Henrique Soares Assumpção e Silva
Matrícula: 2020006620
henriquesoares@dcc.ufmg.br

Trabalho Prático I
Algoritmos I
Universidade Federal de Minas Gerais

2021-11-22

1 Introdução

Durante o mês de Novembro ocorre uma das datas mais relevantes para as empresas de vendas nacionais, a Black Friday. Tendo em vista o grande número de clientes que realizam compras nesse período, grandes empresas do varejo nacional desejam desenvolver um sistema automático para agendamento de clientes para lojas de maneira ótima.
Formalmente, seja um conjunto de clientes $\mathcal{C}$, $|\mathcal{C}| = n$, onde cada $c \in \mathcal{C}$ é uma 5-tupla $(id_c, i_c, uf_c, u_c, (x_c, y_c))$, sendo id_c um número identificador único, i_c a idade do cliente, uf_c seu estado de origem, u_c sua forma de pagamento mais frequente e (x_c, y_c) uma 2-tupla identificando as coordenadas geográficas do cliente, e seja um conjunto de lojas $\mathcal{L}$, $|\mathcal{L}| = m$, onde cada $l \in \mathcal{L}$ é uma 3-tupla $(id_l, (x_l, y_l), l_e)$, sendo que id_l e (x_l, y_l) são análogas aos atributos previamente descritos para os clientes e l_e representa o estoque da loja. Deseja-se criar um algoritmo capaz de, dado os conjuntos $\mathcal{C}$ e $\mathcal{L}$ de entrada, computar um casamento estável ótimo do ponto de vista das lojas.

2 Modelagem

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection, no sistema operacional Windows 10. O programa foi inicialmente implementado e compilado em uma máquina com 16GB de RAM em um processador Intel Core I5-9400F.

2.1 Classes e Estruturas

Para a implementação do algoritmo requisitado, foram criadas classes e estruturas para auxiliar no desenvolvimento da lógica do programa. Seguem as descrições de cada uma dessas construções:

- `Struct Point`: Estrutura simples que representa um ponto em duas dimensões. Foi utilizada para armazenar os dados x e y das lojas e clientes. Possui apenas métodos construtores e destrutor.
- `Class Client`: Classe que representa um cliente. Armazena os valores da 5-tupla $(id_c, i_c, uf_c, u_c, (x_c, y_c))$ que define o cliente. Além disso, também realiza o cálculo do valor do ticket do cliente(t_c) em seu construtor, dado por:
$$t_c = \frac{(|60 - i_c| + \phi_{uf}(uf_c))}{\phi_u(u_c)} \tag{1}$$
Onde ϕ_u e ϕ_{uf} são funções que mapeiam a forma de pagamento mais frequente e o estado de origem de um cliente para números inteiros, respectivamente. A classe possui apenas métodos `get` para alguns de seus atributos privados($id_c, t_c e(x_c, y_c)$) e métodos construtor e destrutor padrão.
- `Class Store`: Classe que representa uma loja. Armazena os valores da 3-tupla $(id_l, (x_l, y_l), l_e)$ que define a loja. Além disso, também possui um atributo de tipo lista que contém o conjunto ordenado dos clientes preferidos por essa loja, de acordo com os critérios estabelecidos no enunciado. A classe possui um método `get` e `set` para essa lista, sendo que o método `get` retorna e remove o primeiro cliente da lista, e o método `set` copia os elementos de um vetor ordenado de clientes para a lista em questão. Além disso, a classe possui métodos `get` para todos os atributos que inicialmente definem a loja. Por fim, a classe também possui métodos para informar se a lista de preferências está vazia e para subtrair um determinado valor do estoque atual da loja.

2.2 Funções

Além das estruturas descritas na seção anterior, também foi-se desenvolvido funções específicas para o funcionamento do algoritmo. Seguem suas descrições:

- `bool client_bigger(Client &c1,Client &c2)`: Função que compara dois clientes e determina se o primeiro($c1$) tem precedência sobre o segundo($c2$), isto é, determina se o primeiro cliente possui um valor de ticket maior ou se seu identificador único é menor que o segundo cliente.

- `int grid_distance(Point p1, Point p2)`: Função que calcula a distância entre dois pontos, de acordo com as diretrizes determinadas pelo enunciado do problema. Dados dois pontos $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$, calcula a distância($D(p_1, p_2)$):

$$D(p_1, p_2) = max(|x_1 - x_2|, |y_1 - y_2|) - 1 \quad (2)$$

  O valor 1 é subtraido ao final da operação de máximo para atender à métrica definida no enunciado do trabalho, isto é, considera-se apenas os pontos intermediários entre os dois pontos em questão no cálculo da distância, e portanto o movimento de "mover-se do ponto adjacente à p_2 para p_2" não é contado no cálculo da distância.

- `read_data`: Função criada para realizar a leitura dos dados de entrada. Essa função tem funcionamento extremamente simples e serve apenas para tornar a função principal `main` menos carregada.

2.3 Programa principal

Esta seção ia descrever o funcionamento da função principal `main` do programa desenvolvido. Segue uma explicação passo a passo de seu funcionamento:

1. Criam-se dois vetores que deverão armazenar os clientes e lojas existentes, e então esses vetores são preenchidos na função `read_data`. Como o enunciado do programa garante que quaisquer dados relacionados às coordenadas de clientes ou lojas irão respeitar os limites dados pelas dimensões da grid, essas dimensões em si não são necessárias para o funcionamento do programa, tendo em vista que é possível garantir o formato correto das entradas pelo enunciado. Assim sendo, esses valores não armazenados em nenhuma variável do programa.

2. Cria-se uma cópia do vetor de clientes e então ordena-se essa cópia em ordem decrescente de acordo com os critérios estabelecidos em `bool client_bigger`. Essa cópia representa a lista de preferências das lojas. Esse vetor então é passado como parâmetro para o método `set_preferences` de cada uma das lojas no vetor de lojas. Além disso, também cria-se uma lista, que irá agir como uma fila, com os ids de todas as lojas.

3. Cria-se um array com o tamanho igual ao número de clientes, e então o inicializa com o valor -1. Esse array irá armazenar o ids das lojas as quais os clientes estão alocados em uma determinada iteração do algoritmo. Ou seja, espera-se que, ao final do algoritmo, esse array contenha um casamento estável entre as lojas e os clientes.

Após estes procedimentos iniciais, se prossegue para o loop do algoritmo desenvolvido para realizar o casamento estável entre as lojas e clientes. Esse algoritmo sera descrito e discutido em detalhes na próxima seção.

3 Solução

Para solucionar o problema em questão, desenvolveu-se uma adaptação do Algoritmo de Gale-Shapley[1] para o problema de casamento instável. O algoritmo original considera um número igual de clientes(que no contexto do algoritmo são estudantes) e lojas(que no contexto do algoritmo são hospitais). Além disso, o algoritmo original considera que pode existir ao máximo um pareamento entre clientes e lojas, ou seja, toda loja tem um estoque igual a 1. Tendo em vista essas diferenças, foi necessário adaptar o algoritmo de Gale-Shapley para o contexto do trabalho, de forma a garantir que um casamento estável ainda seja atingido ao final de sua execução.
O algoritmo proposto para o resolver o problema itera sobre cada loja s, percorrendo sua lista de preferências e pareando s com clientes. Caso um cliente c da lista de s não esteja pareado com nenhuma loja, ele se pareia com s. Caso esse cliente já esteja pareado com alguma outra loja k, é feita uma checagem para saber se c prefere s a k, e, no caso positivo, se remove o pareamento de k com c e se cria um novo pareamento entre s e c. O cliente então é removido da lista de preferências de s e a loja k é adicionada na fila de lojas que ainda não estão cheias. O algoritmo para de checar por pareamentos para s no momento em que sua lista de preferências fica vazia - que significa que s já fez ofertas de agendamento a todos os clientes - ou no momento em que s fica cheia. Esse procedimento é repetido até que a fila de lojas não-cheias fique vazia. Vale notar que, no momento em que s já fez propostas de agendamento a todos os clientes, s é removida da fila de lojas não-cheias, mesmo que a mesma possua espaço livre, pois todos os clientes que preferem s já estarão pareados com a mesma e todos os outros clientes que não estão com s preferem outras lojas.
O algoritmo descrito é apresentado em pseudo-código a seguir:

Algorithm 1 Algoritmo de Gale-Shapley modificado

Require: $n, m > 0$
Ensure: Casamento estável em $\mathcal{M}$
  $\mathcal{U} \leftarrow \{s_0, s_1, ..., s_{m-1}\}$ ▷ Fila contendo as lojas que ainda não estão cheias
  $\mathcal{M} \leftarrow \varnothing$
  while $\mathcal{U} \neq \varnothing$ **do**
    $s \leftarrow$primeira loja em $\mathcal{U}$
    if s tiver lista de preferências vazia **then**
      remova o primeiro elemento de $\mathcal{U}$ ▷ Remove s de $\mathcal{U}$ caso a loja já tenha tentado todos os clientes
      $s \leftarrow$primeira loja em $\mathcal{U}$
    end if
    $c \leftarrow$primeiro cliente na lista de preferências de s ▷ Também remove esse cliente da lista de s
    if c não estiver alocado à nenhuma loja **then**
      adicione $s - c$ a $\mathcal{M}$
      diminua o estoque de s em 1
      if estoque de s for igual a zero **then**
        remova o primeiro elemento de $\mathcal{U}$ ▷ Remove s de $\mathcal{U}$ caso a loja esteja cheia
      end if
    else
      $k \leftarrow$ loja que está pareada com c
      if c prefere s a k **then**
        remova $s - k$ de $\mathcal{M}$
        adicione $s - c$ a $\mathcal{M}$
        aumente o estoque de k em 1
        diminua o estoque de s em 1
        adicione k ao final de $\mathcal{U}$
        if estoque de s for igual a zero **then**
          remova o primeiro elemento de $\mathcal{U}$ ▷ Remove s de $\mathcal{U}$ caso a loja esteja cheia
        end if
      end if
    end if
  end while

Agora será feita uma análise da corretude do problema. Existem dois cenários que classificam um pareamento como inválido:

1. "Um cliente c_1 está agendado para uma loja s_1, mas há uma loja s_2 mais próxima que possui estoque disponível para atender sua demanda ou possui um agendamento para um cliente c_2 de ticket menor que c_1"

2. "Há um cliente sem agendamento mas ainda existe estoque disponível para atendê-lo em alguma loja"

Primeiramente, considera-se o cenário 1. Iremos provar que tal cenário é impossível por construção. Imagine que após k iterações o algoritmo tenha acabado em tál cenário. Se a loja s_2 possui estoque disponível, ela necessáriamente estará na fila de lojas não cheias, por definição do algoritmo. Assim sendo, s_2 irá tentar se parear com os clientes na ordem de preferência até que ela esteja cheia. Se s_2 se parear a c_1, o cenário 1 não é mais o caso, pois o cliente agora está pareado com uma loja a qual ele prefere com relação à s_1, e se s_2 não se parear a c_1, o cenário também não é mais verdade pois a loja agora está cheia de clientes aos quais ela prefere com relação à c_1. Agora considere a situação de s_2 possuir um cliente c_2 de ticket menor que c_1, ou seja, s_2 prefere c_1 à c_2. Essa situação é impossível pela própria definição do algoritmo. A loja s_2 fez propostas aos clientes de acordo com sua lista de preferências, i.e. valor do ticket, até que ela se encontre cheia, ou seja, a loja s_2 está necessariamente pareada com os clientes mais próximos e de maior ticket, e portanto, se c_1 não foi pareado a s_2, ou seu ticket é menor que de $c2$ ou ele está mais próximo de s_1. Portanto, mostramos que o cenário 1 é impossível dentro do contexto do algoritmo proposto.
Agora consideremos o cenário 2. Iremos provar que tal cenário é impossível também por construção. Imagine que o algoritmo, após i iterações, se encontre em uma situação onde existe um cliente c que não possui agendamento e um conjunto $\mathcal{S} = \{s_0, s_1, ..., s_k\}$ de lojas não cheias, tal que $l_{s_i} > 0 \ \forall i \in \{0, 1, 2, ..., k\}$. Sabemos, por definição do algoritmo, que $\mathcal{S} \subseteq \mathcal{U}$, e portanto todas as lojas de $\mathcal{S}$ irão tentar se parear com seus clientes preferidos até que o número de clientes acabe ou até que todas as lojas fiquem cheias. Caso o número de clientes acabe, significa que existe alguma loja em $\mathcal{S}$ que está pareada com c, e portanto o cenário 2 se torna falso. Caso contrário, se todas as lojas ficarem cheias, ou alguma delas se pareou com c, também tornando falso o cenário 2, ou c continua sem pareamento, i.e. todas as lojas em $\mathcal{S}$ preferiam a outros clientes

com respeito à c. Nesse caso, o cenário 2 também se torna falso, pois não existem mais lojas não cheias que poderiam acomodar c. Portanto, mostramos que o cenário 2 é impossível dentro do contexto do algoritmo proposto.
Assim sendo, mostramos que todos os cenários que caracterizam um algoritmo falho são impossíveis para o algoritmo proposto, e portanto sua corretude está comprovada.

4 Análise de Complexidade

Nesta seção, será realizada a análise de complexidade de tempo do algoritmo proposto. Por motivos de simplicidade, a análise em questão será feita com respeito ao pior caso possível de execução do algoritmo. Considera-se n como o número de clientes dados como entrada e m como o número de lojas. Podemos definir $\mathcal{L} = \{s_1, s_2, ..., s_m\}$ como o conjunto de lojas em questão e $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ como o conjunto de clientes. A cada loja s_i está associado um estoque l_{s_i}, e o algoritmo realizará iterações até todas as lojas fiquem cheias ou até que todos os clientes estejam agendados a alguma loja. Independentemente da situação, no pior cenário possível cada loja $s_i \in \mathcal{L}$ deverá percorrer toda sua lista de prioridades na execução do algoritmo. Lembra-se que, no momento em que uma loja termina de percorrer toda sua lista de prioridades, ela é removida da fila de lojas não cheias, indepentemente se ela estivar cheia ou não, pois se ela não estiver cheia significa que todos os clientes que preferem ela à alguma outra loja já foram agendados a esta, e portanto não faz sentido realizar mais propostas.
Assim sendo, cada uma das m lojas deverá percorrer uma lista de tamanho n até que o algoritmo se encerre. Como em cada iteração do algoritmo são realizadas apenas operações que podem ser assumidas como tendo ordem de complexidade de tempo constante - nenhum loop é realizado dentro do loop while do algoritmo, ou seja, o número de operações não varia de acordo com o tamanho do elemento da iteração -, pode-se afirmar que o fator determinante na ordem de complexidade deste algoritmo é o seu número de iterações. Portanto, conclui-se que no pior cenário o algoritmo realizará $m \cdot n$ operações, e logo sua ordem de complexidade de tempo de pior caso é $\mathcal{O}(m \cdot n)$.
Em relação ao programa principal, i.e. função main, também podemos derivar sua ordem de complexidade de tempo de pior caso. Assim como descrito na seção 2.3, o programa principal precisa inicializar cada elemento de um vetor de lojas com uma lista de prioridades, operação que possui ordem de complexidade de $\mathcal{O}(m \cdot n)$, tendo em vista o tamanho dos vetores. Além disso, uma cópia do vetor de clientes é criada e ordenada utilizando o algoritmo de ordenação provido pela biblioteca padrão da linguagem C++, que possui ordem de complexidade de tempo de $\mathcal{O}(n \cdot log(n))$. Portanto, obtemos a seguinte expressão para a ordem de complexidade de tempo de pior caso para a função main do programa:

$$\mathcal{O}(m \cdot n) + \mathcal{O}(m \cdot n) + \mathcal{O}(n \cdot log(n)) = \mathcal{O}(max(m \cdot n, n \cdot log(n))) \quad (3)$$

References

[1] Almeida, J. (2021). Slides virtuais da disciplina de Algoritmos 1. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

[2] Cormen, Thomas H. Algoritmos: Teoria e Prática, Editora Campus, v. 2 p. 296, 2002

A Instruções para compilação e execução

Os arquivos foram estruturados de acordo com o exigido pelo enunciado, ou seja, para executar o programa principal basta seguir os seguintes passos:

- Utilizando um terminal, execute o arquivo `Makefile`, através do comando `make`
- Após isso, execute os seguintes passos de acordo com o sistema operacional utilizado:
- No terminal Linux, execute o comando:
  `./tp01.out <TARGET_FILE>`
  Onde `<TARGET_FILE>` indica o nome do arquivo com os dados.
- Por fim, execute o comando `make clean` para remover os arquivos `.o` e `.out` gerados.