Introdução à Inteligência Artificial - Projeto 2

RAISSA CAVALCANTE CORREIA
RAYSA MASSON BENATTI

I. INTRODUÇÃO

O presente trabalho apresenta a descrição das conclusões e estratégias adotadas para a resolução do Projeto 2 da disciplina Introdução à Inteligência Artificial (MC906), ministrada no primeiro semestre de 2019 na Unicamp pela professora Esther Luna Colombini.

Conforme a especificação, o grupo deveria escolher um problema da literatura e apresentar uma solução para ele com base em técnicas de computação evolutiva, avaliando essa solução de acordo com parâmetros distintos. O grupo optou por estudar e tentar solucionar o **problema do caixeiro viajante**.

Na seção II do relatório, trazemos a definição do problema. A seção III dedica-se à descrição do modelo evolutivo escolhido para buscar soluções e especificidades da implementação, bem como da função de *fitness* e variações sobre os parâmetros adotadas. A seção IV descreve os resultados observados. Finalmente, na seção V, discutimos tais resultados e apresentamos conclusões. As referências encontram-se ao final do trabalho.

II. O PROBLEMA

O problema do caixeiro viajante (PCV) é apresentado na literatura de teoria da computação como um exemplo canônico de problema NP-completo, isto é, ainda não foram encontrados algoritmos eficientes para solucioná-lo de maneira inequivocadamente ótima, tampouco provado que tal algoritmo não possa existir [1]. Contudo, há abordagens para atingir soluções suficientemente boas com eficiência, sendo a estratégia da computação evolutiva uma dessas abordagens.

O PCV consiste em encontrar a menor distância possível a ser percorrida em um grafo, em que se passa por cada vértice exatamente uma vez, exceto pelo primeiro vértice, que deve ser necessariamente o mesmo que o último. A distância é computada somando-se o custo de cada aresta no caminho. Em geral, isso se traduz para uma situação da vida real considerando que cada vértice do grafo é uma cidade, e para cada par de cidades, há uma distância conhecida entre elas (ou algum valor que represente o custo de ir de uma à outra) — daí o nome do problema, dado que a solução traria otimização de custo para um suposto vendedor que deve passar por determinadas localidades.

Há, ainda, outras situações concretas que podem ser modeladas com o PCV. Em [1], por exemplo, os autores imaginam uma empresa de transporte por caminhão com um armazém central, em que todos os dias o caminhão é carregado, sai do armazém, passa pelos locais onde deve efetuar entregas e retorna ao armazém no fim do dia. Para redução de custos, seria interessante otimizar o caminho feito pelo caminhão, selecionando uma ordem de parada nas localidades.

III. SOLUÇÃO COM COMPUTAÇÃO EVOLUTIVA

Para buscar uma solução para o PCV, foi implementado um algoritmo genético baseado na sugestão publicada por Eric Stoltz em [2], sobre a qual o grupo adicionou alterações e funcionalidades, que serão descritas oportunamente. O algoritmo indicado foi escrito em Python, razão pela qual o grupo adotou a linguagem em sua solução. Destacam-se, também, como justificativas para a escolha da linguagem: suporte a boas bibliotecas de visualização e tratamento de dados, como Matplotlib e Pandas; suporte a orientação a objetos; simplicidade de sintaxe.

Nessa abordagem, cada cidade no mapa (vértice do grafo) é entendida como um **gene** e representada por suas coordenadas (x, y), definidas em uma classe `City`. A classe também tem um método para calcular a distância euclidiana entre duas cidades — dada por $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ —, o que representa o custo de uma aresta entre elas.

Um **indivíduo**, ou **cromossomo**, é uma rota que satisfaça às restrições especificadas na seção II, ou seja, que estabeleça um ciclo hamiltoniano entre as cidades. Uma **população** é uma coleção de indivíduos; nesse caso, uma coleção de rotas que satisfaçam às restrições do problema. **Pais** são duas rotas que são combinadas para formar uma nova rota através de alguma técnica de ***crossover***, ou **reprodução sexuada**. Um ***pool* de acasalamento** é o conjunto de pais escolhido através de mecanismos de **seleção** para criar a próxima população, isto é, a próxima **geração** de rotas possíveis.

Além da reprodução sexuada, o algoritmo usa alguma técnica de **mutação** para introduzir variabilidade na população. Isso é importante para evitar, por exemplo, uma convergência prematura para uma solução menos adequada que aquela que o algoritmo tem potencial de oferecer. O algoritmo também faz uso do **elitismo**: um método para

garantir que os melhores indivíduos de uma geração sejam carregados para a próxima.

A. Algoritmo-base

O algoritmo executa os seguintes passos:

- Criar população;
- Determinar *fitness*;
- Selecionar o *pool* de acasalamento;
- Reproduzir;
- Realizar mutações;
- Repetir até que seja atingido o critério de parada.

O critério de parada foi estabelecido como um número fixo de gerações. A população inicial é criada selecionando-se aleatoriamente um número determinado de rotas entre cidades de uma lista. A lista de cidades é preenchida com auxílio da função `random.random()`, criando-se uma determinada quantidade de cidades com valores aleatórios de $(x, y) \in [0, 200)$.

A função de *fitness*, por sua vez, é determinada dentro de uma classe `Fitness`. A classe recebe como parâmetro uma rota `route` e tem dois métodos: `routeDistance()` calcula a distância total da rota, e `routeFitness()` calcula a *fitness*, definida como o inverso da distância. Assim, quanto menor a rota, maior o valor da *fitness* e, portanto, melhor esse caminho é de acordo com o objetivo da solução: otimizar o custo total. A função auxiliar `rankRoutes(population)` recebe uma lista de rotas e as ordena decrescentemente de acordo com o valor da *fitness* de cada uma.

Para a seleção do *pool* de acasalamento, uma parte da população — cuja quantidade é definida pela variável `eliteSize` — é necessariamente inserida numa lista `selectionResults`. O restante é selecionado através de uma implementação do algoritmo da roleta, em que a cada rota da lista é associado um peso que depende de sua *fitness*.

Cada uma das rotas, já ranqueadas decrescentemente, é inserida como uma linha em um *dataframe* de colunas *Index* e *Fitness*, com auxílio da biblioteca Pandas. Em seguida, calcula-se, para cada linha, a soma cumulativa cum_sum dos valores de *fitness*. Finalmente, esse valor é ajustado conforme a expressão: $cum\_perc = 100 \cdot \frac{cum\_sum}{f\_sum}$, em que f_sum é a soma de todos os valores de *fitness*. Ou seja: quanto maior o valor da *fitness*, menor o valor de cum_perc, que pode variar de 0 a 100.

Assim, itera-se sobre a população ranqueada extraindo-se, para posterior reprodução, indivíduos de acordo com esse peso. Um número aleatório `pick` entre 0 e 100 é escolhido para ser comparado com o valor de cum_perc; se `pick` $\leq cum\_perc$ em uma iteração, a rota sob avaliação é selecionada. Dessa maneira, qualquer cromossomo pode ser escolhido, mas aqueles com maior *fitness* têm maiores chances, uma vez que aparecem antes na lista. O número de iterações é equivalente à quantidade de indivíduos na população subtraída do valor de `eliteSize`.

Os indivíduos selecionados são, então, inseridos na lista `matingpool`, o que é feito no método `matingPool(population, selectionResults)`. Com a escolha feita, é possível realizar a reprodução sexuada, que é feita no método `breedPopulation(matingpool, eliteSize)`. O método em questão copia as rotas de elite para a variável que representa a nova geração, e os demais indivíduos desse conjunto são criados com o método `breed(parent1, parent2)`.

Para manter a nova geração válida em relação às restrições, isso é, criando indivíduos que passem por cada cidade exatamente uma vez, o método `breed` implementa um algoritmo de *crossover* ordenado, em que um subconjunto do pai 1 é copiado para o filho, e o restante de seus genes é preenchido com os genes do pai 2, em ordem, da direita para a esquerda, excluindo-se os genes do pai 2 que já tenham sido adicionados ao filho vindos do pai 1.

Outro método que o algoritmo utiliza para introduzir variabilidade na nova geração é o de mutação, feita, aqui, simplesmente trocando-se duas cidades de lugar em uma rota, a uma taxa definida pela variável `mutationRate`. Para cada indivíduo da nova geração, o método `mutatePopulation(population, mutationRate)` invoca o método `mutate(individual, mutationRate)`, que itera sobre o tamanho de uma lista de cidades e, a cada iteração, sorteia um número entre 0 e 1. Caso esse número seja menor que a taxa de mutação, a troca é feita; assim, quanto menor a taxa, menores as chances de uma mutação ocorrer.

Com isso, o algoritmo tem todos os elementos necessários para criar novas gerações a partir de uma população inicial. Isso é feito até que uma quantidade de iterações definida seja atingida. O código original imprime a menor distância da primeira população e a menor distância da última população, além de construir um gráfico que mostra a evolução das menores distâncias de cada geração, com auxílio da biblioteca Matplotlib.

A configuração original de valores proposta pelo algoritmo é:

- Tamanho de uma população: 100;
- Número de cidades: 25;
- Tamanho da elite: 20;
- Taxa de mutação: 0.01;
- Número de gerações: 500.

B. Modificações

1) Personalização: A primeira modificação introduzida no algoritmo foi a possibilidade de o usuário personalizar os valores do número de cidades (limitado a 30), tamanho da população (limitado a 150), tamanho da elite (limitado a 30), taxa de mutação (entre 0.001 e 0.1) e número de gerações (limitado a 1000).

Com isso, é possível testar diferentes configurações do algoritmo, determinadas pelo usuário, e visualizar resultados diversos. O usuário é informado sobre os limites aceitáveis de cada variável; caso eles sejam ultrapassados, o código atribui à variável em questão seu máximo valor possível.

O usuário também tem a opção de executar o código em sua versão original, descrita na seção III-A, ou na versão alternativa, descrita nesta seção III-B, com técnicas diferentes de seleção, *crossover* e mutação.

2) Visualização de dados e métricas de desempenho: O algoritmo original imprime as melhores distâncias inicial e final, além de desenhar um gráfico que mostra a evolução das melhores distâncias entre gerações (em azul).

Para complementar a visualização de dados, o grupo incluiu, no gráfico exibido, a evolução das piores distâncias entre gerações (em laranja). Acrescentamos, ainda, a impressão das piores distâncias e das distâncias médias na primeira e na última gerações.

Também foi incluído o cálculo do tempo total gasto pelo algoritmo até parar, com auxílio da função `time.time()`. Com isso, tornou-se possível comparar o desempenho do algoritmo em diferentes configurações, com parâmetros distintos.

3) Técnica de seleção: A seleção alternativa é feita no método `selectionAlt(popRanked, eliteSize)`. O elitismo, que necessariamente carrega os melhores indivíduos para a geração seguinte, é mantido. Para a escolha dos demais pais, optou-se pela seleção aleatória, com auxílio da função `random.choice()`.

4) Técnica de crossover: A técnica alternativa de reprodução sexuada é executada pelo método `breedPopulationAlt(matingpool, eliteSize)`, idêntico ao método original, exceto pelo detalhe de invocar o método `breedAlt(parent1, parent2)` em vez do método original de *crossover*.

O método `breedAlt(parent1, parent2)`, por sua vez, implementa a técnica de *crossover* cíclico, conforme descrito em [3]. Trata-se de um método em que as posições de algumas das cidades (genes) em um filho originam-se das posições dessas cidades em um dos pais; as demais posições são preenchidas copiando-as do pai restante.

5) Técnica de mutação: A técnica alternativa de mutação é implementada no método `mutateAlt(individual, mutationRate)`, invocado por `mutatePopulationAlt(population, mutationRate)`. Nele, em vez de trocar duas cidades de posição, trocam-se duas sequências de cidades de posição, tomando-se o cuidado de manter a primeira cidade intacta.

As sequências a ser trocadas são definidas traçando-se um divisor de posições — representado por um índice j — em um indivíduo. Essa divisão é feita na metade do tamanho da rota, o que é calculado com a operação de divisão inteira, de modo a lidar também com as rotas de tamanho ímpar. Em seguida, trocam-se de lugar a primeira e a segunda metades.

IV. Resultados

A. Versão original vs. Versão modificada

O primeiro teste realizado objetivou comparar a versão do algoritmo proposto originalmente com a versão que apresenta técnicas alternativas de mutação, seleção e *crossover*, o que chamamos aqui de versão modificada ou alternativa. Para compará-las, ambas as versões foram executadas com os valores *default* propostos, descritos na seção III-A.

Os resultados apresentados foram os seguintes:

Tabela I
Resultados do teste A

Parâmetros	Versão original	Versão modificada
Initial best distance	1948.8639073637612	2189.4099702480335
Initial worst distance	3030.6283212310514	2948.7896053587524
Initial average distance	2504.666908761309	2550.23593569763
Final best distance	769.9048993220708	1117.5232682940798
Final worst distance	1969.12069296721	2065.4895402805478
Final average distance	1056.9492762219097	1238.2373802507211
Time elapsed in seconds	19.209402799606323	8.048295974731445

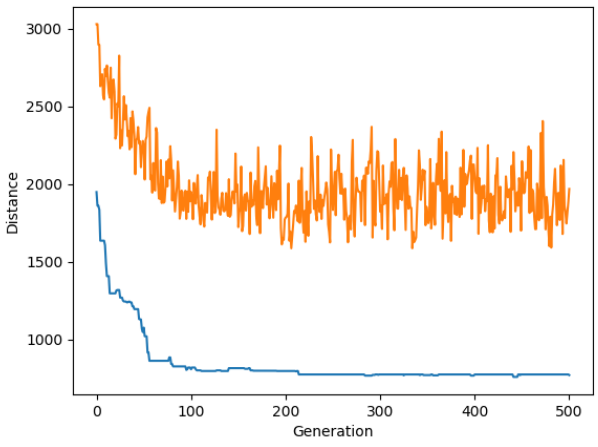

Figura 1. Versão original com os parâmetros *default*.

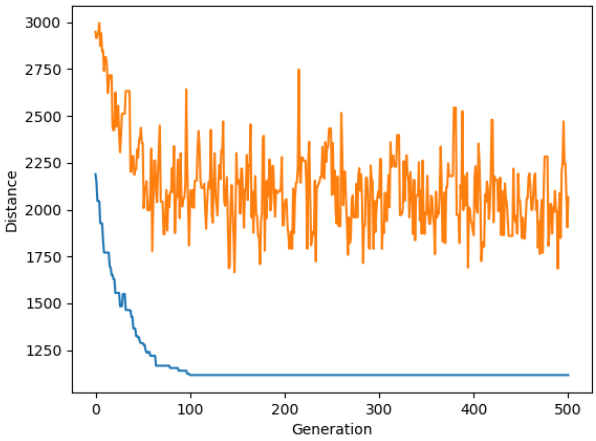

Figura 2. Versão modificada com os parâmetros *default*.

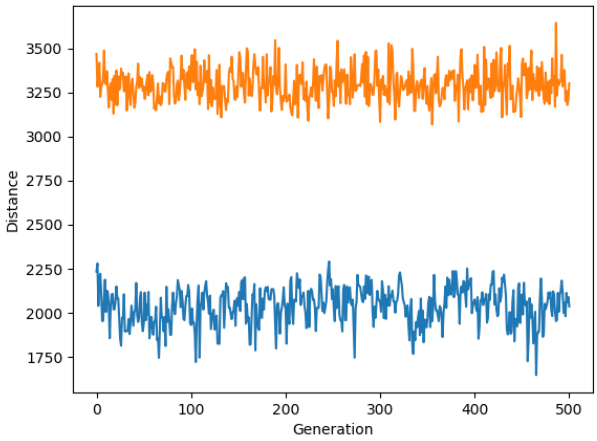

Figura 4. Resultados com taxa de mutação = 10%.

B. Alta taxa de mutação vs. Baixa taxa de mutação

O segundo teste foi realizado somente com a versão original do algoritmo. Foram utilizados todos os valores *default* descritos na seção III-A, à exceção da taxa de mutação: aqui, comparamos a menor taxa possível de mutação (0,1%) com a maior taxa possível (10%).

Os resultados apresentados foram os seguintes:

Tabela II
RESULTADOS DO TESTE B

Parâmetros	Mutação = 0,1%	Mutação = 10%
Initial best distance	1957.3952270489017	2234.8299421727897
Initial worst distance	2961.9036450846666	3468.482586979323
Initial average distance	2376.918750305613	2821.7856776446138
Final best distance	791.138042546842	2036.489398626242
Final worst distance	1572.6824364042004	3302.16578759801
Final average distance	1022.6557396765146	2655.7432327989973
Time elapsed in seconds	19.300386428833008	19.775139331817627

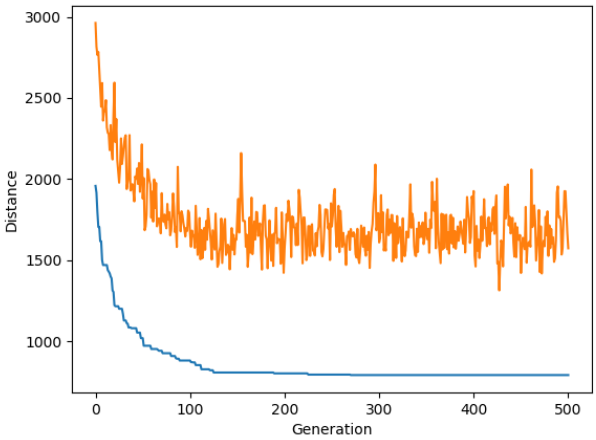

Figura 3. Resultados com taxa de mutação = 0,1%.

Dada a ausência de convergência no último caso, o algoritmo foi executado novamente, com os mesmos parâmetros, exceto o número de gerações, que foi alterado para 1000 (isto é, dobrado). Os resultados, porém, não foram significativamente diferentes, com o único destaque sendo o aumento do tempo total de execução (40.056416511535645):

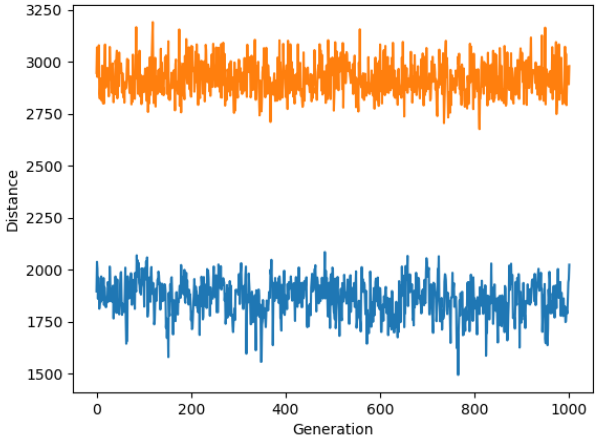

Figura 5. Resultados com taxa de mutação = 10% e número de gerações = 1000.

C. Grande proporção da elite vs. Baixa proporção da elite

O terceiro teste foi realizado com ambas as versões do algoritmo. O objetivo, aqui, foi comparar o comportamento do algoritmo quando modificamos significativamente a proporção entre o tamanho da elite e o tamanho da população. Para tanto, todos os parâmetros foram mantidos nos valores *default* — inclusive o tamanho da população, configurado em 100 —, exceto o tamanho da elite, que foi testado ora com o

valor 5, ora com o valor 30.

Os resultados apresentados foram os seguintes:

Tabela III
RESULTADOS DO TESTE C - VERSÃO ORIGINAL DO ALGORITMO

Parâmetros	eliteSize = 5	eliteSize = 30
Initial best distance	2086.521365029282	1937.2600250362325
Initial worst distance	3110.0000601654688	3054.293939220791
Initial average distance	2665.187974861362	2407.998805164129
Final best distance	1020.5691378469068	877.1164038367368
Final worst distance	2412.8470362934313	1594.5879907392525
Final average distance	1674.8109676181787	1077.3464931778417
Time elapsed in seconds	19.957830905914307	18.20543932914734

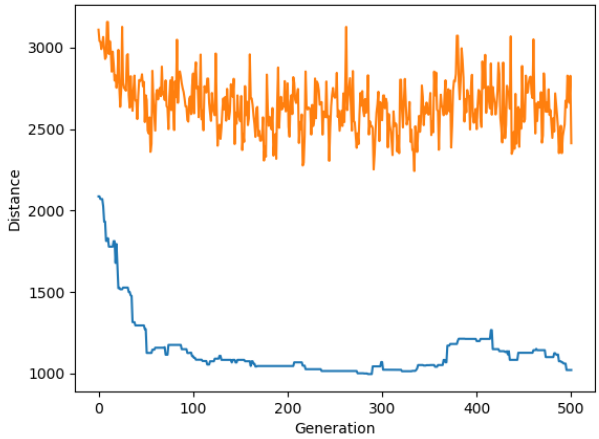

Figura 6. Resultados para o algoritmo original com eliteSize = 5.

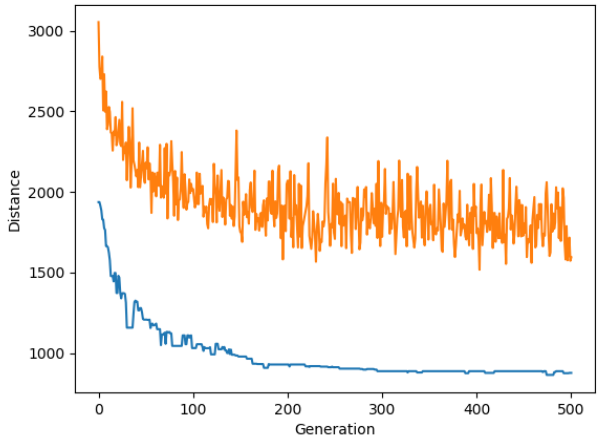

Figura 7. Resultados para o algoritmo original com eliteSize = 30.

Tabela IV
RESULTADOS DO TESTE C - VERSÃO ALTERNATIVA DO ALGORITMO

Parâmetros	eliteSize = 5	eliteSize = 30
Initial best distance	2220.7330762634588	2004.7045905263972
Initial worst distance	3216.105910796282	3229.3269881949886
Initial average distance	2721.3089776174997	2604.3915196927355
Final best distance	1003.8844795466074	1006.6329675159233
Final worst distance	2477.4400852240665	1697.7947899476694
Final average distance	1421.4248590889456	1087.3890669815182
Time elapsed in seconds	8.285218477249146	8.081169843673706

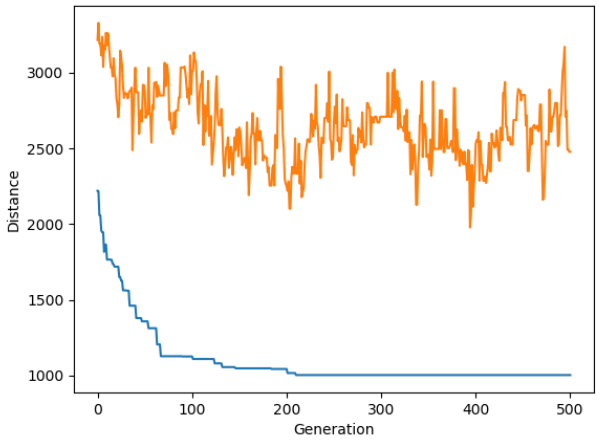

Figura 8. Resultados para o algoritmo alternativo com eliteSize = 5.

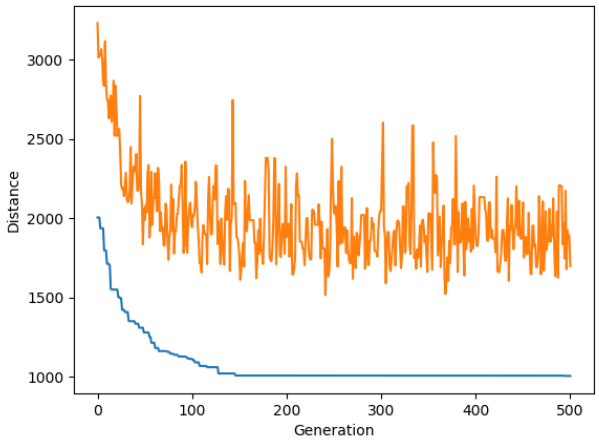

Figura 9. Resultados para o algoritmo alternativo com eliteSize = 30.

V. DISCUSSÃO E CONCLUSÕES

De modo geral, o trabalho realizado mostrou que a abordagem da programação genética provê boas soluções para o problema descrito, desde que observados determinados parâmetros.

Tanto a proposta original do algoritmo quanto a alternativa, modificada pelo grupo, exibiram, com os valores

default, convergência para uma solução. A versão alternativa terminou sua execução em menos tempo e começou a convergir antes da versão original, com aproximadamente 100 gerações. Dado que ambas as versões foram testadas com as mesmas taxas de mutação, e que a seleção na versão alternativa se dá de maneira aleatória sem modificar o elitismo – o que, em tese, não é melhor que a técnica de seleção original –, a melhora se deve provavelmente à implementação da técnica do *crossover* cíclico.

O algoritmo mostrou-se sensível a alterações na taxa de mutação – o que é esperado, dado que a mutação introduz variabilidade na população e, se realizada em demasia, produz resultados com maior aleatoriedade. A configuração da taxa de mutação para 10% já foi suficiente para que o algoritmo não convergisse para uma solução de maneira alguma, mesmo quando dobramos o número de gerações. É interessante notar, porém, que as melhores distâncias permanecem consistentemente melhores que as piores distâncias, ao longo de todas as gerações, o que fica claro ao observar as figuras 4 e 5.

Quanto à proporção da população a ser selecionada por elitismo, observamos que, para uma elite de tamanho grande, ambas as versões do algoritmo convergem para soluções igualmente boas a partir de aproximadamente 200 gerações. Não nos parece se tratar de uma convergência prematura, dado que o resultado exibido é consistente com o de outras execuções, com proporções médias da elite (valores *default*). A diferença apresentada entre as versões é, novamente, o tempo de execução: aqui, também, a versão modificada revelou-se mais rápida.

O algoritmo parece mais sensível, contudo, à adoção de uma elite pequena em relação à população. Isso faz sentido quando lembramos que o elitismo garante de maneira inequívoca a seleção de indivíduos mais aptos; assim, sem muitos indivíduos pertencentes a uma elite, torna-se mais provável que uma população apresente uma proporção maior de indivíduos menos aptos. Curiosamente, a versão original – que realiza uma seleção baseada na *fitness* – mostrou-se mais sensível a essa alteração e menos propensa a convergir que o algoritmo alternativo, que seleciona indivíduos aleatoriamente para além da elite. Novamente, uma hipótese plausível é a adoção da técnica de *crossover* cíclico para realizar a reprodução sexuada. Com isso, torna-se claro que, na abordagem proposta, o elitismo é um mecanismo importante para garantir a convergência.

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Algoritmos: Teoria e Prática [Tradução da segunda edição americana]*. Elsevier, 2002. [Online]. Available: https://www.cin.ufpe.br/~ara/algoritmos-%20portugu%EAs-%20cormen.pdf 1

[2] E. Stoltz, "Evolution of a salesman: A complete genetic algorithm tutorial for Python," 2018. [Online]. Available: https://bit.ly/2P5wvuR 1

[3] O. Abdoun and J. Abouchabaka, "A comparative study of adaptive crossover operators for genetic algorithms to resolve the traveling salesman problem," *International Journal of Computer Applications*, vol. 31, no. 11, pp. 49–57, 2011. [Online]. Available: https://arxiv.org/pdf/1203.3097.pdf 3