

Trabalho de Inteligência Artificial

Alunos:
Daniel Leite Dantas
Filipe Alves Sampaio

09/12/2020

Introdução

O jogo dos 8, também conhecido com 8-Puzzle, é um quebra-cabeça deslizante que consiste em um quadro de peças quadradas numeradas em ordem aleatória com uma peça faltando. O quebra-cabeça também existe em outros tamanhos. Veja na figura 1 um exemplo do jogo.

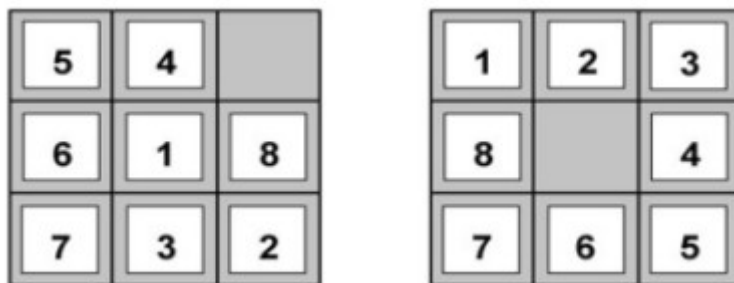


Figura1. Modelo do jogo dos 8.

O quebra-cabeça n é um problema clássico para modelagem de algoritmos envolvendo heurísticas. As heurísticas comumente usadas para este problema incluem contar o número de ladrilhos extraviados e encontrar a soma das distâncias Manhattan entre cada bloco e sua posição na configuração da meta.

O jogo consiste em iniciar de um estado e chegar a um estado definido como solução. Por exemplo, observe na figura 1, a imagem a esquerda representa um possível estado inicial e a imagem a direita representa o estado meta.

Assim, este trabalho tem como objetivo analisar quatro algoritmos de busca, que são: busca em largura, busca em profundidade, busca gulosa e busca A*. Esses algoritmos foram implementados afim de aferir seus desempenhos em relação a sua capacidade de resolver o jogo. As métricas de análise de cada algoritmo usados foram três cálculos de custos diferentes, que são: custo de caminho, que calcula a profundidade ou nível onde foi encontrado o estado meta, custo de espaço, que calcula a quantidade de nós da árvore que ainda não foram expandidos e custo de tempo, que calcula a quantidade de nós expandidos.

Metodologia

Nosso trabalho consistiu em duas etapas. Primeiro implementamos todos os algoritmos de busca e executamos alguns testes utilizando o terminal do SO. Depois começamos a planejar uma interface gráfica para apresentar os resultados calculados, o tabuleiro onde será inserido o estado inicial e por onde será mostrado os passos que cada busca irá realizar.

Assim, implementamos os quatro algoritmos de busca requisitados pelo trabalho, que foram os algoritmos de busca em largura, de profundidade, busca gulosa e busca A*. Em todos os algoritmos foi requisitado que fosse feito escolhas de movimentos aleatoriamente. Só foi aplicado essa escolha nos algoritmos de busca gulosa e A*, devido a problemas com loops que chamamos de “loop global”, onde uma sequência de movimentos era sempre feita. Também implementamos um mecanismo de verificação dos movimentos possíveis a partir do estado selecionado, onde nenhum dos próximos possíveis estados poderia ser igual ao que os originou.

No algoritmo de busca em largura implementamos uma lista onde sempre é inserido um estado novo no fim da fila. Então é selecionado o primeiro nó da fila, é feito o que chamamos de “expansão”, que é a verificação dos movimentos possíveis a partir daquele estado. Então após a

expansão, é inserido todos ao final da fila e depois o estado pai é removido da estrutura. Logo após é selecionado então o próximo estado sendo o primeiro estado da fila. Isso é feito até que o estado meta seja alcançado. Observe na figura 2 o trecho de código que implementa a busca em largura.

```
58 while (aberto.Count > 0 && !achouMeta)
59 {
60     // nó visitado é contabilizado
61     noFinal += 1;
62
63     // adiciona pai atual para a solução
64     solucaoPais.Add(_current);
65
66     // Console.WriteLine("Profundidade atual: " + _current.prof);
67     if (isMeta(_current))
68     {
69         break;
70     }
71
72     if (aberto.Count > 0)
73     {
74         expande(_current);
75         // Console.WriteLine("Possib. de movimento na profundidade atual: " + _current.prof);
76         // Console.WriteLine("-----");
77     }
78
79     aberto.Remove(aberto.First());
80
81     if (aberto.Count > 0)
82         _current = aberto.First();
83 }
```

Figura 2. Trecho onde é implementado a busca em largura.

Salientamos novamente que não utilizamos escolha aleatória nesse algoritmo. Somente implementamos a busca e fizemos com que os nós filhos jamais fossem iguais ao pai.

No algoritmo de busca em profundidade implementamos uma lista (que funcionará com pilha) que guardará todos os nós que serão trabalhados no processo de busca. Primeiro é inserido o estado inicial na pilha e logo depois é feito a expansão dos estados possíveis do primeiro elemento do topo da pilha. Após isso, o estado pai é removido da pilha e cada estado possível é inserido na estrutura, onde o próximo nó selecionado será o último inserido na mesma estrutura. Assim é feito até que seja encontrado o estado meta. Veja na figura 3 o trecho que implementa essa busca.

```

127 while (aberto.Count > 0 && !achouMeta)
128 {
129     // nó visitado é contabilizado (custo de caminho)
130     noFinal += 1;
131
132     // adiciona pai atual para a solução
133     solucaoPais.Add(_current);
134
135     // Console.WriteLine("Profundidade atual: " + _current.prof);
136     //5. Checa se o estado é meta
137     if (isMeta(_current))
138     {
139         break;
140     }
141
142     //6. Se tiverem estados abertos, remove o ultimo
143     if (aberto.Count > 0)
144     {
145         aberto.Remove(aberto.Last());
146     }
147
148     //7. Se a profundidade máxima não foi alcançada: expande o nó
149     if (_current.prof < profMax)
150     {
151         expande(_current);
152         // Console.WriteLine("Possib. de movimento na profundidade atual: " + _current.prof);
153         // Console.WriteLine("-----");
154     }
155
156     //8. Se tiverem estados abertos seta o topo da pilha como estado atual
157     if (aberto.Count > 0)
158     {
159         _current = aberto.Last();
160     }
161 }

```

Figura 3. Trecho de código que implementa a busca em profundidade.

Observe que no nosso algoritmo de busca em profundidade também não implementamos escolhas aleatórias dos estados filhos para a expansão. Ao invés disso decidimos limitar a árvore até a profundidade 30 e fazer com que a busca fosse visitando nó a nó até encontrar a resposta, no limite da profundidade informada. Isso foi feito para tentar evitar estouro de pilha e também para tentar achar a possível melhor resposta em nós de profundidades mais razas. Esse algoritmo também implementa a expansão do nó pai e faz seus filhos evitarem repetir o estado do pai.

Resaltamos que analisamos a possibilidade do algoritmo não encontrar a resposta na profundidade informada. E para resolver isso decidimos adicionar um auto-ajuste da busca, onde caso ele não encontrasse o nó meta, ele chamaria a própria busca de forma recursiva e iteraria mais uma unidade á profundidade. Depois procuraria a resposta na nova profundidade e, caso não achasse, faria o mesmo processo, até achar a resposta na suposta melhor profundidade.

No algoritmo de busca gulosa decidimos utilizar a heurística da distância Manhattan para calcular o “peso” ou “distância” de cada estado selecionado até o estado marcado como meta. Em resumo, o algoritmo implementado funciona semelhante a busca em largura que implementamos, utiliza inclusive a mesma estrutura de fila, mas o ponto crucial da busca gulosa é que ao invés dele sempre selecionar o próximo primeiro elemento da fila, ele escolhe, dentre os estados expandidos, o que tiver menor distância Manhattan. Caso haja filhos com distâncias iguais, ele selecionará aleatoriamente entre eles. Veja abaixo na figura 4 e 5 o trecho de código que a implementa.

Nesse algoritmo implementamos a expansão como nos algoritmos anteriores, onde os novos estados expandidos jamas serão iguais ao seu pai que os geraram.

```

208 while (aberto.Count > 0 && !achouMeta)
209 {
210     // nó visitado é contabilizado
211     noFinal += 1;
212
213     // adiciona pai atual para a solução
214     solucaoPais.Add(_current);
215
216     // Console.WriteLine("Profundidade atual: " + _current.prof);
217     if (isMeta(_current))
218     {
219         break;
220     }
221
222     if (aberto.Count > 0)
223     {
224         // Console.WriteLine("Extendendo jogadas possíveis: \n");
225         expande(_current);
226         // Console.WriteLine("Possib. de movimento na profundidade atual: " + _current.prof);
227         // Console.WriteLine("-----");
228     }
229
230     aberto.Remove(_current);
231
232     abertoTemp.Remove(_current);
233
234     if (aberto.Count > 0)
235     {
236         int menorCusto = ManhattanDistance(aberto.First());
237         int proximoNo = 0;
238

```

Figura 4. Trecho de código que implementa a busca gulosa com heurística Manhattan.

```

239         for (int i = 1; i < aberto.Count; i++)
240         {
241             Random gen = new Random();
242             int prob = gen.Next(2);
243             if (ManhattanDistance(aberto[i]) < menorCusto)
244             {
245                 menorCusto = ManhattanDistance(aberto[i]);
246                 proximoNo = i;
247                 continue;
248             }
249             else if (ManhattanDistance(aberto[i]) == menorCusto && prob != 0)
250             {
251                 menorCusto = ManhattanDistance(aberto[i]);
252                 proximoNo = i;
253                 continue;
254             }
255         }
256
257         _current = aberto[proximoNo];
258         aberto = new List<No> { };
259         aberto.Add(_current);
260         // Console.WriteLine("\nno selecionado: \n");
261         // _current.mostraValores();
262         // Console.WriteLine("====");
263         // System.Threading.Thread.Sleep(1000);
264     }
265 }

```

Figura 5. Trecho de código que implementa a busca gulosa com heurística Manhattan.

E por fim, no algoritmo de busca A* reutilizamos o algoritmo de busca anteriormente citado (busca gulosa) e mudamos a forma de seleção do próximo estado, onde ao invés dele selecionar somente os estados filhos atuais, ele irá considerar todos os estados guardados na fila, e selecionará o de menor peso. Também implementa o caso de haver estados com mesmos pesos, selecionando aleatoriamente entre eles. Veja na figura 6 o trecho de código.

```
303 while (aberto.Count > 0 && !achouMeta)
304 {
305     // adiciona pai atual para a solução
306     solucaoPais.Add(_current);
307
308     // Console.WriteLine("Profundidade atual: " + _current.prof);
309     if (isMeta(_current))
310     {
311         break;
312     }
313
314     if (aberto.Count > 0)
315     {
316         // Console.WriteLine("Expandindo jogadas possiveis: \n");
317         expande(_current);
318         // Console.WriteLine("Possib. de movimento na profundidade atual: " + _current.prof);
319         // Console.WriteLine("-----");
320     }
321
322     // remove o nó atual para não dar loop
323     aberto.Remove(_current);
324     // adiciona nó atual na lista fechada para evitar repetir estados
325     fechado.Add(_current);
326
327     if (aberto.Count > 0)
328     {
329         int menorCusto = aberto.First().f; // f(n)
330         int proximoNo = 0;
```

Figura 6a. Trecho de código que implementa a busca A*.

```
332 for (int i = 1; i < aberto.Count; i++)
333 {
334     Random gen = new Random();
335     int prob = gen.Next(2);
336
337     // if (fechado.Find(x => x.Equals(aberto[i])) != null)
338     // {
339     //     Console.WriteLine("\n** ACHOOOOOOU **\n");
340     //     continue;
341     // }
342
343     if (aberto[i].f < menorCusto)
344     {
345         menorCusto = aberto[i].f;
346         proximoNo = i;
347         continue;
348     }
349     else if (aberto[i].f == menorCusto && prob != 0)
350     {
351         menorCusto = aberto[i].f;
352         proximoNo = i;
353         continue;
354     }
355 }
356
357 _current = aberto[proximoNo];
```

Figura 6b. Trecho de código que implementa a busca A*.

Levantamos uma pesquisa sobre a solubilidade de cada estado inicial dado. Dois matemáticos, Johnson & Story (1879) usaram um argumento de paridade para mostrar que metade das posições iniciais para o quebra-cabeça n são impossíveis de resolver, não importa quantos movimentos sejam feitos. Isso é feito considerando uma função da configuração do bloco que é invariável em qualquer movimento válido e, em seguida, usando isso para particionar o espaço de todos os estados rotulados possíveis em duas classes de equivalência de estados alcançáveis e inacessíveis. Com isso, adicionamos a cada uma das buscas, um algoritmo que verificasse o estado inicial informado para solução. Caso o estado informado der paridade par, seria possível obter uma resposta, caso contrário, seria informado uma mensagem pelo terminal informando a impossibilidade de resolver o jogo a partir daquele estado.

Resultados

Nos nossos testes executamos alguns estados diferentes, mas neste documento iremos mostrar detalhadamente apenas um, para simplificar o documento e facilitar a leitura.

Na figura 7 é possível observar os exemplos informados para execução dos testes. Dentre eles, o mais a esquerda, batizado como “7.1” será detalhado nesse relatório. Os demais só serão informados seus desempenhos em uma tabela.

4	1	2
	5	3
6	7	8

	7	2
1	4	3
6	8	5

1	2	3
4	7	5
6	8	

Figura 7. Exemplos de execuções dos algoritmos de busca (exemplo 7.1 o mais a esquerda, 7.2 o do meio e 7.3 o mais a direita).

Como informado na introdução deste documento, dividimos o trabalho em duas etapas. Primeiro construímos os algoritmos em um projeto a parte e os testamos usando o terminal do SO (*sistema operacional*). Depois implementamos uma UI (*user interface*) para apresentação gráfica dos resultados, como requisitado pelo trabalho.

Na execução de teste usando o terminal, obtivemos os seguintes resultados para cada um dos algoritmos:

	Tempo de execução	Custo de caminho	Custo de espaço	Custo de tempo
Largura	00:00:00.03	6	40	46
Profundidade	00:00:00.09	30	24	11952
Gulosa	00:00:00.004	6	5	6
A*	00:00:00.003	5	5	5

Tabela 1. Resultados dos custos obtidos para a execução do exemplo 7.1.

	Tempo de execução	Custo de caminho	Custo de espaço	Custo de tempo
Largura	00:00:00.01	9	168	259
Profundidade	00:00:00.30	29	20	45493
Gulosa	00:00:00.004	9	8	9
A*	00:00:00.004	8	8	8

Tabela 2. Resultados dos custos obtidos para a execução do exemplo 7.2.

	Tempo de execução	Custo de caminho	Custo de espaço	Custo de tempo
Largura	00:00:00.007	3	4	5
Profundidade	00:00:00.008	29	24	29
Gulosa	00:00:00.004	3	3	3
A*	00:00:00.004	2	3	2

Tabela 3. Resultados dos custos obtidos para a execução do exemplo 7.3.

Para a impressão dos passos até a resolução de cada algoritmo, decidimos apresentar no terminal até a profundidade 6, pois para chegar até a solução, todos os algoritmos acabam visitando muitos nós e ficaria difícil de entender. Assim, observe na figura 8 um trecho do que é printado na tela do terminal, a sequência até a profundidade 3 do algoritmo de busca em largura.

Profundidade: 1 412 053 678	Profundidade: 2 412 503 678	Profundidade: 2 012 453 678
Profundidade: 2 412 503 678	Profundidade: 3 412 530 678	Profundidade: 3 102 453 678
Profundidade: 2 012 453 678	Profundidade: 3 402 513 678	Profundidade: 2 412 653 078
Profundidade: 2 412 653 078	Profundidade: 3 412 573 608	Profundidade: 3 412 653 708

Figura 8. Sequências de escolhas que a busca em largura executa.

Na figura 9 observamos a impressão até a profundidade 3 do algoritmo em profundidade. Observe também na figura 10 que mostramos a parte onde a busca chega ao final da profundidade máxima e começa a procurar nos outros galhos da árvore.

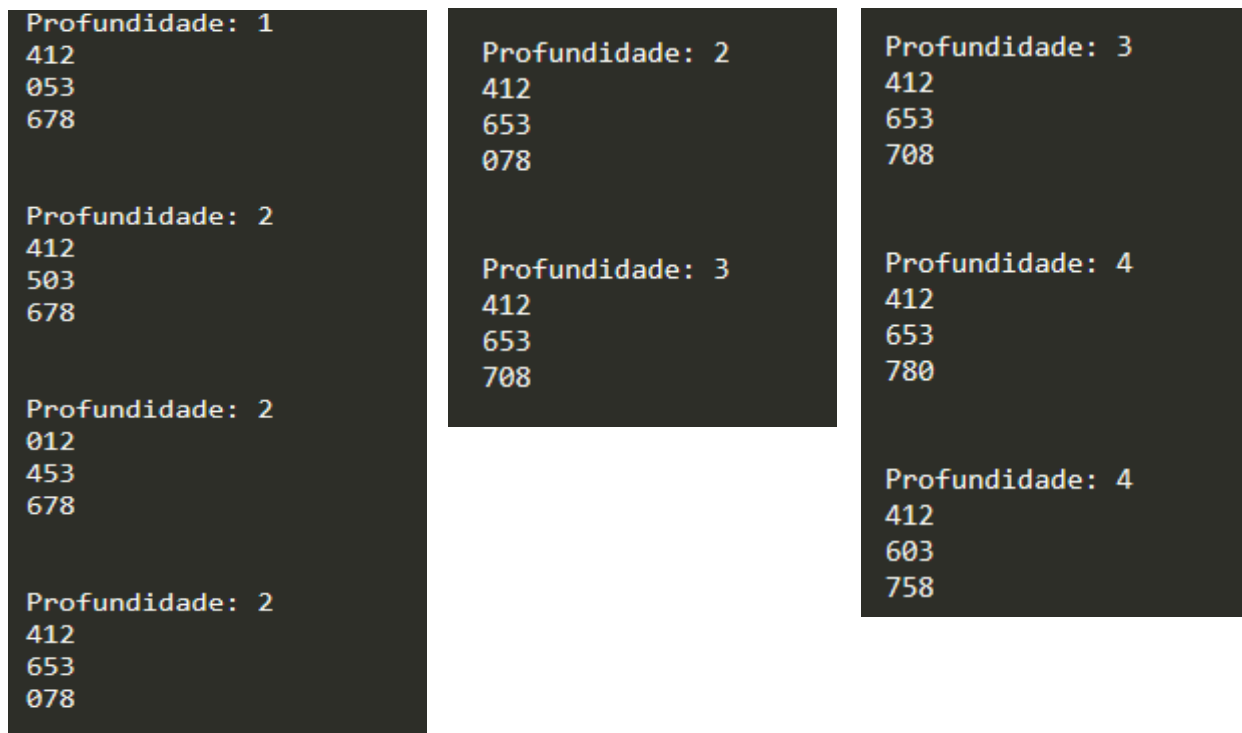


Figura 9. Sequências de escolhas que a busca em profundidade executa.

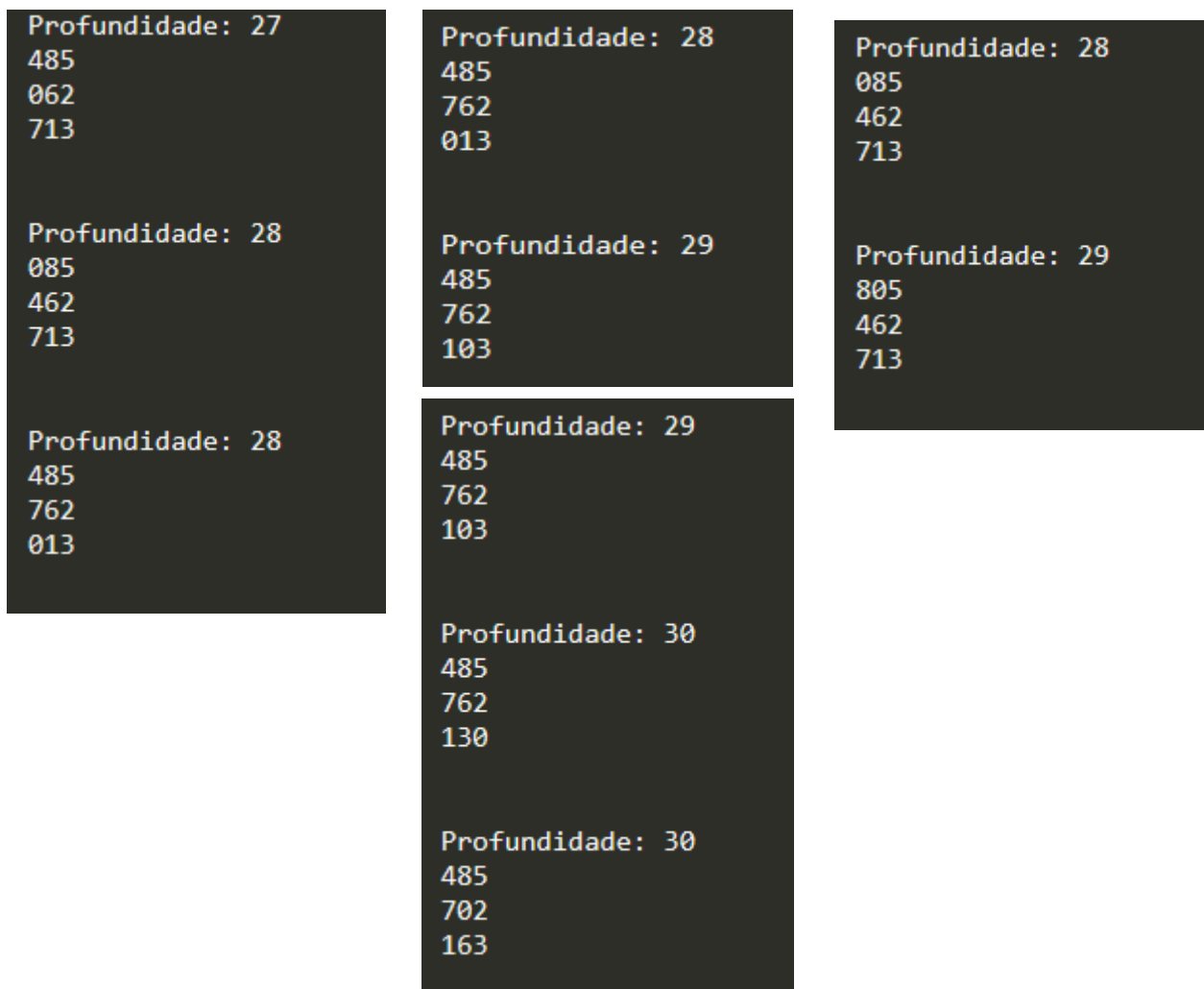
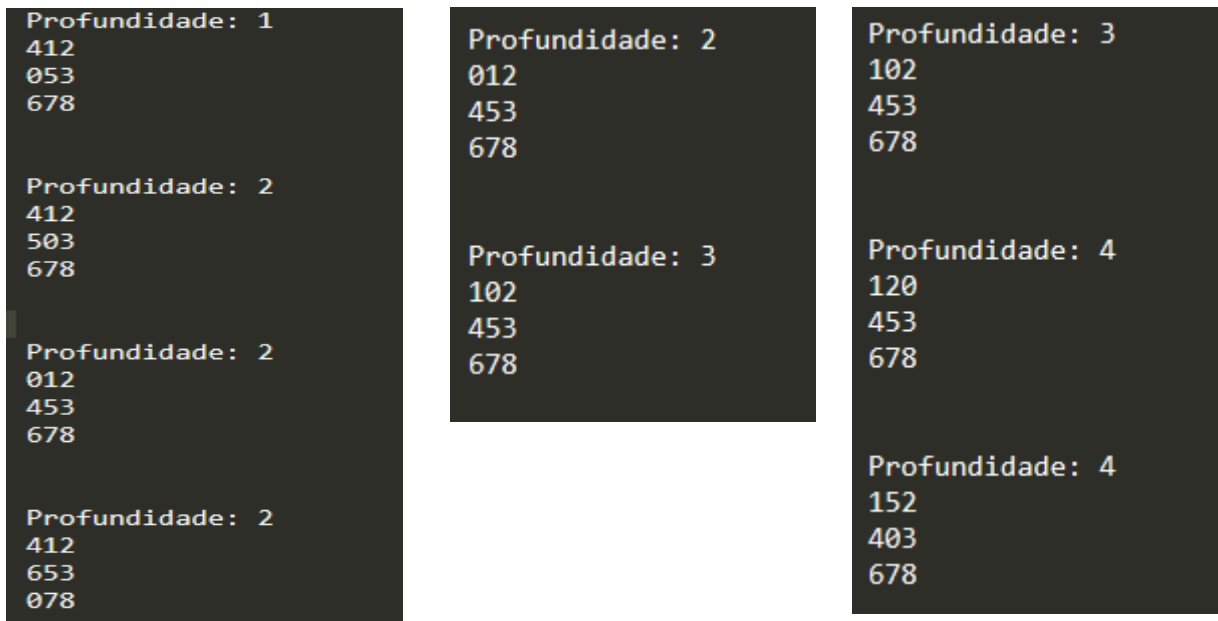


Figura 10. Sequências de escolhas que a busca em profundidade executa, voltando na árvore e procurando a resposta em outros galhos da árvore.

Fazer com que a busca voltasse nos galhos fez com que a busca desse fim a um loop de movimentos, e isso foi o suficiente para não modificarmos mais o código da busca em profundidade.

Observe na figura 11 os resultados da impressão no terminal dos passos até a profundidade 3 no algoritmo de busca gulosa. Ele executa passos semelhantes ao de profundidade, mas não realiza *backtracking* (ação que a busca faz ao voltar nos galhos da árvore).



```
Profundidade: 1
412
053
678

Profundidade: 2
412
503
678

Profundidade: 2
012
453
678

Profundidade: 2
412
653
078

Profundidade: 2
012
453
678

Profundidade: 3
102
453
678

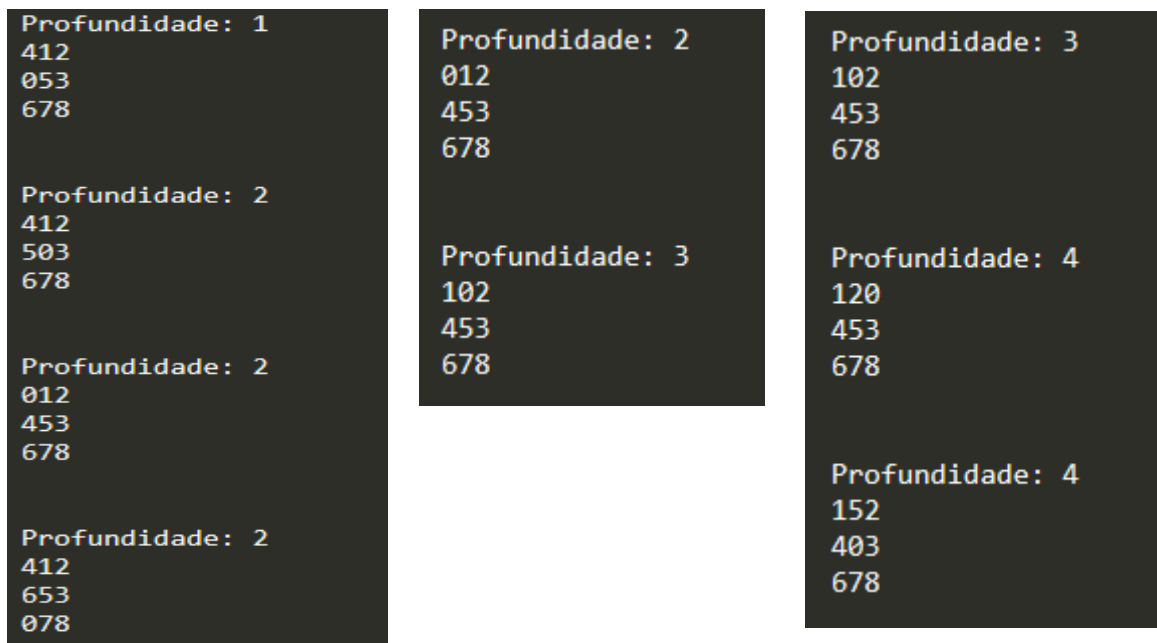
Profundidade: 3
102
453
678

Profundidade: 4
120
453
678

Profundidade: 4
152
403
678
```

Figura 11. Sequências de escolhas que a busca gulosa executa.

E por fim, na figura 12 a sequência de escolhas que o algoritmo A* realiza.



```
Profundidade: 1
412
053
678

Profundidade: 2
412
503
678

Profundidade: 2
012
453
678

Profundidade: 2
412
653
078

Profundidade: 2
012
453
678

Profundidade: 3
102
453
678

Profundidade: 3
102
453
678

Profundidade: 4
120
453
678

Profundidade: 4
152
403
678
```

Figura 12. Sequências de escolhas que a busca A* executa.

Observe nas figuras abaixo o mesmo exemplo 7.1 sendo executado na UI desenvolvida.

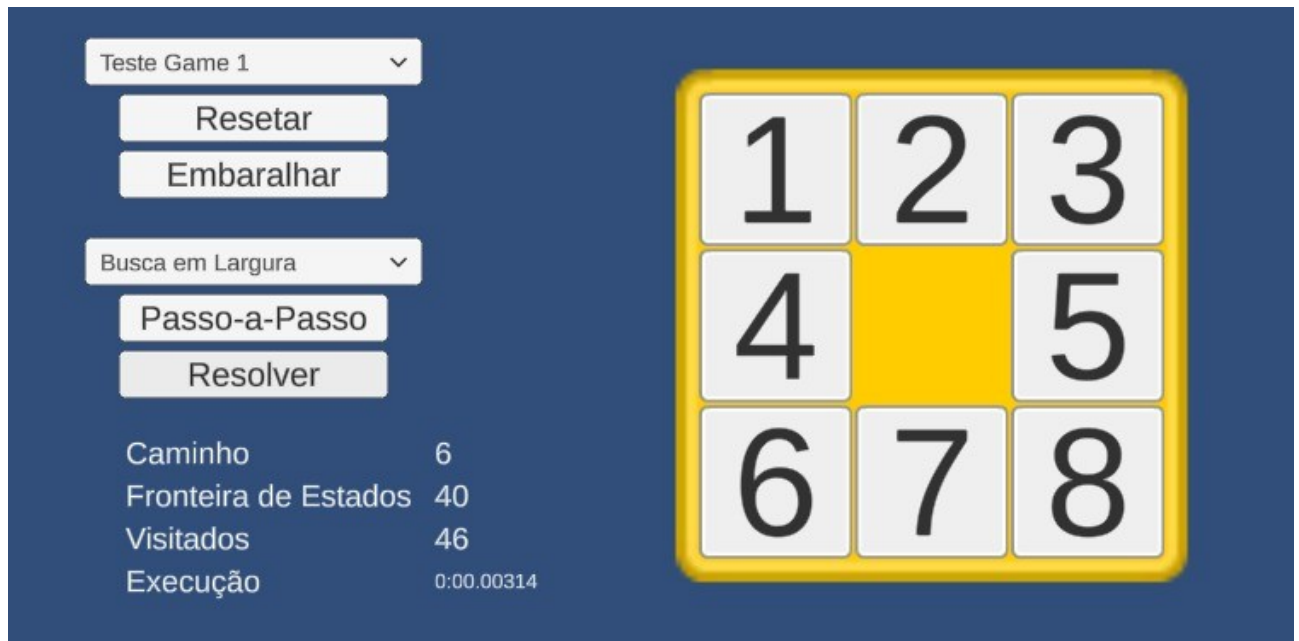


Figura 12. Tela da UI mostrando a execução da busca em largura do exemplo da figura 7 (7.1).

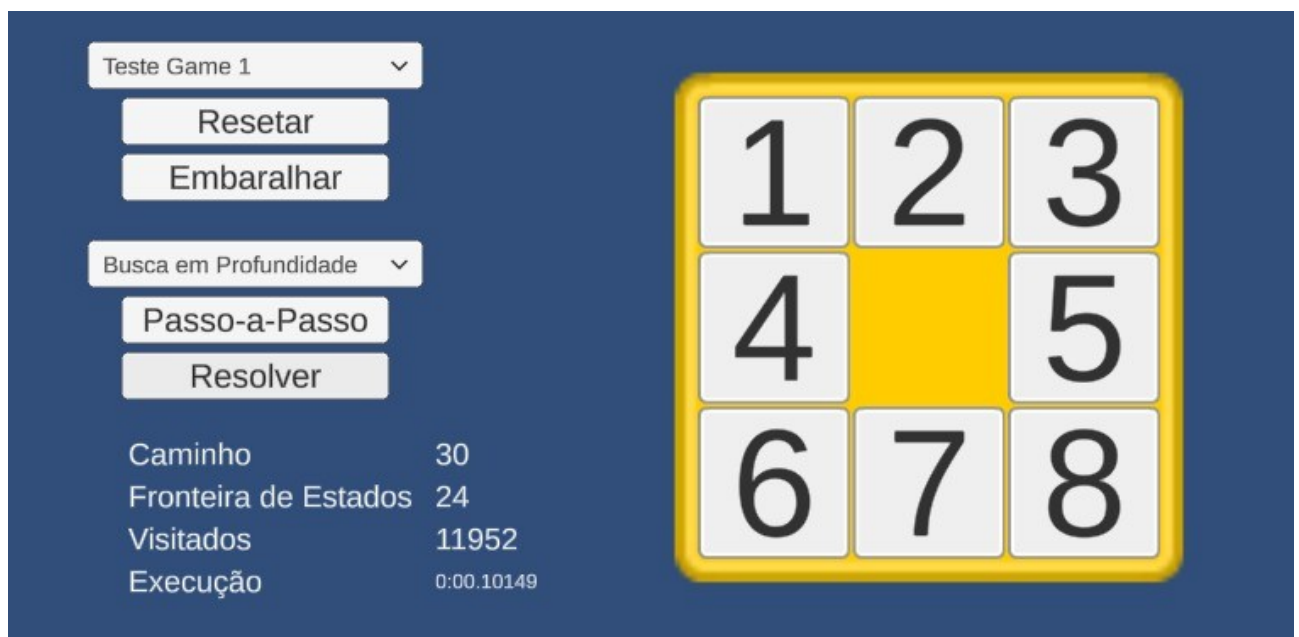


Figura 13. Tela da UI mostrando a execução da busca em profundidade do exemplo da figura 7 (7.1).

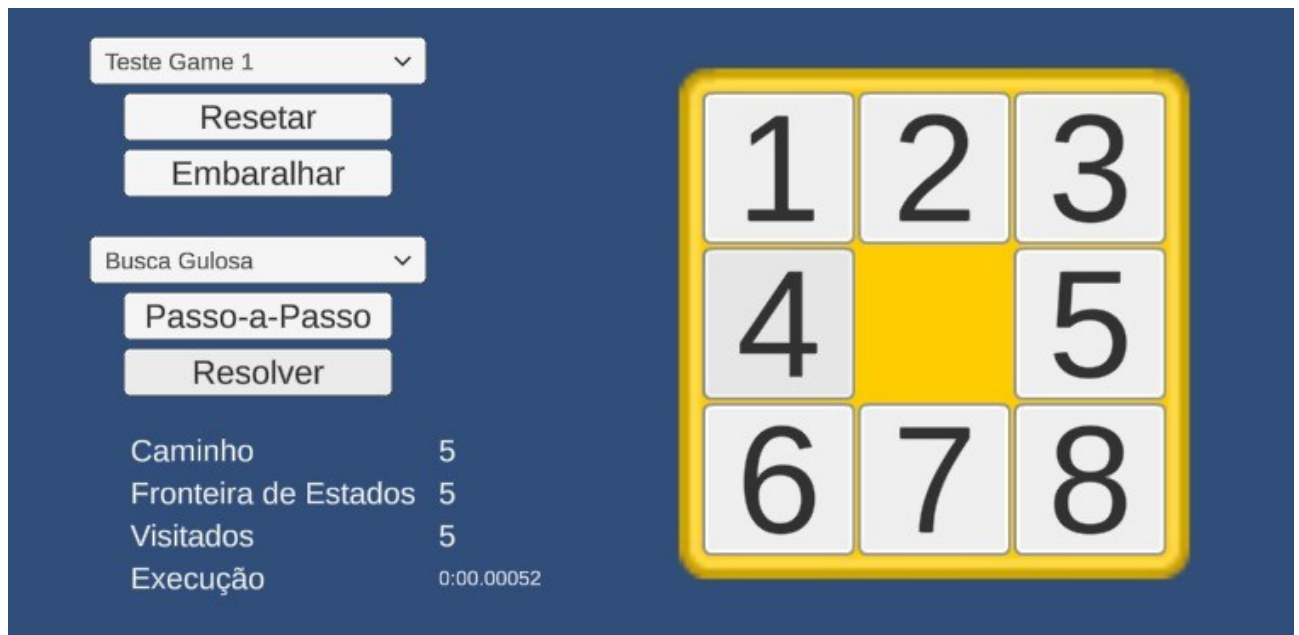


Figura 12. Tela da UI mostrando a execução da busca gulosa do exemplo da figura 7 (7.1).

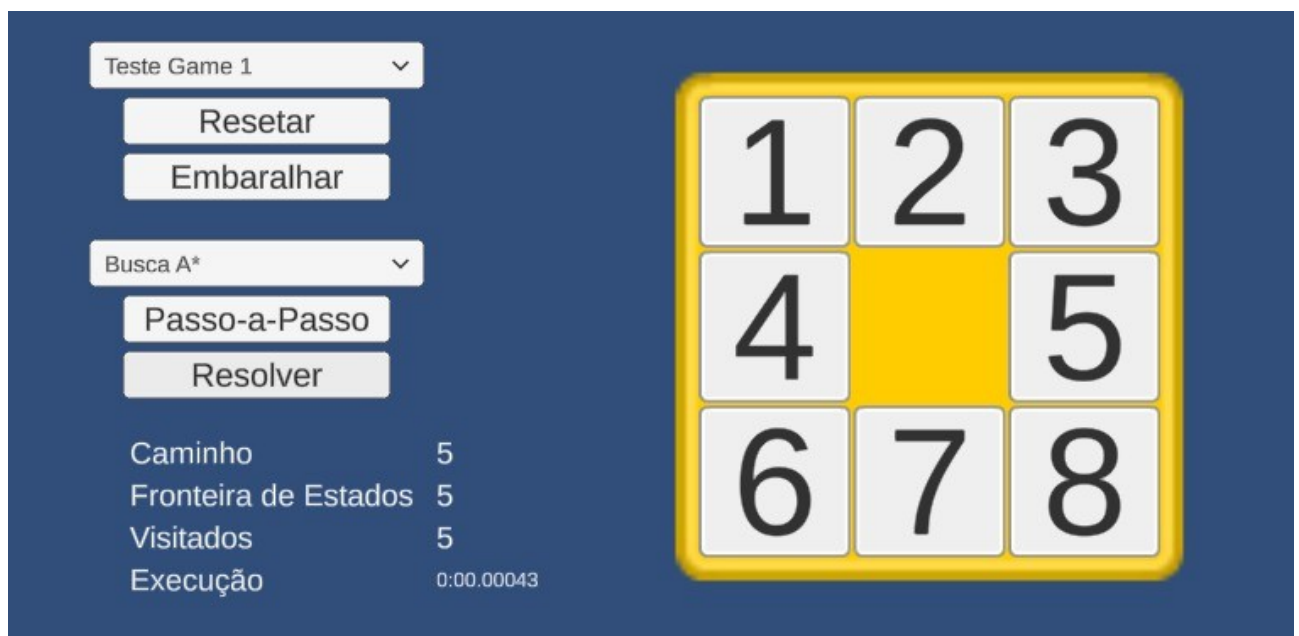


Figura 12. Tela da UI mostrando a execução da busca A* do exemplo da figura 7 (7.1).

Fazendo uma análise de desempenho, temos a seguinte tabela comparativa, levando em consideração sua admissibilidade.

	Completeza	Otimização
Largura	x	
Profundidade	x	
Gulosa	x	
A*	x	x

Tabela 4. Comparativo de completeza e otimização dos algoritmos implementados.