Documentação do trabalho prático 1 da disciplina de Algoritmos 1

Caio César Silva - 2019075681

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

caiocs2019@ufmg.br

1. Modelagem computacional

No projeto, foram criadas e utilizadas 3 classes, de forma que cada uma possui sua responsabilidade. As classes de abstração são as classes Pessoa e PostoVacinacao, que representam as entidades envolvidas na solução. A outra classe é a classe Dados, onde ficam concentradas as regras de negócio, ou seja, processamento e comparação de dados. Todas as operações são chamadas no arquivo Main.

A organização das pastas e do Makefile estão na forma como nos foi ensinado na disciplina de Estrutura de Dados. Os headers das classes ficam na pasta “include”, a implementação de cada classe fica pasta “src”, quando as classes são compiladas, os arquivos com formato “.o” ficam na pasta “obj”. Na pasta “bin” fica o arquivo gerado para execução (formato “.out”) do programa, assim como deve ficar o arquivo de entrada.

Cada classe possui atributos públicos e privados, de forma que para acessar ou alterar um atributo privado, existe uma função getter ou setter para isso.

2. Instruções de compilação e execução

Configurações do meu computador e do compilador:

- Sistema Operacional: Windows 10 Pro x64
- Linguagem de programação utilizada: C++
- Compilador utilizado: g++ através do terminal WSL com Ubuntu instalado, utilizando o Makefile fornecido pela equipe da disciplina de Estrutura de Dados e alterado para as solicitações do trabalho atual.
- Processador/RAM: Intel Core i3-9100F 3.60GHz / 8GB RAM

Instruções de execução:

- Acesse o diretório raíz do projeto, nele, haverão as pastas **bin**, **include**, **obj**, **src** e o arquivo **Makefile**.
- Utilizando um terminal Linux, certifique-se de que você tenha o make e o g++ instalado, caso não, rode os seguintes comandos:
  - sudo apt-get install build-essential
  - sudo apt-get install make
  - sudo apt-get install g++
- Com o make instalado, rode o seguinte comando:
  - make
- Dessa forma, os arquivos com formato “.o” serão gerados na pasta **obj** e o arquivo **tp01.out** será gerado na pasta **bin**.
- Agora, navegue a até a pasta **bin** usando o comando:
  - cd bin

- Coloque seu arquivo de entrada nesta pasta, este arquivo deve estar no formato **“.txt”**.
- Com os arquivos na pasta **bin** rode o comando:
    - ./tp01.out < nomedoarquivo.txt
    - Onde o parâmetro **nomedoarquivo.txt** seria o nome do seu arquivo, acompanhado do formato dele (txt).
- Com isso, os saída do projeto será impressa no terminal,
- Caso queira utilizar outros arquivos de entrada é só substituir pelos que estão na pasta bin e rodar o comando acima, trocando o nome do arquivo no parâmetro do comando, se necessário.
- Caso ocorra qualquer problema com sua execução, entre em contato comigo pelo e-mail: caiocs2019@ufmg.br

3. Estrutura de dados e algoritmos

Explicando primeiramente as classes de abstração, temos a classe **Pessoa**, que possui atributos como ID, Idade, coordenadas X/Y e um indicador informando se está alocada em um posto ou não.
Também temos a classe **PostoVacinacao**, que semelhantemente, possui um ID e coordenadas X/Y. Também possui o número de vagas, e um atributo para a distância relativa a pessoa que estamos comparando no momento, esse atributo, **DistanciaTemp** (possui Temp na nomenclatura para indicar que é algo temporário), só é preenchido no momento que chamamos a função **CalcularDistancia**, que recebe como parâmetros a posição X/Y da pessoa que queremos comparar. Outro detalhe importante dessa função **CalcularDistancia** é que ela deveria implementar a mesma fórmula de distância euclidiana, mas por motivos de inconsistências nos resultados, não usamos a raiz quadrada (função sqrt). A classe **PostoVacinacao** também possui uma lista de IDs das pessoas alocadas nele.
Já a classe **Dados** não possui atributos, somente métodos que irão realizar manipulações nos vetores de **PostoVacinacao** e **Pessoa**. Os métodos **ProcessaEntrada**, **ProcessaPessoas** e **ProcessaPostosVacinacao** irão cuidar de realizar a leitura da entrada e preencher os vetores.

Realizamos o processo de alocação utilizando o método **AlocaPessoasPostosVacinacao**, que ordena as pessoas por idade (mais velhas primeiro), e para cada pessoa, recupera seus postos de vacinação mais próximos, e caso tenham vagas disponíveis e essa pessoa já não esteja alocada, realiza a alocação. O fato das pessoas já estarem ordenadas pela idade e já possuirmos a informação de quais são os postos mais próximos de cada pessoa, isso nos impede de já ter uma pessoa mais nova ocupando a vaga dela no posto. O algoritmo funciona da seguinte maneira:

Ordena lista de pessoas com base nas maiores idades
Enquanto** (i <= Número de pessoas) **faça
    Enquanto** (j <= Número de postos) **faça
        Registrar distância do posto[j] da pessoa[i]
    Fim Enquanto
    Ordenar lista de postos com base na distância da pessoa[i]
    Enquanto** (j <= Número de postos) **faça
        Se** (posto[j] tiver vagas disponíveis e pessoa[i] não estiver alocada) **faça
            Aloca pessoa[i] no posto[j] usando seu ID
            Altera indicador de alocação da pessoa[i] para verdadeiro
        Fim Se
    Fim Enquanto
Fim Enquanto

A complexidade assintótica de tempo para esse método é de :
Θ((i*log2(i) + (i * (j + (j * log2(j))))
onde i é o número de pessoas e j é o número de postos. É dessa forma pois eu realizo a ordenação das i pessoas utilizando o sort() e para cada pessoa, eu realizo duas iterações pelos seus postos, e entre essas iterações, eu realizo uma ordenação nos postos.

Obs: Um detalhe sobre a função sort, no terceiro parâmetro enviado para ela nós criamos funções auxiliares para determinar quais seriam as comparações a serem realizadas com base nas regras de negócio definidas: ***ComparacaoDistancia****,* ***ComparacaoId*** *e* ***ComparacaoIdade****. Além disso, de acordo com a documentação do C++, a função sort() possui complexidade de tempo de N * log2 (N), onde N é o número de elementos do array.*

Por último, nós realizamos a impressão da saída utilizando o método **ImprimirRelatorioAlocacao**:

***Ordena** lista de postos com base no seu ID*
Enquanto** (j <= Número de postos) **faça
    Imprimir ID do posto[j]
    Enquanto** (i <= Número de pessoas alocadas no posto[j]) **faça
        Se** (i == Número de pessoas alocadas no posto[j] - 1) **faça
            Imprimir ID da pessoa[i]
        Se não
            Imprimir ID da pessoa[i] concatenado com " "
        Fim Se
    Fim Enquanto
    Imprimir quebra de linha
Fim Enquanto

A complexidade assintótica de tempo para este método é de **Θ((j*log2(j) + (j * i))** onde j é o número de postos e i é o número de pessoas alocadas em cada posto. Isto segue da seguinte lógica, eu realizo uma ordenação dos j postos e para cada posto, eu realizo outra iteração nas i pessoas alocadas nele.

Logo após isso, no próprio arquivo Main.cpp, eu realizo uma iteração pela lista de postos e pela lista de pessoas limpando a memória, assim como também limpo a classe Dados, seguindo o seguinte algoritmo

Enquanto *(i <= Número de pessoas)* ***faça***
    Deletar pessoa[i]
Fim Enquanto
Enquanto *(j <= Número de postos)* ***faça***
    Deletar posto[j]
Fim Enquanto
Deletar Dados

Este procedimento possui uma complexidade assintótica de tempo de **Θ(j + i),** onde j é o número de postos e i o número de pessoas, e como nenhuma iteração está aninhada, esse cálculo acaba não escalando tanto como os anteriores.

Com isso, temos que a complexidade assintótica geral da aplicação é de
Θ((i*log2(i) + (i * (j + (j * log2(j))))
pois é a função com maior complexidade da nossa solução.