

# BCC241 - Projeto e Análise de Algoritmos

## Clique, Conjunto Independente e SAT

**Felipe Braz, Matheus Peixoto, Pedro Henrique Oliveira**

Universidade Federal de Ouro Preto, UFOP

Departamento de Computação, DECOM

Professor : **Anderson Almeida Ferreira**



# Conteúdo

Introdução

Clique

Conjunto Independente

SAT

# Introdução

## Clique, Conjunto independente e SAT

- ▶ A Satisfabilidade, ou SAT, é um problema de grande importância prática, com aplicações em diversas áreas, como teste de circuitos e design de software.
- ▶ O problema do clique consiste em encontrar um subgrafo completo em que todos vértices estão conextados entre si, sendo um desafio NP-completo com aplicações em várias áreas.
- ▶ O problema do conjunto independente pode ser reduzido ao problema do Clique, 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.

# Clique

## Problema do Clique

Dado um grafo, o objetivo é encontrar um conjunto máximo de vértices tal que todas as possíveis arestas entre eles estejam presentes. Para isso, foi utilizado a estratégia *branch and bound* para encontrar a solução do problema.

## Definição do problema

- ▶ **Variáveis para a solução:**  $X_1, \dots, X_n$ , 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.

## Leitura do Problema

- ▶ O problema foi lido e armazenado em uma única variável, contendo o número de vértices e a matriz de adjacência do grafo.
- ▶ O vetor que armazena a solução inicial foi iniciado com todas as posições possuindo valores  $-1$ , para representar que a mesma começa vazia.



## Geração Inicial da Melhor Solução

```
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  
13
```

## Função que executa o branch and bound

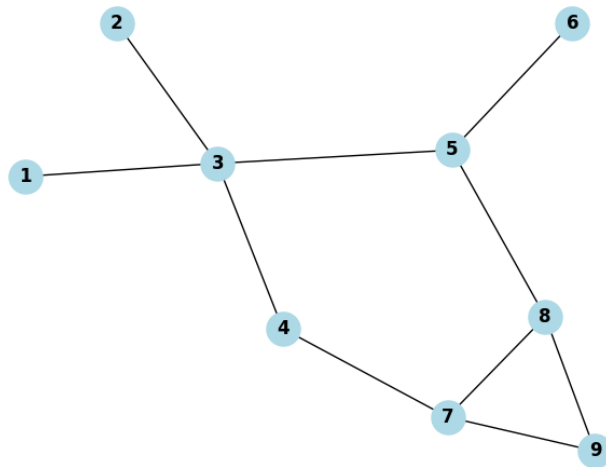
```
1 def branchAndBoundClique(solucao, i, problema, melhor):
2     if eCompleta(solucao, problema):
3         melhor[:] = solucao
4         return melhor
5
6     else:
7         for j in range(2):
8             solucao[i] = j
9
10            if (eConsistente(solucao, problema, i) and ePromissora(solucao,
11                problema, melhor, i)):
12                melhor = branchAndBoundClique(solucao, i+1, problema, melhor)
13
14            solucao[i] = -1
15
16        return melhor
```

## Função que verifica a consistência da solução

```
1 def eConsistente(solucao, problema, i):
2     verticesNaSolucao = []
3
4     for j in range(i+1):
5         if solucao[j] == 1:
6             verticesNaSolucao.append(j)
7
8     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
14
```

## Função que verifica se a solução é promissora

```
1 def ePromissora(solucao, problema, melhor, i):
2     numVerticesMelhor = sum(melhor)
3     numVerticesMaximoSolucao = 0 # armazena o numero maximo de vertices
4     # possiveis nessa solucao
5     for j in range(i+1):
6         numVerticesMaximoSolucao += solucao[j]
7
8     numVerticesMaximoSolucao += (problema[0][0] - (i+1))
9
10    return numVerticesMaximoSolucao > numVerticesMelhor
11
```



**Figura:** Grafo da instância de entrada.

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

100%

### Saída da instância de entrada

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

```
Vértices presentes no clique:
```

```
7
```

```
8
```

```
9
```

```
Tempo de execução: 0.001000 segundos
```

## Conjunto Independente

## O problema do 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.

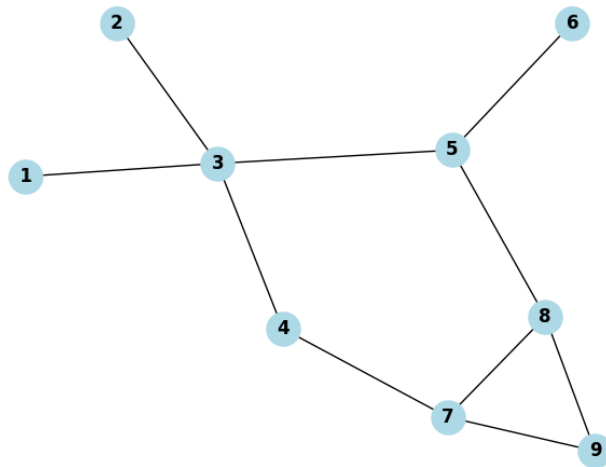


## Passos para a solução

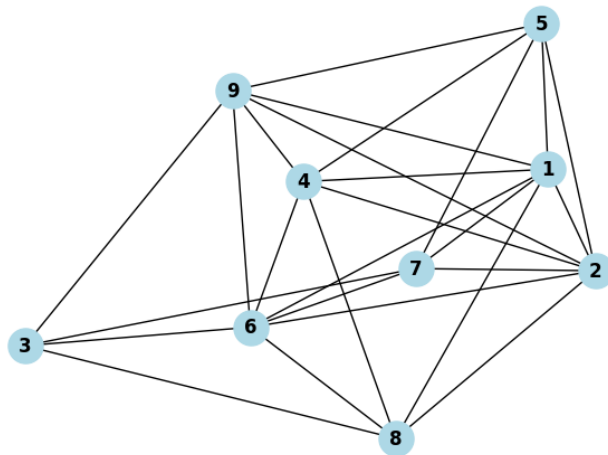
- ▶ **Complemento do grafo:** O complemento de um grafo é um grafo onde as arestas que estavam presentes no grafo original são removidas e as arestas que não estavam presentes são adicionadas.
- ▶ **Redução:** Dado um grafo  $G$ , o problema do conjunto independente em  $G$  pode ser resolvido encontrando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

## Função geraComplemento

```
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
17
```



**Figura:** Grafo da instância de entrada.



**Figura:** Grafo complemento da instância de entrada.

## Saída

### Saída da instância

Conjunto independente máximo: 1, 2, 4, 6, 9

Tempo de execução: 0.002493 segundos

# SAT

## O problema SAT

Dada uma fórmula booleana na forma normal conjuntiva, é necessário encontrar, com *backtracking*, algum resultado que satisfaça a mesma. Caso não seja possível, informa tal possibilidade

## Definição do problema

- ▶ **Variáveis para a solução:**  $X_1, \dots, X_n$ , onde  $X_i$  indica o valor verdade para determinada variável do problema;
- ▶ **Domínio para as variáveis da solução:**  $\{\text{False}, \text{True}\}$ ;
- ▶ **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



## Entrada

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

Arquivo de entrada para a fórmula

4

1 0 1 -1

0 1 -1 1

-1 1 0 0

0 0 -1 1

## Backtracking

```
1 def backtrack(solucao, i, problema):
2     dominio = [False, True]
3     if verificar_solucao(problema, solucao):
4         return True
5     else:
6         i = i + 1
7         candidatos = construir_candidatos(solucao, i, problema, dominio)
8         for c in candidatos:
9             solucao[i-1] = c
10            finished = backtrack(solucao, i, problema)
11            if finished:
12                return True
13            solucao[i-1] = None
14    return False
15
```

## Construir candidatos

- ▶ Iterar sobre cada valor do domínio e colocá-lo na solução
- ▶ Verificar cada cláusula do problema
- ▶ Verificar se as variáveis já atribuídas compõem a cláusula
- ▶ Se comporem, verificar se a cláusula é satisfeita.
- ▶ Se todas as cláusulas forem satisfeitas, o valor pode ser suficiente.
- ▶ Se uma cláusula não for, tal valor do domínio não poderá participar da solução.

## Construir candidatos

```
1 def construir_candidatos(solucao, i, problema, dominio):
2     if i > len(solucao): return []
3     candidatos = dominio.copy()
4     s_temp = solucao.copy()
5     qtd_variaveis = len(problema[0])
6     for c in dominio:
7         s_temp[i-1] = c
8         for clausula in problema:
9             satisfeita = False
10            if clausula_computavel(i, clausula, qtd_variaveis):
11                for variavel_atual, literal in enumerate(clausula):
12                    if literal == 1:
13                        satisfeita = satisfeita or s_temp[variavel_atual]
14                    elif literal == 0:
15                        satisfeita = satisfeita or not s_temp[variavel_atual]
16            else: continue
17            if not satisfeita:
18                candidatos.pop(0)
19                break
20     return candidatos
21
```

## Cláusula computável

- ▶ A cláusula a seguir não poderá ser computável.
- ▶ Estamos analisando o valor do True e a cláusula ficará falsa.
- ▶ Não sabemos se ela será verdadeira ou falsa no fim do processamento.

Cláusula: 1 -1 0 0 1

Solução: [False, False, True, \_, \_]

i = 3

```
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  
5
```

## Exemplo instância

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

### Saída da instância

Solução não encontrada

Tempo de execução: 0.000503 segundos

## Exemplo instância

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

### Saída da instância

Solução encontrada: [False, True, False, True]

Tempo de execução: 0.000151 segundos

## Exemplo instância

### Entrada da instância

Variáveis: 18

Cláusulas: 4096

### Saída da instância

Solução encontrada: [False, False, False, False, False, False, False, False, False, False, False, False, False, False, True, False, False]

Tempo de execução: 0.111010 segundos



## Exemplo instância

### Entrada da instância

Variáveis: 10

Cláusulas: 59049

### Saída da instância

Solução não encontrada

Tempo de execução: 56.896429 segundos