Relatório Speed Run | AED

Professores:
João Manuel Rodrigues
Tomás Oliveira e Silva
Joaquim Madeira
Pedro Cirne
Pedro Lavrador

Speed Run

Paulo Macedo,102620 - 33.33%
Afonso Baixo, 103511 - 33.33%
Luís Leal, 108237 - 33.33%

Departamento de eletrónica telecomunicações e informática
Universidade de Aveiro
5 dezembro de 2022

Índice

Lista de Figuras

Capítulo 1

Introdução

Inserido na unidade curricular Algoritmos e Estrutura de Dados, este relatório serve o propósito de analisar e explicar o código produzido para resolver o problema "Speed Run", proposto pelos professores.

Havendo uma estrada com um determinado número de segmentos, o problema baseia-se em chegar à posição final com o menor esforço e menor quantidade de movimentos. Para chegar a essa posição, teremos de respeitar os limites de velocidade impostos por cada um dos segmentos da estrada. Assim, o objetivo será ajustar as velocidades de modo a respeitar os limites e chegar à meta o mais rapidamente possível, com velocidade igual à inicial.

1.1 Pré-requisitos

De forma a compilar o programa, é necessário ter um compilador de C como o gcc instalado na máquina local.

1.2 Compilar

O seguintes comandos compilam o programa speedrun (speedrun.c) em que o `<executable_filename>` será o programa executável:

```
cc -Wall -O2 -D_use_zlib_=0 speed_run.c -o <executable_filename> -lm
```

Outra forma de compilar usando o zlib:

```
cc -Wall -O2 -D_use_zlib_=1 speed_run.c -o <executable_filename> -lm -lz
```

1.3 Executar

Opções:

```
-ex ......................... Produz um exemplo do resultado esperado;

<seed> ...................... Usa o valor introduzido de forma a gerar a estrada;

< > ......................... Usa um valor predefinido, "0xAED2022" para gerar a estrada.
```

Capítulo 2

Métodos de implementação

2.1 Brute Force

Quanto à função fornecida (que designamos por Brute Force), esta implementa uma solução recursiva para o problema, que testa todas as velocidades possíveis para todas posições, pela ordem : reduzir, manter, acelerar; levando a um tempo de execução bastante elevado.

A complexidade computacional deste método é $\mathcal{O}(2^n)$.

```
   static void solution_1_recursion(int move_number,int position,int speed,int final_position)
{
  int i,new_speed;
  // record move
  solution_1_count++;
  solution_1.positions[move_number] = position;
  // is it a solution?
  if(position == final_position && speed == 1)
  {
    // is it a better solution?
    if(move_number < solution_1_best.n_moves)
    {
      solution_1_best = solution_1;
      solution_1_best.n_moves = move_number;
    }
    return;
  }
  // no, try all legal speeds
  for(new_speed = speed - 1;new_speed <= speed + 1;new_speed++)
    if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for(i = 0;i <= new_speed && new_speed <= max_road_speed[position + i];i++)
        ;
      if(i > new_speed)
        solution_1_recursion(move_number + 1,position + new_speed,new_speed,final_position);
    }
}
```

2.2 Recursivo

Em primeiro lugar, é feita a verificação da posição tendo em conta a posição final (o próximo passo a dar não pode ultrapassar o limite da estrada) e também em que posição deve começar a reduzir a velocidade para chegar ao final do troço com velocidade 1. Esta última condição é feita tendo em conta a seguinte expressão:

$$\sum_{n=1}^{speed-1} n < final\_positon - position$$

Em segundo lugar, é analisado o próximo movimento, desde a posição atual até onde o salto foi feito e, se for encontrado um segmento da estrada com velocidade superior à permitida, este movimento é descartado.
Em todas estas exigências caso uma não seja verificada, a função atual termina e executa a anterior na *stack*.
De forma a controlar o fluxo das várias chamadas recursivas é feita uma prioridade de chamada. Uma vez que pretendemos terminar a estrada com o menor número de movimentos possíveis, essa prioridade será: aumentar a velocidade > manter a velocidade > diminuir a velocidade.

A complexidade computacional deste método é $\mathcal{O}(n \log n)$.

Estrutura *solution_ t*:

```
typedef struct
{
  int n_moves;                          // the number of moves
  int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
}
solution_t;
```

Variáveis:

- (int) 'move_number' : forma de saber quantos passos foram dados;
- (int) 'position' : posição em que se encontra na estrada;
- (int) 'speed' : velocidade a que circula no dado momento;
- (int) 'final_position' : última posição da estrada, não podendo esta ser excedida;
- (static *solution_t*) 'solution' : guarda as posições de todos os movimentos feitos;
- (static *solution_t*) 'best_solution' : caso encontre uma melhor forma de resolver o problema, vai ser a estrutura indicada para guardar o caminho feito;
- (static unsigned long) 'solution_count' : determina o esforço feito pela função para chegar à solução ideal.

```
  int veryNicefunction(int move_number,int position,int speed,int final_position){
// check if is an available jump
 if((speed * (speed - 1)) / 2 > final_position - position || max_road_speed[position] < speed)
     return 0;
 for(int p = position + 1 - speed;p < position ;p++)
   if(max_road_speed[p] < speed) return 0;
 // record move
 solution_count++;
```

```
  solution.positions[move_number] = position;
   // is it a solution?
  if(position == final_position && speed == 1)
  {
   // is it a better solution?
    if(move_number < best_solution.n_moves)
    {
      best_solution= solution;
      best_solution.n_moves = move_number;
    }
    return 1;
  }
  if(veryNicefunction(move_number + 1,position + speed + 1,speed + 1,final_position)) return 1;
  if(veryNicefunction(move_number + 1,position + speed,speed,final_position)) return 1;
  if(veryNicefunction(move_number + 1,position + speed - 1,speed - 1,final_position)) return 1;
  return 0;
}
```

Apesar deste método ser bastante compacto e de fácil compreensão, apresenta algumas restrições. Por vezes, quando esta faz um passo que não é permitido, a função regressa à posição que é permitida pelas condições, armazenada em *stack* e, ocasionalmente, haverá um grande esforço para regressar até essa. Outra restrição, é o facto das sucessivas chamadas à função serem armazenadas na *stack* e, portanto, caso o problema tivesse um tamanho na ordem de 10^6 o programa iria apresentar a mensagem "Segmentation fault (core dumped)"(caso testado), ou seja, para uma estrada com muitos segmentos facilmente a stack chegava à sua capacidade total.

2.3 Aprimorado

Sendo esta uma versão melhorada da função fornecida 2.1, a solução começa pela inicialização de uma cópia da estrada no elemento "int position_speed_limit[1 + _max_ road_size_]", de 2 elementos "int careful_with_limit"e "int limit_position" e das restantes variáveis necessárias.
Quando é ultrapassado um limite, será registado o valor desse mesmo limite na variável "careful_with_limit" e a sua respetiva posição em "limit_position".
De seguida, é registado que um 'move' foi efetuado, registando os respetivos valores da velocidade e posição. Após o registo, verifica-se se já a posição final já foi alcançada. Se sim, os valores registados são guardados na classe "solution_t best_solution", caso contrário, procedemos aos cálculos. Primeiro, é calculado o "new_speed", verificando que velocidades podem ser tomadas através do array cópia, depois, se nenhuma das velocidades for legal para continuar, é verificado onde está o limite que impede o avanço, guardando a sua posição e o seu valor.
É calculado o somatório tal como em 2.2, de maneira a conseguir abrandar a tempo.
No caso de um limite ter sido ultrapassado, o passo seguinte é encontrar uma posição ideal para regressar, percorrendo "X"posições atrás 1 a 1 e verificando se o valor da velocidade nessa posição é inferior ou igual ao valor dos limites impostos. Se sim, retornaremos a essa posição.
Caso o "new_speed"seja legal, é avançado um "move"com essa velocidade. Este processo é repetido quando é ultrapassada a final_ position visto que, teóricamente, esta tem um limite = 1.

A complexidade deste método é $\mathcal{O}(n \log n)$.

```
typedef struct
{
  int n_moves;                          // the number of moves
  int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
}
solution_t;
```

Variáveis:

- (static *solution_t*) 'solution_group' : estrutura referente à solução que está a ser desenvolvida à qual estão associados o 'n_moves' e o 'positions[ ]';
- (static *solution_t*) 'best_solution' : estrutura com a solução ótima;
- (static unsigned long) 'solution_group_count' : esforço dispendido pela solução;
- (static int) 'position_speed_limit[1+_max_road_size_]' : cópia do array max_road_size[ ];
- (int) 'move_number' : número do movimento;
- (int) 'position' : posição atual;
- (int) 'speed' : velocidade atual;
- (int) 'final_position' : posição final da estrada;
- (int) 'n_moves' : número de movimentos;
- (int) 'new_speed' : nova velocidade a ser adquirida;
- (int) 'careful_with_limit' : onde está guardado o valor de um limite encontrado;
- (int) 'limit_position' : onde está guardado a posição (index) do limite encontrado;
- (int) 'aux_new_speed', auxSpeed, auxPosition : valores associados às variáveis correspondentes com o objetivo de efetuar cálculos sem alterar o valor original.

```
static void improved_solution(int move_number, int position, int speed, int final_position){
  // Starting point: 0,0,0,final_position
  int new_speed, i;
  position_speed_limit[final_position] = 1;
  // Record a Move
  solution_group_count++;
  solution_group.positions[move_number] = position;
  solution_group.position_speed[move_number] = speed;
  // Is it a solution?
  if(position == final_position && speed == 1)
  {
    for(i=0; i<=36 && final_position-i>=0; i++){
      position_speed_limit[final_position-i] = max_road_speed[final_position-i];
    }
    best_solution = solution_group;
    best_solution.n_moves = move_number;
    return;
  }
  //Calculate new_speed
  int aux_new_speed;
```

```
if(speed < _max_road_speed) aux_new_speed = speed + 1;
else aux_new_speed = speed;
for(new_speed = speed+1, i=1; new_speed>=speed-1 && new_speed>=1; i++){
  if(position_speed_limit[position+i] < new_speed){
    new_speed--;
    i = -1;
    continue;
  }
  if(i == new_speed){
    aux_new_speed = new_speed;
    break;
  }
}
new_speed = aux_new_speed;

if(new_speed >= 1 && new_speed <= _max_road_speed_ ){
  int min_road_limit = 10;
  for(i = 1; i <= new_speed; i++){
    if(new_speed > position_speed_limit[position + i] && position_speed_limit[position+i] <
        min_road_limit){
 min_road_limit = position_speed_limit[position+i];
 limit_position = position + i;
    }
  };
  // If it busted because new_speed > max_road_speed
  if (new_speed > min_road_limit){
    if(position + new_speed > final_position){ careful_with_limit = 1; limit_position =
        final_position;}
    else{careful_with_limit = min_road_limit;}
    // Decrease speed
    int auxSpeed = careful_with_limit + 1;
    int auxPosition = careful_with_limit + 1;

    int sum = ( (8-careful_with_limit+1)/2 ) * (careful_with_limit + 8);
    for(i = 1; i<=sum && limit_position - i >= 0; i++){ // Saves speed limits in order to
        decrease speed on time
 if(i == auxPosition){
   if(position_speed_limit[limit_position - i] > auxSpeed){
     position_speed_limit[limit_position - i] = auxSpeed;
   }
     auxPosition += auxSpeed + 1;
     auxSpeed += 1;
 }
 else{
   if(position_speed_limit[limit_position - i] > auxSpeed){
     position_speed_limit[limit_position - i] = auxSpeed;
   }
 }
    }
    //Finds the ideal position to comeback
    for(i=1; i<=sum && limit_position-i > 0; i++){
    //
    // verifies if the speed of the current postion is acceptable, to obtain the better
        speed
 if(solution_group.position_speed[move_number-i] <=
     position_speed_limit[solution_group.positions[move_number-i]]){
```

```
      return improved_solution(move_number-i, solution_group.positions[move_number-i],
          solution_group.position_speed[move_number-i], final_position);
    }
      }
    }
    // If it did pass trough all road_limits
    return improved_solution(move_number+1, position+new_speed, new_speed, final_position);
  }
}
```

A função seguinte só é chamada uma vez e, para efeitos de simplicidade e evitar desorganizar a estrutura da estrada original, é feita a sua chamada na *main* do programa.

```
static void init_position_speed_limit(void){
  for(int i=0; i<=_max_road_size_; i++){
    position_speed_limit[i] = max_road_speed[i];
  }
}
```

Esta solução tem uma característica bastante útil devido ao auxílio da cópia da estrada original, pois é com esta cópia que será organizada a estrada de uma maneira mais eficiente, reduzindo as margens de erro.
Enquanto o programa estiver a ser executado, o array continua guardado. Isto irá facilitar a execução repetitiva da mesma função, mesmo que tenha uma posição final diferente.
Um ponto negativo desta função é, que apesar de ter a mínima quantidade de *moves*, esta só será alcançada devido à técnica de aceleração, pela qual se testam primeiro os valores em que "new_speed" é maior e, só após esse valor ser testado, é que se descobre que se trata de uma velocidade ilegal, o que faz com que tenhamos de experimentar o valor decrementado. Isto irá exigir um maior esforço da função.

2.4 Iterativo

O pensamento lógico associado a esta solução é, em tudo, semelhante à da solução apresentada no ponto anterior 2.3, excepto, pelo facto de os processos de registo de movimento, cálculo de velocidade e atualização das variáveis serem executados de forma iterativa, ou seja, é executado um ciclo com a condição de que a posição a ser analisada não é a posição final e a velocidade adoptada é diferente de 1.
A prioridade desta implementação será sempre acelerar, moderar e diminuir a velocidade, por esta ordem e, como tal, após incrementar a velocidade e com recurso ao array 'position_speed_limit[]', a velocidade adquirida é comparada com a velocidade imposta na posição 'position_speed_limit[position + i]', até que a variável 'i' seja igual a 'new_speed'. Caso a velocidade limite seja inferior à velocidade adoptada, 'new_speed' será decrementada, ficando com o valor de 'speed', ou seja, ficando em velocidade cruzeiro. Caso esta velocidade não seja apropriada, então 'new_speed' será decrementada novamente.
Quando a condição 'i == new_speed' for verificada, 'speed' ficará com o valor de 'new_speed', a 'position' irá avançar o número de posições equivalente ao valor de 'new_speed' e o 'move_number' será incrementado.

A complexidade computacional deste método é $\mathcal{O}(n \log n)$.

```
typedef struct
{
  int n_moves;                          // the number of moves
  int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
  int position_speed[1 + _max_road_size_];
}
solution_t_iterative;
```

Variáveis:

- (static *solution_t_iterative*) 'solution_2' : estrutura referente à solução que está a ser desenvolvida à qual estão associados o 'n_moves' e o 'positions[1 + max_road_size]';
- (static *solution_t_iterative*) 'solution_2_best' : estrutura com a solução mais otimizada;
- (static unsigned long) solution_2_count : esforço dispendido pela solução;
- (static int) 'position_speed_limit[1+_max_road_size_]' : cópia do array max_road_size[1 + _max_road_size];
- (static int) 'careful_with_limit' : valor de um limite encontrado;
- (static int) 'initialize' : variável utilizada para controlar a inicialização do array 'position_speed_limit[1 + _max_road_size]';
- (static int) 'final_pos_speed' : posição final da estrada,
- (int) 'move_number' : número do movimento;
- (int) 'position' : posição atual;
- (int) 'speed' : velocidade atual;
- (int) 'final_position' : posição final da estrada;
- (int) 'n_moves' : número de movimentos;
- (int) 'new_speed' : nova velocidade;
- (int) 'aux_speed' : valor da velocidade utilizada nos cálculos;
- (int) 'aux_position': valor da posição utilizada nos cálculos.

```
static void solution_2_iterative(int move_number, int position, int speed, int final_position)
{
  int new_speed, i;
  // Initializing a copy of max_road_speed[]
  if (initialize == 0)
  {
    for (i = 0; i <= _max_road_size_; i++)
      position_speed_limit[i] = max_road_speed[i];
    // Assigning every limit
    for (i = final_position; i >= 0; i--)
    {
      int aux_speed, aux_position, sum;
      aux_speed = aux_position = careful_with_limit + 1;
      // Summation of the numbers from the limit to 8
      sum = ((8 - careful_with_limit + 1) / 2) * (careful_with_limit + 8);
```

```
    // Saves the limits of the previous positions to slow down on time
    for (int j = 1; j <= sum && i - j >= 0; j++)
    {
      if (j == aux_position)
      {
        if (position_speed_limit[i - j] >= aux_speed)
          position_speed_limit[i - j] = aux_speed;
        else
        {
          i -= j;
          break;
        }
        aux_position += ++aux_speed;
      }
      else
      {
        if (position_speed_limit[i - j] >= aux_speed)
          position_speed_limit[i - j] = aux_speed;
        else
        {
          i -= j;
          break;
        }
      }
    }
    initialize = 1;
  }
}

final_pos_speed = position_speed_limit[final_position];
position_speed_limit[final_position] = 1;
int aux_speed, aux_position;
aux_speed = aux_position = 2;
// Saves the limits of the previous positions to slow down on time
// j <= 36 because j = 1 and the summation is (1 + 2 + 3 + ... + 8) = 36
for (int j = 1, i = final_position; j <= 36 && i - j >= 0; j++)
{
  if (j == aux_position)
  {
    if (position_speed_limit[i - j] >= aux_speed)
    {
      position_speed_limit[i - j] = aux_speed;
      aux_position += ++aux_speed;
    }
    else
    {
      if ((final_position <= 50) || (final_position <= 100 && j >= 5) || (final_position <=
          200 && j >= 10) || j >= 20)
        break;
      aux_speed = position_speed_limit[i - j] + 1;
      aux_position = j + aux_speed;
    }
  }
  else
  {
    if (position_speed_limit[i - j] >= aux_speed)
```

```
      position_speed_limit[i - j] = aux_speed;
    else
    {
      if ((final_position <= 50) || (final_position <= 100 && j >= 5) || (final_position <=
          200 && j >= 10) || j >= 20)
        break;
      aux_speed = position_speed_limit[i - j] + 1;
      aux_position = j + aux_speed;
    }
  }
}

while (position != final_position || speed != 1)
{
  // Record move
  solution_2_count++;
  solution_2.positions[move_number] = position;
  solution_2.position_speed[move_number] = speed;
  // Calculate new_speed
  for (new_speed = speed + 1, i = 1; new_speed > speed - 1 && new_speed >= 1; i++)
  {
    // If any speed limit from position to position + new_speed is smaller than new_speed
    // the speed is decreased
    if (position_speed_limit[position + i] < new_speed)
    {
      new_speed--;
      i = -1;
      continue;
    }
    // When i == new_speed the value of speed will be updated to new_speed
    if (i == new_speed)
      break;
  }
  // Speed, position and move_number values are updated
  speed = new_speed;
  position += speed;
  move_number++;
}
// Is it a solution?
position_speed_limit[final_position] = final_pos_speed;
aux_speed = aux_position = final_pos_speed + 1;
solution_2_count++;
solution_2.positions[move_number] = position;
solution_2.position_speed[move_number] = speed;

for (int j = 1, i = final_position; j <= 36 && i - j >= 0; j++)
{
  if (j == aux_position)
  {
    if (max_road_speed[i - j] >= aux_speed)
    {
      position_speed_limit[i - j] = aux_speed;
      aux_position += ++aux_speed;
    }
    else
    {
```

```
      position_speed_limit[i - j] = max_road_speed[i - j];
      aux_speed = max_road_speed[i - j] + 1;
      aux_position = j + aux_speed;
    }
  }
  else
  {
    if (max_road_speed[i - j] >= aux_speed)
      position_speed_limit[i - j] = aux_speed;
    else
    {
      position_speed_limit[i - j] = max_road_speed[i - j];
      aux_speed = max_road_speed[i - j] + 1;
      aux_position = j + aux_speed;
    }
  }
 }
 solution_2_best = solution_2;
 solution_2_best.n_moves = move_number;
 return;
}
```

Esta solução revelou ser a mais rápida entre as soluções propostas e o seu esforço é ideal, sendo este maior apenas um valor em relação ao número de movimentos.
Em relação a outras implementações esta possui uma característica que influencia de forma negativa a velocidade, que é o facto de executar um ciclo de tamanho 1 + _max_road_size_, que corresponderá a uma cópia do *array* max_road_speed, feita de forma a não alterar os valores do *array* original e trocar os valores originais da estrada nos ficheiros de output.

Capítulo 3

Resultados

Os resultados obtidos foram transformados em gráficos através de um programa em *MATLAB*.
O excerto de código que se segue é um exemplo de um programa em *MATLAB* para gerar os gráficos das várias soluções apresentadas.

```
clear;
clc;

S_102620 = load("<ficheiro texto>");
S_103511 = load("<ficheiro texto>");
S_108237 = load("<ficheiro texto>");
hf = figure();
x2 = S_102620(:,1);
y2 = S_102620(:,4);
X2 = [ x2.^2, x2.*log(x2),x2,0*x2+1 ];
w2 = pinv(X2)*y2;
format long
w2
plot_1 = plot(x2,y2,'*b',x2,X2*w2,'LineWidth',2,'b');
hold on

x0 = S_103511(:,1);
y0 = S_103511(:,4);
X0 = [ x0.^2, x0.*log(x0),x0,0*x0+1 ];
w0 = pinv(X0)*y0;
format long
w0
plot_2 = plot(x0,y0,'*g',x0,X0*w0,'LineWidth',2,'g');
hold on

x1 = S_108237(:,1);
y1 = S_108237(:,4);
X1 = [ x1.^2, x1.*log(x1),x1,0*x1+1 ];
w1 = pinv(X1)*y1;
format long
w1
plot_3 = plot(x1,y1,'*r',x1,X1*w1,'LineWidth',2,'r');

title("Performance of all solutions");
xlabel("n");
ylabel("Execution Time");
```

```
legend([plot_1,plot_2,plot_3],"102620-values","102620","103511-values","103511","108237-
    values","108237")

print(hf, "all_solutions", "-dpdflatexstandalone");
system("pdflatex all_solutions");
open all_solutions.pdf
```

3.1 Resultados das soluções implementadas

Para comparar o desempenho de todas as alternativas de resolução, foram agrupados os seus dados (tempo de execução e tamanho da estrada), produzindo o seguinte gráfico:

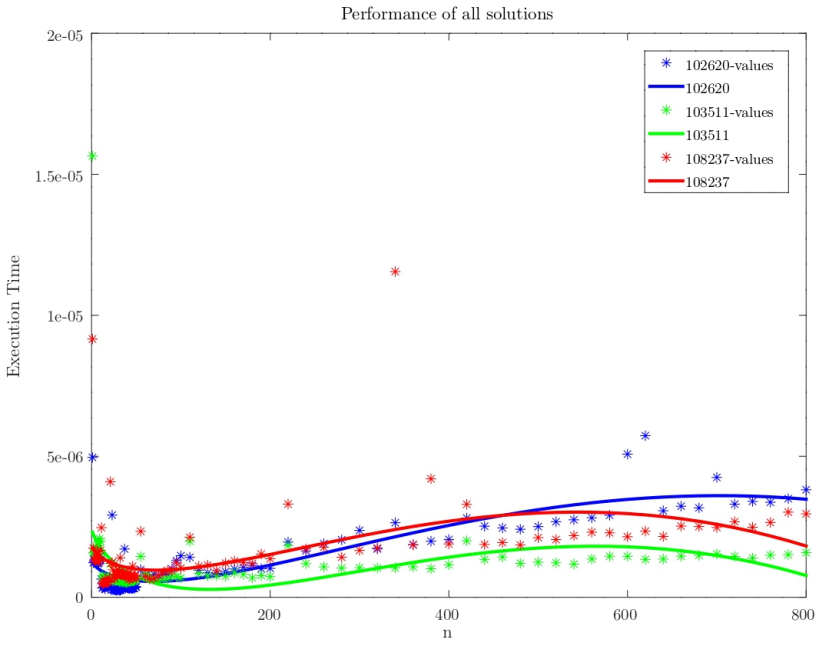

Figura 3.1: Gráfico de desempenho de todas as soluções encontradas.

Através da análise do gráfico, podemos verificar o aumento do tempo de execução em função do tamanho da estrada e, desta forma, comprovar a complexidade computacional de cada método.

3.2 Resultados da solução fornecida

Para satisfazer qualquer curiosidade sobre qual será o tempo de execução quando o $n = 800$, visto que esta solução tem um péssimo desempenho, foi produzido o seguinte gráfico:

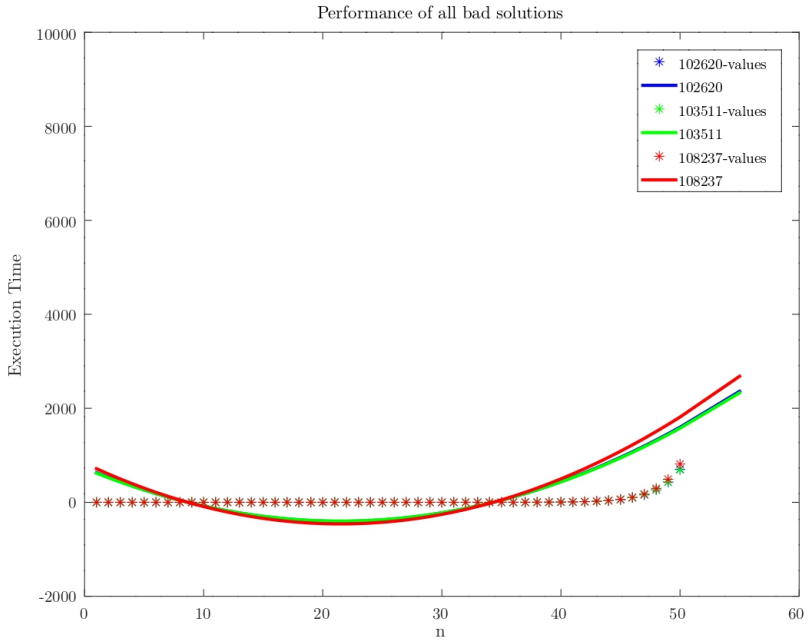

Figura 3.2: Gráfico de execução do método Brute Force para difirentes execuções2.1.

No gráfico, podemos verificar que o tempo de execução cresce consideravelmente e, portanto, só foram recolhidos dados até que o tempo de execução excedesse 1 hora para resolver um determinado problema. Recorrendo ao *MATLAB* foi encontrada a equação para determinar o tempo de execução. Neste gráfico, temos várias parábolas, no entanto, só será considerada a que apresenta pior desempenho, que será a que utiliza <seed>, 108237 1.3.

A equação é a seguinte:

$$\log T(n) = \log(0.5096n) - \log 18.88\ (s) \Leftrightarrow$$
$$\Leftrightarrow T(n) = e^{0.5096n - 18.88}\ (s)$$

Tendo a equação, pode-se calcular o tempo de execução para uma estrada com 800 segmentos.

$$T(800) = 7,14 \times 10^{168}(s)$$

$$\frac{7.14 \times 10^{168}}{3600 \times 24 \times 365.25} \approx 2.26 \times 10^{161}\ (anos)$$

Portanto, a função dada demoraria 2.26×10^{161} anos a encontrar a solução.

3.3 Algumas soluções

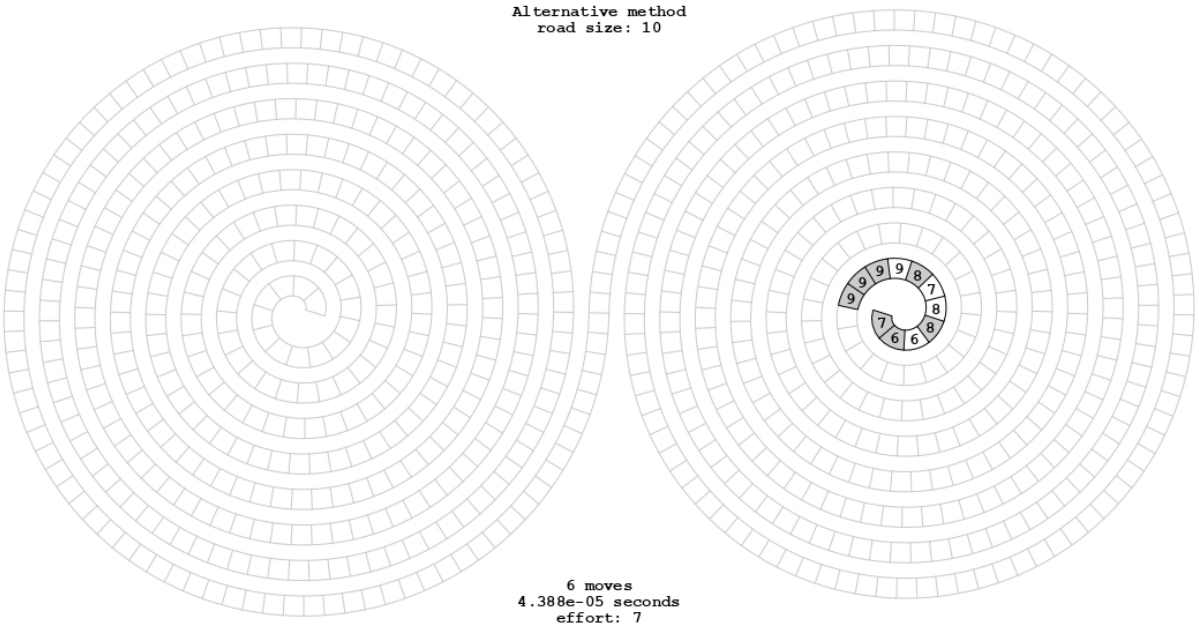

Figura 3.3: Mapa da solução do método recursivo 2.2 com 10 segmentos, e a *seed* 102620 1.3.

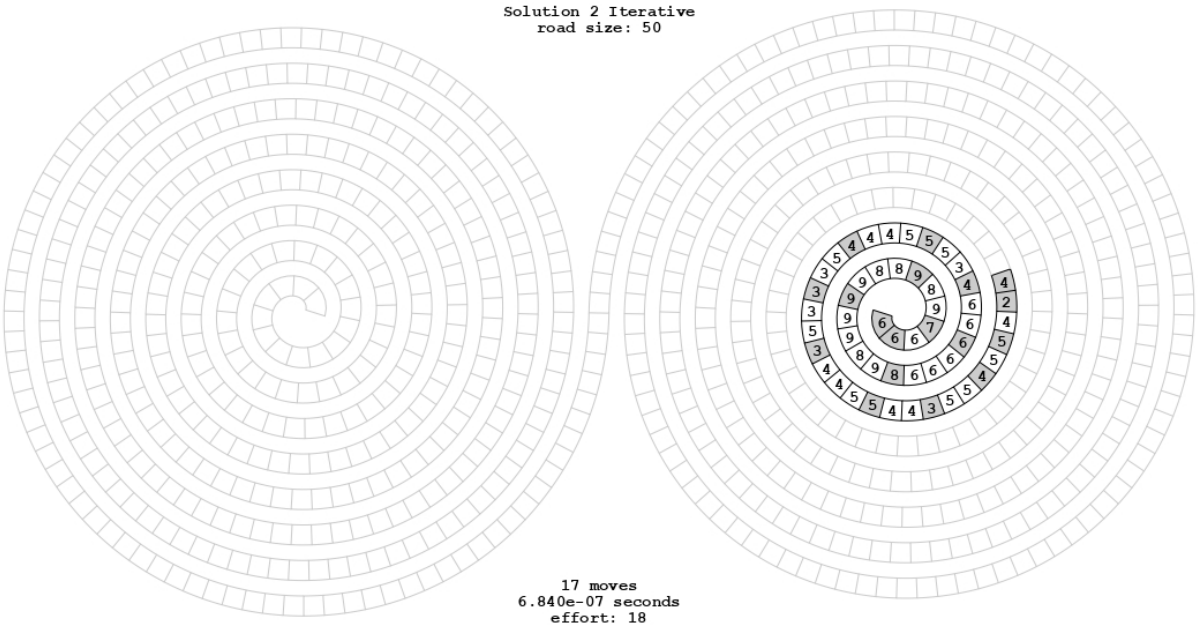

Figura 3.4: Mapa da solução do método iterativo 2.4 com 50 segmentos, e a *seed* 103511 1.3.

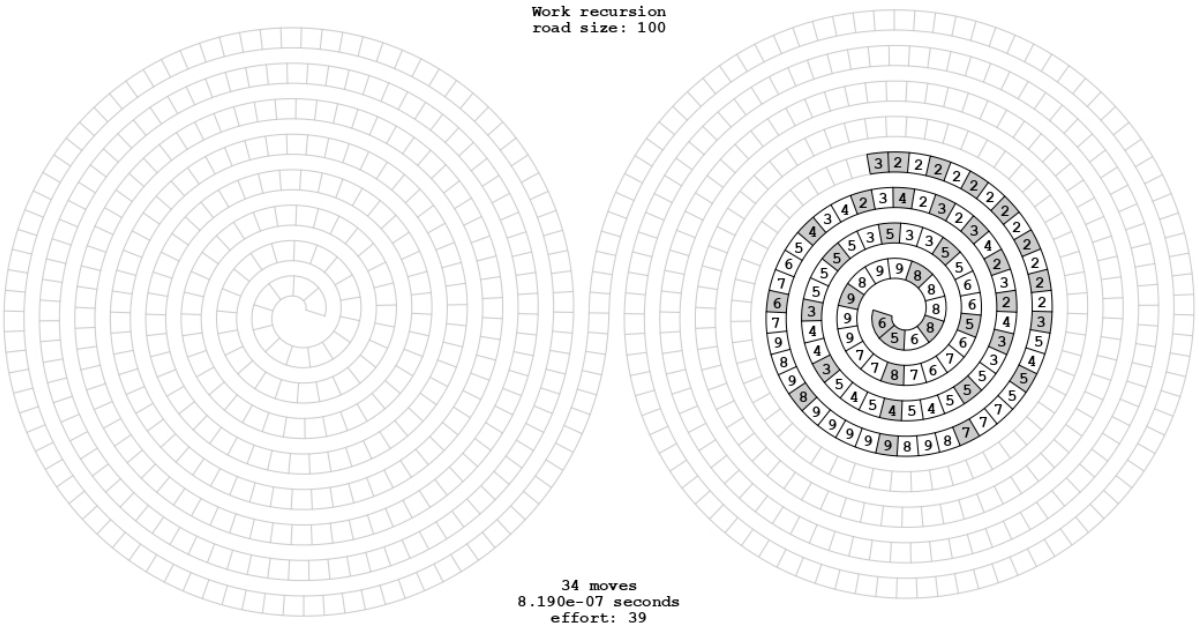

Figura 3.5: Mapa da solução do método aprimorado 2.3 com 100 segmentos, e a *seed* 108237 1.3.

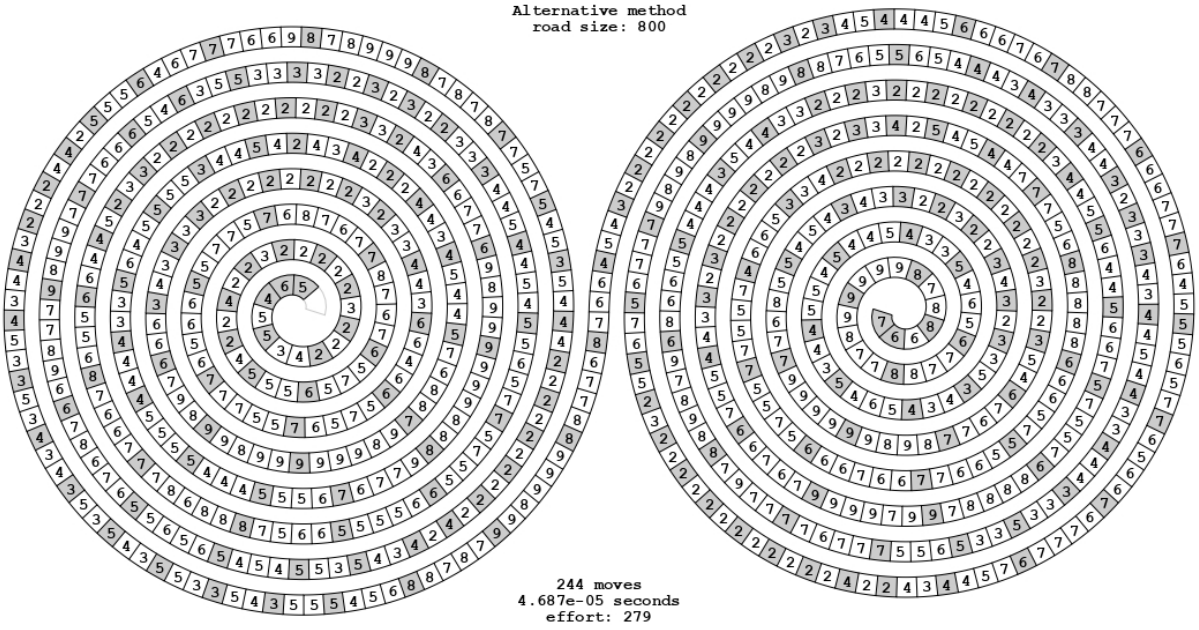

Figura 3.6: Mapa da solução do método recursivo 2.2 com 800 segmentos, e a *seed* 102620 1.3.

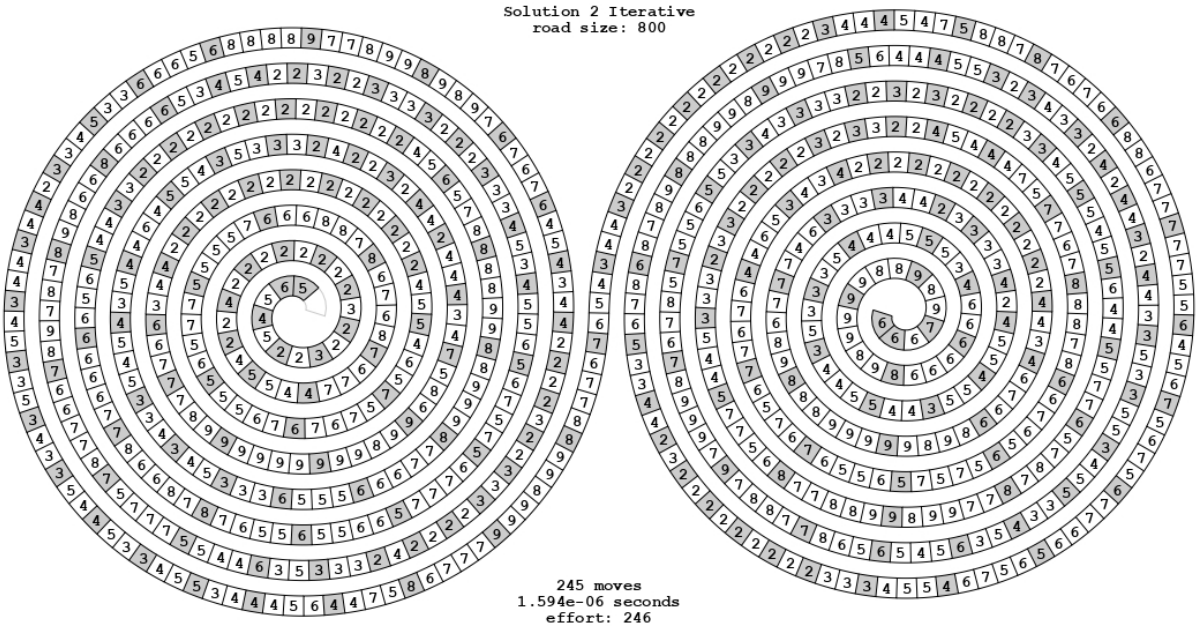

Figura 3.7: Mapa da solução do método iterativo 2.4 com 800 segmentos, e a *seed* 103511 1.3.

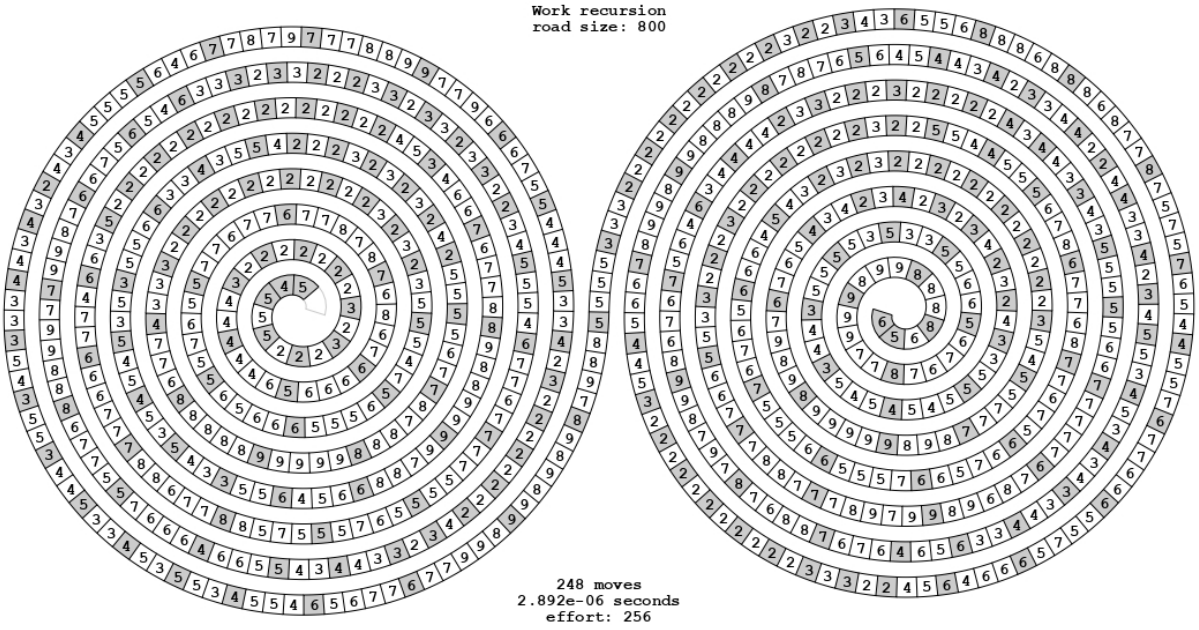

Figura 3.8: Mapa da solução do método aprimorado 2.3 com 800 segmentos, e a *seed* 108237 1.3.

Capítulo 4

Conclusão

Este relatório introduziu várias abordagens ao problema 'Speed Run'. Não só se aprendeu a olhar para o problema de maneiras diferentes, como também a analisar as diferenças de cada método, os pontos fracos e fortes.
Foram estudadas várias formas de resolver este problema, tais como: um método totalmente recursivo 2.2, outro totalmente iterativo 2.4 e, por fim, um método que desenvolve a solução fornecida e melhora o seu tempo de execução 2.3.
Recorrendo ao método totalmente recursivo, podemos verificar que é o mais lento de todos, no entanto apresenta uma solução bastante compacta do problema.
Os outros dois métodos implementam o mesmo raciocínio, mas de formas diferentes, um recorre à recursão, o outro é totalmente iterativo e, a nível de desempenho, é o melhor.
Em suma, todos os algoritmos apresentam soluções válidas, uns mais compactos que outros, com precisões e tempos de execução diferentes.

Bibliografia

[1] *SILVA, Tomás Oliveira e.* ***Lecture notes****: Algorithms and Data Structures (AED — Algoritmos e Estruturas de Dados) LEC, LEI, LECI, 2022/2023.*

[2] *Jeffrey J. McConnell -* ***Analysis of Algorithms_ An Active Learning Approach*** *-Jones & Bartlett Publishers (2001)*

[3] *https://www.mathworks.com/help/matlab/getting-started-with-matlab.html?s_tid=CRUX_lftnav*