Universidade Estadual de Campinas

Instituto de Matemática, Estatística e
Computação Científica

MARTYNA JOANNA RUCINSKA PEREIRA

Análise de Algoritmo para Problema de Particionamento de Posets

CAMPINAS

2018

MARTYNA JOANNA RUCINSKA PEREIRA

Análise de algoritmo para problema de particionamento de posets

Dissertação apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestra em Matemática.

Orientador: Marcelo Firer

Este exemplar corresponde à versão final da Dissertação defendida pela aluna Martyna Joanna Rucinska Pereira, e orientada pelo Prof. Dr. Marcelo Firer.

CAMPINAS

2018

Agência(s) de fomento e nº(s) de processo(s): CNPq, 134559/2016-9

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Márcia Pillon D'Aloia - CRB 8/5180

R828a

Rucinska Pereira, Martyna Joanna, 1989-
Análise de algoritmo para problema de particionamento de posets / Martyna Joanna Rucinska Pereira. – Campinas, SP : [s.n.], 2018.

Orientador: Marcelo Firer.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica.

1. Problema de particionamento de posets. 2. Particionamento de números. 3. Conjuntos parcialmente ordenados. 4. Karmarkar-Karp, Algoritmo de. I. Firer, Marcelo, 1961-. II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Computação Científica. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Analysis of an algorithm for poset partitioning problem
Palavras-chave em inglês:
Poset partitioning problem
Number partitioning
Partially ordered sets
Karmarkar-karp, Algorithm
Área de concentração: Matemática
Titulação: Mestra em Matemática
Banca examinadora:
Marcelo Firer [Orientador]
Rafael Gregorio Lucas D'Oliveira
Marcio Argollo Ferreira de Menezes
Data de defesa: 15-08-2018
Programa de Pós-Graduação: Matemática

Dissertação de Mestrado defendida em 15 de agosto de 2018 e aprovada pela banca examinadora composta pelos Profs. Drs.

Prof(a). Dr(a). MARCELO FIRER

Prof(a). Dr(a). RAFAEL GREGORIO LUCAS D'OLIVEIRA

Prof(a). Dr(a). MARCIO ARGOLLO FERREIRA DE MENEZES

A Ata da Defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria de Pós-Graduação do Instituto de Matemática, Estatística e Computação Científica.

Dedico este trabalho inteiramente à minha família: ao meu marido e companheiro de todas as horas e aos meus filhos Pedro, que sempre traz muita alegria, e Hanna, que está prestes a chegar neste mundo.

Agradecimentos

Agradeço a todas as pessoas e instituições que de alguma forma contribuíram para a realização deste trabalho, em especial:

Ao CNPq pela bolsa concedida.

Ao meu orientador Marcelo Firer, por ter me acolhido, por ter permitido a realização deste trabalho e pela orientação.

Ao meu marido, Anderson, que me deu apoio e incentivo na hora certa, aos meus sogros que estavam sempre dispostos a ajudar, aos meus pais que, mesmo de longe, estavam sempre presentes, e a toda minha família que, com muito carinho, não mediu esforço para que eu chegasse até esta etapa de vida.

Resumo

Este trabalho é dedicado ao estudo do desempenho de um algoritmo para resolver o problema de particionamento de posets, que chamaremos de algoritmo de Karmarkar-Karp Generalizado Completo (KKGC).

O problema de particionamento de posets é uma generalização do famoso problema NP-difícil, que é o problema da partição. O problema de particionamento de posets pode ser interpretado como um problema de distribuição das tarefas (processamento paralelo com tarefas em comum): dadas duas máquinas idênticas e uma lista de tarefas em que certas tarefas dependem de outras, precisamos distribuí-las entre as máquinas de tal modo que o trabalho seja feito o mais rápido possível. As dependências entre as tarefas geram um poset, no qual uma tarefa T é maior do que uma tarefa U, se T depende de U. Por ser uma generalização do processamento paralelo associado ao problema de particionamento clássico, o problema de particionamento de posets, surgido no contexto de códigos corretores de erros, torna-se relevante também no contexto de ciências de computação.

Estudamos o desempenho do algoritmo KKGC do ponto de vista de tempo de execução e precisão da solução. Baseamos a nossa análise na análise feita pelo Richard E. Korf em "A complete anytime algorithm for number partitioning" sobre o algoritmo de Karmarkar-Karp Completo (KKC) que é o algoritmo mais eficiênte conhecido usado para resolver o problema da partição, já que o problema de particionamento de posets é uma generalização do problema da partição, e o algoritmo KKGC é uma certa generalização do algoritmo KKC.

Palavras-chave: problema de particionamento de posets, particionamento de números, conjuntos parcialmente ordenados, algoritmo de karmarkar-karp

Abstract

This work is dedicated to the study of the performance of an algorithm to solve the poset partitioning problem, which we will call the Karmarkar-Karp Generalized Complete algorithm (KKGC).

The poset partitioning problem is a generalization of the famous NP-difficult problem, which is the partition problem (number partitioning). The poset partitioning problem can be interpreted as a task distribution problem (parallel processing with common tasks): given two identical machines and a list of tasks in which certain tasks depend on others, we need to distribute them between the machines so that the work is done as quickly as possible. The dependencies between the tasks generate a poset, in which a task T is larger than a task U, if T depends on U. Because it is a generalization of parallel processing associated with the classical partitioning problem, the poset partitioning problem that emerged in the context of error-correcting codes, becomes relevant also in the context of computer science.

We studied the performance of the KKGC algorithm from a runtime point of view and solution accuracy. We base our analysis on the analysis done by Richard E. Korf in "A complete anytime algorithm for number partitioning" on the Karmarkar-Karp Complete (KKC) algorithm which is the most efficient known algorithm used to solve the partition problem, since the poset partitioning problem is a generalization of the partition problem, and the KKGC algorithm is a certain generalization of the KKC algorithm.

Keywords: poset partitioning problem, number partitioning, partially ordered sets, karmarkar-karp algorithm

Lista de Figuras

1 Balança de dois pratos. . . . 14
2 Desempenho dos algoritmos KK e ganancioso 17
3 Árvore para particionar o conjunto $\{8, 7, 6, 5, 4\}$ 19
4 Árvore para particionar o conjunto $\{8, 7, 6, 5, 4\}$ com os ramos podados 19
5 Ordem dos caminhos construidos da árvore 20

6 Diagrama de Hasse de um poset 23
7 Diagrama de Hasse do poset P 25
8 Poset com uma partição que não é ótima com a discrepância 0 26
9 Poset com a partição ótima cuja discrepância não é a menor possível 27
10 Árvore de busca para o Exemplo 3.11 33
11 Árvore de busca para o Exemplo 3.12 35

12 Desempenho do CKK apresentado pelo Korf 37
13 Um DAG com 6 vértices 40
14 Um DAG e o diagrama de Hasse do poset obtido a partir dele 41
15 Um poset representado pelo diagrama de Hasse e o DAG obtido a partir dele . 41
16 Os passos para construir o vetor dos outpoints de um DAG 43
17 Diagrama de Hasse de um poset após de enumerar os vértices do grafo correspondente 44
18 Comparação do tempo de execução dos algoritmos força-bruta e KKGC 50
19 Comparação do tempo de execução dos algoritmos KKG e ganancioso 51
20 Comparação do desempenho dos algoritmos KKG e ganancioso 52
21 Comparação do tempo de execução dos algoritmos KKGC e força-bruta 54
22 Viabilidade do algoritmo KKGC_{10000} dos posets com $n = 100$ 55
23 Viabilidade do algoritmo KKGC_{10000} dos posets com $n = 500$ 56
24 Viabilidade do algoritmo KKGC_{10000} dos posets com $n = 1000$ 56
25 Distribuição da solução ótima entre as folhas para os posetes de tamanho 100 57
26 Comparação da eficiência dos algoritmos: ganancioso, KKG, KKGC_4 e KKGC_{64} 59
27 Comparação da eficiência dos algoritmos: ganancioso e KKGC_r 60

Sumário

1 Introdução **11**

2 Problema de particionamento **13**

2.1 Particionamento de números 13

2.2 Busca por força bruta 15

2.3 Algoritmo ganancioso 15

2.4 Heurística de Karmarkar-Karp 16

2.5 Algoritmo de Karmarkar-Karp Completo 18

3 Problema de particionamento de posets **21**

3.1 Conjuntos parcialmente ordenados 21

3.2 Surgimento do algoritmo KKGC 24

3.2.1 Caso dos ideais disjuntos 24

3.2.2 Caso geral 26

3.3 Algoritmo de Karmarkar-Karp Generalizado Completo 27

4 Desempenho do algoritmo KKGC **36**

4.1 Métodos de análise 36

4.2 Geração de posets 38

4.2.1 Posets e DAGs 39

4.2.2 Distribuição de posets 45

4.2.3 Algoritmo 47

4.3 Resultados 48

4.3.1 Casos fáceis 49

4.3.2 Casos difíceis 53

5 Perspectivas **62**

Referências **64**

Apêndice A **65**

1 Introdução

Um dos mais famosos problemas NP-difíceis é o problema da partição (neste trabalho chamado de problema de particionamento) da ciência de computação. A tarefa é particionar um conjunto finito S de números naturais em dois subconjuntos S_1, S_2 de tal forma que a diferença entre a soma dos elementos de S_1 e a soma dos elementos de S_2 seja minimizada. Ele pode ser visto como a distribuição das tarefas entre as duas máquinas idênticas com o objetivo de minimizar o tempo de execução de todo processo.

No texto [1], foi apresentada a generalização deste problema para o problema de particionamento de posets. Assim como o problema de particionamento clássico, este também pode ser interpretado como um problema de distribuição das tarefas (processamento paralelo com tarefas em comum): dadas duas máquinas idênticas e uma lista de tarefas em que certas tarefas dependem de outras, precisamos distribui-las entre as máquinas de tal modo que o trabalho seja feito o mais rápido possível. As dependências entre as tarefas geram um poset, no qual uma tarefa T é maior do que uma tarefa U, se T depende de U. Por ser uma generalização do processamento paralelo associado ao problema de particionamento clássico, o problema de particionamento de posets, surgido no contexto de códigos corretores de erros, torna-se relevante também no contexto de ciências de computação.

O nosso trabalho será dedicado ao estudo do desempenho de um algoritmo desenvolvido em [1] para resolver o problema de particionamento de posets, que chamaremos de algoritmo de Karmarkar-Karp Generalizado Completo (KKGC). Este algoritmo é uma certa generalização do algoritmo Karmarkar-Karp Completo (KKC) que é usado no problema de particionamento clássico. Assim como no caso do algoritmo KKC, por problema de particionamento de posets ser um problema NP-difícil, temos somente uma heurística. Estudamos o desempenho dela do ponto de vista de tempo de execução e precisão da solução. Baseamos a nossa análise na análise feita pelo Richard E. Korf em [4] sobre o algoritmo KKC.

A partir dos resultados que obtivemos, concluímos que mesmo que o problema de particionamento de posets seja um problema NP-difícil, a maioria das instâncias é fácil de ser resolvida, pois mais de $99,6\%$ dos posets têm no máximo 3 elementos maximais. Portanto, em casos genéricos não é necessário usar um algoritmo mais sofisticado que o algoritmo

de busca por força bruta. Já nos casos onde um poset possui muitos elementos maximais, uma restrição do KKGC, que chamamos de KKGC$_r$, apresenta grande vantagem sobre o algoritmo de força-bruta e, dependendo do valor de r, obtém melhores resultados que o algoritmo ganancioso.

A primeira parte do trabalho (Capítulos 2 e 3) consiste em apresentar os problemas de particionamento clássico e de posets, e introduzir o algoritmo KKGC a ser estudado.

A segunda parte (Capítulo 4) será completamente destinada ao testes feitos e resultados obtidos da análise do desempenho do algoritmo, mencinando também as dificuldades encontradas no caminho.

2 Problema de particionamento

Neste capítulo falamos sobre o famoso problema NP-difícil da ciência de computação que é o problema de particionamento. Em vários textos, ele é chamado de problema da partição, porém, nós vamos se referir a ele como o problema de particionamento para não ser confundido com o problema da teoria dos números que é a partição de um inteiro. Às vezes, chamaremos ele também de problema de particionamento clássico para enfatizar a diferença entre este e o problema de particionamento de posets que vamos introduzir no Capítulo 3. Por ser um problema NP-difícil, não existem algoritmos eficiêntes que garantam o resultado ótimo.

Na primeira seção introduzimos o problema de particionamento e nas seções seguinte apresentamos alguns algoritmos para resolvé-lo, entre eles, busca por força bruta, algoritmo ganancioso, heurística de Karmarkar-Karp, algoritmo de Karmarkar-Karp Completo (KKC).

2.1 Particionamento de números

Considere um conjunto finito de números, sem perda de generalidade podemos considerá-los naturais. *Será que existe uma partição dele em dois subconjuntos A e B, tal que a soma dos elementos de conjunto A é igual a soma dos elementos de conjunto B?*

Esta pergunta representa o **problema de particionamento**. Em geral, a resposta é *não*. De modo mais preciso, podemos formular o problema da seguinte maneira: dada uma lista finita S de números inteiros positivos, queremos achar uma partição de $S = S_1 \dot\cup S_2$ que minimize a expressão

$$\Delta(S_1, S_2) = \left| \sum_{x \in S_1} x - \sum_{y \in S_1} y \right|$$

chamada de **discrepância**. Falamos que tal partição tem a **discrepância mínima** e a denotamos por

$$\Delta^*(S) = \min_{S_1 \dot\cup S_2 = S} \Delta(S_1, S_2),$$

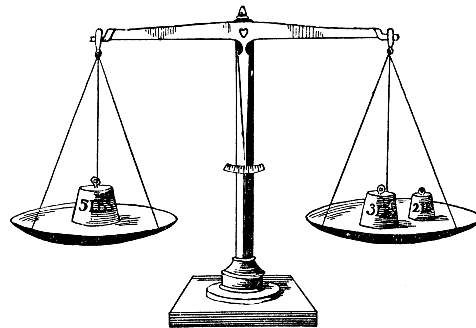
onde $\dot\cup$ é a união disjunta entre dois conjuntos.

Uma partição com a discrepância mínima é chamada de **partição ótima**.

Quando a soma dos elementos do conjunto é par, o menor valor possível para a discrepância é 0, e quando a soma é ímpar o menor valor possível é 1. Porém, isto não significa que um conjunto qualquer admite uma partição com a discrepância mínima igual a 0 ou 1 (basta pensar num conjunto com um elemento só). No entanto, se um conjunto admite tal partição, falamos que esta é a **partição perfeita**. Portanto, se um algoritmo encontrar uma partição cuja discrepância é 0 ou 1, podemos pará-lo.

Uma forma ilustrativa de apresentar o problema de particionamento é imaginar a balança de dois pratos e alguns pesos (Figura 1). A tarefa é distribuir todos os pesinhos entre os dois pratos de tal maneira que a balança fique mais equilibrada possível.

Figura 1: Balança de dois pratos.



Fonte: `http://etc.usf.edu/clipart/50100/50142/50142_add_scale.htm`

Um exemplo da aplicação do problema de particionamento é o **processamento paralelo**: dadas duas máquinas idênticas, uma lista de tarefas e o tempo que a máquina leva para executar cada tarefa, precisamos distribuir as tarefas entre as máquinas de tal modo que o trabalho seja feito o mais rápido possível. A solução deste problema é bem importante porque reduz o tempo de execução das tarefas e economizar tempo significa aumentar a produtividade.

O problema de particionamento é um clássico problema NP-difícil. Por definição, um problema é NP se é solucionável em tempo polinomial por uma máquina de Turing não determinista. Um problema é NP-difícil se um algoritmo para resolvê-lo pode ser traduzido em um para resolver qualquer outro problema NP. NP-difícil significa, portanto, "pelo menos tão difícil quanto qualquer outro problema NP", embora, na verdade, possa ser mais difícil. Como para o problema de particionamento há várias heurísticas que acham a solução, seja otimamente ou aproximadamente, ele tem sido chamado de *problema NP-difícil mais fácil*.

2.2 Busca por força bruta

Um dos algoritmos mais simples e intuitivos que existe é a **busca por força bruta**. É uma técnica de solução de problemas muito geral que no caso de particionamento de números consiste em computar todas as possíveis somas dos subconjuntos da lista a ser particionada e retornar o subconjunto cuja soma é mais próxima da metade da soma de todos os elementos da lista.

O algoritmo de força-bruta vai sempre achar a solução do problema e a implementação dele é simples. Porém o custo é proporcional ao número de candidatos à solução que é 2^n onde n é o número dos elementos na lista, pois existem 2^n subconjuntos de um conjunto de n elementos.

Exemplo 2.1. *Seja* $S = \{4, 5, 6, 7, 8\}$ *a lista de números a ser particionada. O algoritmo de força-bruta vai primeiro achar todos os subconjuntos de* S*, que são* $2^5 = 32$ *no total e depois computar a soma desses subconjuntos. No nosso exemplo, estamos procurando um subconjunto cuja soma é mais próxima de* $15 = \frac{4+5+6+7+8}{2}$*. Um dos subconjuntos cuja soma é exatamente* 15 *é* $S_1 = \{7, 8\}$ *e o complementar dele* $S_2 = \{4, 5, 6\}$*. Portanto* $\Delta(S_1, S_2) = 0$*. Este exemplo admite a partição perfeita.*

2.3 Algoritmo ganancioso

Outra abordagem do problema é o **algoritmo ganancioso** (ou guloso) que primeiro classifica os números em ordem decrescente e coloca o número maior arbitrariamente num dos dois subconjuntos. Depois itera sobre os elementos restantes e coloca cada um no subconjunto com a menor soma até que todos os números sejam distribuídos.

Essa heurística retorna um resultado bem mais rápido que o algoritmo de força-bruta, porém não garante que ele seja a melhor partição possível. Basta analisar a mesma lista S do Exemplo 2.1.

Exemplo 2.2. *Depois de ordenar a lista* S *obtemos* $\{8, 7, 6, 5, 4\}$*. Colocamos* 8 *em subconjunto* S_1*, depois* 7 *em* S_2 *já que ele está vazio. Em seguida designamos* 6 *a* S_2 *pois* $8 > 7$*. Depois colocamos* 5 *em* S_1 *pois* $8 < 13 = 7 + 6$ *e no final, como a soma dos elementos de* S_1 *e* S_2 *é igual, podemos designar o número* 4 *a qualquer subconjunto, que seja* $4 \in S_1$*. Obtemos a seguinte partição:* $S_1 = \{8, 5, 4\}$ *e* $S_2 = \{7, 6\}$ *com as somas* 17 *e* 13 *respectivamente, ou seja,* $\Delta(S_1, S_2) = 4$*. Mas essa não é a melhor partição possível, pois sabemos que este conjunto admite a partição perfeita* $\{7, 8\} \cup \{4, 5, 6\}$*.*

2.4 Heurística de Karmarkar-Karp

A melhor heurística conhecida atualmente para o problema de particionamento é a **heurística de Karmarkar-Karp (KK)** introduzida em [2], chamada também de *método de diferenciação.* O procedimento é o seguinte: comece por reordanar a lista a ser particionada em ordem decrescente, pegue os dois maiores elementos x_i e x_j da lista e substitua-os pelo elemento $|x_i - x_j|$. Esta ação representa a decisão de que os elementos serão colocados em subconjuntos diferentes. Depois de repetir a operação $n-1$ vezes, a partição será feita e a discrepância dela será igual ao valor do único elemento que ficou na lista.

Como a maioria dos algoritmos começa por classificar os números da lista em ordem descrescente, a partir daqui assumiremos que a lista já vem ordenada. Isto é, cada vez que escrevermos $S = \{x_1, x_2, ..., x_n\}$ assumiremos que $x_1 \geqslant x_2 \geqslant ... \geqslant x_n$.

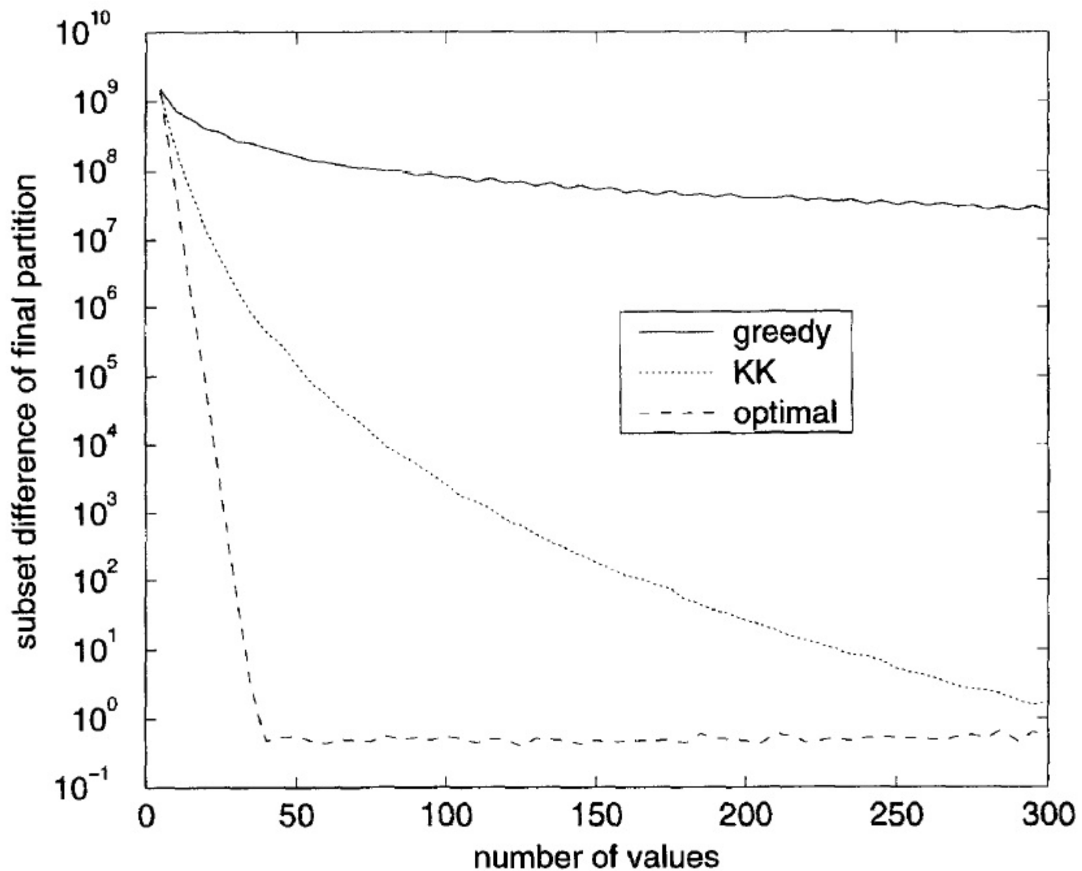
Exemplo 2.3. *Seja* $S = \{8, 7, 6, 5, 4\}$ *o conjunto de números a ser particionado. Seguindo o método de diferenciação, o algoritmo vai dar os seguintes passos:* $\{8, 7, 6, 5, 4\}$, $\{6, 5, 4, 1\}$, $\{4, 1, 1\}$, $\{3, 1\}$, $\{2\}$ *obtendo a partição* $\{8, 6\} \cup \{7, 5, 4\}$ *com a discrepância* $\Delta(S_1, S_2) = 2$. *Ou seja, a heurística KK também não encontra a melhor solução nesse caso. Porém o resultado é melhor do que o do algoritmo ganancioso.*

A vantagem do algoritmo KK sobre o algoritmo ganancioso foi discutida e demonstrada em [4] o que apresenta a Figura 2. Os conjuntos a serem particionados consistem em números inteiros aleatórios distribuidos uniformemente de 0 até 10^{13}. O eixo horizontal representa o tamanho do conjunto particionado e o eixo vertical, o valor da discrepância entre as partições encontradas por cada algoritmo. Cada ponto nas duas linhas superiores representa a média de discrepância de 1000 instâncias do problema, enquanto os pontos na linha inferior são a média de 100 instâncias.

À medida que o tamanho dos conjuntos aumenta, a discrepância encontrada pela heurística KK torna-se substancialmente melhor que a discrepância encontrada pela heurríscica gananciosa. O gráfico mostra também que junto com o crescimento do tamanho dos conjuntos, cresce também a probabilidade de existência da partição perfeita (para $n \geqslant 40$ a probabilidade é muito grande) e solução encontrada pela heurística KK se aproxima da solução ótima.

A diferença entre as soluções encontradas por essas duas heurísticas pode ser explicada pelo fato de que a discrepância entre as partições é do tamanho do último elemento a ser designado. No caso do algoritmo ganancioso, esse elemento é o menor número da lista, enquanto no algoritmo KK, o tamanho dos números que restam é reduzido dramaticamente pelas repetidas operações de diferenciação. Com isso, conforme o aumento do

Figura 2: Desempenho dos algoritmos KK e ganancioso.



Fonte: [4, Korf, *A complete anytime algorithm for number partitioning*]

tamanho dos conjuntos a serem particionados, o último elemento a ser designado tende ser bem menor do que o menor elemento do conjunto original.

O tempo de execução dos ambos algoritmos é polinomial, mas já vimos que o resultado que eles retornam é somente aproximado. Ainda em [4], foi apresentada a maneira de como estender essas heurísticas para algoritmos completos (chamados de *complete anytime algorithms*) que vão melhorando a solução conforme mais tempo ficarem rodando e que garantem que em um certo momento a solução encontrada será ótima. Na seção seguinte apresentaremos a extensão da heurística KK. A extensão do algoritmo ganancioso não será abordada, já que os resultados que ela consegue obter não superam na média os resultados da extensão do algoritmo KK.

Antes de continuarmos, para facilitar o entendimento, adotaremos a seguinte notação: Dada uma partição, definimos o subconjunto que contém o maior elemento como o subconjunto dos positivos, e o outro como o dos negativos. A partir daí a partição pode ser representada como uma sequência de símbolos $+$ e $-$ onde um $+$ $(-)$ na i-ésima posição

nos diz que o i-ésimo elemento da lista pertence ao subconjunto dos positivos (negativos). Em particular, o primeiro símbolo da sequêcia é sempre $+$.

Exemplo 2.4. *Seja* $S = \{8, 7, 6, 5, 4\}$ *o conjunto de números a ser particionado. Então a partição* $\{8, 7\} \cup \{6, 5, 4\}$*, seguindo a notação adotada, pode ser representada por* $++---$*. Podemos interpretar esta notação também como as diretrizes para calcular a discrepância:*

$$\Delta(++---) = |+8+7-6-5-4| = 0$$

Agora vamos focar um pouco nos casos pequenos do problema, até 5 números a serem particionados. Em [1] foi demonstrado que num conjunto S com 4 elementos ou menos existe uma partição com a discrepância mínima tal que os dois maiores elementos de S não pertencem ao mesmo subconjunto. Para um conjunto com 3 números, a partição ótima é óbvia: $+--$. Para um conjunto com 4 números, foi mostrado que a solução ótima é garantida por uma das partições: $+---$ ou $+--+$. No mesmo trabalho foi provado ainda que para um conjunto com 5 elementos a solução ótima é encontrada pelo algorimo KK ou ela tem a forma $++---$.

Todas essas informações serão utilizadas na seção a seguir.

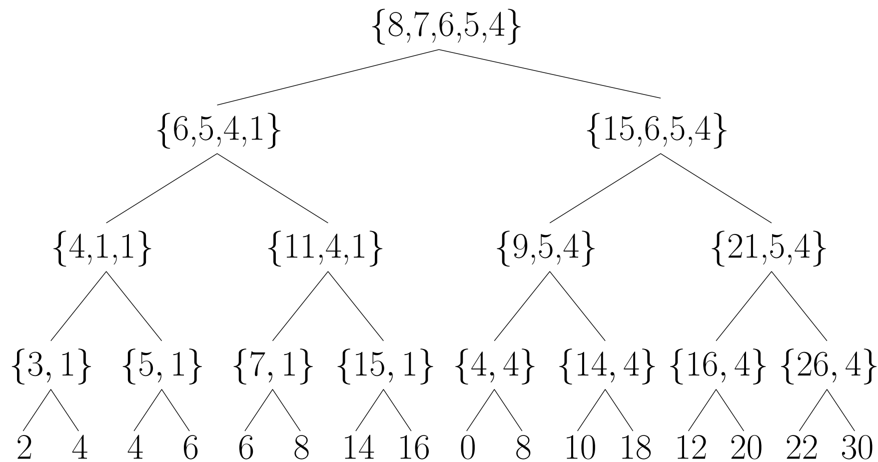
2.5 Algoritmo de Karmarkar-Karp Completo

A extensão da heurística KK é chamada de **algoritmo de Karmarkar-Karp Completo (KKC)**. A base do algoritmo KKC é a construção de uma árvore de busca onde o primeiro nodo é a lista de n números a ser particionada. Sairão dele dois ramos, ramo esquerdo com a lista onde os dois maiores elementos foram substituídos pela sua diferença, e ramo direito com a lista onde os dois maiores elementos foram substituídos pela sua soma (representando que os elementos foram colocados no mesmo subconjunto). Repetiremos a operação para cada nodo e no final teremos 2^{n-1} folhas com os possíveis valores para a discrepância. Enfim, o problema de particionamento se torna um problema de busca em uma árvore binária.

A Figura 3 apresenta a árvore de busca para o problema de particionamento do conjunto $\{8, 7, 6, 5, 4\}$ seguindo as instruções do algoritmo KKC.

Um aspecto muito importante da busca em uma árvore é encontrar os critérios para podar os ramos. Na seção anterior mencionamos que o conjunto com 3 elementos tem a partição ótima da forma $+--$. Sendo assim, ao encontrarmos um nodo com 3 números, já sabemos o valor da folha com solução ótima. Foi dito também que ao particionar um conjunto com 4 números, a solução ótima é garantida pela diferenciação. Portanto, não é preciso analisar o ramo direito saindo de um nodo com 4 números. No caso de conjunto

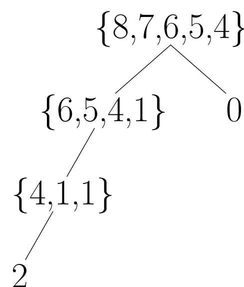
Figura 3: Árvore para particionar o conjunto $\{8,7,6,5,4\}$.



com 5 números, se tomarmos o ramo que vai para a direita, a solução ótima vai ter forma $++---$.

Um outro critério útil é verificar se o maior elemento do conjunto é maior do que a soma dos elementos restantes, pois sendo assim, a solução ótima tem forma $+--...-$.

Figura 4: Árvore para particionar o conjunto $\{8,7,6,5,4\}$ com os ramos podados.



A Figura 4 apresenta a árvore de busca podada para o problema de particionamento do conjunto $\{8,7,6,5,4\}$.

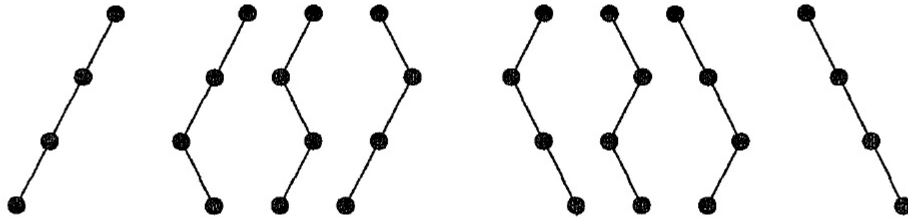
Em geral, quando não existe a solução perfeita, temos que construir a árvore inteira para achar a solução ótima. Por isso, nesse caso, a ordem em que percorremos as folhas não importa. Porém, se existir uma partição perfeita, a ordem de busca torna-se importante, pois quanto mais cedo a encontrarmos, mais cedo poderemos terminar.

Em [3] estão comparadas duas das melhores abordagens para a busca de uma árvore gerada pelo algoritmo KKC.

A primeira abordagem é bem útil quando precisamos percorrer a árvore inteira, ou seja, nos casos quando a probabilidade de existir uma partição perfeira é pequena. Ela é

chamada de *busca em profundidade* (**DFS**, Depth-First Search em inglês) e percorre as folhas da árvore da esquerda para direita.

Figura 5: Caminhos com os desvios 0, 1, 2 e 3.



Fonte: [3, Korf, *Improved Limited Discrepancy Search*]

A outra abordagem é chamada de *busca com discrepância limitada aprimorada* (**ILDS**, Improved Limited Discrepancy Search em inglês). A discrepância nesse caso é um ramo direito da árvore binária. Para evitar confusão, usaremos o termo *desvio* para denotar a quantidade de opções pelo ramo direito. O primeiro caminho gerado pelo ILDS é o mais à esquerda. Em seguida, são gerados aqueles caminhos que contém um ramo direito (do início até a folha). Depois o ILDS gera os caminhos com dois ramos direitos, ect. Assim, o último caminho sendo gerado é aquele mais à direita. A Figura 5 apresenta os conjuntos de caminhos com os desvios 0, 1, 2 e 3 de uma árvore binária de profundiade três.

Essa busca usa fortemente a heurística KK e se mostra eficiente quando a probabilidade de existir uma partição perfeita é grande. Nestes casos, o ILDS encontra as partições perfeitas mais rápido, em média, do que o DFS, e portanto, pode terminar a busca antes.

3 Problema de particionamento de posets

Neste capítulo focaremos na introdução das definições e conceitos relacionados ao problema de particionamento de conjuntos parcialmente ordenados (posets) e na apresentação do algoritmo de Karmarkar-Karp Generalizado Completo para resolvé-lo.

Vale a pena destacar que este problema surgiu no contexto de códigos corretores de erros, ao estudar raio de empacotamento de códigos poset.

Na primeira seção introduzimos os conceitos básicos de conjuntos parcialmente ordenados.

Na segunda mostramos que o problema de particionamento de posets é de fato uma generalização do problema de particionamento clássico.

Na terceira seção fazemos a apresentação do algoritmo de Karmarkar-Karp Generalizado que é uma extensão da heurística de Karmarkar-Karp, e do algoritmo Karmarkar-Karp Generalizado Completo que é uma extensão do algoritmo Karmarkar-Karp Completo.

3.1 Conjuntos parcialmente ordenados

Definição 3.1. *Um* ***conjunto parcialmente ordenado****, também chamado de* ***poset*** *(em inglês,* ***p****artially* ***o****rdered* ***set****), é uma dupla $P = (X, \preceq)$ onde X é um conjunto e $\preceq$ é uma operação binária em X, chamada de* ***ordem parcial****, que satisfaz as seguintes propriedades para todo $x, y, z \in X$:*

1. *$x \preceq x$ (reflexividade).*
2. *Se $x \preceq y$ e $y \preceq x$, então $x = y$ (anti-simetria).*
3. *Se $x \preceq y$ e $y \preceq z$, então $x \preceq z$ (transitividade).*

Na prática, é normal identificar o poset P com o conjunto X. Outro hábito bem comum é dizer que *x é menor que ou igual a y* se $x \preceq y$. Imitando o comportamento do símbolo $\leqslant$,

entende-se $x \succeq y$ como $y \preceq x$, $x \prec y$ como $x \preceq y$ e $x \neq y$, e $x \succ y$ como $y \prec x$. Falaremos de um poset de tamanho n se o conjunto sobre qual está definida a ordem parcial tiver cardinalidade n.

Exemplo 3.1. *Um caso especial de uma ordem parcial é uma ordem total, isto é, em que cada dois elementos do conjunto em questão são comparáveis. Em particular, os números reais, ordenados pela relação padrão $\leqslant$, formam um conjunto parcialmente ordenado (que é também totalmente ordenado).*

Exemplo 3.2. *Um conjunto de todos os subconjuntos de um conjunto dado (seu conjunto das partes) com a relação de inclusão como ordem formam um poset.*

Exemplo 3.3. *O conjunto de numeros naturais com a relação de divisibilidade é um outro exemplo de um poset. Aqui não podemos falar da ordem total, porque, por exemplo, não existe a relação entre os números 4 e 10, já que nenhum é divisor do outro.*

Como mostra o Exemplo 3.3, nem todos os elementos de um poset estão em relação, i.e. podem existir dois elementos $x, y \in P$ para quais não vale $x \preceq y$ nem $y \preceq x$. Neste caso dizemos que x e y são *incomparáveis*, caso contrário, se existir uma relação entre x e y dizemos que estes são *comparáveis*. Com isso, não podemos falar de um elemento máximo ou mínimo de um poset. Porém, existe um conceito de elementos maximais e minimais.

Definição 3.2. *Um elemento $a \in P$ é dito **elemento minimal** se $x \in P$ tal que $x \preceq a$, então $x = a$. Em outras palavras, $a \in P$ é um elemento minimal se nenhum $x \in P$ o precede estritamente.*
*Um elemento $a \in P$ é dito **elemento maximal** se $x \in P$ tal que $x \succeq a$, então $x = a$. Em outras palavras, $a \in P$ é um elemento minimal se nenhum $x \in P$ o sucede estritamente.*

No nosso trabalho, estaremos mais interessados nos elementos maximais de um poset.

Definição 3.3. *Sejam P um poset e $x, y \in P$, então dizemos que y **cobre** x se e somente se $x \prec y$ e não existe outro $z \in P$ tal que $x \prec z \prec y$.*

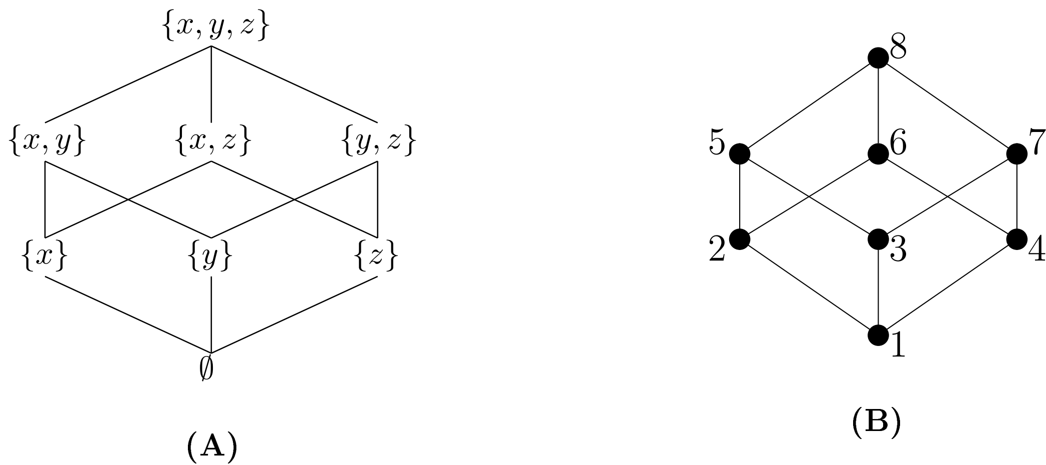
Exemplo 3.4. *Considere o conjunto dos números naturais com a ordem definida pela relação $\leqslant$, então o elemento que cobre um número n é o sucessor $n + 1$.*

Neste texto, trabalharemos somente com os posets finitos. Portanto, sem perda de generalidade, daqui por diante, vamos sempre considerar um poset finito sobre o conjunto $\{1, 2, ..., n\}$.

Exemplo 3.5. *Seja $P = \{a, b, c\}$ um poset com a ordem $a \preceq c$, então se substituirmos a por 1, b por 2 e c por 3, mantendo a ordem dada, obtemos um poset sobre o conjunto $\{1, 2, 3\}$ que é isomorfo a P.*

O **diagrama de Hasse** é uma representação gráfica de um poset. Nesse gráfico, os vértices representam elementos de P e dois elementos $x, y \in P$ são ligados por uma aresta se, e somente se, y cobre x. Os elementos menores são desenhados abaixo dos elementos maiores. Veja os diagramas de Hasse de um poset na Figura 6.

Figura 6: Diagrama de Hasse do poset do Exemplo 3.2 para o conjunto $S = \{x, y, z\}$. Neste caso $P = (\mathcal{P}(S), \subseteq)$, onde $\mathcal{P}(S) = \big\{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\big\}$, veja (A). A figura (B) apresenta um poset isomorfo a P definido sobre o conjunto $\{1, 2, ..., 8\}$ depois de enumerar os elementos de P.



Definição 3.4. *Seja P um poset de tamanho n. A* ***matriz de adjacência*** *de P é uma matriz quadrada $A_{n\times n} = (a_{ij})$ onde*

$$a_{ij} = \begin{cases} 1 & \text{se } i \preceq j, \\ 0 & \text{caso contrário.} \end{cases}$$

Dado um diagrama de Hasse de um poset, se enumerarmos os elementos da esquerda para a direita e de baixo para cima, a matriz de adjacência do poset terá a forma de uma matriz triangular superior.

Exemplo 3.6. *A matriz de adjacência do poset apresentado na Figura 6(B) tem a seguinte forma*

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Definição 3.5. *Seja P um poset. Um **ideal** em P é um subconjunto $I \subseteq P$ tal que se $y \in I$ e $x \preceq y$, então $x \in I$.*

Definição 3.6. *Dado um subconjunto qualquer A de um poset P, definimos o **ideal gerado pelo conjunto** como sendo o menor ideal $I \subseteq P$ que o contém e o denotamos por $\langle A \rangle$.*

Assim como falamos dos elementos maximais de um poset, um subconjunto de um poset também possui tais elementos. Denotaremos o conjunto dos elementos maximais de um conjunto A por M_A.

3.2 Surgimento do algoritmo KKGC

O algoritmo KKGC surgiu, a priori, para determinar o raio de empacotamento de um código poset em [1]. Porém, no mesmo trabalho (Capítulo 3) foi demonstrado que para resolver este problema, precisamos antes encontrar uma partição ótima de um poset. Portanto, vamos direto ao ponto, omitindo o problema original. Para interessados em achar o raio de empacotamento de um código poset, recomendamos a leitura do trabalho [1].

Antes de seguirmos, precisamos definir mais um conceito.

Definição 3.7. *Sejam P um poset e $A \subseteq P$. Definimos o P-**peso** de A por*

$$\omega_P(A) = |\langle A \rangle|,$$

a cardinalidade do ideal gerado por A.

O peso de um conjunto $A \subseteq P$ é determinado pelos seus elementos maximais, pois o ideal gerado pelos dois é igual, ou seja $\omega_P(A) = \omega_P(M_A)$.

3.2.1 Caso dos ideais disjuntos

O resultado a seguir demonstra que o problema de particionamento de posets, cujos ideias gerados por diferentes elementos maximais são disjuntos, é equivalente a resolver um problema de particionamento clássico.

Teorema 3.1. *[1, Teorema 5, p. 42] Sejam P um poset e $M_P = \{x_1, x_2, ..., x_n\}$ o conjunto dos elementos maximais de P tal que $\langle x_i \rangle \cap \langle x_j \rangle = \emptyset$ se $i \neq j$, e seja $w_i := \omega_P(x_i)$. Então, encontrar a partição ótima de P é equivalente a resolver o problema de particionamento de números para $S = \{w_1, w_2, ..., w_n\}$.*

Demonstração. Seja (A, B) uma partição de M_P. Definimos $S_1 = \{w_i | x_i \in A\}$ e $S_2 = \{w_i | x_i \in B\}$. Sendo assim, (S_1, S_2) é uma partição de S. Além disso, como $\langle x_i \rangle \cap \langle x_j \rangle = \emptyset$ se $i \neq j$, temos que

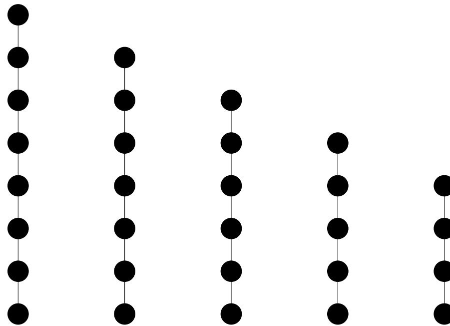
$$\omega_P(A) = \sum_{x \in A} \omega_P(x) = \sum_{x_i \in A} \omega_P(x_i) = \sum_{w_i \in S_1} w_i = \sum_{w \in S_1} w.$$

Analogamente obtemos $\omega_P(B) = \sum_{w \in S_2} w$.
Portanto, encontrar uma partição de (A, B) de M_P que minimize $\max\{\omega_P(A), \omega_P(B)\}$ é equivalente ao encontrar uma partição (S_1, S_2) de S que minimize $\max\{\sum_{w \in S_1} w, \sum_{w \in S_2} w\}$.
□

Vale observar que a condição de que os ideias gerados por diferentes elementos maximais sejam disjuntos equivale a dizer que cada componente conexa do diagrama de Hasse do poset P possui um único elemento maximal.

Um exemplo de transformação de um problema de particionamento clássico em um problema de particionamento de posets está apresentado na Figura 7.

Figura 7: *Diagrama de Hasse para o poset P que podemos obter transformando o problema de particionar a lista $S = \{8, 7, 6, 5, 4\}$ em um problema de particionamento de posetes.*



Aplicando o conceito da discrepância do problema de particionamento para este caso, sendo (A, B) uma partição de elementos maximais de um poset P, foi definida em [1] a **discrepância** entre A e B como

$$\Delta(A, B) = |\omega_P(A) - \omega_P(B)|,$$

e a **discrepância mínima** de P como

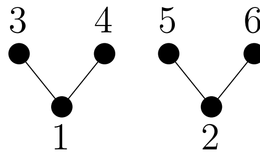
$$\Delta^*(P) = \min_{X \dot{\cup} Y = M_P} \Delta(X, Y),$$

onde (X, Y) é uma partição de M_P.

3.2.2 Caso geral

No caso dos posets genéricos temos que considerar a possibilidade de intersecção entre os ideais gerados por elementos maximais. Portanto, o problema não é mais equivalente a minimizar a discrepância. Podemos perceber isso no exemplo apresentado na Figura 8.

Figura 8: *No poset apresentado abaixo, uma partição com a discrepância igual a zero é* $\{3,5\} \cup \{4,6\}$. *Porém, ela não é ótima, pois o maior dos pesos dos dois subconjuntos é* 4, *enquanto na partição* $\{3,4\} \cup \{5,6\}$, *cuja discrepância também é zero, o maior dos pesos é igual a* 3.



No caso do poset da Figura 8 temos duas partições, $\{3,5\} \cup \{4,6\}$ e $\{3,4\} \cup \{5,6\}$, que têm discrepância zero, portanto mínima. Pensando em processamento paralelo, no primeiro caso, a cada processador estão sendo atribuídas quatro tarefas e no segundo apenas três. Isto nós leva a obrigação de refinarmos o conceito de otimalidade, onde consideramos não apenas a discordância (balanço entre os processadores), mas a necessidade de ambos eventualmente terem de processar a mesma tarefa. Para isto, introduzimos o conceito de *discordância*.

Definição 3.8. *Seja* P *um poset e* (A, B) *uma partição de* M_P, *o conjunto dos elementos maximais. Definimos a* ***discordância*** *entre* A *e* B *como*

$$\Lambda(A, B) = \Delta(A, B) + |\,\langle A \rangle \cap \langle B \rangle\,|,$$

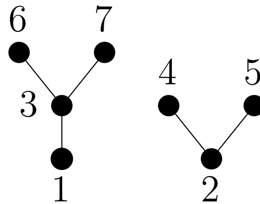
e a ***discordância mínima*** *de* P *como*

$$\Lambda^*(P) = \min_{X \cup Y = M_P} \Lambda(X, Y).$$

Outro exemplo de poset que mostra que a discepância mínima não define a partição ótima, está apresentada na Figura 9.

No caso geral, de modo análogo ao da discrepância, para determinar uma partição ótima temos que achar a discordância mínima. Podemos fazer isto usando o algoritmo desenvolvido no trabalho [1] que é uma extenção de KKC para posets genéricos. Chamaremos ele de **algoritmo de Karmarkar-Karp Generalizado Completo (KKGC)**. O procedimento dele é apresentado na seção seguinte.

Figura 9: *Neste poset, a partição* $\{6,4\} \cup \{7,5\}$ *tem discrepância igual a* 0 *e peso máximo igual a* 5*. Porém, a partição ótima* $\{6,7\} \cup \{4,5\}$ *tem peso máximo igual a* 4 *e a discrepância igual a* 1.



3.3 Algoritmo de Karmarkar-Karp Generalizado Completo

Devemos observar aqui que no trabalo [1], este algoritmo é chamado somente de algoritmo de Karmarkar-Karp Generalizado. Neste trabalho, também usaremos o nome de algoritmo de Karmarkar-Karp Generalizado, porém vai ter outro significado para nós. Explicaremos o motivo e a diferença entre os dois algoritmos no final desta seção.

Antes de apresentarmos o algoritmo, precisamos introduzir alguns conceitos definidos no trabalho [1].

Definição 3.9. *Seja* $P = (X, \preceq)$ *um poset, onde* $X = \{1, 2, ..., n\}$*, e seja* $x \in P$*. Então o **vetor de adjacência** de* x *é*

$$\hat{x} = \begin{pmatrix} \hat{x}^{(1)} \\ \hat{x}^{(2)} \\ \dots \\ \hat{x}^{(n)} \end{pmatrix}, \quad onde \quad \hat{x}^{(i)} = \begin{cases} 1 & se\ i \preceq x \\ 0 & caso\ contrário \end{cases}.$$

Se A_P é uma matriz de adjacência de um poset P de tamanho n, então as colunas de A_P são formadas pelos vetores de adjacência, isto é

$$A_P = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \hat{1} & \hat{2} & \dots & \hat{n} \\ \vdots & \vdots & & \vdots \end{pmatrix}.$$

A seguir, daremos uma definição dos operadores de diferenciação e associação que, no algoritmo KKGC que iremos apresentar em breve, vão agir nos vetores de adjacência dos elementos maximais de um poset a ser particionado. Esta é a relação entre os algoritmos

KKC e KKGC. No algoritmo KKC, construímos os nodos da árvore substituindo os dois elementos da lista, subtraindo um do outro ou os somando. O algoritmo KKGC vai proceder de forma análoga, substituindo os vetores de adjacência dos elementos maximais. Se usarmos o operador de diferenciação, esta ação significará colocar os elementos maximais associados aos vetores que substituímos nos conjuntos diferentes. Se agirmos com o operador de associação, aqueles elementos maximais serão colocados no mesmo conjunto.

Definição 3.10. *Seja $X = \{0, 1, -1, i\}$. Os* ***operadores de diferenciação e associação*** *são definidos pelas seguintes tabelas:*

$\ominus$	*0*	*1*	*-1*	*i*
0	*0*	*-1*	*1*	*i*
1	*1*	*i*	*1*	*i*
-1	*-1*	*-1*	*i*	*i*
i	*i*	*i*	*i*	*i*

$\oplus$	*0*	*1*	*-1*	*i*
0	*0*	*1*	*-1*	*i*
1	*1*	*1*	*i*	*i*
-1	*-1*	*i*	*-1*	*i*
i	*i*	*i*	*i*	*i*

O valor de $x \ominus y$ encontra-se na linha correspondente a x e na coluna correspondente a y. É o mesmo para o caso da associação.
No caso de dois vetores $\hat{x}, \hat{y} \in X^n$, $n \in \mathbb{N}$, os operadores são definidos coordenada por coordenada, isto é, para $i = 1, 2, ..., n$

$$(\hat{x} \ominus \hat{y})^{(i)} = \hat{x}^{(i)} \ominus \hat{y}^{(i)} \qquad e \qquad (\hat{x} \oplus \hat{y})^{(i)} = \hat{x}^{(i)} \oplus \hat{y}^{(i)}.$$

Vale constatar que os símbolos $0, 1, -1, i$ são puramente formais, porém o signicativo deles será obtido mais adiante.

Exemplo 3.7. *Operações de diferenciação e de associação agindo nos vetores com os elementos do conjunto X:*

$$\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \ominus \begin{pmatrix} i \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \ominus i \\ 0 \ominus 1 \\ -1 \ominus -1 \end{pmatrix} = \begin{pmatrix} i \\ -1 \\ i \end{pmatrix}$$

As seguintes propriedades das operação de diferenciação e associação seguem diretamente da definição, para quaisquer $x, y, z \in \{0, 1, -1, i\}$ temos:

1. $x \oplus y = y \oplus x$,
2. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$,
3. $0 \oplus x = x \oplus 0 = x$,
4. $\oplus(x \oplus y) = \oplus x \oplus y$,
5. $\oplus(x \ominus y) = \oplus x \ominus y$,

6. $\ominus(x \oplus y) = \ominus x \ominus y$,
7. $\ominus(x \ominus y) = \ominus x \oplus y$.

Como os operadores agem coordenada por coordenada, as propriedades listadas acima valem também no caso vetorial.

Na propriedade (3) podemos observar que o símbolo 0 é tratado como zero nos números inteiros, isto é, é um elemento néutro de associação.

Definição 3.11. *Seja P um poset e seja $M_P = \{x_1, x_2, ..., x_m\}$ o conjunto dos elementos maximais de P. Então a* ***lista de vetores associada*** *a M_P é $(\hat{x}_1, \hat{x}_2, ..., \hat{x}_m)$.*

Definição 3.12. *Seja $(\hat{x}_1, \hat{x}_2, ..., \hat{x}_m)$ a lista de vetores associada aos elementos maximais de um poset. Uma* ***expressão simples****, envolvendo os elementos da lista e as operações $\oplus$ e $\ominus$, é uma sequência $\hat{v}$ da forma*

$$\hat{v} = *_1 \ \hat{x}_{k_1} \ \ *_2 \ \ \hat{x}_{k_2} \ ... \ \ *_l \ \ \hat{x}_{k_l},$$

onde $k_i \in \{1, 2, ..., m\}$ e $*_i = \oplus$ ou $*_i = \ominus$.

A *partição associada a uma expressão simples* é a partição (A, B) definida da seguinte forma:

$$A = \{x_{k_i} \mid *_i = \oplus\} \qquad \text{e} \qquad B = \{x_{k_i} \mid *_i = \ominus\}.$$

O conjunto A é chamado de *conjunto primário* e o B de *conjunto secundário.*

A *partição associada a uma expressão qualquer* é a partição associada a expressão simples, obtida transformando a expressão original, utilizando as propridades dos operadores.

Como depois de calcular uma expressão obtemos um vetor, por simplicidade, usaremos a notação de um vetor para uma expressão. Denotaremos o conjunto primário de uma expressão $\hat{v}$ por $Primario(\hat{v})$ e o conjunto secundário por $Secundario(\hat{v})$.

Exemplo 3.8. *Seja P um poset e $M_P = \{x_1, x_2, x_3, x_4\}$ o conjunto dos elementos maximais de P. A lista de vetores associada a M_P é $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4)$. Seja $\hat{v} = (\hat{x}_1 \ominus \hat{x}_2) \ominus (\hat{x}_3 \ominus \hat{x}_4)$ uma expressão, então a forma simples de $\hat{v}$ é*

$$\oplus \ \hat{x}_1 \ \ominus \ \hat{x}_2 \ \ominus \ \hat{x}_3 \ \oplus \ \hat{x}_4$$

e a partição associada a expressão $\hat{v}$ é $\big(\{x_1, x_4\}, \{x_2, x_3\}\big)$, onde $Primario(\hat{v}) = \{x_1, x_4\}$ e $Secundario(\hat{v}) = \{x_2, x_3\}$.

Definição 3.13. *Seja $\hat{v} \in \{0, 1, -1, i\}^n$. Definimos a* ***função soma*** *das entradas do vetor $\hat{v}$ como*

$$S(\hat{v}) = \sum_{k=1}^{n} \hat{v}^{(k)}.$$

Exemplo 3.9. *Se* $\hat{v} = (1, 1, -1, i, 1, i)$*, então* $S(\hat{v}) = 2 + 2i$*.*

Mesmo que as operações $\ominus$ e $\oplus$ tenham sido definidas apenas formalmente, podemos considerar os elementos $0, 1, -1, i$ como números complexos. Deste modo a soma $S(\hat{v})$ torna-se um número complexo e faz sentido falar das partes real e imaginária de $S(\hat{v})$ que denotaremos por $\Re\big(S(\hat{v})\big)$ e $\Im\big(S(\hat{v})\big)$ respectivamente.

O seguinte teorema é um resultado bem importante obtido em [1, Teorema 8, p. 54], pois mostra a relação entre a discrepância e discordância e as partes real e imaginária da função soma definida acima.

Teorema 3.2. *Sejam* P *um poset de tamanho* n*,* $M_P = \{x_1, x_2, ..., x_m\}$ *o conjunto dos elementos maximais de* P*,* $(\hat{x}_1, \hat{x}_2, ..., \hat{x}_m)$ *a lista de vetores associada a* M_P *e* $\hat{v}$ *uma expressão utilizando todos os vetores da lista associada uma única vez. Seja* (A, B) *a partição de* M_P *associada a expressão* $\hat{v}$*. Então*

$$\Delta(A, B) = |\Re\big(S(\hat{v})\big)| \quad e \quad |\langle A\rangle \cap \langle B\rangle| = \Im\big(S(\hat{v})\big)$$

de modo que

$$\Lambda(A, B) = |\Re\big(S(\hat{v})\big)| + \Im\big(S(\hat{v})\big).$$

Definição 3.14. *Sejam* P *um poset de tamanho* n*,* $M_P = \{x_1, x_2, ..., x_m\}$ *o conjunto dos elementos maximais de* P *e* $(\hat{x}_1, \hat{x}_2, ..., \hat{x}_m)$ *a lista de vetores associada a* M_P*. Então a* ***matriz-raio*** *de* P *é*

$$R_P = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \hat{x}_1 & \hat{x}_2 & ... & \hat{x}_m \\ \vdots & \vdots & & \vdots \end{pmatrix},$$

Isto quer dizer que a matriz-raio de um poset P tem como colunas os vetores de adjacência dos elementos maximais de P. Logo, podemos construir a matriz-raio a partir da matriz de adjacência eliminando as colunas que não se referem aos elementos maximais.

Exemplo 3.10. *Seja* P *um poset de tamanho* 7 *com a seguinte matriz de adjacência*

$$A_P = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Em geral, podemos ler a matriz de adjacência por colunas, sendo que cada 1 *na* j*-ésima coluna aponta os elementos menores que (ou iguais a) elemento* j*, ou por linhas, sendo que cada* 1 *na* i*-ésima linha aponta os elementos maiores que (ou iguais a) elemento* i*. Portanto, podemos encontrar os elementos maximais, verificando quais linhas têm somente uma entrada igual a* 1*.*
No nosso exemplo, estas linhas são $5, 6$ *e* 7*. Logo, o conjunto dos elementos maximais de* P *é* $M_P = \{5, 6, 7\}$ *e por isso a matriz-raio de* P *é formada pelas colunas* $5, 6$ *e* 7 *da matriz de adjacência. Assim*

$$R_P = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Agora estamos prontos para apresentar o procedimento do algoritmo de **Karmarkar-Karp Generalizado Completo (KKGC)** para resolver o problema de particionamento de um poset P. Temos os seguintes passos:

1. Construir a matriz de adjacência do poset P.
2. Achar o conjunto dos elementos maximais M_P.
3. Construir a matriz-raio R_P.
4. Construir uma árvore de busca, onde o primeiro nodo é a matriz R_P, da seguinte maneira:
   unimos dois vetores colunas da matriz em cada nodo por ramo: o ramo esquerdo substitui os vetores usando o operador de diferenciação, enquanto o ramo direito substitui-los usando o operador de associação; continuamos criando ramos até obtermos uma matriz coluna (um vetor).
5. Calcular $S(\hat{v})$ para cada folha (nodo terminal) com um vetor $\hat{v}$.
6. Calcular $\Lambda(\hat{v})$ para cada $\hat{v}$ e escolher a folha com a menor discordância.

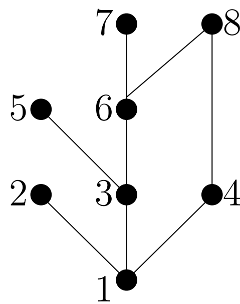
Para escolher os dois vetores da matriz-raio a serem substituídos usamos o critério de PLDM (*Poset Largest Differencing Method* em inglês) que diz o seguinte: o primeiro vetor $\hat{v}$ é aquele que maximiza $\Lambda(\hat{v})$ e o segundo vetor $\hat{w}$ é aquele que minimiza $\Lambda(\hat{v} \ominus \hat{w})$. Este critério foi introduzido em [1].

Enfim, para achar as partições seguimos as instruções abaixo. Seja $\hat{v} = (v_1, v_2, ..., v_n)$ o vetor de uma folha da árvore construída, então para $k = 1, 2, ..., n$ temos:

1. $v_k = 0 \iff k \notin A \cup B$,
2. $v_k = 1 \iff k \in A - B$,
3. $v_k = -1 \iff k \in B - A$,
4. $v_k = i \iff k \in A \cap B$,

sendo (A, B) a partição do poset P, onde $A = Primario(\hat{v})$ e $B = Secundario(\hat{v})$. Para achar a partição ótima basta pegar o vetor $\hat{v}$ da folha com a menor discrepância. Definimos uma *partição perfeita* se a discordância for igual a 0 ou 1.

Exemplo 3.11. *Vamos particionar o poset P com o diagrama de Hasse apresentado abaixo, seguindo os passos do algoritmo KKGC.*



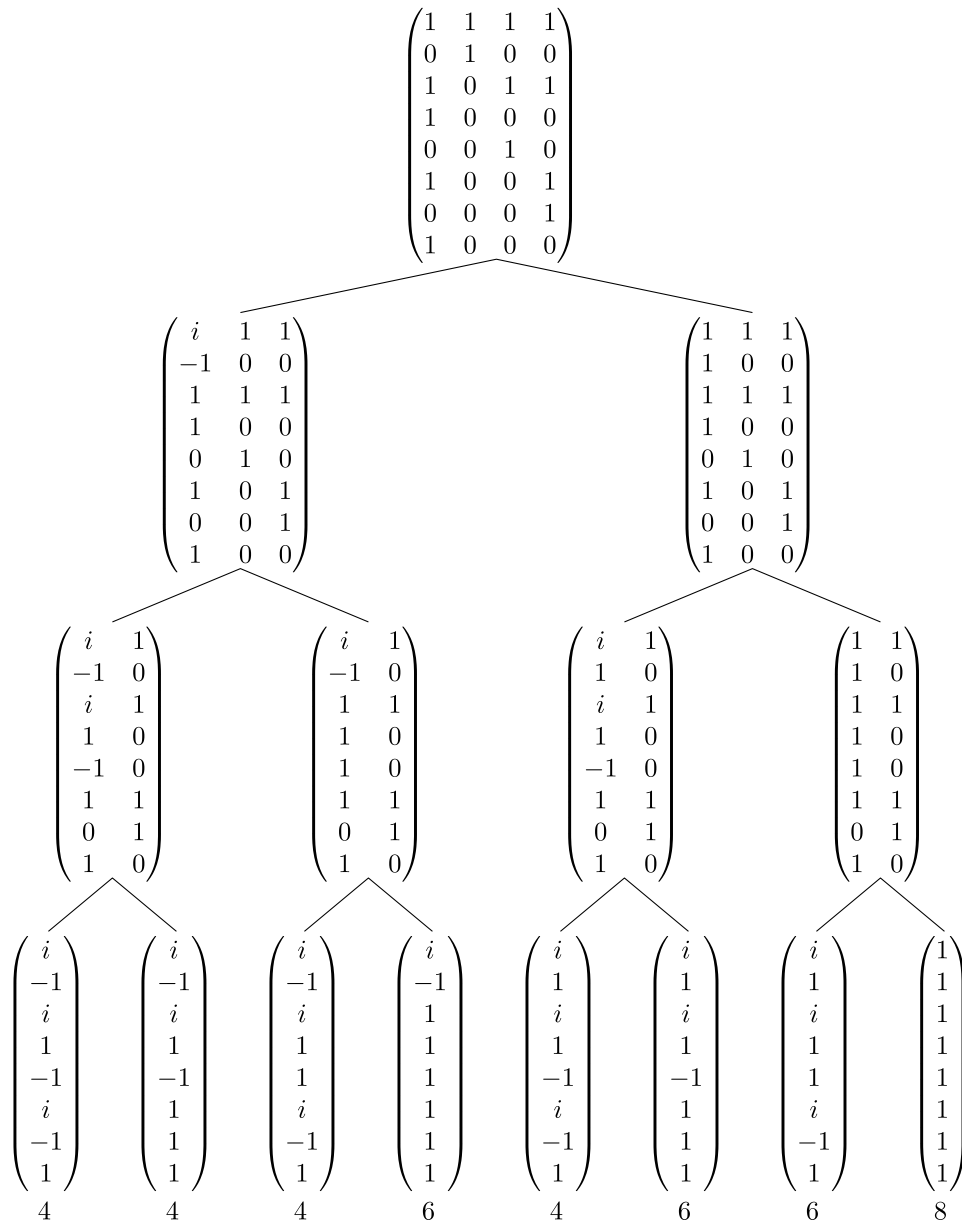
Primeiro precisamos construir a matriz de adjacência do poset P. Ela tem a forma

$$A_P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Os elementos maximais de P são $2, 5, 7, 8$, logo a matriz-raio será formada pelas colunas $2, 5, 7$ e 8 de A_P que são os vetores de adjacência associados aos elementos maximais. A matriz-raio será o primeiro nodo da árvore de busca. Construindo os ramos esquerdos e direitos seguindo as instruções do KKGC e usando o critério PLDM, obtemos a árvore apresentada na Figura 10. Calculando o valor da discordância para cada folha, concluímos que $\Lambda^*(P) = 4$ e uma das partições ótimas é $\big(\{4, 8\}, \{2, 5, 7\}\big)$, sendo $\{4, 8\}$ o conjunto primário e $\{2, 5, 7\}$ o conjunto secundário de $\hat{v} = (i, -1, i, 1, -1, i, -1, 1)$.

Como sugere o nome do algoritmo KKGC, ele é completo, ou seja, garante que a solução encontrada seja ótima. Mas para que isto aconteca temos que construir a árvore

Figura 10: Árvore construida pelo algoritmo KKGC para o poset P do Exemplo 3.11.



inteira (a menos que encontrarmos uma folha com a discordância igual a 0 ou 1 que indicaria a partição perfeita). Porém, dependendo do tamanho do conjunto M_P, o conjunto dos elementos maximais do poset, isto nem sempre será viável, pois o número de todas as folhas possíveis é exponencial ao tamanho de M_P (é igual a 2^{M_P}).

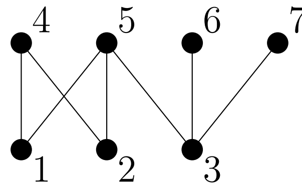
Por isso, achamos interessante dar uma certa importância para o procedimento de encontrar a primeira folha a esquerda. Chamaremos esta parte do KKGC de algoritmo de **Karmarkar-Karp Generalizado (KKG)**, pois pelo fato de usar somente o operador da diferenciação, torna-se análogo ao algoritmo KK do problema de particionamento clássico.

No caso do poset do Exemplo 3.11, o algoritmo KKG encontraria a partição ótima, pois a primeira folha da árvore atinge o valor da discordância mínima. Porém, semelhante ao algoritmo KK, o KKG é somente uma heurística e por isso nem sempre dá o melhor resultado o que acontece no Exemplo 3.12.

Exemplo 3.12. *Seja P um poset com a seguinte matriz de adjacência:*

$$A_P = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Logo, podemos apresentar P com o seguinte diagrama de Hasse



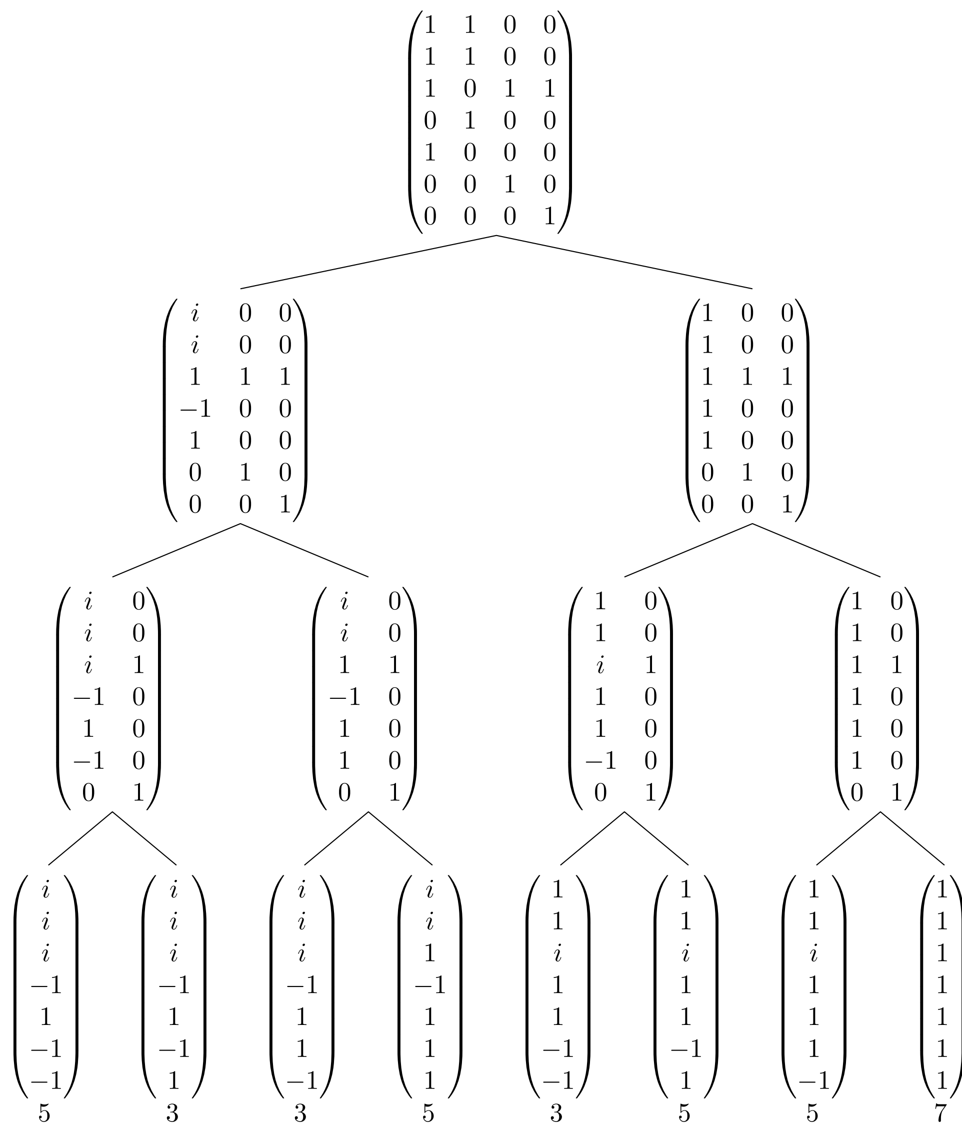
A árvore de busca construida pelo algoritmo KKGC para o poset P encontra-se na Figura 11. Neste caso o KKG não acha a partição ótima, pois $\Lambda^*(P) = 3 < 5$ que é o valor da discordância da primeira folha que representa o algoritmo KKG.

Como vimos no caso do algoritmo KKC, é importante achar os critérios para podar os ramos. No texto [1] foi proposto um critério para fazer a poda, porém, não foi estudado neste trabalho.

Vale mencionar também que na nossa implementação, usamos a DFS (busca em profundidade) para percorrer a árvore, isto é, percorremos as folhas da esquerda à direita.

Antes de seguirmos, para os nossos testes, é importante definir mais um algoritmo decorrente do algoritmo KKGC. Vamos denotar ele por $\mathbf{KKGC}_r$, onde r significará a quantidade das folhas que queremos percorrer antes de retornar uma solução. Portanto, podemos falar que KKGC_r é uma restrição de KKGC. Como um poset com k elementos maximais tem 2^{k-1} folhas, então identificaremos $\text{KKGC}_{2^{k-1}}$ com KKGC e KKGC_1 com KKG.

Figura 11: Árvore de busca do Exemplo 3.12.



4 | Desempenho do algoritmo KKGC

Neste capítulo divulgamos os resultados dos testes realizados com os algoritmos de Karmarkar-Karp Generalizado (KKG) e Karmarkar-Karp Generalizado Completo (KKGC) que medem o desempenho destes do ponto de vista do tempo de execução e da precisão da solução em comparação ao algoritmo de força-bruta e ao algoritmo ganancioso.

Na primeira seção apresentamos os métodos de análise do algoritmo Karmarkar-Karp Completo (KKC) que serviram como a base para os nossos testes, já que o KKGC é a extensão do KKC.

Na segunda seção falamos sobre o maior problema encontrado na implementação do algoritmo, que foi a geração dos posets para podermos realizar os testes.

A terceira seção está inteiramente dedicada a apresentação dos testes feitos e os resultados encontrados. Nessa seção introduzimos o algoritmo de busca por força bruta e o algoritmo ganancioso, que serviram para avaliar a eficiência dos algoritmos KKG e KKGC.

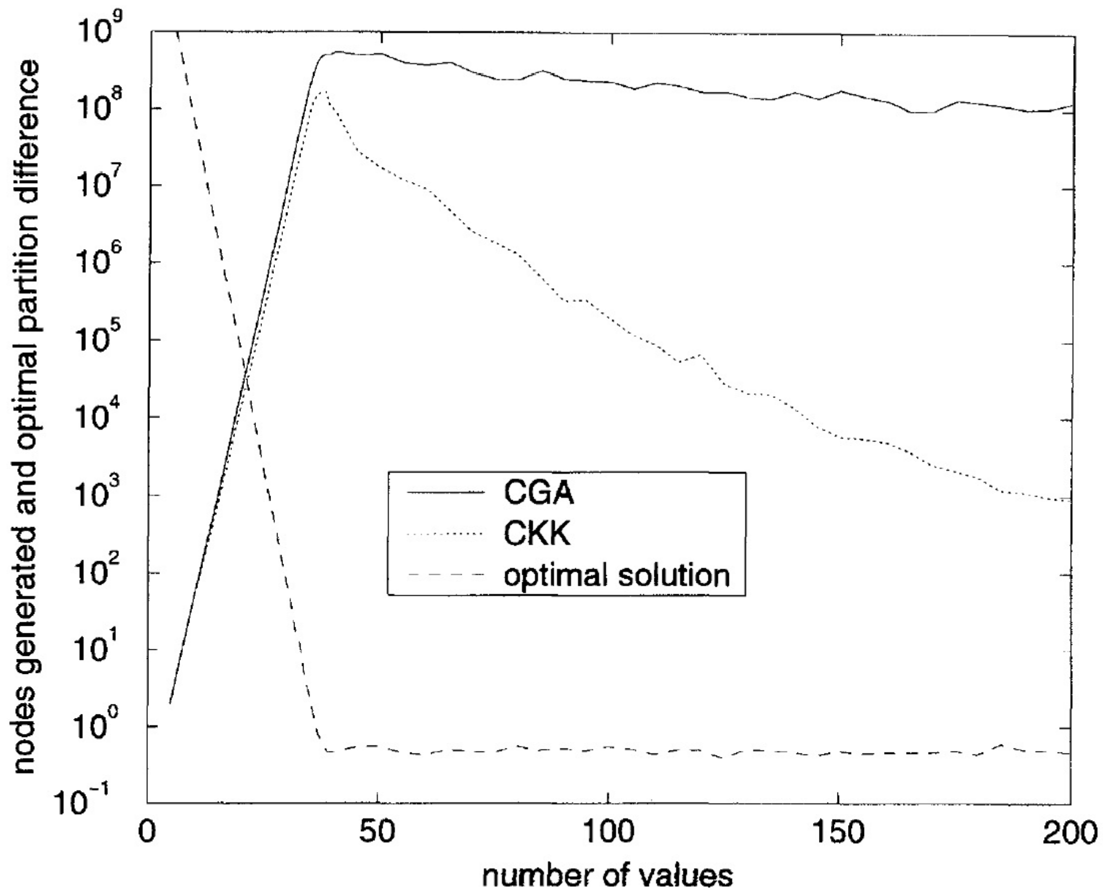
4.1 Métodos de análise

Os métodos de análise que aplicamos ao algoritmo KKGC estão baseados na análise feita pelo Richard E. Korf em [4] sobre o algoritmo KKC, já que o KKGC é uma generalização dele.

O resultado dos testes feitos pelo Korf está apresentado na Figura 12. Os testes foram feitos para o algoritmo de Karmarkar-Karp Completo e um algoritmo que é a extensão da heurística gananciosa, chamado de *Algoritmo Ganancioso Completo* (CGA). O que nós interesa é o comportamento do algoritmo KKC.

Em seus testes, Korf implementou o algoritmo para achar a partição ótima dos conjuntos de números inteiros escolhidos aleatoriamente e distribuidos uniformemente entre 0 e 10^{13}. Cada ponto no gráfico é a média de 100 instâncias arbitrárias do problema. O eixo horizontal mostra a quantidade de números que foram particionados, com pontos de dados para conjuntos de tamanho 5 a 200, com incrmentos de 5 e, com mais detalhes, de

Figura 12: Nodos gerados para particionar de forma ótima os conjuntos de números inteiros com 10 dígitos.



Fonte: [4, Korf, *A complete anytime algorithm for number partitioning*]

30 a 40 com incrementos de 1. O eixo vertical mostra o número de nodos gerados pelos dois algoritmos. A linha tracejada mostra uma outra medida que é a discrepância média da partição ótima, representada no eixo vertical na mesma escala neste caso.

Existem duas regiões diferentes deste gráfico, dependendo de quantos números foram particionados. Com menos de 30 números, não foram encontradas partições perfeitas, enquanto com 40 ou mais inteiros, uma partição perfeita foi encontrada em todos os casos. A discrepância da partição ótima de um conjunto com 40 números ou mais é igual $0,5$ em média, uma vez que a quantidade de partições ótimas com a discrepância 0 ou 1 é aproximadamente igual.

Sem uma partição perfeita, o comportamento do algoritmo KKC é independente da precisão dos números. A melhoria de desempenho é mais dramática quando existe uma partição perfeita. Nesse caso, à medida que o tamanho do problema aumenta, o tempo de execução do KKC cai precipitadamente. Korf rodou o algoritmo KKC em problemas de 10 dígitos ($x_i \leqslant 10^{10}$) até o tamanho 300 ($n \leqslant 300$), onde a solução da heurística KK

é quase sempre ótima, e o número de nodos gerados se aproxima do número de inteiros sendo particionados.

A intuição por trás da observação de que grandes problemas são mais fáceis de resolver do que os de tamanho intermediário é bem simples. Dados n inteiros, o número de diferentes subconjuntos desses inteiros é 2^n. Se os números inteiros variarem de 0 a m, o número de diferentes somas possíveis de subconjuntos será menor que nm, já que nm é a soma máxima possível de subconjuntos. Se m for mantido constante enquanto n for aumentado, o número de subconjuntos diferentes crescerá exponencialmente, enquanto o número de somas de subconjuntos diferentes crescerá apenas linearmente. Portanto, deve haver muitos subconjuntos com a mesma soma de subconjunto. Em particular, a frequência de partições perfeitas aumenta com o aumento de n, tornando-as mais fáceis de encontrar.

A princípio, antes de ter começado os nossos testes com o algoritmo KKGC, a ideia era repetir os testes do Korf adaptando-os para o caso de posets. Ou seja, queríamos rodar o algoritmo para particionar os posets genéricos de tamanho 5 a 300, com 100 instâncias para cada tamanho de poset e exibir os resultados num gráfico para depois poder analisá-lo.

Porém, ao gerar os posets aleatórios de tamanho n, descobrimos que a maioria deles (representanto aproximadamente $99,7\%$ dos casos) têm no máximo 3 elementos maximais (veja Seção 4.2). Isso faz com que o problema de particionamento de posets fique fácil de se resolver, já que na maior parte dos casos, temos no máximo $2^3 = 8$ possíveis subconjuntos dos elementos maximais. Logo, até o algoritmo da busca por força bruta leva pouco tempo para achar a partição ótima. Mesmo assim, fizemos os testes com esses casos, veja a Subseção 4.3.1.

No entanto, com essa informação, concluímos que os casos difíces são muito raros e partimos para outra abordagem do problema. Dessa vez, não estávamos mais interessados em gerar os posets aleatórios e sim, na quantidade dos elementos maximais que eles tinham.

Os resultados dos testes estão apresentados na Seção 4.3.

4.2 Geração de posets

Um dois maiores problemas que encontramos na implementação dos testes para o algoritmo KKGC foi a geração dos posets aleatórios que fossem distribuidos uniformemente. Após várias buscas e tentativas por conta própria, não conseguimos achar nenhum texto científico que apresentasse uma solução a este exato problema. Porém, nos deparamos com

o texto [5], onde encontramos um pseudocódigo para geração uniforme de grafos acíclicos dirigidos (em inglês: directed acyclic graph - DAG) aleatórios. Para podermos justificar a aplicação do algoritmo encontrado para gerar os posets aleatórios, vamos explicar a equivalência entre DAGs e posets. Esta é uma relação bem conhecida e intuitiva, porém poucos textos descrevem como ela é construída de fato. Por isso, decidimos demonstrá-la de forma mais detalhada neste trabalho.

4.2.1 Posets e DAGs

Definição 4.1. *Um **grafo dirigido** ou direcionado $G = (V, E)$ consiste em um conjunto não vazio V cujos elementos são chamados de vértices ou nodos, e um conjunto E de arcos ou arestas (direcionadas). Cada aresta $e \in E$ é especificada por par ordenado de vértices (u, v), onde $u, v \in V$.*

Uma aresta que conecta um vértice a ele mesmo é chamada de *laço*. As arestas que possuem os mesmos vértices como extermidades são chamadas de *arestas múltiplas*. Um grafo dirigido que não tem nenhum laço nem arestas múltiplas é dito **simples**.

Chamaremos de **outpoints** os vêrtices que não possuem as arestas de entrada, isto é, se $v_i \in V$ for um outpoint, então não existe $v_j \in V$ tal que $(v_j, v_i) \in E$.

Definição 4.2. *Um **caminnho** (dirigido) de comprimento $k-1$ num grafo direcionado G é uma sequência de vértices $v_1, v_2, ..., v_k$ distintos (exeto possivelmente o primeiro e o último), onde $(v_i, v_{i+1}) \in E$ para cada $i < k$. Um caminho fechado ou **ciclo** (direcionado) é um caminho onde o vértice inicial (v_1) é igual ao final (v_k).*

O primeiro vértice num caminho é chamado de *vértice inicial* e o último é chamado de *vértice final*.

Definição 4.3. *Um grafo direcionado $G = (V, E)$ é chamado de **grafo dirigido acíclico (DAG)** se não possuir ciclos.*

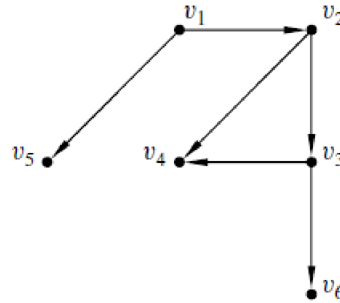
Para mostrar que existe uma correspondência entre DAGs e posets (finitos), temos que conseguir transformar um DAG qualquer em um poset e vice-versa.

$\Rightarrow$) Dado um grafo dirigido acíclico $G = (V, E)$, definimos uma relação $\preceq$ entre os vértices do conjunto $V = \{v_1, v_2, ..., v_n\}$ da seguinte maneira:

$$v_i \preceq v_j \;\Leftrightarrow\; v_i = v_j \text{ ou existe um caminho com inicio em } v_j \text{ e final em } v_i \text{ em } G.$$

É fácil verificar que esta relação satisfaz as propriedades da ordem parcial da Definição 3.1. Portanto, o conjunto V junto com a relação de ordem $\preceq$ formam um poset.

Figura 13: Um grafo dirigido acíclico com 6 vértices.



Fonte: [6]

$\Leftarrow$) A partir de um poset finito $P = (X, \preceq)$ construímos um grafo $G = (V, E)$. Definimos o conjunto dos vértices como sendo $V = X = \{v_1, v_2, ..., v_n\}$ e dizemos que um par de vértices (v_j, v_i) pertence ao conjunto das arestas se e somente se $v_i \preceq v_j$ e $v_i \neq v_j$, onde $v_i, v_j \in V$. Com a condição $v_i \neq v_j$ garantimos que o grafo não tenha laços. Ele não possui também ciclos direcinados, pois se tivesse, existiria uma sequência de elementos de P tal que $v_{i_1} \preceq v_{i_2} \preceq ... \preceq v_{i_m} \preceq v_{i_1}$ onde $v_{i_j} \neq v_{i_k}$ para todo $i_j \neq i_k$. Mas pela transitividade e antisimetria da relação $\preceq$ teríamos que $v_{i_1} = v_{i_2} = ... = v_{i_k}$ que contradiz a suposição de que os elementos sejam distintos. Logo, o grafo dirigido obtido é acíclico.

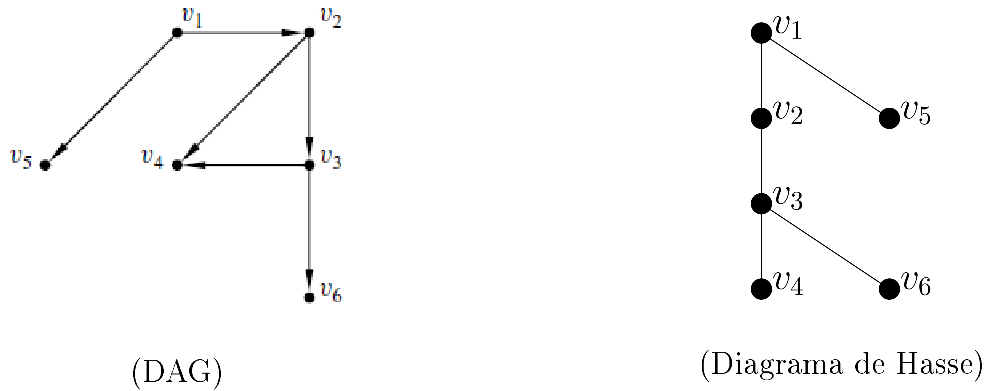
É importante observar que com as relações definidas acima, tendo um par de DAG e poset que correspondem, os outpoints do DAG correspondem aos elementos maximais do poset.

Exemplo 4.1. *Faremos o procedimento de transformar um DAG num poset para o grafo apresentado na Figura 13. Podemos falar de modo geral que, dado um vértice v_j, todos os elementos menores que ele são aqueles vértices v_i que são atingíveis a partir de v_j. Logo, a ordem do grafo será definida pelas seguintes relações:*

$v_1 \preceq v_1$	$v_2 \preceq v_2$	$v_3 \preceq v_3$
$v_2 \preceq v_1$	$v_3 \preceq v_2$	$v_4 \preceq v_3$
$v_3 \preceq v_1$	$v_4 \preceq v_2$	$v_6 \preceq v_3$
$v_4 \preceq v_1$	$v_6 \preceq v_2$	
$v_5 \preceq v_1$		
$v_6 \preceq v_1$		

Já vimos que podemos visualisar um poset por meio do diagrama de Hasse. Na Figura 14 apresentamos, junto ao DAG da Figura 13, o diagrama de Hasse do poset obtido a partir dele no Exemplo 4.1.

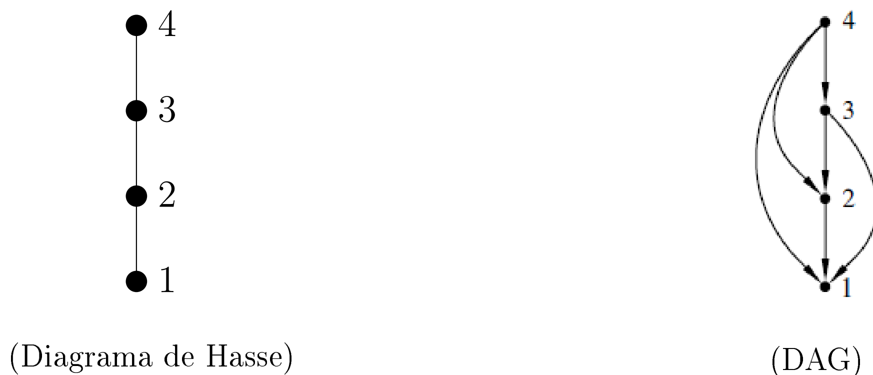
Figura 14: Grafo acíclico dirigido da Figura 13 e o diagrama de Hasse do poset obtido a partir dele (veja Exemplo 4.1).



Agora daremos um exemplo de como transformar um poset no grafo dirigido acíclico.

Exemplo 4.2. *Considere o poset P com a relação de $\leqslant$ definida sobre o conjunto* $\{1,2,3,4\}$*. Então o grafo G relacionado a P terá como vértices os números* $1,2,3$ *e* 4 *e o conjunto das arestas será* $E = \{(2,1),(3,2),(3,1),(4,3),(4,2),(4,1)\}$*. A comparação do DAG obtido com o diagrama de Hasse do poset P está apresentada na Figura 15.*

Figura 15: Diagrama de Hasse do poset P do Exemplo 4.2 e o DAG associado a P.



Dado que neste texto trabalhamos com os posets, a parte que nos interessa mais é obter um poset a partir do DAG. Quando chegarmos na parte de implementação, veremos que para fazer isto o algoritmo opera nas matrizes de adjacência dos grafos e posets. A matriz de adjacência de um poset está definida na Seção 3.1 (Definição 3.4). Agora, daremos a definição da matriz de adjacência de um grafo dirigido.

Definição 4.4. *Seja $G = (V,E)$ um grafo dirigido. A* ***matriz de adjacência*** *do grafo G é uma matriz cujas linhas e colunas são rotuladas por vértices, com* 1 *ou* 0 *na posição* (v_i, v_j) *conforme existir um aresta* $(v_i, v_j) \in E$ *ou não.*

Vale observar que a matriz de adjacência de um grafo simples tem que ter obrigatoriamente 0's na diagonal, já que este tipo de grafo não tem laços.

Exemplo 4.3. *A matriz de adjacência do grafo da Figura 13 tem a seguinte forma*

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Para que tenhamos a concordância com o artigo [5] em como representar um DAG por meio da matriz de adjacência, queremos obter uma matriz triangular inferior. Para fazer isso, basta reordenar os vêrtices do grafo conforme explicamos.

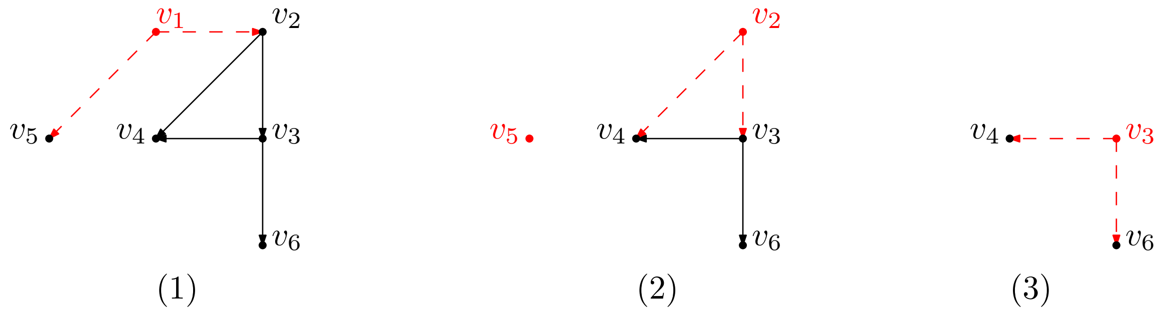
Seja $G = (V, E)$ um DAG com n vértices quais k_1 deles são outpoints. Depois de removermos todos os k_1 outpoints e todas as arestas de saída deles, obtemos um grafo menor com $n - k_1$ vértices e com um outro número de outpoints. Repetimos este passo $I - 1$ vezes até obtermos um grafo sem arestas, considerando k_i $(i = 1, 2, ..., I - 1)$ como sendo o número de outpoints removidos em cada etapa, e anotamos o número dos últimos vértices que sobraram por k_I. Então podemos falar de um **vetor dos outpoints** $K = (k_1, k_2, ..., k_I)$, onde $\sum_{i=1}^{I} k_i = n$.

Exemplo 4.4. *Vamos construir o vetor dos outpoints para o DAG da Figura 13. Os grafos que obtemos em cada etapa estão apresentados na Figura 16. No primeiro passo, o grafo tem somente um outpoint v_1, portanto $k_1 = 1$. Depois de remover v_1 e as arestas de saída dele, obtemos um grafo cujos outpoints são v_2 e v_5, logo $k_2 = 2$. Removendo estes, o único vértice que vira o outpoint do novo grafo é v_3, com isso $k_3 = 1$. No final sobram dois vértices v_4 e v_6, portanto neste caso temos $I = 4$ e $k_4 = 2$. O vetor dos outpoints é $K = (1, 2, 1, 2)$. Verificamos que $\sum_{i=1}^{4} k_i = 1 + 2 + 1 + 2 = 6$ que é exatamente o número dos vértices do grafo original.*

Tendo definido o vetor dos outpoints, podemos usá-lo para transformar a matriz de adjacência de um grafo numa matriz triangular inferior. Basta colocar os vértices do grafo na ordem inversa do que os removemos para obter o vetor dos outpoints.

Exemplo 4.5. *No caso do grafo da Figura 13, para transformar a matriz de adjacência em uma matriz triangular inferior, a ordem dos vértices teria que ser: $v_4, v_6, v_3, v_2, v_5, v_1$*

Figura 16: Os passos de construir o vetor dos outpoints para o Exemplo 4.4. Os vértices que removemos em cada passo (junto com as arestas) estão em vermelho.



(veja Exemplo 4.4). Após de reordenar as linhas e colunas da matriz que obtemos no Exemplo 4.3, a matriz de adjacência do grafo tem a forma de uma matriz triangular inferior, como desejado.

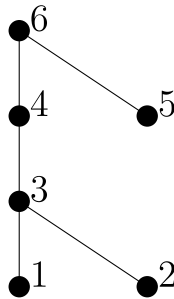
$$A_G = \begin{array}{c} \\ v_4 \\ v_6 \\ v_3 \\ v_2 \\ v_5 \\ v_1 \end{array} \begin{array}{c} \begin{array}{cccccc} v_4 & v_6 & v_3 & v_2 & v_5 & v_1 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Se enumerarmos os vértices nesta ordem, o diagrama de Hasse do poset correspontente estará enumerado da esquerda para a direita e de baixo para cima (Figura 17). Logo, a matriz de adjacência do poset terá a forma de uma matriz triangular superior.

$$A_P = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Se quisermos transformar a matriz de adjacência A_G de um DAG na matriz de adjacência A_P do poset correspondente, primeiro temos que garantir que a primeira matriz

Figura 17: Diagrama de Hasse do poset correspondente ao grafo da Figura 13, após de enumerar os vértices da forma apresentada no Exemplo 4.5.



esteja na forma de uma matriz triangular inferior. Suponhamos que enumeramos os vértices de 1 a n, onde n - número total dos vértices no grafo. O próximo passo é alterar a matriz A_G e colocar 1 em cada posição (i, i), pois a ordem parcial é reflexiva, e na posição (i, j) se o vértice j for atingível a partir do vértice i. Para descobrir quais são os vértices j-atingíveis a partir do vértice i, analisamos as linhas de todos os vértices conectados ao vértice i. Fazemos isto percorrendo as linhas da matriz, de cima para baixo. Por último, temos que considerar a matriz transposta, tendo em mente que os outpoints do grafo tornam-se os elementos maximais do poset.

Exemplo 4.6. *Vamos mostrar os passos para transformar a matriz de adjacência do grafo obtida no Exemplo 4.5 na matriz de adjacencia do poset. Vejamos que, de fato, a matriz final é igual ao matriz de adjacência do poset apresentada no Exemplo 4.5.*

$$A_G = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \longrightarrow$$

$$\longrightarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\top} \longrightarrow \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = A_P$$

Agora, como já mostramos a equivalência entre os grafos dirigidos acíclicos e os conjuntos parcialmente ordenados, e como transformar a matriz de adjacência de um DAG na matriz de adjacência de um poset, podemos prosseguir e dizer sobre a experiência que tivemos com as informações e o algoritmo encontrados em [5]. Porém, devemos dizer que adaptamos todas as nomenclaturas para a linguagem dos posets.

4.2.2 Distribuição de posets

Todos os conceitos introduzidos nesta subseção estão extraídos do texto [5] que apresenta os métodos de gerar um DAG arbirtário. Como temos a equivalência entre DAGs e posets, usamos um desses métodos, chamado de *método de enumeração*, que é um método recursivo, para gerar um poset aleatório.

Seja $a_{n,k}$ o número de posets rotulados[1] com n elementos dos quais k são maximais ($1 \leqslant k \leqslant n$). Então, dado n, podemos recursivamente calcular o número de todos os posets usando as seguintes equações:

$$a_{n,k} = \binom{n}{k} b_{n,k}, \qquad b_{n,k} = \sum_{s=1}^{n-k} (2^k - 1)^s 2^{k(n-k-s)} a_{n-k,s},$$

onde $a_{n,n} = 1$ (o poset sem relação estrita) e $b_{n,k}$ as quantidades auxiliares com $b_{n,n} = 1$. O número total de posets com n elementos é

$$a_n = \sum_{k=1}^{n} a_{n,k}.$$

A separação em termos de elementos maximais é uma chave para distribuir os posets, considerando-se todos os rótulos possíveis.

Como um exemplo, o número de posets de tamanho 5 especificado por número de elementos maximais está apresentado na Tabela 1.

Tabela 1: O número de posets com 5 elementos onde k são maximais.

k	1	2	3	4	5
$a_{5,k}$	16885	10710	1610	75	1

O primeiro passo do algoritmo apresentado em [5] é computar todos os valores de $a_{n,k}$ (e $b_{n,k}$), assim como as somas totais a_n. Porém, com as limitações de máquina, conseguimos calcular os valores somente até $n = 42$. Ou seja, usando este método direto de achar

[1] Os posets são rotulados no sentido que, dados dois posets $(P, \preceq_P)$ e $(Q, \preceq_Q)$ definidos por $1 \preceq_P 2$ e $2 \preceq_Q 1$ respectivamente, apesar de isomorfos, são contados duas vezes.

a distribuição uniforme, não conseguiríamos fazer os testes para os posets com mais de 42 elementos. No entanto, os números que obtivemos foram bem interessantes.

Tabela 2: Distribuição real dos posets (em %).

Tamanho do poset (n) \ Número de elementos maximais (k)	1	2	3	4	5	6	7	8	9	10
5	57,665	36,577	5,4984	0,2561	0,00342					
6	57,506	36,608	5,6015	0,2797	0,00492	$2,64 \cdot 10^{-5}$				
7	57,457	36,617	5,6328	0,2869	0,00543	$3,87 \cdot 10^{-5}$	$8,78 \cdot 10^{-8}$			
8	57,443	36,620	5,6423	0,2892	0,00559	$4,30 \cdot 10^{-5}$	$1,30 \cdot 10^{-7}$	$1,28 \cdot 10^{-10}$		
9	57,438	36,621	5,6452	0,2899	0,00564	$4,44 \cdot 10^{-5}$	$1,44 \cdot 10^{-7}$	$1,89 \cdot 10^{-10}$	$8,24 \cdot 10^{-14}$	
10	57,437	36,621	5,6461	0,2901	0,00566	$4,48 \cdot 10^{-5}$	$1,49 \cdot 10^{-7}$	$2,11 \cdot 10^{-10}$	$1,22 \cdot 10^{-13}$	$2,40 \cdot 10^{-17}$
...										
42	57,436	36,621	5,6465	0,2902	0,00567	$4,50 \cdot 10^{-5}$	$1,55 \cdot 10^{-7}$	$2,21 \cdot 10^{-10}$	$1,43 \cdot 10^{-13}$	$4,16 \cdot 10^{-17}$

A Tabela 2 mostra a proporção percentual entre $a_{n,k}$ e a_n onde $n = \{5, 6, ..., 10, 42\}$ e $1 \leqslant k \leqslant 10$. Umas das primeiras observações que vem à mente é que os números em colunas parecem estar convergindo. De fato, foi provado em [7] que a fração $\frac{a_{n,k}}{a_n}$ converge se n tende a infinito e para $n \geqslant 20$ temos que $|A_k - \frac{a_{n,k}}{a_n}| < 10^{-10}$, onde $A_k = \lim_{n\to\infty} \frac{a_{n,k}}{a_n}$. Portanto, para $n \geqslant 20$, devido as limitações computacionais de máquina, a distribuição de k pode ser obtida diretamente da distribuição limitante apresentada na Tabela 3.

Tabela 3: A ocorrência relativa do número de elementos maximais em posets grandes. A_k foi multiplicado por 10^{10}. A_8 é aproximadamente $2,2 \cdot 10^{-12}$, então $k > 7$ pode ser excluído neste nível de precisão.

A_1	5743623733
A_2	3662136932
A_3	564645435
A_4	29023072
A_5	566517
A_6	4496
A_7	15
A_8	0

Fonte: [5, *Uniform random generation of large acyclic digraphs*, p. 7]

Outra observação, interessante para o nosso caso, é que somando os três primeiros

valores da Tabela 3, obtemos

$$A_1 + A_2 + A_3 = 0,5743... + 0,3662... + 0,0564... \approx 0,9969.$$

Isto significa que, para n grande (que seja $n \geqslant 20$), aproximadamente 99,7% dos posets de tamanho n têm menos que 4 elementos maximais.

Neste ponto é importante observar que, apesar do problema de particionamento de posets ser um problema NP-difícil, temos que as instâncias difíceis são raras (menos que $0,004 > 1 - 0,9969$).

4.2.3 Algoritmo

Com todas as informações que obtivemos sobre a distribuição dos posets, concluímos que não vale a pena analisar o comportamento do algoritmo KKGC em posets aleatórios, pois o problema de particionamento de posets com até 3 elementos maximais (que seria a grande maioria dos casos) é muito fácil de se resolver. Portanto, não nos interessa gerar um poset arbitrário com a distribuição uniforme, e sim, gerar um poset de tamanho n com dado número de elementos maximais k para analisarmos o desempenho do KKGC conforme k varia.

O algoritmo que implementamos para construir um poset de tamanho n e número de elementos maximais k ($1 \leqslant k \leqslant n$) está baseado no pseudocódigo apresentado em [5] que gera um DAG. Como vimos que existe uma correspondência entre DAGs e posets (veja Subseção 4.2.1), depois de obter um DAG, o transformamos em um poset, lembrando que os elementos maximais do poset correspondem aos outpoints do grafo.

Podemos dividir o algoritmo implementado em três etapas:

Primeiro, temos que gerar os outpoints de forma recursiva. O pseudocódigo gerador de DAG ($GdeDAG$), usa direto os valores de $a_{n,k}$ e $b_{n,k}$ para fazer isso de maneira que a distribuição seja mais real possível. Nós, porém, simplificamos a recursão considerando o fato de que 99,7% dos outpoins são 1 (57,43%), 2 (36,62%) ou 3 (5,65%). Usamos esta informação em casos onde $n > 3$, e nos 0,3% que restaram, damos a mesma chance de aparecer aos números entre 4 e n. Para $n \leqslant 3$ usamos a distribuição real apresentada na Tabela 4.

Logo, geramos um vetor dos outpoints K da seguinte maneira: $K(1) = k$ dado; escolhemos $K(2)$ como se fóssemos gerar os outpoints para um DAG menor do que original, com $n_1 = n - K(1) = n - k$ elementos; escolhemos $K(3)$ como se o DAG tivesse $n_2 = n_1 - K(2)$ elementos e assim continuamos e paramos quando $n_j = n_{j-1} - K(j)$ for menor que 1 para algum j.

Tabela 4: Distribuição real dos posets/DAGs de até 3 elementos.

n \ k	1	2	3
1	100%		
2	66,67%	33,33%	
3	60%	36%	4%

Tendo em mãos o vetor K de comprimento I, podemos definir as arestas entre os vértices. Isso também é feito da forma recursiva prevista no GdeDAG, que constrói uma matriz de adjacência de um DAG. O algoritmo começa a partir dos dois útimos elementos do vetor K e vai descendo os níveis. Pega cada vértice do nível mais baixo e decide se existe uma aresta ou não entre ele e algum vértice dos níveis mais altos por atribuir aleatoriamente 1 ou 0 na matriz de adjacência, sendo que os vértices do nível mais baixo devem estar conectados a pelo menos um vértice dos níveis mais altos, já que estes (os vértices do nível mais baixo) são considerados os outpoints. Esta parte termina quando percorrermos o vetor K inteiro do fim até o começo, obtendo a matriz de adjacência do DAG que é uma matriz triangular inferior.

O último passo é transformar a matriz de adjacência do DAG na matriz de adjacência do poset correspondente. Fazemos isto do jeito que descrevemos no final da Subseção 4.2.1. Assim, obtemos a matriz de adjacência de um poset de tamanho n com k elementos maximais.

4.3 Resultados

Podemos dividir os nossos resultados em duas partes: os casos fáceis, que analisam o comportamento do algoritmo KKGC no caso dos posets com até 3 elementos maximais, e os casos difíceis (mais raros) com os posets com mais de 3 elementos maximais. Para analisar o desempenho do algoritmo, em cada caso usamos uma abordagem diferente dos testes.

Dependendo do caso, vamos comparar o algoritmo KKGC a uma busca por força bruta ou um algoritmo ganancioso, portanto agora descrevemos como estes algoritmos agem no caso de problema de particionamento de posets.

O algoritmo da *busca por força bruta* simplesmente acha todas as possíveis partições do conjunto de elementos maximais, calcula a discordância de cada par de subconjuntos usando a Definição 3.8 e retorna a partição com o menor valor da discordância que é a partição ótima.

Já o *algoritmo ganancioso*, assim como no problema de particionamento clássico, não garante que o resultado seja ótimo. Primeiro, ele classifica os elementos maximais por peso em ordem decrescente. Em seguida, pega os dois primeiros elementos da lista (os elementos "mais pesados") e os coloca em subconjuntos diferentes. Depois pega o terceiro elemento e o coloca em subconjunto cujo peso é menor (subconjunto "mais leve"). Assim, continua colocando os elementos, um por um, sempre no subconjunto "mais leve", até distribuir todos elementos. No final, calcula o valor da discordância e retorna a partição encontrada.

Para implementar todos os algritmos e realizar os testes usamos o software **GNU Octave**, que é uma das principais alternativas gratuitas para o MATLAB. Os códigos-fonte dos algoritmos encontram-se no Apêndice A.

4.3.1 Casos fáceis

Consideramos um caso fácil aquele em que o algoritmo está sendo aplicado para particionar um poset com no máximo 3 elementos maximais. Portanto, temos somente três opções:

- 1 elemento maximal:

  Nesta situação, a partição é óbvia. A única maneira de particionar um poset de tamanho n com 1 elemento maximal é colocar o poset inteiro (que é o ideal gerado por este único elemento maximal) num subconjunto, enquanto o outro fica vazio. A discordância desta partiçao sempre será igual a n.

- 2 elementos maximais:

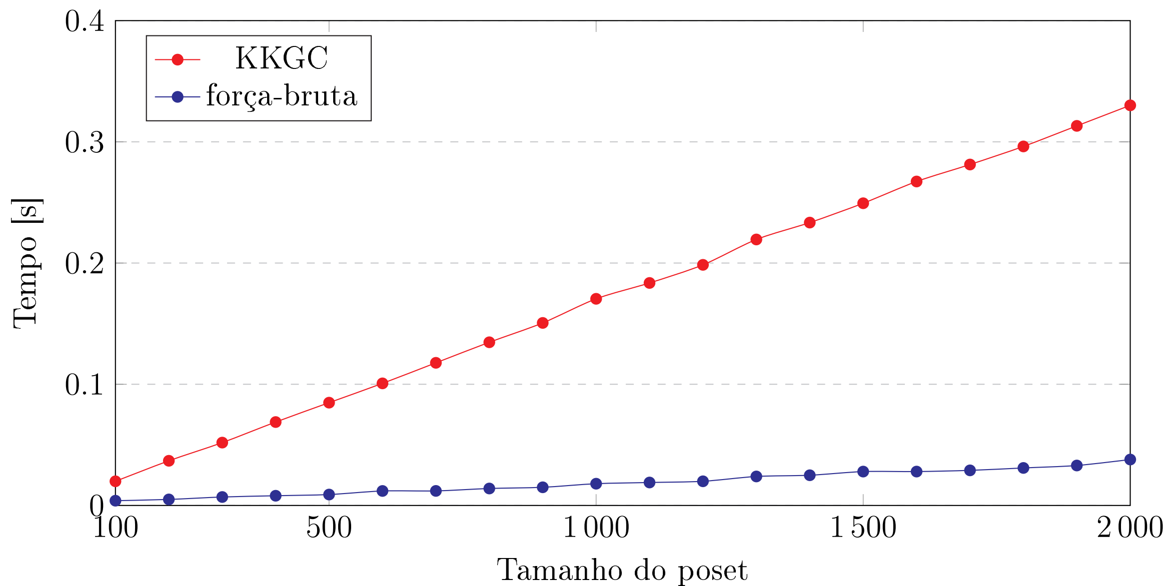
  No caso de um poset com 2 elementos maximais, a partição ótima é aquela que separa os elementos maximais.

- 3 elementos maximais:

  Neste cenário, não há padrão único. Portanto, na presente subseção, analisaremos o desempenho do algoritmo KKGC sobre os posets com exatamente 3 elementos maximais.

Como suponhamos que estamos lidando com um poset com 3 elementos maximais, não faz muito sentido comparar o desempenho do algoritmo KKGC com o algoritmo da força-bruta, já que temos somente $2^3 = 8$ possíveis subconjuntos dos elementos maximais, ou seja, 4 partições diferentes. Na nossa implementação, o algoritmo de busca por força bruta retorna o resultado mais rápido que o KKGC neste caso (veja a Figura 18), o que é

Figura 18: Comparação do tempo de execução entre os algoritmos força-bruta e KKGC feita com os posets com 3 elementos maximais de tamanho entre 100 e 2000 com o incremento de 100. Conforme o tamanho do poset aumentar, o tempo de execução do algoritmo KKGC cresce mais rápido que o do algoritmo de força-bruta. Podemos justificar esta diferença pelo simples fato de que o algoritmo KKGC faz operações nos vetores em cada passo. Porém, em ambos os casos, o crescimento é linear.



esperado, pois a implementação do KKGC tem um custo fixo da operação de diferenciação.

Com esta observação, achamos interessante verificar quais são os nodos da árvore de busca construida pelo KKGC que retornam a solução ótima. Descobrimos que aproximadamente em 96% instâncias do problema, a solução ótima está representada pela primeira folha (veja a Tabela 5), ou seja, em aproximadamente 96% das vezes, o algoritmo KKG retorna a partição ótima no caso de particionamento de posets com 3 elementos maximais.

Portanto, concluímos que uma análise mais razoável neste cenário (os posets com 3 elementos maximais) seria a comparação de desempenho entre o KKG e o algoritmo ganancioso.

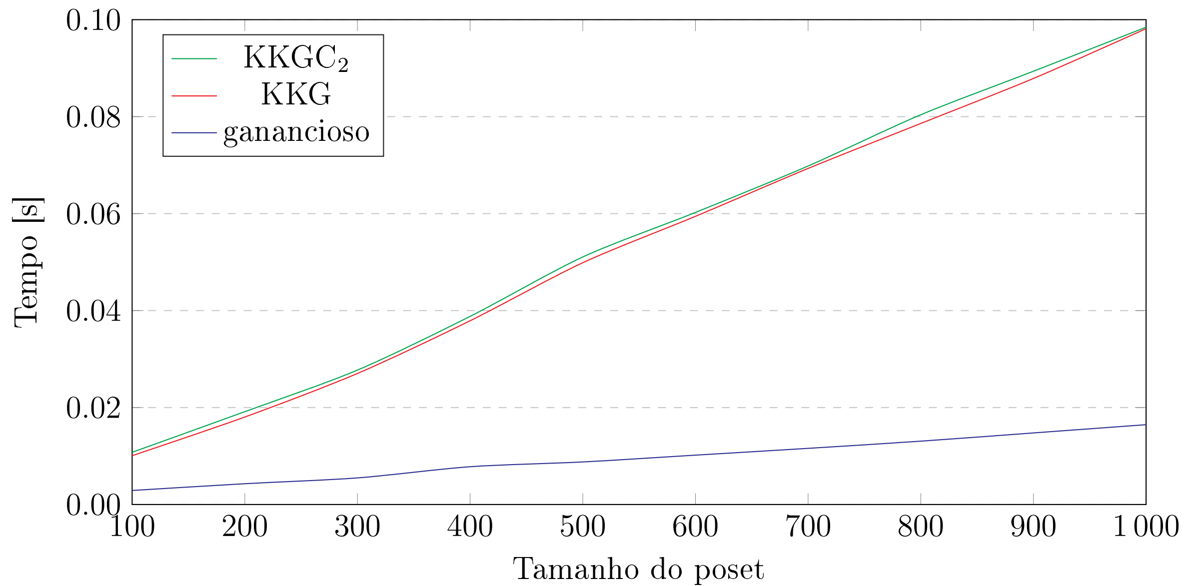
A priori, considerando a nossa implementação, o algoritmo ganancioso age mais rápido (veja a Figura 19). Porém, não sabemos ainda se ele tem alguma vantagem sobre o KKG em termos da partição obtida.

Os resultados da comparação do desempenho em termos de discordância entre os dois algoritmos (KKG e ganancioso) encontram-se na Figura 20. Os testes foram feitos com os posets com 3 elementos maximais de tamanho entre 100 e 2000 com o incremento de 100, analisando 100 instâncias de cada tamanho. No gráfico, está apresentado o número das instâncias de cada tamanho de poset em que o KKG ou o algoritmo ganancioso obteve

Tabela 5: Distribuição da solução ótima entre as folhas da árvore de busca construída pelo algoritmo KKGC no caso dos posets com 3 elementos maximais. O teste foi feito com os posets de tamanho n analisando s instâncias de cada tamanho. Para $n = 1000$ geramos somente 100 instâncias, pois gerar um poset de tamanho tão grande levou muito tempo.

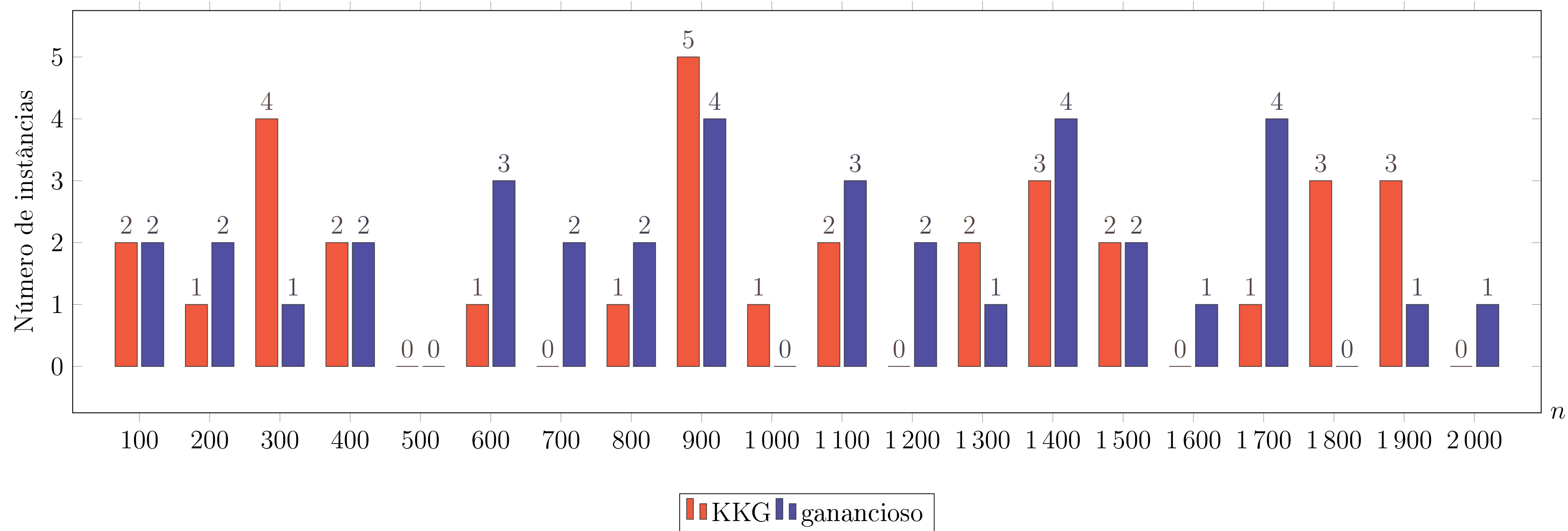
n	Folhas				s
	1	2	3	4	
10	970	30	0	0	1000
100	957	44	0	0	1000
500	957	44	0	0	1000
1000	98	2	0	0	100

Figura 19: Comparação do tempo de execução entre os algoritmos KKG, $KKGC_2$ e ganancioso feita com os posets com 3 elementos maximais de tamanho entre 100 e 2000 com o incremento de 100. Igual no caso da Figura 18, conforme o tamanho do poset aumentar, o tempo de execução do algoritmo KKG cresce mais rápido devido às operações nos vetores, porém o crescimento é linear.



melhor resultado (achou a partição com menor discordância), representado pelas cores vermelho ou azul respectivamente.

Figura 20: Comparação do desempenho dos algoritmos KKG e ganancioso feita com os posets de tamanho n com 3 elementos maximais.



Analisando os resultados apresentados na Figura 20 não é possível concluir qual algoritmo tem melhor desempenho no caso dos posets com 3 elementos maximais. De fato, os valores encontrados na Tabela 6, que apresenta a eficiência do algoritmo ganancioso em achar a partição ótima, e comparados com os da Tabela 5 não sugerem claramente qual dos algoritmos retorna a solução ótima com maior frequência. Levando isto em consideração, e o fato de que o algoritmo ganancioso age mais rápido, concluímos que não é viável o uso do algoritmo KKG para resolver o problema de particionamento de posets com 3 elementos maximais. No entanto, sabemos que o KKGC_2 sempre garante a solução ótima se o poset tem 3 elementos maximais.

Tabela 6: Eficiência do algoritmo ganancioso no caso dos posets com 3 elementos maximais. A tabela apresenta o número de partições ótimas encontradas pelo algoritmo. O teste foi feito com os posets de tamanho n analisando s instâncias de cada tamanho.

n	Número de partições ótimas	s
10	969	1000
100	963	1000
500	956	1000
1000	98	100

Assim, eventual interesse nos algoritmos KKGC_r deve ser restrito aos casos difíceis ($k > 3$), que já sabemos serem raros.

4.3.2 Casos difíceis

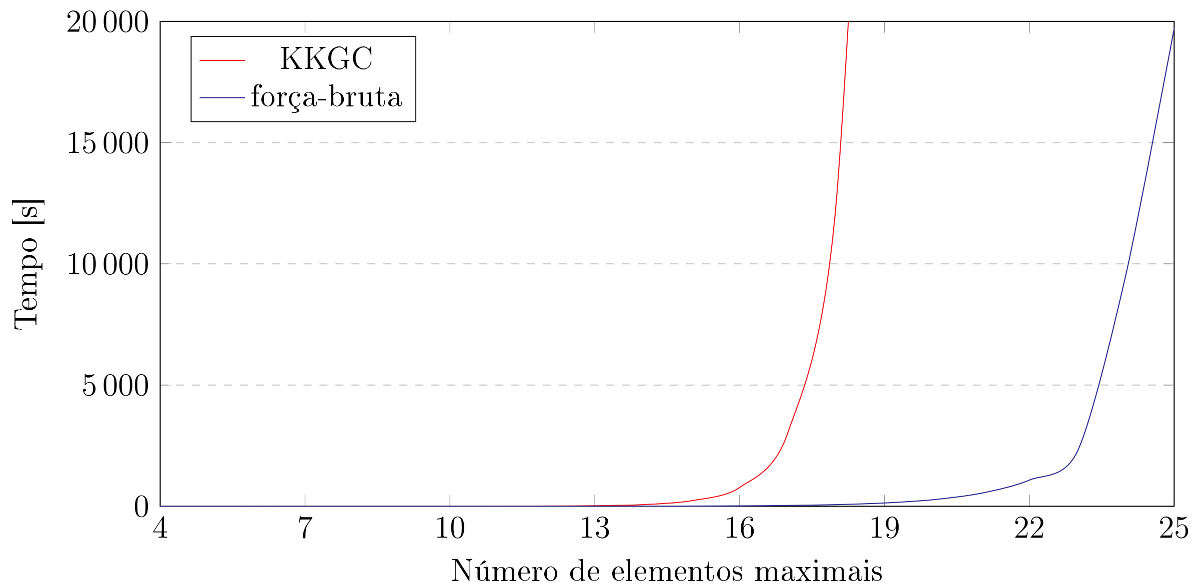
Neste trabalho, consideramos um caso "difícil" de particionar um poset se este tiver ao menos 4 elementos maximais. Porém, os casos realmente difíces podem ser considerados aqueles para quais o algoritmo de força-bruta não consegue dar uma resposta. Nesta situação usamos outra abordagem para analisar o desempenho do algoritmo KKGC: fixamos o tamanho de poset e vamos aumentando o número de elementos maximais.

Primeiro, comparamos o tempo de execução dos algoritmos KKGC e força-bruta conforme aumentamos o número de elementos maximais. Fizemos testes com os posets de tamanho 100, aumentando, a partir de 4, o número de elementos maximais. Os resultados estão apresentados na Figura 21.

Como esperado, observamos crescimento exponencial no tempo de execução dos ambos algoritmos. Pórem, o algoritmo KKGC explode mais rápido. No caso do algoritmo de força-bruta, conseguimos chegar num poset com 25 elementos maximais. O algoritmo

levou aproximadamente 5 horas e 30 minutos para dar o resultado, enquanto num poset com $k = 24$ demorou 2 horas e 40 minutos e num poset com 23 elementos maximais, levou 40 minutos. Já o algoritmo KKGC demorou 50 minutos para particionar um poset com 17 elementos maximais, e 3 horas e 30 minutos no caso de um poset com 18 elementos maximais.

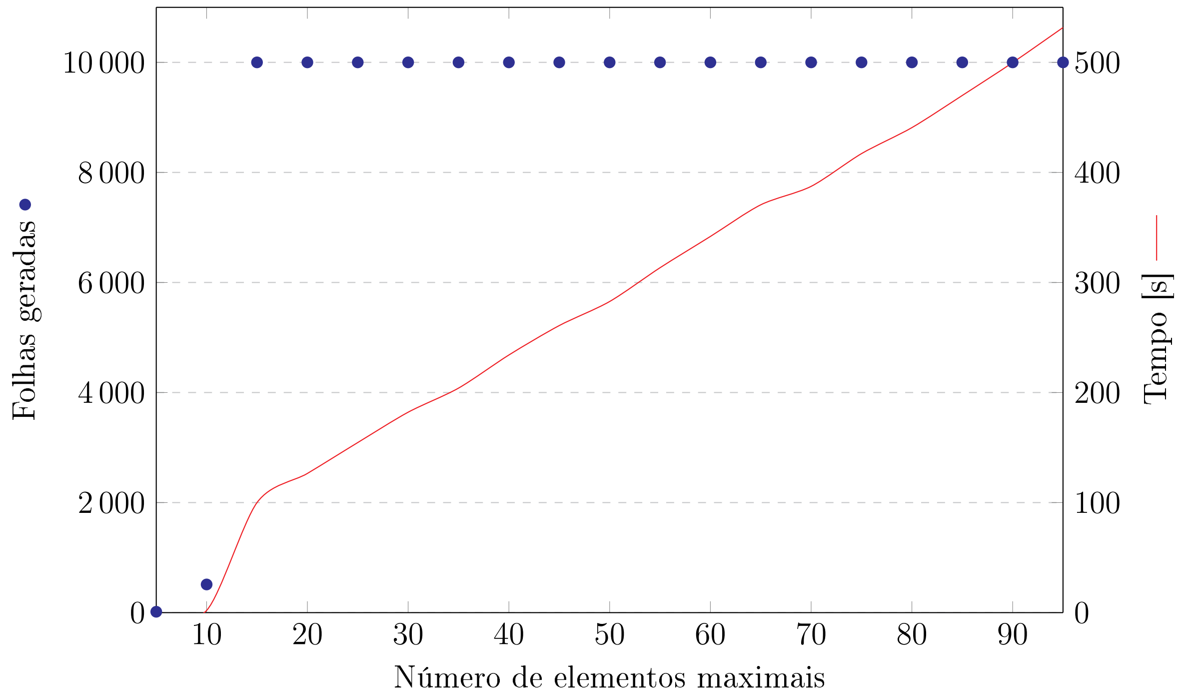
Figura 21: Comparação do tempo de execução entre os algoritmos KKGC e força-bruta.



Poderíamos concluir aqui que o KKGC não tem nenhuma vantagem sobre o algoritmo de força-bruta, temos que lembrar que o algoritmo de força-bruta tem que analisar todas as possíveis soluções do problema antes de dar a resposta. A função que usamos na nossa implementação que acha todos os subconjuntos (o conjunto das partes) do conjunto de elementos maximais, no software que estamos usando, está implementada somente para um conjunto com até 32 elementos. Isto significa que, mesmo se tivermos tempo para esperar o resultado, não coseguiríamos particionar um poset de forma ótima usando o algoritmo de força-bruta se este tiver mais que 32 elementos maximais. Além disso, não é preciso rodar o algoritmo KKGC inteiro para obter algum resultado. Logo, podemos parar o algoritmo na hora que quisermos, tendo em mente que quanto mais tempo ele rodar, a qualidade da solução vai melhorando.

Justamente por isso, no Capítulo 3, introduzimos o conceito do algoritmo KKGC_r que é a restrição do algoritmo completo a produzir e analisar somente r folhas. As Figuras 22, 23, 24 apresentam a viabilidade do algoritmo KKGC_r, onde fixamos $r = 10000$, com os poset de tamanho 100, 500 e 1000 respectivamente, dependendo do número de elementos maximais. Limitamos também o tempo máximo de execução do algoritmo para 1 hora.

Figura 22: Viabilidade do algoritmo $KKGC_{10000}$ testada com posets de tamanho 100.



Os dados apresentados na Figura 22 são os resultados dos testes feitos com posets de tamanho 100, aumentando o número de elementos maximais k de 5 a 95, com incremento de 5. Colocamos duas informações num gráfico só: o número das folhas geradas (eixo vertical à esquerda, bolinhas azuis) e o tempo de execução do algoritmo (eixo vertical à direita, linha vermelha). O eixo horizontal representa o número de elementos maximais do poset. Os dois primeiros casos, onde $k = 5$ e $k = 10$, retornam a solução ótima, pois o número total das folhas é $2^{5-1} = 2^4 = 16$ e $2^{10-1} = 2^9 = 512$ respectivamente e ambos são menores que $r = 10000$. Logo, o algoritmo consegue percorrer a árvore inteira e retornar a solução ótima. Já no caso onde $k = 15$, o número total das folhas seria $2^{15-1} = 2^{14} = 16384 > 10000 = r$. Portanto, o algoritmo para quando atinge a folha r se $k \geqslant 15$. Vale lembrar que limitamos o tempo de construção das folhas até 1 hora, ou seja, o maior valor que pode aparecer no eixo vertical à direita é 3600 s. Porém, lidando com um poset de tamanho 100, o caso mais complicado testado, $k = 95$, é executado em menos que 9 minutos. Logo, podemos concluir que o algoritmo $KKGC_{10000}$ não tem nenhuma dificuldade em retornar uma solução para particionar um poset de tamanho 100 com qualquer número de elementos maximais.

A Figura 23 apresenta os resultados dos testes feitos com posets de tamanho 500, aumentando o número de elementos maximais de k de 25 a 475 com incremento de 25. Vemos no gráfico que o tempo de execução cresce linearmente até atingir o valor máximo de 3600 s no caso de um poset com 450 elementos maximais onde o algoritmo não conseguiu

Figura 23: Viabilidade do algoritmo $KKGC_{10000}$ testada com posets de tamanho 500.

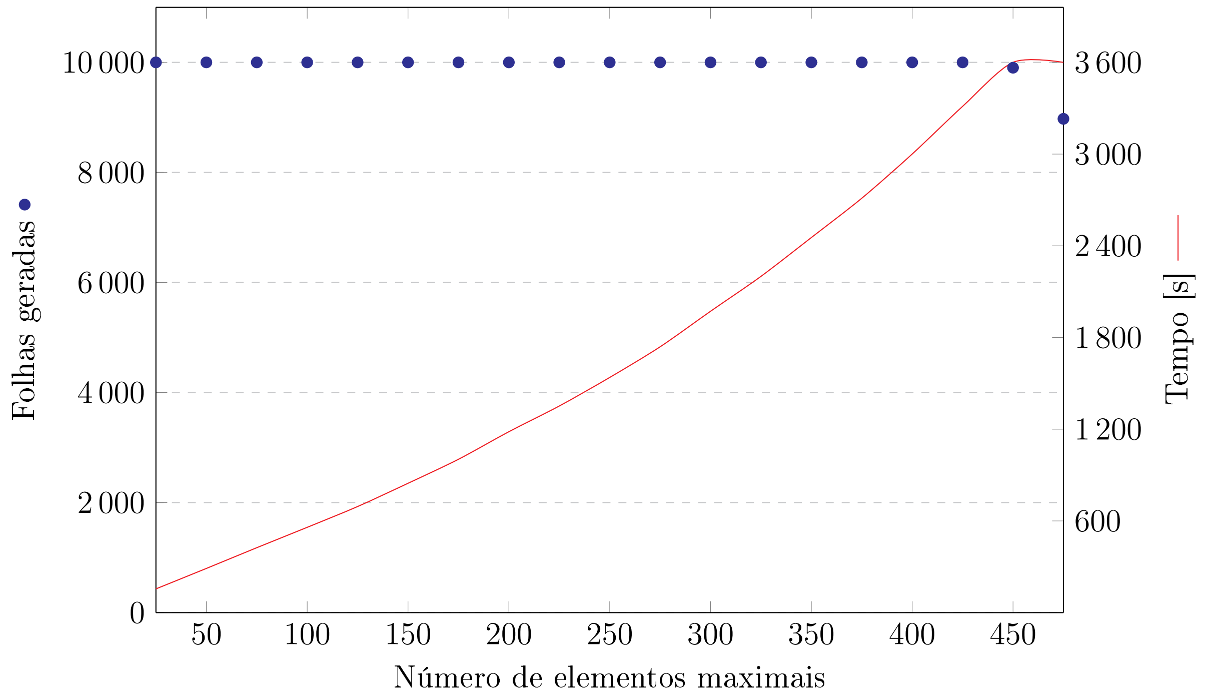
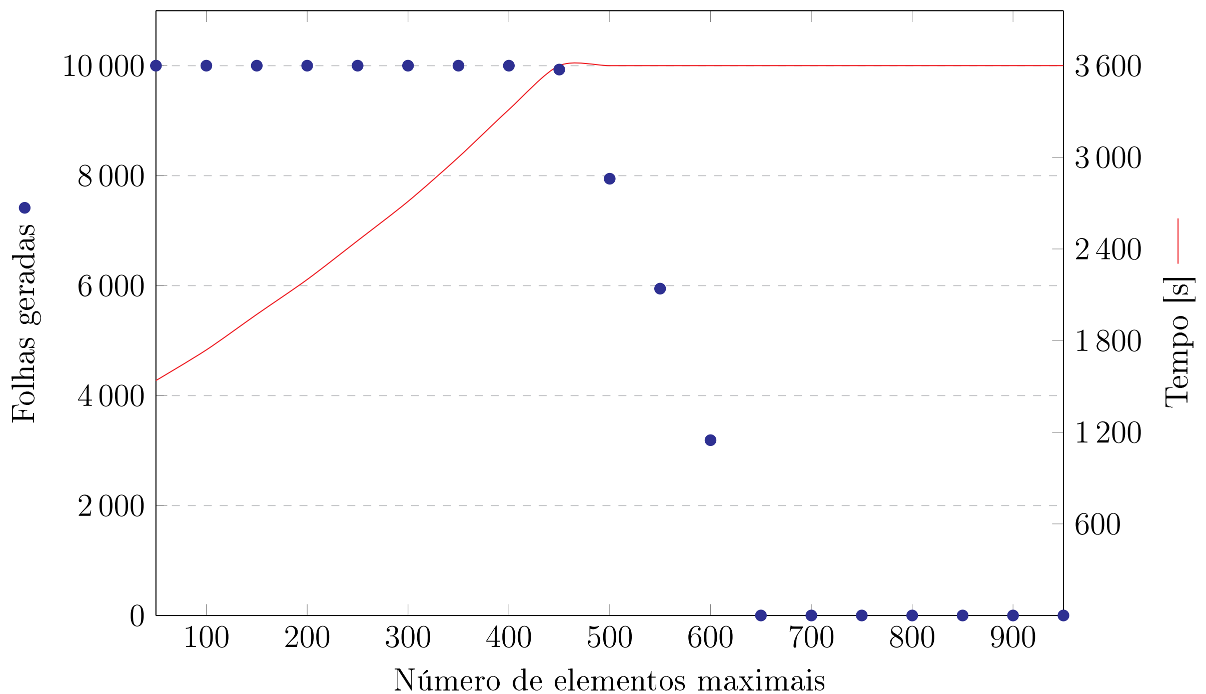


Figura 24: Viabilidade do algoritmo $KKGC_{10000}$ testada com posets de tamanho 1000.

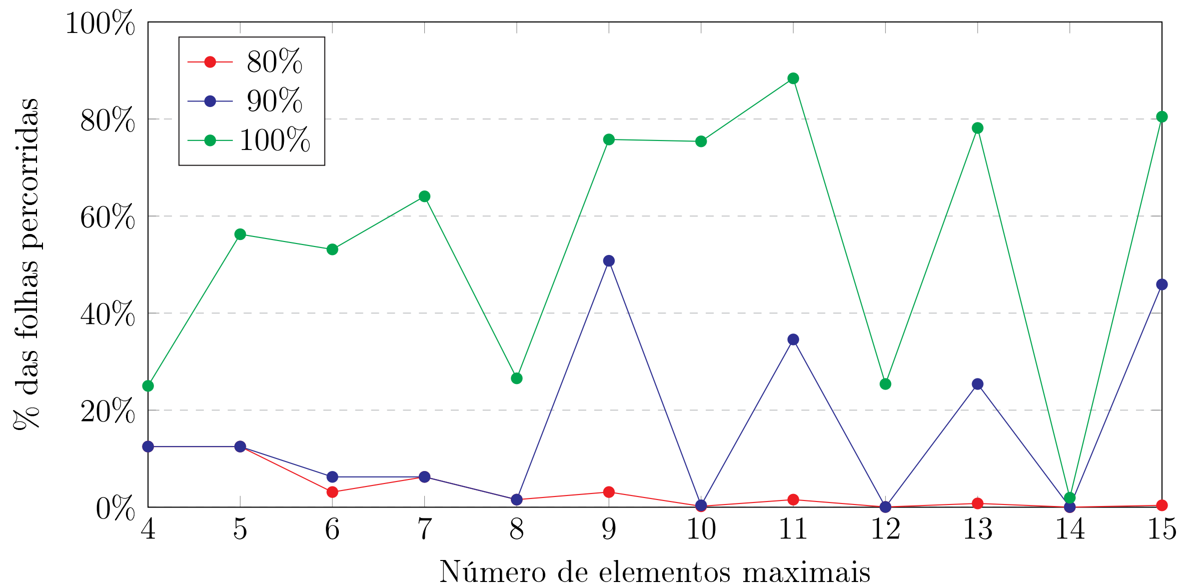


percorrer todas as 10000 folhas em menos que 1 hora. A situação se repete no caso de um poset com $k = 475$, onde o algoritmo conseguiu analisar um pouco menos que 9000 em 1 hora. Nestes dois últimos casos, observamos o decrescimento da eficiência do algoritmo.

O último gráfico do mesmo grupo dos testes, apresentado na Figura 24, reflete a eficiência do algoritmo KKGC_{10000} no caso dos posets de tamanho 1000 com o número de elementos maximais entre 50 e 950 com incremento de 50. Observamos a queda do desepmenho do algoritmo na mesma faixa dos números de elementos maximais que no caso dos posets de tamanho 500, que começa por volta de $k = 450$. A queda é relativamente rápida, tanto que com $k = 650$, em 1 hora, o algoritmo só consegue construir a primeira folha, e quando $k > 750$, nem a primeira folha pode ser construida no tempo que definimos.

É relevante mencionar um outro motivo que comprova que usar o algoritmo KKGC_r pode ser viável e interessante. Nos casos que conseguimos rodar o KKGC completo no tempo razoável, vamos dizer que estes casos são os posets com até 15 elementos maximais, a maioria das soluções foi obtida das primeiras folhas da árvore. O gráfico apresentado na Figura 25 mostra essa distribuição da solução ótima entre as folhas da árvore.

Figura 25: Distribuição da solução ótima entre as folhas da árvore para os posetes de tamanho 100 e k elementos maximais ($4 \leqslant k \leqslant 15$).



Os testes foram feitos com os posets de tamanho 100, aumentando o número de elementos maximais k de 4 a 15 e analisando 120 instâncias para cada k. O eixo horizontal representa o número de elementos maximais, e o eixo vertical - a porcentagem dos números das folhas com a solução ótima comparados com a quantidade total das folhas da árvore.

Como podemos ver, a grande parte dos casos (80%) se concentra nas primeiras folhas da árvore, por exemplo, se pegarmos os posets com mais de 5 elementos maximais, as folhas que retornaram a solução ótima encontram-se entre menos que 5% das primeiras

folhas.

Já se olharmos 100% das instâncias geradas, o valor no gráfico representa a pior instância entre todas, isto é, uma instância em que a partição ótima foi encontrada numa folha posterior. Por exemplo, num poset com 7 elementos maximais, esta folha representa 64% de todas as folhas da árvore, enquanto no caso do poset com 11 elementos maximais, são 88% das folhas. Mesmo que eventualmente existem as instâncias que acharam a solução ótima somente nas últimas folhas, estas são poucas (menos que 10%).

Um fenômeno interessante que observamos neste teste é que aparentemente os posets com um número par de elementos maximais são mais fáceis de particionar, pois olhando somente os posets com $k = 4, 6, 8, 10, 12, 14$, em 95% das instâncias analisadas para cada k, a solução ótima foi encontrada nas folhas número 1, 2 ou 4.

Quando rodamos o algoritmo ganancioso usando as mesmas instâncias do teste anterior, este também apresentou maior facilidade em encontrar a solução ótima nos casos dos posets com o número par de elemetnos maximais. Veja a Tabela 7, que mostra claramente que o algoritmo ganancioso acerta a resposta com muita frequência quando k for par.

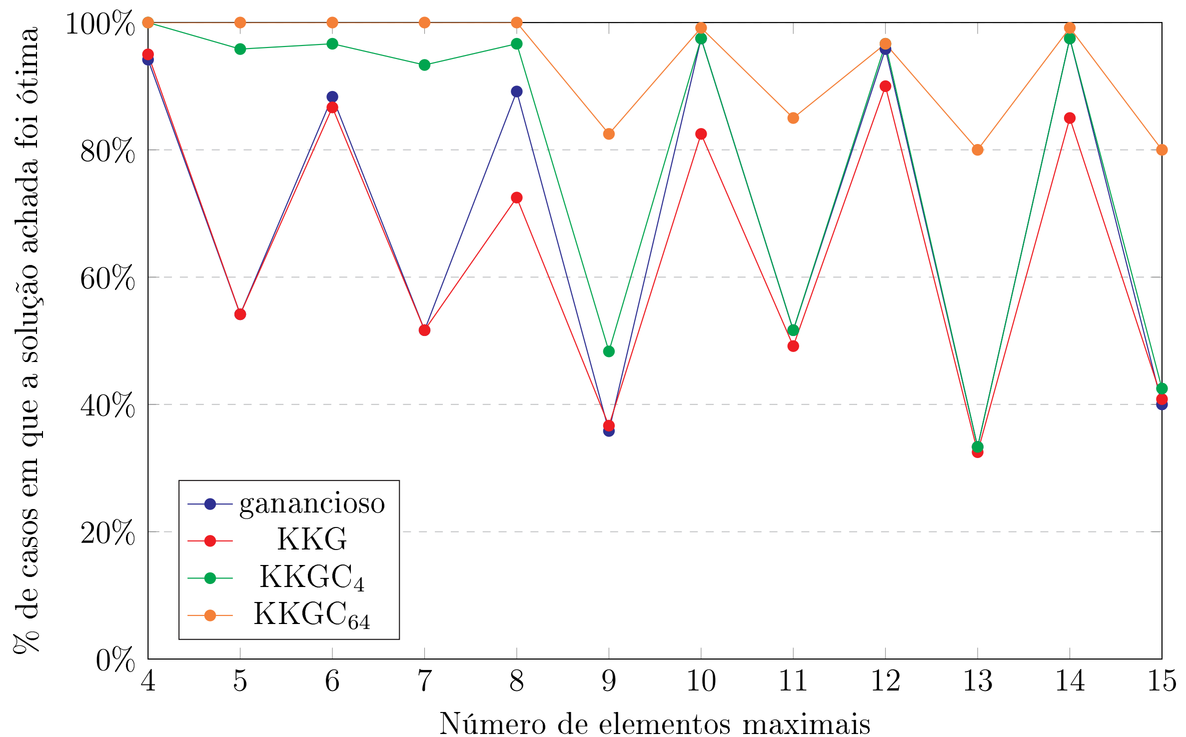
Tabela 7: A porcentagem de vezes que o algoritmo ganancioso encontra a solução ótima nos posets de tamanho 100 com k elementos maximais.

k	%
4	94, 17%
5	54, 17%
6	88, 33%
7	51, 67%
8	89, 17%
9	35, 83%
10	97, 50%
11	51, 67%
12	95, 83%
13	33, 33%
14	97, 50%
15	40, 00%

Comparando a eficiência do algoritmo KKG com o algoritmo ganancioso, rodados neste mesmo conjunto de posets, novamente, não conseguimos claramente dizer qual deles é mais vantajoso. Porém, em média, considerando todas as instâncias para k entre 4 e 15, o algoritmo ganancioso encontrou a solução ótima em 69, 1% casos, enquanto o KKG encontrou-a em 64, 72% casos. Já os algoritmos KKGC_4 e KKGC_{64}, ambos superam

o algoritmo ganancioso em média, encontrando a solução ótima em $79,16\%$ e $93,54\%$ dos casos, respectivamente. Logo, como esperado, a solução vai melhorando conforme aumentarmos r do algoritmo KKGC_r.

Figura 26: Comparação da eficiência dos algoritmos: ganancioso, KKG, KKGC_4 e KKGC_{64} para os posets de tamanho 100 e k elementos maximais ($4 \leqslant k \leqslant 15$).



A Figura 26 mostra o acerto de cada algoritmo em retornar a solução ótima, dependendo do número de elementos maximais. Mais uma vez podemos observar maior facilidade de encontrar a solução ótima em caso de um número par de elementos maximais.

A última análise que fizemos, foi comparar o desempenho das duas heurísticas, o algoritmo ganancioso e o KKGC_r, ao lidarmos com um poset de tamanho 100, aumentando o número de elementos maximais k de 5 a 95 com incremento de 5 e gerando 100 instâncias para cada k. A Figura 27 apresenta os resultados do teste.

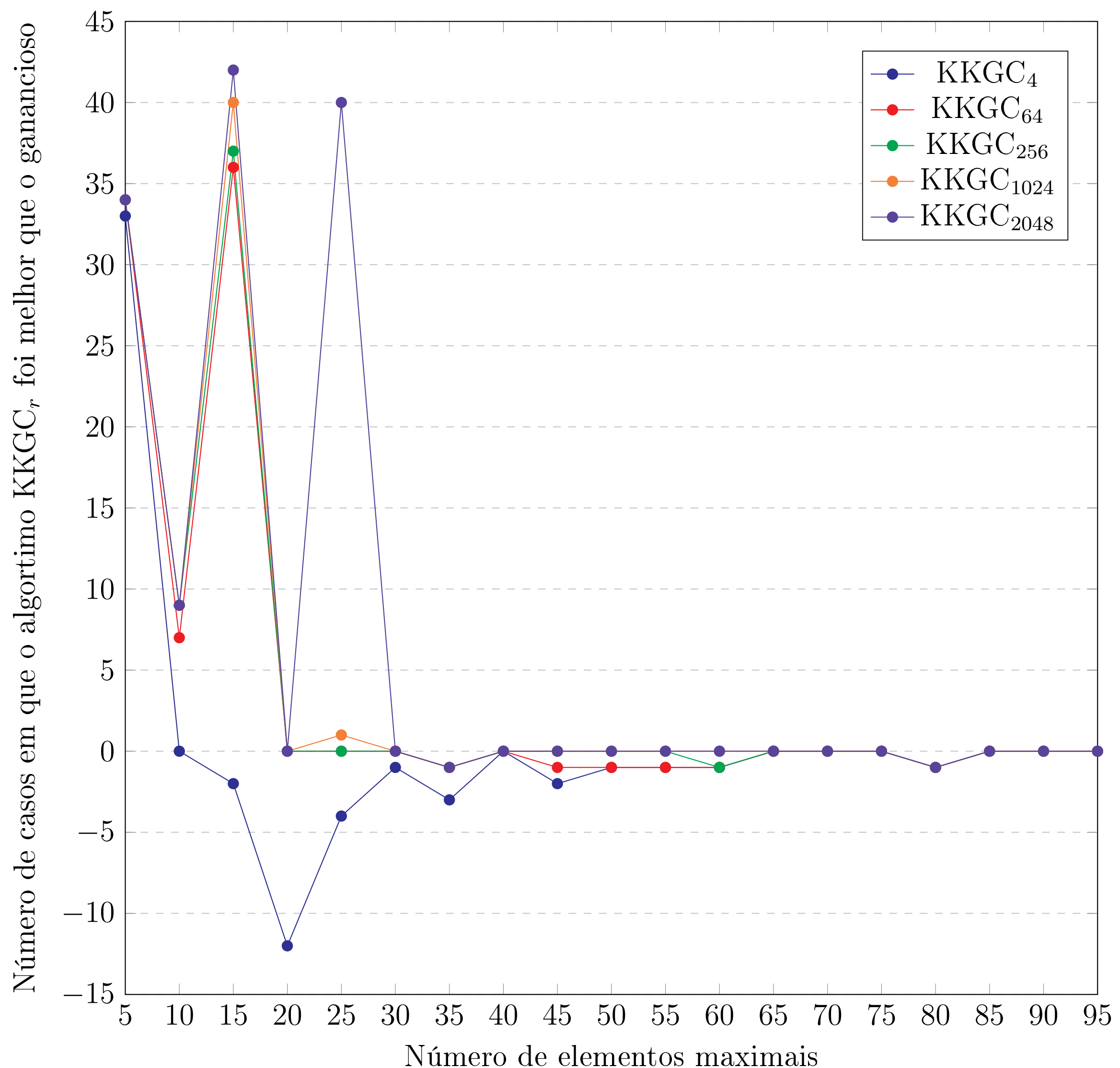
Para cada k, comparamos o valor da discordância em cada instância retornado pelos algoritmos ganancioso e KKGC_r com $r \in \{4, 64, 256, 1024, 2048\}$, ou seja, calculamos

$\gamma_r = \Lambda_{KKGC_r} - \Lambda_{gan}$ e definimos

$$\Gamma_r = \begin{cases} 1 & \text{se } \gamma_r < 0 \qquad \text{(vantagem do algoritmo KKGC}_r\text{)}, \\ 0 & \text{se } \gamma_r = 0, \\ \text{-1} & \text{se } \gamma_r > 0 \qquad \text{(vantagem do algoritmo ganancioso)}. \end{cases}$$

Emfim, somamos os valores de Γ_r levando em consideração todas as instâncias. O valor desta some está representado por eixo vertical. Os números positivos mostram a quantidade média das instâncias em que o KKGC$_r$ obteve melhor resultado, e os números negativos referem se ao melhor desempenho do algoritmo ganancioso.

Figura 27: Comparação da eficiência dos algoritmos ganancioso e KKGC$_r$ para os posets de tamanho 100 e k elementos maximais ($5 \leqslant k \leqslant 95$).



Como esperado, a vantagem do algoritmo KKGC$_r$ cresce conforme aumentamos r. Podemos perceber isto de forma muito clara no caso dos posets com 25 elementos ma-

ximais. O algoritmo $KKGC_4$ foi em média menos eficiênte que o algortimo ganancioso. Já os algoritmos $KKGC_{64}$ e $KKGC_{256}$ retornaram o mesmo valor da discordância que o algoritmo ganancioso, enquanto o $KKGC_{1024}$ apresentou leve vantagem (um caso em 100). No entanto, ao rodarmos o $KKGC_{2048}$, observamos grande crescimento do desempenho ao comparação do algoritmo ganancioso (40% das instâncias). Se quisermos ter alguma melhora no caso de posets com 35 elementos maximais, teríamos que aumentar r ainda mais, para percorrer maior quantidade das folhas.

Como notamos antes, aqui também percebemos que no caso de número par de elementos maximais não temos muita mudança conforme aumentamos r. Novamente isto nós leva a pensar que ambos os algoritmos têm uma facilidade em retornar a solução ótima com muita frequência nestes casos.

5 | Perspectivas

Como vimos no Capítulo 4, a maior parte das instâncias do problema de particionamento de posets são fáceis de serem resolvidas, pois aproximadamente $99,7\%$ dos posets têm no máximo 3 elementos maximais. Com isso, o algoritmo KKGC torna-se inútil nos casos gerais. Portanto, vale a pena investigar e definir as instâncias realmente difíceis, onde o algoritmo de força-bruta não consegue agir, ou seja, os posets têm muitos elementos maximais, e onde a probabilidade do algoritmo ganancioso dar a resposta certa é baixa, aparentemente restrito aos casos em que o número de elementos maximais é ímpar.

Imaginamos também que os poset que possam provocar uma resposta errada do algoritmo ganancioso sejam aqueles em que os pesos dos elementos maximais sejam muito diferentes um do outro, ou então aqueles em que os elementos que dependem do mesmo elemento não se encontrem novamente, isto é, que não exista um elemento que depende destes (formato de uma árvore).

Outro aspecto interessante de experimentar seria aplicar a abordagem LDS de percorrer a árvore (aquela que dá preferência aos ramos esquerdos). Poderíamos repetir os mesmos testes feitos neste trabalho que usaram a DSF na construção da árvore (construção dos nodos da esquerda à direita). Talvez na abordagem LDS, como esta prioriza a operação de diferenciação, as folhas com a solução ótima concentrem-se ainda mais nas primeiras folhas geradas.

Uma questão importante na construção de uma árvore de busca é, sem dúvida, encontrar os critérios de podar os ramos. Como mencionamos no Capítulo 3, um critério foi proposto no trabalho [1]. Seria apropriado estudá-lo e aplicá-lo na nossa implementação. Espera-se que isso diminuiria o número de nodos gerados e, consequentemente, impactaria no tempo de obter a resposta. Talvez, seria possível encontrar outros critérios de poda.

Falando da implementação do algoritmo KKGC em sí, um dos elementos que podem ser melhorados é o jeito de elaborar as operações usadas para construir a árvore de busca. Estas agem nas coordenadas dos vetores, uma por uma. Isto impacta no tempo de execu-

ção do algoritmo quando aumentarmos muito o tamanho do poset (porém o crescimento é linear). Acreditamos que isso pode ser corrigido, para que as operações ajam direto nos vetores, agilizando dessa forma a geração dos nodos.

Por último, seria interessante fazer um estudo sobre a aplicação do algoritmo. Porém, imaginamos que isso será mais fácil depois de definir os casos em quais os algoritmos KKGC_r apresentam muita vantagem sobre o algoritmo de força-bruta e o algoritmo ganancioso, ou seja, definir as instâncias de fato difíceis.

Espera-se que existam umas famílias de posets que representam situações reais, onde os algoritmos KKGC_r possam ser aplicados.

Referências

[1] R. G. L. D'OLIVEIRA and M. FIRER. *Raio de Empacotamento de Códigos Poset*. IMECC-UNICAMP, 2012.

[2] N. KARMARKAR and R. M. KARP. *The differencing method of set partitioning*. Computer Science Division (EECS), University of California, Berkley, Tech. Rep., 1982.

[3] R. E. KORF. *Improved Limited Discrepancy Search*, pages 286–291. Proceedings of AAAI-96, 1996.

[4] R. E. KORF. *A complete anytime algorithm for number partitioning*, volume 106, pages 181–203. Artificial Intelligence, 1998.

[5] J. KUIPERS and G. MOFFA. *Uniform random generation of large acyclic digraphs*. arXiv preprint arXiv:1202.6590, 2012.

[6] E. LEHMAN, F. T. LEIGHTON, and A. R. MEYER. *Mathematics for Computer Science*, chapter 6.1, 7.6, pages 189–192, 226–229. MIT, 2010.

[7] V. LISKOVETS. On the number of maximal vertices of a random acyclic digraph. *Theory of Probability and Its Applications*, (20):401–409, 1976.

Apêndice A

Todos os algoritmos foram implementados em GNU Octave. Para melhor compreensão dos códigos, abordamos a seguinte lógica: todas as funções internas do programa estão em azul, todos os comentários estão em verde, e os nomes das funções que nós implementamos estão de cor laranja.

Código-fonte do algoritmo de busca por força bruta

```
function [S1,S2,dis] = brute_force(P)

  % function [S1,S2,dis] = brute_force(P)
  %
  % brute force algorithm
  %
  % input arguments:
  %     P    - poset
  %
  % output parameters:
  %     S1   - subset 1
  %     S2   - subset 2
  %    dis   - discordancy
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 07/05/2018

  n = length(P); % weight of the poset = number of elelemnts
  max_el = maximal(P); % finds maximal elements
  M_raio = P(:,max_el) % matriz de raio

  subsets = powerset(max_el); % list of all the subsets of maximal elements
  m = length(subsets); % number of all the subsets = 2^nr_max

  d = zeros(m/2,1); % list of discordancy for all pairs of subsets
  for i = 1:(m/2)
    d(i) = discordancyG(P,subsets{i},subsets{m-i+1});
  end
  [dis idx] = min(d);
  % subsets with maximal elements only
  S1 = subsets{idx};
  S2 = subsets{m-idx+1};
  % subsets with all elements
```

```
  S1 = find(any(P(:,S1),2));
  S2 = find(any(P(:,S2),2));
end
```

Código-fonte do algoritmo ganancioso

```
function [S1,S2,dis] = greedy(P)

  % function [S1,S2,dis] = brute_force(P)
  %
  % greedy algorithm
  %
  % input arguments:
  %     P    - poset
  %
  % output parameters:
  %     S1   - subset 1
  %     S2   - subset 2
  %    dis   - discordancy
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 07/05/2018

  n = length(P); % weight of the poset = number of elelemnts
  max = maximal(P); % finds maximal elements
  nr_max = length(max); % number of maximal elements

  weight = zeros(nr_max,1); % vector of weights of maximal elements
  for i = 1:nr_max
    weight(i) = length(find(P(:,max(i))));
  end

  if (nr_max == 1) % if only one max el. => one option of division
    S1 = [1:n]';
    S2 = [];
    wS1 = n;
    wS2 = 0;
    dis = abs(wS1 - wS2);
  else
    [weight,idx_sort] = sort(weight,'descend');
    max = max(idx_sort); % sort the max elements by weight

    % put the two "haviest" max elements in separate subsets
    S1 = max(1);
    S2 = max(2);
    wS1 = weight(1);
    wS2 = weight(2);

    for i = 1:(nr_max-2) % assigning the rest of max elements
                         % to the "lighter" subset
      if (wS1 <= wS2)
        S1 = union(S1,max(i+2));
        wS1 = length(find(any(P(:,S1),2)));
```

```
      else
        S2 = union(S2,max(i+2));
        wS2 = length(find(any(P(:,S2),2)));
      end
    end
    S1 = union(S1,find(any(P(:,S1),2))); % 1st subset
    S2 = union(S2,find(any(P(:,S2),2))); % 2nd subset
    dis = abs(wS1 - wS2) + length(intersect(S1,S2)); % discordancy
end
```

Código-fonte do algoritmo KKGC

```
function [idx,dis] = cgkk(P,r)

  % function [idx,dis] = cgkk(P,r)
  %
  % complete generalized karmarkar-karp algorithm
  %
  % input arguments:
  %     P    - poset
  %     r    - desired number of terminal nodes
  %
  % output parameters:
  %    idx   - number of the leaf with best discordancy
  %    dis   - discordancy
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 15/05/2018

  global node nr;

  n = length(P);
  m = maximal(P);

  R = P(:,m); % radius matrix

  if (length(m) == 1) % case there is only one max element
    dis = sum(R); % no imaginary unit, so
                  % discordancy = sum of the elements of the vector
    idx = 1;
  else
    [idx,dis] = build_tree(R,r); % finding the idx and the discordancy
                                 % of the node representing the optimal solution
  end
end
```

Construção da árvore de busca

```
function [idx,d] = build_tree(R,r)

  % function v = build_tree(R)
  %
```

```
% builds the searching tree
%
% input arguments:
%    R  - radius matrix
%    r  - desired number of terminal nodes
%
% output parameters:
%   idx - number that defines the optimal node
%
% (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
% 24/06/2018

global node nr;

[m n] = size(R); % n - number of maximal elements

% we know for sure that there is more than one maximal element

% number-matrix form
node = cell([n 1]); % the list of nodes
nr = cell([n 1]); % the list of indexes of i

nr{1,1} = []; % first node in a number-matrix form
              % with nr empty because no operation was done so far
node{1,1} = R;

which_nodes = zeros(n,1); % # levels of nodes = # maximal elements

% tic; % possible time limit
% builds nodes, in the end builds the first terminal node
for i = 2:n
  build_node(m,i,1);
end

d = abs(sum(node{n,1})) + length(nr{n,1});
idx = 1;

if ((d == 0) || (d == 1))

else
  j = 2;
  p = 2^(n-1);
  if (r == 0)
    r = p; % total number of terminal nodes
  end
  % while ( (j <= r) && (toc < 3600) && (j <= p)) % with time limit
  while ( (j <= r) && (j <= p) ) % without time limit
    % checks which nodes to build to obtain the j-th terminal node
    which_nodes(n) = j;
    for i = n-1:-1:1
      which_nodes(i) = ceil(which_nodes(i+1)/2);
    end

    % builds nodes (if empty) to in the end build the j-th terminal node
```

```
      for i = 1:n

        % catch an error in case the node we're checking doesn't exist yet
        try
          check = isempty(node{i,which_nodes(i)});
        catch
          check = 1;
        end

        % build the node if it's empty or doesn't exist
        if ( check )
          build_node(m,i,which_nodes(i));
        end
      end

      % computing the discordancy of a terminal node
      dis = abs(sum(node{n,j})) + length(nr{n,j});

      if (dis < d)
        d = dis;
        idx = j;
        if ((d == 0) || (d == 1))
          break;
        end
      end

      j++;
    endwhile
  end
end
```

Discordância de uma folha

```
function dis = discordancy(x)

  % function dis = discordancy(x)
  %
  % calculates the discordancy of a vector
  %
  % input arguments:
  %     x   -   vector
  %
  % output parameters:
  %    dis  -   number value
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 15/05/2017

  sum = sum(x);

  dis = abs(real(sum)) + imag(sum);
end
```

Aplicação do critério PLDM

```
function A = pldm(A)

  % function A = pldm(A)
  %
  % applies the pldm criterion to a matrix, i.e. switches the columns of a matrix:
  % on the first column puts the vector/column v that maximizes discordancy(v),
  % on the second column puts the vector/column W that minimizes discordancy(minus(v,w))
  %
  % input arguments:
  %    A - matrix
  %
  % output parameters:
  %    A - matrix
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  [m n] = size(A);

  if n == 1
    break;
  else

    d = zeros(n,1);
    for j = 1:n
      d(j) = discordancy(A(:,j));
    end

    [value, k] = max(d);
    h = A(:,1);
    A(:,1) = A(:,k);
    A(:,k) = h;

    d(1) = m;
    for j = 2:n
        d(j) = discordancy(minus_gkk(A(:,1),A(:,j)));
      end
    end

    [value, k] = min(d);
    h = A(:,2);
    A(:,2) = A(:,k);
    A(:,k) = h;
end
```

Construção de um nodo

```
function [] = build_node(size,m,n)

  % function = bulid_node(m)
  %
```

```
  % bulild a certain node from a left or a right branch
  %
  % input arguments:
  %  size - size of the poset
  %    m  - level of a node in the list
  %    n  - position of a node in the list
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 19/04/2018

  global node nr;

  s = ceil(n/2);

  if ( mod(n,2) == 0 )
    % builds right branch
    [nr{m,n}, node{m,n}] = nr_matrix(size,nr{m-1,s},branchR(pldm(node{m-1,s})));
  else
    % builds left branch
    [nr{m,n}, node{m,n}] = nr_matrix(size,nr{m-1,s},branchL(pldm(node{m-1,s})));
  end
end
```

Construção do ramo esquerdo

```
function B = branchL(A)

  % function B = branchL(A)
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  [m n] = size(A);
  B = zeros(m,n-1);
  B(:,1) = minus_gkk(A(:,1),A(:,2));
  B(:,2:n-1) = A(:,3:n);
end
```

Construção do ramo direito

```
function B = branchR(A)

  % function B = branchR(A)
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  [m n] = size(A);
  B = zeros(m,n-1);
  B(:,1) = plus_gkk(A(:,1),A(:,2));
  B(:,2:n-1) = A(:,3:n);
end
```

Operação de diferenciação

```
function z = minus_gkk(x,y)

  % function z = minus(x,y)
  %
  % operation of differentiation on vectors coordinate by coordinate
  %
  % input arguments:
  %  x, y - vectors
  %
  % output parameters:
  %    z  - vector
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  n = length(x);
  z = zeros(n,1);

  for j = 1:n
    z(j) = differ(x(j),y(j));
  end
end
```

```
function c = differ(a,b)

  % function c = diff(a,b)
  %
  % operation of differentiation
  %
  % input arguments:
  %  a, b - elements of {0, 1, -1, i}
  %
  % output parameters:
  %    c  - element of {0, 1, -1, i}
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  c = i;

  if ( a != i )
    switch b
      case 0
        c = a;
      case 1
        if ( a != 1 )
          c = -1;
        end
      case -1
        if ( a != -1 )
          c = 1;
        end
```

```
    end
  end
end
```

Operação de associação

```
function z = plus_gkk(x,y)

  % function z = plus(x,y)
  %
  % operation of addition on vectors coordinate by coordinate
  %
  % input arguments:
  %  x, y - vectors
  %
  % output parameters:
  %    z  - vector
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  n = length(x);
  z = zeros(n,1);

  for j = 1:n
    z(j) = add(x(j),y(j));
  end
end
```

```
function c = add(a,b)

  % function c = add(a,b)
  %
  % operation of addition
  %
  % input arguments:
  %  a, b - elements of {0, 1, -1, i}
  %
  % output parameters:
  %    c  - element of {0, 1, -1, i}
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 09/10/2017

  c = i;

  if ( b != i )
    switch a
      case 0
        c = b;
      case 1
        if ( b != -1 )
          c = 1;
        end
```

```
      case -1
        if ( b != 1 )
          c = -1;
        end
    end
  end
end
```

Transformação de um nodo na forma de número-matriz

```
function [idx_new,A] = nr_matrix(m,nr_idx,A)

  % function [nr_idx,A] = nr_matrix(nr_idx,A)
  %
  % converts a matrix into a into number-matrix form, i.e.
  % removes the rows that contains i and saves their positions
  %
  % input arguments:
  %    m     - size of the poset
  %  nr_idx - vector of indexes of i removed
  %    A     - matrix
  %
  % output parameters:
  %  nr_idx - vector of indexes of  i removed
  %    A     - matrix
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 15/05/2018

  idx = find( A(:,1) == i ); % indexes with i

  idx_new = zeros(m,1);
  idx_new(nr_idx) = nr_idx; % puts old indexes on right place

  x = find(idx_new == 0); % finds empty places
  idx_new(x(idx)) = x(idx); % puts new indexes on right places

  idx_new = idx_new(idx_new ~= 0); % removes empty entries

  A(idx,:) = []; % removes rows where we found i
end
```

Códigos-fonte das funções auxiliares

Função que calcula a disordância usando a definição

```
function d = discordancyG(P,A,B)

  % function d = discordancyG(A,B)
  %
  % general formula for discordancy
  %
```

```
  % input arguments:
  %    P    - poset (adjacency matrix)
  %    A    - subset 1 (list of elements)
  %    B    - subset 2 (list of elements)
  %
  % output parameters:
  %    d   - value of discordancy
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 07/05/2018

  el_A = find(any(P(:,A),2)); % elements of the ideal generated by A
  el_B = find(any(P(:,/b),2)); % elements of the ideal generated by B
  discrepancy = abs(length(el_A) - length(el_B)); % value of discrepancy
  x = length(intersect(el_A,el_B)); % number of elements in common
  d = discrepancy(P,A,B) + x; % discordancy
end
```

Função que retorna os elementos maximais de um poset

```
function [m] = maximal(poset)

  % function [m] = maximal(poset)
  %
  % finds maximal elements of a poset
  %
  % input arguments:
  %  poset - adjacency matrix of a poset
  %
  % output parameters:
  %    m    - vector with a list of maximal elements
  %
  % (c) Martyna Joanna Rucinska Pereira <ra190651@ime.unicamp.br>
  % 04/10/2017

  n = length(poset);
  sum = zeros(n,1);
  for j = 1:n
    sum(1:j) = sum(1:j) + poset(1:j,j);
  end

  m = find(sum==1); % index of maximal elements
end
```