Trabalho Prático 1
Algoritmos I

Luís Felipe Ramos Ferreira

2019022553

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

lframos_ferreira@outlook.com

1. Introdução

A proposta do trabalho prático 1, da disciplina de Algoritmos, foi implementar um algoritmo que, ao receber as coordenadas de um mapa, as disposições de bicicletas, visitantes e obstáculos nele, além de uma lista de preferências de visitantes por bicicletas, fosse realizado um cálculo de casamento/pareamento e, dessa forma, alocar cada bicicleta a um visitante e vice-versa, respeitando a seguinte condição de estabilidade:

Se uma pessoa p1 foi alocada a uma bicicleta b1,então:

- Se há uma bicicleta b2 que p1 considere preferível em relação a b1, então há uma pessoa p2 mais próxima de b2 para a qual b2 foi alocada;

- Se há uma pessoa p2 que está mais próxima de b1 do que p1, então p2 foi alocada para alguma bicicleta b2 que ela considera preferível a b1.

A modelagem computacional, assim como a implementação deste problema, serão discutidos ao longo desta documentação.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection, no sistema operacional Windows.

2.1. Modelagem computacional

O problema dado pelo enunciado do trabalho prático pode ser separado em duas partes, de forma a garantir seu entendimento e simplificar sua implementação.

Em primeiro lugar, é necessário obter as listas de preferências de cada uma das entidades envolvidas no sistema. A lista com os ranqueamentos dos visitantes para cada bicicleta já é dada pela entrada do programa, portanto podemos nos preocupar com isso posteriormente. A lista de "ranqueamentos" das bicicletas, por sua vez, deve ser obtida analisando o mapa da lagoa.

Para isso, é necessário calcular as distâncias entre cada bicicleta e cada um dos visitantes, construindo assim os ranqueamentos de cada bicicleta para cada visitante. A maneira escolhida para resolver isso foi de interpretar o mapa da lagoa como um grafo, onde cada coordenada (i, j) da matriz é um vértice, o qual está ligado, por arestas, aos vértices nas coordenadas (i - 1, j), (i + 1, j), (i, j - 1) e (i, j + 1), se, e somente se, as

coordenadas desses vértices são válidas no mapa (isto é, estão dentro dos limites do mapa e não são obstáculos). Para melhor compreensão dessa análise do mapa, vejamos o seguinte exemplo apresentado no enunciado:

- Mapa de entrada da lagoa, de tamanho 4x5:

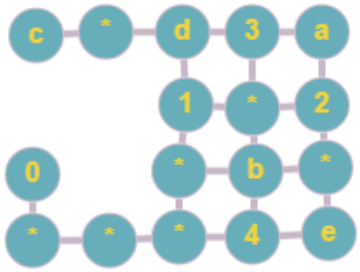
  c*d3a

  --1*2

  0-*b*

  ***4e

- Grafo do mapa, de acordo com a interpretação explicada:



  Note que o vértice (0, 0), representado pela letra c, não está conectado a nenhum vértice a cima, a baixo ou a esquerda dele, pois estas são posições inválidas no mapa. A sua direita, por sua vez, é a coordenada (0, 1), que está dentro dos limites do mapa e não é um obstáculo. Portanto, existe uma aresta conectando ambos vértices.

Essa interpretação permite utilizar algoritmos de grafos para cálculo de menor distância entre dois vértices. O algoritmo escolhido, que será detalhado no próximo tópico, é o algoritmo de busca em largura que, em grafos não ponderados, pode ser utilizado para encontrar o menor caminho do vértice S ao vértice T. Utilizando este método, é possível calcular as menores distâncias entre cada bicicleta e cada um dos visitantes e, assim, gerar os ranqueamentos desejados.

Após ter acesso a ambas listas de ranqueamentos, é necessário obter as listas de preferências, a partir delas. Ordenar estas listas, de acordo com as especificações de preferência de cada entidade, já permite construir as matrizes de preferência de ambas. No caso, os visitantes atribuíram uma nota às bicicletas e quanto maior a nota, maior sua preferência por ela. As bicicletas, por sua vez, possuem preferência por visitantes que estão a distâncias menores dela. Em ambas as situações, o empate é resolvido por meio da preferência pelo menor ID da entidade.

A segunda parte da resolução do problema envolve realizar o casamento estável entre bicicletas e visitantes, de acordo com as matrizes de preferências obtidas anteriormente. A melhor escolha para lidar com esse tipo de situação é o algoritmo de Gale-Shapley, que será abordado com mais detalhes no próximo tópico. Por meio desse algoritmo, é possível encontrar o emparelhamento estável de bicicletas e visitantes para a entrada do programa.

2.2. Estruturas de Dados e algoritmos

Assim como na descrição da modelagem computacional do problema, para abordar as estruturas de dados e algoritmos utilizados, vamos separar ele em duas partes.

No entanto, para que a explicação fique mais clara, é importante citar que, embora na teoria os visitantes sejam representados por letras, valores inteiros que

representam as letras foram usados para o armazenamento dos valores referentes a eles. A letra ‘a’, por exemplo, foi representada pelo número 0, a letra ‘b’ pelo número 1, e etc. Outra constatação importante é de que os pares de coordenadas utilizados para navegar pelo mapa/grafo foram usados como a estrutura par *(pair*) da biblioteca padrão da linguagem C++.

A primeira parte do trabalho se refere à encontrar as listas de ranqueamento e preferências de cada entidade envolvida no problema. As listas de cada entidade foram armazenadas em um vetor (*vector*), da biblioteca padrão, que contém inteiros. As matrizes de ranqueamento e preferências, por sua vez, são vetores de vetores, que irão armazenar as listas já descritas. Essa abordagem foi utilizada pois as estruturas de dados da biblioteca padrão possuem poderosas funcionalidades, que facilitariam a implementação do trabalho.

Como especificado, o mapa da lagoa foi interpretado como um grafo, e o algoritmo de busca em largura (BFS ou *breadth first search*) foi utilizado para encontrar o menor caminho entre os vértices desejados. Segue abaixo o pseudocódigo do algoritmo de BFS, retirado do livro-texto da disciplina.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

A implementação desse algoritmo foi feita de maneira iterativa, utilizando a estrutura de dados de uma Fila (*queue*), disponibilizada na biblioteca padrão, assim como uma matriz declarada de forma estática, a qual armazena quais posições da matriz já foram visitadas, junto com a distância que tais posições estão da coordenada inicial.

Em primeiro lugar, para cada uma das bicicletas, as coordenadas de sua posição são o vértice de início da busca em largura. Essa primeira coordenada é então adicionada à fila de vértices assim como na matriz de vértices já visitados,onde possui valor 0, uma vez que está a uma distância nula de si mesma. Dado estas especificações, um laço *while* começa a ser percorrido. Enquanto houver um elemento na fila, o elemento do topo da fila é pego e são testados todos os movimentos que podem ser feitos a partir dele. Se o movimento resultar em um movimento inválido, ele é ignorado. Caso contrário, duas coisas podem ocorrer:

- A posição é ocupada por um espaço vazio/outra bicicleta, que pode ser percorrido, e então esse espaço é adicionado à fila.
- A posição é ocupada por um caractere, checado pela função isalpha, da biblioteca *cctype*, que representa um visitante. Então a distância da pessoa ao vértice de origem é pega e adicionada ao vetor de distâncias entre visitante - bicicleta, e o espaço é adicionado à fila.

Em ambos os casos, ao ser lido como uma posição válida, o vértice é marcado como visitado e sua distância ao vértice de origem é computada como a distância do vértice anterior somado a um.

Quando todos os movimentos possíveis de serem feitos a partir de um vértice são checados, esse vértice, que está no topo da fila, é retirado dela. Assim, garantimos que vamos checar todos os vértices do grafo, isto é, todos os espaços possíveis de alcançar a partir da origem.

Essa abordagem é feita para cada uma das coordenadas que contêm bicicletas e,

dessa maneira, adicionaremos a lista de distâncias de cada bicicleta à matriz auxiliar. Ao fim de todo esse processo, como já temos a matriz de ranqueamentos dos visitantes pela entrada e calculamos a das bicicletas, precisamos manipular essas duas estruturas e gerar as matrizes de preferências para cada entidade.

No entanto, ao analisar as especificações do trabalho e o algoritmo de Gale Shapley, é possível notar que a matriz de preferências das bicicletas não é necessária para o cálculo do casamento. A matriz com as distâncias de cada bicicleta para cada visitante são suficientes para tal. No entanto, a matriz de preferências foi calculada mesmo assim, no intuito de facilitar os testes de unidade.

A forma encontrada de gerar essas matrizes foi ordenar as listas de ranqueamentos de cada entidade, de acordo com as especificações do enunciado. No entanto, a ordenação mudaria os valores de ranqueamento de posição, mas não armazenaria os índices anteriores desses valores, os quais representam o ID de cada entidade. Para contornar essa situação, foi criado um vetor auxiliar na função de ordenação, o qual armazena um par (valor, índice pré-ordenação). Isso garante que, ao terminar a ordenação, teremos acesso aos IDs de cada entidade na posição correta nas listas de preferências.

Para ordenar os ranqueamentos, foi utilizada a função *stable_sort*, da biblioteca *algorithm*, pois ela garante uma ordenação eficiente e estável. Para cada entidade, foi criada uma função de comparação auxiliar, que é passada como parâmetro para a função de ordenação. Isso foi feito uma vez que cada entidade possui critérios diferentes de prioridade. Assim, cada uma das matrizes de ranqueamentos passa por esse processo, que irá retornar a matriz de preferências desejada para os cálculos restantes do trabalho.

A próxima etapa é, como já abordado, realizar um emparelhamento estável entre as duas entidades do trabalho. Para isso, foi utilizado o algoritmo de casamento de Gale-Shapley, abordado nas aulas, uma vez que ele garante sempre um emparelhamento estável. Segue abaixo o pseudocódigo do algoritmo, disponibilizado no livro texto da disciplina.

```
Initially all m ∈ M and w ∈ W are free
While there is a man m who is free and hasn't proposed to
every woman
   Choose such a man m
   Let w be the highest-ranked woman in m's preference list
      to whom m has not yet proposed
   If w is free then
      (m, w) become engaged
   Else w is currently engaged to m'
      If w prefers m' to m then
         m remains free
      Else w prefers m to m'
         (m, w) become engaged
         m' becomes free
      Endif
   Endif
Endwhile
Return the set S of engaged pairs
```

A implementação do algoritmo de Gale-Shapley foi feita seguindo o que o pseudocódigo e a lógica do algoritmo dizem. Para checar quais visitantes ainda não tinham sido emparelhados, usou-se a estrutura de dados pilha (s*tack*). Para armazenar as alocações de pares enquanto o algoritmo funcionava, dois vetores, um para bicicletas e outro para visitantes, armazenavam os IDs dos pares de cada entidade no índice

referente à própria entidade. Por fim, para checar quantas propostas já haviam sido feitas por cada visitante, a fim de garantir que a próxima proposta do visitante seja para a sua próxima prioridade, foi utilizado também um vetor.

Uma alteração específica foi feita para otimizar a função. Na segunda condição do loop principal, é checado se a bicicleta b1, que recebe a proposta do visitante v2, prefere v2 ao seu par atual, v1. Para garantir que essa checagem fosse feita em tempo constante, ao invés de passar a matriz de preferências das bicicletas como parâmetro da função, a matriz que contém as distâncias entre bicicletas e visitantes foi utilizada, visto que seu acesso já estava viabilizado desde as primeiras etapas do trabalho.

Ao fim da execução do algoritmo de Gale-Shapley, temos acesso ao emparelhamento estável desejado e podemos imprimir na saída o resultado. De forma a imprimir os símbolos corretos para os visitantes, os IDs com valor inteiro que os representavam foram transformados nos caracteres desejados durante a impressão dos resultados.

3. Análise de Complexidade de Tempo

Função 'moveBike': essa função realiza apenas operações constantes, portanto sua complexidade de tempo é constante.

Função 'outOfBounds': essa função realiza apenas operações constantes, portanto sua complexidade de tempo é constante.

Função 'getDistancesBikeToVisitors': essa função implementa o algoritmo de busca em largura. Sua entrada é quadrática, baseada no tamanho do mapa. No entanto, em um pior caso em que não existem obstáculos, o grafo gerado pelo mapa (de acordo com a interpretação apresentado anteriormente) irá possuir MN vértices, onde M é o número de linhas e N é o número de colunas no mapa. Dado essa constatação, podemos entender que o número de vértices será dado por este produto, sempre, e podemos chamá-lo de V. O algoritmo, da forma que é implementado, checa, para cada vértice, 4 possíveis movimentos. Portanto, o número de comparações/checagens feitas é igual a 4V. Como 4 é uma constante, ao analisarmos a complexidade, podemos dizer que o algoritmo possui uma complexidade de tempo linear para o número de vértices do grafo/mapa, ou seja, pertence a O(V).

Função 'getDistancesBikesToVisitors': essa função basicamente chama a função anterior para cada uma das N bicicletas, portanto, sua complexidade de tempo é O(NV).

Função 'getPreferenceMatrix': essa função percorre um laço para cada N listas de ranqueamentos (onde N é o número de visitantes/bicicletas) em uma matriz. Para cada uma dessas listas, é utilizado o algoritmo de ordenação *stable_sort*, da biblioteca *algorithm* que, segundo sua documentação, possui complexidade de tempo O(Nlog(N)) para o tamanho da lista de ranqueamentos (que nesse caso também é N). Portanto, sua complexidade de tempo final $O(N^2 log(N))$. No entanto, é importante ressaltar que, segundo o enunciado do trabalho, o número N sempre será menor ou igual a 10. Nesse sentido, pode-se fazer uma interpretação de que esse valor é constante, resultando numa complexidade de tempo constante para a função.

Função 'bikePrefersVisitorToCurrentMatch': essa função realiza apenas operações constantes, portanto sua complexidade de tempo é constante.

Função 'GaleShapley': o algoritmo, como explicitado anteriormente, roda de forma que cada visitante irá fazer propostas pelas bicicletas e, conforme as especificações, os pares serão formados. No pior caso, cada visitante terá que propor para todas as bicicletas. Em outras palavras, toda a matriz de preferências dos visitantes terá que ser percorrida. Dessa maneira, a complexidade de tempo dessa função é $O(N^2)$.

4. Compilação e execução

- Acesse o diretório TP
- No terminal, utilize os seguintes comandos, para gerar os arquivos *object* da implementação no diretório ‘obj’ e, então, gerar o arquivo executável nomeado ‘tp01’ na pasta ‘bin’.

➢ g++ -std=c++17 -Wall -c -Iinclude -o obj/PreferenceList.o src/PreferenceList.cpp
➢ g++ -std=c++17 -Wall -c -Iinclude -o obj/GaleShapley.o src/GaleShapley.cpp
➢ g++ -std=c++17 -Wall -c -Iinclude -o obj/main.o src/main.cpp
➢ g++ -o bin/tp01 obj/main.o obj/PreferenceList.o obj/GaleShapley.o -lm

- Para executar o programa, digite no terminal ‘bin/tp01’ e, em seguida, passe os valores de entrada especificados pelo trabalho.