Universidade de Brasília

Disciplina: Estrutura de Dados

Semestre: 2/2018

Alunos: Artur Hugo Cunha Pereira – 18/0030400

Thiago Chaves Monteiro de Melo – 18/0055127

Projeto de Programação 01

1. Problema e solução:

O problema apresentado foi criar um sistema de priorização de atendimento para um laboratório, a fim de organizar a ordem de atendimento de seus clientes. Os atendimentos disponíveis são segmentados em cinco tipos. Cada tipo é atendido por um guichê diferente e demora um certo tempo de atendimento:

- Hemograma simples – guichê: 0, tempo: 5 unidades;
- Pré-natal – guichê: 1, tempo: 10 unidades;
- Vacinação – guichê: 2, tempo: 8 unidades;
- Hemograma completo – guichê: 3, tempo: 7 unidades;
- Entrega de resultados – guichê: 4, tempo: 2 unidades.

Além de organizar os clientes por tempo de chegada, seria necessário levar em conta o número de prioridade de cada um, de forma que alguém com maior prioridade que chegou depois fosse atendido antes de quem tivesse menor prioridade. Esse nível de preferência leva em conta a condição física e a idade do indivíduo:

- Gestante: 2 pontos;
- Deficiente: 2 pontos;
- Saudável: 1 ponto;
- Idade:
    - 80 anos ou mais: 3 pontos;
    - 65 a 79 anos: 2 pontos;
    - Menos de 65: 1 ponto.

Foi considerado que cada cliente poderia receber prioridade de apenas uma das três condições físicas. Essa prioridade é somada com a prioridade por idade para determinar a prioridade total.

A abordagem adota para resolver esse problema foi programar um *software* que simulasse um dia de atendimento no laboratório. Para isso, o primeiro passo foi abstrair a informação dos clientes e dos guichês em estruturas de dados adequadas. Foi utilizada uma estrutura de fila que guardava os registros de todos os clientes a serem atendidos no dia por ordem de chegada. Ao longo da execução do programa, um contador de tempo era incrementado e o cliente que chegasse nesse tempo era "trazido para dentro do laboratório", sendo alocado na fila do guichê adequado. A ideia foi basicamente simular a passagem do tempo, movendo os clientes entre as filas de acordo com sua necessidade de atendimento. Na seção 3, o algoritmo é explicado de maneira mais detalhada.

2. Os dados da entrada e saída:

Os dados que o programa recebe como entrada foram divididos em dois arquivos: o primeiro contendo a configuração dos guichês abertos; o segundo, as informações de cada cliente por ordem de chegada. Para detalhes sobre a execução, vide seção 5.

No primeiro arquivo, a primeira linha corresponde ao número de guichês abertos. As demais linhas apresentam apenas um dígito, referente ao serviço de cada guichê.

Já no segundo arquivo, cada linha apresenta uma série de números que codificam as informações de cada cliente. A ordem dessas informações em cada linha é a seguinte: ordem de chega, idade, serviço e condição física. Dessa forma, um cliente que chega no tempo 12, tem 23 anos, quer um hemograma simples e é saudável, seria codificado no arquivo de entrada como **12 23 0 2**.

A saída desse programa é registrada em um arquivo de saída passado como último argumento na linha de comando. A primeira linha mostra o temo de médio dos clientes e a quantidade de clientes atendidos por unidade de tempo, ambos com duas casas decimais de precisão. A linhas subsequentes apresentam informações de atendimento sobre cada cliente, seguindo a mesma ordem da entrada (temo de chegada). Cada uma dessas linhas contém o número do guichê em que o cliente foi atendido (posição do guichê no arquivo de entrada, não relacionado com o serviço do guichê), sua prioridade, seu tempo de espera e o serviço que ele usou, nessa ordem. Para exemplificar, se o cliente do parágrafo anterior tiver esperado 8 unidades de tempo e fora atendido no guichê 2, sua linha correspondente no arquivo de saída seria **2 2 8 0**.

2.1 – Como foram geradas as cargas de entrada:

As cargas de entrada para a validação do programa foram geradas por um software à parte (GeraArquivos.c), que gera clientes aleatório com tempos de chegada que distam um valor aleatório de 0 a 5 entre si. Esse software também permite gerar horários de pico, diminuindo essa distância entre tempos de chegada para de 0 a 2 durante o intervalo de tempo escolhido para o pico.

3. O programa:

3.1 – Módulos do programa:

O programa foi modularizado em quatro partes: o programa principal que que é executado (main.c), as funções de leitura de entrada e geração de saída (inout.c), a lógica do simulador que é chamado no programa principal (simulador.c), e as funções que atuam sobre as estruturas de fila e de clientes (fila.c e clientes.c).

Quanto à interdependência dos módulos, o módulo de clientes chama funções do módulo de fila, enquanto o módulo de entrada e saída chama funções de fila e de clientes. Já o simulador chama apenas funções de fila. O programa principal chama o simulador e funções do módulo de fila.

3.2 – TADs, estruturas de dados e funções:

Os tipos abstratos de dados e estruturas que foram definidos para a abstração do problema em questão foram os seguintes:

- TAD Fila;

- TAD Cliente;
- Estrutura para guichê;
- Estrutura de lista;
- Estrutura para informações gerais do simulador.

Essa estrutura para informações gerais foi usada para registrar dados como tempo médio de espera e número de clientes atendidos por unidade de tempo. A estrutura do guichê contém seu número, o número do seu serviço e o tempo em que começou o último atendimento desse guichê.

O TAD de cliente é composto pela estrutura de cliente, que armazena:

- Tempo de chegada;
- Tempo de início do atendimento;
- Tempo de espera;
- Prioridade;
- Serviço desejado;
- Número do guichê para onde foi encaminhado;
- Número do cliente na ordem do arquivo de entrada.

As funções que atuam sobre essa estrutura se encontram no módulo clientes.c e são as seguintes:

- CriaCliente: aloca espaço para um novo ciente;
- PassaDados: recebe as informações da estrutura de clientes e escreve no espaço alocado por CriaCliente;
- CalculaTempodeEsperaMedio: recebe uma fila de clientes e calculado o tempo médio de espera desses clientes.

O TAD de fila consiste da estrutura de fila, que possui um tamanho, uma célula *tail*, que indica o fim da fila (onde os elementos novos são adicionados) e uma célula *head*, que indica o início da fila (de onde os elementos são tirados). Todos os elementos da lista, ou células, são estruturas do tipo lista, que possuem um ponteiro para o próximo elemento e um ponteiro para onde a informação contida nessa célula está escrita na memória. O diagrama que segue ilustra essa estrutura, sendo que cada seta é o ponteiro para o próximo elemento (cada célula ainda teria um ponteiro para a sua informação, mas isso foi omitido para não poluir visualmente a ilustração):

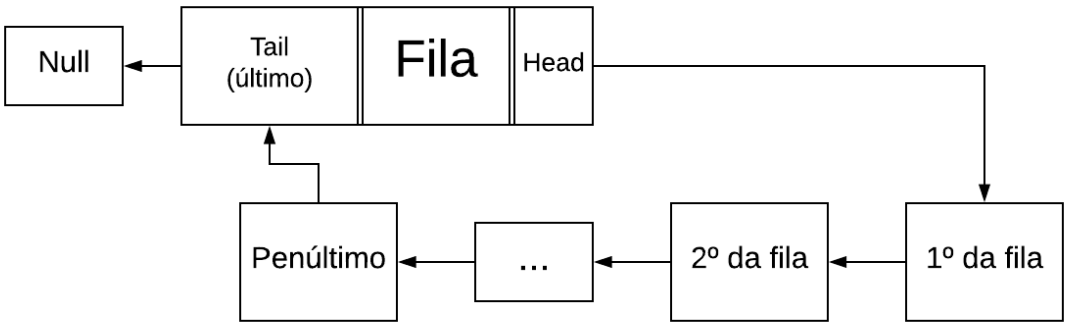

Figura 1, ilustração da estrutura de fila.

Vale observar que o último (*tail*) aponta para o endereço zero da memória, pois não existe ninguém depois do último da fila. Além disso, o primeiro da fila não é a *head*, e sim o seu próximo elemento.

As funções que atuam sobre as filas são:

- CriaFilaVazia: aloca espaço para uma fila vazia, com os ponteiros de *head* e *tail* apontando para o mesmo lugar;
- InserirNaFila: insere um elemento na fila e atualiza a posição do *tail*;
- TiraElementoDaFila: remove o primeiro elemento e atualiza a posição do *head*;
- FVazia: verifica se a fila está vazia;
- FreeFila: desaloja a memória alocada para uma fila;
- CriaFilas: cria um vetor com um número específico de filas;
- SeparaGuiches: separa os guichês lidos na entrada em um vetor com uma fila para cada serviço;
- InsereClientePrioridade: insere o cliente na fila do serviço adequado, colocando-o na posição adequada de acordo com sua prioridade;
- InsereClienteNum: insere o cliente na fila de clientes atendidos, seguindo a ordem do arquivo de entrada.

Por último, o módulo inout.c contém as funções que leem os arquivos de entrada dos guichês e dos clientes e a função que escreve o arquivo de saída, respectivamente:

- LeDadosGuiches;
- LeDadosClientes;
- EscreveRelatório.

O módulo simulador.c possui apenas uma função, que será tratada com maior detalhe na seção 3.3.

3.3 – O algoritmo do simulador:

O simulador consiste de um laço principal (linha 43) que itera até que não haja mais clientes nessa fila e não haja mais cliente sendo atendidos. A cada execução do laço principal, um contador de tempo é incrementado. A função do simulador retorna um ponteiro para a fila de clientes já atendidos.

Para a lógica do simulador, a estrutura de fila foi utilizada para organizar as demais estruturas. Ao ler o arquivo dos clientes, as informações são armazenadas em uma fila de clientes chegando no laboratório, que é passada como parâmetro do simulador. Ao ler o arquivo de guichês, o mesmo procedimento é feito e a fila resultante também é passada para o simulador. Todas essas passagens são feitas por referência, tal como a passagem do terceiro parâmetro: uma variável que receberá, ao fim da função, o valor final do contador de tempo.

Antes do laço principal do simulador, são alocados três vetores de cinco filas: um deles representando as filas de espera para cada serviço, outro representando os guichês livres para cada tipo de serviço, e outro para os guichês ocupados.

O primeiro laço interno (linha 47) que aparece dentro do principal verifica se o primeiro cliente da fila de clientes chegando tem o tempo de chegada igual ao do contador. Caso tenha, esse cliente “entra no laboratório”, sendo removido da fila inicial. Então esse laço itera novamente vendo todo os clientes que chegaram no mesmo horário. Ao não encontrar um (o próximo cliente chegou depois ou a fila está vazia), o laço é encerrado. Quando um cliente é removido da fila de chegada, ele é imediatamente alocado em uma das cinco filas do vetor de clientes dentro do laboratório, de acordo com o serviço desejado e levando em conta sua prioridade.

Em seguida, há um laço (linha 58) que itera uma vez para cada tipo de serviço (nesse caso, cinco vezes). Dentro dele são encontrados outros dois laços (linhas 60 e 69). O primeiro deles passa pela fila de guichês ocupados do respectivo serviço, verificando os guichês que foram liberados. A lógica desse laço é mesma do primeiro laço interno: ele para ao encontrar um guichê que não foi liberado quando a fila estiver vazia; quando um guichê é liberado, ele é tirado dessa fila e colocado na fila de guichês livres. Já o segundo laço, itera enquanto houver algum guichê livre para o dado serviço. Se houver algum cliente na fila desse serviço, o primeiro da fila é atendido, ou seja, removido da fila de dentro do laboratório e colocado na fila de clientes já atendidos (retorno da função). As informações desse cliente atendido são então atualizadas, registrando o tempo de início de seu atendimento e o número do guichê que o atendeu. O guichê que agora está em uso passa a registrar também o tempo do início desse atendimento e é trocado para a fila de guichês ocupados.

Por fim, há um pequeno laço (linha 87) que passa sobre o vetor de filas de guichês ocupados verificando se há alguma que não está vazia. Dessa forma, pode-se saber se ainda há algum guichê ocupado e, consequentemente, algum cliente sendo atendido (uma das condições do laço principal).

Para ilustrar de uma maneira mais simplificada todo esse processo do laço principal, segue a ilustração em fluxograma:

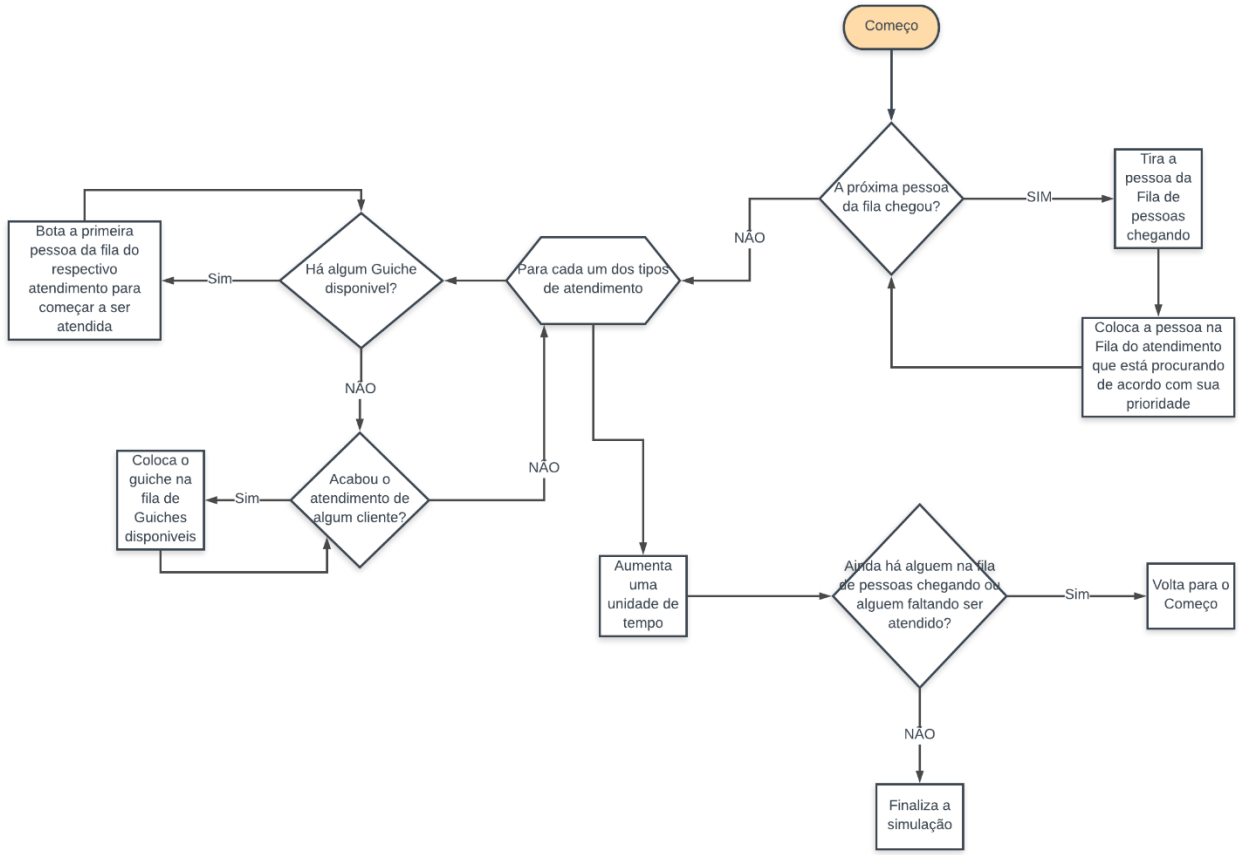

Figura 2, fluxograma do simulador.

3.4 – A complexidade:

Para a análise de complexidade do programa, foi analisada inicialmente a complexidade de cada função.

Começando pelo módulo de entrada e saída, temos que as três funções presentes são **O(n)**, com **n** correspondendo ao número de linhas do arquivo (para as funções de leitura) ou o número de elementos na fila recebida como parâmetro (para a função de escrever o relatório de saída).

Para o módulo de filas, a maioria realiza um número constante de ações significativas, como chamar funções para alocar ou desalojar espaço na memória, no caso das funções CriaFilaVazia, InserirNaFila, TiraElementoDaFila, ou uma simples comparação, como no caso de FVazia. Essas funções são, portanto, **O(1)**.

Há ainda, nesse mesmo módulo, as funções lineares: CriaFilas, que chama CriaFilaVazia para cada fila a ser criada; SeparaGuiches, que chama CriaFilas e ainda executado outro laço de complexidade O(n); InsereClientePrioridade e InsereClienteNum, que, no pior caso, precisam percorrer a fila toda para inserir cliente no lugar certo; e FreeFila, que percorre uma fila inteira dando *free* em cada elemento. Esse grupo de funções **O(n)** fecha o módulo de filas.

No módulo de cliente, há apenas três funções. A função de criar cliente realiza apenas uma alocação de memória, enquanto a PassaDados realiza uma série de atribuições. Portanto, essas duas funções são **O(1)**. Já a função de calcular o tempo de espera médio percorre um fila inteira para pegar o tempo de espera de cada cliente para fazer a média, o que faz dela **O(n)**, com **n** sendo o número de clientes (tamanho da fila).

Com esses três módulos analisados, resta apenas determinar a complexidade do simulador para concluir a complexidade do programa todo. Fora do laço principal, ele chama funções lineares do módulo de fila: CriaFilas (linhas 22 e 23) e SeparaGuiches (linha 20). Porém, como o número de filas passado como parâmetro de CriaFilas é constante, essa função passa a ser O(1). Há também, no fim da função, um laço que itera esse mesmo número de vez e que chama FreeFila três vezes. Tem-se, até então, uma complexidade O(n) fora do *loop* principal.

Para analisar a complexidade do laço principal, serão levadas em conta as chamadas de funções lineares. Uma maneira mais prática para calcular essa complexidade é ver quantas vezes essas funções serão chamadas no total, ao invés de tentar estimar quantas vezes o *loop* aconteceria e quantas vezes as funções seriam chamadas por ocorrência do laço. Para tal, é preciso analisar cada laço interno.

Para o primeiro laço (linha 47), a função de inserir cliente por prioridade será chamada uma vez para cada cliente (ou seja, n vezes), pois caso o primeiro da fila ainda não tenha chegado, o laço é encerrado. Portanto esse laço contribui com $n*O(n) = O(n^2)$.

Dentro do laço *for* (linha 58) temos dois outros laços. Como o *for* acontece um número constante de vezes, ele não vem a influenciar na análise assintótica da complexidade, portanto serão analisados os seus laços internos de maneira independente. O primeiro (linha 60), chama duas funções por cliente, seguindo a mesma ideia do parágrafo anterior. Porém, essas funções chamadas são O(1), fazendo com que ele contribua com $n*O(1) = O(n)$. Já o segundo (linha 69) fará chamadas de funções caso haja algum cliente esperando nas filas de dentro do laboratório e a cada vez que essa condição é satisfeita, o cliente atendido é removido da fila. Dessa forma, a condição é satisfeita n vezes. Como ele chama uma função linear (InsereClientesNum), sua contribuição para a complexidade do laço principal é $n*O(n) = O(n^2)$.

Por fim, há um último *loop* (linha 87) que repete um número constante de vezes e não faz chamada de nenhuma função linear. Logo, ele não influi no comportamento assintótico da complexidade em análise.

Somando todas as complexidades discutidas dos laços internos, tem-se que a complexidade do laço principal é $O(n^2) + O(n^2) + O(n) = O(n^2)$. Somando essa complexidade à complexidade externa ao laço, conclui-se que a complexidade do simulador é $O(n) + O(n^2) = \mathbf{O(n^2)}$.

Com base nesse resultado e sabendo que o programa principal (main.c) faz chamadas independentes de diversas funções de diversos módulo, pode-se concluir que o programa tem uma complexidade $\mathbf{O(n^2)}$.

4. Avaliando algoritmo:

Com o intuito de facilitar a compreensão e deixar as explicações menos confusas, todos os exemplos de entrada e saída que serão citados estão na pasta “testes”, que foi entregue junto ao relatório, e as referências feitas a eles terão o mesmo nome.

Primeiramente para conferirmos que a lógica do simulador está funcionando como foi previsto, foi utilizado o Clientes1.txt como carga de trabalho, juntamente com o Guiches1.txt para a configuração dos guichês. Utilizando essa carga pequena juntamente com só um guichê de cada tipo, é possível comprovar que a lógica feita para atender de acordo com as prioridades está correta. Ao analisar a distribuição dos pacientes que estão chegando, é possível ver que o terceiro na lista de chegada possui uma prioridade maior em relação a pessoa que chegou imediatamente antes dele. Tendo em mente que ambos estão a procura do mesmo atendimento e que os dois não podem ser atendidos imediatamente ao chegarem pois o guichê está ocupado pelo primeiro paciente listado no arquivo, o paciente com maior prioridade deve ter menos tempo de espera que os demais. Esse resultado esperado foi relatado ao observar o relatório de saída referente a essa configuração de entrada, Relatorio1.txt, onde a quarta pessoa, a com maior prioridade entre as primeiras a chegar e que não foram imediatamente atendidas, é a com menor tempo de espera.

Agora, para mostrar a eficiência do algoritmo ao adicionar mais guichês, foi utilizado novamente a carga de trabalho anterior, Clientes1.txt, juntamente com uma nova configuração de guichês, Guiches2.txt. Com isso, a única diferença do citado anteriormente foi o acréscimo de um guichê do atendimento de tipo 0, que é a maior demanda nessa carga de trabalho. Comparando a diferença do tempo de espera médio anterior no Relatorio1.txt com o novo tempo de espera médio, no Relatorio2.txt, é possível ver como esse pequeno acréscimo foi muito significativo. Houve uma diminuição de 68% do tempo médio de espera e um aumento significativo na quantidade de clientes atendidos por unidade de tempo. Outro fato importante a ser ressaltado neste teste é que os maiores tempos de espera, antes de 14 e 13, caíram para somente 0 e 3, respectivamente. Considerando os fatos mostrados, é possível concluir que o algoritmo consegue responder de maneira eficaz a uma quantidade variada de guichês, tornando o tempo de espera o menor possível, e sempre assegurando as prioridades então sendo levadas em conta.

Além disso, para confirmar a veracidade da complexidade do programa, foram recolhidas amostras do tempo de execução do programa para quantidades variadas de

clientes, deixando fixo a configuração dos guichês como no Guiches1.txt, para que sempre esteja havendo um pior caso e os resultados possam ser comparadas de maneira igual. Com os dados obtidos, foi feito um gráfico da relação quantidade de clientes pelo tempo, em segundos, de execução:

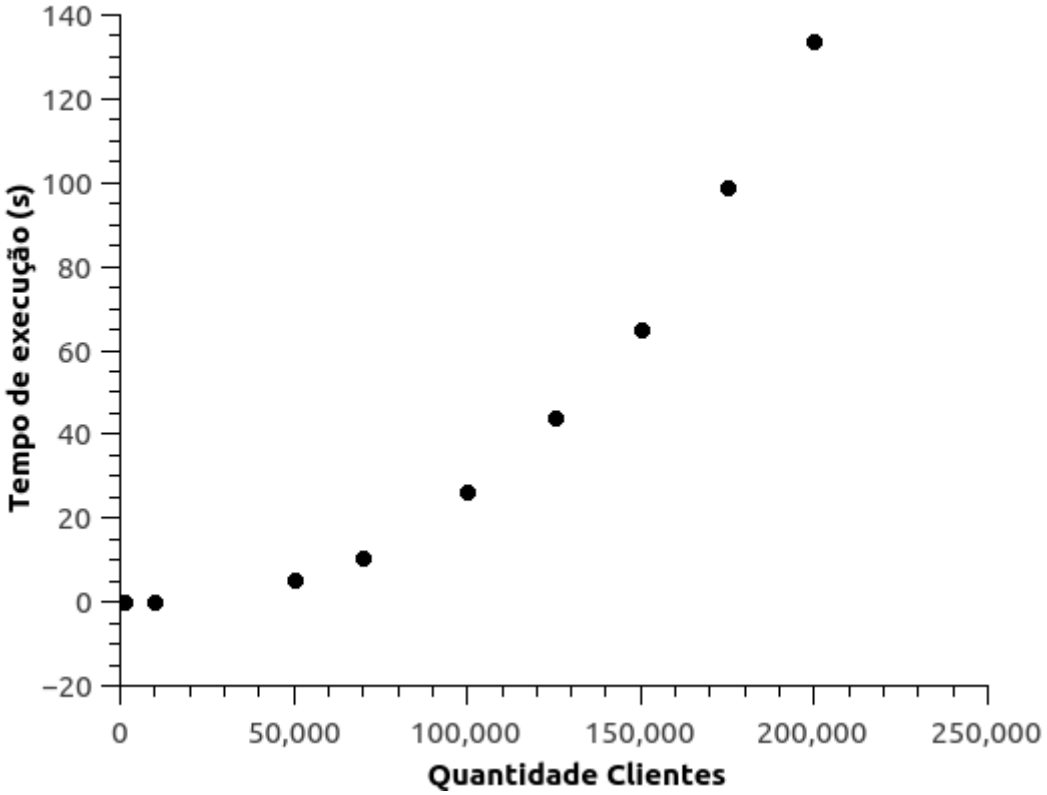

Figura 3, gráfico: Tempo X Quantidade de Clientes.

Analisando a curva descrita no gráfico acima, é possível ver que foi formado uma curva com formato parabólico. Com o auxílio do software Qtiplot, foram feitos diversos ajustes de curvas para os dados plotados. Assim, foi descoberto que o melhor ajuste é dado pela curva formada por um polinômio de grau 2. Dessa forma pôde ser verificada a complexidade do programa com dados experimentais, sendo ela realmente **O(n²)**.

Para uma análise de um problema mais real foi utilizado o Clientes2.txt como carga de clientes. Nesse exemplo, é visto um claro horário de pico no instante 134, onde 35 pessoas chegam ao mesmo tempo na clínica, começando da pessoa de índice 49. Primeiramente foi feita a saída usando o Guiche1.txt que representa o pior caso da configuração dos guichês. Ao ser feito a execução, foi obtido o Relatorio3.txt, em que é possível verificar aumentos nos tempos de espera dos pacientes a partir da pessoa de índice 49 (situado na linha 50). Um fato interessante para ser ressaltado é os elevados tempos de espera acabaram se propagando até a última pessoa a ser atendida após o horário de pico. Assim, é possível verificar como em situações reais, um horário de pico, além de aumentar a insatisfação dos pacientes nesse próprio horário, pode afetar as próximas pessoas que estão chegando fora dessa faixa de tempo.

Utilizando essa mesma carga de clientes, agora foi adicionado mais um guichê de cada tipo, Guiches3.txt, totalizando 10 guichês. Tendo feito isso, foi gerado o

Relatorio4.txt. Nesse, é possível ver que apesar de ainda existirem alguns tempos de espera altos durante o horário de pico, o problema não mais é propagado para fora desse. Vale constatar que houve uma queda de 71% no tempo de espera médio e ainda que os tempos de espera das últimas pessoas nos arquivos saíram de 20 e 23 para 0 e 0.

Vendo isso, conseguimos constatar problemas presentes em laboratórios reais: horários de pico criando tempos de espera absurdos para todos os pacientes que chegam durante e após o termino do mesmo. Para esse tipo de situação, foi comprovado, por meio desses testes que uma boa solução é criar mais guichês de atendimento nos horários específicos de pico para que não haja essa propagação de tempo de espera falada anteriormente.

Continuando com as analises, foram testados casos extremos, para analisar como o programa se comporta diante deles.

Foi utilizado o Clientes3.txt, juntamente com Guiches4.txt para simular uma situação onde há um número muito maior de guichês comparado ao número de clientes total. Nesse teste criado existem 30 pacientes, chegando em horários distintos e 50 guichês distribuídos de forma aleatória entre os tipos de atendimento. Ao gerar a saída dessa combinação no Relatorio5.txt, obteve-se o que era esperado, não houve tempo de espera algum para nenhum cliente. Esse tipo de exemplo é importante para ressaltar a funcionalidade do programa mesmo com uma grande quantidade de guichês sendo dada para utilizar.

O próximo exemplo de caso extremo foi feito utilizando uma carga que está toda dentro de um horário de pico. Para isso, foi utilizado o Clientes4.txt como carga de trabalho onde foram colocados 1000 clientes todas chegando no instante 4 e Guiches1.txt como configuração dos guichês para que pudesse ser feita a análise do pior caso. Com isso foi obtido o Relatorio6.txt, e nele é possível ver claramente que todos os menores tempos de espera são pertencentes às pessoas com maior prioridade. Estando no inicio ou no final do arquivo de carga, as pessoas com menor prioridade são as que possuem os tempos mais altos, chegando até a 1900 unidades de tempo para algumas que se situavam no fim do arquivo com a menor prioridade possível. Tal fato comprova mais uma vez a funcionalidade do algoritmo para atender em ordem certa os clientes de acordo com suas prioridades.

Por último, foi feito uma mistura dos últimos 2 exemplos. O objetivo foi entender como o programa consegue lidar com uma quantidade muito grande de guichês e muitas pessoas chegando em um horário igual. Para isso foi utilizado o Clientes4.txt junto ao Guiches4.txt. Após gerar o Relatorio7.txt como saída dessa combinação foram constatados alguns pontos importantes. Em relação ao do Relatorio6.txt, onde existia apenas um guichê de cada tipo, o tempo de espera médio diminuiu em cerca de 10 vezes, e a quantidade de pacientes atendidos por unidade de tempo aumentou em aproximadamente 9 vezes. Além disso, nenhum tempo de espera passa de 200 unidades de tempo.

Vendo os dados analisados, podemos certamente concluir que o algoritmo empregado para a construção do simulador é tanto eficiente como também eficaz para o manuseio das filas de cada tipo de atendimento. Conseguir encaixar todas as pessoas para serem atendidas da forma mais rápida possível sem que haja desrespeito com suas respectivas prioridades são uma das principais qualidades que foram procuradas, e alcançadas conforme mostrado.

5. Execução do software:

Como o programa todo foi desenvolvido utilizando a linguagem C, antes da sua execução ele necessita que haja a compilação do mesmo. Para isso, estando com o terminal no mesmo diretório que o código fonte, digite:

> make

Isso fará com que seja criado vários arquivos com extensão .o e o executável "sim_senhas". Para a utilização correta do mesmo, é necessário que na linha de comando do terminal seja digitado:

>./sim_senhas <arquivo_configuracao> <arquivo_carga_trabalho> <arquivo_saída>

Note que as palavras entre <> devem ser substituídas pelos arquivos .txt que serão lidos e escritos na execução.

Por último, caso seja necessário a criação de novos arquivos de entrada, no terminal de o comando:

>make GeraArquivos

Isso criará um executável chamado Teladeclientes que facilita interativamente a criação de distribuições de entradas de clientes e guichês.