

Relatório EP2 - MAC0121

João Gabriel Basi - N° USP: 9793801

1. O programa

Utiliza a técnica de backtracking para achar uma solução para o jogo de tabuleiro "Resta Um". O programa recebe as dimensões do tabuleiro e um matriz com 0, -1 e 1, representando "sem buraco", "buraco sem peça" e "buraco com peça" respectivamente, e retorna os movimentos realizados para resolver o tabuleiro ou "Impossível" se não há como resolvê-lo.

2. As funções

Foi criada uma struct pilha que contém um vetor de triplas (a posição da peça e o movimento executado), além de variáveis para o topo da pilha e seu tamanho máximo.

Foram criadas também algumas funções sobre essa struct:

- *criaPilha*: Aloca uma pilha de número de linhas especificado;
- *pilhaVazia*: Verifica se a pilha está vazia;
- *pilhaCheia*: Verifica se a pilha está cheia;
- *pilhaEstourou*: Verifica se a pilha tem mais camadas do que seu máximo, ou seja, se o número de peças retiradas é maior que o de buracos no tabuleiro final;
- *empilha*: Empilha uma posição e um movimento na pilha;
- *desempilha*: Desempilha a posição e o movimento que estão no topo da pilha;
- *imprimePilha*: Imprime as posições de início e fim de cada movimento guardado na pilha;
- *freePilha*: Desaloca uma pilha.

Foram criadas funções para ajudar na manipulação de matrizes e vetores:

- *criaVetor*: Aloca um vetor de tamanho especificado;
- *criaMatriz*: Aloca uma matriz de número de linhas e colunas especificados;
- *podeMover*: Verifica se a peça pode ser movida e indica a direção do movimento;
- *mexe*: Executa ou volta um movimento;
- *freeMatriz*: Desaloca uma matriz.

E foram criadas algumas funções para verificações sobre matrizes:

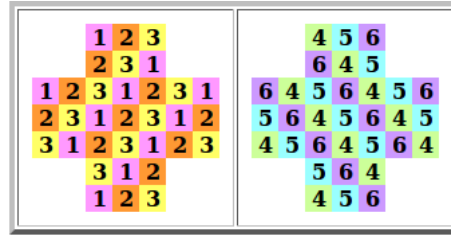
- *concluido*: Verifica se as posições que estavam livres no começo estão ocupadas;

- *ehPossivel*: Faz alguns testes (especificados no item 3) na distribuição das peças do tabuleiro para filtrar alguns tabuleiros insolúveis;
- *floodFill*: Utilizada pela função *ehPossivel* para verificar se o tabuleiro tem partes desconexas;

3. Conceitos matemáticos e simplificações utilizados

Na função *ehPossivel* foram feitos três testes para identificar se um tabuleiro tem potencial para ser resolvido:

- O primeiro teste checa se N° de peças $\geq 2 \cdot N^\circ$ de espaços iniciais, pois, para cada buraco sem peça do tabuleiro inicial é preciso de, no mínimo, um movimento, e cada movimento precisa de duas peças para ser executado, então, se o tabuleiro tem n buracos, é preciso de, pelo menos, $2n$ peças para resolvê-lo.
- O segundo teste leva em conta a classe de posições do tabuleiro. Conforme foi descrito no site Recmath[1], a classe de posições do tabuleiro é definida enumerando as diagonais do tabuleiro do seguinte modo:



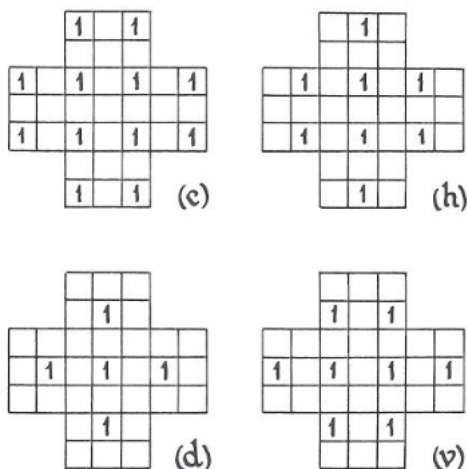
Numeração no resta um tradicional[1]

A partir disso, definimos uma função N_i sobre o tabuleiro, que retorna a quantidade de casas ocupadas marcadas com número i , e a função T , que retorna o total de casas ocupadas. Com isso definimos a classe de posições do tabuleiro como sendo a 6-upla da forma $(T - N_1, T - N_2, T - N_3, T - N_4, T - N_5, T - N_6) \bmod 2$ (no site é colocado como a "paridade" desses números, mas no programa eu considere como módulo 2). Definidas as classes, observamos que a cada movimento executado a paridade dos númes dessa 6-upla não muda. Pegando como exemplo o tabuleiro da imagem, com a posição central livre e as outras ocupadas, vemos que, ao mexer qualquer peça para a posição central, os N_2 e N_5 aumentam em 1 e os N_1, N_3, N_4, N_6 e T diminuem em 1, então, para os N s que diminuíram, temos que $N - 1 - (T - 1) \equiv N - T \bmod 2$, e para os que aumentaram $N + 1 - (T - 1) \equiv N - T \bmod 2$, vemos que essa regra vale para todo o tabuleiro e, concluímos que, a partir de um tabuleiro com certa classe de posição, só é possível atingir tabuleiros com a mesma classe, então temos que os tabuleiros final e inicial têm que estar na mesma classe, se não o tabuleiro é impossível. Lembrando que esse teste indicar que um tabuleiro é impossível é suficiente para que ele seja impossível, mas não é necessário.

- O terceiro leva em conta a possibilidade de tabuleiros com partes desconexas, checando com a função *floodFill* se, sempre quando há uma parte desconexa, ela tem no mínimo uma casa vazia, para que todas as peças possam ser movidas.

Outra simplificação utilizada foi uma função contadora de recursos (também chamada de Pagoda) introduzida pela primeira vez no volume 4 do livro "Winning ways for your mathematical plays". Para calcular a função temos que atribuir um valor de forma estratégica para cada casa do tabuleiro. O valor da função em um determinado

tabuleiro é a soma dos valores das casas ocupadas. Como os valores das casas não são negativos e uma peça sempre é retirada do tabuleiro por movimento, o valor da função não pode aumentar, portanto, se o valor do tabuleiro atual for menor que o valor do tabuleiro final, o conjunto de movimentos executados não resolverá o jogo.



Exemplos de funções Pagoda[2]

Utilizando essa lógica, podemos marcar um tabuleiro inteiro com 1s, então o valor da função em determinado tabuleiro é o seu número de peças. Já que o número de peças não pode ser aumentado, se o valor da função no tabuleiro atual for menor que o valor no tabuleiro final, o tabuleiro não pode ser resolvido.

No programa a função *pilhaEstourou* checa essa condição a cada movimento, e, se ela retornar 1, ele desmpilha o último movimento.

4. Informações sobre os testes realizados

Considerando como núcleo a formação 3x3 central do tabuleiro tradicional (imagem do item anterior) com a posição central livre e as outras ocupadas, e como braços as quatro partes restantes, escrevemos que um tabuleiro é *nnnn* se tem o núcleo e 4 braços 3xn, contados a partir do de cima em sentido horário (notação encontrada no site citado no item 3).

- Testei com um tabuleiro 1111, que passou nos testes da *ehPossivel*, mas, depois de 140seg, o programa concluiu que ele é impossível;
- Testei para um tabuleiro 1111 com o buraco central e mais dois opostos e adjacentes a ele, que o programa concluiu ser impossível em 25seg;
- Testei com o tabuleiro 2222 que ele resolveu em pouco menos de 1seg;
- Testei em um 6x4 com um buraco em vários lugares diferentes, que ele resolveu em pouco menos de 1seg;
- Testei para um 2121, que ele resolveu em 130seg.
- Testei para o tabuleiro francês, um tabuleiro com um núcleo 5x5, e 4 braços 3x1 (também descrito em [1]), e o programa concluiu, ainda no teste das classes, que o tabuleiro é impossível.

5. Prós e contras

- **Prós**

- Identifica boa parte dos tabuleiros que são impossíveis no começo do programa, deixando de gastar tempo tentando resolvê-los;
- Otimiza um pouco o tempo de execução de tabuleiros com muitos buracos checando a condição da *pilhaEstourou*.

- **Contras**

- Tentei implementar uma otimização usando uma função Pagoda um pouco melhor, mas, a pesar dessa solução diminuir o número de movimentos feitos, ela não diminui o tempo gasto na resolução, então não a implementei e o programa continua sem muita otimização.

Referências

- [1] Imagem retirada do site <http://recmath.org/pegsolitaire/index.html#pre>
- [2] Imagens retiradas do livro "Winning ways for your mathematical plays" vol. 4