BikeShapley - Documentação

Autor: João Pedro Fernandes Silva
Emails: joaofernandes@dcc.ufmg.br / joopedrof.silva@gmail.com
Belo Horizonte – MG – Brazil

---

1. Introdução

Essa documentação lida com o problema apresentado no trabalho prático 01 da matéria de Algoritmos I da turma TN 2022/1 da Universidade Federal de Minas Gerais. O problema lida com a situação fictícia de um sistema de aluguel de bicicletas, assim os usuários desse sistema devem ser alocados às bicicletas de maneira ótima, gerando pares estáveis que respeitem as listas de preferência de cada usuário em conjunto com a distância de cada bicicleta.

2. Implementação

Para solução do problema, foram projetadas duas classes principais, *Bicycle* e *User*, que abstraem as funcionalidades dos usuários e bicicletas no problema e têm as seguintes especificações de atributos:

- *Bicycle:* Classe que define uma bicicleta. Tem como atributos variáveis inteiras "_id" e "_match", que salvam respectivamente, o número identificador único da bicicleta e o identificador único do usuário que foi relacionado à bicicleta. Além destas variáveis, a classe *Bicycle* salva dois vetores de inteiros "*vector*<*int*>_prefList" e "*vector*<*int*> _rankList" que salvam respectivamente, a lista de preferência de cada bicicleta(em relação ao _id dos usuários) em ordem crescente e a posição de cada usuário nessa lista. A lista de preferência das bicicletas é calculada a partir da distância entre cada usuário, como ela é gerada será explicada mais à frente.
- *User:* Classe que define um usuário. Tem como atributos variáveis inteiras "_id" e "_match", que salvam respectivamente, o número identificador único do usuário e o identificador único da bicicleta que foi relacionada ao usuário. Além destas variáveis, a classe *User* salva um vetor de inteiros "*vector*<*int*> _prefList" que salva a lista de preferência do usuário(em relação ao _id das bicicletas) em ordem crescente e um iterador "it" para este vetor que aponta para a última bicicleta que foi feita uma proposta. A lógica de proposta às bicicletas será explicada mais à frente.

Para facilitar a implementação da solução, foram especificados os seguintes tipos de dado, que estão descritos no arquivo de cabeçalho "DataTypes.h":

- *typedef pair<int, int> coord*: Tipo de dado coordenada (linha, coluna) para salvar um ponto no mapa.

- *typedef pair<int, int> people*: Tipo de dado pessoa (id, level) onde level é a distância da pessoa em relação à uma bicicleta.
- *typedef std::pair<coord, int>coordAt*: Tipo de dado coordAt, para salvar uma coordenada relacionada a uma distância.

A seguir serão detalhadas as funções de geração das listas de preferências de usuários de bicicletas, estas funções foram implementadas no arquivo de cabeçalho "genPrefList.h".

- *std::vector<people>* genPrefList_Bicycle(char** matrix, int maxRow, int maxCollum, coord init, unsigned int numPeople)*: Função que gera a lista de preferências de uma bicicleta qualquer. A lista de preferências de uma bicicleta é gerada a partir da distância da coordenada *init* (coordenada da bicicleta que se quer gerar a lista) até todas os usuários do mapa, sendo *_prefList[0]* o usuário com maior preferência e *_prefList[numPeople]* o usuário com menor preferência. Para realizar a pesquisa de usuários no mapa, a função realiza uma busca BFS de forma recursiva a partir da coordenada *init* e retorna um vetor do tipo *people* com a variável *first* sendo o id da pessoa encontrada e *second* a distância da pessoa em relação à bicicleta. Em caso de empate entre a distância das pessoas, a pessoa de menor _id tem a preferência, a função trata esses casos e retorna a lista de preferências ordenada independente de empates na distância.
- *void genPrefList_User(std::vector<int>& initPref):* Função que, a partir de uma lista de pontuação para as bicicletas, gera a lista de preferências do usuário. A lista de notas é lida diretamente do arquivo de entrada e funciona de forma que *initPref[bicycle]* é um valor inteiro diretamente proporcional ao interesse da pessoa pela bicicleta *bicycle*. A função *genPrefList_User* modifica essa lista de forma a retornar uma lista de preferência da forma *initPref[0]* sendo o id da bicicleta de maior preferência e *_prefList[numPeople]* a bicicleta de menor preferência.

Para a classe *Bicycle* foram desenvolvidos três métodos principais:

- *bool Bicycle::genPrefList(vector<people>* prefList)* – A partir da lista ordenada *prefList* gerada na função *genPrefList_Bicycle()*, copia o _id de cada pessoa e salva na lista de preferência da bicicleta. Retorna 1 se a lista for gerada com sucesso e 0 caso contrário.
- *bool Bicycle::genRankList()* – Gera uma lista de rankings "*vector<int>_rankList"* de pessoas para uma bicicleta da forma *_rankList[_prefList[i]] = i.* Essa lista é usada para comparar a posição de duas pessoa na lista de preferências em tempo constante. Retorna 1 se a lista for gerada com sucesso e 0 caso contrário.
- *int Bicycle::tryMatch(int m)* –Analisa a proposta da pessoa de id *m*, e caso *m* tenha um rank maior que a pessoa que está alocada, aloca *m* no lugar. Caso ainda não tenha ninguém alocado, *_match* recebe *m* e a função

retorna -2. Caso a proposta seja aceita no lugar da pessoa que já está relacionada, *_match* recebe *m* e a função retorna a pessoa que foi trocada. Caso a proposta seja negada, retorna -1.

Já para a estrutura *User* foi implementado um método principal descrito a seguir:

- *int User::getItemFromPref()* – Computa o _id do próximo item a ser proposto pela lista de preferências de um usuário. Caso seja um item válido, retorna este item, e caso já tenha feito propostas a todas as bicicletas retorna -1.

A partir dessas estruturas o programa principal irá funcionar de forma a declarar e preencher um vetor de *U* usuários e um vetor de *B* bicicletas além gerar e preencher os respectivos vetores de preferência. Assim será aplicado o algoritmo de Gale-Shapley, com os usuários realizando propostas sequenciais às bicicletas e enquanto houver algum usuário que não tenha sido alocado a alguma bicicleta, novas propostas são feitas até que todos os pares estáveis sejam computados.

O programa foi implementado na linguagem C++11, compilado utilizando GCC (tdm64-1) 4.9.2 utilizando a opção do compilador -std=c++11, desenvolvido e testado em uma máquina com sistema operacional Windows 10 Home Single Language, 8GB de memória RAM e processador Intel Core i5-8250U CPU @ 1.60GHz 1.80GHz.

3. Instruções de compilação e execução

No diretório do Makefile execute os comandos, sendo <entrada.txt> o arquivo texto de entrada:

$ make

$ ./tp01 <entrada.txt>

4. Pseudocódigo

Algoritmo* 01 – *TP*01 *pseudocódigo

$Seja\ U = (u_0, u_1, \dots, u_n)_{usuarios}$
$Seja\ B = (b_0, b_1, \dots, b_n)_{bicicletas}$
$i \leftarrow 0$
Enquanto $i < n$ ***faça***:
    $u_i.preferencia = GeraPrefUsuario()$
    $b_i.preferencia = GeraPrefBicicleta()$
    $b_i.match = \emptyset$
    $i \leftarrow i + 1$
fim enquanto
Enquanto$(existe\ \ algum\ \ u\ que\ não\ foi\ alocado\ e\ nao\ propôs\ a\ todas\ as\ bicletas)$:
 $b \leftarrow primeira\ \ bicicleta\ \ na\ \ lista\ \ de\ \ preferencia\ \ de\ \ u\ \ na\ \ qual\ u\ ainda\ não\ propôs$
 $m \leftarrow b.match$
    se $m = \emptyset$:
        $b.match \leftarrow u$
    ou se $b.rank(u) > b.rank(m)$:
        $b.match \leftarrow u$
        $desaloca\ \ m$
    senao:
        $b\ \ nega\ \ u$
 fim enquanto
fim

5. Análise de Complexidade

- ***Bicycle* – Complexidade de espaço:** Cada objeto *Bicycle* declarado aloca dois vetores inteiros, sendo um de preferências e outro de rankings, sendo ambos de tamanho ***n*,** sendo *n* a quantidade de usuários que se tem no problema. Assim o custo de espaço para cada objeto *Bicycle* é O(*n*).
- ***Bicycle* – Complexidade de tempo:** Todos os métodos da classe *Bicycle* são dominados assintoticamente pelas funções *Bicycle::genPrefList()* que percorre uma lista de tamanho *n* copiando os valores para a sua lista de preferência. Dessa forma tem complexidade de tempo O(*n*).
- ***User* – Complexidade de espaço**: Cada objeto *User* declarado aloca um vetor inteiro de preferências de tamanho ***n*,** sendo *n* a quantidade de lojas que se tem no problema. Assim o custo de espaço para cada objeto *Pessoa* é O(*n*).
- ***User* – Complexidade de tempo**: Todos os métodos da classe *User* tem complexidade de tempo constante O(1).
- **genPrefList_Bicycle() – Complexidade de espaço:** A função salva *n* valores em uma lista ordenada. Assim a complexidade de espaço é O(*n*).
- **genPrefList_Bicycle() – Complexidade de tempo:** A função *genPrefList_Bicycle()* faz uma busca BFS em um mapa de tamanho de *rc*, sendo *r* o a quantidade de linhas do mapa e *c* a quantidade de colunas,

além de adicionar as pessoas de forma ordenada em uma lista de tamanho *n*, essa inserção ordenada tem custo O(n^2) e a busca BFS tem custo O(rc), dessa forma a função tem custo de tempo O($rc + n^2$).

- ***main* – Complexidade de espaço**: A função main gera o vetor de usuários e bicicletas, ambos de tamanho *n*, além de salvar uma matriz de tamanho *r x c*, assim sua complexidade de espaço é O($rc + n$).
- ***main* – Complexidade de tempo:** Além das funções já citadas, o main executa o algoritmo de Gale-Shapley em uma lista de tamanho *n*, o algoritmo de Gale-Shapley tem complexidade assintótica O(n^2). Porém o main invoca a função *genPrefList_Bicycle()*, que domina assintoticamente as outras funções no escopo com complexidade O($rc + n^2$).

6. Conclusão

Este trabalho lidou com o problema de casamento estável entre dois grupos distintos (usuários e bicicletas), na qual a abordagem utilizada para resolução foi a criação de estruturas que continham listas de preferências que a partir da ordenação e ranqueamento dos itens das listas, e para a geração dessa lista foi necessário abstrair conhecimentos de algoritmos comumente utilizados em grafos para realizar uma busca BFS em uma matriz. A execução deste trabalho foi importante para visualizar na prática a implementação do algoritmo de Gale Shapley e ser criativo para implementar o BFS em uma matriz e entender que pequenas mudanças no problema(como o mapa ter posições que não podem ser acessadas) podem mudar completamente a solução adotada.