FUNDAÇÃO GETULIO VARGAS

ESCOLA DE MATEMÁTICA APLICADA

MESTRADO 2015.1

ESTRUTURA DE DADOS E SEUS ALGORITMOS

Prof Alexandre Rademaker

Apresentação e discussão dos exercícios selecionados

ALUNOS:

KIZZY TERRA

OTTO TAVARES

RIO DE JANEIRO

JUNHO DE 2015

1 Introdução

Este relatório reúne comentários e implementações dos exercícios selecionados, cujos enunciados foram previamente apresentados em sala de aula. O objetivo deste documento é, acima de tudo, apresentar as interessantes discussões que emergiram a medida que os exercícios foram sendo analisados e solucionados, ultrapassando a mera exposição da resposta final encontrada para cada questão escolhida.

Cabe ressaltar que alguns dos exercícios selecionados foram utilizados apenas como ponto de partida para discussões mais enriquecedoras a fim de proporcionar maior aprendizado e, possibilitando que os conhecimentos adquiridos ao longo do curso fossem, de fato, colocados em prática.

2 Exercícios selecionados

2.1 Exercício Selecionado 1

Este exercício foi extraído do livro *Algorithms* [1], capítulo 6, sendo o exercício 6.1. O enunciado deste exercício refere-se a um conhecido problema de programação dinâmica: subsequência de soma máxima. Este problema consiste em encontrar a subsequência de números com maior soma dentro de uma lista unidimensional de números dada.

Este problema possui duas principais variações unidimensionais: na primeira, consideram-se apenas subsequências contíguas da lista de números; na segunda, por sua vez, as subsequências podem envolver números que não são contíguos.

O problema da subsequência contígua de soma máxima foi proposto pela primeira vez em 1977 por Ulf Grenander da *Brown University,* como um modelo simplificado do problema de estimativa de padrões por máxima verossimilhança em imagem digitalizadas.

O problema original do estimador de máxima verossimilhança é na verdade um problema de subsequência de soma máxima bidimensional, mas foi proposto por Grenander em sua forma simplificada (unidimensional) visto que o problema bidimensional exigia muito mais tempo para ser resolvido. Uma solução de tempo linear para o problema unidimensional foi encontrada pouco tempo depois que o problema foi proposto por Jay Kadane da *Carnegie-Mellon University.*[2]

2.1.1 A primeira versão

Problema: Subsequência contígua de soma máxima

Entrada: Uma lista de números a_1, a_2, a_3, ... , a_n

Saída: A subsequência contígua de soma máxima (uma subsequência de tamanho zero possui soma zero)

Discussão

A solução mais ingênua que se pode dar para esta questão consiste de um algoritmo força bruta que calcula todas as possíveis combinações de subsequências, calcula a soma para cada uma delas e retorna a subsequência correspondente a maior soma encontrada. A seguir é apresentada a implementação desta abordagem ingênua em Python.

Implementação

Descrição das variáveis utilizadas

L: lista de números (entrada)

sizeL: tamanho da entrada L

max_sum: armazena o valor da soma máxima. É inicializada com o valor do primeiro elemento da lista de números e é atualizada, durante a execução, a medida que são encontradas somas maiores

max_subsequence: armazena os elementos que compõe a subsequência de soma máxima

current_sum: armazena o valor da soma para cada iteração de i (laço *for* intermediário)

```
def cubic(L):
    sizeL = len(L)
    if sizeL == 0:
        return L
    max_sum = L[0]
    max_subsequence = L[:1]
    for j in range(1, sizeL):
        for i in range(j):
            current_sum = 0
            for k in range(i,j):
                current_sum += L[k]
            if current_sum > max_sum:
                max_sum = current_sum
                max_subsequence = L[i:j]
    return max_subsequence
```

O método implementado possui três laços *for* encadeados e cada um deles irá iterar no máximo N vezes, em que N é o número de elementos da lista de números dada como entrada, portanto a complexidade deste algoritmo é $O(n^3)$.

Solução quadrática

A solução cúbica apresentada é claramente ineficiente e por essa razão, basta refletir um pouco para perceber que podemos modificá-la facilmente e torná-la ligeiramente melhor. Isto pode ser feito com uma simples remoção do laço *for* mais interno, como segue:

```
def quadratic(L):
    sizeL = len(L)
    if sizeL == 0:
        return L
    max_sum = L[0]
    max_subsequence = L[:1]
    for j in range(sizeL):
        current_sum = 0
        for i in range(j, sizeL):
            current_sum += L[i]
            if current_sum > max_sum:
                max_sum = current_sum
                max_subsequence = L[i:j+1]
    return max_subsequence
```

Esta implementação possui dois laços *for* encadeados e cada um será executado N vezes, no pior caso, logo a complexidade do algoritmo apresentado é $O(n^2)$.

Embora o algoritmo quadrático seja mais eficiente do que o algoritmo cúbico, é possível propor soluções ainda melhores. A seguir será discutida uma solução linear similiar à solução proposta por Kadane na ocasião em que o problema foi proposto.

A solução linear

A idéia para esta implementação é utilizar a técnica de programação dinâmica, isto é, construir a solução de um problema através da solução de subproblemas associados.

Neste contexto, a principal idéia é considerar que para cada elemento I da lista de números a solução procurada (subsequência contígua de soma máxima) pode ser encontrada a partir da solução calculada para o elemento anterior *I-1* (se

existir), beneficiando-se do fato de que duas somas parciais vizinhas computadas ao longo da execução diferem por apenas um elemento.

Implementação

Descrição das variáveis utilizadas

startSoFar: armazena o índice do elemento que inicia a subsequência de maior soma

startPrevious: armazena o índice do elemento que inicia a subsequência de maior soma encontrada na iteração anterior

endSoFar: armazena o índice do elemento que encerra a subsequência de maior soma

maxSoFar: armazena o valor da subsoma máxima. É inicializada com o valor do primeiro elemento da lista de números e é atualizada, durante a execução, a medida que são encontradas somas maiores

maxPrevious: armazena o valor da subsoma máxima encontrada na iteração anterior. É inicializada com o valor do primeiro elemento da lista de números e é atualizada, durante a execução, a medida que são encontradas somas maiores

O algoritmo inicia-se ao definir as variáveis, conforme descrito, e em seguida implementa um laço *for* para calcular a solução do problema para cada elemento I (cada iteração calcula a resposta para o problema como se a lista de números terminasse no elemento I), para isso verifica-se qual subsequência possui maior soma: a subsequência composta por todos os elementos até I ($S[I-1] \cup L[I]$) ou a subsequência que se inicia em I ($S[I] = \{L[I]\}$).

Uma vez encontrada a melhor solução para a iteração as variáveis auxiliares são devidamente atualizadas: caso a subsoma encotrada ao final de uma determinada iteração (*maxPrevious*) seja maior que a subsoma máxima encontrada em todas as iterações anteriores (*maxSoFar*), dai as variáveis *maxSoFar*, *startSoFar* e *endSoFar* são modificadas.

```
if maxPrevious + L[i] >= L[i]:
   maxPrevious = maxPrevious + L[i]
else:
   startPrevious = i
   maxPrevious = L[i]
if maxPrevious >= maxSoFar:
   maxSoFar = maxPrevious
   endSoFar = i
   startSoFar = startPrevious
```

Ao final de todas as iterações a subsequência de soma máxima é retornada. O método completo é apresentado a seguir:

```
def linear(L):
    startSoFar = 0
    startPrevious = 0
    endSoFar = 0
    maxSoFar = L[0]
    maxPrevious = L[0]
    for i in range(1,len(L)):
        if maxPrevious + L[i] >= L[i]:
            maxPrevious = maxPrevious + L[i]
        else:
            startPrevious = i
            maxPrevious = L[i]
        if maxPrevious >= maxSoFar:
            maxSoFar = maxPrevious
            endSoFar = i
            startSoFar = startPrevious
    return L[startSoFar:endSoFar+1]
```

No método apresentado, o laço *for* é o único bloco da implementação que depende do tamanho da entrada e este laço será executado N vezes, e, para cada iteração, irá executar um número constante de operações O(1). Portanto, a complexidade desta solução é $O(n)$.

2.1.2 A segunda versão

Problema: Subsequência de soma máxima

Entrada: Uma lista de números a_1, a_2, a_3, ... , a_n
Saída: A subsequência (não necessariamente contígua) de maior soma

Discussão

Para esta versão do problema, a qual os números que compõe a sequência a ser retornada podem ser não-adjacentes, as soluções apresentadas na seção anterior não podem ser utilizadas, visto que em tais implementações, memorizava-se a solução parcial encontrada a cada iteração para que fosse utilizada no cálculo da solução parcial da iteração seguinte.

Entretanto, para que se possa encontrar a solução correta para este caso, é necessário calcular a solução parcial em uma determinada iteração em relação a todas as iterações anteriores, e não apenas em relação a solução parcial imediatamente anteiror como fazia-se na seção 2.1.1. A seguir será apresentada uma solução com complexidade de tempo quadrática no tamanho da entrada.

Uma solução quadrática

Esta implementação utiliza a técnica de programação dinâmica através do armazenamento de resultados intermediários calculados ao longo da execução. A idéia deste algoritmo consiste em calcular a solução para uma determinada iteração levando-se em consideração todos os resultados previamente calculados de modo que a subsequência de soma máxima encontrada para a iteração será:

$$subseqMax(i) = max\{max\{subseqMax(k) \cup elemento(i);\, k < i\},\ elemento(i)\}$$

Isto é, a subsequência de soma máxima para uma dada iteração I é calculada comparando as somas das subsequências encontradas nas iterações anteriores adicionadas do elemento I, com o próprio elemento I.

Estas comparações executadas a cada iteração tornam esta solução quadrática, uma vez que o número de comparações por iteração é (I-1) e são realizadas N iterações no total, onde N é o tamanho da entrada, logo a complexidade de tempo do algoritmo é $O(n^2)$. Nesta solução, a complexidade de espaço também depende do tamanho da entrada, uma vez que são armazenados os resultados intermediários obtidos para as N iterações, tornando a complexidade espacial $O(n)$.

Implementação

Descrição das variáveis utilizadas

L : lista de números (entrada)

sizeL : tamanho da entrada

maxSoFar_index: armazena o índice do último elemento da subsequência de soma máxima. É inicializado com zero, sendo atualizado a medida que subsequências com somas maiores são encontradas

maxSum: array de tamanho N que armazena em cada posição k o valor da soma da subsquência de soma máxima encontrada para a posição k. É inicializado com os elementos da entrada L

subsequence: array de tamanho N utlilizado para recuperar os elementos que compõe a subsequência de soma máxima. Cada posição corresponde a um elemento i e armazena o índice do elemento que o antecede na subsequência de soma máxima. Os elementos deste array são inicializados com valor -1.

max_subsequence: armazena os elementos que compõe a subsequência de soma máxima

A algortimo inicia-se com ao definir as variáveis, conforme descrito, e em seguida implementa dois laços *for* encadeados para encontrar os resultados inter-

mediários para cada um dos elementos da lista, segundo a equação apresentada acima.

```
    for i in range(1,sizeL):
        for j in range(i):
            if maxSum[j] + L[i] >= maxSum[i]:
                subsequence[i] = j
                maxSum[i] = maxSum[j] + L[i]
        if maxSum[i] >= maxSum[maxSoFar_index]:
            maxSoFar_index = i
```

Ao final de todas as iterações recupera-se a subsequência de soma máxima utilizando-se o array *subsequence* e é armazenada no array *max_subsequence* para ser retornada.

```
k = maxSoFar_index
while k != -1 :
    max_subsequence.append(L[k])
    k = subsequence[k]
```

O método completo é apresentado a seguir:

```
def quadratic(L):
    sizeL = len(L)
    if sizeL < 1:
        return L
    maxSoFar_index = 0
    maxSum = []
    subsequence = []
    for i in range(sizeL):
        maxSum.append(L[i])
    for i in range(sizeL):
        subsequence.append(-1)
    for i in range(1,sizeL):
        for j in range(i):
            if maxSum[j] + L[i] >= maxSum[i]:
                subsequence[i] = j
                maxSum[i] = maxSum[j] + L[i]
        if maxSum[i] >= maxSum[maxSoFar_index]:
            maxSoFar_index = i
    k = maxSoFar_index
    max_subsequence=[]
    while k != -1 :
        max_subsequence.append(L[k])
        k = subsequence[k]
     max_subsequence.reverse()
    return max_subsequence
```

2.1.3 Análise dos tempos de execução

Nesta seção apresentamos uma comparação dos tempos de execução obtidos para as diferentes implementações apresentadas para diferentes tamanhos de entrada. Os algoritmos foram implementados na linguagem de programação Python. A execução foi realizada em uma máquina com processador 1,3 GHz Intel Core i5, memória de 4 GB 1600 MHz DDR3 e sistema Operacional: OS X Yosemite (10.10.3).

Subsequências Contíguas

Tamanho da entrada	Algoritmo cúbico	Algoritmo Quadrático	Algoritmo linear
N	$O(N^3)$	$O(N^2)$	$O(N)$
10^1	0.014329 segundos	0.014370 segundos	0.014080 segundos
10^2	0.045706 segundos	0.017277 segundos	0.017037 segundos
10^3	14.5 segundos	0.089748 segundos	0.046395 segundos
10^4	3.5 horas	7.71 segundos	0.095831 segundos
10^5	—	13.85 minutos	0,.81422 segundos

Tabela 1: Tempos de execução

Subsequências Não-Contíguas

Tamanho da entrada	Algoritmo Quadrático
N	$O(N^2)$
10^1	0.014250 segundos
10^2	0.018624 segundos
10^3	0.147 segundos
10^4	13.9 segundos
10^5	24.567 minutos

Tabela 2: Tempos de execução

2.2 Exercício Selecionado 2

O Exercício a seguir foi extraído do livro *Algorithm Design* [3], capítulo 4, que aborda o tema de algoritmos gulosos, sendo o exercício de número 7. Sua escolha foi pautada pelo interesse em enunciar e provar a eficiência de um algoritmo guloso.

Dado um problema, um algoritmo guloso tem como principal característica construir uma solução em pequenos passos, de modo a escolher uma ação a cada passo sob um determinado critério, com o objetivo de otimizar a solução final. Dessa forma, ao final da rotina (gulosa), deve-se retornar como saída uma solução ótima. É possível identificar diferentes critérios para resolução de um

problema, por intermédio de um algoritmo guloso, porém, não necessariamente, o critério escolhido nos leva à solução ótima. Por essa razão, após construir um algoritmo guloso, se faz necessário provar que a solução proposta por tal algoritmo é ótima, sendo esse o nosso objetivo com esse exercício.

2.2.1 Enunciado

Existe um conjunto de N trabalhos diferentes $J_1, J_2, ..., J_n$ que devem ser executados por N PCs. Porém, antes de poder ser executado em algum dos PCs, os trabalhos devem ser pré-processados em um único supercomputador, o qual só pode pré-processar um trabalho de cada vez. Cada um dos trabalhos J_i leva um tempo P_i para ser finalizado pelo supercomputador. Após este tempo P_i, cada job J_i é encaminhado para um PC, o qual leva um tempo f_i para finalizar o trabalho.

Pergunta-se: Qual a melhor forma de alocar os trabalhos dos supercomputadores e dos PCs de modo a minimizar o tempo total necessário pra finalizar o proceassamentos de todos os trabalhos?

2.2.2 Prova da Otimalidade

A primeira etapa será analisar os jobs do supercomputador. Como já dito no enunciado, P_i será o valor da duração dos jobs do supercomputador e F_i o valor duração dos jobs dos PCs.

Uma vez que todos os jobs de um PC iniciam necessariamente após o job de um supercomputador, e como os jobs do supercomputador devem ser ordenados de modo que nao haja sobreposição entre seus valores, podemos inferir que a ordem dos jobs do supercomputador é indiferente, pois dadas as hipóteses listadas, o que de fato irá interferir na otimalidade da soma dos tempos dos jobs em questão, será a ordem dos trabalhos dos PCs.

Com efeito, nosso algoritmo A, exposto em código python na próxima seção, ordena a duração dos jobs feitos pelos PCs em ordem decrescente. Em resumo, nosso critério será escolher a cada passo o job de um PC de maior valor. Através desse algoritmo, vamos ser capazes de alocar os jobs dos supercomputadores e dos PCs de modo a gastar o menor tempo possível nessa tarefa, como será provado a seguir:

Passo Base: Seja k a iteração no tempo, temos nesse passo k = 1.

Seja F_i^* o job de maior duração feito por um PC, supõe-se um algoritmo O que não posiciona o job F_i^* na primeira posição, isto é, após o termino do job do primeiro supercomputador. Isso nos leva à conclusão de que o algoritmo O não ordena os trabalhos dos PCs em ordem decrescente dos valores de duração. Aqui, destaca-se o fato de que para um job de um PC começar o de um su-

percomputador deve terminar e, no passo base, apenas um job P_1 foi concluído por um supercomputador. Seja T o tempo gasto pelo algoritmo A implementado nesse artigo e seja T‘ o tempo gasto pelo algoritmo O, pode-se afirmar que:

$$T = P_1 + F_i^* < P_{-1} + F_i^* = T^{'}$$

Ao considerar P_{-1}como P não um, sabemos que tal job começará necessariamente após P_1. Ou seja, se F_i^* for posicionado em outro lugar que não com P_1 por O, o algoritmo A termina em menor tempo. Por outro lado, se F_i^* é posicionado com P_1 por O, como em A, temos que os dois fazem a mesma escolha ótima T. Assim o algoritmo A mantém a propriedade de "stays ahead"[1] *de algoritmos gulosos.*

Passo Indutivo: Aqui, considera-se que as iterações k =< n são válidas.

A partir da hipótese indutiva podemos dizer que:

$$T = P_{k+1} + F_{i-k}^* < P_{k+j} + F_{i-k}^* = T^{'},\ sendo\ j \geq 1$$

Considera-se F_{i-k}^* como o job de maior duração na iteração k. Esta desigualdade é válida pois P_{n+j} necessariamente começa após P_{n+1} o que faz do tempo em O ser maior do que em A. Se j = 1, então o algoritmo O possui tempo igual ao tempo de A, o que garante novamente a propriedade de "stays ahead". Dessa forma, pode-se concluir que nosso algoritmo é ótimo.

2.2.3 Implementação

A entrada do problema é uma lista de jobs representados por tuplas[2] do tipo (p_i, f_i) em que o primeiro elemento é o tempo que o supercomputador leva para processar o job e o segundo é o tempo de execução que PC escolhido leva para análise de tais dados.

Já a saída é uma lista de jobs -representados por tuplas do tipo (p_i, f_i)- na ordem ótima em que devem ser executados para minimizar o tempo total de execução. Isto posto, vamos analisar as funções criadas para resolução do problema.

É importante destacar que estamos implementando um algoritmo guloso, que já teve sua otimalidade demonstrada na seção anterior. Após a prova de

[1] Método de demonstração, o qual compara-se as soluções parciais que o algoritmo guloso constrói com as soluções parciais propostas por um algoritmo suposto ótimo. O objetivo é mostrar que o algoritmo guloso possui soluções iguais ou melhores, do ponto de vista de otimalidade, frente ao algoritmo suposto ótimo.

[2] Uma tupla é uma estrutura de dados finitos e ordenados. Uma propriedade paritucular da tupla está no fato de ser imutável, o que se mostra útil para análise do presente artigo.

otimalidade do algoritmo, a implementação consiste em uma variação do mergesort, pois estamos interessados na ordem descrescente de elementos que serão expostos na lista de tuplas.

Para tornar mais claro o argumento, vamos expor um exemplo de como será o dado de input. A lista a seguir expõe tuplas com os valores de tempo tomado pelo supercomputador para processamento dos dados na primeira casa, enquanto a segunda casa leva o tempo tomado pelos PCs para análise dos dados em questão.

```
lista = [(24,5),(31,9),(57,22),(39,56),(5,1),(82,43)]
```

A primeira função analisada corresponde àquela responsável por comparar duas tuplas no caso base e é denominada de *compare*. Ela toma os valores dos tempos do PC (i.e. segunda casa) como parâmetro. Essa função tem custo de complexidade O(1).

```
def compare(a, b):
    return a[1] >= b[1]
```

A segunda função analisada é a *merge* modificada. A função *merge*, revisitada pelo problema, expõe como caso passo base o controle de quando umas das listas possui tamanho nulo, retornando a outra lista de input. Até aqui a estrutura do algoritmo é semelhante ao de um *merge* tradicional. Porém, a partir do caso base, o nossa função de comparação faz o argumento recursivo retornar os elementos em ordem inversa, o que nos proporciona ordenar os jobs dos PCs em ordem decrescente, como segue:

```
def merge(lista_rst, lista_nd):
    if len(lista_rst) is 0:
        return(lista_nd)
    elif len(lista_nd) is 0:
        return(lista_rst)
    else:
    #Modificacao para iterar entre tuplas e ordenar em
        ordem decrescente dos valores dos jobs dos PCs
        if compare(lista_rst[0], lista_nd[0]):
            temp = [lista_rst[0]]
            temp.extend(merge(lista_rst[1:],lista_nd))
            return(temp)
        else:
            temp = [lista_nd[0]]
            temp.extend(merge(lista_rst,lista_nd[1:]))
            return(temp)
```

Por fim, a função *mergesort_ inverse* é responsável por receber a lista exposta acima e chamar a função *merge* aplicada ao *mergesort_ inverse* da metade das listas. Esse argumento possui complexidade da ordem de $O(n \log n)$ e retorna os tempos dos jobs dos PCs em ordem decrescente.

```
def mergesort_inverse(lista):
    if len(lista) <= 1:
        return(lista)
    else:
        med = len(lista) // 2
        result=merge(mergesort_inverse(lista[:med]),
            mergesort_inverse(lista[med:]))
        return(result)
```

2.2.4 Conclusão

Destaca-se que esse problema teve sua escolha aclamada por abordar conceitos de Algoritmos Gulosos e de Divisão e Conquista, sendo a prova de otimalidade fundamentada pelo primeiro e a implementação pelo segundo. Dado que devemos minimizar o tempo dos jobs dos computadores, possuímos uma função objetivo que deve ser minimizada e, para isso, utilizamos os conceitos de algoritmos gulosos, de modo a escolher um critério, que será utilizado por passo, minimizando o tempo de processamento em questão.

Ao concluir a prova de otimalidade, é fácil ver que a implementação se resume ao ordenamento dos dados de input, o que nos leva à luz da implementação de um Mergesort modificado de custo $O(n \log n)$ de complexidade.

2.3 Exercício Selecionado 3

Esse exercício foi extraído do livro *Algorithm Design* de *John Kleinberg* e de *Eva Tardos,* capítulo 5, sendo o exercício de número 3 o selecionado.

2.3.1 Enunciado

Supõe-se que esteja fazendo uma consultoria a um banco, que está preocupado em detectar possíveis fraudes, e esse banco surge com o seguinte problema:

O banco possui uma coleção de N cartões, os quais foram confiscados, pois o banco suspeita que tais cartões estejam sendo usados em uma fraude. Cada cartão de crédito é um pequeno pedaço de plástico, contendo uma tira magnética com um dado criptografado. Cada conta pode possuir uma série de cartões de crédito, e vamos dizer que dois cartões são equivalentes se eles correspondem à

mesma conta. É muito difícil ler o número criptografado, mas o banco possui uma tecnologia que faz o seguinte teste:

Toma dois cartões, faz algumas contas e determina se os cartões são equivalentes.

Isto posto, a pergunta é a seguinte: Entre a coleção de N cartões, existe um conjunto maior que N/2 cartões que acusarão equivalência no teste? Considera-se a hipótese de a única operação possível que se possa fazer para testar os cartões seja pegar dois deles, plugar na máquina, e daí verificar a equivalência.

Nossa abordagem no presente artigo será a de resolução do exercício proposto pelo livro, para posteriormente aplicarmos essa resolução a um problema real, evolvendo um dataset com dados de educação superior no Brasil.

2.3.2 Abordagem Linear

Inspirado em um problema real, buscamos resolver o problema exposto no enunciado através de uma solução linear, tomando algumas hipóteses simplificadoras para facilitar a análise. O primeiro passo consiste na estruturação do dado de forma conveninente, para que a partir do dado estruturado sejamos capazes de capturar os valores que apresentam igualdade com custo de complexidade linear.

Após a estruturação do dado vamos explicar como se deu a construção das funções responsáveis por comparação dos dados, contagem dos casos com igualdade, e da função mestre, responsável por chamar todas as demais e verificar sob determinado parâmetro se há fraude. Destaca-se que nossa solução usará o zero como parâmetro ao invés de metade da lista, pois somos capazes de retornar o número exato de fraudes.

Assim caso esse número seja diferente de zero, a função retorna um valor lógico *True,* além de retornar *False* caso contrário. A pertinência do input parâmetro consiste na extensão do algoritmo para casos mais gerais comparados ao exercício em questão. Para resolver o exercício proposto basta-se truncar o parâmetro no valor zero e estar atento apenas ao valor lógico de output. No entanto, em outras aplicações, pode ser pertinente tolerar um certo nível maior do que zero de "fraudes".

2.3.3 Implementação

Com efeito, o algoritmo leva em consideração algumas hipóteses simplificadoras. Para que seja linear e retorne o valor desejado, os dados, aqui representando códigos de cartões de crédito, quando iguais, e em número par, devem estar justapostos em pares, não importando a posição de tais pares. Caso haja um número ímpar de determinado código, deve-se justapor o número par máximo

de códigos em pares e o valor restante deve ser justaposto com outro valor que tenha sobrado de uma situação semelhante, isto é, de um conjunto de códigos iguais ímpares. Caso não haja esse outro valor, deve-se justapor o valor que tenha sobrado com a variável nula.

A seguir se expõe como é feita a construção da lista, atendendo às condições descritas acima.

```
lst = [1201,1201,1308,1308,1201,1403,1308]
lst_pares = []
pare = len(lst) + 1
teste_impar = len(lst) % 2 != 0

if teste_impar:
   pare = len(lst)
   cont = 1

while cont != pare:
   lst_pares.append((lst[cont - 1],lst[cont]))
   cont += 2

if teste_impar:
   lst_pares.append((lst[len(lst) - 1], None))
```

A lista construída para o exemplo ilustrado acima tem o seguinte formato:

```
[(1201, 1201), (1308, 1308), (1201, 1403), (1308,None)]
```

Após a construção da lista, vamos mostrar como se deu a construção da função responsável por comparar os valores dos códigos analisados. A função *compara,* quando chamada, toma dois valores e verifica sua igualdade. Caso haja igualdade ela retorna o valor lógico *True* e retorna *False* caso contrário. Essa função possui custo de complexidade da ordem O(1). Segue o código:

```
def compara(valor1, valor2):
   if valor1 == valor2:
      return(True)
   else:
      return(False)
```

Além da função compara, se faz necessário definir outras duas funções de teste. A primeira seria a função *testa_ igual* e a segunda seria a *testa_ dif.* A construção dessas funções se mostrou necessária, pois dado que possuímos uma lista de tuplas, temos interesse em criar uma lista com tuplas que representem apenas pares iguais, ou que tenham um par entre um código de um cartão e um

valor nulo e, outra lista, que apresente apenas pares com valores diferentes entre si. Dai, de posse dessas funções, basta utilizarmos o argumento linear filter, que será explicado com mais calma a seguir, para que sejamos capazes de criar as listas citadas. É importante destacar que ambas as funções possuem custo de complexidade O(1). Seguem-nas:

```
def testa_igual(lista):
    if lista[0] == lista[1]:
        return(lista[0])
    elif lista[0] is None:
        return(lista[1])
    elif lista[1] is None:
        return(lista[0])

def testa_dif(lista):
    if lista[0] != lista[1]:
        return((lista[0],lista[1]))
```

Nesse passo, vamos descrever a construção da função de maior importância da implementação em questão, a função *contagem*. Será por intermédio dessa função que seremos capazes de contar os códigos que extrapolam o número permitido de igualdades, e assim, retornarmos o número de fraudes.

Dado que o input são as listas, já citadas, dos pares de valores iguais (restante) e a dos pares de valores distindos (deletado),[3]o primeiro passo da função *contagem* decorre de tomar tais listas e desconstruí-las de modo a criar duas novas listas com os valores dos casos em que já se tem pares definidos (restante_lista) ou um valor que tenha sobrado, para o caso de a lista ter tamanho ímpar, e a outra lista ser justamente os valores que não estavam justapostas aos seus pares (deletado_lista).

A partir daí, toma-se cada valor da primeira lista citada(restante_lista) e busca-se na segunda lista citada (deletado_lista) por repetições. Caso uma repetição seja encontrada, conta-se uma fraude e deleta-se o valor encontrado da lista de busca (i.e. deletado_lista). Dessa forma, no primeiro passo tem-se complexidade O(n/2), no segundo O(n/4), e assim por diante, além de O(n-1) para realização dos testes, dando um total de no máximo O(2n) de complexidade, pro pior caso de se percorrer toda a lista duas vezes. Segue o código:

[3]O nome restante se dá pelo fato de o argumento funcional que constrói tal lista, retornar apenas os valores restantes que atendem ao critério de igualdade seus valores, critério esse definido através da construção da função *testa_igual*. Enquanto o nome deletado recorre justamente do fato de que esses são os valores que não estão justpostos em igualdade sendo deletados da primeira lista.

```
def contagem(restante, deletado):
    repeticoes = 0
    restante_lista = []
    deletado_lista = []
    #necessario transformas as listas que estavam em
        tuplas em apenas listas, para manter linearidade.
    for valor_tupla in restante:
        restante_lista.append(valor_tupla[0])
    for valor_tupla in deletado:
        deletado_lista.extend([valor_tupla[0],valor_tupla
            [1]])
    for valor in restante_lista:
        cont = 0
        if valor is None:
            next
        while cont != len(deletado_lista):
            if compara(valor,deletado_lista[cont]) is True
                :
            repeticoes += 1
            deletado_lista.remove(deletado_lista[cont])
            cont -= 1
        cont += 1
    return(repeticoes)
```

Dai, a função *teste* é responsável por receber a lista de tuplas, criada no primeiro passo, e o parâmetro de comparação e, a partir dai, chamar as funções já explicadas, de modo a encontrar a solução desejada, que retornará se há fraude, e o número de fraudes em questão.

Através de um argumento funcional denominado filter, na linguagem python, toma-se a lista de input e utiliza-se o critério da função *testa_ igual* para criar a função das tuplas de códigos de cartões de crédito iguais justapostos, que será a variável restante demandada pela função *contagem.* De forma análoga, o mesmo argumento filter toma a mesma lista e utilza do critério da função *testa_ dif,* para criar a lista das tuplas que não apresentam elementos iguais, lista que será utilizada pela funão contagem no input denominado deletado.

A partir daí, a função contagem devolve o número de fraudes, caso haja, e assim a função *teste* compara com o parâmetro dado, retornando o argumento lógico *True,* caso o extrapole e o argumento *False,* caso contrário. Além disso a função retorna o número de "fraudes" encontradas.

A função criada resolve o problema exposto pelo livro, além de apresentar um arcobouço geral, que porporciona a aplicação a outros tipos de análise. Como será visto a seguir, esse código será alterado, e talvez até simplificado, para

resolução de um problema real, urgido durante a análise de um dataset.

2.3.4 Dataset

Surgiu o desafio de analisar um dataset que apresenta dados de docentes brasileiros em atividade em todos os programas de matemática registrados na CAPES. O desafio consiste em analisar quantos docentes que pesquisam e lecionam no mesmo centro que se titularam. Essa pergunta é pertinente pois existe uma recomendação da CAPES de que a proporção de professores que se titularam e são pesquisadores de um dado programa não seja maior que um determinado parâmetro que varia para cada área de pesquisa. Aqui, vamos considerar este parâmetro igual a dois terços do tamanho total do dataset.

Dessa forma, acionamos uma base de dados que nos apresenta as instituições as quais os docentes se titularam, e os centros que os mesmos trabalharam entre 2004 e 2012. Nosso desafio foi comparar os casos, verificar a igualdade entre eles e reportar se o número de docentes nos programas de matemática no Brasil violam a regra. Lançamos mão de analisar instituicão por instituição por fugir do escopo da análise, isto é, da contextualização entre o exercicio proposto pelo livro e o problema real que nos propusemos a solucionar.

Um dos desafios da base está no fato de que as strings que correspondem às instituições que os docentes se titularam não estão expostas da mesma forma que as strings dos centros que os docentes compõem nos anos da análise. Assim, se fez necessário um tratamento das strings e a criação de uma função responsável por comparações que ao invés de levar um argumento lógico como balizador, levou em consideração a função de Levenshtein, função abordada mais adiante, para verificar a proximidade das strings.

No entanto, comparada à abordagem linear exposta acima, o fato de o dataset apresentar as respectivas instituições em que os professores atuam nos anos de análise já casadas com as instituições de titulação facilita a estruturação do algoritmo. Dito de outra forma, não existem pares desencontrados, ou valores ímpares de características de um mesmo docente, pois cada linha corresponde a um docente diferente e busca-se encontrar o número de semelhanças sobre um determinado critério; por essa razão, a função *contagem* - apresentada na implementação da abordagem linear - será revisitada e simplificada.

Com isso, a análise do dataset, teve como desafio a aplicação do caso abstrato ao caso real, por intermédio do relaxamento de algumas hipóteses lançadas na abordagem linar acima. Segue um recorte do dataset para melhor expor a natureza do problema:

Out[6]:

	Docente_OK	Id_Ano	Nome_da_IES	Titulação_IES
0	ALEXANDRE ANANIN	2004	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
1	ALEXANDRE ANANIN	2005	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
2	ALEXANDER KONSTANTINOVICH KUSHPEL	2005	UNIVERSIDADE ESTADUAL DE CAMPINAS	Moscow University / MOUNIVER
3	ALEXANDRE ANANIN	2006	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
4	ALEXANDER KONSTANTINOVICH KUSHPEL	2006	UNIVERSIDADE ESTADUAL DE CAMPINAS	Moscow University / MOUNIVER
5	ALEXANDER KONSTANTINOVICH KUSHPEL	2007	UNIVERSIDADE ESTADUAL DE CAMPINAS	Moscow University / MOUNIVER
6	ALEXANDRE ANANIN	2007	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
7	ALEXANDRE ANANIN	2008	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
8	ALEXANDER KONSTANTINOVICH KUSHPEL	2008	UNIVERSIDADE ESTADUAL DE CAMPINAS	Moscow University / MOUNIVER
9	ALEXANDRE ANANIN	2009	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
10	ALEXANDER KONSTANTINOVICH KUSHPEL	2009	UNIVERSIDADE ESTADUAL DE CAMPINAS	Moscow University / MOUNIVER
11	ALEXANDRE ANANIN	2010	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
12	ALEXANDRE ANANIN	2011	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
13	ALEXANDRE ANANIN	2012	UNIVERSIDADE ESTADUAL DE CAMPINAS	Instituto de Matematica de Seccao Siberiana da...
14	IGOR MOZOLEVSKI	2004	UNIVERSIDADE FEDERAL DE SANTA CATARINA	Belorussian State University / BSU

Figura 1: Recorte do Dataset

2.3.5 Implementação

Isto posto, buscamos na versão linear do algoritmo de resolução da questão proposta no livro inspiração para resolução de tal problema. Somos capazes de criar uma lista de tuplas, as quais farão uma relação entre a instituição que o docente atua e a instituição a qual ele se tituolou. Uma vez, que essa lista de tuplas é criada, somos capazes de passar por cada tupla, comparando em tempo linear, a igualdade entre as strings. Como já dito acima, essa comparação é feita por intermedio do algoritmo de Levenshtein.

A teoria da distância de Levenshtein entre duas strings, diz respeito ao número de operações necessárias para transformar uma string em outra. Em nossa análise, tomamos como razoável uma distância de Leveinshtein de até dois. Segue o algoritmo:

A primeira parte diz respeito à importação das bibliotecas necessárias para análise. A *xlrd* é utilizada para importar os dados de uma planilha em excel, enquanto a biblioteca *re* é responsável por fazer o tratamento de strings via expressões regulares, enquanto a *Levenshtein* diz respeito à distância homonima descrita acima.

```
import xlrd
import re
import Levenshtein
```

Deixamos a segunda parte da análise para descrever como estruturamos

o dado. Como já descrito, buscamos as construção de uma lista de tuplas, justapondo as instituições que os docentes trabalham e qual eles se titularam respectivamente, de modo que a comparação de custo O(1) seja repetida n vezes, o que nos traz um custo linear para análise de toda base em questão. Assim, segue o exemplo:

```
#Exemplo meramente ilustrativo, pois não está no
    dataset real.
lst = [("UFRJ","UERJ"), ("EMAP","EBAPE"), ("PUC Rio","
    PUC-MG"), ("UFF","FGV"), ("EBEF","EPGE"), ("UFRRJ","
    UFMG")]
```

Se faz necessário mostrar como limpamos os dados na terceira parte da análise. Em primeiro lugar importamos os dados de uma planilha em excel[4], posteriormente necessitamos de expressões regulares para limpar as strings das instituições as quais os docentes se titularam pois essa parte do dado veio em formatação não usual. Por fim, transformamos todas as strings em maiusculas e construimos a lista de tuplas como exposto no exemplo acima. A descrição corresponde ao seguinte código:

```
matematica_wb = xlrd.open_workbook("Matematica.xlsx",
    encoding_override = "utf8" )
matematica_ws = matematica_wb.sheet_by_index(0)
instituicao = matematica_ws.col(4)
titulacao = matematica_ws.col(52)
titulacao_IES = []
cont = 0
for valor in titulacao:
   valor_split = re.split("/",str(valor))
   titulacao_IES.append((str(instituicao[cont]).upper()
       ,valor_split[0].upper()))
   cont += 1

#Iniciamos no indice 1, pois não nos interessamos no
    titulo.
lst_tit_ies = titulacao_IES[1:]
```

A partir de agora, serão expostas as funções necessárias para verificar a semelhança entre as strings, realizar a contagem entre elas, e por fim verificar o número de casos de igualdade. A primeira função a ser abordade será a de comparações, que nesse caso será chamada de *teste_igual*. A função toma

[4]A base de dados é confidencial. Faz parte de um projeto de pesquisa junto à EBAPE sobre métodos de avaliação da qualidade da educação Superior no Brasil.

dois valores, compara a distância de Levenshtein entre eles e retorna o número um caso esse nível de distância seja abaixo de 2 e retorna o número zero caso contrário.

Repara-se que a função *teste_ igual* é uma variação da função *compara* explicada na implementação da abordagem linear. A alteração se faz necessária, pois estamos interessados em contar casos que consideramos próximos, assim, de forma recursiva vamos chamar tal função para comparar e retornar tais valores que proporcionarão a capacidade de contagem.

Além disso, a complexidade dessa função é de O(1), pois toma dois valores e os compara através de um argumento lógico. Segue o código da função em questão.

```
#A variavel valor é uma tupla
def testa_igual(valor):
    distancia_str = Levenshtein.distance(valor[0],valor
        [1])
    if distancia_str <= 2:
        return(1)
    else:
        return(0)
```

Uma vez que a função *teste_ igual* está definida podemos descrever a função *contagem*. A função contagem toma uma lista de tuplas, e itera sobre cada elemento dessa lista de forma a otimizar a contagem do número de valores iguais entre instituições de trabalho e de titulação. Essa simplificação é possível em decorrência da maneira com que os dados dos dataset estão estruturados.

Dessa forma, o algoritmo retorna exatamente o número de casos em que a distância de Levenshtein das strings é menor ou igual a dois para todas os docentes nos anos de análise.

```
def contagem(lista):
    cont = 0
    for lst in lista:
        cont += testa_igual(lst)
    return cont
```

Por fim, expomos a função responsável por receber a lista com os dados e chamar as funções já explicadas anteriormente, para que com os outputs das funções já explicadas, ela possa fazer a comparação entre o tamanho de casos de igualdade o parâmetro de análise.

Como já citado, buscamos comparar a incidência de casos em que há igualdade de docentes com dois terços do número de docentes no pais. Assim, o nosso

parametro será o valor correspondente a dois terços do número de observações totais. Caso seja maior, retornamos o lógico *True,* responsável por designar que mais de dois terços dos professores brasileiros se titularam na mesma insituição que trabalham. Caso a função contagem não extrapole o parametro dado como input, o retorno será o lógico *False.*

```
def verifica(universidades,parametro):
    if contagem(universidades) > parametro:
        return(True,contagem(universidades))
    else:
        return(False,contagem(universidades))
```

2.3.6 Conclusão

Em resumo, busca-se através desse algortimo enfrentar um desafio real de análise de dados de educação superior no Brasil. A resolução em tempo linear do exercício proposto, já explicado acima, serviu de inspiração para criação desse código, que nos levou a inferir que 2105 docentes se titularam e trabalham no mesmo centro.

Destaca-se que a análise é uma aproximação que pode estar levando em consideração repetições ou deixando de tomar casos de interesse, uma vez que as strings não estão expostas no mesmo formato e o dado está em painel. Apesar dos desafios, foi construído um código em tempo de complexidade $O(n)$, que retornou o lógico *False,* pois a base tem tamanho de 9322 observações.

2.4 Exercício Selecionado 4

Este exercício foi extraído do capítulo 8 do livro *Algorithms* [1], sendo o exercício 8.3. Este capítulo discute problemas da classe NP e da classe NP-completo, destacando a redução de problemas NP-completo conhecidos para outros problemas de interesse, com a finalidade de mostrar que os problemas de interesse também são da classe NP-completo. O problema apresentado a seguir é, justamente, um problema de redução e sua escolha foi motivada pelo interesse em buscar um maior entendimento sobre o assunto.

2.4.1 Enunciado

STINGY SAT é o nome dado ao seguinte problema: Dado um conjunto de cláusulas (cada cláusula é uma disjunção de literais) e um inteiro k, encontre uma solução que satisfaça o conjunto de cláusulas com no máximo k variáveis iguais a *true*, se esta solução existir.

Pede-se para provar que *STINGY SAT* é NP-completo.

2.4.2 Discussão

Para provar que um dado problema é NP-completo, podemos mostrar que é possível reduzir um problema conhecido da classe NP-completo para este problema. Sendo assim, neste contexto, basta reduzirmos o problema SAT para o problema STINGY SAT.

Uma breve descrição de SAT

O problema de satisfabilidade booleana (SAT) consiste de determinar se existe uma valoração para as variáveis de uma dada fórmula na forma normal conjuntiva (conjunto de cláusulas que consistem em disjunções de literais) tal que esta valoração torne a fórmula verdadeira, ou seja a valoração satisfaz a fórmula que é dita satisfatível.

A redução SAT→STINGY SAT

A redução do problema de satisfabilidade para o problema *mesquinho* (tradução livre) de satisfabilidade ocorre de maneira simples, uma vez que a entrada do problema de SAT é bastante similar à entrada do problema STINGY SAT, pois as duas entradas diferem apenas pelo inteiro k. O inteiro k será determinado pelo número N de literais presentes na fórmula fornecida como entrada para o problema SAT, de forma que o problema STINGY SAT irá retornar se é possível satisfazer a fórmula dada com no máximo N variáveis verdadeiras.

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee ... \vee a_n) \\ entrada\, de\, SAT \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} (a_1 \vee a_2 \vee ... \vee a_n),\ inteiro\, k = n \\ entrada\, de\, STINGY\ SAT \end{array} \right\}$$

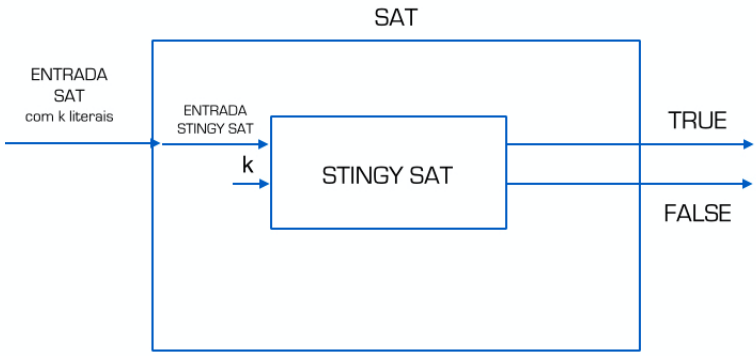

Figura 2: Redução SAT-STINGY SAT

Implementação

Tendo sido provado que o problema STINGY SAT é NP-completo é possível implementar a redução em ambos sentidos: SAT→STINGY SAT e STINGY SAT→SAT. Para testar a redução STINGY SAT→SAT implementada utilizou-se o pacote *pycosat* do Python o qual implementa a resolução do problema SAT. Este pacote representa as cláusulas como conjunto de lista de inteiros que representam os literais da fórmula,tal que:

$$(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C) \, equivale \, a \, [\,[1,2,3],\,[-1,-2,3]\,]$$

Para tornar a construção das fórmulas mais intuitivas, implementou-se uma classe para construção das fórmulas, de forma que o usuário entre com a seguinte fórmula:

```
clause = cnf.AND(cnf.OR("A",cnf.NOT("E"),"D"),cnf.OR(
    cnf.NOT("A"),"E","C","D"),cnf.OR(cnf.NOT("C"),cnf.
    NOT("B")))
```

Esta seja transformada em uma entrada equivalente do pacote *pycosat:*

```
clause =[[1,-5,4],[-1,5,3,4],[-3,-2]]
```

Os métodos implementados são apresentados a seguir e cosistem de um simples parser para a fórmula de entrada. Cada uma das operações (AND, OR e NOT) foi implementada como um método que pode receber um número variável de parâmetros. Os parâmetros são strings (literais) ou um valor de retorno de uma outra operação (operação entre literais). Os métodos compõe uma classe denominada CNF que possui uma atributo do tipo dicionário chamada *variables* que relaciona os literais fornecidos em forma de string com os inteiros a serem passados para o resolvedor do *pycosat;* além disso a classe possui um atributo *clause* que armazena a f'órmula na forma de lista de inteiros (compatível com *pycosat*).

```
class CNF:
   #Inicialização do dicionário de variáveis da fórmula
        e da fórmula
   def __init__(self):
      self.variables = {}
      self.clause = []
   #Cada parâmetro da função AND é uma dijunção de
       literais (lista de literais retornados pela
       função OR)
```

```
def AND(self, *args):
    for arg in args:
        if isinstance(arg, list):
            self.clause.append(arg)
        else:
            return "Invalid CNF format"
    return self.clause
#Cada parâmetro da função OR é uma string que
    corresponde a um literal ou
#um inteiro (retornado pela função NOT) que
    corresponde ao valor negado de um literal
def OR(self, *args):
    clause=[]
    for arg in args:
        #verifica se o valor é um inteiro
        if isinstance(arg, int):
            clause.append(arg)
        else:
            #verifica se o parâmetro é uma string
            if isinstance(arg, basestring):
                #armazena a string se ela não estiver no
                     dicionário
                if not arg in self.variables:
                    self.variables[arg] = len(self.
                        variables)
                clause.append(self.variables[arg])
            else:
                return "Invalid CNF format"
    return clause
#Este parâmetro é uma string que corresponde a um
    literal
def NOT(self,variable):
    #verifica se o parâmetro é uma string
    if isinstance(variable, basestring):
        #armazena a string se ela não estiver no
            dicionário
        if not variable in self.variables:
            self.variables[variable] = len(self.
                variables)
        return -self.variables[variable]
    else:
        return "Invalid CNF format"
#Retorna o número de literais presentes na fórmula
```

```
    def NUMBER_OF_VARIABLES(self):
        return len(self.variables)
```

As reduções implementadas são apresentadas a seguir, a redução de SAT para STINGY SAT foi implementada segundo a equivalência entre as entradas explicada anteriormente. A idéia da redução do problema STINGY SAT para o SAT consiste de analisar as valorações que satisfazem a fórmula fornecida como entrada, retornadas pelo SAT e verificar se para alguma dessas valorações existem no máximo k variáveis verdadeiras. Se houver, então o STINGY SAT deve retornar verdadeiro, se não houver deve retornar falso.

```
#recebe a instância da classe CNF criada para armazenar
     a fórmula de entrada
def reduceSatToStingySAT(cnf):
  return STINGYSAT(cfn.clause, cnf.NUMBER_OF_VARIABLES
      ())
#recebe a instância da classe CNF criada para armazenar
     a fórmula de entrada
def reduceStingySatToSAT(cnf, k):
    solutions = SAT(cnf.clause)
    #Conta o número de variáveis verdadeir para cada
        valoração que satisfaz a fórmula
    for solution in solutions:
        #se encontra uma valoração com k variáveis
            verdadeiras retorna verdadeiro
        if k == countTrueVariables(solution):
            return "TRUE"
    return "FALSE"
```

Finalmente, apresenta-se o método resolvedor de SAT no qual utiliza-se o método *itersolve* do pacote *pycosat,* este método retorna todas as valorações que satisfazem uma determinada fórmula.

```
def SAT(clause):
    return list(pycosat.itersolve(clause))
```

3 Conclusão

Em resumo, esse artigo aborda os diferentes conceitos estudados no curso de Estuturas de Dados e Algoritmos de uma forma não convencional, pois busca-se a contextualização de problemas reais com a abordagem de soluções de problemas clássicos. Além disso, busca discutir problemas não tratáveis de forma

criteriosa, de modo a explorar o tema de algoritmos desde os casos mais simples, até o mais complexos, sem deixar de encarar com relevância a aplicação das resoluções discutidas a possíveis problemas encontrados em situações reais, como no caso do dataset que aborda dados de educação superior no Brasil. Dessa forma, a oportunidade de escolha dos exercícios nos livros citados na referência bibliográfica e a busca da relação entre tais exercícios e a aplicação prática pode ser apontada como força motriz das discussões vistas aqui.

Referências

[1] DASGUPTA, S.; PAPADIMITRIOU, C. H.; VAZIRANI, U. V. Algorithms. 2006.

[2] BENTLEY, J. Programming pearls: Algorithm Design Techniques. *Communications of the ACM*, v. 27, n. 9, p. 865–873, 1984. ISSN 00010782.

[3] KLEINBERG, J.; TARDOS, E. *Algorithm Design*. [S.l.: s.n.], 2005. ISBN 9780321295354.