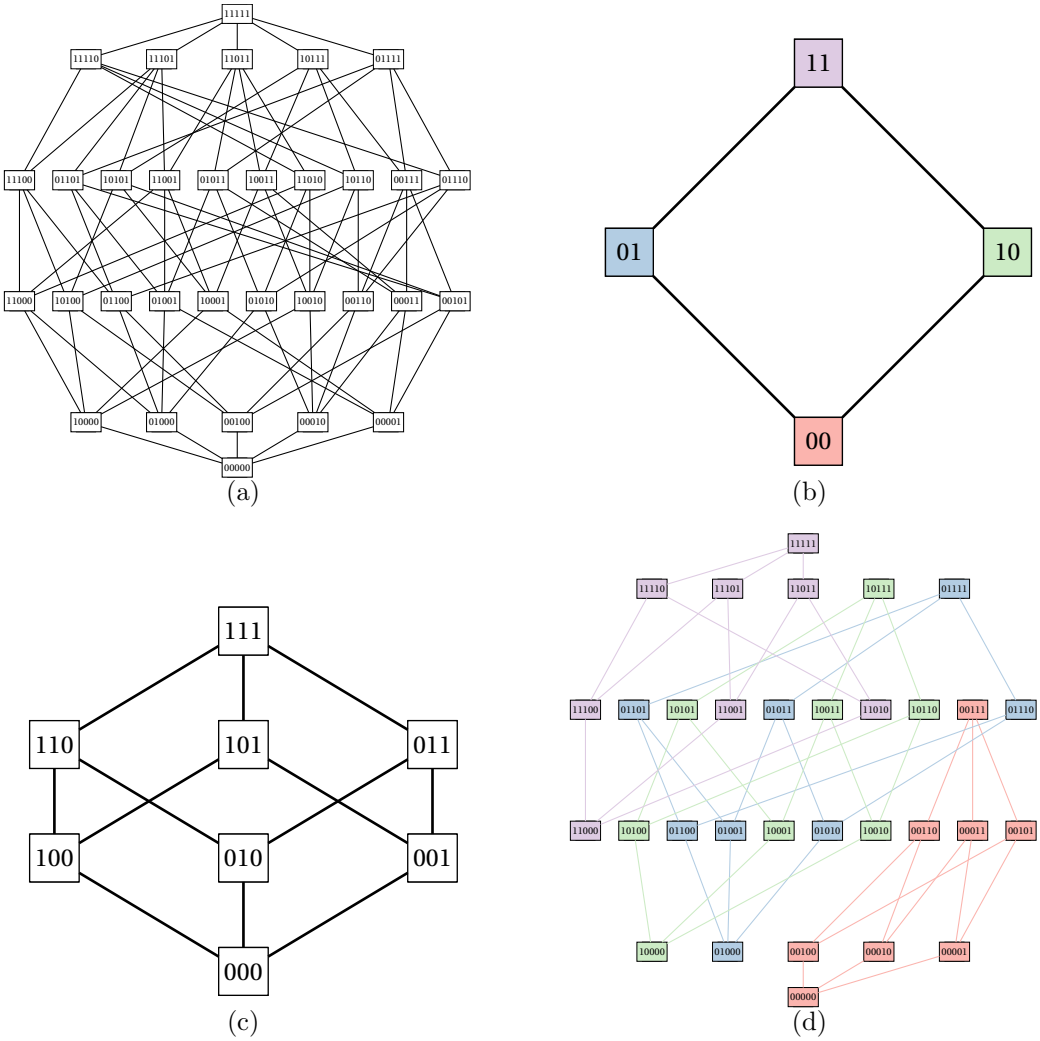


Figura 4.1: Exemplo de particionamento feito pelo algoritmo **PUCS** *em uma instância com cinco características; o reticulado Booleano desta instância é representado na figura 4.1(a). Neste particionamento, as duas primeiras variáveis formam o conjunto de variáveis fixadas, definindo o reticulado externo (figura 4.1(b)) enquanto as outras três definem os reticulados internos, que são cópias do reticulado da figura 4.1(c). A figura 4.1(d) mostra o reticulado Booleano original, sem as arestas que ligam duas partes diferentes, e a cor de cada nó representa a qual parte tal nó pertence, de acordo com as cores do reticulado externo em 4.1(b) Note que, de fato, cada parte forma um reticulado pequeno de mesmo tamanho e com mesma estrutura que o reticulado da figura 4.1(c)*

representa o próprio conjunto de características P, enquanto que a ponta superior representa o conjunto de características $P \cup \overline{S'}$.

Teorema 4.2.1 (Critério de poda para o reticulado externo do algoritmo PUCS)**.** *Sejam S um conjunto de características e S' um conjunto de variáveis fixas no particionamento definido pelo algoritmo* **PUCS***. Dados $P, Q \in \mathcal{P}(S')$ dois elementos do reticulado externo com $Q \subseteq P$; se a ponta inferior do reticulado interno de P tem custo maior do que a ponta inferior do reticulado interno de Q, então todas as partes do intervalo $[P, \mathcal{P}(S')]$ tem apenas conjuntos de características com custo maior do que o custo da ponta inferior de Q.*

Demonstração. Se o custo da ponta inferior do reticulado interno de P é maior do que a de Q, então:

$$c_Q(\emptyset) < c_P(\emptyset)$$
$$c(\emptyset \cup Q) < c(\emptyset \cup P)$$
$$c(Q) < c(P)$$

Como $Q \subseteq P$, temos que existe uma cadeia que passa pelas pontas inferiores de Q e P. Além disso, para qualquer conjunto de características $X \in \mathcal{P}(S)$, com $P \subseteq X$, a hipótese de curva em U garante que:

$$c(P) \leq max\{c(Q), c(X)\}$$

e como $c(P) > c(Q)$, temos que $c(X) \geq c(P)$, isto é, qualquer elemento do reticulado Booleano original que cobre a ponta inferior de P tem custo estritamente maior do que o custo da ponta inferior de Q. Note que para qualquer parte R do intervalo $[P, \mathcal{P}(S')]$, vale que $P \subseteq R$, e como a ponta inferior de P não contém nenhum elemento de $\overline{S'}$, então qualquer conjunto de características da parte R cobre a ponta inferior de P e portanto tem custo estritamente maior do que o custo da ponta inferior da parte Q. □

Teorema 4.2.2 (Critério dual de poda para o reticulado externo do algoritmo `PUCS`)**.** *Sejam S um conjunto de características e S' um conjunto de variáveis fixas no particionamento definido pelo algoritmo* `PUCS`*. Dados $P, Q \in \mathcal{P}(S')$ dois elementos do reticulado externo com $Q \subseteq P$; se a ponta superior do reticulado interno de P tem custo menor do que a ponta superior do reticulado interno de Q, então todas as partes do intervalo $[\emptyset, Q]$ tem apenas conjuntos de características com custo maior do que o custo da ponta superior de P.*

Demonstração. Se o custo da ponta superior do reticulado interno de Q é maior do que a de P, então:

$$c_P(\overline{S'}) < c_Q(\overline{S'})$$
$$c(\overline{S'} \cup P) < c(\overline{S'} \cup Q)$$

Como $Q \subseteq P$, temos que existe uma cadeia que passa pelas pontas superiores de Q e P. Além disso, para qualquer conjunto de características $X \in \mathcal{P}(S)$, com $\{Q \cup \overline{S'}\} \supseteq X$, a hipótese de curva em U garante que:

$$c(Q \cup \overline{S'}) \leq max\{c(P \cup \overline{S'}), c(X)\}$$

e como $c(Q \cup \overline{S'}) > c(P \cup \overline{S'})$, temos que $c(X) \geq c(Q \cup \overline{S'})$, isto é, qualquer elemento do reticulado Booleano original que é coberto pela ponta superior de Q tem custo estritamente maior do que o custo da ponta superior de P. Note que para qualquer parte R do intervalo $[\emptyset, Q]$, vale que $R \subseteq Q$, e como a ponta superior de Q contém todos os elementos de $\overline{S'}$, então qualquer conjunto de características da parte R é coberto pela ponta superior de Q e portanto tem custo estritamente maior do que o custo da ponta superior da parte P. □

4.2.2 Passeio aleatório no reticulado externo

O passeio do `PUCS` se inicia escolhendo arbitrariamente um nó inicial que pertence ao espaço de busca, então a cada passo escolhe-se aleatoriamente um vizinho do nó corrente, que também deve pertencer ao espaço de busca, e verificam-se as condições de poda. Caso elas sejam verdadeiras, o procedimento de poda elimina parte do reticulado; se o vizinho escolhido foi removido nesta etapa, então escolhe-se outro vizinho. O vizinho escolhido torna-se então o nó corrente e o procedimento é repetido até que não seja possível escolher um vizinho; quando isto ocorre e o espaço ainda não foi esgotado, escolhe-se novamente um início de passeio arbitrariamente. Todo nó visitado é automaticamente removido do espaço de busca, e os passeios aleatórios são repetidos até que o espaço de busca tenha sido esgotado, ou seja, todo nó foi ou visitado ou removido em alguma poda.

Durante a realização dos passeios aleatórios, precisamos armazenar quais são as partes candidatas a conterem o mínimo global. As podas deste algoritmo removem do espaço de busca apenas partes que obrigatoriamente não contém o mínimo global, portanto qualquer outra parte é candidata a conter tal elemento. Desta forma, como toda parte é visitada ou podada, temos que o conjunto de nós visitados e não podados é exatamente o conjunto de candidatos a conterem o subconjunto de custo ótimo.

As figuras 4.2 - 4.4 mostram a dinâmica do `PUCS` ao resolver uma instância do problema U-Curve.

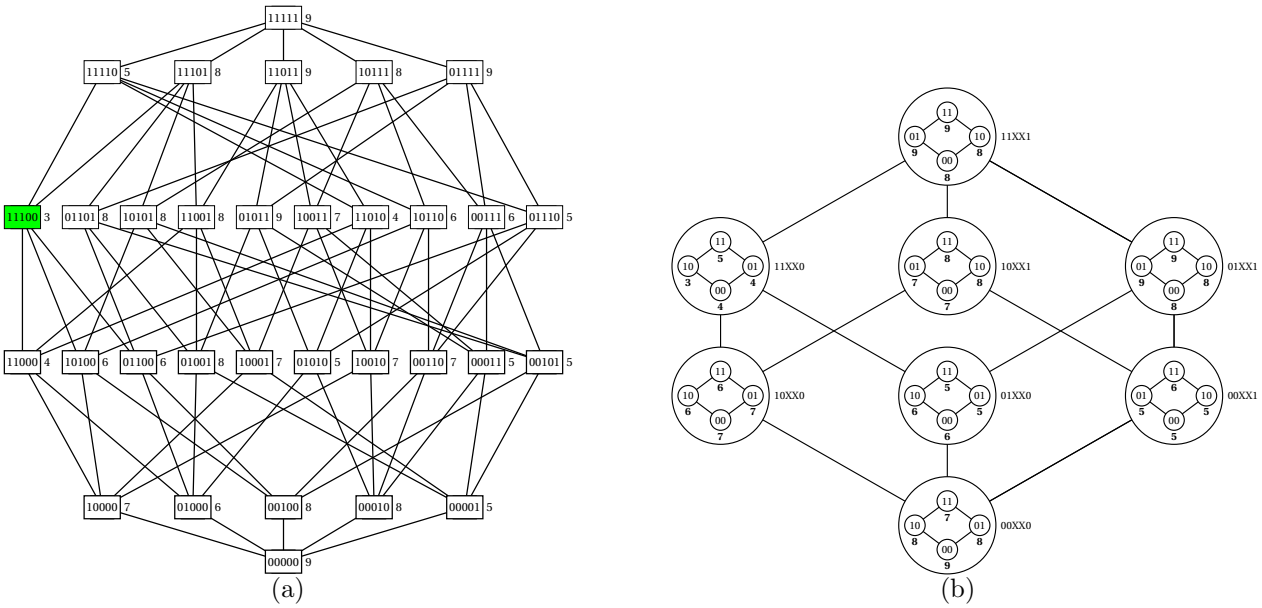


Figura 4.2: Uma instância do problema U-Curve e o seu particionamento quando o conjunto de variáveis fixas S' é composto pela primeira, segunda e última variável. No reticulado externo, denotamos por X don't cares, que são as variáveis livres do particionamento. O subconjunto colorido em verde é o elemento de custo mínimo desta instância.

- Figura 4.2(a): uma instância do problema U-Curve com cinco características e com a função de custo anotada ao lado dos nós do reticulado.
- Figura 4.2(b): o particionamento do espaço de busca quando são fixadas a primeira, segunda e última característica.

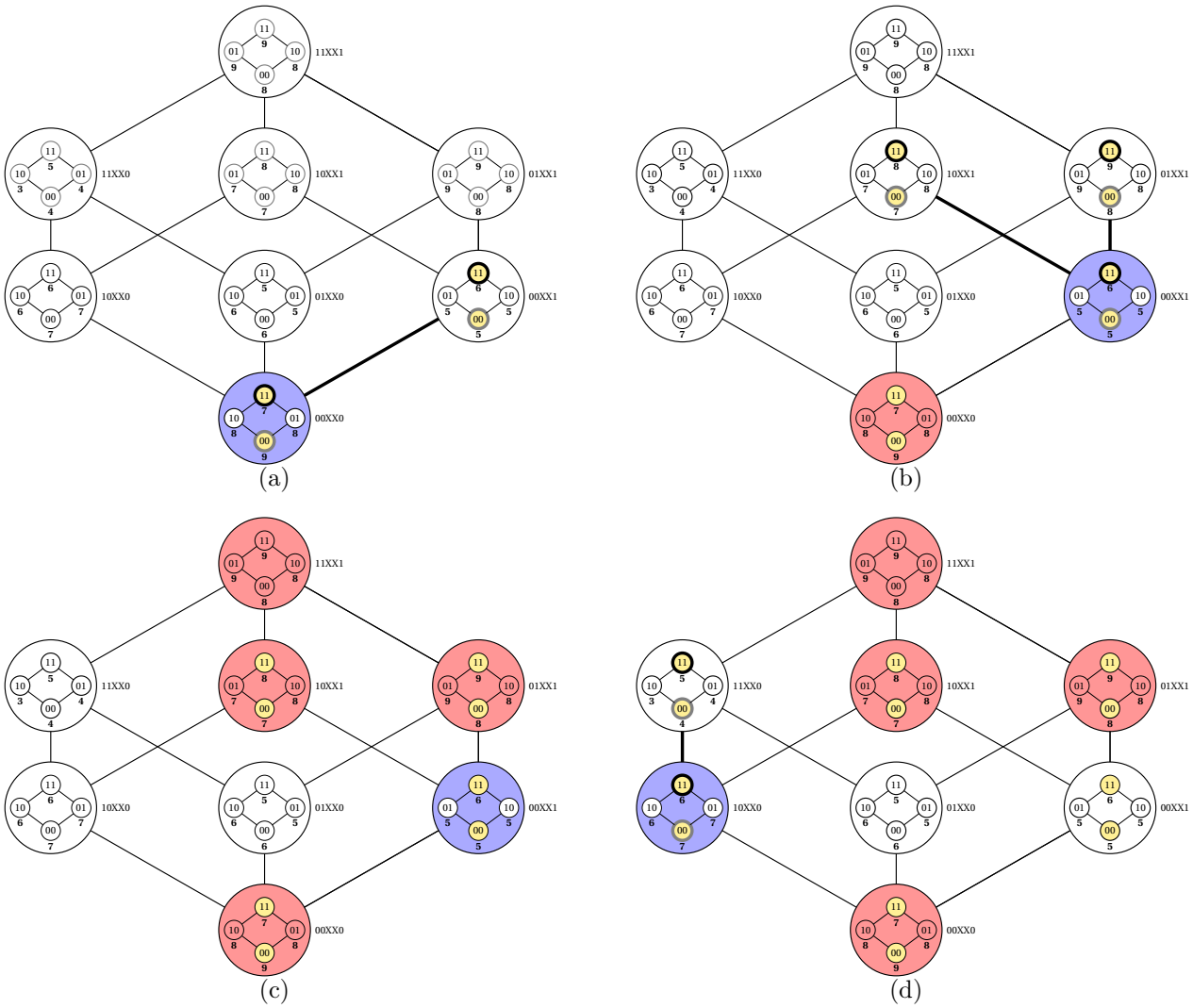


Figura 4.3: Dinâmica do algoritmo `PUCS` ao resolver a instância apresentada na figura 4.2

- Figura 4.3(a) a parte 00XX0 é escolhida arbitrariamente para ser o início do passeio aleatório. O vizinho 00XX1 é escolhido aleatoriamente como candidato a ser o próximo nó do passeio. Como o custo da ponta superior de 00XX1 (6) é menor do que o custo da ponta superior de 00XX0 (7), o intervalo de partes $[000, 000]$ é removido do espaço de busca.

- Figura 4.3(b) a pontas inferiores das partes 10XX1 (7) e 01XX1 (8) têm custo maior do que a ponta inferior de 00XX1 (5), portanto os intervalos de partes $[101, 111]$ e $[011, 111]$ são removidos do espaço de busca.

- Figura 4.3(c) todos os vizinhos de 00XX1 foram podados, portanto esta parte torna-se candidata a conter o mínimo, e iniciamos um novo passeio.

- Figura 4.3(d) a parte 10XX0 é escolhida arbitrariamente como início de passeio. O custo da ponta superior de 10XX0 (6) é maior do que o custo da ponta superior de 11XX0 (5), portanto o intervalo de partes $[000, 100]$ é removido do espaço de busca.

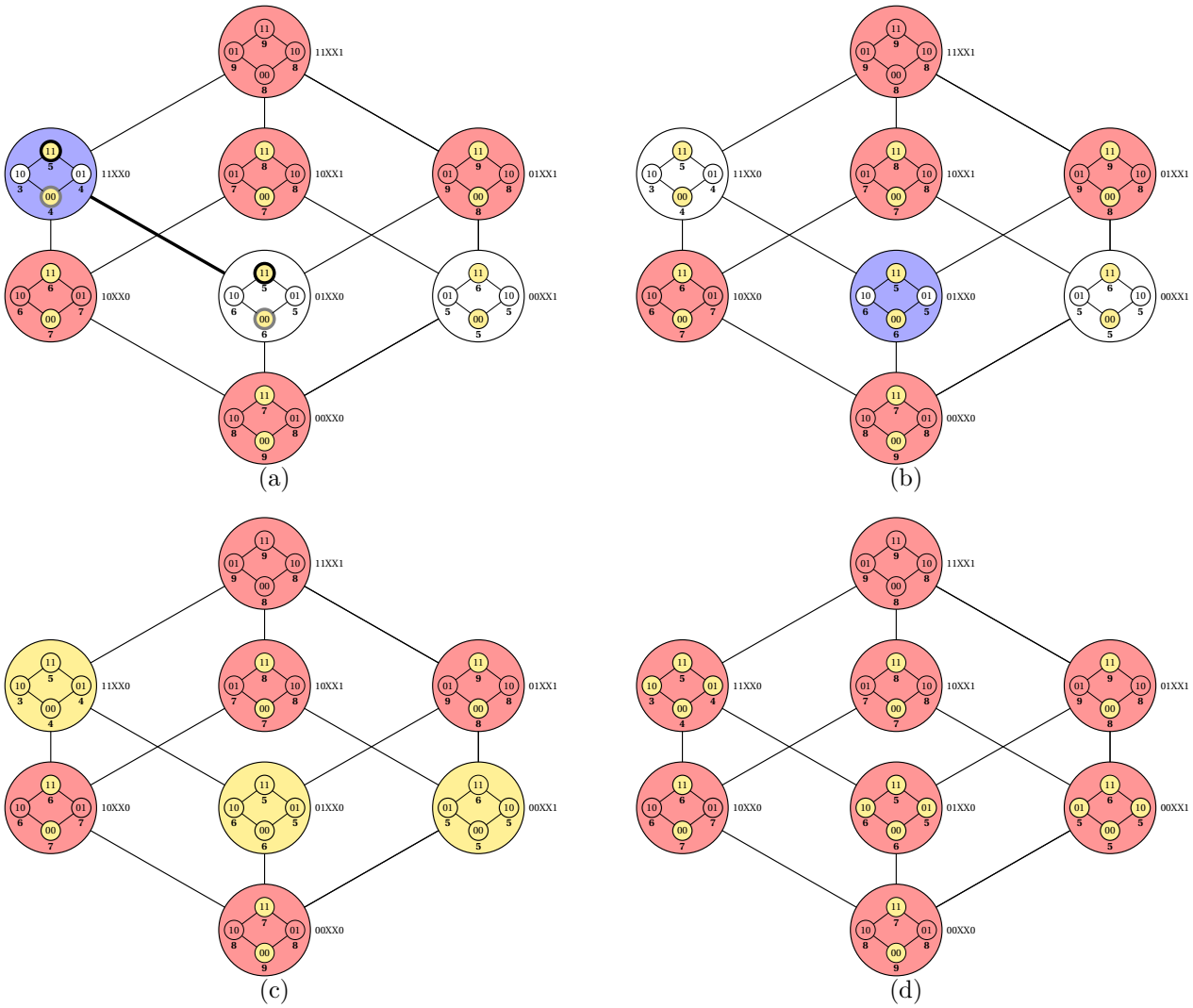


Figura 4.4: Continuação da figura 4.3

- Figura 4.4(a) Nenhuma poda é realizada ao comparar 11XX0 com 01XX0, portanto a última torna-se o nó corrente e a primeira torna-se candidata a conter o mínimo.
- Figura 4.4(b) A parte 01XX0 torna-se o nó corrente, mas como não possui vizinho no espaço de busca é removida do espaço de busca e torna-se candidata a conter o mínimo.
- Figura 4.4(c) Todo nó foi removido do espaço de busca por podas ou por visitas. Resolve-se as partes candidatas a conter o mínimo.
- Figura 4.4(d) O elemento de custo mínimo entre as partes é 11100, que é de fato a solução ótima do problema com custo 3.

4.2.3 Solução das partes

Ao fim do passeio aleatório, teremos uma coleção de partes que precisam ser resolvidas para se obter o conjunto de custo mínimo. Nesta etapa, o `PUCS` constrói para cada parte uma instância auxiliar do problema U-Curve que é equivalente ao problema original, porém restrito

a parte de interesse. Seja $\langle S, c\rangle$ a instância do problema original e S' o conjunto de variáveis fixas no particionamento feito pelo PUCS neste problema, então, dada uma parte $A \in \mathcal{P}(S')$, o conjunto de custo mínimo nesta parte é exatamente a solução ótima do problema U-Curve auxiliar $\langle \overline{S'}, c_A\rangle$ em que $c_A(X) = c(X \cup A)$ para qualquer $X \in \mathcal{P}(\overline{S'})$.

Para solucionar os problemas auxiliares, podemos chamar um outro algoritmo de seleção de características, ótimo ou sub-ótimo, e podemos inclusive chamar o próprio PUCS, tornando o algoritmo recursivo. Chamamos o último algoritmo na sequência de chamadas recursivas de **algoritmo base**; o PUCS é algoritmo base apenas no caso em que cada parte contém apenas um elemento. A escolha do algoritmo base é crítica no desempenho da chamada do PUCS no que diz respeito a uso de recursos computacionais e também na qualidade da solução obtida.

4.3 Parâmetros de funcionamento

Na seção anterior apresentamos a dinâmica básica do algoritmo PUCS, porém por simplicidade não definimos alguns parâmetros que regem o funcionamento do mesmo. Apesar de ser fácil entender a dinâmica do algoritmo sem conhecer estes parâmetros, eles tem papel crítico no desempenho do mesmo, tanto no quesito de uso de recursos computacionais quanto na qualidade da solução encontrada. Estes parâmetros são p, l, e algoritmo base.

O parâmetro p define a quantidade de variáveis fixas no particionamento do espaço de busca e deve estar contido no intervalo $(0, 1]$, sendo a proporção de variáveis que devem ser fixadas; desta forma:

$$|S'| = \lceil |S| * p \rceil$$
$$|\overline{S'}| = |S| - \lceil |S| * p \rceil$$

Portanto, quanto maior o p, maior o tamanho do reticulado externo ($|\mathcal{P}(S')|$) e menor o tamanho dos reticulados internos ($|\mathcal{P}(\overline{S'})|$) e vice-versa. Note que quando p é pequeno o algoritmo PUCS deve ser semelhante ao algorítmo base, já que o tamanho das partes continua semelhante; quando o p é grande, o particionamento é mais "fino" porque as partes se tornam menores e consequentemente há mais partes.

Como vimos na seção 4.2.3, a estrutura criada no particionamento do problema nos permite fazer chamadas recursivas do PUCS. O parâmetro l determina a quantidade de chamadas recursivas que acontecerão até que o algoritmo base seja chamado. Ao fazer chamadas recursivas estamos particionando o espaço de busca seguidas vezes e portanto, assim como o parâmetro p, quando aumentamos o valor de l, o tamanho da parte que será resolvida pelo algoritmo base diminui.

O algoritmo base determina como as partes serão resolvidas. Note que os teoremas 4.2.1 e 4.2.2 garantem que se a hipótese U-Curve for verdadeira, então todas as partes que foram podadas do espaço de busca não contém o mínimo global, portanto se o algoritmo base é ótimo, então o PUCS também é ótimo. Se o algoritmo base for uma heurística, então o PUCS também se comporta como uma heurística, entretanto é provável que a solução encontrada pelo PUCS seja melhor do que a solução dada pelo algoritmo base. Dizemos que isto é provável porque o PUCS faz diversas chamadas ao algoritmo base, uma para cada parte candidata, portanto percorre mais nós do que uma chamada única do algoritmo base.

4.4 Implementação do algoritmo

O algoritmo `PUCS` foi implementado no arcabouço *featsel*, usando a linguagem `C++`. Nesta seção apresentaremos detalhes sobre sua implementação

4.4.1 Controle do espaço de busca

Sempre que um nó do reticulado externo é podado ou visitado ele deve ser removido do espaço de busca, e representar este espaço explicitamente não é uma boa solução devido ao seu tamanho, que é exponencial em relação a quantidade de características fixas. A estrutura de dados utilizada deve ser eficiente tanto para inserções (de intervalos e de pontos do reticulado) quanto para consultas. Escolhemos para nossa implementação usar a estrutura de dados de diagramas de decisão binária reduzidos e ordenados (*Reduced Ordered Binary Decision Diagram* (ROBDD)) [Bry86].

4.4.2 Paralelização do código

Usamos a biblioteca *OpenMP* na paralelização do código. Esta biblioteca nos permite paralelizar o algoritmo de maneira fácil, com anotações que indicam ao compilador como blocos do código fonte podem ser processados paralelamente.

O `PUCS` foi criado com o intuito de ser um algoritmo paralelo para resolver o problema U-Curve. A particionamento do espaço foi feito exatamente para que o processo de paralelização do código fosse simples, facilitando a distribuição de trabalho entre threads e usando o mínimo de comunicação entre as mesmas. Desta forma, para paralelizar o código, basta indicar ao compilador que o particionamento e passeio pelo reticulado deve ser feito pela thread principal enquanto a solução de cada parte pode ser realizada por qualquer outra thread, usando a estrutura de *tasks* da biblioteca *OpenMP*. Desta forma, sempre que o algoritmo visita uma parte que não é podada, cria-se uma *task* que deve solucionar tal parte.

Esta abordagem pode causar cálculos supérfluos, pois partes resolvidas podem ser podadas no decorrer dos passeios aleatórios, no entanto, para evitar tais recálculos seria necessário esperar o percorrimento de todo reticulado externo antes de se resolver as partes. Esta segunda abordagem tornaria necessário armazenar e manter atualizada uma lista de partes candidatas, o que pode ser caro computacionalmente; além disso, como o passeio é feito apenas por uma thread, todas as outras seriam subutilizadas durante o passeio.

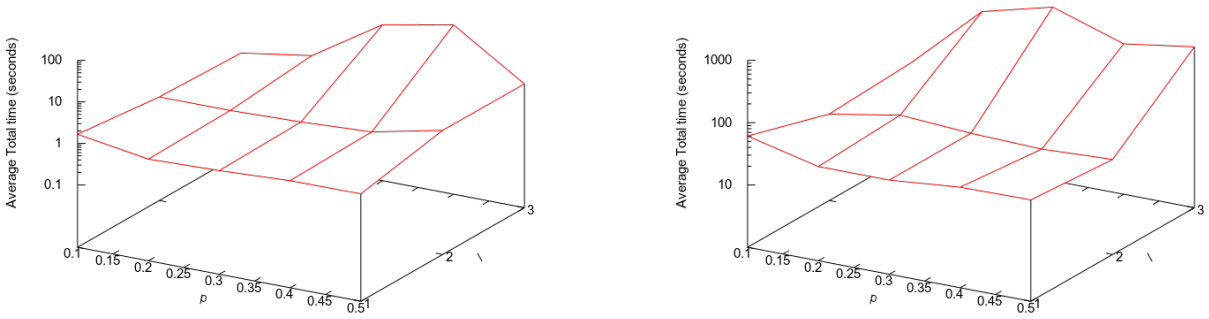
4.5 Ajuste de parâmetros

Antes de discutir o desempenho do algoritmo, precisamos entender como os parâmetros devem ser ajustados para cada tipo de instância do problema U-Curve. Consideramos instâncias pequenas aquelas que podem ser resolvidas otimamente. Estas instâncias costumam ter no máximo por volta de 30 características e algoritmos ingênuos como a busca exaustiva podem, já para estas instâncias, se tornar muito caros computacionalmente. Já as instâncias grandes, que não podem ser resolvidas por algoritmos ótimos, costumam ter mais do que 30 características; utilizam-se para estas algoritmos sub-ótimos, heurísticas.

Instâncias pequenas

Instâncias pequenas podem ser resolvidas otimamente por algoritmos como o `UBB`, `PFS` ou até a busca exaustiva. Portanto, para estas instâncias, vamos usar parâmetros no `PUCS` que garantem que ele será ótimo. Para fazer isto, basta utilizar como algoritmo base um algoritmo ótimo. A decisão de qual algoritmo ótimo usar como base depende dos recursos computacionais disponíveis e da topologia do problema; se, por exemplo, a função de custo é complicada, devemos escolher um algoritmo que faz poucos cálculos de custo, como o `PFS` ou `UBB-PFS`.

Para os outros dois parâmetros, p e l, devemos escolher de forma que a granularidade da partição seja moderada. Como estas instâncias são pequenas, uma partição muito fina pode criar muitas partes pequenas, fazendo com que o trabalho de se criar e escalonar estas várias tarefas pequenas comprometa o tempo total de execução. Por outro lado, se utilizarmos valores pequenos para p e l é possível que o particionamento seja muito pouco fino, não gerando um número de tarefas que use todo o poder computacional da máquina usada.



(a) (b)

Figura 4.5: Gráficos com o tempo de execução médio de acordo com os parâmetros p e l. A figura 4.5(a) mostra este valor para uma instância artificial com 20 características enquanto a figura 4.5(b) mostra este valor para uma instância de 25 características.

Melhorar estes gráficos para justificar o uso de p

A figura 4.5 mostra como o tempo de execução evolui quando aumentamos o valor dos parâmetros p e l.

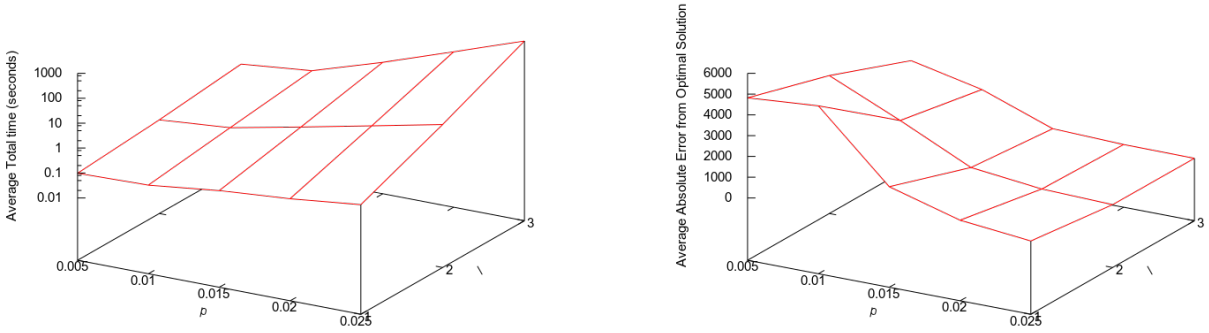
Instâncias grandes

Instâncias grandes costumam não ser resolvidas por algoritmos ótimos, pois estes demandam recursos computacionais as vezes não disponíveis. Desta forma, resolve-se o problema U-Curve com heurísticas, como o `Sequential Forward Floating Selection` (`SFFS`) [PNK94] ou `Best-First Search` (`BFS`) [KJ97]. Para o `PUCS` se tornar uma heurística, basta utilizar um destes algoritmos como algoritmo base.

Quando o `PUCS` é ótimo, os parâmetros p e l interferem apenas no tempo de execução do algoritmo, mas quando tornamos este algoritmo uma heurística, estes parâmetros passam a interferir também na qualidade das soluções obtidas. Quanto mais fino é o particionamento feito, mais partes são geradas e resolvidas pelo `PUCS`, e como cada parte é resolvida com uma

chamada do algoritmo base, temos que quanto mais granular é o particionamento, mais percorrimentos disjuntos acontecerão no reticulado. Desta maneira, quando aumentamos o p e l, mais buscas são feitas e é provável que a solução encontrada seja melhor.

Entretanto, não podemos nos esquecer que estes parâmetros ainda influenciam o tempo de execução do algoritmo. Desta maneira não podemos aumentar os valores destes parâmetros arbitrariamente até atingir a solução com qualidade necessária, e sim aumentar estes parâmetros enquanto houverem recursos computacionais compatíveis.



(a) (b)

Figura 4.6: Desempenho do algoritmo `PUCS` *para uma instância artificial de 200 características utilizando o* `SFFS` *como algoritmo base e variando os parâmetros p e l. A figura 4.6(a) mostra o tempo médio de execução, note que este aumenta quando aumentamos os parâmetros. A figura 4.6(b) mostra a diferença absoluta média entre o custo do conjunto obtido e do conjunto ótimo.*

A figura 4.6 mostra como o tempo de execução e qualidade da solução se comportam quando aumentamos os valores de p e l. Confirmando as nossas expectativas, quando amentamos p e l a qualidade da solução encontrada é mais perto da ótima (o erro diminui, no gráfico da figura 4.6(b)), e o tempo de execução aumenta.

4.6 Experimentos com instâncias artificiais

Nesta seção apresentamos testes feitos com o `PUCS` ao solucionar instâncias artificiais do problema U-Curve onde função de custo utilizada é a de soma de subconjuntos. Estes testes foram feitos no arcabouço *featsel* e foram rodados em uma servidora de 64 cores.

Como parâmetro p utilizamos um valor tal que o número de variáveis fixas é sempre 10, o que não compromete a escalabilidade e deve ser o suficiente para gerar um número de partes que façam todos os cores da servidora trabalharem.

4.6.1 Experimentos ótimos

Estes experimentos são feitos em instâncias do problema U-Curve de tamanho pequeno, que podem ser resolvidos por algoritmos ótimos. Comparamos o `PUCS` com o `UBB`, `PFS` e também com o algoritmo criado na seção 3.5, o `UBB-PFS`. Como parâmetros para estas instâncias, fixamos. Utilizamos como parâmetros: p tal que número de variáveis fixas é 10; $l = 2$; e o `UBB` como algoritmo base.

Instance		Total time (sec)			
$\|S\|$	$2^{\|S\|}$	UBB	PFS	UBB-PFS	PUCS
10	1024	0.008 ± 0.017	0.013 ± 0.014	0.024 ± 0.004	0.026 ± 0.023
11	2048	0.007 ± 0.002	0.017 ± 0.004	0.027 ± 0.005	0.631 ± 0.316
12	4096	0.010 ± 0.003	0.025 ± 0.008	0.034 ± 0.007	0.814 ± 0.570
13	8192	0.015 ± 0.006	0.052 ± 0.017	0.048 ± 0.012	1.099 ± 0.562
14	16384	0.025 ± 0.013	0.095 ± 0.039	0.074 ± 0.023	1.816 ± 1.145
15	32768	0.039 ± 0.029	0.179 ± 0.075	0.114 ± 0.042	2.088 ± 1.107
16	65536	0.106 ± 0.053	0.386 ± 0.159	0.214 ± 0.078	3.591 ± 1.874
17	131072	0.222 ± 0.105	0.708 ± 0.368	0.358 ± 0.181	3.370 ± 2.228
18	262144	0.311 ± 0.240	1.302 ± 0.762	0.666 ± 0.346	5.683 ± 3.479
19	524288	0.731 ± 0.447	2.867 ± 1.661	1.381 ± 0.794	7.158 ± 7.202
20	1048576	1.542 ± 0.886	6.119 ± 3.403	2.873 ± 1.537	15.634 ± 24.817

Tabela 4.1: Comparação de tempo de execução de algoritmos ótimos para o problema U-Curve. O `PUCS` foi o algoritmo mais lento para estas instâncias.

Instance		# Calls of cost function			
$\|S\|$	$2^{\|S\|}$	UBB	PFS	UBB-PFS	PUCS
10	1024	617.7 ± 368.0	621.5 ± 147.1	603.3 ± 228.1	730.6 ± 319.4
11	2048	1225.5 ± 748.6	1125.1 ± 354.9	1163.5 ± 450.5	1268.4 ± 782.2
12	4096	2396.1 ± 1724.9	1811.1 ± 680.9	1982.7 ± 772.1	1693.9 ± 1501.7
13	8192	5144.4 ± 3182.5	3982.3 ± 1399.0	4292.6 ± 1592.1	4125.2 ± 2859.2
14	16384	9777.9 ± 6350.5	7559.3 ± 3308.7	8262.6 ± 3395.1	7879.4 ± 6534.4
15	32768	16496.8 ± 13431.8	13866.9 ± 6098.5	14390.0 ± 6576.1	14693.8 ± 11279.2
16	65536	46811.7 ± 23795.2	28667.9 ± 12109.5	31038.2 ± 11381.7	30742.6 ± 22862.0
17	131072	99140.0 ± 46959.9	49896.5 ± 26087.6	53171.0 ± 26159.2	54582.4 ± 47866.6
18	262144	141399.9 ± 108893.4	89345.1 ± 50833.7	96310.4 ± 47545.5	95279.0 ± 88709.4
19	524288	339805.7 ± 206774.3	186964.7 ± 108756.5	199350.3 ± 101879.9	201333.1 ± 174767.8
20	1048576	709486.2 ± 404264.5	390060.3 ± 218822.5	406496.1 ± 208354.8	445381.0 ± 358644.6

Tabela 4.2: Comparação de número de chamadas da função custo em algoritmos ótimos do problema U-Curve. O `PUCS` faz menos chamadas da função custo que o `UBB`, porém faz mais chamadas do que o `PFS` e `PFS-UBB`.

A tabela 4.1 mostra o tempo de execução médio dos algoritmos `UBB`, `PFS`, `UBB-PFS` e `PUCS`. O último foi o algoritmo que gastou mais tempo para estas instâncias. A tabela 4.2 mostra, para os mesmos algoritmos, o número médio de chamadas da função custo. O `PUCS` computou menos nós do que o `UBB`, e computou pouco mais nós do que o `PFS` e `UBB-PFS`.

4.6.2 Experimentos sub-ótimos

Estes experimentos foram feitas em instâncias do problema U-Curve que são grandes e geralmente não podem ser resolvidas por algoritmos ótimos. Comparamos o desempenho do `PUCS` contra as heurísticas `SFFS` [PNK94] e `BFS` [KJ97]. Utilizamos como parâmetros: p tal que o número de variáveis fixas é 10; $l = 1$; e como algoritmo base, utilizamos a heurística `Sequential Forward Selection SFS` [Whi71].

A figura 4.7(a) compara o tempo de execução do `PUCS` com heurísticas. Apesar do desvio padrão grande, podemos observar que, com os parâmetros citados, o `PUCS` foi o algoritmo mais lento entre os testados. A figura 4.7(b) mostra para cada uma das três heurísticas testadas a proporção de vezes em que o algoritmo achou a solução de menor custo entre os três. Neste

teste, o PUCS teve proporção de melhor solução de 100% para todos os tamanhos de instâncias testadas, isto é, a solução encontrada por este algoritmo teve custo sempre menor ou igual do que as soluções encontradas pelo SFFS e BFS.

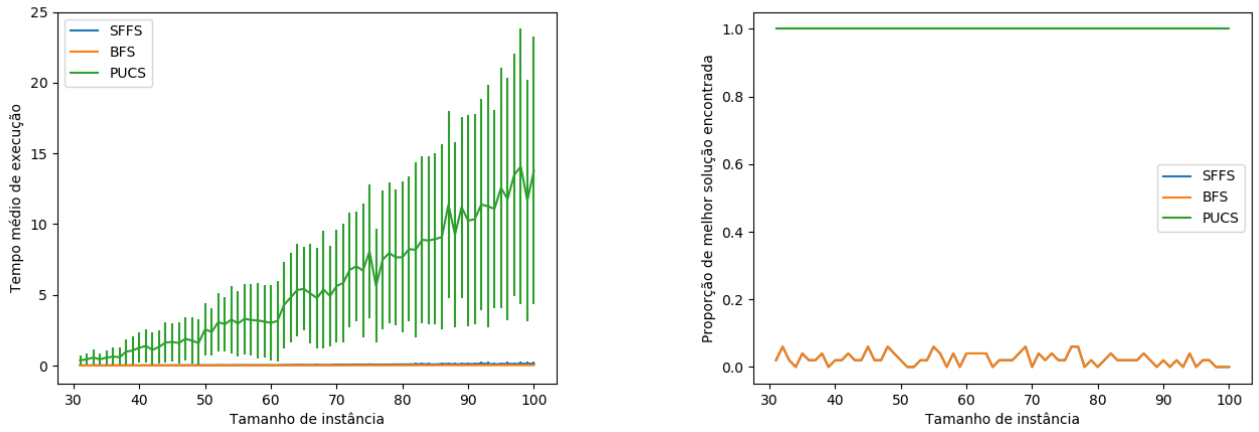


Figura 4.7: Comparação entre os algoritmos SFFS, BFS e PUCS. A figura 4.7(a) mostra o tempo médio de execução dos algoritmos. A figura 4.7(b) mostra a proporção de vezes em que cada algoritmo achou a melhor resposta entre os três.

4.6.3 Comentários sobre os experimentos

Os experimentos ótimos mostraram que o desempenho do PUCS foi pior do que outros algoritmos ótimos. Isto significa que o trabalho do particionamento, percorrimento do reticulado externo, e do controle de concorrência não são compensados pelos ganhos de processamento paralelo.

Capítulo 5

Exemplos de aplicação em aprendizado de máquina

Com o objetivo de testar os algoritmos de seleção de características em problemas reais, decidimos aplicá-los a problemas de seleção de modelos de aprendizado computacional. Usamos as características selecionadas para definir um modelo de aprendizado e calculamos o erro de classificação médio para este modelo, usando validação cruzada. Os dados utilizados estão disponíveis no University of California Irvine (UCI) Machine Learning Repository [Lic13].

5.1 Seleção de características em aprendizado de máquina

Os dados disponíveis no repositório UCI Machine Learning Repository são dados de treinamento e teste de aprendizado de máquina supervisionado. Estes dados são compostos por múltiplos exemplos de tuplas com valores de características do conjunto S e o rótulo da tupla observada, Y. Um modelo de aprendizado determina um conjunto de possíveis classificadores para o problema; quando usamos seleção de características, indicando que um conjunto $X \in \mathcal{P}(S)$ descreve melhor os dados, estamos escolhendo como modelo de aprendizado o conjunto de classificadores que leva em consideração apenas as características de X.

Para que o conjunto X de características selecionadas de fato determine um bom conjunto de classificadores, precisamos construir o problema de seleção de características de maneira que isto aconteça. Então, além de determinar que o conjunto de características seja S, precisamos escolher a função de custo c para o problema. Como discutido na seção 2.2, a função de custo entropia condicional média (MCE) faz bem este papel. Além disso, a função MCE induz curvas em U, com poucas violações, nas cadeias do reticulado Booleano. Por conta das violações, este problema não é o U-Curve, porém, como visto em [Rei12], podemos aproximar a solução do problema original pela solução encontrada assumindo que estamos no caso U-Curve.

A função de custo MCE que utilizamos neste trabalho, disponível no arcabouço *feat-sel* [Rei+17], assume que os valores das características são discretos, portanto ainda é necessário discretizar os dados de treinamento do UCI que têm variáveis contínuas para fazermos a seleção de características no arcabouço. A estratégia que utilizamos para tal tarefa é de discretizar os valores de características contínuas por quartis, mapeando seus valores para um inteiro entre 0 e 3. Existem entretanto outras estratégias mais elaboradas, como a de Fayyad e Irani [FI93] que separa dados por classes minimizando a entropia média.

5.2 Support Vector Machine com kernel linear

A seleção de características é apenas uma etapa da seleção de modelo de aprendizado. Vamos então especificar mais o conjunto de possíveis classificadores para os problemas que trataremos neste capítulo. Fazemos isto ao decidir que o modelo de aprendizado que usaremos é o de Support Vector Machine (SVM) [CL11].

Mais especificamente, vamos trabalhar com classificadores SVM com kernel linear e multiclasse. O funcionamento de um classificador SVM binário (duas classes) é simples: dado um conjunto de dados de treinamento com dois rótulos possíveis, o classificador determina um hiperplano que separa no espaço os dados de treinamento, assim para classificar um dado, basta responder em qual lado do hiperplano este dado está localizado. Se houverem k classes, então podemos criar $k(k-1)$ classificadores binários para cada par de classes e então, para rotular um dado, utilizamos um esquema de votação em que a classe mais votada dentre todos os estes classificadores dá o rótulo ao ponto.

O modelo de SVM linear tem um parâmetro C que determina como o hiperplano é posicionado no espaço de pontos de treinamento. Para valores pequenos de C o hiperplano será posicionado dando preferência a ter uma margem grande para pontos de treinamento corretamente classificados, mesmo que isto implique em pontos de treinamento mal classificados; para valores maiores de C, a preferência é dada para classificar corretamente os dados de treinamento, mesmo que a margem do hiperplano seja pequena. Em nosso trabalho, utilizamos $C = 100$, pois este parâmetro é regulador, isto é, diminui a complexidade do modelo para se obter melhores resultados com poucas amostras; mas isto é o que queremos fazer também com seleção de características, portanto o valor alto de C garante que a diminuição da complexidade do modelo é feita de fato pela seleção de características.

5.3 Validação de modelos

Para fazer a validação dos modelos de aprendizado gerados, precisamos estimar o erro médio de classificadores treinados em cada um destes modelos. Para fazer isto é necessário separar os dados entre dados de treinamento e dados de teste, mas como o número de amostras é geralmente pequeno, fazemos o procedimento de classificação e estimação de erro para várias escolhas de conjuntos de treinamento e teste; chamamos este tipo de validação, que usa o mesmo exemplo como teste e treinamento, de validação cruzada.

Para as instâncias em que o número de exemplos é superior a 100, usaremos a validação cruzada 10-*fold*, para outras utilizaremos a validação *leave-one-out*. A validação cruzada 10-*fold* separa as amostras em 10 grupos e usa cada um deles para calcular o erro do classificador treinado com os outros nove. Na abordagem *leave-one-out* separamos as n amostras em n grupos e fazemos o mesmo procedimento.

5.4 Experimentos com problemas de classificação

Nesta seção apresentamos projetos de classificadores feitos com seleção de características para conjuntos de dados do University of California Irvine (UCI) Machine Learning Repository [Lic13]. Após a fase de seleção de modelo, feita com seleção de características, fazemos a validação cruzada de modelos, como definidos na seção 5.2, utilizando a biblioteca libsvm [CL11].

5.4.1 Descrição dos conjuntos de dados

Iris

Este conjunto de dados é mais famoso do repositório UCI, e apresenta dados de plantas com flor do gênero Iris, conhecidas popularmente como lírio. Os dados descrevem cada planta com quatro variáveis, que são comprimento e largura de pétalas e sépalas, e as rotulam em três tipos: *Iris setosa*, *Iris Versicolor* e *Iris Virginica*. Este conjunto possui 150 exemplos de rotulações.

Wine

Este conjunto de dados descreve vinhos em 13 variáveis, como porcentagem alcoólica, cor, intensidade de cor, entre outros. Os 178 exemplos de vinhos são classificados entre três tipos.

Thoracic Surgery

Este conjunto de dados classifica dados de pacientes que passaram por uma ressecção pulmonar. Este procedimento é um tratamento de câncer de pulmão em que se remove do paciente os tecidos do pulmão que são cancerígenos, assim como os tecidos saudáveis das suas periferias. Os pacientes são classificados em duas classes complementares que indicam se o paciente sobreviveu por um ano após o procedimento. Os dados possuem 470 exemplos e descrevem os pacientes em 17 atributos.

Zoo

Este conjunto de dados contém exemplos de classificação de animais de zoológico. Os animais são classificados em 7 possíveis grupos e são descritos em 17 variáveis que podem ser Booleanas, como presença de pelos, ser aquático, etc., ou inteiras, como a quantidade de pernas. Este conjunto possui 101 exemplos classificados.

Breast Cancer

O objetivo deste conjunto de dados é classificar amostras de biópsia de tumores mamários. Estas amostras contém informações de 31 variáveis e são classificadas em tumores benignos ou malignos. Este conjunto possui 700 exemplos de classificação de biópsia.

Lung Cancer

Estes dados apresentam exemplos de classificação de tipos de câncer de pulmão. São utilizadas 56 características para descrever um paciente, mas nenhuma delas é especificada pelos autores deste conjunto de dados. Existem 3 possíveis rótulos para cada paciente, e um total de 32 exemplos.

Promoters

Chamamos de promotora uma sequência de nucleotídeos (A, T, C, G) do DNA que indica o começo de uma região da fita onde deve haver a transcrição. Este conjunto de dados contém 106 exemplos de sequências de DNA da bactéria *E. coli*. Cada sequência é formada por 57 variáveis, que são nucleotídeos, e é rotulada em promotora ou não promotora.

5.4.2 Resultados

Capítulo 6

Conclusão

Bibliografia

[AG+18] Esmaeil Atashpaz-Gargari, Marcelo S. Reis, Ulisses M. Braga-Neto, Junior Barrera e Edward R. Dougherty. «A fast Branch-and-Bound algorithm for U-curve feature selection». Em: *Pattern Recognition* 73.Supplement C (2018), pp. 172 –188. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.08.013. URL: http://www.sciencedirect.com/science/article/pii/S0031320317303254.

[Bar+07] J. Barrera, R. M. Cesar-Jr, D.C. Martins-Jr, R.Z.N Vencio, E. F. Merino, M. M. Yamamoto, F. G. Leonardi, C. A. B. Pereira e H. A. Portillo. «Constructing probabilistic genetic networks of *Plasmidium falciparum* from dynamical expression signals of the intraerythrocytic development cycle.» Em: *Methods of Microarray Data Analysis* V (2007), pp. 11–26.

[Bry86] R. E. Bryant. «Graph-Based Algorithms for Boolean Function Manipulation». Em: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. ISSN: 0018-9340. DOI: 10.1109/TC.1986.1676819.

[CL11] Chih-Chung Chang e Chih-Jen Lin. «LIBSVM: A Library for Support Vector Machines». Em: *ACM Trans. Intell. Syst. Technol.* 2.3 (mai. de 2011), 27:1–27:27. ISSN: 2157-6904. DOI: 10.1145/1961189.1961199. URL: http://doi.acm.org/10.1145/1961189.1961199.

[DMJ06] R.M. Cesar-Jr an J. Barrera D.C. Martins-Jr. «W-operator window design by minimization of mean conditional entropy». Em: *Patter Analysis & Applications* (2006), pp. 139–153.

[FI93] Usama M. Fayyad e Keki B. Irani. «Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning». Em: (1993).

[JCJB04] D. C. Martins Jr, R. M. Cesar-Jr e J. Barrera. «W-operator window design by maximization of training data information». Em: *Computer Graphics and Image Processing, 2004. Proceedings. 17th Brazilian Symposium* (2004), pp. 162–169.

[KJ97] Ron Kohavi e George H. John. «Wrappers for feature subset selection». Em: *Artificial Intelligence* 97.1 (1997). Relevance, pp. 273 –324. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(97)00043-X. URL: http://www.sciencedirect.com/science/article/pii/S000437029700043X.

[Lic13] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: http://archive.ics.uci.edu/ml.

[PNK94] P. Pudil, J. Novovičová e J. Kittler. «Floating search methods in feature selection». Em: *Pattern Recognition Letters* 15.11 (1994), pp. 1119 –1125. ISSN: 0167-8655. DOI: https://doi.org/10.1016/0167-8655(94)90127-9. URL: http://www.sciencedirect.com/science/article/pii/0167865594901279.

[Rei+17] Marcelo S. Reis, Gustavo Estrela, Carlos Eduardo Ferreira e Junior Barrera. «featsel: A framework for benchmarking of feature selection algorithms and cost functions». Em: *SoftwareX* 6 (2017), pp. 193 –197. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2017.07.005. URL: https://www.sciencedirect.com/science/article/pii/S2352711017300286.

[RFB14] M. S. Reis, C. E. Ferreira e J. Barrera. «The U-curve optimization problem: improvements on the original algorithm and time complexity analysis». Em: *ArXiv e-prints* (jul. de 2014). arXiv: 1407.6067 [cs.LG].

[Whi71] A. W. Whitney. «A Direct Method of Nonparametric Measurement Selection». Em: *IEEE Transactions on Computers* C-20.9 (1971), pp. 1100–1103. ISSN: 0018-9340. DOI: 10.1109/T-C.1971.223410.

[Rei12] M. S. Reis. «Minimization of decomposable in U-shaped curves functions defined on poset chains – algorithms and applications». Tese de doutoramento. University of Sao Paulo, 2012.