UFMG

Universidade Federal de Minas Gerais

Inteligência Artificial
Mestrado em Ciência da Computação

8 PUZZLE - RELATÓRIO

Autor:
Daniel Martins Reis

Matrícula:
2019697267

1 8 Puzzle

O presente trabalho consiste em implementar e comparar os diferentes métodos de busca apresentados durante o curso da disciplina de Inteligência Artificial ministrado pelo professor Luiz Chaimowicz, aplicando-os a um toy problem.

O quebra-cabeça das oito peças (8-Puzzle) é composto por uma moldura 3x3 contendo um conjunto de peças numeradas de 1 a 8 e um espaço vazio. O propósito do jogo resume-se a posicionar as peças em uma determinada ordem (Figura 1) apenas deslizando-as pela moldura. Sabe-se que o problema de encontrar uma solução com o menor número de passos no caso geral, denominado N-Puzzle, é NP-Difícil [1].

2 Implementação

Foram implementados os seguintes algoritmos para o N Puzzle:

- Busca sem informação:

  Breadth-first search;

  Iterative deepening search;

  Uniform-cost search.

- Busca com informação:

  A* search;

  Greedy best-first search.

- Busca local:

  Hill Climbing, permitindo movimentos laterais.

Foram utilizadas duas heurísticas distintas, sendo elas:

- Número de quadrados em uma posição errada: Percorre cada linha e coluna da matriz e contabiliza cada vez que o valor na mesma posição na matriz objetivo e na matriz atual são diferentes nas duas matrizes.
- Distância de Manhattan: Percorre cada linha e coluna da matriz e contabiliza a soma das distâncias que separam os quadrados das posições finais. Ou seja, calcula a distância em x e em y de um elemento numa posição da matriz atual para a posição que ele deveria ficar, ou seja, como está na matriz objetivo.

A geração dos filhos ocorre da seguinte maneira:

1. Procura-se o elemento '0' na matriz, ou seja, a posição do espaço em branco;
2. A partir dessa posição, verifica-se as possibilidades na seguinte ordem:

   Mover para esquerda => retroceder uma posição na coluna (x - -);

   Mover para direita => retroceder uma posição na coluna (x++);

   Mover para cima => retroceder uma posição na coluna (y - -);

   Mover para baixo => retroceder uma posição na coluna (y++);
3. Em cada uma deles, é clonado matriz pai e realizado o movimento;
4. Elimina-se então a configuração gerada que for igual ao avó, senão pode gerar loop no algoritmo, pois voltaria a uma configuração anterior.

Foi utilizada a estrutura de dados "Node"para todos algoritmos. Dentro de um nó, armazenamos as seguintes informações:

- root => índice do pai no vetor de nós;
- matrix => configuração, ou seja, matriz que representa o estado;
- cost => custo real dessa configuração;
- costH => custo heurístico dessa configuração.

Em todos algoritmos, os nós são armazenados dentro de uma lista de nós, que funciona como uma lista dinâmica. Dessa maneira, a lista denominada "fronteira"e "explorados"apenas armazenam índices desses nós, ou seja, qual a posição do nó na lista de nós.

A lista de explorados, funciona como uma HASH, em que, ao inserir uma configuração (matriz), verificamos:

1. Posição do elemento "0"na matriz;
2. Posição do elemento "2"na matriz;
3. Posição do elemento "4"na matriz;
4. Posição do elemento "6"na matriz;
5. Posição do elemento "8"na matriz.

Nesse caso, 0 representa o espaço vazio. Essa posição está num intervalo de 0 a 8, contabilizado da seguinte maneira:

- Posição (0,0) => vale 0;
- Posição (0,1) => vale 1;
- Posição (0,2) => vale 2;
- Posição (1,0) => vale 3;
- Posição (1,1) => vale 4;
- Posição (1,2) => vale 5;
- Posição (2,0) => vale 6;
- Posição (2,1) => vale 7;
- Posição (2,2) => vale 8;

A HASH então pode ser descrita da seguinte forma:

1. Lista: Posição do elemento "0"na matriz:
   [[pos=0] [pos=1] [pos=2] [pos=3] [pos=4] [pos=5] [pos=6] [pos=7] [pos=8]]
2. Cada sublista: Posição do elemento "2"na matriz:
   [[pos=0] [pos=1] [pos=2] [pos=3] [pos=4] [pos=5] [pos=6] [pos=7] [pos=8]]
3. Cada subsublista: Posição do elemento "4"na matriz:
   [[pos=0] [pos=1] [pos=2] [pos=3] [pos=4] [pos=5] [pos=6] [pos=7] [pos=8]]
4. Cada subsubsublista: Posição do elemento "6"na matriz:
   [[pos=0] [pos=1] [pos=2] [pos=3] [pos=4] [pos=5] [pos=6] [pos=7] [pos=8]]
5. Cada subsubsubsublista: Posição do elemento "8"na matriz:
   [[pos=0] [pos=1] [pos=2] [pos=3] [pos=4] [pos=5] [pos=6] [pos=7] [pos=8]]

Dessa maneira, é uma HASH de tamanho 9 x 9 x 9 x 9 x 9, pois cada posição está definida entre 0 e 8. Assim, para setar um nó explorado ou testar se um nó já foi explorado, basta acessar a lista de explorados explored[A][B][C][D][E] em que:

- A => Posição do elemento "0"na matriz;
- B => Posição do elemento "2"na matriz;
- C => Posição do elemento "4"na matriz;
- D => Posição do elemento "6"na matriz;
- E => Posição do elemento "8"na matriz.

Seu uso, torna a busca bem mais rápida, visto que não é mais preciso percorrer os n nós da lista, mas somente apenas a lista de nós que estão nesse índice.

Na busca de custo uniforme, no qual precisamos sempre escolher o nó de menor custo, o que se faz é inserir ordenado e retirar sempre o primeiro. Ressalta-se que para melhoria de desempenho, a função que identifica em qual posição devemos inserir o nó, é basicamente uma busca binária no vetor, na qual analisamos:

- Se o elemento tem custo total menor que o do primeiro => Insere no início;
- Se o elemento tem custo total maior que o do último => Insere no fim;
- Senão, calcula o meio (m) e:

  Verifica se o elemento está no intervalo entre listElement[m] e listElement[m+1] => Insere na posição m+1;

  Verifica se o elemento está no intervalo entre listElement[m-1] e listElement[m] => Insere na posição m;

  Verifica se o elemento é maior listElement[m] => l=m, ou seja, reduz a esquerda pra posição de meio atual e refaz a busca na nova metade;

  Verifica se o elemento é menor listElement[m] => r=m, ou seja, reduz a direita pra posição de meio atual e refaz a busca na nova metade.

No Hill Climbing Search e no Greedy Best First Search a inserção ocorre sempre no final, e a busca pelo menor é basicamente uma busca sequencial na lista de nós de forma a encontrar o nó de menor custo.

No Depth Search foi otimizado o espaço utilizado, de maneira que dado que você encontrou uma configuração que já foi explorada, mas agora tem um custo menor, o que se faz é apenas atualizar no vetor de nós esse nó, ou seja, setar pra ele o novo custo e o novo pai.

Posteriormente é descrito de forma geral um pseudocódigo da implementação de cada algoritmo realizada.

2.1 Breadth First Search

Características:

- Expande o nó mais raso ainda não expandido;
- Expande a raiz, depois os sucessores da raiz, depois os sucessores dos sucessores...
- FIFO

Pseudocódigo implementado:

```
    Verifica se o no atual e igual ao objetivo
        Retorna o resultado
    Inicializa a lista de nos como vazia
    Inicializa a fronteira
    Insere na fronteira o primeiro no (No inicial)
    Inicializa a lista de nos explorados como vazia
    Loop => Enquanto a fronteira nao estiver vazia
        Atualiza no => Seleciona o primeiro elemento da fronteira
        Marca o no como explorado
        Explora cada filho gerado
            Verifica se o no nao esta entre os explorados e nao esta
            na fronteira:
                Verifica se o no e igual a solucao objetivo
                    Insere o resultado na lista de nos
                    Insere o resultado na fronteira
                    Atualiza o indice do ultimo no
                    Retorna o resultado
                Adiciona no na fronteira
```

Análise:

- Completo: sim;
- Ótimo: sim, pois nesse caso o custo é uma função não decrescente da profundidade do nodo, ou seja, aumenta em 1 para cada passo;
- Complexidade de espaço: Total de nós gerados, pois todos nós são mantidos na memória;

- Complexidade de tempo: Considerando branch factor b, solução no nível d. $O(b^d)$.

2.2 Uniform Cost Search

Características:

- Semelhante ao BFS, mas expande o nodo d menor custo até o momento;
- Uso de uma fila de prioridades (Heap)

Pseudocódigo implementado:

```
    Define o custo real do no inicial como zero
    Inicializa a lista de nos como vazia
    Inicializa a fronteira
    Insere na fronteira o primeiro no (No inicial)
    Inicializa a lista de nos explorados como vazia
    Loop => Enquanto a fronteira nao estiver vazia
        Atualiza no => Seleciona o no de menor custo (Apenas real)
        da fronteira
        Verifica se o no atual e igual ao objetivo
            Retorna resultado
        Marca o no como explorado
        Explora cada filho gerado
            Verifica se o no esta entre os explorados
            Verifica se o no esta na fronteira
            Posicao em que o no deve ser inserido para inserir
            ordenado
            Se o no nao foi explorado e nem esta na fronteira
                Adiciona no na lista de nos
                Adiciona o indice do no na fronteira
            Se o no nao foi explorado mas ja esta na fronteira
                Atualiza aquele no utilizando o novo pai e o novo
                custo => Isso porque foi identificado um novo
                caminho ate ele com menor custo
                Atualiza o custo do no
                Atualiza o pai do no
                Remove esse no da fronteira e insere novamente
                com o custo atualizado
            Incrementa o custo do caminho
```

Análise:

- Completo: sim;
- Ótimo: sim, pois segue o menor custo através da fila de prioridade;
- Complexidade de espaço: Total de nós gerados, pois todos nós são mantidos na memória;

- Complexidade de tempo: Considerando C* como o custo da solução ótima e que cada passo tem um custo de pelo menos $\epsilon, nopiorcaso : O(b^{1+C*/\epsilon})$.

2.3 Depth Search

Características:

- Expande o nó mais profundo;
- LIFO

Pseudocódigo implementado:

```
Verifica se o no atual e igual ao objetivo
    Mostra resultado e retorna verdadeiro
Inicializa a lista de nos como vazia
Inicializa a fronteira
Adiciona no na lista de nos com profundidade zero
Insere na fronteira o primeiro no (No inicial)
Inicializa a lista de nos explorados como vazia
Loop => Enquanto a fronteira nao estiver vazia
    Atualiza no => Seleciona o ultimo elemento da fronteira
    Marca o no como explorado
    Verifica se o no e igual a solucao objetivo
        Mostra resultado e retorna verdadeiro
    Se a profundidade dos filhos e menor que o limite, podemos
    gerar mais filhos
    Explora cada filho gerado
        Verifica se o no esta entre os explorados
        Se ja cheguei nesse no antes e ja explorei
            Verifico se ja cheguei nesse no com um custo maior.
            Se sim, insiro ele na fronteira
                Atualizo o custo do no
                Atualiza o pai do no
                Adiciona o indice do no na fronteira
        Se nao
            Verifica se o no esta na fronteira
            Se ja cheguei nesse no antes mas ele ainda nao
            foi explorado, ou seja, ele esta na fronteira
                Verifico se ja cheguei nesse no com um custo
                maior. Se sim, insiro ele na fronteira
                    Atualizo o custo do no
                    Atualiza o pai do no
                    Adiciona o indice do no na fronteira
        Se esse no nao foi nem explorado e nem esta na fronteira
```

```
                Adiciona no na lista de nos com profundidade=depth+1
                Adiciona o indice do no na fronteira
    Retorna falso pois nao encontrou o no objetivo
```

Análise:

- Completo: não - Falha com caminhos infinitos ou loops;
- Ótimo: não;
- Complexidade de espaço: Considerando branch factor b e m como a profundidade máxima é O(bm): linear!

- Complexidade de tempo: O(b^m).

2.4 Iterative Deepening Search

Características:

- Faz repetidas buscas em profundidade, aumentando o limite a cada iteração;
- Combina os benefícios da BFS e DFS;

Pseudocódigo implementado:

```
    Comeca a contar o tempo total da busca em profundidade
    Inicializa a profundidade como 1
    Enquanto a profundidade e menor ou igual a profundidade maxima
        Comeca a contar o tempo da busca na Tenta encontrar o no
        objetivo numa profundidade maxima depth
        Termina de contar o tempo da busca na profundidade depth
        Se o resultado e verdadeiro, finaliza
        Avanca a profundidade
    Termina de contar e mostra o tempo total da busca
```

Análise:

- Completo: sim;
- Ótimo: sim quando chegar ao depth;

- Complexidade de tempo: (d)b+(d-1)b^2 + $(d-2)b^3$ + ... + $(1)b^d$.

2.5 A Star Search

Características:

- A escolha do nodo a ser investigado é feita considerando-se: custo real de chegar ao nó atual + estimativa do custo do nó atual para o nó objetivo.

Pseudocódigo implementado:

```
Seta o custo real como zero
Define o custo heuristico com base na heuristica selecionada
=> Heuristica 1 ou 2
    Inicializa a lista de nos como vazia
    Inicializa a fronteira
    Insere na fronteira o primeiro no (No inicial)
    Inicializa a lista de nos explorados como vazia
    Loop => Enquanto a fronteira nao estiver vazia
        Atualiza no => Seleciona o no de menor custo (real +
        heuristico) da fronteira
        Verifica se o no e igual a solucao objetivo
            Retorna o resultado
        Marca o no como explorado (fechados)
        Explora cada filho gerado
            Se o no nao esta entre os fechados
                Verifica se o no ja foi aberto (fronteira)
                Verifica se o no esta na fronteira
                Calcula o custo heurisitico
                Posicaoo em que o no deve ser inserido para
                inserir ordenado
                Se o no ainda nao foi aberto
                    Adiciona no na lista de nos
                    Adiciona o indice do no na fronteira
                    (abertos)
                Se o no ja foi aberto
                    Se o custo real agora e menor que o custo
                    real anterior, encontrei um caminho melhor
                        Atualiza o custo para se chegar nesse
                        no atraves do novo pai
                        Atualiza o pai do no
                        Remove esse no da fronteira e insere
                        novamente com o custo atualizado
        Toda vez que descer um ramo na arvore, avanca o custo
```

```
real
```

Análise:

- Completo: sim;
- Ótimo: sim (se a heurística é admissível);
- Complexidade de espaço: Todos os nós são mantidos na memória;
- Complexidade de tempo: Apesar disso tudo, dependendo da heurística, o número de nodos expandidos ainda pode ser uma função exponencial da solução.

Nota: ambas as heurísticas utilizadas são admissíveis, pois nelas o custo indicado pela heurística é menor ou igual ao custo real para o objetivo, como provado no exemplo abaixo:

Estado atual:

$$\begin{bmatrix} 7 & 2 & 4 \\ 5 & 0 & 6 \\ 8 & 3 & 1 \end{bmatrix}$$

Estado objetivo:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Heurística 1: número de peças na posição errada. No exemplo acima, h1=8. Portanto, é admissível porque cada peça deverá ser movimentada pelo menos uma vez.

Heurística 2: soma das distâncias de cada peça às suas posição correta. Não considera diagonais, pois considera a Distância de Manhattan. No exemplo acima, h2= 3+1+2+2+2+3+3+2 = 18. Portantot, é admissível porque cada ação movimenta cada peça apenas um passo mais próximo do objetivo.

2.6 Greedy Best First Search

Características:

- Preocupa apenas com a função heurística.

Pseudocódigo implementado:

```
    Calcula o custo heurisitico
    Inicializa a lista de nos como vazia
    Inicializa a fronteira
    Insere na fronteira o primeiro no (No inicial)
    Inicializa a lista de nos explorados como vazia
    Loop => Enquanto a fronteira nao estiver vazia
        Atualiza no => Seleciona o no de menor custo (Apenas
        heuristico) da fronteira
        Verifica se o no e igual a solucao objetivo
            Retorna o resultado
        Marca o no como explorado
        Esvazia a fronteira, pois so deve ter em cada iteracao
        os filhos gerados
        Explora cada filho gerado
            Calcula o custo heurisitico
            Se o no nao foi explorado
                Adiciona no na fronteira
    Retorna o resultado
```

Análise:

- Completo: não, pois pode entrar em loop;
- Ótimo: não;

- Complexidade de tempo e espaço: b é o fator de expansão e m a profundidade máxima da árvore de busca => $O(b^m)$.

2.7 Hill Climbing Search

Características:

- Move na direção do incremento da função, terminando quando acha um pico;
- Nessa implementação, quando encontra um pico, ele realiza um movimento lateral.

Pseudocódigo implementado:

```
    Calcula o custo heurisitico
    Inicializa a lista de nos como vazia
    Inicializa a fronteira
```

```
    Insere na fronteira o primeiro no (No inicial)
    Inicializa a lista de nos explorados como vazia
    Loop => Enquanto a fronteira nao estiver vazia
        Inicializa todas variaveis
        Movimento lateral
            Atualiza no => Seleciona o no de menor custo (Apenas
            heuristico) da fronteira
            Se o filho e pior que o pai, a busca para, ou seja,
            realiza um movimento lateral e continua
        Verifica se o no e igual a solucao objetivo
            Retorna o resultado
        Marca o no como explorado
        Explora cada filho gerado
            Calcula o custo heurisitico
            Se o no nao foi explorado
                Adiciona no na fronteira
    Retorna o resultado
```

Análise:

- Completo: não, pois pode entrar em loop;
- Ótimo: não.

3 Testes de execução

Os resultados descritos posteriormente foram obtidos executando com as seguintes configurações:

Estado inicial:

$$\begin{bmatrix} 8 & 6 & 7 \\ 2 & 5 & 4 \\ 3 & 0 & 1 \end{bmatrix}$$

Estado objetivo:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Observe abaixo como executar o código:

```
python tp_ia_daniel_reis.py 3 3 [[8,6,7],[2,5,4],[3,0,1]]
                            [[1,2,3],[4,5,6],[7,8,0]] 0 31
```

Note que os 2 primeiros parâmetros se remetem ao tamanho da matriz. Logo, os próximos 2 parâmetros se remetem aos valores da matriz inicial e da matriz objetivo respectivamente. O próximo parâmetro se refere ao algoritmo escolhido para executar, sendo:

- 0 - Executar todos os algoritmos;
- 1 - Executar apenas o Breadth First Search;
- 2 - Executar apenas o Uniform Cost Search;
- 3 - Executar apenas o Iterative Deepening Search;
- 4 - Executar apenas o A Star Search Heuristic 1;
- 5 - Executar apenas o A Star Search Heuristic 2;
- 6 - Executar apenas o Greedy Best First Search Heuristic 1;
- 7 - Executar apenas o Greedy Best First Search Heuristic 2;
- 8 - Executar apenas o Hill Climbing Search Heuristic 1;
- 9 - Executar apenas o Hill Climbing Search Heuristic 2.

O último parâmetro se refere ao nome do arquivo a ser salvo com a solução do problema obtida pelo(s) algoritmo(s) executado(s). Para mais detalhes dos resultados da execução, acesse o arquivo README.

É importante ressaltar que o código apresentado nesse trabalho foi desenvolvido para suportar um N puzzle. Uma execução com um tamanho 4x4 fica:

```
python tp_ia_daniel_reis.py 4 4
    [[1,2,3,4],[5,6,7,8],[9,10,11,12],[0,13,14,15]]
    [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,0]] 0 3
```

4 Resultados

A tabela 1 sumariza os resultados da execução com a entrada do 8 puzzle:

Tabela 1: Dados da execução de todos algoritmos - 8 puzzle

Algoritmo	Tempo execução	Custo	Nós	Nós explorados
breadthFirstSearch	1334.41 s	31	181440	181410
uniformCostSearch	2559.74 s	31	181440	181438
iterativeDeepeningSearch	371.85 s	31	175260	175248
AStarSearchH1	1278.30 s	31	181440	181439
AStarSearchH2	1288.91 s	31	181440	181439
greedyBestFirstSearchH1	0.0461 s	319	575	319
greedyBestFirstSearchH2	0.0279 s	71	119	72
hillclimbingSearchH1	73.353 s	39	42392	27196
hillclimbingSearchH2	0.0546 s	43	385	246

Tabela 2: % de nós explorados x Nós explorados/segundo

Algoritmo	% Explorado	Nós explorados/segundo
breadthFirstSearch	99,98 %	135,95
uniformCostSearch	100,00 %	70,88
iterativeDeepeningSearch	99,99 %	471,29
AStarSearchH1	100,00 %	141,94
AStarSearchH2	100,00 %	140,77
greedyBestFirstSearchH1	55,48 %	6922,62
greedyBestFirstSearchH2	60,50 %	2580,96
hillclimbingSearchH1	64,15 %	370,76
hillclimbingSearchH2	63,90 %	4503,98

Durante o desenvolvimento da busca em profundidade iterativa, notou-se que a mesma é muito demorada perante as demais, visto que acaba ocupando muita memória explorando os nós, o que faz com que a uma profundidade aproximadamente maior do que 27, o algoritmo fique muito lento. Na sua implementação então, como mencionado anteriormente, o espaço foi otimizado da seguinte forma: Caso seja explorado algum nó que já está na lista de nós explorados, apenas substitue-se os valores de pai e custo do nó, ao invés de criar mais um espaço de memória pra ele.

Dessa forma, o algoritmo acabou se tornando mais rápido que os demais, pois a busca em profundidade em si não verifica se o nó já foi explorado ou está na fronteira.

Outro ponto que influenciou bastante no tempo de execução dos algoritmos foi o fato de ter criado uma HASH para a lista de explorados. Com isso, o tempo de busca reduziu drasticamente, tornando a execução de todos algoritmos que a utilizam muito mais rápida.

Nota-se que os algoritmos que utilizam apenas custo real, acabam demorando mais para encontrar a solução do problema. De fato, o custo heurístico acaba direcionando-o no percurso. Com relação a heurística, a de número 2 acabou resolvendo o problema em menor tempo.

Percebe-se também que os algoritmos baseados somente em heurística podem retornar uma resposta ruim mas muito rápido, como é o caso do Greedy Best First Search, ou também pode demorar um pouco e mesmo assim retornar uma resposta ruim, como o HillClimbing.

Executando o N puzzle com tamanho 4x4, já dá pra perceber que o nível de dificuldade para resolução do problema é maior. O 5x5, piora mais ainda. Dessa forma, quanto maior o tamanho acaba se tornando muito mais lento, pois como mencionado anteriormente, o N puzzle é um problema não determinístico em tempo polinominal. Também, é de suma importância mencionar que existem algumas entradas aleatórias que não tem solução. Para algumas entradas que necessitam de poucos movimentos, como na mostrado abaixo, o N puzzle é resolvido de forma rápida para uma entrada que tem uma solução simples e com poucos passos.

Entrada:

```
python tp_ia_daniel_reis.py 4 4
[[1,2,3,4],[5,6,7,8],[9,10,11,12],[0,13,14,15]]
[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,0]] 0 38
```

Entrada:

```
python tp_ia_daniel_reis.py 5 5
[[1,2,3,4,5],[6,7,0,8,10],[12,13,14,9,15],
[11,16,18,19,20],[21,17,22,23,24]]
[[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],
[16,17,18,19,20],[21,22,23,24,0]]
0 11
```

Tabela 3: Dados da execução de todos algoritmos - 15 puzzle

Algoritmo	Tempo execução	Custo	Nós	Nós explorados
breadthFirstSearch	0.46 s	2	6	3
uniformCostSearch	0.49 s	2	18	7
iterativeDeepeningSearch	1.01 s	2	9	8
AStarSearchH1	0.45 s	2	18	7
AStarSearchH2	0.49 s	2	18	7
greedyBestFirstSearchH1	0.48 s	2	6	2
greedyBestFirstSearchH2	0.46 s	2	6	2
hillclimbingSearchH1	0.51 s	52	397	187
hillclimbingSearchH2	0.51 s	176	381	186

Tabela 4: Dados da execução de todos algoritmos - 24 puzzle

Algoritmo	Tempo execução	Custo	Nós	Nós explorados
breadthFirstSearch	81.28 s	11	24637	10745
uniformCostSearch	594.68 s	11	66819	29503
iterativeDeepeningSearch	57.14 s	11	24594	24587
AStarSearchH1	575.49 s	11	67804	29998
AStarSearchH2	585.01 s	11	67804	29998
greedyBestFirstSearchH1	4.38 s	11	30	11
greedyBestFirstSearchH2	4.19 s	11	30	11
hillclimbingSearchH1	4.23 s	11	148	65
hillclimbingSearchH2	4.57 s	351	795	379

Com o N puzzle, percebe-se que até os algortimos que levam em consideração apenas o custo heurístico levam um tempo maior para resolver o problema. De fato, não são apresentados aqui dados do N puzzle com instâncias maiores, simplesmente devido ao tempo de execução ser muito extenso.