Relatório de MAC0121 - EP2

Juliano Garcia de Oliveira Nº USP: 9277086

---

1. Especificação do Exercício

O EP2 consiste na resolução de um clássico jogo de nome "Resta Um" ou "peg solitaire". O jogo a ser resolvido é uma versão generalizada do Resta Um, no qual o jogo está resolvido quando todas as peças ocupam o lugar dos buracos vazios iniciais, e onde havia as peças iniciais no final há buracos vazios. As dimensões e o formato do tabuleiro são passados pela entrada padrão para o programa.

OBS: A especificação do EP diz que ele deve ser feito utilizando a técnica de Backtrack, porém não menciona (nem proíbe) o uso da técnica de Backtrack usando recursão (que foi aprendida em aula). Todo o meu EP tinha sido desenvolvido utilizando esta técnica de Backtrack, porém **3 dias antes da data de entrega** , o monitor decidiu que não aceitaria o EP feito usando recursão, apenas com pilhas (mesmo que a técnica de Backtrack com recursão tivesse sido ensinada em classe). Portanto meu EP foi convertido em menos de 3 dias para a versão usando pilhas, para ser entregue na "especificação" de última hora corretamente.

2. Algoritmo

O algoritmo usa uma pilha para implementar o Backtrack. A pilha é composta de um vetor que em cada casa armazena uma coordenada da peça e o movimento feita por ela. O algoritmo funciona do seguinte modo:

- Enquanto ainda houver peças com movimentos disponíveis:
    - Tentar executar um movimento com a peça:
        - Se conseguiu executar o movimento: Empilha o movimento + peça e volta a percorrer o tabuleiro desde o início;
        - Senão, continua a percorrer o tabuleiro;
- Se o tabuleiro satisfaz a condição de vitória , para o algoritmo e imprime os movimentos;
- Se não, se a pilha de movimentos está vazia, quer dizer que tudo foi testado e não se conseguiu a solução. Neste caso é impresso a mensagem "impossível".
- Se a pilha não está vazia, desempilha-se o último movimento e volta para o início do algoritmo, que irá continuar com a última peça e tentar fazer o próximo movimento disponível (**Backtracking**).

3. Funções e estruturas de dados

O EP consiste de 2 arquivos principais (excluindo o *header*):

- *restaUm.c*
- *dataStructs.c*

Utiliza-se duas estruturas de dados principais (excluindo-se as já nativas do C, como vetores e matrizes):

- Pilha de movimentos : Armazena *structs* do tipo *pMovData* , que nada mais é que a coordenada da peça e o número do movimento executado;
- Vetor de posições : Armazena *structs* do tipo *pos*, que é uma posição (com um "*x*" e um "*y*") .

As funções possuem nomes bem autoexplicativos e são divididas em escopos distintos. A explicação de cada função está mais detalhada nos comentários do próprio código *restaUm.c* .

<u>Resta um :</u>

- **Tabuleiro**
  - *newBoard( )* : Cria o tabuleiro (matriz);
  - *getBoardData( )* : Óbtem o número de peças e buracos vazios do tabuleiro, além de preencher um vetor de posições com as posições dos buracos vazios do tabuleiro.
- **Resta Um e movimentos**
  - *solvePeg( )* : Função principal que resolve o Resta Um usando Backtrack e a pilha de movimentos;
  - *doMove( )* : Move um peça se for possível;
  - *undoMove( )* : Desfaz um movimento;
  - *canMove( )* : Verifica se é possível efetuar um movimento;
  - *isSolved( )* : Verifica se o tabuleiro está resolvido (também usa a função abaixo);
  - *allPegsAreHoles( )* : Verifica se todos os buracos vazios estão ocupados por peças.
- **Impressão e miscelânea**
  - *printSolution( )* : Imprime a solução de acordo com a especificação do EP2;
  - *destroy( )* : Libera a memória das estruturas de dados usadas no programa.

As funções do arquivo *dataStructs.c* são comuns e básicas e não precisam de descrição aqui.

4. Otimização e resultados

Primeiramente, as verificações básicas para descobrir se um tabuleiro é impossível não são tão necessárias, já que saem rapidamente do algoritmo de Backtrack, não sendo necessário escrever outra função somente para verificar isto.

A maioria das otimizações executadas no EP2 foram para diminuir a utilização do **espaço** consumido pelo programa. A escolha dos tipos e como montar a pilha de movimentos foi feita pensando nesse fator. Temos os seguintes tipos e sua utilização:

- ***minINT*** : Usa o tipo "char" nativo do C. Como a matriz só é composta de 0, 1 ou -1, e os movimentos só podem ser 1, 2, 3 ou 4, todas essas variáveis não precisam ocupar o espaço que um **int** padrão ocuparia. Portanto, a matriz do tabuleiro e os movimentos possuem tipo **minINT** , para diminuir ao mínimo o consumo de espaço utilizado pelo programa.
- ***jCoord*** : A pilha dos movimentos só precisa de um inteiro *j* que é a coordenada de uma peça, diferentamente da maioria das implementações que provavelmente deve ter usado dois inteiros. A conversão do *j* para as coordenadas na matriz é simples e rápida, sendo simplesmente " tab[*j/n][j%n*] " a coordenada correspondente, onde *n é o número de colunas da matriz.*

O problema do resta Um é **NP completo[1]**, portanto não é trivial simplificar o problema. Achei vários modos de simplificar a árvore de possibilidades do programa.

O mais simples seria utilizar um *bitmask* para cada matriz que eu verificasse impossível de resolver. O *bitmask* da matriz seria armazenado em um *set* ou em uma *Hash Table* , e a cada iteração do algoritmo, verificar se o tabuleiro em questão está contido no meu *set / Hash Table* , e fazer o Backtracking caso estivesse. O problema é implementar tal simplificação na linguagem C, que não possui nenhum desses tipos (*set* e *Hash tables*) por padrão. Fiz um programa do Resta Um em Python que usava o tipo *set* do Python, mas para implementá-lo em C eu teria que ter conhecimentos de "Árvores Rubro negras" ou "Árvores de Busca", que ainda não vimos no curso. Portanto a melhor otimização que consegui achar ainda não possuo o conhecimento necessário para implementá-la.

Outra otimização seria usar o que se chama *pagoda Function* [2] , que basicamente calcula uma soma usando a paridade das peças no tabuleiro. Não consegui achar muito mais sobre esta função, e o que achei valia apenas para casos quando temos apenas um buraco no tabuleiro. Por fim, outras otimizações usavam conceitos de programação linear e DFS, e no artigo *Integer Programming Based Algorithms for Peg Solitaire Problems* há a utilização de várias dessas técnicas avançadas, mas parecem muito avançados para a proposta deste EP e difícil de entender o funcionamento.

Testes e resultados :

Os testes feitos incluem a matriz 7 x 7 descrita no enunciado do EP2, matrizes menores para teste de funcionalidade e corretude (3 x 3 com várias configurações diferentes e 4 x 4 com 2 buracos). Para estes casos os resultados não demoram muito, levando na média 20 segundos para emitir a resposta quando fosse possível, e um pouco mais de tempo (cerca de 2 minutos) para casos impossíveis. O tempo cresce rapidamente e se torna inviável para resolver tabuleiros com dimensões maiores que 8 x 8, ou ainda tabuleiros com dimensões menores porém com vários buracos vazios. Vários desses testes não terminaram após 1 hora de teste, quando foi abortada a execução do algoritmo.

Os resultados foram satisfatórios para o tipo de técnica usada (Backtracking), já que esta técnica é conhecida por testar todas as possibilidades e não ser a mais eficiente de todas, o resultado não surpreende. A técnica de Backtracking foi aplicada corretamente e infelizmente por limitações de linguagem e (principalmente) do meu conhecimento, não consegui aplicar otimizações mais avançadas como o *Dynamic Programming* e *Memoization*, que consegui utilizar no EP1.

Fontes:

[1]: Uehara, R.; Iwata, S. (1990), "Generalized Hi-Q is NP-complete", Trans. IEICE, 73: 270–273

[2]: Berlekamp, E. Conway, J. H. and Guy, R. K.: Winning Ways for Mathematical Plays. Academic Press London 1982

Integer Programming Based Algorithms for Peg Solitaire Problems, Masashi KIYOMI. Department of Mathematical Engineering and Information Physics, University of Tokyo.

George's Peg solitaire Page.