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 também 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 acessando 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 arquivos e pastas:

- 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** (*Cascading Style Sheet*)
    - **style.css**: arquivo de estilo da página index.html.
    - **rules.css**: arquivo de estilo da página rules.html.
  - Pasta **js** (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 CSP 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 (imagens).

- Pasta ***resources***
  - Pasta ***images***
    - **coronavirus.svg**
    - **infectedperson.svg**
    - **obstacle.svg**
    - **person.svg**
    - **removeosbstacle.svg**

Solução de problemas por restrições

Esta seção foi escrita baseada no conteúdo do capítulo VI do livro "*Artificial Intelligence A Modern Approach Third Edition*", de Stuart Russel e Peter Norving (edição publicada 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 a Figura 1.

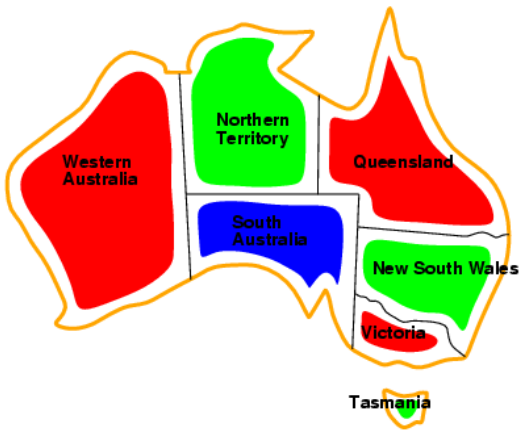

Figura 1. Resolução do problema de pintura de um mapa com restrição de cores diferentes para regiões fronteiriças.

No exemplo da Figura 1, somente três cores foram utilizadas: vermelho, verde e azul. Com isso, para a modelagem de um CSP, esses 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 a Figura 2).

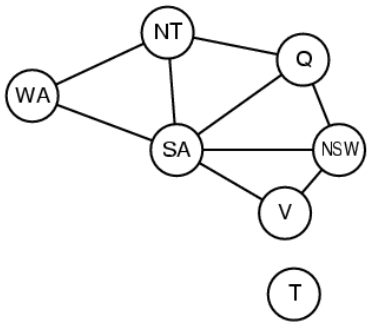

Figura 2. Possível representação em grafo das restrições do problema de pintura das regiões de um mapa com restrição de cores diferentes para regiões fronteiriças.

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 de 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 distintas. A Figura 4 apresenta um diagrama de classes da aplicação, que também pode ser encontrado a partir do *link* https://app.creately.com/diagram/RfcR0T5qDaD/view. A seguir, uma explicação breve de cada uma das classes da aplicação e suas funções.

- **Graph**: para criar objetos que possibilitam armazenar dados em uma estrutura de grafo.
- **Node**: para criar objetos que compõem um grafo.
- **CSP**: para criar objetos que possibilitam resolver diferentes problemas a partir da implementação do algoritmo de CSP apresentado da seção anterior.
- **Board**: uma extensão da classe Graph, permite a criação de objetos que representam o tabuleiro do jogo em uma estrutura de grafo.
- **BoardNode**: uma extensão da classe Node, permite a criação de objetos que compõem o grafo que representa o tabuleiro.
- **BoardPosition**: para criar objetos que armazenar as coordenadas da posição de cada casa do tabuleiro, bem como se há algum elemento nessa posição.
- **ElementType**: usado para representar no grafo os elementos que podem ser colocados no tabuleiro.
- **GameView**: permite criar o objeto que realiza todas as manipulações de interface: criação do tabuleiro, animações, adicionar elementos, remover elementos.
- **GameViewController**: permite criar o objeto que gerencia todas as ações do jogador: clique em uma posição do tabuleiro, clique no botão de jogar, clique em elementos do jogo.

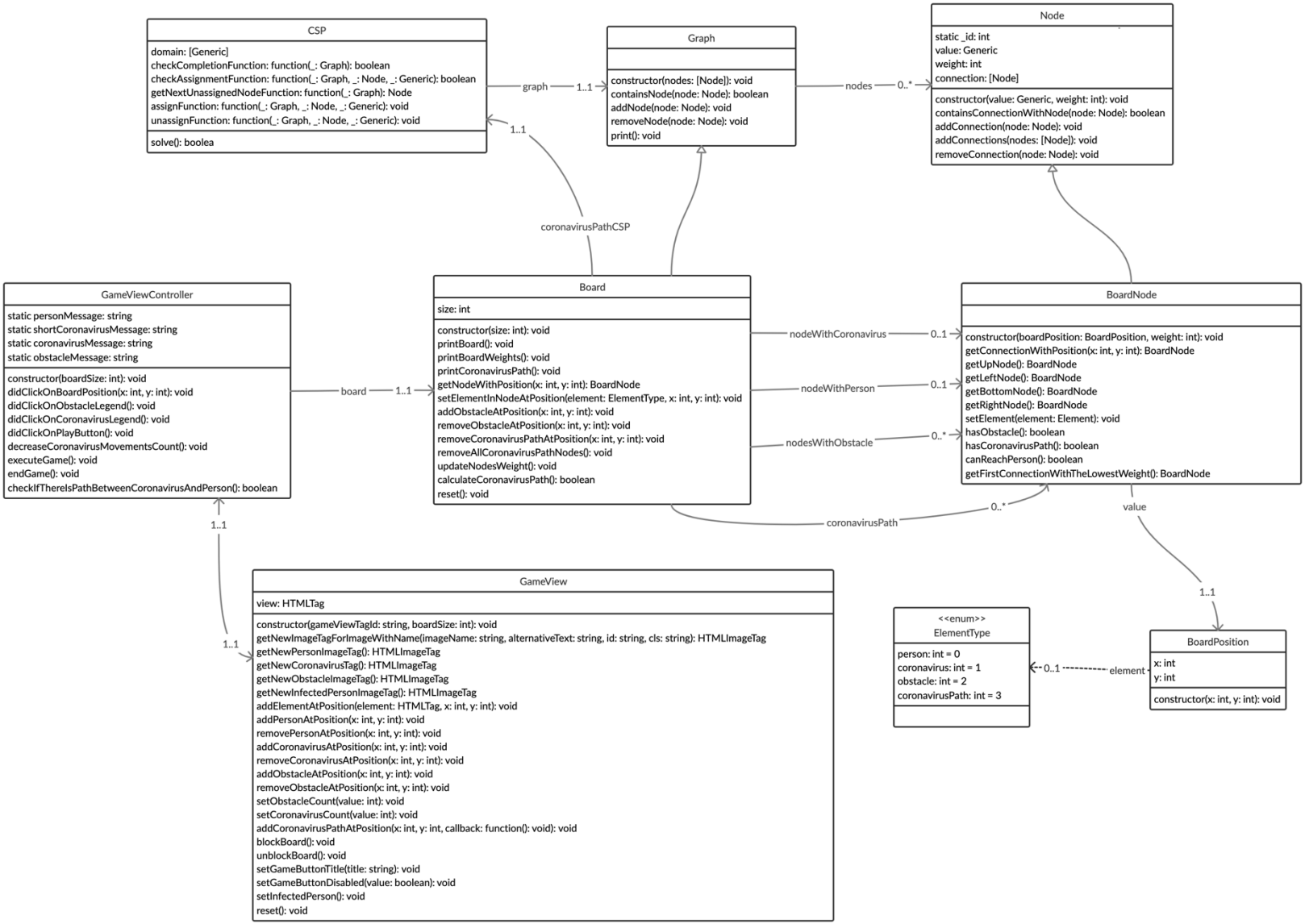

Figura 4. Diagrama de classes da aplicação.

Caminho do Coronavírus até a Pessoa

Esta seção explica o algoritmo implementado para definir o caminho da Coronavírus até a Pessoa no jogo. Ele é dividido em dois passos principais. Primeiro, o peso de cada nó no grafo que representa o tabuleiro é atualizado. E por último, o menor caminho entre o Coronávírus e a Pessoa é computador.

O processo de atualização do peso dos nós do tabuleiro começa a partir do nó no qual a pessoa se encontra. O peso deste nó é zero. A partir dele, o peso dos nós vizinhos é computado. O peso do nó vizinho à um nó é igual ao peso do nó atual mais uma unidade. Assim, os nós vizinhos ao nó no qual a pessoa se encontra terão peso um. Esse processo é repetido recursivamente para todos os nós do tabuleiro.

Inicialmente todos os nós possuem peso nulo. Caso um nó possua um obstáculo, seu peso continua nulo, porque essa posição não pode ser usada como parte do caminho – o mesmo se sucede para o nó no qual o Coronavírus se encontra. Ao final desse processo, todos os nós terão um peso. Veja um exemplo de resultado desse processo na tabela a seguir (em um tabuleiro 5 por 5 com a Pessoa e o Coronavírus nas duas extremidades da diagonal principal.

Pessoa	1	2	3	4
1	1	2	3	4
2	2	2	3	4
3	3	3	3	4
4	4	4	4	Coronavírus

A seguir, o mesmo exemplo, mas agora com alguns obstáculos no tabuleiro.

Pessoa	1	2	3	4
1	1	2	Obstáculo	5
2	2	Obstáculo	7	6
2	2	Obstáculo	8	Obstáculo
2	2	3	4	Coronavírus

A seguir, o mesmo exemplo, mas agora isolando o Coronavírus da Pessoa. Não é possível calcular um caminho entre os dois pontos. Esse cenário é evitado no jogo, e caso o jogador o reproduza um alerta é exibido dizendo que ele não pode fazer isso.

Pessoa	1	2	3	Obstáculo
1	1	2	Obstáculo	
2	2	Obstáculo		
2	Obstáculo			
Obstáculo				Coronavírus

A partir dessa tabela é possível computar o menor caminho entre a Pessoa e o Coronavírus. Na aplicação isso é feito começando a partir do nó onde se está o Coronavírus. Então, busca-se sempre o vizinho cujo peso é o menor, sendo que deve ser menor que o peso do nó atual também. Ao encontrar, o nó é marcado como possuindo um elemento do tipo “Caminho do Coronavírus”, e é armazenado em uma lista para que seja possível realizar as animações posteriormente.

A atualização do peso dos nós do tabuleiro é feito na classe ***Board***, na função ***updateNodesWeight***. A definição do menor caminho é feita na mesma classe, na função ***calculateCoronavirusPath***. Esta, utiliza um CSP para resolver o problema. O objeto que executa o CSP para a definição do caminho é construído no construtor da classe ***Board***. A seguir são explicados dos componentes.

- **Domínio**: uma lista contendo o valor definido em ***ElementType.coronavirusPath***.
- ***checkCompletionFunction***: verifica se o último nó do caminho do Coronavírus já pode alcançar o nó no qual a Pessoa está.
- ***checkAssignmentFunction***: verifica se o nó não possuí um obstáculo.
- **g*etNextUnassignedNodeFunction***: caso seja o início do caminho, retorna o primeiro nó, vizinho do nó no qual está o Coronavírus, cujo peso é o menor entre todos os vizinhos. Se o caminho já estiver em construção, realiza a mesma operação, mas com o último nó do caminho.
- ***assignFunction***: insere o elemento no nó e o armazena como último nó do caminho do Coronavírus.
- ***unassignFunction***: remove o elemento do nó e o remove da lista do caminho do Coronavírus.