# Scape Coronavírus: implementação de um jogo utilizando a técnica de solução de problemas por restrições (CSP - *Constraint Satisfaction Problem*).

O presente documento descreve tecnicamente a aplicação desenvolvida para o projeto final da disciplina de Inteligência Artificial (IA) do Programa de Pós-Graduação em Engenharia Elétrica e Computação (PPGEEC) da Universidade Presbiteriana Mackenzie (UPM).

A aplicação é um jogo com o tema Coronavírus. O mesmo foi nomeado “Scape Coronavírus”. Consiste em uma aplicação *web* que pode ser acessada através do *link* https://rsg73626.github.io/docs/scape-coronavirus/. Sua implementação (código) pode ser acessada através do *link* https://github.com/rsg73626/scape-coronavirus.

Visto que o presente documento descreve tecnicamente a aplicação criada, não será explicado quais as regras do jogo nem como jogar. Essas informações podem ser encontradas na opção “Regras e Como Jogar” presente na página da aplicação (cujo endereço na *web* foi disponibilizado logo acima); ou aceassando diretamente o *link* https://rsg73626.github.io/docs/scape-coronavirus/rules.html. Assim, o objetivo desse documento é descrever as nuances do código desenvolvido bem como das tecnologias utilizadas.

## Organização do projeto

O projeto possui a seguinte estrutura de pastas e arquivos:

- Pasta raiz
  - **index.html**: página inicial do jogo.
  - **rules.html**: página com as regras do jogo e como jogar.
  - **Scape Coronavírus - Documentação.pdf**: este documento.
  - Pasta **css**: contém arquivos de estilo.
    - **style.css**: arquivo de estilo da página index.html.
    - **rules.css**: arquivo de estilo da página rules.html.
  - Pasta **js**: contém arquivos JavaScript.
    - Pasta **graph**
      - **graph.js**: implementação da estrutura de dados grafo.

- **graph-test.js**: criação e manipulação de um grafo usando a implementação criada no arquivo graph.js.

- Pasta **csp**
  - **csp.js**: implementação de um algoritmo de *Constraint Satsfaction Problem* genérico para resolução de qualquer problema usando essa técnica.
  - **map-colors.js**: resolução do problema de colorir um mapa com restrição de cores diferentes para regiões fronteiriças, utilizando a implementação de *Constraint Satisfaction Problem* criada no arquivo csp.js.
- Pasta **game**
  - **board.js**: implementação de uma estrutura de dados para representar o tabuleiro do jogo. A implementação nesse arquivo estende a estrutura de dados criada no arquivo graph.js na pasta graph.
  - **view.js**: implementação de uma classe para criação de toda a interface do jogo e execução de rotinas de acordo com as ações do jogador.
  - **game.js**: implementação de função para inicialização da interface do jogo. A função criada é chamada no fim da página index.html, na pasta raiz.
- Pasta **service**
  - **resources.js**: implementação de uma estrutura auxiliar para a leitura dos arquivos utilizados na aplicação (as imagens).

- Pasta **resources**
  - Pasta **images**: contém as imagens utilizadas no projeto.

## Solução de problemas por restrições

**Esta seção foi escrita baseada no conteúdo do capitulo VI do livro "*Artificial Intelligence A Modern Approach Third Edition*", de Stuart Russel e Peter Norving (edição públicada em 2010). Este foi o livro texto da disciplina durante o semestre em que este documento foi escrito (primeiro semestre de 2020).**

Como sugere o nome, esta técnica visa encontrar a solução de problemas a partir da criação de um algoritmo que encontre os valores soluções para o problema que satisfaçam todas as restrições (condições obrigatórias) apresentadas. Para tanto, faz-se necessário modelar o problema. Três elementos fundamentais para essa modelagem são:

- as variáveis: paras as quais serão atribuídos valores do domínio até que se encontre o conjunto de valores que não infringe nenhuma restrição;
- o domínio: conjunto contendo os valores que podem ser atribuídos às variáveis;
- e as restrições: condições obrigatórias que devem ser respeitas pela solução encontrada.

A partir desses três elementos é realizado uma busca utilizando *backtracking* para tentar encontrar a solução do problema.

Um exemplo clássico da literatura é o problema de definir as cores das regiões de um mapa levando em consideração que duas regiões vizinhas não podem ter cores iguais. Veja Figura 1.

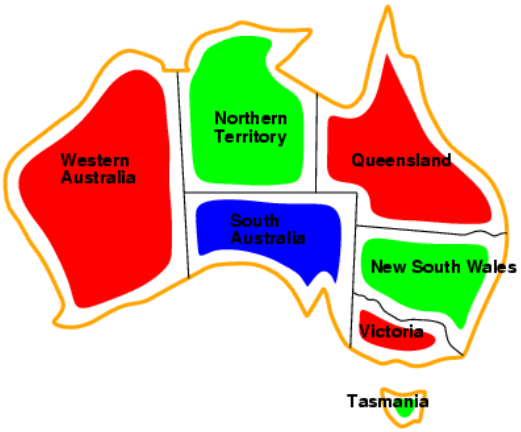


*Figura 1. Resolução do problema de pintura de um mapa com restrição de cores diferentes para regiões fronteiriças.*

No exemplo da Figura 1, somente três cores foram utilizadas: vermelho, verde e azul. Com isso, para a modelagem de um CSP, estes seriam os valores do elemento **domínio**. Os outros dois elementos são a lista com o nome de cada região, para as **variáveis**, e as ligações entre as regiões que fazem fronteira, para as **restrições** (tais ligações podem ser apresentadas em forma de grafo – veja Figura 2).

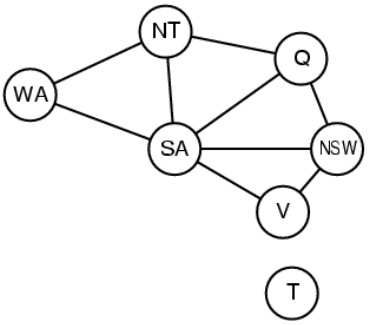


*Figura 2. Possível representação em grafo das restrições do problema de pintura das regiões de um mapa com restrição de cores diferentes para regiões fronteiriças.*

A Figura 3 apresenta um pseudocódigo para implementação de um algoritmo de CSP.

```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to Constraints[csp] then
         add { var = value } to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failue then return result
         remove { var = value } from assignment
   return failure
```

*Figura 3. Pseudocódigo para implementação de um algoritmo de CSP.*

A partir do pseudocódigo da Figura 3, é possível identificar alguns componentes importantes do algoritmo, isola-los e pensar em uma implementação genérica. Aqui, cada um desses componentes (com exceção do **domínio**) será pensado como uma função:

- Função pare verificar se o problema foi resolvido.
- Função para pegar a próxima variável sem valor.
- Uma lista contendo os valores do domínio do problema.
- Função para verificar se uma variável pode receber um valor do domínio.
- Função para atribuir um valor do domínio para uma variável.
- Função para remover um valor do domínio para uma variável.

## Implementação

Todo o projeto foi implementado utilizando somente HTML, CSS e JavaScript, sem biblioteca ou *framework* de terceiro. A maior parte da programação feita em JavaScript segue o paradigma Orientado a Objetos, com a definição de diferentes classes para criação de objetos com responsabilidades bem definidas. As subseções a seguir explicam um pouco sobre a implementação das classes criadas.

### Grafo

A implementação desse tipo abstrato de dado (TAD) está dividida em duas classes. Node, que representa um nó do grafo e possui um identificador (gerado automaticamente e utilizado pela aplicação para facilitar algumas rotinas – não deve ser alterado pelo desenvolvedor), um valor, um peso e as conexões com outros nós do grafo. Graph, que possui uma lista de nós. Ambas as classes possuem funções para a criação e manipulação de diferentes instâncias de grafos.

É importante dizer que, o valor da propriedade *value* de um nó do grafo não precisa ser obrigatoriamente de um tipo primitivo (um inteiro, um ponto flutuante, um booleano, etc). Essa propriedade pode ser outro objeto, ou até mesmo uma função, porque a linguagem permite essa flexibilidade. Portanto, o tipo do valor que será atribuído para cada nó é definido de acordo com a necessidade do problema que deseja solucionar.

### CSP

Essa classe possui a implementação do algoritmo apresentado no pseudocódigo da Figura 3. Ela foi implementada de maneira genérica e necessita de alguns parâmetros para a criação de instâncias a partir dela. Uma vez modelado o problema que se quer resolver como uma CSP, e instanciado o objeto, o algoritmo para a sua resolução pode ser executado invocando a função ***solve*** da instância criada – essa função retorna um valor booleano indicando se foi possível encontrar a solução do problema ou não. A seguir, da um dos argumentos que devem ser passados na construção do objeto são explicados (os valores entre parêntesis no final são os nomes de cada um dos argumentos na implementação).

- Um grafo, contendo os dados do problema (*graph*).

- Uma lista, contendo os valores do domínio do problema (*domain*).
- Uma função para verificar se o problema foi solucionado. Essa função é invocada durante a execução da implementação do pseudocódigo da Figura 3, que é executado a partir da chamada da função *solve*. Ela deve receber um grafo e retornar um valor booleano (*checkCompletionFunction*).
- Uma função para verificar se é possível atribuir um valor do domínio à um nó do grafo. Essa função é invocada durante a execução da implementação do pseudocódigo da Figura 3, que é executado a partir da chamada da função *solve*. Ela deve receber um grafo, um nó do grafo, um valor do domínio e retornar um valor booleano (*checkAssignmentFunction*).
- Uma função para pegar o próximo nó do grafo que ainda não possui valor. Essa função é invocada durante a execução da implementação do pseudocódigo da Figura 3, que é executado a partir da chamada da função *solve*. Ela deve receber um grafo e retornar um nó do grafo ou nulo (*getNextUnassignedNodeFunction*).
- Uma função para atribuir um valor do domínio à um nó do grafo. Essa função é invocada durante a execução da implementação do pseudocódigo da Figura 3, que é executado a partir da chamada da função *solve*. Ela deve receber um grafo, um nó do grafo, um valor do domínio e não possui retorno (*assignFunction*).
- Uma função para remover um valor do domínio de um nó do grafo. Essa função é invocada durante a execução da implementação do pseudocódigo da Figura 3, que é executado a partir da chamada da função *solve*. Ela deve receber um grafo, um nó do grafo, um valor do domínio e não possui retorno (*unassignFunction*).

**CSP – exemplo**

No arquivo **"./js/csp/map-colors.js"**, a classe apresentada anteriormente foi utilizada para resolver o problema de pintura de um mapa com a restrição de cores diferentes para áreas fronteiriças (problema apresentado na seção da Figura 1). Para tanto, foi criado um grafo representando as cinco regiões do Brasil, e as ligações entre elas. Também foi criado uma nova classe, *Place*, com as propriedades *name* e *color*, para ser utilizada como tipo de valor dos nós do grafo. Dessa maneira, cada região do país é representada por um nó do grafo, cujo valor é uma instância de *Place*, contendo o nome da região e a propriedade *color*

inicialmente nula. A seguir, a explicação da implementação de cada um dos argumentos do CSP para resolver esse problema em específico:

- ***graph***: representa o mapa das cinco regiões do brasil.
- ***domain***: uma lista contendo as *strings red*, *green* e *blue* (cores para pintar o mapa).
- ***checkCompletionFunction***: verifica se todos os nós do grafo já possuem uma cor atribuída. Se sim, significa que a solução foi encontrada.
- ***checkAssignmentFunction***: verifica se o nó possui alguma conexão cujo valor da cor é igual ao valor que está se tentando atribuir para o nó em questão. Se encontrar alguma conexão com esse valor, significa que não é possível atribuir esse valor para esse nó.
- ***getNextUnassignedNodeFunction***: retorna o primeiro nó do grafo cujo valor da propriedade *color* ainda é nulo.
- ***assignFunction***: atribui o valor recebido à propriedade *color* do objeto na propriedade *value* do nó do grafo em questão.
- ***unassignFunction***: atribui nulo à propriedade *color* do objeto na propriedade *value* do nó do grafo em questão.

**GameBoard**

No arquivo **"./js/game/board.js"**, as classes BoardPosition, BoardNode e Board, assim como o objeto auxiliar ElementType, foram criados.

BoardPosition possui as propriedades *x*, *y* e *element* (cujo valor, inicialmente nulo, deve ser uma das opções definidas em ElementType). Objetos dessa classe representam uma posição no tabuleiro e se há algum elemento ou não nessa posição. BoardNode é uma extensão da classe Node e seu valor deve ser um objeto da classe BoardPosition. Essa classe também define algumas funções auxiliares para a manipulação dos nós, como por exemplo, pegar um vizinho à direita, à esquerda, acima ou abaixo da posição de um nó, de acordo com os valores da coordenada do objeto BoardPosition, armazenado na propriedade *element* de cada instância de BoardNode. ElementType define os valores *person*, *coronavirus*, *obstacle* e *coronaVirusPath*. Cada um deles representa um dos elementos que podem estar no tabuleiro: a pessoa (triangulo amarelo), o coronavírus (circulo verde), os

obstáculos (losangos cinzas) e o caminho percorrido pelo coronavírus para chegar à pessoa (cirulos verdes).

**GameView**

**GameViewController**