Técnica Fenícia para Economia de Combustíveis Baseada em Algoritmos de Busca e Caminhamento  
  
Mateus F. Poletto\*  
Escola Politécnica – PUCRS

05 de junho de 2023  
  
 **Resumo**

Irá ser retratado no artigo a seguir a solução para um problema comercial que atingiu os fenícios conforme descrito pelo enunciado da atividade. O problema em si fala sobre como que é possível descobrir a maneira mais barata para se viajar até os parceiros comercias deles e retornar até em casa a salvo e de maneira mais econômica possível, de maneira a maximizar os lucros.  
 A viagem é feita com uma embarcação pelo mar e o mapa que eles foram guiados era formato por pontos (água = navegável), asteriscos (terra, etc = Não navegável), e números que representavam os portos. Podendo navegar apenas na vertical e horizontal, a solução adotada foi primeiramente antes de percorrer o mapa e decidir a rota mais barata foi eliminar todos os pontos do mapa onde eram impossíveis de serem alcançados por navegação marinha e também já definir quais portos seriam possíveis parceiros de negócios. Após isso foi executado o Algoritmo de Dijkstra com o mapa resultante da eliminação dos pontos inacessíveis, cada chamada do algoritmo recebia o porto de partida e o chegada, de maneira que ao final das execuções é encontrado o combustível mínimo necessário.

**Introdução**

Esse artigo irá retratar uma maneira eficiente e tecnológica de resolver um problema que a civilização fenícia sofria nos velhos tempos, como praticar comercio marítimo de maneira eficiente e econômica.

Sempre que uma embarcação fenícia gostaria de sair pelo mar para fazer negócios era disponibilizado um mapa da região a ser praticado o comércio no seguinte formato:

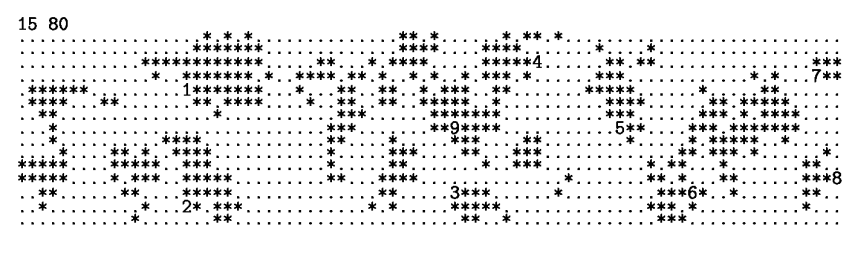
* Pontos “ . ” representavam água (navegável);
* Asteriscos “ \* ” representavam terra ou algo não navegável;
* Números de 1 a 9 representam portos, sendo o número 1 sempre o porto dos fenícios;

Porém, antes de sair pelo mar era necessário definir a rota a ser praticada e qual portos seriam definidos como possível parceiro de negócio para visitar. A rota precisava ser a mais econômica e seguir as seguintes regras:

* A movimentação é somente com movimentos para Norte, Sul, Leste ou Oeste;
* Deve se seguir um padrão de negócios em que os portos a seriam visitados devem ser atingidos de maneira crescente, ou seja, primeiro a ser visitado é o 2, depois 3, e assim por adiante;

Trazendo então esse problema para os dias atuais, a solução obtida foi ler o mapa em um programa de computador escrito por mim (Mateus), e dentro do mapa, apagar todos os pontos em que nunca conseguiríamos atingir por navegação utilizando o Algoritmo de Caminhamento por Largura. Após realizar essa passada no mapa então era jogado o novo mapa com somente áreas navegáveis para ser analisado pelo Algoritmo de Dijkstra, o qual nós retornava a distância mínima do primeiro porto até o segundo e assim por adiante. No final, a soma dos resultados nos retorna o combustível mínimo para o trajeto.

Exemplo de Mapa a ser analisado:

**Linguagem De Programação Escolhida:**

* Java 17.0.6 2023-01-17 LTS;

**Desenvolvimento**

Após uma análise minuciosa do problema, percebi que até ao realizar a leitura do mapa para o programa já era necessário começar com uma simplificação do mapa. Para facilitar as próximas etapas que viriam pela frente do problema foi resolvido remover todos os asteriscos do mapa, fazendo em que o array bidimensional aonde estavam sendo salvos os nodos (posições do mapa) por posição de linha (x) e coluna (y) ficasse vazio (null) naquela posição específica.

Além disso, foram salvos em cada posição da matriz um objeto que representa um ponto do mapa. Cada objeto contém um “boolean” (visitado) que será utilizado principalmente para o Algoritmo de Caminhamento em Largura, um “inteiro” (numeroPorto) que contém o número do porto caso seja um, se não for contém 0, uma referência as seus nodos vizinhos ao norte, sul, leste e oeste se tiver, sua localização na matriz (x e y) e mais um “inteiro” (combustível) para armazenar o valor de combustível que foi necessário para chegar até aquele nodo.

Ao finalizar a leitura devolvemos o array bidimensional com as referências para os nodos (posições do mapa), uma referência para o nodo que possui a terra dos fenícios (ponto de partida e chegada) e a quantidade de linhas e colunas que o mapa original tinha.

**Pseudocódigo**

função leituraArquivo(nomeDoArquivo):

mapa é uma lista vazia

terraDosFenicios é null

ajudaParaLerArquivo abre nomeDoArquivo

primeiraLinha é ajudaParaLerArquivo.lerLinha()

linhas, colunas é primeiraLinha.separadoPorEspaço()

mapa é nova lista com 'linhas' de altura e 'colunas' de largura

para cada linha em ajudaParaLerArquivo:

para cada coluna em linha:

se coluna é '.' então:

colocar nova localização vazia em mapa na posição [linhaAtual, colunaAtual]

senão se coluna é '\*' então:

continuar para o próximo

senão:

numeroPorto é o número que está na coluna

se numeroPorto é 1 então:

colocar a terra dos Fenícios em mapa na posição [linhaAtual, colunaAtual]

terraDosFenicios é esta nova localização

senão:

colocar novo porto com número 'numeroPorto' em mapa na posição [linhaAtual, colunaAtual]

linhaAtual aumenta em 1

se algo deu errado:

mostrar o erro

retornar resultadoLeitura(linhas, colunas, mapa, terraDosFenicios)

**Problema 1:**

Entretanto um problema ainda estava deixando a solução para o problema distante, como fazer para que nosso Algoritmo de Dijkstra não demore muito para rodar quando existir portos em que não podem ser alcançados por navegação?

Essa pergunta perseguiu o resultado final por muito tempo até ser definido que mesmo possuindo o mapa em mãos já simplificado, precisaríamos simplificar mais ainda ele removendo todos os nodos não alcançados na cadeia geral de nodos e tendo em mãos todos os portos possíveis para se fazer negócios. Tudo isso era necessário para rodarmos depois com absoluta certeza que somente aqueles portos e aqueles nodos resultantes seriam possíveis de ser alcançados com o Algoritmo de Dijkstra.

Portanto, essa simplificação foi feita utilizando o Algoritmo de Caminhamento por Largura, em que todos os nodos que conseguíamos atingir foram marcados o “booleano” (visitado) como verdadeiro e sempre que encontrado um porto seu número era salvo na lista de Portos a serem visitados. Após essa primeira parte, foi iterado na matriz fazendo com que todos os nodos em que não tinham sido marcados com verdadeiro em seu “booleano” fossem eliminados da matriz.

**Pseudocódigo**

função fazCaminhamentoDeLarguraDoMapa OU exploreOMapa():

cria uma lista de direções para cima, para baixo, para a esquerda e para a direita

faz todos os lugares no mapa como 'não visitados'

cria uma linha de espera para os lugares que queremos visitar

começa pela terra dos Fenícios e dizemos que já visitamos ela

adiciona a terra dos Fenícios na nossa linha de espera

enquanto ainda tem lugares na nossa linha de espera:

vamos ao primeiro lugar na linha de espera e o tiramos da linha

para cada direção em nossa lista de direções:

olhamos para o lugar naquela direção

se esse lugar é parte do nosso mapa e ainda não o visitamos:

dizemos que já visitamos esse lugar

adiciona esse lugar na nossa linha de espera

se esse lugar é um porto:

adiciona esse porto na nossa lista de portos para negociar

lembra que esse lugar é o vizinho do nosso lugar atual na direção que olhamos

depois de visitar todos os lugares que conseguimos:

para cada lugar no nosso mapa:

se não visitamos esse lugar:

remove ele do nosso mapa

dizemos que a terra dos Fenícios tem 0 de combustível

**Problema 2:**

O segundo e último problema enfrentado foi referente a adaptação do Algoritmo de Dijkstra para a situação comercial dos fenícios que estamos trabalhando. De início começamos definindo todos os locais existentes no mapa com o valor Inteiro máximo disponível na linguagem de programação em seus combustíveis, o único a não ser marcado foi o ponto de partida (porto dos fenícios). Além disso, para chamadas futuras, o porto de partida da chamada é setado com valor de combustível igual a 0.

Após isso criamos uma fila de prioridade em que o nodo com menor número de combustível e sempre o preferido a sair chamado na lista. Então adicionamos o nodo atual em que partimos a essa fila. Com a fila criada e com um valor já adicionado vamos criar um laço de repetição em que até que a fila de prioridade não acabe ou o laço seja quebrado vamos fazer o seguinte:

* Definimos as 4 posições que podem ser possíveis de aquele nodo ter vizinhos;
* Para cada vizinho fazemos o seguinte tratamento;
  + Se a posição “ x “ do vizinho for maior ou igual a zero e menor que a quantidade total de linhas do mapa, e, a posição “ y “ do vizinho for maior ou igual a zero e for menor que a quantidade de colunas, então deixamos prosseguir;
  + Então buscamos esse vizinho, se esse vizinho existir então continuamos a execução;
  + É criado então uma variável auxiliar de combustível que guarda o valor do nodo atual que estamos + 1 que será o combustível que guardaremos no vizinho;
  + Se o combustível da variável auxiliar for menor que o combustível que o vizinho já possui setado, então continuamos a execução;
  + Setamos então o novo valor de combustível para o vizinho;
  + Então verificamos se esse vizinho não é o porto que estamos buscando chegar no momento, se não for, apenas adicionamos o vizinho a lista de prioridade, porém se for, seguimos o seguinte:
    - Se a nossa lista com portos para negociar ainda possuir valores, chamamos novamente nosso Algoritmo de Dijkstra modificado passando o vizinho (que é o porto atual que chegamos) e o próximo porto na nossa lista para negociar;
    - Se já visitamos todos os portos em que poderíamos negociar então fazemos nossa última chamada do nosso algoritmo modificado passando o vizinho (que é o porto atual que chegamos) e o número do porto dos fenícios;
      * Ao acabar qualquer uma das chamadas anteriores é quebrado o ciclo daquela execução;
* No final, é somado os valores dos combustíveis que foram salvos em cada porto e retornado o resultado que é o combustível mínimo para a viagem;

**Pseudocódigo**

função visiteOsParceirosDeNegocio(portoAtual, próximoPorto):

para cada lugar no mapa:

se o lugar não é o porto atual:

coloque muito combustível nesse lugar

coloque zero de combustível no porto atual

crie uma linha de espera para os lugares que queremos visitar, ordenando do lugar com menos combustível para o lugar com mais combustível

adicione o porto atual à linha de espera

enquanto ainda tem lugares na linha de espera:

vá para o lugar com menos combustível na linha de espera e o tire da linha

para cada vizinho do lugar que estamos visitando:

se o vizinho é parte do nosso mapa:

novaQuantidadeDeCombustível é a quantidade de combustível do lugar que estamos visitando +1

se a novaQuantidadeDeCombustível é menor que a quantidade de combustível do vizinho:

coloque a novaQuantidadeDeCombustível no vizinho

se o vizinho é o próximo porto:

adicione a novaQuantidadeDeCombustível na nossa lista de gasolina por porto

se ainda tem portos para visitar:

visite o próximo porto na nossa lista de portos para visitar

senão:

visite a terra dos Fenícios

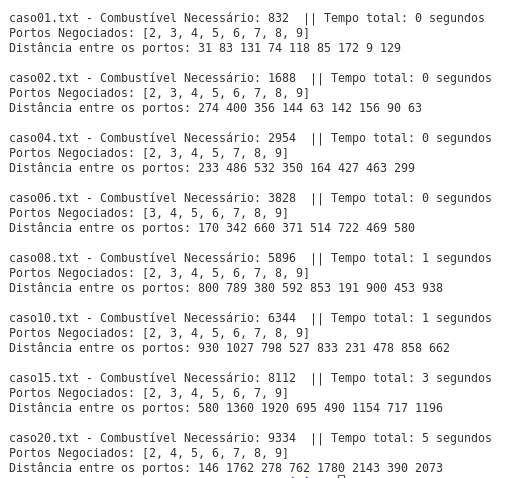
pare de olhar os outros vizinhos

senão:

adicione o vizinho na nossa linha de espera

**Resultados:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CASO | COMBUSTÍVEL NECESSÁRIO | TEMPO DE EXECUÇÃO (em segundos ) | PORTOS NEGOCIADOS | DISTANCIA ENTRE OS PORTOS (na sequência) |
| Caso01.txt | 832 | T < 0 | [2, 3, 4, 5, 6, 7, 8, 9] | 31 83 131 74 118 85 172 9 129 |
| Caso02.txt | 1688 | T < 0 | [2, 3, 4, 5, 6, 7, 8, 9] | 274 400 356 144 63 142 156 90 63 |
| Caso04.txt | 2954 | T < 0 | [2, 3, 4, 5, 7, 8, 9] | 233 486 532 350 164 427 463 299 |
| Caso06.txt | 3828 | T < 0 | [3, 4, 5, 6, 7, 8, 9] | 170 342 660 371 514 722 469 580 |
| Caso08.txt | 5896 | 1 | [2, 3, 4, 5, 6, 7, 8, 9] | 800 789 380 592 853 191 900 453 938 |
| Caso10.txt | 6344 | 1 | [2, 3, 4, 5, 6, 7, 8, 9] | 930 1027 798 527 833 231 478 858 662 |
| Caso15.txt | 8112 | 3 | [2, 3, 4, 5, 6, 7, 9] | 580 1360 1920 695 490 1154 717 1196 |
| Caso20.txt | 9334 | 5 | [2, 4, 5, 6, 7, 8, 9] | 146 1762 278 762 1780 2143 390 2073 |



**Conclusão:** Mesmo com um algoritmo que em partes possa parecer desnecessário, as mesmas são muito importantes pois ajudaram na visualização da lógica e os próximos passos a serem tomados. Além de as mesmas poderem em um momento futuro fazerem parte de uma adição de funcionalidade extra na aplicação.

Entre todos os desafios para completar esse programa julgo que o mais complexo foi decidir como acabar com o problema de quando temos um porto que não podemos negociar. Estava muito pensativo em relação a isso e quase cheguei a perguntar possíveis soluções á colegas, porém ao estar discutindo o trabalho com um amigo, acabei chegando na solução para acabar com esse problema, mesmo que o mesmo não me passou dicas, agradeço o apoio “moral” a meus pensamentos meu caro Ricardo.

Em minha experiência, julgo que o programa chegou perto da perfeição, existe alguns detalhes que poderiam ser melhor estudados e modificados no futuro para diminuir lentamente a complexidade assintótica do programa e acabar diminuindo seu custo, porém estou muito contente com o resultado obtido com o que foi desenvolvido.

**Referências:**<https://moodle.pucrs.br/pluginfile.php/4310926/mod_resource/content/1/artdemo.pdf>  
<https://www.youtube.com/watch?v=09_LlHjoEiY>  
[https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm](https://en.wikipedia.org/wiki/Dijkstra's_algorithm)  
<https://pt.wikipedia.org/wiki/Busca_em_largura>  
Algoritmo implementado em Java encontrado no perfil do Github: https://github.com/mati-fp