universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Speed Run

Algoritmos e Estruturas de Dados
2022

Professor Tomás Silva e Professor Pedro Lavrador

Trabalho realizado por:

João Nuno da Silva Luís (107403) | 50%

Diana Raquel Rodrigues Miranda (107457) | 50%

Índice

Índice de figuras

Introdução

Este trabalho foi realizado no âmbito da disciplina de Algoritmos e Estruturas de Dados do 2º ano da Licenciatura em Engenharia Informática.

Foi-nos proposto desenvolver um algoritmo em C que determinasse o número mínimo de movimentos necessários para alcançar a posição final. No entanto, têm de ser respeitadas certas regras, que são as seguintes:

- O carro só pode aumentar 1 velocidade, reduzir 1 velocidade ou mantê-la;

- O carro começa no primeiro segmento da estrada com uma velocidade 0 e tem de atingir o último segmento de estrada com uma velocidade de um;

- O carro não pode em nenhum momento passar num segmento de estrada com uma velocidade superior à nela permitida.

Com isto em mente, temos que a finalidade principal do nosso trabalho é conseguir otimizar o algoritmo fornecido pelo professor de modo a tornar possível atingir a posição 800. Para além disto, se for possível, desenvolver um novo algoritmo que resolva o problema com o menor tempo de execução possível.

Algoritmo fornecido

O algoritmo fornecido segue o conceito de depth first search, que é um algoritmo utilizado para realizar uma procura em árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder.

Posto isto, temos no nosso caso de estudo (Speed run) o nó raiz como a primeira posição de onde o carro irá arrancar e desse nó irão sair três novos nós, um com a opção de aumentar a velocidade, um com a opção de a manter e outro com a opção de a diminuir, e só depois de todas as possibilidades terem sido percorridas, é que o algoritmo vai retroceder e escolher o melhor caminho.

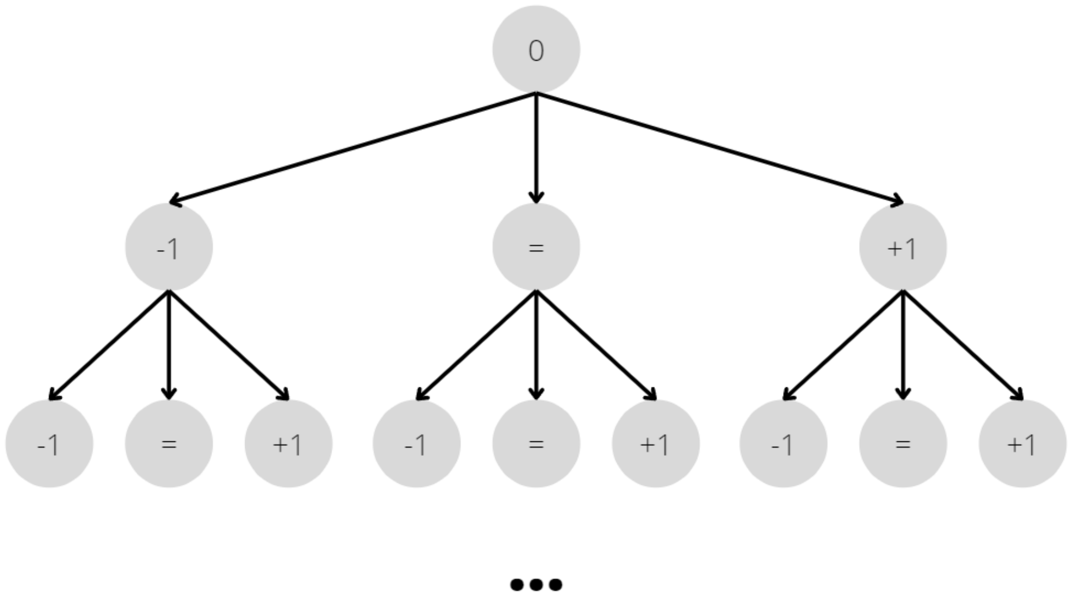

Figura 1 - Exemplo gráfico da árvore percorrida pela função fornecida.

Fazendo o gráfico do tempo de execução do algoritmo fornecido, antes de fazer qualquer otimização, obtemos um gráfico exponencial a partir da posição 40.

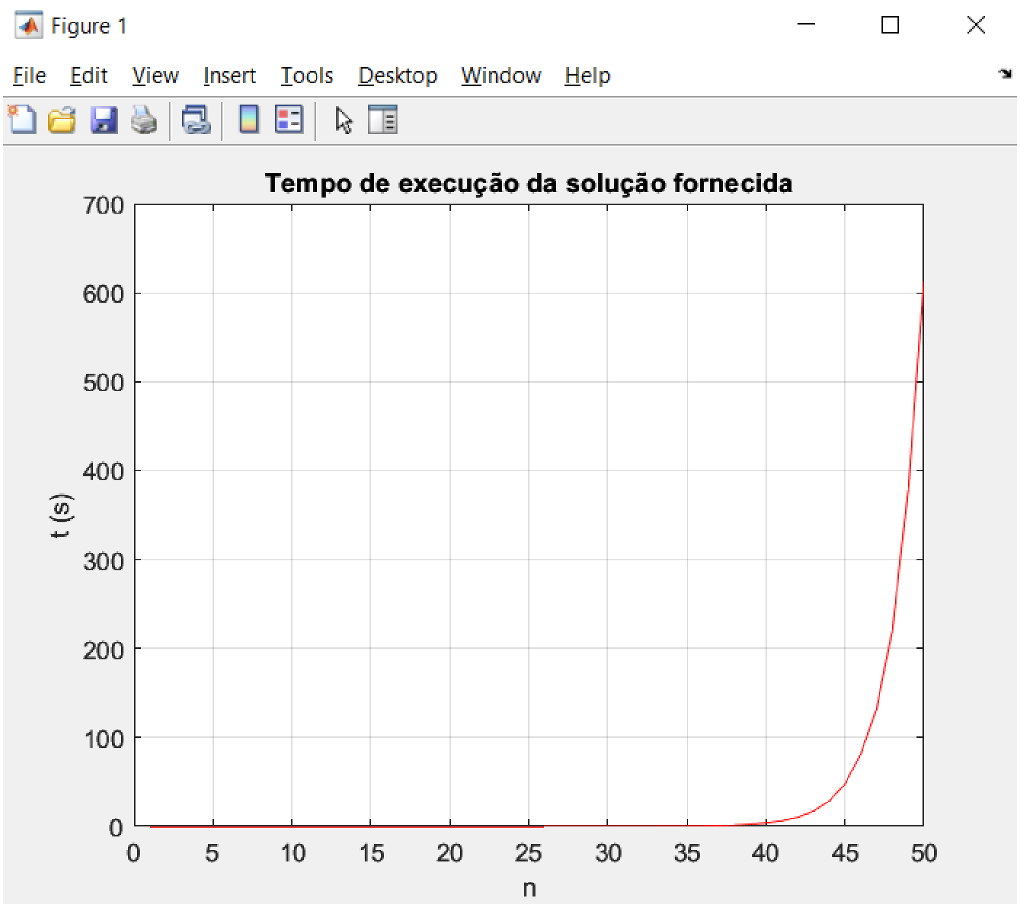

Figura 2- Tempo de execução da solução fornecida

Este tipo de gráfico não é o melhor para visualizarmos os nossos dados, nem calcular a reta de ajuste, pelo que podemos aplicar um logaritmo na base dez ao eixo dos yy (eixo dos tempos), e ao mesmo tempo calcular a reta de ajuste aos dados obtidos, que nos permite fazer uma previsão do tempo de execução deste algoritmo até à posição final.

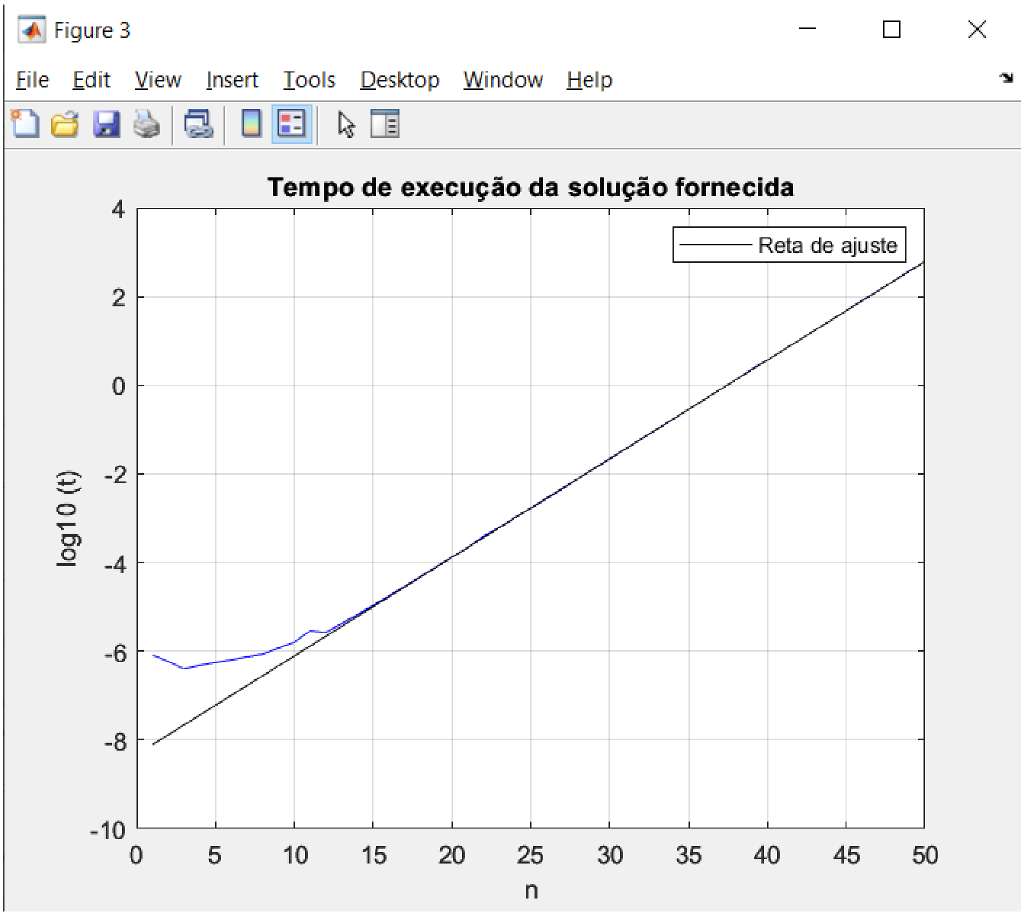

Figura 3- Reta de ajuste aos dados da solução fornecida

Através desta resta de ajuste, calculamos que a solução fornecida iria demorar 1.114e+162 segundos a chegar à posição 800.

Estes gráficos provam que apesar do algoritmo ser capaz de chegar à solução correta, o mesmo irá demorar muitíssimo tempo a resolver o problema para as 800 posições.

O nosso método para tornar este algoritmo mais eficiente foi pensar numa maneira de otimizar a pesquisa em árvore e diminuir o número de ramos visitados.

1ª melhoria – Tentar acelerar primeiro.

Como o nosso objetivo é chegar à posição final o mais rápido possível, vamos visitar primeiro o nó em que se verifica o aumento da velocidade e depois visitamos os outros.

Para isso, alterámos o seguinte pedaço de código:

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

Para:

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

Assim, vai começar a sua pesquisa sempre pelo nó que acelera.

Com o código fornecido, sem nenhuma alteração, e em execução durante uma hora (com o número mecanográfico 107457) é possível chegar à posição 50 em 6.121e+02 segundos. Com esta alteração foi possível, durante o mesmo tempo de execução, chegar à posição 50 em 5.935e+02 segundos. É uma melhoria mínima, pois mesmo a começar a pesquisa pelo nó que acelera o algoritmo vai verificar todos os ramos da árvore possíveis, o que não melhora muito o tempo de execução.

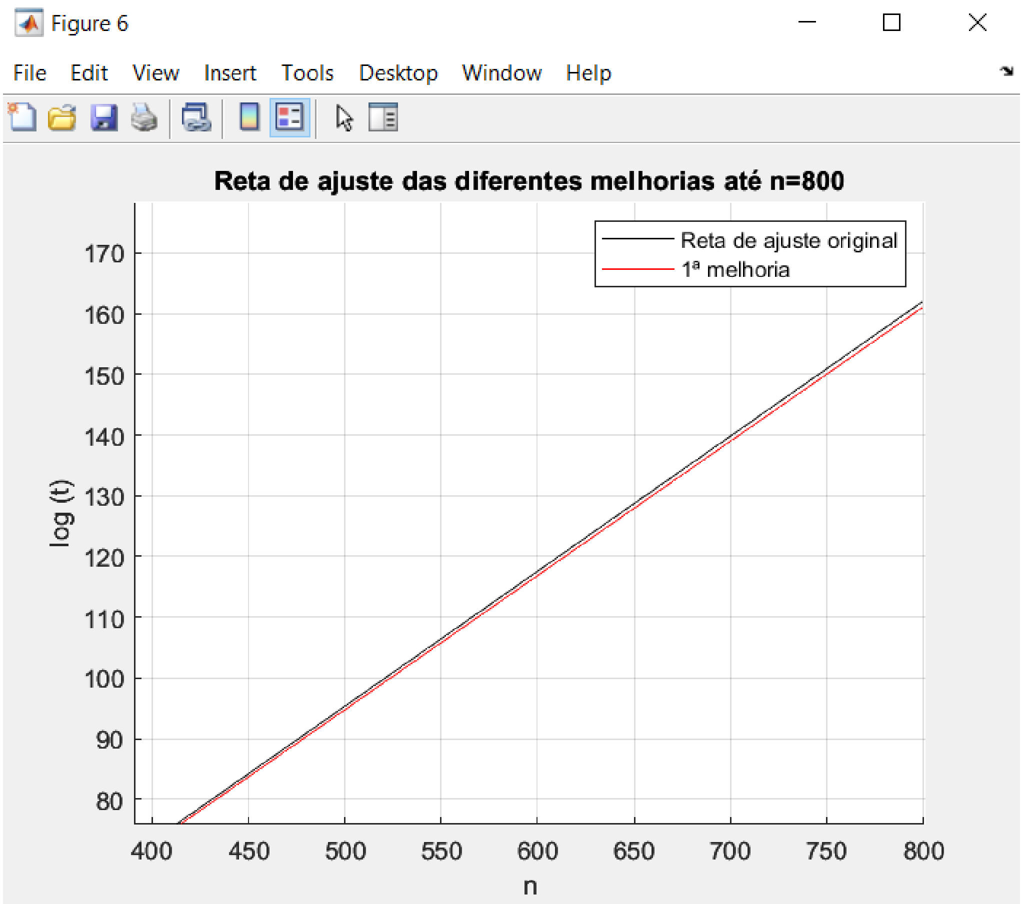

Figura 4- Comparação da reta de ajuste até à posição n=800

2ª melhoria – Acrescentar um if no código da função fornecida.

```
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(move_number >= solution_1_best.n_moves){
        return;
      }

      solution_1_otimized_recursion(move_number + 1, position + new_speed, new_speed, final_position);
    }
  }
```

Este if verifica se o número de movimentos da solução que está a ser vista nesse momento já é maior do que o número de movimentos total da "melhor" solução anteriormente guardada. Se for maior, então o algoritmo pode parar essa procura, pois já não nos interessa uma vez que já temos uma solução melhor, o que torna possível cortar alguns ramos da árvore.

Apesar de esta melhoria ser bastante simples, pois só acrescentámos duas linhas de código, é uma melhoria que nos permite reduzir o tempo de execução para metade. Com isto, durante uma hora de execução (com o número mecanográfico 107457), já conseguimos chegar à posição 95 em 1.009e+03 segundos.

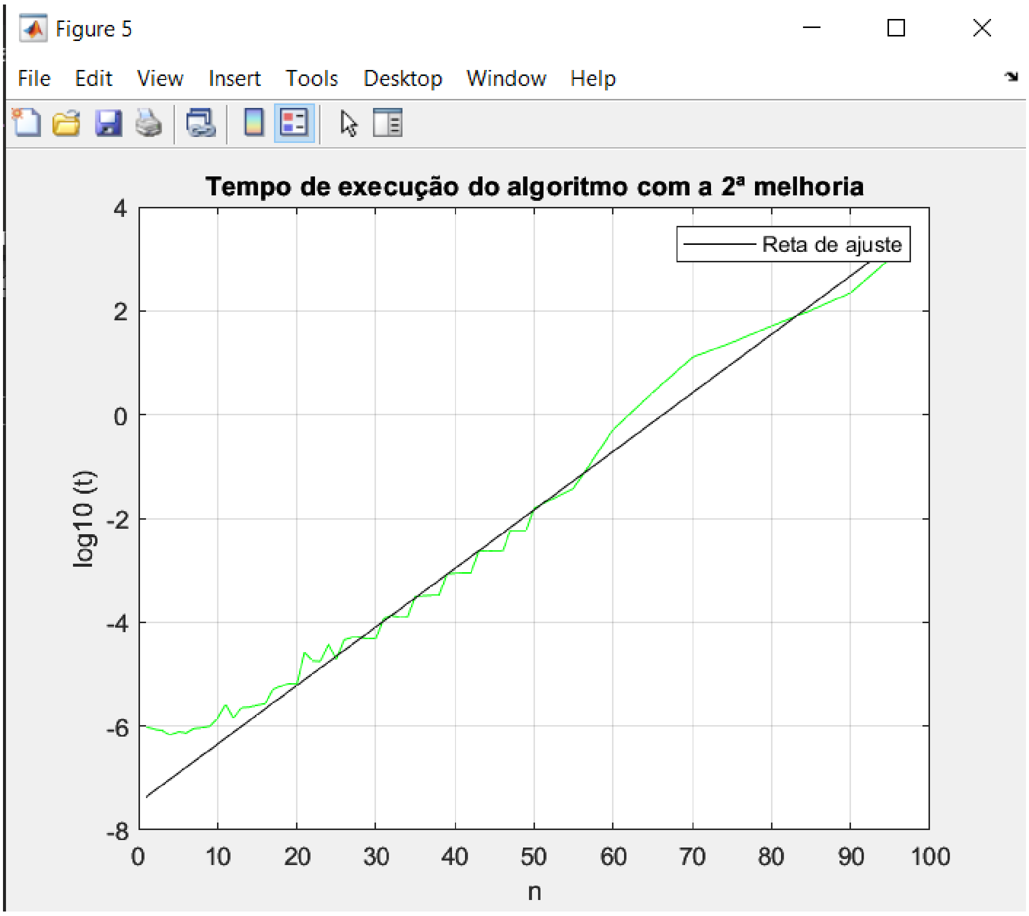

Figura 5- Reta de ajuste aos dados da 2ª melhoria

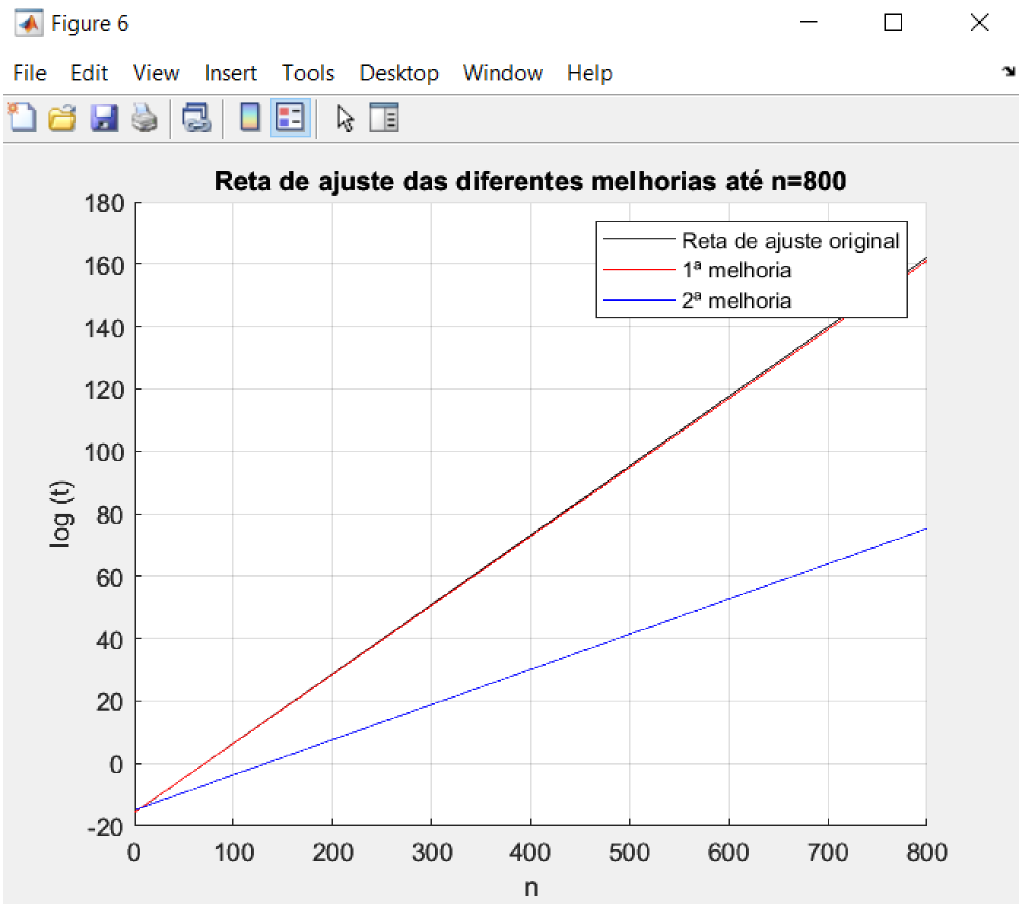

Figura 6- Comparação das diferentes retas de ajuste.

A 2° melhoria permite reduzir o tempo de execução para metade.

3ª melhoria – Acrescentar um outro if no código da função fornecida.

```
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(move_number >= solution_1_best.n_moves){
        return;
      }

      if(solution_1.positions[move_number] < solution_1_best.positions[move_number]){
        return;
      }

      solution_1_otimized_recursion(move_number + 1, position + new_speed, new_speed, final_position);
    }
  }
```

Neste segundo if, vai ser verificado se numa determinada posição foi possível, na solução já guardada, passar com uma velocidade maior do a que está a ser vista naquele momento. Se tiver sido possível então podemos abandonar a pesquisa desse ramo, pois interessa-nos andar sempre com a velocidade máxima possível.

Esta melhoria, também sendo simples, torna possível, juntamente com as outras melhorias resolver o problema para as 800 posições, sendo possível agora chegar à posição 800 em 1.203e-05 segundos (com o número mecanográfico 107457).

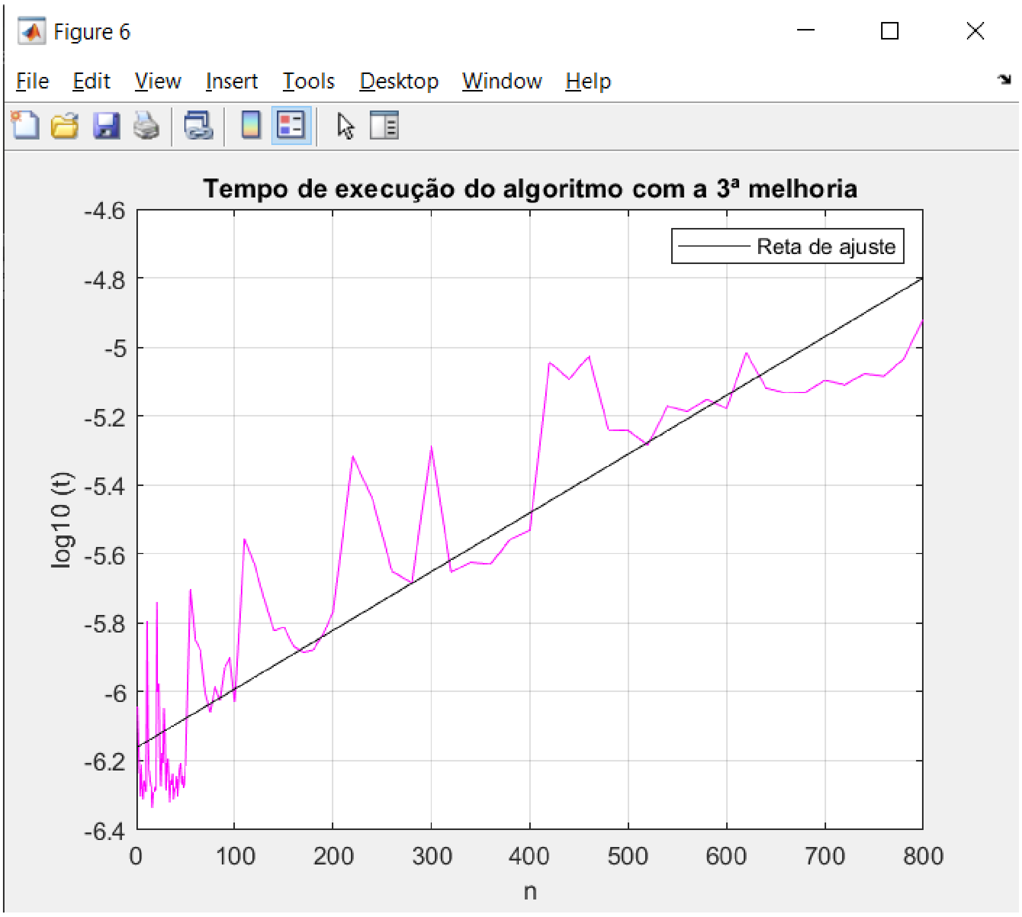

Figura 7 - Reta de ajuste aos dados da 3ª melhoria

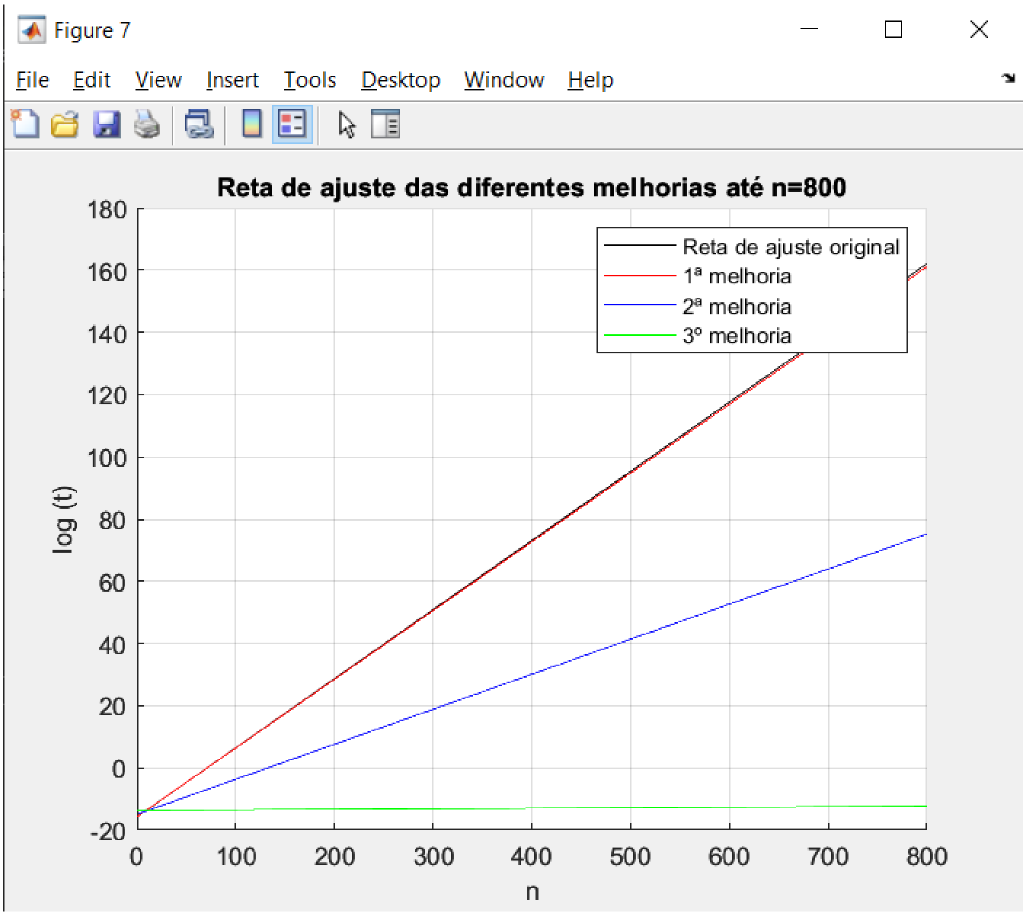

Figura 8- Comparação das diferentes retas de ajuste.

A 2ª e a 3ª melhoria foram baseadas no algoritmo de Branch and Bound, uma vez que a sua função é descartar logo um ramo se essa solução for pior que a "melhor" solução anteriormente encontrada e guardada.

Segundo Algoritmo

Este algoritmo foi criado pensando numa maneira de descobrir a melhor solução para o problema percorrendo a estrada toda uma única vez.

Posto isto, começamos por criar um ciclo while que permitisse percorrer toda a estrada. Dentro desse ciclo temos um if que vai verificar se é possível aumentar, diminuir ou manter a velocidade.

```
static void solution_2(int move_number, int position, int speed, int final_position)
{

  while ((position != final_position))
  {

    solution_2.positions[move_number] = position;

    if (respect_limits(position, speed + 1, final_position) == 1) //verificar se pode subir a velocidade
    {
      speed++;
      move_number++;
      position += speed;

    }else if(respect_limits(position, speed, final_position) == 1) //verificar se pode manter a velocidade
    {
      move_number++;
      position += speed;
    }else //pode diminuir a velocidade
    {
      speed--;
      move_number++;
      position += speed;
    }

  }

  solution_2.positions[move_number] = position;
  return;

}
```

Figura 9 - Função da Solução 2

Para conseguir fazer uma verificação que garantisse que em nenhum momento se iria desrespeitar as regras da estrada criámos uma função (respect_limits) que tem como parâmetros de entrada a posição onde se encontra, a velocidade a que está a tentar seguir e a posição final da estrada. O objetivo desta é verificar se a velocidade que está a tentar seguir é válida, e para essa avaliação, o algoritmo verifica, se com essa velocidade teria tempo de travar se já se encontrasse perto do fim da estrada. Verifica, também, se essa velocidade respeita a velocidade de todos os segmentos por onde vai passar até chegar à posição seguinte.

Se a velocidade cumprir estes dois requisitos a função vai retornar o valor 1, caso contrário retorna o valor 0.

```c
static int respect_limits(int position, int speed,int final_position) //verificar se respeita os limites de velocidade
{

  for(int s = speed; s >= 1; s--){                                  //verifica se consegue desacelearar até 1, caso esteja perto do fim da estrada

    for(int i = 0;i<=s;i++){                                        //verifica a velocidade em cada segmento da estrada por onde passa

      if(((position + i) > final_position) || max_road_speed[position + i] < s ){ //verifica se a posição atual + i é maior que a posição final
        return 0;                                                    //ou se a velocidade máxima da estrada é menor que a velocidade atual
      }
    }
    position += s;
  }
  return 1;
}
```

Figura 10 - Função usada na solução 2 para verificar a velocidade

Com esta solução temos que o tempo de execução para a posição 800 é 2.815e-06 segundos (com o número mecanográfico 107457).

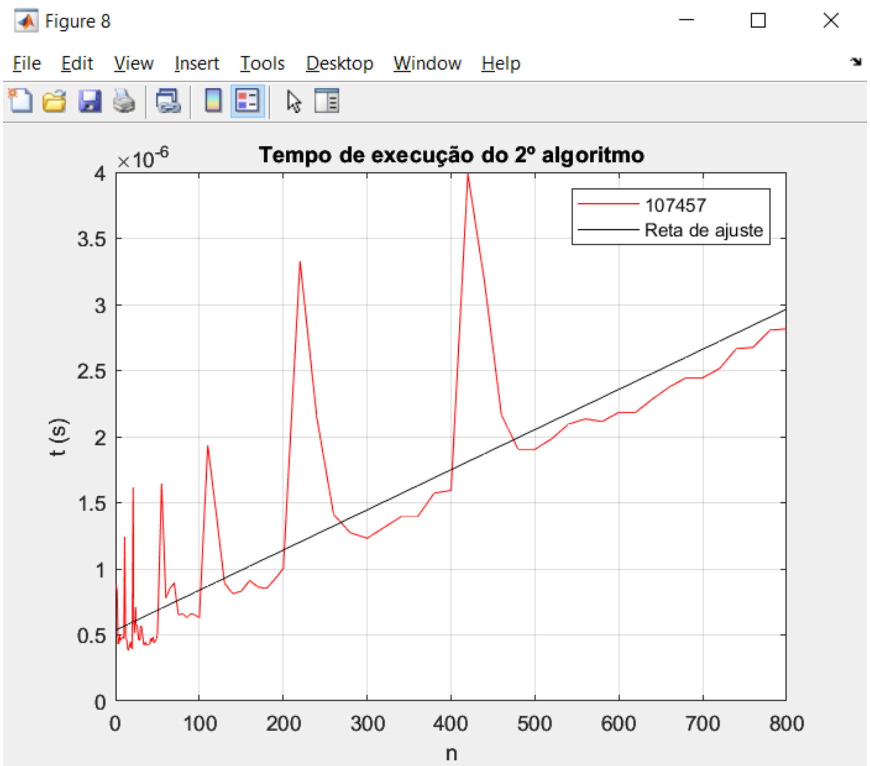

Figura 11-Reta de ajuste aos dados do 2º algoritmo

De notar que na Figura 11 não foi necessário aplicar logaritmos à construção da reta de ajuste, pois a mesma não é exponencial. Assim sendo, temos um gráfico do tempo (t(s)) de execução em função do número de segmentos da estrada (n).

Terceiro Algoritmo – Programação Dinâmica

Para este algoritmo utilizámos a programação dinâmica, que consiste em dividir um problema de otimização em subproblemas mais simples e guardar a solução para cada, de modo que cada subproblema seja resolvido só uma vez.

No contexto do problema em estudo, o que começámos por pensar foi, por exemplo, numa estrada com 10 segmentos os saltos que o carro vai dar até começar a travar, por já estar perto do fim da estrada, vão ser os mesmo que numa estrada com 20 segmentos até essa posição, e assim em diante.

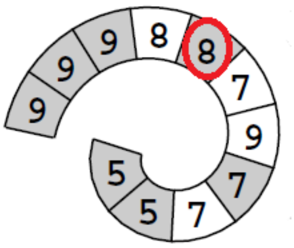

Figura 12 - 10 segmentos

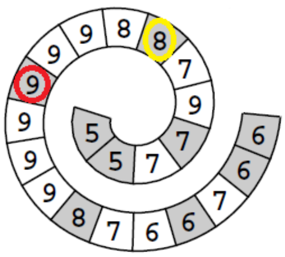

Figura 13 - 20 segmentos

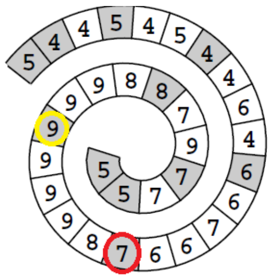

Figura 14 - 30 segmentos

As figuras acima demonstram o que foi descrito anteriormente, a posição marcada a vermelho é a posição onde o carro começa a travar, por já se encontrar perto do fim, e a amarelo está marcado a posição onde o algoritmo vai começar a nova procura, pois, as posições que estão para trás são iguais às já pesquisadas anteriormente.

Com este pensamento, criámos um algoritmo que segue a mesma ideia do segundo algoritmo, mas neste é criado um array com as posições onde o carro passou até à posição onde começa a travar, por se encontrar perto do fim da estrada, e é também guardado a velocidade com que ia nessa posição. Assim, na próxima pesquisa é reaproveitado esse array, e em vez da pesquisa começar no inico da estrada com velocidade 0, a pesquisa é iniciada na última posição desse array e com a velocidade guardada, uma vez que até aí as posições de paragem vão ser sempre as mesmas, o que torna a procura mais rápida e eficiente, evitando assim que faça duas vezes a mesma pesquisa.

Para esta solução foi criada uma estrutura nova (solution3_t) para conseguir guardar o número de saltos, a velocidade, a posição em que ficou e o array das posições onde passou, e poder assim usar esses valores quando necessário.

```c
typedef struct
{
  int move_number;
  int speed;
  int position; // the positions (the
  int positions[1 + _max_road_size_];
} solution3_t;
```

Figura 15 - Estrutura criada para a solução 3

Foi também reaproveitado o código da solução 2, fazendo só algumas alterações, que são:

- Dentro do while foi acrescentado um if que só é executado na primeira vez em que o carro chega a uma posição com uma velocidade onde tem de começar a reduzir para respeitar os limites até ao fim da estrada.

```c
static void solution_3(int move_number, int position, int speed, int final_position)
{
  int a = 0;
  while ((position != final_position))
  {
    solution_3_count++;
    solution3.positions[move_number] = position;

    int res = respect_limitsV2(position, speed + 1, final_position); //Iniciar a variável res verificando se pode acelerar

    if (res == 1 && a == 0){
      solution3.move_number = move_number;
      solution3.position = position;
      solution3.speed = speed;
      a = 1;  //Flag para só executar este if a primeira vez que ele tiver de travar por já se encontrar perto da posição final
    }

    if (res == 0)
    {
      speed++;
      move_number++;
      position += speed;

    }else if(respect_limitsV2(position, speed, final_position) == 0) //verificar se pode manter a velocidade
    {
      move_number++;
      position += speed;
    }else //pode diminuir a velocidade
    {
      speed--;
      move_number++;
      position += speed;
    }

  }

  solution3.positions[move_number] = position;

  solution_3_best = solution3;
  solution_3_best.move_number = move_number;

  return;
}
```

Figura 16 - Função da Solução 3

- Na função que verifica os limites de velocidade foram alterados os valores de retorno. Como para esse algoritmo é necessário saber exatamente quando é que a verificação falha por se ultrapassar a posição final da estrada é retornado 1 quando isso acontece. De seguida é verificado se não é ultrapassado o limite de velocidade de nenhum segmento de estrada por onde passa, se for retorna 2. Por fim, se a velocidade passar estas duas verificações o valor de retorno é 0.

```c
static int respect_limitsV2(int position, int speed,int final_position) //verificar se não execede a velocidade em nenhuma estrada por onde passa
{
  solution_3_count++;
  //verifica se a soma de as posições ao desacelerar não ultrapassa a posição final
  if(((position + (speed*(speed+1))/2) > final_position)){
    return 1;
  }

  //Verifica se a velocidade máxima de cada segmento de estrada por onde passa é respeitada
  for(int s = speed; s >= 1; s--){
    for(int i = 0;i<=s;i++){
      if (max_road_speed[position + i] < s ){
        return 2;
      }
    }
    position += s;
  }
  return 0;
}
```

Figura 17 - Função usada na solução 3 para verificar a velocidade

Alterámos também a função que executa o código do terceiro algoritmo, para chamar a solução com o número de saltos, a posição e a velocidade guardadas na estrutura da solução 3.

```c
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.move_number = final_position + 100;
  solution_3(solution3.move_number, solution3.position, solution3.speed, final_position);
  solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}
```

Figura 18 - Função que chama a função da solução 3

Com este algoritmo temos que o tempo de execução para a posição 800 é 6.310e-07 segundos (com o número mecanográfico 107457).

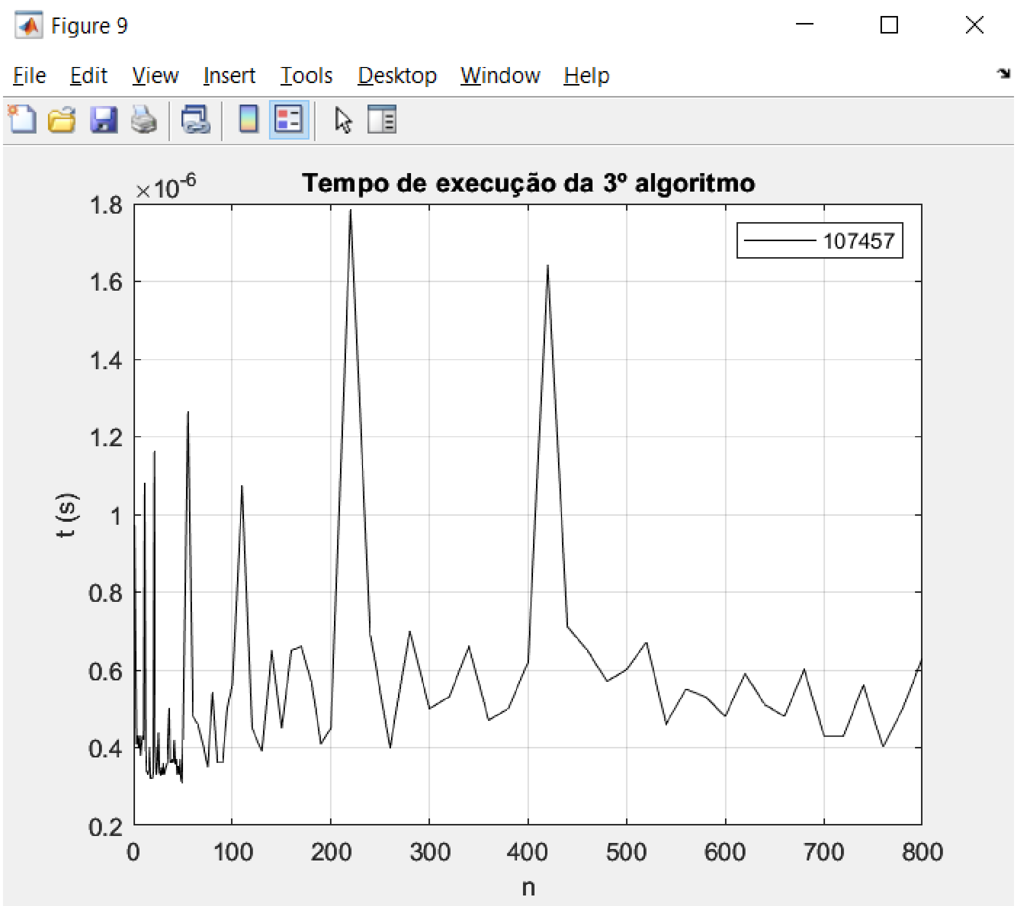

Figura 19- Dados do 3º algoritmo

Para este algoritmo, não fizemos a resta de ajuste aos dados visto que o gráfico é inconstante e tem bastante ruído. O gráfico é decrescente devido ao facto de que sempre que se inicia uma nova procura com um novo número de posição final o tempo é começado a zero, no entanto as primeiras posições possíveis já foram vistas e guardadas num array logo o tempo que demorou a fazer essa pesquisa anteriormente não vai ser contabilizado. Isto provoca estas inconsistências no gráfico.

Resultados

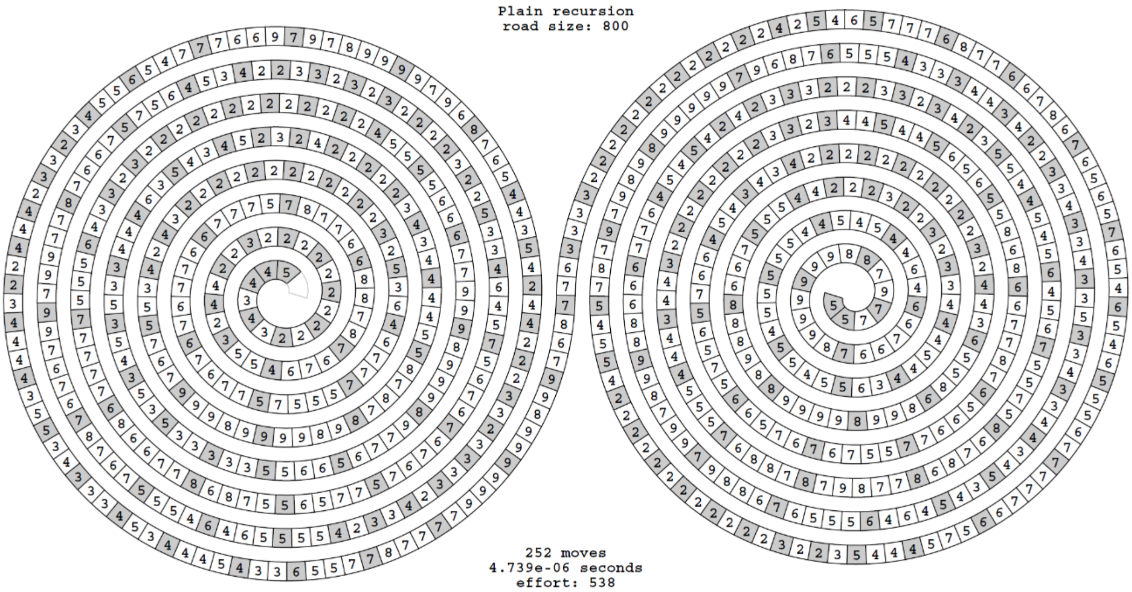

Figura 20 - Resultados com solução do professor otimizada

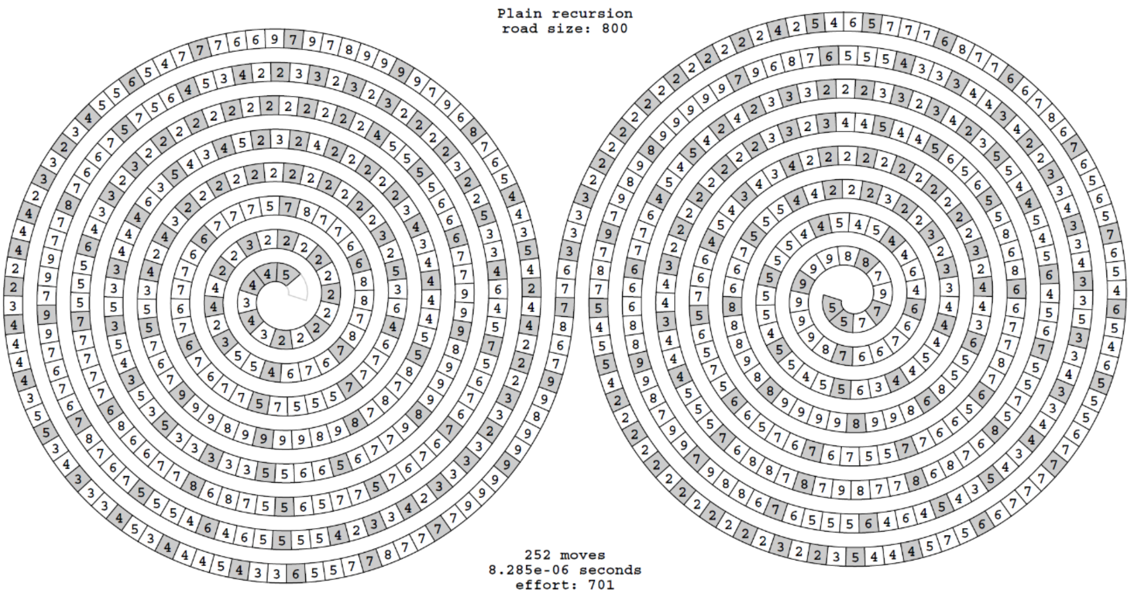

Figura 21 - Resultados com a segunda solução

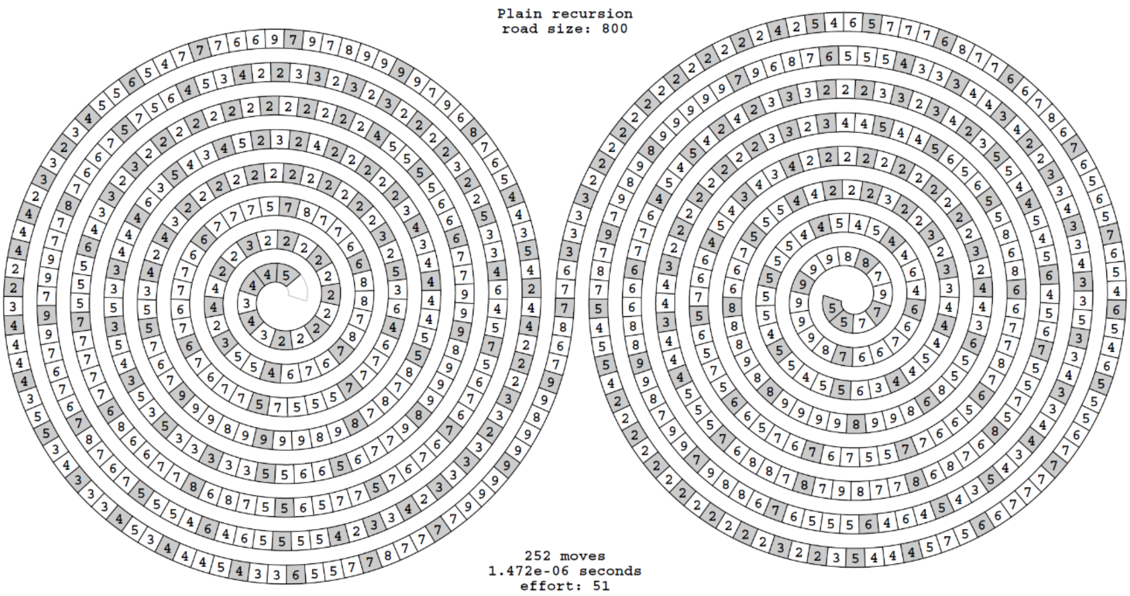

Figura 22 - Resultados com a terceira solução (Programação dinâmica)

Através destas imagens vemos que os resultados das 3 solução são iguais, ou seja, todos dão 252 saltos (com o número mecanográfico 107457). No entanto, os tempos de execução variam, o que nos permite concluir que a terceira solução é a melhor, uma vez que tem um tempo de execução menor. Estes resultados vão de encontro à teoria, pois esta solução usa programação dinâmica, que permite que a melhor solução para o problema seja combinada com a melhor solução pesquisada e guardada anteriormente, evitando assim que se volte a calcular uma coisa já antes vista, enquanto os outros dois algoritmos vão sempre calcular a solução partindo do zero.

- **Tempos de execução finais**

1º Algoritmo:

n	sol	plain recursion count	plain recursion cpu time
1	1	2	9.010e-07
2	2	3	7.110e-07
3	3	5	5.810e-07
4	3	5	5.010e-07
5	4	7	6.120e-07
6	4	8	5.210e-07
7	5	10	4.910e-07
8	5	12	5.510e-07
9	5	10	5.310e-07
10	6	13	5.110e-07
11	6	15	1.593e-06
12	6	13	7.120e-07
13	7	17	5.910e-07
14	7	20	5.510e-07
15	7	20	5.310e-07
16	7	16	4.610e-07
17	8	20	5.110e-07
18	8	23	5.110e-07
19	8	24	5.310e-07
20	8	21	5.210e-07
21	9	25	1.824e-06
22	9	29	1.002e-06
23	9	30	1.051e-06
24	9	28	6.410e-07
25	9	21	5.310e-07
26	10	25	6.610e-07
27	10	29	6.210e-07
28	10	29	9.010e-07
29	10	26	7.610e-07
30	10	21	5.210e-07
31	11	25	6.210e-07
32	11	28	6.410e-07
33	11	27	5.410e-07
34	11	23	4.800e-07
35	12	27	5.510e-07
36	12	30	5.410e-07
37	12	29	5.810e-07
38	12	25	4.910e-07
39	13	29	5.210e-07
40	13	32	5.310e-07
41	13	31	5.710e-07
42	13	27	5.010e-07
43	14	30	5.610e-07
44	14	32	6.010e-07
45	14	31	6.210e-07
46	14	28	5.410e-07
47	15	31	5.710e-07
48	15	33	5.310e-07
49	15	31	5.410e-07
50	16	34	6.110e-07
55	17	33	1.973e-06
60	20	39	1.413e-06
65	22	46	1.323e-06
70	24	51	9.920e-07
75	25	57	8.710e-07
80	26	66	1.031e-06
85	27	69	9.410e-07
90	28	61	1.183e-06
95	30	62	1.252e-06
100	33	68	9.310e-07
110	37	79	2.776e-06
120	40	90	2.345e-06
130	42	100	1.854e-06
140	44	105	1.503e-06
150	46	111	1.543e-06
160	49	101	1.353e-06
170	54	114	1.303e-06
180	57	124	1.323e-06
190	59	128	1.473e-06
200	61	140	1.703e-06
220	68	139	4.809e-06
240	76	163	3.627e-06
260	80	172	2.234e-06
280	86	178	2.073e-06
300	93	198	5.139e-06
320	97	201	2.224e-06
340	107	223	2.375e-06
360	111	231	2.344e-06
380	116	252	2.766e-06
400	125	268	2.945e-06
420	130	282	9.026e-06
440	136	291	8.075e-06
460	144	307	9.407e-06
480	150	329	5.751e-06
500	156	331	5.730e-06
520	164	349	5.189e-06
540	169	379	6.733e-06
560	175	370	6.502e-06
580	182	390	7.053e-06
600	186	399	6.632e-06
620	193	406	9.628e-06
640	200	429	7.604e-06
660	205	433	7.353e-06
680	212	446	7.384e-06
700	220	467	8.005e-06
720	224	476	7.765e-06
740	234	498	8.366e-06
760	238	511	8.235e-06
780	242	521	9.227e-06
800	252	538	1.206e-05

2º Algoritmo:

n	sol	plain recursion count	plain recursion cpu time
1	1	2	8.710e-07
2	2	5	8.220e-07
3	3	8	4.310e-07
4	3	7	4.310e-07
5	4	10	4.900e-07
6	4	10	4.600e-07
7	5	13	4.700e-07
8	5	13	4.810e-07
9	5	12	4.710e-07
10	6	15	4.910e-07
11	6	15	1.243e-06
12	6	15	4.710e-07
13	7	18	4.710e-07
14	7	18	4.310e-07
15	7	18	3.810e-07
16	7	17	4.010e-07
17	8	20	4.210e-07
18	8	20	4.510e-07
19	8	20	4.000e-07
20	8	20	4.000e-07
21	9	23	1.613e-06
22	9	23	6.010e-07
23	9	23	5.200e-07
24	9	23	7.120e-07
25	9	22	5.810e-07
26	10	25	5.610e-07
27	10	25	4.910e-07
28	10	25	4.600e-07
29	10	25	4.610e-07
30	10	25	5.710e-07
31	11	28	5.610e-07
32	11	28	5.110e-07
33	11	28	4.610e-07
34	11	28	4.210e-07
35	12	31	4.400e-07
36	12	31	4.410e-07
37	12	31	4.210e-07
38	12	31	4.310e-07
39	13	34	4.210e-07
40	13	34	4.300e-07
41	13	34	4.310e-07
42	13	34	4.710e-07
43	14	37	4.510e-07
44	14	37	4.710e-07
45	14	37	4.810e-07
46	14	37	4.400e-07
47	15	40	4.510e-07
48	15	40	4.600e-07
49	15	40	4.710e-07
50	16	43	5.010e-07
55	17	45	1.644e-06
60	20	54	7.810e-07
65	22	60	8.520e-07
70	24	65	8.920e-07
75	25	67	6.510e-07
80	26	70	6.610e-07
85	27	72	6.310e-07
90	28	75	6.610e-07
95	30	81	6.510e-07
100	33	90	6.310e-07
110	37	101	1.934e-06
120	40	109	1.423e-06
130	42	114	8.910e-07
140	44	120	8.110e-07
150	46	126	8.310e-07
160	49	134	9.120e-07
170	54	149	8.620e-07
180	57	157	8.520e-07
190	59	161	9.220e-07
200	61	167	1.002e-06
220	68	187	3.326e-06
240	76	209	2.144e-06
260	80	219	1.412e-06
280	86	237	1.272e-06
300	93	256	1.232e-06
320	97	266	1.312e-06
340	107	295	1.393e-06
360	111	305	1.393e-06
380	116	319	1.573e-06
400	125	346	1.593e-06
420	130	358	3.987e-06
440	136	375	3.166e-06
460	144	399	2.164e-06
480	150	414	1.904e-06
500	156	431	1.904e-06
520	164	453	1.984e-06
540	169	467	2.094e-06
560	175	484	2.134e-06
580	182	503	2.114e-06
600	186	514	2.184e-06
620	193	534	2.184e-06
640	200	553	2.284e-06
660	205	567	2.375e-06
680	212	588	2.445e-06
700	220	610	2.445e-06
720	224	621	2.515e-06
740	234	650	2.665e-06
760	238	660	2.675e-06
780	242	671	2.805e-06
800	252	701	2.815e-06

3º Algoritmo

n	sol	plain recursion count	plain recursion cpu time
1	1	2	9.720e-07
2	2	5	6.820e-07
3	3	6	4.110e-07
4	3	5	4.310e-07
5	4	6	4.010e-07
6	4	6	4.310e-07
7	5	9	3.810e-07
8	5	9	4.110e-07
9	5	8	4.310e-07
10	6	9	4.210e-07
11	6	9	1.082e-06
12	6	9	3.910e-07
13	7	12	3.410e-07
14	7	12	3.310e-07
15	7	12	3.410e-07
16	7	11	4.010e-07
17	8	12	3.210e-07
18	8	12	3.200e-07
19	8	12	3.200e-07
20	8	12	3.400e-07
21	9	15	1.162e-06
22	9	15	4.010e-07
23	9	15	3.310e-07
24	9	15	3.610e-07
25	9	14	4.400e-07
26	10	15	3.400e-07
27	10	15	3.300e-07
28	10	15	3.500e-07
29	10	15	3.310e-07
30	10	15	3.610e-07
31	11	18	3.310e-07
32	11	18	3.410e-07
33	11	18	3.500e-07
34	11	18	3.600e-07
35	12	21	4.310e-07
36	12	21	5.010e-07
37	12	18	3.610e-07
38	12	18	3.610e-07
39	13	21	3.710e-07
40	13	21	3.610e-07
41	13	21	4.210e-07
42	13	15	3.600e-07
43	14	18	3.710e-07
44	14	15	3.310e-07
45	14	15	3.510e-07
46	14	15	3.300e-07
47	15	18	3.700e-07
48	15	15	3.210e-07
49	15	15	3.110e-07
50	16	18	4.210e-07
55	17	20	1.263e-06
60	20	21	4.810e-07
65	22	27	4.610e-07
70	24	14	4.110e-07
75	25	14	3.510e-07
80	26	15	5.410e-07
85	27	17	3.610e-07
90	28	18	3.610e-07
95	30	24	5.010e-07
100	33	30	5.610e-07
110	37	20	1.072e-06
120	40	17	4.510e-07
130	42	17	3.910e-07
140	44	21	6.510e-07
150	46	24	4.510e-07
160	49	26	6.510e-07
170	54	39	6.610e-07
180	57	17	5.710e-07
190	59	16	4.110e-07
200	61	18	4.510e-07
220	68	38	1.783e-06
240	76	31	6.910e-07
260	80	22	4.000e-07
280	86	36	7.010e-07
300	93	31	5.010e-07
320	97	22	5.310e-07
340	107	47	6.610e-07
360	111	19	4.710e-07
380	116	29	5.010e-07
400	125	45	6.210e-07
420	130	21	1.643e-06
440	136	29	7.110e-07
460	144	39	6.510e-07
480	150	24	5.710e-07
500	156	32	6.010e-07
520	164	52	6.720e-07
540	169	23	4.610e-07
560	175	35	5.510e-07
580	182	49	5.310e-07
600	186	23	4.810e-07
620	193	38	5.910e-07
640	200	49	5.110e-07
660	205	26	4.810e-07
680	212	42	6.010e-07
700	220	37	4.310e-07
720	224	23	4.310e-07
740	234	47	5.610e-07
760	238	19	4.010e-07
780	242	26	5.010e-07
800	252	51	6.310e-07

Conclusões finais

Com a realização deste trabalho, conseguimos aprofundar bastante os nossos conhecimentos em linguagem C e também adquirir novos conhecimentos sobre alguns algoritmos como o depth first search, o branch and bound e programação dinâmica. Antes do início da execução do projeto tínhamos pouco conhecimento sobre a otimização de algoritmos e como é que uma simples alteração num código consegue fazer tanta diferença na complexidade computacional e no tempo de execução.

Sentimos alguma dificuldade a entender a solução já fornecida devido à função recursiva e à falta de experiência de trabalho na linguagem em C. No entanto essas dificuldades foram diminuindo à medida que tínhamos mais aulas e que fazíamos mais pesquisas sobre a matéria, conseguindo assim ultrapassá-las.

Por fim, podemos afirmar que os objetivos propostos foram alcançados com sucesso, visto que conseguimos implementar 2 soluções de forma eficiente e otimizar a solução já fornecida com sucesso, chegando assim a ter 3 algoritmos que atingem a posição final com o menor número de saltos no tempo de execução de microssegundos.

Web grafia

- https://brilliant.org/wiki/depth-first-search-dfs/ consultado em 10/11/2022.
- https://www.baeldung.com/cs/branch-and-bound consultado em 10/11/2022.
- https://web.stanford.edu/class/cs106b-8/lectures/backtracking-optimization/Lecture13.pdf consultado em 22/11/2022.
- https://www.freecodecamp.org/news/demystifying-dynamic-programming-3efafb8d4296/#:~:text=Dynamic%20Programming%20Defined,example%20of%20a%20sub-problem consultado em 23/11/2022

Apêndice

Código C

- Speed_run.c

peed_run.c

```c
//
// AED, August 2022 (Tomás Oliveira e Silva)
//
// First practical assignement (speed run)
//
// Compile using either
//   cc -Wall -O2 -D_use_zlib_=0 solution_speed_run.c -lm
// or
//   cc -Wall -O2 -D_use_zlib_=1 solution_speed_run.c -lm -lz
//
// Place your student numbers and names here
//   N.Mec. 107457  Name: Diana Raquel Rodrigues Miranda
//   N.Mec. 107403 Name: João NUno da Silva Luís
//
//
// static configuration
//
#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 "../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;

typedef struct
{
  int move_number;                          // the number of moves (the number of positions is one more than the number of moves)
  int speed;
  int position; // the positions (the first one must be zero)
  int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
} solution3_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

//Solution 2
static solution_t solution_2, solution_2_best;
static double solution_2_elapsed_time; // time it took to solve the problem
static unsigned long solution_2_count; // effort dispended solving the problem

//Solution 3
static solution3_t solution3, 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
```

```c
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 void solution_1_otimized_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)
      {
        if(move_number >= solution_1_best.n_moves){
          return;
        }

        if(solution_1.positions[move_number] < solution_1_best.positions[move_number]){
          return;
        }

        solution_1_otimized_recursion(move_number + 1, position + new_speed, new_speed, final_position);
      }
    }

}

static int respect_limits(int position, int speed,int final_position) //verificar se não execede a velocidade em nenhuma estrada por onde passa
{
  solution_2_count++;
  for(int s = speed; s >= 1; s--){
    for(int i = 0;i<=s;i++){
      if(((position + i) > final_position) || max_road_speed[position + i] < s ){
        return 0;
      }
    }
    position += s;
  }
  return 1;
}
```

```
static void solution2(int move_number, int position, int speed, int final_position)
{
  while ((position != final_position))
  {
    solution_2_count++;
    solution_2.positions[move_number] = position;

    if (respect_limits(position, speed + 1, final_position) == 1) //verificar se pode subir a velocidade
    {
      speed++;
      move_number++;
      position += speed;

    }else if(respect_limits(position, speed, final_position) == 1) //verificar se pode manter a velocidade
    {
      move_number++;
      position += speed;
    }else //pode diminuir a velocidade
    {
      speed--;
      move_number++;
      position += speed;
    }

  }

  solution_2.positions[move_number] = position;
  //para  pintar a casa final
  solution_2_best = solution_2;
  solution_2_best.n_moves = move_number;

  return;
}

static int respect_limitsV2(int position, int speed,int final_position) //verificar se não execede a velocidade em nenhuma estrada por onde passa
{
  solution_3_count++;
  //verifica se a soma de as posições ao desacelerar não ultrapassa a posição final
  if(((position + (speed*(speed+1))/2) > final_position)){
    return 1;
  }

  //Verifica se a velocidade máxima de cada segmento de estrada por onde passa é respeitada
  for(int s = speed; s >= 1; s--){
    for(int i = 0;i<=s;i++){
      if (max_road_speed[position + i] < s ){
        return 2;
      }
    }
    position += s;
  }
  return 0;
}

static void solution_3(int move_number, int position, int speed, int final_position)
{
  int a = 0;
  while ((position != final_position))
  {
    solution_3_count++;
    solution3.positions[move_number] = position;

    int res = respect_limitsV2(position, speed + 1, final_position); //Iniciar a variável res verificando se pode acelerar

    if (res == 1 && a == 0){
      solution3.move_number = move_number;
      solution3.position = position;
      solution3.speed = speed;
      a = 1;  //Flag para só executar este if a primeira vez que ele tiver de travar por já se encontrar perto da posição final
    }

    if (res == 0)
    {
      speed++;
      move_number++;
      position += speed;

    }else if(respect_limitsV2(position, speed, final_position) == 0) //verificar se pode manter a velocidade
    {
      move_number++;
      position += speed;
    }else //pode diminuir a velocidade
    {
      speed--;
      move_number++;
      position += speed;
    }

  }
```

```c
    solution3.positions[move_number] = position;

    solution_3_best = solution3;
    solution_3_best.move_number = move_number;

    return;

}

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_otimized_recursion(0,0,0,final_position);
    solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;
}

static void solve_2(int final_position)
{
    if (final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr, "solve_1: bad final_position\n");
        exit(1);
    }

    solution_2_elapsed_time = cpu_time();
    solution_2_count = 0ul;
    solution_2_best.n_moves = final_position + 100;
    solution2(0, 0, 0, final_position);
    solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}

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.move_number = final_position + 100;
    solution_3(solution3.move_number, solution3.position, solution3.speed, final_position);
    solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}
```

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

  srandom(0xAED2022);
  init_road_speeds();
  final_position = 30;
  //solve_3(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;
  solution_1_elapsed_time = 0.0;
  solution_2_elapsed_time = 0.0;
  solution_3_elapsed_time = 0.0;
  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);
  //printf("%3d ",final_position);
  // 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("                                |");
  }

  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);

  //second solution method (less bad)

  if(solution_2_elapsed_time < _time_limit_)
  {
    solve_2(final_position);
    if(print_this_one != 0)
    {
      sprintf(file_name,"%03d_2.pdf",final_position);
      make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_2_best.n_moves,&solution_2_best.positions[0],solution_2_elapsed_time,solution_2_count,"Plain recursion");
    }
    printf(" %3d %16lu %9.3e |",solution_2_best.n_moves,solution_2_count,solution_2_elapsed_time);
  }
  else
  {
    solution_2_best.n_moves = -1;
    printf("                                |");
  }

   //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);
   //third solution method (less bad)
   if(solution_3_elapsed_time < _time_limit_)
   {
     solve_3(final_position);
     if(print_this_one != 0)
     {
       sprintf(file_name,"%03d_3.pdf",final_position);
       make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_3_best.move_number,&solution_3_best.positions[0],solution_3_elapsed_time,solution_3_count,"Plain recursion");
     }
     printf(" %3d %16lu %9.3e |",solution_3_best.move_number,solution_3_count,solution_3_elapsed_time);
     //printf("%3d %16lu %9.3e",solution_3_best.move_number,solution_3_count,solution_3_elapsed_time);
   }
   else
   {
     solution_3_best.move_number = -1;
     printf("                                |");
   }

   // 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_
}
```

Código MATLAB

- Execution_time.m

```
clear;clc;
DATA = load("SolucaoProf1hour.txt");
SoFor = load("SolProfOtimizadaFor.txt");
ForE1If= load("SolProfOtimizadaForE1If.txt");
ForE2If= load("SolProfOtimizadaForE2If.txt");
Sol2 = load("Solution2_107457.txt");
Sol3 = load("Solution3_107457.txt");

n = DATA(:,1); % selecionar dos dados do .txt a primeira coluna com os valores de n
t = DATA(:,4); % selecionar dos dados do .txt a quarta coluna com os valores de n
%% Gráficos dos algoritmos originais
figure(1)
plot(n,t,"r") % gráfico super exponencial a partir do x=40
title("Tempo de execução da solução fornecida");
xlabel("n");
ylabel("t (s)")
grid on

figure(2)
semilogy(n,t,"g")
title("Tempo de execução do algoritmo");
xlabel("n");
ylabel("semilogy")
grid on

figure(3)
plot(n,log10(t),"b") % quase o mesmo que o plot do semilogy, mudando os valores do eixo y
title("Tempo de execução da solução fornecida");
xlabel("n");
ylabel("log10 (t)")
grid on

t_log =log10(t);
N = [n(20:end) 1+0*n(20:end)]; % começa no 20, porque é a partir desse n que a reta fica mais estável,
% para a reta de ajuste apanhar a maior parte dos dados
Coefs = pinv(N)*t_log(20:end); % matriz de regressão

hold on
Ntotal = [n n*0+1];
% regra de ajuste aos dados
P2= plot(n, Ntotal*Coefs, "k");
legend(P2,"Reta de ajuste")
hold off

t800_log = [800 1]* Coefs;

% gráfico para as 800 posições
% temos de calcular os t's ate a essa posição e não só o t=800
n= 1:800;

for i=n
    t(i)= [i 1]*Coefs;
    t(i)= 10.^t(i) / 3600 / 24 /365;
end
t_log =log10(t);
%% construir o grafico para a modificação do FOR
n_for = SoFor(:,1);
t_for = SoFor(:,4);

figure(4)
plot(n_for,log10(t_for),"b")
t_log_for =log10(t_for);
N = [n_for(20:end) 1+0*n_for(20:end)];
Coefs = pinv(N)*t_log_for(20:end); % matriz de regressão

hold on
```

```
Ntotal = [n_for n_for*0+1];
% regra de ajuste aos dados
P2= plot(n_for, Ntotal*Coefs, "k");
title("Tempo de execução do algoritmo com a 1ª melhoria");
xlabel("n");
ylabel("log10 (t)")
legend(P2,"Reta de ajuste")
grid on
hold off
t800_log_for = [800 1]* Coefs;

n= 1:800;
for i=n
    t_for(i)= [i 1]*Coefs;
    t_for(i)= 10.^t_for(i) / 3600 / 24 /365;
end
t_log_for =log10(t_for);
%% construir o grafico para a 2ª melhoria: FOR mais 1 IF
n_F1if = ForE1If(:,1);
t_F1if = ForE1If(:,4);

figure(5)
plot(n_F1if,log10(t_F1if),"g")
t_log_F1if =log10(t_F1if);
N = [n_F1if(20:end) 1+0*n_F1if(20:end)];
Coefs = pinv(N)*t_log_F1if(20:end); % matriz de regressão

hold on
Ntotal = [n_F1if n_F1if*0+1];
% regra de ajuste aos dados
P2= plot(n_F1if, Ntotal*Coefs, "k");
title("Tempo de execução do algoritmo com a 2ª melhoria");
xlabel("n");
ylabel("log10 (t)")
legend(P2,"Reta de ajuste")
grid on
hold off
t800_log_F1if = [800 1]* Coefs;

n= 1:800;
for i=n
    t_F1if(i)= [i 1]*Coefs;
    t_F1if(i)= 10.^t_F1if(i) / 3600 / 24 /365;
end
t_log_F1if =log10(t_F1if);

%% construir o grafico para a 2ª melhoria: FOR mais 2 IF
n_F2if = ForE2If(:,1);
t_F2if = ForE2If(:,4);

figure(6)
plot(n_F2if,log10(t_F2if),"m")
TempoRealPos800_log = log10(t_F2if(100,1)); % ir buscar o valor real do tempo demorado na posição 800 (mas em log)

t_log_F2if =log10(t_F2if);
N = [n_F2if(20:end) 1+0*n_F2if(20:end)];
Coefs = pinv(N)*t_log_F2if(20:end); % matriz de regressão

hold on
Ntotal = [n_F2if n_F2if*0+1];
% regra de ajuste aos dados
plot(n_F2if, Ntotal*Coefs, "k");
title("Tempo de execução do algoritmo com a 3ª melhoria");
xlabel("n");
ylabel("log10 (t)")
legend("107457","Reta de ajuste")
```

```
grid on
hold off
t800_log_F2if = [800 1]* Coefs;

n= 1:800;
for i=n
    t_F2if(i)= [i 1]*Coefs;
    t_F2if(i)= 10.^t_F2if(i) / 3600 / 24 /365;
end
t_log_F2if =log10(t_F2if);
%% Todos os gráficos dos diferentes algoritmos para 800 n's da solução 1
figure(7)
hold on
plot(n,t_log, "k")
plot(n,t_log_for, "r")
plot(n,t_log_F1if, "b")
plot(n,t_log_F2if, "g")
xlabel("n");
ylabel("log (t)");
title("Reta de ajuste das diferentes melhorias até n=800")
legend("Reta de ajuste original","1ª melhoria","2ª melhoria","3º melhoria");
grid on
hold off
%% construir o grafico para a 2º solução
n_Sol2 = Sol2(:,1);
t_Sol2 = Sol2(:,4);
figure(8)
plot(n_Sol2,t_Sol2,"r")
title("Tempo de execução do 2º algoritmo");
xlabel("n");
ylabel("t (s)")
grid on
hold on
N = [n_Sol2(20:end) 1+0*n_Sol2(20:end)];
Coefs = pinv(N)*t_Sol2(20:end); % matriz de regressão

hold on
Ntotal = [n_Sol2 n_Sol2*0+1];
% regra de ajuste aos dados
plot(n_Sol2, Ntotal*Coefs, "k");
legend("107457","Reta de ajuste")
%% construir o grafico para a 3º solução
n_Sol3 = Sol3(:,1);
t_Sol3 = Sol3(:,4);
figure(9)
plot(n_Sol3,t_Sol3,"k")
%axis([0 800 0 0.000010]) % faria sentido acrecentar um axis propositado para parecer constante??
title("Tempo de execução da 3º algoritmo");
xlabel("n");
ylabel("t (s)")
grid on
legend("107457")
```