Realatório: Projeto de Computação Evolutiva

Alunos: 109660 Guilherme Ribeiro, 106328 Luís Andrade,
109496 Marta Machado, 109508 Vasco Marques

Outubro 2023

Unidade Curricular: Elementos de Programação
Professor Carlos Caleiro

Conteúdo

1 Introdução

Este projeto, desenvolvido no âmbito da cadeira de Elementos de Programação, procura desenvolver, em Python, um programa baseado no paradigma de computação evolutiva. Consequentemente, este programa centra-se em encontrar uma solução aproximada do problema do número cromático.
O problema do número cromático consiste em encontrar o menor número de cores distintas para colorir um grafo, sendo um problema para o qual ainda não se tem uma solução eficiente. Note-se que uma coloração consiste numa atribuição de cores aos nós do grafo de modo a que os nós ligados por um arco tenham cores distintas entre si.

1.1 Objetivo

Assim, tem-se como objetivo simular a evolução de uma população inicial de possíveis colorações do grafo G dado (K- número de indivíduos dado), durante um tempo limite fornecido. Após a simulação, a solução apresentará o melhor adaptado dos indivíduos da população final (se houver sobreviventes), isto é, a coloração válida que utiliza um menor número de cores.
Durante a simulação, cada indivíduo pode sofrer mutações, reproduzir-se ou morrer, considerando os eventos avaliação, evolução e seleção, geridos por uma agenda (cadeia de acontecimentos pendentes).

1.2 Organização

O projeto assenta no método de programação modular por camadas, centrado nos dados, onde se divide o problema em subproblemas mais simples, que se traduzem em módulos implementados de forma independente.
Deste modo, procedeu-se, primeiramente, à descrição dos tipos de dados e respetivas operações. Seguindo-se o desenvolvimento do simulador (programa abstrato) sobre a camada que disponibliza os tipos de dados e a implementação dos módulos. Por fim, ao integrar o simulador com os módulos desenvolvidos, testou-se o programa com vários conjuntos de dados representativos e discutiu-se os resultados.

2 Densenvolvimento do Projeto

2.1 Tipos de dados

Como referido anteriormente, o projeto iniciou-se com a descrição dos tipos de dados e a definição das operações a eles associados, sendo, posteriormente, concebidos módulos que os implementam de forma independente.

2.1.1 Grafos

Um grafo é uma estrutura que traduz a relação dos objetos de um determinado conjunto, sendo constituido por nós (vértices) e arcos (arestas que correspondem a pares dos nós).
Neste caso, na primeira implementação, um grafo (g) é uma lista de listas. Temos que g[i] corresponde ao nó i+1 (pois, numa lista, a primeira posição é a posição 0) e que os elementos de g[i] são os vértices ligados a i+1. Tem-se que o número de vértices do grafo é len(g). No caso da segunda implementação, o grafo é uma lista que, na posição 0, indica o número de vértices do grafo e, na segunda posição, contém uma lista de tuplos que correspondem às arestas do grafo.
Estas duas implementações garantem, ao testar, que os resultados obtidos são independentes da implementação do módulo, um aspeto fundamental da programação em camadas. É importante destacar que cada vértice é um número natural. Além disso, com estas implementaçôes, cada aresta aparece duas vezes. Por exemplo, na primeira implementação, considerando a aresta entre os vértices 1 e 3, temos que 3 aparece na lista de 1 (g[0]) e 1 na lista de 3 (g[2]), enquanto que, na segunda implementação, aparece tanto o tuplo (1,3) como o (3,1). Concluiu-se também que o grafo não podia ter arestas que fossem de um vértice para o próprio vértice, pois, assim, nenhuma coloração seria válida.
As operações definidas para manipular o grafo foram:

- new(n) – criação de um grafo com n vértices e sem arestas;
- addedge(g,x,y)– adiciona a aresta (x,y) ao grafo g, só adicionando se ela não existir ou se não for para o próprio vértice, e acrescentando,na primeira implementação, tanto a g[x-1] como a g[y-1] e na segunda implementação os pares (x,y) e (y,x) a g[1];
- deledge(g,x,y) - remove a aresta (x,y) ao grafo g, só eliminando nas mesmas condições que o addege; /item dim(g) – devolve o número de vértices de g (len(g) – primeira implementação; g[0] – segunda implementação);
- emptyQ(g) - verifica se o gráfico é vazio (não tem arestas);
- edge(g,x,y) - verifica se a aresta (x,y) existe em g;

- graphQ(g) - verifica se g é um grafo, ou seja:

  Para grafos1:

  - g é uma lista
  - os elementos de g são listas
  - as listas não têm mais elementos do que o número de vértices do grafo
  - os elementos das listas (vértices) são inteiros e estão ente 1 e o comprimento de g
  - se x está em g[y-1], então y está em g[x-1]

    Para grafos2:
  - g é uma lista
  - a lista tem duas posições
  - a primeira posição é um inteiro
  - a segunda tem uma lista
  - essa lista é vazia ou formada por tuplos
  - verifica que cada tuplo tem dois elementos
  - confirma se as arestas são entre vértices que não pertencem ao grafo (*)
  - verifica que cada aresta (x,y) tem um par no sentido oposto (y,x)
  - confirma que não há arestas para o próprio vértice do grafo

    (*) Para este propósito tem-se uma função extra:

    * list.nodes(g) – uma função que numa lista coloca todos os vértices que têm ligação com outros vértices

2.1.2 Colorações

Como já foi referido, tem-se que uma coloração de um grafo é uma atribuição de cores aos vértices do grafo, sendo válida se os nós ligados por um arco tenham cores distintas entre si.
Na implementação, estabeleceu-se que uma coloração (c) corresponde a uma lista com dois elementos: c[0] é o grafo em questão e c[1] é a lista das cores de cada vértice. Tem-se que cada cor é um número natural e que c[1][i] é a cor do vértice i+1.
No desenvolvimento desta implementação, foi importado o módulo grafos.
Neste caso, foram consideradas as seguintes operações:

- new(g,w) - cria uma coloração de g, segundo as cores dadas por w;
- grafo(c) - devolve o grafo associado à coloração c;

- cor(c,v,n) - altera a cor do vértice v para n na coloração c;
- show(c) - devolve os pares (vértice,cor) da coloração c;
- num-erros(c) - devolve o número de erros da coloração c (ou seja, as vezes em que os nós ligados pela mesma aresta têm a mesma cor);
- num.cores(c) - devolve o número de cores distintas usadas em c, ao procurar o mínimo numa cópia da lista das cores, eliminado as suas ocorrências, sucessivamente, até a lista ficar vazia, contabilizando o número de eliminações;
- copyC(c) - cria uma cópia da coloração c.

2.1.3 Indivíduos

Um indivíduo trata-se de uma possível coloração de g, tendo um identificador único e irrepetível.
Na implementação, tem-se que corresponde a uma lista i com 3 elementos: i[0] é a coloração correspondente, i[1] é o momento em que nasceram, i[2] é o identificador.
No desenvolvimento desta implementação, foi importado o módulo grafos, colorações e random.
Está associado às seguintes operações:

- new(c,t,n) - cria um indivíduo com a coloração c, nascido no momento t, com o identificador n (devolve o indivíduo criado);
- coef(i) - devolve o coeficiente de adaptação de i (1);
- idade(i) - devolve o momento de criação de i;
- mutation(i) - altera uma cor na coloração de i (devolve o indivíduo evoluído);
- ident(i) - devolve o identificador do indivíduo;
- colour(i) - devolve a coloração do indivíduo;
- copyI(i) - copia indivíduo.

Nota: Coeficiente de adaptação
Como descrito no enunciado do projeto, o coeficiente de adaptação de um indivíduo com uma coloração c num dado instante é:

$$\begin{cases} \frac{\text{Número de nós do grafo}}{\text{Número de cores distintas da coloração c}} & \text{se } C \text{ é uma coloração válida de } G \\ \frac{1}{1+\text{número de erros da coloração}} & \text{caso contrário} \end{cases} \quad (1)$$

2.1.4 População

A população corresponde ao conjunto de indivíduos.
Foi implementada como uma lista p de indivíduos, importando-se os módulos grafos, colorações, random e indíduos.
Tem-se as seguintes operações:

- new() - cria uma população nova (devolvendo a população criada);
- dim(p) - devolve a dimensão de p (número de elementos);
- identQ(p,i) – verifica se algum indivíduo de p tem o identificador i (tem-se 0 como o identificador que identifica todos os indivíduos);
- ident(p,i) - devolve o indivíduo da população p com o identificador i (só devolvendo o valor, se o indivíduo existir), não podendo i ser 0, ao contrário da anterior;
- best(p) - devolve o indivíduo de p com melhor coeficiente de adaptação;
- worst(p) - devolve o indivíduo de p com pior coeficiente de adaptação;
- addI(p,i) - adiciona o indivíduo i à população p (devolvendo p);
- kill(p,i) - tira o indivíduo i da lista p (devolvendo a lista alterada).

2.1.5 Eventos

Um evento representa um acontecimento que atua sobre o sistema alterando o seu estado. Os eventos vão afetar os indivíduos, levando à evolução da população. Estes são caracterizados pelo seu tipo (de acordo com o enunciado, há três tipos de eventos), pelo indivíduo sobre o qual atuam (podendo ocorrer em mais de um indivíduo) e pelo instante em que ocorrem.
Assim, na implementação, tem-se um evento como uma lista e com 3 elementos: e[0] é o tipo do evento; e[1] é o identificador do indivíduo a quem o evento vai acontecer; e[2] é o instante em que acontece.
Verificam-se as seguintes operações:

- event(k,i,t) - cria um evento do tipo k que vai acontecer ao indivíduo com identificador i, no instante t (devolvendo o evento criado);
- kind(e) - devolve o tipo do evento e;
- ident(e) - devolve o identificador do indivíduo sobre o qual vai incidir o evento e;
- time(e) - devolve o instante do evento e.

2.1.6 CAP

A cadeia de acontecimentos pendentes funciona, na simulação discreta estocástica, como uma agenda que sequencia os eventos, guardando todos os eventos programados que ainda não aconteceram.
Assim, foi implementada como uma lista c dos eventos, sendo necessário importar o módulo eventos, ficando ordenada pelos instantes dos eventos. É importante destacar que o nome da agenda muda com as operações add e delete.
A CAP está munida das seguintes operações:

- new() - cria uma agenda vazia (devolve a agenda criada);
- add(e,c) - junta o evento e à agenda c (devolve a agenda atualizada, mantendo-a ordenada por instante dos eventos);
- nextE(c) - devolve o próximo evento da agenda c;
- delete(c) - elimina o primeiro evento da CAP c (devolve a CAP atualizada).

2.2 Simulador

O simulador foi construido de acordo com a técnica de simulação digital estocástica por sequenciação de eventos pendentes.
Começou-se por importar os módulos necessários (definidos anteriormente -fig.1- e os necessários aos cáculos de probabilidade a partir de math).

Figura 1:

```
import grafos as graph
import colorações as color
import indivíduos as ind
import população as pop
import eventos as event
import CAP as cap
```

```
import random as random
from math import pi
from math import log
from math import e
from math import atan
```

De seguida, definiu-se a função exprandom, que retorna um valor aleatório exponecial de valor médio m (Figura 2)

Tem-se, depois, a definição da função simulador. Esta função requer 6 variáveis, sendo elas:

- G – o grafo
- K - dimensão da população inicial;
- Tfim – duração da simulação;

Figura 2:

```
def exprandom(m):
    x=random.random()
    return -m*log(x)
```

- Tritmo – parâmetro associado ao evento evolução;
- Tlimiar – parâmetro associado ao evento avaliação;
- Tfiltro – parâmetro associado ao evento seleção.

Verificando-se se estas variáveis são válidas, isto é, se correspondem aos tipos de dados pretendidos, aparecendo uma mensagem de erro caso contrário. Seguidamente, iniciam-se as variáveis:

- CurrentTime – o instante em que a simulação está a ocorrer (iniciado em 0);
- Identificador – o identificador do próximo indivíduo a ser criado (iniciado em 1);
- População – onde se cria a lista de indivíduos, com a atribuição de uma coloração uniforme (uma cor distinta para cada nó de G);
- CAP – cria a lista de eventos.

Ora, os eventos podem ser de três tipos. Os primeiros dois:
1. Avaliação (cuja cadência corresponde a uma variável aleatória exponencial com valor médio Tlimiar e que resulta na morte do indiduo se for vivo nesse instante, com a probabilidade:

$$1 - \frac{2}{\pi} \arctan\left((1+A)^{1+\frac{8}{(1+I)}}\right)$$

, com A – coeficiente de adaptação do indivíduo - e I – a sua idade);

2. Evolução (associado a uma variável aleatória exponencial de valor médio Tritmo, resultando numa mutação (alteração da coloração) com probabilidade Prob ou numa reprodução do indivíduo com probalidade 1-Prob). Tem-se Prob=

$$\frac{1}{1+e^{\frac{K-T}{10}}}$$

, com T sendo o tamanho da população no momento (figura 3).

Sendo que, depois, marca-se a evolução do próximo indivíduo em função do Tritmo. Estes dois eventos estão associados a um certo indivíduo (pontuais), sendo criados e acrescentados à CAP para cada elemento da população.

Figura 3:

```
if event.kind(EventoAtual) == "avaliação": #se o proximo evento for avaliação

    avaliado = pop.ident(população, event.ident(EventoAtual)) #define o indivíduo a ser avaliado
    A = ind.coef(avaliado) #calcula o coeficiente de adaptação do avaliado
    I = CurrentTime - ind.idade(avaliado) #calcula a idade do avaliado

    if random.random() < (1-(2/pi)*atan((1+A)**(1+8/(1+I)))): #probabilidade do avaliado morrer
        população = pop.kill(população, avaliado) #retira o avaliado da população

    else: #se não morrer, marca-se a próxima avaliação
        avaliação = event.event("avaliação", ind.ident(avaliado), CurrentTime + exprandom(Tlimiar))
        CAP = cap.add(avaliação, CAP)
```

Figura 4:

```
elif event.kind(EventoAtual) == "evolução": #se o próximo evento for uma evolução

    avaliado = pop.ident(população, event.ident(EventoAtual)) #individuo a ser avaliado
    T = pop.dim(população) #calcula a dimensão da população atual

    if random.random() < 1/(1+e**((K-T)/10)): #probabilidade de haver uma mutação
        novo = ind.copyI(avaliado) #cria uma cópia do avaliado
        novo = ind.mutation(novo) #faz a mutação do indivíduo novo
        if ind.coef(novo) > ind.coef(avaliado): #substitui o avaliado pelo novo, se o novo for melhor
            população = pop.kill(população, avaliado) #elimina o avaliado
            população = pop.addI(população, novo) #adiciona o novo
            #não é preciso atualizar eventos, porque o novo tem o mesmo identificador (na prática, é o mesmo indivíduo)

    else: #se não houver uma mutação, há uma reprodução
        novo = ind.copyI(avaliado) #faz uma cópia do indivíduo avaliado
        filho = ind.new(ind.colour(novo), CurrentTime, identificador) #cria o filho, com base na cópia do avaliado
        filho = ind.mutation(filho) #provoca uma mutação no filho
        população = pop.addI(população, filho) #adiciona o filho à população
        avaliação = event.event("avaliação", identificador, CurrentTime + exprandom(Tlimiar)) #próxima avaliação do filho
        CAP = cap.add(avaliação, CAP)
        evolução = event.event("evolução", identificador, CurrentTime + exprandom(Tritmo)) #próxima evolução do filho
        CAP = cap.add(evolução, CAP)
        identificador += 1
```

3. Seleção – leva à morte de todos os indivíduos que nesse instante tenham uma coloração inválida e dos indivíduos com pior coeficiente de adaptação (assim a população não excede 3/2 K indivíduos), com cadência correspondente à constante Tfiltro.
Ou seja, trata-se de uma seleção global que afeta todos os indivíduos, sendo depois criado e adicionado o evento à CAP (ver Figura 5).

Assim, tem-se o decorrer da simulação, enquanto ainda existirem indivíduos ou enquanto o tempo limite não tiver sido atingido. Assim, o próximo evento

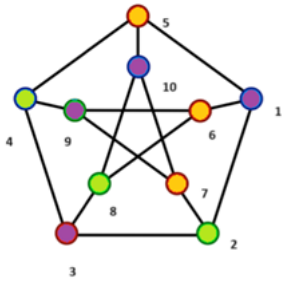
Figura 5:

```
else: #se o próximo evento for uma seleção
    check=True
    while pop.dim(população)!=0 and check:
        if ind.coef(pop.worst(população)) < 1 or pop.dim(população) > K*3/2:
            #elimina indivíduos enquanto houver indivíduos não válidos (com coeficientes <1)
            #elimina indivíduos enquanto a população for muito grande
            população=pop.kill(população,pop.worst(população)) #elimina o pior indivíduo
        else:
             check=False #se estiver tudo OK, para o ciclo

        seleção = event.event("seleção", 0, CurrentTime + Tfiltro) #próximo evento de seleção
        CAP = cap.add(seleção, CAP)
```

ocorre de acordo com o seu tipo, é eliminado da CAP, atualizando-se o tempo e sendo acrescentado os eventos à CAP e assim sucessivamente.
Para a simulação terminar temos que ou o número de indivíduos é zero ou chegou o tempo limite.
No primeiro caso é a presentada a mensagem de que foram eliminados todos os indivíduos. Por outro lado, no segundo caso, é apresentado o resultado do problema, ou seja, apresenta a coloração do vencedor e o número cromático.
Assim, tem como output do grafo de Peterson(g) com 10 indivíduos, 100 Tfim, 1 Tritmo, 1 Tlimiar e 1 Tfiltro:

Figura 6:



Que corresponde ao seguinte:

Figura 7:

```
simulador (g,10,100,1,1,1)

Vértice 1 : cor nº 3
Vértice 2 : cor nº 4
Vértice 3 : cor nº 3
Vértice 4 : cor nº 4
Vértice 5 : cor nº 2
Vértice 6 : cor nº 2
Vértice 7 : cor nº 2
Vértice 8 : cor nº 4
Vértice 9 : cor nº 3
Vértice 10 : cor nº 3

3
```

3 Experimentação e Discussão dos Resultados

Esta parte final do projeto centrou-se em duas partes:
Uma parte de testagem com vários tipos de grafos, procurando mostrar o bom funcionamento do programa e as diferenças encontradas;
E uma segunda parte que, tomando como caso base o grafo de Petersen, fez variar as outras variáveis, averiguando os efeitos significativos destas alterações.
Em cada grafo realizaram-se vários testes para se obter um resultado significativo.

3.1 Grafo Nulo (o número de vértices é 0)

Verifica-se a impossibilidade de realizar a simulação neste grafo, pois o coeficiente de adaptação teria o denominador 0, dado este corresponder ao número de cores que é 0.

3.2 Grafo Trivial (com um único vértice e sem arestas)

Como seria de esperar, independentemente dos valores das outras variáveis, o resultado do número cromático deu sempre 1 (em 20 testes), dado ter apenas um vértice.
Ver exemplo na Figura 8.

3.3 Grafo só com uma aresta

Ora, o resultado esperado será 2: de forma à coloração ser válida, é necessário que os vértices ligados pela aresta tenham cores diferentes, podendo os outros vértices (se existirem) ter a mesma cor do que um destes.
Tal foi confirmado para os testes, sendo que, desde que o Tfim seja suficientemente grande em comparação às outras variáveis de tempo, tal verifica-se

Figura 8:

```
g=graph.new(1)

simulador (g,10,100,1,1,1)

Vértice 1 : cor nº 1

1
```

maioritariamente.
Por exemplo:

Figura 9:

```
g=graph.new(7)
g=graph.addedge(g,1,2)

simulador (g,10,50,2,3,4)

Vértice 1 : cor nº 1
Vértice 2 : cor nº 3
Vértice 3 : cor nº 3
Vértice 4 : cor nº 1
Vértice 5 : cor nº 1
Vértice 6 : cor nº 1
Vértice 7 : cor nº 3

2
```

3.4 Grafos cíclicos

Para o grafo cíclico com quatro vértices (quadrado), o esperado seria 2, verificando-se que é uma coloração válida. De facto, em 20 testes, este foi o valor predominante, obtendo-se 3 cores em 2 dos testes.

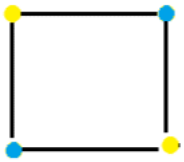
Figura 10:

```
simulador (g,10,10,1,1,1)

Vértice 1 : cor nº 1
Vértice 2 : cor nº 4
Vértice 3 : cor nº 1
Vértice 4 : cor nº 4

2
```

Figura 11:



3.5 Grafo Planar (sem cruzamento de arestas)

Tomou-se como exemplo o grafo:

Figura 12:

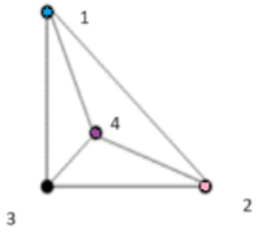
Ora, os grafos planares são abordados no Teorema das Quatro Cores que estabelece que o número cromático de qualquer grafo planar é no máximo 4. Os resultados obtidos vão de acordo com este teorema, pois, tomando o grafo indicado, verifica-se que o maior valor obtido maioritariamente é 4:

Figura 13:

```
simulador (g,10,10,1,1,1)

Vértice 1 : cor nº 1
Vértice 2 : cor nº 2
Vértice 3 : cor nº 3
Vértice 4 : cor nº 4

4
```



3.6 Árvore

As árvores têm muitas aplicações, nomeadamente em estruturas de dados.Para este exemplo, tomou-se a Figura 14.

Observa-se que, neste caso, o número cromático é 2. Tal foi confirmado com os testes (utilizando um Tfim suficientemente grande em ralação às outras variáveis de tempo) (ver figura 15)

Posteriormente, ao alterar os valores das variáveis, obtiveram-se resultados distintos em alguns casos. Por exemplo, com simulador(g,100,100,50,50,50) o

Figura 14:

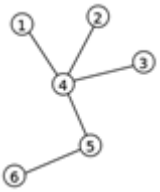


Figura 15:

```
simulador (g,100,1000,50,20,40)

Vértice 1 : cor nº 1
Vértice 2 : cor nº 1
Vértice 3 : cor nº 1
Vértice 4 : cor nº 4
Vértice 5 : cor nº 1
Vértice 6 : cor nº 4

2
```

valor predominante, em 20 testes, foi 3.

Figura 16:

```
simulador (g,100,100,50,50,50)

Vértice 1 : cor nº 1
Vértice 2 : cor nº 2
Vértice 3 : cor nº 1
Vértice 4 : cor nº 4
Vértice 5 : cor nº 1
Vértice 6 : cor nº 2

3
```

Tal deve-se ao facto de o Tfim ser reduzido em comparação com as outras variáveis de tempo, sendo, neste caso, o dobro. Assim, realça-se a proporção entre as variáveis, que será discutida mais à frente.

3.7 Grafo de Peterson

Sabe-se que o número cromático do grafo é 3. Primeiro, tomou-se como caso base: simulador (Petersen,10,100,1,1,1), que em 20 testes deu sempre como número cromático 3.
Por exemplo:

Figura 17:

```
simulador (g,10,100,1,1,1)

Vértice 1 : cor nº 3
Vértice 2 : cor nº 4
Vértice 3 : cor nº 3
Vértice 4 : cor nº 4
Vértice 5 : cor nº 2
Vértice 6 : cor nº 2
Vértice 7 : cor nº 2
Vértice 8 : cor nº 4
Vértice 9 : cor nº 3
Vértice 10 : cor nº 3

3
```

Aletrar Tfim: De seguida alterou-se o Tfim, tomando simulador (Petersen,10,Tfim,1,1,1), elaborando a seguinte tabela com os resultados.

Tabela 1: Tabela de Resultados

Tfim	3 cores	4 cores	5 cores	6 cores	nº testes	nº erros
100	20	0	0	0	20	0
50	16	4	0	0	20	4
45	11	9	0	0	20	9
40	10	10	0	0	20	10
30	4	16	0	0	20	16
20	0	15	5	0	20	25
10	0	1	15	4	20	43

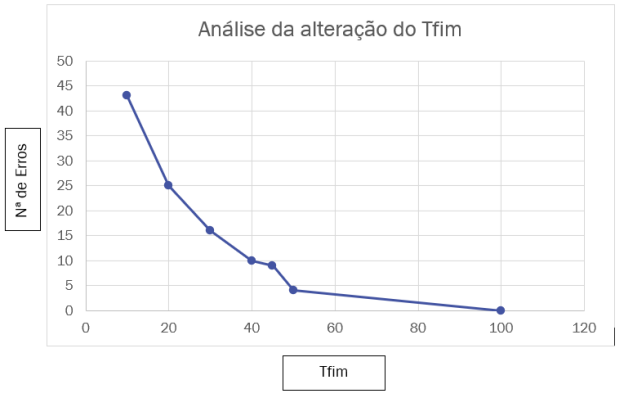
Nota: ocorrências com 4 cores valem 1 erro, ocorrências com 5 cores valem 2 erros, ocorrências com 6 ou mais cores valem 3 erros.

Conclusões:

- A relação Tfim/sucesso é inversa
- A relação é inconstante: do 50 para o 40 há mais 6 erros; do 40 para o 30 são 6 e do 30 para o 20 são 9; do 20 para o 10 são 18.
- De Tfim=30 para Tfim=20, a moda continua a ser 4 cores, mas o número de 3 e de 5 cores muda consideravelmente – de modo geral, comportam-se da mesma maneira, mas olhar para os detalhes revela que Tfim=20 tem mais erros

Resumo: como esperado, menos tempo de simulação leva a uma população que tem menos tempo de evoluir, e, portanto, tem mais erros, mas essa relação é inconstante; em todos os casos, nunca houve resultados inválidos.

Figura 18:



Alterar K

Tendo simulador(Peterson, K, 40, 1,1,1)
Escolheu-se Tfim = 40 porque é o caso em que acerta aproximadamente metade das vezes com as variáveis do caso base (se fosse Tfim=100 provavelmente nunca errava).

Tabela 2: Tabela de Resultados

K	3 cores	4 cores	5 cores	nº testes	nº erros
5	6	14	0	20	14
10	10	10	0	20	10
20	10	10	0	20	10
30	8	12	0	20	12
40	4	16	0	20	16

Conclusões:

- Como seria de esperar, K não tem tanto impacto nos resultados como Tfim
- Mesmo assim, tem alguma influência:
  - Populações muito pequenas têm mais erros (K=5)
  - Por outro lado, populações maiores também apresentam bastantes erros (K=40),
  - Sendo uma possível explicação o evento da seleção, que impede que a população cresça em demasia;
  - Verifica-se que populações de dimensão média têm um número mais reduzido de erros

Alterar Tlimiar, Tritmo e Tlimiar simultaneamente

Com simulador(Petersen,10,100,a,a,a)

Usamos novamente o caso base, de modo a mostrar as diferenças em proporção com o tempo da evolução.

Tabela 3: Tabela de Resultados

a	3 cores	4 cores	5 cores	6 cores	nº testes	nº erros
1	20	0	0	0	20	0
2	10	10	0	0	20	10
3	2	17	1	0	20	19
4	1	13	6	0	20	25
6	0	9	10	1	20	32
8	0	3	12	5	20	42
10	0	2	11	7	20	45

Conclusões:

• Verifica-se que, quanto maior o valor de a, menor é a frequência dos eventos.

• Assim, associa-se a mais erros como era de esperar, pois os eventos têm como objetivo a evolução da população, não alcançando o menor número cromático.

Alterar só Tritmo

Tem-se o simulador(Peterson,10,100,Tritmo,1,1)

Procura-se ver a partir de que Tritmo a população seria exterminada.

Mais precisamente, espera-se que, com o tempo das reproduções maior do que os restantes (associados às mortes dos indivíduos/população), a população seja toda eliminada.

Tabela 4: Tabela de Resultados

Tritmo	3 cores	4 cores	5 cores	6 ou mais cores	extermínio	nº testes
1	20	0	0	0	0	20
2	8	11	1	0	0	20
5	0	2	1	3	14	20
7	0	0	1	0	19	20
8	0	0	0	0	20	20
10	0	0	0	0	20	20

Conclusões:

• Confirmou-se o esperado, sendo que a partir de Tritmo=8, tal verifica-se. No entanto, tal não está completamente garantido, pois a simulação não é estocástica.

Notas finais

Após a realização dos testes tanto para vários gráficos distintos como, de forma

mais específica, para o grafo de Petersen, e da análise dos resultados obtidos, notou-se um certo padrão relativamente aos valores de entrada e à eficácia de o programa acertar o número cromática.
Mais precisamente, conclui-se que o que influencia mais são as proporções das variáveis relativamente umas às outras, obtendo-se resultados muitos distintos para o mesmo grafo e alterando apenas uma das variáveis.
Por exemplo, um Tfim maior relativamente às outras variáveis sugere um resultado mais fidedigno (com menos erros), pois resulta de uma evolução longa onde os vários eventos ocorreram.
Além disso, a experimentação serviu para confirmar muitos dos resultados espectados, revelando a eficiência do programa.
Por fim, é de notar que estas conclusões não são necessariamente universais: por se tratar de uma simulação estocástica, vários testes iguais podem ter resultados diferentes.

4 Conclusão

Concluiu-se, assim, o projeto desenvolvido no âmbito da cadeira, cumprindo os objetivos desejados. Nomeadamente, demonstrou-se o conhecimento da técnica de simulação digital estocástica por sequenciação de eventos pendentes, aplicando-a às várias implementações dos módulos, que foram criados de forma independente.
Por outro lado, os resultados obtidos apoiam o que era espectado, o que demonstra a eficácia da solução encontrada para resolver o problema proposto.
Assim, deu-se por finalizado o projeto que fomentou a capacidade de organização em equipa e o domínio das técnicas de programação lecionadas.