TP2 - Conway's Game of Life

Novembro, 2022

Bruno Miguel Ferreira Fernandes - a95972

Hugo Filipe de Sá Rocha - a96463

Definir valores de input do problema

In [40]:
```
from pysmt.shortcuts import *
from pysmt.typing import INT
from z3 import *
import random


global N
N = 5
global centro
centro = (random.randint(2,N-2),random.randint(2,N-2))
global p
p = 0.5
```

Funções auxiliares

- gera_aleatorio() - Gera número aleatório 0 ou 1,com probabilidade p de sair 1 e (1-p) de sair 0.
- nao_quad_central() - Gera a lista das posições da matriz que não estão contidas no quadrado central, preenchidas com 0 no estado inicial.

In [41]:
```
def gera_aleatorio():
        r = random.randint(1,100)
        if r <= p*100:
            return 1
        else:
            return 0


def nao_quad_central():
    res = [(i,j) for i in range(1,N+1) for j in range(1,N+1)]
    for i in range(centro[0]-1,centro[0]+2):
          for j in range(centro[1]-1, centro[1]+2):
                res.remove((i,j))
    return list(res)

global nqc
nqc = nao_quad_central()


global pos
pos = [f'({i},{j})' for i in range(1,N+1) for j in range(1,N+1)]
```

A seguinte função cria a k-ésima cópia das variáveis de estado, agrupadas num dicionário que nos permite aceder às mesmas pelo nome:

In [42]:
```
def declare(k):
```

```
    state = {}
    state['pc'] = Int('pc'+str(k))
    for i in range(1,N+1):
        for j in range(1,N+1):
            state[f'({i},{j})'] = Int('('+str(i)+','+str(j)+')')
    for i in range(N+1):
        state[f'(0,{i})'] = Int('(0,'+str(i)+')')
    for i in range(1,N+1):
        state[f'({i},0)'] = Int('('+str(i)+',0)')
    return state
```

A seguinte função `init`, dado um possível estado do programa (um dicionário de variáveis), devolve um predicado do pySMT que testa se esse estado é um possível estado inicial do programa.

In [43]:
```
#verifica se é estado inicial( pc == 0, posicoes do quadrado central a 1,
#bordas aleatorias, colocar as restantes posicoes a 0)
def init(state):
    return And(
        state['pc'] == 0,
        (And([(state[f'({i},{j})'] == 1) for i in range(centro[0]-1,centro[0]+2)
                                         for j in range(centro[1]-1, centro[1]+2)])),
        (And([(state[f'(0,{i})'] == gera_aleatorio()) for i in range(N+1)])),
        (And([(state[f'({i},0)'] == gera_aleatorio()) for i in range(1,N+1) ])),
        (And([(state[f'({i},{j})'] == 0) for (i,j) in nqc]))
    )
```

A seguinte função `vizinhos_vivos`, dada uma célula e um estado, retorna o número de vizinhos vivos dessa célula no estado state.

In [44]:
```
#funçao que dada uma string do tipo '(x,y)' e um estado, retorna o numero
#de vizinhos vivos da celula (x,y) no estado state
def vizinhos_vivos(celstr, state):
    x = int(celstr[1])
    y = int(celstr[3])
    moves = [(x+1,y), (x,y+1), (x-1,y), (x,y-1)]
    return len(list(filter(lambda t: t[0] >= 0 and t[0] <= N and t[1] >= 0 and t[1] <= N
                           and state[f'({t[0]},{t[1]})'] == 1,moves)))
```

A seguinte função `nasce`, garante que as células nascem nas condições abaixo:

- Caso a célula possua 3 vizinhos vivos a célula passa de 0 para 1.

In [45]:
```
def nasce(curr,prox):

    return And([(prox[v] == 1)          for v in pos
                                        if curr[v] == 0
                                        if vizinhos_vivos(v,curr) == 3
                ]
               )
```

A seguinte função `sobrevive`, garante que as células sobrevivem (mantém-se viva) nas condições abaixo:

- Caso a célula possua 2 ou 3 vizinhos vivos, caso contrário morre ou continua morta.

In [46]:
```
def sobrevive(curr,prox):

    return And([(prox[v] == 1)            for v in pos
                                          if curr[v] == 1
```

```
                                        if vizinhos_vivos(v,curr) == 3
                                        or vizinhos_vivos(v,curr) == 2
             ]
            )
```

A seguinte função `morre`, garante que as células morrem nas condições abaixo:

- Caso a célula não possua 2 nem 3 vizinhos vivos.

In [47]:
```
def morre(curr,prox):

    return And([(prox[v] == 0)              for v in pos
                                            if curr[v] == 1
                                            if UGE(vizinhos_vivos(v,curr),4)
                                            or Not(UGE(vizinhos_vivos(v,curr),2))
             ]
            )
```

A seguinte função `continua_morta`, garante que as células permanecem mortas nas condições abaixo:

- Caso a célula continue sem possuir nem 2, nem 3 vizinhos, então permanece no seu estado morta.

In [48]:
```
def continua_morta(curr,prox):

    return And([(prox[v] == 0)              for v in pos
                                            if curr[v] == 0
                                            if UGE(vizinhos_vivos(v,curr),4)
                                            or Not(UGE(vizinhos_vivos(v,curr),2))]
            )
```

A seguinte função `trans`, dados dois possíveis estados do programa, devolve um predicado do pySMT que testa se é possível transitar do primeiro para o segundo estado.

In [49]:
```
def trans(curr,prox):


    t0 = And(
             prox['pc'] == curr['pc']+1,
             nasce(curr,prox),
             sobrevive(curr,prox),
             morre(curr,prox),
             continua_morta(curr,prox)

    )


    t1 = And(
              prox['pc'] == curr['pc'],
              prox == curr

         )




    return Or(t0,t1)
```

A função `gera_traco`, dada uma função que gera uma cópia das variáveis do estado, um predicado que

testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, e um número positivo k gera um possível traço de execução do programa de tamanho k.

In [50]:
```
def gera_traco(declare,init,trans,k):

        s = Solver()

        trace = [declare(i) for i in range(k)]

        s.add(init(trace[0]))

        for i in range(k-1):
            s.add(trans(trace[i], trace[i+1]))

        if s.check() == sat:
            m = s.model()
            for i in range(k):
                print("Passo", i)
                for v in trace[i]:
                    print(v, "=", m[trace[i][v]])
                print("----------------")

        else:
            print("Não foi possível gerar o traço.\n")



gera_traco(declare,init,trans,5)
```

```
Passo 0
pc = 0
(1,1) = 1
(1,2) = 1
(1,3) = 1
(1,4) = 0
(1,5) = 0
(2,1) = 1
(2,2) = 1
(2,3) = 1
(2,4) = 0
(2,5) = 0
(3,1) = 1
(3,2) = 1
(3,3) = 1
(3,4) = 0
(3,5) = 0
(4,1) = 0
(4,2) = 0
(4,3) = 0
(4,4) = 0
(4,5) = 0
(5,1) = 0
(5,2) = 0
(5,3) = 0
(5,4) = 0
(5,5) = 0
(0,0) = 1
(0,1) = 1
(0,2) = 0
(0,3) = 1
(0,4) = 1
(0,5) = 1
(1,0) = 1
(2,0) = 0
(3,0) = 0
```

```
(4,0) = 1
(5,0) = 0
----------------
Passo 1
pc = 1
(1,1) = 1
(1,2) = 1
(1,3) = 1
(1,4) = 0
(1,5) = 0
(2,1) = 1
(2,2) = 1
(2,3) = 1
(2,4) = 0
(2,5) = 0
(3,1) = 1
(3,2) = 1
(3,3) = 1
(3,4) = 0
(3,5) = 0
(4,1) = 0
(4,2) = 0
(4,3) = 0
(4,4) = 0
(4,5) = 0
(5,1) = 0
(5,2) = 0
(5,3) = 0
(5,4) = 0
(5,5) = 0
(0,0) = 1
(0,1) = 1
(0,2) = 0
(0,3) = 1
(0,4) = 1
(0,5) = 1
(1,0) = 1
(2,0) = 0
(3,0) = 0
(4,0) = 1
(5,0) = 0
----------------
Passo 2
pc = 2
(1,1) = 1
(1,2) = 1
(1,3) = 1
(1,4) = 0
(1,5) = 0
(2,1) = 1
(2,2) = 1
(2,3) = 1
(2,4) = 0
(2,5) = 0
(3,1) = 1
(3,2) = 1
(3,3) = 1
(3,4) = 0
(3,5) = 0
(4,1) = 0
(4,2) = 0
(4,3) = 0
(4,4) = 0
(4,5) = 0
(5,1) = 0
(5,2) = 0
```

```
(5,3) = 0
(5,4) = 0
(5,5) = 0
(0,0) = 1
(0,1) = 1
(0,2) = 0
(0,3) = 1
(0,4) = 1
(0,5) = 1
(1,0) = 1
(2,0) = 0
(3,0) = 0
(4,0) = 1
(5,0) = 0
----------------
Passo 3
pc = 3
(1,1) = 1
(1,2) = 1
(1,3) = 1
(1,4) = 0
(1,5) = 0
(2,1) = 1
(2,2) = 1
(2,3) = 1
(2,4) = 0
(2,5) = 0
(3,1) = 1
(3,2) = 1
(3,3) = 1
(3,4) = 0
(3,5) = 0
(4,1) = 0
(4,2) = 0
(4,3) = 0
(4,4) = 0
(4,5) = 0
(5,1) = 0
(5,2) = 0
(5,3) = 0
(5,4) = 0
(5,5) = 0
(0,0) = 1
(0,1) = 1
(0,2) = 0
(0,3) = 1
(0,4) = 1
(0,5) = 1
(1,0) = 1
(2,0) = 0
(3,0) = 0
(4,0) = 1
(5,0) = 0
----------------
Passo 4
pc = 4
(1,1) = 1
(1,2) = 1
(1,3) = 1
(1,4) = 0
(1,5) = 0
(2,1) = 1
(2,2) = 1
(2,3) = 1
(2,4) = 0
(2,5) = 0
```

```
(3,1) = 1
(3,2) = 1
(3,3) = 1
(3,4) = 0
(3,5) = 0
(4,1) = 0
(4,2) = 0
(4,3) = 0
(4,4) = 0
(4,5) = 0
(5,1) = 0
(5,2) = 0
(5,3) = 0
(5,4) = 0
(5,5) = 0
(0,0) = 1
(0,1) = 1
(0,2) = 0
(0,3) = 1
(0,4) = 1
(0,5) = 1
(1,0) = 1
(2,0) = 0
(3,0) = 0
(4,0) = 1
(5,0) = 0
----------------
```

A função `testa_prop_1`, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, um predicado a testar e um número positivo k, testa se a propriedade seguinte é satisfeita:

- Todos os estados acessíveis contém pelo menos uma célula viva.

In [51]:
```
def testa_prop_1(declare,init,trans,inv,K):
    s = Solver()


    trace = [declare(i) for i in range(K)]


    s.add(init(trace[0]))

    for i in range(K-1):
         s.add(trans(trace[i], trace[i+1]))

    s.add(Not(And([inv(trace[i]) for i in range(K-1)])))

    if s.check() == sat:
                print("A propriedade 1 não é satisfeita.\n")
                return

    else:
        print(f'A propriedade 1 verifica-se.\n')


def inv1(state):

    for v in state:
        if v not in ['pc']:
            if state[v] == 0:
                continue
            else:
                return True
```

```
    return False

testa_prop_1(declare,init,trans,inv1,5)
```

```
A propriedade 1 verifica-se.
```

A função `testa_prop_2`, dada uma função que gera uma cópia das variáveis do estado, um predicado que testa se um estado é inicial, um predicado que testa se um par de estados é uma transição válida, um predicado a testar e um número positivo k, testa se a propriedade seguinte é satisfeita:

- Toda a célula normal está viva pelo menos uma vez em algum estado acessível.

In [52]:
```
def testa_prop_2(declare,init,trans,inv,K):
    s = Solver()


    trace = [declare(i) for i in range(K)]


    s.add(init(trace[0]))

    for i in range(K-1):
         s.add(trans(trace[i], trace[i+1]))

    res = [f'({i},{j})' for i in range(1, N+1) for j in range(1,N+1)]

    for cel in res:
        s.add(Not(Or([inv(trace[i], cel) for i in range(K-1)])))

    if s.check() == sat:
                print("A propriedade 2 não é satisfeita.\n")
                return

    else:
        print(f'A propriedade 2 verifica-se.\n')

def inv2(state, cel):
    if state[cel] == 1:
        return True
    return False

testa_prop_2(declare,init,trans,inv2,5)
```

```
A propriedade 2 não é satisfeita.
```