TRABALHO PRÁTICO 0

Passeio do Cavalo

Wagner Vinícius Guimarães Lopes

Departamento de Ciência da Computacão – Universidade Federal de Minas Gerais (UFMG)
wagnerl@dcc.ufmg.br

***Resumo**: Este documento descreve um sistema que encontra um caminho valido para um cavalo de xadrez em um tabuleiro de tamanho N x M. Para que o caminho seja válido o cavalo deve passar em todas as casas do tabuleiro sem repetir nenhuma. O sistema também utiliza uma estratégia determinística e outra aleatória e exibe algumas informações sobre o comportamento do algoritmo durante a busca da solução. Essas informações foram analisadas com o objetivo de comparar o desempenho dos dois algoritmos.*

1 INTRODUÇÃO

Neste documento esta descrito a implementação de um sistema que, dado um tabuleiro de tamanho N x M e uma posição inicial, tenta encontrar um caminho pelo qual o cavalo do xadrez percorre todas as posições passando por cada uma delas uma única vez. Os principais objetivos do trabalho são familializar com as tecnologias utilizadas nos demais trabalhos práticos e com os médodos de avaliação que serão utilizados pelos monitores nos demais trabalhos práticos.

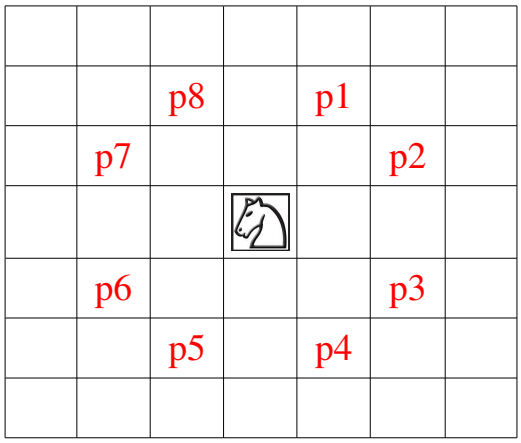
O núcleo do sistema, que é o algoritmo que busca o caminho do cavalo no tabuleiro utiliza a técnica do *backtracking*. Essa técnica consiste em tentar avançar o máximo possível em uma direção no tabuleiro, se o algoritmo chega em uma posição onde não é possível mais realizar movimentos para completar o caminho, os passos são desfeitos até que se encontre um passo anterior com uma possibilidade nova de caminho. Foram implementadas dois algoritmos utilizando a técnica. O primeiro é um algoritmo determinístico que sempre encontrará o mesmo caminho para o mesmo tabuleiro com a mesma posição inicial. O segundo algoritmo utiliza um método aleatório para a escolha dos passos que o cavalo irá seguir. Nesse segundo método existe a possibilidade de se encontrar caminhos diferentes para os mesmos dados de entrada.

2 ANALISE DO PROBLEMA

1. **Problema:**

   Encontrar, se existir, um caminho em um tabuleiro N x M por onde o cavalo do xadres possa, apatir de uma coordenada inicial, passar por todas as cassa sem repetir nenhuma. Se não houve caminho válido, o sistema deve imprimir uma mensagem informando.

2. **Possibilidades de movimentação do cavalo:**



Na analise acima, o cavalo localizado em uma posição “p0” qualquer possui oito possibilidades de movimentação para os pontos destacados em vermelho fazendo as seguintes movimentações na matriz:

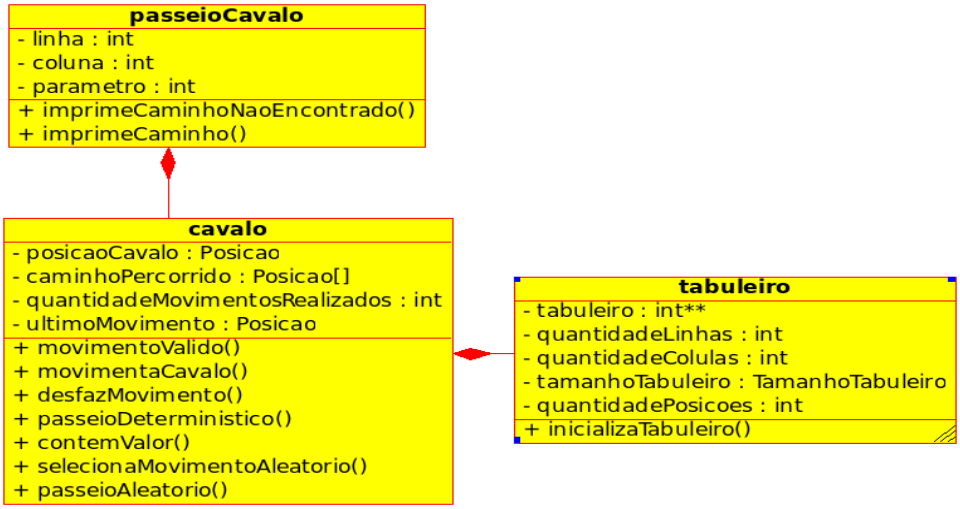
Movimento	deslocamentoLinha	deslocamentoColuna
1	1	2
2	2	1
3	2	-1
4	1	-2
5	-1	-2
6	-2	-1
7	-2	1
8	-1	2

Para realizar a movimentação, criei um vetor de oito posições com os oito deslocamentos possíveis.

3 SOLUÇÃO PROPOSTA

A solução proposta foi a implementação de algoritmos que utilizam o principio do *backtracking* com recursão. Tanto a versão determinística quanto a aleatória possui um while responsávem por testar todas as possibilidades de movimento partindo de uma determinada posição. Isso possibilita que a pilha de execução do método recursivo armazene somente o sub-caminho que está sendo testado.

Para a implementação dessa solução o sistema foi dividido em três módulos. Um módulo de controle, um que representa o cavalo e outro que representa o tabuleiro, como descritos no diagrama de abaixo:



1 Descrição das principais funções:

1.1 cavalo.movimentoValido:

1.1.1 Verifica se o movimento selecionado pode ser executado

1.2 cavalo.movimentaCavalo:

1.2.1 Faz o cavalo se movimentar para frente no caminho

1.3 cavalo.desfazMovimento:

1.3.1 Faz o cavalo retroceder um movimento

1.4 passeioDeterminístico:

1.4.1 Algoritmo determinístico de busca pelo caminho. A estratégia se resume em tentar realizar os movimentos em sentido horário. Isso quer dizer que o primeiro movimento testado será o que está na posição 1hora em relação a posição atual do cavalo, depois 2h, 4h, 5h, etc.

1.5 cavalo.selecionaMovimentoAleatorio

1.5.1 Seleciona, aleatoriamente, o próximo movimento a ser testado. Funcionamento:

1.5.1.1 Recebe um vedor booleano de 8 posições representando as possibilidades de movimento do cavalo.

1.5.1.2 Recebe a quantidade de movimentos já utilizados.

1.5.1.3 Calcula a quantidade de movimentos restantes

1.5.1.4 Utiliza a função rand(time(NULL)) para gerar um número “n” aleatório entre 1 e o número de movimentos restantes.

1.5.1.5 Procura a n-esima posição livre no vetor.

1.5.1.6 Retorna o índice da n-esima posição livre do vetor. Esse será o índice do próximo movimento a ser testado.

1.5.2 cavalo.passeioAleatorio

1.5.2.1 Algoritmo aleatório de busca pelo caminho.

1.6 tabuleiro.inicializaTabuleiro

1.6.1 Aloca espaço para a matriz N x M inicializando todas as posições com zero.

4 Implementação

1. **Código**
   1. **Arquivos**
      1. **passeioCavalo.c**: Módulo de controle. Esse módulo trata as entradas de dados, chama a função que implementa a estratégia selecionada pelo usuário e imprime o resultado.
      2. **cavalo.h**: Implementa as funcionalidades do cavalo descritas na seção anterior
      3. **tabuleiro.h**: TAD que representa o tabuleiro. Possui um vetor bidimencional que representa o tabuleiro, quantidade de linhas, columas, quantidade de posições e uma função que faz a alocação do tabuleiro de acordo com o tamanho selecionado pelo usuário.
2. **Compilação**
   1. Na pasta com os arquivos digite “make”
3. **Execução**
   1. Para executar o programa é necessário digitar:
      1. ./PasseioDoCavalo -n <linas> -m <colunas> -i <linhaInicial> -j <colunaInicial> -e <estrategia> -s <arquivoSaida>
4. Formado da saída

Solução:

5	8	3	18	15	12	21	24
2	17	6	9	22	19	14	11
7	4	1	16	13	10	23	20

Métricas de performance:

Total de movimentos realizados: 57417

Total de movimentos desfeitos: 57393

Quantidade de movimentos antes do primeiro retrocesso: 20

Tempo de execucao: 0.030s

5 ANALISE DE COMPLEXIDADE

Como descrito na seção “SOLUÇÃO PROPOSTA”, os algoritmos que analisam o tabuleiro são algoritmos recursivos mas que possuen um while interno que testa todas as oito possibilidades de movimentação partindo de uma determinada posição. Encontraremos a função de complexidade dada pelo pior caso de execução. Esse pior caso acontecerá, por exemplo, quando o algoritmo testar todas as possibilidades antes de descobrir que o problema não tem solução para as dimenções do tabuleiro. Nesse caso, independente se o cavalo está na borda do tabuleiro ou está no centro, os algoritmos implementados irão testar todas as oito possibilidades, mesmo que seja para descobrir que a posição está fora do tabuleiro, pra depois procurar o próximo movimento. Nesse caso, teremos oito movimentos analisados para cada posição. Sendo assim, temos a seguinte função de complexidade:

Função de complexidade = $8^{\text{colunas x linhas}}$

Ou seja, temos um algorítmo de complexidade exponendial. Isso dificulta muito a análise de tabuleiros de tamanho maior, demandando muito tempo para que se consiga uma resposta

6 PROBLEMAS DURANTE A IMPLEMENTAÇÃO

A maior dificuldade encontrada foi modelar o backtracking de forma a fazer o cavalo passear corretamente. Minha primeira tentativa foi chamar a função recursivamente para todas as possibilidades de movimentos, mas o almento do tamanho da pilha de execução dificultou a implementação. Por fim, decidi fazer um while para testar as oito possibilidades de movimentos de uma posição e chamar a função recursiva somente quanto um caminho válido fosse encontrado. Essa estratégia facilitou o controle da pilha, da movimentação do cavalo e diminuiu a quantidade de memória utilizada, uma vez que a pilha só armazenaria o caminho percorrido pelo cavalo.

7 AVALIAÇÃO EXPERIMENTAL

Nessa seção vamos avaliar as estratégias de busca pelo caminho no tabuleiro. Durante o processamento dos dados, o sistema coleta informações que são apresentados no final da execução e são eles:

Total de movimentos realizados

Quantidade de movimentos antes do primeiro retrocesso

Tempo de execucao

Durante a análise do algoritmo aleatório, percebi que a função rand consumia muito tempo de processamento. Sendo assim, incluí a métrica de performance abaixo que contabiliza, exclusivamente, o tempo gasto pela função:

Tempo da funcao rand

Os testes foram executados em um computador com a seguinte configuração

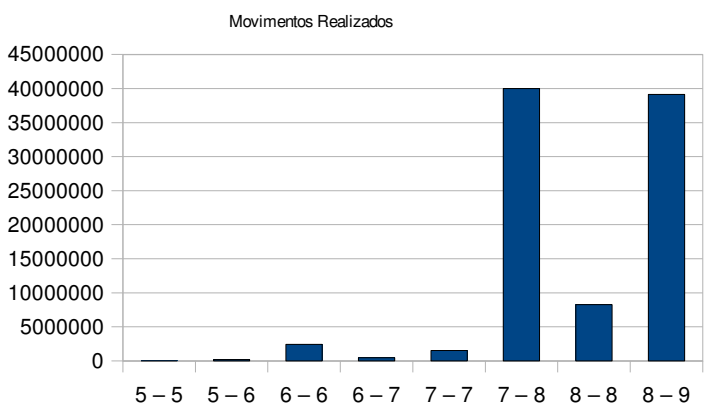
Processador: Intel Dual-Core 1,86GHZ – 1MB cache

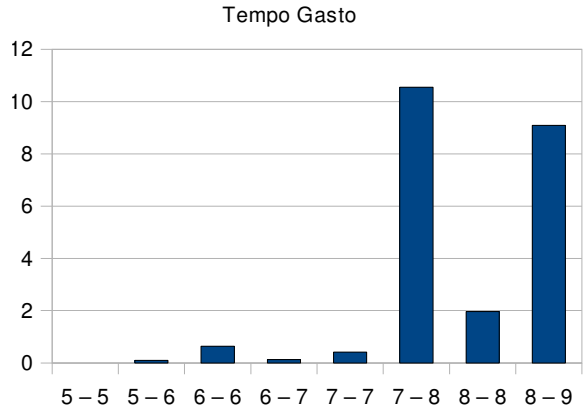
Memória: 1GB

Estratégia determinística

Para essa estrategia foram executados os seguintes testes abaixo:

Linhas – Colunas	Movimentos Realizados	Movimentos sem retrocesso	Tempo gasto
5 – 5	9909	15	0
5 – 6	180789	22	0,1
6 – 6	2436915	30	0,64
6 – 7	483364	22	0,13
7 – 7	1510656	42	0,41
7 – 8	39981691	26	10,55
8 – 8	8250733	36	1,96
8 – 9	39134287	44	9,09





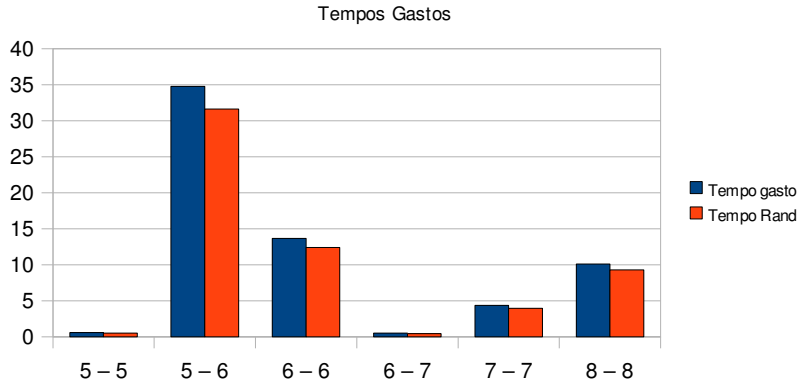
Analisando os dados acima, percebemos que, como era de se esperar, o tempo gasto varia de acordo com a quantidade de movimentos realizados. Issó pode ser visto comparando os gráficos Movimentos Realizados e Tempo Gasto.

Outra constatação interessante que se pode observar pelos gráficos é que, em alguns formatos de tabuleiro, o algoritmo consegue encontrar a resposta mais do que tabuleiros de tamanho menos. Isso ocorre, por exemplo, no tabuleiro 8 x 8, onde o tempo gasto foi muito menor que o necessário para encontrar a solução do tabuleiro 7 x 8.

Estrategia Aleatoria

Para essa estrategia foram executados os seguintes testes

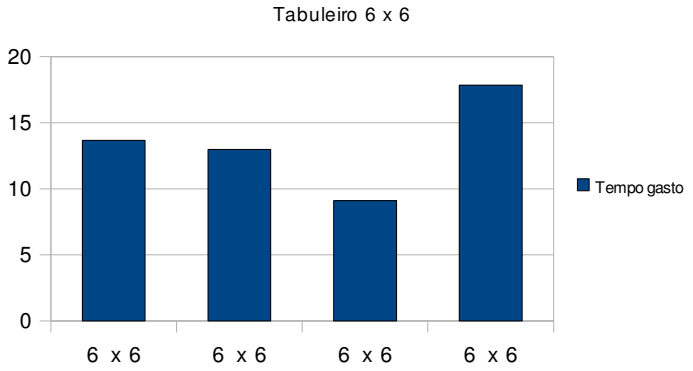
Linhas – Colunas	Movimentos Realizados	Movimentos sem retrocesso	Tempo gasto	Tempo Rand	Tempo sem o Rand	% Rand
5 – 5	12443	11	0,57	0,53	0,04	0,93
5 – 6	822413	21	34,79	31,62	3,17	0,91
6 – 6	320300	26	13,67	12,39	1,28	0,91
6 – 7	11247	33	0,51	0,45	0,06	0,88
7 – 7	98988	27	4,36	3,96	0,4	0,91
7 – 8	4925331	34	206840	186440	20400	0,9
8 – 8	258281	16	10,09	9,3	0,79	0,92



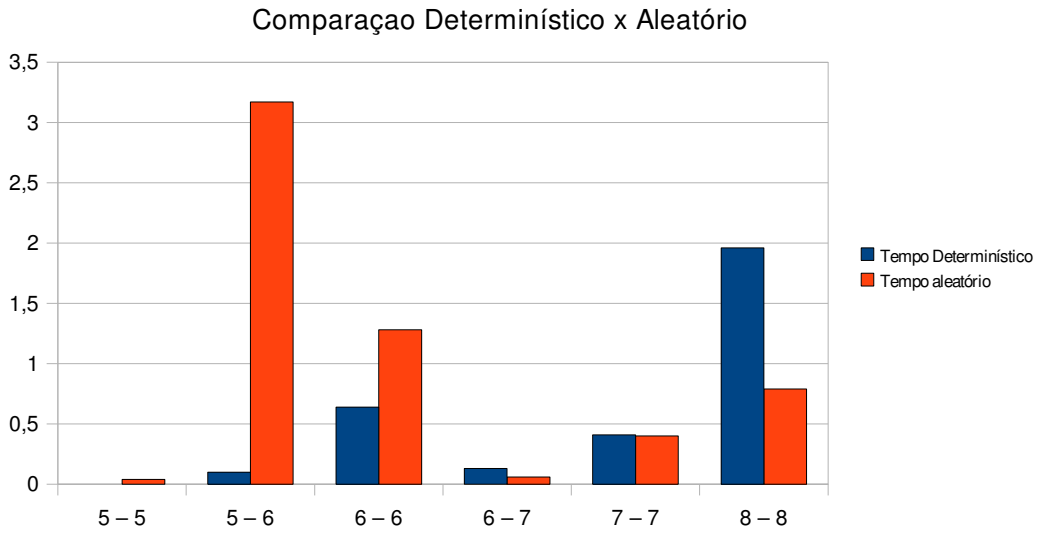
Pelo gráfico ficou comprovado que a função rand está consuminto a maior parte do tempo de processamento do algoritmo de busca aleatório. Pelas medições, é possivel perceber que mais de 90% do tempo de processamento foi gasto nessa função. Isso foi um complicador na hora de executar os testes porque estes demorarim muito tempo para ser executados.

Também por esse motivo, vou comparar o algoritmo determinístico e aleatório, posteriormente, despresando esses tempos.

Como podemos perceber, a estrategia Aleatória apresenta uma grande variação dos tempos de busca. Por esse motivo, a coluna da tabela 7x8 foi removida do gráfico, uma vez que, por ter apresentado um tempo muito maior que os dos outros tabuleiros, essa barra dificultaria a visualização das demais. Sendo assim, para comprovar a aleatoriedade dos tempos gastos, decidi realizar vários testes com o tabuleiro 6 x 6. Os resultados desses testes estão no gráfico abaixo e comprovam a aleatoriedade dos tempos.



Comparação entre estratégias Determinística e Aleatória



No gráfico acima, já está descontado o tempo gasto pela função rand na estratégia Aleatória.

Como podemos perceber no gráfico acima, não existe uma regra que defina qual estratégia é mais eficiente. Como podemos notar, em alguns dos casos de teste a estratégian Determinística foi mais rápida e em outros textes aconteceu o oposto.

8 CONCLUSÃO

A implementação do trabalho permitiu assimilar a técnica do backtracking e a recursão. Alem disso, serviu para relembrar as características da linguagem, o que será muito importante para o desenvolvimento dos demais trabalhos práticos.

9 REFERÊNCIAS

Livro: Título: Projeto de algoritmos: Com implementações em Pascal e C
Autor: Ziviane, Nivio.
Publicação: São Paulo - 1993