INF05010 Otimização Combinatória
Trabalho Final:
Algoritmo Genético para o Problema da Seleção da Maior Distância Mínima Total

Lucas Nunes Alegre

Junho de 2018

1 Introdução

Este trabalho tem como objetivo a implementação de uma meta-heurística para a resolução do problema da Seleção da Maior Distância Mínima Total(MDMT). A meta-heurística implementada foi o algoritmo genético, cujos detalhes serão apresentados na seção 3.

O problema da maior distância mínima total é definido como: dados um grafo bi-partido completo $G = (M \dot{\cup} L, A)$ com distâncias d_a nas arestas $a \in A$ e uma constante $l \leqslant |L|$, o objetivo é encontrar uma seleção de vértices $S \subseteq L$ com exatamente l elementos tal que a soma das distâncias mínimas $\sum_{m \in M} d(m)$ onde $d(m) = min_{s \in S} d_{ms}$ é maximizado.

O problema é conhecidamente NP-Difícil.

2 Formulação do MDMT como Programa Inteiro

Variáveis: $x_i \in \{0, 1\} \quad \forall\ i \in L$ onde:

$$x_i = \begin{cases} 1 & \text{Caso o vértice } i \in S \\ 0 & \text{Caso contrário.} \end{cases}$$

$s_m \in \mathbb{Z}_+ \quad \forall m \in M$: Distância mínima do vertice m até algum vértice $\in S$

Função Objetivo:

$$\max \sum_{m \in M} s_m$$

Restrições:

$$\sum_{i \in L} x_i = l \tag{1}$$

$$s_m \leqslant d_{mi} + C(1 - x_i) \qquad \forall m \in M, \quad \forall i \in L \tag{2}$$

$$x_i \in \{0, 1\} \qquad \forall i \in L \tag{3}$$

$$s_m \in \mathbb{Z}_+ \qquad \forall m \in M \tag{4}$$

A restrição (1) garante que serão escolhidos exatamente l vértices.

A restrição (2) garante que s_m seja a menor distância entre o vértice m e algum vértice escolhido em L. Nesta restrição, caso um vértice $i \in L$ não foi selecionado($x_i = 0$), somamos uma constante grande $C \geq max_{m \in M, i \in L} d_{mi}$ no lado direito.

As restrições (3) e (4) definem o domínio das variáveis.

3 O Algoritmo Genético

Para a resolução do problema da Seleção da Maior Distância Mínima Total foram definidos os seguintes elementos para o algoritmo genético:

Cromossomo: Um vetor binário de genes $c = (c_1, c_2, ..., c_{|L|})$ de $1..|L|$.

Gene: Cada c_i com valor 0 ou 1, indicando a presença ou não do vértice i no conjunto S.

População: Um conjunto de cromossomos.

3.1 Parâmetros

Os seguintes parâmetros foram levados em consideração na implementação do algoritmo genético:

population_size : O número de cromossomos na população, número inteiro maior que 1.

elitism_rate : O percentual de cromossomos com maior valor de fitness da geração atual que serão automaticamente passados para a geração seguinte sem sofrerem crossover ou mutação, número real entre 0 e 1.

mutation_rate : Taxa de mutação, número real entre 0 e 1.

max_non_improving_generations : Critério de terminação, número inteiro maior do que 1, o algoritmo irá terminar a execução se forem passadas max_no_improving_generations que não aumentaram o fitness da melhor solução atual.

time_limit : Tempo máximo de execução do algoritmo em segundos.

random_seed : Semente randômica inicial, utilizada para gerar nímeros aleatórios durante a execução. É garantido que, para uma mesma semente e mesmos parâmetros, a solução seja igual em diferentes execuções.

3.2 Pseudocódigo

Algorithm 1 Algoritmo Genético

```
1:  população ← PopulaçãoInicialAleatória()
2:  while not CritériosDeParada() do
3:      nova_população ← ∅
4:      while tamanho(nova_população) < (1 - elitism_rate) · population_size do
5:          pai1, pai2 ← EscolheDoisCromossomos()
6:          filho1, filho2 ← Crossover(pai1, pai2)
7:          nova_população ← nova_população ∪ filho1 ∪ filho2
8:      for cromossomo in nova_população do
9:          cromossomo ← TornaFactível(cromossomo)
10:         if random[0, 1) < mutation_rate then
11:             cromossomo ← Mutação(cromossomo)
12:         if Fitness(cromossomo) > Fitness(melhor_solução) then
13:             melhor_solução ← cromossomo
14:     k ← population_size − tamanho(nova_população)
15:     população ← nova_população ∪ K-Melhores(população, k)
```

3.3 Geração da População Inicial

São gerados *population_size* vetores binários aleatórios com exatamente l bits em 1, garantindo a factibilidade de cada cromossomo, e calcula-se o fitness de cada solução.

3.4 Método de Seleção para o Crossover

A escolha de indivíduos para crossover é feita através do Método do Torneio[1]: seleciona-se k indivíduos aleatórios da população e retorna-se o indivíduo de maior fitness entre os k selecionados. O método é executado duas vezes para selecionar dois indivíduos que serão recombinados no crossover. O valor de k escolhido foi 4, pois apresentou bons resultados em avaliações empíricas.

3.5 Crossover

Utilizou-se o método de Uniform Crossover, como descrito em [2]. Dado dois pais p1 e p2 selecionados pelo método de seleção, gera-se dois filhos, f1 e f2, inicialmente iguais aos respectivos pais. Cria-se então, uma máscara binária aleatória de tamanho dos cromossomos, e para cada bit i da máscara, se o bit for 1 realiza-se o swap do i-ésimo bit do f1 com o de f2, se o bit for 0, continua. Desse modo, para cada bit, há 50% de chance de cada filho "herdar"o bit de p1 e 50% de p2.

É importante observar que esse método não mantém a factibilidade das soluções uma vez que os filhos gerados podem ter um número diferente de l bits em 1. Portanto, após o crossover, se o filho haver mais de l bits, realiza-se bit flips de 1 para 0 até que o filho tenha l bits em 1. O mesmo raciocínio aplica-se da maneira inversa para o caso do filho haver selecionado menos de l vértices.

3.6 Mutação

Utilizou-se a técnica de Swap-Mutation[3]. Duas posições aleatórias do vetor binário são selecionadas e têm seus valores trocados. Essa operação de mutação mantém a factibilidade das soluções, pois o vetor resultante continua com a mesma quantidade de bits em 1 e em 0 após a mutação.

3.7 Critério de Parada

O algoritmo para sua execução, retornando a melhor solução encontrada até o momento, caso se ultapasse *time_limit* segundos desde seu início, ou caso se atinja *max_non_improving_generations* gerações sem haver aumento no fitness da melhor solução.

4 Implementação

Todo código criado e utilizado na realização desse trabalho se encontra disponível em `https://github.com/LucasAlegre/GeneticAlgorithm-MDMT`.

4.1 Plataforma

O trabalho foi implementado e testado em uma máquina virtual com sistema operacional Linux Ubuntu(64-bit) e sistema hospedeiro Windows 10, com um processador Intel(R) Core(TM) i5-7500 CPU, com acesso a 2 núcleos físicos de 3.40GHz, com cache L2 de 1MB e 5GB de memória.

A linguagem de programação utilizada foi Python 3.6.3 com uso da biblioteca NumPy 1.14.5, que executa código pré-compilado da linguagem C.

4.2 Estruturas de Dados

O grafo da instância é representado por uma matriz de inteiros de tamanho $|M| \times |L|$, a melhor representação dado que trata-se de um grafo bi-partido completo, pois cada célula da matriz representa o peso de uma aresta de um vértice em M para um vértice em L.

Cada cromossomo é representado como um array de booleanos de tamanho $|L|$ com l posições com valor True e as restantes com valor False, sendo True e False igualmente operados como 1 e 0 na linguagem Python. Cada cromossomo também tem como atributo seu tamanho e seu valor de Fitness para evitar recalcular esses valores a cada acesso. Uma população é representada como uma lista de cromossomos.

4.3 Função de Fitness

O fitness de um cromossomo é o valor da função objetivo do problema aplicada aos genes do cromossomo, que representam uma possível solução. O cálculo é feito percorrendo as $|M|$ linhas da matriz do grafo da instância e calculando o somatório do valor mínimo de cada linha levando em consideração apenas as colunas selecionadas na solução. Portanto, o cálculo de fitness de um cromossomo tem complexidade $O(|M| \cdot l)$.

5 Testes dos Parâmetros

Nos seguintes testes, variou-se os parâmetros do algoritmo genético separadamente para determinar os melhores valores. Cada teste foi executado 5 a 10 vezes com sementes aleatórias distintas na instância mdmt49.112.A, que foi escolhida por ser uma instância que se demonstrou difícil de obter bons resultados e que possui valor de l intermediário entre as outras instâncias. Foi medido para cada teste o desvio padrão médio em relação ao best-known value(BKV) e o tempo de execução médio entre as execuções. Os parâmetros foram variados a partir da seguinte configuração inicial: *population_size* = 40, *elitism_rate* = 0.1, *mutation_rate* = 0.3 e *max_non_improving_generations* = 400.

5.1 Teste do Parâmetro *population_size*

O valor de *population_size* foi variado entre 20 e 80 com incrementos de 10. A Figura 1 mostra os resultados.

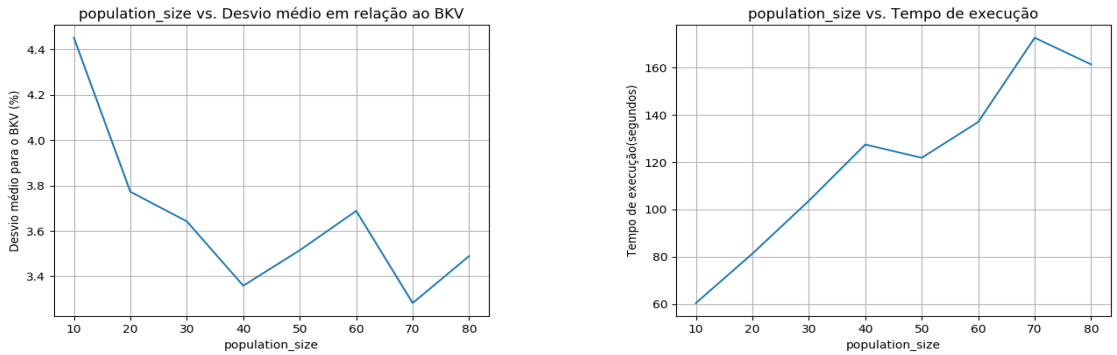

Figura 1: Variação do parâmetro *population_size*

Este parâmetro demonstrou ser um dos mais importantes do algoritmo. Com base nos resultados, população de tamanho 40 mostrou ter o melhor custo benefício entre desvio em relação ao BKV e tempo de execução. Valores menores que 40 mantêm pouca variabilidade na população e tendem a atingir mínimos locais. Entretanto, valores maiores que 40 aumentam consideravelmente o tempo de execução do algoritmo sem resultar em soluções melhores.

5.2 Teste do Parâmetro *elitism_rate*

O valor do parâmetro *elitism_rate* foi variado de 0.0 a 0.5 com incrementos de 0.1. A Figura 2 mostra os resultados.

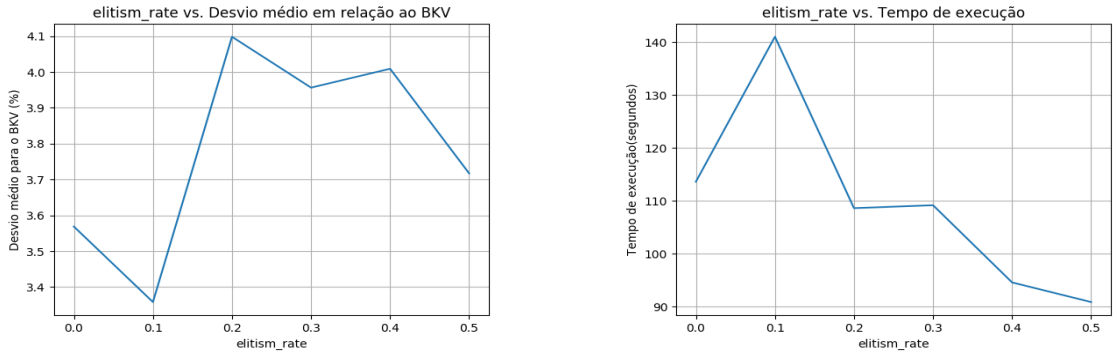

Figura 2: Variação do parâmetro *elitism_size*

Analisando a Figura 2, fica claro que taxas de elitismo altas não demonstraram ser boas opções. Isso deve-se ao fato de que muitas soluções da geração passadas são mantidas e menos soluções novas são geradas, causando a estagnação em mínimos locais mais facilmente. Taxa de elitismo com valor 0 também resulta em soluções piores que as com o melhor valor obtido(0.1), pois pode-se perder as soluções com valores altos de fitness de uma geração para a próxima.

5.3 Teste do Parâmetro *mutation_rate*

O valor do parâmetro *mutation_rate* foi variado de 0.0 a 1.0 com incrementos de 0.1. A Figura 3 mostra os resultados.

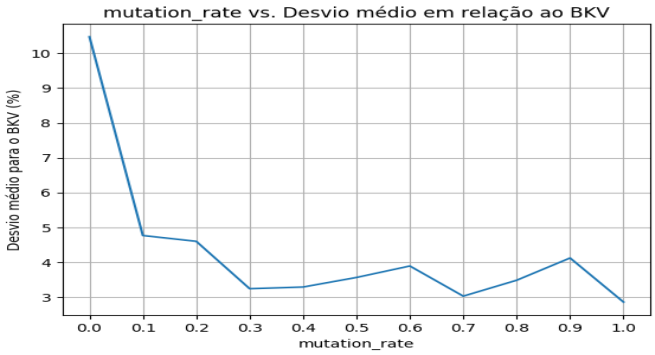

Figura 3: Variação do parâmetro *mutation_size*

Com base nos resultados, percebe-se que taxas menores que 0.3 obtêm resultados ruins, uma vez que a variabilidade introduzida na população é muito baixa e facilmente fica presa em mínimos locais. Valores maiores que 0.3 obtiveram resultados semelhantes entre si. Todavia, é preferível evitar valores muito altos, pois pode-se perder boas soluções resultantes do crossover de duas soluções com valor alto de fitness caso as soluções filhas sempre sofram mutação.

5.4 Teste do Parâmetro *max_non_improving_generations*

O parâmetro *max_non_improving_generations* foi variado de 100 a 500 com incrementos de 100. A Figura 4 mostra os resultados.

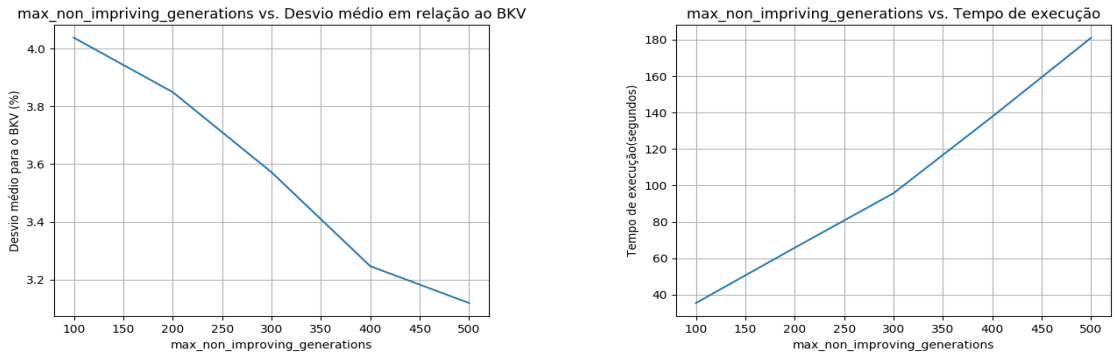

Figura 4: Variação do parâmetro *max_non_improving_generations*

Fica claro que o tempo de execução cresce de forma quase linear conforme aumentamos o parâmetro, pois com valores maiores o algoritmo executa mais gerações na tentativa de obter uma melhor solução. Entretanto, do valor 400 para o 500 a inclinação da curva é menor, assim considerou-se o valor 400 como um bom valor para o parâmetro.

6 Testes das Instâncias

As instâncias testadas nesse trabalho foram as disponíveis em `http://www.inf.ufrgs.br/~mrpritt/oc/mdmt.zip`. Todas possuem cardinalidade da partição M e L do grafo igual a 450, tendo como diferença os pesos das arestas e a constante l. Para cada instância, comparou-se o BKV com os resultados obtidos via solver e via algoritmo genético com a melhor configuração obtida nos testes dos parâmetros.

6.1 Execução com o Solver CPLEX

Os seguintes testes das instâncias, após formuladas como programas inteiros, foram executadas no solver CPLEX com um tempo limite de 30 minutos. A versão utilizada do solver foi a 12.8 para Linux 64 bits. O objetivo é comparar os resultados obtidos pelo algoritmo genético com os obtidos via solver. A Tabela 1 mostra os resultados.

Tabela 1: Resultados com o solver CPLEX

Instância	BKV	Valor Obtido	Desvio para Referência (%)
mdmt39.112.A	5935	5157	13.11
mdmt39.112.B	6198	5588	9.84
mdmt39.225.A	4354	3272	24.85
mdmt39.225.B	4260	3873	9.08
mdmt40.56.A	8211	6244	23.96
mdmt40.56.B	8022	7506	6.43
mdmt40.112.A	6271	4988	20.46
mdmt40.112.B	6198	5759	7.08
mdmt40.225.A	4550	4303	5.43
mdmt40.225.B	4492	4286	4.59

É importante destacar que todas as instâncias se executadas no solver por mais de 30 minutos acabavam por consumir toda a memória RAM disponível - encerrando o processo, não sendo possível achar a solução ótima sem a utilização de memória auxiliar em disco.

6.2 Teste com o Algoritmo Genético

Os seguintes testes foram executados com os melhores valores encontrados nos testes dos parâmetros: *population_size* = 40, *elitism_rate* = 0.1, *mutation_rate* = 0.3 e *max_non_improving_generations* = 400. Cada instância foi executada 5 vezes com sementes aleatórias distintas. Na Tabela 2 são apresentados os resultados médios das 5 execuções assim como a melhor solução encontrada em cada instância.

Tabela 2: Resultados com o algoritmo genético

Instância	BKV	Valor Médio Obtido	Desvio Médio para Referência (%)	Melhor Valor Obtido	Desvio do Melhor Valor para Referência (%)	Tempo de Execução Médio (segundos)
mdmt39.112.A	5935	6024.2	-1.50	6090	-2.61	105.40
mdmt39.112.B	6198	5758.8	7.09	5806	6.32	89.23
mdmt39.225.A	4354	4134.2	5.05	4208	3.35	141.67
mdmt39.225.B	4260	4219.4	0.95	4309	-1.15	132.81
mdmt40.56.A	8211	7845.4	4.45	7963	3.02	76.66
mdmt40.56.B	8022	7997.2	0.31	8100	-0.97	300.15
mdmt40.112.A	6271	6067.4	3.25	6097	2.77	122.60
mdmt40.112.B	6198	6047.8	2.42	6137	0.98	109.29
mdmt40.225.A	4550	4415.8	2.95	4452	2.15	145.23
mdmt40.225.B	4492	4469.8	0.49	4488	0.09	143.80

6.3 Comparação dos testes das instâncias

A Figura 5 compara os resultados obtidos via CPLEX e algoritmo genético(AG) relatados nas seções 6.1 e 6.2.

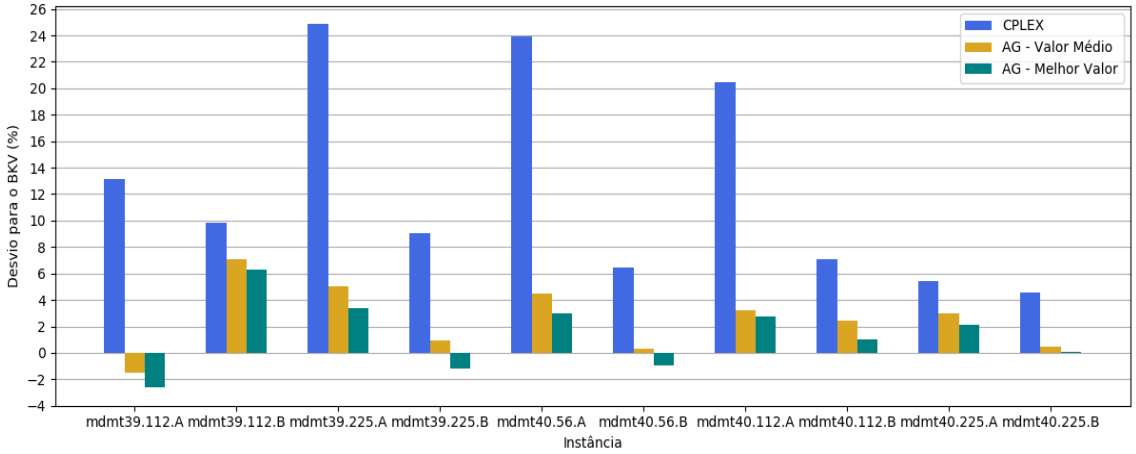

Figura 5: Comparação dos testes das instâncias

7 Conclusões

Através dos resultados obtidos, pode-se concluir que para as instâncias do MDMT testadas, a resolução via algoritmo genético é uma opção consideravelmente mais proveitosa que a via solver. Executando o algoritmo genético por cerca de 2 minutos obteve-se valores de soluções até 20% maiores que as encontradas no CPLEX com tempo limite de 30 minutos. Ainda, nos testes da seção 6.2 ultrapassou-se o BKV em 3 das 10 instâncias.

Por meio dos testes da seção 5, notou-se também que o algoritmo genético é bastante sensível aos valores de parâmetros escolhidos, fazendo-se necessário o teste de diversas combinações de valores para encontrar a configuração com melhores soluções.

Como possível melhoria, poderia se testar métodos de mutação que causam maiores perturbações nas soluções que a técnica de Swap-Mutation. Assim, evitaria-se mínimos locais mais facilmente.

Acredita-se que poderia obter-se melhores soluções e ultrapassar o BKV em mais instâncias se o algoritmo fosse executado por mais tempo e com um valor maior do parâmetro *max_non_improving_generations*.

Em vista dos pontos citados, considera-se que foi possível implementar uma boa configuração da meta-heurística escolhida, obtendo-se valores satisfatórios para a grande maioria das instâncias.

Referências

[1] Thomas Back, D.B Fogel, and Z Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. 2000.

[2] Williams E.A. and Crossley W.A. Empirically-derived population size and mutation rate guidelines for a genetic algorithm with uniform crossover. *Chawdhry P.K., Roy R., Pant R.K. (eds) Soft Computing in Engineering Design and Manufacturing. Springer, London*, 1998.

[3] Nitasha Soni and Dr Tapas Kumar. Study of various mutation operators in genetic algorithms. *(IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5*, 2014.