Universidade Estadual de Maringá
Centro de Tecnologia - Departamento de Informática
Ciência da Computação
6903 - Modelagem e Otimização Algoritmica
Professor Ademir Constantino

Segunda Avaliação

Aluno: Rafael Cortez Sanches
RA: 82357

Maringá, 11 de janeiro de 2016

Conteúdo

1 Resumo

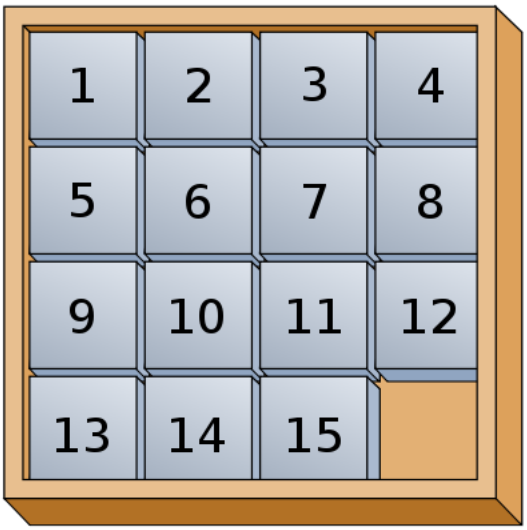
Nesse trabalho, foi implementado o algoritmo A* (A-Estrela) para solucionar o problema do tabuleiro de 15 peças, com o objetivo de analisar o comportamento da solução para diversos casos, utilizando-se diferentes heurísticas.

Para a implementação, a linguagem escolhida foi C++. Os testes foram realizados em executáveis gerados pelo compilador GCC, sendo que tanto a compilação quanto a execução foram realizadas em Linux Ubuntu de 64 bits.

Foram utilizadas cinco heurísticas no total, aplicadas a 10 diferentes casos de teste. Mesmo com a mais eficiente das heurísticas, quatro desses casos falharam por estouro de memória.

2 O Problema

Tem-se um tabuleiro com 15 peças móveis e um espaço vazio, o qual permite que as peças deslizem para ocupá-lo, levando o tabuleiro a uma nova configuração. A figura a seguir ilustra esse mecanismo:



No exemplo ilustrado, as peças 12 e 15 poderiam ser deslizadas até o espaço vazio, trocando a posição da peça com essa brecha. O problema consiste no seguinte enunciado:

Dado um tabuleiro com suas peças fora de ordem, qual o menor número de movimentos necessário para levar o tabuleiro de volta a seu estado ordenado?

Para os casos de teste do problema, foi considerado como estado ordenado o seguinte tabuleiro, com os números dispostos ordenadamente em "caracol":

Tabela 1: Tabuleiro ordenado

1	2	3	4
12	13	14	5
11	0	15	6
10	9	8	7

Sendo a peça "0"o espaço vazio.

3 O Algoritmo A*

Se fossem listados todos os estados possíveis do tabuleiro, teria-se um total de $16! = 10,461,394,944,000$ estados. Buscar a melhor solução em um grafo com esse número de nós seria impraticável pelos métodos clássicos de busca em largura e em profundidade.

O algoritmo A* foi baseado no algoritmo de Dijkstra, que é usado para encontrar caminhos mínimos em grafos com arestas de peso não negativo. O A* tem seu processo de busca agilizado por se utilizar de heurísticas para avaliar o melhor caminho a se percorrer, eliminando computações desnecessárias.

3.1 Grafo Problema

Os nós do grafo problema são configurações diferentes do tabuleiro de 15 peças. A partir de uma dessas, é possível trocar o espaço em branco com peças adjascentes a ele, gerando de 2 a 4 sucessores.

Para a aplicação do A* no problema do tabuleiro de 15 peças, definimos os seguintes elementos:

t_f = Tabuleiro final, com os números dispostos em caracol, como ilustrado na seção 2.
t_i = Tabuleiro inicial, a partir do qual as peças devem ser deslocadas para se chegar no tabuleiro final.
A = Conjunto dos nós abertos, tabuleiros que podem estar na solução ótima.
F = Conjunto dos nós fechados, tabuleiros que já foram avaliados.
$g(t)$ = Atributo de um tabuleiro t, usado para identificar o número de passos utilizados para se chegar em t a partir de t_i.
$h'(t)$ = Atributo de um tabuleiro t, retornado pela função heurística do problema. O valor de $h'()$ é uma estimativa de quão próximo de t_f está o tabuleiro t.
$f(t)$ = Atributo de t, definido por $f(t) = g(t) + h'(t)$.

3.2 Resolução do Problema

Inicialmente, o nó t_i é construído e inserido em A. Seu valor $g(t_i)$ é 0 e ele não possui tabuleiros que o precedam.

No laço principal do algoritmo, o nó com menor valor de $f(t)$ é extraído de A e inserido em F. Em seguida, seus sucessores t_s são gerados e avaliados de acordo com as seguintes condições:

- t_s **já está em A:** nesse caso, verifica se o nó antigo possui valor de $g(t_s)$ maior que o recém descoberto. Isso significa que o algoritmo achou um caminho mais curto de t_i a t_s. Para registrar essa mudança, atualiza-se $g(t_s)$.

- t_s **está em F:** verifica-se $g(t_s)$ como acima. Se o novo valor de $g(t_s)$ é menor que o antigo, remove-se o antigo t_s de F e insere-se o novo t_s em A.

- t_s **não está nem em A, nem em F:** encontrou-se um novo nó para ser avaliado. Insere-se t_s no conjunto A.

- t_s **é o nó final do problema (t_f):** o tabuleiro foi resolvido com o menor número de movimentos. O nó principal é interrompido e a solução final é avaliada através de um backtracking a partir de t_f.

3.3 Heurísticas Utilizadas

Para qualquer nó t valor de $f(t)$ depende diretamente de $h'(t)$, que é um valor arbitrado de acordo com a heurística utilizada. Como o valor de $f(t)$ influencia em quais nós do conjunto A serão vasculhados primeiro, especula-se que a escolha da heurística tenha um grande impacto na execução do algoritmo.

Nesse trabalho, foram implementadas e avaliadas 5 heurísticas diferentes para o A* aplicado ao problema do tabuleiro de 15 peças:

$h'1(t)$: número de peças foras de seu lugar na configuração final.
$h'2(t)$: número de peças fora de ordem na sequência numérica das 15 peças, seguindo a ordem das posições no tabuleiro.
$h'3(t)$: para cada peça fora de seu lugar somar a distância retangular (quantidade de deslocamentos) para colocar em seu devido lugar. Neste caso considera-se que o caminho esteja livre para fazer o menor número de movimentos.
$h'4(t)$: $p1 \times h'1(t) + p2 \times h'2(t) + p3 \times h'3(t)$, sendo que $p1$, $p2$, $p3$ são pesos (números reais) tais que $p1 + p2 + p3 = 1$. A escolha desses pesos deverá ser realizada conforme os resultado dos experimentos.
$h'5(t)$: $max(h'1(t), h'2(t), h'3(t))$.

4 Implementação

Para a implementação do algoritmo enunciado, foi utilizada a linguagem C++. Sua escolha se deve pela eficiência de execução e pelos tipos abstratos de dados já implementados em sua biblioteca padrão.

4.1 Classe Tabuleiro

Os nós do grafo problema foram definidos pela classe "Tabuleiro", contida nos arquivos "*tabuleiro.h*"e "*tabuleiro.cpp*". A seguir, ela está representada de forma resumida (sem getters e setters):

```
1  class Tabuleiro
2  {
3          private:
4                  string id;
5                  int g;
6                  int f;
7                  int heuristica;
8                  string p;
9
10         public:
11                 int casa[16];
12                 Tabuleiro();
13                 ~Tabuleiro();
14                 const bool operator<(const Tabuleiro rhs) const;
15                 string Calcula_ID();
16                 int Calcula_H();
17                 int Le_Tabuleiro_STDIN();
18                 void Imprime_Tabuleiro();
19                 void Gerar_Vizinhos(vector <Tabuleiro> &lista);
20 };
```

A *string id* é uma cadeia de caractéres única para cada tabuleiro. Seu valor é usado para indexar os tabuleiros em um dicionário (std::map), o qual é utilizado na função principal para mapear os conjuntos A e F. Esse identificador é calculado pelo método *int Calcula_ID()*.

O inteiro *int heuristica* representa a heuristica utilizada para obter o valor de $f(t)$ para o tabuleiro em questão. Ele assume valores constantes, definidos em macros no início do arquivo de cabeçalho "*tabuleiro.h*".

A *string p* guarda o identificador do predecessor de t, para que seja feito o backtracking ao fim da execução, caso seja encontrada uma solução.

Os valores de $g(t)$ e $f(t)$ são atributos da classe, sendo que $h'(t)$ pode ser acessado pela operação $h' = f - g$. O valor de $f(t)$ é atualizado quando se chama o método *int Calcula_H()*.

O operador "<"foi definido para objetos da classe Tabuleiro para que fosse possível utilizá-los em uma fila de prioridade (std::priority_queue). Essa estrutura permite que o nó de menor f seja acessado rapidamente.

O método *void Gerar_Vizinhos(vector <Tabuleiro>)* constrói os tabuleiros

sucessores do tabuleiro em questão, inserindo-os em um vetor de objetos Tabuleiro, o qual é passado por referência ao método.

4.2 Função Principal

A função principal do código, apresentada no arquivo "*main.cpp*", contém os procedimentos necessários para executar o algoritmo A* a partir de um tabuleiro inicial.

Os conjuntos A e F foram mapeados com auxílio de dicionários do tipo std::map. Para essas estruturas, foram utilizadas as *strings* identificadoras dos nós como chaves de consulta. A biblioteca padrão do C++ implementa dicionários como árvores binárias de busca, o que permite que seja verificado em tempo $O(lg(n))$ se um elemento está não em um dado conjunto.

Para mapear o conjunto A, também há a necessidade de se consultar rapidamente o nó com menor valor de $f(t)$. Para isso, criou-se uma fila de prioridade para armazenar os mesmos nós do dicionário A, o que permite acessar o menor $f(t)$ em tempo $O(lg(n))$. Essa fila de prioridade é denotada por "*priority_queue <Tabuleiro> PQueue*".

4.3 Redundâncias em PQueue

Inicialmente, implementou-se a manutenção da fila de prioridade da seguinte forma:

- Quando havia a necessidade de alterar o valor de $g(t)$ de um elemento no conjunto A, uma operação "DECREASE_KEY(t)"era chamada. Esse procedimento está enunciado em CORMEN et al, página 118, e tem custo $O(lg(n))$ em tempo de execução.

- Para encontrar esse elemento no heap, era necessária uma busca linear, uma vez que os elementos do heap não estão ordenados. Esse procedimento, por sua vez, custa $O(n)$ em tempo de execução.

Entretanto, para melhorar o tempo de execução, foi descartada a busca linear enunciada acima. Dessa forma, quando um nó pertecente a A é encontrado com um novo valor de $g(t)$, esse nó é simplesmente adicionado à fila de prioridade, deixando na fila um nó de mesma configuração, exceto pelo valor de $g(t)$. A presença desses nós foi denominada como **redundâncias em PQueue**.

Essas redundâncias foram tratadas da seguinte forma: a cada vez que o nó de menor $f(t)$ é sacado do heap, é avaliado se ele pertence ou não ao conjunto A. Caso não pertença, ele é uma redundância, sendo descartado. Esse procedimento é repetido até se encontrar um nó não redundante.

Apesar de acrescentar uma certa carga de operações à função principal, esse tratamento de redundâncias se utiliza de funções de busca em dicionário e extração de valor mínimo em heap. Ambos tipos de operação custam $O(lg(n))$ em tempo de execução, que é assintoticamente menor que $O(n)$. Empiricamente, essa solução mostrou-se melhor em eficiência.

4.4 Pesos de h'4(t)

Os testes com as três primeiras heurísticas mostraram uma eficiência demasiadamente superior de $h'3(t)$ em relação às outras duas, sendo $h'2(t)$ a menos eficiente. Tendo em vista esses resultados, escolheu-se os seguintes valores de peso para $h'4(t)$:

$p1 = 0.08$
$p2 = 0.02$
$p3 = 0.90$

Esses pesos estão definidos como macros, presentes no cabeçalho "*tabuleiro.h*".

5 Testes

Na execução do programa, pode-se selecionar a heuristica pelo parâmetro "−− *heuristica* H", em que "H"é um número de 1 a 5 (O valor padrão é 3). Algumas informações extras podem ser obtidas durante a execução se acrescentado o parâmetro "−− *eco*". Entre essas informações estão: tamanho dos conjuntos A e F, tamanho de PQueue, número de redundâncias tratadas e tempo de execução.

Mais informações de como compilar e executar o código estão contidas no arquivo *"readme.txt"*, que se encontra no diretório dos arquivos-fonte.

Os testes a seguir foram realizados em um ambiente Linux, SO versão de 64 bits, 4GB de RAM de sistema.

Para os casos em que a memória utilizada pelo processo excedeu a quantidade de RAM disponibilizada pelo sistema, optou-se por abortar o processo, uma vez que o uso de memória swap deixa o processo muito mais lento. Nesses casos, o tempo de execução foi marcado com um "−".

Tabela 2: Tempo de execução (em segundos)

	Heurística 1	Heurística 2	Heurística 3	Heurística 4	Heurística 5
Caso 1	< 0.01	1.57	< 0.01	< 0.01	−
Caso 2	−	−	15.75	16.89	−
Caso 3	−	−	55.96	109.53	−
Caso 4	< 0.01	32.17	< 0.01	< 0.01	−
Caso 5	3.35	−	0.06	0.14	−
Caso 6	−	−	−	−	−
Caso 7	−	−	−	−	−
Caso 8	−	−	−	−	−
Caso 9	−	−	−	−	−
Caso 10	−	−	127.08	129.70	−

Tabela 3: Uso máximo de memória (em MB)

	Heurística 1	Heurística 2	Heurística 3	Heurística 4	Heurística 5
Caso 1	3.24	45.26	3.24	3.26	2,797.88
Caso 2	2,782.05	2,767.46	368.69	389.84	2,797.77
Caso 3	2,774.96	2,765.51	1,210.16	2,398.58	2,766.53
Caso 4	3.14	687.65	3.31	3.15	2,736.37
Caso 5	88.76	2,757.79	5.19	7.52	2,729.52
Caso 6	2,771.25	2,758.28	2,769.60	2,805.26	2,728.71
Caso 7	2,772.73	2,770.37	2,766.63	2,803.12	2,730.2
Caso 8	2,779.50	2,780.82	2,762.08	2,802.41	2,727.43
Caso 9	2,777.38	2,778.92	2,761.42	2,798.98	2,727.7
Caso 10	2,775.52	2,769.46	2,496.38	2,645.8	2,726.16

A tabela a seguir mostra o número mínimo de movimentos encontrado para cada um dos casos:

Tabela 4: Número Mínimo de Movimentos

Caso	Número de Movimentos
Caso 1	8
Caso 2	36
Caso 3	41
Caso 4	13
Caso 5	28
Caso 6	?
Caso 7	?
Caso 8	?
Caso 9	?
Caso 10	42

O número mínimo de movimentos para os casos 6, 7, 8 e 9 não foi identificado pelo programa. Observando a disposição dos dados, especula-se que esses valores sejam maiores que 42.

6 Conclusões

Os resultados mostraram $h'3(t)$ como a mais eficiente heurística para o A* aplicado ao problema do tabuleiro de 15 casas. Para o caso 5, por exemplo, ela é 56 vezes mais eficiente que $h'1(t)$ e 2 vezes mais eficiente que $h'4(t)$.

Mesmo $h'3(t)$ sendo uma boa heurística, tanto o tempo de execução quanto o uso de memória aumentam muito a partir de 42 peças, o que impossibilitou a análise dos casos 6, 7, 8 e 9.

As seguintes sugestões são enumeradas para otimizar trabalhos futuros com o código apresentado:

- **Utilizar tipos abstratos de dados mais eficientes para A e F:** No trabalho foram utilizados dicionários std::map para verificar a presença de nós nos conjuntos a partir da string identificadora de cada nó. Essa consulta poderia ser mais rápida em um código que utilizasse árvores rubro-negras ou tabelas hash para implementar os conjuntos do problema.
- **Eliminar a necessidade do tratamento de redundâncias:** Se fosse utilizado um TAD que eliminasse os casos de redundância e simultaneamente permitisse rápida consulta ao nó de menor $f(t)$, a execução da função principal seria agilizada.
- **Podar nós de A e F:** Se fossem identificados nós que certamente não entrariam na solução final, esses poderiam ser "podados"dos conjuntos A e F, o que diminuiria o consumo de memória. Entretanto, não se sabe quais tipos de nós são passíveis de serem removidos do problema durante a execução.

7 Referências

CORMEM, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos: Teoria e Prática**. 3ª Edição. Elsevier, 2012.