Problema Max-Cut: Algoritmos Aleatorizados

Hugo Veríssimo - 124348 - hugoverissimo@ua.pt

***Abstract* – This report presents the implementation and comparison of three randomized algorithms to solve the Maximum Weight Cut problem: a random cut algorithm, a simulated annealing algorithm, and a random greedy algorithm. The problem consists of dividing a graph into two complementary subsets in order to maximize the sum of the weights of the edges connecting the two subsets. The analysis of the algorithms was performed based on different metrics, such as the number of basic operations, execution time, and the accuracy of the solutions obtained, on graphs of varying sizes. The results show a balance between computational complexity and solution quality, indicating that the choice of the most suitable algorithm depends on the specific characteristics of each problem.**

***Resumo* – Este relatório apresenta a implementação e comparação de três algoritmos aleatorizados para resolver o problema do *Maximum Weight Cut*: um algoritmo de Corte Aleatório, um algoritmo de *Simulated Annealing* e um algoritmo Guloso Aleatório. O problema consiste em dividir um grafo em dois subconjuntos complementares de modo a maximizar a soma dos pesos das arestas que ligam os dois subconjuntos. A análise dos algoritmos foi realizada com base em diferentes métricas, tais como número de operações básicas, tempo de execução e precisão das soluções obtidas, em grafos de diferentes dimensões. Os resultados mostram um equilíbrio entre a complexidade computacional e a qualidade da solução, indicando que a escolha do algoritmo mais adequado depende das características específicas de cada problema.**

I. Introdução

O problema *Maximum Weight Cut* é um problema de otimização, que tem como objetivo encontrar o corte mais pesado num grafo não direcionado $G(V, E)$, onde $|V| = n$ vértices e $|E| = m$ arestas. Este corte envolve dividir os vértices do grafo em dois subconjuntos disjuntos S e T, sendo que o peso do corte é a soma dos pesos das arestas que ligam os vértices de S aos vértices de T: $|E(S, T)|$ [1].

No passado relatório foram analisados algoritmos determinísticos para resolver o problema *Maximum Weight Cut*, nomeadamente a Pesquisa Exaustiva e a heurística gulosa. Neste relatório, serão analisados novos algoritmos com um certo grau de aleatoriedade, com o objetivo de encontrar um algoritmo que otimize o equilíbrio entre a complexidade computacional e a qualidade da solução obtida. Os algoritmos a serem analisados são o algoritmo de Corte Aleatório, o algoritmo de *Simulated Annealing* e o algoritmo Guloso Aleatório.

Para além disso, será também analisada a complexidade de cada algoritmo, o número de operações básicas realizadas, o tempo de execução, o número de soluções testadas e a precisão das soluções obtidas, com o intuito de comparar os diferentes algoritmos e determinar o mais eficiente e eficaz na resolução do problema em questão. Os resultados alcançados com estes novos algoritmos, serão também comparados com os resultados obtidos com os algoritmos determinísticos, analisados no relatório anterior.

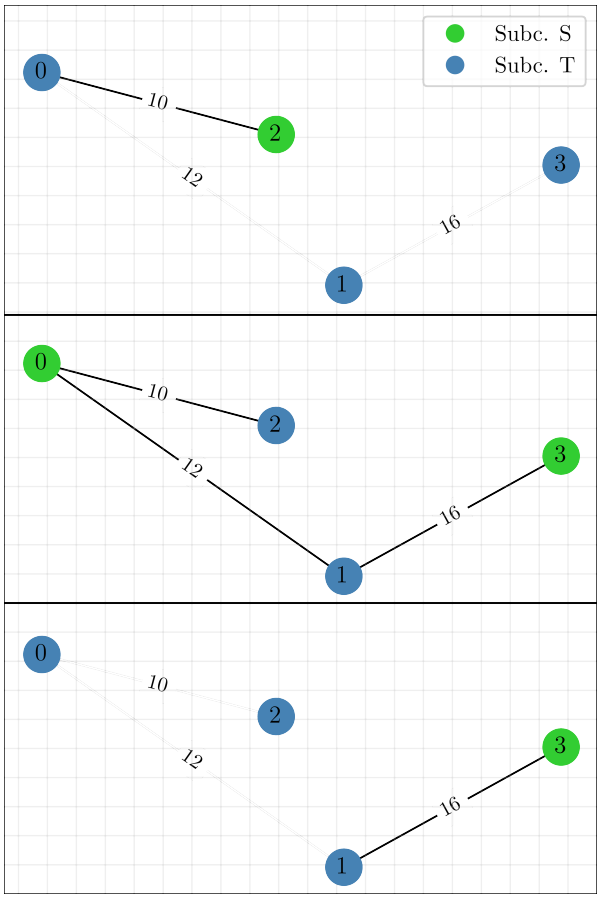

Fig. 1: Exemplo ilustrativo da aplicação dos algoritmos de Corte Aleatório, *Simulated Annealing* e Guloso Aleatório, ao grafo *0004_500.graphml*, pela respetiva ordem. Note-se que os parâmetros foram ajustados para fins demonstrativos, pelo que os resultados apresentados podem não corresponder aos reais.

II. METODOLOGIA DA ANÁLISE

Com o intuito de analisar o problema em destaque, implementar os algoritmos referidos e comparar os resultados obtidos, foi utilizada a linguagem de programação *Python*, devido à vasta variedade de bibliotecas que contém, facilitando, assim, a implementação eficiente e simplificada dos algoritmos necessários.

A análise desenvolvida tem por base o ficheiro `benchmark.py`, responsável por implementar os algoritmos em estudo, acompanhar as métricas relevantes para os resultados a analisar e guardar as mesmas num ficheiro *.xlsx* (*Excel*). Não obstante, também foram usados outros ficheiros, tais como `benchmarks (mergedf).ipynb` e `benchmarks (plots).ipynb`, que contribuiram para a análise dos resultados obtidos.

Para a realização da análise dos algoritmos criados, foram utilizados grafos gerados aleatoriamente com a semente 124348, diferentes números de vértices e densidade de arestas, através do ficheiro `graphs.py`, e os grafos da coleção *Gset*, disponibilizada por Yinyu Ye [2].

III. ALGORITMO DE CORTE ALEATÓRIO

O primeiro algoritmo a ser implementado foi um algoritmo de Corte Aleatório, que consiste em gerar várias soluções aleatórias, comparar as mesmas e escolher a melhor entre elas [1].

Este é um algoritmo computacionalmente leve, pela sua simplicidade, mas não garante a obtenção da solução ótima, devido à sua natureza aleatória, sendo que a probabilidade de encontrar a mesma, assumindo que é única, é dada por

$$1 - \left(1 - \frac{1}{2^{n-1}}\right)^{solutions}$$

onde n corresponde ao número de vértices e *solutions* ao número de soluções a gerar. Pode-se facilmente verificar que, para grafos de grandes dimensões, esta probabilidade decresce exponencialmente, tornando o algoritmo cada vez menos preciso.

Pelo facto do algoritmo gerar muitas soluções aleatórias, é importante garantir que não existem soluções repetidas a serem testadas, evitando o cálculo do peso do corte, uma operação computacionalmente cara. Para isso foi criado um *set* onde são guardadas as soluções já testadas, e cada vez que uma solução é gerada, a mesma só será testada depois de ser verificado que não é uma repetição.

Atendendo à paragem do algoritmo, este tem dois critérios de paragem, interrompendo o processo assim que um deles é verificado. O primeiro, e mais provável em grafos de grandes dimensões, é quando o número de soluções geradas atinge o limite, definido pelo utilizador. O segundo critério, é verificado quando todas as soluções possíveis são testadas, ou seja, quando o *set* que acompanha as soluções testadas contém 2^n elementos.

Este algoritmo pode ser então traduzido para o seguinte pseudocódigo:

Algoritmo 1 Corte Aleatório

Entrada:
- lista de arestas e respetivos pesos (*edges*)
- número de vértices (*n_nodes*)
- número de soluções a gerar (*solutions*)

Saída: subconjuntos S e T, peso do corte (*weight*)

```
best_solution ← None
weight ← 0
seen_solutions ← empty set
for i ← 1 to solutions do
    partition ← random partition of the nodes
    if length(seen_solutions) = 2^n_nodes then
        break
    end if
    partition_hash ← hash the partition
    if partition_hash ∈ seen_solutions then
        continue
    end if
    Add partition_hash to seen_solutions
    new_cut_weight ← compute the cut weight
    if new_cut_weight > weight then
        weight ← new_cut_weight
        best_solution ← copy of partition
    end if
end for
S ← set of nodes assigned to 0 in best_solution
T ← set of nodes assigned to 1 in best_solution
return S, T, weight
```

Pode-se observar que a parte computacionalmente mais custosa deste algoritmo é o ciclo, que é responsável por gerar cortes aleatórios e calcular o peso dos mesmos. A complexidade deste ciclo é dado por $O(n+m)$, por percorrer todos os vértices, atribuindo-os a um dos subconjuntos, e por calcular o peso do corte, percorrendo todas as arestas. Assim, a complexidade final do algoritmo é dada por $O((n+m) \times \texttt{solutions})$, visto que o ciclo corre no máximo `solutions` vezes. Esta complexidade pode ser simplificada para $O(m)$, visto que $m \gg n$ para grafos de grandes dimensões, e que `solutions` é uma constante.

A complexidade referida é confirmada pela análise experimental apresentada na Fig. 2, onde se pode observar que o número de operações básicas realizadas é linear em relação ao número de arestas do grafo, para diferentes números de soluções geradas. Nesta figura as retas apresentadas são dadas pela função

$$f(m) = m \times \text{MS}$$

onde MS é o número de soluções a gerar (`solutions`). Quanto aos pontos acima das retas que representam a complexidade do algoritmo, estes são devido ao facto de que para grafos de menores dimensões é possível que não se possa simplificar a complexidade para $O(m)$, visto que n e m são da mesma ordem de grandeza.

Além disso, também se pode observar que o número de operações básicas realizadas é menor para um menor número de soluções geradas, o que é esperado, visto que o algoritmo realiza menos iterações.

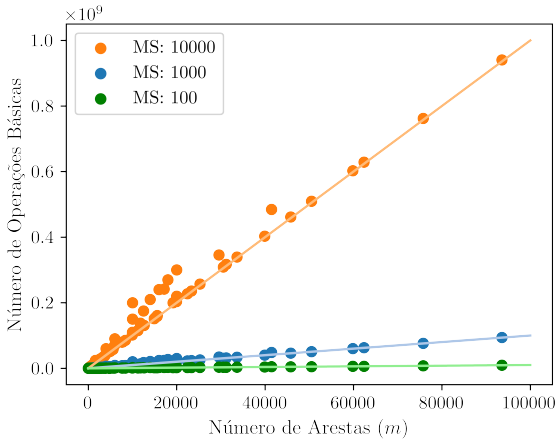

Fig. 2: Número de operações básicas efetuadas pelo algoritmo de Corte Aleatório em função do número de arestas do grafo, para diferentes valores de `solutions` (MS).

Quanto ao número de soluções testadas, a partir da Fig. 3 e da análise do pseudocódigo, é possível verificar que o número é dado por $\min(2^n, \texttt{solutions})$.

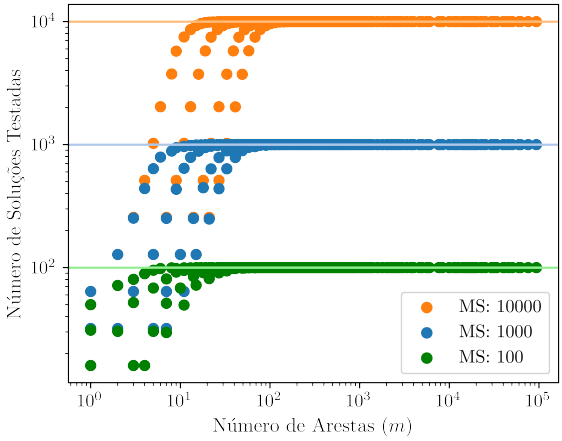

Fig. 3: Número de soluções testadas pelo algoritmo de Corte Aleatório em função do número de arestas do grafo, para diferentes valores de `solutions` (MS). As retas horizontais são dadas por $f(m) = MS$.

IV. Algoritmo de Simulated Annealing

O segundo algoritmo a ser implementado foi o algoritmo de *Simulated Annealing* (SA), uma heurística de pesquisa aleatória, que consiste em encontrar soluções aproximadas para problemas de otimização combinatória [3]. Este método consite em gerar uma solução inicial aleatória, e a partir desta, gerar soluções vizinhas, que são obtidas a partir da solução atual, compará-las, aceitando as melhores e algumas das piores, com uma probabilidade que decresce ao longo das iterações, até que a temperatura (parâmetro do algoritmo), que vai arrefecendo ao longo das iterações a uma determinada taxa de arrefecimento (parâmetro do algoritmo), seja menor que um determinado valor, por exemplo 10^{-3}, sendo este o critério de paragem do algoritmo [4].

Assim, é possível verificar que este algoritmo tem como componente aleatória a seleção de uma solução inicial e a aceitação de soluções piores, e como parte determinística a diminuição da probabilidade de aceitação de soluções piores ao longo das iterações e a garantia de aceitação de soluções melhores.

Na Fig. 4, pode-se observar o comportamento decrescente da probabilidade de aceitação de soluções piores ao longo das iterações do algoritmo de SA. Os pontos cinzentos representam números aleatórios gerados dentro do intervalo $[0, 1]$ a cada iteração. A proposta de uma nova solução é aceite se esta for melhor que a solução atual ou se o valor aleatório (ponto cinzento) estiver abaixo da *curva* dada por $\exp(\Delta\text{Peso do Corte}/T)$ (pontos azuis). Esta *curva* reflete a probabilidade de aceitação, que diminui à medida que a temperatura decresce, limitando cada vez mais a aceitação de soluções subótimas.

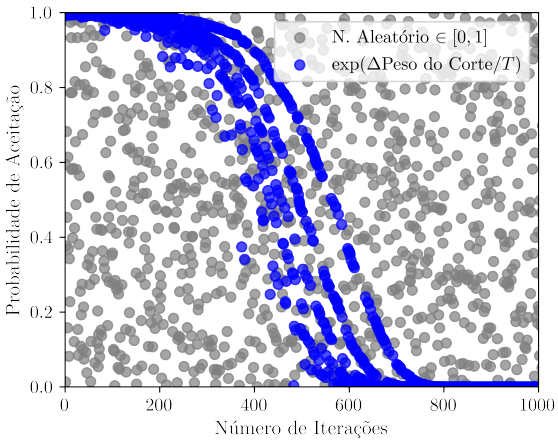

Fig. 4: Probabilidade de aceitação de uma solução subótima, em função do número de iterações, quando o algoritmo de SA é aplicado com a semente 124348, ao grafo G59.

É também importante referir que o algoritmo não garante a não repetição de soluções previamente testadas e que não foi implementado um mecanismo para tal, visto que a probabilidade de testar a mesma solução é baixa para grafos de grandes dimensões e o processo de comparação de soluções já testadas prejudicaria a eficiência do algoritmo.

O algoritmo de *Simulated Annealing* pode ser examinado em detalhe no seguinte pseudocódigo:

Algoritmo 2 *Simulated Annealing*

Entrada:
- lista de arestas e respetivos pesos (*edges*)
- temperatura (*Temp*)
- taxa de arrefecimento (*cooling_rate*)

Saída: subconjuntos S e T, peso do corte (*best_cut*)

```
partition ← random partition of the nodes
best_partition ← partition
current_cut ← compute the cut weight
best_cut ← current_cut
while Temp > 10^-3 do
    node ← randomly select a node
    Flip the partition of node in partition
    new_cut ← compute the new cut weight
    cost_diff ← new_cut − current_cut
    if cost_diff > 0 or random number ∈ [0, 1] < e^(cost_diff/Temp) then   ▷ Accept the move
        current_cut ← new_cut
        if new_cut > best_cut then
            best_cut ← new_cut
            best_partition ← partition
        end if
    else                                                                   ▷ Reject the move
        Revert the partition of node in partition
    end if
    Temp ← Temp × cooling_rate
end while
S ← set of nodes assigned to 0 in best_partition
T ← set of nodes assigned to 1 in best_partition
return S, T, best_cut
```

Tal como foi referido na descrição do algoritmo, pode-se verificar que este é sensível à solução inicial, pelo que será interessante executar o algoritmo várias vezes, cobrindo uma maior área do espaço de soluções.

Para além disso, através do pseudocódigo, é possível analisar a complexidade do algoritmo em questão. As operações computacionalmente mais custosas encontram-se dentro do ciclo, que só termina após a temperatura ser inferior a 10^{-3}. Assim, torna-se importante calcular o total de iterações (k) que o ciclo irá realizar, o que pode ser feito através da seguinte equação:

$$\texttt{Temp}_0 \cdot (\texttt{cooling\_rate})^k \leq 10^{-3}$$
$$\Leftrightarrow k = \left\lceil \frac{\log\left(\frac{10^{-3}}{\texttt{Temp}_0}\right)}{\log(\texttt{cooling\_rate})} \right\rceil$$

Quanto à complexidade dentro do ciclo, a mesma é dada por $O(m)$, visto que a operação mais custosa é o cálculo do peso do corte, que percorre todas as arestas do grafo. Assim, a complexidade final do algoritmo é dada por $O(m \times k)$, e pelo facto de k depender apenas da temperatura inicial e da taxa de arrefecimento, a complexidade final pode ser aproximada por $O(m)$.

Através da Fig. 5 que representa o número de operações básicas realizadas em função do número de arestas do grafo, para diferentes temperaturas iniciais e taxas de arrefecimento, pode-se verificar que a complexidade do algoritmo é linear em relação ao número de arestas do grafo. As curvas apresentadas são dadas pela função $f(m) = m \times k$, que consegue descrever o comportamento do algoritmo para grafos de diferentes dimensões, confirmando a complexidade calculada. Também se pode observar que são executadas mais operações básicas para temperaturas iniciais e taxas de arrefecimento mais altas, o que é esperado, visto que o algoritmo realiza mais iterações (maior k).

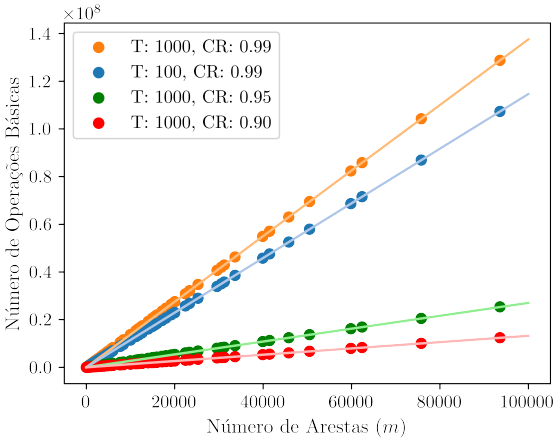

Fig. 5: Número de operações básicas efetuadas pelo algoritmo de SA em função do número de arestas do grafo, para diferentes valores de `Temp` (T) e de `cooling_rate` (CR).

Quanto ao número de soluções testadas, a partir da Fig. 6, é possível verificar que o mesmo é constante e é igual a k, ou seja, é testada uma solução por iteração do algoritmo.

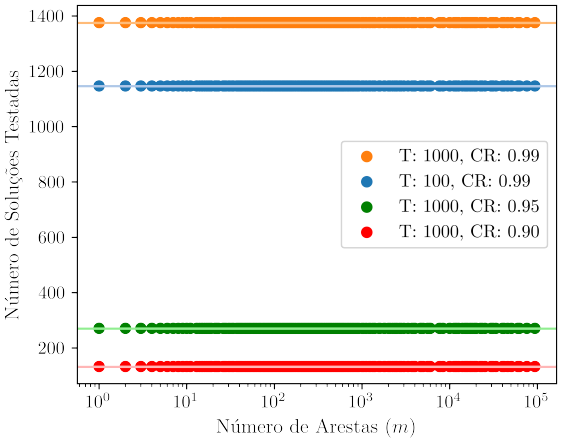

Fig. 6: Número de soluções testadas pelo algoritmo de SA em função do número de arestas do grafo, para diferentes valores de `Temp` (T) e de `cooling_rate` (CR). As retas horizontais são dadas por $f(m) = k$.

V. Algoritmo Guloso Aleatório

Por fim, o terceiro algoritmo a ser implementado foi um algoritmo Guloso Aleatório, que segue uma heurística baseada numa abordagem gulosa, não garantido encontrar a solução ótima. Este algoritmo itera sobre todos os vértices do grafo e, para cada vértice, troca a sua partição, verificando se a nova configuração melhora a solução atual. Caso o peso do corte com o vértice na partição oposta seja maior que o atual, a solução é atualizada. O processo continua até que uma iteração completa seja realizada sem encontrar melhorias, momento em que o algoritmo termina.

Devido ao critério de paragem do algoritmo, este pode correr indefinidamente dado a natureza aleatória da solução inicial, que pode estar a um grande número de iterações da solução estável que o algoritmo procura. Por isso, é adicionado um fator de ajuste do máximo de iterações (*itLim*) ao algoritmo, tornando o número máximo de iterações do mesmo $m \times itLim$.

Este algoritmo pode ser representado em pseudocódigo da seguinte forma:

Algoritmo 3 Guloso Aleatório

Entrada:
- lista de arestas e respetivos pesos (*edges*)
- número de vértices (*n_nodes*)
- fator de ajuste do máximo de iterações (*itLim*)

Saída: subconjuntos S e T, peso do corte (*weight*)

```
partition ← random partition of the nodes
cut_weight ← compute the cut weight
improved ← True
it_limit ← len(edges) × itLim
while improved and it_limit > 0 do
    it_limit ← it_limit −1
    improved ← False
    for node in range(n_nodes) do
        Flip the partition of node in partition
        new_cut_weight ← compute the cut weight
        if new_cut_weight > cut_weight then
            cut_weight ← new_cut_weight
            improved ← True
            break      ▷ Stop iteration for this node
        end if
        Revert the partition of node in partition
    end for
end while
S ← Set of nodes assigned to 0 in partition
T ← Set of nodes assigned to 1 in partition
return S, T, cut_weight
```

É também importante referir que, pelo facto da única componente aleatória deste algoritmo ser a criação de uma solução inicial, e como todas as iterações realizam alterações em direção à melhor solução, o algoritmo nunca irá testar a mesma solução mais que uma vez, pelo que manter um registo sobre as soluções já testadas não é necessário. Pela mesma razão, o algoritmo é sensível à solução inicial, pelo que será interessante executar o algoritmo várias vezes, de modo a cobrir uma maior área do espaço de soluções, garantido uma maior probabilidade de encontrar uma solução próxima da ótima, ou até a própria.

Quanto à complexidade deste algoritmo, através da análise do pseudocódigo é possível verificar que, pelo facto das operações mais custosas serem percorrer todos os vértices ($O(n)$) e calcular o novo peso do corte, ao mudar o vértice de partição ($O(m)$), e ambas se encontrarem dentro do ciclo principal, que itera no máximo $m \times$ `itLim` vezes, a complexidade deste algoritmo é dada por $O(m \times \texttt{itLim} \times n \times m)$. Esta pode ser reescrita e simplificada de diferentes maneiras, entre elas $O(m^2 \times n)$, visto que `itLim` é uma constante.

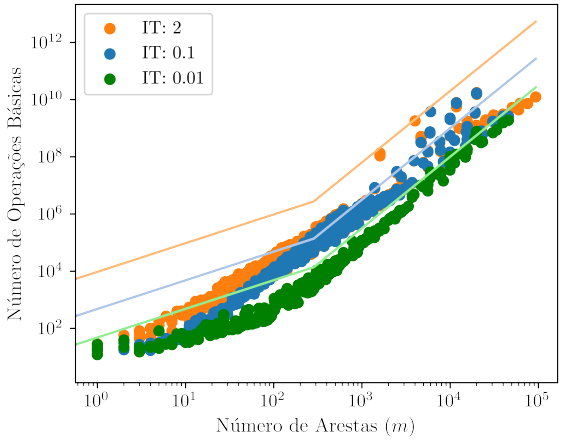

Fig. 7: Número de operações básicas efetuadas pelo algoritmo Guloso Aleatório em função do número de arestas, para diferentes valores de `itLim` (IT).

Através da Fig. 7, que apresenta o número de operações básicas efutadas pelo algoritmo em função do número de arestas, para diferentes valores de `itLim` (IT), é possível validar a complexidade do algoritmo calculada analiticamente. Pode-se observar que as curvas apresentadas, dadas pela função

$$f(m) = m^{2.5} \times \text{IT}$$

conseguem descrever o comportamento do algoritmo para grafos de diferentes dimensões, confirmando a complexidade calculada. É importante referir que a complexidade calculada analiticamente foi simplificada para $O(m^{2.5} \times \texttt{itLim})$, visto que para grafos com maior número de vértices e de arestas, onde é mais interessante aplicar este tipo de algoritmos, tem-se que

$$m \approx \frac{n(n-1)}{2} \Rightarrow n \approx \sqrt{m}$$

Quanto ao número de soluções testadas, a partir da análise do pseudocódigo, é possível verificar que o número é variável, devido à natureza do algoritmo. Sempre que é encontrada uma solução melhor que a atual, no algoritmo, este termina a iteração que está a ser efetuada, sem testar mais soluções, pelo que se um dos primeiros vértices testados melhorar a solução, o

algoritmo não irá testar os restantes, fazendo com que sejam testadas menos soluções. De qualquer forma, é possível verificar que o número de soluções testadas nunca será superior a $m \times \texttt{itLim} \times n$, visto que a cada iteração do ciclo principal ($m \times \texttt{itLim}$ iterações máximas), são testadas no máximo n soluções (testar todos os vértices). Na Fig. 8 é validada a análise feita, onde é apresentado o número de soluções testadas em função do número de arestas do grafo e a função $f(m) = m^{1.5} \times \text{IT}$, para diferentes valores de `itLim` (IT). É de notar que foi utilizada mais uma vez a aproximação $n \approx \sqrt{m}$.

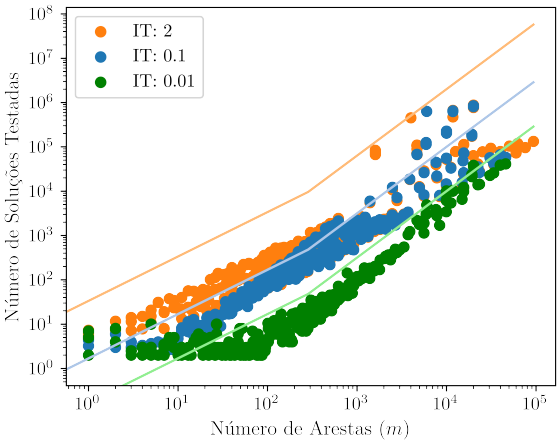

Fig. 8: Número de soluções testadas pelo algoritmo Guloso Aleatório em função do número de arestas do grafo, para diferentes valores de `itLim` (IT). As curvas apresentadas são dadas por $f(m) = m^{1.5} \times \text{IT}$.

VI. Análise dos Resultados

Após a implementação e execução dos algoritmos em estudo, torna-se possível a análise dos resultados obtidos, nomeadamente uma análise de complexidade entre todos os algoritmos, através do número de operações básicas realizadas, do tempo de execução dos algoritmos e das soluções obtidas. Para além da análise dos algoritmos em estudo neste relatório, também é realizada uma comparação com os algoritmos determinísticos, analisados no relatório anterior, nomeadamente a Pesquisa Exaustiva e a Pesquisa Gulosa.

Para esta análise, foram utilizados o conjunto de grafos gerados aleatoriamente com a semente 124348, a que será chamado *Gpeq*, por serem grafos de menores dimensões, e os grafos do conjunto *Gset*. Entre os diferentes grafos, podem ser encontrados diversos números de vértices e densidades de arestas, garantindo a representação de uma ampla gama de configurações estruturais e cenários computacionais.

Dada a componente aleatória dos algoritmos, para assegurar resultados mais precisos e consistentes, cada grafo foi processado múltiplas vezes, calculando-se a média para as diferentes métricas (número de operações básicas, precisão, etc.), com exceção do tempo de execução, para o qual foi calculado a mediana. A escolha da mediana para esta métrica deve-se à sua robustez contra valores atípicos que podem surgir devido à falta de controlo sobre o ambiente de execução dos algoritmos.

A. Análise do Número de Operações

Atendendo ao número de operações básicas que um algoritmo executa, esta métrica ao estar diretamente relacionada com a complexidade de um algoritmo, já foi analisada anteriormente, na secção correspondente a cada algoritmo, de modo a validar a complexidade calculada formalmente.

Na Tabela I, pode-se observar a complexidade dos algoritmos tendo por base o número de operações básicas realizadas, permitindo a comparação da eficiência computacional entre os diferentes algoritmos implementados.

TABELA I: Complexidade dos algoritmos pelo número de operações básicas.

Algoritmo	**Complexidade**
Pesquisa Exaustiva	$O(2^n)$
Pesquisa Gulosa	$O(m \log m)$
Corte Aleatório	$O(m)$
Simulated Annealing	$O(m)$
Guloso Aleatório	$O(m^2 \times n)$

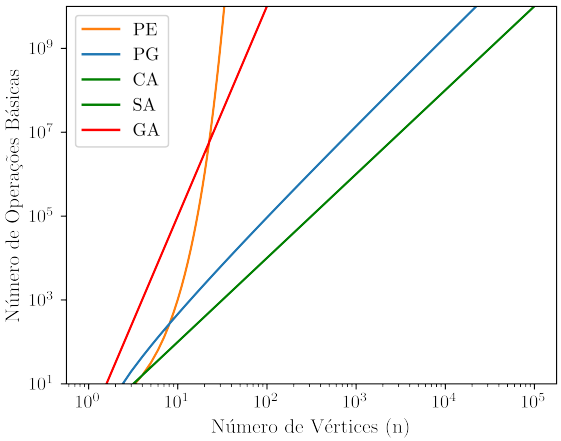

Fig. 9: Número total de operações básicas realizadas pelos diferentes algoritmos, em função do número de vértices do grafo. A legenda apresentada indica as iniciais dos algoritmos.

Tendo em conta que para grafos de maiores dimensões, com elevado número de arestas, a relação entre n e m é dada por $n^2 \approx m$, torna-se possível realizar a visualização apresentada na Fig. 9. Pode-se verificar a ordem pela qual os algoritmos se tornam impraticáveis devido ao seu elevado número de operações, sendo a Pesquisa Exaustiva (PE) o primeiro a tornar-se impraticável, seguido do Guloso Aleatório (GA), Pesquisa

Gulosa (PG) e, por fim, o Corte Aleatório (CA) e o Simulated Annealing (SA), com igual complexidade. Apesar desta análise ser possível através da Tabela I, a visualização gráfica permite uma melhor compreensão da complexidade dos algoritmos.

B. Análise do Tempo de Execução

Quanto à análise do tempo de execução de cada algoritmo, esta métrica por ser mais sensível a fatores externos, como a carga do sistema, não terá a comparação entre os algoritmos estudados neste relatório face aos estudados no relatório anterior, pelo facto da igualdade das condições de execução não ser garantida.

Atendendo aos algoritmos abordados ao longo deste relatório, todos eles foram implementados sob as condições mais homogéneas possíveis, de modo a garantir a igualdade de condições de execução. Quanto ao *hardware* usado, o mesmo manteve-se constante, sendo que todas as execuções foram realizadas num *MacBook Air*, com um processador *Apple M1* e 16GB de memória RAM.

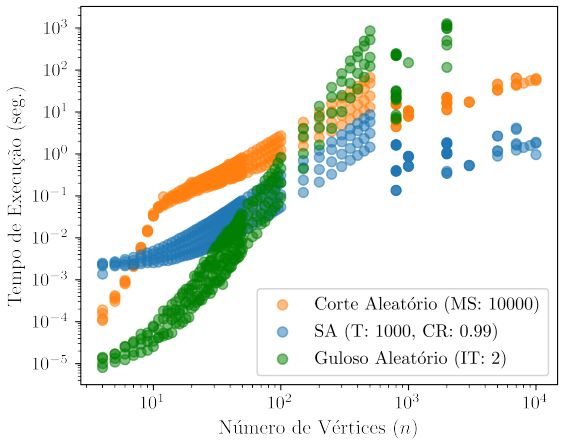

Fig. 10: Tempo de execução, em segundos, dos diferentes algoritmos em função do número de vértices do grafo.

Através da análise da Fig. 10, que apresenta o tempo de execução para um exemplo de cada algoritmo, para não sobrecarregar tanto a análise, em função do número de vértices do grafo, é possível verificar certas *irregularidades* nas *curvas* formadas pelos pontos referentes a cada um dos algoritmos.

No caso do algoritmo de Corte Aleatório (pontos laranja), os menores tempos de execução observados para grafos com número reduzido de vértices, podem ser explicados pelo facto do algoritmo não atingir o número máximo de soluções a gerar, por atingir todas as soluções possíveis, previamente.

Quanto às *irregularidades* observadas nos grafos de maiores dimensões, que pertencem ao conjunto *Gset*, é possível que estejam relacionadas com a forma como é lido o ficheiro que contém o grafo, o que pode, de alguma forma, impactar o desempenho do *hardware* durante a execução do algoritmo ou com a densidade de arestas dos grafos.

Com o objetivo de ajustar a função que melhor descreve o comportamento do tempo de execução dos algoritmos, foi criado um intervalo para o número de vértices de modo a ignorar as *irregularidades* observadas. Através da análise da Fig. 11, onde é realizado esse ajuste, é possível, de forma não muito clara, devido à escala do eixos, verificar o comportamento polinomial dos tempos de execução dos algoritmos e que o grau polinomial do algoritmo Guloso Aleatório é superior ao dos restantes.

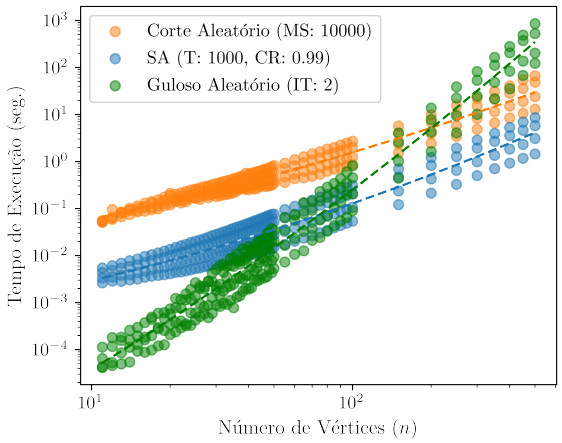

Fig. 11: Tempo de execução, em segundos, dos diferentes algoritmos em função do número de vértices de grafos selecionados.

Ao ajustar diferentes funções polinomiais aos tempos de execução dos algoritmos, verificou-se que o comportamento do tempo de execução é melhor descrito por uma função polinomial de quinto grau, no caso do algoritmo Guloso Aleatório, enquanto que para os demais algoritmos, a melhor descrição é dada por uma função polinomial de segundo grau. Estes resultados estão em conformidade com o esperado, considerando a complexidade dos algoritmos em função do número de vértices, através da aproximação $m \approx n^2$.

C. Análise das Soluções Obtidas

Tendo em conta as soluções obtidas pelos diferentes algoritmos, torna-se interessante analisar a precisão das soluções, ou seja, comparar os resultados aproximados com as soluções ótimas (ou conhecidas) dos grafos em estudo. Para tal, foi calculada a precisão das soluções obtidas, em percentagem, em relação à solução ótima, para cada um dos algoritmos, e para cada grafo.

De modo a discretizar o desempenho dos algoritmos em grafos de diferentes dimensões, foi calculada a média e desvio padrão das precisões do conjunto de grafos *Gpeq* e *Gset*, relembrando que os conjuntos contêm os grafos de menores e maiores dimensões, respetivamente.

Na Tabela II pode-se verificar a média e o desvio

padrão da precisão das soluções obtidas pelos diferentes algoritmos, para os diferentes conjuntos de grafos. É de notar que a Pesquisa Exaustiva não foi executada para os grafos do conjunto *Gset*, devido ao elevado número de operações básicas que o algoritmo executa, tornando-se impraticável para grafos de maiores dimensões. Os algoritmos que não completaram a execução para todos os grafos de um derminado conjunto, devido ao tempo de execução elevado, foram assinalados com um asterisco (*).

TABELA II: Média e desvio padrão (entre parênteses) da precisão das soluções obtidas pelos diferentes algoritmos, para os diferentes conjuntos de grafos.

Algoritmo	**Gpeq**	**Gset**
Pesquisa Exaustiva	100% (0)	–
Pesquisa Gulosa	93% (6.9)	61% (23.1)
Corte Aleat. (MS: 10^2)	92% (6.3)	41% (35.6)
Corte Aleat. (MS: 10^3)	96% (4.5)	42% (35.0)
Corte Aleat. (MS: 10^4)	98% (2.4)	43% (34.6)
SA (T: 10^2, CR: 0.99)	99% (1.3)	58% (28.7)
SA (T: 10^3, CR: 0.9)	97% (3.1)	40% (36.1)
SA (T: 10^3, CR: 0.95)	98% (2.6)	44% (34.4)
SA (T: 10^3, CR: 0.99)	99% (1.3)	59% (28.5)
Gul. Aleat. (IT: 10^{-2})	75% (14.5)	58% (30.1)*
Gul. Aleat. (IT: 10^{-1})	84% (12.4)	90% (7.7)*
Gul. Aleat. (IT: 2)	96% (3.7)	88% (8.4)*

Pode-se verificar que, de uma forma geral, todos os algoritmos apresentam uma precisão elevada para os grafos do conjunto *Gpeq*, o que não é muito interessante, visto que para grafos deste conjunto, a Pesquisa Exaustiva não tem grandes problemas de eficiência e garante a solução ótima.

Por outro lado, para os grafos do conjunto *Gset*, a Pesquisa Exaustiva torna-se impraticável, pelo que a precisão dos restantes algoritmos é mais relevante. Observa-se que o algoritmo Guloso Aleatório é aquele que apresenta melhores resultados para este conjunto, mas o facto de não ter completado a execução para todos os grafos, devido ao tempo de execução elevado, reforça a importância do equilíbrio entre a precisão e a complexidade do algoritmo. O segundo algoritmo em destaque é a Pesquisa Gulosa, que por sua vez, é o algoritmo com maior complexidade, excluindo a Pesquisa Exaustiva e o Guloso Aleatório.

É interessante observar que, para o conjunto de grafos *Gset*, a ordem de precisão dos algoritmos é a mesma que a ordem de complexidade, reforçando mais uma vez, a existência de um equilíbrio entre a precisão e a eficiência computacional dos algoritmos.

VII. Conclusão

Em suma, pode-se apurar que existem diversos algoritmos para a resolução do problema *Maximum Weight Cut*, tanto determinísticos como aleatorizados, cada um com as suas vantagens e desvantagens.

Através da análise dos resultados obtidos com a implementação das diferentes abordagens, foi possível verificar que há um equilíbrio entre a precisão e a eficiência computacional dos algoritmos. Enquanto que algoritmos como a Pesquisa Exaustiva e o Guloso Aleatório apresentam uma precisão elevada, são algoritmos com uma complexidade elevada, tornando-se impraticáveis para grafos de maiores dimensões. Por outro lado, algoritmos como o Corte Aleatório e o *Simulated Annealing* apresentam uma precisão mais baixa, mas com uma complexidade correspondente, tornando-se mais eficientes para grafos de maiores dimensões.

Isto revela a importância de escolher o algoritmo mais adequado para o problema em questão, tendo em conta as suas características, como o número de vértices e arestas do grafo, o tempo disponível para a execução do algoritmo e a precisão desejada para a solução obtida.

Bibliografia

[1] Anupam Gupta, "15-854: Approximations algorithms", 2014, https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec02.pdf. Accessed: 2024-11-28.

[2] Yinyu Y. e S. Karisch, "Gset: A collection of graphs for benchmarking", Stanford University, n.d., https://web.stanford.edu/ yyye/yyye/Gset/. Accessed: 2024-11-02.

[3] Tor G. J. Myklebust, "Solving maximum cut problems by simulated annealing", 2015, https://arxiv.org/abs/1505.03068. Accessed: 2024-11-28.

[4] Galen Hajime Sasaki, "Optimization by simulated annealing: A time-complexity analysis.", Defense Technical Information Center, 1987, https://apps.dtic.mil/sti/pdfs/ADA185547.pdf. Accessed: 2024-11-28.