UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Gabriel Simmel Nascimento
Rafael Augusto Monteiro

Relatório
16 Sliding Puzzle Game

São Carlos

1. O Programa

1.1. Instruções de Compilação

Utilize o comando Make na pasta raíz do projeto para compilar o programa

```
make
```

1.2. Instruções de Execução

Utilize o comando Make run para executar o programa. O programa pode receber o número máximo de passos da solução como argumento. No exemplo abaixo, o número passado foi 30.

```
make run 30
```

Caso o número não seja passado, o programa utilizará 50 passos como padrão. O exemplo abaixo ilustra a execução com 50 passos ao máximo

```
make run
```

Para executar os casos de teste disponíveis, utilize o comando "<"

```
make run Y < ../testeX.in
```

Onde Y é o número de passos desejado e X é um número entre 1 e 3 (recomendamos apenas 1 e 2 para força bruta).

1.3. Estruturas de Dados

Para cada solução descrita abaixo, a estrutura de dados *Table* foi utilizada para armazenar um estado do tabuleiro. Ainda, para a solução com A*, foi utilizada uma estrutura auxiliar *Node*, que guarda um estado do tabuleiro, junto com o custo para alcançá-lo e o conjunto de passos que levaram à esse estado.

Para um dado tabuleiro, é possível calcular a distância de Manhattan para o estado buscado. A função *ManhattanDist* realiza esse cálculo.

1.4. Visão Geral do Programa

O programa receberá da *stdin* um número *n* de casos de teste. Em seguida, será feita a leitura de *n* matrizes 4x4 contendo um tabuleiro de jogo a ser resolvido. O programa tentará resolver cada tabuleiro lido chamando a função correspondente à lógica utilizada. Em seguida, o programa exibirá o caminho encontrado até a solução ou exibirá a string "*This puzzle is not solvable*." caso nenhuma solução seja encontrada.

1.5. Força Bruta (DFS)

A implementação força bruta consiste em explorar todo o espaço de possibilidades utilizando chamadas recursivas. Vendo o espaço de possíveis movimentos como um grafo, o algoritmo executa uma busca em profundidade (Depth First Search, ou DFS). A função *solves*, que realiza a DFS, realiza três passos importantes:

1. Verifica se o estado atual é o buscado (ou seja, se a distância de Manhattan é zero). Caso seja, retorna *true*.
2. Verifica se o número de passos dados é maior ou igual ao número máximo de passos dado. Caso seja, retorna *false*.
3. Chama a função *solves* para todas as direções possíveis. Caso o retorno dessa chamada seja *true*, acrescenta a direção à solução.

Dessa forma, o programa realiza chamadas que retornam *false* ao atingir um estado de profundidade máxima e que não é uma solução, e quando uma chamada retorna *true,* as chamadas que à chamaram acrescentam a letra com a direção utilizada para alcançar aquele estado na solução e retornam *true.*

Ao final, basta inverter a string de solução para encontrar o caminho correto (pois as chamadas recursivas marcam o caminho partindo do final e indo em direção ao começo).

1.6. Implementação Sofisticada (A*)

A implementação mais sofisticada utiliza uma busca A* ao invés de uma DFS. Para isso, é utilizada uma fila de prioridade que leva em consideração um custo *f(x)* para cada estado. O custo *f(x)* é dado por $f(x) = g(x) + h(x)$, onde:

- *f(x)*: Custo de um estado *x* do tabuleiro.
- *g(x)*: Número de passos para atingir um estado *x*
- *h(x)*: Distância de Manhattan do estado *x* até o estado final buscado.

A heurística *h(x)* é admissível pois subestima o custo para alcançar o estado buscado (a distância de Manhattan considera o número mínimo de movimentos horizontais e verticais para que cada peça do tabuleiro alcance a posição correta do tabuleiro, que sempre será menor ou igual ao número de passos necessários).

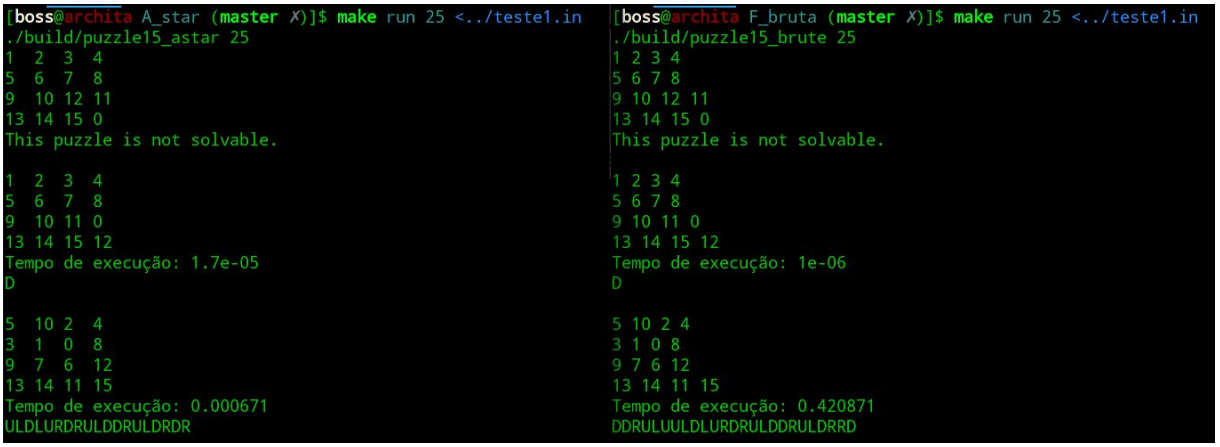
A execução começa inserindo o estado inicial na fila de prioridades. Em seguida, as seguintes operações são realizadas enquanto ainda houver estados disponíveis na fila de prioridades:

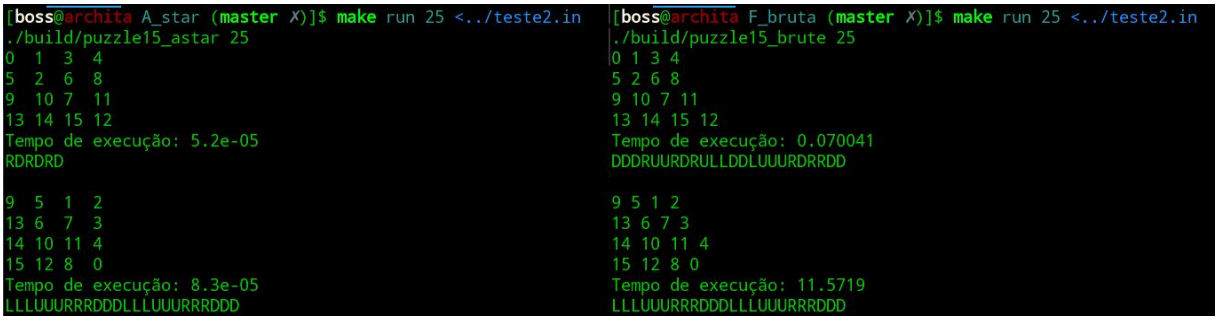
1. Retira-se o estado *n* de menor custo
2. Verifica-se se o estado é o estado buscado. Caso seja, retorna a string contendo o caminho até encontrar aquele estado
3. Verifica-se se o estado *n* excede o número máximo de passos. Caso exceda, a execução retorna ao passo 1.
4. Empilham-se todos os estados relativos aos possíveis movimentos a partir do estado *n.*

Ao final, a função retornará uma string vazia (caso nenhuma string seja encontrada) ou uma string contendo os passos até a solução encontrada.

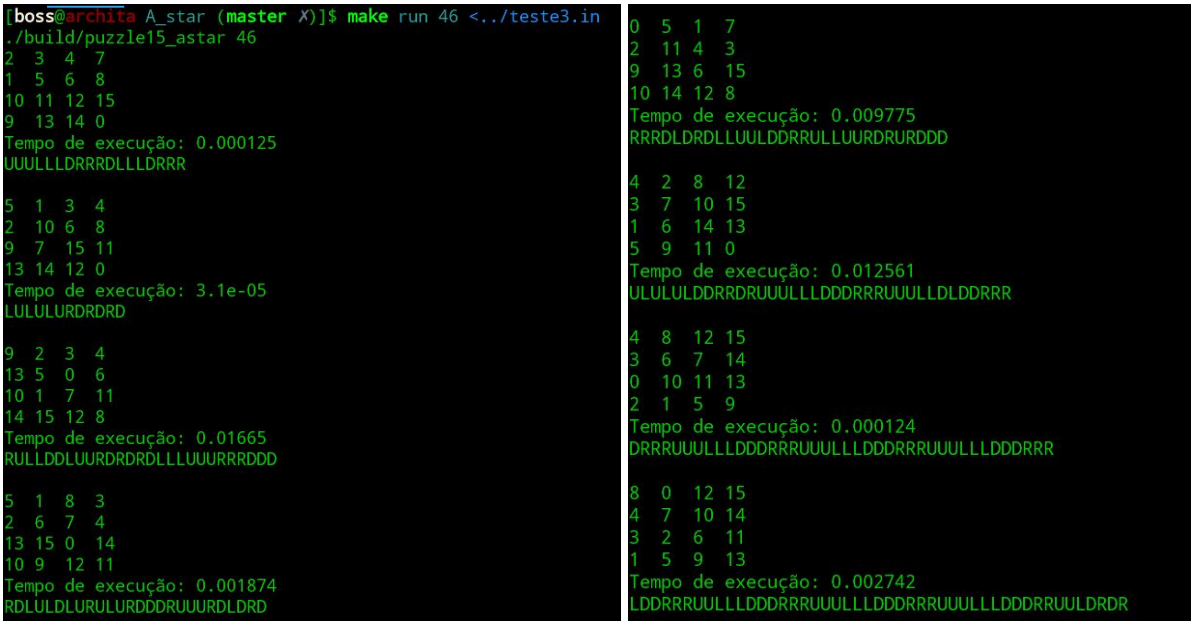
2. Execução e Testes

As execuções abaixo foram feitas comparando o tempo de execução entre o algoritmo A* (esquerda) e o algoritmo de força bruta (direita). Para tornar o tempo de execução do algoritmo de força bruta tolerável, foram utilizados no máximo 25 passos.





A imagem abaixo mostra a execução do algoritmo A* para casos mais complexos. Foram utilizados no máximo 46 passos.



Os casos de teste para ambos os programas estão disponíveis nos arquivos *“teste1.in”* e *“teste2.in”*. Os casos de teste para o algoritmo A* está disponível no arquivo *“teste3.in”*. Para executar os casos de teste via makefile, basta executar o seguinte comando make, a pa:

```
make run < ../testeX.in
```

Onde X é o número do arquivo de teste a ser utilizado.

3. Comentários sobre as soluções

Tanto a solução de força bruta quanto a solução mais sofisticada possuem a complexidade *O(n) = 4^n*, onde *n* é o número máximo de passos permitidos. Porém, A solução com A* permite uma busca mais inteligente, que alcança o resultado de forma mais rápida que a força bruta.

Com o intuito de minimizar o tempo de execução, o algoritmo A* finalizará a busca caso o custo do movimento de um estado selecionado seja maior que o limite definido. Isso ocorre pois o custo *f(x)* de um estado até o destino sempre será menor que o número de passos necessários para atingir o destino.

Ainda, por usarmos uma heurística admissível, é garantido que o algoritmo A* encontrará uma resposta ótima para o número de passos dado. Enquanto isso, o algoritmo DFS retornará a primeira solução encontrada, que não necessariamente será ótima.