UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Cecília Regina Oliveira de Assis

Resolução de Problemas via Busca
RELATÓRIO

Minas Gerais, Brasil

2017

Sumário

1 Introdução

1.1 O Problema

O problema abordado se trata do jogo "Quebra-cabeças deslizante", contendo 8 peças. Neste, é necessário que o jogar faça movimentos até que um estado meta seja atingido. No objetivo atual, o estado desejado requer que as peças se encontrem em ordem crescente: `0, 1, 2, 3, 4, 5, 6, 7, 8`; onde o valor `0` representa o elemento vazio e é o único que pode trocar de lugar com os demais elementos.

Logo, todos os movimentos ocorrem de forma adjacente ao `0`, podendo ser para: cima, baixo, esquerda ou direita.

1.2 O Algoritmo

O algoritmo propõe a resolução do problema através do mecanismo de busca `A star`, também conhecido como `A estrela` ou `A*`. O mesmo apresenta a expansão de estados a partir do menor valor retornado por uma função: `f(n)`, que avalia a custo real do caminho (`g(n)`), juntamente com uma heurística (`h(n)`). Tal heurística é responsável por representar, da forma mais abrange possível, os estados permitidos pelo problema e o quão próximo estes estão da solução.

2 Problema

Conforme já levantado, o problema apresenta um estado inicial, com os elementos em qualquer posição, e procura por uma solução onde as peças se encontrem em ordem crescente, a partir do zero.

2.1 Formalismo de Espaço de Estados

2.1.1 Estado Inicial

Uma dada configuração do quebra-cabeça deslizante, onde as peças se encontram de forma desordenada.

2.1.2 Ações

As possíveis ações são:

- para cima;
- para baixo;
- esquerda;
- direita,

onde cada ação representa um movimento de deslize de certa peça em direção ao espaço vazio.

2.1.3 Modelo de Transição

A resposta a cada uma das ações pode ser:

- (Cima(x, 0) = Inverte(x,0));
- (Baixo(x, 0) = Inverte(x,0));
- (Esquerda(x, 0) = Inverte(x,0));
- (Direita(x, 0) = Inverte(x,0)),

onde `x` denomina a peça que está sendo/será movimentada, `0` o espaço vazio e `Inverte` a inversão de posições entre `x` e `0`.

2.1.4 Teste de Meta

Nesta etapa, o teste/algoritmo averígua se a configuração atual do tabuleiro corresponde a configuração esperada, aquela onde as peças se encontram organizadas do menor (0) para o maior valor (8).

2.1.5 Função Custo

Como a busca `A*` utiliza heurísticas para auxilar no cálculo da função custo, a mesma é denotada por `f(n) = g(n) + h(n)`. Aqui, `g(n)` apresenta o custo do caminho, que é igual a quantidade de movimentos realizados até a expansão daquele estado, onde cada movimento tem custo 1 (um) e `h(n)` retrata a heurística, o quão próximo da solução o estado atual está.

Para este problema, a heurística que melhor representa os estados é soma das distâncias entre a posição atual de uma peça e o local onde ela deveria estar. O cálculo é feito através da `Distância de Manhattan`.

3 Algoritmo

3.1 Sobre

O algoritmo desenvolvido foi divido em sub-módulos que tratam de forma individual cada etapa da busca:

1. `arquivo: initial_state.rb`: Inicialmente, um vetor com elementos de zero até a quantidade de peças desejadas para o tabuleiro é criado, de modo que os valores se encontrem em posições aleatórias;
2. `arquivo: possible_solution.rb`: Então cálculos são realizados a fim de determinar a possível solução ou não frente a ordem das peças;
3. `arquivo: build_matrix.rb`: Caso possível, uma matriz é criada, contendo o valor de cada componente e a posição que ele ocupa dentro da mesma. A posição do (0) também é computada;
4. `arquivo: main.rb`: Com isso estado e `f(estado)` são alocados em uma fila de prioridade, que ordenada segundo o menor valor de `f(n)`.
5. `arquivo: run.rb`: Após a conclusão das inicializações, o algoritmo `A*` propriamente dito se inicia, onde, consecutivamente:
   5.1. Uma verificação valida se o estado meta foi alcançado ou não:
      i. Caso positivo, o algoritmo finaliza e apresenta os movimentos realizados e a quantidade deles;
      ii. Caso negativo, o algoritmo verifica se tal estado ainda não foi visitado ou se já foi e seu valor de `f(n)` é menor que o apresentado pelo topo da fila:
         A. `arquivo: expand.rb`: Se sim, o presente estado é expandido, segundo os movimentos possíveis;
         B. Se não, isso significa que uma melhor opção já foi escolhida e aquele estado pode ser ignorado.

3.2 Resultados

A implementação atual alcança seu resultado rapidamente, em menos de 10ms, quando utilizada a verificação dos estados. Tecnicamente o resultado ótimo é adquirido, pois os passos da busca `A*` foram seguidos corretamente, porém, durante diversas pesquisas, nada foi achado na Internet para comparação das saídas apontadas pelo código.

3.3 Complexidade

A complexidade do algoritmo em termos de tempo e memória, depende da heurística, pois a mesma, como já foi dito, é aquela que auxilia na busca pela solução, informando o quão próximo o algoritmo esta desta. Caso o espaço não tenha limites, a complexidade se torna exponencial, com todos os nós expandidos:

$$O(b^d)$$

, sendo `b` o fator de ramificação (WELLING, 2012).

Fazendo o uso de uma boa heurística, que permite o corte de ramificações desnecessárias, a complexidade se torna polinomial, dado que o espaço de busca é uma árvore e há somente um estado meta:

$$O(\log h^*(x))$$

, para:

$$|h(x) - h^*(x)|$$

, onde `h*(x)` é a heurística ótima, aquela que apresenta o custo exato para alcance do objetivo(WIKIPEDIA,).

3.4 Heurísticas

Consoante o que já foi discutido, a decisão da heurística correta é a chave para o bom resultado do algoritmo.

Aqui foi adotada a soma das distâncias entre a posição atual de uma peça e o local onde ela deveria estar, através da `Distância de Manhattan`.

Esta é caracterizada enquanto uma heurística admissível por nunca sobrepor o valor máximo possível para ela, dado que efetua seus cálculos a partir do posicionamento dos elementos dentro da matriz.

4 Conclusão

Foi bastante interessante perceber que sem a implementação das verificações de condição, o tempo levado pelo algoritmo para apresentar uma resposta tangia mais de 1 segundo, chegando a quase 1 segundo e meio.

No mais, a codificação e entendimento do problema se saíram bem mais fácies do que o esperado, e a modularização dos métodos foi um dos pontos altos para clareza do código e dos passos a serem seguidos.

Referências

WELLING, M. *A* Search*. 2012. <https://www.ics.uci.edu/~welling/teaching/ICS175winter12/A-starSearch.pdf>. (Accessed on 10/08/2017). Citado na página 6.

WIKIPEDIA. *A* search algorithm*. <https://en.wikipedia.org/wiki/A*_search_algorithm#Complexity>. Citado na página 6.