Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Trabalho Prático

BCC241 - Projeto de Análise de Algoritmos

Felipe Braz Marques - 22.1.4030
Matheus Peixoto Ribeiro Vieira - 22.1.4104
Pedro Henrique Rabelo Leão de Oliveira - 22.1.4022

Professor: Anderson Almeida Ferreira

Ouro Preto

21 de setembro de 2024

Sumário

Lista de Figuras

Lista de Códigos Fonte

1 Introdução

Neste trabalho, propõe-se a implementação de algoritmos que resolvem três diferentes problemas utilizando as técnicas de backtracking, branch and bound e redução. Os problemas abordados são: Satisfabilidade, Clique e Conjunto Independente. A Satisfabilidade, ou SAT, é um problema de grande importância prática, com aplicações em diversas áreas, como teste de circuitos e design de software. Um exemplo clássico de SAT é a busca por uma atribuição de valores de verdade para variáveis em uma fórmula booleana na forma normal conjuntiva (CNF). Além disso, como destacado por [1], os problemas de Clique e Conjunto Independente podem ser reduzidos um ao outro de maneira simples, pois um conjunto de nós que forma um Conjunto Independente em um grafo é equivalente a um conjunto que forma um Clique em seu grafo complementar. Dessa forma, a solução de um dos problemas pode ser obtida a partir do outro, facilitando o processo de implementação.

A codificação foi feita em Python sem o uso de bibliotecas adicionais para a implementação dos algoritmos.

1.1 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- Linguagem utilizada: Python
- Biblioteca para exibição dos grafos: Matplotlib e networkx

2 Clique

Para resolver o problema do Clique, onde o objetivo é encontrar um conjunto máximo de vértices tal que todas as possíveis arestas entre eles estejam presentes, foi utilizado uma solução Branch and Bound onde foram definidos os seguintes pontos do problema:

- **Variáveis para a solução:** $X_1, \ldots, X_n$, onde X_i representa um vértice pertencente ao grafo.
- **Domínio para as variáveis da solução:** $\{0, 1\}$, onde 0 indica que aquele vértice não faz parte do clique máximo e 1 indica o contrário.
- **Restrições:** Todos os vértices representados pelas variáveis que possuem valor 1 devem estar conectados entre si através de uma aresta.
- **Objetivo:** Obter o maior conjunto possível de vértices tal que todas as possíveis arestas entre eles estejam presentes.

Primeiro, o problema foi lido através de uma função e armazenado em uma variável na qual guarda o número de vértices do grafo e a matriz de adjacência do mesmo.

Depois, como melhor solução de início foi gerado um vetor através de uma função que varre a matriz de adjacência em busca de uma célula na qual indica que dois vértices estão conectados. Quando encontra, faz com que esse vetor tenha o valor 1 nas posições que representam esses respectivos vértices e valor 0 nas demais posições. Caso não encontre, o vetor terá apenas a primeira posição com o valor 1 e as demais com o valor 0, indicando um clique com apenas o primeiro vértice do grafo.

```
1  def geraSolucao(problema):
2      solucao = [0] * problema[0][0]
3
4      for i in range(problema[0][0]):
5          for j in range(problema[0][0]):
6              if(problema[i+1][j] == 1):
7                  solucao[i] = 1
8                  solucao[j] = 1
9                  return solucao
10
11     solucao[0] = 1
12     return solucao
```

Código 1: Função que gera a melhor solução inicialmente

A solução inicial é um vetor com o número de posições igual ao número de vértices do grafo, com todas elas possuindo o valor -1, representando que as variáveis estão "livres", sem nenhum valor atribuído a elas, e, portanto, a solução inicial está vazia.

Por fim, a função principal é chamada para encontrar a melhor solução, ou seja, o clique máximo, utilizando branch and bound como estratégia.

```
def branchAndBoundClique(solucao, i, problema, melhor):
    if eCompleta(solucao, problema):
        melhor[:] = solucao
        return melhor

    else:
        for j in range(2):
            solucao[i] = j

            if(eConsistente(solucao, problema, i) and
               ePromissora(solucao, problema, melhor, i)):
                melhor = branchAndBoundClique(solucao, i+1,
                   problema, melhor)

            solucao[i] = -1

        return melhor
```

Código 2: Função que encontra o clique máximo através de branch and bound

Para a verificar se a solução é completa, verifica-se se nenhuma posição do vetor solução possui o valor -1, o que significa que todas as variáveis tem valores 0 ou 1 atribuídos a elas.

Já para verificar se uma solução é consistente, verifica-se, através da matriz de adjacência, se os vértices representados pelas variáveis que possuem valor 1 na solução possuem todas as possíveis arestas entre eles. Caso contrário, a solução é inconsistente e ocorre a poda daquele ramo.

```
def eConsistente(solucao, problema, i):
    verticesNaSolucao = []

    for j in range(i+1):
        if solucao[j] == 1:
            verticesNaSolucao.append(j)

    for j in range(len(verticesNaSolucao)-1):
```

```
9           for k in range(j+1, len(verticesNaSolucao)):
10              if(problema[verticesNaSolucao[j]+1][
                    verticesNaSolucao[k]] == 0):
11                  return False
12
13      return True
```

Código 3: Função que verifica a consistência da solução até o momento

Para a verificar se uma solução é promissora, adotamos a estratégia de somar o número de vértices com valor 1 no vetor da solução, ou seja, os vértices pertencentes ao clique até então, com o restante de posições do vetor sem valor atribuído, ou seja, com o valor -1. Dessa forma, obtemos o número máximo de vértices que estarão presentes naquela solução e, posteriormente, comparamos ele com o número de vértices do clique da melhor solução até o momento. Para obtermos esse último, apenas somamos os valores presentes nas variáveis da solução, uma vez que elas possuem valor 0 ou 1, e, portanto, a soma terá o valor do número de vértices presentes nesse clique.

```
1   def ePromissora(solucao, problema, melhor, i):
2       numVerticesMelhor = sum(melhor)
3       numVerticesMaximoSolucao = 0 # armazena o numero maximo de
            vertices possiveis nessa solucao
4
5       for j in range(i+1):
6           numVerticesMaximoSolucao += solucao[j]
7
8       numVerticesMaximoSolucao += (problema[0][0] - (i+1))
9
10      return numVerticesMaximoSolucao > numVerticesMelhor
```

Código 4: Função que verifica se a solução até o momento é promissora

2.1 Exemplos de instância do problema

2.1.1 Instância 1

Arquivo de entrada para a instância 1

```
9
0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
1 1 0 1 1 0 0 0 0
0 0 1 0 0 0 1 0 0
0 0 1 0 0 1 0 1 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 1 1
0 0 0 0 1 0 1 0 1
0 0 0 0 0 0 1 1 0
```

Saída da instância 1

```
[0, 0, 0, 0, 0, 0, 1, 1, 1]
Vértices presentes no clique:
7
8
9
Tempo de execução:  0.001000 segundos
```

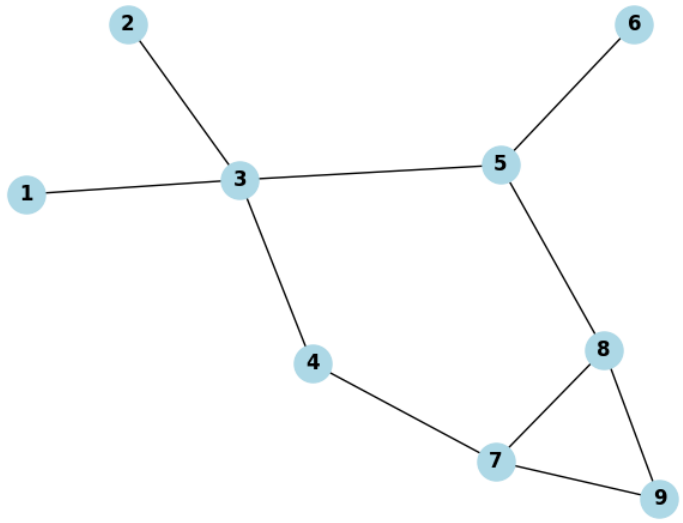

Figura 1: Grafo da instância 1 de entrada.

2.1.2 Instância 2

Arquivo de entrada para a instância 2

```
5
0 1 1 1 0
1 0 0 0 0
0 1 0 1 1
1 0 1 0 1
0 0 1 1 0
```

Saída da instância 2

```
[0, 0, 1, 1, 1]
Vértices presentes no clique:
3
4
5
Tempo de execução:  0.000995 segundos
```

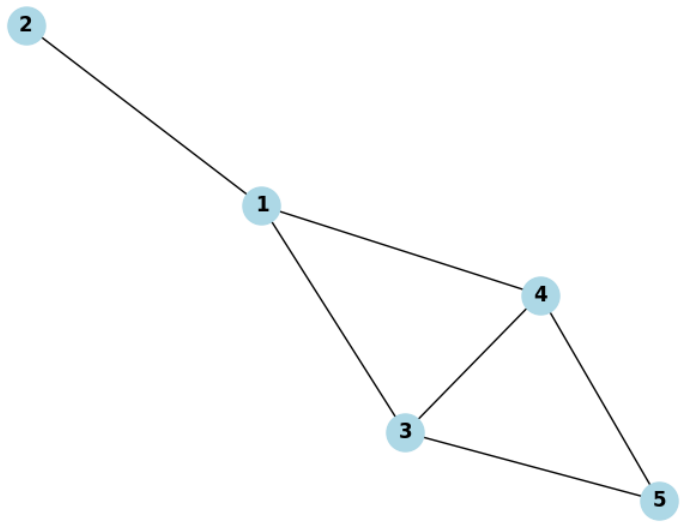

Figura 2: Grafo da instância 2 de entrada.

2.1.3 Instância 3

Arquivo de entrada para a instância 3

```
10
0 1 0 0 0 1 1 0 0 0
1 0 0 0 1 0 1 0 1 0
0 0 0 1 0 0 1 1 1 1
0 0 1 0 1 1 0 1 1 0
0 1 0 1 0 0 0 0 0 1
1 0 0 1 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 1
0 0 1 1 0 0 0 0 0 1
0 1 1 1 0 0 0 0 0 0
0 0 1 0 1 0 1 1 0 0
```

Saída da instância 3

```
[0, 0, 1, 0, 0, 0, 0, 1, 0, 1]
Vértices presentes no clique:
3
8
10
Tempo de execução:  0.000997 segundos
```

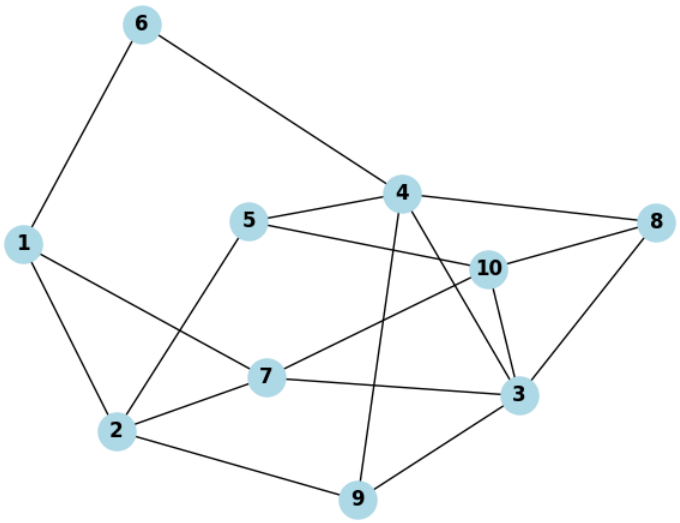

Figura 3: Grafo da instância 3 de entrada.

3 Conjunto Independente

Para resolver o problema do Conjunto Independente (ou Conjunto Estável) por meio de uma redução polinomial ao problema do Clique, iremos usar a relação entre esses dois problemas. Sabemos que um conjunto independente de um grafo é equivalente a um clique no complemento desse grafo.

3.1 Lógica da função

Passos para a solução

- **Complemento do grafo:** O complemento de um grafo é um grafo onde as arestas que estavam presentes no grafo original são removidas e as arestas que não estavam presentes são adicionadas.
- **Redução:** Dado um grafo GGG, o problema do conjunto independente em GGG pode ser resolvido encontrandoo o clique máximo no complemento do grafo G'G'G'.
- **Aproveitar a implementação do clique:** Após calcular o complemento do grafo, aplicamos o algoritmo de clique já implementado.

3.2 Função geraComplemento

O Código 5 apresenta a função que foi implementada para gerar o complemento do grafo.

```
1  def geraComplemento(problema):
2      numVertices = problema[0][0]  # Numero de vertices (primeira
           linha da matriz)
3
4      # Inicializa a matriz de complemento
5      complemento = [[0] * numVertices for _ in range(numVertices)
           ]
6
7      # Preenche a matriz de complemento
8      for i in range(numVertices):
9          for j in range(numVertices):
10             if i != j:  # Nao considerar a diagonal principal
11                 complemento[i][j] = 1 - problema[i + 1][j]
```

```
12
13      # Adiciona o numero de vertices como a primeira linha da
            matriz de complemento
14      complemento.insert(0, [numVertices])
15
16      return complemento
```

Código 5: Função que gera o complemento do grafo para o problema do conjunto independente

A função recebe como parâmetro de entrada uma matriz de adjacência que representa o grafo original. A primeira linha da matriz contém o número de vértices do grafo, e as linhas subsequentes indicam a presença de arestas entre os vértices. Um valor 1 indica que há uma aresta entre dois vértices, e um valor 0 indica que não há aresta.

Passos da função:

- **Captura o número de vértices:** O número de vértices do grafo é obtido da primeira linha da matriz problema. A linha problema[0][0] armazena essa informação.
- **Inicializa a matriz do complemento:**

```
complemento = [[0] * numVertices for _ in range(numVertices)]
```

  Nessa parte a função cria uma nova matriz quadrada de dimensão numVertices * numVertices, inicialmente preenchida com zeros. Essa matriz será preenchida com informações do complemento do grafo.

- **Preenche a matriz de complemento:**

```
for i in range(numVertices):
    for j in range(numVertices):
        if i != j:
            complemento[i][j] = 1 - problema[i + 1][j]
```

  A função utiliza dois loops aninhados para percorrer todos os pares de vértices (i, j) do grafo e, para cada par, verifica se os vértices são diferentes. Isso é necessário porque um vértice não pode ser adjacente a si mesmo. Para os pares de vértices distintos, a função calcula o complemento da adjacência: se há uma arestas entre dois vértices no grafo original, o complemento remove essa aresta (1 vira 0), e se não há aresta, o complemento a adiciona (0 vira 1).

- **Adiciona o número de vértices como a primeira linha:**

```
complemento.insert(0, [numVertices])
```

Após gerar a matriz de adjacência do complemento, a função insere o número de vértices como primeira linha da matriz, para manter o mesmo formato da matriz original (onde a primeira linha sempre indica a quantidade de vértices).

3.3 Exemplos de instância do problema

3.3.1 Instância 1

Arquivo de entrada para a instância 1

```
9
0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
1 1 0 1 1 0 0 0 0
0 0 1 0 0 0 1 0 0
0 0 1 0 0 1 0 1 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 1 1
0 0 0 0 1 0 1 0 1
0 0 0 0 0 0 1 1 0
```

Saída da instância 1

```
Conjunto independente máximo:  1, 2, 4, 6, 9
Tempo de execução:  0.002493 segundos
```

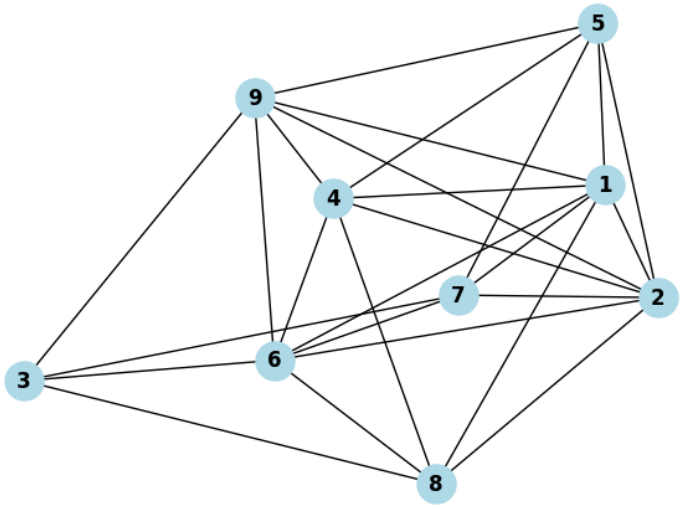

Figura 4: Grafo complemento da instância 1 de entrada.

3.3.2 Instância 2

Arquivo de entrada para a instância 2

```
5
0 1 1 1 0
1 0 0 0 0
0 1 0 1 1
1 0 1 0 1
0 0 1 1 0
```

Saída da instância 2

```
Conjunto independente máximo:  1, 3, 5 Tempo de execução:
0.001508 segundos
```

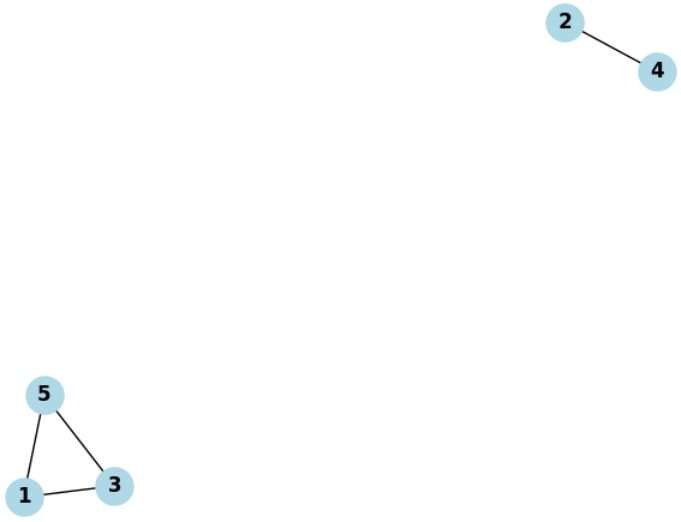

Figura 5: Grafo complemento da instância 2 de entrada.

3.3.3 Instância 3

Arquivo de entrada para a instância 3

```
10
0 1 0 0 0 1 1 0 0 0
1 0 0 0 1 0 1 0 1 0
0 0 0 1 0 0 1 1 1 1
0 0 1 0 1 1 0 1 1 0
0 1 0 1 0 0 0 0 0 1
1 0 0 1 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 1
0 0 1 1 0 0 0 0 0 1
0 1 1 1 0 0 0 0 0 0
0 0 1 0 1 0 1 1 0 0
```

Saída da instância 3

```
Conjunto independente máximo:  5, 6, 7, 8, 9
Tempo de execução:  0.002523 segundos
```

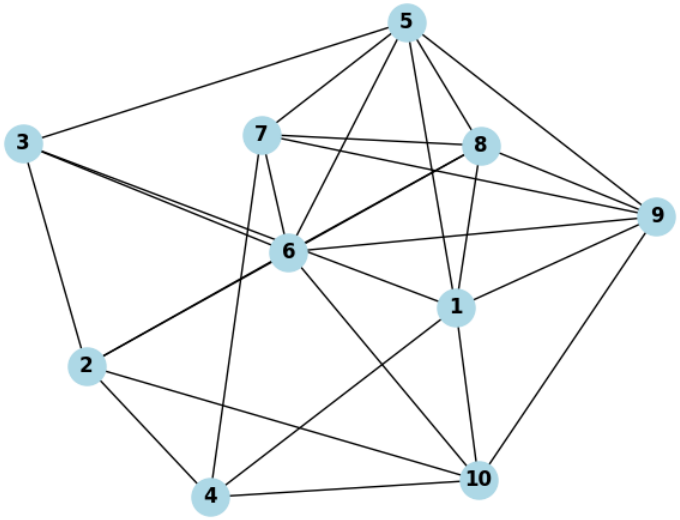

Figura 6: Grafo complemento da instância 3 de entrada.

4 Satisfabilidade

Para o problema da satisfabilidade (SAT), dada uma fórmula booleana na forma normal conjuntiva, é necessário encontrar, com *backtracking*, algum resultado que satisfaça a mesma, ou, caso não seja possível satisfazer, informar a não possibilidade de satisfabilidade.

Assim, para tal problema, podemos descrevê-lo da seguinte forma:

- **Variáveis para a solução:** $X_1, \ldots, X_n$, onde X_i indica o valor verdade para determinada variável do problema;
- **Domínio para as variáveis da solução:** {False, True}, indicando os valores lógicos falso e verdadeiro, respectivamente;
- **Restrições:** Todas as cláusulas devem resultar em verdadeiro
- **Objetivo:** Obter algum conjunto de valores do domínio para as variáveis de forma que todas as cláusulas sejam satisfeitas

Para a entrada do programa, é lido um arquivo onde a primeira linha indica uma quantidade N de variáveis e, as demais, indicam as cláusulas da fórmula. Cada cláusula é formada N valores inteiros, onde 1 representa a presença da variável na entrada, 0 indica a negação e -1 indica a ausência.

A seguir temos o exemplo de entrada presente em um arquivo de entrada para a seguinte fórmula:

$$(A \vee \neg B \vee C) \wedge (\neg A \vee B \vee D) \wedge (B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

Arquivo de entrada para a fórmula

```
4
1 0 1 -1
0 1 -1 1
-1 1 0 0
0 0 -1 1
```

Após a realização da leitura do arquivo, teremos uma lista de listas, onde a lista externa indica o problema como um todo e, cada lista interna, indica as clásulas.

Entrada do programa para a fórmula fornecida

```
[[1, 0, 1, -1],
[0, 1, -1, 1],
[-1, 1, 0, 0],
[0, 0, -1, 1]]
```

Para o início da solução, é gerado um vetor `solucao` com tamanho igual à quantidade de variáveis do problema SAT, onde cada posição é inicializada como nula.

A seguir, é chamada a função `backtrack`, passando como parâmetros `solucao`, `i`, indicando a última variável que teve valor atribuído, sendo que, neste primeiro momento será zero, e `problema`, indicando a entrada lida.

Na função `backtrack`, primeiro verificamos se a `solucao` é uma solução final, caso seja, retornamos o valor `True` indicando que foi encontrada uma solução e encerramos todas as camadas recursivas retornando este valor.

Caso a solução parcial não seja final, então identificamos os valores candidatos para a próxima variável a ser verificada e chamamos recursivamente a função `backtrack`, onde os valores candidatos são pertencentes ao domínio do problema.

4.1 Verificar solução final

Para verificar se uma solução parcial é uma solução final, utilizamos a função `verificar_solucao`, que recebe a instância do problema e a solução atual.

Primeiramente, verificamos se todos os valores da solução atual são diferentes de nulo, indicando se o *bactracking* chegou a uma solução completa ou não. Caso uma variável seja nula, retornamos o valor falso, caso contrário, passaremos a verificar todas as cláusulas, onde, caso uma seja falsa, já retornamos que não é uma solução, pois não terá como satisfazer a fórmula desejada, e, ao satisfazer, retornamos que, de fato, é uma solução para o problema.

4.2 Gerar candidatos

Para a construção de candidatos realizada na função `construir_candidatos`, recebemos a solução parcial, a variável atual que está sendo verificada, o problema e o domínio.

Primeiramente, verificamos se todas as variáveis já tiveram valores atribuídos. Caso tenham, então retornamos uma lista vazia. Assim, não tendo candidatos, a função de `backtrack` será finalizada para este ramo de opções.

Agora, não retornando prematuramente, iremos atribuir, para a i-ésima variável,

cada valor do domínio e verificar se, com o esse valor, a fórmula não terá a sua restrição violada.

Dessa forma, com o valor atribuído e iterando sobre todas as cláusulas, verificamos se elas podem ser computáveis, ou seja, se todos o valor de todas as variáveis atribuidas até o momento estão presentes na mesma.

No exemplo a seguir, verificamos que a cláusula não pode ser computada ainda, pois, com `i = 3`, estamos verificando a viabilidade do valor `True`. Todavia, com uma simples inspeção, verificamos que, até o momento, a cláusula é falsa. Mas como nada sabemos sobre as duas últimas variáveis, nada podemos afirmar, pois ela pode se tornar verdadeira ou continuar falsa. Logo, tal cláusula não pode ser computável no momento.

```
Cláusula:  1 -1 0 0 1
Solução:  [False, False, True, _, _]
i = 3
```

Em formato de código, para realizar tal verificação, usamos a função 6, onde verificamos se, da i-ésima variável em diante, todas têm valores -1, indicando que não estão presentes na fórmula. Assim, as próximas variáveis não impactarão no cálculo.

```
1 def clausula_computavel(i, clausula, qtd_variaveis):
2     for j in range(i, qtd_variaveis):
3         if clausula[j-1] == -1:
4             return False
```

Código 6: Verificando se cláusula é computável

Vale ressaltar que, a presença ou ausência das variáveis anteriores ao valor `i` não são necessárias verificações, pois garantimos que elas já foram atribuidas em passos anteriores do *backtracking*.

Agora, sabendo que a cláusula pode ser computável, verificamos se ela será satisfeita, para tanto, iteramos sobre todas as variáveis e verificamos o resultado ao final. A cláusula sendo verdadeira, verificamos a próxima, caso contrário, encerramos para tal valor do domínio e o retiramos da lista de candidatos, pois não será possível satisfazer a fórmula.

4.3 Exemplos de instância

4.3.1 Instância 1

$$(A \vee B \vee C) \wedge (A \vee \neg B) \wedge (B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$$

Entrada da instância 1

```
3
1 1 1
1 0 -1
-1 1 0
0 -1 1
0 0 0
```

Saída da instância 1

```
Solução não encontrada
Tempo de execução:  0.000503 segundos
```

4.3.2 Instância 2

$$(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$$

Entrada da instância 2

```
3
1 1 1
0 0 0
```

Saída da instância 2

```
Solução encontrada:  [False, False, True]
Tempo de execução:  0.000090 segundos
```

4.3.3 Instância 3

$$(A \vee \neg B \vee C) \wedge (\neg A \vee B \vee D) \wedge (B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

Entrada da instância 3

```
4
1 0 1 -1
0 1 -1 1
-1 1 0 0
0 0 -1 1
```

Saída da instância 3

```
Solução encontrada:  [False, False, False, False]
Tempo de execução:  0.000175 segundos
```

4.3.4 Instância 4

$$(\neg B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg A \vee B) \wedge (\neg B \vee \neg C) \wedge (A \vee C) \wedge (A \vee B)$$

Entrada da instância 4

```
3
-1 0 0
0 -1 1
0 1 -1
-1 0 0
1 -1 1
1 1 -1
```

Saída da instância 4

```
Solução não encontrada
Tempo de execução:  0.000192 segundos
```

4.3.5 Instância 5

$$(A \vee B \vee C \vee D) \wedge (\neg A \vee \neg B \vee C) \wedge (A \vee \neg C \vee D) \wedge (\neg B \vee D)$$

Entrada da instância 5

```
4
1 1 1 1
0 0 1 -1
1 -1 0 1
-1 0 -1 1
```

Saída da instância 5

```
Solução encontrada:  [False, False, False, True]
Tempo de execução:  0.000134 segundos
```

4.3.6 Instância 6

$$(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (A \vee \neg B \vee D) \wedge (\neg A \vee B \vee \neg D) \wedge (A \vee \neg C \vee D) \wedge (\neg A \vee C \vee \neg D) \wedge (\neg A \vee \neg B \vee C \vee D) \wedge (A \vee B \vee \neg C \vee \neg D)$$

Entrada da instância 6

```
4
1 1 1 -1
0 0 0 -1
1 0 -1 1
0 1 -1 0
1 -1 0 1
0 -1 1 0
0 0 1 1
1 1 0 0
```

Saída da instância 6

```
Solução encontrada:  [False, True, False, True]
Tempo de execução:  0.000151 segundos
```

4.3.7 Instância 7

$(A \vee B \vee \neg C \vee D \vee E) \wedge (\neg A \vee \neg B \vee C \vee \neg E) \wedge (A \vee \neg C \vee \neg D \vee E) \wedge (\neg A \vee B \vee D) \wedge (C \vee \neg D \vee \neg E)$

Entrada da instância 7

```
5
1 1 0 1 1
0 0 1 -1 0
1 -1 0 0 1
0 1 -1 1 -1
-1 -1 1 0 0
```

Saída da instância 7

```
Solução encontrada:  [False, False, False, False, False]
Tempo de execução:  0.000105 segundos
```

4.3.8 Instância 8

$(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (A \vee \neg B \vee D) \wedge (\neg A \vee B \vee \neg D) \wedge (A \vee \neg C \vee D) \wedge (\neg A \vee C \vee \neg D) \wedge (\neg A \vee \neg B \vee C \vee D) \wedge (A \vee B \vee \neg C \vee \neg D)$

Entrada da instância 8

```
4
1 1 -1 -1
0 0 -1 -1
1 -1 1 -1
-1 -1 0 -1
0 1 0 1
-1 -1 -1 0
```

Saída da instância 8

```
Solução encontrada:  [True, False, False, False]
Tempo de execução:  0.000208 segundos
```

4.3.9 Instância 9

$$(\neg A \vee B) \wedge (A \vee \neg B) \wedge (A \vee B) \wedge (\neg A \vee \neg B)$$

Entrada da instância 9

```
2
0 1
1 0
1 1
0 0
```

Saída da instância 9

```
Solução não encontrada
Tempo de execução:  0.000091 segundos
```

4.3.10 Instância 10

$$(\neg A)$$

Entrada da instância 10

```
1
0
```

Saída da instância 10

```
Solução encontrada:  [False]
Tempo de execução:  0.000102 segundos
```

5 Considerações Finais

Com o fim do desenvolvimento do trabalho, desenvolvemos nossos conhecimentos sobre branch and bound, redução polinomial e backtracking ao colocar, na prática, os conhecimentos antes apresentados de forma teórica.

Ademais, o trabalho também foi útil para relembrarmos conceitos da Teoria dos Grafos, como o fato de que o conjunto independente máximo poder ser calculado a partir do complemento de um grafo e executando o algoritmo do Clique Máximo para o mesmo.

Referências

[1] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006.