Universidade Federal de Ouro Preto - *Campus* Morro do Cruzeiro
Departamento de Computação - DECOM
Docente: Anderson Almeida Ferreira
BCC241 - Projeto e Análise de Algoritmos - Turma 11

Trabalho Prático: Clique máximo, Satisfatibilidade (SAT) e Conjunto Independente Máximo

Documentação

LUCAS CHAGAS MOREIRA - 22.1.4109

NICOLAS EXPEDITO LANA MENDES - 22.1.4028

PEDRO MORAIS FERNANDES - 22.1.4020

Ouro Preto, Setembro de 2024

Sumário

Introdução

Este trabalho prático visa a implementação de três programas para solucionar três problemas clássicos: Clique máximo, Satisfatibilidade (SAT) e Conjunto Independente Máximo.

Descrevendo resumidamente cada problema, temos:

- Clique máximo: Dado um grafo previamente fornecido, o objetivo é determinar o maior subconjunto de vértices onde todos os vértices possuem arestas para todos os demais vértices.

- Satisfatibilidade (SAT): Dada uma fórmula booleana na forma normal conjuntiva (CNF), o objetivo é determinar um conjunto de valores-verdade para as variáveis presentes de modo que a fórmula tenha valor-verdade verdadeiro ou, caso não seja possível, informar que não existe tão conjunto.

- Conjunto Independente Máximo: Dado um grafo previamente fornecido, o objetivo é determinar o maior subconjunto de vértices que não possuem nenhuma aresta entre si.

Segundo os requisitos do trabalho, o programa para solucionar o problema do Clique Máximo deve utilizar a técnica de **Branch and Bound**, o programa para problema da Satisfatibilidade (SAT) deve utilizar **Backtracking** e o programa para o problema do Conjunto Independente Máximo deve utilizar **Redução Com Custo Polinomial** para o problema do Clique Máximo, previamente implementado.

Clique máximo

Tendo em vista o problema do clique apresentado acima que é proposto a resolução a partir do algoritmo de *Branch and Bound,* mostra-se a análise do código a seguir:

- Função *Main()*

```python
if __name__ == "__main__":
```

```
        arquivos = ['input/maxcliqueinput/input1.txt',
                'input/maxcliqueinput/input2.txt',
                'input/maxcliqueinput/input3.txt'
                ]

        for arquivo in arquivos:
        # Lê o grafo de um arquivo
        with open(arquivo, 'r') as f:
                n = int(f.readline().strip())
                grafo = [list(map(int, linha.split())) for linha in
f.readlines()]

        # Mede o tempo de execução
        start_time = time.time()
        clique_maximo = maxclique(grafo, n)
        end_time = time.time()

        # Imprime o clique máximo encontrado e o tempo de execução
        print(f"Arquivo: {arquivo}")
        print("Clique máximo:", clique_maximo[0])
        print(f"Tempo de execução: {end_time - start_time:.7f} segundos\n")
```

A função main possui uma lista que contém o caminho dos arquivos de entrada para a amostra do código.Tal processo irá iterar esta lista para que a função **maxclique(grafo, n)** seja chamada para cada um dos inputs acima e assim apresentar a solução para cada um, sendo que seus parâmetros virão a partir dos arquivos de entrada.

Posteriormente, ao final da execução do algoritmo, irá ser imprimido o o resultado do clique máximo e também o tempo de execução.

- Função *maxclique(grafo, n)*

```
Python
def maxclique(grafo, n):
        vertices = list(range(n))

        melhor_clique = [[]]
        melhor_clique.append(melhorInicialGuloso([], vertices, grafo, n))
#Solução melhor inicial
```

```
        branch_and_bound([], vertices, melhor_clique, grafo) #Solução inicial
= []

        return melhor_clique
```

Esta função irá fazer uma lista de vértices a partir da primeira linha de cada arquivo de input que informa o tamanho da instância.Após isso, há a chamada da função **melhorInicialGuloso(clique_atual, vertices_restantes, grafo, n)**, que basicamente irá fazer uma busca gulosa para criar uma solução inicial para a chamada da função **branch_and_bound([], vertices, melhor_clique, grafo)** na linha seguinte, justamente para impedir que o algoritmo comece sem uma solução.

- Função *melhorInicialGuloso(clique_atual, vertices_restantes, grafo, n)*

```
Python
def melhorInicialGuloso(clique_atual, vertices_restantes, grafo, n):
        if(len(vertices_restantes) == 0):
        return clique_atual

        v = vertices_restantes[0] #Melhor solução inicial começará a partir do
primeiro vértice
        novos_vertices_restantes = vertices_com_arestas_com_v(v,
vertices_restantes, grafo)
        clique_atual.append(v)
        melhorSolucaoInicial = melhorInicialGuloso(clique_atual,
novos_vertices_restantes, grafo, n)

        return melhorSolucaoInicial
```

Esta função, a partir do primeiro vértice, irá retornar uma solução inicial que beneficiará o algoritmo de *Branch and Bound*, fornecendo assim um limite para a função, o que consequentemente, ajuda na poda das soluções mais adiante no algoritmo.Ademais, a função irá checar as conexões entre os vértices do grafo recebido na entrada com a função **vertices_com_arestas_com_v(v, vertices, grafo)** e através da recursão da função, irá gerar a solução inicial para o algoritmo de *Branch and Bound*.

- Função *vertices_com_arestas_com_v(v, vertices, grafo)*

```python
def vertices_com_arestas_com_v(v, vertices, grafo):
	vertices_com_arestas_com_v = []
	for w in vertices:
	if grafo[v][w] == 1: #Vértice com aresta com vértice V
		vertices_com_arestas_com_v.append(w)
	return vertices_com_arestas_com_v
```

Tal função irá basicamente ver todas as conexões que um determinado vértice do grafo tem com os outros vértices em questão, retornando uma lista de vértices.

- Função *branch_and_bound(clique_atual, vertices_restantes, melhor_clique, grafo)*

```python
def branch_and_bound(clique_atual, vertices_restantes, melhor_clique,
grafo):
	if eCompleta(vertices_restantes):
	melhor_clique[0] = clique_atual[:]
	return

	for v in vertices_restantes:
	novos_vertices_restantes = vertices_com_arestas_com_v(v,
vertices_restantes, grafo) #Adiciona todos os vértices como promissores que
possuem aresta com o vértice v, garante que todas as soluções serão
consistentes
	clique_atual.append(v) #Adiciona v ao conjunto de clique atual para
eventualmente explorar uma solução que o contém
	if(ePromissora(clique_atual, novos_vertices_restantes,
melhor_clique)): #Poda
		branch_and_bound(clique_atual, novos_vertices_restantes,
melhor_clique, grafo)
	clique_atual.pop() #Remove v do clique atual para explorar uma
eventual solução utilizando o próximo vértice do for
```

A função começa chamando a função **eCompleta(vertices_restantes)**, que verificará se a solução é completa para então assumir a forma do melhor clique ao

final da recursão.À vista disso, o algoritmo segue a sua execução do *Branch and Bound,* para cada vértice do grafo, irá chamar a função **vertices_com_arestas_com_v(v, vertices, grafo)**, que irá checar as conexões do vértice e após isso, tal vértice irá ser posto no clique atual para que eventualmente uma solução seja explorada com este vértice.

Com a função prosseguindo, irá ser chamada a função **ePromissora(clique_atual, vertices_candidatos, melhor_clique)**, que irá verificar se a solução do clique atual é promissora e assim podendo chamar a recursão da função **branch_and_bound** novamente e assim continuar a execução do algoritmo para a formação do clique, caso não seja promissora, tal função irá funcionar como uma poda das soluções e assim , otimizar o algoritmo e não permitir perca de tempo com soluções não promissoras para a solução do problema em questão.

- Função *eCompleta(vertices_restantes)*

```python
def eCompleta(vertices_restantes): #Verifica se a solução é completa, ou seja, se não há mais vértices a serem adicionados ao clique
    if len(vertices_restantes) == 0:
        return True
    return False
```

A função ***eCompleta(vertices_restantes)*** simplesmente verifica se a quantidade de vértices restantes é igual a 0. Caso seja 0, isso significa que não há mais nenhum vértice a ser colocado, e a função retorna **true**; caso contrário, a função retorna **false**.

- Função *ePromissora(clique_atual, novos_vertices_restantes, melhor_clique)*

```python
def ePromissora(clique_atual, vertices_candidatos, melhor_clique): #Verifica se o eventual clique tem potencial de ser melhor que a solução melhor atual
    if (len(clique_atual)+len(vertices_candidatos)) > len(melhor_clique[0]):
        return True
    return False
```

A função ***ePromissora(clique_atual, novos_vertices_restantes, melhor_clique)*** compara a quantidade de vértices que já estão no clique atual,

somada à quantidade de vértices restantes que podem possivelmente entrar no clique, com a quantidade de vértices presente no melhor clique. Caso o clique atual tenha potencial para receber mais vértices do que o melhor clique, ele é considerado promissor, e a função retorna **true**; caso contrário, a função retorna **False**.

Resultados Obtidos

- Output para primeira instância (10 vértices)

```
Unset
Arquivo: input/maxcliqueinput/input1.txt
Clique máximo: [0, 4, 7, 9]
Tempo de execução: 0.0000660 segundos
```

- Output para segunda instância (50 vértices)

```
Unset
Arquivo: input/maxcliqueinput/input2.txt
Clique máximo: [0, 10, 18, 26, 31, 36, 37, 47]
Tempo de execução: 0.1536267 segundos
```

- Output para terceira instância (140 vértices)

```
Unset
Arquivo: input/maxcliqueinput/input3.txt
Clique máximo: [2, 3, 12, 17, 20, 23, 27, 68, 101, 124]
Tempo de execução: 214.6982052 segundos
```

Satisfabilidade

- Função *Main()*

```python
if __name__ == "__main__":
    # Lista de arquivos de entrada
    arquivos = ['input/satinput/input1.txt',
                'input/satinput/input2.txt',
                'input/satinput/input3.txt']

    for nome_arquivo in arquivos:
        clausulas, numero_variaveis = ler_entrada_arquivo(nome_arquivo)
        if clausulas and numero_variaveis:
            resultado = satisfatibilidade(clausulas, numero_variaveis)
            print(f"Resultado para {nome_arquivo}: {resultado}")
        else:
            print(f"Erro ao processar a entrada do arquivo {nome_arquivo}.")
```

A função principal do programa executa um for each passando por todo arquivo da lista arquivos, chamando a função ler_entrada_arquivo() para cada .txt, que representa uma instância para o problema SAT.

Posteriormente, ainda dentro da mesma iteração, a função avalia se as cláusulas e o número de variáveis foram obtidos por meio da leitura do arquivo, o que está sendo feito dentro do if.

Seguidamente, a função chama a função satisfatibilidade(), passando como argumento as cláusulas e o número de variáveis como argumento.

Por fim, caso a função receba as entradas corretamente, a função principal printa o nome do arquivo e o resultado obtido. Caso contrário, printa comunicando a existência de um erro com as entradas do problema.

- Função *ler_entrada_arquivo(nome_arquivo)*

```python
def ler_entrada_arquivo(nome_arquivo):
    try:
```

```
        with open(nome_arquivo, 'r') as f:
            numero_variaveis = int(f.readline().strip())
            clausulas = []
            for linha in f:
                clausula = list(map(int, linha.strip().split()))
                clausulas.append(clausula)
        return clausulas, numero_variaveis
    except FileNotFoundError:
        print(f"Erro: O arquivo '{nome_arquivo}' não foi encontrado.")
        return [], 0
    except Exception as e:
        print(f"Erro ao ler o arquivo: {e}")
        return [], 0
```

A função **ler_entrada_arquivo(nome_arquivo)** recebe como parâmetro o nome_arquivo, que diz respeito ao endereço relativo do arquivo .txt com a instância do problema.

Posteriormente, a função, caso consiga abrir o arquivo corretamente, retornará uma lista de listas, representando as cláusulas, onde em cada lista estará o respectivo literal referente a sua variável (0 [negada], 1[sem negação] ou -1 [não está presente na cláusula]). Além disso, retornará também o número de variáveis presentes na fórmula.

Caso ocorra algum erro, a função executa um print, comunicando qual erro ocorreu.

- Função *satisfatiblidade(clausulas, numero_variaveis)*

```
Python
def satisfatibilidade(clausulas, numero_variaveis):
    variaveis = [None] * numero_variaveis
    start_time = time.time()
    satisfaz = backtrack(clausulas, variaveis, numero_variaveis)
    end_time = time.time()

    if satisfaz:
        return f"Satisfazível. Atribuição: {variaveis}. Tempo de execução:
{end_time - start_time:.7f} segundos"
```

```
    else:
        return f"Não satisfazível. Tempo de execução: {end_time -
start_time:.7f} segundos"
```

A função ***satisfatiblidade(clausulas, numero_variaveis)*** mede o tempo de execução e chama a função principal do programa. Ademais, recebe como parâmetros: **clausulas**, que corresponde a uma lista de lista, representando as cláusulas da fórmula. Seu conteúdo foi anteriormente descrito na função ***ler_entrada_arquivo(nome_arquivo)***. E **numero_variaveis**, que corresponde a quantidade de variáveis presente na fórmula.

Seguidamente, a função cria uma lista, cujas posições contém None para indicar que a variável não tem um valor atribuído. A quantidade de elementos da lista será igual ao valor de **numero_variaveis**.

Depois, a função inicia a contagem do tempo de execução do algoritmo e, posteriormente, chama a função principal **backtrack(clausulas, variaveis, numero_variaveis)**, passando como argumento: **clausulas**, a lista de listas gerada pela função ***ler_entrada_arquivo(nome_arquivo)***, **variaveis**, a lista inicialmente criada para representar os valores-verdade atribuídos a cada variável (False ou True) e o número de variáveis.

Após o término da execução da função **backtrack(clausulas, variaveis, numero_variaveis)**, o programa encerra a contagem do tempo de execução e verifica o valor da variável **satisfaz,** que receberá o retorno da função **backtrack(clausulas, variaveis, numero_variaveis),** que será True, caso a fórmula seja satisfazível, ou False, caso a fórmula não seja satisfazível.

Caso o valor de **satisfaz** seja True, o programa executa um print, informando os valores-verdades para cada variável que satisfazem a fórmula, bem como o tempo de execução da função em segundos, com uma precisão de 7 casas decimais. Caso contrário, o programa executa um print, informando que a fórmula não é satisfazível e o tempo de execução da função com a mesma precisão e unidade de tempo caso fosse satisfazível.

- Função *backtrack(clausulas, variaveis, numero_variaveis)*

```python
def backtrack(clausulas, variaveis, numero_variaveis):
    #Verifica se a solução parcial é consistente, estratégia para evitar
busca exaustiva
    if not is_partial_solution_is_valid(clausulas, variaveis):
        return False

    #Se todas as variáveis estão atribuídas, verifica se as clausulas são
satisfeitas simultaneamente
    if None not in variaveis:
        return verificar_formula(clausulas, variaveis)

    #Próxima variável a ser atribuída
    index_prox = variaveis.index(None)

    for valor in [True, False]:
        #Proximo passo do backtrack
        variaveis[index_prox] = valor
        if backtrack(clausulas, variaveis, numero_variaveis):
            return True
        variaveis[index_prox] = None #Caso a tentativa tenha gerado uma
solução que não satisfaz, retorna o passo anterior

    return False
```

A função ***backtrack(clausulas, variaveis, numero_variaveis)*** realiza o algoritmo de backtracking propriamente dito e recebe como parâmetros: **clausulas**, uma lista de listas, representando as cláusulas da fórmula. Seu conteúdo foi anteriormente descrito na função ***ler_entrada_arquivo(nome_arquivo)***, **variaveis**, a lista criada para representar os valores-verdade atribuídos a cada variável (False ou True) e **numero_variaveis,** que corresponde ao número de variáveis presente na fórmula.

Primeiramente, a função verifica se a solução parcial que está sendo gerada, chamando a função **is_partial_solution_is_valid(clausulas, variaveis)**, passando como parâmetros: **clausulas** e **variaveis**., durante os passos do backtracking, é consistente. Desse modo, evita-se que o backtracking se torne uma busca exaustiva, pois, se uma solução parcial não for consistente, este programa, como está implementado, não seguirá aprofundando os passos do backtracking a partir desta solução, partindo para o ramo oposto da árvore de passos, evitando a geração de mais soluções inconsistentes, identificando isso previamente. Caso a função chamada retorne **False,** significa que a solução parcial não é consistente, logo a função ***backtrack(clausulas, variaveis, numero_variaveis)*** retorna **False** também, fazendo com que o próximo passo do backtrack não gere mais soluções inconsistentes a partir de uma solução sabidamente inconsistente. Caso a função

chamada retorne **True,** significa que a solução parcial é consistente e que o backtrack pode dar um próximo passo a partir dela.

Seguidamente, caso não haja o valor None no vetor **variaveis** que representa as atribuições de valores-verdade para as variáveis, significa que todas as variáveis possuem um valor-verdade atribuído. Desse modo, o programa verifica se tais valores satisfazem a fórmula, chamado a função **verificar_formula(clausulas, variaveis)**, passando como parâmetro **clausulas** e **variaveis**. Esta função retorna **True,** caso os valores-verdade atribuídos satisfaçam a fórmula. Neste caso, a função **b*acktrack(clausulas, variaveis, numero_variaveis)*** também retorna **True,** indicando que encontrou um conjunto de valores-verdade para as variáveis que satisfaz a fórmula. Caso a função de verificação retorne **False,** isso significa que os valores não satisfazem a fórmula.

Depois, o programa obterá o índice da primeira variável no vetor **variaveis** com um valor None, ou seja, uma variável que não possui um valor-verdade atribuído, e armazenará na variável **index_prox.** A partir deste índice, o backtracking irá atribuir valores para essa variável para dar prosseguimento aos próximos passos.

Na sequência, o programa faz um for entre os valores **True** e **False,** que serão atribuídos à variável referente ao **index_prox**. Seguidamente, o valor referente a iteração é atribuído a variável.

Consequentemente, o programa chama recursivamente a função **backtrack(clausulas, variaveis, numero_variaveis),** passando como parâmetros: clausulas, uma lista de listas, representando as cláusulas da fórmula. Seu conteúdo foi anteriormente descrito na função ***ler_entrada_arquivo(nome_arquivo),*** **variaveis**, a lista criada para representar os valores-verdade atribuídos a cada variável (False ou True) e **numero_variaveis,** que corresponde ao número de variáveis presente na fórmula. Tal chamada recursiva ocorre dentro de um if, ou seja, caso a função **backtrack(clausulas, variaveis, numero_variaveis)** retorne **True,** a função própria que a chamou também retorna **True**, sinalizando que um conjunto de valores-verdade para as variáveis que satisfaz a fórmula foi encontrado.

Caso, em algum momento, a função **backtrack(clausulas, variaveis, numero_variaveis),** dentro do condicional do if, retorne **False,** a função irá atribuir None para a variável referente ao **index_prox**, indicando que o backtracking terá que voltar um passo e atribuir um valor diferente, gerando outras soluções.

Por fim, caso a função não retorne **True** em nenhum dos momentos e condições descritos anteriormente, a função retorna **False,** indicando que não há um conjunto de valores-verdade que satisfaz a fórmula.

- Função *is_partial_solution_is_valid(clausulas, variaveis)*

```python
is_partial_solution_is_valid(clausulas, variaveis):
    for clausula in clausulas:
            if -1 in clausula: #Caso haja uma variável (ou mais) não
presentes em uma cláusula
                #Verifica-se se as demais variáveis presentes possuem
valores
                if
ha_atribuicao_para_todas_variaveis_validas_na_clausula(clausula, variaveis):
                    satisfied = False
                    for i, literal in enumerate(clausula):
                        if literal == 1 and variaveis[i]:  # Variável
positiva e verdadeira = True
                            satisfied = True
                            break

                        elif literal == 0 and not variaveis[i]:  # Variável
negada e falsa = True
                            satisfied = True
                            break

                    # Se a cláusula não é satisfeita, significa que não faz
sentido expandir o backtrack a partir de tal solução parcial
                    if not satisfied:
                        return False  # Poda: Se já sabemos que não podemos
satisfazer a cláusula

    return True  # Ainda pode ser possível satisfazer a fórmula
```

A função **is_partial_solution_is_valid(clausulas, variaveis)** verifica se uma solução parcial é consistente recebe como parâmetros: **clausulas**, uma lista de listas, representando as cláusulas da fórmula. Seu conteúdo foi anteriormente descrito na função ***ler_entrada_arquivo(nome_arquivo)***, **variaveis**, a lista criada para representar os valores-verdade atribuídos a cada variável (False ou True).

Inicialmente, a função faz um for por todas as cláusulas e, a cada iteração, ela começa verificando se há um **-1** na cláusula, ou seja, se há alguma variável da fórmula que não está presente na cláusula. Isso se dá pela estratégia que nós decidimos adotar para verificar a consistência de uma solução parcial, evitando que o backtracking se torne uma busca exaustiva.

Seguidamente, caso haja **-1** na fórmula, a função chama a função **ha_atribuicao_para_todas_variaveis_validas_na_clausula(clausula, variaveis),**

passando como parâmetro **clausula** e **variaveis**. Basicamente, essa função verifica se há algum valor atribuído para todas as variáveis presentes em tal cláusula. Caso haja, retorna **True,** caso contrário, retorna **False.**

Na sequência, resumidamente, caso haja valores-verdade definidos para as variáveis presentes na cláusula, verifica-se se os valores definidos satisfazem tal cláusula (dentro de um for síncrono nos literais da cláusula e nas variáveis), pois, nesse caso, não é necessário a definição de valores-verdade para todas variáveis da fórmula. Isso é feito nas linhas subsequentes que verificam se caso o literal seja 1 e o valor da variável correspondente seja **True** ou caso o literal seja 0 (variável negada) e o valor da variável seja **False.** Caso transcorra em uma destas situações, a variável **satisfied** recebe **True**, caso contrário, a variável **satisfied** permanece como **False,** indicando uma solução parcial inconsistente.

Ao final da iteração, verifica-se o valor de **satisfied.** Caso seja **False,** a função retorna **False,** indicando que encontrou uma solução parcial inconsistente. Caso contrário, passa para próxima iteração.

Após percorrer todas as cláusulas e o valor de **satisfied** assumir **True** em todas as iterações, a função retorna o valor **True,** indicando que a solução parcial é consistente.

- Função *verificar_formula(clausulas, variaveis)*

```python
def verificar_formula(clausulas, variaveis): #Verifica se os valores atribuídos às variáveis satisfazem todas as cláusulas simultaneamente
    for clausula in clausulas:
        satisfaz = False
        for i, literal in enumerate(clausula):
            if literal == 1 and variaveis[i]:  # Variável positiva e verdadeira = True
                satisfaz = True
                break #Pula para analisar a próxima clásula
            elif literal == 0 and not variaveis[i]:  # Variável negada e falsa = True
                satisfaz = True
                break #Pula para analisar a próxima clásula

        if not satisfaz: # Se alguma cláusula não for satisfeita, retorna falso
            return False
```

```
    return True  # Todas as cláusulas foram satisfeitas
```

A função **verificar_formula(clausulas, variaveis)** verifica se um conjunto de valores-verdade para as variáveis satisfaz a fórmula e recebe como parâmetros: **clausulas**, uma lista de listas, representando as cláusulas da fórmula. Seu conteúdo foi anteriormente descrito na função ***ler_entrada_arquivo(nome_arquivo)***, **variaveis**, a lista criada para representar os valores-verdade atribuídos a cada variável (False ou True).

Primeiramente, a função faz um for entre as cláusulas e atribui o valor **False** para a variável **satisfaz**. Posteriormente, há um for síncrono entre os literais da cláusula e as variáveis, onde verifica-se se algum literal é satisfeito pelo valor da variável referente, que seria o caso em que o literal seja 1 e o valor da variável correspondente seja **True** ou caso o literal seja 0 (variável negada) e o valor da variável seja **False.** Caso uma dessas condições seja satisfeita, a variável **satisfaz** recebe **True,** caso contrário permanece com **False.**

Ao final da iteração da cláusula, verifica o valor de **satisfaz**, caso seja **False,** a função retorna **False,** informando que o conjunto de valores-verdade para as variáveis não satisfaz a fórmula. Caso contrário, segue para próxima iteração.

Caso o valor de **satisfaz** assuma **True** nas iterações, a função retorna **True,** indicando que o conjunto de valores-verdade satisfaz a fórmula.

- Função *ha_atribuicao_para_todas_variaveis_validas_na_clausula(clausula, variaveis)*

```Python
def ha_atribuicao_para_todas_variaveis_validas_na_clausula(clausula,
variaveis):
    for i, literal in enumerate(clausula):
        if literal != -1 and variaveis[i] == None:
            return False
    return True
```

A função **ha_atribuicao_para_todas_variaveis_validas_na_clausula(clausula, variaveis)** verifica se todas as variáveis presentes em uma cláusula possuem um valor-verdade atribuído e recebe como parâmetros: **clausulas**, uma lista de listas, representando as cláusulas da fórmula. Seu conteúdo foi anteriormente descrito na função ***ler_entrada_arquivo(nome_arquivo)***, **variaveis**, a lista criada para representar os valores-verdade atribuídos a cada variável (False ou True).

Primeiramente, há um for síncrono entre os literais da cláusula e os valores-verdade das variáveis. A cada iteração, caso haja algum literal diferente de **-1,** ou seja, a variável está presente na cláusula, e a respectiva variável não tenha um valor-verdade atribuído, a função retorna **False.**

Caso a função percorra todos os literais de todas as cláusulas sem cair na condição anteriormente descrita, a função retorna **True.**

Resultados Obtidos

- Output da primeira instância (20 variáveis na fórmula)

Unset
```
Resultado para input/satinput/input1.txt: Satisfazível. Atribuição: [True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True]. Tempo de execução: 0.0007083
segundos
```

- Output da segunda instância (70 variáveis na fórmula)

Unset
```
Resultado para input/satinput/input2.txt: Satisfazível. Atribuição: [True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True]. Tempo de execução:
0.0205586 segundos
```

- Output da terceira instância (280 variáveis na fórmula)

```
Unset
Resultado para input/satinput/input3.txt: Satisfazível. Atribuição: [True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True]. Tempo de execução: 0.9531214 segundos
```

Conjunto Independente Máximo

- Função *Main()*

```python
Python
if __name__ == "__main__":
    arquivos = ['input/independentconjinput/input1.txt',
                'input/independentconjinput/input2.txt',
                'input/independentconjinput/input3.txt']
```

```
    for arquivo in arquivos:
        # Lê o grafo de um arquivo
        with open(arquivo, 'r') as f:
            n = int(f.readline().strip())
            grafo = [list(map(int, linha.split())) for linha in
f.readlines()]

        # Mede o tempo de execução
        start_time = time.time()
        conjunto_independente_maximo = conjunto_independente(grafo, n)
        end_time = time.time()

        # Imprime o conjunto independente máximo encontrado e o tempo de
execução
        print(f"Arquivo: {arquivo}")
        print("Conjunto independente máximo:",
conjunto_independente_maximo[0])
        print(f"Tempo de execução: {end_time - start_time:.8f} segundos\n")
```

A “função main” contém o caminho para as pastas que possuem as entradas para o código, em formato de lista.

A função itera sobre a lista, chamando a função ***conjunto_independente(grafo,tamanho_do_grafo)*** para cada entrada.

- Função *conjunto_independente(grafo,tamanho_do_grafo)*

```
Python
def conjunto_independente(grafo, n):
    grafo_complemento = complemento_grafo(grafo, n)
    return maxclique(grafo_complemento, n)
```

A função ***conjunto_independente(grafo,tamanho_do_grafo)*** recebe um grafo e seu tamanho como parâmetros. A partir disso, a redução é realizada para o problema do clique máximo por meio da transformação do grafo recebido em seu complemento, utilizando a função ***complemento_grafo(grafo,tamanho_do_grafo)*** e repassando o complemento para a função ***maxclique(grafo_complemento,tamanho_do_grafo)***.

- Função *complemento_grafo(grafo,tamanho_do_grafo)*

```python
def complemento_grafo(grafo, n):
   for i in range(n):
       for j in range(n):
           if i != j: #Desconsiderando a presença de "1" na diagonal
principal, conforme orientado pelo professor
            if grafo[i][j] == 0:
               grafo[i][j] = 1
            else:
               grafo[i][j] = 0
   return grafo
```

A função ***complemento_grafo(grafo,tamanho_do_grafo)*** recebe um grafo e seu tamanho como parâmetros. A função itera sobre a matriz de adjacências, simplesmente invertendo os valores: o que era “1” é transformado em “0”, e o que era “0” é transformado em “1”. Ao final, o grafo complementar é retornado.

Resultados Obtidos

- Output da primeira instância (15 vértices)

```
Arquivo: input/independentconjinput/input1.txt
Conjunto independente máximo: [2, 3, 6, 10, 11]
Tempo de execução: 0.00082445 segundos
```

- Output da segunda instância (80 vértices)

```
Arquivo: input/independentconjinput/input2.txt
Conjunto independente máximo: [0, 3, 10, 21, 32, 36, 37, 40]
Tempo de execução: 2.16887045 segundos
```

- Output da terceira instância (150 vértices)

```
Unset
Arquivo: input/independentconjinput/input3.txt
Conjunto independente máximo: [0, 12, 44, 54, 58, 82, 87, 106, 109, 123]
Tempo de execução: 207.91675878 segundos
```

Conclusão

Desse modo, após a fase de projeção e implementação dos algoritmos, juntamente com o teste com as instâncias apresentadas, conclui-se que para o problema do clique máximo, os tempos de execução variaram entre 0.0000660 e 214.6982052 segundos, com a técnica de Branch and Bound permitindo uma exploração sistemática e otimizada do espaço de soluções. Essa eficiência pode ser atribuída ao uso da estratégia de poda, decidida pelo grupo, descrita anteriormente, que reduz substancialmente o número de combinações a serem geradas e exploradas, influindo em uma rápida convergência para a solução ótima. Entretanto, para a instância de 140 vértices, o algoritmo apresentou um tempo significativamente maior em relação às demais instâncias, constituindo a existência de uma relação entre o aumento do tamanho da instância de entrada com o aumento do tempo de execução do algoritmo.

Para o problema de Satisfatibilidade (SAT), a implementação utilizando a técnica do Backtracking apresentou tempos de execução que variaram de 0.0007083 a 0.9531214 segundos. A natureza recursiva e a capacidade de identificar e tratar caminhos não promissores, presente na estratégia de validar soluções parciais não consistentes, descrita anteriormente, contribuíram, de modo significativo para a eficiência do algoritmo, viabilizando que o programa identificasse, em tempo hábil e otimizado, um conjunto de valores-verdade para as variáveis satisfazível, quando aplicável, e a não existência do mesmo, quando aplicável, independente da variação do tamanho da instância.

No que se refere ao Conjunto Independente Máximo, a utilização da redução com custo polinomial somado ao algoritmo do Clique Máximo obtiveram tempos de execução que variaram de 0.00082445 a 207.91675878 segundos, sendo este último tempo, explicitamente maior que os demais, oriundo da instância com 150 vértices, a maior instância testada. Nesse caso, o tamanho da instância certamente influiu na geração de mais soluções no algoritmo Branch and Bound, resultando no aumento substancial de tempo de execução. Este foi o caso mais notório com base na métrica de tempo de execução. Para as demais instâncias, o tempo de execução está factível e razoavelmente rápido.

Em resumo, os tempos de execução obtidos em cada abordagem refletem a eficiência dos algoritmos utilizados. Esses resultados indicam que a escolha dos algoritmos não só garantiu soluções corretas, mas também otimizou o desempenho, sugerindo que futuras implementações podem se beneficiar de ajustes finos e potencialização das técnicas abordadas, principalmente na estratégia de poda do Branch and Bound e na estratégia de evitar a busca exaustiva no Backtracking, especialmente em cenários de maior escala.