Trabalho final de Projeto e Análise de Algoritmos

Arthur Carvalho Basílio[1], Gisele de Sousa Ribeiro[1], Lucas Danillo B. P. Soares[1]

[1]Departamento de Computação - Universidade Federal do Piauí (UFPI) - Teresina, PI - Brasil

{basilio.arthur, gisele, lucasdanillo}@ufpi.edu.br

Resumo. *Este relatório descreve os aspectos relacionados ao trabalho final da disciplina de Projeto e Análise de Algoritmos. É abordado o problema do Conjunto Máximo Independente (CMI), que pertence à classe dos problemas NP-completo estudados pela Ciência da Computação. A partir da investigação de uma aplicação prática, é proposto um algoritmo heurístico que obtém uma boa solução para os problemas de CMI em um tempo de execução aceitável.*

1. Introdução

Na Teoria dos Grafos, um conjunto independente é um conjunto de vértices de um grafo no qual nenhum par de vértices é adjacente. Ou seja, é um conjunto V de vértices em que nenhum par é interligado por uma aresta. O tamanho de um conjunto independente é o número de vértices que ele contém.

Um CMI não é um subconjunto próprio de qualquer outro conjunto independente, logo, possui o maior tamanho possível para um determinado grafo G. Esse tamanho é chamado de número de independência de G, e é denotado por α(G). O problema de encontrar tal conjunto é chamado de Problema do Conjunto Independente Máximo e é um problema de otimização NP-completo. Dito isso, é improvável que exista um algoritmo eficiente para encontrar um Conjunto Máximo Independente.

Nas seguintes seções, são apresentadas uma proposta de aplicação prática para o problema de CMI e sua implementação, bem como um algoritmo simples (*baseline*) que resolve instâncias do problema, porém em tempo inviável. Além disso, é provado que ele pertence à classe de problemas NP-completo. Por fim, é mostrada a implementação e análise de um algoritmo heurístico que soluciona problemas de CMI.

2. Aplicação prática proposta

A aplicação selecionada lida com a otimização da distribuição geográfica dos dispositivos que compõem uma Rede de Sensores sem Fio (RSSF), a fim de diminuir custos de implementação. As RSSFs são, basicamente, redes sem fio compostas por dispositivos autônomos dotados de sensores, os quais são distribuídos espacialmente com o intuito de monitorar o ambiente no qual estão imersos. A RSSF proposta neste trabalho utiliza três tipos de dispositivos: *end device, router* e *coordinator*. A principal diferença entre eles é que um *end device* não é capaz de rotear tráfego, enquanto *routers* têm essa funcionalidade adicional. O dispositivo *coordinator,* além dessas características, também é responsável por formar a rede e configurá-la de forma adequada. A rede é composta, dessa forma, por apenas um dispositivo *coordinator,* que se comunica diretamente com os *routers* dispostos em seu alcance de comunicação e por *end devices,* que por sua vez se ligam aos *routers.*

Diante disso, a aplicação prática proposta neste trabalho sugere a implementação de uma RSSF capaz de sensoriar uma área específica da Universidade Federal do Piauí com a finalidade de obter médias de temperatura naquela região e, a partir desses dados, alertar à estação base qualquer anomalia nas médias que indique temperaturas relativamente altas. Com a finalidade de monitorar a maior área possível usando o mínimo de dispositivos

sensores, sobre essa região, serão mapeadas localidades distintas que representarão os possíveis pontos de alocação dos nós *routers,* todos ao alcance do nó central *coordinator* (15 a 30 metros).

Essas localidades serão representadas como vértices no grafo desta aplicação. Cada aresta do grafo ligará dois vértices que se encontram a uma distância de 15 a 30 metros um do outro. Após a construção do grafo e a aplicação do algoritmo desenvolvido para determinar o conjunto independente máximo, espera-se como resultado a menor quantidade de sensores (por evitar a interseção entre a área alcançada por cada sensor) para cobrir a maior área possível (pela busca do conjunto independente máximo) dentro da abrangência do nó central. Dessa forma, serão utilizados os recursos mínimos para a implementação do projeto.

3. Implementação de uma instância do problema

A instância inicial do problema é um cenário fictício de uma Rede de Sensores Sem Fio (RSSF) situada no Centro de Ciências da Educação (CCE) da Universidade Federal do Piauí, com o propósito de obter médias de temperatura das localidades onde os nós se encontram. Qualquer anomalia nas médias de temperatura é alertada à estação base.

A aplicação do algoritmo de identificação do CMI nesta instância se justifica em um cenário em que há um conjunto de salas no CCE com patrimônios de grande valor e, portanto, precisam ser monitoradas, a fim de evitar prejuízos. A UFPI não possui à disposição um grande orçamento para compra de nós sensores, portanto, é necessário definir a localização de cada nó sensor de forma a monitorar a maior quantidade de salas possível. Além disso, para minimizar a redundância dos dados coletados, o que ocasiona no consumo desnecessário de energia pelos sensores, é preciso instalá-los de maneira que não haja nós sensores cobrindo a mesma região. A representação da instância inicial do problema em questão pode ser observada na forma de um grafo na Figura 1.

O vértice triangular na figura representa a estação base, que está ao alcance de todos os nós da rede e é imprescindível para o funcionamento da aplicação, portanto, não fará parte da instância do problema a ser resolvido com o algoritmo de CMI. Os vértices circulares são as localidades candidatas em que os nós da RSSF podem ser instalados, os quais poderão ser reduzidos após a aplicação do algoritmo. As arestas indicam em quais localidades teriam raios de atuação sobrepostos caso tivessem sensores instalados nas posições indicadas pelos nós circulares – esses raios de atuação variam entre 15 e 30 metros. É justamente a relação dada por essa sobreposição que justifica o uso do algoritmo de CMI no problema.

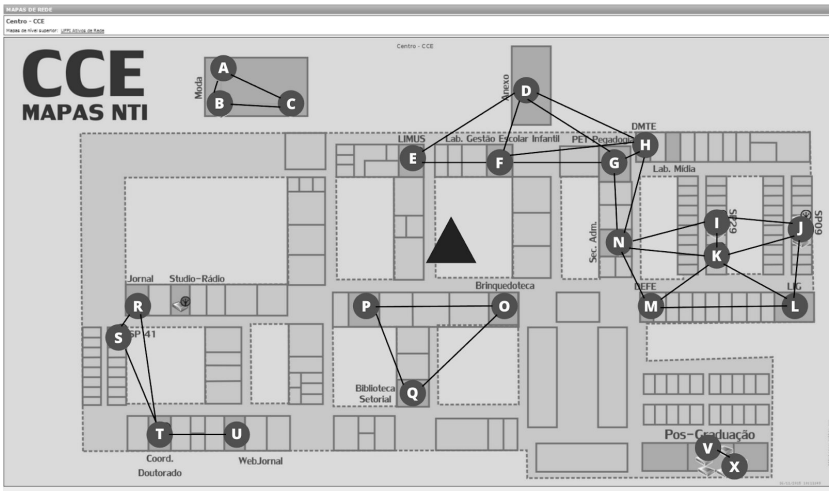

Figura 1. Representação da instância do problema através de um grafo, que será resolvido com CMI.

Com o objetivo de tornar o problema ilustrado na Figura 1 uma entrada compatível com o código desenvolvido (na linguagem de programação *Python*), tal grafo foi estruturado por meio da estrutura de dados *Dicionário* e assume a forma mostrada na Figura 2.

```
graph = {"a": ["b", "c"],
         "b": ["a", "c"],
         "c": ["a", "b"],
         "d": ["e", "f", "g", "h"],
         "e": ["d", "f"],
         "f": ["e", "d", "g", "h"],
         "g": ["d", "f", "h", "n"],
         "h": ["d", "f", "g", "n"],
         "i": ["j", "k", "n"],
         "j": ["i", "k", "l"],
         "k": ["i", "j", "l", "m", "n"],
         "l": ["j", "k", "m"],
         "m": ["k", "l", "n"],
         "n": ["i", "g", "h", "k", "m"],
         "o": ["p", "q"],
         "p": ["o", "q"],
         "q": ["o", "p"],
         "r": ["s", "t"],
         "s": ["r", "t"],
         "t": ["r", "s", "u"],
         "u": ["t"],
         "v": ["x"],
         "x": ["v"]
        }
```

Figura 2. Representação da instância do problema

Cada chave do dicionário, encontrada do lado esquerdo da associação, corresponde a um vértice do problema. O lado direito contém uma lista com todos os outros vértices com os quais aquele vértice está associado. Esses dados são inseridos no dicionário <*graph*>, presente no arquivo *mis.py*, no diretório "maximum-independent-set-algorithm" do arquivo fonte deste trabalho. Uma vez efetuadas as alterações, basta salvar o arquivo, executar o prompt de comando no diretório em que o arquivo *mis.py* está armazenado e executar o comando <*python3 mis.py*>. Vale ressaltar que, para essa execução, é necessário que o interpretador *Python 3* e as bibliotecas *networkx*, *matplotlib, itertools, random* e *time* estejam devidamente disponíveis na máquina.

O resultado da execução são duas imagens de extensão *.png,* salvas no diretório do código fonte, contendo as respostas geradas pelo algoritmo força bruta e pelo algoritmo heurístico.

4. Implementação baseline

O algoritmo simples desenvolvido para a resolução do problema foi implementado com a linguagem *Python* e sua estrutura está descrita nos próximos parágrafos desta seção.

A execução do algoritmo é iniciada com a chamada do procedimento *maxIndSetBF(),* observado na Figura 3. Nele, a função *findsubsets()*, mostrada na Figura 4, é chamada para a geração de uma lista contendo todas as combinações entre os vértices do grafo recebido por parâmetro. As combinações são calculadas com o auxílio da função *itertools.combinations(s, i),* que recebe por parâmetro o grafo a ser analisado e o tamanho das combinações a serem calculadas. Os tamanhos dos subconjuntos variam de *n* até 1 (sendo *n* a quantidade de vértices do grafo analisado). Estes são adicionados em uma lista e, por conta da ordem de execução que a função foi criada, estão ordenados de maneira decrescente em função do tamanho.

Uma vez gerada a lista de combinações, tem-se uma lista de listas. Nestas, encontram-se as combinações de determinado tamanho, isto é, a primeira conterá apenas a combinação de maior tamanho (aquela que contém todos os vértices do grafo), a segunda conterá combinações de tamanho correspondente à quantidade de vértices decrementado em um e assim por diante, até que a última conterá listas compostas por apenas um vértice. Com essa configuração, percorre-se cada um dos subconjuntos e verifica-se se existe alguma aresta entre os vértices de um mesmo subconjunto. Se existir, a variável *flag* é marcada como *True* e, dessa forma, o primeiro subconjunto que finalizar sua iteração e não marcar a *flag* é retornado. Uma vez que a lista está em ordem decrescente de tamanho, o primeiro Conjunto Independente encontrado corresponderá ao Conjunto Máximo Independente. Além disso, quando um grafo vazio é passado como parâmetro para esse algoritmo, ela retornará nulo.

```
def maxIndSetBF(graph):
    subsets_lists = findsubsets(graph)

    for subset_list in subsets_lists:
        for subset in subset_list:
            flag = False
            for vertex in subset:
                for vertex_else in subset:
                    if(vertex_else != vertex):
                        if vertex_else in graph[vertex]:
                            flag = True
                            break
                if(flag):
                    break
            if (not flag):
                return subset
    return None
```

Figura 3. Função *maxIndSetBF()*

```
def findsubsets(s):
    subsets = []
    for i in range(len(s), 0, -1):
        subsets.append(list(itertools.combinations(s, i)))
    return subsets
```

Figura 4. Função *findsubsets()*

O resultado do algoritmo para o problema proposto pode ser visualizado na Figura 5. Como esperado, o algoritmo retorna um grafo que corresponde ao conjunto independente máximo encontrado.

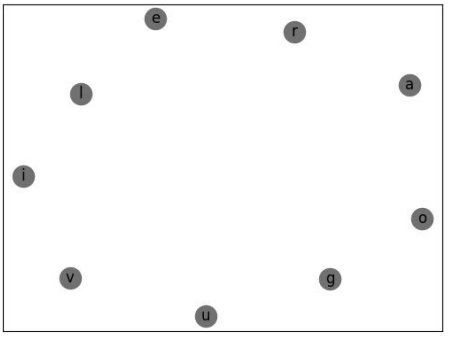

Figura 5. Conjunto Máximo Independente calculado pelo algoritmo força bruta

A complexidade do algoritmo no pior caso corresponde a $O(2^n \times n^2)$. O termo 2^n corresponde à quantidade de combinações de vértices do grafo analisado, enquanto n^2 corresponde à quantidade de comparações necessária para verificar se existem arestas entre os vértices de cada subconjunto.

5. NP-completo

Para provar que um problema é NP-completo, é necessário mostrar que ele pertence tanto à classe dos problemas NP quanto à dos NP-difícil. O Conjunto Máximo Independente (CMI) pode ser caracterizado como um problema de otimização, visto que o objetivo é encontrar o maior conjunto de vértices tais que esses não possuam arestas em comum.

Para desenvolver a prova de que o CMI é da classe NP-completo, é preciso restringi-lo a um problema de decisão. Essa simplificação se justifica pelo fato de que, para o estudo teórico da complexidade de algoritmos, considera-se problemas cujo resultado da computação seja "sim" ou "não", o que caracteriza um problema de decisão. Tais problemas, quando apresentam uma dada solução que é verificada facilmente (em tempo polinomial), satisfazem a característica da classe NP.

No caso do CMI, é possível simplificá-lo para um problema de Conjunto Independente Grande (CIG): dado um grafo *G* e um número *k,* verificar se G possui um conjunto independente com *k* ou mais vértices. Portanto, se for provado que a simplificação do problema de otimização faz parte da classe NP-completo, fica demonstrado que o original também fará.

O primeiro passo para provar que um problema é NP-completo é mostrar que ele é NP. Para um problema ser NP, deve ser possível verificar uma solução para uma instância sua em tempo polinomial. Ou seja, no caso do CIG, dado um grafo *G* e um conjunto *V* de vértices, deve ser possível verificar se *V* é independente e possui *k* ou mais vértices. Para verificar isso, basta contar a quantidade de elementos do conjunto e percorrê-lo, verificando se há uma aresta ligando algum par de vértices do conjunto. Isso pode ser verificado em tempo polinomial. Logo, é possível afirmar que CIG faz parte da classe de problemas NP.

O segundo passo é provar que o problema também é NP-difícil. Para isso, é necessário que ele seja pelo menos tão difícil quanto qualquer outro problema pertencente à classe NP. Para efeitos de simplificação, o segundo passo será válido ao provar que, para um problema X, existe um problema pertencente à classe dos NP-completo tal que este seja polinomialmente redutível a X. Ou seja, é possível transformar, em tempo polinomial, um problema NP-completo *P1* em um problema *P2*, solucioná-lo e transformar, também em tempo polinomial, a solução obtida para *P2* em uma solução para *P1*.

No caso do CIG, o problema NP-completo conhecido que será utilizado é a Clique: dado um grafo *G*, um subconjunto de vértices *V* forma uma clique se todo par de vértices é adjacente. O problema de decisão seria, dado um grafo *G* e um número inteiro *k* maior que zero*,* verificar se existe uma clique no grafo com ao menos *k* vértices.

A Figura 6 apresenta um grafo *G* e seu grafo complementar *G'*. Este é obtido a partir da remoção das arestas existentes em *G* e da inserção de arestas que ligam os vértices que não estavam conectados. Dado *k* igual a 3, é possível obter um CIG de tamanho três no grafo complementar *G'*. Na instância exibida na figura, os vértices em destaque (*b*, *c*, *d*) no grafo complementar *G'* formam um CIG, pois não existem arestas entre eles. Uma vez obtida uma solução para o conjunto independente grande em G', esta corresponde a uma solução para o problema original (Clique) em G.

Nesse sentido, a transformação polinomial de *P1* para *P2* se justifica pelo fato de que é possível obter G' em tempo polinomial ($O(V^2)$) e, além disso, se *G'* possui um CIG de tamanho maior ou igual a k, *G* possui uma clique do mesmo tamanho.

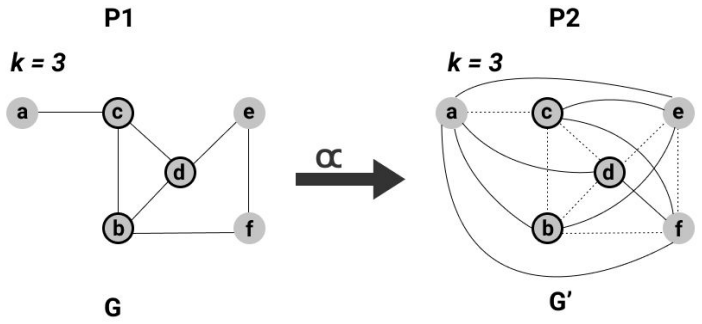

Figura 6. Transformação polinomial *P1∝P2*

Portanto, uma vez que uma Clique, problema conhecidamente NP-completo, é redutível polinomialmente a um CIG e esse se caracteriza como um problema NP, conclui-se que o CIG é um problema NP-completo. Por fim, pelo fato de um CIG se tratar de uma simplificação do problema de otimização conhecido como CMI, constata-se, por consequência, que o Conjunto Máximo Independente é um problema NP-completo.

6. Implementação do algoritmo heurístico

Com o objetivo de obter uma solução ótima para instâncias grandes do problema de CMI em um tempo aceitável, foi implementado um algoritmo heurístico guloso. Este percorre todos os vértices da instância, que compõem o conjunto de candidatos, e decide a solução ótima local verificando se um determinado vértice deve pertencer ao conjunto solução ou não. Após a decisão, o algoritmo resolve, recursivamente, os subproblemas gerados. Uma vez decidido, não haverá reconsideração. Espera-se que, determinando soluções ótimas locais, o resultado final corresponda à solução ótima global. Nesse sentido, o restante desta seção trata dos detalhes de implementação da abordagem.

A execução do algoritmo inicia com a chamada da função *maxIndSetHA(),* apresentada na figura 7, que recebe como argumento um grafo estruturado em um dicionário da linguagem de programação *Python* - especificado na seção 3. Nessa estrutura, cada vértice é representado por uma chave. Cada chave está associada a uma lista com todos os vértices que se conectam a ela através de uma aresta.

```
def maxIndSetHA(graph):
    v = degree(graph)

    if(v == -1):
        return graph

    graph1 = copy.deepcopy(graph)
    graph2 = copy.deepcopy(graph)

    n1 = delete(graph1, v)
    n2 = deleteNeighbors(graph2, v)

    return maxIndSetHA(n1) if(len(n1) > len(n2)) else maxIndSetHA(n2)
```

Figura 7. Função *maxIndSetHA()*

Cada vez que essa função é executada, é buscado o vértice de maior grau existente no grafo recebido como argumento. Isso é feito através da chamada da função *degree(),* representada na Figura 8, cujo retorno é o vértice de maior grau encontrado ou -1, caso todos

os vértices possuam grau 0. Uma vez identificado o vértice de maior grau, duas possibilidades são verificadas: deletar esse vértice do grafo e consequentemente todas as arestas relacionadas a ele ou remover do grafo todos os vizinhos desse vértice e suas respectivas arestas.

```
def degree(graph):
    l = 0
    v = -1
    for i in graph.keys():
        if(len(graph[i]) > l):
            l = len(graph[i])
            v = i
    return v
```

Figura 8. Função *degree()*

A primeira possibilidade é executada obtendo um novo grafo a partir da chamada da função *delete(),* exibida na Figura 9. Essa função remove um vértice de um grafo, ambos passados por parâmetro. Além disso, a função percorre as listas correspondentes às arestas dos demais vértices do grafo e, se houver uma referência àquele vértice removido anteriormente, ela é removida também.

```
def delete(graph, v):
    graph.pop(v)
    for i in graph.keys():
        for j in graph[i]:
            if(j == v):
                graph[i].remove(j)
    return graph
```

Figura 9. Função *delete()*

A segunda possibilidade é através da obtenção de um novo grafo com a execução da função *deleteNeighbors()*, apresentada na Figura 10. Essa função percorre toda a lista de vizinhos do vértice passado a ela como parâmetro e faz uma chamada da função *delete()* para remover do grafo todos os vértices vizinhos encontrados.

```
def deleteNeighbors(graph, v):
    for i in range(0, len(graph[v])):
        graph = delete(graph, graph[v][0])
    return graph
```

Figura 10. Função *deleteNeighbors()*

Com os dois grafos obtidos, a função *maxIndSetHA()* retornará uma chamada a ela mesma. Desta vez, o grafo passado como parâmetro será aquele obtido através da primeira estratégia caso ele tenha uma quantidade maior de vértices ou o grafo obtido com a segunda abordagem, caso contrário.

Esse procedimento é realizado de maneira recursiva até que restem somente vértices que não possuam nenhum vizinho, ou seja, até que a função *degree()* retorne -1. Dessa maneira, a função principal *maxIndSetHA()* devolve o grafo resultante.

O resultado do algoritmo para o problema proposto pode ser visualizado na Figura 11.

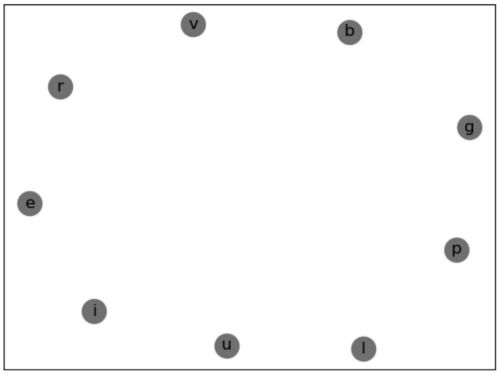

Figura 11. Conjunto Máximo Independente calculado pelo algoritmo heurístico

A complexidade do algoritmo no pior caso corresponde a $O(n^4)$. A função *delete()* tem complexidade $O(n^2)$, *deleteNeighbors()* tem $O(n^3)$ e *degree()* tem $O(n)$. Nesse sentido, a equação de recorrência da função *maxIndSetHA()* corresponde a $T(n) = T(n-1) + n^3 + n^2 + n$, onde *T(n - 1)* se refere à chamada recursiva presente na função, pois a cada chamada um vértice é deletado do grafo. Seguindo a regra das somas, a equação final de recorrência pode ser simplificada da seguinte forma: $T(n) = T(n-1) + O(n^3)$. Considerando o pior caso, haverá n chamadas recursivas da função, logo, a complexidade da mesma é $O(n^4)$.

7. Testes de desempenho

Os algoritmos foram implementados na linguagem de programação *Python*, versão 3.8.5. O sistema operacional no qual os testes foram executados é o *Ubuntu 20.04.1 LTS*, do tipo 64 bits. O computador possui o processador *AMD Ryzen 3 3200g* (4 núcleos e 3600 MHz), duas memórias RAM do tipo DDR4 de tamanho 8192 MB e velocidade de 3200 MHz.

Os algoritmos força bruta e heurístico foram executados três vezes com o objetivo de analisar o tempo de execução para grafos com diferentes quantidades de vértices: 4, 8, 12, 16, 20 e 24. O tempo de execução obtido é resultado da média aritmética das três execuções. Para análise da qualidade das soluções, os algoritmos foram executados uma vez e a quantidade de vértices obtida na resposta é verificada.

7.1. Tempo de execução

O tempo de execução dos algoritmos implementados foram medidos para grafos gerados aleatoriamente. A Tabela 1 e Figura 12 mostram os resultados obtidos em milisegundos. Observa-se que o tempo de execução do algoritmo força bruta aumenta de forma exponencial, enquanto a abordagem heurística tem crescimento polinomial (quarta potência).

Tabela 1. Comparativo dos algoritmos Força Bruta e Heurístico quanto a tempo de execução.

Quantidade de Vértices do Grafo	Tempo médio de Execução do Algoritmo Força Bruta (ms)	Tempo médio de Execução do Algoritmo Heurístico (ms)	Razão AH/AFB
4	0,0093	0,0324	348,03%
8	0,1039	0,2749	264,67%
12	1,5104	0,6333	41,93%
16	16,1115	0,3755	2,33%
20	426,3495	2,9330	0,69%
24	8661,9539	2,6886	0,03%

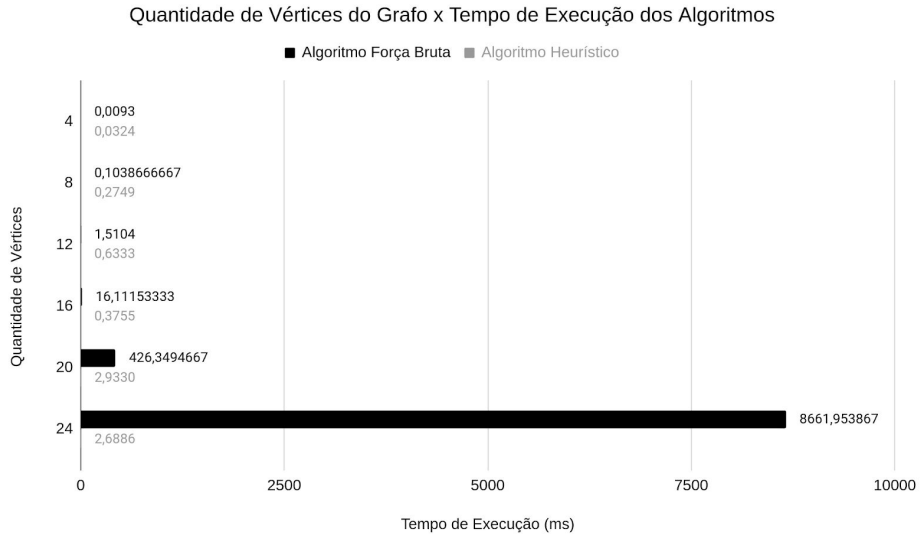

Figura 12. Gráfico do tempo de execução em função da quantidade de vértices do grafo.

7.2. Qualidade da solução

Ao adotar uma heurística gulosa como alternativa para solução do problema em tempo hábil, a otimalidade da solução não é garantida para todos os casos. O objetivo desta seção é demonstrar que existem instâncias em que a heurística escolhida não resulta na solução ótima, ou seja, no maior conjunto independente possível.

Na Tabela 2, que compara os resultados dos dois algoritmos para instâncias geradas aleatoriamente, observa-se que na maioria dos testes realizados o algoritmo heurístico obtém um CMI para o problema, uma vez que encontra um conjunto independente com a mesma quantidade de vértices da solução ótima garantida pelo algoritmo força bruta. No entanto, para determinadas instâncias, o algoritmo heurístico não consegue resultar em uma solução ótima, como é o caso da instância de 20 vértices.

Tabela 2. Comparativo dos tamanhos dos conjuntos independentes gerados pelos algoritmos Força Bruta e Heurístico.

Quantidade de Vértices do Grafo	Tamanho do Conjunto Independente resultante - Algoritmo Força Bruta	Tamanho do Conjunto Independente resultante - Algoritmo Heurístico
4	3	3
8	5	5
12	9	9
16	7	7
20	7	6
24	4	4

A Figura 13 apresenta o resultado provido pelos algoritmos força bruta e heurístico dado um mesmo grafo original. Como mencionado na seção 6, o algoritmo heurístico sempre escolhe o vértice de maior grau para, somente após essa seleção, realizar a análise e a decisão de remoção ou inclusão desse vértice no grafo correspondente à solução final. Em casos onde mais de um vértice apresenta grau máximo, o critério de decisão corresponde à ordem de

aparição desses vértices; para um grafo onde eles estão em ordem alfabética, o primeiro deles nessa ordem é o escolhido.

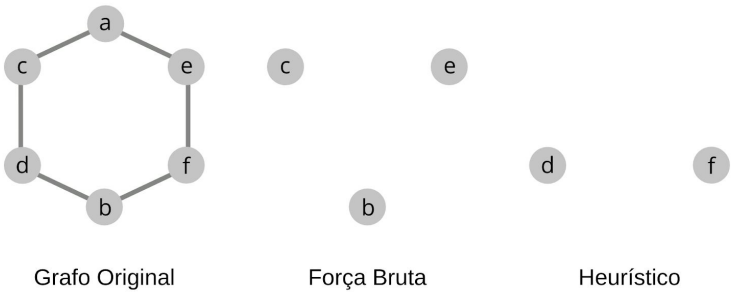

Figura 13. Teste de otimalidade entre FB e H.

Portanto, para o grafo original apresentado na Figura 13, a sequência de remoção de vértices pelo algoritmo heurístico será: $a \rightarrow b \rightarrow c \rightarrow e$. Isso resulta, como ilustrado pela mesma figura, em um conjunto independente formado somente pelos vértices *d* e *f*. O algoritmo força bruta, por sua vez, devido ao fato de sempre testar todas as possibilidades, resulta em um conjunto independente formado por três vértices (*b*, *c* e *e*, no exemplo) que corresponde, para o grafo original em questão, a um conjunto máximo independente.

8. Conclusão

Partindo de uma aplicação prática para o problema do Conjunto Máximo Independente, foi possível, neste trabalho prático, através da aplicação dos conceitos sobre classes de problemas, verificar que o CMI pertence à classe NP-Completo. Como abordagem inicial, foi implementado um algoritmo *baseline* adotando a estratégia força bruta para resolução do problema; a partir dele foi possível constatar que o seu tempo de execução cresce exponencialmente em função do tamanho da entrada.

Uma heurística gulosa foi escolhida como alternativa, onde a garantia de otimalidade foi abdicada em prol da necessidade de se obter resultados em tempo polinomial. Tendo ambas as abordagens implementadas, foi possível averiguar a otimização do tempo despendido para a resolução do problema, através da comparação das execuções dos algoritmos heurístico e força bruta. Por fim, ao confrontar o conjunto independente retornado por ambos, foi possível evidenciar as mínimas diferenças entre a otimalidade das respostas.

9. Bibliografia

Cormen, Thomas H. et al. (2009) "**Introduction to algorithms",** MIT press.