Universidade Federal
de São João del-Rei

Departamento de Ciência da Computação

Vítor Augusto Niess Soares Fonseca

Relatório Algoritmo Genético Para Caixeiro Viajante

São João del-Rei, Agosto de 2024

Sumário

1 Introdução

Os algoritmos genéticos (AGs) são uma classe de algoritmos de otimização baseados em princípios biológicos de evolução natural. Inspirados no processo de seleção natural, os AGs são capazes de resolver uma ampla variedade de problemas de otimização em diferentes domínios.

O problema do caixeiro viajante (PCV) é um problema clássico de otimização combinatória que envolve encontrar o caminho mais curto que permite a um caixeiro viajante visitar um conjunto de cidades exatamente uma vez e retornar à cidade de origem. Este problema é representado por uma matriz de distâncias entre as cidades e a solução ótima é a rota que minimiza a distância total percorrida. Apesar de sua simplicidade conceitual, o PCV é NP-difícil, o que significa que não existe uma solução algorítmica eficiente para encontrar a solução ótima em tempo polinomial para instâncias de tamanho arbitrário.

Neste trabalho, propomos a utilização de algoritmos genéticos para abordar o problema do caixeiro viajante. Especificamente, investigaremos o impacto de diferentes métodos de seleção de pais e operadores de cruzamento de indivíduos na eficácia do algoritmo genético em encontrar soluções para o problema descrito. Duas abordagens distintas serão comparadas: o método de seleção de pais pelo torneio e pelo método da roleta, utilizando o operador de crossover OX (Order Crossover).

2 Metodologia

Para abordar o problema do caixeiro viajante utilizando algoritmos genéticos, implementamos um conjunto de funções em Python que compõem o núcleo do nosso algoritmo. As principais etapas do algoritmo incluem a geração da população inicial, a avaliação dos indivíduos, a seleção dos pais, o cruzamento, a mutação e a geração da próxima geração. Cada indivíduo na população é representado por uma permutação de números inteiros, onde cada número inteiro corresponde a uma cidade. A ordem dos números na permutação representa a ordem em que as cidades serão visitadas pelo caixeiro viajante.

2.1 Geração da População Inicial

A função `genIniPop(popSize, cNum)` gera a população inicial de tamanho `popSize`, onde cada indivíduo é uma permutação aleatória de `cNum` cidades. Esta função utiliza a função `random.sample()` para criar indivíduos únicos na população inicial.

2.2 Avaliação dos Indivíduos

A função `fitness(pop, disM)` calcula o fitness de cada indivíduo na população. O fitness é definido como a distância total percorrida pelo caixeiro viajante ao visitar todas as cidades na ordem especificada pelo indivíduo. A função retorna uma lista contendo o valor de fitness para cada indivíduo.

2.3 Seleção dos Pais

Implementamos dois métodos de seleção de pais: o método do torneio e o método da roleta. Cada método tem suas próprias características e estratégias para selecionar os indivíduos que participarão do cruzamento.

2.3.1 Método do Torneio

A função `tourn(popSize, fit)` realiza um torneio entre dois candidatos selecionados aleatoriamente da população. Este método funciona da seguinte maneira:

- Dois indivíduos (candidatos) são selecionados aleatoriamente da população.
- Uma competição é realizada entre os dois candidatos, onde o indivíduo com melhor fitness tem uma maior chance de ser escolhido.
- A porcentagem de vitória (vicP) determina a probabilidade de o candidato com melhor fitness vencer. Se um número aleatório gerado for maior que vicP, o indivíduo com pior fitness é escolhido.
- Este processo é repetido até que todos os pais necessários sejam selecionados.

O método do torneio é eficiente e tem a vantagem de ser simples de implementar. Ele também permite controlar a pressão de seleção ajustando a porcentagem de vitória.

2.3.2 Método da Roleta

A função `roul(popSize, fit)` utiliza o método da roleta para selecionar os pais. Este método funciona da seguinte maneira:

- O valor de fitness de cada indivíduo é invertido (1/fitness) para que indivíduos com menor distância total (melhor fitness) tenham maior probabilidade de serem selecionados.
- A soma total dos valores invertidos de fitness é calculada.
- A probabilidade de seleção de cada indivíduo é proporcional ao seu valor invertido de fitness dividido pela soma total.
- Uma roleta virtual é criada onde cada indivíduo ocupa uma porção da roleta proporcional à sua probabilidade de seleção.
- Um número aleatório é gerado e o indivíduo correspondente à porção da roleta onde o número caiu é selecionado.
- Este processo é repetido até que todos os pais necessários sejam selecionados.

O método da roleta é intuitivo e assegura que todos os indivíduos têm uma chance de serem selecionados, com os melhores indivíduos tendo maiores probabilidades. No entanto, ele pode ser menos eficiente que o método do torneio em termos de tempo de execução para grandes populações.

2.4 Cruzamento e Mutação

A função `crossover(pop, fat, cNum, crossRate)` realiza o cruzamento de dois pais utilizando o operador OX (Order Crossover), que será explicado com mais detalhes posteriormente. A função `mutate(pop, cNum, mutRate)` realiza a mutação dos indivíduos, onde cada gene tem uma probabilidade de ser trocado com outro gene no indivíduo de acordo com a taxa de mutação `mutRate`.

2.4.1 Cruzamento OX (Order Crossover)

O operador OX (Order Crossover), mantém a ordem relativa dos elementos nos filhos. O processo é descrito da seguinte maneira:

- Dois pontos de corte são selecionados aleatoriamente nos pais.
- Os elementos entre os pontos de corte de um pai são copiados diretamente para o filho.
- Os elementos restantes são preenchidos com base na ordem em que aparecem no segundo pai, começando a partir do segundo ponto de corte.

Por exemplo, se os pais são `P1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]` e `P2 = [9, 8, 7, 6, 5, 4, 3, 2, 1]` e os pontos de corte são 3 e 6, o processo seria:

- Copiar os elementos entre os pontos de corte do pai 1 para o filho: `F1 = [X, X, X, 4, 5, 6, X, X, X]`.
- Preencher os elementos restantes na ordem em que aparecem no pai 2: `F1 = [9, 8, 7, 4, 5, 6, 3, 2, 1]`.

O operador OX é eficaz em manter a estrutura dos pais, o que é crucial para problemas como o caixeiro viajante, onde a ordem dos elementos é importante.

2.5 Geração da Próxima Geração

A função `execAlgGen(popSize, genNum, cNum, pop, disM, type, mutRate, crossRate)` combina todas as etapas anteriores para gerar a próxima geração de indivíduos. Esta função calcula o fitness da população atual, seleciona os pais, realiza o cruzamento e a mutação, e mantém o melhor indivíduo (elitismo) para a próxima geração.

2.6 Parâmetros e Execução

Definimos os parâmetros do algoritmo, incluindo o tamanho da população (`popSize`), o número de gerações (`genNum`), a taxa de mutação (`mutRate`), a taxa de cruzamento (`crossRate`) e o método de seleção de pais (`selMet`). Em seguida, executamos o algoritmo por um número específico de gerações, registrando o melhor fitness de cada geração e salvando-os em um arquivo ordenado de forma decrescente, utilizado para posterior análise.

3 Resultados e Discussões

Para avaliar o desempenho dos algoritmos genéticos na resolução do problema, foram realizados testes utilizando diferentes configurações. Cada teste consistiu em executar o algoritmo com o método de seleção de pais pelo torneio, uma vez que o método da roleta foi desconsiderado devido a resultados inconsistentes. Foram avaliadas diferentes taxas de mutação (1%, 5% e 10%) e variações no tamanho da população.

A taxa de mutação que apresentou melhores resultados foi de 1% (0.01), visto que taxas de mutação mais altas não mostraram vantagens significativas e, em alguns casos, prejudicaram o desempenho do algoritmo. Gráficos de desempenho para cada taxa de mutação estão apresentados a seguir.

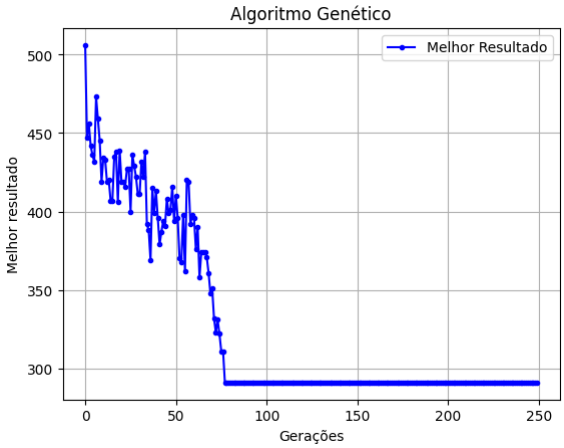

Figura 1: Desempenho do algoritmo com taxa de mutação de 1%.

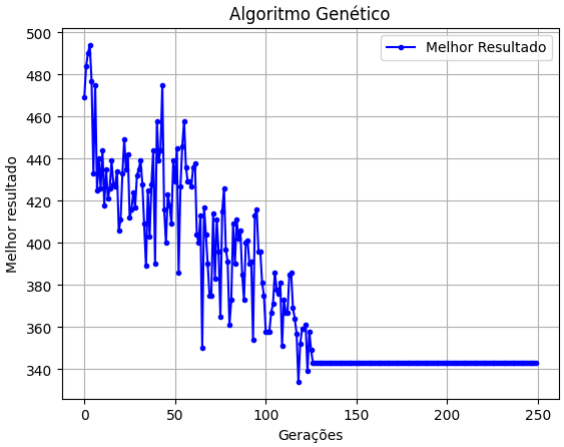

Figura 2: Desempenho do algoritmo com taxa de mutação de 5%.

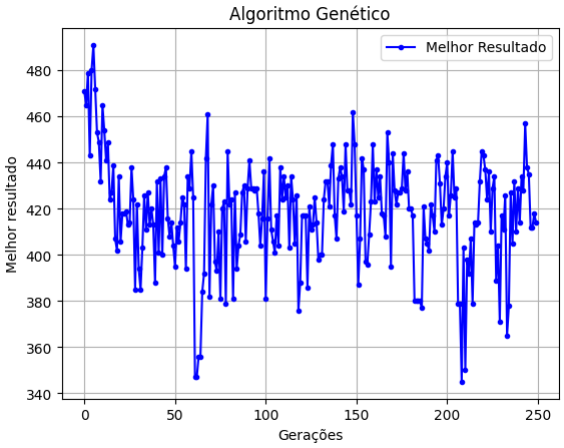

Figura 3: Desempenho do algoritmo com taxa de mutação de 10%.

Obtida a taxa de mutação ótima de 1%, mais testes foram feitos, conforme o número de gerações: 250, 500 e 800. A partir deles, as seguintes observações foram feitas:

- Com 250 gerações:
  - 100 indivíduos: 90 gerações para encontrar o melhor indivíduo.

  - 200 indivíduos: 80 gerações para encontrar o melhor indivíduo.
  - 300 indivíduos: 60 gerações para encontrar o melhor indivíduo.

- Com 500 gerações:
  - 100 indivíduos: 120 gerações para encontrar o melhor indivíduo.
  - 200 indivíduos: 70 gerações para encontrar o melhor indivíduo.
  - 300 indivíduos: 60 gerações para encontrar o melhor indivíduo.

- Com 800 gerações:
  - 100 indivíduos: 90 gerações para encontrar o melhor indivíduo.
  - 200 indivíduos: 90 gerações para encontrar o melhor indivíduo.
  - 300 indivíduos: 60 gerações para encontrar o melhor indivíduo.

Os resultados indicam que o número de gerações não teve um impacto significativo na obtenção do melhor indivíduo, já que o melhor indivíduo foi encontrado consistentemente até, em média, a centésima geração. O fator mais influente na obtenção de melhores resultados foi o tamanho da população.

Portanto, os resultados demonstram que a configuração ótima para o algoritmo genético neste contexto é com uma taxa de mutação de 1% e uma população de 300 indivíduos, pois esses parâmetros proporcionaram a convergência mais rápida e eficaz para encontrar o melhor indivíduo.

4 Considerações Finais

Este trabalho proporcionou uma experiência valiosa no desenvolvimento e análise de algoritmos genéticos aplicados.

Os testes realizados com diferentes tamanhos de população e taxas de mutação demonstraram que a configuração do algoritmo genético tem um impacto significativo na sua eficácia. No caso específico do trablho, observou-se que o número de gerações não influenciou substancialmente o tempo necessário para encontrar o melhor indivíduo, sendo o tamanho da população o fator mais determinante para a obtenção de resultados melhores. Além disso, a taxa de mutação de 1% (0.01) revelou-se a mais eficaz, promovendo uma convergência mais rápida e estável para a solução ótima.

Com a análise e os resultados obtidos, fica evidente que a escolha adequada dos parâmetros utilizados, como o tamanho da população e a taxa de mutação, pode otimizar significativamente o desempenho do algoritmo, oferecendo uma abordagem mais eficiente para problemas de otimização combinatória.

Em suma, este trabalho não só ampliou a compreensão sobre algoritmos genéticos e sua aplicação a problemas de otimização, mas também ofereceu aprendizados para ajustes de parâmetros e estratégias de implementação. O conhecimento adquirido durante este estudo é valioso para futuras pesquisas em algoritmos evolutivos e em outras abordagens de resolução de problemas de otimização mais complexos.