Maximum Weighted Cut – Análise de Soluções de Otimização

Guilherme Antunes

***Resumo* – O problema do corte máximo ponderado consiste em encontrar uma linha que corte arestas de um grafo, separando o mesmo em dois subconjuntos complementares, e que maximize o valor desse corte que é dado pela soma dos pesos das arestas cortadas. Neste relatório são exploradas duas abordagens principais: uma de pesquisa exaustiva e uma de pesquisa voraz, sendo que são abordadas duas heurísticas com a abordagem de pesquisa voraz. Foi realizada uma análise formal e experimental de cada uma delas de forma a compará-las quanto aos parâmetros considerados.**

***Abstract* – The problem of weighted maximum cut consists of finding a line that cuts multiple edges of a graph, separating it into two complementary subsets, and maximizing the value of this cut, which is given by the sum of the weights of the cut edges. This report explores two main approaches: an exhaustive search and a greedy search, with two heuristics being addressed within the greedy search approach. A formal and experimental analysis of each approach was conducted to compare them in terms of the considered parameters.**

Keywords – Maximum Weighted Cut, Exhaustive Search, Greedy Search, Greedy Heuristics, Graphs

I. INTRODUÇÃO

O problema de otimização Maximum Weighted Cut, ou Corte Máximo Ponderado, consiste em, para um dado grafo ponderado não orientado $G(V, E)$ com n vértices e m arestas, queremos determinar qual é o corte que divide o grafo G em dois conjuntos complementares S e T e que maximize a soma dos pesos das arestas que conectam os conjuntos S e T [1]. Este problema também pode ser descrito como a solução que maximiza a seguinte função [3]:

$$\sum_{i<k}^{n} w_{ik} - \sum_{i<k}^{n} w_{ik} z_i z_k$$

$$\begin{cases} z_i = +1, i \in S \\ z_i = -1, i \in T \end{cases}$$

Onde:

- n é o número de vértices no grafo G e $1 < [i\,, k] < n$
- w_{ik} é o peso da aresta que liga os vértices i e k
- z_i é um valor arbitrário que varia mediante o subconjunto em que o vértice se encontra

Podemos observar que ao somatório dos pesos de todas as arestas é removido o somatório dos pesos das arestas que conectam os diferentes subconjuntos S e T. O segundo somatório não pode ser 0 uma vez que isso significaria não existir um corte, tornando S igual a G. Esta função é o resultado da decomposição da seguinte:

$$\sum_{ik}^{n} \frac{1 - z_i z_k}{2} w_{ik}$$

Aqui é possível perceber que a variação de z anula ou aplica ao somatório final o peso da aresta que liga os vértices i e k, valorizando assim as arestas que conectam os dois subconjuntos ao invés das que estão dentro dos mesmos.

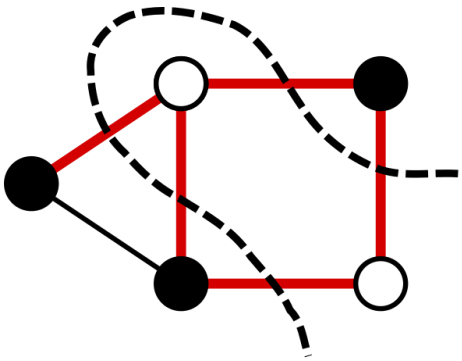

Fig. 1 – Exemplo visual da aplicação do Maximum Weighted Cut a um grafo

Observando a Fig. 1 e aplicando o raciocínio enunciado, assumindo que este será o nosso grafo G os vértices brancos fariam parte do subconjunto S e os pretos fariam parte do subconjunto T, tendo os vértices de cada conjunto polaridades opostas, ajudando-nos a determinar o corte máximo representado pela linha a tracejado. Se assumirmos que todas as arestas da Fig. 1 têm o mesmo peso podemos observar que o traçado do corte maximiza o número de arestas por onde este passa de forma a formar dois subconjuntos complementares.

A fim de estudar a resolução deste problema para qualquer grafo $G(V, E)$ foram consideradas 2 abordagens principais:

pesquisa exaustiva e pesquisa voraz. Estes algoritmos foram desenvolvidos em Python 3.12 e o seu código encontra-se no script `run.py` que segue em anexo a este documento.

II. Pesquisa Exaustiva

Um algoritmo de pesquisa exaustiva consiste precisamente em explorar exaustivamente todas as possibilidades de solução de um problema para encontrar a melhor possível. Para este problema em específico iremos explorar todos os pares de conjuntos complementares S e T de forma a encontrar aquele que maximiza o valor do corte para o grafo G.

Assim, a abordagem definida para explorar todas as possibilidades existentes em cada grafo consistiu em percorrer todos os vértices do grafo, calcular todas as combinações possíveis entre esse vértice e os restantes do grafo principal e para cada combinação é calculado o peso do corte necessário para formular o subconjunto correspondente a esta. Caso o peso do corte seja superior ao do último corte máximo registado então o conjunto de maior corte é substituído pelo atual assim como o próprio valor do corte máximo. Por fim tomamos como melhor solução a que vence todas as comparações a que é sujeita. Para colocar em prova este conceito foi desenvolvido e utilizado o código seguinte:

```
def exhaustive_search(G):
    nodes = G.nodes
    n = G.number_of_nodes()
    max_cut_weight = 0
    max_cut = None

    for i in range(n):
        for subset in combinations(nodes, i):
            cut_weight = nx.cut_size(G, subset, weight='weight')
            if cut_weight > max_cut_weight:
                max_cut_weight = cut_weight
                max_cut = subset

    return max_cut_weight, max_cut
```

Como já referido, este algoritmo é o único que calcula sempre a melhor solução possível para o grafo a que é submetido, no entanto torna-se demasiado dispendioso quer em termos computacionais quer em termos de tempo para grafos mais complexos.

A. Análise Formal

A complexidade do algoritmo utilizado é $O(2^n)$ uma vez que os dois *for* em conjunto apresentam uma complexidade de aproximadamente $2^n \times n$, já que estamos a multiplicar um *for* de complexidade $O(n)$ e os resultados da função que gera as combinações de vértices que tem complexidade $O(2^n)$. A função *nx.cut_size()*, para grafos não orientados, tem complexidade $O(n^2)$ já que corresponde à complexidade de calcular o número de arestas que é dado pela expressão $\frac{n\times(n-1)}{2} \times p, p \in [0,1]$ sendo p a densidade de arestas relativa ao número máximo possível delas no grafo e n a quantidade de vértices no grafo. Assim espera-se um crescimento exponencial dos recursos consumidos pelo programa.

Esta complexidade aplica-se para o caso melhor, pior e médio uma vez que um algoritmo de pesquisa exaustiva percorre todas as possibilidades, alcançando sempre o pior caso.

B. Análise Experimental

Para verificar os resultados estipulados no processo de análise formal foram gerados grafos com uma quantidade de vértices a variar entre 2 e 20, arestas com pesos aleatórios inteiros entre 1 e 100 e com uma densidade de 12,5%, 25%, 50% e 75% relativamente ao número máximo de arestas que poderiam existir num grafo com *n* vértices. Os vértices foram gerados sem ter em conta os limites de coordenadas estabelecidos entre 1 e 100 para números inteiros, ao invés foram gerados vértices livres de coordenadas, libertando-os de questões de proximidade geográfica, sendo grafos livres do conceito de espaço físico como o conhecemos. Para garantir que o número de vértices nunca era superior ao estipulado pelo problema nunca foram gerados grafos com mais de 10000 vértices.

Foram tidas em conta três métricas para avaliação: o tempo de execução, o número de configurações/soluções testadas e o número de operações básicas. Neste algoritmo o número de iterações é o mesmo que o número de soluções testadas uma vez que a cada iteração se testou uma solução diferente. As operações básicas a ter em conta para este algoritmo foram as comparações, sendo que existe uma para cada solução testada. O tempo de execução acabou por limitar o número máximo de vértices a testar em 20, para este algoritmo, uma vez que os tempos de execução a partir deste ponto são demasiado elevados para serem aprazíveis. O gráfico do tempo de execução foi realizado sem as contagens referidas anteriormente, de forma a minimizar o tempo de execução para a pesquisa em cada grafo.

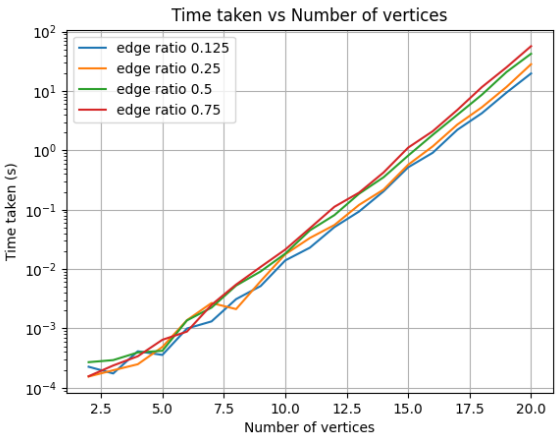

Figura 2 - Pesquisa Exaustiva: Tempo de execução por grafo em escala logarítmica

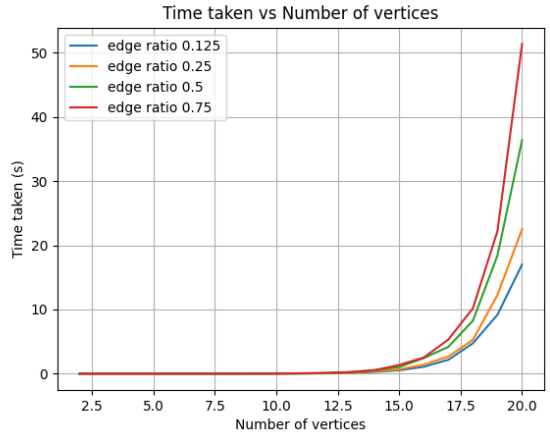

Figura 3 - Pesquisa Exaustiva: Tempo de execução por grafo em escala linear

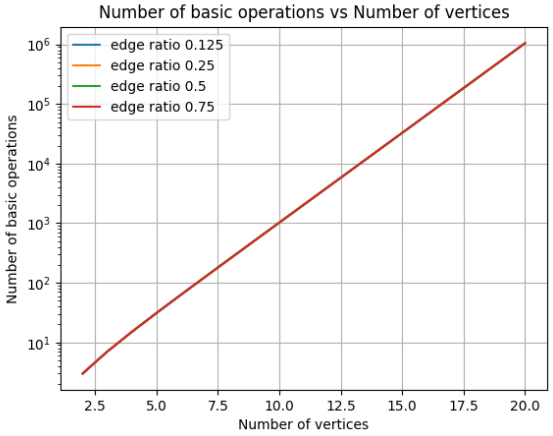

Fig. 4 – Pesquisa Exaustiva: Número de operações básicas por grafo

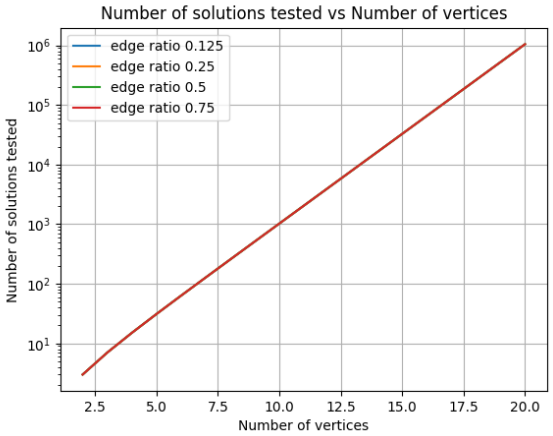

Fig 5 - Pesquisa Exaustiva: Número de soluções testadas por grafo

Todos os gráficos usam uma escala logarítmica no eixo y, excepro o da figura 3, sendo que é possível verificar um crescimento exponencial em cada um dos gráficos já que este se assemelha a um crescimento linear, mas numa escala de logaritmo. Este crescimento exponencial é visível na figura 3 que corresponde aos mesmos dados da figura 2, mas com escala linear no eixo de y.

Como referido também é possível visualizar que nas figuras 4 e 5 o número de configurações é exatamente o mesmo que o número de operações básicas, assim como para qualquer densidade de arestas, o que revela que a variação destes valores é independente do número de arestas e dependente do número de vértices. Também na figura 2 é possível observar que apesar dos grafos com o mesmo número de vértices terem densidades de arestas diferentes a forma como as linhas crescem são semelhantes.

A operação que não está a ser tida em conta, *nx.cut_size()*, por sua vez tem tanto mais iterações quanto mais arestas o grafo tiver. No entanto como todas essas iterações são realizadas para produzir um valor único não verificamos alterações no número de operações básicas e apenas no tempo que o algoritmo demora a executar.

III. Pesquisa voraz

Com o objetivo de encontrar soluções para grafos maiores, com mais vértices e arestas, e para comparar as performances do algoritmo de pesquisa exaustiva foram desenvolvidos 3 algoritmos de pesquisa voraz, sendo usadas 2 heurísticas diferentes e uma variação de uma delas com pré-processamento dos dados. Enquanto os primeiros dois algoritmos foram desenvolvidos com uma heurística bastante básica e desenvolvida apenas para prova de conceito, que será referida por Simple Greedy, a segunda heurística foi retirada do artigo *On greedy construction heuristics for the MAX-CUT problema* [2], nomeadamente a 3ª variação da aproximação do algoritmo de Sahni-Gonzalez, que será referida como SG3.

IV. Pesquisa voraz – Simple Greedy

Para este algoritmo foi aplicada a seguinte estratégia: começar com o grafo completo, adicionar um vértice, sem critério específico, a um subconjunto, calcular o peso do corte que definirá esse subconjunto, caso o valor do corte seja maior que o último máximo registado o subconjunto feito pelo corte é atualizado, assim como o valor do mesmo. Em caso negativo o vértice é removido do corte, mantendo apenas assim os vértices, que quando são adicionados, garantam a maximização do corte.

O código seguinte implementa este raciocínio:

```
def simple_greedy_search(G):

    nodes = G.nodes
    max_cut_weight = 0
    max_cut = None
    cut_weight = 0
    cut = set()

    for node in nodes:
        if node not in cut:
            cut.add(node)
            cut_weight = nx.cut_size(G, cut, weight='weight')
            if cut_weight > max_cut_weight:
                max_cut_weight = cut_weight
                max_cut = cut.copy()
```

```
        else:
            cut.remove(node)

    return max_cut_weight, max_cut
```

Este algoritmo consegue resolver grafos até 256 vértices e com densidades de arestas até 75% com um tempo de execução sempre abaixo de 2 segundos. No entanto é de realçar que, assim como esperado, o algoritmo não encontra sempre a melhor solução para o problema, e por isso tratamo-la apenas como ótima.

Tomando como ponto importante o facto de o resultado não ser efetivamente o melhor é importante tentar retornar o melhor resultado, tentando sacrificar o menor tempo de execução possível. Posto isto, reutilizando a mesma heurística foi implementado um pré-ordenamento dos *nodes*. Este ordenamento consistiu em ordená-los por ordem decrescente do valor da soma do peso das arestas que o conectam a outros vértices. Desta maneira pretende-se procurar entre as arestas mais "apetecíveis" ao corte antes de procurar entre as que têm peso menor. Assim no código anterior *nodes* é inicializado da seguinte maneira:

```
nodes = sorted(G.nodes, key=lambda x: sum([G.get_edge_data(x, y).get('weight', 0) for y in G.adj[x]]), reverse=True)
```

Assim como no algoritmo voraz sem pré-ordenamento não é garantido que aplicar este processo vá aproximar o valor obtido do melhor, no entanto, uma vez que quando um vértice é excluído não volta a ser considerado para o corte, ordená-los pelo seu potencial valor pode ajudar a minimizar a diferença para o resultado efetivamente melhor.

A. Análise Formal

Como é possível concluir pelo código a complexidade do algoritmo é $O(n^3)$ uma vez que apenas temos um *for* até n, número de vértices, e a função *nx.cut_size()* cuja sua complexidade $O(n^2)$ já foi explicada na análise do algoritmo de pesquisa exaustiva. Novamente esta complexidade é a mesma para os melhores, médio e pior caso, já que não existe uma condição que termine o loop antes daquilo que foi programado inicialmente, percorrendo os n vértices do grafo. Espera-se assim um crescimento cúbico dos recursos consumidos, claramente inferior ao crescimento gerado pelo algoritmo de pesquisa exaustiva.

B. Análise Experimental

Os grafos a usar para teste foram gerados da mesma maneira que para o algoritmo de pesquisa exaustiva, com a exceção de que para este algoritmo podemos avaliar confortavelmente grafos até 256 vértices, uma vez que a execução do algoritmo para cada grafo é bastante mais rápida e menos complexa que no algoritmo anterior. Às três métricas anteriormente estudadas, para o estudo deste algoritmo iremos também tomar em consideração o valor máximo do corte obtido para cada grafo por cada uma das variações do algoritmo, a fim de comparar as performances entre eles. Dada a rápida execução do algoritmo, os tempos representados nos gráficos contemplam também a contagem do número de soluções e de operações básicas, que será considerado o mesmo tipo que foi no algoritmo anterior, as comparações.

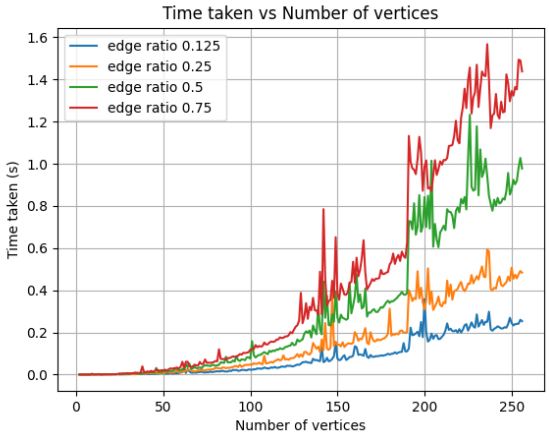

Fig 6 – Greedy Simple: Tempo de execução por grafo

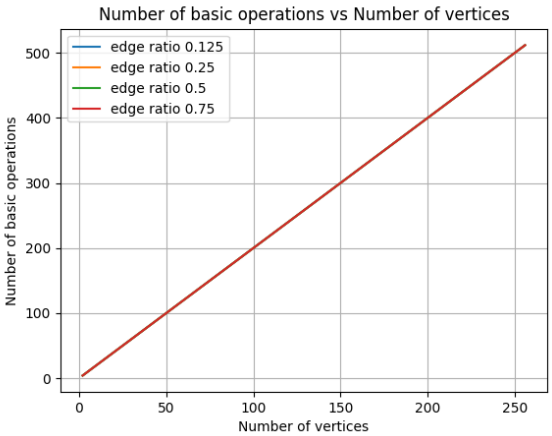

Fig 7 – Greedy Simple: Número de operações básicas por grafo

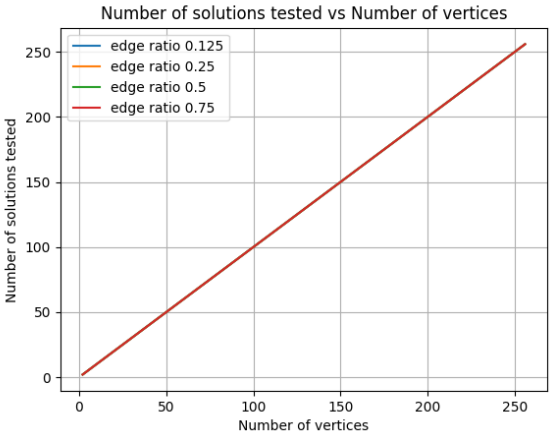

Figura 8 - Número de soluções testadas por grafo

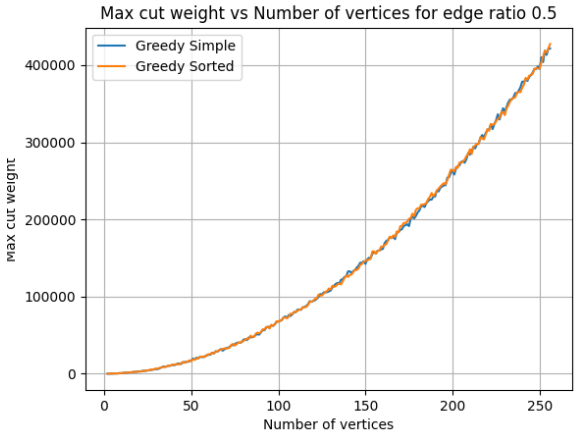

Figura 9 - Valor do corte máximo ponderado por grafo

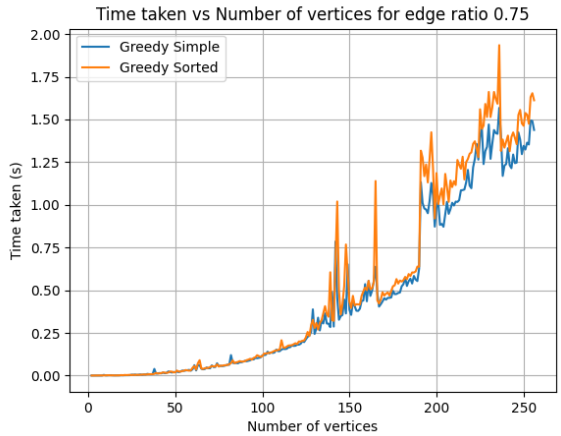

Figura 10 - Tempo de execução por grafo para 0.75 de densidade

Algo que salta claramente à vista, e que já foi referido, é o alcance muito superior que o algoritmo consegue fazer em termos da dimensão dos grafos que consegue avaliar, que é feita num intervalo de tempo radicalmente reduzido em comparação ao algoritmo de pesquisa exaustiva.

Tendo em consideração que todos os gráficos das figuras 6 a 10 têm uma escala linear no eixo y podemos verificar que existe um crescimento cúbico do tempo de execução por grafo representado na figura 6. Sendo que novamente não consideramos as operações que ocorrem dentro da função *nx.cut_size()* as figuras 7 e 8 demonstram o crescimento linear da estatística que representam, respetivamente número de operações básicas e número de configurações testadas, que novamente não varia com o número de arestas, e que condiz com a complexidade do *for* de complexidade n existente no algoritmo.

Analisando o gráfico da figura 9 onde temos representados os valores de corte máximo ponderado obtidos por cada uma das variações desta pesquisa voraz para grafos com 75% de densidade de arestas, podemos verificar visualmente que a linha laranja, do Simple Greedy preordenado, não está clara e tão recorrentemente acima da azul como se planeava aquando a sua implementação. Na tabela seguinte podemos ver inclusive que o valor máximo do corte máximo até é bastante repartido entre ambos. Isto, aliado ao tempo de execução superior que o Greedy Sorted tem e que é possível visualizar na figura 10 para os grafos de densidade de arestas a 75%, configura que a ordenação prévia dos grafos quanto à soma dos pesos das arestas que conectam este a outros não produz resultados satisfatórios e que por isso não compensa o gasto extra de tempo relativamente ao algoritmo base.

Edge Rate	Greedy Simple	Greedy Sorted
0.125	125	121
0.25	121	127
0.5	118	130
0.75	104	145

Tabela 1 - Número de ocorrências do corte máximo ponderado superior por algoritmo por densidade de arestas

V. Pesquisa voraz – Sahni-Gonzalez 3

Como já referido esta heurística foi retirada do artigo *On greedy construction heuristics for the MAX-CUT problem* [2] e adaptada de pseudo-código para Python 3.12. A estratégia enunciada no artigo consiste em selecionar os vértices que são ligados pela aresta de maior peso e colocar cada um num subconjunto diferente, removendo-os assim do grafo principal G, percorrer todos os vértices do grafo G e para cada vértice calcular um valor de score que é dado pelo módulo da diferença entre os pesos das arestas que ligam cada vértice do grafo G a cada um dos subconjuntos. Após o cálculo dos scores para todos os vértices existentes em G é escolhido o vértice que maximiza a diferença entre os pesos das arestas que o ligam a cada um dos subconjuntos. Caso o peso da aresta para S1 seja maior o vértice é adicionado a $S2$, de modo a cortar a aresta de maior peso, e se o peso da aresta para $S2$ for maior então o vértice é adicionado a $S1$. Por fim o vértice é removido do grafo G e é adicionado ao valor do corte máximo o valor da aresta escolhida para ser cortada.

Em termos de código o seguinte algoritmo foi desenvolvido:

```
def sg3_greedy_search(G):
    V = set(G.nodes)
    S1, S2 = set(), set()

    if len(G.edges) == 0:
        return S1, S2, 0

    max_weight_edge = max(G.edges(data=True), key=lambda x: x[2]['weight'])
    x, y = max_weight_edge[0], max_weight_edge[1]
    S1.add(x)
    S2.add(y)
    V.remove(x)
    V.remove(y)

    def get_edge_weight(graph, node1, node2):
        if graph.has_edge(node1, node2):
            edge_data = graph.get_edge_data(node1, node2)
```

```
            return edge_data.get('weight', 0)
        else:
            return 0

    scores = {}
    for i in V:
        scores[i] = 0
        for j in S1:
            scores[i] += get_edge_weight(G,i,j)
        for j in S2:
            scores[i] = abs(scores[i] - get_edge_weight(G,i,j))

    for _ in range(G.number_of_nodes() - 2):
        if len(scores) == 0:
            break
        i_max = max(scores, key=scores.get)
        if max(get_edge_weight(G,i_max,j) for j in S1) > max(get_edge_weight(G,i_max,j) for j in S2):
            S2.add(i_max)
        else:
            S1.add(i_max)
        del scores[i_max]
        for i in V:
            scores[i] = get_edge_weight(G,i,i_max)
        V.remove(i_max)

    return nx.cut_size(G, S1, S2, weight='weight'), S1
```

O código foi adaptado de forma a que os scores sejam todos calculados antes de entrar no loop com mais linhas sendo esse score atualizado sempre que é removido um vértice do grafo original apenas para as arestas que podem conter esse vértice maximizante. Também não é calculado incrementalmente o valor do corte, sendo este calculado apenas no fim do algoritmo pela função *nx.cut_size()*, que corre apenas uma vez. Aquando do cálculo dos pesos das arestas que saem de um vértice para os dois subconjuntos, caso a aresta não exista é considerado um valor de 0 para o peso da aresta no cálculo do score.

A. Análise Formal

Apesar de este algoritmo usar bastantes funções nativas de Python 3.12 a complexidade do algoritmo é de $O(n^2)$ pois o valor de complexidade máximo é atingido dentro do *for* com mais linhas que percorre $n-2$ iterações, e em cada iteração desse *for* é chamada a função *max* que tem complexidade $O(n)$ e outro *for* de complexidade $O(n)$, fazendo $O(2n^2)$. No entanto como n diminui 1 unidade por iteração apenas serão executadas metade das $2n^2$ iterações, configurando assim uma complexidade máxima de $O(n^2)$, média de $O(n^2)$ e mínima de $O(1)$ para quando não existem arestas no grafo. Podemos então esperar um crescimento quadrático, com um gráfico semelhante a um arco de parábola quanto ao consumo de recursos computacionais.

B. Análise Experimental

Repetindo a mesma forma de gerar grafos, mas desta vez até 1024 vértices, beneficiando da eficiência do algoritmo, iremos tomar as mesmas três métricas consideradas no algoritmo de pesquisa exaustiva. À semelhança do que aconteceu na análise de resultados do algoritmo de pesquisa voraz anterior as operações básicas que estão a ser consideradas são as comparações e os tempos de execução apresentados contemplam novamente o cálculo do número de soluções testadas e do número de operações básicas.

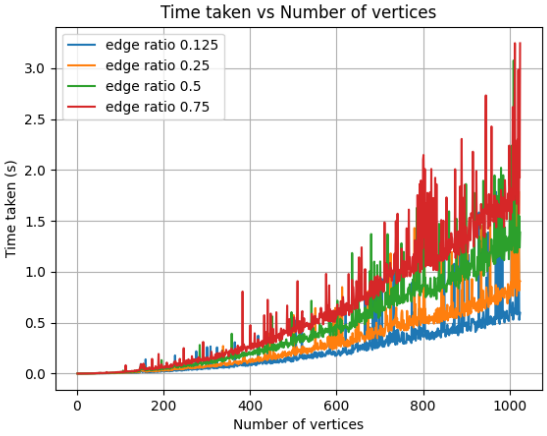

Fig 11 - Sahni-Gonzalez 3: Tempo de execução por grafo

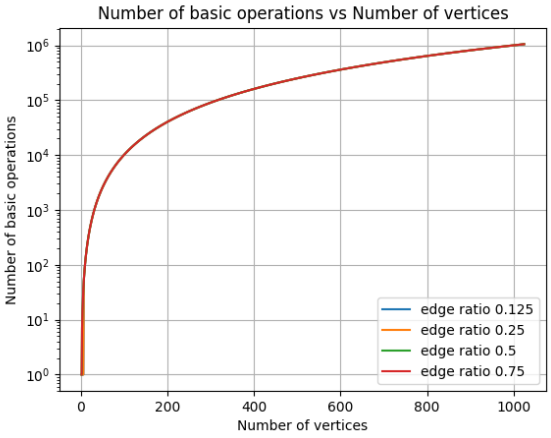

Fig 12 - Sahni-Gonzalez 3: Número de operações básicas por grafo em escala log

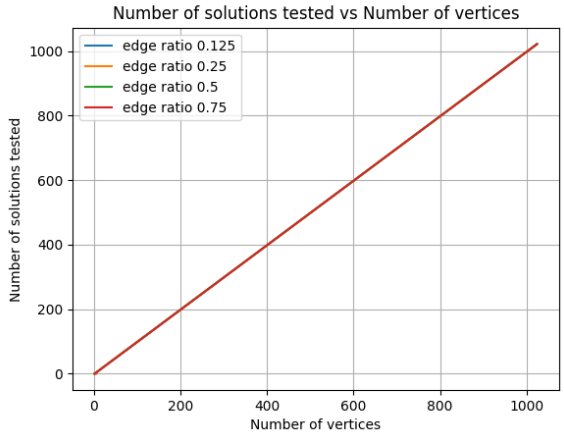

Fig 13 - Sahni-Gonzalez 3: Número de soluções testadas por grafo

Sabendo que as figuras 11 e 13 usam uma escala y linear e a 12 uma escala logarítmica podemos começar por verificar pelo gráfico da variação do tempo de execução que este efetivamente varia conforme uma parábola, comprovando assim o crescimento quadrático previsto. Pela primeira vez a variação do número de operações básicas assemelha-se a uma expressão logarítmica enquanto a variação do número de soluções testadas é diretamente proporcional ao número de vértices numa proporção de 1, como já acontecia no Simple Greedy. Novamente, nenhuma destas métricas depende do número de arestas no grafo assim como verificamos mais uma vez um aumento do alcance do algoritmo já que fornece tempos de execução satisfatórios para grafos com elevados números de vértices.

VI. Exploração de resultados

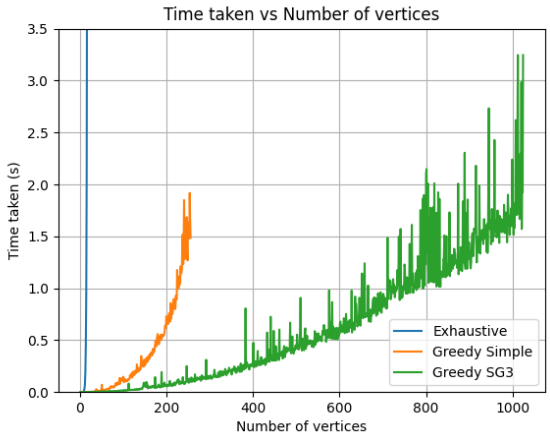

Fig 14 – Variação do tempo de execução para os 3 algoritmos com edge ratio = 0.75

Neste gráfico da figura 14 é claramente percetível a redução do tempo de execução mediante a diminuição da complexidade do algoritmo. O gráfico teve de ser limitado a y =3.5 s para termos uma melhor perceção da forma como os Greedys variam. Sabendo que para $n = 20$ o tempo de execução do algoritmo de pesquisa exaustiva é cerca de 50s é difícil pensar em utilizar este algoritmo como boa solução para problema apenas por uma análise a este gráfico. O único motivo para o fazer continua a ser a garantia de que nos oferece o melhor resultado.

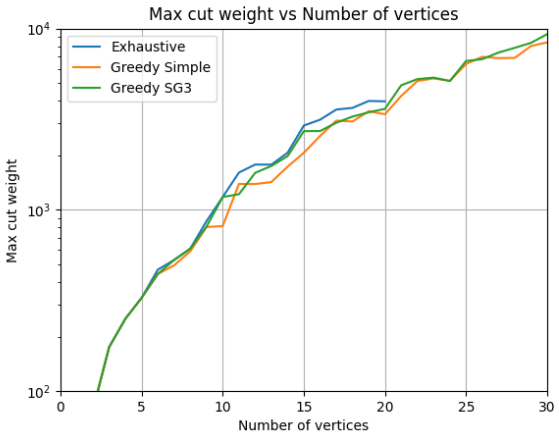

Fig 15 – Variação do Corte Máximo Ponderado para os 3 algoritmos com edge ratio = 0.75 e x=[0,30]

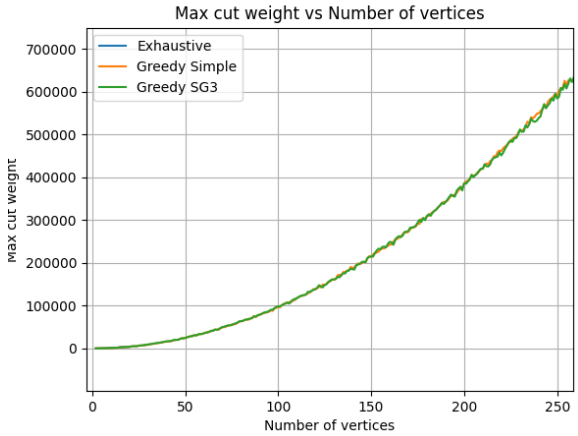

Fig 16 – Variação do Corte Máximo Ponderado para os 3 algoritmos com edge ratio = 0.75 e x=[0,260]

Comparando as figuras 15 e 16 podemos verificar que ambos os greedys têm aproximações bastante interessantes do valor do corte máximo ponderado, não diferindo muito eles próprios um do outro. Tendo isto em conta e o tempo de execução o algoritmo baseado na abordagem Sahni-Gonzalez 3 parece ser efetivamente a melhor opção, das estudadas para resolver um problema de otimização como este.

VII. Conclusão

Foram consideradas diferentes abordagens para tentar otimizar uma solução. Apesar de o algoritmo de pesquisa exaustiva ser muito dispendioso temporal e computacionalmente demonstrou ser o único capaz de produzir resultados fiáveis no que toca ao real valor do corte máximo ponderado. A falta de eficiência do algoritmo

deve-se em muito à sua complexidade $O(2^n)$ que é muito superior às dos algoritmos vorazes $O(n^3)$ e $O(n^2)$. Os algoritmos vorazes embora não garantam a melhor solução são capazes de produzir resultados muito satisfatórios e ótimos uma vez que normalmente retornam valores de corte relativamente perto do máximo e fazem-no de uma maneira bastante mais rápida e exigindo menos capacidade da máquina que os está a correr. Podemos tentar implementar extras aos algoritmos que, não aumentando significativamente a sua complexidade, possam ajudá-los a produzir resultados mais consistentemente melhores, à semelhança da tentativa de ordenar os vértices antes de os percorrer. No entanto é necessário avaliar bem os impactos dessas tentativas pois o esforço extra que estas exigem podem não produzir efeitos práticos suficientes para o justificar.

REFERÊNCIAS

[1] AA_2324_Trab_1.pdf

[2] On greedy construction heuristics for the MAX-CUT problem

[3] 21.Classical optimization: MaxCut problem

[4] Maximum Cut Problem, MAX-CUT

[5] Maximum Cut

[6] Corte Max 2011