Speed Run

Algoritmos e Estruturas de Dados

Prof. Tomás Oliveira e Silva
Prof. João Manuel Rodrigues

Gonçalo F. Couto Sousa - Nº Mec: 108133
Liliana P. Cruz Ribeira - Nº Mec: 108713

Departamento de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Índice

Introdução

Este relatório foi realizado no âmbito de expor os resultados do Primeiro Trabalho Prático da cadeira Algoritmos e Estruturas de Dados (AED). O objetivo deste trabalho consiste na elaboração de um programa que respondesse ao problema dado pelo Professor, que passa por arranjar um algoritmo aprimorado para determinar o número mínimo de movimentos necessários para que um "carro" percorresse uma estrada segmentada e chegasse à sua posição final, seguindo certos parâmetros bem definidos.

Estes parâmetros incluem várias limitações e variáveis que influenciam a maneira como o "carro" irá percorrer o seu caminho:

- Este anda numa estrada composta por vários segmentos do mesmo tamanho, sendo que cada segmento possui o seu limite de velocidade.
- A velocidade a que o carro se desloca será ditada pelo número de segmentos que este poderá avançar num único movimento, sem ultrapassar o limite de velocidade definido para cada segmento.
- O caminho percorrido pelo carro é também restringido pelo facto deste iniciar a sua marcha com uma velocidade de 0 e acabar com uma velocidade de 1 e pelo facto do carro apenas possuir 3 tipos de movimentos possíveis: - Aumentar a velocidade por 1; - Manter a velocidade; - Reduzir a velocidade por 1;

Para isto, foi disponibilizada uma pasta que inclui vários ficheiros: **makefile**, **make_custom_pdf.c**, **elapsed_time.h** e **speed_run.c**.

Metodologia

Procedemos à modificação da função fornecida **solution_1_recursion** e à implementação de novas funções de modo a otimizar a resolução para o nosso problema. Dentro da função main também foi desenvolvida uma maneira simplista para correr a testagem dos métodos desenvolvidos. Caso o utilizador introduza como último argumento (p.e depois do número mecanográfico) o termo **solution1**, irá ser corrida a **Solution_1_recursion**, caso introduza o termo **solution2v1,** irá ser corrida a **Solution_2v1_recursionTest**, caso introduza o termo **solution6**, irá ser corrida a **Solution_6_Dinamic**, caso não seja especificada uma solução, irá ser corrida a **solution_3_SmartWay**.

Solution_1_recursion

A solução já fornecida, **solution_1_recursion**, é pouco eficiente e incapaz de passar por testes com instâncias finais elevadas uma vez que é utilizado o método conhecido como **brute force**, conhecido em português como abordagem exaustiva ou de força bruta. É o método mais lento, comparativamente aos próximos, pois tem de percorrer todas as opções para encontrar a solução certa.

Esta possui uma variável **move_number**, que regista o total de movimentos e o aumenta em cada instância do movimento. O array **solution_1.positions** regista a posição a que o "carro" chega em cada movimento que irá fazer.

Recorrendo a uma condição **if/else**, irá-se verificar se o carro chega à posição final com a velocidade de 1, tal como pedido: - Caso isso aconteça, verificar-se-á que o caminho percorrido é o mais eficaz (verificando que o número de movimentos que acabou de calcular é menor do que o **best.n_moves** que tem atualmente). Caso seja, este passa a ser considerado a melhor solução (**solution_1_best**) e registar-se-á também o número de movimentos totais nesta iteração (**solution_1_best.n_moves = move_number**)

- Caso isso não aconteça, é utilizado um ciclo **for** que irá experimentar todas as velocidades possíveis para encontrar movimentos permitidos (no formato de uma **tree (Fig. 1)**)**.** Este tenta primeiro diminuir a velocidade, depois tenta manter a velocidade constante e só depois irá tentar aumentar a

velocidade, o que faz com que não seja nada eficiente para tentar arranjar o melhor caminho possível). Este ciclo, posteriormente, irá utilizar outra condição **if/else** que verificará que a **new_speed** não ultrapassa a velocidade mínima permitida (1) e a velocidade máxima permitida (9) e que a posição seguinte com o acrescento da nova velocidade não será superior à posição final (**position + new_speed <= final_position**).

Caso estas condições se verifiquem, este irá entrar num ciclo **for** que aparentemente não é util, mas, no entanto, permitirá tirar o valor de i (que vai depender da condição dentro do **for**). Este valor de **i** garante que a **max_road_speed** das posições intermédias não ultrapassam a nova velocidade.

Este valor de **i** irá posteriormente ser usado numa condição **if** para conferir se realmente todos os segmentos passam todas as condições. Se isto acontecer, este avança para o próximo movimento da função recursiva.

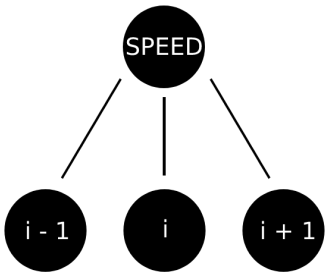

Fig. 1

```
      static solution_t solution_1, solution_1_best;
      static double solution_1_elapsed_time; // time it took to solve
the problem
      static unsigned long solution_1_count; // effort dispended
solving the problem

      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);
          }
      }
```

Solution_2v1_recursionTeste

A segunda função implementada utiliza também o método ***brute force***, mas, desta vez, "esperta" (*clever*). Este método evita que a recursão continue quando já é possível descobrir a solução do problema, não sendo necessário analisar todas as somas do problema.

Partimos então da função dada (**solution_1_recursion**) e considerámos que o ciclo **for** responsável por experimentar todas as velocidades possíveis para encontrar movimentos permitidos (no formato de uma **tree**) está a percorrer as 3 opções de velocidades (diminuir, manter e aumentar) pela ordem menos eficiente, e a simples modificação do ciclo para `for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)` irá fazer com que se encontre o melhor caminho mais rapidamente e por consequência seja mais eficiente, uma vez que

este irá agora primeiro tentar aumentar a velocidade e só caso isto não seja viável tentar manter e depois diminuir (como podemos ver na **fig. 2**) , fazendo com que o programa não tenha de correr todas as soluções possíveis, avançando logo para a melhor.

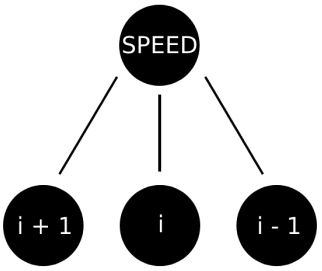

Fig. 2

```
static solution_t solution_2v1, solution_2v1_best;
static double solution_2v1_elapsed_time; // time it took to solve the
problem
static unsigned long solution_2v1_count; // effort dispended solving
the problem

static void solution_2v1_recursionTeste(int move_number, int position,
int speed, int final_position)
{

 if(solution_2v1_best.positions[move_number] > position) return;
  int i, new_speed;

 // record move
 solution_2v1_count++;
 solution_2v1.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   if (move_number < solution_2v1_best.n_moves)
   {
     solution_2v1_best = solution_2v1;
     solution_2v1_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_2v1_recursionTeste(move_number + 1, position +
new_speed, new_speed, final_position);
      }
    }
}
```

Solution_3_SmartWay

Para a implementação da primeira função feita de raiz optámos por uma metodologia em que a função iria sempre tentar aumentar a velocidade e só não o faria caso não fosse possível. No entanto, aumentar a velocidade nem sempre será a melhor opção, uma vez que poderá gerar um caminho não otimizado, e é aí que a nossa primeira solução irá solucionar esse problema de maneira inteligente.

A nossa função é gananciosa, uma vez que irá sempre tentar fazer com que seja aumentada a velocidade, a diferença é que ao invés de andar logo e só depois, caso der errado, voltar para trás, ela "prevê o futuro".

Ou seja, à medida que aumenta a velocidade esta irá também prever a "distância de travagem", isto é, se é possível ir diminuindo a velocidade para que se chegue à posição final com velocidade 1.

Isto faz com que nunca seja necessário voltar atrás caso não seja possível travar a tempo, o que por si só já otimiza a solução, mas também assegurará que é mantido o aumento de velocidade até à posição mais longe possível, sem que a distância de travagem fique comprometida, otimizando o tempo de execução (**Fig. 3**)

Para inicializar esta função começámos por definir a variável do struct **solution_3_best**, a variável **solution_3_elapsed_time** e a variável **solution_3_count.**

```
static solution_t solution_3_best;
static double solution_3_elapsed_time; // time it took to solve the
problem
static unsigned long solution_3_count; // effort dispended solving the
problem
```

Seguidamente irá ser iniciado um ciclo while que irá confirmar que a posição em questão é inferior à posição final. Depois iniciamos um ciclo **for** semelhante ao usado na função anterior, que irá fazer com que se tente sempre aumentar a velocidade e só caso não seja possível, manter a velocidade atual ou diminuí-la. A variável **brk** toma valor 0 e utilizamos uma condição **if** para testar se a nova velocidade ultrapassa a velocidade máxima permitida. Caso isso aconteça, então passamos para a próxima iteração do ciclo **for**.

```
while (position < final_position)
 {

   for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
   {
     brk = 0;

     if (new_speed > _max_road_speed_) // testa se ultrapassa a
velocidade maxima
     {
       continue;
     }

     position_test = position;
```

Em seguida utilizamos outro ciclo **for** em que a variável **speed_test** percorre todos os valores desde **speed+1** (movimento ideal) até à velocidade 1 (que é a velocidade de travagem). Dentro desse ciclo encontramos outra condição **if** que irá verificar se não é ultrapassada a posição final. Caso seja, a variável **brk** irá tomar valor 1.

```
for (speed_test = new_speed; speed_test >= 1; speed_test--)
      {
        if ((position_test + speed_test) > final_position)
        {
          brk = 1;
          break;
        }
```

Posteriormente, utilizamos outro ciclo **for** que irá testar a posição de travagem, que inclui dentro deste uma condição **if** que irá verificar se a velocidade máxima de cada segmento da estrada é ultrapassado. Caso isso aconteça a variável **brk** toma o valor 1 e saímos da condição e do ciclo. A última operação do ciclo **for** em que nos encontramos é somar à variável **position_test** a variável **speed_test**.

```
            for (md_position = 0; md_position <= speed_test;
md_position++) // testar posicao de travagem
        {
          if (speed_test > max_road_speed[position_test + md_position])
// passa a velocidade da casa?
          {
            brk = 1;
            break;
          }
        }

        position_test += speed_test; // future_position --> posicao
teste, i --> speed teste
      }
```

Só nos resta uma condição **if** para atualizar as variáveis:

- speed irá tomar o valor da variável **new_speed**;
- A variável position irá ser acrescentado o valor de speed;
- Irá haver um incremento na variável **solution_3_count**;
- A variável **solution_3_best** guarda a posição no array **position** com o index **move_number**. Em seguida **move_number** irá incrementar para a próxima iteração;
- O atribuito **n_moves** irá ser incrementado na **solution_3_best**;

```
        if (!brk) // não deu errado? temos o proximo passo :) -->
atualizar as variaveis
        {
          speed = new_speed;
          position += speed;
          solution_3_count++;
          solution_3_best.positions[move_number++] = position;
          solution_3_best.n_moves++;
          break;
        }
      }
    }
}
```

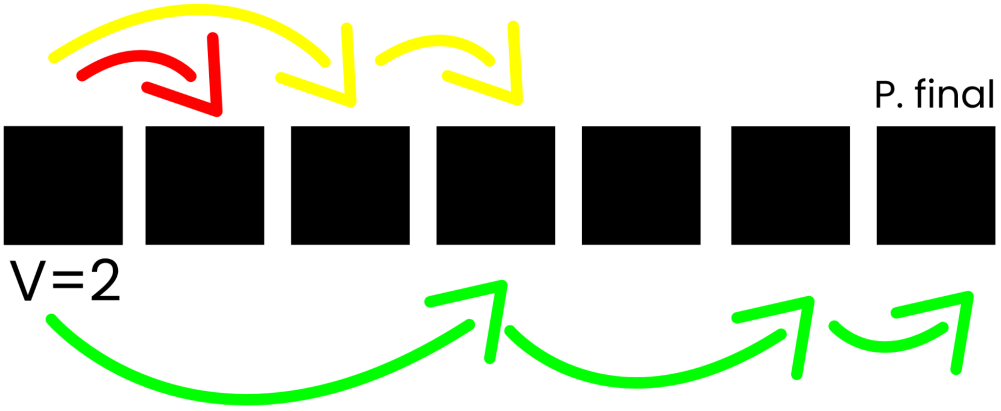

Fig. 3 - Legenda: Testa aumentar a velocidade (Movimento ideal); Testa manter a velocidade; Testa reduzir a velocidade.

Solution_6_Dinamic

Nota: foram realizadas mais duas tentativas de resolução do problema que acabaram por ser cortadas porque não conseguiram ser executadas eficientemente, pelo que esta solução se chama **solution_6_Dinamic** mas no entanto é a 4ª solução apresentada.

Esta solução corresponde à tentativa de resolução do problema de uma maneira dinâmica.

Para começar declaramos algumas variaveis estáticas:

```
// save the values for the dinamic function
static int posD = 0;
static int spdD = 0;
static int movD = 0;
static solution_t solution_6_best;
static double solution_6_elapsed_time; // time it took to solve the
problem
static unsigned long solution_6_count; // effort dispended solving the
problem
```

-posD → posição guardada para uma próxima chamada da função
-spdD → speed guardada para uma próxima chamada da função
-movD → posição guardada para uma próxima chamada da função

Foram também declaradas algumas variáveis dentro da função como **rd** (que corresponde à opção que irá ser escolhida. **rd**=2 corresponde a diminuir a velocidade, **rd**=1 corresponde a manter a velocidade e **rd**=0 corresponde a diminuir a velocidade) e **fp** (que indica que se estivermos próximos da posição final fp será diferente de 0).

```
static void solution_6_Dinamic(int move_number, int position, int
speed, int final_position)
{
 int speed_test, md_position, new_speed, position_test;
 solution_6_best.n_moves = move_number;
 int rd;
 int fp = 0;
```

Em seguida utilizamos um ciclo **while** que irá ser repetido até se chegar à posição final. Dentro deste encontramos uma condição **if** que será executada caso **rd** == 0 ou **fp** == 0. Dentro da condição a variável **new_speed** será igual a **speed** + 1 e entramos noutro **if** para testar que a posição final não é ultrapassada.

```
 while (position < final_position)
 {
   rd = 0;
   // increase
   if (rd == 0 || fp == 0)
   {
```

```
      new_speed = speed + 1;

      if (new_speed <= _max_road_speed_) // testa se não ultrapassa o
final TODO: n(n+1)2
      {
        position_test = position;
```

Dentro deste encontramos um ciclo **for** que, como na solução anterior, percorre todos os valor desde **speed+1** até 1. Dentro do ciclo encontramos outro **if** que será executado caso **position_test + speed_test** seja superior à posição final (**final_position**) e que define **fp** e **rd** com valor 1.

```
        for (speed_test = new_speed; speed_test >= 1; speed_test--)
        {
          if ((position_test + speed_test) > final_position)
          {
            fp = 1;
            rd = 1;
            break;
          }
```

Seguidamente temos outro ciclo para testar a posição de travagem, que inclui dentro de si uma condição que irá verificar se a velocidade máxima de cada segmento da estrada é ultrapassado, e, caso seja, faz **rd** = 1. Fora da condição **if** anterior à variável **position_test** é somada o valor de **speed_test**.

```
          for (md_position = 0; md_position <= speed_test;
md_position++)
          {
            if (speed_test > max_road_speed[position_test +
md_position])
            {
              rd = 1;
              break;
            }
          }

          position_test += speed_test;
        }
      }
```

Caso não sejam verificadas as condições da condição que testa se não é ultrapassada a posição final, então **rd** toma valor 1.

```
        else
        {
          rd = 1;
        }
      }
```

A seguir, caso **rd** == 1 ou **fp** == 1 temos uma condição **if** em que possui outro ciclo **for** igual ao que anteriormente percorria todos os valor desde **speed+1** até 1, sendo que dentro dele possui tudo igual ao anterior, mas onde as variáveis **fp** e **rd** tomavam valor 1, passarão agora a tomar o valor 2.

```
      if (rd == 1 || fp == 1)
      {
        new_speed = speed;
        position_test = position;

        for (speed_test = new_speed; speed_test >= 1; speed_test--)
        {
          if ((position_test + speed_test) > final_position)
          {
            fp = 2;
            rd = 2;
            break;
          }
          for (md_position = 0; md_position <= speed_test;
md_position++)
          {
            if (speed_test > max_road_speed[position_test +
md_position])
            {
              rd = 2;
              break;
            }
          }

          position_test += speed_test;
        }
```

```
    }
```

Depois encontramos outra condição **if** que ocorre caso **fp** == 2 ou **rd** == 2, sendo que esta apenas faz com que **new_speed** seja igual a **speed - 1** por ser a última opção.

```
    if (rd == 2 || fp == 2)
    {
      new_speed = speed - 1;
    }
```

Depois apenas nos resta guardar os valores de cada movimento. Se não chegámos perto da posição final (onde começa a diminuir para acabar com velocidade igual a 0), então guardamos nas variaveis estáticas os valores do move atual, para serem usados na próxima chamada para uma posição maior.

```
    speed = new_speed;
    position += speed;
    solution_6_count++;
    solution_6_best.positions[move_number++] = position;
    solution_6_best.n_moves++;
    if (fp == 0)
    {
      movD = move_number;
      posD = position;
      spdD = speed;
    }
  }
}
```

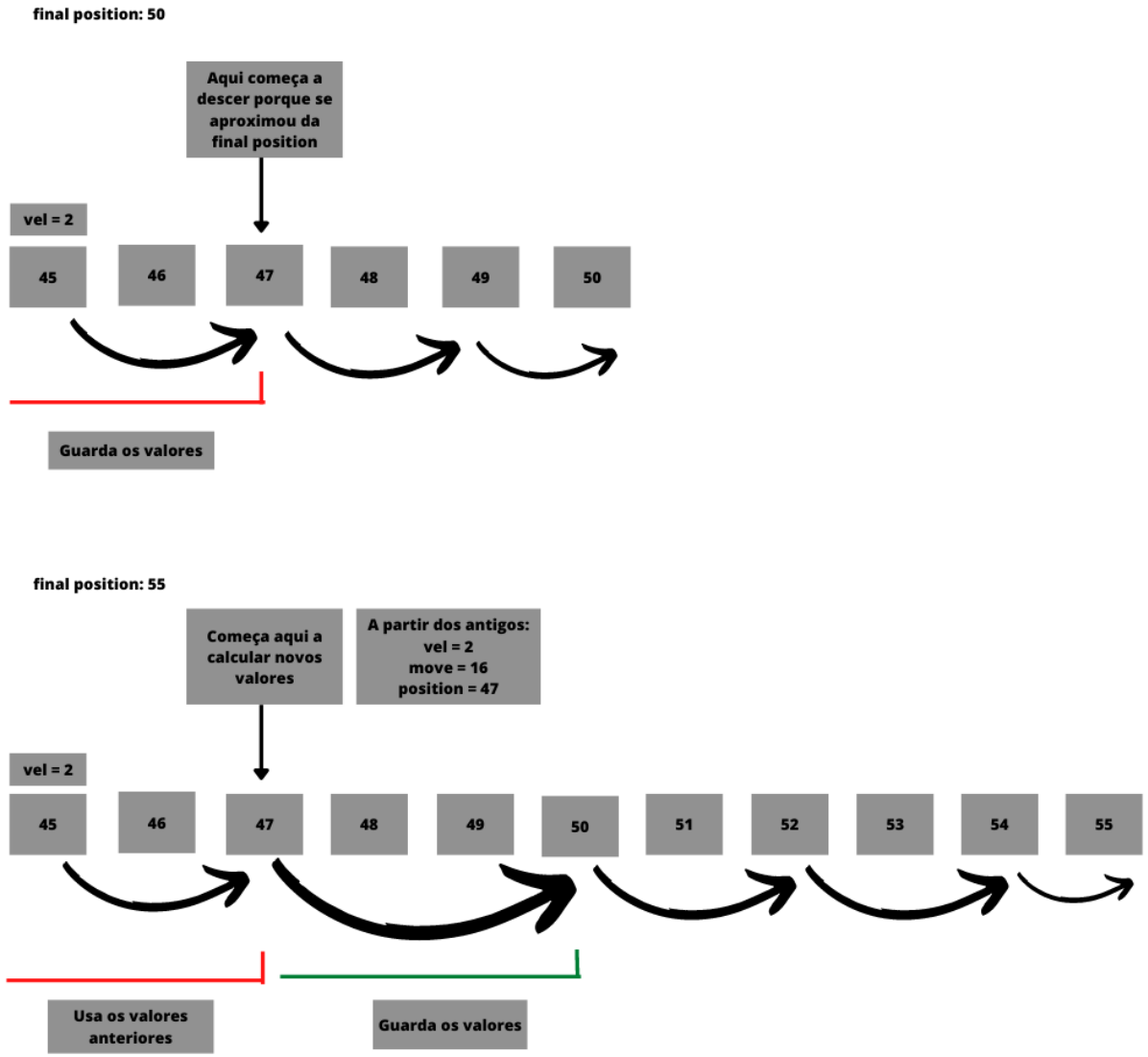

Fig. 4 -Método usado na solution_6_Dinamic

Main

Na função **main** encontramos vários prints que farão com que o programa e os seus dados sejam mostrados no terminal, sendo que **n** corresponde ao número de movimentos, **sol** ao número de movimento executados para chegar à solução, **count** corresponde ao esforço para encontrar a solução e **cpu time** o tempo de resolução.

Tratamento dos Resultados

Para testar os métodos implementados, testámos cada um para dois argumentos diferentes, correspondendo ao número mecanográfico de cada membro deste grupo (108133 e 108713).

Ao vermos os tempos de resolução para cada posição da estrada no terminal, bem como nos PDF's gerados para determinadas posições finais, conseguimos organizar estes numa tabela Excel que melhor organizava os dados obtidos. Posteriormente, e para cada solução (**solution_1_recursion**, **solution_2v1_recursionTeste**, **solution_3_SmartWay** e **solution_6_Dinamic**) obtivémos os gráficos do **tempo de resolução** em função da **posição**.

problema	tempos em segundos 108133	tempos em segundos 108713
0	0	**0**
5	1,27E-06	1,22E-06
10	6,84E-06	4,99E-06
15	1,49E-05	1,38E-05
20	1,74E-04	1,79E-04
25	2,19E-03	2,25E-03
30	2,85E-02	2,85E-02
35	3,65E-01	3,62E-01
40	4,81E+00	4,45E+00
45	6,14E+01	6,13E+01

Tabela 1 - Solution_1_recursion

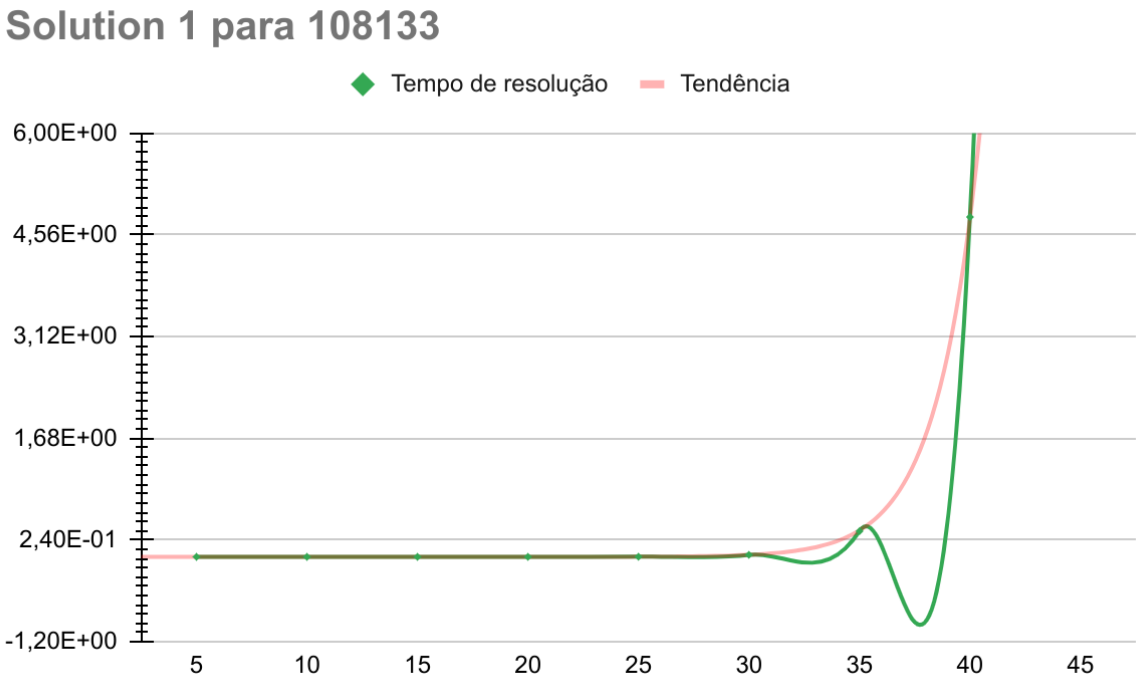

Gráfico 1 - Solution_1_recursion para o argumento 108133

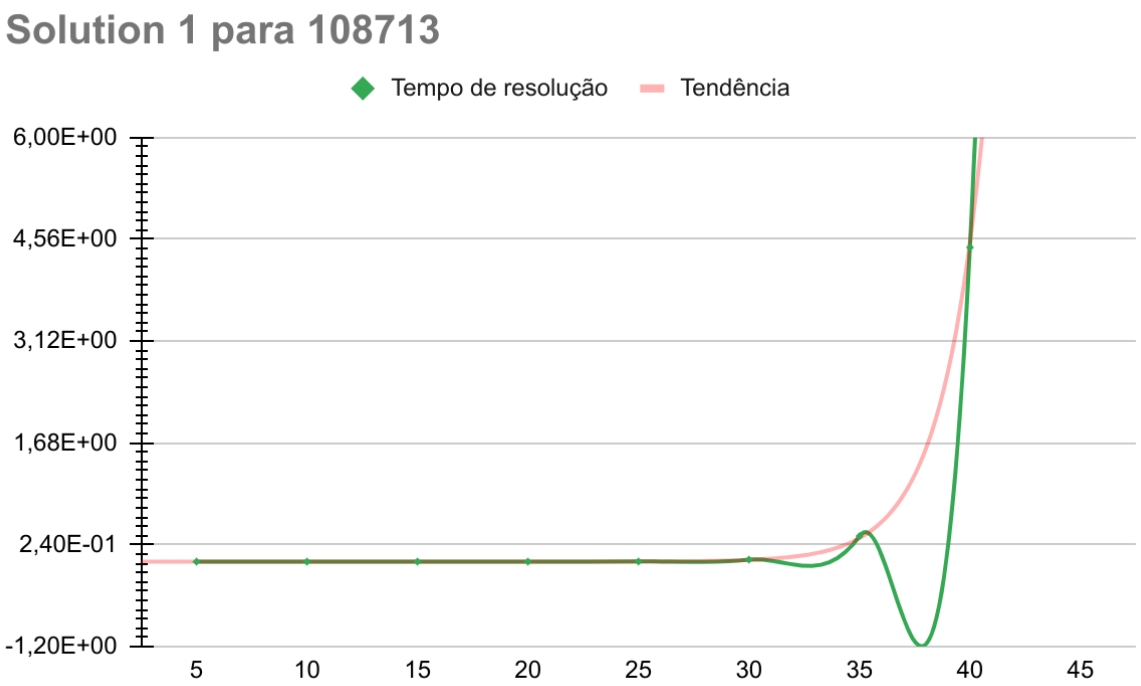

Gráfico 2 - Solution_1_recursion para o argumento 108713

Podemos então concluir que para a resolução já dada (**solution _1_recursion**) é bastante ineficaz na resolução do problema, uma vez que o tempo de resolução é consideravelmente superior ao das outras soluções e esta só consegue desenvolver posições superiores a 50 com tempos de resolução bastante longos.

problema	tempos em segundos 108133	tempos em segundos 108713
0	0	**0**
10	1,16E-06	1,38E-06
20	4,28E-07	4,29E-07
50	4,54E-07	4,88E-07
100	8,59E-07	9,07E-07
200	2,15E-06	1,70E-06
400	3,09E-06	2,97E-06
800	6,41E-06	5,82E-06

Tabela 2 - Solution_2v1_recursionTeste

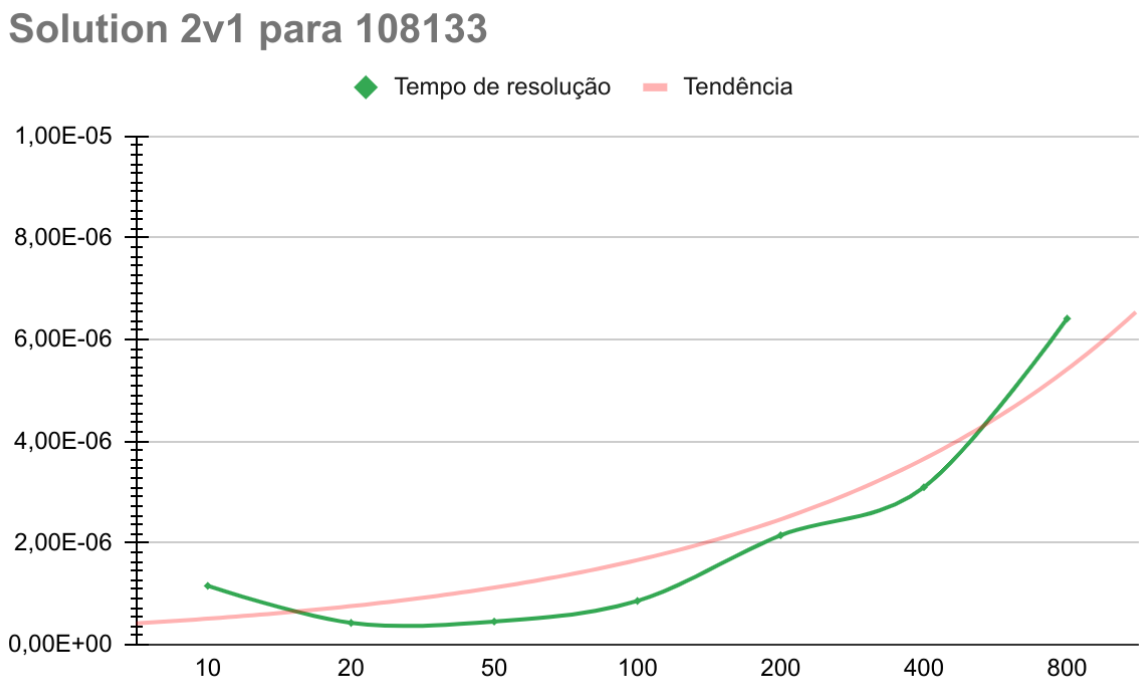

Gráfico 3 - Solution_2v1_recursionTeste para o argumento 108133

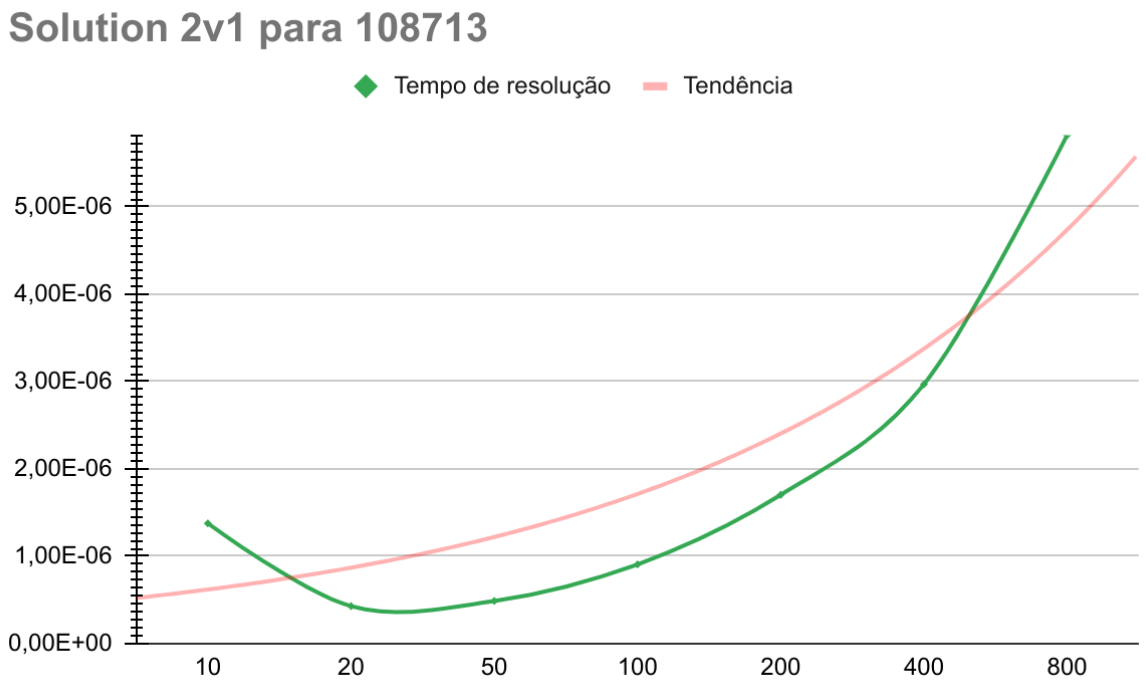

Gráfico 4 - Solution_2v1_recursionTeste para o argumento 1081387

Analisando os resultados da **solution_2v1_recursionTeste** podemos concluir que esta é significativamente mais eficaz que a solução 1. Esta já é capaz que chegar à posição 800 para ambos os testes e o seu tempo de execução também foi significativamente melhorado.

problema	tempos em segundos 108133	tempos em segundos 108713
0	0	**0**
10	4,15E-07	7,11E-07
20	3,76E-07	7,53E-07
50	9,61E-07	7,99E-07
100	1,71E-06	1,49E-06
200	2,10E-06	1,50E-06
400	3,83E-06	2,96E-06
800	6,41E-06	8,24E-06

Tabela 3 - Solution_3_SmartWay

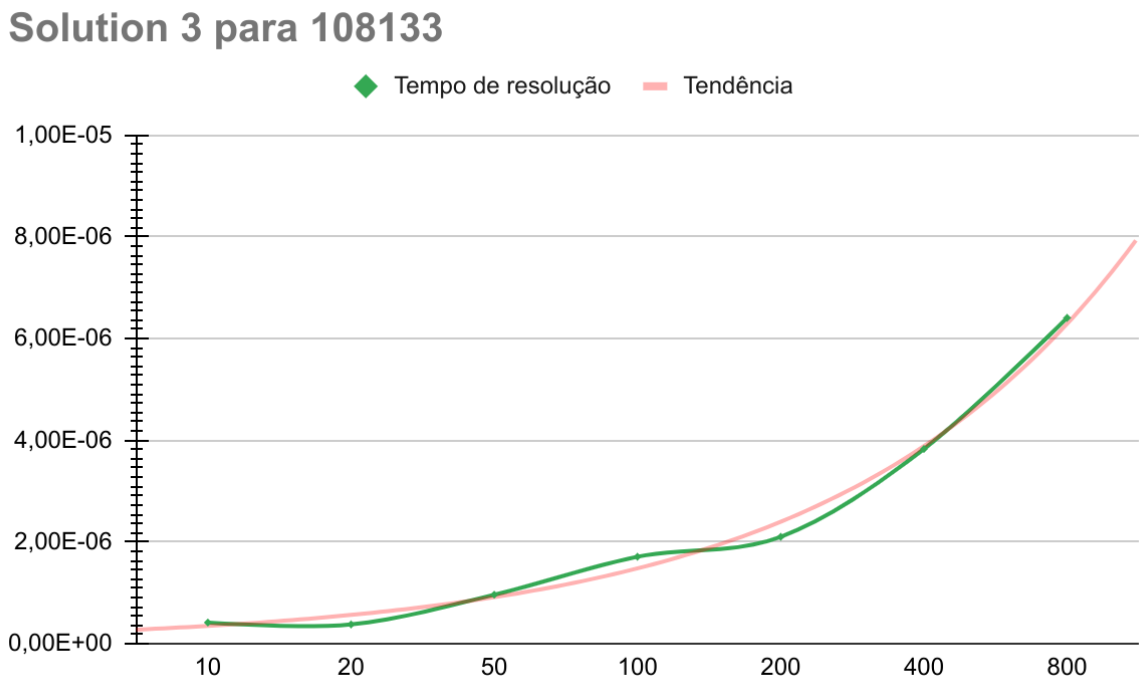

Gráfico 5 - Solution_3_SmartWay para o argumento 108133

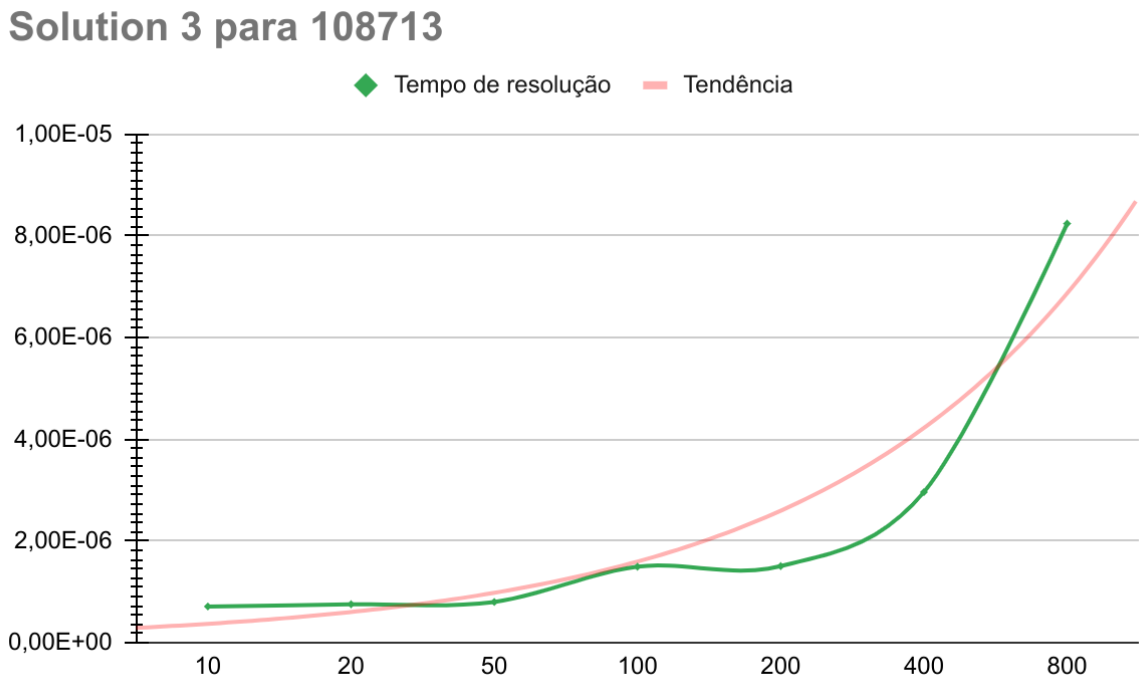

Gráfico 6 - Solution_3_SmartWay para o argumento 108713

Ao analisar a testagem da terceira solução concluímos que esta é capaz de resolução com tempos de resolução significativamente inferiores às restantes

solução, especialmente em posições maiores. Conseguimos assim verificar que é a solução mais otimizada.

problema	tempos em segundos 108133	tempos em segundos 108713
0	0	**0**
10	7,33E-07	6,82E-07
20	2,01E-06	1,37E-06
50	8,44E-07	8,12E-07
100	1,82E-06	1,53E-06
200	1,07E-06	1,06E-06
400	1,82E-06	2,17E-06
800	1,71E-06	1,58E-06

Tabela 4 - Solution_6_Dinamic

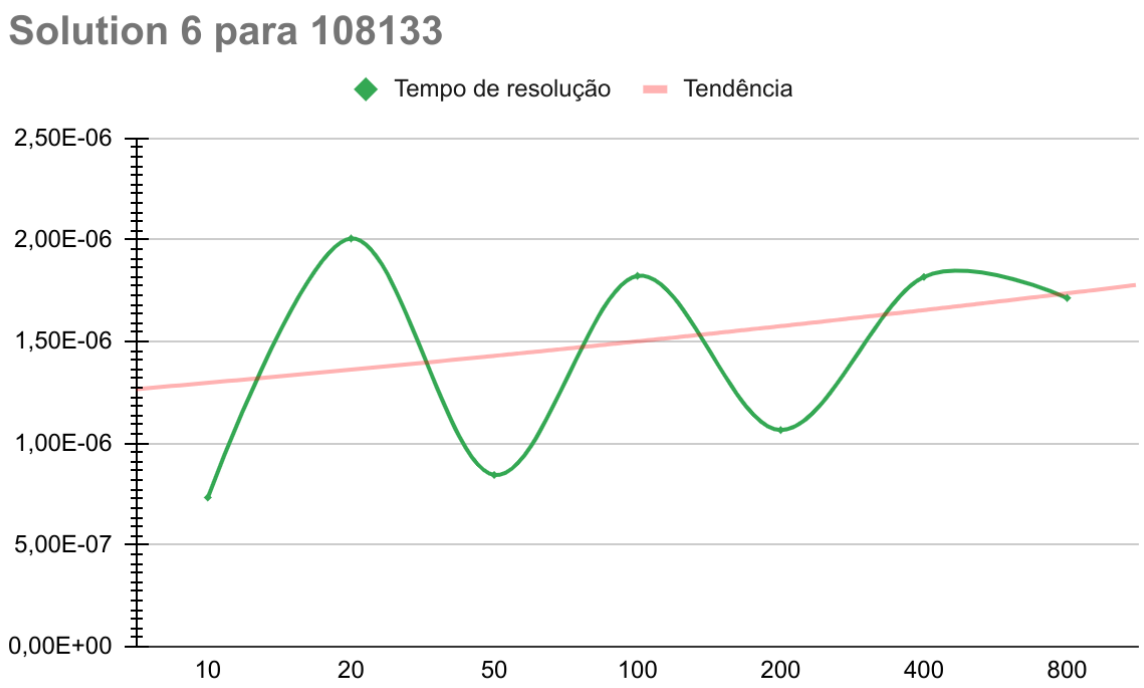

Gráfico 7 - Solution_6_Dinamic para o argumento 108133

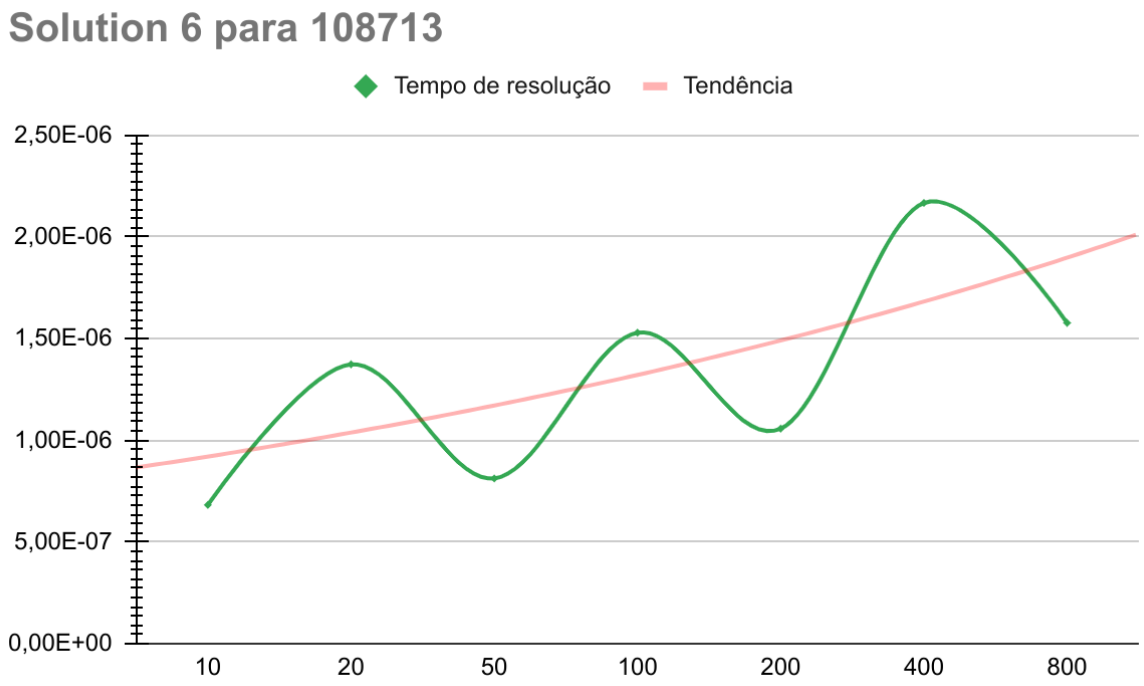

Gráfico 8 - Solution_6_Dinamic para o argumento 108713

Ao analisar os gráficos para esta solução podemos concluir que este apresenta tempos de resolução inconstante consoante a posição final e que apresenta tempos de execução superiores aos da solução anterior.

Em seguida apresentamos gráficos que melhor ilustram a comparação entre as 4 soluções para os valores das primeiras posições finais nos PDF's gerados, uma vez que a primeira solução só é capaz de alcançar a posição 50 com um tempo de resolução muito superior ao das restantes soluções.

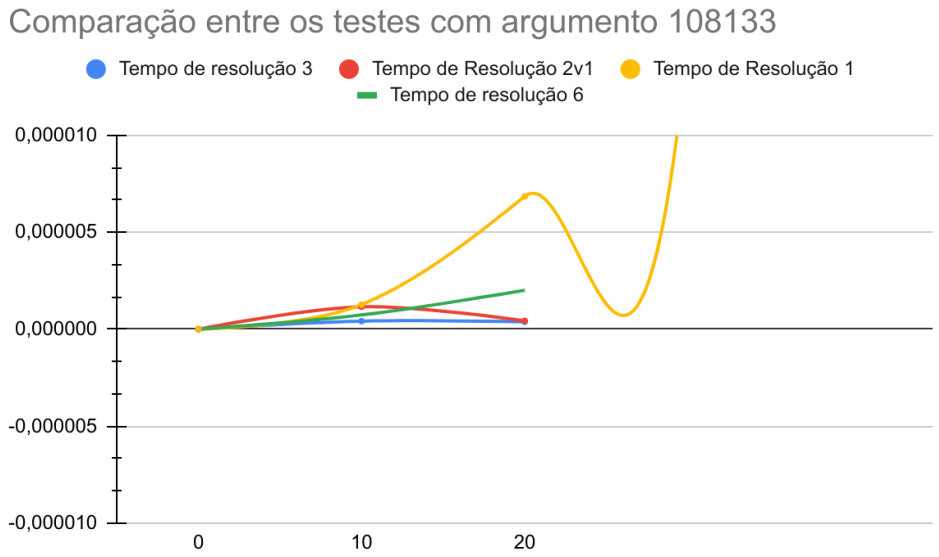

Gráfico 9 - Comparação dos métodos implementados para o argumento 108133

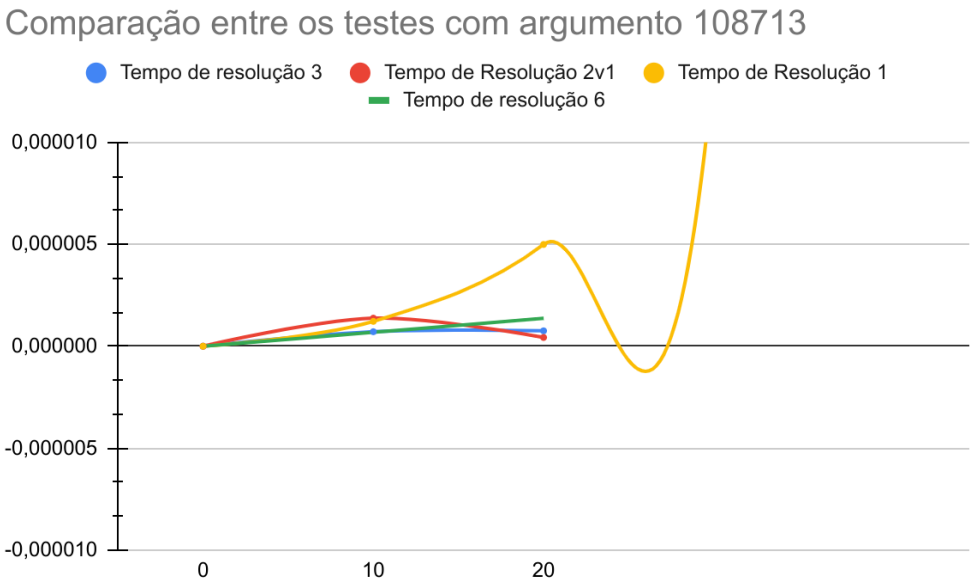

Gráfico 10 - Comparação dos métodos implementados para o argumento 108713

Código Final

```
#define _max_road_size_ 800 // the maximum problem size
#define _min_road_speed_ 2  // must not be smaller than 1, shouldnot be
smaller than 2
#define _max_road_speed_ 9  // must not be larger than 9 (only because
of the PDF figure)

//
// include files --- as this is a small project, we include the PDF
generation code directly from make_custom_pdf.c
//

#include <math.h>
#include <stdio.h>
#include <string.h> //usado para podermos escolher que funcção vamos
correr
#include "../P02/elapsed_time.h"
#include "make_custom_pdf.c"

//
// road stuff
//

static int max_road_speed[1 + _max_road_size_]; // positions
0.._max_road_size_

static void init_road_speeds(void)
{
 double speed;
 int i;

 for (i = 0; i <= _max_road_size_; i++)
 {
   speed = (double)_max_road_speed_ * (0.55 + 0.30 * sin(0.11 *
(double)i) + 0.10 * sin(0.17 * (double)i + 1.0) + 0.15 * sin(0.19 *
(double)i));
   max_road_speed[i] = (int)floor(0.5 + speed) + (int)((unsigned
int)random() % 3u) - 1;
```

```
    if (max_road_speed[i] < _min_road_speed_)
      max_road_speed[i] = _min_road_speed_;
    if (max_road_speed[i] > _max_road_speed_)
      max_road_speed[i] = _max_road_speed_;
  }
}

//
// description of a solution
//

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

//
// the (very inefficient) recursive solution given to the students
//

static solution_t solution_1, solution_1_best;
static double solution_1_elapsed_time; // time it took to solve the
problem
static unsigned long solution_1_count; // effort dispended solving the
problem

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);
    }
}

static solution_t solution_2v1, solution_2v1_best;
static double solution_2v1_elapsed_time; // time it took to solve the
problem
static unsigned long solution_2v1_count; // effort dispended solving
the problem

static void solution_2v1_recursionTeste(int move_number, int position,
int speed, int final_position)
{

  if (solution_2v1_best.positions[move_number] > position)
    return;

  int i, new_speed;

  // record move
  solution_2v1_count++;
  solution_2v1.positions[move_number] = position;
  // is it a solution?
  if (position == final_position && speed == 1)
  {
    if (move_number < solution_2v1_best.n_moves)
    {
      solution_2v1_best = solution_2v1;
```

```
      solution_2v1_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_2v1_recursionTeste(move_number + 1, position +
new_speed, new_speed, final_position);
      }
    }
}

static solution_t solution_3_best;
static double solution_3_elapsed_time; // time it took to solve the
problem
static unsigned long solution_3_count; // effort dispended solving the
problem

static void solution_3_SmartWay(int move_number, int position, int
speed, int final_position)
{
  int speed_test, md_position, new_speed, position_test, brk;
  solution_3_best.n_moves = 0;

  while (position < final_position)
  {

    for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    {
      brk = 0;

      if (new_speed > _max_road_speed_) // testa se ultrapassa a
velocidade maxima
```

```
      {
        continue;
      }

      position_test = position;

      for (speed_test = new_speed; speed_test >= 1; speed_test--) // i
percorre todos os valores desde speed+1 (movimento ideal) até 1... -->
velocidades assumidas na travagem
      {

        if ((position_test + speed_test) > final_position)
        {
          brk = 1;
          break;
        }
        for (md_position = 0; md_position <= speed_test; md_position++)
// testar posicao de travagem
        {
          if (speed_test > max_road_speed[position_test + md_position])
// passa a velocidade da casa?
          {
            brk = 1;
            break;
          }
        }

        position_test += speed_test; // future_position --> posicao
teste, i --> speed teste
      }

      if (!brk) // não deu errado? temos o proximo passo :) -->
atualizar as variaveis
      {
        speed = new_speed;
        position += speed;
        solution_3_count++;
        solution_3_best.positions[move_number++] = position;
        solution_3_best.n_moves++;
        break;
      }
    }
  }
```

```
}

// save the values for the dinamic function
static int posD = 0;
static int spdD = 0;
static int movD = 0;
static solution_t solution_6_best;
static double solution_6_elapsed_time; // time it took to solve the
problem
static unsigned long solution_6_count; // effort dispended solving the
problem

static void solution_6_Dinamic(int move_number, int position, int
speed, int final_position)
{
 // define some variables
 int speed_test, md_position, new_speed, position_test;
 // save de values from the save value of move_number
 solution_6_best.n_moves = move_number;
 // rd --> choose what option will the car choose (decrease rd=2, keep
rd=1 or increase rd=0)
 int rd;
 // fp --> if the car is close to the final point fp different from 0
 int fp = 0;

 // repite this process until it gets to the final position
 while (position < final_position)
 {
   rd = 0;
   // increase
   if (rd == 0 || fp == 0)
   {
     new_speed = speed + 1;

     if (new_speed <= _max_road_speed_) // testa se não ultrapassa o
final TODO: n(n+1)2
     {
       position_test = position;

       for (speed_test = new_speed; speed_test >= 1; speed_test--) // i
percorre todos os valores desde speed+1 (movimento ideal) até 1... -->
velocidades assumidas na travagem
       {
```

```
          if ((position_test + speed_test) > final_position)
          {
            fp = 1;
            rd = 1;
            break;
          }
          for (md_position = 0; md_position <= speed_test;
md_position++) // testar posicao de travagem
          {
            if (speed_test > max_road_speed[position_test +
md_position]) // passa a velocidade da casa?
            {
              rd = 1;
              break;
            }
          }

          position_test += speed_test; // future_position --> posicao
teste, i --> speed teste
        }
      }
      else
      {
        rd = 1;
      }
    }
    // keep
    if (rd == 1 || fp == 1)
    {
      new_speed = speed;
      position_test = position;

      for (speed_test = new_speed; speed_test >= 1; speed_test--) // i
percorre todos os valores desde speed+1 (movimento ideal) até 1... -->
velocidades assumidas na travagem
      {
        if ((position_test + speed_test) > final_position)
        {
          fp = 2;
          rd = 2;
          break;
        }
```

```
        for (md_position = 0; md_position <= speed_test; md_position++)
// testar posicao de travagem
        {
          if (speed_test > max_road_speed[position_test + md_position])
// passa a velocidade da casa?
          {
            rd = 2;
            break;
          }
        }

        position_test += speed_test; // future_position --> posicao
teste, i --> speed teste
      }
    }
    // decrease (dont run any testes because its the last option)
    if (rd == 2 || fp == 2)
    {
      new_speed = speed - 1;
    }
    // save the values of this movement
    speed = new_speed;
    position += speed;
    solution_6_count++;
    solution_6_best.positions[move_number++] = position;
    solution_6_best.n_moves++;
    // if the car didnt get close to the final position, we save the
current move to use in the next call for the bigger position
    if (fp == 0)
    {
      movD = move_number;
      posD = position;
      spdD = speed;
    }
  }
}

static void solve_1(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_1: bad final_position\n");
    exit(1);
```

```
 }
 solution_1_elapsed_time = cpu_time();
 solution_1_count = 0ul;
 solution_1_best.n_moves = final_position + 100;
 solution_1_recursion(movD, posD, spdD, final_position);
 solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;
}

// função 2v1
static void solve_2v1(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
   fprintf(stderr, "solve_1: bad final_position\n");
   exit(1);
 }
 solution_2v1_elapsed_time = cpu_time();
 solution_2v1_count = 0ul;
 solution_2v1_best.n_moves = final_position + 100;
 solution_2v1_recursionTeste(0, 0, 0, final_position);
 solution_2v1_elapsed_time = cpu_time() - solution_2v1_elapsed_time;
}

// função 3
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_best.n_moves = final_position + 100;
 solution_3_SmartWay(0, 0, 0, final_position);
 solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}

// função 3
static void solve_6(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
```

```
    fprintf(stderr, "solve_1: bad final_position\n");
    exit(1);
  }
  solution_6_elapsed_time = cpu_time();
  solution_6_count = 0ul;
  solution_6_best.n_moves = final_position + 100;
  // fazer os moves novos
  // em vez de começar com os valores 0,0,0,final_position começa com os
valores guardados
  solution_6_Dinamic(movD, posD, spdD, final_position);
  // solution_6_Dinamic(0, 0, 0, final_position);
  solution_6_elapsed_time = cpu_time() - solution_6_elapsed_time;
}

//
// example of the slides
//

static void example(void)
{
  int i, final_position;

  srandom(0xAED2022);
  init_road_speeds();
  final_position = 30;
  solve_1(final_position);
  make_custom_pdf_file("example.pdf", final_position,
&max_road_speed[0], solution_1_best.n_moves,
&solution_1_best.positions[0], solution_1_elapsed_time,
solution_1_count, "Plain recursion");
  printf("mad road speeds:");
  for (i = 0; i <= final_position; i++)
    printf(" %d", max_road_speed[i]);
  printf("\n");
  printf("positions:");
  for (i = 0; i <= solution_1_best.n_moves; i++)
    printf(" %d", solution_1_best.positions[i]);
  printf("\n");
}

//
// main program
//
```

```
int main(int argc, char *argv[argc + 1])
{
#define _time_limit_ 3600.0
 int n_mec, final_position, print_this_one;
 char file_name[64];

 // generate the example data
 if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'e' && argv[1][2]
== 'x')
 {
   example();
   return 0;
 }
 // initialization
 n_mec = (argc < 2) ? 0xAED2022 : atoi(argv[1]);
 srandom((unsigned int)n_mec);
 init_road_speeds();
 // run all solution methods for all interesting sizes of the problem
 final_position = 1;

 printf("    + --- ---------------- --------- +\n");
 printf("    |                plain recursion |\n");
 printf("--- + --- ---------------- --------- +\n");
 printf("  n | sol            count  cpu time |\n");
 printf("--- + --- ---------------- --------- +\n");
 while (final_position <= _max_road_size_ /* && final_position <= 20*/)
 {
   print_this_one = (final_position == 10 || final_position == 20 ||
final_position == 50 || final_position == 100 || final_position == 200
|| final_position == 400 || final_position == 800) ? 1 : 0;
   printf("%3d |", final_position);

   if (strcmp(argv[argc - 1], "solution1") == 0)
   {
     solution_1_elapsed_time = 0.0;
     // first solution method (very bad)
     if (solution_1_elapsed_time < _time_limit_)
     {
       solve_1(final_position);
       if (print_this_one != 0)
       {
         sprintf(file_name, "%03d_1.pdf", final_position);
```

```
        make_custom_pdf_file(file_name, final_position,
&max_road_speed[0], solution_1_best.n_moves,
&solution_1_best.positions[0], solution_1_elapsed_time,
solution_1_count, "Plain recursion");
      }
      printf(" %3d %16lu %9.3e |", solution_1_best.n_moves,
solution_1_count, solution_1_elapsed_time);
    }
    else
    {
      solution_1_best.n_moves = -1;
      printf("                                  |");
    }
  }
  else if (strcmp(argv[argc - 1], "solution2v1") == 0)
  {
    solution_2v1_elapsed_time = 0.0;
    // first solution method (very bad)
    if (solution_2v1_elapsed_time < _time_limit_)
    {
      solve_2v1(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_1.pdf", final_position);
        make_custom_pdf_file(file_name, final_position,
&max_road_speed[0], solution_2v1_best.n_moves,
&solution_2v1_best.positions[0], solution_2v1_elapsed_time,
solution_2v1_count, "recursion but smart");
      }
      printf(" %3d %16lu %9.3e |", solution_2v1_best.n_moves,
solution_2v1_count, solution_2v1_elapsed_time);
    }
    else
    {
      solution_2v1_best.n_moves = -1;
      printf("                                  |");
    }
  }
  else if (strcmp(argv[argc - 1], "solution6") == 0)
  {
    // six solution  method (dinamic one)
    solution_6_elapsed_time = 0.0;
    if (solution_6_elapsed_time < _time_limit_)
```

```
      {
        solve_6(final_position);
        if (print_this_one != 0)
        {
          sprintf(file_name, "%03d_1.pdf", final_position);
          // TODO: solucao que dado uma position mostra as anteriores
          make_custom_pdf_file(file_name, final_position,
&max_road_speed[0], solution_6_best.n_moves,
&solution_6_best.positions[0], solution_6_elapsed_time,
solution_6_count, "Dinamic");
        }
        printf(" %3d %16lu %9.3e |", solution_6_best.n_moves,
solution_6_count, solution_6_elapsed_time);
      }
      else
      {
        solution_6_best.n_moves = -1;
        printf("                                  |");
      }
    }
    else
    {
      // third solution  method (better one)
      solution_3_elapsed_time = 0.0;
      if (solution_3_elapsed_time < _time_limit_)
      {
        solve_3(final_position);
        if (print_this_one != 0)
        {
          sprintf(file_name, "%03d_1.pdf", final_position);
          make_custom_pdf_file(file_name, final_position,
&max_road_speed[0], solution_3_best.n_moves,
&solution_3_best.positions[0], solution_3_elapsed_time,
solution_3_count, "Best Way");
        }
        printf(" %3d %16lu %9.3e |", solution_3_best.n_moves,
solution_3_count, solution_3_elapsed_time);
      }
      else
      {
        solution_3_best.n_moves = -1;
        printf("                                  |");
      }
```

```
    }
    // second solution method (less bad)
    //....

    // done
    printf("\n");
    fflush(stdout);
    // new final_position
    if (final_position < 50)
      final_position += 1;
    else if (final_position < 100)
      final_position += 5;
    else if (final_position < 200)
      final_position += 10;
    else
      final_position += 20;
  }
  printf("--- + --- ---------------- --------- +\n");
  return 0;
#undef _time_limit_
}
```

Conclusão

Para terminar, podemos afirmar que conseguimos encontrar uma solução eficiente para o problema dado, aprimorando o código fornecido e criando métodos de resolução mais eficientes, como comprovado nos diversos testes realizados. Conseguimos ao longo deste trabalho prático melhorar a nossa capacidade de programação na linguagem C e pôr em prática os conteúdos e metodologias dadas aulas práticas e teóricas desta cadeira.

Bibliografia

Para além de consultarmos os slides da disciplina, que possuem apontamentos teóricos e guiões práticos, também consultámos alguns sites web face a alguma dificuldade de implementação do código, mais notavelmente os seguintes sites:

- **StackOverflow:** https://stackoverflow.com/
- **W3Schools:** https://www.w3schools.com/c/index.php