

# Algoritmos Genéticos - alguns problemas combinatórios

Carolina Ribeiro Xavier

Fevereiro de 2021

## 1 Problema da Mochila binária

O problema da mochila binária consiste em um problema de maximização da utilidade dos itens levados( $v$ ) restrito à capacidade da mochila( $c$ ).

Formalmente, dados dois vetores  $p$  e  $v$  de  $n$  posições e um número  $c$ , deseja-se encontrar um subconjunto  $X$  de  $\{0, 1, \dots, n - 1\}$  que maximize  $v(X)$  sob a restrição  $\sum p(X) \leq c$ .

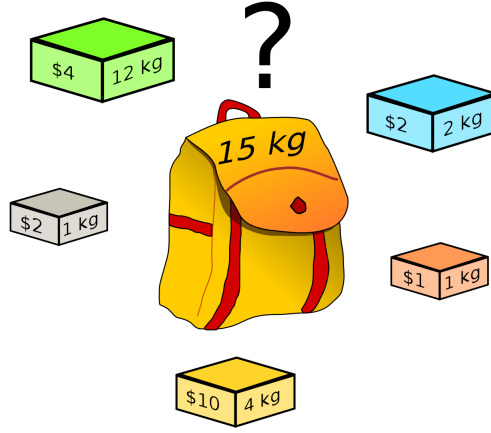


Figura 1: Ilustração do problema da Mochila

Diremos que  $0, 1, 2, \dots, n - 1$  são os objetos do problema,  $p[i]$  é o peso do objeto  $i$ , e que  $v[i]$  é o valor (utilidade) do objeto  $i$ . Diremos que  $c$  é a capacidade da instância. Diremos ainda que um subconjunto  $X$  de  $\{0, 1, 2, \dots, n\}$  é uma mochila se  $\sum p(X) \leq c$ . O valor de uma mochila  $X$  é dado pela função  $v(X)$ , e nosso objetivo é encontrar uma mochila de valor máximo, e.g maximizar  $v(X)$ .

$$v(X) = \sum_{\forall i \in X} v[i] \quad (1)$$

## 2 Questões de projeto

Como já foi dito, existem várias questões que podem mudar de acordo com o problema a ser resolvido, as mais importantes são as questões relativas à função objetivo escolhida e a representação, sendo que a segunda reflete na implementação de detalhes de todos os operadores genéticos. As principais questões são as listadas a seguir:

- Função objetivo com **penalização**;
- Representação;
- Estratégia de seleção; ✓
- Cruzamento, pode ser feito o de  $n$  pontos; ✓
- Mutação, pode ser feita a de negação do bit; ✓
- Elitismo (usar o elitismo de 1); ✓

Valores como tamanho da população e número máximo de gerações do AG também devem ser testados para verificar o melhor cenário para solução do problema que se quer resolver.

## 2.1 Função objetivo

A função objetivo é uma função que avalia a proximidade do indivíduo da solução do problema, o problema tratado na primeira parte deste tutorial é o problema da mochila binária, um problema de maximização sujeito à restrições. O valor de *fitness* ou aptidão do indivíduo será uma composição da utilidade da instância com uma penalização, caso a instância não seja uma solução viável (não é uma mochila).

Se a instância é viável, ou seja, o peso dos elementos em  $X$  não excedem a capacidade  $c$ , o fitness é dado somente pela utilidade:

$$fitness = \sum_{\forall i \in X} v[i] \quad (2)$$

**Caso contrário, faremos uma penalização.**

$$fitness = \sum_{\forall i \in X} v[i] * (1 - (\sum_{\forall i \in X} p[i] - c)/c) \quad (3)$$

Observe que a segunda parte da função calcula o per-

centual de peso excedido, em seguida reduz a utilidade na proporção inversa. Este tipo de penalização é muito usado quando você pode ter alguma tolerância a violação das restrições.

Caso o seu AG esteja constantemente sendo dominado por soluções inviáveis, e isso não possa ser permitido, você pode usar uma penalização mais drástica, que não permitirá nunca que uma solução viável tenha *fitness* maior que uma não viável, pois tornará o *fitness* de todas as soluções inviáveis.

$$fitness = \sum_{\forall i \in X} v[i] - (\sum_{\forall i \in X} v[i] * (\sum_{\forall i \in X} p[i] - c)) \quad (4)$$

## 2.2 Representação

Para este problema temos duas alternativas:

- vetor binário indexado pelo id do objeto, onde o zero significa que o objeto não faz parte da solução e o 1 indica que ele faz parte da solução;

- vetor com os objetos presentes na solução, onde cada posição é um objeto. (este só deve ser usado se o número de objetos disponíveis for muito grande e inviabilizar a opção anterior, pois teremos que admitir operações que varie o tamanho da solução e que não permita repetição de itens).

## 2.3 Estratégia de seleção

Você pode testar as duas formas de seleção que vimos até agora, e pode ainda fazer umas variações para ajustar o método que você escolheu.

Dicas para melhorias dos métodos:

- suavização da roleta (experimentem isso quando o mesmo pai está sendo selecionado várias vezes devido a uma grande diferença do *fitness* dele para os dos demais indivíduos, isso diminui a pressão seletiva, dando oportunidades a indivíduos com *fitness* piores, você pode usar o *rank*, por exemplo);
- torneio com o número de indivíduos maior (experimente quando o algoritmo estiver estagnado por

muitas gerações sem aproximação do ótimo, em geral com a média de *fitness* bastante variável), isso aumenta a pressão seletiva;

## 2.4 Cruzamento

O cruzamento é uma operação muito importante para **intensificação** do espaço de busca, ele acontece entre dois indivíduos selecionados por algum critério.

O tipo cruzamento mais comum para representação por códigos binários é o cruzamento de  $n$  pontos. Nesse tipo de cruzamento são gerados, em geral, dois filhos para comporem a nova população. Um exemplo de cruzamento de  $n=2$  pontos pode ser ilustrado pela Figura 2.

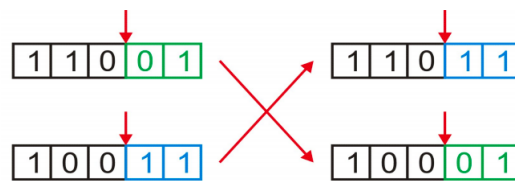


Figura 2: Cruzamento de um ponto

Este tipo de cruzamento pode ser usado nas duas representações, mas deve-se observar na segunda opção se

não houve repetição de itens.

## 2.5 Mutação

A operação de mutação tem o papel de **diversificação** da população, muitas vezes os indivíduos ficam presos a ótimos locais e precisam de uma perturbação para que seja possível escapar desses locais.

Após a aplicação do operador de cruzamento, o operador de mutação é aplicado na população intermediária. Para cada indivíduo (por vezes para cada gene ou alelo do indivíduo) da população é sorteado um valor entre 0 e 1, caso esse valor seja menor ou igual a taxa de mutação estipulada o indivíduo (ou seu gene/alelo) é substituído. No caso da representação binária, pode-se manter a ideia anterior de inverter o bit, e no caso da representação por id de objetos pode-se substituir o objeto do cromossomo por outro aleatório, ou ainda remover(ou adicionar) um objeto aleatório.

Para isso é preciso fixar uma taxa de mutação e/ou alguns critérios de como ela pode variar durante a execução do algoritmo. Valores comuns para essa taxa não costu-



podem ultrapassar 20%, sendo mais comuns valores de 1% a 10%.

**Se o seu algoritmo está estagnado, você pode tentar após um número pré-fixado de gerações estagnadas, aumentar a taxa de mutação por algumas (poucas) gerações para dar diversidade à sua solução, depois retornar a taxa padrão.**

### 3 Problema do Caixeiro viajante

O problema do caixeiro viajante consiste na busca por um circuito que possua a menor distância, começando numa cidade qualquer, entre várias, e visitando todas cidades, cada uma precisamente uma vez, voltando então para a cidade de origem.

Dado um conjunto  $C = \{c_1, \dots, c_n\}$  de  $n$  cidades  $c_i$  e uma matriz de distâncias  $(\rho_{ij})$ , onde  $\rho_{ij} = \rho(c_i, c_j)$  ( $i, j \in \{1, \dots, n\}$ ,  $\rho_{ij} = \rho_{ji}$ ,  $\rho_{ii} = 0$ ), a tarefa passa por encontrar a permutação  $\pi \in S_n = \{s : \{1, \dots, n\} \rightarrow \{1, \dots, n\}\}$  que faça com que a função objetivo (distância

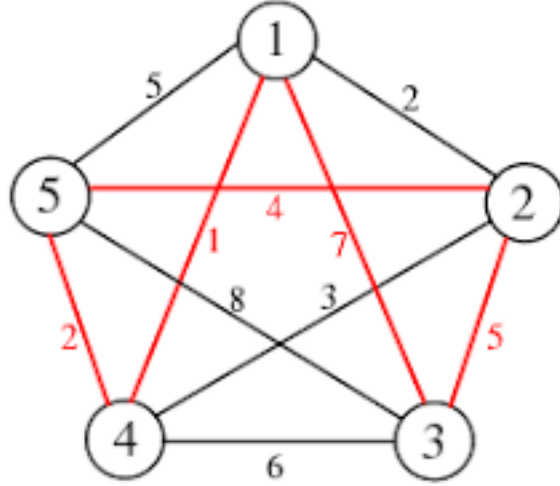


Figura 3: Grafo completo para aplicação do problema do caixeiro viajante

do circuito)  $f : S_n \rightarrow \mathbb{R}$ , onde:

$$f(\pi) = \sum_{i=1}^{n-1} \rho(\pi(i), \pi(i+1)) + \rho(\pi(n), \pi(1)), \quad (5)$$

seja mínima.

## 4 Questões de projeto

Como já foi dito, existem várias questões que podem mudar de acordo com o problema a ser resolvido, as mais importantes são as questões relativas à função objetivo escolhida e a representação, sendo que a segunda reflete na implementação de detalhes de todos os operadores genéticos. As principais questões são as listadas a seguir:

- Função objetivo;
- Representação - permutação das cidades;
- Estratégia de seleção; ✓
- **Cruzamento;**
- Mutação;
- Elitismo; ✓

Valores como tamanho da população e número máximo de gerações do AG também devem ser testados para verificar o melhor cenário para solução do problema que se quer resolver.

## 4.1 Função objetivo

A função objetivo do problema do caixeiro viajante será direta, basta fazer o cálculo da distância do percurso dado por 5.

## 4.2 Representação

A representação da solução do problema do caixeiro viajante será feita por um vetor de  $n$  posições, que contenha alguma permutação dos vértices da instância a ser otimizada (minimização).

## 4.3 Cruzamento

Para o cruzamento vamos mudar completamente o que vínhamos fazendo por se tratar de um problema de permutação. Veremos um cruzamento baseado em ordem, o *ox-crossover*.

Neste operador, sortearmos dois pontos, os valores entre estes dois pontos serão preservados nos filhos na mesma posição que eles ocorrem nos pais. Como são dois

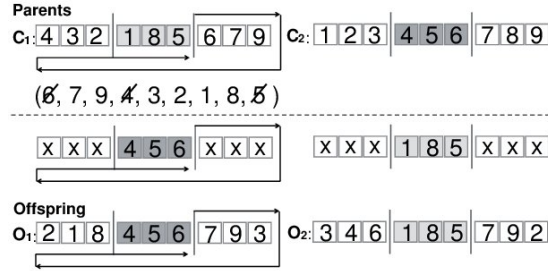


Figura 4: Cruzamento baseado em ordem

filhos, gerados por dois pais, cada filho herda o trecho de um dos pais, o restante do filho será preenchido de acordo com a ordem que os índices ocorrem no segundo pai(o que passou o trecho para o outro filho).

Esse preenchimento é feito de forma a não ocorrer repetição no indivíduo, observe a Figura 4, o O1 está com o trecho preservado de C2 e as posições marcadas na segunda linha com x, serão preenchidas de acordo com C1. Partindo da posições subsequente ao segundo ponto sorteado, as posições recebem o índice de C1 se ele ainda não fizer parte de O1. Neste exemplo, ele não recebe o valor 6, recebe os valores 7 e 8, passa para o início do pai C1, não recebe o valor 4, recebe o valor 3, e passa a preencher o início de O1, recebendo os valores 2, 1 e 8.

Observe que dessa forma, a maioria das arestas potencialmente boas nos pais que passaram por uma seleção, permanecem nos filhos, as alterações permitem uma busca em soluções próximas as soluções encontradas até agora.

## 4.4 Mutação

O operador de mutação será bastante simples. Para cada gene do cromossomo sortearemos um valor  $r \in [0, 1]$ , se o  $r \leq pm$ , sortearemos a posição de outro gene e faremos a troca dos valores desses dois genes.

**Pode ocorrer de haver mais de uma troca em um mesmo indivíduo, mas é importante que você faça uma troca de cada vez para garantir a manutenção de todos os valores da permutação.**

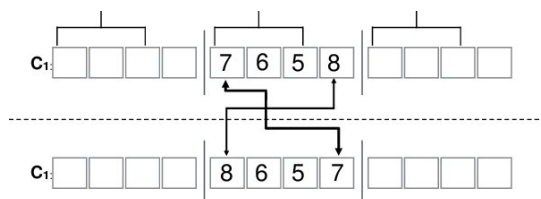


Figura 5: Mutação por troca de posição

## 5 Implementação - escolha uma dupla e entreguem os dois problemas

### 5.1 Mochila binária

Agora vamos implementar um AG simples com representação à sua escolha para a mochila, e seguirá os passos do fluxograma da Figura 6.

Você deve definir a estrutura de dados que você irá armazenar os indivíduos e seus respectivos valores de *fitness*.

Defina os valores listados de acordo com o experimento fatorial que vimos na última tarefa e use a alguma instância encontrada **aqui**.

Cada instância possui três arquivos, sendo:  
pxx\_c.txt, a capacidade da mochila;  
pxx\_w.txt, pesos dos objetos;  
pxx\_s.txt, valor ótimo para que você avalie a solução do seu AG.

- Tamanho da população;

- Número máximo de gerações a serem executadas;
- Critério de seleção de pais;
- Taxa de cruzamento;
- Taxa de mutação;
- Elitismo.

## 5.2 Caixeiro Viajante

Agora vamos implementar um AG simples para a caixeiro viajante, que seguirá os passos do fluxograma da Figura 6.

Você deve definir a estrutura de dados que você irá armazenar os indivíduos e seus respectivos valores de *fitness*.

Defina os valores listados de acordo com o experimento fatorial que vimos na última tarefa e use a alguma instância encontrada **aqui** ou **aqui**.

As instâncias que melhor representam o problema no primeiro link são:



LAU15 - que possui o resultado da solução ótima e a matriz de distâncias (lau15\_dist.txt)

SGB128 - que não traz a solução ótima, mas possui a matriz de distâncias (sgb128\_dist.txt)

No segundo link você deve se atentar aos dados pos-suam problemas simétricos.

- Tamanho da população;
- Número máximo de gerações a serem executadas;
- Critério de seleção de pais;
- Taxa de cruzamento;
- Taxa de mutação;
- Elitismo.

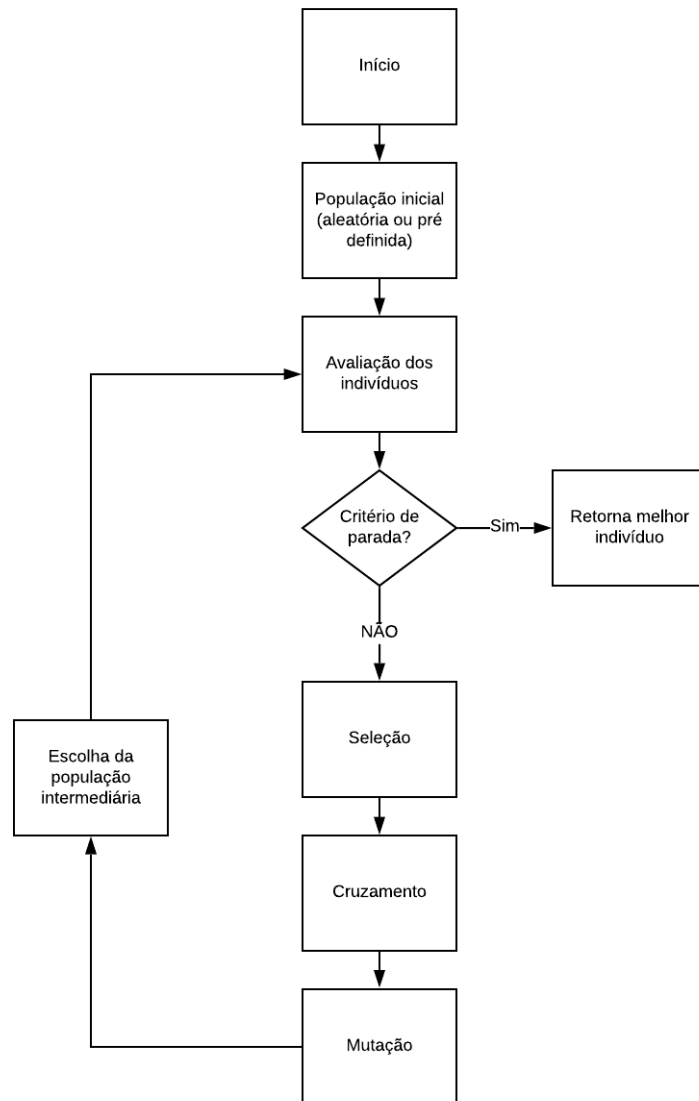


Figura 6: Fluxograma de um algoritmo genético básico