Resolutor *8 Puzzle*

Bruno Alano Medina
(Dated: October 26, 2015)

Desenvolvimento de Projeto para resolução do problema conhecido como *Sliding Puzzle*, utilizando a linguagem de programação C e estrutura de dados especializadas.

Utilização: `make && ./8puzzle < test.txt`

I. INTRODUÇÃO

Para o desenvolvimento deste projeto, acreditei que a melhor estrutura de dados até o momento ensinada seria a utilização de uma Fila (*Queue*). Utilizando uma técnica chamada de Busca em Largura, podemos retornar o resultado em poucas operações sem a necessidade da implementação de uma heurística. Caso fossemos implementar uma heuristíca, a que mostra maior performance neste tipo de problema é o algoritmo `A*`, porém, poderia ser implementado algo mais simplista como `minimax`.

II. BOAS PRÁTICAS

O projeto foi inteiramente documento utilizando o padrão de documentação ANSI C, e uma mesclagem com o estilo de Java Doc, também sendo aceito por diversos geradores de documentação para C, como tais utilizado pela *Google*.

A convenção de nomeclatura para variáveis utilizada foi KR C, onde variáveis possuem suas partes nominais separadas por *underscores* e constantes com todas as letras capitalizadas.

Ao gerenciamento de compilação foi utilizado o GNU Make, porém, pela simplicidade de tal arquivo, ele pode ser utilizado por qualquer demais program que simule a mesma atividade. Poderia ser utilizado *softwares* mais complexos, como `CMake` ou `SCons`, porém, não foi visto a necessidade de tal.

III. ESTRUTURA DE DADOS

A. Queue

A estrutura de Fila foi implementada de modo que mantenha a performance da mesma e sua compatibilidade com qualquer tipo de dados. Para tanto, utilizamos a recepção de ponteiro para `void`, em que no caso de sistemas 64 bits, suporta qualquer dado em 8 bytes, e em 32 bits em 4 bytes.

A nomenclatura das funções segue o padrão de tal estrutura.

B. Stack

A estrutura de Pilha também foi implementada de modo a suportar qualquer tipo de dado, sendo o `cast` deixado ao seu utilizador.

IV. ALGORITMO

O Algoritmo baseia-se nos estados do jogo (*Ver estrutura State*). Baseando-se no Estado atual, ele gerá no máximo 4 possíveis movimentos à fim de não retornar em um processo anterior (parente), portanto, geralmente são gerados 3 movimentos.

Estes 3 possíveis Estados são enviados à uma queue onde iremos iterar sobre a mesma, a fim de verificar todos os Estados nela presente procurando se o jogo finalizou. Utilizando o método *dequeue*, retiramos o item prioritário e geramos seus possíveis movimentos, adicionando-os ao fim da queue. Desse modo, sempre verificaremos todo o nível da árvore antes de descê-la, evitando então uma Busca em Profundide, em que acarretaria em um *overflow* de memória (exceto se aplicar heurística).

V. MELHORIAS

A única melhoria essencial ao projeto seria implementar o algoritmo A*, onde utilizariamos uma fila prioritário, implementando-a com uma Fibonacci Heap. Por ser um trabalho universitário, não viu-se a necessidade de tal implementação.

Uma melhoria em código seria utilizar um método menos manual para a verificação do fim do jogo, como a função *is_finished* no arquivo *src/game.c*. Uma função que foi otimizada neste quesito por mim foi a *generate_state_sucessors* no arquivo *src/state.c*.

VI. CRÉDITOS

Foi utilizado o livro *Introduction to Algorithms, Cormen et al.* e *Algorithms, Sedgewick* como referência na implementação de algumas estruturas.

O código foi publicado em meu Github pessoal(github.com/brunoalano), onde todas as funções foram implementadas pelo mesmo.