Introdução a Inteligência Artificial DCC024
Aluno: Paulo Henrique Moreira Melo

Para implementação dos algoritmos, foram criadas estruturas comuns a todos. Os nós dos algoritmos são objetos da classe **State,** cujos atributos armazenam dados importantes para a execução dos algoritmos.

State.state : int list. Estado do nó.

State.parent : State. Nó que foi expandido para gerar o nó.

State.move: int. Número referente a direção do movimento que gerou o estado do nó. 1 para cima, 2 para baixo, 3 para esquerda e 4 para direita.

State.depth:int. Profundidade do nó

State.cost:int. Custo do nó.

State.key:int. Soma da heurística do estado do nó com o custo do nó. Usado apenas nos algoritmos que usam função heurística.

State.map:string. Concatenação dos elementos do estado. Facilita comparação entre estados.

Foram criadas funções que são utilizadas por diferentes algoritmos

expand(X:node)->List of States Toma um objeto da classe State como argumento, e instancia até 4 novos nós (referentes às ações de deslizar as teclas do jogo para cima, para baixo, para esquerda e para direita). Os novos nós são instanciados com seu State.state alterado, com seu State.parent = node (o nó recebido como parâmetro) e State.depth acrescido de 1. O tamanho da lista retornada pode variar entre 2 e 4 elementos.

move(state:list of int, position:int)->list of int|None Usando a codificação já mencionada (1 para cima, 2 para baixo…) para **position,** troca a posição do elemento 0 com outro elemento, simulando o deslizamento de peças no jogo 8-puzzle. Pode retornar ao novo estado, ou, em caso do movimento ser ilegal, retornar nulo.

Estrutura de dados muito usadas

Queue

No Python, objeto da classe deque. Permite que, ao usarmos, **deque.popleft(),** possamos pegar o primeiro elemento e implementar o método first-in, first-out

Stack

No python, usamos list. Ao usarmos, **list.pop(),** implementa-se last-in,first-out.

Heap

Árvore binária cujos filhos sempre são maiores ou iguais aos pais. A utilidade dela é que com custo computacional reduzido conseguimos garantir que o primeiro elemento da heap seja sempre o menor. Se fossemos usar lista e darmos **sort()** a cada nova inserção, o custo computacional seria imenso e demoraria muito tempo.

Algoritmos

bfs(Breadth First Search)

O algoritmo mantém uma fila de nós ainda não expandidos (**queue**). Ele expande o primeiro nó de queue (o primeiro que entrou na lista). Assim, ele expandirá os nós mais superficiais antes de se aprofundar em nós mais fundos. O algoritmo checa primeiro se o nó expandido está em **explored** (um Set de strings). Se não tiver, ele adiciona o node.map ao Set e adiciona o nó ao queue. O algoritmo chega ao fim quando a fila fica vazia ou quando se encontra o nó com o estado procurado.

Iterative deepening search

Introdução a Inteligência Artificial DCC024
Aluno: Paulo Henrique Moreira Melo

Usei duas funções para este algoritmo. **Ida(start_state),** que recebe como argumento a sequência de inteiros referentes ao Estado do nó inicial. A função começa um while-loop e dentro deste loop chama **dls_mod(start_state, threshold)**, passando como parâmetro, o estado inicial e o **threshold.** O threshold é a profundidade máxima que a busca em profundidade irá realizar. Inicialmente o threshold será 1 e a medida que loop repetirá, será o threshold será acrescido de 1. Se o **dls_mod** encontrar o **goal_state** dentre os nós expandidos, ele retornará a lista com a fronteira, caso contrário retornará None. Portanto, a condição de parada do while-loop é checar se o tipo de retorno de dls_mod é uma lista (no python uma funcao pode retornar tipos diferentes). Se for lista, o while-loop pára, significando que o goal_node foi encontrado.

Uniform Cost Search

Usa-se o estrutura de dados heap e cada nó da heap é uma tupla (state.cost, State), assim ao dar **heappop(),** sempre pegamos o elemento com menor custo.

Ele expande o nó. O algoritmo checa se os nós filhos estão em **explored** (um Set de strings). Se não tiver, ele adiciona o node.map ao Set e dá **heappush(heap,entry)** na entry, que é uma tupla(state.cost, State). O algoritmo chega ao fim quando a fila fica vazia ou quando se encontra o nó com o estado procurado.

Algoritmos de busca com informação

Os algoritmos de buscas com informação fizeram uso de **duas heurísticas diferentes**

H1(state)->int Esta função pega um dado estado e calcula quantos elementos (inteiros de 1 a 8) estão com index diferente do index destes mesmos elementos na lista de inteiro do estado objetivo

H2(state)->int Esta função pega um dado estado e calcula o somatório da distância manhattan que cada elemento está da posição do objetivo. Para isto, considera-se que o ambiente é **um grid 3x3.**

A*

O algoritmo também mantém um set de elementos explorados (explored).

Ele começa por calcular a key do nó inicial. Como o nó inicial possui custo zero, sua key é o resultado da função heurística. Para este algoritmo, usamos a estrutura de dados **Heap.** A **Heap** consegue manter o nó com a **menor** key (custo+heurística) na frente e o faz com custo computacional reduzido, pois ele não precisa reorganizar todos os novos elementos a cada nova inserção. Cada node da nossa heap é uma tupla de 3 elementos **(!!!)**,sendo o primeiro elemento State.key, o segundo elemento State.move e terceiro elemento o objeto da classe State. O algoritmo funciona dentro while-loop que se encerra quando a heap acaba.

O algoritmo começa pegando o primeiro elemento (o com menor key) da heap com **heappop(heap)**, daí ele compara se este elemento possui o estado objetivo, caso contrário ele expande este elemento com a função expand. Ela calcula o valor da heurística dos novos nós e soma ao State.cost, guardando esta soma em State.key. Daí, usando **heappush(heap, (key,move,nó)),** ele acrescenta a heap.

Uma preocupação importante para o algoritmo A* é de quando achamos um estado já encontrado, entretanto desta vez o encontramos com por um caminho com **State.cost menor**, Assim, atualizamos na heap e colocamos o node com state.cost menor.

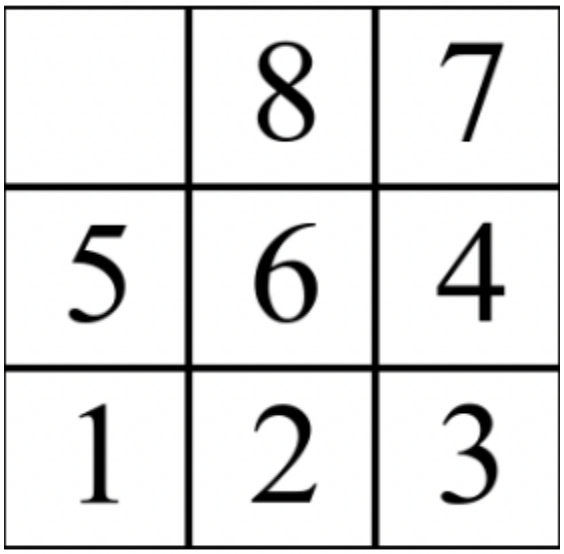
Greedy Search

Introdução a Inteligência Artificial DCC024
Aluno: Paulo Henrique Moreira Melo

O greedy search funciona de forma similar ao A*, mas a diferença é que não se usa o custo de caminho. Assim, ele sempre escolhe o nó na fronteira com menor heurística.

Análises

Exemplo: Passos mínimos para solução: 30



Solução: 30

Análise das Heurística:
h1([0,8,7,5,6,4,1,2,3])->8
h2([0,8,7,5,6,4,1,2,3])->16
(já foi explicado acima o funcionamento das duas heurísticas)
h2 retorna um valor maior que h1 em todos os casos (pois, no mínimo, cada peça fora do lugar precisa deslizar 1 espaço para chegar ao index objetivo), portanto h2 é uma heurística mais eficiente que h1. Ambas são admissíveis, pois retornam um valor menor que o custo real (30)

Análise dos algoritmos

30	BFS	DFS	IDS	UCS	A*h1	A*h2	GS h1	GS h2
Goal depth	30	41110	32	30	30	30	76	84
Expaded nodes	181.264	145.480	200743	181264	109.145	10.360	279	408
Maior fronteira	24.048	42.826	28	24114	24732	5139	181	276
RAM usage	157253.6	177307.6	18366.4	163971.0	212303.8	212434.9	9437.1	10354.6
time	8.9	1.6	2.6	3.28	10.6	1.0	0.003	0.009
Max depth	31	65982	32	31	30	30	76	115

Introdução a Inteligência Artificial DCC024
Aluno: Paulo Henrique Moreira Melo

Considerando que toda ação possui custo 1, é esperado que o **BFS** e o **UCS** se comportem de maneira similar, o que foi exatamente o que aconteceu. Seria esperado que o UCS fosse um pouco menos eficiente, pois ele teria que fazer a comparação dos valores dos custos e dar **sort()** na lista. Na primeira versão do algoritmo que fiz, acontecia exatamente isso (UCS levava 500 segundos para o exemplo de solução em 30 passos!). Mas na nova versão, optei por usar a estrutura de dados **heap** e o tempo de execução foi bem menor. Eu implementei o algoritmo **DFS** a título de curiosidade, vê-se sua tendência a encontrar soluções extremamente subótimas. **IDS**, como previsto, chega a solução ótima expandindo um número de nós maior que o **BFS.** E como vantagem nota-se que o tamanho máximo da fronteira do **IDS** fica extremamente reduzida. Fica claro que dos algoritmos sem informação IDS possui a melhor performance.

Nas buscas com informação, viu-se que o A* com heurística ruim foi, de todos os algoritmos, aquele com execução mais demorada. Muito possivelmente, pois estamos chamando uma função heurística a cada novo nó e como esta heurística não é tão boa, o número de nós expandidos não reduz o bastante. Com heurística boa, o resultado foi mais positivo. O Greedysearch tendeu a expandir menos nós que A*, o que faz sentido pois ele escolhe sempre opção com menor heurística.

Nós expandidos em função do número mínimo de passos para objetivo