BCC241 - Projeto e Análise de Algoritmos

Clique, Conjunto Independente e SAT

Felipe Braz Marques, Matheus Peixoto Ribeiro Vieira, Pedro Henrique Rabelo Leão de Oliveira

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Professor : **Anderson Almeida Ferreira**

Conteúdo

Introdução

Clique, Conjunto indpendente 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 indpendente 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:** $X_1, \ldots, X_n$, 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.

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

```
def geraSolucao(problema):
    solucao = [0] * problema[0][0]

    for i in range(problema[0][0]):
        for j in range(problema[0][0]):
            if(problema[i+1][j] == 1):
                solucao[i] = 1
                solucao[j] = 1
                return solucao

    solucao[0] = 1
    return solucao
```

Função que executa o branch and bound

```
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

```

Função que verifica a consistência da solução

```
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):
        for k in range(j+1, len(verticesNaSolucao)):
            if(problema[verticesNaSolucao[j]+1][verticesNaSolucao[k]] == 0):
                return False

    return True
```

Função que verifica se a solução é promissora

```
def ePromissora(solucao, problema, melhor, i):
    numVerticesMelhor = sum(melhor)
    numVerticesMaximoSolucao = 0 # armazena o numero maximo de vertices possiveis nessa solucao

    for j in range(i+1):
        numVerticesMaximoSolucao += solucao[j]

    numVerticesMaximoSolucao += (problema[0][0] - (i+1))

    return numVerticesMaximoSolucao > numVerticesMelhor

```

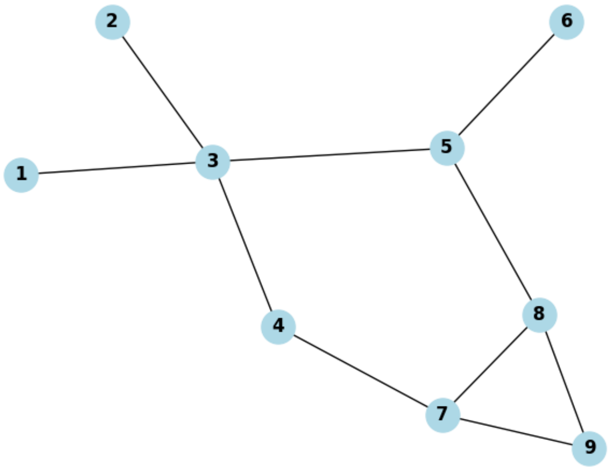

Figura: Grafo da instância de entrada.

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 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.

Função geraComplemento

```
def geraComplemento(problema):
    numVertices = problema[0][0]  # Numero de vertices (primeira linha da matriz)

    # Inicializa a matriz de complemento
    complemento = [[0] * numVertices for _ in range(numVertices)]

    # Preenche a matriz de complemento
    for i in range(numVertices):
        for j in range(numVertices):
            if i != j:  # Nao considerar a diagonal principal
                complemento[i][j] = 1 - problema[i + 1][j]

    # Adiciona o numero de vertices como a primeira linha da matriz de complemento
    complemento.insert(0, [numVertices])

    return complemento
```

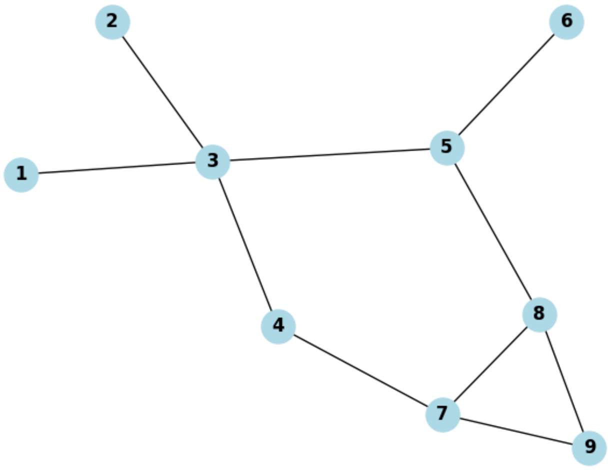

Figura: Grafo da instância de entrada.

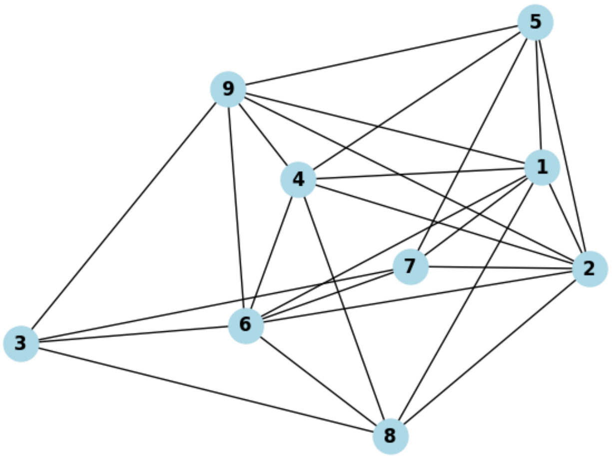

Figura: Grafo complemento da instância de entrada.

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:** $X_1, \ldots, X_n$, onde X_i indica o valor verdade para determinada variável do problema;
- **Domínio para as variáveis da solução:** {False, 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

```python
def backtrack(solucao, i, problema):
    dominio = [False, True]
    if verificar_solucao(problema, solucao):
        return True
    else:
        i = i + 1
        candidatos = construir_candidatos(solucao, i, problema, dominio)
        for c in candidatos:
            solucao[i-1] = c
            finished = backtrack(solucao, i, problema)
            if finished:
                return True
            solucao[i-1] = None
    return False
```

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

```
def construir_candidatos(solucao, i, problema, dominio):
    if i > len(solucao): return []
    candidatos = dominio.copy()
    s_temp = solucao.copy()
    qtd_variaveis = len(problema[0])
    for c in dominio:
        s_temp[i-1] = c
        for clausula in problema:
            satisfeita = False
            if clausula_computavel(i, clausula, qtd_variaveis):
                for variavel_atual, literal in enumerate(clausula):
                    if literal == 1:
                        satisfeita = satisfeita or s_temp[variavel_atual]
                    elif literal == 0:
                        satisfeita = satisfeita or not s_temp[variavel_atual]
            else:continue
            if not satisfeita:
                candidatos.pop(0)
                break
    return candidatos
```

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
```

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

```

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

$$(A \lor B \lor C) \land (\neg A \lor \neg B \lor \neg C) \land (A \lor \neg B \lor D) \land (\neg A \lor B \lor \neg D) \land (A \lor \neg C \lor D) \land$$
$$(\neg A \lor C \lor \neg D) \land (\neg A \lor \neg B \lor C \lor D) \land (A \lor B \lor \neg C \lor \neg D)$$

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, 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
```