Projeto de Computação Evolutiva
k–partições perfeitas

Eduardo Guerreiro, 107036
Filipe Viseu, 107091
Mariana Costa, 107094
Tiago Mourão, 107096

Professor	*Carlos Caleiro*
Disciplina:	Elementos de programação
Data:	Novembro 2022

Índice

1 Introdução

Uma $k-$partição perfeita de uma lista w de inteiros positivos é uma divisão da lista em k sublistas disjuntas de tal forma que a soma de cada sublista é igual.É possível avaliar o quão próxima uma $k-$partição está de uma $k-$partição perfeita pelo quão próximo de zero é o seu *coeficiente de inadaptação*. Se definirmos $S(\ell)$ como a soma de uma lista ℓ, o coeficiente de uma k–partição $x_1, x_2, \ldots, x_k$ de w é dado por

$$\frac{\sum_{i=1}^{k} \left| S(x_i) - S(w)/k \right|}{k}.$$

Determinar se w admite uma k–partição perfeita é um problema muito difícil, não existindo nenhum algoritmo eficiente para o resolver. Conjectura-se que não pode ser resolvido em tempo polinomial.

Neste projeto, abordamos o problema das k-partições perfeitas através do paradigma de computação evolutiva. Desenvolvemos uma simulação discreta estocástica que simula uma população de indivíduos, cada um associado a uma $k-$partição de w, que evolui de forma a encontrar uma solução para o problema pretendido. Para tal, através de três acontecimentos com tempos aleatórios (mutação, reprodução e morte), os indivíduos e a população são alterados com o objetivo de melhorar os coeficientes de inadaptação das partições, aproximando-as assim de uma partição perfeita. Se ao longo da simulação for encontrada uma tal partição, a simulação é interrompida e esta partição é devolvida; caso contrário, no final do programa são devolvidas as partições com menor coeficiente de inadaptação.

O projeto foi desenvolvido de acordo com o método de programação modular, por camadas, sempre respeitando a independência entre o Tipo de Dados e a Aplicação. As classes e os métodos criados são explicados abaixo, bem como o simulador. No final do relatório são discutidos os resultados obtidos e apresentadas algumas considerações finais.

2 Tipos de Dados

2.1 Partição

A classe `particao` representa uma $k-$partição da lista original w e é munida dos métodos:

- `__init__()` – inicializa a partição;
- `generate(k, w)` – devolve uma $k-$partição aleatória da lista w;
- `troca(i, b)` – passa o elemento na $i-$ésima posicão de w para o bloco b e devolve a nova particao;
- `elem_ordem(x)` – retorna o bloco a que pertence o x-ésimo elemento de w;
- `soma_blocos(w, k)` – retorna uma lista cuja $i-$ésima posição é a soma dos elementos do $(i+1)-$ésimo bloco da partição;
- `coef(w, k)` – retorna o coeficiente de inadaptação da partição;
- `view(w, k)` – retorna a lista das sublistas formadas pelos elementos de cada bloco da partição.

2.2 Indivíduo

A classe `indv` representa um indivíduo e é munida dos métodos:

- `__init__(k, w, part, tempo, listamagica, nif)` – inicializa um indivíduo com uma partição associada, uma lista mágica e um número nif identificador do indíviduo;
- `getnif()` – retorna o nif do indivíduo;
- `getparticao()` – retorna a partição associada ao indivíduo;
- `create_listamagica(w, k, tempo)` – retorna uma lista cujo $i-$ésimo elemento é o `tempo` se o $i-$ésimo bloco for perfeito, e -1, caso contrário;
- `update_listamagica(w, k, tempo)` – retorna a lista mágica atualizada com o tempo de criação dos novos blocos perfeitos;
- `getlista_magica()` – retorna a lista mágica associada ao indivíduo.

2.3 População

A classe populacao representa um conjunto de Indivíduos e é munida dos métodos:

- `__init__()` – inicializa a população vazia;
- `getindvlist()` – retorna a lista dos indivíduos da população;
- `getmaxnif()` – retorna o maior nif da população;
- `getniflist()` – retorna a lista dos nif's de todos os indíviduos vivos da população;
- `add(indv)` – adiciona o indivíduo `indv` à população e retorna a nova população;
- `mata(indv)` – remove o indivíduo `indv` da população e retorna a nova população;
- `substitui(new, old)` – substitui o indivíduo `old` pelo índividuo `new` na população e retorna a nova população;
- `procura(indv)` – retorna o índice do indivíduo, usando pesquisa binária; `indv` da população;
- `vivoQ(nif)` – retorna `True` se existir na população um indíviduo com nif igual ao argumento e `False` caso contrário;

- `vazioQ()` – retorna `True` se a população estiver vazia e `False` caso contrário.

2.4 Evento

A classe `evento` representa um acontecimento e é munida dos métodos:

- `__init__(tipo,alvo,time)` – inicializa o evento do tipo `tipo`, sobre o indíviduo `alvo`, no tempo `time`;
- `getime()` – retorna o tempo a que se dará o evento;
- `getalvo()` – retorna o indivíduo sobre o qual se dará o evento;
- `gettipo()` – retorna o tipo de evento;

2.5 CAP

A classe `cap` representa uma agenda dos eventos pendentes e é munida dos métodos:

- `__init__()` – inicializa a cap vazia;
- `proximo()` – retorna o primeiro evento na cap;
- `apaga()` – apaga o primeiro evento da cap e devolve a nova cap;
- `view()` – permite saber a informação guardada na cap;
- `adicionar(evento)` – adiciona o evento `evento` à cap.

2.6 Classes Alternativas

De forma a testar a independência entre o tipo de dados e a aplicação, criámos implementações alternativas dos tipos de dados (cuja implementação era a nosso critério). Para as classes alternativas, optámos por utilizar dicionários para as estruturas de dados. Estas implementações quando aplicadas ao simulador obtiveram resultados positivos. Ver:

- `individuo_alt.py`
- `populacao_alt.py`
- `evento_alt.py`
- `cap_alt.py`

3 Aplicação

Resumimos neste capítulo o funcionamento do simulador construído.

3.1 Funções Auxiliares

Na construção do simulador definimos três funções auxiliares que facilitam a leitura do algoritmo implementado, bem como a sua explicação.

3.1.1 Exprandom

Definimos a função auxiliar `exprandom(c)` que permitirá gerar aleatoriamente números de valor exponencial médio c:

```
def exprandom(c):
    return -c*log(random())
```

3.1.2 Mutação

A função `mutacao` é responsável pela mutação de uma partição `part`.

```
def mutacao(w,k,part):

    soma_e_pos = [[0] for i in range(k)]

    for j in range(len(w)):
        soma_e_pos[part.elem_ordem(j)-1][0] += w[j]
        soma_e_pos[part.elem_ordem(j)-1] += [j]
```

O programa constrói a lista `soma_e_pos` com k sublistas. Cada sublista guarda informação sobre um dos k blocos de w: nomeadamente, `soma_e_pos`$[i-1]$ contém na posição 0 a soma de todos os elementos do bloco i e nas posições seguintes contem a posição dos elementos de w que pertencem ao bloco i.

```
    perf = sum(w)//k

    lista_sup = []
    lista_inf = []

    for x in range(k):
        if soma_e_pos[x][0] > perf:
            lista_sup.append(x)
        elif soma_e_pos[x][0] < perf:
            lista_inf.append(x)
```

As sublistas são filtradas de acordo com a soma dos elementos do bloco correspondente: se a soma for superior/inferior à de um bloco perfeito, é colocado o índice dessa sublista na lista `lista_sup`/`lista_inf` respetivamente. Como a partição `part` não é uma partição perfeita, obrigatoriamente existe um bloco com soma superior e outro com soma inferior à de um bloco perfeito.

```
    nb_maior = lista_sup[randint(0,len(lista_sup)-1)]
    nb_menor = lista_inf[randint(0,len(lista_inf)-1)]
```

```
    bloco_menor = soma_e_pos[nb_menor]
    x = soma_e_pos[nb_maior][randint(1,len(soma_e_pos[nb_maior])-1)]

    targetsum = w[x] - (soma_e_pos[nb_maior][0] - bloco_menor.pop(0))/2

    soma = 0
```

De seguida, o programa cria as variáveis:

- `nb_maior` e `nb_menor` são os índices das sublistas aleatoriamente escolhidas da `lista_sup` e da `lista_inf` respetivamente;
- `bloco_menor` é a lista das posições dos elementos de w pertencentes ao bloco com soma inferior escolhido;
- x é a posição do elemento do bloco com soma superior escolhido;
- `targetsum`, é a quantia definida por

$$\texttt{targetsum} = x - \frac{\text{diferença da soma dos dois blocos}}{2};$$

- `soma` é a soma dos elementos escolhidos do `bloco_menor`. Inicialmente, como ainda não foram escolhidos elementos, este valor começa em 0.

```
    while soma < targetsum and bloco_menor != []:
        proximo = bloco_menor.pop(randint(0, len(bloco_menor) - 1))
        soma += w[proximo]
        part.troca(proximo, nb_maior + 1)
    part.troca(x, nb_menor + 1)

    return part
```

Enquanto `soma` $<$ `targetsum` e o bloco menor ainda tiver elementos, o programa escolhe um elemento aleatório do `bloco_menor`, adiciona-o à soma e coloca-o no bloco maior. Por fim, x é colocado no `bloco_menor` e a função retorna a partição mutada.

3.1.3 Reprodução

A função `reproducao` recebe como atrgumento, para além de w e k, os indivíduos `mae` e `pai`.

```
def reproducao(mae, pai, w, k):

    particao_mae = mae.getparticao()
    particao_pai = pai.getparticao()

    magica_mae = mae.getlista_magica()
    magica_pai = pai.getlista_magica()

    if any(map(lambda x: x != -1, magica_pai)):

        filho = deepcopy(particao_mae)

        heranca = deepcopy(magica_mae)

        resto = []
```

```
        for i in range(len(w)):
            if magica_mae[particao_mae.elem_ordem(i) - 1] == -1:
                resto.append([w[i], i])
```

Verifica se o `pai` tem blocos perfeitos; se não tiver a reprodução não produz efeito.

O programa define as variáveis

- `filho`, que no final será a partição do indivíduo resultante da reprodução, começa por ser uma partição igual à da mãe;
- `heranca` será a lista mágica do indivíduo resultante da reprodução. Esta lista começa por ser uma lista igual à lista mágica da mãe, e será apenas atualizado a entrada correspondente ao bloco que o filho vai herdar do pai. A possibilidade de aparecerem blocos perfeitos no final da reprodução não causa problemas dado que a lista mágica do filho vai ser atualizada quando o indivíduo for criado;
- `resto`, que começa como uma lista vazia, e será uma lista de listas em que primeiro elemento de cada lista corresponde a cada elemento de w que não pertence a um bloco perfeito da mãe e o segundo elemento corresponde à sua posição em w.

De seguida, para cada posição i, se o i-ésimo elemento de w não estiver num bloco perfeito da `mae`, é adicionado a `resto` a lista $[w[i], i]$.

```
        blocos_perf_pai = []
        blocos_por_usar = []

        for x in range(1, k+1):
            if magica_pai[x-1] != -1:
                blocos_perf_pai.append(x)

            if magica_mae[x-1] == -1:
                blocos_por_usar.append(x)

        perf_pai = blocos_perf_pai[randint(0,len(blocos_perf_pai)-1)]
        bloco_pai = [w[i] for i in range(len(w)) if particao_pai.elem_ordem
    (i) == perf_pai]

        bloco_to_fill = blocos_por_usar.pop()
```

O programa cria as listas `bloco_perf_pai` e `blocos_por_usar`, que são a lista dos blocos perfeitos do pai e a lista dos blocos ainda vazios do filho e, ainda, a variável `perf_pai`, que é o número do bloco perfeito do pai a herdar pelo filho, a variável `bloco_pai`, que é esse mesmo bloco perfeito, e a variável `bloco_to_fill`, que corresponde ao bloco do `filho` que queremos preencher com os elementos de `bloco_pai`.

```
        possivel = True
        i = 0

        while possivel and i<len(bloco_pai):
            j = 0
            while j < len(resto) and resto[j][0] != bloco_pai[i]:
                j += 1
```

```
            if j == len(resto):
                possivel = False
            else:
                filho.troca(resto[j][1], bloco_to_fill)
                resto = resto[:j] + resto[j+1:]
                i += 1
```

Para cada elemento de `bloco_pai`, procura-se um índice $j <$ `len(resto)` tal que a primeira entrada do j–ésimo elemento de `resto` é igual ao elemento de `bloco_pai` em questão. Se tal índice for encontrado, coloca-se o j–ésimo elemento de w no bloco `bloco_to_fill` do `filho` e retira-se o j–ésimo elemento de `resto`, para que este não volte a ser utilizado. Se tal índice não for encontrado, a reprodução não produz efeito.

```
        if possivel:
            heranca[bloco_to_fill - 1] = magica_pai[perf_pai - 1]
            for s in range(len(resto)):
                filho.troca(resto[s][1], blocos_por_usar[randint(0, len(
   blocos_por_usar)-1)])
            return (True, filho, heranca)

    return (False, False, False)
```

Se o `filho` ainda tiver blocos vazios, os elementos que ainda não estão em nenhum bloco são distribuídos aleatoria e uniformemente pelos blocos vazios do `filho`.

3.2 Simulador

Para a construção do simulador importámos os módulos:

```
from random import random, randint
from math import log, inf
import copy
import time

from funcoes import mutacao, reproducao
from partition import particao
from individuo import indv
from evento import evento
from populacao import populacao
from cap import cap
```

sobres os quais é construída a aplicação.

O simulador é inicializado ao receber como *input*, por ordem: o número de indivíduos, o tempo máximo que o simulador correrá (se não encontrar uma solução), os parâmetros médios para a mutação, reprodução e morte de indíviduos, a lista original w e o parâmetro k.

É definido `agora` $= 0$ e é criada uma população onde são adicionados `NInd` indivíduos. Cada indivíduo, além da variável k e da lista w, guarda uma $k-$partição da lista w, o instante em que foi criado, uma lista vazia (que mais tarde conterá informação sobre os tempos de criação dos seus blocos perfeitos) e um número entre 1 e `NInd` identificador do indíviduo. A variável `agora` representa o "tempo" do simulador.

```
def simulador(NInd, TFim, TMut, TRep, TMor, w, k):
    start = time.time()
    population = populacao()
    agora = 0
    ganhamos = False

    for i in range(1,NInd+1):
        population.add(indv(k, w,new_part.generate(k,w), agora, [], i))
```

A agenda é inicializada e, para cada indivíduo adicionado à população, é verificado se guarda uma k–partição perfeita e agendada a primeira mutação. São agendados os eventos reprodução e morte, atualizado o `evento_atual` e o instante `agora` .

```
  agenda = cap()
  for individuo in population.getindvlist():
        agenda.adicionar(evento("mut", individuo, exprandom(TMut)))

        if individuo.coef(w,k) == 0:
            sol = [individuo.getparticao()]
            ganhamos = True

    agenda.adicionar(evento("rep",[],exprandom(TRep)))
    agenda.adicionar(evento("mor",[], exprandom(TMor)))

    evento_atual = agenda.proximo()
    agora = evento_atual.gettime()
```

Caso não seja encontrada uma k–partição perfeita o simulador prossegue iterativamente até atingir o tempo `TFim` ou até ser encontrada uma solução (`ganhamos == True`). O simulador executa o evento, dependendo do tipo, após isso descarta-o da agenda, executa o próximo evento agendando e atualiza o tempo `agora`. No caso do evento em causa ser uma mutação:

```
if tipo == "mut" and sum(w)%k==0:

            ind = evento_atual.getalvo()
            if population.vivoQ(ind.getnif()):

                particao_indv = ind.getparticao()
                magica = ind.getlista_magica()

                new_part = mutacao(w,k,particao_indv)


                new = indv(k, w, new_part, agora, magica, ind.getnif())
                population.substitui(new,ind)
                agenda.adicionar(evento("mut", new, agora + exprandom(TMut)
   ))

                if new.getparticao().coef(w, k)==0:
                    sol = [new.getparticao()]
                    ganhamos = True
```

é selecionado o indivíduo sobre o qual atua a mutação, verificando-se, pelo `nif`, se está vivo. Com auxílio da função `mutacao`, o indivíduo é mutado. O "indíviduo antigo" é substituído na população pelo indivíduo mutado, o programa agenda uma nova mutação e verifica-se se o indivíduo mutado representa um k–partição perfeita.

Se o evento agendado for uma reprodução:

```
elif tipo == "rep" and sum(w)%k==0 and len(population.getindvlist())>1:

    lista_ind = population.getindvlist()
    mae = lista_ind[randint(0, len(lista_ind)-1)]
    pai = lista_ind[randint(0, len(lista_ind)-1)]
    while mae.getnif() == pai.getnif():
        mae = lista_ind[randint(0, len(lista_ind)-1)]
        pai = lista_ind[randint(0, len(lista_ind)-1)]

    (success,filho,heranca) = reproducao(mae,pai,w,k)

    if success:
        result = indv(k, w, filho, agora, heranca, population.getmaxnif()
   +1)
        population.add(result)
        agenda.adicionar(evento("mut", result, agora + exprandom(TMut)))

        if filho.coef(w, k)==0:
            sol = [result.getparticao()]
            ganhamos = True
        agenda.adicionar(evento("rep", [], agora + exprandom(TRep)))
```

Analogamente, o programa escolhe de modo aleatório uma `mae` e `pai`, garantido que não são o mesmo indivíduo. Tendo dois indivíduos diferentes, a função `reproducao` devolve o triplo `(sucess, filho, herança)`. Se for bem sucedida, o `filho` é incluído na população e tal como acima, é agendanda uma nova mutação para este indivíduo e verificado se representa uma k–partição perfeita. Por fim, se se tratar de uma morte:

```
   else: #(tipo == "mor")
            for individuo in population.getindvlist():
                tempos = [x for x in individuo.getlista_magica() if x!=-1]

                if tempos != []:

                  maximo = tempos[0]
                  minimo = tempos[0]
                  for x in range(1,len(tempos)):
                    if x > maximo:
                      maximo=x
                    if x < minimo:
                      minimo=x

                  idade_recente = agora - maximo
                  formacao_antigo = minimo

                  if idade_recente > 2*formacao_antigo:
                   population.mata(individuo)
```

Para cada indíviduo, a função `morte`, com um ciclo `for`, encontra o máximo e o mínimo dos tempos de criação dos seus blocos perfeitos, e compara-os, eliminando o indivíduo caso o bloco perfeito mais recente tenha uma idade superior ao dobro do instante de formação do seu bloco perfeito mais antigo. Caso, no fim do processo, a população fique vazia, é repovoada com `NInd` indivíduos, são agendadas mutações para cada um, e é verificado por meio do coeficiente de inadaptação se algum deles possui uma k-partição perfeita. Tal como no começo,

são agendadas mutações, reprodução e morte.

```
   if population.vazioQ():
            for i in range(1,NInd+1):
                 individuo = indv(k, w,  particao().generate(k,w),
agora, [], i)
                 population.add(individuo)
                 agenda.adicionar(evento("mut", individuo, agora+
exprandom(TMut)))

                if individuo.getparticao().coef(w,k) == 0:
                    sol = [individuo.getparticao()]
                    ganhamos = True

            agenda.adicionar(evento("rep",[],agora + exprandom(TRep)))
        agenda.adicionar(evento("mor",population, agora + exprandom(
TMor)))
```

Se entretanto for encontrada uma solução exata, o ciclo `while` é interrompido e a solução é apresentada; caso contrário, são calculados os coeficientes de inadaptação e apresentadas as partições que possuem o menor coeficiente de inadaptação.

```
if ganhamos == False:
    coefs = [ind.getparticao().coef(w,k) for ind in population.
getindvlist()]
    minimo = min(coefs)
    sol=[]
    for ind in population.getindvlist():
        if ind.getparticao().coef(w,k)== minimo:
            sol += [ind.getparticao()]

end = time.time()

if len(sol)==1:
    return sol[0].view()

return [x.view() for x in sol]
```

4 Resultados

O tratamento dos dados foi realizado sobre a lista $w = \texttt{range(1,101)}$, $k = 5$ e $\texttt{TFim} = 500$. Em todos os gráficos é variado um dos quatro parâmetros abaixo, sendo os outros mantidos fixos. O valor base destes parâmetros são

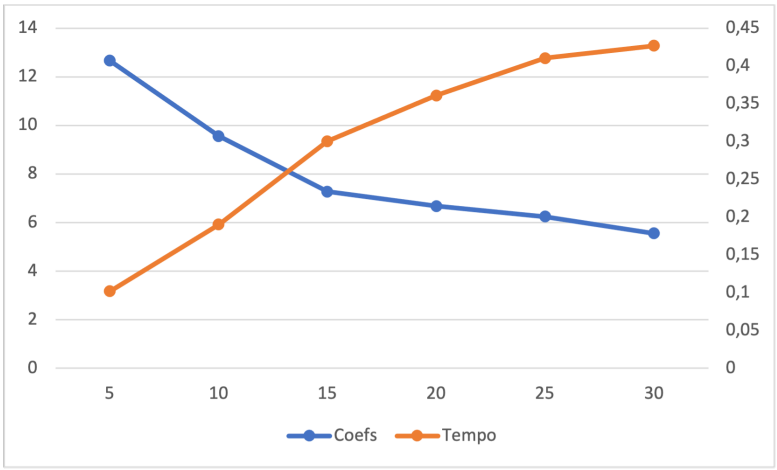
- $\texttt{NInd} = 10$
- $\texttt{TMut} = 5$
- $\texttt{TRep} = 5$
- $\texttt{TMor} = 5$

O objetivo é estudar o impacto desta variação no melhor coeficiente de inadaptação no final do programa e no tempo de execução do programa.

Para estes testes foi feita uma média de 100 ensaios, todos realizados do mesmo computador, para minimizar o efeito da aleatoriedade.

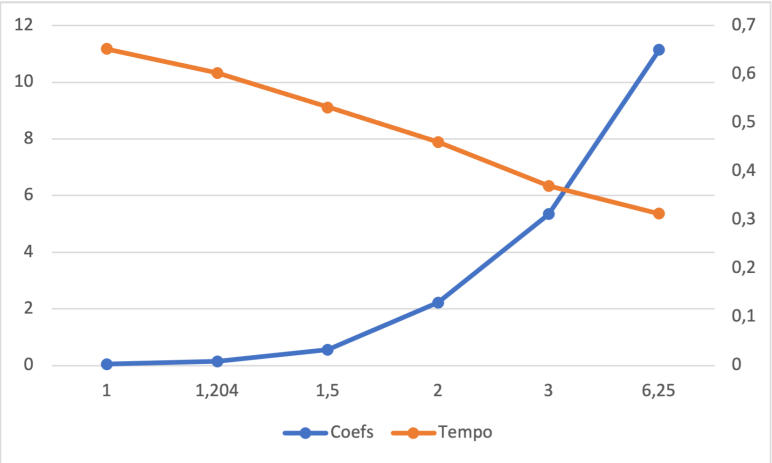
Em todos os gráficos em que é variado um parâmetro de tempo, pretendemos avaliar de forma linear a quantidade de vezes que a população sofre o evento em questão. Dado que a frequência destes eventos variam com os parâmetros de tempo em proporcionalidade inversa, não estamos a variar os tempos de forma linear.

Variação de `NInd`



Variação do coeficiente de inadaptação e do tempo de execução em função do número do número de indivíduos

Variação de `TMut`

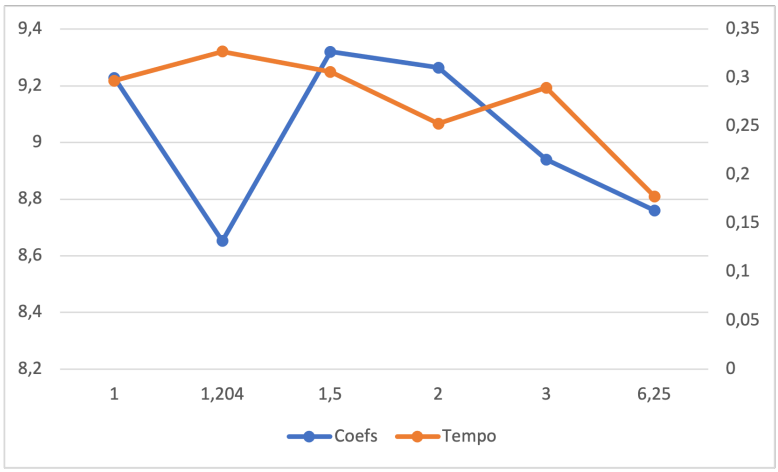


Variação do coeficiente de inadaptação e do tempo de execução em função do número de mutações

Variação de `TRep`

Variação do coeficiente de inadaptação e do tempo de execução em função do número de reproduções

Variação de `TMor`



Variação do coeficiente de inadaptação e do tempo de execução em função do número de mortes

5 Discussão de Resultados

Verifica-se que maiores valores para o número inicial de indivíduos tendem naturalmente a aumentar de forma linear o tempo de execução do programa e a diminuir o coeficiente de inadaptação, visto que um maior número de indivíduos implica uma maior quantidade de mutações e de reproduções.

Uma vez que as mutações e as reproduções tendem a diminuir o coeficiente de inadaptação, era de esperar que aumentar o número de reproduções e mutações (diminuindo `TMut` e `TRep`), tivesse um impacto positivo. Esta expectativa é confirmada pelo gráfico, onde valores menores para os parâmetros `TMut` e `TRep` retornam partições com menor coeficiente de inadaptação.

No caso da morte, seria de esperar que mais mortes diminuíssem o número de indivíduos. Como a mutação representa a grande maioria das operações realizadas pelo programa, menos indivíduos leva a uma diminuição do total de operações e uma diminuição do tempo que o programa leva correr. Em contrapartida, o coeficiente de inadaptação deveria piorar. No entanto, o gráfico apresenta resultados contrários, uma vez que um maior numero de mortes (menor `TMut`) está associado a um maior tempo de execução. Isto deve-se provavelmente ao facto de na simulação estamos a utilizar um número reduzidos de indivíduos e um número excessivo de operações morte.

Concluímos que, em geral, os gráficos confirmam as nossas expectativas.