Licenciatura em Engenharia Informática

Algoritmos e Estruturas de Dados

speed_run

João Catarino
NMec: 93096
%

Rúben Garrido
NMec: 107927
%

Nuno Vieira
NMec: 107283
%

5 de dezembro de 2022

Conteúdo

1 Introdução

Este trabalho consiste em desenvolver algoritmos capazes de resolver o seguinte problema:

Uma estrada encontra-se dividida em vários segmentos de igual tamanho, cada um com uma velocidade pré-definida. O objetivo deste problema é encontrar o menor número de passos necessários para percorrer a estrada, começando no primeiro segmento e terminando no último, com velocidade 1.

Em cada passo, é possível avançar para os segmentos seguintes, com velocidade igual à velocidade de chegada ao segmento atual ou inferior/superior à velocidade atual em 1 unidade. Contudo, em cada iteração, deve-se verificar se a velocidade de partida é igual ou inferior à velocidade dos segmentos seguintes. Caso a velocidade atual seja 1, não é possível reduzir a velocidade.

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 com ordem de grandeza 10^{154} (em anos) para a resolução do caso $n = 800$. Com vista a obter a solução em tempo aceitável, foram criadas três soluções capazes de o fazer na ordem dos microsegundos. A primeira altera apenas ligeiramente a solução original, enquanto que a segunda é não recursiva (dinâmica) e utiliza o princípio da solução anterior com algumas optimizações. A terceira combina as duas soluções anteriores.

2.1 Original

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

Esta 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.

Pelo facto de a solução ser recursiva, à medida que o tamanho do problema aumenta, o número de chamadas recursivas também aumenta, o que leva a um tempo de execução exponencial. Por exemplo, a partir de $n = 35$, o tempo de execução é superior a 1 segundo, pelo que esta não é, portanto, uma solução viável para o problema.

2.2 Original Improved

A presente 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).

```
        return 1;
    }
  return 0;
```

Se uma sequência de chamadas recursivas chegar a uma solução, a última chamada retorna 1.

```
        if (solution_2_recursion(move_number + 1,position +
   new_speed,new_speed,final_position))
        return 1;
```

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

```
  // Solution found
  if(position == final_position && speed == 1)
  {
    solution_2.n_moves = move_number;
    return 1;
  }
```

Deste modo, torna-se imperativo que o programa encontre a melhor solução possível à primeira tentativa. Esta condição 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

O algoritmo Advance and retreat foi feito de raiz. O seu princípio consiste em 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);
}
```

Realizado o passo, a escolha do incremento é guardada num *array*, na posição associada ao número do passo. Desta forma, este *array* guarda as escolhas feitas até ao passo atual.

```
// 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;
```

Quando um passo é impossível de executar a qualquer das velocidades possíveis, 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.

```
    move--;
    speed -= incmax;
    incmax--;
```

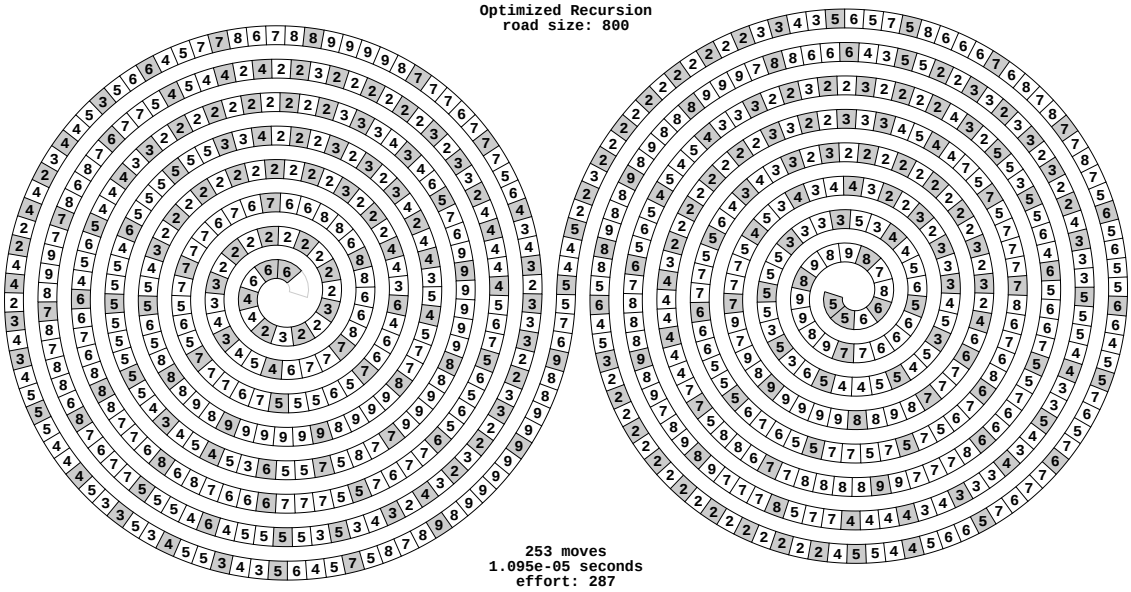
2.4 Combined

Com esta solução, tentou-se aproximar o comportamento do algoritmo *Original Improved* à solução *Advance and Retreat*, com o objetivo de observar diferenças entre uma solução dinâmica e recursiva com lógica semelhante. Para tal, ao invés de, no ciclo `for`, existir uma verificação de *overstep* para passo, utilizou-se a função `valstop`, presente no algoritmo *Advance and retreat*.

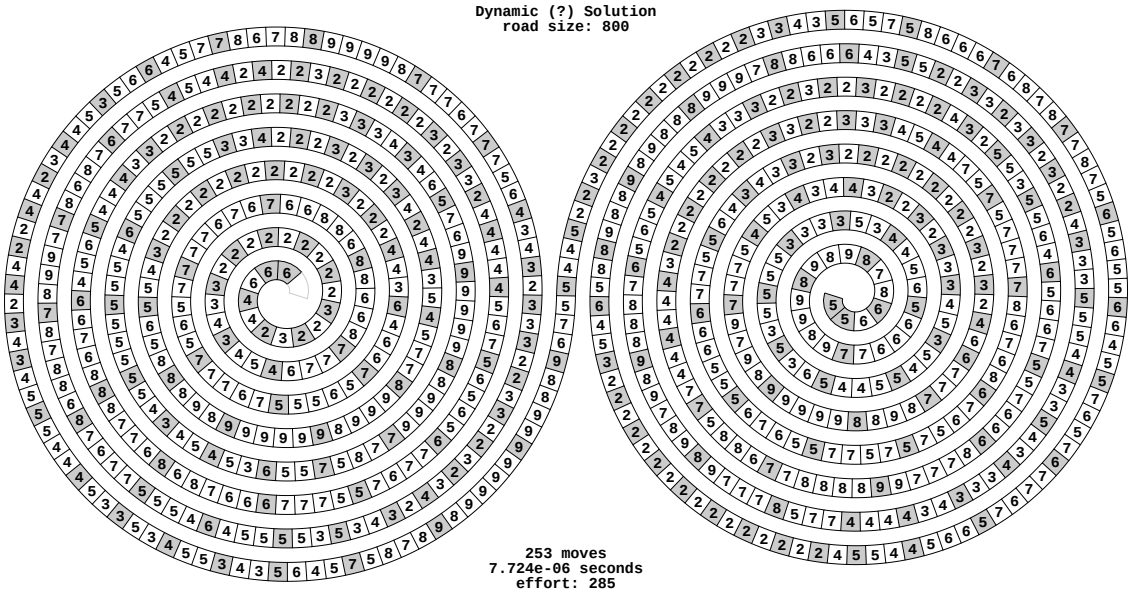
```
for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--)
  if(new_speed >= 1 && new_speed <= _max_road_speed_ && valstop(
 position, new_speed, final_position))
  {
    for(i = 0;i <= new_speed && new_speed <= max_road_speed[
 position + i];i++)
      ;
    if(i > new_speed)
      if (solution_4_recursion(move_number + 1,position +
 new_speed,new_speed,final_position))
      return 1;
  }
return 0;
```

3 PDFs de soluções

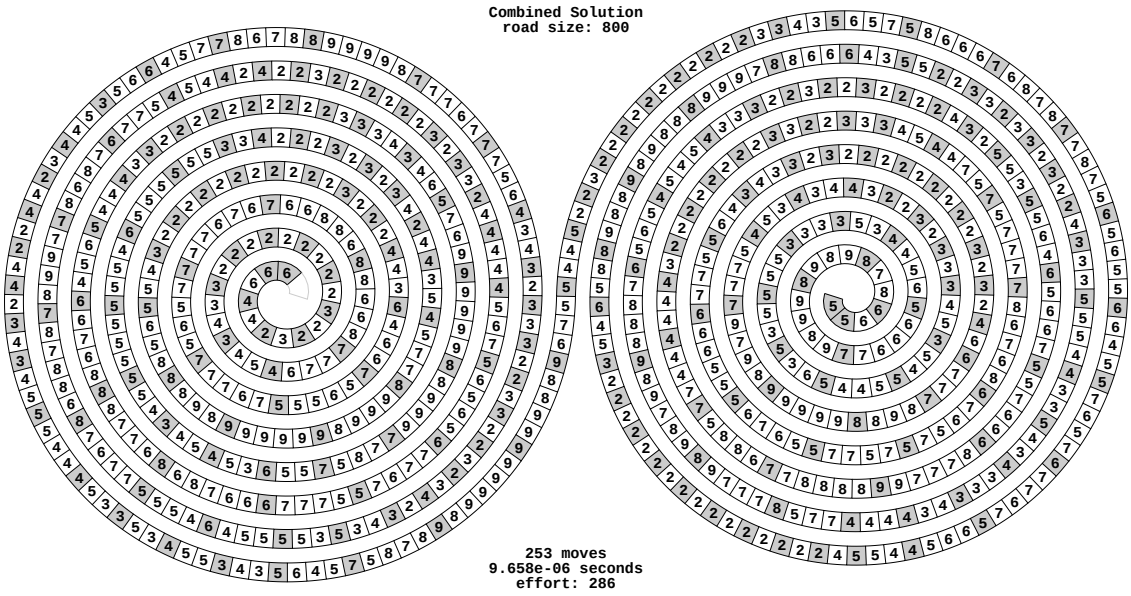
3.1 Original Improved



3.2 Advance and retreat



3.3 Combined


Combined Solution
road size: 800
253 moves
9.658e-06 seconds
effort: 286

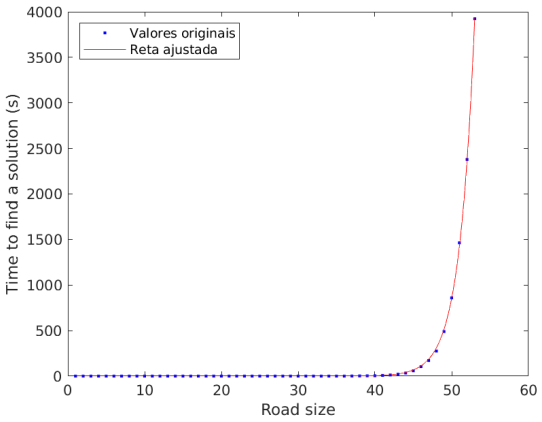
4 Tempos de execução e estimativas

4.1 Original

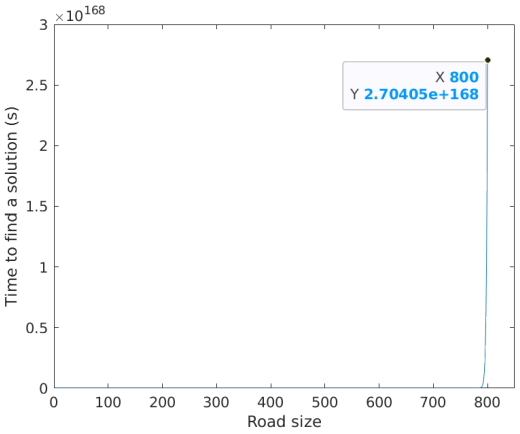
Devido à complexidade do algoritmo, não foi possível obter tempos de execução para $n = 800$, pelo que foi necessário recorrer a estimativas, utilizando o MATLAB.

Procedeu-se ao ajuste de uma curva do tipo $a \times e^{bx}$ aos tempos obtidos até uma hora, utilizando a Curve Fitting Toolbox do MATLAB. De seguida, foi calculado o tempo de execução para $n = 800$, através da equação exponencial determinada anteriormente. A complexidade computacional é, portanto, $O(e^n)$.

Para o número mecanográfico 107927, obteve-se a equação $y = (7.942 \times 10^{-9}) \times e^{0.5081x}$ e uma previsão de 2.704×10^{168} segundos para $n = 800$, o que equivale a, aproximadamente, 8.694×10^{160} anos.



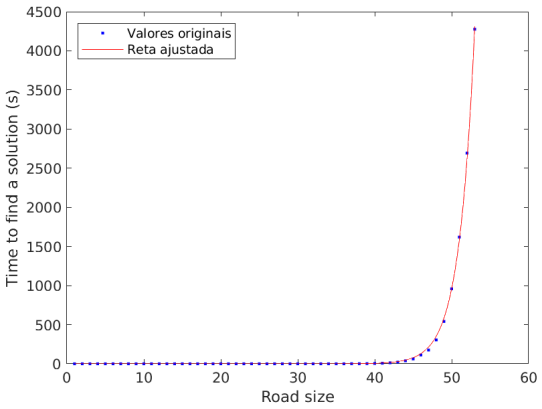
(a) Tempos de execução obtidos e função exponencial ajustada.



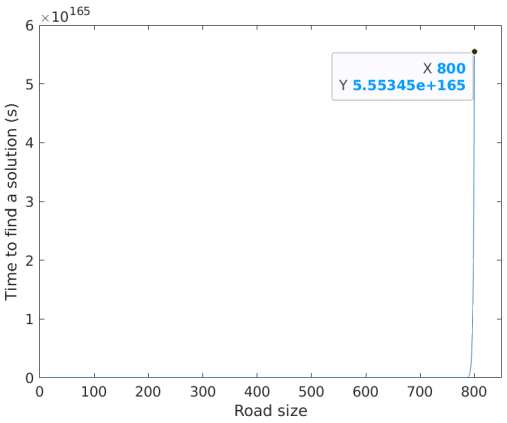
(b) Tempo de execução estimado, recorrendo à função ajudada em (a), para $n = 800$.

Figura 1: Dados do algoritmo original, para o número mecanográfico 107927.

Para o número mecanográfico 93096, obteve-se a equação $y = (1.36 \times 10^{-8}) \times e^{0.4997x}$ e uma previsão de 5.553×10^{165} segundos para $n = 800$, o que equivale a, aproximadamente, 1.785×10^{158} anos.



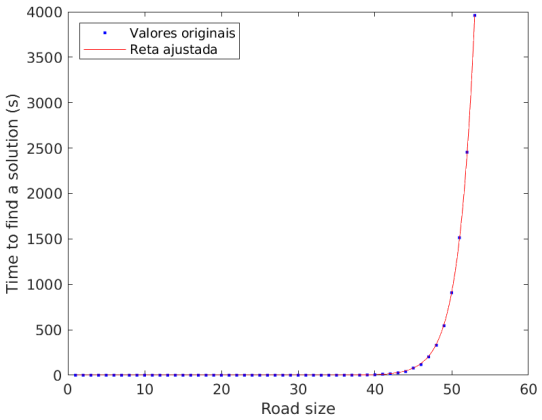
(a) Tempos de execução obtidos e função exponencial ajustada.



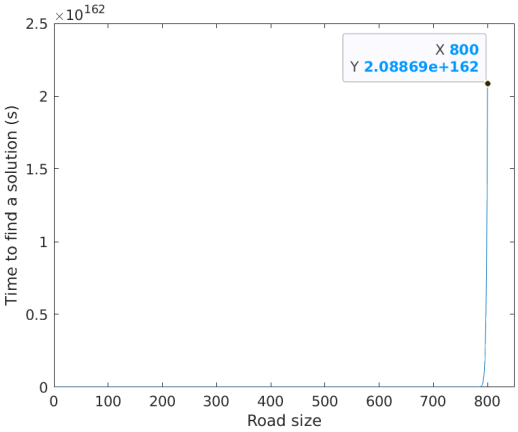
(b) Tempo de execução estimado, recorrendo à função ajudada em (a), para $n = 800$.

Figura 2: Dados do algoritmo original, para o número mecanográfico 93096.

Para o número mecanográfico 107283, obteve-se a equação $y = (2.177 \times 10^{-8}) \times e^{0.4892x}$ e uma previsão de 2.089×10^{162} segundos para $n = 800$, o que equivale a, aproximadamente, 6.715×10^{154} anos.



(a) Tempos de execução obtidos e função exponencial ajustada.



(b) Tempo de execução estimado, recorrendo à função ajudada em (a), para $n = 800$.

Figura 3: Dados do algoritmo original, para o número mecanográfico 107283.

4.2 Original Improved

Pelo facto de o algoritmo Original Improved efetuar cálculos em tempos com ordem de grandeza 10^{-6}, foi possível obter um tempo de execução para $n = 800$.

Contudo, os valores até $n = 60$ não são fiáveis para ajustar uma reta, pois são constituídos, na sua maior parte, por ruído. Este é causado pelo script de medição de tempo que, devido a questões de arquitetura, não consegue reportar valores temporais tão baixos.

A reta de ajuste obtida tem equação $y = (1.069 \times 10^{-8})x + (9.515 \times 10^{-7})$, pelo que este algoritmo tem uma complexidade computacional de $O(n)$.

As alterações indicadas na secção 2.2 são a explicação para uma diferença tão significativa nos resultados, quando comparado com o algoritmo Original.

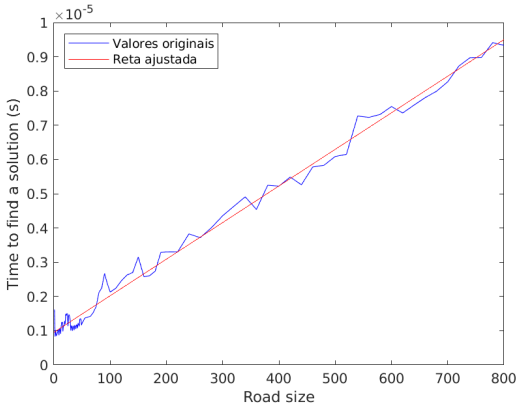


Figura 4: Tempos de execução e função exponencial ajustada.

4.3 Advance and retreat

Tal como a solução anterior, o algoritmo Advance and retreat efetua cálculos em tempos bastante baixos (ordem de grandeza 10^{-6}), até $n = 800$. Assim, os valores iniciais de n constituem igualmente ruído.

A reta de ajuste tem equação $y = (7.464 \times 10^{-9})x + (7.920 \times 10^{-7})$.

Este algoritmo é o mais eficiente, já que, apesar de a complexidade computacional ser $O(n)$, o declive da reta ajustada é o mais baixo (ordem de grandeza 10^{-9}), quando comparado com os declives dos restantes (ordem de grandeza 10^{-8}). A não recursão do algoritmo é, provavelmente, um fator decisivo que contribui para a sua eficiência.

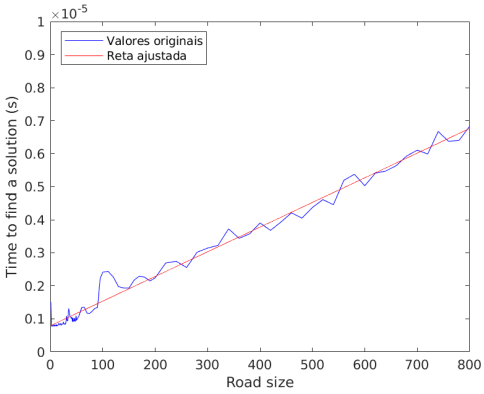


Figura 5: Tempos de execução e função exponencial ajustada.

4.4 Combined

Pela mesma razão das soluções anteriores, o algoritmo Combined também efetua cálculos em tempos bastante baixos, pelo que os valores iniciais de n são considerados, mais uma vez, ruído.

A reta de ajuste tem equação $y = (1.081 \times 10^{-8})x + (7.594 \times 10^{-7})$.

Apesar da alteração efetuada neste algoritmo, o declive da reta de ajuste é aproximadamente igual ao da solução Original Improved (excluem-se pequenas diferenças devido à presença de fatores externos). Assim, este algoritmo tem a mesma eficiência e a mesma complexidade computacional do algoritmo que tem por base ($O(n)$). A explicação para a obtenção destes resultados deve-se ao facto de a complexidade computacional da expressão *boolean* e da função `valstop` ser a mesma, o que não produz qualquer diferença.

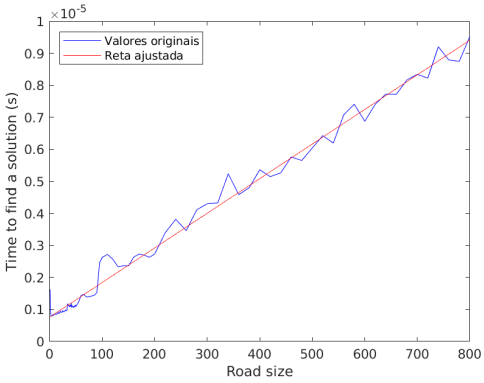


Figura 6: Tempos de execução e função exponencial ajustada.

5 Conclusão

Neste trabalho foi possível estudar a evolução temporal da solução original e determinar uma estimativa de resolução para o caso final. Foi também possível elaborar dois algoritmos capazes de resolver o caso final num intervalo de tempo na ordem dos microsegundos.

Através dos resultados obtidos, foi possível concluir que o algoritmo Advance and retreat é o mais eficiente, uma vez que o declive da reta de ajuste é o mais baixo. Uma das razões para tal acontecer é a natureza dinâmica do algoritmo, ou seja, o facto de a função ir guardando os valores anteriormente calculados num *array*.

Contudo, o algoritmo Original Improved é igualmente eficiente, já que é possível obter um tempo de execução com ordem de grandeza 10^{-6} para $n = 800$.

A solução Combined, que tem por base o algoritmo Original Improved, não apresenta diferenças significativas, pelo que podemos concluir que alterações com a mesma complexidade computacional não produzem resultados diferentes.

6 Código

6.1 Original Improved

```
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;
}
```

6.2 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

// 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;
}
```

6.3 Combined

```
static solution_t solution_4;
static double solution_4_elapsed_time; // time it took to solve the
    problem
static unsigned long solution_4_count; // effort dispended solving
   the problem

static int solution_4_recursion(int move_number,int position,int
   speed,int final_position)
{
  int i,new_speed;

  // record move
  solution_4_count++;
  solution_4.positions[move_number] = position;
  // Solution found
  if(position == final_position && speed == 1)
  {
    solution_4.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_ && valstop(
   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_4_recursion(move_number + 1,position +
   new_speed,new_speed,final_position))
        return 1;
    }
  return 0;
}

static void solve_4(int final_position)
{
  if(final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr,"solve_4: bad final_position\n");
    exit(1);
  }
  solution_4_elapsed_time = cpu_time();
  solution_4_count = 0ul;
  solution_4.n_moves = final_position + 100;
  solution_4_recursion(0,0,0,final_position);
  solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}
```

6.4 Ajuste dos dados obtidos utilizando o MATLAB

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

% Curve fitting
figure(1)
f = fit(n,t,'exp1'); % Alternatively, one can launch the Curve
    Fitting Toolbox
plot(f,n,t,'b.')
xlabel('Road size')
ylabel('Time to find a solution (s)')
legend('Valores originais','Reta ajustada','Location','northwest')

% 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
xlabel('Road size')
ylabel('Time to find a solution (s)')

% Print the value
fprintf('Time needed to solve the problem, with size 800: %e
    seconds or %e years\n', y_fit(end), y_fit(end)/(3600*24*30*12))
```