Algoritmos e Estrutura de Dados III (2019-1)

Trabalho Prático 1

Heitor Lourenço Werneck
heitorwerneck@hotmail.com

30 de Abril de 2019

Conteúdo

1 Introdução

O trabalho a ser apresentado consiste na resolução do problema de planejamento de manobras tendo em vista maximizar a ocupação do estacionamento e minimizar o tempo de espera dos clientes para sair do estacionamento. Como o problema parece ser complexo, inicialmente será abordado uma parte simples da solução. Uma breve descrição do programa:

- O estacionamento é quadrado e tem dimensões 6 por 6.
- Os carros não fazem curva.
- As dimensões são identificadas de X1 a X6 e de Y1 a Y6.
- O estacionamento comporta dois tipos de veículos: carros e caminhões.
- Carros têm dimensão 2 por 1 e os caminhões têm dimensão 3 por 1.
- O objetivo é mover um carro específico, que chamaremos Z para a saída do estacionamento.
- O programa deve receber como entrada a configuração inicial do estacionamento e um conjunto de manobras.
- Deve responder se teve sucesso ou não em retirar o carro Z.

Entrada do programa:

- Arquivo de configuração inicial no qual informa os veículos e suas posições. Os veículos podem estar posicionados paralelo ao eixo X ou ao eixo Y. Cada linha informa sobre um veículo: identificador, tamanho, direção e posição.
- Arquivo de manobras no qual informa os movimentos dos veículos. Cada movimento indica o veículo, a dimensão onde o movimento ocorre e a amplitude do movimento.

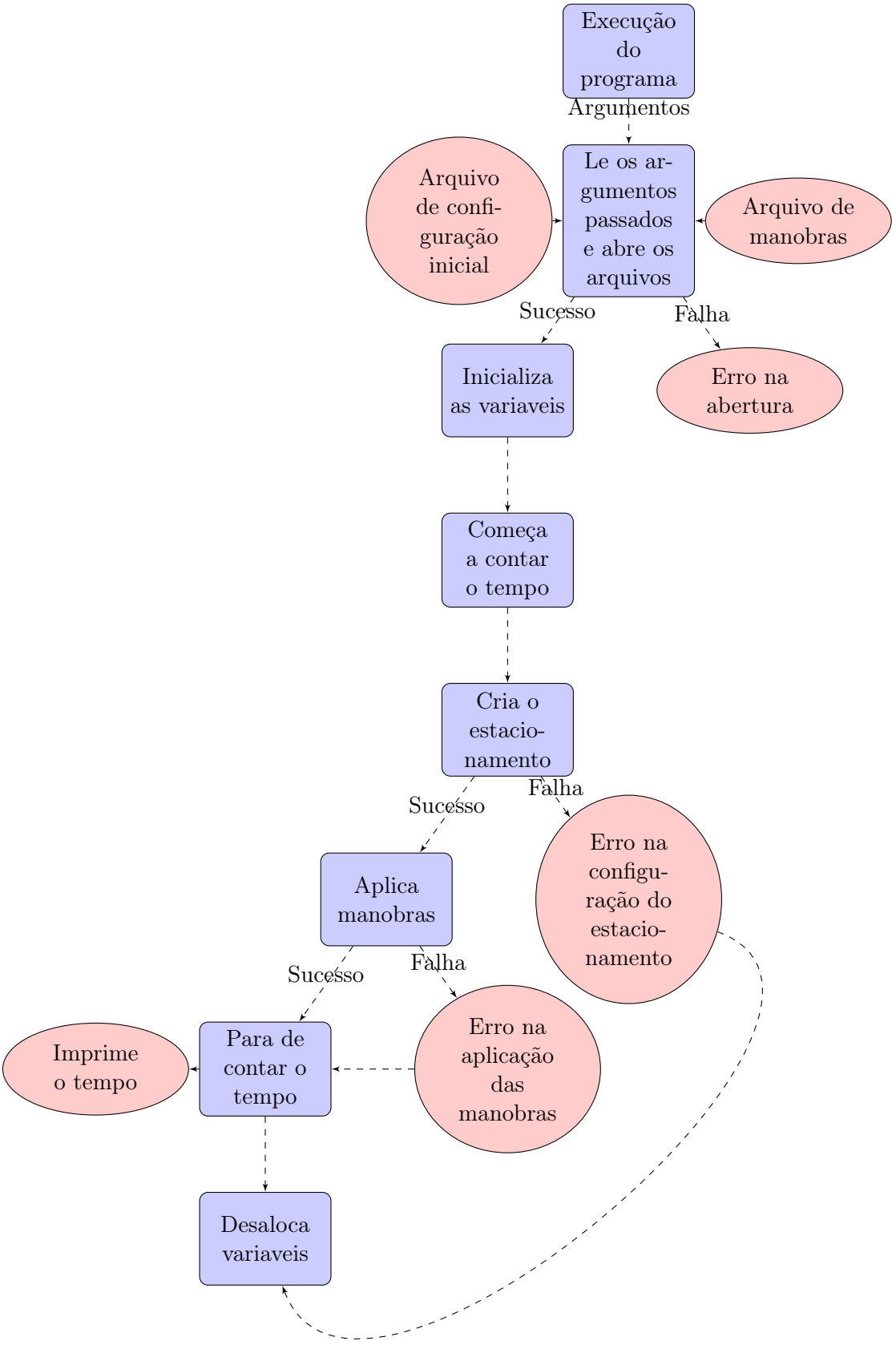
Erros que devem ser alertados:

- Alertar para configurações físicas impossíveis (isto é, dois veículos ocupando a mesma posição).
- Manobras inviáveis (colisões entre veículos e com os muros

do estacionamento).

Logo será criado um simulador de um estacionamento simplificado. Para resolver esse problema foi utilizado estruturas de dados com boa complexidade assintótica para casos gerais.

1.1 Uma visão geral sobre a solução



2 Módulos

2.1 Estruturas de dados

As estruturas de dados são de extrema importância para resolução do problema, elas irão ter grande peso na complexidade do programa e no seu funcionamento.

2.1.1 Tabela de dispersão

A tabela de dispersão [2] foi utilizada pois visto que há referencia aos nomes respectivos dos veículos constantemente no arquivo de manobras, logo uma tabela de dispersão comum garante complexidade de tempo sobre as operações de inserção e busca de $\Theta(1)$ e $O(n)$(Ao longo do documento será utilizado espalhamento comum para o espalhamento com complexidade $O(n)$). É interessante lembrar que existe o espalhamento perfeito (*perfect hashing*) [1] que também foi implementado nesse trabalho que garante complexidade de tempo $O(1)$ para operações de inserção e busca, porém a complexidade de espaço no pior caso é $\Omega(n)$.

A tabela de dispersão normal utilizou a função de espalhamento modular com sondagem linear.

Na análise de resultados será possível ver quão diferente o espalhamento perfeito pode ser no tempo de execução.

O espalhamento perfeito foi utilizado aproveitando-se do fato que os identificadores dos veículos são caracteres, e existem 256 caracteres diferentes, ou seja um numero fixo de entradas possíveis.

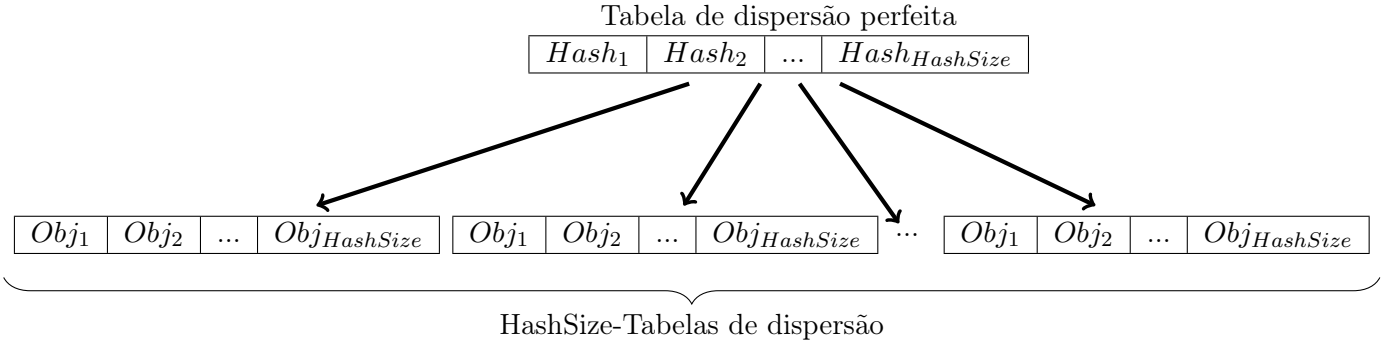
A tabela de dispersão principal contem 16 outras tabelas podendo conter 16 objetos, todas elas não alocadas para uma economia de blocos de caracteres não utilizados.

$$\underbrace{obj.label}_{\text{Rotulo do objeto}} \in \{x : x \in \mathbb{N} \wedge (0 \leq x \leq 255)\}$$

$$\begin{aligned} ASCII\_SIZE &= 256 \\ HashSize &= \frac{ASCII\_SIZE}{16} \\ HashSize &= 16 \end{aligned} \tag{2.1}$$

Um detalhe de implementação interessante e que na criação da tabela de dispersão com espalhamento perfeito já é alocado os blocos de caracteres previstos como sendo os mais utilizados, que são os blocos que contem A e Z.

```
perf_hash->hashs[hash_division('A',TABLE_SIZE)]= new_hash(TABLE_SIZE);
perf_hash->hashs[hash_division('Z',TABLE_SIZE)]= new_hash(TABLE_SIZE);
```



2.1.2 Mapa de Colisão

O mapa de colisão foi feito para trabalhar com a colisão de maneira que ela fosse um evento ou seja, só é preciso checar caso haja alguma mudança no mapa e só ira checar colisões nas áreas de mudança oque garante uma complexidade assintótica ótima. Também é importante utilizá-lo para diminuir o tempo de

busca pelo carro Z para ser O(1) pois caso não haja mapa de colisão com os identificadores dos veículos no mapa esse tempo de busca seria $O(n)$ em uma tabela de dispersão não perfeita.

Como definido na introdução o tamanho do mapa em cada dimensão é 6 e há 2 dimensões.

$$\begin{aligned} MapDimensions &\coloneqq 2 \\ MapSize &\coloneqq 6 \end{aligned} \tag{2.2}$$

```
Y+---+---+---+---+---+---+
6|   |   |   |   |   |   |
 +---+---+---+---+---+---+
5|   |   |   |   |   |   |
 +---+---+---+---+---+---+
4|   |   |   |   | Z | Z |
 +---+---+---+---+---+---+
3|   |   |   |   |   | A |
 +---+---+---+---+---+---+
2|   |   |   |   |   | A |
 +---+---+---+---+---+---+
1|   |   |   |   |   | A |
 +---+---+---+---+---+---+
   1   2   3   4   5   6 X
      Mapa de colisão
```

O mapa possui funções para inserir objetos nele e ao mesmo tempo detectar colisões devido a inserções e também possui função de remoção do mapa.

```
int collission_map_insert(map coll_map,object* obj);
int collission_map_remove(map coll_map,object* obj);
```

2.1.3 Tempo de algoritmo

Essa é a estrutura de dados que guarda os dados relacionados a tempo do algoritmo. Capturando o tempo de usuário, tempo de sistema e o tempo total.

```
struct algorithm_time{
  double utime,stime,total_time;
};
```

Há uma função auxiliar para contar o tempo, possui um parâmetro que auxilia na inicialização da contagem e na parada e outros parâmetros que guardam os dados.

```
enum {START, STOP};
void time_count(struct rusage *resources,char start,algorithm_time *algo_time);
```

2.1.4 Objeto

A estrutura de dados *Object* descreve um objeto que é a generalização de um veículo e pode ser especificado para um veículo. Possui os atributos a seguir:

- Rótulo (Identificador)
- Posições x e y
- Tamanho

Com base na introdução será definido o menor tamanho de um objeto.

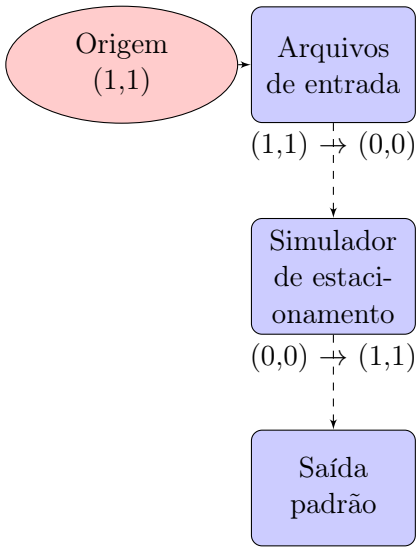
$$MinObjectSize \coloneqq 2 \tag{2.3}$$

- Direção

```
struct object{
  char label,dir;
  int x,y,size;
};
```

Esse tipo de dado contem as funções básicas de leitura e mudança de variáveis. Assim como um pequeno detalhe de implementação que é a normalização das coordenadas, visto que os vetores na linguagem de programação usada começam no índice 0, para se adequar a essa estrutura há uma normalização dos dados quando eles entram e quando eles saem. Ou seja internamente os objetos guardam as coordenadas com a origem em (0,0).

```
int normalize_coordinate(int coord);
int normalize_coordinate_out(int coord);
```



Um *Object* também possui uma função para se mover com base em uma direção e amplitude dada.

```
void obj_move(object *obj,char dir,int amplitude);
```

2.2 Entrada e saída

Esse módulo é responsável pela maior parte das operações de entrada e saída do programa. Possui a função que irá ler os argumentos passados para o programa.

```
void entry_read(int argc, char* argv[],FILE** fpc,FILE** fm);
```

Sendo a entrada padronizada como:

```
parking-simulator(-perfect) -c <arquivo de configuração> & -m <arquivo de manobras> & -d (DEBUG)
```

Este modulo também possui funções para imprimir o mapa de colisão, objetos e o tempo de algoritmo.

```
void print_collission_map(map cmap);
void print_algorithm_time(const algorithm_time *ai,FILE* f);
void obj_print(object* obj);
```

2.3 Lógica de configuração inicial

A lógica de configuração inicial é feita pelo módulo *parking*.

Essa configuração inicial colocara os objetos passados na entrada para o mapa em suas posições indicadas e direção.

Se houver algum tipo de configuração impossível o programa irá detectar.

Há 2 versões da lógica de configuração inicial, cada uma delas se diferenciam somente pela tabela de dispersão utilizada e como o código foi bem estruturado toda a lógica se mantem somente mudando a estrutura de dados.

```
int parking_init(FILE* fpc,hash* objs,map coll_map);
int parking_init_perfect(FILE* fpc,perfect_hash* objs,map coll_map);
```

Seu funcionamento poderá ser visto na secção de análise de complexidade.

2.4 Lógica de manobras

A lógica de manobras é feita pelo módulo *maneuvers*. O processo para aplicação das manobras é ler a linha que contem uma manobra, transformar os dados em elementos computáveis, mover o objeto na direção e amplitude passada movendo passo a passo. Se ocorrer algum erro as estruturas de dados auxiliares irão detectar e é só tratar o erro. E é só repetir as mesmas instruções para cada manobra.

Assim como a lógica de configuração inicial há 2 versões pelo mesmo motivo.

```
int apply_maneuvers(FILE* fm,hash* objs,map coll_map);
int apply_maneuvers_perfect(FILE* fm,perfect_hash* objs,map coll_map);
```

O algoritmo poderá ser visto na secção de análise de complexidade.

2.5 Principal

Existem 2 algoritmos principais assim também como as lógicas que fazem a utilização de todas estruturas e logicas criadas para obter a solução do problema. Seu algoritmo será abordado na secção de análise de complexidade.

3 Análise de complexidade

A análise de complexidade foi feita para descobrir como o algoritmo se comporta com tamanhos de entrada tendendo ao infinito, ou seja, o comportamento assintótico. O foco será a descoberta do pior caso dos algoritmos pois não haverá caso pior que o pior caso.

Será feito a análise dos dois algoritmos utilizando as duas estruturas de dados diferentes, tabela de dispersão com espalhamento perfeito e tabela de dispersão com espalhamento normal.

3.1 Tabela de dispersão comum

3.1.1 Configuração inicial do estacionamento

Primeiro é necessário analisar as funções elementares que aparecem constantemente na resolução do problema.

A primeira delas é o **insert_hash**, essa é a função que insere um elemento na tabela de dispersão.

$$\underbrace{n}_{\substack{\text{quantidade de elementos na tabela de dispersão}}} \in \mathbb{N} \tag{3.1}$$

Algorithm 1 Inserir na tabela de dispersão

Input: hash, obj

```
1:  procedure INSERT_HASH
2:      position = key_division(key, hash.size)                          ▷ O(1)
3:      for i = 0 to hash.size − 1 do                                    ▷ O(n*max(1,1))=O(n)
4:          newPosition = linear_probing(position, i, hash.size)         ▷ O(1)
5:          if hash.objs[newPosition] = null then                        ▷ O(1)
6:              hash.objs[newPosition] = obj
7:              return SUCCESS
8:          end if
9:      end for
10:     return FAIL
11: end procedure
```

$$
\begin{aligned}
& && (3.1)\\
& insert\_hash(n) && \\
& \sum_{i=1}^{n} O(1) && (3.2)\\
& n \cdot O(1) && \\
& O(n) && \\
\therefore\ & insert\_hash(n) \in O(n) &&
\end{aligned}
$$

A interpretação intuitiva da ordem de complexidade dessa função seria que a função é $O(n)$ no pior caso pois podem ocorrer n colisões na inserção.

A próxima função é a inserção do objeto no mapa de colisões. Que é necessária para detectar colisões.

Algorithm 2 Inserção no mapa de colisão

Input: collissionMap, obj

```
1:  procedure COLLISSION_MAP_INSERT
2:      y = obj.y
3:      x = obj.x
4:      if obj.direction = X then dirIsX = 1 else dirIsX = 0
5:      if obj.direction = Y then dirIsY = 1 else dirIsY = 0
6:      for i = 0 to obj.size − 1 do                                     ▷ O(obj.size*max(1,1))=O(obj.size)
7:          if y < collissionMap.SizeY AND x < collissionMap.SizeX AND
8:  collissionMap[y][x] = Empty then                                     ▷ O(1)
9:              collissionMap[y + dirIsY ∗ i][x + dirIsX ∗ i] = obj.label   ▷ O(1)
10:         else
11:             return FAIL
12:         end if
13:     end for
14:     return SUCCESS
15: end procedure
```

$$
\underbrace{ObjSize}_{\substack{\text{tamanho}\\ \text{do objeto}}} \in \{2, 3\} \tag{3.3}
$$

$$
\begin{aligned}
&(3.3)\\
&collission\_map\_insert(ObjSize)\\
&ObjSize \cdot O(1)\\
&O(ObjSize)\\
\therefore\ &collission\_map\_insert(ObjSize) \in O(ObjSize)
\end{aligned}
\tag{3.4}
$$

É importante notar que *ObjSize* faz parte de um conjunto finito logo ele é assintoticamente igual a $O(1)$.

A intuição por trás disso é que existe uma constante maior que todos elementos de um conjunto finito. Ou seja pode-se observar um elemento de um conjunto finito como uma constante.

$$
\begin{aligned}
&\exists(c \in \mathbb{N}^*)\forall ObjSize \mid ObjSize \leq c\\
&c = max(ObjSize)\\
&ObjSize \leq max(ObjSize)\\
\therefore\ &collission\_map\_insert(ObjSize) \in O(1)
\end{aligned}
\tag{3.5}
$$

Já com essas duas funções analisadas pode-se analisar a função **parking_init** que é a função que realiza as instruções contidas no arquivo de configuração inicial do estacionamento.

Algorithm 3 Configuração inicial do estacionamento

Input: fpc(File parking config), objs, collissionMap

```
1:  procedure PARKING_INIT
2:      while NOT EndOfFile(fpc) do                          ▷ Objs*(O(n)+O(ObjSize))
3:          scanFileLine(fpc, label, size, dir, x, y)        ▷ O(1)
4:          obj = new_obj(label, size, dir, x, y)            ▷ O(1)
5:          if insert_hash(objs, obj) = FAIL then            ▷ O(n)
6:              return FAIL
7:          end if
8:          if collission_map_insert(collissionMap, obj) = FAIL then   ▷ O(ObjSize)
9:              return FAIL
10:         end if
11:     end while
12:     return SUCCESS
13: end procedure
```

Como dito na especificação do problema o número de objetos são limitados, visto que o mapa foi definido como 6x6=36 e o menor objeto é de tamanho 2 logo so cabem 18 objetos no mapa no máximo, esse conjunto de 0 a 18 objetos pode ser visto como uma constante.

Seja *Objs* a quantidade de objetos (especificados no arquivo de configuração inicial).

$$
\begin{aligned}
&\underbrace{Objs}_{\substack{\text{quantidade}\\ \text{de objetos}}} \in \mathbb{N}\\
(2.2)Objs \in \{x : x \in \mathbb{N} \wedge (0 \leq x \leq MapSize^{MapDimensions} \div MinObjectSize)\}\\
&Objs \in \{x : x \in \mathbb{N} \wedge (0 \leq x \leq 18)\}\\
&\exists(c \in \mathbb{N}^*)\forall Objs \mid Objs \leq c\\
&c = max(Objs)\\
&Objs \leq max(Objs)\\
&Objs \in O(1)
\end{aligned}
\tag{3.6}
$$

$$
\begin{aligned}
i &\in \{x : x \in \mathbb{N}^* \wedge x \leq Objs\} \\
n &= Objs - i \\
\exists(c \in \mathbb{N}^*)\exists(m \in \mathbb{N}^*)\forall(Objs \geq m) & \\
0 &\leq n \leq c \cdot Objs \\
0 &\leq Objs - i \leq c \cdot Objs \\
m &= 1; c = 1 \\
0 &\leq Objs - i \leq Objs \\
Objs - i &\in O(Objs) \\
n &\in O(Objs) \\
&(3.6) \\
n &\in O(1)
\end{aligned}
\tag{3.7}
$$

$$
\begin{aligned}
& parking\_init(Objs, ObjSize) \\
& \sum_{i=1}^{Objs} (O(n) + O(ObjSize)) \\
(3.5) \implies & \sum_{i=1}^{Objs} O(max(n, 1)) \\
& \sum_{i=1}^{Objs} O(n)
\end{aligned}
\tag{3.8}
$$

$$
\begin{aligned}
(3.7)\ & \sum_{i=1}^{Objs} O(Objs) = Objs \cdot O(Objs) = O(Objs^2) \\
& \therefore parking\_init(Objs, ObjSize) \in O(Objs^2) \\
& \therefore parking\_init(Objs, ObjSize) \in O(1)
\end{aligned}
$$

3.1.2 Aplicação das manobras

Do mesmo modo que foi feito para configuração inicial deverá ser feito para a aplicação das manobras, será analisado primeiro as funções elementares.

A primeira função será a **search_hash**, ela será necessária para fazer a busca pelo rotulo do objeto. Isso será utilizado a cada manobra a ser executada, pois é necessário saber o objeto que corresponde a cada rotulo para realizar a manobra.

Algorithm 4 Busca objeto em uma tabela de dispersão

Input: hash, label

```
1:  procedure SEARCH_HASH
2:      position = key_division(key, hash.size)                    ▷ O(1)
3:      for i = 0 to hash.size − 1 do                              ▷ O(n*max(1,1,1))=O(n)
4:          newPosition = linear_probing(position, i, hash.size)   ▷ O(1)
5:          if exist(hash.objs[newPosition]) =FALSE then           ▷ O(1)
6:              return FAIL
7:          end if
8:          if hash.objs[newPosition].label = label then           ▷ O(1)
9:              return hash.objs[newPosition]
10:         end if
11:     end for
12:     return FAIL
13: end procedure
```

$$
\begin{aligned}
&(3.1)\\
search\_hash(n) &= \\
\sum_{i=1}^{n} O(1) &= \\
n \cdot O(1) &= \\
O(n) &\\
\therefore search\_hash(n) &\in O(n)
\end{aligned}
\tag{3.9}
$$

Além da inserção também será necessário a remoção do objeto do mapa de colisão para simular um movimento do mesmo.

Algorithm 5 Remoção no mapa de colisão

Input: collissionMap, obj

```
1: procedure COLLISSION_MAP_REMOVE
2:     y = obj.y
3:     x = obj.x
4:     if obj.direction = X then dirIsX = 1 else dirIsX = 0
5:     if obj.direction = Y then dirIsY = 1 else dirIsY = 0
6:     for i = 0 to obj.size − 1 do                    ▷ O(ObjSize*max(1,1,1))=O(ObjSize)
7:         if y < collissionMap.SizeY AND x < collissionMap.SizeX then        ▷ O(1)
8:             if collissionMap[y][x] ≠ Empty then                            ▷ O(1)
9:                 collissionMap[y + dirIsY ∗ i][x + dirIsX ∗ i] = Empty      ▷ O(1)
10:            end if
11:        else
12:            return FAIL
13:        end if
14:    end for
15: end procedure
```

$$
\begin{aligned}
&(3.3)\\
collission\_map\_remove(ObjSize) &\\
ObjSize \cdot O(1) = O(ObjSize) &\\
\therefore collission\_map\_remove(ObjSize) &\in O(ObjSize)\\
(3.5)\ \therefore collission\_map\_remove(ObjSize) &\in O(1)
\end{aligned}
\tag{3.10}
$$

Com essas funções analisadas já é possível analisar a função **apply_maneuvers** (função que aplica as manobras).

Seja *maneuvers* a quantidade de manobras e *amplitude* a amplitude do movimento.

$$
\underbrace{maneuvers}_{\substack{\text{quantidade}\\ \text{de manobras}}} \in \mathbb{N} \tag{3.11}
$$

$$
\underbrace{amplitude}_{\substack{\text{amplitude}\\ \text{do movimento}}} \in \{x : x \in \mathbb{N} \wedge (0 \leq x < MapSize)\}; amplitude \in O(1) \tag{3.12}
$$

Algorithm 6 Aplicação das manobras

Input: fm(File maneuvers), objs, collissionMap

```
1: procedure APPLY_MANEUVERS
2:     while NOT EndOfFile(fm) do                                   ▷ O(maneuvers*n)
3:         scanFileLine(fm, label, direction, amplitude)            ▷ O(1)
4:         currentObj = search_hash(objs, label)                    ▷ O(n)=O(Objs)
5:         sense = amplitude/absoluteValue(amplitude)               ▷ O(1)
6:         amplitude = absoluteValue(amplitude)                     ▷ O(1)
7:         for i = 0 to amplitude-1 do                              ▷
   O(amplitude*max(ObjSize,1,ObjSize))=O(amplitude*ObjSize)=O(1)
8:             collission_map_remove(collissionMap, currentObj)     ▷ O(ObjSize)
9:             obj_move(currentObj, direction, currentObj)          ▷ O(1)
10:            if collission_map_insert(collissionMap, currentObj) = FAIL then   ▷ O(ObjSize)
11:                return FAIL
12:            end if
13:        end for
14:    end while
15: end procedure
```

$$
\begin{aligned}
&(3.11)\\
&(3.12)\\
&apply\_maneuvers(maneuvers, amplitude, ObjSize, Objs)\\
&maneuvers \cdot (O(amplitude \cdot max(ObjSize, 1, ObjSize)) + O(n))\\
&maneuvers \cdot (O(amplitude \cdot ObjSize) + O(n))\\
&maneuvers \cdot (O(1) + O(Objs)) \qquad (3.13)\\
&maneuvers \cdot O(Objs)\\
&O(maneuvers \cdot Objs)\\
\therefore\ &apply\_maneuvers(maneuvers, amplitude, ObjSize, Objs) \in O(maneuvers \cdot Objs)\\
&(3.6)\\
\therefore\ &apply\_maneuvers(maneuvers, amplitude, ObjSize, Objs) \in O(maneuvers)
\end{aligned}
$$

É importante deixar um fato explicitado, as funções **parking_init** e **apply_maneuvers** foram explicitadas com duas ordens de complexidade na conclusão da análise, mesmo que tenha chegado nessas complexidades a escolha foi que será priorizado a complexidade que contem o termo *Objs*, pois sem isso não seria deixado explicito que existe um termo que representa a colisão da tabela de dispersão.

E além disso com o termo explicitado se o tamanho do mapa pudesse variar a analise de complexidade trataria esse caso também, logo quando a analise contem o termo *Objs* ela é mais generalista. Ou seja, quando possível será preservado o termo *Objs*.

3.1.3 Principal

Com as principais funções que o programa principal chama analisadas só falta analisar o programa principal.

$$
\begin{aligned}
&(3.1)\\
&parking\_simulator(Objs, maneuvers)\\
&O(max(Objs, Objs^2, maneuvers \cdot Objs))\\
&O(max(1, 1, maneuvers \cdot Objs)) \qquad (3.14)\\
&O(maneuvers \cdot Objs)\\
\therefore\ &parking\_simulator(Objs, maneuvers) \in O(maneuvers \cdot Objs)\\
\therefore\ &parking\_simulator(Objs, maneuvers) \in O(maneuvers)
\end{aligned}
$$

Algorithm 7 Algoritmo principal

Input: fpc,fm ▷ Arquivo de configuração inicial e manobra

1: **procedure** PARKING_SIMULATOR
2: $collissionMap = init\_map()$ ▷ O(1)
3: $fpcLines = file\_lines(fpc)$ ▷ O(1)
4: $objs = new\_hash(fpcLines)$ ▷ O(Objs)
5: $time\_count(START, algoTime)$ ▷ O(1)
6: **if** $parking\_init(fpc, objs, collissionMap)$ **then** ▷ O($Objs^2$)
7: $apply\_maneuvers(fm, objs, collissionMap)$ ▷ O($maneuvers \cdot Objs$)
8: $time\_count(STOP, algoTime)$ ▷ O(1)
9: $print\_algorithm\_time(algoTime, stdout)$ ▷ O(1)
10: **end if**
11: **end procedure**

3.2 Tabela de dispersão com espalhamento perfeito

3.2.1 Configuração inicial do estacionamento

Agora para a tabela dispersão as coisas que irão mudar serão as funções da tabela. A inserção será a primeira delas abordada. Agora como não há colisão o índice dado é absoluto e todas operações serão $O(1)$ como será mostrado a seguir.

Algorithm 8 Inserir na tabela de dispersão com espalhamento perfeito

Input: hash, obj

1: **procedure** INSERT_PERFECT_HASH ▷ O(max(1,1,1,1)) = O(1)
2: $hash\_index = hash\_division(obj.label, HashSize)$ ▷ (2.1) O(1)
3: $obj\_index = key\_division(obj.label, HashSize)$ ▷ O(1)
4: $hash.hashs[hash\_index].objs[obj\_index] = obj$ ▷ O(1)
5: **return** SUCCESS ▷ O(1)
6: **end procedure**

$$\begin{aligned} & insert\_perfect\_hash(n) \\ & O(max(1,1,1,1)) \\ \therefore\ & insert\_perfect\_hash(n) \in O(1) \end{aligned} \tag{3.15}$$

Pela matemática feita pode-se observar que o custo de utilizar a função de inserção não muda com a quantidade de elementos na tabela.

Algorithm 9 Configuração inicial do estacionamento utilizando espalhamento perfeito

Input: fpc(File parking config), objs,collissionMap

1: **procedure** PARKING_INIT_PERFECT
2: **while** NOT EndOfFile(fpc) **do** ▷ O(Objs*max(1,ObjSize)) = O(Objs)
3: $scanFileLine(fpc, label, size, dir, x, y)$ ▷ O(1)
4: $obj = new\_obj(label, size, dir, x, y)$ ▷ O(1)
5: **if** $insert\_perfect\_hash(objs, obj)$ = FAIL **then** ▷ O(1)
6: **return** FAIL
7: **end if**
8: **if** $collission\_map\_insert(collissionMap, obj)$ = FAIL **then** ▷ O(ObjSize)=O(1)
9: **return** FAIL
10: **end if**
11: **end while**
12: **return** SUCCESS
13: **end procedure**

$$
\begin{aligned}
& && (3.6)\\
& parking\_init\_perfect(Objs, ObjSize) && \\
& \sum_{i=1}^{Objs} (O(1) + O(ObjSize)) && \\
& && (3.5)\\
& \sum_{i=1}^{Objs} O(1) && (3.16)\\
& Objs \cdot O(1) && \\
& O(Objs) && \\
\therefore\ & parking\_init\_perfect(Objs, ObjSize) \in O(Objs) && \\
\therefore\ & parking\_init\_perfect(Objs, ObjSize) \in O(1) &&
\end{aligned}
$$

3.2.2 Aplicação das manobras

Agora na busca será a mesma coisa.

Algorithm 10 Buscar na tabela de dispersão com espalhamento perfeito

Input: hash, obj

```
1: procedure SEARCH_PERFECT_HASH                          ▷ O(max(1,1,1)) = O(1)
2:     hash_index = hash_division(obj.label, HashSize)    ▷ O(1)
3:     obj_index = key_division(obj.label, HashSize)      ▷ O(1)
4:     return hash.hashs[hash_index].objs[obj_index]      ▷ O(1)
5: end procedure
```

$$
\begin{aligned}
& search\_perfect\_hash(n) && \\
& O(max(1,1,1)) && (3.17)\\
& O(1) && \\
\therefore\ & search\_perfect\_hash(n) \in O(1) &&
\end{aligned}
$$

Algorithm 11 Aplicação das manobras utilizando espalhamento perfeito

Input: fm(File maneuvers), objs, collissionMap

```
1: procedure APPLY_MANEUVERS_PERFECT
2:     while NOT EndOfFile(fm) do                          ▷ O(maneuvers*1) = O(maneuvers)
3:         scanFileLine(fm, label, direction, amplitude)   ▷ O(1)
4:         currentObj = search_perfect_hash(objs, label)   ▷ O(1)
5:         sense = amplitude/absoluteValue(amplitude)      ▷ O(1)
6:         amplitude = absoluteValue(amplitude)            ▷ O(1)
7:         for i = 0 to amplitude-1 do                     ▷
   O(amplitude*max(ObjSize,1,ObjSize))=O(amplitude*ObjSize)=O(1)
8:             collission_map_remove(collissionMap, currentObj)   ▷ O(ObjSize) = O(1)
9:             obj_move(currentObj, direction, currentObj)        ▷ O(1)
10:            if collission_map_insert(collissionMap, currentObj) = FAIL then ▷ O(ObjSize) = O(1)
11:                return FAIL
12:            end if
13:        end for
14:    end while
15: end procedure
```

$$
\begin{aligned}
&(3.11)\\
&(3.12)\\
&apply\_maneuvers\_perfect(maneuvers, amplitude, ObjSize)\\
&maneuvers \cdot (O(amplitude \cdot max(ObjSize, 1, ObjSize)) + O(1))\\
&maneuvers \cdot (O(amplitude \cdot ObjSize) + O(1))\\
&maneuvers \cdot (O(1) + O(1))\\
&O(maneuvers)\\
\therefore\ &apply\_maneuvers\_perfect(maneuvers, amplitude, ObjSize) \in O(maneuvers)
\end{aligned}
\tag{3.18}
$$

3.2.3 Principal

Com as principais funções que o programa principal chama analisadas so falta analisar o programa principal.

Algorithm 12 Algoritmo principal com espalhamento perfeito

Input: fpc,fm ▷ Arquivo de configuração inicial e manobra

```
1:  procedure PARKING_SIMULATOR_PERFECT
2:      collissionMap = init_map()                                  ▷ O(1)
3:      fpcLines = file_lines(fpc)                                  ▷ O(1)
4:      objs = new_hash(HashSize)                                   ▷ (2.1) O(1)
5:      time_count(START, algoTime)                                 ▷ O(1)
6:      if parking_init_perfect(fpc, objs, collissionMap) then      ▷ O(Objs)
7:          apply_maneuvers_perfect(fm, objs, collissionMap)        ▷ O(maneuvers)
8:          time_count(STOP, algoTime)                              ▷ O(1)
9:          print_algorithm_time(algoTime, stdout)                  ▷ O(1)
10:     end if
11: end procedure
```

$$
\begin{aligned}
&(3.1)\\
&parking\_simulator\_perfect(Objs, maneuvers)\\
&O(max(Objs, maneuvers))\\
&(3.6)\\
&O(maneuvers)\\
\therefore\ &parking\_simulator\_perfect(Objs, maneuvers) \in O(maneuvers)
\end{aligned}
\tag{3.19}
$$

3.3 Conclusão

Foi possível observar que os dois algoritmos possuem a mesma complexidade assintótica, porém especificamente para esse problema pois o mapa não cresce o que implica em uma quantidade limitada de veículos e também pelo formato do rótulo dos veículos há uma limitação na sua quantidade também.

Devido a isto até mesmo as funções relacionadas a tabela de dispersão comum foram mostradas com duas conclusões de ordem de complexidade, uma que compreende o problema generalista e outra o problema especifico que é o feito neste trabalho.

Logo a diferenciação da complexidade dos dois algoritmos se daria por meio da procura pela função de complexidade, como esse é um processo complicado e depende de muitos fatores logo será feito uma abordagem mais simplificada na seção de resultados, será feito regressões lineares para achar os fatores da função.

4 Testes

Para saber se o algoritmo funciona como o esperado foi feito os testes dos possíveis erros. Os erros detectados como fundamentais do problema são os apresentados seguir.

4.1 Finalização

Dado como configuração inicial do estacionamento:

```
Z 2 X X5Y5
```

e o arquivo de manobras sendo:

```
Z Y -1
```

A saída é:

```
Z Reached the end! Y 4 X 6
User time 0.000048s, System time 0.000000s, Total Time 0.000048s
```

Saída correta.

4.2 Colisão

Dado como configuração inicial do estacionamento:

```
Z 2 X X5Y5
T 2 X X4Y6
```

e o arquivo de manobras sendo:

```
Z Y 1
```

Espera-se que colida.
A saída é:

```
Collission happened
Failed in maneuvers apply.
User time 0.000044s, System time 0.000000s, Total Time 0.000044s
```

Saída correta.

4.3 Posição fora do mapa

Dado como configuração inicial do estacionamento:

```
Z 2 X X5Y5
T 2 X X4Y6
R 2 X X2Y3
```

e o arquivo de manobras sendo:

```
Z X 1
```

Espera-se que saia do mapa (ou colida, que é a palavra utilizada na saída de erro).
A saída é:

```
Collission happened
Failed in maneuvers apply.
User time 0.000043s, System time 0.000000s, Total Time 0.000043s
```

Saída correta.

4.4 Inicialização com colisão

Dado como configuração inicial do estacionamento:

```
Z 2 Y X5Y5
T 2 X X4Y6
R 2 X X2Y3
A 2 Y X1Y5
```

e o arquivo de manobras sendo:

```
Z X 0
```

Espera-se que colida:

```
Collission happened in parking configuration file. Objects in same position.
Error in config file.
```

Saída correta.

4.5 Desalocação de memória

Foi feito o teste de vazamento de memória com o programa com erros e sem erros, com a utilização de um programa chamado *valgrind* que detecta blocos de memória não desalocados, e todos deram como resultado que todos blocos de memória alocados foram desalocados.

```
==12259== Memcheck, a memory error detector
==12259== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12259== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==12259== Command: ./parking-simulator -c testes/objs4.txt -m testes/man4.txt
==12259==
==12259==
==12259== HEAP SUMMARY:
==12259==     in use at exit: 0 bytes in 0 blocks
==12259==   total heap usage: 17 allocs, 17 frees, 5,548 bytes allocated
==12259==
==12259== All heap blocks were freed -- no leaks are possible
==12259==
==12259== For counts of detected and suppressed errors, rerun with: -v
==12259== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5 Resultados

Os resultados foram obtidos utilizando alguns algoritmos auxiliares em *shell script*. (veja o Apêndice)

5.1 Manobras

Foi gerado o gráfico com regressão linear a seguir do algoritmo com espalhamento normal e espalhamento perfeito, do tempo total. (Figuras 1 e 2)

É possível ver que a complexidade calculada foi comprovada, os gráficos mostram uma complexidade linear.

Pode-se notar que o espalhamento perfeito possui alguns fatores na equação linear que demonstram ser um pouco mais rápido, porém esse valor é muito pequeno. Isso demonstra que a tabela de dispersão normal possivelmente está na média colidindo muito pouco.

Ou seja por causa da limitação de objetos no mapa o algoritmo com espalhamento perfeito e espalhamento normal se diferenciam por uma constante.

Veja uma simples média de tempo dos dados com 100000 manobras.

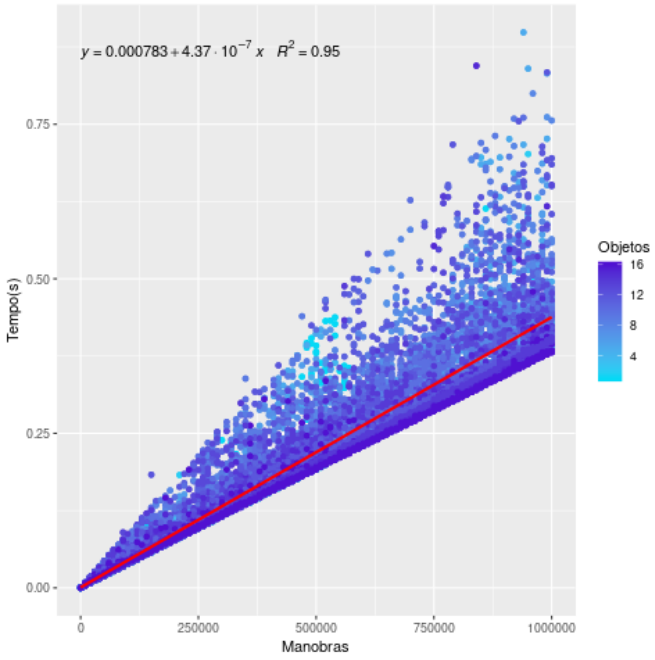


Figura 1: Tempo total do algoritmo com espalhamento normal.

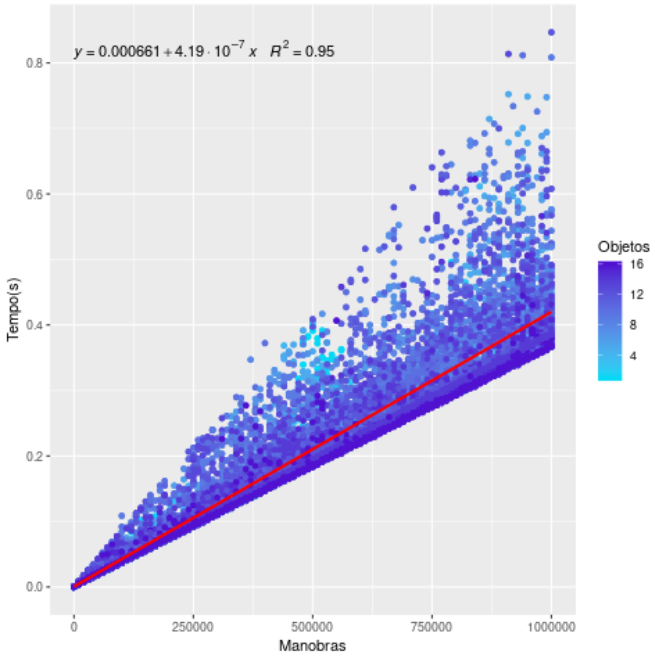


Figura 2: Tempo total do algoritmo com espalhamento perfeito.

Espalhamento normal	Espalhamento perfeito
0.04566633s	0.04402262s

É possível notar que o espalhamento perfeito na média para muitas manobras possui execução mais rápida.

Será analisado se isso se mantém para quantidades menores de manobra, especificamente 1000 manobras.

Espalhamento normal	Espalhamento perfeito
0.00013292s	0.00012228s

Pelo visto o espalhamento perfeito esta conseguindo tempos menores, então só falta uma visão geral dessas médias e intervalo de confiança das mesmas. Observe na figura 3.

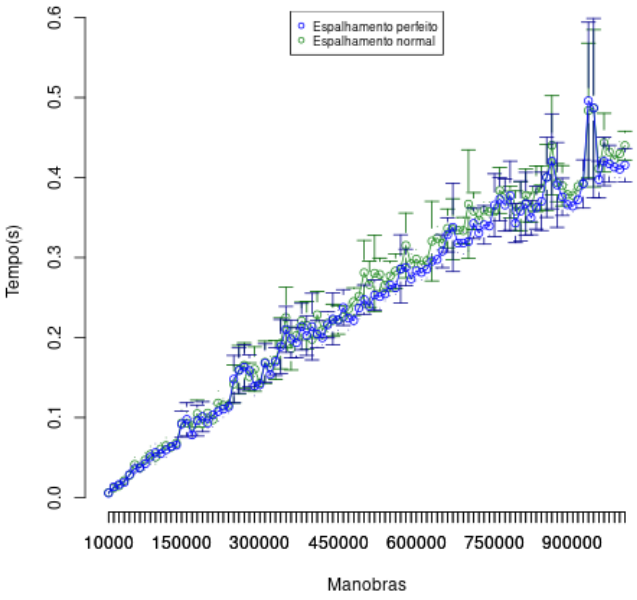


Figura 3: Média dos tempos dos algoritmos.

O gráfico mostra que o programa com espalhamento perfeito na maioria das vezes tem uma média de tempo menor que o programa com espalhamento normal.

Para um conjunto menor de manobras fica evidente que uma domina a outra por uma constante. (Figura 4)

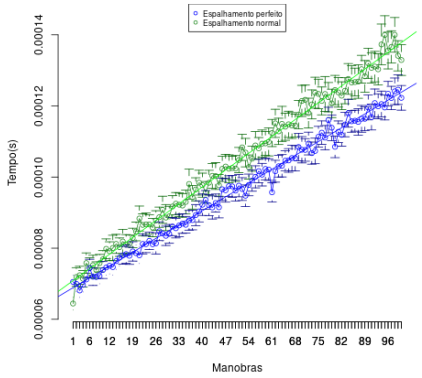


Figura 4: Média dos tempos com pequena quantidade de manobras.

5.2 Objetos

Com a análise do comportamento do custo da função com o aumento do número de manobras só falta analisar o aumento do tempo da função com base no aumento do número de objetos. (Figura 5)

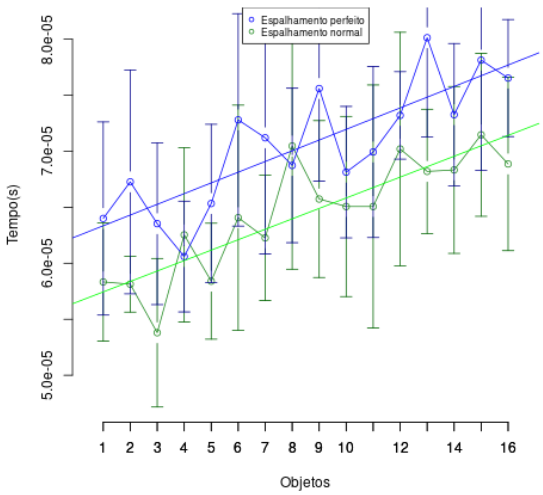


Figura 5: Média dos tempos de acordo com a quantidade de objetos, 1 manobra.

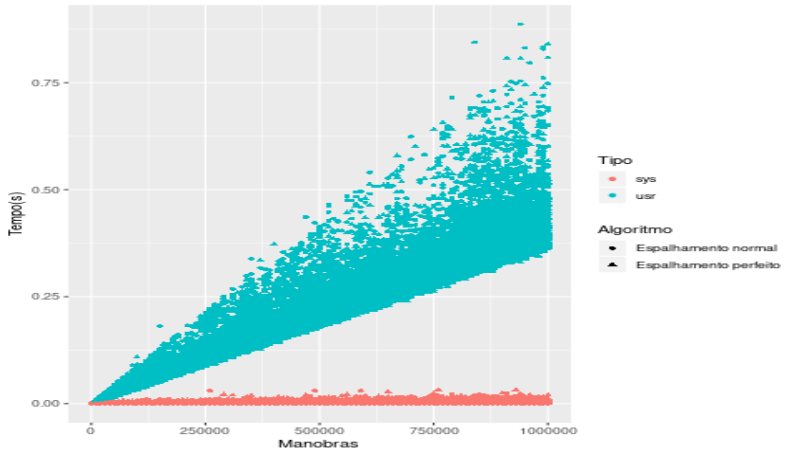


Figura 6: Tempo de usuário e sistema.

A primeira coisa que o gráfico mostra é que para um certo valor pequeno de manobras, que no caso é 1, o algoritmo com espalhamento normal ainda consegue ser mais rápido que o algoritmo com espalhamento perfeito.

E a outra informação que pode ser obtida é que o crescimento do tempo de acordo com o crescimento da quantidade de objetos parece ser um crescimento linear para ambos algoritmos. Como a complexidade do algoritmo com espalhamento normal na iniciação do estacionamento é $O(Objs^2)$ (3.8) o gráfico deveria apresentar esse comportamento, mas por causa do limite de objetos se torna inviável fazer uma regressão polinomial, ou seja a complexidade assintótica $O(1)$ também está comprovada.

5.3 Tempo de sistema e usuário

O tempo de usuário é o tempo associado ao tempo de processamento no modo de usuário ou fora do núcleo do sistema [3]. Ou seja o tempo de usuário contém o tempo processando o código do programa e das bibliotecas que o compõe, tempos de entrada e saída por exemplo não são contabilizados. Já o tempo de sistema é o tempo gasto no núcleo dentro do processo ou simplesmente o tempo gasto no modo sistema no processo, já que o usuário não pode fazer todo tipo de operação possível com os recursos de

um computador o núcleo do sistema é quem faz o intermédio para gerenciar as chamadas de operações privilegiadas do usuário (exemplo I/O).

Um exemplo é se o algoritmo for recursivo as computações locais serão somadas no tempo de usuário e a chamada a função de recursão guardara todas variáveis locais atuais na pilha do sistema e quem gerencia isso é o núcleo do sistema logo somara o tempo dessa operação no tempo de sistema. Visto que não há muitas chamadas ao sistema e também principalmente não há nenhum algoritmo recursivo o tempo no modo sistema se manteve baixo como é mostrado na figura 6.

6 Apêndice

6.1 Shell script para coleta de dados

Algoritmo **data-catch**.

```
data-catch <initial config file> <maneuvers> <execution times> <initial value> <step> <
    ↪ final value>
```

Foi utilizado o gerador de configurações iniciais para fazer mapas de 1 a 16 objetos. E a cada configuração criada era observado se o mapa conseguiria suportar um movimento de "vai e volta"de um carro em especifico, se conseguisse preserva o arquivo, se não utiliza o gerador para a quantidade de objetos em especifico.

```
for ((i=1;i<=16;i++));do ./generator -n $i -o objs"$i".txt; done
```

Com os estacionamentos criados e a manobra sendo valida para todos eles então é so começar a guardar os dados com o auxiliar que foi criado acima.

```
times=20
for ((i=1;i<=16;i++)); do ./data-catch objs"$i".txt manobras.txt $times 10000 10000
    ↪ 1000000; done
for ((i=1;i<=16;i++)); do ./data-catch objs"$i".txt manobras.txt $times 1 1 100; done
```

6.2 Gerador de configurações iniciais

Foi feito um gerador de configurações iniciais. A sua utilização pode ser vista a seguir:

```
generator -n <number of objects> & -o <output file>
```

Referências

[1] Thomas Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 2009.

[2] Paulo Feofiloff. Hashing. `https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html`, 2018. [Online; acessado 16-03-2019].

[3] Wikipedia. Kernel (operating system). `https://en.wikipedia.org/wiki/Kernel_(operating_system)`, 2019. [Online; acessado 21-03-2019].