Algoritmos Avançados – 2º Projeto
Algoritmo Aleatório para o problema Maximum Weight Cut

Mariana Rosa

Resumo **- Este artigo tem como propósito demonstrar o trabalho desenvolvido para o 2º projeto da disciplina Algoritmos Avançados, que consistiu em desenhar e testar um algoritmo de pesquisa aleatório para resolver um problema de grafos: o Maximium Weight Cut.**

I. INTRODUÇÃO

Este trabalho tem como continuação explorar o problema de grafos Maximum Weight Cut, em português, Peso Máximo de Corte [1]. Consiste em, dado um grafo não-direcionado $G=(V, E)$ com o conjunto de vértices $V = [1,2, \dots n]$ e um conjunto de arestas E com um peso $w_{i,j} \geq 0$ para cada $(i,j) \in E$, encontrar o peso máximo dado por um corte em arestas. Formam-se dois conjuntos de vértices, uma bipartição do conjunto V em S e T, tal que $S \cap T = 0$ e $S \cup T = V$. As arestas com os vértices em S, são cortadas e a soma do peso das suas arestas é o resultado do corte. Os vértices do conjunto S é a soma das arestas formados pelos vértices do conjunto S em T: $corte(S, T) = \sum_{i \in S, j \in T} w_{ij}$. O objetivo é encontrar o melhor conjunto S e T para se obter o maior peso possível para o corte. Este problema na teoria não é difícil de explicar, contudo quando vamos tentar resolver já não é bem assim, sendo classificado como um dos NP-hard problem. Matematicamente falando, este problema escreve-se com a seguinte fórmula[2]: $pesoMáximo = max \sum_{i,j=1, i<j}^{n} w_{ij} * y_{ij}$. Onde:

- $y\,ij - x\,i - x\,j \leq 0,\ i, j = 1, 2, \dots, n,\ i < j;$
- $y\,ij + x\,i + x\,j \leq 2,\ i, j = 1, 2, \dots, n,\ i < j;$
- $x\,i \in \{0, 1\},\ i = 1, 2, \dots, n.$

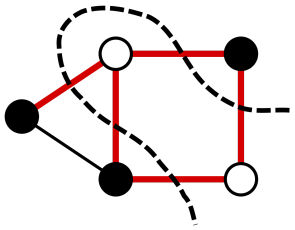

Fig. 1 - [3] Exemplo de demonstração do corte das arestas. Os vértices que estão a preto simbolizam o conjunto S, ou seja, todas as arestas que a estes se ligam, serão cortadas. Sendo o peso do corte igual ao peso (ou custo) de todas as arestas pelo qual o corte passa.

No primeiro trabalho desta disciplina foram desenhados dois algoritmos diferentes para resolver o problema: algoritmo de pesquisa exaustiva e uma pesquisa gulosa. Neste segundo trabalho a abordagem será diferente: explorar e desenhar um algoritmo de pesquisa aleatório. De notar que o grafo a analisar não poderá ter vértices isolados, nem ter menos de 2 vértices ou 3 arestas, se não nenhum dos algoritmos irá funcionar.

II. ESTRUTURA DO CÓDIGO DESENVOLVIDO

Muito do código para a realização deste projeto é reaproveitado do 1º trabalho, tal como o código de fazer *plot* aos resultados, os resultados do 1ª trabalho também vão ser utilizados para compararmos os algoritmos.
O problema está dividido em dois ficheiros escritos em *Python* e podem ser corridos da seguinte maneira:

```
· $ python3 main_randomized.py
· $ python3 plot_results_randomized.py
· $ python3 graph_generator_randomize.py
```

No ficheiro *main_randomized.py* é onde está toda a elaboração do algoritmo de pesquisa aleatório, encontra-se também as funções já explicadas anteriormente para calcular o resultado através da força bruta e da pesquisa gulosa, mas com um upgrade, onde o peso das arestas é calculado previamente.
Os ficheiros *plot_results_randomized* e *graph_generator_randomize* são semelhantes ao do 1º projeto, apenas foram adaptados para este trabalho para analisar resultados e gerar grafos, respetivamente.
No primeiro relatório encontra-se a explicação de como estes algoritmos funcionam, neste relatório irá só estar explicado as novas funções criadas para o algoritmo de pesquisa aleatória, que foram:

- *find_cut_randomized_simples(vertex,edges)*: primeira tentativa de um algoritmo aleatório muito simples;

- *find_cut_randomized_algorithm(vertex,edges,weight)* : retorna uma solução para um grafo (o subset S, T e o valor do máximo corte). Uma grande mudança no código deste trabalho, é que o peso das arestas agora é calculado previamente

e passado como argumento da função, fazendo com que o código seja muito mais rápido.

perform_randomized_algorithm(vertices, edges,weights) : retorna a melhor solução depois de iterar o algoritmo um determinado número de vezes, para ver qual a melhor solução aleatória.

· *save_solution(vertices,edges,filepath):* função que permite resolver um grafo através dos 3 algoritmos desenhados até agora. Todos os resultados encontram-se na pasta *solutions*. Recolhe dados sobre o número de vértices, arestas, o número do maior peso de corte, que vértices estão presentes no conjunto *S*, no conjunto *T*, o número de iterações realizadas e o tempo de execução de cada função em cada algoritmo. Foram aproveitadas as soluções do 1º trabalho, sendo apenas recolhidas as informações relativas aos algoritmos novos.

· *analyse_file(filename):* função para gerar pesos aleatórios para os grafos fornecidos pelo docente.

Na pasta *solutions* encontram-se os resultados para estes novos algoritmos e na pasta *solutions_1stproject* os resultados do 1º projeto. Na pasta *graficos* encontram-se os gráficos gerados para a análise do algoritmo.

II. ALGORITMO DE PESQUISA ALEATÓRIA

Um algoritmo de pesquisa aleatório[4], é um algoritmo não-determinístico, uma vez que faz decisões aleatórias para encontrar uma solução, com o objetivo de diminuir a complexidade computacional para tentar chegar rapidamente a uma boa solução. Para o algoritmo desenhado, foi usada a *seed* 98390. Numa primeira abordagem, foi criado um algoritmo simples, baseado num trabalho feito por Alexander Dimitrakakis[5], onde este criou um algoritmo aleatório para o Max Cut problem, ao apenas "atirar uma moeda ao ar" para decidir se os vértices iriam para o conjunto S ou T.

Algoritmo 1: Algoritmo de Pesquisa Aleatória Simples (versão 1)

Input: *(V, E),* sendo que *V* e *E* formam um Grafo *G,* com pesos (custo) w_{ij}, $\forall i, j \in V, i \neq j$
Output: Uma lista formada pelo *custo* do corte *(S,T),* os conjuntos *S* e *T*. De notar que, $S \cap T = 0$ e $S \cup T = V$
weights ←Lista do custo (w_{ij}) de todas as arestas, sendo, w_{ij}, $\forall i, j \in V, i \neq j$
cut_weight ←0
dic_iterations_max_cut ←{}, dicionário onde as chaves são as iterações e o valor o maior corte possível no grafo

for *i=1 ... range(V)* **do**
    Flip a coin *ri* ∈ {"heads", "tails"} :
    if *ri* = "heads"**do**
        adicionar i a S
    else do
        adicionar i a T
cut_weight ←calcular peso do corte com S e T
return cut_weight, S, T

Contudo, com este algoritmo, obtia-se resultados baixos de peso máximo quando comparados com a solução ideal (resultado fornecido pelo algoritmo de força bruta). Pelo facto do algoritmo ser totalmente aleatório, sem nenhum critério de seleção para alcançar resultados que realmente pudessem ser valores aceitáveis para o corte máximo, foi criado um outro algoritmo mais elaborado.

Este consiste em escolher aleatóriamente, *i* vezes, onde *i* é igual ao número de metade de arestas existentes no grafo, duas arestas, *e1* e *e2,* cada uma composta por vértices *v1* e *v2*. Depois, comparamos o peso entre as mesmas de cada vez. Se forem diferentes uma da outra, então, vamos ver o peso de cada uma e consoante a que for mais pesada (a que tiver um maior peso), vamos atribuir ao conjunto S *v1* e a T *v2*. Se estas tiverem o mesmo peso, então aplica-se a lógica do algoritmo anterior de "atirar uma moeda" para decidir qual adicionar a cada conjunto. Numa explicação sucinta de código seria o seguinte:

Algoritmo 2: Algoritmo de Pesquisa Aleatória versão 2 (mais elaborado)

Input: *(V, E,weights),* sendo que *V* e *E* formam um Grafo *G,* com pesos (custo) num dicionário w_{ij}, $\forall i, j \in V, i \neq j$
Output: Uma lista formada pelo *custo* do corte *(S,T),* juntamente com os conjuntos *S* e *T*. De notar que, $S \cap T = 0$ e $S \cup T = V$
subset_s ←Lista de vértices v em que v∈ *V*
subset_t ←Lista de vértices v em que v∈ *V*
weights ←Lista do custo (w_{ij}) de todas as arestas, sendo, w_{ij}, $\forall i, j \in V, i \neq j$
for *i=1 ... range(len(E)/2)* **do**
    e1 = escolher aleatóriamente uma aresta de E
    e2 = escolher aleatóriamente uma aresta de E
    if e1 ≠ e2 **do**
        if $\textbf{weight}_{e1} > \textbf{weight}_{e2}$ **do**
            if v1 e v2, vértices que compõe *e1*, verificar se v1 ou v2 não estão no *subset_s* nem v2 no *subset_t* **do**
                subset_s ← *subset_s* + $\{v1_{e1}\}$
                subset_t ← *subset_t* + $\{v2_{e1}\}$
        if $\textbf{weight}_{e1} < \textbf{weight}_{e2}$ **do**
            if v1 e v2, vértices que compõe *e2*, verificar se v1 ou v2 não estão no *subset_s* nem v2 no *subset_t* **do**
                subset_s ← *subset_s* + $\{v1_{e2}\}$
                subset_t ← *subset_t* + $\{v2_{e2}\}$

if $weight_{e1} = weight_{e2}$ **do**

Flip a coin *ri* ∈ {"heads", "tails"} :

if *ri* = "heads"**do**

subset_s ← *subset_s* + {$v1_{e1}$}

subset_t ← *subset_t* + {$v2_{e1}$}

else do

subset_s ← *subset_s* + {$v1_{e2}$}

subset_t ← *subset_t* + {$v2_{e2}$}

for *v* em V **do**

if v ∉ *subset_s* **and** v ∉ *subset_t* **do**

subset_t ← *subset_t* + {v}

cut_weight ←calcular peso do corte com *subset_s* e *subset_t*

return cut_weight, *subset_s*, *subset_t*

Para ter a certeza que encontramos a melhor solução possível vamos efetuar o Algoritmo 2 *n* vezes na função *perform_randomized_algorithm*. Sendo *n* decidido consoante o número de arestas existentes no grafo. Isto pois, um maior número de iterações implica maior gasto computacional. Por exemplo, para chegar a uma solução do grafo fornecido pelo docente *SWMediumG* demorou mais de 6 horas, algo completamente inviável. Após algum raciocínio, chegou-se a uma conclusão do que poderia ser um número viável para n:

- Se $|E| \leq 15$, $n = 20\ \% * 2^{Nº\ total\ de\ arestas}$
- Se $|E| \geq 16$ e $|E| < 25$, $n = 10\ \% * 2^{Nº\ total\ de\ arestas}$
- Se $|E| \geq 25$, $n = 1000$

Contudo, não foi possível testar estes valores com os restantes grafos maiores fornecidos pelo docente, aparecendo no terminal "*killed"*, mesmo sendo *n* reduzido a 1000 ou 10.000 para números mais elevados.

Estes valores foram escolhidos, uma vez que 2^N é o número possível de soluções deste algoritmo, e como não queremos testar para todas as possibilidades por questões de complexidade computacional, vamos apenas fazer para uma percentagem deles. O valor exato foi escolhido tendo em conta que seria um número fixo, uma vez que vamos lidar com grafos muito maiores, logo muito mais informação.

IV. RESULTADOS

Para fazer uma análise de todos os resultados, foram recolhidos dados de 56 grafos, com vértices de *n*=2 a *n*=24, com quantidade de arestas diferentes (12.5%, 25%, 50% ou 75%) . Contudo, como dito anteriormente, os grafos só conseguiam correr o algoritmo se tivessem mais de dois vértices e três arestas e não conterem vértices isolados. Restando apenas 42.

Para avaliar a *performance* deste algoritmo, primeiramente a análise será feita ao comparar um grafo com os três algoritmos. Este grafo tem cerca de 8 vértices e 17 arestas. À primeira vista, o algoritmo de pesquisa aleatória 1 é descartado por ser a pior solução, analisaremos a fundo a versão 2 pois este de facto encontrou a solução ótima, mas para este exemplo o seu tempo de execução é bem superior ao algoritmo de força bruta por fazer mais de 100.000 iterações.

Algoritmo	Nº de iterações	Peso máximo do corte	Tempo de execução
Força Bruta	162	111 (peso ideal)	0.002
Pesquisa gulosa	15	73	5.102e-05s
Pesquisa aleatória versão 1	8	59	5.126e-05s
Pesquisa aleatória versão 2	13123	111	0.047

Tabela 1 - Resolução do grafo com 8 vértices e 17 arestas com os 4 algoritmos desenhados até agora.

Iremos dividir a análise por partes:

· Análise do grafo **SWtinyG** fornecido pelo docente resolvida por pesquisa aleatória versão 1 VS Grafo resolvida por pesquisa aleatória versão 2 VS Brute Force
· Avaliar o número de iterações;
· Avaliar o tempo de execução;
· Avaliar os resultados obtidos do corte máximo;
· Características do algoritmo de pesquisa aleatório desenvolvido
· Avaliar a aplicação dos algoritmos para problemas muitos maiores.

*A. Análise Grafo **SWtinyG** resolvida por pesquisa aleatória versão 1 VS Grafo resolvida por pesquisa aleatória versão 2 VS Brute Force*

Vamos comparar o grafo SWtinyG fornecido pelo docente que tem 13 vértices e 13 arestas e obter soluções dos 3 algoritmos desenhados. Sendo o Brute Force o que retorna a melhor solução, vamos ver em que medida os algoritmos elaborados para este projeto retornam uma boa solução ou não.

Algoritmo	Nº de iterações	Tempo de execução	Peso do corte
Versão 1	13	7.44e-05s	94
Versão 2	1657	0.049	133
Brute Force	4095	0.09s	149

Tabela 2 - Comparação entre os algoritmos de pesquisa aleatória e o de força bruta.

Como podemos observar pela tabela, o algoritmo de força bruta idealiza o corte máximo como 129, apesar da 1ª versão ter encontrado uma solução não muito longe da solução ideal rapidamente, a versão 2 é sem dúvida muito melhor. Apesar de ter um tempo de execução muito superior ao da 1ª versão, alcançou um resultado similar ao do algoritmo da força bruta em exatamente metade do tempo e com muito menos iterações. Assim, a partir de agora iremos descartar a 1ª versão do algoritmo de pesquisa aleatória e focar em analisar o 2º

B. Análise do número de iterações (número de operações básicas)

Observando a tabela 1, podemos ver claramente que naquele exemplo o número de iterações é muito superior ao da força bruta, uma vez que definimos as regras em cima.
Mas e se formos comparar apenas com o algoritmo de pesquisa aleatória? Na tabela 3 observamos que o número de iterações para este algoritmo não é exponencial na sua forma geral, uma vez que quando existem mais de 25 arestas o número de iterações na função *perform_randomized_algorithm* reduz para o 1000.

Nº de vértices	Nº de arestas	Nº de iterações
9	8	64
11	12	836
12	15	6572
14	19	52451
15	23	838886
16	27	1029
17	29	1031
18	35	1035
19	20	104886

Tabela 3 - Alguns dos resultados obtidos ao percorrer o algoritmo de pesquisa aleatório

O gráfico resultante (Fig. 2) não diz resultados concretos, contudo quando vamos comparar com os outros dois algoritmos (Fig. 5) do trabalho passado conseguimos ver que tem muito muito menos que o algoritmo de força bruta (Fig. 3). Porém, nem conseguimos observar os resultados da pesquisa gulosa, isto pois o como podemos observar na Fig. 5 estes valores são bastante reduzidos, levando a que este seja o vencedor na complexidade computacional, muito mais simples e menos exigente.

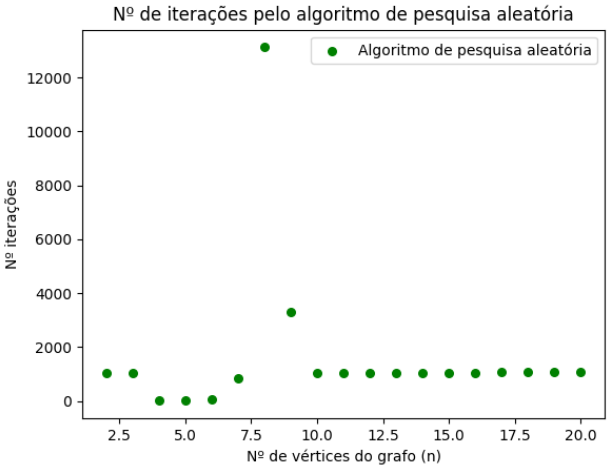

Fig. 2 -Gráfico de análise do número de iterações por força bruta

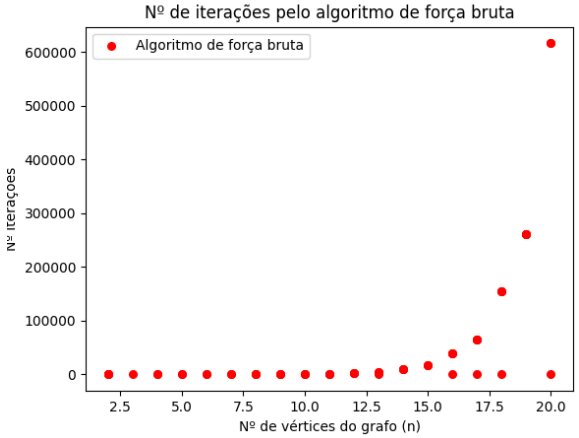

Fig. 3 -Gráfico de análise do número de iterações por força bruta

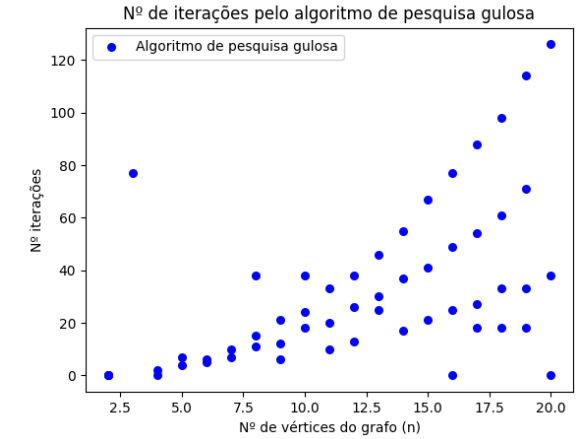

Fig. 4 -Gráfico de análise do número de iterações por pesquisa gulosa

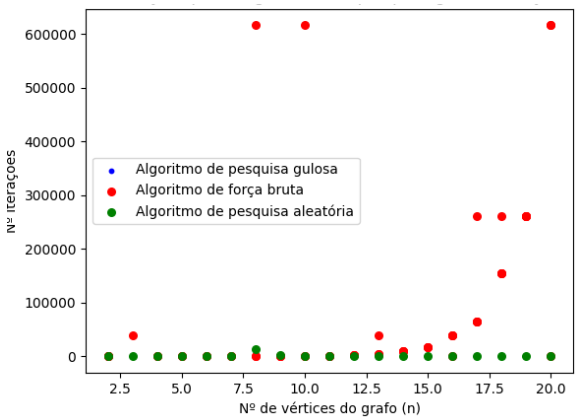

Fig. 5 -Gráfico de análise do número de iterações dos três algoritmos

C. Análise dos tempos de execução

Ao comparar os três algoritmos (Fig. 6), não há dúvida que o algoritmo de pesquisa gulosa continua a ser o mais rápido e o algoritmo de pesquisa aleatória (Fig. 7) consegue ser mais rápido que o *brute force* em alguns casos, não em todos, por causa das condições do número de iterações.

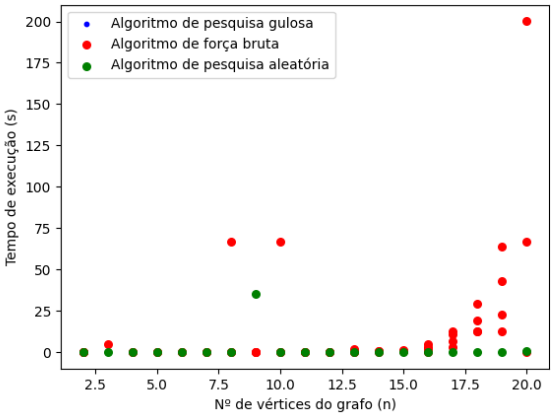

Fig. 6 -Gráfico de análise do tempo de execução por força bruta, pesquisa gulosa e pesquisa aleatória

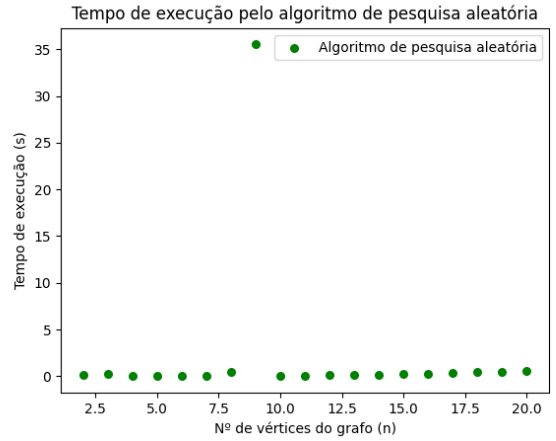

Fig. 7 - Gráfico a avaliar o tempo de execução do algoritmo de pesquisa aleatória

D. Análise dos resultados obtidos do corte máximo

Na tabela 1 conseguimos comprovar que o algoritmo de pesquisa aleatória conseguiu chegar à solução ideal, contudo demorou mais tempo. No que toca a chegar a resultados do peso máximo, este algoritmo é sem dúvida eficiente, conseguindo resultados muito semelhantes ou até mesmo iguais à solução ideal.

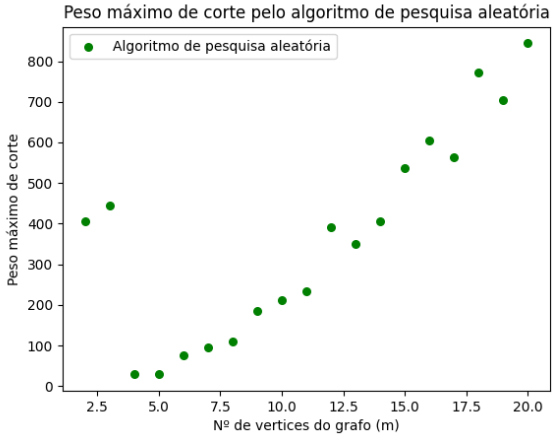

Fig. 8- Gráfico dos resultados do peso máximo obtido pelo algoritmo de pesquisa aleatória

Podemos ver ao sobrepor os algoritmos que consegue chegar a soluções bastante elevadas, ao mesmo nível que o algoritmo de força bruta. Os resultados de pesquisa gulosa novamente não são visíveis no gráfico pois não tem soluções muito boas (Fig. 9 .

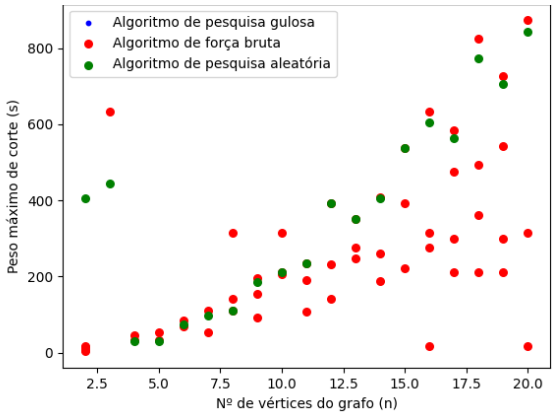

Fig. 9- Gráfico a sobrepor os resultados do peso máximo obtido pelos três algoritmos

E. Características do algoritmo de pesquisa aleatório desenvolvido

Este algoritmo em termos de complexidade computacional é cerca de $O(n^3)$. Sendo mais pesado em relação aos outros algoritmos desenvolvidos, contudo, dependendo do número de arestas presentes no grafo, pode ser melhor para grafos com maior número de arestas. Pois a partir das 25 arestas o número de iterações passa para pelo menos 1000, diminuindo imenso a complexidade computacional e consequentemente o tempo de execução.

F. Avaliar a aplicação dos algoritmos para problemas muitos maiores

Na geração dos grafos para análise dos seus limites, consegui gerar soluções para um grafo com 25 vértices e 207 arestas em 0.83 segundos. Algo que não era possível para o algoritmo de força bruta. Contudo, aumentando a fasquia, para analisar os grafos fornecidos pelo docente (o Medium com 250 vértices e 1273 arestas) o programa ficou a correr durante muitas horas, tornando-se inviável, logo já nem avancei para testes ao Large.

Através do ficheiro *graph_generator_randomized.py* foram criados grafos e fomos testando o limite:

Vértices	Arestas	Tempo
30	198	0.95s
51	924	12s
75	2029	50.24s
120	5266	≃ 6 minutos

Tabela 4 - Tentativas de encontrar o limite da função de pesquisa aleatória

Observando a tabela 4, analisamos que o grafo com 120 vértices e 5266 arestas já demora quase 6 minutos, significa que já exige muita complexidade computacional, tornando já inviável a aplicação deste algoritmo para grafod com estas (ou superiores) características por já demorar bastante tempo. Ou seja, para problemas muito maiores, este algoritmo não seria de todo uma boa opção. Ganhando aqui novamente o algoritmo de pesquisa gulosa.

V. Conclusão

Por toda a análise realizada em relação aos algoritmos de pesquisa aleatória, os resultados sobre o problema em si definitivamente foram melhores com o algoritmo de pesquisa aleatória versão 2.

No primeiro projeto, chegámos à conclusão que o algoritmo de força bruta apesar de chegar à solução ideal tinha uma complexidade computacional não muito favorável. O algoritmo desenhado para este trabalho, permitiu atingir valores altos para o peso do corte, por vezes em tempos mais baixos (para grafos maiores) e com muito menos iterações logo menos complexidade computacional exigida).

Para concluir, dependendo das características do grafo, o utilizador poderá preferir utilizar o *brute force* para grafos com menos de 20 vértices ou o algoritmo de pesquisa aleatória para grafos com menos de 120 vértices. Para grafos maiores, recomenda-se a utilização do algoritmo de pesquisa gulosa.

Referências

[1] P. Festa, P.M. Pardalos, M.G.C. Resende & C.C. Ribeiro (2002) Randomized heuristics for the Max-Cut problem, Optimization Methods and Software, 17:6, 1033-1058, DOI: 10.1080/1055678021000090033

[2] Sera Kahruman, Elif Kolotoglu, Sergiy Butenko and Illya V. Hicks (2002) On greedy construction heuristics for the MAX-CUT problem

[3] Corte Máximo - Wikipédia

[4] Geeks for Geeks | Randomized Algorithm

[5] Lecture 6 Overview 1 Max Cut