Inteligência Artificial

Thiago Henrique Leite da Silva, RA: 139920

- **Problema dos Missionários e Canibais**

Sobre o Problema:

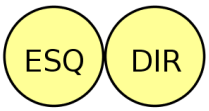
O problema dos Missionários e Canibais é muito antigo, e se trata de três missionários e três canibais que precisam passar para o outro lado da margem através de um barco com capacidade para 2, sendo que, se houver ao menos um de cada na margem, em nenhum momento pode haver mais canibais do que missionários. O barco não pode cruzar o rio sem ninguém, precisa de ao menos um tripulante.

Primeiros passos:

Logo de início podemos perceber que os estados possíveis do problema são finitos e não são muitos, portanto, podemos elencar todos eles para perceber quais as ocorrências e peculiaridades presentes neles que podem nos ajudar.

Criei um diagrama de estados utilizando o JFLAP, e ao todo encontrei 39 estados diferentes possíveis, onde levamos em conta a quantidade de missionários e canibais na direita e esquerda, e também a posição do barco.

Na representação utilizada, cada estado é composto por dois círculos, onde cada um deles representa uma das margens, esquerda e direita.



Já o barco é representado por um círculo circunscrito na margem em que ele estiver:

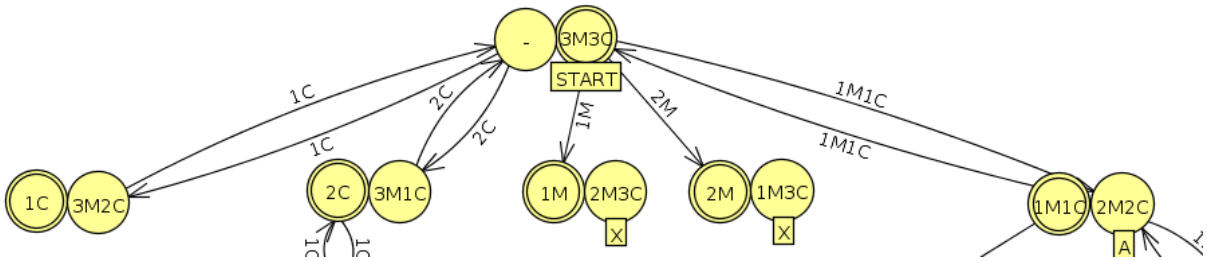
Estando definida a forma de representação, partimos para o estado inicial:

- Começaremos com 3 missionários e 3 canibais na direita. Sabendo que a capacidade do barco é 2, podemos chegar à conclusão de que, para cada estado, temos no máximo 5 possíveis formas de realizar a viagem de uma margem para outra, gerando, portanto, no máximo 5 sucessores para cada estado.

- As possibilidades de transferência são estas:

1. Um canibal;
2. Dois canibais;
3. Um missionário;
4. Dois missionários;
5. Um missionário e um canibal.

Chegamos então nos sucessores do estado inicial. Conseguimos agora montar o diagrama de estados deste exemplo:



Após descobrirmos os sucessores, precisamos verificar se eles continuam sendo válidos, e se porventura chegamos ao estado final, que é nosso critério de parada.

O label [**X]** indica os estados que deixaram de satisfazer a condição de existência. Podemos verificar que das 5 possibilidades, apenas 3 continuarão.

Conseguimos perceber também que a primeira opção, mover 1 canibal, não é interessante, pois no segundo movimento só poderíamos voltar ao estado inicial. E por fim, não podemos repetir duas vezes seguidas o mesmo movimento.

As primeiras linhas de código:

Após ter todas essas informações em mãos, chegou a hora de desenvolver o script que soluciona o problema, decidi não ler a explicação da monitora neste primeiro momento, para não enviesar minha linha de raciocínio.

O que tinha raciocinado para a solução era:

- Gerar o estado inicial
- A partir dele gerar seus 5 sucessores
- Eliminar os sucessores que quebraram as validações
- Verificar se estamos no final
- Para cada um dos sucessores que restaram, gerar também seus sucessores em um loop
- Guardar os estados atuais para não permitirmos que a mesma ação seja tomada duas vezes seguidas

Linguagem escolhida:

Decidi desta vez utilizar Ruby por ser uma linguagem orientada a objetos de alto-nível, e que particularmente gosto muito. Apesar do C ser mais performático, não estávamos desta vez precisando

fazer um código de meio segundo, então acredito que em Ruby conseguiria fazer um código mais enxuto e legível, além de aproveitar alguns recursos da linguagem, como as pilhas, que em C precisaria implementar, o que é uma tarefa fácil, mas ocuparia mais linhas no script. Por fim, algo que a orientação a objetos proporciona e que foi de grande valia para este problema, é permitir fazer métodos próprios para a classe Struct, que foi como decidi representar o game.

Primeiros problemas:

Comentei acima que utilizei Structs para chegar na solução, mas não foi bem assim desde o início. No começo, estava tentando representar o problema com uma Struct de dois Hash's, ou seja, teríamos uma struct **Margin** que receberia dois parâmetros, rigth e left. Rigth e Left possuiriam os campos cannibals e missionaries, que indicariam quantos deles estariam na margem apropriada; ou seja, uma confusão.

```
Margin = Struct.new(:left, :rigth) do
    def init
        left = {missionaries: 0, cannibals: 0}
        rigth = {missionaries: 3, cannibals: 3}
    end
end
```

Dessa forma, estava encontrando muita dificuldade para pensar em como fazer a parte dos sucessores de uma maneira prática, tive muitos problemas para representar os estados, foi onde passei dois dias raciocinando, mas não conseguia evoluir. Percebi o quanto é importante estruturarmos bem um código antes de prosseguirmos, um problema de estrutura é mais difícil de resolver posteriormente.

Virada de Chave:

Foi aí que lembrei da explicação genérica feita pela monitora do semestre passado e fornecida pela professora, ali encontrei a solução para o que estava procurando, a estrutura. O que precisava representar como uma Struct não era a margem, e sim os estados, pois conseguiria também fazer métodos próprios para eles, como verificar se é valido, se é o estado final, se o barco está na direita, se está na esquerda, entre outros que poderiam ter sido implementados. A explicação ajudou demais também na hora de fazer o loop dos sucessores que estava tendo dificuldades.

Tinha agora o código todo estruturado:

- Criei uma Struct State, que possui como parâmetro o número de missionários na direita, número de missionários na esquerda, número de canibais na esquerda, números de canibais na direita, a direção do barco e dois campos para gravar quantos missionários e canibais foram movidos para se chegar a este estado.
- Implementei o método para verificar se um estado é valido e também se ele é final.

```ruby
State = Struct.new(:m_left, :c_left, :m_rigth, :c_rigth, :boat_dir, :m, :c) do
  def left?
    boat_dir == :left
  end

  def rigth?
    boat_dir == :rigth
  end

  def isValid?(miss, can)
    (more_miss_than_can? || m_left.zero? || m_rigth.zero?) && correct_sum?(miss, can) &&
positive_values?
  end

  def isFinal?(missionaries, cannibals)
    m_left == missionaries && c_left == cannibals
  end

  def more_miss_than_can?
    m_left >= c_left && m_rigth >= c_rigth
  end

  def correct_sum?(miss, can)
    m_left + m_rigth == miss && c_left + c_rigth == can
  end

  def positive_values?
    c_left >= 0 && c_rigth >= 0 && m_left >= 0 && m_rigth >= 0
  end
end
```

rigth? e **left?** são métodos autoexplicativos que retornam booleanos referentes a posição atual do barco naquele estado.

IsValid?: Verifica se o nó é válido, ou seja, número de missionários é maior ou igual que o de canibais em ambos os lados, a não ser que o número de missionários seja zero em algum dos lados.

Exemplo: 3 Missionários na esquerda e 3 Canibais na direita. Do lado direito, o número de canibais é maior que o de missionários, mas sabemos que o estado é válido pois não temos missionários na direita.

Também checamos se a soma de missionários na esquerda com missionários na direita dá o número total de missionários, neste exemplo, 3. Análogo para os canibais.

Verificamos também se todos os itens são maiores ou iguais a zero. Sendo assim, estamos assegurando que qualquer estado válido possível passa na validação.

isFinal?: Verificamos se todos os missionários e canibais estão na esquerda.

- Após definida a estrutura, dei início a classe principal, **MissionariosECanibais**, que receberá como parâmetros o número de canibais, missionários e a capacidade do barco (todos estes parâmetros estão relacionados a feature que implementei e que explicarei posteriormente).

```ruby
class MissionariosECanibais

  attr_reader :inicial_state, :pending_states, :checked_states, :boat_capacity,
              :missionaries, :cannibals, :solve

  def initialize(missionaries: 3, cannibals: 3, boat_capacity: 2)
    @missionaries = missionaries
    @cannibals    = cannibals
    @boat_capacity  = boat_capacity
    @pending_states = []
    @checked_states = []
    @solve = []
  end
```

Obs.: O attr_reader define os atributos da classe que poderão ser lidos em qualquer lugar do código, para setá-los utilizamos o @ na frente do nome da variável.

O número de missionários e canibais por default é 3, e a capacidade do barco por default é 2.

Iniciamos também os vetores: estados pendentes e estados checados, além da variável solve, onde guardaremos os movimentos necessários para ganhar o jogo.

- **Corpo do Script**

```ruby
def self.play(*args)
   new(*args).play
 end

 def play
   define_inicial_state
   final_state = breadth_first_search
   finish_game(final_state)
 end
```

O corpo do script é bem simples, possui apenas três métodos que resumem nossa solução. Primeiramente definimos o estado inicial, realizamos a busca em largura (que será explicada mais

adiante) já armazenando o estado final retornado, e encerramos o jogo analisando se o estado final era o desejado.

O método self.play serve para definirmos um método de classe, assim conseguimos chamar o método play que se encontra mais abaixo sem precisar instanciar um objeto da classe.

- **Desenvolvimento**

```ruby
def define_inicial_state
   puts @inicial_state =
   State.new(0, 0, missionaries, cannibals, :rigth, 0, 0)

   puts "\nINÍCIO > #{cannibals} Canibais e #{missionaries}
           Missionários na margem direita!\n\n"

   raise 'Invalid Inicial State!' if cannibals > missionaries
 end
```

Explorando um pouco mais os métodos listados acima, começamos com a definição do estado inicial. Setamos e imprimimos o estado inicial com base nos parâmetros fornecidos pelo usuário.

A feature desenvolvida foi poder variarmos a quantidade de missionários, canibais e até mesmo a capacidade do barco.

Caso o número de canibais seja maior que o de missionários, imprimimos uma Exception na tela, pois o estado inicial será inválido.

- **Função principal**

```ruby
def breadth_first_search
    pending_states.push(inicial_state)
    previous = inicial_state

    while pending_states.any?
      state = pending_states.pop
      register_action(state, previous)
      previous = state

      return state if state.isFinal?(missionaries, cannibals)
      checked_states.push(state)

      sons = successors(state, previous)
      sons.each { |son| pending_states.push(son) if exclude?(checked_states, son) &&
                                                    exclude?(pending_states, son) }
    end
  end
```

O que fazemos agora é uma sequência de movimentos:

- Começamos passando o estado inicial definido anteriormente para pendente
- No início ele também será o estado anterior
- Agora, começamos a busca em largura, ou seja, pra cada estados, vamos passar por todos seus adjacentes, que neste caso, são seus sucessores.
- Começamos empilhando o primeiro estado pendente (o inicial), registramos a ação tomada (neste caso ainda não tomou), setamos o estado anterior.
- Caso o estado em que estamos seja o estado final, encerramos a busca, caso contrário, marcamos o estado como já checado, e vamos agora para seus sucessores.
- Após ter descoberto seus sucessores, vamos iterar sobre todos eles verificando se já não estão pendentes ou se já foram checados, caso nenhum desses dois casos sejam verdade, empilhamos o sucessor na pilha de estados pendentes.
- Vamos fazer isso até que a pilha estaja vazia.

- **Método de busca utilizado (BFS)**

Este algoritmo era uma das opções possíveis para resolver este problema, achei ele interessante pela ordem da busca, por sempre visitar o estado, logo após seus vizinhos, depois os vizinhos dos vizinhos, e assim sucessivamente. O mais importante na verdade é o fato de que a BFS percorre todos os estados ao alcance do estado inicial, não necessariamente todos os estados do problema, e neste caso dei a possibilidade de o usuário alterar os parâmetros iniciais, então fez sentido para mim essa propriedade. Vale lembrar que adaptei o método para utilizar com pilhas, ao invés de filas, pois as filas são do tipo FIFO, ou seja, a cada iteração, em tese, estarei em um estado mais próximo da solução.

- **Definição dos Sucessores**

```ruby
def successors(state, previous)
    successors = Array.new

    if missionaries+cannibals <= @boat_capacity
      return [State.new(state.m_left+missionaries, state.c_left+cannibals,
                        state.m_rigth-missionaries, state.c_rigth-cannibals, :left)]
    end

    (@boat_capacity+1).times do |i|
      (@boat_capacity+1).times do |j|
        next if i.zero? && j.zero? || i+j > @boat_capacity

        if state.rigth?
          new_state = State.new(state.m_left+i, state.c_left+j, state.m_rigth-i,
                                                state.c_rigth-j, :left, i, j)
        else
          new_state = State.new(state.m_left-i, state.c_left-j, state.m_rigth+i,
                                                state.c_rigth+j, :rigth, i, j)
        end

        successors << new_state if new_state.isValid?(missionaries, cannibals) &&
                                   !isEqual?(new_state, previous)
      end
    end

    successors
  end
```

Para definir os sucessores, precisamos do estado atual e do estado anterior.

Iniciamos um array de sucessores para guardar os possíveis novos estados.

Utilizamos agora a capacidade do barco para levar em conta a criação dos sucessores, caso ela seja maior do que a soma dos missionários e canibais, o problema está encerrado, levamos todos de uma vez.

Caso contrário, faremos uma espécie de produto entre vetores, vamos ver na prática para entender melhor:

Capacidade do Barco: 2

Array1 = [0,1,2]

Array2 = [0,1,2]

Produto: (0,0), (0,1) (0,2), (1,0) (1,1), (1,2), (2,0), (2,1), (2,2)

Basta eliminarmos agora os indesejados, ou seja, quando temos dois zeros, ou quando a soma dos elementos do par é maior do que a capacidade do barco, como no caso de (2,1), por exemplo.

Assim temos todas as possibilidades:

(0,1), (0,2), (1,0), (1,1) e (2,0).

Isso funciona para todos os casos.

Tendo as possibilidades, basta subtrairmos ou somarmos na posição adequada, se estamos na direita, vamos para esquerda, então somamos na esquerda e subtraímos na direita, e assim sucessivamente.

Agora criamos os estados se forem válidos e não forem iguais ao estado anterior.

- **Execução**

Com valores default:

```
1: Terminal
[205] pry(main)> MissionariosECanibais.play
#<struct State m_left=0, c_left=0, m_rigth=3, c_rigth=3, boat_dir=:rigth, m=0, c=0>

INÍCIO > 3 Canibais e 3 Missionários na margem direita!

Mova => 1 canibal(is) e 1 missionário(s) para esquerda.
Mova => 1 missionário(s) para direita.
Mova => 2 canibal(is) para esquerda.
Mova => 1 canibal(is) para direita.
Mova => 2 missionário(s) para esquerda.
Mova => 1 canibal(is) e 1 missionário(s) para direita.
Mova => 2 missionário(s) para esquerda.
Mova => 1 canibal(is) para direita.
Mova => 2 canibal(is) para esquerda.
Mova => 1 missionário(s) para direita.
Mova => 1 canibal(is) e 1 missionário(s) para esquerda.

FIM > 3 Canibais e 3 Missionários na margem esquerda!

=> #<struct State m_left=3, c_left=3, m_rigth=0, c_rigth=0, boat_dir=:left, m=1, c=1>
[206] pry(main)>
```

Com a capacidade do barco igual a 1:

```
1: Terminal
[207] pry(main)> MissionariosECanibais.play(boat_capacity: 1)
#<struct State m_left=0, c_left=0, m_rigth=3, c_rigth=3, boat_dir=:rigth, m=0, c=0>

INÍCIO > 3 Canibais e 3 Missionários na margem direita!


É impossível concluir o jogo com os parâmetros fornecidos!
=> nil
[208] pry(main)>
```

Com a capacidade do barco igual a 4:

```
[208] pry(main)> MissionariosECanibais.play(boat_capacity: 4)
#<struct State m_left=0, c_left=0, m_rigth=3, c_rigth=3, boat_dir=:rigth, m=0, c=0>

INÍCIO > 3 Canibais e 3 Missionários na margem direita!

Mova => 1 canibal(is) e 3 missionário(s) para esquerda.
Mova => 3 missionário(s) para direita.
Mova => 1 canibal(is) e 3 missionário(s) para esquerda.
Mova => 3 missionário(s) para direita.
Mova => 1 canibal(is) e 3 missionário(s) para esquerda.

FIM > 3 Canibais e 3 Missionários na margem esquerda!

=> #<struct State m_left=3, c_left=3, m_rigth=0, c_rigth=0, boat_dir=:left, m=3, c=1>
```

Com o 7 missionários, 6 canibais e capacidade do barco 3:

```
1: Terminal
[209] pry(main)> MissionariosECanibais.play(missionaries: 7, cannibals: 6, boat_capacity: 3)
#<struct State m_left=0, c_left=0, m_rigth=7, c_rigth=6, boat_dir=:rigth, m=0, c=0>

INÍCIO > 6 Canibais e 7 Missionários na margem direita!

Mova => 1 canibal(is) e 2 missionário(s) para esquerda.
Mova => 2 missionário(s) para direita.
Mova => 1 canibal(is) e 2 missionário(s) para esquerda.
Mova => 2 missionário(s) para direita.
Mova => 3 missionário(s) para esquerda.
Mova => 1 missionário(s) para direita.
Mova => 1 canibal(is) e 2 missionário(s) para esquerda.
Mova => 1 canibal(is) e 1 missionário(s) para direita.
Mova => 2 canibal(is) e 1 missionário(s) para esquerda.
Mova => 1 canibal(is) para direita.
Mova => 3 missionário(s) para esquerda.
Mova => 3 canibal(is) para direita.
Mova => 1 canibal(is) para esquerda.
Mova => 2 canibal(is) para direita.
Mova => 3 canibal(is) para esquerda.
Mova => 3 missionário(s) para direita.
Mova => 1 canibal(is) e 2 missionário(s) para esquerda.
Mova => 1 missionário(s) para direita.
Mova => 1 canibal(is) e 2 missionário(s) para esquerda.

FIM > 6 Canibais e 7 Missionários na margem esquerda!

=> #<struct State m_left=7, c_left=6, m_rigth=0, c_rigth=0, boat_dir=:left, m=2, c=1>
```

Com 4 canibais e 2 missionários:

```
1: Terminal
[210] pry(main)> MissionariosECanibais.play(missionaries: 2, cannibals: 4)
#<struct State m_left=0, c_left=0, m_rigth=2, c_rigth=4, boat_dir=:rigth, m=0, c=0>

INÍCIO > 4 Canibais e 2 Missionários na margem direita!

RuntimeError: Invalid Inicial State!
```

- **Código completo:**

Anexo à parte. Caso queira testar professora, sugiro usar a mesma versão do Ruby que utilizei, o *ruby 2.7.1p83,* versões anteriores podem esbarrar em problemas na definição do método ou até mesmo na impressão (método puts). Com o ASDF dá para instalar localmente a versão desejada do ruby, ou até mesmo subir um container Docker.

Para testar online teme este site aqui https://www.tutorialspoint.com/execute_ruby_online.php

Que é bem legal, é só colocar o código e a chamada do método ao final, a chamada é desta forma:

MissionariosECanibais.play (com valores default)

MissionariosECanibais.play(missionaries: 6, cannibals: 3, boat_capacity: 2) (variar os parâmetros)

:)