O Problema das k-Partições Perfeitas

Computação evolutiva

Projeto de Elementos de Programação

Número do Aluno	Nome do Aluno
106489:	Martim Lopes
106182:	Daniel Bartolomeu
102563:	Afonso Gouveia
106419:	André Amaro

Licenciatura em Matemática Aplicada e Computação
Instituto Superior Técnico - Alameda

2022/2023

Conteúdo

Introdução

Este projeto foi realizado no âmbito da disciplina de Elementos de Programação com o principal intuito de desenvolver um programa em *Python* que calcule uma solução (exata ou aproximada) para o problema da k-partição perfeita de uma lista não vazia de valores positivos, através do paradigma de computação evolutiva e o princípio da simulação discreta estocástica.

Para um k e uma lista não vazia com elementos positivos $\bar{x}$, o projeto consiste na simulação de evolução da população formada por um número inicial de indivíduos, *NInd* (que são interpretados como k-partições de $\bar{x}$), para um dado intervalo de tempo (*TFim*). O programa termina exibindo as melhores soluções aproximadas presentes na população no instante *TFim* ou se encontrar uma solução exata.

Durante a simulação, a população evolui segundo os processos de *morte*, *mutação* e *reprodução* ditados por processos estocásticos que dependem de parâmetros dados *TMor*, *TMut* e *TRep*. De acordo com estes parâmetros torna-se então possível resolver o problema.

Planeamento

Para programar em camadas, é necessário distinguir os tipos de dados que serão utilizados. Ora, para este programa definimos: a *partição*, o *indivíduo*, a *população*, o *evento* e a *Cadeia de Acontecimentos Pendentes* (*CAP*).
Para além da função que gera valores aleatórios para os processos estocásticos, é também necessário utilizar funções que manipulam os tipos de dados definidos.
Para a partição é necessário ou útil:

- Uma função que permita inicializar a partição com uma distribuição aleatória e uniforme dos elementos pelos blocos;
- Uma função que permita editar a posição dos elementos da partição;
- Uma função que identifique em que bloco está um dado elemento da lista original;
- Uma função que identifique os índices dos elementos de um bloco da partição na lista original;
- Uma função que some os valores de cada bloco da partição;
- Uma função que some os valores de um bloco da partição específico;

Para o indivíduo é necessário ou útil:

- Uma função que inicializa um indivíduo, associando-o a uma partição e definindo o instante de formação dos seus blocos perfeitos, caso existam;
- Uma função que identifique qual a partição associada ao indivíduo;
- Uma função que identifique o instante de formação de um bloco perfeito.

- Uma função que calcule o coeficiente de inadaptação de um indivíduo.
- Uma função que retorne os índices (de 1 a k) dos blocos perfeitos.
- Uma função que transfere um conjunto de elementos da partição para um bloco específico, registando internamente o novo tempo de formação deste e dos blocos de origem dos elementos, caso algum destes seja perfeito.
- Uma função que transfere um elemento de um bloco para outro, registando internamente o novo tempo de formação destes, caso algum seja perfeito.

Para a população é necessário ou útil:

- Uma função que inicialize a população com 0 indivíduos;
- Uma função que adicione indivíduos à população;
- Uma função que remove todos os indivíduos para os quais uma dada condição seja verdadeira;
- Uma função que remove um indivíduo específico;
- Uma função que retorna um maximizante de uma função de output racional;
- Uma função que verifique se um determinado indivíduo pertença à população;
- Uma função que escolha um indivíduo aleatoriamente;
- Uma função que verifique quantos indivíduos a população tem;

Para o evento é necessário ou útil:

- Uma função que inicialize um evento;
- Uma função que retorne o tipo de evento;
- Uma função que retorne o instante em que o evento ocorre;
- Uma função que retorna os objetos a que este evento está associado;

Para a Cadeia de Acontecimentos Pendentes é necessário:

- Uma função que inicialize o *CAP* vazio com tempo final definido;

- Uma função que retorne o evento mais próximo;
- Uma função que apague o evento mais recente;
- Uma função que adicione um evento;
- Uma função que verifique se a *CAP* está vazia;

Resumo da Implementação dos Tipos de Dados

3.1 Partição

A partição foi implementada como uma classe com um atributo $\_lista$, que é uma lista de comprimento $len(X)$, cujo elemento de índice i corresponde ao índice do bloco ao qual está associado o elemento de índice i de uma lista X. Para manter independência entre os tipos e a aplicação foi necessário armazenar uma referência à lista à qual a partição diz respeito no atributo $\_X$ e o número de blocos da partição em $\_numBlocos$.

3.2 Indivíduo

Os indivíduos foram implementados pela classe indivíduo. Esta classe tem os atributos $\_particao$, que armazena uma referência à partição associada ao indivíduo; $\_numBlocos$, que descreve o número de blocos desta partição; $\_Blocos$, uma lista de comprimento $\_numBlocos$ com valores na forma [cond,instante], com cond booleano e instante racional, em que, para um bloco b, $\_Blocos[b-1][0]$ é verdadeiro se e só se b for perfeito, caso no qual, $\_Blocos[b-1][1]$ indica o seu instante de formação; $\_Perf$, que guarda o valor da soma de um bloco perfeito;$\_X$, que é uma referência à lista dos valores da partição.

3.3 População

A classe população tem o atributo $\_listaIndividuos$, que é uma lista dos indivíduos na população, e $\_numIndividuos$ que é um natural que representa o número de indivíduos na população.

3.4 Evento

A classe evento possui três atributos: $\_kind$, $\_instante$ e $\_envolvido$ que armazenam, respetivamente, uma string identificadora do tipo de evento, um número racional relativo ao instante em que o evento ocorre e o objeto (e.g., uma população ou um indivíduo) que está relacionado ao evento.

3.5 Cadeia de Acontecimentos Pendentes(CAP)

A Cadeia de Acontecimentos Pendentes é implementada por uma lista, $\_cadeia$, de eventos ordenada em ordem decrescente do instante em que cada ocorre. Na CAP só constam eventos cujo instante de formação é inferior a valor do atributo $\_tempoLimite$. Também é mantida uma contagem do número de eventos no atributo $\_numEventos$.

Funções sobre os Tipos de Dados

Para melhor compreender o funcionamento do programa e as funções que este usa durante o seu funcionamento, é necessário entender o funcionamento das classes definidas.

4.1 Classe Partição

```
1 def __init__(self,X,k):
2     self._X = X
3     self._numBlocos = k
4     self._lista = [0]*len(X)
5     for i in range(len(X)):
6         self._lista[i] = r.randrange(1,k+1)
```

- **__init(n,k,X)__:** O objeto inicializa com uma lista associada de comprimento *len(X)*, sendo todos os seus elementos associados a um bloco aleatório indexado entre 1 e k. É armazenada uma referência à lista original e ao número de blocos da partição.

```
1 def editar(self,i,x):
2     self._lista[i] = x
```

- **editar(i,x):** O elemento de índice i do *input* inicial passa a estar associado ao bloco de índice x.

```
1 def ler(self,i):
2     return self._lista[i]
```

- **ler(i):** Retorna o índice do bloco a que está associado o elemento de índice i do input inicial.

```
1 def LSomaBlocos(self):
2         res = [0]*self._numBlocos
3         for i in range(len(self._lista)):
4             res[self._lista[i]-1]+=self._X[i]
5         return res
```

- **LsomaBlocos():** Retorna uma lista cujo valor do índice i representa a soma dos elementos do bloco i+1.

```
1 def blocoIndice(self,i):
2     return [x for x in range(len(self._lista)) if self._lista[x
          ]==i]
```

- **blocoIndice(i):** Retorna uma lista com os índices de todos os elementos do input associados ao bloco de índice i.

```
1 def soma(self,I,X):
2     return reduce(lambda a,b:a+X[b],
3     filter(lambda f: self._lista[f] == I,range(len(self._lista)
          )),0)
```

- **soma(I,X):** Retorna a soma dos valores da lista original associados ao bloco de índice I.

4.2 Classe Indivíduo

```
1 def __init__(self,p,k,tempo):
2     self._particao = p
3     self._numBlocos = k
4     self._Blocos = [tempo]*k
```

- **__init__(p,k,X,tempo,Perf):** Inicializa o individuo com partição p, número de blocos k, cujos instantes de formação são predefinidos para *tempo* e com a lista associada X.

```
1 def part(self):
2     return self._particao
```

- **part()**: Retorna a partição atualmente associada ao indivíduo.

```
1 def BlocosPerf(self):
2     return [b for b in range(1,self._numBlocos+1) if self.
          _Blocos[b-1][0]]
```

- **BlocosPerf()**: Retorna lista dos índices dos blocos perfeitos do indivíduo atualmente.

```
1 def coeficiente(self):
2     return reduce(lambda a,b: a + abs(b-self._Perf),self.
          _particao.LSomaBlocos(self._X,self._numBlocos),0)/self.
          _numBlocos
```

- **coeficiente()**: Calcula e retorna o coeficiente de inadaptação do indivíduo

```
1 def instanteFormacao(self,i):
2     return self._Blocos[i-1][1]
```

- **instanteFormacao(i)**: Retorna o instante de formação do bloco de índice i.

```
1 def alterarParticao(self,i,b,instante):
2     blocoOriginal = self._particao.ler(i)
3     self._particao.editar(i,b)
4     self._Blocos[b-1] = [self._particao.soma(b,self._X) == self
          ._Perf,  instante]
5     self._Blocos[blocoOriginal-1] = [self._particao.soma(
          blocoOriginal ,self._X) == self._Perf,  instante]
```

- **alterarParticao(i,b,instante)**: Altera o elemento indice i da particao para o bloco b.

```
1 def associarBloco(self,i,B,instante):
2     alterado = [False] * self._numBlocos
3     for e in B:
4         alterado[self._particao.ler(e)-1] = True
5         self._particao.editar(e,i)
6     self._Blocos[i-1] = [self._particao.soma(i,self._X) == self
          ._Perf,   instante]
7     for x in filter(lambda x: alterado[x], range(self.
          _numBlocos)):
8         self._Blocos[x] = [self._particao.soma(x+1,self._X) ==
              self._Perf,   instante]
```

- **associarBloco(i,B,instante)**: Recebe lista de indices de X (lista dada no input) que associa ao bloco de índice i.

4.3 Classe População

```
1 def __init__(self):
2     self._numIndividuos = 0
3     self._listaIndividuos = []
```

- **__init(n)__:** O objeto inicializa com número de indivíduos igual a 0 e lista de indivíduos vazia.

```
1 def add(self,ind):
2     self._numIndividuos += 1
3     self._listaIndividuos.append(ind)
```

- **add(ind):** Descreve a adição do indivíduo ind à população.

```
1 def remocaoEmMassa(self,P):
2     for i in filter(P,self._listaIndividuos):
3         self._listaIndividuos.remove(i)
4         self._numIndividuos-=1
```

- **remocaoEmMassa(P):** Para cada elemento da população que satisfaça a condição P, esse elemento é removido da população e o número de indivíduos desta é reduzido em uma unidade.

```
1 def remover(self,ind):
2     self._listaIndividuos.remove(ind)
3     self._numIndividuos-=1
```

- **remover(ind):** O indivíduo ind é removido da população e o número de indivíduos desta é reduzido em uma unidade.

```
1 def estaPresente(self, ind):
2     return (ind in self._listaIndividuos)
```

- **estaPresente(ind):** retorna *True* se um indivíduo *ind* está presente na população e *False* caso contrário.

```
1 def randIndividuo(self):
2     import random as r
3     return r.choice(self._listaIndividuos)
```

- **randIndividuo():** Retorna um indivíduo aleatório da população.

```
1 def numIndividuos(self):
2     return self._numIndividuos
```

- **numIndividuos():** Retorna o número de indivíduos da população.

```
1 def numIndividuos(self):
2     return self._numIndividuos
```

- **maximizante(f):** Retorna um maximizante (individuo i da população tal que f(i) $>=$f(e) para qualquer indivíduo e da população) da função f de valores racionais.

```
1 def maximizante(self,f):
2     curMax = self._listaIndividuos[0]
3     for ind in self._listaIndividuos:
4         if f(ind)>f(curMax):
5             curMax = ind
6     return curMax
```

4.4 Classe Evento

```
1 def __init__(self,kind,instante,envolvido):
2     self._kind = kind
3     self._instante = instante
4     self._envolvido = envolvido
```

- **__init__(kind,instante, envolvido):** O objeto inicializa com uma string identificadora do tipo de evento (*kind*), o instante determinado para este ocorrer (*instante*) e os objetos aos quais este evento diz respeito (*envolvido*).

```
1 def kind(self):
2     return self._kind
```

- **kind():** Retorna a string identificadora do tipo de evento.

```
1 def instante(self):
2     return self._instante
```

- **instante():** Retorna o instante em que vai ocorrer o evento.

```
1 def envolvido(self):
2     return self._envolvido
```

- **envolvido():** Retorna o objeto ao qual este evento diz respeito.

4.5 Classe CAP

```
1 def __init__(self,Tfim):
2     self._numEventos = 0
3     self._tempoLimite = Tfim
4     self._cadeia = []
```

- **__init(Tfim)__:** O objeto inicializa com número de eventos associado igual a 0, o tempo limite (instante para o qual eventos com instante de ocorrência superior ao *Tfim* não são adicionados à CAP) indicado no argumento e cadeia (lista de eventos) associada vazia.

```
1 def top(self):
2         return self._cadeia[-1]
```

- **top():** Retorna o evento definido como o seguinte a ocorrer.

```
1 def delete(self):
2     self._cadeia.pop()
3     self._numEventos-=1
```

- **delete():** Faz com que o evento definido para ocorrer no instante menor de entre os eventos pendentes na CAP seja eliminado e reduz em 1 o número de eventos.

```
1 def add(self,E):
2     tempo = E.instante()
3     if tempo <= self._tempoLimite:
4         l = 0
5         r= self._numEventos-1
6         while l<r:
7             m = (l+r)//2
8             if tempo > self._cadeia[m].instante():
9                 r = m
10            else:
11                l = m+1
12
13        self._cadeia.insert(l,E)
14        self._numEventos+=1
```

- **add(E):** Caso o evento E tem associado um instante de ocorrência menor ou igual a um tempo limite definido previamente, adiciona-o à CAP de forma a que esta fique ordenada por ordem decrescente de instantes em que os eventos ocorrerão. Caso o evento a adicionar esteja definido para o mesmo instante que outro já existente na CAP, é adicionado a seguir desse outro evento (de modo a que ocorra mais cedo do que este).

```
1 def empty(self):
2     return self._numEventos == 0
```

- **empty():** Retorna *True* se não houver eventos na CAP e False caso contrário.

Descrição do Simulador

Para utilizar o simulador, chama-se a função principal main.simulador(), fornecendo os respetivos argumentos. São também utilizadas as funções main.aleatoriaExp(), funcoesEvento.mutacao(), funcoesEvento.morte() e funcoesEvento.reproducao().

5.1 Funções auxiliares

- **main.aleatoriaExp(v_medio)**: Devolve uma variável de valor médio v_medio.
- **funcoesEvento.mutacao(ind,tempo,X,k,Perf):** Começa-se por criar duas listas com os índices dos blocos da partição: uma em que a soma dos valores de cada bloco é menor que Perf, e outra em que é maior.
  De seguida seleciona-se aleatória e uniformemente um bloco de cada lista e calcula-se a diferença, designada por *dif*, entre a soma dos valores associados aos blocos. Escolhe-se, também aleatória e uniformemente um valor *x* associado ao bloco selecionado da lista dos blocos cuja soma dos valores associados é maior que Perf.
  Depois, é criada uma nova lista *SeleclistaIndxLinha* à qual serão adicionados os índices dos valores *x'*, que pertencem ao bloco cuja soma dos valores associados é menor que perf. De modo a serem adicionados à nova lista corretamente, são selecionados aleatória e uniformemente enquanto houver valores ainda não selecionados e a soma de todos os selecionados for inferior a $x - \frac{dif}{2}$.
  Por fim, utilizando o método *alterarParticao* o valor *x* é transferido para o bloco cuja soma dos valores associados é menor que Perf. Analogamente, utilizando o método *associarBloco* os valores *x'* são transferidos para o bloco cuja soma dos valores associados é maior que Perf.

- **funcoesEvento.morte(P,tempo):** Usa uma função auxiliar *condicaoDeMorte* que devolve um *booleano* que será analisada pelo método *remocaoEmMassa* para remover todos os índividuos para os quais *condicaoDeMorte* retorna *True*.
  A função auxiliar começa por criar uma lista com os índices dos blocos perfeitos do indivíduo. Se a lista não for vazia, ou seja, se houverem blocos perfeitos, irá procurar qual o bloco perfeito mais velho e mais novo, retornando a condição $2 \times$ *velho* $<$ *tempo* - *novo*. Se for vazia, ou seja, se não houver blocos perfeitos, retorna, obviamente, *False* de modo a que a *remocaoEmMassa* não aconteça.

- **funcoesEvento.reproducao(pai,mae,tempo,X,k,Perf):** A função começa por inicializar o filho como um indivíduo que tem todos os elementos no bloco de índice k, e a condição *existeReprod* como *False*.
  De seguida, se o indivíduo *pai* tem blocos perfeitos *existeReprod* passa a *True*. Verifica-se se o bloco perfeito selecionado do *pai* pode ser reconstruído com os elementos dos blocos imperfeitos da *mãe*.
  Caso seja impossível reconstruí-lo, existeReprod passa de novo a False, caso contrário, o bloco perfeito do *pai* é transferido para o índice 1 do filho. Transferem-se os blocos perfeitos da *mãe* para os índices a seguir (2,3,4,...). Se ainda houver blocos por preencher, procura-se todos os índices não utilizados e distribuem-se pelos blocos restantes. Retorna-se o booleano *existeReprod* e o indivíduo criado, *filho*.

5.2 Função main.simulador(X,k,TFim,TMorte,TMut,TRep,NInd)

Os argumentos desta função têm os seguintes significados:

- **X**: lista de valores da k-partição
- **k**: número de blocos da partição
- **TFim**: instante final da simulação
- **TMorte,TMup,TRep**: tempos médios de ocorrência dos eventos morte, mutação e reprodução, respetivamente.
- **NInd**: número de indivíduos no inicio da simulação

No início da simulação, é calculado, na variável Perf, o valor da soma de um bloco perfeito; é inicializado o indivíduo Res, que servirá para armazenar o resultado exato do problema, caso este surja durante a simulação; é inicializado o booleano particaoEncontrada para False, que, durante a simulação, passará ao valor True, caso uma partição perfeita seja encontrada.

Também será inicializada a CAP, correspondente à variável C, e a população, correspondente à variável P.

```
1     Perf = reduce(lambda x,y: x+y, X,0)//k
2     Res = individuo(particao(X,k),k,X,0,Perf)
3     particaoEncontrada = False
4     C = CAP(TFim)
5     P = populacao()
```

De seguida, é necessário gerar NInd indivíduos com partições aleatórias, testar se algum desses indivíduos tem uma k-partição perfeita (caso no qual particaoEncontrada passa a True e o Res passa ao indivíduo perfeito), e adicioná-los à população, agendando as respetivas mutações. Após isto estar feito, deve-se agendar os eventos de morte e reprodução que são globais.

```
    for it in range(NInd):
        ind = individuo(particao(X,k), k,X, 0.0,Perf)

        if len(ind.BlocosPerf()) == k:
            Res = ind
            particaoEncontrada = True


        P.add(ind)
        C.add(evento("mutacao",aleatoriaExp(TMut),ind))

    C.add(evento("reproducao",aleatoriaExp(TRep), P))
    C.add(evento("morte",aleatoriaExp(TMorte),P))
```

Tendo o estado inicial do simulador sido estabelecido, enquanto a CAP não estiver vazia e uma partição perfeita não for encontrada o seguinte ciclo while será percorrido:

```
    while not (C.empty() or particaoEncontrada):

        evt = C.top()
        C.delete()

        if evt.kind() == 'mutacao' and P.estaPresente(evt.
           envolvido()):
            ...
        elif (evt.kind() == 'reproducao') and (P.numIndividuos
           () > 1):
            ...
        elif evt.kind() == "morte":
            ...
#As reticencias simbolizam um pedaco de codigo omitido
```

Portanto, a cada iteração, é selecionado e guardado em evt o evento mais próximo da CAP, sendo de seguida removido desta

Se o evento for uma mutação, e se o individuo ao qual esta diz respeito ainda está

na população, é aplicada a função funcoesEvento.mutacao() ao indivíduo, sendo depois reagendada uma nova mutação e verificado se este tem uma k-partição perfeita.

```
1     ind = evt.envolvido()
2     mutacao(ind,evt.instante(),X,k,Perf)
3     C.add(evento("mutacao",
4     evt.instante() + aleatoriaExp(TMut),ind))
5     if len(ind.BlocosPerf()) == k:
6         Res = ind
7         particaoEncontrada = True
```

Se for uma reprodução, e existir mais que um indivíduo na população, são selecionados uniformemente desta dois indivíduos distintos. De seguida as variaveis existeReprod e ind são retornadas da função funcoesEvento.reproducao(). Se existeReprod for True, deve-se adicionar ind à população, agendado também a mutação deste. Deve-se verificar se a partição de ind é perfeita. Por fim, agenda-se a próxima reprodução

```
1     Ind1 = P.randIndividuo()
2     Ind2 = P.randIndividuo()
3     while (Ind2 == Ind1):
4         Ind2 = P.randIndividuo()
5     existeReprod, ind = reproducao(Ind1,Ind2,evt.instante(),X,k
         ,Perf)
6     if existeReprod:
7         P.add(ind) C.add(evento("mutacao", evt.instante()+
             aleatoriaExp(TMut),ind))
8         if k==len(ind.BlocosPerf()):
9             Res = ind
10            particaoEncontrada = True
11    C.add(evento("reproducao",
12    evt.instante()+aleatoriaExp(TRep),P))
```

Se o evento for do tipo morte, aplica-se a função morte à população. Caso isto diminua o número de indivíduos para 0, deve-se, de um modo semelhante, estabelecimento do estado inicial, gerar novos NInd indivíduos com partições uniformemente

aleatórias.

```
1     morte(evt.envolvido(),evt.instante())
2     if P.numIndividuos() == 0:
3         for Ind in range(NInd):
4             ind = individuo(particao(X,k),k,X,evt.instante(),
                  Perf)
5             if len(ind.BlocosPerf()) == k:
6                 Res = ind
7                 particaoEncontrada = True
8             else:
9                 P.add(ind) C.add(evento("mutacao",evt.instante
                      ()+aleatoriaExp(TMut),ind))
10    C.add(evento("morte", evt.instante()+aleatoriaExp(TRep),P))
```

Quando o ciclo terminar, se tiver sido encontrado um indivíduo com uma partição perfeita, exibe-se o resultado exato. Caso contrário, exibe-se a partição do indivíduo com menor coeficiente de inadaptação, ou seja, aquele que é maximizante da função f(x) = $\frac{1}{x.coeficiente()}$.

As partições são exibidas na forma de uma lista de k listas, em que cada uma das k listas representa um dos k blocos da partição, sendo os seus elementos, os valores dos elementos dos blocos.

```
1 if particaoEncontrada:
2         print("Resultado Final (Exato): ", [[X[y] for y in
              range(len(X)) if Res.part().ler(y) == x  ] for x in
              range(1,k+1)])
3     else:
4         bestInd= P.maximizante(lambda ind: 1/ind.coeficiente())
5         coef = bestInd.coeficiente()
6         print("Resultado Final (Aproximado): ", [[X[y] for y in
              range(len(X)) if bestInd.part().ler(y) == x] for x
              in range(1,k+1)])
7         print("coef de inadaptacao = ", coef)
```

Discussão de resultados

Para **simulacao(*v*,*k*,*TFim*,*TMorte*,*TMut*,*TRep*,*NInd*)**, com:

- *v = [14175, 15055, 16616, 17495, 18072, 19390, 19731, 22161, 23320, 23717, 26343, 28725, 29127, 32257, 40020, 41867, 43155, 46298, 56734, 57176, 58306, 61848, 65825, 66042, 68634, 69189, 72936, 74287, 74537, 81942, 82027, 82623, 82802, 82988, 90467, 97042, 97507, 99564]*;
- *k = 2*; *TMorte = 30*; *TMut = 20*; *TRep = 2*; *NInd = 400 usado apenas na fig. 6.2.*

Obtivemos os seguintes resultados experimentais:

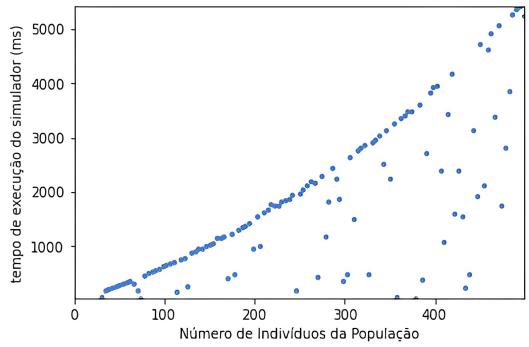

Figura 6.1: Tempo de execução do simulador em função do número de indivíduos da população

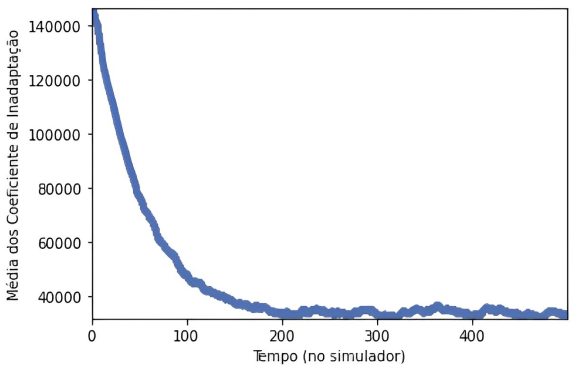

Figura 6.2: Coeficiente de inadaptação em função do tempo

Ao utilizar o simulador, concluiu-se que, para populações entre 1 e 500 indivíduos, a taxa de variação média do tempo de execução do simulador é positiva (fig. 6.1), com uma leve concavidade voltada para cima, indicando que, para quantidades maiores, esta taxa de variação média tem tendência a aumentar.

A fig. 6.2 indica que a média do valor dos coeficientes diminui dramaticamente, evidenciado que são apuradas partições ”mais perto”de partições perfeitas e a ser rejeitadas as com coeficientes de inadaptação maiores.

Para **simulacao(*v*,*k*,*TFim*,*TMorte*,*TMut*,*TRep*,*NInd*)**, com:

- *v = [267, 1, 125, 38, 574, 119, 25, 98, 4, 99, 7, 57, 420, 97, 18, 127, 108, 169, 206, 273, 169, 252, 534, 303, 180, 187, 257, 191]*;
- *k = 7*; *TMorte = 30*; *TMut = 25*; *TRep = 20*; *NInd = 60(usado apenas na fig. 6.4)*.

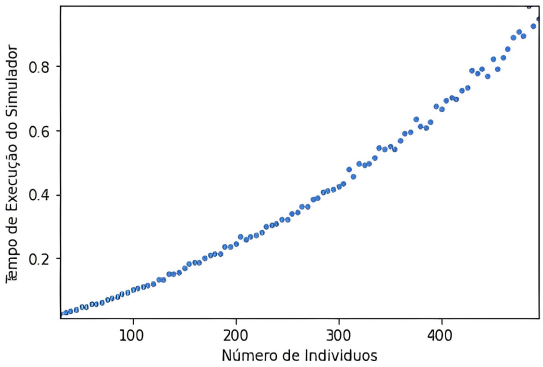

Figura 6.3: Tempo de execução do Simulador em função do número de indivíduos

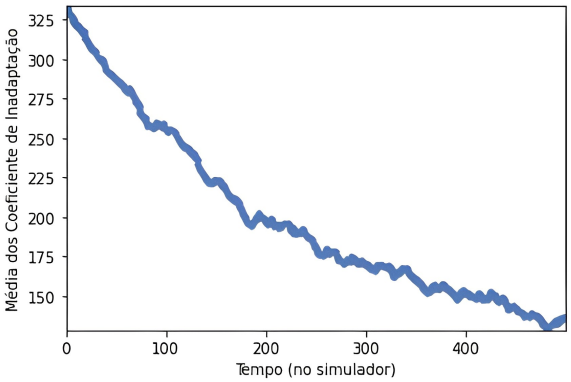

Figura 6.4: Média dos coeficientes de inadaptação em função do tempo

Um crescimento muito semelhante ao da fig. 6.1 acontece na fig. 6.3, reforçando a ideia de que a curva de complexidade temporal deste simulador em função do número de indivíduos se assemelha a estas. Na fig. 6.4 é de novo observável a tendência que os indivíduos têm de desenvolver partições com coeficientes cada vez menores.

Para simulacao(*v*,*k*,*TFim*,*TMorte*,*TMut*,*TRep*,*NInd*), com:

- *v = [5,5,3,4,8,1,9,3,1,7,3,3]*;
- *k = 4*; *TMorte = 5*; *TMut = 3*; *TRep = 8*; *NInd = 20 usado apenas na fig. 6.6*.

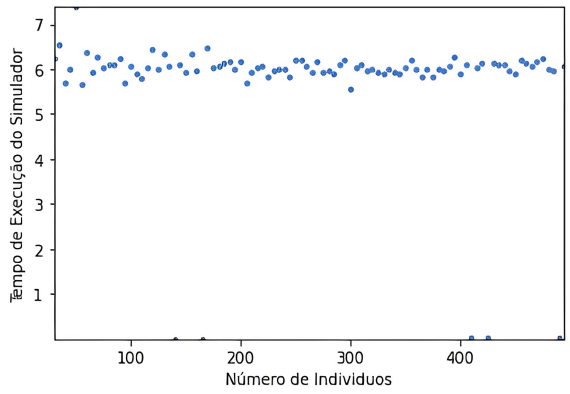

Figura 6.5: Tempo de execução do Simulador em função do número de indivíduos

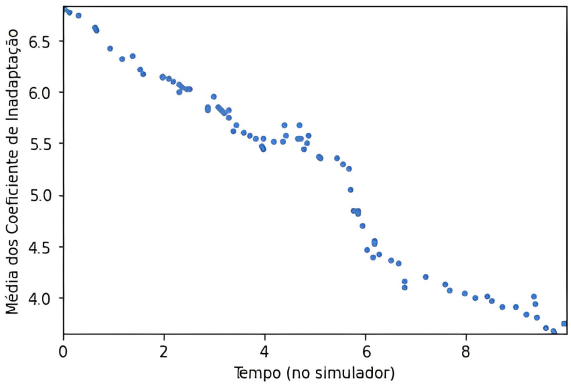

Figura 6.6: Média dos coeficientes de inadaptação em função do tempo

A fig. 6.5 contraria a tendência dos exemplos anteriores de formar curvas convexas.O motivo pelo qual esta curva é aproximadamente constante tem que ver com o facto de ter uma k-partição fácil de calcular, pelo que se encontra quase sempre e rapidamente uma solução exata, num intervalo de tempo pouco ou nada influenciado pelo número de indivíduos no sistema.

A fig. 6.6 reafirma a tendência que a média os coeficientes de inadaptação tem para diminuir com o tempo.

Conclusão

O problema da k-partição perfeita é considerado extremamente difícil, pois não existe uma forma eficiente de o solucionar. Isto deve-se ao facto de ser considerado um problema *NP-completo*, segundo a teoria da complexidade.

Este projeto foi-nos apresentado a fim de desenvolver competências nos vários paradigmas da programação em ambiente *Python*. O uso de classes e tipos de dados permitiu desenvolver um programa que tem como *output* uma aproximação da k-partição perfeita num tempo considerável.

Devido à elevada complexidade deste problema, tivemos de nos dedicar e esforçar consideravelmente, desenvolvendo competências de programação em vários paradigmas e de manipulação de tipos de dados.