1 - ... capa ...

2 - ... contra-capa ...

3 - Introdução:

Nosso trabalho consiste na criação de um código capaz de solucionar o *8-puzzle problem*, brinquedo popularmente conhecido, de modo eficiente. Para isso, utilizamos o algoritmo A\*, que utiliza de uma fila de prioridades, evitando a computação de resultados que não utilizaríamos posteriormente. De modo geral, tal algoritmo gera uma busca diagonal pela resposta, tornando-o mais eficiente.

Utilizamos de algumas funções previamente criadas e contidas na biblioteca *cl-heap,* principalmente para a utilização da fila de prioridades. Entretanto, não utilizamos outra biblioteca, e todas as funções restantes foram criadas exclusivamente para o trabalho. Tendo em vista todo o processo de criação do código, estaremos o explicando posteriormente.

Portanto, deixaremos claro toda a construção do código, além de decisões que tomamos durante o processo. É demasiado importante que o código fique o mais claro possível, para que, em algum outro momento, ele possa ser usado como meio de consulta para alguém interessado em computação e que tenha esse desafio em mente.

4 – Parte 1: decisões

Durante a criação do código, nos deparamos com diversos problemas e, consequentemente, desenvolvemos soluções para tais. Além da apresentação de um novo algoritmo de resolução, recebemos um trabalho do zero, e por mais que a documentação dele fosse estruturada, ainda seria um ponto de dificuldade no começo. Por fim, a apresentação da resposta foi um problema maior do que imaginávamos, mas também encontramos uma solução para tal.

Primeiramente, devemos ressaltar a eficiência do algoritmo A\* para o *puzzle*, tendo em vista que tal busca com peso torna o código bem mais simples e leve. A seguir, teremos uma foto da maneira como o algoritmo funciona para o problema apresentado.

Após entendermos o algoritmo e o problema apresentado, partimos para o começo do código. Buscamos criar uma função principal e, a partir dela, as funções necessárias para a resolução, tal como a determinação de uma medida para o peso que levaríamos em conta na fila de prioridades, e a criação de uma matriz solução. No entanto, essa última ideia foi descartada durante o processo.

Enfim, também utilizamos a *priority-queue* para incluirmos o estado atual do jogo e desmembramos em novos casos, os filhos desse estado – todas as mudanças possíveis. Ademais, durante o processo de criação, para facilitar a utilização de vários códigos, optamos por uma formatação linear da matriz do estado do jogo, de modo que nossa busca continuaria sendo eficiente, e que tornasse alguns códigos triviais.

5 – Parte 2: código

... foto da classe ...

Primeiramente, começaremos criando uma classe *board*, que armazenará o estado atual do jogo, o estado anterior – pai –, a *hamming distance* – método para determinação de peso que aumenta em um para cada número fora de posição –, o número de movimentos naquele estado e, por último, a posição do zero dentro da *board*.

... continuação do código (até hamming dist) ...

Optamos por determinar a *\*game-tree\** – fila de prioridades que contém as boards analisadas –, como um parâmetro fixo, já que a utilizaríamos durante diversas partes do código. Além disso, optamos por definir nomes de fácil entendimento e visualização, tais como *board-obj* e *board-array*, evitando o conflito e erros por confusão. Finalmente, o código da *hamming distance*, como dito anteriormente, responsável por determinar os pesos.

... make-move, enqueue-child, gen-neighbors ...

Tais funções estão conectadas e tem como objetivo principal a geração dos vizinhos – *boards* filhos –, além de armazená-los dentro da *\*game-tree\**. Para isso, os nomes se tornam bem específicos e explicitam de maneira eficiente a função de cada uma delas.

... is-solvable ...

Uma das funções menos eficientes dentro do código, mas de grande importância. Tendo em vista que estaríamos lidando com loops grandes para casos em que o jogo estivesse totalmente embaralhado, deveríamos ser capazes de determinar se um possível estado é, ou não, solucionável. Para o caso do tamanho n ímpar da matriz nxn, só precisamos determinar a soma do número de inversões (linearmente falando, o número de números que deveriam estar na frente de um número i, mas não estão. Determinando tal valor, caso ele seja par, o jogo é solucionável. Caso contrário, não! Para n par, em uma matriz nxn, teremos que lidar com outra situação ...

... solve, solve-aux ...

Utilizando todas as funções supracitadas, criamos a função principal do código, responsável por devolver a solução – ou seja, o número de passos e os passos realizados –, para alcançar o resultado do jogo. Tentamos torna-las o menos preenchidas possível, de modo que fosse fácil de entender o código olhando para seu corpo e observando as diversas funções e a maneira com a qual elas se conectam.

6 – Parte 3: considerações finais

Nós buscamos realizar o trabalho de maneira conjunta, nos encontrando para desenvolver o código. Tanto Matheus, quanto Vinícius foram capazes de trabalhar no código, corrigindo detalhes que o outro não percebia, mas que alguém de fora conseguia visualizar mais facilmente. Não determinamos funções específicas para cada um, mas sempre estivemos trocando diversas ideias sobre as melhores possibilidades para o nosso código, até alcançarmos o resultado.

Acreditamos que esse trabalho foi capaz de nos dar muito mais bagagem para solucionar problemas maiores do que simplesmente um teste de conhecimentos individual. Fomos capazes de discutir ideias, mudar completamente a ideia do código, compartilhar *insights* e buscar a melhor solução possível.

Logo, temos a certeza de que esse trabalho foi de grande aprendizado e nos animamos mais ainda a continuar programando, descobrindo novos códigos, linguagens e algoritmos, tendo em vista a gama de possibilidades que surgem quando nos dedicamos.