Trabalho Prático I - Algoritmos I

Lucas Braz Rossetti
2020041590

Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

lucasbraz@ufmg.br

Modelagem Computacional

Abstraindo-se o necessário do contexto apresentado, o problema de alocação dos clientes às lojas de maneira otimizada é, em suma, uma variante do problema de casamento estável , onde a relação loja - cliente é similar à homem - mulher (ou o contrário) no problema original. Entretanto, neste caso, há uma proporção oscilantes de lojas/clientes, além de que cada loja possui um número de produtos (alocações de clientes) específico, portanto, foi razoável adotar uma versão adaptada do algoritmo de Gale-Shapley.

Cada entrada cai em um dos três casos:

- Estoque global < Número de clientes. Logo haverá clientes sem agendamento
- Estoque global = Número de clientes. Há um matching com todos incluídos
- Estoque global > Número de clientes. Lojas terão estoque sobressalente

Os dois primeiros casos são tratados nas condições originais do algoritmo de Gale-Shapley, mas para incluir o terceiro caso, basta verificar se uma dada loja já iterou por toda a lista de preferências, logo é imediato que, se o seu estoque não foi completamente preenchido, na solução otimizada, essa loja não vai operar na capacidade máxima.

Descrição da Implementação

Dada a complexidade do problema e em prol da construção de um código de boa qualidade, foram adotados os princípios da orientação a objetos e modularização, portanto classes auxiliares foram criadas para manipular a entrada de dados da melhor forma possível. A implementação foi dividida em 7 arquivos principais, distribuídos no seguinte esquema de diretório:

- TP
  |- src <armazena os arquivos de extensão ".cpp">
  |- obj <armazena arquivos objetos produzidos na compilação >
  |- include <armazena os arquivos de extensão ".h">
  Makefile

Cada classe possui um par de arquivos *header-source* , organizados da seguinte forma:

Shop - associada às entidades das lojas , classe descrita no par *<loja.h,loja.cpp>*, ***Client*** - representante das entidades dos clientes, dado pelo par *<cliente.h,cliente.cpp>* e ***Manager***, uma classe encarregada de gerir os objetos das duas anteriores, sendo esta munida de métodos que organizam o principal fluxo de dados do programa e. Esta última classe se encontra no par de arquivos <manager.h, manager.cpp>

Cada objeto ***Shop*** possui um vetor de ***Client*** que representa a atual alocação de clientes, enquanto todos clientes e lojas estão armazenados em dois vetores cujos acessos estão restritos ao ***Manager***. A classe dispõe de uma lista de preferência das lojas prefShop (note que, como o critério de prioridade de clientes não depende de algum dado das lojas, a lista é global - isto é, é a mesma para todas as lojas) que apresenta o id dos clientes em ordem de prioridade, já considerando os critérios de desempate. Há também um vetor clientIter cujo valor no índice *i* representa a posição na lista de preferência do próximo cliente a ser consultado pela loja de id *i*.

O fluxo do programa é cadastro >> agendamento >> impressão, onde cada uma das etapas está representada por um método da classe *Manager*, respectivamente: registration() , schedule() e printSchedule(). A primeira e a terceira funções seguem o padrão de entrada/saída da especificação , enquanto a função schedule() foi construída com base em uma versão adaptada do algoritmo de gale-shapley:

```
while(existe uma loja s que ainda pode agendar um cliente)
        while(s pode agendar um cliente)
                c = primeiro cliente que ainda não foi consultado por s na lista de preferência
                If (c não está agendado a nenhuma loja)
                        agendar c a s
                else if(c está agendado a s' e distância(c,s) < distância(c,s'))
                        desfazer o agendamento c e s'
                        agendar c a s
```

Note que, para a condição dos loops abranger os 3 casos descritos na modelagem do problema, uma loja s “poder agendar um cliente” é equivalente a dizer que s ainda possui estoque disponível e não consultou todos os clientes da lista, como já foi comentado. Neste sentido, duas funções auxiliares foram criadas:

- *nextCustomer(int idS)* - se ainda houver um cliente a ser consultado, retorna a sua posição na lista de preferência, se não, retorna -1.
- *frstNonFull()* - procura por uma loja que ainda tenha estoque disponível e clientes a consultar

O indíce da próxima loja no itinerário é determinado pela segunda função e o loop externo é finalizado quando não há nenhuma loja que tenha estoque disponível e ainda não consultou todos os clientes. De forma semelhante, o loop interno é finalizado de acordo com a condição de existência descrita acima, mas aplicada para uma loja particular s.

Na classe Manager, os demais métodos são funções auxiliares ao cadastro dos dados dos objetos durante a primeira fase de funcionamento do programa. A classe Shop e Cliente possuem, em sua maioria, apenas getters e setters como métodos relevantes e, portanto, a descrição dada pelo código é suficiente.

Análise de Complexidade

Primeiramente, notamos que as variáveis de interesse no cálculo da complexidade do código são apenas o número de lojas e o número de clientes, que denotaremos, respectivamente, por **m** e **n**. O programa é executado primariamente por três métodos da classe Manager na main(): registration() , schedule() e printSchedule() , agora seja T(n) a função que retorna a classe assintótica de um método. Assim, temos:

$$T(main()) = T(registration()) + T(schedule()) + T(printSchedule())$$

Vamos analisar cada método separadamente:

1) $T(registration())$:

A função que realiza o cadastro dos dados executa , entre vários comandos de complexidade constante, cinco operações relevantes ao cálculo:

- Execução do loop de entrada dos dados (operações O(1)) das m lojas: possui complexidade $O(m)$
- Execução do loop de entrada dos dados (operações O(1)) dos n clientes: possui complexidade $O(n)$
- Preenchimento de um vetor com os tickets de cada um dos n clientes: possui complexidade $O(n)$
- Execução do método std::stable_sort sob um vetor de n posições: possui complexidade $O(n * log(n))$
- Preenchimento de um vetor com os ids de cada um dos n clientes: possui complexidade $O(n)$

Logo, temos que

$$T(registration()) = O(m) + 3O(n) + O(n\,log\,(n) = O(m) + O(n\,log\,n)$$

2)$T(schedule())$:

Aqui vamos assumir que o estoque das lojas possui uma distribuição uniforme, isto é, toda loja possui estoque n/m.

O loop exterior é executado pelo menos por toda a lista de lojas, portanto possui complexidade O(m) , mas note que,a cada iteração com a loja s, temos dois blocos com complexidade não constante:

- Um loop que contém apenas operações de complexidade O(1) que é executado enquanto :
  a) há estoque disponível na loja s e
  b) s ainda não tentou agendar com todos os clientes disponíveis
  Considerando que o estoque de cada loja segue uma distribuição uniforme, podemos afirmar que o pior caso ocorre quando a loja precisa iterar por toda a lista de prioridade, isto é, o bloco assume complexidade O(n).
- Para encontrar a próxima loja, a função frstNonFull() de complexidade O(m) é executada

Portanto, considerando que os dois blocos se encontram dentro do loop de complexidade O(m) , teremos que :

$$T(schedule()) = O(m) * (O(n) + O(m)) = O(mn) + O(m^2)$$

3)$T(printSchedule())$:

Novamente vamos considerar que os estoques possuem distribuição uniforme, ou seja, iterar pelo estoque de uma loja qualquer s assume complexidade da ordem O(n/m). Sendo assim, dado que a função printSchedule() imprime cada cliente associado a cada loja, teremos que:

$$T(printSchedule()) = O(m) * O(n/m) = O(n)$$

Finalmente podemos calcular a complexidade geral da execução do programa:

$$T(main()) = T(registration()) + T(schedule()) + T(printSchedule())$$

$$T(main()) = O(m) + O(n\,log(n)) + O(mn) + O(m^2) + O(n)$$

$$T(main()) = O(n\,log(n)) + O(mn) + O(m^2)$$