Algoritmos e Estruturas de Dados

speed_run

João Catarino
NMec: 93096

Rúben Garrido
NMec: 107927

Nuno Vieira
NMec: 107283

novembro de 2022

1 Introdução

Este trabalho tem como objetivo desenvolver algoritmos que sejam capazes de encontrar o menor número de passos necessários para resolver o seguinte problema:

2 Soluções

A solução original utiliza uma função recursiva para verificar todas as combinações de passos a todas as velocidades possíveis dentro dos limites de cada casa, guardando sempre a melhor solução encontrada até ao momento. Esta possui um tempo de execução da ordem das centenas de anos para a resolução do caso $n = 800$. Com vista em obter a solução em tempo aceitável, criámos duas soluções capazes de o fazer na ordem dos microsegundos. A primeira altera apenas ligeiramente a solução original. A segunda é não recursiva e utiliza o princípio da solução anterior com algumas optimizações.

2.1 Original

A solução dada é uma solução recursiva que itera sobre todas as possibilidades de percurso, guardando a melhor solução até ao momento.

Ela parte das soluções de maior número de passos (menor velocidade por passo) para as de menor número de passos (maior velocidade por passo) dentro das regras do problema.

2.2 Original Improved

Esta solução introduz duas pequenas mudanças no código original, que no entanto geram alterações significativas no seu comportamento.

Em primeiro lugar, é introduzido controlo de fluxo através de um valor de retorno *boolean* (implementado como inteiro).

```
static void solution_1_recursion(int move_number,int position,int
    speed,int final_position)
{
  int i,new_speed;
```

Se uma sequência de chamadas recursivas chegar a uma solução, a última chamada retorna 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++)
```

Se for este o caso, as chamadas anteriores retornarão também 1. Isto significa que o programa grava apenas a primeira solução que encontrar.

```
static void solve_1(int final_position)
{
  if(final_position < 1 || final_position > _max_road_size_)
```

Tendo isto, torna-se imperativo que o programa encontre a melhor solução possível à primeira tentativa. Isto implica que o "carro" se mova o mais rápido possível em qualquer passo para obter o menor número de passos. A segunda alteração garante essa condição ao fazer com que o programa itere desde as maiores velocidades para as menores.

```
    {
      for(i = 0;i <= new_speed && new_speed <= max_road_speed[
   position + i];i++)
```

2.3 Advance and retreat

Este algoritmo foi feito de raiz. Tem como princípio tentar a qualquer passo avançar com a velocidade mais alta. Em cada passo, a escolha de velocidade é representada por um incremento (-1, 0 ou 1). O programa começa sempre por tentar o maior incremento. Para verificar passos possíveis, utilizam-se os seguintes métodos:

Calcular a distância de paragem para cada velocidade possível em cada passo para evitar correr para além do fim do trajeto:

```
//  Sum 1 to n: stopping distance going at speed n
static int sum1ton(int n)
{
  return n * (n + 1) / 2;
}

//  Checks if it is possible to stop before or at finalpos going at
    speed from pos
static int valstop(int pos, int speed, int finalpos)
{
  return (pos + sum1ton(speed)) <= finalpos;
}
```

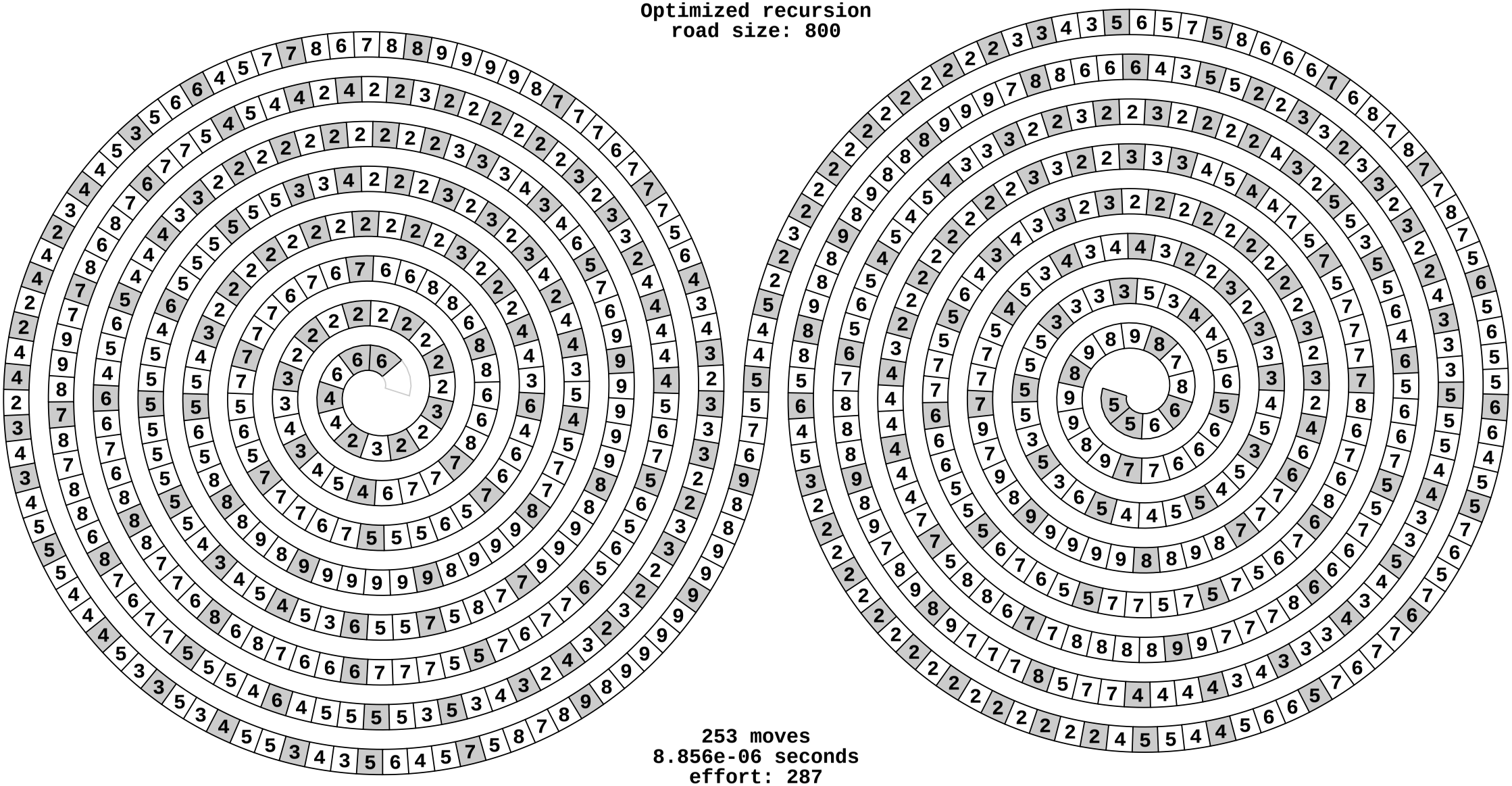
Verificar, a partir do incremento mais alto, se uma "passada" não quebra os limites de velocidade das casas pelas quais passaria:

```
// Checks if step from pos at speed breaks any of the intermediary
   speed limits
static int valstep(int pos, int speed)
{
  int end = pos + speed;
  for (; pos <= end && speed <= max_road_speed[pos]; pos++)
    ;
  return (pos > end);
}
```

Feito o passo, a escolha de incremento é guardada num *array* na posição associada ao número do passo. Desta forma, este *array* guarda as escolhas feitas até ao passo atual. Quando um passo é impossível de executar a qualquer das velocidades possíveis nesse passo, o algoritmo recua um passo e tenta reduzir o incremento de velocidade até que consiga avançar novamente. Não sendo possível avançar com nenhum dos incrementos, o programa recua novamente, e assim sucessivamente.

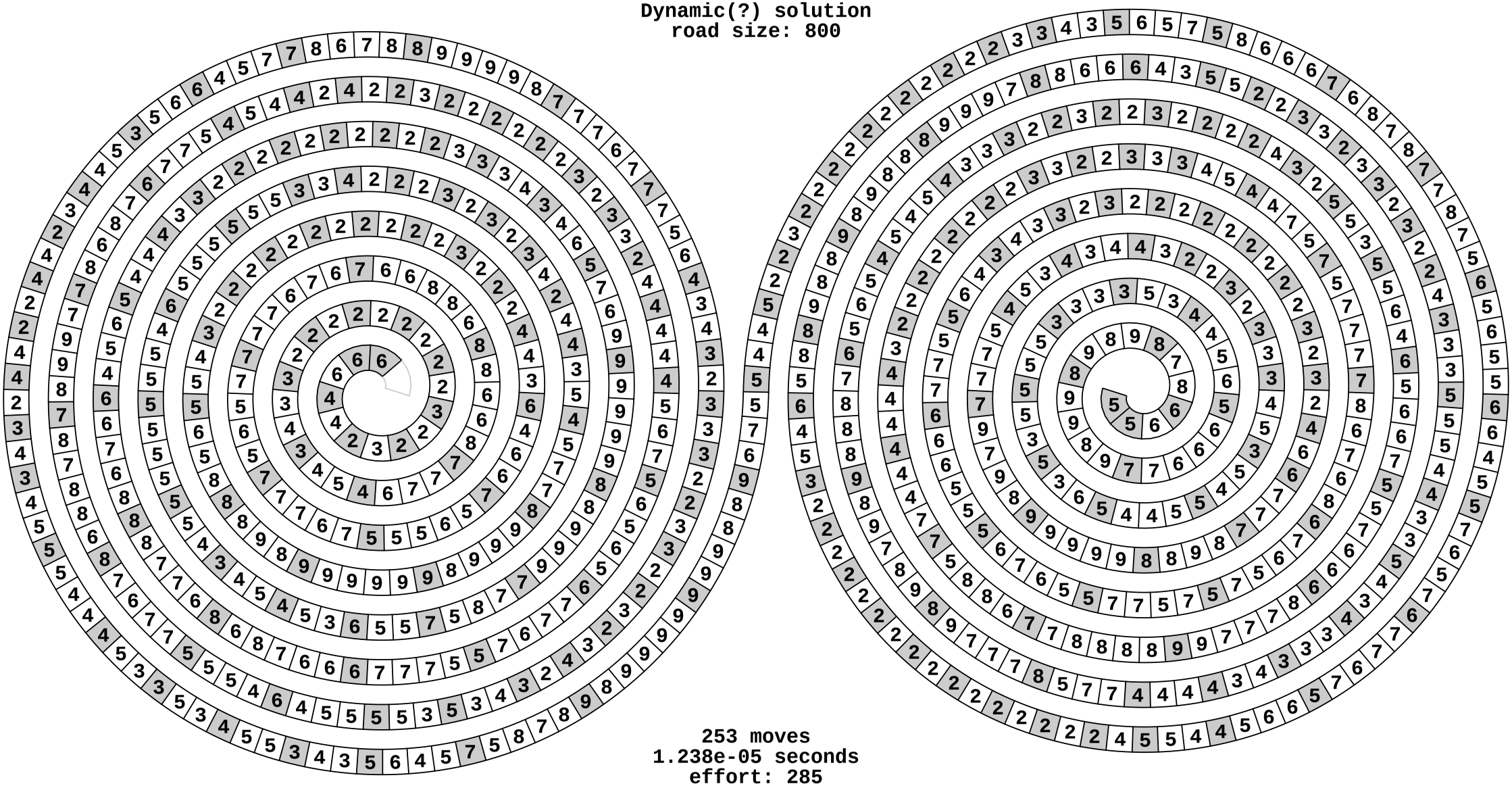
3 PDFs de soluções

Optimized recursion
road size: 800



253 moves
8.856e-06 seconds
effort: 287

Dynamic(?) solution
road size: 800



253 moves
1.238e-05 seconds
effort: 285

4 Tempos de execução e estimativas

4.1 Original

Devido à complexidade do algoritmo, não foi possível obter tempos de execução para $n = 800$, pelo que foi necessário recorrer a estimativas, utilizando o MATLAB.

Em primeiro lugar, procedeu-se ao ajuste de uma função exponencial aos tempos obtidos até $n = 50$. Dado o enorme salto existente de $n = 50$ para $n = 55$, a função a ajustar teria de ser do tipo $a \times e^{bx}$. Assim, utilizando a Curve Fitting Toolbox do MATLAB, obteve-se, para o percurso gerado com o número mecanográfico 107927, a seguinte equação: $y = (6.231 \times 10^{-8}) \times e^{0.4869x}$.

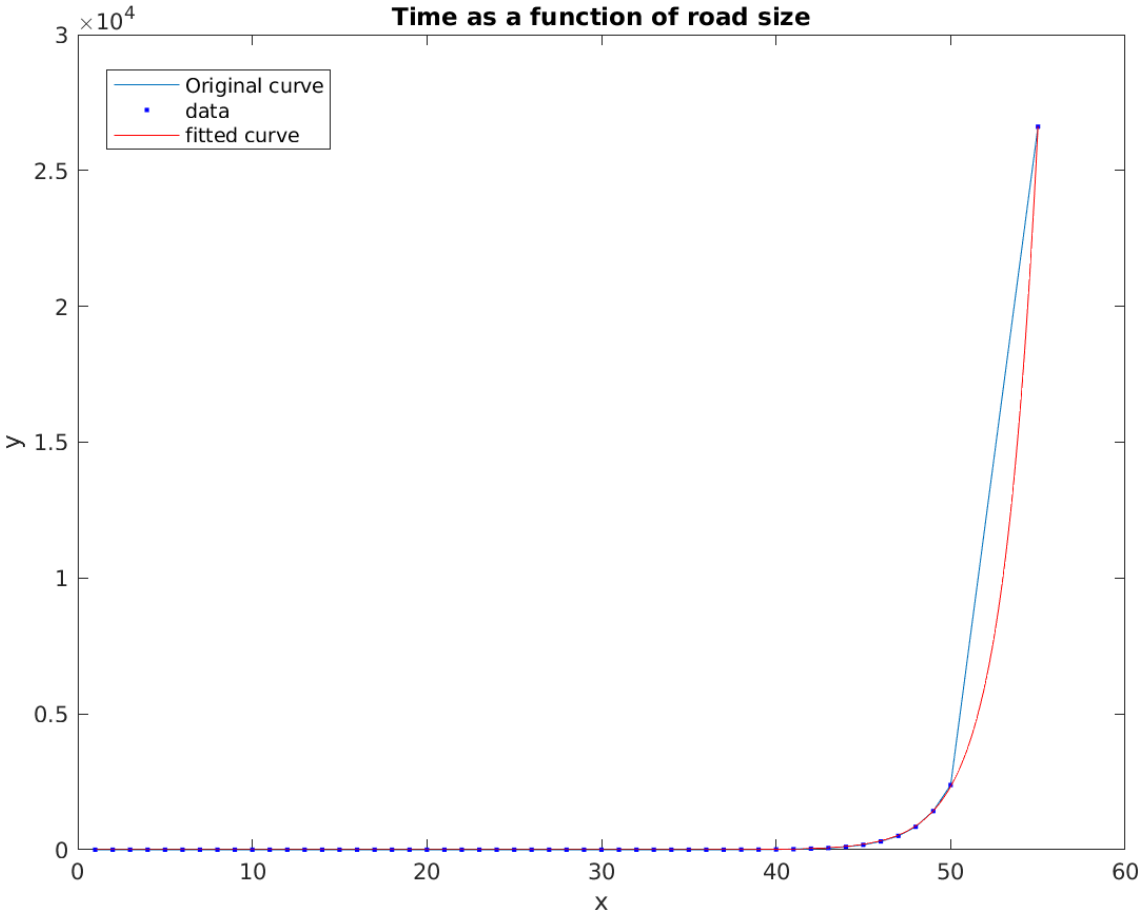


Figure 1: Gráfico com os tempos de execução para o número mecanográfico 107927 e a função exponencial ajustada

Possuindo uma função exponencial que se ajusta aos valores obtidos, foi possível, portanto, estimar os tempos de execução para $n = 800$. Deste modo, o tempo de execução previsto é de $9.233\,97 \times 10^{161}$ segundos, o que equivale a, aproximadamente, $2.968\,74 \times 10^{154}$ anos.

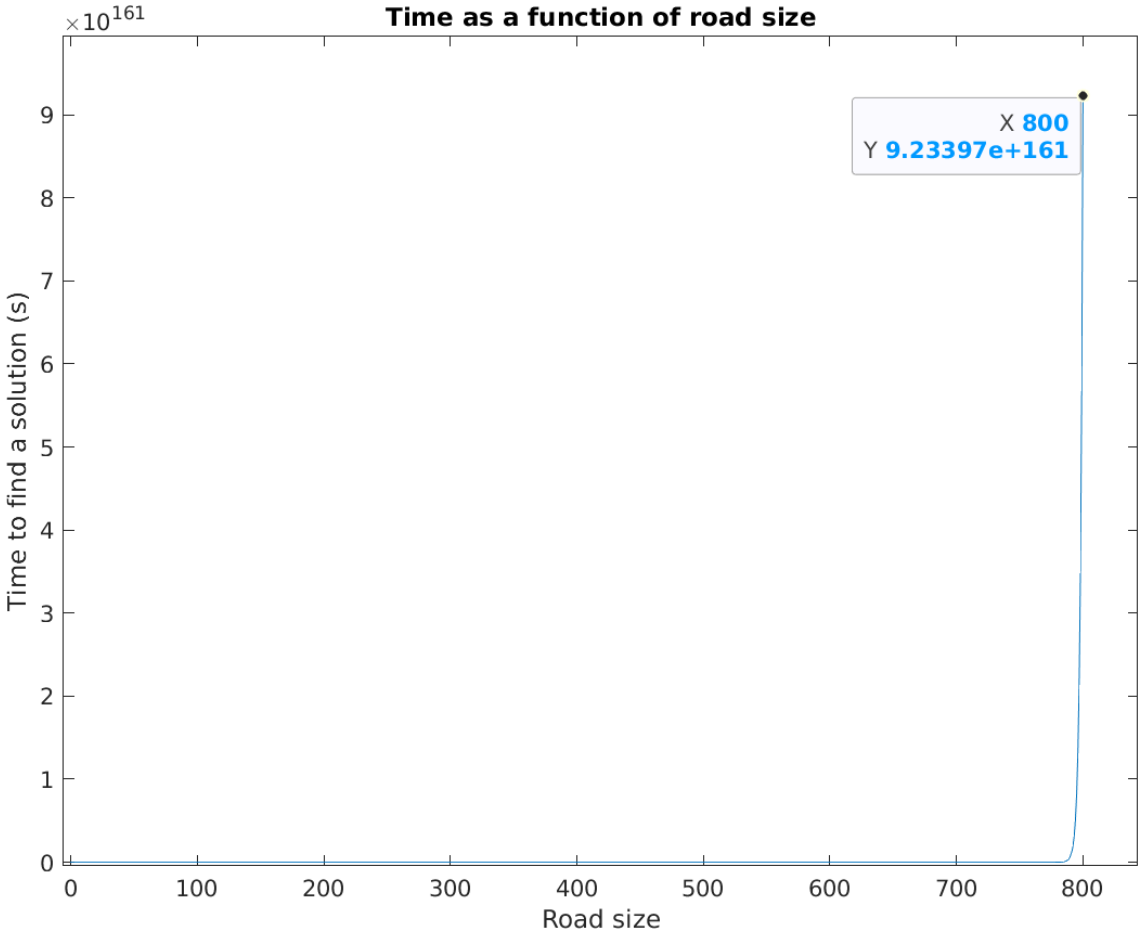


Figure 2: Tempo de execução estimado para $n = 800$, utilizando a solução original

5 Código

5.1 Original Improved

```
// Improved version of original recursive func

static solution_t solution_2;
static double solution_2_elapsed_time; // time it took to solve the
    problem
static unsigned long solution_2_count; // effort dispended solving
   the problem

static int solution_2_recursion(int move_number,int position,int
   speed,int final_position)
{
  int i,new_speed;

  // record move
  solution_2_count++;
  solution_2.positions[move_number] = position;
  // Solution found
  if(position == final_position && speed == 1)
  {
    solution_2.n_moves = move_number;
    return 1;
  }
  // Try all legal speeds. Fastest first
  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)
        if (solution_2_recursion(move_number + 1,position +
   new_speed,new_speed,final_position))
        return 1;
    }
  return 0;
}

static void solve_2(int final_position)
{
  if(final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr,"solve_2: bad final_position\n");
    exit(1);
  }
  solution_2_elapsed_time = cpu_time();
  solution_2_count = 0ul;
  solution_2.n_moves = final_position + 100;
  solution_2_recursion(0,0,0,final_position);
  solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}
```

5.2 Advance and retreat

```
//

static solution_t solution_3;
static double solution_3_elapsed_time; // time it took to solve the
    problem
static unsigned long solution_3_count; // effort dispended solving
   the problem

//  Sum 1 to n: stopping distance going at speed n
static int sum1ton(int n)
{
  return n * (n + 1) / 2;
}

//  Checks if it is possible to stop before or at finalpos going at
    speed from pos
static int valstop(int pos, int speed, int finalpos)
{
  return (pos + sum1ton(speed)) <= finalpos;
}

// Checks if step from pos at speed breaks any of the intermediary
   speed limits
static int valstep(int pos, int speed)
{
  int end = pos + speed;
  for (; pos <= end && speed <= max_road_speed[pos]; pos++)
    ;
  return (pos > end);
}

/*    The solution works by increasing the speed as much as
   possible
  without overstepping the finalpos or breaking any speed limits.

    In a move, if any of the those two checks fail, the program
   attemps to
  mantain or decrease the speed of the car. If the two checks don't
    work
  for any of the possible speeds, the program moves back one step
   and
  retries it with the previous speed reduce by one.
*/

static void solution_3_dynamic(int final_position)
{
  // Current move
  #define move solution_3.n_moves

  // Car position
  #define pos solution_3.positions[move]
  #define nextpos solution_3.positions[move+1]

  // Current speed "choice"
```

```
  #define incmax incmaxes[move]

  // Stores the "choice" taken at every move (slowdown, cruise,
   accel)
  int incmaxes[1 + final_position];

  // Current speed
  int speed = 0;

  pos = 0;
  move = 0;
  incmax = 1;
mainloop:
  while (pos != final_position)
  {
    for (; incmax >= -1; incmax--)
    {
      if (valstop(pos, speed + incmax, final_position) && valstep(
   pos, speed + incmax))
        {
          solution_3_count++;
          // Found valid step, take it
          speed += incmax;
          nextpos = pos + speed;
          move++;
          incmax = 1;
          // Jump to main loop to see if it reaches the
          // end or try the next one, avoiding the fail
          // state after the two fors
          goto mainloop;
        }
    }
    /*
      There are no possible steps in the current move,
      so lets try the previous move with it's speed reduced by one

      Move the program back one move by reverting
      the prev speed change and by decrementing the
      move count. Then, choose the next smaller speed.
    */
    move--;
    speed -= incmax;
    incmax--;
  }
}

static void solve_3(int final_position)
{
  if(final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr,"solve_3: bad final_position\n");
    exit(1);
  }
  solution_3_elapsed_time = cpu_time();
  solution_3_count = 0ul;
  solution_3_dynamic(final_position);
```

```
  solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}
//
// example of the slides
```

5.3 Ajuste dos dados obtidos utilizando o MATLAB

```
table = load("Path_to_file");
n = table(:,1); % Road size
t = table(:,2); % Time to find a solution

% Original data plot
figure(1)
plot(n,t)
legend('Original curve')
title('Time as a function of road size')
xlabel('Road size')
ylabel('Time to find a solution (s)')

% Curve fitting
hold on
f = fit(n,t,'exp1'); % Alternatively, one can launch the Curve
   Fitting Toolbox
plot(f,n,t)

% Extend fitted curve on x-axis to [0,800]
x_fit = 0:800;
coefficients = coeffvalues(f); % Get fitted curve coefficients
y_fit = coefficients(1) * exp(coefficients(2)*x_fit); % Equation of
    fitted curve: a*exp(b*x)
figure(2)
plot(x_fit,y_fit)
xlim([0 850]) % Graph would look empty (due to exponential values)
   if no x-lim was applied
title('Time as a function of road size')
xlabel('Road size')
ylabel('Time to find a solution (s)')
```