TP 01 - Trabalho Prático 1
Algoritmos I

Igor Lacerda Faria da Silva

igorlfs@ufmg.br

1 Introdução

O TP foi desenvolvido em C++, em ambiente Linux (versão do *kernel:* 5.17.4), utilizando algumas estruturas de dados (como o `std::vector`) e algoritmos (`std::stable_sort()`, etc) da biblioteca padrão. Em termos de boas práticas (formatação, *case* de variáveis e afins), foram as usadas as ferramentas `clang-format` e `clang-tidy` para garantir um código limpo e consistente. Foram escritos testes de unidade (uma lista não extensiva) para as principais funções do código, usando a biblioteca `<gtest>`. Os testes podem ser compilados e executados com o comando `make test` (assumindo-se que a biblioteca esteja instalada no sistema).

2 Modelagem Computational do Problema

Em suma, o programa transforma a entrada em duas listas de preferências e faz um casamento estável a partir delas. Em um nível a baixo, isso é realizado em 6 etapas: leitura dos parâmetros (número de visitantes, de linhas e de colunas), que é trivial; leitura do mapa; cálculo de distâncias das bicicletas aos visitantes, com a criação de uma "lista de preferências" das bicicletas para os visitantes; leitura de "pontuações" e criação da lista de preferências dos visitantes; execução do casamento e impressão dos resultados. Estas etapas serão descritas em mais detalhes a seguir.

Seguindo a especificação, a "minha equipe" facilitou consideravelmente a modelagem do mapa da orla, ao transformar uma entidade física em uma matriz de caracteres. Desse forma, não foi necessário ter as ideias mais extraordinárias para lidar com esse aspecto do modelo. De qualquer forma, foi tomado o cuidado de se salvar as posições das bicicletas, que serão úteis na etapa seguinte da execução, e, a título de corretude, conferir se o parâmetro de visitantes de fato corresponde à quantidade presente no mapa.

A priori, a primeira abordagem para se calcular as distâncias das bicicletas para os visitantes, considerando a estrutura "matriz de caracteres", seria simplesmente fazer a soma das diferenças dos módulos das coordenadas na matriz.

Por exemplo, se uma bicicleta 0 está na posição (0,0) e um visitante a está na posição (1,2), pode-se facilmente obter $d_0(a) = |0-1|+|0-2| = 1+2 = 3$. Mas é claro que isso não funciona, porque existem obstáculos. *Coincidentemente*, a busca em largura (BFS) foi estudada e ela é perfeita para essa aplicação; não é nem necessário construir um grafo, uma vez que o deslocamento na matriz é dado por uma unidade na vertical ou na horizontal, o que facilita o uso da própria matriz de caracteres da etapa anterior, dado que as posições das bicicletas são conhecidas. Considere um exemplo de execução, com o seguinte mapa:

$$\begin{pmatrix} 0 & - & a & * \\ * & - & * & * \\ * & * & * & * \\ * & * & - & - \\ * & * & - & b \end{pmatrix} \tag{1}$$

Aplicando a abordagem ingênua, a distância de 0 a a seria 2. Mas começando uma BFS em 0, com as camadas indicadas por um degradê em verde, obtém-se que a distância de 0 a a é na verdade 6. O interessante desse exemplo, além de ilustrar como a BFS funciona em um *lattice*, é o visitante b: não existe um caminho de 0 para esse visitante. Isso também foi levado em consideração na implementação: se não existe um caminho de uma bicicleta para um visitante, a entrada é considerada errônea e o programa é abortado (é claro que, sendo um TP, não se espera que esse tipo de erro aconteça).

Com as distâncias em mãos (ou, similarmente, com os *rankings* de cada visitante para cada bicicleta), é necessário gerar uma lista de preferências. No caso das distâncias, quanto menor a distância, mais a *bike* "gosta" do visitante em questão, enquanto que no *ranking*, uma *pontuação* maior é melhor. De qualquer forma, em essência, é o mesmo problema: a partir de uma matriz de pontuações, criar uma lista de preferências que obedece o critério de ordenação.

Assim, a linha que corresponde às distâncias de uma bicicleta é ordenada e é feita uma correspondência a partir dos índices na linha desordenada. Suponha que no mapa existam 5 bicicletas e 5 visitantes, sendo que as distâncias da bicicleta 0 em relação aos visitantes seja (5,2,1,5,3). Considerando essa quíntupla ordenada como um vetor e ordenando-o: (1,2,3,5,5), que correspondendo aos índices das posições originais resulta na seguinte "lista de preferências": [1] (2,1,4,0,3). Repetindo esse processo para todas as bicicletas e visitantes, as listas de preferências estão criadas e basta fazer o casamento estável.

O casamento estável é feito com o algoritmo de Gale-Shapley (GS), em que os visitantes propõem às bicicletas (ou seja, ele é ótimo para os visitantes), vide *thread* no fórum. O GS produz um casamento tal que nenhuma bicicleta b_1 que foi alocada a um visitante v_1 vai preferir um visitante v_2 (alocado a uma bicicleta b_2) e esse visitante simultaneamente também irá preferir b_1 a b_2.

[1] É claro que um objeto não tem preferências, mas as distâncias se comportam como critério de preferências nesse problema.

3 Estruturas de Dados e Algoritmos

Como comentado na seção anterior, as principais estruturas de dados usadas foram matrizes (seja de inteiros ou caracteres) e vetores, nas diversas etapas de execução do programa, quando não é menos que natural usá-las. Além disso, foi usado um mapa para guardar as posições dos dígitos na matriz de entrada, embora essa decisão tenha sido arbitrária: na maioria das situações, foi suposto um certo *encoding* do significado de letras como posições de vetores, ou seja, foram criados vetores em que a posição 0 corresponde à letra a, a posição 1 à letra b e assim por diante. Nessas instâncias foram usados *typecasts* para obter o comportamento desejado.

Adicionalmente, filas foram usadas em duas instâncias na implementação: na BFS e no GS. Na BFS a fila é usada para garantir que os nós só vão ser visitados quando os nós da camada anterior também já tiverem sido. No GS, a fila é usada para evitar comportamento cúbico ao tentar encontrar um proponente. Em nível mais baixo, o código é dividido em três módulos: a classe `Map` que é responsável pela leitura e outras operações que envolvem o mapa da Orla (como a BFS), a classe `Rank` que é *estática* e agrupa alguns métodos de ordenação e classificação e uma classe `Input`, também estátíca, que agrupa os métodos de leitura e validação da entrada.

A seguir, é descrita a implementação dos algoritmos em pseudo-código:

Algorithm 1 Retorna um casamento estável dadas duas listas de preferência

```
1:  function GaleShapley(ListaPrefHomens, ListaPrefMulheres)
2:      HomensLivres ← Insere todos os homens de ListaPrefHomens
3:      ProxMulher ← Primeira
4:      AtualMarido ← Nenhum
5:      Ranking ← Posições de cada homem com base em ListaPrefMulheres
6:      while HomensLivres != vazio do
7:          Homem ← frente de HomensLivres
8:          Mulher ← próxima mulher livre de Homem
9:          if AtualMarido[Mulher] = Nenhum then
10:             AtualMarido[Mulher] = Homem
11:         else if Mulher prefere Homem em relação ao seu AtualMarido then
12:             AtualMarido[Mulher] = Homem
13:     return AtualMarido
```

Alguns cuidados adicionais são tomados na implementação, como o controle da fila de homens livres e o acompanhamento da próxima "mulher" de todo "homem". Foi usada a analogia de homens e mulheres porque ela é muito canônica. As variáveis definidas antes do laço principal servem para limitar a complexidade da função a $\Theta(n^2)$.

Algorithm 2 Retorna um vetor distância da posição (m,n) a todas os visitantes

```
1: function BFS(m,n)
2:     dist ← inválidos
3:     visitados ← (m,n)
4:     fila ← (m,n)
5:     while fila != vazio do
6:         visitado ← primeiro da fila
7:         if Mapa[m,n] == letra then
8:             atualiza distância para aquela letra
9:         visita vizinhos de visitado se forem válidos, os colocando na fila
10:    return dist
```

O único "truque" na BFS é usar um membro adicional que "toma conta" da distância. Para isso, foi criada uma estrutura cuja única finalidade é essa. Como comentado anteriormente, é checado se o mapa é correto a partir desse método, ou seja, espera-se que nenhuma distância seja inválida no final da execução e é aqui onde aparece um dos *typecasts* (na "atualização" da distância).

4 Análise de Complexidade de Tempo

Até a leitura do mapa, está é a etapa mais cara em termos de complexidade de tempo. Considerando um mapa de M linhas e N colunas, o teto (e piso) na execução do programa até essa etapa é $\Theta(MN)$.

Como o número de visitantes V não pode ser maior que 10 pela especificação (e porque não é possível ler um caractere '10'), o restante dessa análise parte do pressuposto que **o número de visitantes é constante**. Ou seja, se uma função depender **somente do número de visitantes, seu tempo será considerado constante** (pior caso $n = 10$). Esse é o caso do Gale-Shapley, que é quadrático no número de visitantes e a função `setPreferenceList()` que seria $n^2 \lg n$ não fosse a limitação de visitantes (uma vez que ela "ordena" n vetores de tamanho n). A função `readScores()` também entra nessa categoria.

Desse modo, o único candidato a "gargalo" do programa (fora a leitura do mapa), é a função que usa a BFS para calcular as distâncias das bicicletas aos visitantes. Para estudar sua complexidade, consideremos seu pior caso, incluindo possíveis otimizações que não iriam interferir na complexidade assintótica, como parar de buscar quando todos os visitantes já tiverem sido encontrados. No nosso exemplo de pior caso, vamos ter uma bicicleta e um visitante:

$$\begin{pmatrix} 0 & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & a \end{pmatrix} \tag{2}$$

No máximo, para cada vértice são verificados 4 vizinhos, então essa medida

será tomada como constante. E um vértice nunca é visitado mais de uma vez. Quando nenhum vértice está bloqueado de ser visitado (isto é, é um caractere proibido), todos os vértice são visitados. Desse modo, podemos estabelecer como teto para a execução dessa função $\Theta(4MN) = \Theta(MN)$.

Concluímos então que a execução de uma BFS é $\Theta(MN)$, mas o número de vezes que essa função é executada depende do número de visitantes, que como suposto anteriormente, não pode ser maior que 10. Portanto, a complexidade do programa como um todo é $\Theta(MN)$, com o cálculo das distâncias como etapa mais custosa computacionalmente.