UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO - ICMC/USP
BACHARELADO EM CIÊNCIAS DE COMPUTAÇÃO

RELATÓRIO DO PROJETO 1: BUSCA TRADICIONAL E BUSCA A*

SCC0218 - ALGORITMOS AVANÇADOS E APLICAÇÕES
PROF. JOÃO DO ESPÍRITO SANTO BATISTA NETO

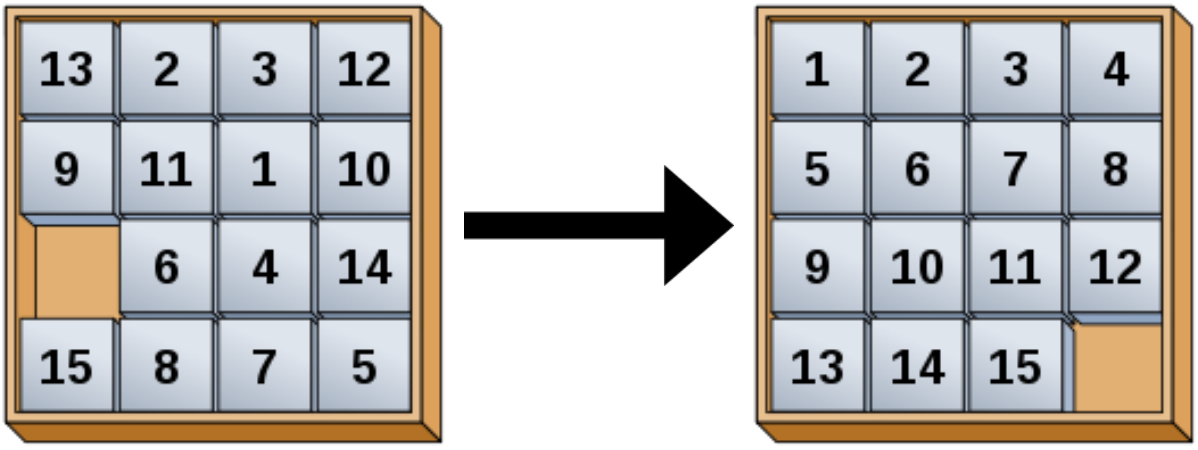
GABRIEL ROMUALDO SILVEIRA PUPO
NÚMERO USP: 9896250

SÃO CARLOS
OUTUBRO/2017

Introdução **2**
1 Detalhamento da lógica implementada 3
1.1 Busca em largura 3
1.2 Busca A* 3
2 Comparação dos tempos de execução 4

Introdução

O 15-puzzle, conhecido no Brasil como “O jogo do 15”, é um tipo de quebra-cabeças que consiste, na sua versão mais tradicional, em uma placa oca com 15 peças numeradas de 1 a 15, que podem ser movidas para cima, para baixo, para a esquerda e para a direita, e um espaço vazio. O objetivo do puzzle é: dada a configuração inicial arbitrária das peças, movê-las de forma a chegar na configuração final em que todas as peças estão em ordem crescente, com o espaço vazio no canto inferior direito.



Configuração inicial Configuração final

O 15-puzzle também pode ser resolvido por um computador, utilizando algoritmos de busca eficientes que, por meio de heurísticas convenientes, procuram caminhos que mais se acercam da configuração final e solucionam o quebra-cabeças no menor número de movimentos possível.

Para o escopo deste projeto, será apresentado o algoritmo de busca A*, um dos mais eficientes para resolver um 15-puzzle. Ele será comparado com outro algoritmo de busca que, apesar de também encontrar o menor caminho possível para a configuração final, é menos eficiente e pode ter um tempo de execução extremamente grande com estados iniciais mais complexos: a busca em largura (do inglês, *breadth-first search*).

1 Detalhamento da lógica implementada

1.1 Busca em largura

O algoritmo de busca em largura inicializa uma fila, que será usada durante a execução e conterá estados do quebra-cabeças a serem visitados e um vector STL, que armazenará todos os estados calculados pelo algoritmo − a fim de checar por arranjos já tentados e impedir *loops* infinitos no algoritmo − e um ponteiro para o estado inicial do *puzzle*, que será inserido na fila e registrado no vector.

Em seguida, a partir do estado inicial, o algoritmo encontra as próximas posições possíveis para o espaço vazio e então verifica se alguma dessas posições resolve o quebra-cabeças. Se esse não for o caso, o próximo estado a ser visitado pela busca é então enfileirado e registrado no vector, contanto que ele já não tenha sido passado anteriormente pelo algoritmo. Isso é repetido para os outros candidatos a serem visitados na próxima iteração da busca. Por fim, o estado atual sai da fila e o próximo estado na primeira posição dela é visitado.

A ineficiência deste algoritmo para esta situação deve-se ao fato da busca ser **desinformada**, ou seja, ela visita todos os arranjos possíveis do quebra-cabeças para tentar chegar na configuração final sem seguir nenhuma heurística que determine o melhor caminho a ser seguido primeiro, tentando ir em todos eles e só mudando de ideia ao se esgotarem a possibilidades de seguir adiante. Logo, o algoritmo acaba checando caminhos indesejáveis em excesso, perdendo muito tempo para solucionar o problema.

1.2 Busca A*

O algoritmo de busca A* utiliza duas listas: uma aberta, que conterá todos os estados gerados pela busca, e uma fechada, que conterá os estados expandidos por ela. Em outras palavras, a aberta − implementada como uma list STL − guarda estados que são sucessores de estados expandidos e que ainda serão processados, e a fechada − implementada como um vector − guarda os que já foram expandidos e não serão mais analisados, para evitar *loops* infinitos.

Para cada estado apto a ser o candidato do próximo movimento, calcula-se o valor de seu **f(n)**, que é a soma de seu **g(n)**, a distância exata da configuração inicial para o estado atual, com o seu **h(n)**, obtido a partir do uso de uma heurística para aproximar a distância do atual para a configuração final. O g(n) do estado atual é igual ao g(n) do estado anterior acrescido de uma unidade, já o seu h(n) é obtido pelo somatório da distância de Manhattan de cada peça para a sua posição desejada no estado final, isto é:

$$h(n) = \sum_{n=1}^{16} |i_{na} - i_{nf}| + |j_{na} - j_{n_f}|$$

em que i_{n_a} e j_{n_a} são a linha i e a coluna j de uma peça n no estado atual, e i_{n_f} e j_{n_f} são a linha i e a coluna j de uma peça n no estado final. Note que o cálculo da distância desconsidera o espaço vazio, a fim de melhorar a consistência da heurística.

O cálculo da distância de Manhattan é uma heurística admissível porque ela só considera movimentos em quatro direções (norte, leste, sul e oeste), o que é compatível com o tabuleiro do 15-puzzle, que impede fisicamente a movimentação das peças nas diagonais. Ele também é mais eficiente que uma outra heurística aplicável no caso do quebra-cabeças, a distância de Hamming − que conta todas as peças fora da posição final em cada caso − pois esta visita muito mais estados indesejados por não considerar o quão longe uma peça no lugar errado está da sua posição final, o que a leva a considerar que um tabuleiro com uma peça fora do lugar, mas longe da sua posição final, tem mais prioridade sobre um com duas peças fora, mas que estão bem mais próximas das suas posições esperadas.

2 Comparação dos tempos de execução

Foram executadas algumas baterias de teste para ver o tempo de execução de cada algoritmo para resolver uma determinada configuração inicial de um 15-puzzle. Gerou-se aleatoriamente alguns arranjos de peças, e cada arranjo variava no número de passos necessários para resolvê-lo.

Para um arranjo resolvível em 5 passos, a diferença era pequena, mas já notável:

```
main@virtualpupo:~/Desktop/15-puzzle$ ./15p-basic
1
1 2 3 4
5 6 0 8
9 11 7 12
13 10 14 15
DLDRR

Execution time for this set: 0.001023 s
```

```
main@virtualpupo:~/Desktop/15-puzzle$ ./15p-astar
1
1 2 3 4
5 6 0 8
9 11 7 12
13 10 14 15
DLDRR

Execution time for this set: 0.000185 s
```

Com um resolvível em 10 passos, a diferença começava a ser bem mais perceptível:

```
main@virtualpupo:~/Desktop/15-puzzle$ ./15p-basic
1
1 2 4 7
5 6 12 3
9 10 0 8
13 14 11 15
URULDRDLDR

Execution time for this set: 0.86884 s
```

```
main@virtualpupo:~/Desktop/15-puzzle$ ./15p-astar
1
1 2 4 7
5 6 12 3
9 10 0 8
13 14 11 15
URULDRDLDR

Execution time for this set: 0.000446 s
```

Já para um resolvível em 15 passos, a diferença era gritante:

```
main@virtualpupo:~/Desktop/15-puzzle$ ./15p-basic
1
1 2 4 7
5 6 12 3
9 10 8 15
0 13 14 11
RRRULURULDRDLDR
Execution time for this case: 937.971 seconds
```

```
main@virtualpupo:~/Desktop/15-puzzle$ ./15p-astar
1
1 2 4 7
5 6 12 3
9 10 8 15
0 13 14 11
RRRULURULDRDLDR

Execution time for this set: 0.000589 s
```

O gráfico a seguir mostra visualmente a crescente disparidade entre a evolução do tempo de execução do algoritmo BFS (em azul) e do A* (em verde), em função do número de passos necessários para resolver um 15-puzzle arbitrário: