TP2_ex2

November 15, 2022

Implementação

Começamos por importar alguns métodos úteis para a resolução do problema

O pygame será utilizado para uma visualização gráfica do problema enunciado Para a utilização do mesmo é necessário instalá-lo (é uma biblioteca externa):

pip install pygame

[ ]:
```
from random import choices
from math import floor
from copy import deepcopy
import pygame
```

Parâmetros do programa

- N: corresponde ao tamanho da grelha de células normais
- p: probabilidade de cada célula "nascer" viva
- c: posição onde o centro com 3 x 3 células vivas iniciais se localiza

[ ]:
```
N = 15
p = 1
c = (5,5)
```

"Solver"

Após analisar o problema, resolvemos não utilizar um solver que foi trabalhado nas aulas. Fizemos o nosso próprio "solver" de forma a modelar o problema de forma mais compreensiva e possívelmente mais eficiente.

Função auxiliar 1 - vizinhos_possiveis

É uma função que recebe como parâmetros os valores das posições de x e y e o tamanho máximo da matriz (N x N) que queremos considerar. A função devolve todos os pares de posições x, y correspondente aos vizinhos dos valores recebidos como parâmetro, de forma a filtrar os casos que não fazem sentido (que saem da matriz)

[ ]:
```
def vizinhos_possiveis(x,y, N):
    v =  [(x+1,y), (x-1,y), (x,y-1), (x,y+1), (x-1, y+1), (x-1,y-1), (x+1, y+1),␣
↪(x+1,y-1)]
    return list(filter(lambda t: t[0] >= 0 and t[0] < N and t[1] >= 0 and t[1] <␣
↪N, v))
```

Função auxiliar 2 - celulas_vizinhas_vivas

É uma função que recebe como parâmetro uma matriz de células e uma lista de vizinhos, isto é, uma lista de pares de posições x e y vizinhos a uma determinada célula. A função devolve a quantidade de células vivas dentre os vizinhos.

[ ]:
```
def celulas_vizinhas_vivas(mapa, vizinhos):
    vivas = 0
    for x,y in vizinhos:
        if mapa[y][x]:
            vivas += 1
    return vivas
```

Função auxiliar 3 - list_to_key

É uma função que recebe como parâmetro uma lista de listas e devolve um tuplo de tuplos com os mesmos elementos. O ponto todo desta função é que as listas são estruturas mutáveis, enquanto que os tuplos são imutáveis. A utilização desta função vai ficar mais clara mais a frente

[ ]:
```
def list_to_key(lista):
    return tuple(tuple(l) for l in lista)
```

Função estado_inicial

É uma função que recebe como parâmetro uma matriz de células, um par que corresponde a posição do centro de células vivas 3 x 3, a probabilidade das células de borda "nascerem" vivas e o tamanho da matriz correspondente as células normais. Esta função não retorna nada, mas altera diretamente a matriz recebida como parâmetro.

[ ]:
```
def estado_inicial(lista_aux, centro, probabilidade, N):
    lista_aux[centro[1]][centro[0]] = 1

    for x,y in vizinhos_possiveis(centro[0],centro[1], N+1):
        lista_aux[y][x] = 1

    lista_aux[0][0] = choices([1, 0], [probabilidade, 1-probabilidade])[0]

    for i in range(1, N + 1):
        lista_aux[i][0] = choices([1, 0], [probabilidade, 1-probabilidade])[0]
        lista_aux[0][i] = choices([1, 0], [probabilidade, 1-probabilidade])[0]
```

Função trans_estado

É uma função que recebe como parâmetro a matriz de células corresponde ao estado atual e o tamanho da parte correspondente as células normais. A função devolve a matriz de células corresponde ao próximo estado.

[ ]:
```
def trans_estado(atual_estado, N):
    prox_estado = deepcopy(atual_estado)
    for i in range(1, N+1):
        for j in range(1, N+1):
```

```
            vizinhos = vizinhos_possiveis(i,j,N)
            vivas = celulas_vizinhas_vivas(atual_estado, vizinhos)
            if not atual_estado[i][j] and vivas == 3:
                prox_estado[i][j] = 1
            elif vivas != 2 and vivas != 3:
                prox_estado[i][j] = 0

    return prox_estado
```

Função verifica_i

É uma função que recebe um dicionário dos estados acessíveis com parâmetro. A função verifica se todos os estados acessíveis contém pelo menos uma célula viva.

[ ]:
```
def verifica_i(estados):
    verifica = True
    for estado in estados.keys():
        aux = [elem for vetor in estado for elem in vetor]
        if sum(aux) == 0:
            verifica = False
            break

    return verifica
```

Função verifica_ii

É uma função que recebe um dicionário dos estados acessíveis como parâmetro e o tamanho da matriz correspondente as células normais. A função verifica que toda a célula normal está viva pelo menos uma vez em algum estado acessível

[ ]:
```
def verifica_ii(estados, N):
    aux = [[0 for j in range(N)] for i in range(N)]
    for estado in estados.keys():
        for i in range(1, N+1):
            for j in range(1, N+1):
                aux[i-1][j-1] += estado[i][j]

    verifica = True
    i = 0
    while verifica and i < N:
        for j in range(N):
            if aux[i][j] == 0:
                verifica = False
                break
        i += 1

    return verifica
```

Criação da máquina de estados finita

Como nossa matriz é finita então nossa máquina também terá um número finito de estados. Basta considerarmos que cada combinação diferente de células vivas e mortas de uma matriz representa um estado único. Como cada estado é único, podemos pensar em estados como chaves de um dicionário, em que o valor associado a chave corresponde o seu estado (valor inteiro). Uma chave tem que ser um estado imutável, por isto, convertemos uma lista de listas em tuplo de tuplos sempre que queremos trabalhar com as chaves. A transição de um mesmo estado para o outro é sempre a mesma e como temos um número finito de estados, sabemos que necessariamente encontraremos um ciclo e este será único, ou seja, a partir daqui não surgirá estados diferentes daqueles que já temos e portanto podemos terminar a construção da máquina. Para facilitar a busca futura, invertemos as chaves com os valores, para aceder diretamente a um estado (inteiro) específico.

[ ]:
```
estados = {}
trans = {}

lista_aux = [[0 for j in range(N+1)] for i in range(N+1)]

estado_inicial(lista_aux, c, p, N)
estados[list_to_key(lista_aux)] = 1

estado = 2

lista_aux = trans_estado(lista_aux, N)
chave = list_to_key(lista_aux)
while chave not in estados:
    estados[chave] = estado
    trans[estado-1] = estado
    estado += 1
    lista_aux = trans_estado(lista_aux, N)
    chave = list_to_key(lista_aux)

trans[estado-1] = estados[chave]

# verificar se as propriedades i e ii são válidas

print(verifica_i(estados))
print(verifica_ii(estados, N))

# Inverte as chaves com os valores

estados = {v: k for k, v in estados.items()}
```

pygame

Não explicaremos aqui os conceitos relativos a esta biblioteca porque foge ao tema do trabalho, mas apresentaremos o código que construímos para a construção da parte que permite visualizar o problema de forma gráfica. É preciso dar run all para funcionar corretamente (escolher simulação manual ou automática, colocando àquela que não foi escolhida em comentário)

[ ]:
```
pygame.init()
```

Estado Inicial

[ ]:
```
MAX_SIMULATION_DIMENSION = 700
BLOCK_SIZE = floor(MAX_SIMULATION_DIMENSION/(N+1))
SIMULATION_DIMENSION = BLOCK_SIZE * (N+1)

def draw_cells(celulas, N):
    for i in range(0, N+1):
        for j in range(0, N+1):
            if celulas[i][j]:
                x = BLOCK_SIZE * i
                y =  BLOCK_SIZE * j + 100
                rect = pygame.draw.rect(screen, (0,0,0), [x, y, BLOCK_SIZE,␣
 ↪BLOCK_SIZE])
                pygame.draw.rect(screen, (255,0,0), rect, 1)

screen = pygame.display.set_mode((SIMULATION_DIMENSION,␣
 ↪SIMULATION_DIMENSION+100))
screen.fill((255,255,255))

font = pygame.font.Font('freesansbold.ttf', 32)
text = font.render('ESTADO 1', True, (255,0,0), (255,255,255))
textRect = text.get_rect()
textRect.center = (SIMULATION_DIMENSION // 2, 50)

estado = 1
draw_cells(estados[estado], N)
pygame.display.update()
```

Simulação Manual (Seta esquerda e direita)

[ ]:
```
"""

run = True

while run:
    screen.fill((255,255,255))
    screen.blit(text, textRect)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

        if event.type == pygame.KEYDOWN:

            if event.key == pygame.K_RIGHT:
```

```
                estado = trans[estado]
                text = font.render('ESTADO ' + str(estado), True, (255,0,0),␣
↪(255,255,255))

            if event.key == pygame.K_LEFT:
                if estado >= 2:
                    estado = estado - 1
                    text = font.render('ESTADO ' + str(estado), True, (255,0,0),␣
↪(255,255,255))

    draw_cells(estados[estado], N)
    pygame.display.update()
"""
```

Simulação Automática

[ ]:
```
run = True

MOVEEVENT = pygame.USEREVENT+1
T = 250
pygame.time.set_timer(MOVEEVENT, T)
estado = 1

while run:
    screen.fill((255,255,255))
    screen.blit(text, textRect)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

        if event.type == MOVEEVENT:
            estado = trans[estado]
            text = font.render('ESTADO ' + str(estado), True, (255,0,0),␣
↪(255,255,255))

    draw_cells(estados[estado], N)
    pygame.display.update()
```

Fim do pygame

[ ]:
```
pygame.quit()
```