Relatório do TP1 de Introdução à Inteligência Artificial

Nome: Lucas Xavier Veneroso
Matrícula: 2016065138

1 - O esqueleto por trás

Primeiramente temos que falar sobre o esqueleto por trás dos algoritmos: as funções compartilhadas e similaridades nas implementações. Para começar podemos perceber que criamos uma classe conhecida como Node. Essa classe serve para nos dar uma estrutura que contém em si valores úteis relacionados a qualquer nó da árvore:

vector - Este atributo é o principal do nó. Nele está armazenado o estado do vetor que estamos ordenando.
cost - Aqui temos o custo para se chegar do nó anterior a este. Sua utilidade é óbvia.
heuristic - Aqui é armazenado o custo da heurística associado ao estado em que o vetor está. Falaremos mais sobre a heurística numa seção posterior
nav_cost - O custo total de se navegar até aquele nó. Extremamente importante para o UCS e A* e para retornar o valor do custo total da solução.
father - Este definitivamente requer uma explicação menos óbvia. father existe para indicar quem é o nó progenitor do nó atual. Usamos esse atributo principalmente para indicar o caminho pelo qual chegamos na solução: começando pela solução e subindo pelos progenitores até chegar na raiz, nós temos todos os estados intermediários.

Agora que falamos sobre a classe dos nós, podemos discutir a principal função utilizada integralmente por todos os algoritmos: a função que expande um nó: expandNode. Para isso nós começamos com h (heurística) = 0 e um vetor s (sons/filhos) vazio. Agora nós varremos o vetor que foi dado. Começamos com o primeiro elemento ([0]) e comparamos esse com todos à direita. Se há um elemento menor que este, uma troca pode ser feita e, após a heurística ser calculada, o novo estado é adicionado na lista dos filhos com o custo definido se a troca foi de vizinhos ou não. Note no código que o vetor auxiliar newNode é uma cópia do vetor original dado. Um cuidado para que não alteremos o vetor dado diretamente por meio da criação de seus filhos. Após repetir esse processo com os itens do vetor (comparando com todos aqueles à sua direita) nós teremos todos os filhos desse nó e o nó é considerado expandido.

Outra função usada por todos algoritmos é a printIntermidiate que simplesmente, se PRINT foi uma opção colocada na entrada, imprime os nós intermediários até a solução. Grande parte do funcionamento deste código é explicado na discussão do atributo father da classe Node. Como começamos da solução e subimos pelos seus pais nós temos que imprimir começando do final, por isso o uso do pop().

2 - Como fazer a busca?

Nesta seção iremos falar sobre os diferentes algoritmos para navegar a árvore de escolha. Primeiramente vamos começar falando sobre a grande similaridade entre eles. Todos eles usam uma estrutura parecida para expandir seus nós, expandindo a fronteira enquanto não se acha uma solução, e todos ignoram os nós repetidos que já foram explorados. Agora vamos abordar um a um:

BFS - BFS ou Breadth-First Search é o algoritmo que busca na árvore por nível. Para a implementação deste no algoritmo, nós podemos simplesmente continuar chamando a função para se expandir um nó em elementos da fronteira até achar uma solução. Isso pode ser feito porque a expansão adiciona todos os filhos como próximos nós, ou seja, temos que explorar todos os filhos em um nível antes de passar para o próximo.

IDS - O IDS ou Iterative Deepening Search foi o algoritmo mais complexo de se implementar, por ser efetivamente um híbrido entre o BFS e o DFS. O aspecto de se construir uma mesma seção de árvore de novo e de novo apenas aumentando a profundidade de cada tentativa me levou à ideia de se implementar por meio de recursão. Falaremos mais sobre a função recursiva pois essa é a parte principal deste algoritmo. A função recursiva pega várias variáveis de entrada: f (frontier/fronteira) , e (lista dos estados expandidos, resetada a cada passo do IDS, pois recomeçamos a árvore do 0), n_exp (por causa de problemas com manipulação de variáveis globais, um vetor é passado e, como estamos em Python, a função consegue alterar este vetor diretamente), i (uma variável versátil que indica em que nível da árvore estamos e possibilita recursão, sendo uma das condições de parada da recursão e um identificador se estamos na raiz para dar o return final), d (depth/profundidade da árvore que é incrementada a cada passo do IDS) e r (sendo a variável encarregada de finalizar o processo como um todo ao achar a solução). Uma coisa a se destacar nessa função recursiva é o loop envolvendo n_expansion_level, um loop que executa uma recursão para cada nó em um nível. É claro que ao começar com um dos filhos este chama a função recursiva, o que faz com que sempre começamos os returns pelo filho mais fundo.

UCS - UCS também conhecido como Uniform Cost Search é um dos algoritmos com implementação mais simples. Tudo que precisamos fazer é, após toda expansão, ordenar os nós da fronteira com base no custo de se chegar até aquele determinado

nó (foi utilizada uma função lambda passada como chave para um sort), e depois basta sempre expandir o primeiro membro da fronteira.

A* - Assim como o UCS o A* é de implementação bem simples e sucinta. Tudo o que foi feito a partir do UCS foi adicionar o valor da heurística à função lambda.

Greedy - A busca gulosa, Greedy Best-first Search é a busca mais rápida e eficiente, se preocupando apenas em expandir nós filhos do estado com menor heurística. Isso reduz drasticamente a quantidade de nós explorados e na implementação, ao expandir um nó, ao invés de adicionarmos os filhos na fronteira, esses filhos se tornam a nova fronteira.

3 - A heurística

Agora vamos discutir a heurística, muitas vezes mencionada até aqui. A heurística usada nesse TP é dada pela quantidade de elementos fora do lugar. Podemos determinar isso comparando o elemento com sua posição (1 deveria estar na 1° posição, 2 na 2ª, etc). Isso nos oferece um chão para o custo até a solução. Isso se deve a alguns fatores. O menor número de elementos fora do lugar é 2 já que é impossível se ter apenas um número fora do lugar em um vetor desses já que este elemento deslocado estará ocupando a posição “correta”de outro. Com isso nós temos uma heurística = 2 e um custo mínimo = 2 (caso os elementos a serem trocados sejam vizinhos). Para qualquer número de elementos deslocados maior que dois o número de trocas será pelo menos n/2 cada uma com custo mínimo de 2 dando um custo total mínimo de n onde n é o número de elementos em posições erradas e, assim, sendo também o valor de nossa heurística. Confirmando que nossa heurística é sempre menor ou igual ao custo real, sendo assim admissível.

4 - Alguns exemplos

Vamos dar um exemplo para cada algoritmo, utilizando as entradas de tamanho 8.

BFS - entrada: B 8 4 1 2 3 8 7 5 6 PRINT
saida: 16 141
[4, 1, 2, 3, 8, 7, 5, 6]
[1, 4, 2, 3, 8, 7, 5, 6]
[1, 2, 4, 3, 8, 7, 5, 6]
[1, 2, 3, 4, 8, 7, 5, 6]
[1, 2, 3, 4, 7, 8, 5, 6]
[1, 2, 3, 4, 5, 8, 7, 6]
[1, 2, 3, 4, 5, 6, 7, 8]

IDS - entrada: I 8 2 7 4 1 8 3 6 5 PRINT

saida: 22 1715

[2, 7, 4, 1, 8, 3, 6, 5]
[1, 7, 4, 2, 8, 3, 6, 5]
[1, 4, 7, 2, 8, 3, 6, 5]
[1, 2, 7, 4, 8, 3, 6, 5]
[1, 2, 6, 4, 8, 3, 7, 5]
[1, 2, 3, 4, 8, 6, 7, 5]
[1, 2, 3, 4, 5, 6, 7, 8]

UCS - entrada: U 8 3 5 4 7 2 1 8 6 PRINT

saida: 18 840

[3, 5, 4, 7, 2, 1, 8, 6]
[3, 5, 4, 7, 2, 1, 6, 8]
[3, 5, 4, 1, 2, 7, 6, 8]
[3, 5, 1, 4, 2, 7, 6, 8]
[3, 5, 1, 4, 2, 6, 7, 8]
[1, 5, 3, 4, 2, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]

A* - entrada: A 8 3 8 4 2 7 5 6 1 PRINT

saida: 14 163

[3, 8, 4, 2, 7, 5, 6, 1]
[3, 8, 4, 2, 5, 7, 6, 1]
[3, 8, 4, 2, 5, 6, 7, 1]
[3, 8, 2, 4, 5, 6, 7, 1]
[3, 1, 2, 4, 5, 6, 7, 8]
[1, 3, 2, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]

Greed - entrada: G 8 6 8 3 5 1 7 4 2 PRINT

saida: 18 6

[6, 8, 3, 5, 1, 7, 4, 2]
[6, 2, 3, 5, 1, 7, 4, 8]
[1, 2, 3, 5, 6, 7, 4, 8]
[1, 2, 3, 4, 6, 7, 5, 8]
[1, 2, 3, 4, 5, 7, 6, 8]
[1, 2, 3, 4, 5, 6, 7, 8]

5 - Comparação em Tabela

	tam = 5	tam = 10	tam = 15	tam = 20
BFS	exp = 29	exp = 388	**X**	**X**
IDS	exp = 41	exp = 618	**X**	**X**
UCS	exp = 42	exp = 438	**X**	**X**
A*	exp = 14	exp = 82	**X**	**X**
Greed	exp = 4	exp = 6	exp = 14	exp = 18

vetor tam 5 = 3 5 2 4 1
vetor tam 10 = 1 2 4 5 9 3 10 8 6 7
vetor tam 15 = 7 10 4 15 3 11 12 9 13 5 8 2 14 6 1
vetor tam 20 = 2 20 9 6 8 18 15 17 5 14 1 7 4 16 13 10 3 19 11 12

Se o tempo de execução for maior que 10s: **X**