Trabalho Prático 1
Algoritmos 1

Vinicius Silva Gomes

Universidade Federal de Minas Gerais
vinicius.gomes@dcc.ufmg.br

18 de maio de 2022

1 Introdução

O trabalho contextualiza um problema de alocação de bicicletas para pessoas num parque. É pedido a confecção de um software que, dado o conjunto de bicicletas preferidas dos visitantes e dado o mapa do parque, que contém a localização das pessoas, das bicicletas e das áreas inacessíveis no parque; seja capaz de alocar uma bicicleta para cada pessoa.

Sempre é possível alocar uma bicicleta para cada pessoa. No entanto, o software deve, além disso, retornar a escolha mais justa de bicicleta para um visitante. A "escolha mais justa"é definida através da relação entre a preferência do visitante para com a bicicleta e distância dele até a mesma.

Dessa forma, este artigo tem como objetivo documentar a solução deste problema, enfatizando como foi feita a Modelagem Computacional do problema, quais as características e algoritmos utilizados na Implementação e qual a Complexidade Assintótica desses algoritmos.

2 Modelagem Computacional

Para modelar o problema, foram feitas análises a respeito das entradas que ele admite e do retorno esperado. Inicialmente, é possível perceber que o software precisa retornar, após algum processamento, um conjunto de pares de visitantes e bicicletas. Um algoritmo conhecido que lida com esse cenário é o algoritmo de Gale-Shapley: uma solução para o problema do *emparelhamento estável* linear em função da entrada e que retorna um resultado ótimo em tempo polinomial.

Assim, para computar o emparelhamento através desse algoritmo, será necessário montar duas tabelas de preferência, uma para os visitantes e uma para as bicicletas. É importante ressaltar, também, que a definição de "estável"para esse problema é ligeiramente diferente da definição original do algoritmo. Para tanto, será necessário fazer algumas adaptações no algoritmo.

A tabela de preferências dos visitantes em relação às bicicletas é informada pela entrada. No entanto, para utilizá-la, será necessário ordenar os resultados em ordem decrescente, visando facilitar a execução do algoritmo. Além disso, o mapa do parque e o fato de que o critério de desempate para uma bicicleta preferida por dois visitantes é a distância entre eles e a bicicleta, nos sugere que talvez seja possível montar uma tabela de preferências para as bicicletas a partir dessa distância entre ela e os visitantes. Iremos seguir por essa linha de raciocínio.

Para calcular as distâncias, podemos traduzir um mapa informado em um grafo, interpretando a matriz como um grid a ser percorrido (grafo está implícito no grid). Assim, podemos percorrer esse grid, identificando as menores distâncias entre os visitantes e as bicicletas. Um algoritmo conhecido que computa a distância mínima entre nós de um grafo é a busca em largura (BFS), do inglês *breadth-first search*. Dessa forma, podemos executar uma BFS para cada bicicleta e armazenar a distância mínima obtida até cada visitante. Após isso, a segunda tabela necessária para executar o algoritmo de Gale-Shapley seria obtida.

Com a entrada processada, basta executar o algoritmo de Gale-Shapley e obter o emparelhamento estável para cada conjunto de visitantes e bicicletas. Os detalhes específicos referentes a implementação de cada passo descrito nessa modelagem serão melhor desenvolvidos na seção 3.

3 Implementação

O programa foi desenvolvido na linguagem **C++** e compilado pelo compilador **G++**, da **GNU Compiler Collection**. Algumas variáveis de entrada aparecerão com frequência nos pseudocódigos: V, número de visitantes/bicicletas; e N e M, as dimensões do mapa. A seguir, os principais algoritmos do programa serão discutidos.

3.1 `main()`

A função `main` recebe uma entrada do problema, armazenando o grafo e a tabela de preferências em matrizes do tipo `vector<vector<char>>` e `vector<vector<int>>`, respectivamente, ordena a tabela de preferências dos visitantes e a tabela de distâncias das bicicletas através dos métodos `sort_preferences()` e `sort_distances()`, respectivamente. Após isso, ela computa o *emparelhamento estável*, por meio da função `stable_matching()`, e exibe os resultados.

Para representação das matrizes foi utilizado a estrutura `vector`, que basicamente computa um vetor de tipo genérico definido na declaração. Ele terá acesso a uma posição em $\mathcal{O}(1)$ e oferecerá alguns facilitadores, como o método `push_back`, para inserção.

3.2 `sort_preferences()`

Ordena as preferências dos visitantes, recebendo como entrada a matriz de preferências informada pelo problema. Para tanto, o algoritmo retorna uma matriz de pares ordenados em ordem decrescente, do tipo `vector<vector<pair<int, int>>>`, nos quais a primeira componente do par é a preferência do visitante por aquela bicicleta e a segunda é o identificador da bicicleta. A ordenação foi feita através do método `sort()`, implementado pela biblioteca STL.

Essa estrutura para a tabela foi utilizada pois existe a necessidade de salvar o identificador ao ordenar as preferências, visto que a referência à bicicleta através da posição no vetor é perdida com a ordenação. Assim, é possível ordenar as preferências armazenando-as em uma matriz e acessar tanto o ID quanto o valor da preferência de maneira prática. O pseudocódigo do algoritmo 1 mostra como essa ordenação é feita.

Algorithm 1 Ordena as preferências dos visitantes

Input: V e a matriz de preferências original M
Output: Matriz de pares P com as preferências ordenadas

```
P ← inicializa a matriz vazia
for i < V do
    T ← inicializa um vetor temporário vazio
    for j < V do
        pair ← (M[i][j], j)        ▷ M[i][j] = preferência; j = ID da bicicleta
        Insere pair em T
    end for
    T ← T ordenado em ordem decrescente do valor das preferências
    Insere T em P
end for
return P
```

3.3 `bfs()`

Percorre o grafo através de um vetor de movimentos `vector<pair<int, int>> moviment = ((-1, 0), (1, 0), (0, -1), (0, 1))`. Esse vetor irá variar a célula atual em uma das 4 direções possíveis e a nova célula será verificada. Caso a célula possa ser acessada (verificação feita pela função `valid_movement()`, que será discutida na subseção 3.6), ela será inserida numa fila do tipo `queue<pair<int, int>>`, que armazena as posições do grafo a serem processadas. Esse nó visitado é marcado, para que ele não seja processado de novo e a distância da bicicleta a cada nó do grafo também é calculada.

Caso uma célula possa ser acessada e, além disso, represente um visitante, o ID desse visitante e a sua distância até a bicicleta fonte da BFS são armazenados em um vetor de inteiros. A referência ao identificador do visitante é feita pela posição daquela distância nesse vetor.

Após processar o grafo, a BFS retorna esse vetor `vector<int>`, que será usado pela função `sort_distances()` para computar a tabela de distâncias. O pseudocódigo do algoritmo 2 mostra como esse percorrimento/processamento é feito.

Algorithm 2 Calcula a distância de uma bicicleta a todos os visitantes

Input: V, N, M o mapa G e bicicleta fonte s
Output: Vetor U com as distâncias calculadas

```
Inicializa o vetor U, a matriz de células visitadas S e a matriz de camadas L
M ← ((−1, 0), (1, 0), (0, −1), (0, 1))
while Existe alguma célula para percorrer do
    cell ← primeira célula em Q
    for moviment ← M do
        temp ← cell + moviment
        if temp é uma posição acessável then
            Insere temp em Q
            Marca temp como visitado e calcula sua distância até cell
            if temp é um visitante then
                U[temp] ← L[temp]
            end if
        end if
    end for
end while
return U
```

3.4 `sort_distances()`

Cria uma matriz do tipo `vector<vector<int>>` com as distâncias de cada visitante a cada bicicleta. A distância de uma bicicleta i a um visitante j é armazenada na posição $[i][j]$ da matriz, onde i e j são os respectivos IDs (ID do visitante é convertido do tipo `char` para `int` através da tabela $ASCII$).

Para calcular as distâncias, o método `find_bike_coords()`, que será melhor descrito na subseção 3.6, irá percorrer o mapa e guardar as coordenadas de cada bicicleta. Após isso, uma BFS será executada a partir de cada bicicleta encontrada e a sua distância até cada um dos visitantes será computada. O retorno da BFS será concatenado na matriz resultado e a tabela de distâncias terá sido montada.

O pseudocódigo do algoritmo 3 mostra como essa criação é feita.

Algorithm 3 Ordena as distâncias das bicicletas aos visitantes

Input: V, N, M o mapa G
Output: Matriz W com as preferências de distâncias

```
W ← inicializa a matriz vazia
B ← find_bike_coords()
for bike ← B do
    D ← bfs()
    Insere D em W
end for
return W
```

3.5 `stable_matching()`

Realiza o emparelhamento estável entre os visitantes e as bicicletas. Recebe as matrizes de preferência dos visitantes e da distâncias e computa o emparelhamento. Para isso, essa função utiliza uma fila do

tipo `queue<int>`, que armazena os IDs dos visitantes livres a serem emparelhados, e uma matriz do tipo `vector<vector<bool>>` que armazena se o i-ésimo visitante já fez uma proposta para a j-ésima bicicleta.

Enquanto a fila tiver algum visitante, ela escolhe uma bicicleta que ele ainda não propôs, na matriz booleana `already_proposed` e verifica se essa bicicleta já possui um par. Caso não, esse visitante livre fará par com essa bicicleta. Caso sim, as distâncias e os IDs desses visitantes serão comparados. Se a distância do visitante livre à bicicleta for menor que a distância do par à bicicleta, a bicicleta troca de par e o antigo par se torna livre. Se a distância for a mesma, aquele que tiver o menor ID fará par com a bicicleta e o outro se tornará/continuará livre. Caso a distância do visitante à bicicleta seja maior que a do par à bicicleta, o visitante continua livre.

O emparelhamento retorna um `map<char, int>` que relaciona um visitante e o seu respectivo. No entanto, durante toda a execução do Stable Matching, é usado um `map` do tipo `map<int, char>`. Isso foi feito para que seja possível verificar o par de uma bicicleta em tempo logarítmico, evitando ter que percorrer o `map` toda vez para verificar se uma bicicleta possui par. Após o fim do emparelhamento, o `map` terá suas componentes invertidas, aproveitando a ordenação para imprimir a saída do programa, e o retorno da função será feito.

O pseudocódigo do algoritmo 4 mostra como o emparelhamento é feito.

Algorithm 4 Emparelhamento estável dos visitantes e bicicletas

Input: V e as matrizes de preferências
Output: Mapa com os emparelhamentos

```
while Existe um visitante v que possui alguma bicicleta para propor do
    b ← bicicleta melhor ranqueada da lista de v que ele ainda não propôs
    if b não possui par then
        (v, b) se tornam um par
    else
        v' ← par atual de b
        if v está mais próximo de b do que v' then
            (v, b) se tornam um par
            v' se torna livre
        else
            if v e v' estão a mesma distância de b then
                b faz par com o que possuir o menor ID
                O outro visitante se torna livre
            end if
        end if
    end if
end while
return S
```

3.6 Funções auxiliares

Ao longo do código, algumas funções auxiliares foram utilizadas para compor funções maiores no programa. Esta subseção vai listar cada uma delas e resumir a sua utilização dentro do programa.

- `sort_comparison()`: redefine a comparação da função `sort()` da biblioteca STL para que ela ordene pares em ordem decrescente;
- `valid_movement()`: retorna um `booleano` que diz se a célula de destino após um movimento da BFS é uma célula navegável;
- `is_visitor()`: retorna um `booleano` que diz se o caractere passado como parâmetro representa um visitante no grafo;
- `is_bike()`: retorna um `booleano` que diz se o caractere passado como parâmetro representa uma bicicleta no grafo;

- `find_bikes_coords()`: percorre o mapa identificando as células que contém bicicletas e retorna um `map<int, pair<int, int>>` composto pelo identificador da bicicleta e suas coordenadas no grafo. O uso do `map` auxiliará a BFS a computar e concatenar as distâncias de uma bicicleta aos visitantes já na ordem correta dos IDs.

4 Análise de Complexidade

Para definir a complexidade do algoritmo, serão usados alguns padrões para se referir as variáveis do problema. Iremos denotar V como o número de bicicletas/visitantes, N e M como as dimensões do grafo e G como o grid que representa o mapa/grafo.

Os algoritmos foram separados e suas complexidades serão analisadas individualmente, a princípio. Ao final, será dado a complexidade total do algoritmo implementado que resolve o problema proposto.

- Para realizar a **ordenação da tabela de preferências dos visitantes**, as tarefas mais custosas são percorrer dois *for loop's* de 0 a V, para criar a matriz de preferências, e ordenar um vetor de pares em ordem decrescente. A complexidade para essas atividades é $\mathcal{O}(V^2)$ e $\mathcal{O}(V \log V)$, respectivamente. Assim, a complexidade desse trecho é $\mathcal{O}(V^2)$.

- A **BFS** percorre o grafo inteiro para calcular as distâncias entre as células. Sendo assim, a complexidade de uma BFS individual sob grid implementado por uma matriz é $\mathcal{O}(N \cdot M)$. A função `find_bikes_coords()` também percorre o grid todo para encontrar as coordenadas das bicicletas, portanto também é $\mathcal{O}(N \cdot M)$

- A função que **cria a tabela de preferências das bicicletas** executa uma BFS para cada bicicleta identificada no mapa. Como existem V bicicletas e a complexidade da BFS já foi descrita previamente, esse trecho tem complexidade de $\mathcal{O}(V \cdot (N \cdot M))$.

- Por fim, o **algoritmo que implementa o Stable Matching** tem o pior caso quando cada visitante precisa "propor"para cada bicicleta de sua lista de preferências, antes do emparelhamento estável ser alcançado. Essa tarefa tem complexidade de $\mathcal{O}(V^2)$.

Dessa forma, ao analisar o programa com todos esses algoritmos, é possível constatar que ele possui complexidade de $\mathcal{O}(2 \cdot V^2 + 2 \cdot (N \cdot M) + V \cdot (N \cdot M)) \in \mathcal{O}(V^2 + V \cdot (N \cdot M))$.

5 Compilando o Código

Para compilar o código, foi usado um `makefile` com o código:

```
all:
    g++ main.cpp -Wall -o tp01
```

Desse modo, caso o `makefile` não esteja disponível no ambiente de testes, o comando `g++ main.cpp -Wall -o tp01` deverá ser utilizado para compilar, e o comando `./tp01 < ./test_cases/1.txt` para executar o programa com alguma entrada de exemplo "1.txt".