Guilherme Dytz dos Santos - 244475
Lucas Spagnolo Bombana - 314879

Trabalho Final
INF05010 - Otimização Combinatória
Prof. Dr. Marcus Ritt

2021/1

1 Introdução

O objetivo deste trabalho foi implementar uma meta-heurística para resolver o Problema de Emparelhamento Diversificado (definido na Seção 2). A meta-heurística escolhida foi a Simulated Annealing (SA), cujos detalhes serão expandidos na Seção 4.

2 Definição do Problema

Dado um grafo não-direcionado $G = (V, A)$, onde cada aresta $a \in A$ possui um tipo t_a, deseja-se encontrar um emparelhamento $M \subseteq A$ tal que todas as arestas $m \in M$ possuem tipos diferentes, além disso, deseja-se que esse emparelhamento possua o maior número de arestas possíveis. Sabe-se que este problema é conhecidamente $\mathcal{NP}$-Difícil.

3 Formulação do programa inteiro

Seja o grafo $G = (V, A)$ definido na Seção 2.

3.1 Variáveis de Decisão

São necessárias variáveis associadas a cada aresta do grafo. Essas variáveis são definidas da seguinte forma:

$$x_a \in \mathbb{B} \rightarrow 1 \text{ caso a aresta } a \text{ seja selecionada, } 0 \text{ caso contrário.} \qquad \forall a \in A$$

3.2 Função Objetivo

Como o objetivo é maximizar o número de arestas selecionadas, basta fazer a seguinte função objetivo:

$$\textbf{maximiza} \quad \sum_{a \in A} x_a$$

3.3 Restrições

Agora é necessário garantir que as arestas selecionadas formam um emparelhamento e possuam tipos diferentes dentro desse emparelhamento.

1. **Para garantir o emparelhamento:** Fixando cada vértice $u \in V$, tem-se que E_u é o conjunto de arestas que incidem sobre u, então para as restrições que garantem um emparelhamento tem-se:

$$\sum_{a=(u,v) \in E_u} x_a \leq 1 \qquad \forall u \in V.$$

2. **Para garantir que cada aresta possua um tipo diferente:** Seja T o conjunto de todos os tipos de arestas. Para cada tipo $t \in T$, existe um conjunto de arestas E_t que possuem esse tipo. Com isso, basta fazer as seguintes restrições que garantem tipos diferentes no emparelhamento:

$$\sum_{a \in E_t} x_a \leq 1 \qquad \forall t \in T.$$

4 Principais Elementos da Abordagem

Nessa seção, serão detalhados os principais elementos da abordagem definida pelo grupo.

4.1 Solução Inicial

Essa seção apresenta ideias sobre a geração da solução inicial utilizada, bem como a implementação do algoritmo empregado e algumas imagens que buscam exemplificá-lo de uma maneira mais visual.

4.1.1 Iteração do algoritmo guloso

Considere como exemplo o grafo da Figura 1. Por motivos de simplicidade, todas as arestas desse grafo possuem tipos distintos. O processo para o caso em que arestas possuem tipos iguais é um caso que pode ser verificado juntamente à etapa de checagem de vértices já utilizados.

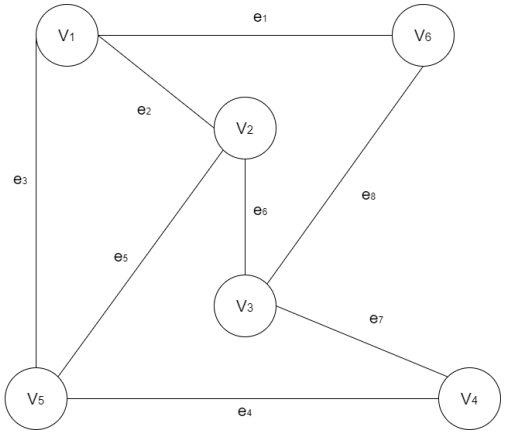

Figura 1: Grafo Inicial

O primeiro passo para o algoritmo guloso é ordenar o conjunto de arestas recebidos, em ordem crescente, com base no grau médio dos vértices em que a aresta incide. Um exemplo dessa ideia pode ser vista ao selecionar a aresta e_1, a qual incide nos vértices V_1 e V_6. Isso significa que o valor (grau) dessa aresta é a média dos valores $deggre(V_1) = 3$ e $deggre(V_6) = 2$, ou seja 2.5. Deve-se salientar que a função $deggre$ retorna o grau de um determinado vértice $v \in V$.

Para ordenar esse conjunto supracitado, fica-se evidente a necessidade de calcular os valores de todas arestas. O resultado de tais cálculos para o grafo da Figura 1 está demonstrado na Tabela 1.

Aresta	Grau Médio de Vértices	Total
e_1	$avg\_degree = \frac{deggre(V_1)+deggre(V_6)}{2}$	2.5
e_2	$avg\_degree = \frac{deggre(V_1)+deggre(V_2)}{2}$	3
e_3	$avg\_degree = \frac{deggre(V_1)+deggre(V_5)}{2}$	3
e_4	$avg\_degree = \frac{deggre(V_4)+deggre(V_5)}{2}$	2.5
e_5	$avg\_degree = \frac{deggre(V_2)+deggre(V_5)}{2}$	3
e_6	$avg\_degree = \frac{deggre(V_2)+deggre(V_3)}{2}$	3

Tabela 1: Grau médio de cada aresta do grafo da Figura 1

O conjunto ordenado com base nos valores obtidos é $\{e_1, e_3, e_2, e_4, e_5, e_6\}$. Nesse caso, percebe-se que os valores foram desempatados por meio de seu *index*, de forma crescente, porém, no algoritmo implementado, esse ordenamento é feito pela função *sorted*, a qual já vem implementada dentro da linguagem utilizada. Em outras palavras, a responsabilidade de ordenar arestas com valores empatados é delegada para a própria linguagem.

Para o grafo em questão, o conjunto de solução inicial seria inicializado com o primeiro elemento das arestas ordenadas, ou seja $\{e_1\}$. Após isso, itera-se de forma linear sobre todo o conjunto, verificando a possibilidade de inserção da aresta atual no emparelhamento.

É nesse momento que checa-se se os vértices da aresta atual já fazem parte de alguma aresta do conjunto solução ou se o tipo da aresta já foi utilizado. Se a resposta para tal asserção é verdadeira, então continua-se o laço, caso contrário, uma nova aresta é inserida na solução. A terminação do algoritmo é dado quando não há mais arestas que podem ser inseridas.

Isso significa que, com base nesse algoritmo, o emparelhamento formado como solução inicial é $\{e_1, e_4, e_6\}$. Uma breve descrição é dada da seguinte forma: A aresta e_2 não pode ser utilizada, por causa de V_1 já fazer

parte de e_1, o mesmo vale para e_3. Enquanto que e_5 não entra para o conjunto solução, uma vez que e_4 deixa o vértice V_5 indisponível.

Pode-se afirmar que foi utilizado uma abordagem simplista para a construção da solução inicial, porém algumas ideias que podem ser implementadas para estender este trabalho é diminuir o grau dos vértices com base na simetria ou descontar, do grau de um dado vértice, a quantidade de arestas ligadas a ele que já foram inseridas na solução.

4.1.2 Algoritmo guloso

O algoritmo guloso construído é da seguinte forma: Seja $edges$ um conjunto de arestas que foram carregadas na memória e $deggre\_counter$ um dicionário que contém o grau de todos os vértices pertencentes ao grafo G, obtém-se, como retorno, um conjunto de arestas ($edges'$) que formam um emparelhamento no grafo. O pseudo-código é apresentado no Algortimo 1

Algorithm 1 Algoritmo Guloso para construção da Solução Inicial

```
1:  function GREEDY_INITIAL_SOLUTION(edges, deggre_counter)
2:      for e in edges do
3:          e.deggre ← (deggre_counter[e.vertex_u] + deggre_counter[e.vertex_v])/2
4:      end for
5:      sorted_edges ← sort_by_deggre(edges)
6:      edges' ← {}
7:      if |sorted_edges| > 0 then
8:          edges' ← edges' + {sorted_edges[0]}
9:          for edge in sorted_edges do
10:             for e in edges' do
11:                 can_increase ← share_attributes(edge, e)
12:                 if ¬can_increase then
13:                     edges' ← edges' + {edge}
14:                 end if
15:             end for
16:         end for
17:     end if
18:     return edges'
19: end function
```

4.2 Vizinhança

Seja S um conjunto de arestas que configura uma solução inicial para o problema, podemos definir a vizinhança da seguinte forma: sorteia-se uma aresta $a \in A$ e, com isso, temos dois casos:

- Caso $a \in S$, então retiramos a de S e retorna o conjunto $S' = S - \{a\}$ como vizinho.
- Caso $a \notin S$, então é preciso testar se a satisfaz as restrições do emparelhamento diversificado para podermos adicionar a em S:
  - Caso a adição da aresta a em S satisfaça as restrições do emparelhamento diversificado, então retorna $S' = S + \{a\}$ como vizinho.
  - Caso a adição da aresta a em S não satisfaça as restrições, então é feito um novo sorteio e são feitos os mesmos testes novamente até que a possa ser adicionada ou removida do conjunto S.

Vale ressaltar que esse método pode entrar em *loop* caso não haja arestas possíveis a serem adicionadas, então estabelece-se um critério de parada de I iterações para terminar o método, retornando assim o próprio conjunto S. Por simplicidade, o valor de I é definido como sendo o tamanho do conjunto de arestas do grafo, ou seja, $I = |A|$.

4.2.1 Algoritmo da Vizinhança

O Algoritmo 2 apresenta a lógica implementada para encontrar um vizinho de uma solução, sendo que ele recebe como entrada a lista A de arestas do grafo e a solução atual S, retornando sua solução vizinha S'.

A função *share_attributes(a, a')* returna $true$ caso as arestas a e a' compartilham um vértice ou possuem o mesmo tipo e $false$ caso contrário.

Algorithm 2 Algoritmo de Busca para Vizinhança

```
1: function GET_NEIGHBOR(S, A)
2:     S' ← S                                   ▷ Caso a solução não aumente ou diminua
3:     for i = 0; i < |A|; i ← i + 1 do
4:         a ← random_edge(A)
5:         if a ∈ S then
6:             S' ← S − {a}
7:             break
8:         else
9:             can_increase ← true
10:            for a' in S do
11:                can_increase ← ¬share_attributes(a', a)
12:                if ¬can_increase then
13:                    break
14:                end if
15:            end for
16:            if can_increase then
17:                S' ← S + {a}
18:                break
19:            end if
20:        end if
21:    end for
22:    return S'
23: end function
```

4.3 Critério de Parada para o SA

Considerando que a heurística é a de *Simulated Annealing*, o critério de parada é uma temperatura mínima, quando a solução já está estabilizada em um máximo local. Mais detalhes sobre o valor da temperatura mínima estipulada podem ser vistos na Seção 7.2.

5 Parâmetros

É necessário definir alguns parâmetros utilizados no método e quais as suas funções, sendo esses parâmetros apresentados a seguir:

- **metropolis_it**: quantidade de iterações do Metropolis, quando a temperatura é mantida constante.
- **init_temp**: temperatura inicial.
- **end_temp**: temperatura final.
- **discount**: fator de desconto de temperatura a cada execução do Metropolis.

6 Implementação

6.1 Plataforma de implementação

A linguagem de programação utilizada foi *Python* em sua *versão 3.8.2*. Além disso, fez-se necessário a instalação da biblioteca pandas, a qual realiza a coleta dos dados das simulações para serem salvos no formato *csv*. Como plataforma para rodar os experimentos, dada a necessidade de executar muitas repetições que levariam uma grande quantidade de tempo, optou-se por usar o serviço *Google Cloud*, no qual configurou-se uma máquina virtual (VM) em um servidor em São Paulo. Essa VM possui as seguintes configurações: Processador Intel(R) Xeon(R) CPU @ 3.10GHz, Cascade Lake (com 8 vCPU para rodar múltiplos experimentos ao mesmo tempo), memória RAM de 32 Gb e sistema operacional Debian GNU/Linux 10.

6.2 Estruturas de dados utilizada

Para representar os dados de entrada presentes no arquivo, utilizou-se uma lista (equivalente a um *array*, para a linguagem utilizada) que aloca as arestas do grafo. Para cada aresta, utilizou-se uma classe **Edge**, composta

de dois vértices (u e v), um tipo (t_a) e um grau. O grau contém o valor do grau médio dos vértices incididos pela aresta. Para representar as soluções, utilizou-se a estrutura de dados Set fornecida pela linguagem Python.

7 Resultados

Nesta seção estão presentes os resultados obtidos ao aplicar a implementação proposta às instâncias do problema disponibilizadas pelo professor.

Instância	GLPK	SA	BKV
RM01	2165	2467	2436
RM02	2202	2463	2432
RM03	2183	2470	2436
RM04	2155	2464	2439
RM05	2188	2465	2436
RM06	2153	2463	2431
RM07	2157	2465	2437
RM08	2156	2466	2433
RM09	2197	2466	2436
RM10	2184	2464	2429
RM11	2134	2465	2434
RM12	2188	2468	2440

Tabela 2: Valores das soluções encontradas com o otimizador (GLPK), com o Simulated Annealing (SA), assim como os melhores valores conhecidos disponibilizados pelo professor (BKV).

7.1 Otimizador

A formulação inteira apresentada na Seção 3 foi implementada no otimizador GLPK, por meio da linguagem de programação Julia. Estipulou-se um limite de 30 minutos[1] para executar cada instância do problema no otimizador, visto que o tempo para resolver o programa inteiro pode ser consideravelmente grande. Os resultados de cada instância no otimizador estão apresentados na coluna GLPK da Tabela 2.

7.2 Simulated Annealing

Os valores para cada parâmetro utilizado na heurística de *Simulated Annealing* são os seguintes:

- **metropolis_it**: 500. Esse valor é o suficiente para ter uma garantia de que o método possa encontrar arestas possíveis de serem inseridas quando a solução já está próxima de um máximo local.
- **init_temp**: 2. Como a solução inicial já é consideravelmente boa devido ao algoritmo guloso, colocar uma temperatura muito alta poderia fazer com que o método aceitasse muitas soluções piores, inutilizando a necessidade do guloso. Além disso, considerar uma temperatura menor que ainda permita soluções piores acabou fazendo com que a busca pudesse melhorar a partir de soluções que já são melhores que a trivial[2].
- **end_temp**: 0.01. Esse valor garante que seja praticamente impossível (3.72×10^{-42} %) que uma solução pior que a atual seja considerada pelo Metropolis.
- **discount**: 0.99, o que gera em torno de 500 repetições do Metropolis. Dado que já partiu-se de uma solução consideravelmente boa e também pelas limitações de tempo (com esse fator de desconto, leva em torno de uma hora para rodar cada conjunto de experimentos), esse valor foi suficiente.

Dada a característica estocástica do método de *Simulated Annealing*, é necessário de rodá-lo mais de uma vez com o objetivo de buscar soluções melhores para o problema. Dessa maneira, foram feitas 10 execuções de cada instância com os parâmetros setados de acordo com o que foi discutido na Seção 7.2. A partir dessas execuções, extraiu-se o melhor resultado de cada instância para ser considerado como o valor obtido pelo método. Os resultados obtidos foram descritos na Tabela 2, a qual apresenta os valores obtidos para

[1] testes com até 2h acabaram gerando o mesmo resultado, eliminando a necessidade de aumentar o tempo
[2] Solução que não possui nenhuma aresta

cada instância com o otimizador (GLPK)[3], com o Simulated Annealing (SA) e também os melhores valores conhecidos, fornecidos pelo professor (BKV).

7.3 Comparação dos Resultados

Uma comparação mais visual entre os métodos pode ser observada nas Figuras 2, 3 e 4. É possível notar que o método que obteve melhores soluções foi o *Simulated Annealing*, tendo resultados melhores até mesmo em relação ao BKV. Já o otimizador obteve soluções consideravelmente piores, algo que acontece sistematicamente para todas as instâncias do problema.

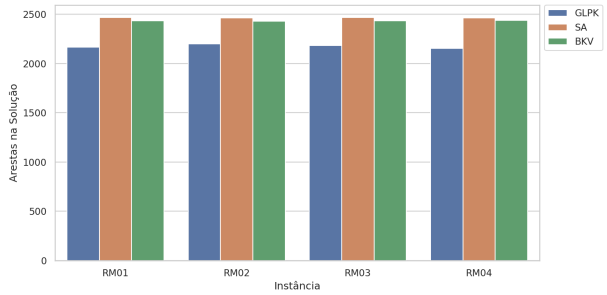

Figura 2: Comparação das instâncias RM01 a RM04

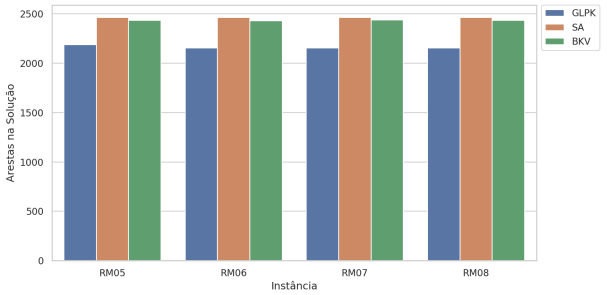

Figura 3: Comparação das instâncias RM05 a RM08

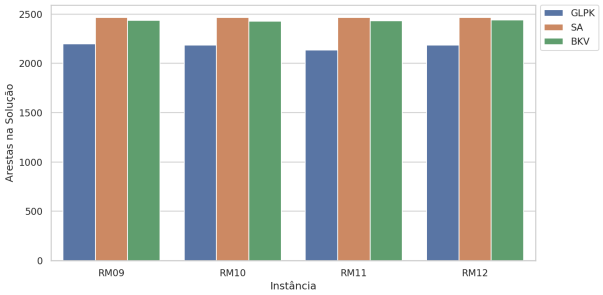

Figura 4: Comparação das instâncias RM09 a RM12

8 Conclusões

Pode-se considerar que a meta-heurística proposta teve uma boa performance, dado que os resultados foram melhores que os valores conhecidos anteriormente. Em compensação, a performance do otimizador pareceu um tanto abaixo do esperado, visto que com basicamente o mesmo tempo[4] a meta-heurística proposta obteve resultados muito melhores. Concluí-se então que foi deveras vantajoso utilizar a meta-heurística em detrimento do otimizador em se tratando de instâncias consideravelmente grandes do problema.

A implementação da heurística foi relativamente simples, o que proporcionou mais tempo para análise e ajuste de parâmetros do método a fim de se obter melhores resultados. Dada a linguagem de programação usada, acredita-se que a utilização de uma linguagem de programação compilada para o método poderia trazer benefícios em questão de tempo para rodar os experimentos, o que possibilitaria a obtenção de resultados ainda melhores e talvez até a otimalidade em algumas instâncias, visto que proporcionaria mais execuções do Metropolis com um fator de desconto que reduza a temperatura mais lentamente.

Como um todo, acredita-se que o trabalho apresentado obteve bons resultados, trazendo um bom entendimento da meta-heuristíca de *Simulated Annealing* e, com isso, foi bem sucedido.

[3]Com critério de parada por tempo, sendo 30 minutos o limite para cada execução
[4]Dado que aumentar o tempo do *solver* não proporcionou valores melhores