Maximum Weighted Cut – Otimização por algoritmos randomizados

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 de pesquisa aleatória: o método de Monte Carlo e o método de escolher um nó aleatório e desse nó escolher o nó que se liga ao primeiro pela aresta mais pesada. Foi realizada uma análise formal e experimental de cada uma delas de forma a compará-las quanto aos parâmetros considerados e quanto aos resultados obtidos pelos algoritmos desenvolvidos no trabalho anterior**

***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 random research approaches: the Monte Carlo method and the method of choosing a random node and, from that node, selecting the node that is connected to the first one by the heaviest edge. A formal and experimental analysis of each approach was conducted to compare them in terms of the considered parameters and the results obtained by the algorithms developed in the previous work.**

Keywords – Maximum Weighted Cut, Monte Carlo, Randomized Algorithms, 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.

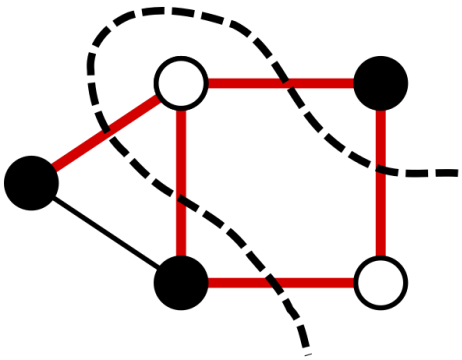

Fig. 1 – Exemplo visual da aplicação do Maximum Weighted Cut a um grafo

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.

Na continuação do trabalho anterior foram desenvolvidos 2 algoritmos de pesquisa aleatória a fim de comparar os seus resultados com os obtidos pelos algoritmos desenvolvidos anteriormente.

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 aleatória

Um algoritmo de pesquisa aleatória é um algoritmo e uma técnica de otimização não determinística que consiste em explorar o conjunto de soluções possíveis através da seleção aleatória de pontos deste e para posterior avaliação do resultado obtido. Esta técnica é utilizada quando a complexidade do problema e o tamanho do conjunto de soluções são muito elevados.

Para este problema em específico iremos selecionar aleatoriamente, com diferentes abordagens, os pares de conjuntos complementares S e T de forma a encontrar aquele que mais aproxima o valor do corte para o grafo G. Foi desenvolvido um código base que tem por objetivo limitar cada uma das abordagens definidas, quer quanto ao número de tentativas, quer quanto ao tempo de execução.

A função recebe o grafo a percorrer G, a percentagem do número máximo de iterações que pode ser percorrida, que em caso de omissão será 100%, o tempo máximo que se pretende que o algoritmo esteja em execução, por padrão será infinito, permitindo percorrer todas as iterações possíveis, e a função de geração do corte e dos conjuntos complementares. Após verificar se o grafo possui arestas que possam ser cortadas as variáveis necessárias são inicializadas, incluindo o número máximo de tentativas, que corresponde a $\frac{2^N}{2} \times \frac{\max\_tries}{100}$, sendo N o número de nós no grafo, 2^N corresponde ao número total de combinações que é possível fazer com esses N grafos, que é dividido por 2 uma vez que são sempre gerados 2 conjuntos complementares, ou seja, são testadas 2 combinações de cada vez, e $\frac{\max\_tries}{100}$ a percentagem do número combinações máximo de combinações a testar. Mesmo quando max_$tries$ seja 100 é muito improvável que todas as combinações sejam testadas uma vez que estas são geradas aleatoriamente e a probabilidade de em cada tentativa se gerar uma solução nova em todas as iterações diminui com o aumento do número de nós. No entanto isto não é um problema pois o objetivo dos algoritmos de pesquisa aleatória não é percorrer todas as possibilidades e sim apenas algumas. Também é inicializado um set que irá garantir que cada solução não é testada mais do que uma vez.

É gerado um corte aleatório que, como já referido, será gerado com uma componente de aleatoriedade e que irá diferir de abordagem para abordagem. Verifica-se se a solução produzida pelo algoritmo aleatório já foi testada, caso já o tenha sido é gerada uma solução nova, caso não tenha sido avalia-se se o corte produzido tem um corte maior que o anteriormente registado como maior, em caso afirmativo o novo corte substitui o antigo, e termina adicionando ambos os conjuntos originados pelo corte ao set de conjuntos testados. Este processo é repetido tantas vezes quantas aquelas que ficou definido aquando do cálculo do número máximo de soluções a testar, ou até que o limite de tempo de execução seja atingido, caso este seja passado ao programa.

```
def random_search(G, max_tries = 100, max_time = float("inf"), random_cut_func=None):

    if G.number_of_edges == 0:
        return 0, 0, 0
    max_tries = 2**G.number_of_nodes()//2*max_tries//100
    tested_sol = set()
    best_cut_weight = 0
    n_iter = 0
    start = time.perf_counter()

    while n_iter < max_tries and time.perf_counter() - start < max_time:
        n_iter += 1
        S1, S2 = random_cut_func(G)
        if S1 in tested_sol:
            continue
        cut_weight = nx.cut_size(G, S1, S2, weight='weight')
        if cut_weight > best_cut_weight:
            best_cut_weight = cut_weight
        tested_sol.add(S1)
        tested_sol.add(S2)

    return best_cut_weight, n_sol, n_oper
```

As abordagens que serão estudadas em seguida correspondem à função de geração aleatória do corte que é representada nesta função pela variável $random\_cut\_func$.

III. Método de Monte Carlo

Um método, aproximação ou algoritmo de Monte Carlo consiste em utilizar a aleatoriedade dos dados para gerar uma solução para problemas que à partida são problemas determinísticos. Para este caso específico a solução gerada consiste em 2 conjuntos complementares do grafo original G aos quais está associado um corte, cujo seu peso é aquilo que pretendemos maximizar.

A opção por este algoritmo deveu-se ao seu aprofundamento durante as aulas da unidade curricular e por ser um algoritmo relativamente simples de implementar e que consegue produzir resultados satisfatórios, mesmo que não sejam os melhores possíveis.

Para a otimização deste problema em específico por métodos de Monte Carlo cada nó do grafo G será atribuído a um dos conjuntos definidos pelo corte com base numa

função pseudoaleatória que gera um número pseudoaleatório entre 0 e 1, a decisão a qual dos conjuntos o nó é atribuído é dada pela condição de o número obtido ser maior ou menor que 0.5, assim como no lançamento de uma moeda perfeita.

```
def monte_carlo_random_cut(G):

    S1, S2 = set(), set()
    for i in G.nodes:
        if random.random() < 0.5:
            S1.add(i)
        else:
            S2.add(i)

    return tuple(S1), tuple(S2)
```

A. Análise Formal

Como é observável pelo código a complexidade do algoritmo de geração dos conjuntos do corte é de $O(n)$. Aliado à função que executa esta, onde temos um $while$ que pode a executar, a mesma executa um outro for de até $\frac{2^N}{2}$ iterações. Como já estudado no trabalho anterior a função $nx.cut\_size$ tem complexidade $O(n^2)$ para grafos não orientados, no entanto, para o pior caso, esta complexidade acaba por não ser a maior no algoritmo. No melhor dos casos a complexidade será sempre $O(n^2)$ uma vez que é a função de maior complexidade e que não apresenta resultados melhores que essa complexidade.

Sendo que por predefinição o algoritmo irá gerar o mesmo número de soluções que o de pesquisa exaustiva é de esperar um desempenho relativamente próximo deste se nenhum limitador for colocado.

B. Análise Experimental

Para verificar os resultados estipulados no processo de análise formal foram gerados exatamente os mesmos grafos com as mesmas condições que no trabalho anterior. Para além destes também foram providenciados, por parte do docente da unidade curricular, mais alguns grafos que também serão utilizados, e devidamente referidos, para avaliar o desempenho das soluções desenvolvidas.

Para a formulação deste relatório foram tidas em conta quatro métricas para avaliação: o tempo de execução, o número de configurações/soluções testadas, o número efetivo de iterações percorridas e o número de operações básicas que foram definidas como sendo a determinação de um novo corte máximo.

Inicialmente foi feita uma comparação básica entre os resultados de pesquisa exaustiva e os resultados deste algoritmo de Monte Carlo para verificar a capacidade de este gerar soluções para pequenos grafos.

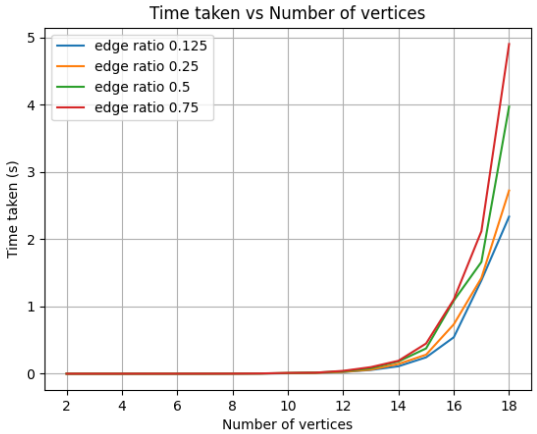

Figura 2 – Método de Monte Carlo: Tempo de execução por grafo

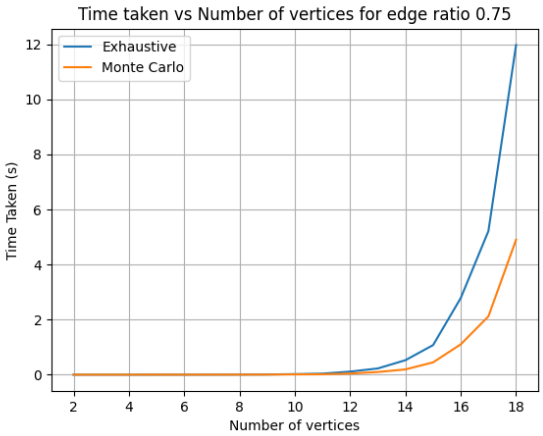

Figura 3 - Tempo de execução por grafo para 0.75 de densidade para Pesquisa Exaustiva e Método de Monte Carlo

Em termos de valores do peso do corte máximo podemos verificar que o Método de Monte Carlo conseguiu igualar o algoritmo de pesquisa exaustiva em 85% dos grafos com vértices a variar entre os 2 e 18 por grafo e pelo gráfico da figura 3 podemos observar que a sua execução demora menos de metade do tempo que o outro algoritmo em comparação.

Apesar de demorar menos tempo que o algoritmo de pesquisa exaustiva, 5 segundos já é muito tempo para arriscar percorrer grafos maiores. Para limitar o número de execuções os resultados do algoritmo foram estudados de maneira a verificar qual a percentagem do número máximo de iterações que este precisou de percorrer para chegar ao maior valor de corte.

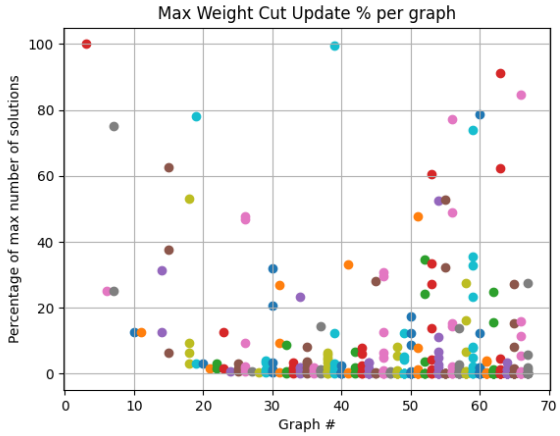

Figura 4 – Percentagem do número máximo de soluções em que é detetado um novo máximo do peso de corte Monte Carlo

Pelo mapa da figura 4 podemos perceber que a maioria dos grafos não atualiza o valor de corte máximo depois dos 19% e os grafos que o fazem fazem-no relativamente poucas vezes, indicando que o último valor de corte registado já é bastante elevado. Assim, podemos limitar o número máximo de soluções testadas a 19% do número máximo, reduzindo o tempo de pesquisa para cerca de $\frac{1}{5}$ daquele que seria expectável. No entanto como estamos perante um crescimento exponencial do tempo de pesquisa rapidamente os valores escalarão novamente e por isso o valor será definido em 10%.

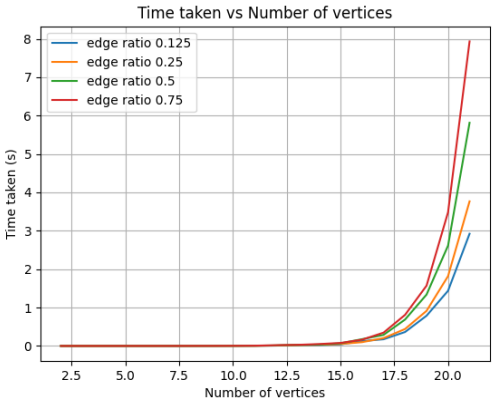

Figura 5 – Método de Monte Carlo: Tempo de execução por grafo para 10% do número máximo de soluções

Como esperado o tempo de execução diminuiu bastante, mas devido ao crescimento exponencial o tempo de execução volta a escalar bastante para os grafos a seguir, sendo que para um grafo de 21 vértices e uma percentagem de arestas de 75% o algoritmo demora 8 segundos a percorrer 10% das soluções possíveis. A única maneira de contrariar este crescimento é colocando uma limitação ao tempo de execução, que iremos definir como 2 segundos de tempo máximo de execução. Esta opção irá encontrar uma solução mais rapidamente para grafos maiores, mas com menor probabilidade de ser a solução ideal. A fim de realizar o estudo da capacidade do algoritmo de Monte Carlo de gerar soluções ótimas, impondo-lhe uma limitação temporal, os resultados obtidos serão comparados com o algoritmo de pesquisa gulosa Sahni-Gonzalez 3, explorado no primeiro trabalho.

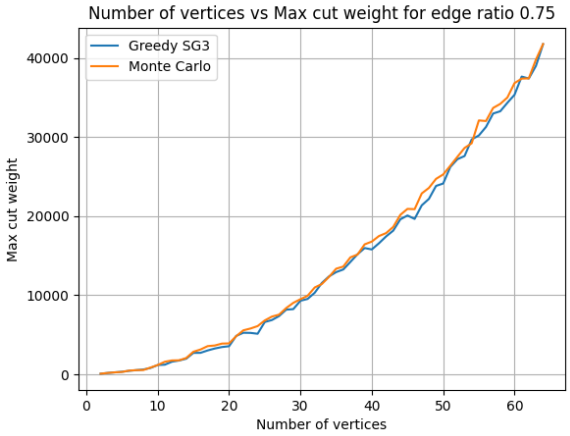

Figura 6 – Variação do Corte Máximo Ponderado para os algoritmos Greedy SG3 e Monte Carlo para grafos com edge ratio = 0.75

Como é possível observar pelo gráfico da figura 6, mesmo com o tempo de execução limitado a 2 segundos, o método de Monte Carlo consegue atingir a maioria das vezes valores de corte máximo ponderado superiores ao algoritmo de pesquisa gulosa, para grafos com até 64 nós, garantindo assim utilidade relativamente ao algoritmo derrotado nesta comparação.

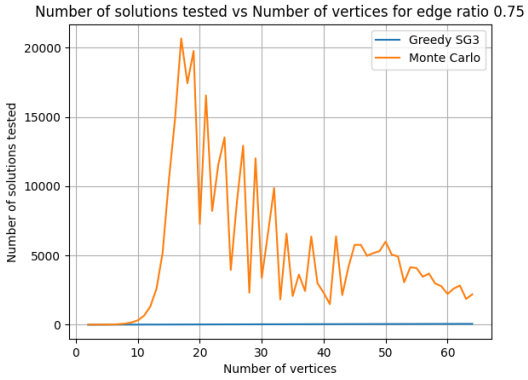

Figura 7 – Soluções testadas para os algoritmos Greedy SG3 e Monte Carlo para grafos com edge ratio = 0.75

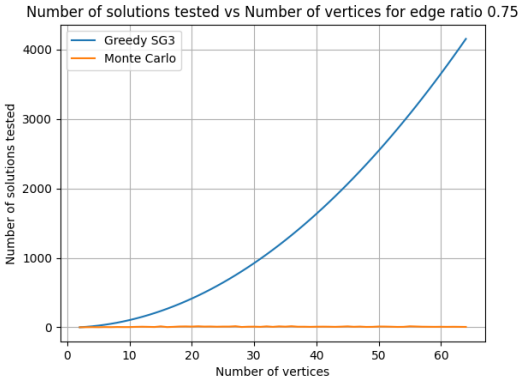

Figura 8 – Operações básicas contabilizadas para os algoritmos Greedy SG3 e Monte Carlo para grafos com edge ratio = 0.75

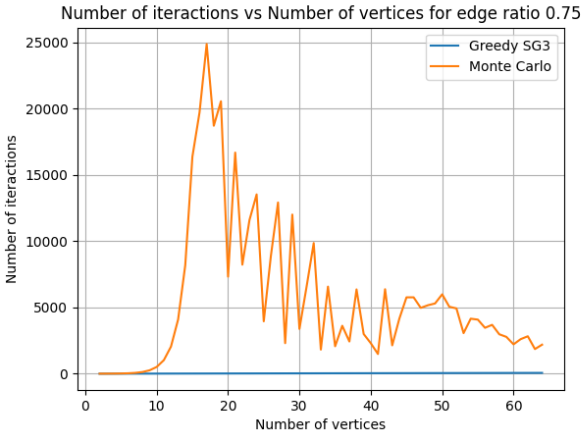

Figura 9 – Número de iterações percorridas para os algoritmos Greedy SG3 e Monte Carlo para grafos com edge ratio = 0.75

Em termos de métricas podemos verificar que o algoritmo greedy fez muito menos iterações que o algoritmo de Monte Carlo, assim como testou todas elas, fez igualmente muito menos testes a soluções que o algoritmo de Monte Carlo, no entanto encontrou muito mais vezes um valor de corte máximo ponderado novo.

IV. Método Nó Aleatório Melhor Aresta

Com o objetivo de maximizar mais rapidamente o valor de corte máximo foi desenvolvido um algoritmo de pesquisa aleatória que, depois de escolher aleatoriamente um nó, adiciona-o a um dos conjuntos e adiciona ao outro conjunto o nó ligado pela aresta de maior peso ao escolhido. A escolha aleatória é implementada através do embaralhamento do conjunto inicial de nós, após isto é retirado sempre o primeiro da lista de vértices. Para evitar duplicações e conflitos cada um dos nós é removido da cópia do grafo inicial e é sempre verificado se o nó da iteração não foi já colocado no segundo conjunto. Caso o nó ainda não esteja em nenhum dos subconjuntos e este ainda possua vizinhos então o vizinho ligado pela aresta de maior peso é adicionado ao conjunto complementar.

```
def random_node_choose_best_edge_cut(G: nx.Graph):

    G = G.copy()
    S1, S2 = set(), set()
    nodes = list(G.nodes)
    random.shuffle(nodes)

    for i in nodes:
        if i in S2:
            continue
        S1.add(i)
        neighbors = list(G.neighbors(i))
        if len(neighbors) > 0:
            max_neighbor = max(neighbors, key=lambda x: G.get_edge_data(i, x).get('weight', 0))
            S2.add(max_neighbor)
            G.remove_node(max_neighbor)
        G.remove_node(i)

    return tuple(S1), tuple(S2)
```

A. Análise Formal

Em termos de complexidade este algoritmo de geração de conjuntos é igual ao de Monte Carlo, $O(n)$, já que apenas temos um for que percorre todos os nós. Como a função que percorre as soluções geradas foi definida como sendo a mesma para os algoritmos randomizados então as complexidades envolvidas na solução, bem como o número de soluções geradas, serão os mesmos do método de Monte Carlo.

Como já referido, o desenvolvimento deste algoritmo prendeu-se com a tentativa de adicionar algum critério às decisões aleatórias que continuam a ser tomadas, tendo em vista alcançar valores de corte mais altos e com um número menor de soluções testadas para o fazer.

B. Análise Experimental

Os grafos utilizados para testar esta abordagem foram exatamente os mesmos utilizados para testar a abordagem de Monte Carlo assim como foram mantidas as métricas utilizadas nesse estudo. A fim de perceber novamente a capacidade de alcançar um bom valor de corte máximo para grafos menores foram comparados os resultados entre o método de Nó Aleatório Melhor Aresta (NAMA) e os resultados do algoritmo de pesquisa exaustiva.

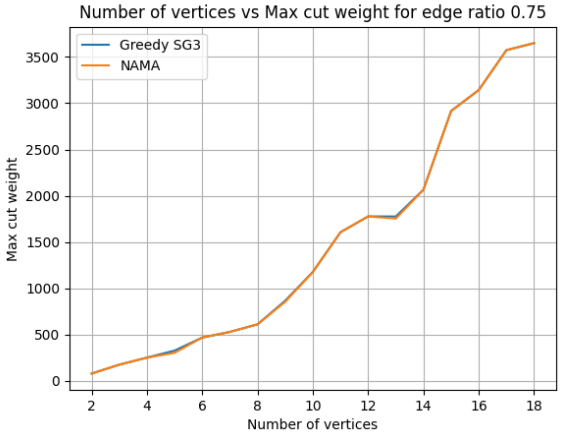

Figura 10 – Variação do Corte Máximo Ponderado para os algoritmos Pesquisa Exaustiva e NAMA para grafos com edge ratio = 0.75

Os dados da figura 9 permitem-nos ter confiança nos valores produzidos pelo NAMA uma vez que alcança valores iguais ou muito próximos do algoritmo de pesquisa exaustiva. No entanto o NAMA demorou, em todos os grafos, cerca de 4 a 5 vezes mais tempo a percorrer todas as soluções, o que nos leva novamente à necessidade de cortar uma percentagem das soluções testadas.

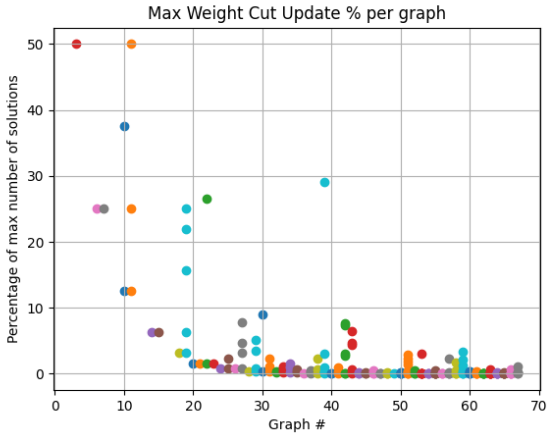

Figura 11 - Percentagem do número máximo de soluções em que é detetado um novo máximo do peso de corte NAMA

Pelo mapa da figura 10 é possível verificar que o objetivo do algoritmo em alcançar mais rapidamente uma solução ótima é atingido, já que são muito poucos os grafos que precisam de mais de 3% das soluções testadas para atingir o valor ótimo do corte máximo ponderado.

Novamente, estamos a tratar de um problema de crescimento exponencial, o que irá resultar num crescimento de tempo idêntico ao registado no estudo do algoritmo de Monte Carlo. Por isso iremos novamente definir como tempo máximo de execução 2 segundos e comparar os resultados obtidos com o algoritmo de pesquisa gulosa.

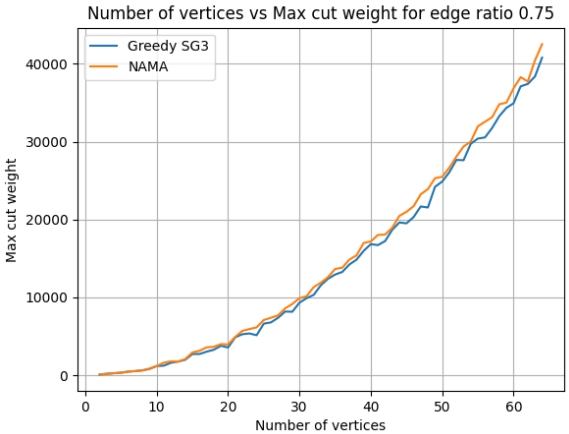

Figura 12 - Variação do Corte Máximo Ponderado para os algoritmos Greedy SG3 e NAMA para grafos com edge ratio = 0.75

Pela figura 11 é possível verificar que o NAMA alcança sempre resultados de corte máximo ponderado superiores aos do SG3, mesmo limitado a 2 segundos de execução. A explicação está precisamente no facto de o greedy testar muito menos soluções já que vai eliminando candidatos à medida que testa novas soluções, mesmo que a solução ainda não tenha sido testada, enquanto o NAMA nunca descarta soluções que não tenham sido já testadas e prioriza propositadamente o corte das arestas mais "pesadas".

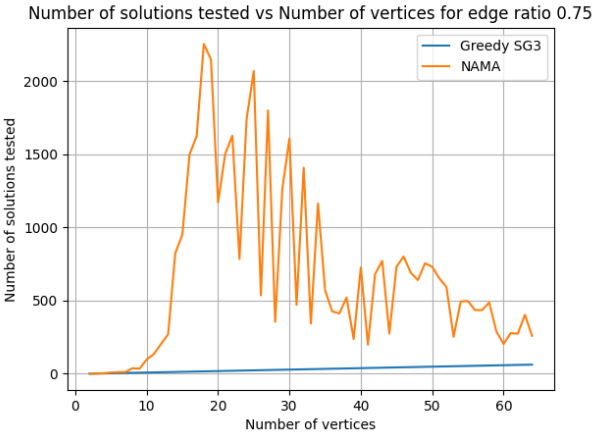

Figura 13 – Soluções testadas para os algoritmos Greedy SG3 e NAMA para grafos com edge ratio = 0.75

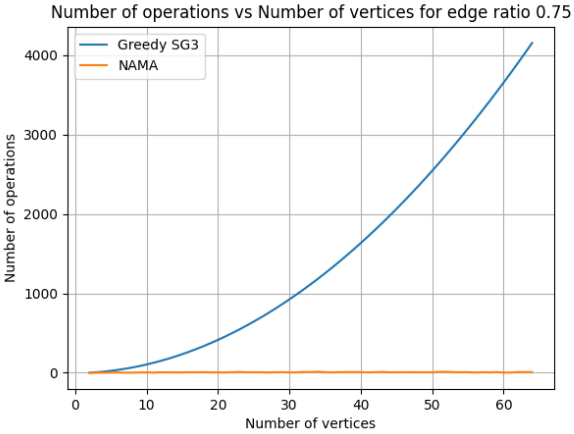

Figura 14 – Operações básicas contabilizadas para os algoritmos Greedy SG3 e NAMA para grafos com edge ratio = 0.75

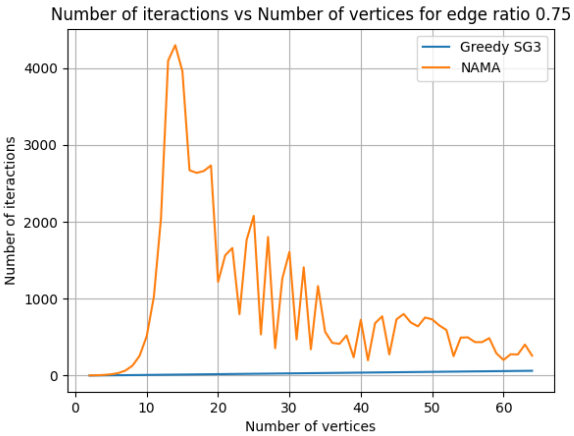

Figura 15 – Número de iterações percorridas para os algoritmos Greedy SG3 e NAMA para grafos com edge ratio = 0.75

Em termos de métricas podemos verificar que são bastante semelhantes às verificadas no estudo do algoritmo de Monte Carlo, não fossem ambos algoritmos de pesquisa aleatória. Também pelas métricas é possível verificar o que foi enunciado de que o SG3 acaba por avaliar muito poucas soluções enquanto o NAMA avalia todas as que pode em 2 segundos.

Numa tentativa de perceber se quando a métrica das soluções testadas pelo greedy é superior à do NAMA os resultados do peso do corte máximo ponderado continuam ou não a ser favoráveis ao algoritmo randomizado foi procurado o ponto em que a inversão acontecia e testado a partir dele.

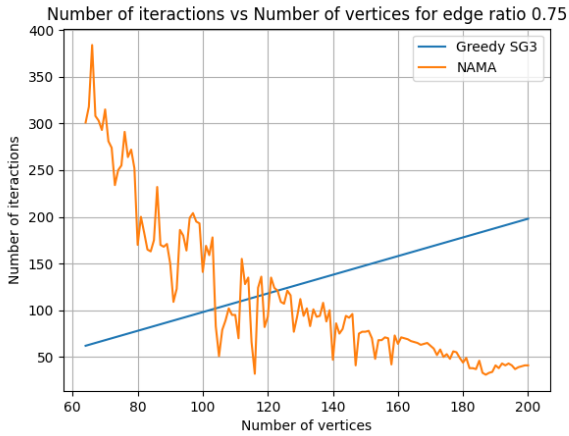

Figura 16 – Número de iterações percorridas para os algoritmos Greedy SG3 e NAMA para grafos com edge ratio = 0.75 e x=[64,200]

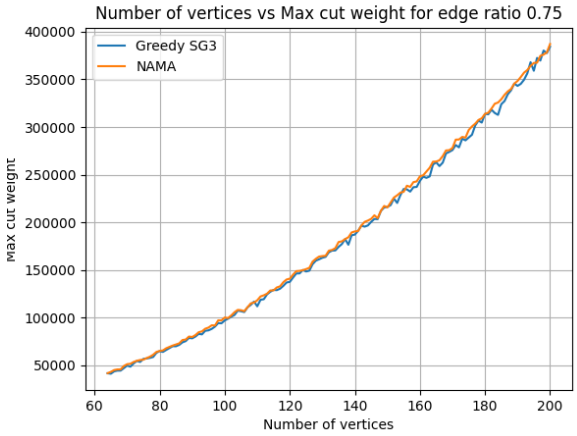

Figura 17 – Variação do Corte Máximo Ponderado para os algoritmos Greedy SG3 e NAMA para grafos com edge ratio = 0.75 e x=[64,200]

Pela figura 16 é possível observar que o número de iterações do SG3 supera o do NAMA pouco depois dos 120 vértices. No entanto pela figura 17 vemos que o NAMA continua quase sempre com resultados de corte máximo ponderado superiores aos do SG3, o que permite concluir que o método utilizado para acelerar o processo de descobrir o valor ótimo efetivamente funciona.

V. Exploração de resultados

A fim de testar as abordagens para grafos mais dispendiosos computacionalmente foi percorrido o grafo SW10000EWD, com 10 mil vértices, para os algoritmos de Monte Carlo e NAMA com um limite de 20 segundos e para o Greedy SG3 sem limite.

	Peso do corte	Soluções testadas	Operações básicas	Iterações
Greedy SG3	499.43429	9998	100009995	9998
Monte Carlo	415.35449	447	3	447
NAMA	460.76882	61	7	61

Tabela 1 – Resultados da pesquisa no grafo SW10000EWD

Tendo por base a Tabela 1, como já era expectável o NAMA foi capaz de testar muito menos soluções do que o Monte Carlo, no entanto, e como já tinha demonstrado na análise anterior, foi capaz de alcançar um valor de corte máximo ponderado superior e atualizou o valor do corte mais do sobro das vezes que o Monte Carlo para apenas 13% das iterações, que acabaram também por ser todas testadas. Já o Greedy SG3 alcançou estes resultados com 1 minuto de execução.

VI. Conclusão

Foram desenvolvidos 2 algoritmos diferentes para tentar resolver o problema que o trabalho nos propunha, o método de Monte Carlo, que serviu inicialmente como uma base de trabalho, e o método de escolher um nó aleatório e dele selecionar a aresta com mais peso, para adicionar algum critério ao método de Monte Carlo e assim chegar mais rapidamente a uma solução. A realidade é que estes algoritmos se demonstraram bastante bons, mesmo quando limitados no tempo de execução ou no número de iterações, alcançando valores de corte máximo ponderado superiores ao próprio Greedy SG3, principalmente o NAMA.

Mais uma vez, o único algoritmo capaz de garantir o valor correto é o de pesquisa exaustiva, mas os algoritmos de pesquisa aleatória mostraram-se bastante capazes de gerar soluções igualmente boas e bastante rápido. Algoritmos puramente aleatórios correm sempre o risco de não encontrar logo a melhor solução que são capazes, como o caso de Monte Carlo, mas quando adicionamos uma "heurística" podemos verificar que o processo de obter o melhor peso do corte máximo ponderado fica bastante otimizado, como vemos no NAMA.

Referências

[1] AA_2324_Trab_2.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
[7] Monte_Carlo_method
[8] Randomized Algorithms