Trabalho Prático 1 em Algoritmos 1

Gustavo Freitas Cunha

Matrícula: 2020054498

Departamento de Ciência da Computação - Instituto de Ciências Exatas -
Universidade Federal de Minas Gerais
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1. Modelagem

Para resolução do problema proposto, o mapa foi implementado como uma matriz de caracteres e, para calcular a distância, cada célula foi abstraída a um nó (implementado como uma classe) de um grafo, em que é aplicado o algoritmo de Busca em Largura (BFS) para encontrar o caminho mais curto entre um visitante e uma bicicleta, considerando os obstáculos como nós já inicialmente "explorados", conforme a definição desse algoritmo. Haverá uma aresta entre uma célula/nó e outra, caso a célula vizinha na matriz não seja um obstáculo e esteja dentro dos limites do mapa. Visitantes e Bicicletas foram abstraídos como classes, cada qual com seus atributos de acordo com a especificação.

A alocação entre bicicletas e visitantes foi feita com base no algoritmo de Gale-Shapley, considerando as duas entidades do problema, sendo o visitante a parte que "propõe" o casamento/alocação, isto é, a alocação final é ótima para os visitantes.

Detalhes dos algoritmos utilizados serão dados na seção 3.

2. Estruturas de Dados

Os nós, aqui representando células do mapa, utilizados no algoritmo BFS ganharam uma classe, contendo um construtor, um par ordenado, referente à sua coordenada cartesiana no mapa e um inteiro distância, que representará a distância do visitante no qual o BFS começa.

A entidade Bicicleta foi implementada como uma classe de mesmo nome, contendo id, idVis (id do visitante alocado atualmente, inicializado com -1) e um par ordenado localizacao.

Também foi criada uma struct preferencia, contendo um inteiro preferencia, que é a nota atribuída a uma bicicleta (class Bicicleta), segundo elemento da struct.

Os visitantes também foram implementados como uma classe, contendo id, idBike (id da bicicleta com que ele está alocado atualmente, inicializado com -1), um par ordenado chamado localizacao, e uma lista de preferencias, implementada com o container deque.

Observação é que as bicicletas foram tratadas normalmente com seus id's numéricos, mas os visitantes tiveram seus id's literais convertidos para números também, onde 'a' é mapeado para '0', 'b' para '1', e assim por diante. A classe visitante tem o método alfaId() para realizar esta conversão. Somente ao final da execução do programa é que os números são convertidos de volta para letras, para impressão dos resultados da alocação.

3. Algoritmos

Para resolução do problema de alocar bicicletas a visitantes, foi utilizado o algoritmo de **Gale-Shapley**, que é não determinístico, já que este é um clássico algoritmo resolvedor do problema de casamento estável, considerando as condições de estabilidade definidas neste problema. No caso deste problema, o não determinismo quer dizer que a ordem em que os visitantes solicitam a locação de uma bicicleta não importa para a alocação final. Segue seu pseudocódigo:

```
Gale-Shapley (Lista de preferência dos visitantes){
    M:= Lista dos pares alocados (inicialmente vazia)
    while(algum visitante v esteja alocado e sua lista de preferências não esteja
vazia){
            b:= Primeira bicicleta na lista do visitante v
            if(b está disponível){
                    Adiciona-se v-b a M
            }
            else if(b está mais perto de v do que seu par v' atual){
                    Substitui-se v'-b por v-b em M
            }
            else{
                    Sem alocação
            }
            Remove b da lista de preferências de v
    }
    return M
}
```

O critério de desempate para a distância entre a bicicleta e o visitante é o id numérico do visitante. O "casamento" é feito com aquele de menor id, conforme estabelecido nas regras de especificação do projeto.

Para resolução do problema de encontrar o caminho mínimo entre um visitante e uma bicicleta, considerando as especificações do mapa, foi utilizada uma adaptação do algoritmo de **Busca em Profundidade em grafos, o BFS**, já que este é um algoritmo linear e clássico para tratamento de caminhos em grafos não direcionados de maneira geral. Note que foi utilizada a modelagem descrita na seção 1 para isso. Segue abaixo seu pseudo-código:

```
BFS (Visitante vis, Bicicleta bike, mapa){
     distancia[] //vetor de controle de distância das células a vis

     explorado[] //vetor de controle de navegação no mapa
     explorado[vis] = true
     para cada obstáculo o no mapa, set explorado[o] = true
     para cada outra célula x no mapa, set explorado[x] = false

     camadas := lista contendo inicialmente apenas vis
     while(camadas não está vazio){
            aux := camadas[0]
            Remove o primeiro item de camadas
            if(elemento nas coordenadas de aux for bike){
                   return distancia[aux]
            }
            above := célula uma unidade acima de aux
            if(above está em mapa and explorado[above] = false){
                   distancia[above] = distancia[aux] + 1
                   add above ao final de camadas
            }
            under:= célula uma unidade abaixo de aux
            if(under está em mapa and explorado[under] = false){
                   distancia[under] = distancia[aux] + 1
                   add under ao final de camadas
            }
            right:= célula uma unidade à direita de aux
            if(right está em mapa and explorado[right] = false){
                   distancia[right] = distancia[aux] + 1
                   add right ao final de camadas
            }
            left:= célula uma unidade à esquerda de aux
            if(left está em mapa and explorado[left] = false){
                   distancia[left] = distancia[aux] + 1
                   add left ao final de camadas
            }
     }
}
```

4. Análise de Complexidade

A solução implementada possui complexidade assintótica $O(V^2 * max(V, MN))$, sendo V o número de visitantes que, por sua vez, é igual ao número de bicicletas, N o número de linhas do mapa e M o número de colunas do mapa.

Note que um processamento no mapa, qualquer que seja ele, é sempre em função de seu número de linhas e colunas, isto é, $O(NM)$. Assim, encontrar um visitante/bicicleta no mapa é $O(NM)$. Esta solução foi implementada de tal forma que um visitante/bicicleta é procurado uma só vez no mapa. Logo, temos, até aqui, uma complexidade de $2V * O(NM) = O(VNM)$, já que temos V visitantes e V bicicletas.

Já o cálculo da distância entre um visitante e uma bicicleta é linear em função do número de arestas (aqui modelado com número de vizinhos válidos que cada célula pode ter, isto é, 4; assim, temos até $4MN$ arestas) somado ao número de nós/vértices (NM), por ser uma busca em largura. Assim, temos $O(4MN + MN) = O(MN)$.

O algoritmo de Gale-Shapley, utilizado para fazer as alocações visitante-bicicleta tem, conforme visto em aula, uma complexidade $O(V^2)$, já que cada visitante pode propor a todas as V bicicletas e existem também V visitantes. No caso desta implementação, a alocação/realocação de pares, é $O(1)$, por utilizar índices/atributos das classes implementadas. Além disso, o controle das propostas, também é $O(1)$, já que a lista de preferências foi implementada como uma lista encadeada e a remoção no início tem custo constante.

Agora, para que se deduza qual será o próximo visitante "a propor" para uma bicicleta, é feita uma varredura no vetor de visitantes (tamanho V) até que se encontre um visitante sem par e que não tenha proposto a todas as bicicletas da sua lista de preferências. Essa varredura é claramente linear em V, logo, a cada iteração, temos um custo $O(V)$.

Note que, a cada iteração, pode ser necessário calcular a distância entre visitante e bicicleta - $O(MN)$. Desse modo, teremos a complexidade total dessa adaptação de Gale-Shapley igual a $V^2 * (O(V) + O(MN)) = O(V^2 * max(V, MN))$.

Como $O(V^2 * max(V, MN)) > O(VMN)$, esta implementação custa $O(V^2 * max(V, MN))$.

5. Instruções para Compilação e Execução

Este programa havia sido implementado inicialmente para ser compilado utilizando um arquivo Makefile, todo modularizado e dividido em diretórios e arquivos. Devido à indisponibilidade de suporte para isso nas máquinas do CRC, tive que concatenar tudo em uma pasta só e a compilação e linkagem deverá ser manual. Seguem instruções para compilá-lo, linká-lo e executá-lo dessa maneira.

Primeiro, com o terminal no diretório **TP**, execute o seguinte comando, para compilação e geração do arquivo object (.o), de nome `main.o`:

```
g++ -std=c++17 -lstdc++fs -Wall -g -o main.o -c main.cpp
```

Na sequência, entre com o comando a seguir, para linkagem e geração do arquivo executável, de nome `tp01`:

```
g++ -g -o tp01 main.o -lm
```

Agora o projeto está pronto para ser executado. Ainda com o terminal no mesmo diretório, basta entrar com o comando:

```
./tp01 < <arquivo de entrada>.txt
```

É necessário que o arquivo de entrada também esteja no diretório TP, caso contrário, é preciso indicar seu caminho no terminal.

Conforme solicitado, os resultados serão impressos na saída padrão (stdout).

Por motivos de possível incompatibilidade na arquitetura de computadores, os arquivos objects e o executável gerados em meu computador não foram entregues, sendo gerados por você ao executar os dois primeiros comandos descritos acima.

O arquivo Makefile criado por mim também não foi enviado já que, infelizmente, não será utilizado.