Universidade Federal de São João del-Rei
Algoritmo e Estrutura de Dados III
Trabalho Prático 1

Felipe Francisco
Thales Mrad

1. Introdução

A agilidade dos processos é algo imprescindível dentro das empresas. O grande volume de informações, a necessidade de realizar multitarefas e a exigência por inovação no mercado fazem com que os colaboradores tenham que lidar com diversas funções. No entanto, algumas atividades podem — e devem — ser otimizadas por meio da automação de serviços.

Além de promover uma melhora crescente da performance da organização, a implementação de boas tecnologias traz uma série de outras vantagens.

A automação de serviços nada mais é do que promover a implementação de recursos tecnológicos em um negócio. Ou seja, é a utilização de ferramentas e soluções tecnológicas com o intuito de simplificar, agilizar os processos internos e reduzir seus custos, bem como lidar com a grande demanda de dados e manipulá-los com precisão.

Assim, este presente trabalho prático consiste na implementação de um programa que realize o planejamento das manobras de um estacionamento para um empresa fictícia chamada TremDaFolia, tendo em vista maximizar a ocupação do estacionamento e minimizar o tempo de espera dos clientes para sair do estacionamento.

Dada a implementação do programa este trabalho tem, também, como objetivo analisar o desempenho do algoritmo em diferentes cenários, considerando as seguintes métricas de desempenho:

- análise da complexidade das rotinas;
- análise dos tempos de computação e tempos de entrada e saída.

2. Implementação

Premissas iniciais:

- O estacionamento é quadrado e tem dimensões 6 por 6;
- Os veículos não fazem curvas;

- O estacionamento comporta dois tipos de veículos: carros (2x1) e caminhões (3x1);
- Arquivo configuração inicial do estacionamento: identificador, tamanho, direção e posição;
- Arquivo de manobras: identificador do veículo, a dimensão onde o movimento ocorre e a amplitude do movimento;
- Detectar e alertar para configurações físicas impossíveis e manobras inviáveis.
- Objetivo: mover o veículo Z, para a saída do estacionamento.

Estrutura de dados:

O algoritmo foi implementado com quatro módulos, sendo eles:

Módulo 1: trata-se dos procedimentos de entrada e saída;
Módulo 2: contém a lógica de configuração do estacionamento;
Módulo 3: contém a lógica de manobras;
Módulo 4: contém o programa principal.

O módulo 1, por tratar dos procedimentos de entrada e saída, interage com todos os outros módulos, e é nele que são declaradas as *structs Auto* (especificação das características do veículo), *Movimento* (especificação das configurações de cada manobra) *e Arquivos* (contém os ponteiros para o arquivo de configuração inicial e manobras).

O mapa do estacionamento foi implementado como uma matriz de 6 colunas e 6 linhas, onde que nas posições retratadas pelo número 1 possuem veículos, enquanto que nas posições que contém o número 0 estão disponíveis/vagas.

Estacionamento sem nenhum veículo:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Estacionamento com quatro veículos dispostos arbitrariamente:

0	**1**	0	**1**	**1**	**1**
0	**1**	0	0	0	0
0	**1**	0	0	**1**	**1**
0	0	0	0	0	0
0	0	**1**	0	0	0
0	0	**1**	0	0	0

Funções e Procedimentos

As funções implementadas para serem utilizadas pelo programa principal são: (a indentação representa a hierarquia das funções e suas sub-funções)

void contaTempoProcessador(double *utime, double *stime);
Recebe dois ponteiros para variáveis do tipo double, que iram marcar o tempo atual de usuário e em modo de sistema, essa função utiliza o comando *getrusage()*. A função retorna os recursos correntemente utilizados pelo processo, em particular os tempos de processamento (tempo de CPU) em modo de usuário e em modo sistema, fornecendo valores com granularidades de segundos e microsegundos.

Arquivos* argumentosEntrada(int argc, char* argv[]);
Recebe os argumentos pela linha de comando, e através da função *getopt(argc, argv, "m:v")* abre os arquivos de configuração inicial (-v) e o arquivo de manobra (-m).

int criaMapa();**
Cria uma matriz 6x6 que será usada para realizar toda a lógica de configuração inicial e movimentação de veículos, e inicializa cada elemento como 0.

Auto* leituraConfigInicial(FILE* arq_veiculos);
Recebe o ponteiro do arquivo de configuração inicial, para que primeiramente, seja calculado a quantidade de veículos pela função *calculaQtdVeiculos()*, é alocado dinamicamente uma variável da *struct Auto* de tamanho *n,* onde *n* é a quantidade de veículos. Em seguida usa-se o *rewind()*, para apontar novamente para o início do arquivo para que ele possa ser percorrido preenchendo as características de cada veículo, este trabalho é realizado pela função *setVeiculo().*

> **int calculaQtdVeiculos(FILE* arq_veiculos);**
> Recebe o ponteiro do arquivo de configuração inicial, onde que enquanto não chegar no fim do arquivo, se houve um '\n' ou atingir o fim do arquivo a quantidade de veículos é incrementada. A função retorna um inteiro que corresponde à quantidade de veículos.
>
> **void setVeiculo(int qtdVeiculos, FILE *arq_veiculos, Auto *veiculo);**
> Recebe a quantidade de veiculos, o ponteiro para o arquivo de configuração inicial e o vetor da *struct Auto*, a função é responsável por ler todo o conteúdo do arquivo e armazenar no vetor as características de cada carro (ID, tamanho, direção, posição X e posição Y).

int configInicialMapa(int qtdVeiculos, Auto* veiculo, int **mapa);
Recebe a quantidade de veiculos, o vetor contendo os veículos, e o mapa. Nesta função todos os veículos são colocados em suas respectivas posições, mas sempre antes

verificando na função *verificaEspacoMapa()* se a posição requerida está disponível, caso não esteja retorna-se 0 para o programa principal.

int verificaEspacoMapa(Auto veiculo, int **mapa);
Recebe um veículo, e o mapa do estacionamento. Primeiramente verifica se a direção do veículo é X ou Y, então é checado se a posição dada pelo arquivo de configuração inicial e a seguinte a ela está vazia, isto, quando o veículo é um carro, quando é um caminhão é checado, ainda, a próxima posição na direção correspondente ao veículo. O algoritmo retorna 0 caso esteja ocupada a posição e 1 caso esteja disponível.

int leituraExecucaoManobra(Auto *veiculo, int **mapa, FILE* arq_manobras);
A primeira parte desta função, é responsável por percorrer todo o arquivo de manobras, onde que enquanto não estiver no seu fim, a sub-função *realizaManobra()* é chamada lendo cada manobra individualmente e realizando a movimentação do veículo. A segunda parte da função é pegar o inteiro retornado pela *realizaManobra()* e verificar seu valor, se for diferente de 1, significa que o algoritmo não obteve sucesso em realizar a manobra, neste caso a manobra pode ter sido inviável por motivos de colisão ou o arquivo de manobras apresentava incompatibilidade com o arquivo de configuração. Por fim, na terceira parte é checado se o veículo Z chegou até a saída do estacionamento, por meio da sub-função *verificaSaidaZ()*.

int realizaManobra(int qtdVeiculos, Auto *veiculo, Movimento manobra, int **mapa);
Em um *loop*, que passa por todos os veículos, se a ID do veículo em questão for igual à ID da manobra, a função chama a *verificaTrajeto()* para analisar se o percurso a ser feito pelo veículo está livre. Caso não esteja, a função r*ealizaManobra()* termina retornando 2 (colisão com o muro) ou 3 (colisão com outro veículo) para leituraExecucaoManobra(). Por outro lado, caso esteja livre, a sub-função *apagaPosAnterior()* é acionada apagando a posição atual do veículo para que em seguida a *movimentaVeiculo()* realize a movimentação para a posição futura.

int verificaTrajeto(Auto veiculo, int **mapa, Movimento manobra);

Primeiro a função verifica a direção do veículo e da manobra, se forem iguais, a trajetória que o automóvel possivelmente fará é uma linha. Caso as direções forem diferentes, a trajetória será uma submatriz da matriz do estacionamento. É atribuído a variável inicio (do tipo inteiro) a posição imediatamente após o veículo ou imediatamente anterior ao veículo (caso o sinal da manobra seja '+' a manobra seguirá sentido positivo crescente da matriz o começo do trajeto será o espaço após, se for negativo seguirá o sentido negativo e o começo do trajeto será o espaço anterior ao veículo) e atribuído a variável final (também do tipo inteiro) o espaço correspondente ao lugar que será ocupado pelo automóvel após a realização da manobra. É chamada a função *verificaColisaoParede()* para checar se a manobra saíria dos limites do estacionamento, se

transpassasse a função termina e retorna 2. Por último são chamadas às funções *verificaLinhaLivre()* ou *verificaMatrizLivre()* para checar se existe outro automóvel no caminho do veículo que está executando a manobra. Se existir, retorna 3 (caso de colisão com outro veículo). Se o trajeto estiver limpo, a função termina e retorna o valor 1, indicando que a manobra pode ser efetuada.

void apagaPosAnterior(Auto veiculo, int **mapa);
Recebe o veículo que realizará a movimentação e atribui 0 à sua posição atual na matriz.

int movimentaVeiculo(Auto *veiculo, int **mapa, Movimento manobra);
A função recebe um ponteiro da estrutura do veículo e modifica nela a posição inicial do mesmo através da adição da amplitude do movimento à coordenada correspondente a direção da manobra.

int verificaSaidaZ(Auto *veiculo, int qtdVeiculos);
Esta função procura pelo automóvel identificado como Z entre todos os veículos, encontrado o veículo, o algoritmo precisa reconhecer se ele é um carro ou caminhão e se está na direção X ou Y, para poder aplicar a lógica correta. Esta lógica consiste em analisar se as posições ocupadas pelo veículo Z é uma das que dão acesso à saída do estacionamento, se sim, é retornado 1.

void imprimeTempo(double user_time, double system_time);
Recebe a diferença entre tempo de usuário final e inicial e a diferença entre tempo de sistema final e inicial, todos estes tempos são calculados pela função *contaTempo(),* para que possam ser imprimidos por esta função.

void imprimeMapa(int **mapa);
Imprime o mapa do estacionamento.

Programa Principal

A função main é executada recebendo os parâmetros passados via linha de comando:

int main(int argc, char *argv[]), Onde:

argc – é um valor inteiro que indica a quantidade de argumentos que foram passados ao chamar o programa.
argv – é um vetor de char que contém os argumentos, um para cada string passada na linha de comando (estas strings passadas pela linha de comando, correspondem ao nome dos arquivos que serão manipulados pelo programa).

Em seguida é marcado o tempo inicial, para no fim ser subtraído do tempo final e ser possível conhecer o tempo gasto de execução.

Marcado o tempo inicial, os argumentos recebidos por linha de comando são reconhecidos como arquivos de leitura e são abertos.

É criado o mapa do estacionamento, onde serão dispostos os veículos e serão manipuladas todas as manobras.

O arquivo de configuração inicial, que contém as especificações de cada automóvel é lido, e os veículos são armazenados em um ponteiro da struct Auto.

Em seguida, o programa passa por uma estrutura condicional, que basicamente checa se houve problemas no posicionamento dos carros no estacionamento ou se o arquivo de manobras está vazio, se acusar verdadeiro para alguma dessas situações o programa termina aí. Não tem mais o que fazer, pois se os veículos não estão posicionados corretamente no estacionamento algo está errado no arquivo de configuração inicial. Por outro lado, se o arquivo de manobra estiver vazio, o programa perde o objetivo que é o de realizar manobras e checar se o veículo chegou até à saída.

Caso o esteja tudo certo, continuamos e o próximo passo é a leitura e simultaneamente a execução das manobras.

Marcamos novamente o tempo, agora, simbolicamente intitulado tempo final. Pegamos o tempo final, subtraímos do tempo inicial e temos o tempo de execução do programa. Importante salientar que este não é o tempo de relógio, é o tempo de processamento calculado pela CPU.

Por fim, imprime-se o mapa do estacionamento, após realizadas todas as operações de manobras, tenha o veículo chegado até o destino ou não.

Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em sete arquivos principais: seis deles implementam o TAD enquanto o arquivo main.c implementa o programa principal.

O compilador utilizado foi o GCC (versão: Debian 7.3.0-19) no sistema operacional Linux. Para executar o programa, primeiramente deve compilá-lo digitando *make* a partir da linha de comando e depois: ./tp -m [arquivo de manobra] -v [arquivo de configuração inicial].

3. Análise de complexidade

Análise de complexidade das principais funções:

- **criaMapa()**

A função possui 2 laços aninhados que vão de 0 até SIZEMAP (que é uma constante de valor 6), sua ordem de complexidade seria O($SIZEMAP^2$). Como SIZEMAP é constante, independente de ser elevada ao quadrado, não deixa de ser uma constante, logo a ordem de complexidade é O(1).

- **leituraConfigInicial()**

De acordo com a ordem de complexidade de suas sub-funções calculaQtdVeiculos() e setVeiculo(), temos O(n) e O(n), pela regra da soma temos O(max(n, n), logo a ordem de complexidade geral é O(n).

➢ **calculaQtdVeiculos()**

Percorre um arquivo de tamanho n até o fim, todas as outras instruções internas levam tempo constante para serem executadas, tendo assim sua ordem de complexidade em todos os casos O(n).

➢ **setVeiculo()**

A função possui um laço que vai de 0 à uma quantidade de veículos *n, O(n),* dentro deste laço a outro, porém diferentemente do primeiro, este laço vai do 0 ao fim de uma linha do arquivo, as linhas deste arquivo seguem um padrão de string do tipo [id][espaço][direção][espaço][amplitude] e por isso não passam de 5 caracteres, tornando-o O(1). Dentro destes dois laços aninhados temos apenas estruturas condicionais e atribuições. O comando de atribuição leva um tempo constante para ser executado, bem como a na avaliação da decisão na estrutura condicional. Sendo assim O(n x 1) = O(n), conforme a regra do produto para notação O.

- **configInicialMapa()**

Esta função percorre todos os veículos, em um laço que vai de 0 até uma quantidade *n-1* de veículos, onde que dentro do laço há apenas instruções de atribuições e comparações, assim temos O(max(n, 1, …, 1) = O(n), no pior caso. Contudo é O(1), no melhor caso, que ocorre quando na primeira iteração do laço, o algoritmo detecta uma configuração física impossível através da *verificaEspacoMapa(),* e a função termina.

- **leituraExecucaoManobra()**

Essa função em particular se divide em várias sub-funções com complexidades variadas. A maioria delas tem ordem de complexidade O(1), pois fazem operações baseadas no tamanho dos veículos, na amplitude das manobras ou no tamanho do estacionamento que são todos constantes. As outras funções dependem da quantidade de veículos (n), e ainda há uma passagem pela quantidade de manobras no arquivo (m) que são variáveis. A função tem 3 “escopos”, o primeiro que é um while de complexidade O(m*n), a parte intermediária

de comparações e atribuições O(1) e a verificaSaidaZ() que é O(n). Então a complexidade da função leituraExecucaoManobra() é O(max(mn,1,n)) que é O (m*n), conforme a regra da soma para notação O.

- **imprimeMapa()**

Segue a mesma complexidade da função criaMapa: possui 2 laços aninhados que vão de 0 até SIZEMAP (que é uma constante de valor 6), sua ordem de complexidade seria $O(SIZEMAP^2)$. Então sua complexidade também é O(1).

- **Programa Principal**

O programa principal apenas chama as funções analisadas acima e algumas outras funções O(1) não citadas no tópico de análise de complexidade, sendo assim, a complexidade geral do programa, no pior caso, é a soma de todas as complexidades O(max(1, 1, 1, n, n, m*n, 1, 1, 1) = O(m*n). Entretanto, o *n* é limitado, pois há um limite de veículos que é possível colocar em um espaço finito, enquanto que o *m* (quantidade de manobras) não é limitado, pensando em um volume muito grande de dados de entrada, temos que o algoritmo possui ordem assintótica O(m).

4. Testes

O algoritmo foi testado para diversas entradas diferentes, com situações que levam o programa a ter desfechos distintos, e seu funcionamento correspondeu ao esperado. Os testes foram realizados em um Intel Core i5 7ª geração i5-7200U, com 8gb de memória RAM.

Abaixo está alguns testes realizados, separados por tabelas, cada uma representando um direcionamento que o programa pode tomar dependendo dos arquivos de entrada:

TABELA 1. Veiculo Z chegando até a saída:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 3 Y X1Y3 Y 2 Y X6Y1 X 3 Y X3Y1 W 2 Y X5Y2 V 2 X X4Y1 U 2 X X2Y4	Y Y 4 W X 1 Z Y 1 X X -2 W Y -1 U Y -3 Z X 4 Z Y -2 Y X -5 Z Y 2 Z X 1	O veículo Z obteve sucesso em chegar até à saída. 0.000089s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000089s (tempo total)

Z 2 X X1Y1 Y 3 X X3Y3 X 3 X X4Y4 W 3 X X2Y5 V 2 X X4Y6	Z X 2 Y X -2 Z X 2 X X -2 Z Y 3	O veículo Z obteve sucesso em chegar até à saída. 0.000081s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000081s (tempo total)

TABELA 2. Veículo Z não chegando até a saída:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 2 X X2Y5 A 2 X X1Y2 Q 2 Y X6Y3 D 2 Y X4Y1 X 2 X X4Y3 K 2 X X4Y4 B 2 Y X3Y1 C 3 X X3Y6 Y 2 X X5Y2 P 2 X X4Y5 E 2 Y X1Y3 V 2 X X1Y6 R 2 Y X3Y3 F 2 Y X2Y3 G 2 X X5Y1 H 2 X X1Y1	Z X -1	O veículo Z não obteve sucesso em chegar até à saída. 0.000104s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000104s (tempo total)
Z 3 Y X5Y1 Y 3 Y X4Y4 X 2 Y X6Y4 W 3 X X1Y3 V 3 Y X1Y4 U 2 X X5Y6 T 2 X X2Y1 S 2 X X2Y6	X Y -3 Z Y 2 Z X 1	O veículo Z não obteve sucesso em chegar até à saída. 0.000100s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000100s (tempo total)

TABELA 3. Colisão entre veículos:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 3 Y X1Y3 Y 2 Y X6Y1 X 3 Y X3Y1 W 2 Y X5Y2 V 2 X X4Y1 U 2 X X2Y4	W X 1 Y Y 4	Conjunto de manobras inviável! MOTIVO: Colisão com um veículo. .000097s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000097s (tempo total)
Z 2 Y X6Y4 Y 3 X X1Y6	Z Y 1 V Y 3	Conjunto de manobras inviável! MOTIVO: Colisão com um veículo.

X 3 X X4Y6 W 2 X X2Y5 V 2 X X3Y2	Y Y -4	0.000082s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000082s (tempo total)

TABELA 4. Colisão entre veículo e muro:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 3 X X1Y3 V 3 X X2Y5	Z X 3 V X -2	Conjunto de manobras inviável! MOTIVO: Colisão com o muro. 0.000080s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000080s (tempo total)
Z 2 X X4Y6 Y 2 Y X2Y3 X 2 Y X3Y4 W 3 X X1Y2 V 2 Y X4Y4	Z X -2 V Y -2 W X 4	Conjunto de manobras inviável! MOTIVO: Colisão com o muro. 0.000105s (tempo de usuário) + 0.000000s (tempo de sistema) = 0.000105s (tempo total)

TABELA 5. Incompatibilidade entre dados entre os arquivos:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 3 Y X1Y3	Y Y 4 W X 1 Z Y 1 X X -2 W Y -1 U Y -3 Z X 4 Z Y -2 Y X -5 Z Y 2 Z X 1	INCOMPATIBILIDADE DE DADOS... Não há veículo identificado como Y. 0.000000s (tempo de usuário) + 0.000083s (tempo de sistema) = 0.000083s (tempo total)

TABELA 6. Configuração física inicial impossível:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 3 Y X3Y2 Y 3 X X3Y4	Z Y 1 Y X 2	Configuração física impossível (dois veículos ocupando a mesma posição inicial ou um veículo estrapolando o limite do estacionamento).

TABELA 7. Arquivo de manobras vazio:

Arquivo de configuração inicial	Arquivo de manobras	Saída
Z 2 X X2Y5 A 2 X X1Y2 Q 2 Y X6Y3		O arquivo de manobras está vazio!

Tempo

Tempo de usuário (user time): Quantidade de tempo da CPU gasta com os processos fora do kernel (em modo de usuário), o tempo gasto para executar o programa apenas. Para mudar de modo de usuário para o modo kernel, o computador para o programa. Esse tempo que o programa fica bloqueado (esperando a entrada e saída por exemplo) não é contabilizado e outros processos também não.

Tempo de sistema (system time): É a quantidade de tempo da CPU gasta com os processos dentro do kernel como alocar memória (o malloc se encaixa nessa definição porém também realiza ações em modo de usuário que conta no usertime) ou acessar algum hardware. É referente ao tempo em modo kernel, tempo no qual tem acesso irrestrito ao hardware do computador.

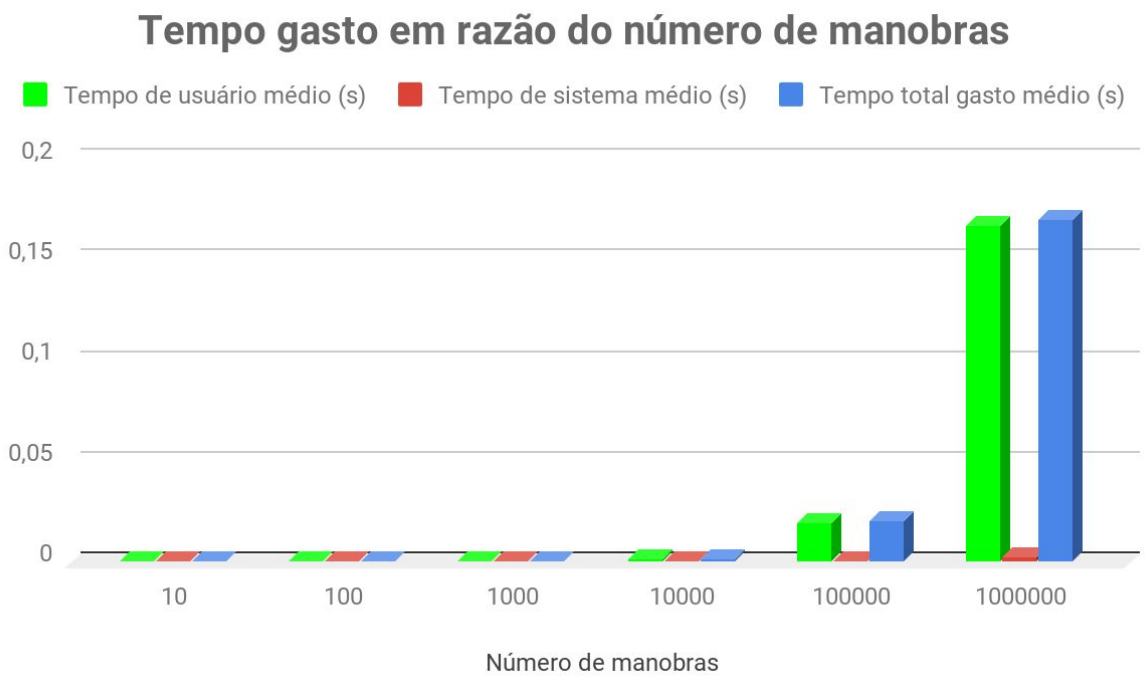
Usertime e systemtime são tempos de execução da CPU, ou seja, a soma deles não resulta no tempo real de relógio decorrido da execução do programa. Na verdade pode até ultrapassar o tempo real em casos com multi-processadores.

Para analisar o crescimento do tempo gasto de execução em relação ao número de manobras, foi realizado testes com diferentes números de manobras, no qual o programa foi executado 10 vezes para cada parâmetro e a média aritmética do tempo gasto foi calculada. Segue na tabela abaixo este estudo:

TABELA 8. Tempo médio gasto em razão do número de manobras.

Número de manobras	Tempo de usuário médio (s)	Tempo de sistema médio (s)	Tempo total gasto médio (s)
10	0,000078	0,000024	0,000102
100	0,000083	0,000023	0,000106
1000	0,000259	0,000000	0,000259
10000	0,001491	0,000333	0,001824
100000	0,019346	0,000625	0,019971
1000000	0,166936	0,002718	0,169654

GRÁFICO 1. Tempo gasto em razão do número de manobras.



Nota-se que o tempo de usuario média é grotescamente maior que o tempo de sistema, isso se dá devido ao fato de que durante a execução do programa a maior parte do tempo é gasta com processos fora do kernel, executando instruções do próprio programa e apenas uma pequena parte é gasta no kernel.

Memória

Relatório *valgrind*:

```
==11845== HEAP SUMMARY:
==11845==     in use at exit: 0 bytes in 0 blocks
==11845==   total heap usage: 14 allocs, 14 frees, 10,808 bytes allocated
```

De acordo com o relatório do valgrind foram alocadas 14 variáveis dinamicamente e todas foram desalocadas corretamente.

5. Conclusão

O programa implementado neste trabalho provou automatizar a checagem das manobras tornando o serviço do TremDaFolia mais rápido e permitindo que o estacionamento atenda ainda mais pessoas e maximize seu lucro.

6. Referências Bibliográficas

Como a automação de serviços pode melhorar a performance da sua empresa? Entenda! - Disponível em: < https://gerencianet.com.br/blog/automacao-de-servicos-para-empresas/ > Acesso em: 16 de mar. 2019.

Argumentos em linha de comando - Disponível em: < http://linguagemc.com.br/argumentos-em-linha-de-comando/ > Acesso em: 18 de mar. 2019.

ZIVIANI, Nívio. Projeto de Algoritmos. 3ª Edição. São Paulo, 2011.