Computação Evolucionária

Métodos de extração de motifs

- Métodos exatos:
  - Procura exaustiva;
  - Branch & Bound
- Métodos heurísticos
  - Tipo consenso
- Algoritmos estocásticos:
  - Algoritmos heurísticos estocásticos;
  - Gibbs sampling
  - **Algoritmos evolucionários**

Algoritmos Genéticos e Evolucionários: o que são ?

- Família de **modelos computacionais** usados nos campos da Optimização, das Ciências da Computação, da Aprendizagem, da Inteligência Artificial, da Engenharia, etc.

- Mimetizam a **evolução das espécies por seleção natural** (cf. teorias de Darwin) – implementam operadores para criar novos indivíduos bem como operadores de seleção dos indivíduos mais aptos - **EVOLUCIONÁRIOS**.

Algoritmos Genéticos e Evolucionários: o que são ?

- Trabalham com **populações**, onde cada indivíduos representa uma solução para um dado problema.
- Soluções geralmente codificadas numa sequência de símbolos de um dado alfabeto (genoma) que permite a aplicação de operadores.
- Novas soluções criadas a partir de operadores de mutação ou recombinação sexuada – **GENÉTICOS**.

Codificação

- Processo de **representação das soluções** para o problema alvo, num dado alfabeto;
- Inicialmente, usado apenas o alfabeto binário (cf. **Algoritmos Genéticos** de Holland, 1975);
- Progressivamente, adoptados outros alfabetos de representação: nºs inteiros, nºs reais, permutações, árvores de expressões, etc;
- Tipicamente, genomas constituídos por sequências **lineares** (cromossomas) de valores (**genes**).
- A "qualidade" de um algoritmo evolucionário depende em grande parte a forma como as soluções são codificadas.

Codificação - Exemplos

- Motifs
  - Vetor de posições iniciais s

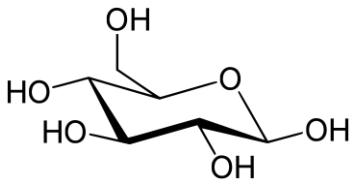
s ={3,10,5,11,13,4,1} → [3,10,5,11,13,4,1]

  - Consenso

A C G T A C G T → [1,2,4,3,1,2,4,3]

1 = A
2 = C
3 = T
4 = G

- Moléculas



Glucose ($C_6H_{12}O_6$)

•SMILE (lista de caracteres):

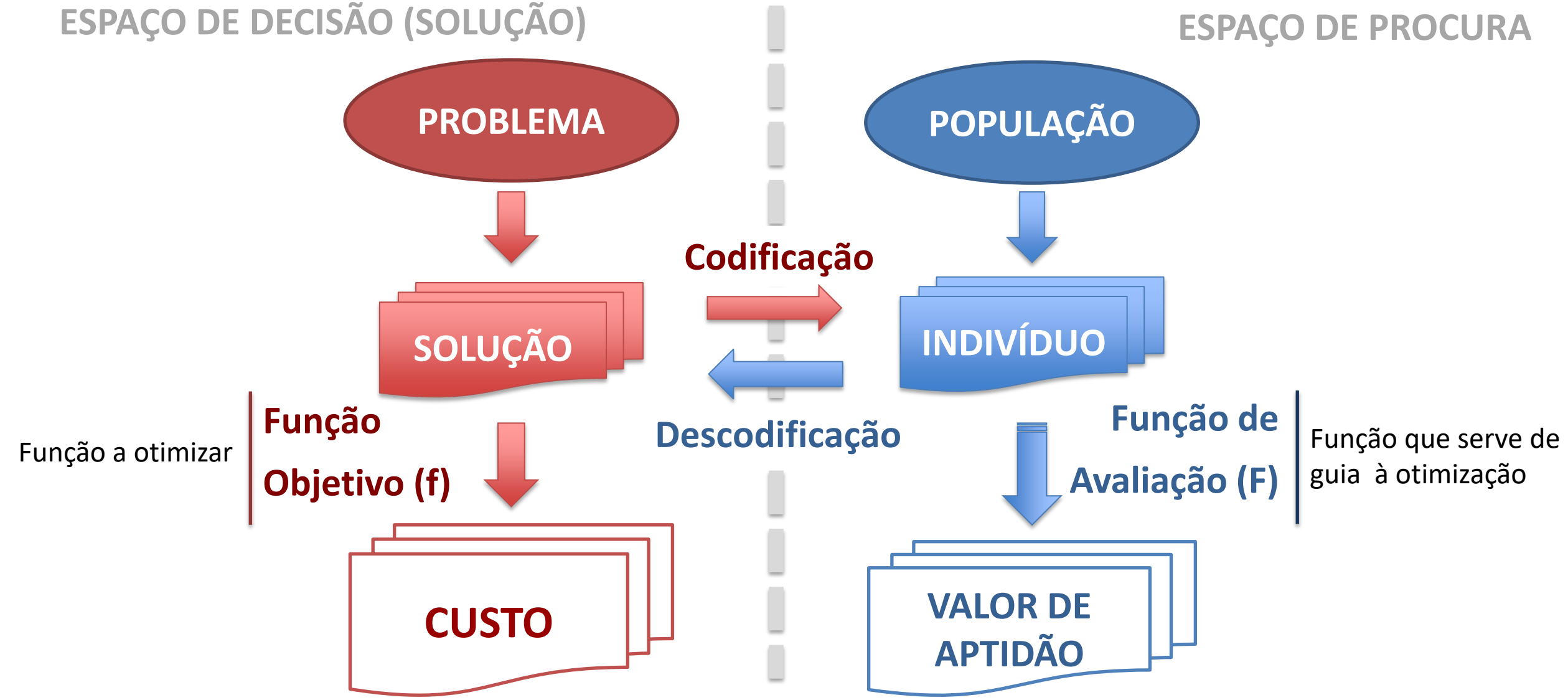
OC[C@@H](O1)[C@@H](O)[C@H](O)[C@@H](O)[C@@H](O)1

•Lista de inteiros usando uma gramática:

[75,11,123,11,19,37,63,82,12,67,33,41,2,16,59,23,54,19,55,12,13,12]

•Lista de reais usando usando um *Varational Auto Encoder*;
•Lista de inteiros representando um grafo,
•etc …

Codificação e Avaliação



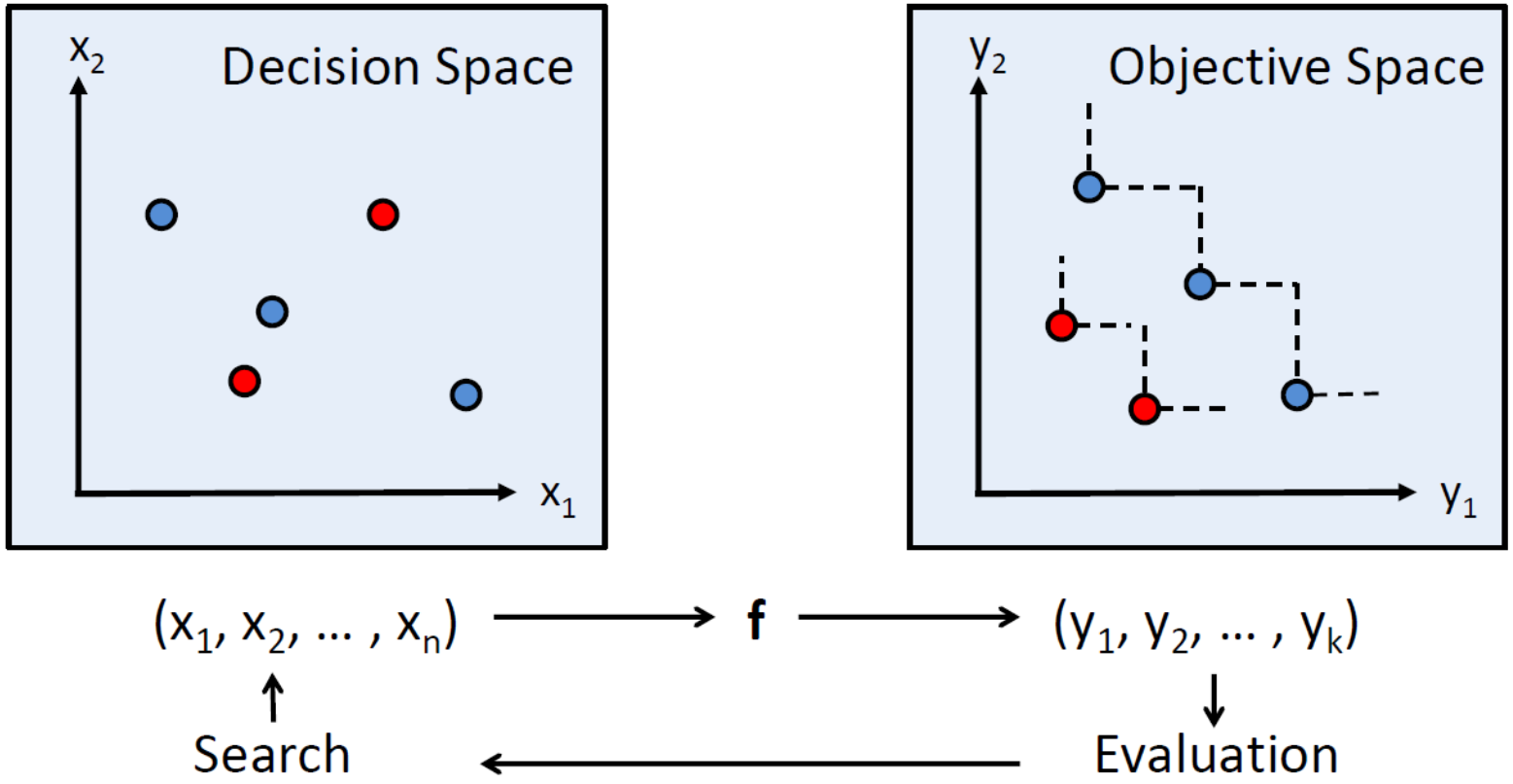
Função de avaliação

- Função que recebe como entrada o genoma de um indivíduo e retorna o seu valor de aptidão (valor real não negativo):

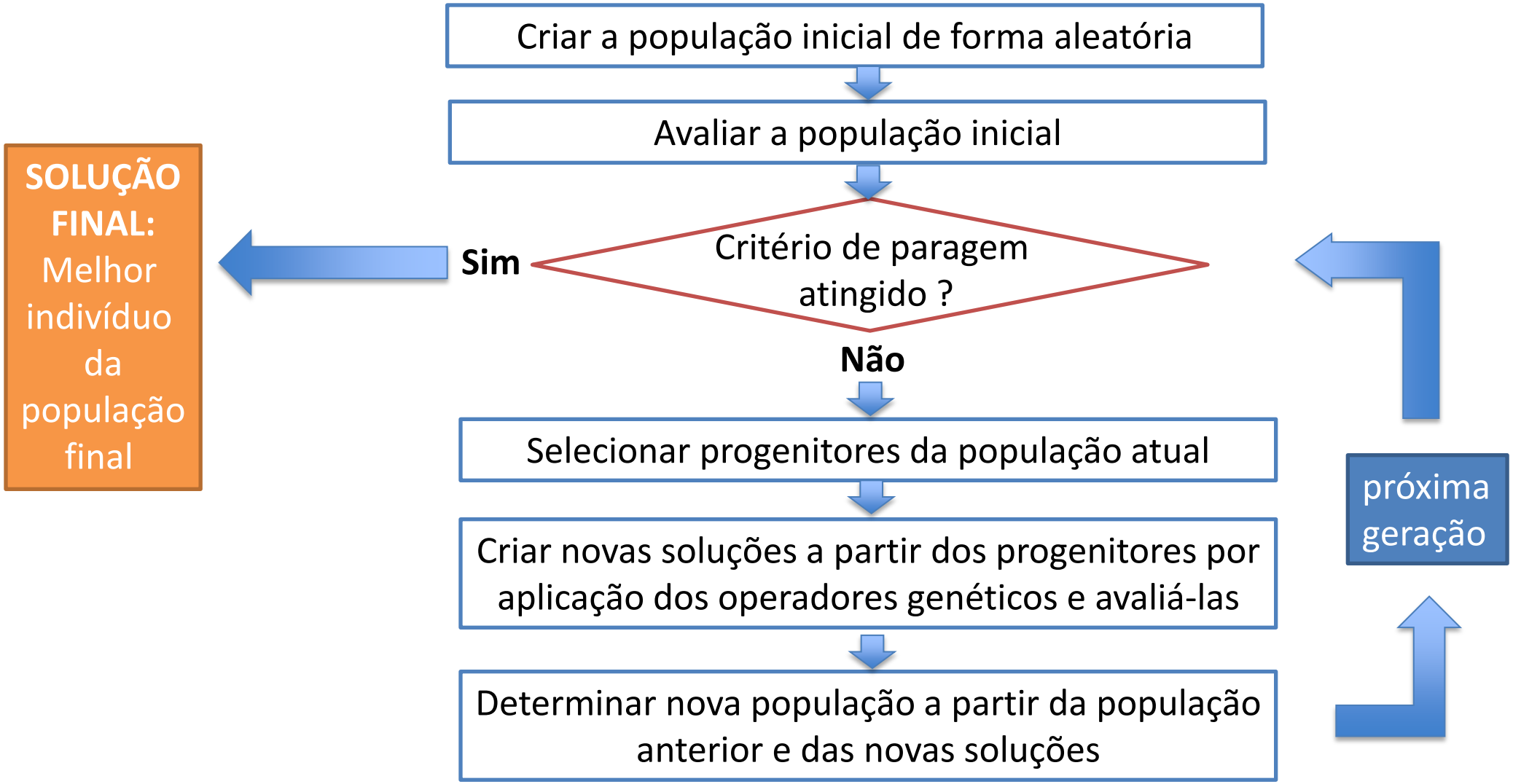
  F: *Indivíduo* $\rightarrow \mathbb{R}_0^+$

- **Valor de aptidão** de um indivíduo mede a qualidade da solução codificada por este na resolução do problema dado.
- Função de avaliação é dependente do problema a resolver: construída pelo processo de descodificação das soluções, seguida pelo cálculo do valor da função objetivo.
- Os Algoritmos Genéticos necessitam de uma função de avaliação para escalonar custos em valores reais positivos, que **permitam aplicar mecanismos de seleção**.
- São também utilizadas para penalizar indivíduos que representam **soluções inválidas**.

Espaços de Decisão e Objetivo



Estrutura de um algoritmo evolucionário

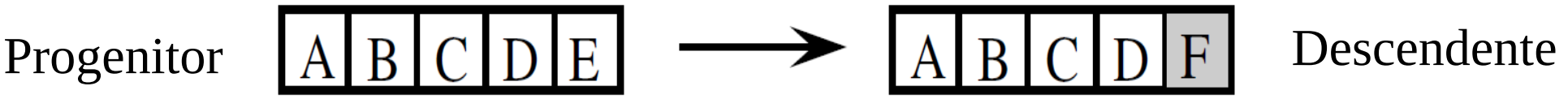


Operadores genéticos

- Necessário criar diversidade nas populações dos AEs – formas de criar novas soluções => **operadores genéticos**
- Matematicamente: funções que recebem um ou mais indivíduos e retornam um ou mais novos indivíduos:
  - ***Operador***: Indivíduo^{M} x Parâmetros $\rightarrow$ Individuo^{N}
- Operadores mais usados:
  - **mutação** (M=N=1)
  - **cruzamento** (M=N=2).

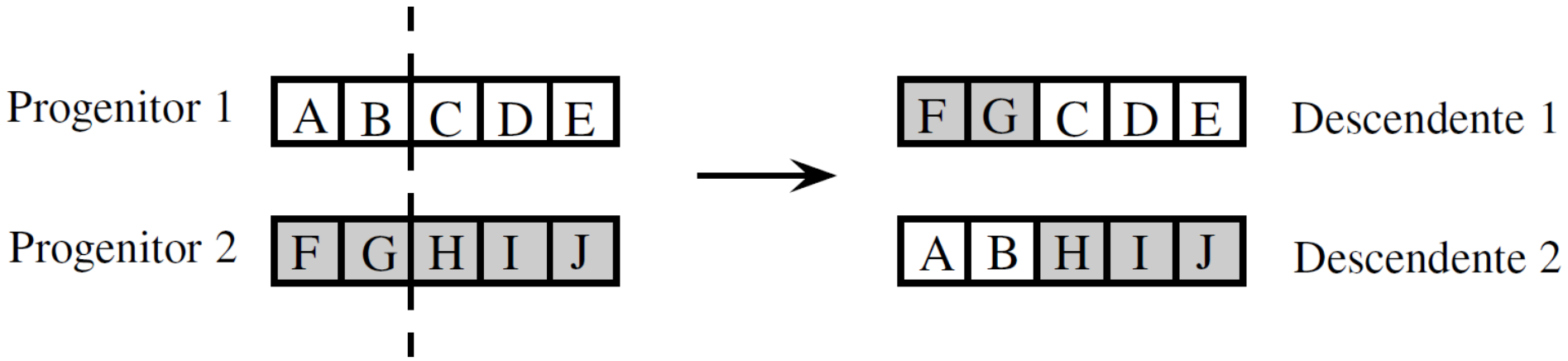
Mutação

- Novo indivíduo similar ao progenitor, com uma pequena alteração no seu genoma;
- Exemplo de **Mutação**



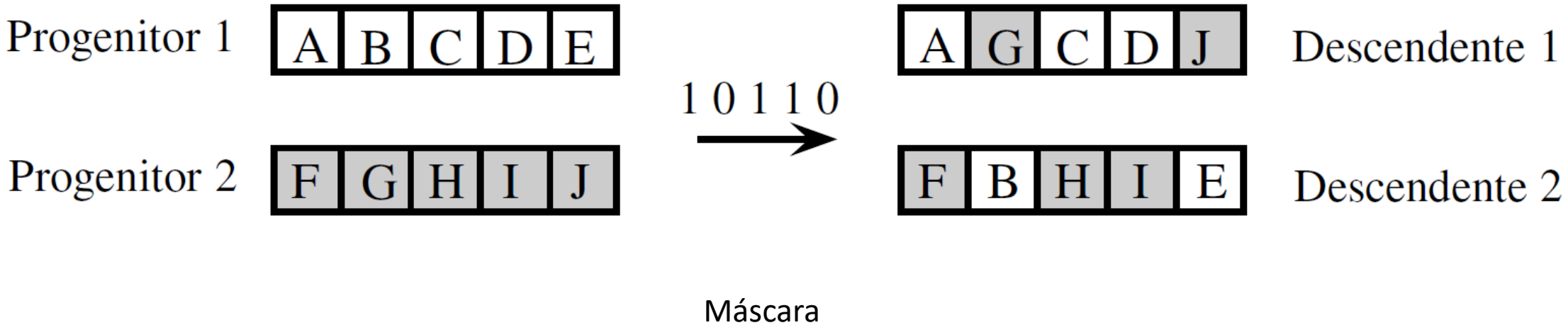
Cruzamento

- Progenitores recombinam a informação genética dos progenitores
- Exemplo: **Cruzamento de um ponto**
  - *swap* dos genomas de dois progenitores a partir de um ponto aleatório).



Cruzamento

- Exemplo: **Cruzamento uniforme**
  - A troca dos genes entre dois progenitores é efetuado de acordo com uma máscara aleatória

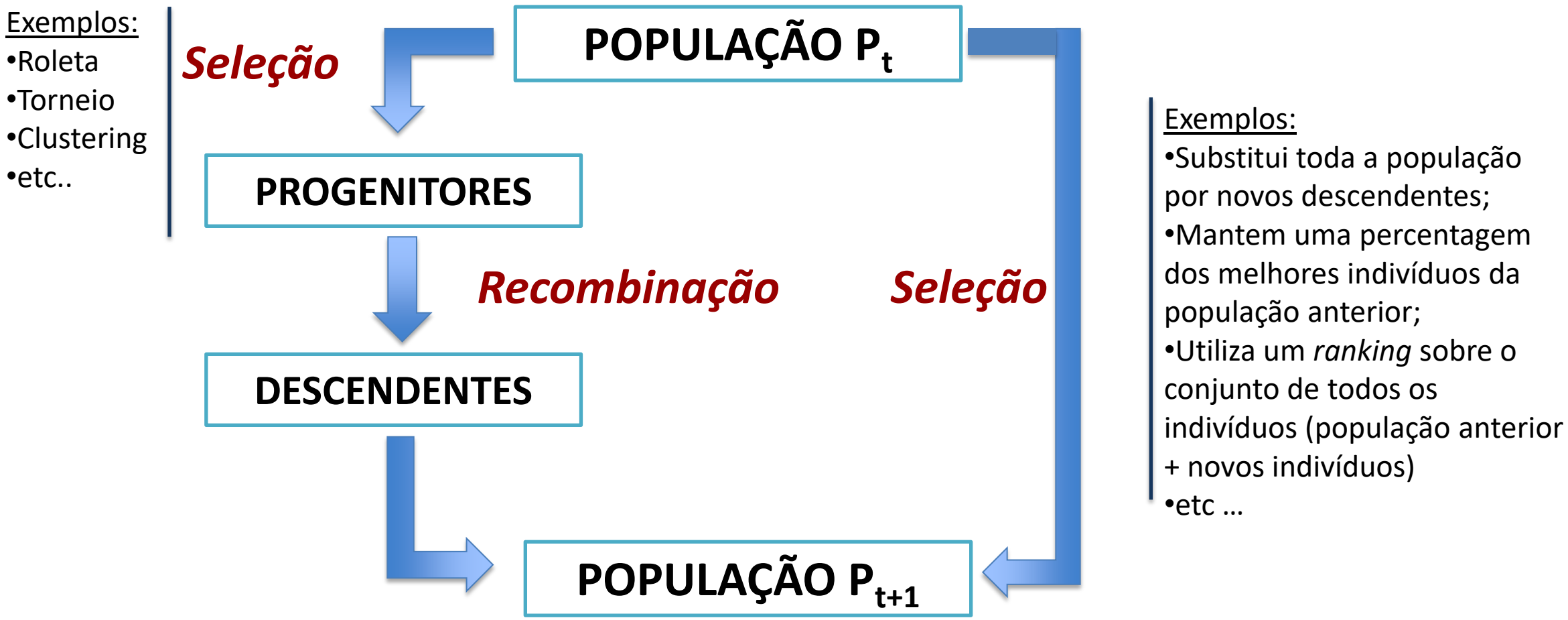


Seleção

- Processo de escolha de indivíduos da população para:
  - Serem **progenitores**, i.e., darem origem a novos indivíduos através da aplicação de operadores genéticos;
  - **Sobreviverem** de uma geração para a seguinte

- Processo de seleção é implícito na natureza.
- Nos Algoritmos Evolucionários, o processo baseia-se no valor de aptidão dos indivíduos – melhores indivíduos têm maior hipótese de serem selecionados, embora o processo seja **estocástico**.

Ciclo de vida de um algoritmo evolucionário: reinserção

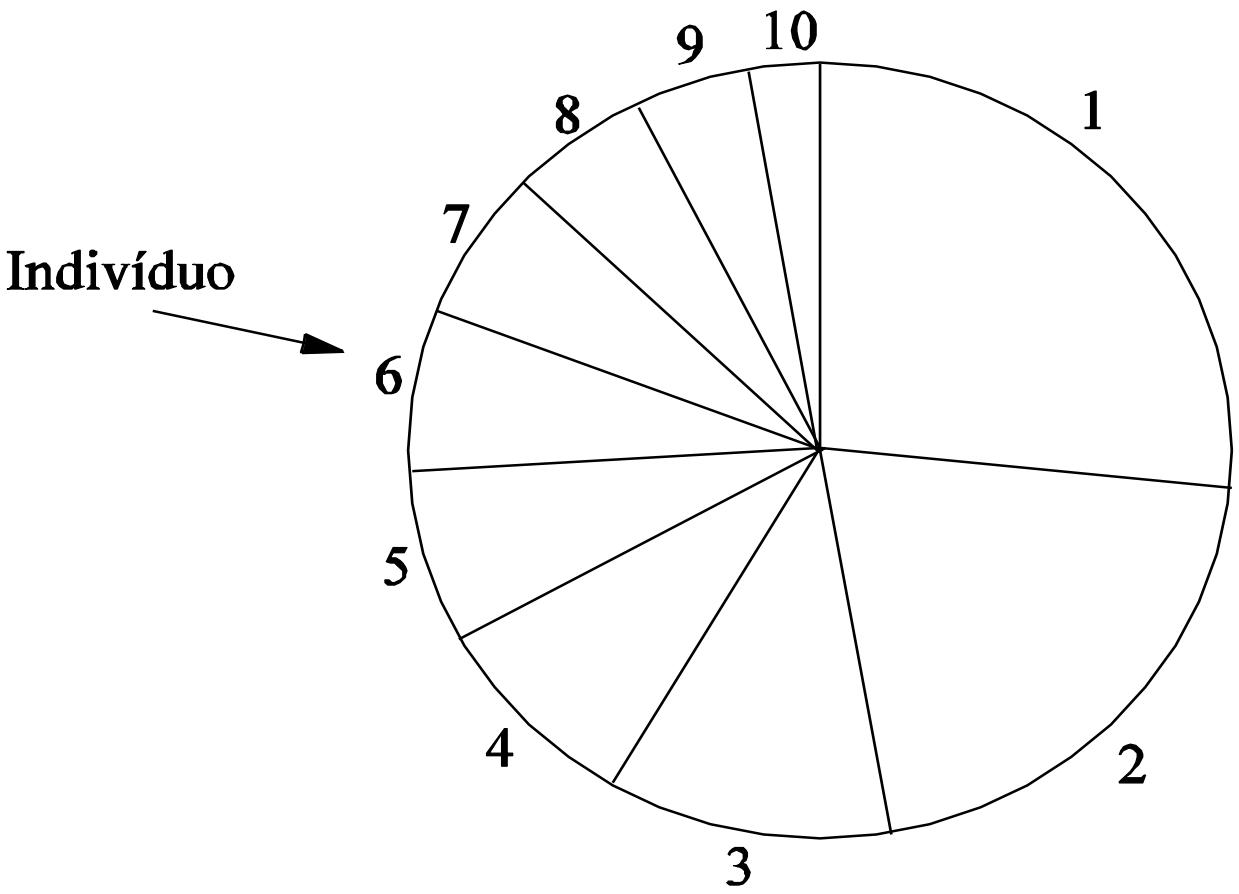


Parâmetros definem o nº de indivíduos que passa em cada ramo.

Métodos de seleção

- 1ª fase: **Conversão dos valores de aptidão**
  - Podem usar-se os valores de aptidão diretamente
  - Em muitos casos, tal opção leva a uma pressão de seleção baixa, dado o pequeno intervalo de valores
  - Outras opções: normalização de valores num intervalo fixo (e.g. [0,1]), ordenação (ranking)
- 2ª fase: **Amostragem**
  - escolha dos indivíduos a partir do valor anterior através de um método estocástico – e.g. roleta
  - Alternativa de seleção local: torneio, clustering

Seleção por roleta



Valor de Selecção	Indivíduo
5,3	1
4,4	2
2,1	3
1,5	4
1,4	5
1,4	6
1,3	7
1,3	8
0,8	9
0,5	10

Implementação de Algoritmos evolucionários

- Vamos criar um conjunto de classes python para implementar Algoritmos evolucionários (Genéticos)
- Numa primeira versão, vamos usar apenas representações binárias.
- Como 1º exemplo, vamos criar um AE para um problema óbvio: contar 1's, i.e., a função de avaliação será o nº de 1's no genoma, sendo a solução ótima um genoma só com 1's.
- Vamos implementar com base em 3 camadas: indivíduos, população, AE.

Implementação de Algoritmos Evolucionários - indivíduos

```
class Indiv:

    def __init__(self, size, genes=[], lb=0, ub=1):
        self.lb = lb
        self.ub = ub
        self.genes = genes
        self.fitness = None
        if not self.genes:
            self.initRandom(size)
```

Classe para implementar indivíduos com representações binárias

Atributos:

- ***genes*** – genoma
- ***fitness*** – guarda valor de aptidão
- ***lb/ub*** – limites inferior e superior do intervalo para representação de genes.

```
    def initRandom(self, size):
        self.genes = []
        for _ in range(size):
            self.genes.append(randint(self.lb, self.ub))
```

Inicialização aleatória

Implementação de Algoritmos Evolucionários – operadores genéticos

- Os operadores de mutação dependem da representação.

- Exemplo:
  Mutação para representações binárias que altera um único gene.

Progenitor: `[0,1,1,0,0,1,0,1]`

Descendente: `[0,1,1,0,1,1,0,1]`

```
def mutation(self):
        s = len(self.genes)
        pos = randint(0, s-1)
        if self.genes[pos] == 0:
            self.genes[pos] = 1
        else:
            self.genes[pos] = 0
```

Implementação de Algoritmos Evolucionários – operadores genéticos

- Cruzamento de um ponto

Progenitor 1

[0,1,1,0,0,1,0,1]

Progenitor 2

[1,0,0,0,1,1,1,0]

Descendente 1

[0,1,1,0,1,1,1,0]

Descendente 2

[1,0,0,0,0,1,0,1]

```
def crossover(self, indiv2):
    return self.one_pt_crossover(indiv2)

def one_pt_crossover(self, indiv2):
    offsp1 = []
    offsp2 = []

    s = len(self.genes)
    pos = randint(0, s-1)
    for i in range(pos):
        offsp1.append(self.genes[i])
        offsp2.append(indiv2.genes[i])
    for i in range(pos, s):
        offsp2.append(self.genes[i])
        offsp1.append(indiv2.genes[i])
    res1 = self.__class__(s, offsp1, self.lb, self.ub)
    res2 = self.__class__(s, offsp2, self.lb, self.ub)
    return res1, res2
```

Implementação de Algoritmos Evolucionários - população

```
class Popul:

    def __init__(self, popsize, indsize,indivs=[]):
        self.popsize = popsize
        self.indsize = indsize

        if indivs:
            self.indivs = indivs
        else:
            self.initRandomPop()

    def getIndiv(self, index):
        return self.indivs[index]

    def initRandomPop(self):
        self.indivs = []
        for _ in range(self.popsize):
            indiv_i = Indiv(self.indsize, [])
            self.indivs.append(indiv_i)
```

Classe para implementar **populações** de indivíduos com representações binárias

Atributos:

- ***popsize***– nº indivíduos da população
- ***indsize*** – tamanho dos indivíduos
- ***indivs***– indivíduos

Inicialização aleatória

Implementação de Algoritmos Evolucionários - população

```
    def getFitnesses(self, indivs=None):
        fitnesses = []
        if not indivs:
            indivs = self.indivs
        for ind in indivs:
            fitnesses.append(ind.getFitness())
        return fitnesses

    def bestSolution(self):
        return max(self.indivs)

    def bestFitness(self):
        indv = self.bestSolution()
        return indv.getFitness()
```

Funções de manipulação de **valores de aptidão**

Implementação de Algoritmos Evolucionários - população

Funções de **seleção**

```
def selection(self, n, indivs=None):
    res = []
    fitnesses = list(self.getFitnesses(indivs))
    for _ in range(n):
        sel = self.roulette(fitnesses)
        fitnesses[sel] = 0.0
        res.append(sel)
    return res

def roulette(self, f):
    tot = sum(f)
    val = random()
    acum = 0.0
    ind = 0
    while acum < val:
        acum += (f[ind] / tot)
        ind += 1
    return ind-1
```

```
fitnesses = list(self.linscaling(
                    self.getFitnesses(indivs)))
```

Normalização do valor de aptidão para [0,1]

```
def linscaling(self, fitnesses):
    mx = max(fitnesses)
    mn = min(fitnesses)
    res = []
    for f in fitnesses:
        val = (f-mn)/(mx-mn)
        res.append(val)
    return res
```

Implementação de Algoritmos Evolucionários - população

```
    def recombination(self, parents, noffspring):
        offspring = []
        new_inds = 0
        while new_inds < noffspring:
            parent1 = self.indivs[parents[new_inds]]
            parent2 = self.indivs[parents[new_inds+1]]
            offsp1, offsp2 = parent1.crossover(parent2)
            offsp1.mutation()
            offsp2.mutation()
            offspring.append(offsp1)
            offspring.append(offsp2)
            new_inds += 2
        return offspring
```

Função de **recombinação**

Usa **cruzamento** para criar novas soluções

Aplica **mutação** a cada nova solução

Implementação de Algoritmos Evolucionários - população

```
    def reinsertion(self, offspring):
        tokeep = self.selection(self.popsize-len(offspring))
        ind_offsp = 0
        for i in range(self.popsize):
            if i not in tokeep:
                self.indivs[i] = offspring[ind_offsp]
                ind_offsp += 1
```

Função de **reinserção**

Implementação de Algoritmos Evolucionários - AE

```
class EvolAlgorithm:

    def __init__(self, popsize, numits,
                 noffspring, indsize):
        self.popsize = popsize
        self.numits = numits
        self.noffspring = noffspring
        self.indsize = indsize

    def initPopul(self, indsize):
        self.popul = Popul(self.popsize, indsize)
```

Classe para implementar **algoritmos evolucionários** com representações binárias

Implementação de Algoritmos Evolucionários - AE

```
    def evaluate(self, indivs):
        for i in range(len(indivs)):
            ind = indivs[i]
            fit = 0.0
            for x in ind.getGenes():
                if x == 1:
                    fit += 1.0
            ind.setFitness(fit)
        return None
```

Classe para implementar **algoritmos evolucionários** com representações binárias

Função de avaliação para o problema de contar 1’s

Implementação de Algoritmos Evolucionários - AE

```
    def iteration(self):
        parents = self.popul.selection(self.noffspring)
        offspring = self.popul.recombination(parents, self.noffspring)
        self.evaluate(offspring)
        self.popul.reinsertion(offspring)

    def run(self):
        self.initPopul(self.indsize)
        self.evaluate(self.popul.indivs)
        self.bestsol = self.popul.bestSolution()
        for i in range(self.numits+1):
            self.iteration()
            bs = self.popul.bestSolution()
            if bs > self.bestsol:
                self.bestsol = bs
            print("Iteration:", i, " ", "Best: ", self.bestsol )
```

Classe para implementar **algoritmos evolucionários** com representações binárias

Ciclo principal do AE

Procura de motifs: algoritmos evolucionários

Algoritmos evolucionários

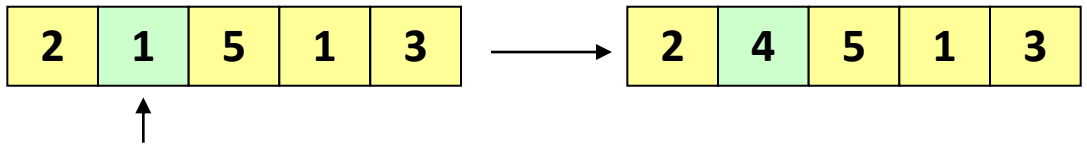
- Uma alternativa aos algoritmos já estudados para inferência de motifs passa pela utilização de algoritmos evolucionários para a resolução do problema
- Neste caso, é necessário representar soluções para o problema em alfabetos que possam ser usados em algoritmos evolucionários

Representações discretas/ inteiras

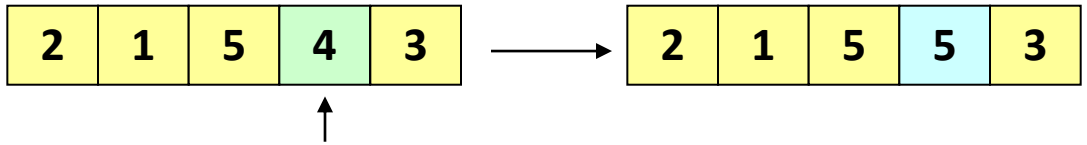
- Representação linear com N valores **inteiros** em cada gene
- Em cada posição, valor poderá estar entre 0 e M
- Pode ser usado no caso dos motifs para representar diretamente a solução: cada gene é a posição inicial do motif numa sequência (indivíduos têm tamanho igual ao nº de sequências)
- Operadores genéticos de cruzamento semelhantes aos usados para qualquer representação linear (e.g. binárias)
- Operadores de mutação distintos

Operadores genéticos: representações inteiras

Mutação aleatória

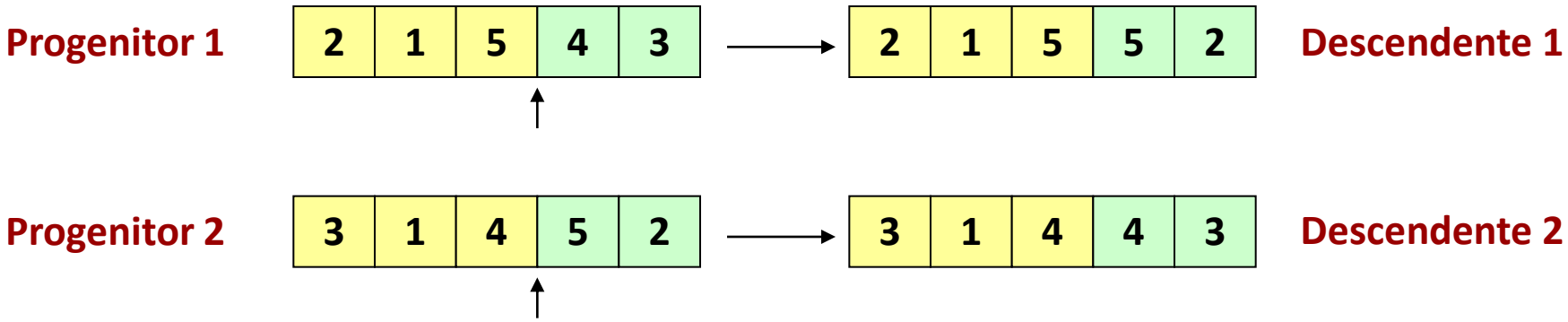


Mutação "elemento seguinte"



Operadores genéticos: representações inteiras

Cruzamento 1 ponto



Implementação do algoritmo evolucionário

- Vamos implementar, em primeiro lugar, representações inteiras
  - Estendendo a classe *Indiv* com uma nova classe *IndivInt*
  - Estendendo a classe *Popul* com uma nova classe *PopulInt*
- Em seguida, iremos implementar uma classe estendendo *EvolAlgorithm* onde definiremos a função de avaliação para este problema

Implementação do algoritmo evolucionários: indivíduos

```
class IndivInt (Indiv):

    def initRandom(self, size):
        self.genes = []
        for _ in range(size):
            self.genes.append(randint(0, self.ub))

    def mutation(self):
        s = len(self.genes)
        pos = randint(0, s-1)
        self.genes[pos] = randint(0, self.ub)
```

A class ***IndivInt*** estende a class ***Indiv*** herdando todos os seus métodos.

Só é necessário implementar o método *mutation* (override) pois os genes assumem valores em intervalos diferentes dos definidos em ***Indiv***.

Implementação do algoritmo evolucionários: população

```
class PopulInt(Popul):

    def __init__(self, popsize, indsize, ub, indivs=[]):
        self.ub = ub
        Popul.__init__(self, popsize, indsize, indivs)

    def initRandomPop(self):
        self.indivs = []
        for _ in range(self.popsize):
            indiv_i = IndivInt(self.indsize, [], 0, self.ub)
            self.indivs.append(indiv_i)
```

Estende a class ***Popul*** herdando todos os seus métodos.

Só é necessário implementar o método *initRandomPop* (override) para usar representações inteiras.

Implementação do AE

```
class EAMotifsInt (EvolAlgorithm):
    def __init__(self, popsize, numits, noffspring, filename):
        self.motifs = MotifFinding()
        self.motifs.readFile(filename, "dna")
        indsize = len(self.motifs)
        EvolAlgorithm.__init__(self, popsize, numits, noffspring, indsize
)

    def initPopul(self, indsize):
        maxvalue = self.motifs.seqSize(0) - self.motifs.motifSize
        self.popul = PopulInt(self.popsize, indsize,
                              maxvalue, [])

    def evaluate(self, indivs):
        for i in range(len(indivs)):
            ind = indivs[i]
            sol = ind.getGenes()
            fit = self.motifs.score(sol)
            ind.setFitness(fit)
```

As soluções são vetores de posições iniciais representadas como lista de inteiros (*IndivInt*)

Cada indivíduo é avaliado calculando o score aditivo do alinhamento definido pelas posições iniciais.

Procura-se o encontrar o vetor de posições iniciais com o melhor score (fitness).

Representação real

- Uma alternativa para os Algoritmos Evolucionários passa pelo uso de uma **representação real**, onde se representa diretamente o perfil (a PWM)
- Neste tipo de representação, os indivíduos são constituídos por uma lista de nºs reais, entre um limite inferior e superior
- Tal como nas representações inteiras, os operadores de cruzamento podem ser os mesmos das outras representações lineares, havendo necessidade de definir novos operadores de mutação

Exercício

- Deverá implementar, em primeiro lugar, representações reais
  - Estendendo a classe *Indiv* com uma nova classe *IndivReal,* onde deve redefinir o construtor os métodos *initRandom* e *mutation* (a mutação poderá ser substituir um gene selecionado aleatoriamente por um valor aleatório no intervalo admissível)
  - Estendendo a classe *Popul* com uma nova classe *PopulReal,* onde deve redefinir o construtor e o método *initRandomPop.* Note que neste caso, deve ter dois atributos na população que guardam o limite inferior e superior dos valores admissíveis; neste problema poderá usar 0 e 1

Exercício

- Em seguida deverá criar uma classe que estende a classe *EvolAlgorithm* e que redefine o construtor, o método *initPopul* e o método *evaluate*
- Note que, neste caso, o tamanho do indivíduo deverá ser: L x A, onde L é o tamanho do motif e A é o nº de símbolos do alfabeto
- No método *evaluate* deverá, para cada indivíduo:
  - Construir uma matriz de dimensão A x L (a PWM)
  - Normalizar a matriz para que a soma de cada coluna seja 1
  - Determinar a posição mais provável do motif representado por essa PWM em cada sequência, construindo o vetor ***s***
  - Calcular o score da solução ***s*** construída no passo anterior (este será a fitness)