Atividade Avaliativa T2

O Caranguejo Perdido

Gilberto Luis Koerbes Junior

Resumo

Algoritmos de busca e caminhamento em grafos estão presentes no nosso cotidiano. Sejam no roteamento por roteadores de redes, rotas para viajantes, ou até mesmo um caranguejo que busca sair de um tabuleiro no menor caminho possível. Nesse relatório será apresentada uma solução para este último problema com algoritmos de busca em largura.

INTRODUÇÃO

Este artigo visa apresentar uma solução para o problema proposto em aula, um desafio baseado em tabuleiros em que um caranguejo C precisa encontrar a saída S utilizando o menor espaço. Na primeira seção, será aprofundado o problema proposto. Na segunda seção uma modelagem para solução. Na terceira, o processo, execução e exemplo dos algoritmos implementados. Ante fim, os resultados objetivos, e como final as conclusões obtidas com o desenvolvimento deste artigo.

1. O PROBLEMA

Conforme proposto pelo enunciado da atividade, o problema busca uma solução de caminhos mínimos, um caranguejo em um tabuleiro com objetivo de uma saída S com restrições durante o percurso de locais que não podem ser utilizados, um tipo de labirinto, e saltos de movimentação fixos conforme a orientação do movimento, duas posições em vertical e horizonta, e apenas uma na diagonal.

2. MODELAGEM DO PROBLEMA

Analisando a movimentação do caranguejo conforme as restrições imposta, pode se perceber que os caminhos permitos formam um um tabuleiro, assim, o caranguejo acaba podendo por se movimento apenas em posições de mesma cor.

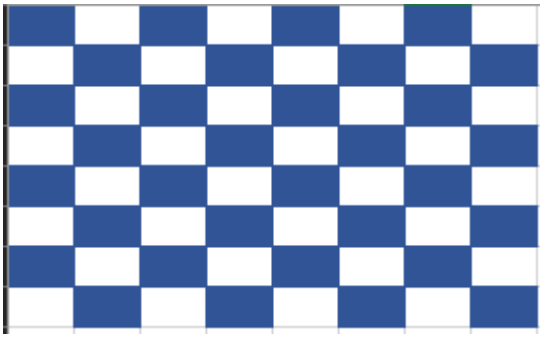

Figura 1 Tabuleiro 8x8

Tabuleiro 8x8

Os algoritmos considerados para a modelagem foram o algoritmo de busca em largura, também conhecido por *Breadth-First Search* (BFS) e o algoritmo de dijkstra. Como o problema considera que qualquer movimento tem peso 1, indiferente da distância saltada, foi descartado o algoritmo de dijkstra, pois este utiliza desse recurso para verificar os caminhos mínimos.

Assim, o algoritmo possui como pré-requisitos as marcações dos vértices visitados e as arrestas vizinhas. O pseudo-código sugere que sejam enfileirados os nós visitados, assim, como as filas utilizam o princípio de FIFO (*Fisrt In – First-Out*), esses nós vão sendo desenfileirados e processados seus vértices. Para que não haja um loop entre os nós visitados, é adicionado um rótulo (*label*) ao vértice, assim, se o mesmo já foi visitado, não será novamente. Desse modo, assume-se também que, se esse vértice já foi visitado, o caminho mais próximo (em saltos) foi o pai (*father*)que o encontrou.

No problema modelo, os vértices a serem considerados são os pontos de mesma cor, e as arrestas os movimentos permitidos entre esses vértices.

1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8
6,1	6,2	6,3	6,4	6,5	C	6,7	6,8
7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8
8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8

Figura 2 Matriz com posições

Considerando que o caranguejo esteja em 6,6 suas possíveis movimentações são 5,5 4,6 6,8 7,7 8,6 e 6,4. Assim essas posições adjacentes são colocadas em fila e marcadas como visitadas, quando retidas, seu processamento será repetido em seus vizinhos. A marcação é importante para evitar que loops aconteçam, como por exemplo, 6,6 visita seu vizinho 5,5, e

em seguida, 5,5 visita seus vizinhos, voltando a posição 6,6, e assim sucessivamente. A adição de um rótulo (*label*) evita esse problema. A visita se sucede da seguinte forma:

Quadrantes mais claros são posições com *labels* e as setas indicam as opções disponíveis(vértice 5,5).

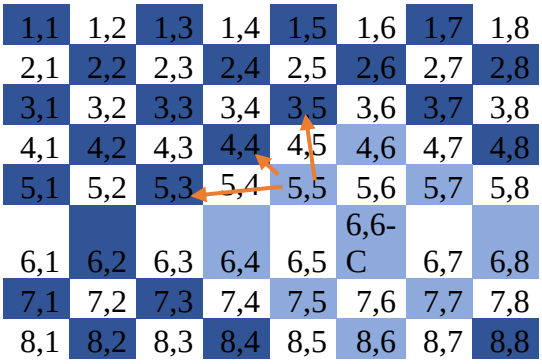

Figura 3 Matriz com vértices marcados

3. IMPLEMENTAÇÃO:

Os códigos gerados estão disponíveis em https://github.com/gilbertokoerbes/AlestII em repositório público(após data limite de entrega deste trabalho). O programa está divido com as seguintes classes:

- App;
- Leitura;
- Vertices;
- BuscaCaminho;

A classe *App* faz apenas as chamadas para as outras classes organizando a execução do programa. A classe leitura lê os arquivos de testes. Como base, utiliza uma matriz de lista de *Array (ArrayList)*, implementado de forma dinâmica, sendo um *array* dentro de *array*. O *array* mais externo representa as linhas, e ao acessar uma posição, internamente há um novo *array* que representa as colunas, formando por fim a matriz. Cada combinação (i,j) armazena uma instância de *Vertices*, uma representação de nodo.

A classe *Vertices* foi implementada justamente para que dado um objeto, possa se inserir atributos. Possui os seguintes atributos, além dos *getters* e *setters* necessários para manipulação destes dados:

- Integer Position[2];
- Integer Father[2];
- Boolean Visited;
- String Element;

Ao ser realizada a leitura, para fins de controle, é setado a posição (position) do nodo/vértice igualmente conforme sua representação na Matriz, sendo Position[] um pequeno array de duas

posições, a posição [0]guarda o valor de i e [1] a posição j, sendo uma forma simplificada de mapear os dois valores juntos. Na leitura também, é armazenado o elemento lido para, mais frente, ser utilizado durante a busca.

Os atributos *visited* e *father* são atualizados após ocorrer a busca. Visited representa o rótulo (*label*) para controle se o nodo já foi visitado e *father* (pai) mantém um controle com a posição do seu buscador (uma cópia de position do pai que o acessou), assim, facilita após a execução a busca pelo caminho percorrido.

Não foram criados objetos que representassem as arrestas, de forma que mais a frente, o controle sobre estes acessos são feitos pelo código de busca.

Carregado todo arquivo de entrada e armazenadas as informações na classe Vertices, a classe BuscaCaminho recebe essa instância carregada com os dados. Possui um método construtor que recebe essa instancia e dispara toda busca.

Inicialmente, faz uma validação se as posições do caranguejo e saída possuem caminho possível. Essa validação é tirada a partir do módulo das posições. Por exemplo C= 6,6 e S= 2,2, o cálculo do modulo de 6 (em i e j) é 0, da mesma forma que 2 (em i e j) resulta em 0. Esse calculo representa o tabuleiro apresentado acima e a compatibilidade entre os saltos e caminhos do caranguejo.

```
private boolean validaPosicao(){
    if(((((ler.CaranguejoI))%2)==(((ler.SaidaI))%2)) && (((ler.CaranguejoJ%2))==((ler.SaidaJ%2)))) return true;
    else return false;
}
```

Figura 4 Fragmento de código - BuscaCaminho

Após essa validação, se os resultados entre a saída e caranguejo for valida (verdadeira) o programa executa a busca.

```
private void realizaBusca(){

    int iniciali = ler.CaranguejoI;
    int inicialj = ler.CaranguejoJ;
    int finali = ler.SaidaI;
    int finalj = ler.SaidaJ;
    fila.add(ler.MatrizList.get(iniciali).get(inicialj));
    while (fila.size()>0){
        Vertices u = fila.poll();
        int[] pos = u.getPosition();

        processa(u);
        u.setVisited();

        if(pos[0]==finali && pos[1]==finalj){
           //encontrou saida
            break;
        }
```

Figura 5 Fragmento de código - BuscaCaminho

Como o algoritmo de busca em largura necessita de um ponto inicial , passamos a posição da matriz que esta o caranguejo, esse é enfileirado, e ao entrar no laço *while*, é desenfileirado e processado. A etapa de processamento consiste em verificar suas arrestas.

```java
private void processa(Vertices u){
    int[] pos = u.getPosition();
    int i = pos[0];
    int j = pos[1];
    BFS( (i-1), (j-1), pos);//left-up
    BFS( (i-1), (j+1), pos);//right-up
    BFS( (i+1), (j+1), pos);//right-down
    BFS( (i+1), (j-1), pos);//left-down
    BFS( (i-2), (j), pos);//up
    BFS( (i),   (j+2), pos);//right
    BFS( (i+2), (j), pos);//down
    BFS( (i),   (j-2), pos);//left
```

Figura 6 Fragmento de código - BuscaCaminho

Processa é um método processa é um intermediário que direciona a que vértices serão acessados. Como dito anteriormente, não houve implementação de arrestas para ligação entre os vértices, sendo esse método o responsável.

Por fim, é feita a análise do vértice direcionado pelo método processa. Esse faz uma validação dos valores recebidos e do vértice a ser analisado para que o valores de posição não estourem as bordas (menor que 0 e maior que o tamanho do *array*), valida se o vértice já foi visitado, e se é possível que o mesmo seja um “caminho”, ou seja, não ocupando espaços com “x”. Caso atendidas as condições, o vértices tem seus atributos setados (como *father* e *visited*), e enfileirado para ser processado mais à frente.

Assim, antes tínhamos um fila F={u} agora teremos uma fila sem esse vértice, mas com os seus vizinhos aguardando seu processamento F={v1,v2,v3,v4,v5,v6,v7,v8}. Essa fila possui um crescimento inicial grande, aumentando em até 8x no primeiro ciclo, mas a medida do seu processamento, atinge um tamanho máximo e em seguida, é diminuído ligeiramente, vistos que os vértices processados, não geram novos enfileiramentos já que seus vizinhos poderão estar visitados. Ainda otimizando o algoritmo, caso o vértice analisado seja a posição de saída, o loop é parado, evitando o processamento de vértices na fila, utilizando processamento desnecessário.

4. RESULTADOS OBTIDOS:

Para melhor análise, o algoritmo substitui os caracteres do caminho de “.” por “|” melhorando a visualização do caminho percorrido.

Vejamos:

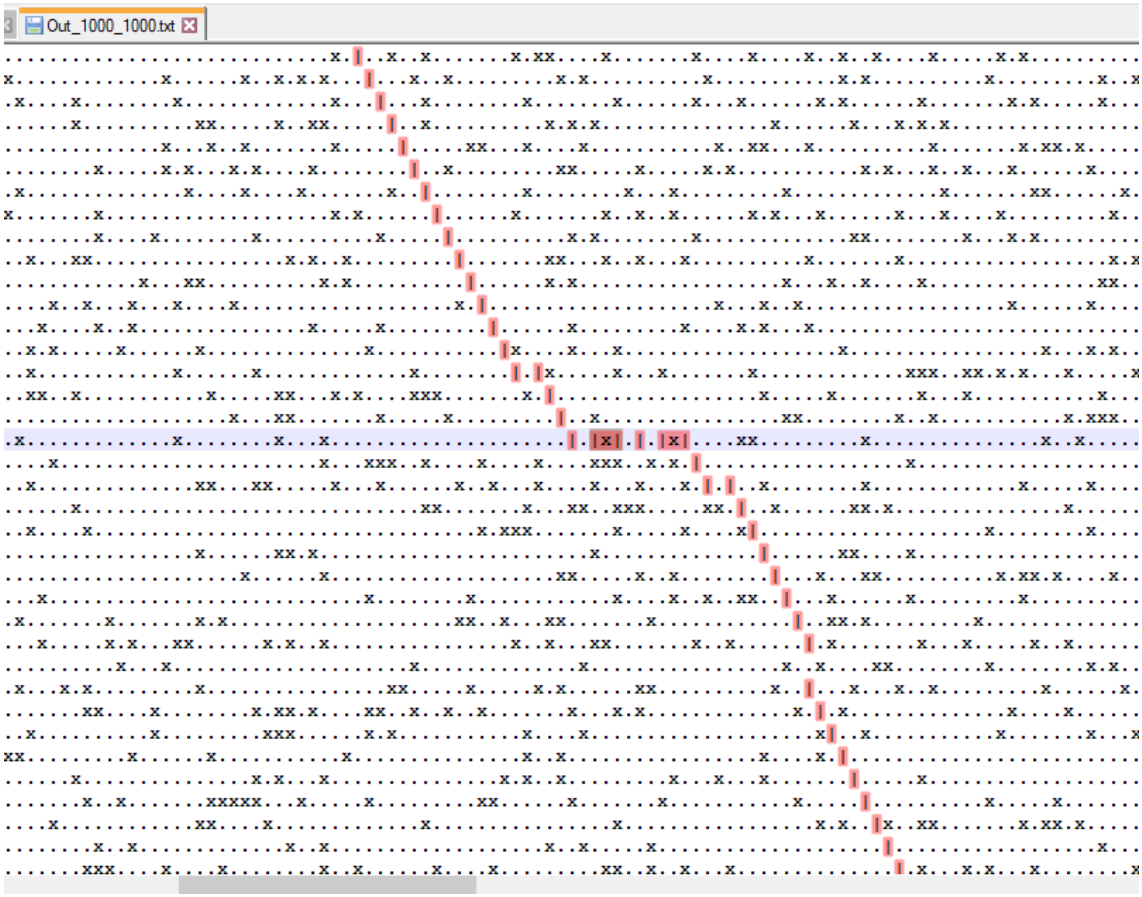

Figura 7 Arquivo de saída 1000_1000

O algoritmo é implementado de forma que consiga contornar possíveis travamentos pois, de fato, se um nodo que está buscando arrestas não tem saída, o mesmo “morre”. Assim, só restam os caminhos uteis para o caranguejo.

O repositório disponibilizado no github possui a saída dos arquivos de forma resumida. Apenas o arquivo 75_75_4 apresentou problemas de execução. Os demais, mesmo os que foram identificados que não possuíam caminhos, foram executaram. De forma sucinta foram obtidos os seguintes resultados:

Arquivo	Possibilidade de saída	Caminho total
50_50.txt	Sim	19
75_75.txt	Não	--
75_75_1.txt	Não	--
75_75_2.txt	Não	--
75_75_3.txt	Não	--
75_75_4.txt	-ERRO DE EXECUÇÃO-	--
75_75_5.txt	Sim	37
100_50.txt	Sim	25
100_100.txt	Não	--
250_250.txt	Não	--
640_480.txt	Sim	230
1000_1000.txt	Sim	315

Tabela 1 Resultados

CONCLUSÕES

Com o presente trabalho foi possível compreender de forma prática o uso dos algoritmos de caminhamento em grafos. Tanto nos utilizados, quanto naqueles descartados, que de certa forma, contribuíram no aprendizado e análise do problema pois não atenderiam a proposta solicitada.

O trabalho trouxe boa contextualização do problema e aplicação, desafiando além da implementação, a modelação desde a leitura do arquivo e a forma de tratar os dados obtidos e a ligação entre eles. Sem sombra de dúvida há maneiras otimizadas de implementação do algoritmo que não foram aplicadas, mas por fim, está foi a solução obtida com resultados.