universidade de aveiro deti departamento de electrónica, telecomunicações e informática

# Trabalho Prático 1

# Speed Run

## Relatório

## Turma P8


**Realizado por:**

Diogo Falcão Nº108712 - Licenciatura em Engenharia Informática – **50%**

José Gameiro Nº108840 - Licenciatura em Engenharia Informática – **50%**

# *Índice:*

# *1. Introdução*

Para o projeto "Speed Run" da cadeira Algoritmos e Estruturas de Dados, foi-nos proposto otimizar um programa com base no código fornecido pelo professor. O objetivo deste programa é calcular, para uma estrada constituída por blocos com velocidade máximas, o número mínimo de movimentos de um carro desde o início ao fim de toda a estrada.

Existem alguns aspetos a ter em conta: toda os blocos da estrada são aproximadamente da mesma largura e como já referido, têm todos uma velocidade máxima estabelecida aleatória entre um e nove. A velocidade é medida pela quantidade de blocos que o carro irá fazer num único movimento. Em cada movimento, o carro pode diminuir, manter ou aumentar a sua velocidade uma unidade. Por fim, no último bloco, tem de se garantir que a velocidade do carro vai ser 1, para depois descer para 0 na última casa e assim, parar.

Este relatório irá demonstrará que existem soluções recursivas e iterativas mais eficientes através de uma breve explicação de cada solução, gráficos dos tempos de execução e ficheiros PDF gerados por estas.

## 1.1. Conceitos Base

**Branch and Bound** é uma técnica algorítmica que é utilizado muitas vezes para resolver problemas de otimização combinatória. Os problemas em que se aplica esta técnica são do tipo exponencial em termos de complexidade algorítmica que podem necessitar da análise de todas as permutações possíveis. Esta técnica é utilizada frequentemente para problemas deste género pois consegue encontrar uma solução de forma bastante rápida.

A abordagem **Back Tracking** é utilizada para resolver problemas de forma recursiva, onde se tenta construir elaborar uma solução de forma incremental, removendo as soluções que falham nas restrições criadas a qualquer momento da execução da solução.

**Tabulation** é um método utilizado para a resolução de problemas com programação dinâmica. Este método baseia-se em criar uma tabela e ao longo do tempo ir preenchendo-a com dados do problema em questão, para depois se poder encontrar uma solução, com base no resultado da tabela. É considerado um método ascendente, ou seja, começamos por resolver o problema pelos casos mais simples inserindo na tabela os dados e completando com os dados dos casos seguintes até ao topo da tabela. Esta implementação é considerada iterativa.

**Memoization** é uma forma de armazenar os dados de um problema em cache, que será utilizada em programação dinâmica. O objetivo da cache é melhorar o desempenho de programas e manter os dados acessíveis para que possam ser utilizados em futuros algoritmos. Isto faz com que o esforço de calcular novamente a solução para o mesmo problema seja removido.

# *2. Desenvolvimento*

## 2.1. Solution_1_Recursion (Disponibilizada pelo professor)

```
78  static void solution_1_recursion(int move_number,int position,int speed,int final_position)
79  {
80    int i,new_speed;
81
82    // record move
83    solution_1_count++;
84    solution_1.positions[move_number] = position;
85    // is it a solution?
86    if(position == final_position && speed == 1)
87    {
88      // is it a better solution?
89      if(move_number < solution_1_best.n_moves)
90      {
91        solution_1_best = solution_1;
92        solution_1_best.n_moves = move_number;
93      }
94      return;
95    }
96    // no, try all legal speeds
97    for(new_speed = speed - 1;new_speed <= speed + 1;new_speed++)
98      if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
99      {
00        for(i = 0;i <= new_speed && new_speed <= max_road_speed[position + i];i++)
01          ;
02        if(i > new_speed)
03          solution_1_recursion(move_number + 1,position + new_speed,new_speed,final_position);
04      }
05  }
```

**Fig.1 –** Solução recursiva fornecida pelo professor

Esta função dada, é uma função recursiva que usa 4 inteiros como argumentos, “move_number” – o número de blocos que o carro irá percorrer em um único movimento, “position” – posição em que o carro se encontra no movimento atual, “speed” – velocidade do carro numa determinada posição e “final_position” – posição final a que o carro terá de chegar. Em cada iteração, a função atualiza todos estes valores (à exceção da final position) e conta o número de movimentos, guardando-os em “solution_1”.

Se a posição não for igual à posição final e a velocidade não for igual a 1, ou seja, enquanto o carro não parar, testa-se qual a melhor opção entre descer velocidade, manter velocidade ou aumentar velocidade. Para cada uma destas opções, segue-se um “if statement” que averigua se a nova velocidade se situa entre o intervalo 0 a 9 e se a nova posição é menor ou igual à posição final.

Em seguida, se a nova velocidade cumprir todos os requisitos anteriormente impostos, existe um ciclo for a percorrer todos os blocos da estrada que a nova velocidade irá avançar. Aqui, certifica-se que em nenhum destes blocos, a velocidade é ultrapassada.

Finalmente, se o último “if statement” se verificar, a função irá ser executada novamente com os argumentos “move_number”, “position” e “new_speed” atualizados.

A função recursiva "solution_1_recursion", faz uma pesquisa em árvore, isto é, cada vez que se executa a função, traça-se um caminho possível até à casa final. Se durante este percurso alguma casa não verificar as condições impostas, a função vai ter de voltar ao nó anterior da árvore e calcular a partir daí outras possibilidades de caminho.

No início do programa, estes "andar para a frente e para trás" não exigem muitos cálculos e são relativamente rápidos de calcular. No entanto, a cada passo, a árvore vai ficando cada vez mais complexa e a pesquisa torna-se cada vez mais intensiva. Através da imagem abaixo é possível constatar o referido.

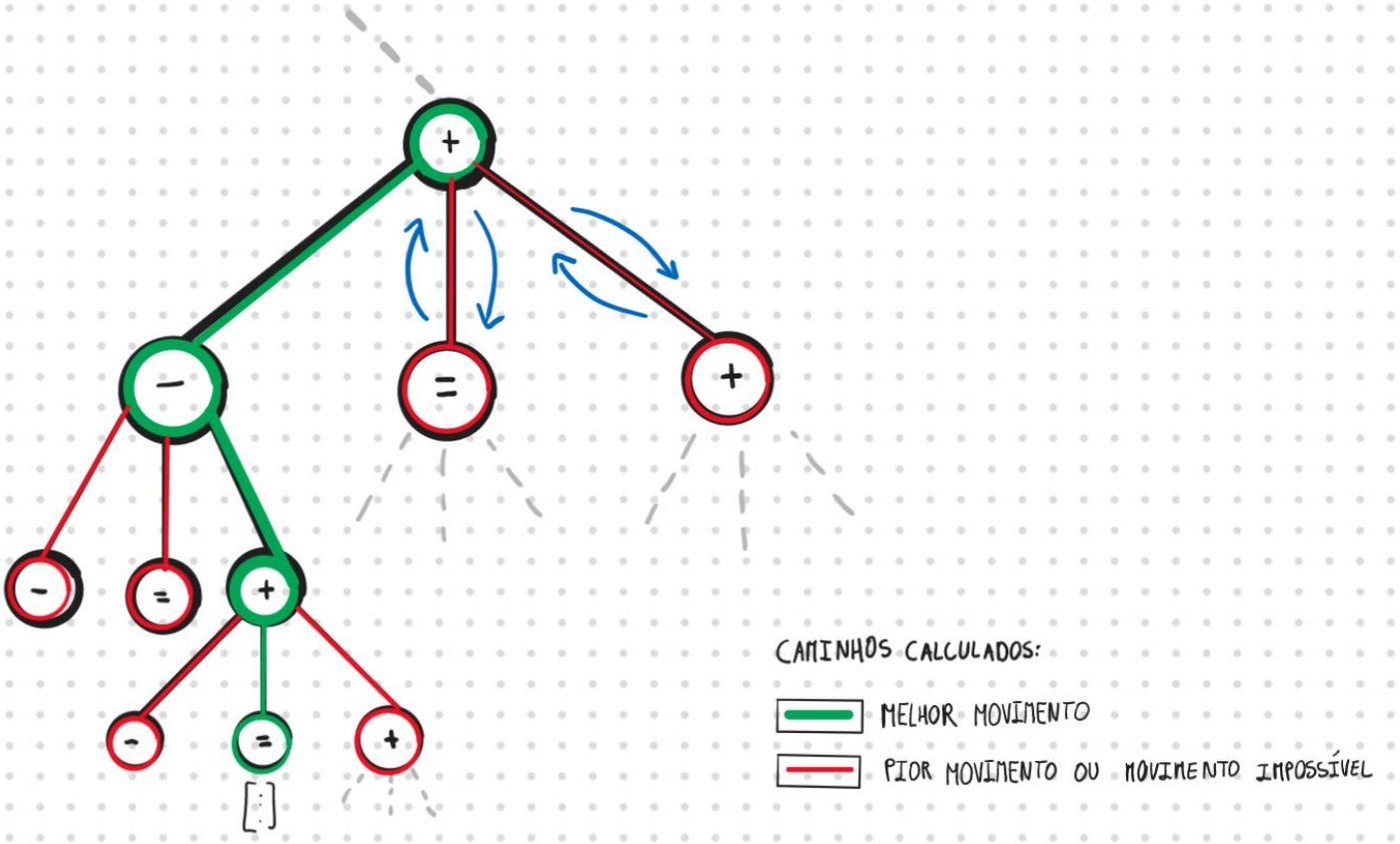


**Fig.2 –** Ilustração da pesquisa em árvore

Para cada nó são gerados outros três. Estes representam as velocidades possíveis: pelo nó da esquerda diminui-se a velocidade do nó anterior, pelo nó do meio, mantém-se a velocidade e pelo nó da direita, aumenta-se a velocidade.

O algoritmo verifica para o primeiro nó gerado (com a velocidade mais pequena), se o caminho é possível. Se sim, avança para o próximo nó, se não, volta ao nó anterior e testa para outra velocidade. Na prática, ao correr o programa com a função dada, observamos que o tempo do CPU aumenta exponencialmente.

Mostra-se que esta função tem uma pesquisa lenta visto que imposto o limite de 1 hora, o programa apenas chegou a


**Fig.3 –** Gráfico do tempo de execução da função recursiva fornecida pelo professor

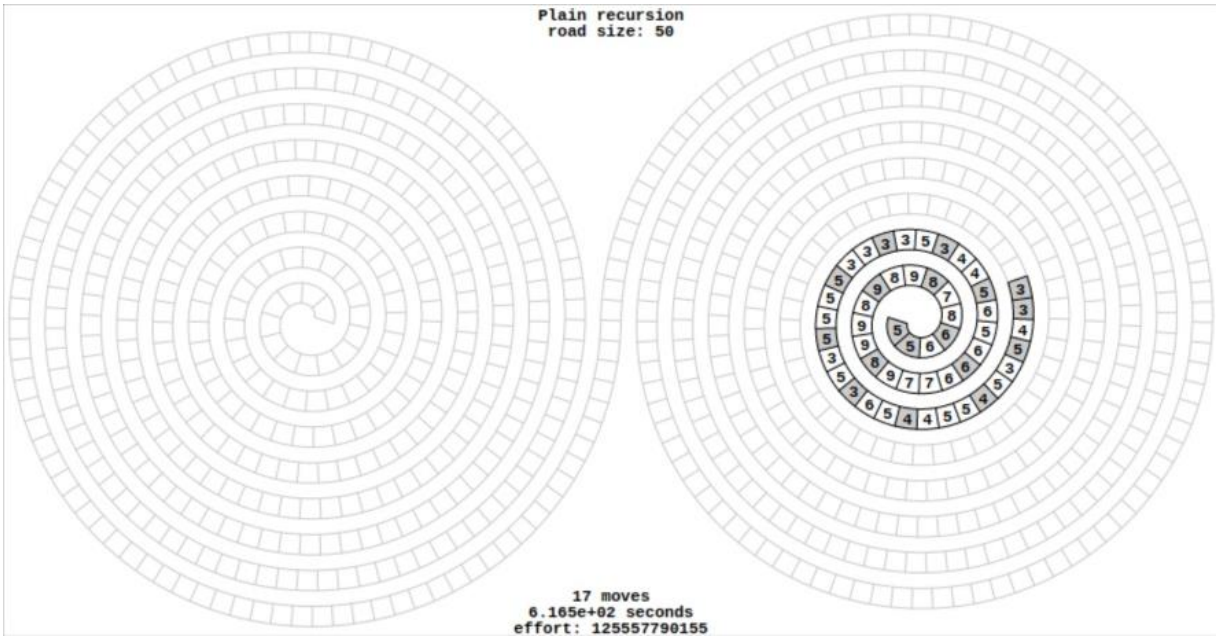


**Fig.4 –** Ficheiro pdf gerado pela função recursiva fornecida pelo professor

## 2.2. Solution_1_recursion (melhorada)

Seguindo o problema de recursividade e instabilidade da função dada, a nossa primeira ideia foi a de mudar a ordem pela qual a função testa a melhor velocidade para o carro. A função calcularia primeiro um possível caminho aumentado a velocidade do carro, depois testaria um caminho quando ficasse com a mesma velocidade do carro e por último, calcularia um caminho se descêssemos a velocidade do carro.

Em teoria, a velocidade de execução deste algoritmo ia aumentar, no entanto, mais tarde, percebemos que independentemente da ordem pela qual a função testa as velocidades, o melhor caminho poderá ser constituído por uma mesma redução ou aumento das velocidades, calculadas apenas por ordens diferentes.

Desta forma, decidimos incluir um novo if statement que retorna assim que possível uma solução de um caminho, sem necessitar de calcular as outras opções. A condição retorna imediatamente o cálculo do primeiro caminho válido.

```
if (solution_1_best.positions[move_number] > solution_1.positions[move_number])
  return;
```

**Fig.5 –** Condição lógica da função recursiva melhorada

Esta aperfeiçoamento à função resulta no uso de uma perspetiva algorítmica “Branch and Bound”, que resolveu este problema de uma maneira relativamente rápida, diminuindo drasticamente o tempo de execução.

De modo a incluir programação dinâmica, tentámos implementar uma abordagem algorítmica chamada de Back Tracking. Esta abordagem vem tentar tornar ainda mais eficiente o programa. Testámos este método implementando um array booleano de 3 dimensões (move_number, position e speed) chamado visited. À medida que o programa ia correndo, com esta solução, era guardado no array o move_number, a posição e a velocidade. Criámos uma condição em que caso a solução já tivesse encontrado uma solução com os valores de move_number, posição e velocidade, este iria utilizar a solução já encontrada para aqueles valores.  Sempre que esta solução era chamada, seria necessário limpar o array visited visto que na posição inicial, todas as variáveis presentes no array são 0. Ao correrm esta solução verificámos que o esforço e o tempo de execução aumentaram por isso decidimos não a incluir na solução.

```
120  static void solve_1(int final_position)
121  {
122    // Clear the visited array
123    // for (int i = 0; i < final_position; i++)
124    //   for (int j = 0 ; j < final_position; j++)
125    //     for (int k = 0; k < _max_road_speed_; k++)
126    //       visited[i][j][k] = 0;
127
```

```
// backtrack
if (visited[move_number][position][speed] == 1)
  return;

visited[move_number][position][speed] = 1;
```

**Fig.6 –** Código implementado para a abordagem Back Tracking

Esta função apresenta uma evolução completa em relação aos tempos de execução com a função dada, como é possível verificar no gráfico abaixo.

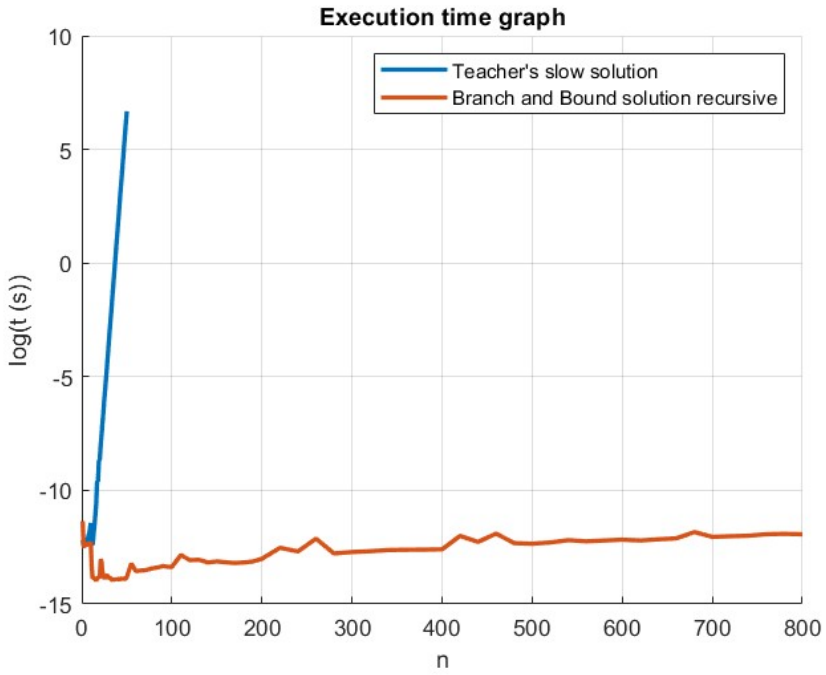


**Fig.7 –** Gráficos do tempo de execução da função recursiva melhorada e lenta

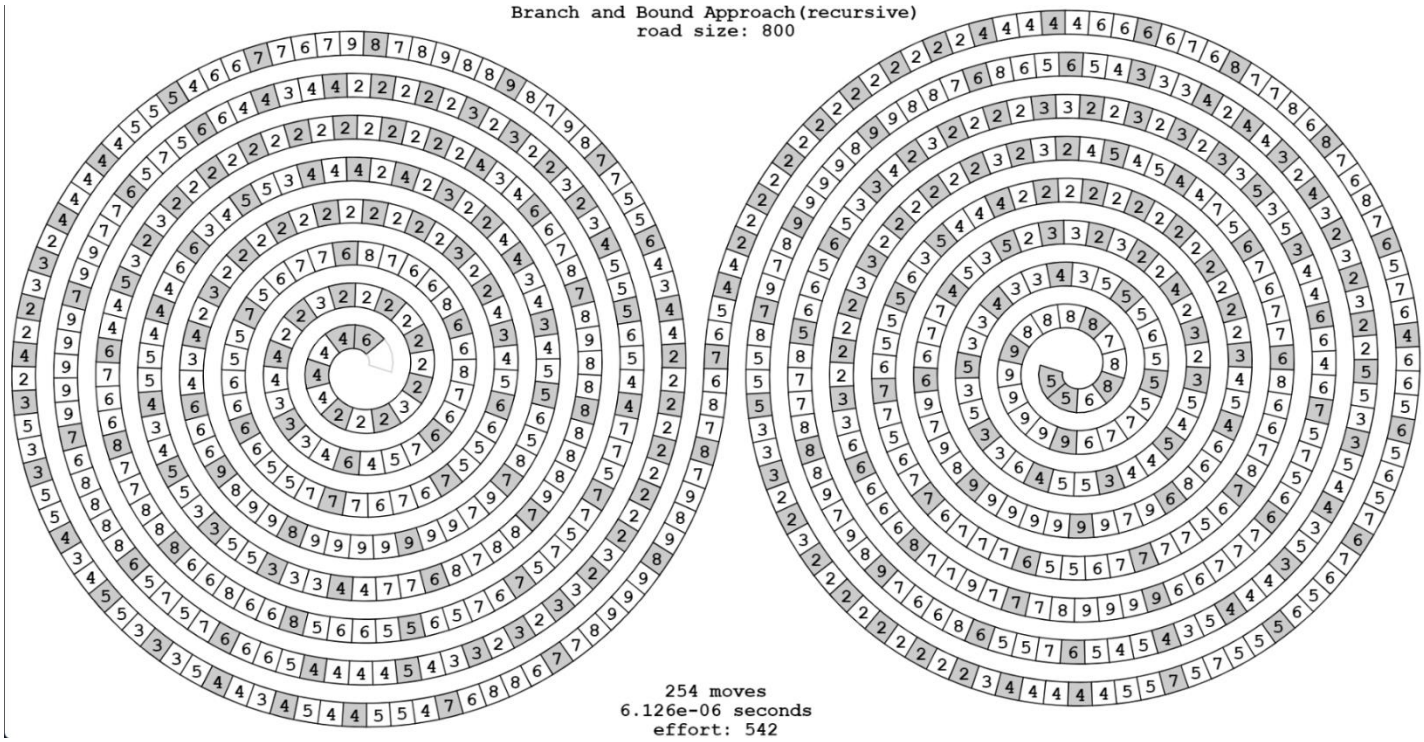


**Fig.8 –** Ficheiro pdf gerado pela função recursiva fornecida melhorada

## 2.3. Solution_2_Recursion

Desenvolvemos outra solução recursiva parecida com a anterior. Esta tem um modo de correr diferenciado: pretende atualizar os valores retornados desde o pior até ao melhor. Para isto, adicionámos uma condição if caso esta já tenha encontrado uma solução melhor, ou seja, uma solução com menos esforço irá parar a execução.

Em vez de se calcular todas as diferentes combinações do caminho que o carro tem de percorrer, a função retorna instantaneamente uma solução, porque esta "será a melhor solução". Desta forma, na primeira execução, o efford situa-se nos 800, depois na segunda execução reduzirá e assim por diante. Quando numa execução, a solução para certos movimentos já foi encontrada, retorna ou seja interrompe a sua execução.

Comparando esta solução com a função dada e com a função recursiva anterior, nota-se que, por muito relativa que seja a diferença, uma ligeira progressão a nível de tempo de execução.

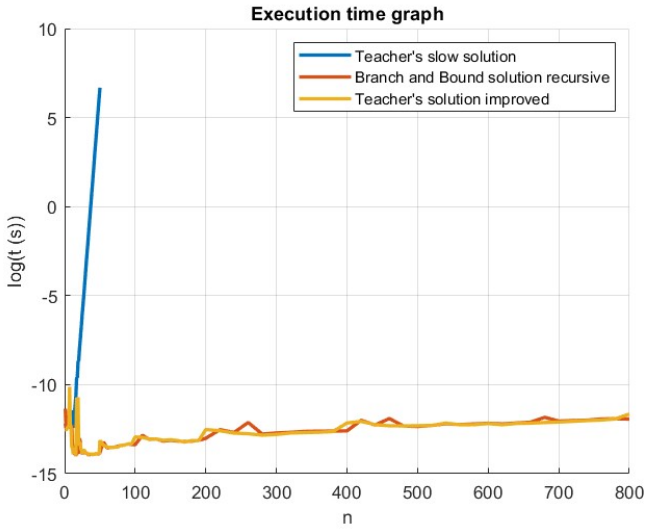


**Fig.9 –** Gráficos do tempo de execução das soluções recursivas criadas

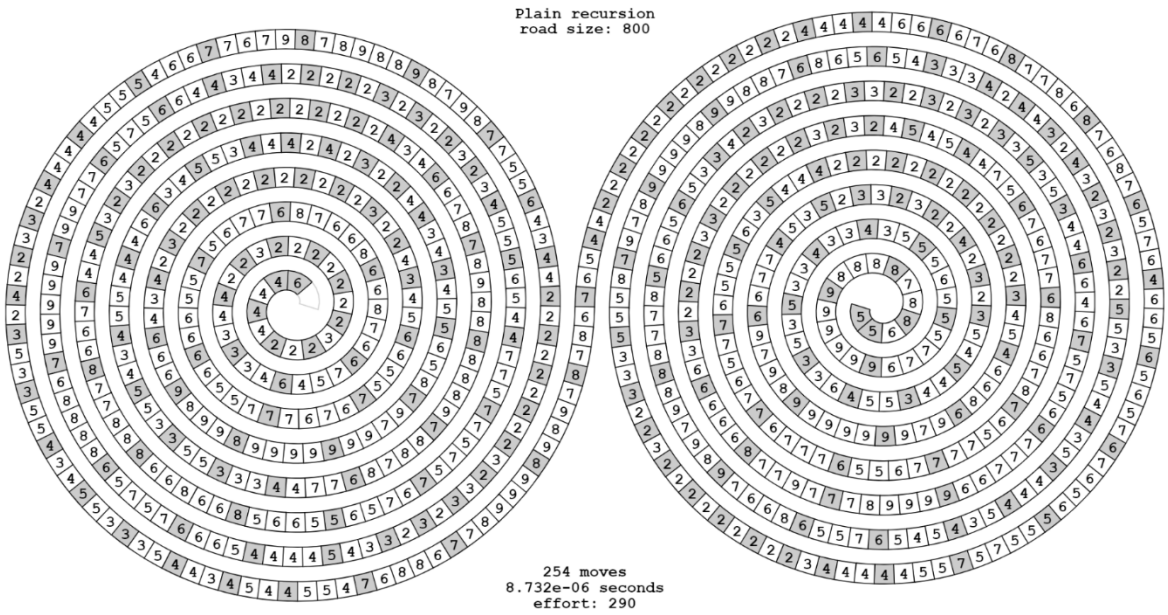


**Fig.10 –** Ficheiro pdf gerado pela segunda função recursiva criada

## 2.4. Solution_3_Iterative

Antes de criar qualquer solução iterativa, já sabíamos que em norma, as funções iterativas são mais rápido que as recursivas. Isto porque nas funções iterativas, não precisamos de guardar imediatamente um resultado na stack. Significa, deste modo, menos instruções e assim, menos ciclos de CPU.

A nossa função iterativa tem por base a distância de travagem, ou seja, este algoritmo, para cada velocidade possível e começando sempre pela velocidade maior, irá calcular a distância de travagem, sendo esta o somatório das velocidades, começando na maior velocidade e ir decrementando uma unidade até esta ser igual a um. Calculada a distância de travagem (a distância que este precisa de percorrer para conseguir parar), é verifica-se se se consegue percorrer a distância de travagem sem exceder nenhum limite de velocidade. Caso exceda algum, irá verificar para outra velocidade, fazendo o mesmo algoritmo para esta. Caso não exceda, avança para a próxima posição com novos valores.

```
207  static void solution_3_iterative(int move_number, int position, int speed, int final_position)
208  {
209    int i, new_speed,j;
210    int new_pos;
211    while (position < final_position)
212    {
213      for (new_speed = speed + 1; new_speed >= 1; new_speed--)
214      {
215        int break_distance = new_speed * (new_speed + 1) / 2;
216        if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + break_distance <= final_position)
217        {
218          // check break distance
219          new_pos = position;
220          for (i = new_speed ; i >= 1 ; i--) {// de velocidade maxima até 1
221            for (j = 0; j <= i && i <= max_road_speed[new_pos + j]; j++);
222            if (j <= i) {
223              break; // nao respeitou um limite i <= max_road_speed[new_pos + j]
224            }
225            new_pos = new_pos + i;
226          }
227
228          if (i == 0) // se a velocidade for válida
229          {
230            position += new_speed;
231            speed = new_speed;
232            solution_3_count++;
233            solution_3.positions[move_number++] = position;
234            solution_3.n_moves = move_number;
235            break;
236          }
237        }
238      }
239    }
240    solution_3_best = solution_3;
241  }
```

**Fig.11** – Código da primeira solução iterativa criada

O algoritmo que desenvolvemos tem por base um ciclo while que irá ser repetido enquanto a posição atual for menor do que a posição final. Dentro deste ciclo utiliza-se a possibilidade de a velocidade aumentar uma unidade como primeira tentativa para o algoritmo ser mais rápido e, com esta velocidade, é calculada a distância de travagem. É verificado se a nova velocidade é maior ou igual do que um, se esta também é menor do que a velocidade máxima possível, ou seja, nove e se a soma desta com a distância de travagem é menor ou igual do que a posição final, se esta condição se verificar avança para a próxima instrução, caso não se verifique testa a mesma condição para o próximo caso (a velocidade manter-se). Se a condição anterior se verificar, avança para os dois ciclos for's presentes nas linhas 220 e 221 em que é avaliado para cada velocidade se é excedido algum limite no número de casas percorridas. Se todos os limites se cumprirem é guardado a posição e a velocidade e executado novamente o ciclo while com os novos valores.

Ao comparamos esta função com todas as outras já apresentadas, observamos que esta é a melhor até à data, dado principalmente à diferença da quantidade de instruções entre funções recursivas e iterativas.

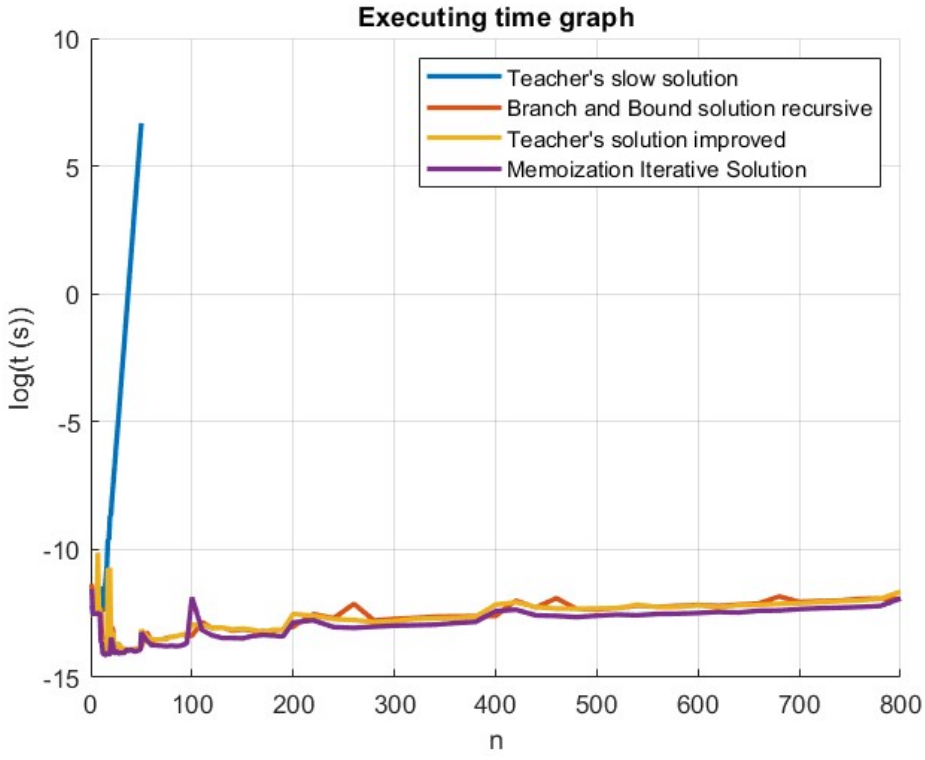


**Fig.12 –** Gráficos do tempo de execução das soluções recursivas mais a solução iterativa

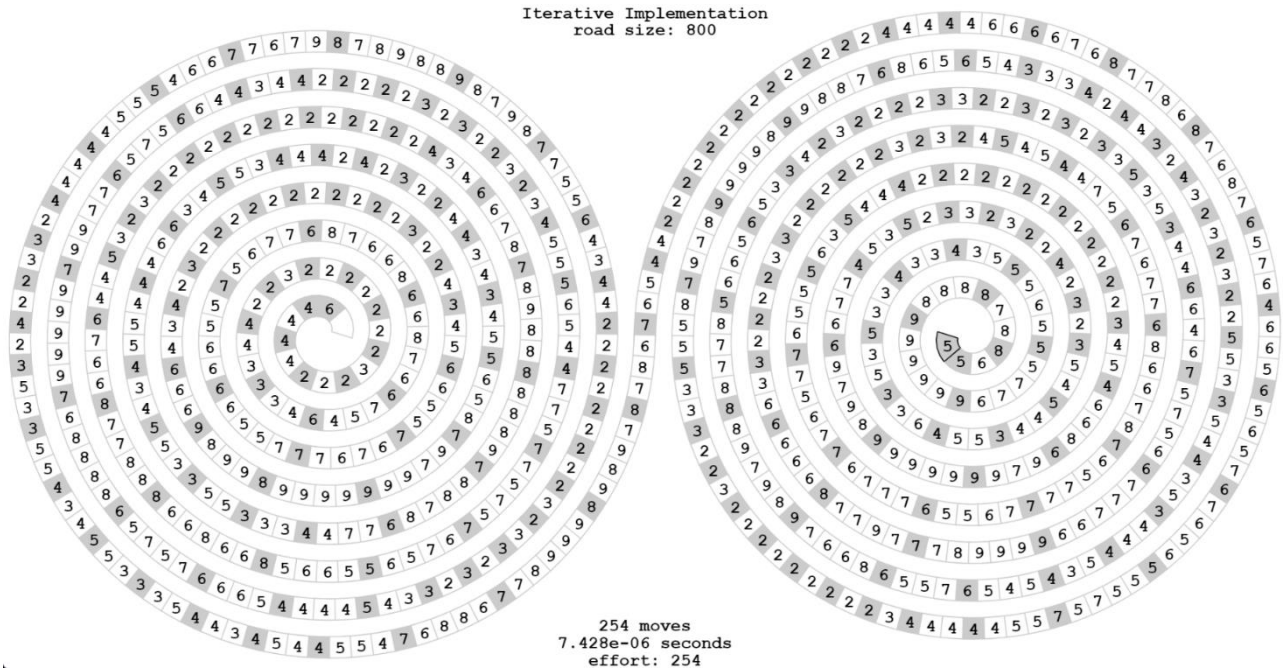


**Fig.13 –** Ficheiro pdf gerado pela primeira solução iterativa criada

## 2.5. Solution_4_Iterative (Programação Dinâmica)

Na última solução desenvolvida, procurámos incluir efetivamente um método de programação dinâmica para levar mais além os tempos de execução e otimização do programa. Com a programação dinâmica, tende-se a guardar soluções de problemas para uso futuro.

Para esta função iterativa, a linha de pensamento é muito parecida à função anterior "solution_3_iterative". Usámos novamente um ciclo while que itera enquanto a posição for menor do que a posição final. Depois, são testadas as velocidades possíveis na linha 272 através de um ciclo for e calcula-se a distância que o carro necessita para travar logo depois. De seguida, conclui-se que as soluções nas quais o carro para só depois da posição final, não são válidas e por isso, é forçada uma redução através da variável "reducing", que vai ter o valor 1. Após isto, a partir da linha 285, se a velocidade for válida, verifica-se a distância de travagem ao percorrer as velocidades desde a velocidade máxima até 1 e para cada velocidade, se esta respeita o limite de imposto. Se a distância de travagem das velocidades não for correta, o carro não poderá andar com a sua velocidade atual.

```
265  static void solution_4_iterative(int move_number, int position, int speed, int final_position)
266  {
267    int i, new_speed,j;
268    int new_pos;
269    int reducing = 0;
270    while (position < final_position)
271    {
272      for (new_speed = speed + 1; new_speed >= 1; new_speed--)
273      {
274        int break_distance = new_speed * (new_speed + 1) / 2; // Calculates the distance that the car will travel until it needs to break
275
276        if (position + break_distance > final_position) { // If the car is going to break after the final position, it will not be a valid solution
277          if (reducing == 0) {
278            last_move_number = move_number;
279            last_speed = speed;
280            last_position = position;
281          }
282          reducing = 1;
283          continue;
284        }
285        if (new_speed >= 1 && new_speed <= _max_road_speed_)
286        {
287          // check break distance
288          new_pos = position;
289          for (i = new_speed ; i >= 1 ; i--) {// de velocidade maxima até 1
290            for (j = 0; j <= i && i <= max_road_speed[new_pos + j]; j++);
291            if (j <= i) {
292              break; // nao respeitou um limite i <= max_road_speed[new_pos + j]
293            }
294            new_pos = new_pos + i;
295          }
296
297          if (i == 0) // se a velocidade for válida
298          {
299            position += new_speed;
300            speed = new_speed;
301            solution_4_count++;
302            solution_4.positions[move_number++] = position;
303            solution_4.n_moves = move_number;
304            break;
305          }
306        }
307      }
308    }
309    solution_4_best = solution_4;
310  }
```

**Fig.14** – Código da segunda função iterativa criada

Mas é no método estático “solve_4”, que mais se difere. Este método usa o resultado da melhor solução da solution_3_iterarive e calcula onde é que é necessário começar a reduzir a velocidade. Nota-se que enquanto a posição final for menor do que 1 ou maior que o tamanho máximo da estrada, não é contabilizada.

No anexo observa-se que ao chamar a função “solution_4_iterative”, em vez de se inicializar as variáveis com zero de novo, a função irá ser chamada já com o “last_move_number”, “last_position” e “last_speed”. É desta forma que se reaproveita o código previamente calculado e o tempo de execução diminui.

Assim, numa última comparação, chegamos à conclusão que juntando funções iterativas com programação dinâmica, a diferença entre todos os outros métodos usados é abismal.

```
312  //  solve_4 uses the result of the best solution found and it calculates were does it need to start reducing its speed
313  static void solve_4(int final_position)
314  {
315    if (final_position < 1 || final_position > _max_road_size_) // enquanto a posição final for menor que 1 ou maior que o tamanho maximo da estrada
316    {
317      fprintf(stderr, "solve_4: bad final_position\n");
318      exit(1);
319    }
320
321    // If the last solution was not the best solution, it will start from the beginning
322    solution_4_elapsed_time = cpu_time();
323    solution_4_count = 0ul; // effort dispended solving the problem
324    solution_4_best.n_moves = final_position; // best solution found
325    solution_4_iterative(last_move_number, last_position, last_speed, final_position); // calculates the best solution
326    solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time; // time it took to solve the problem
327  }
```

**Fig.15** – Código da função solve_4

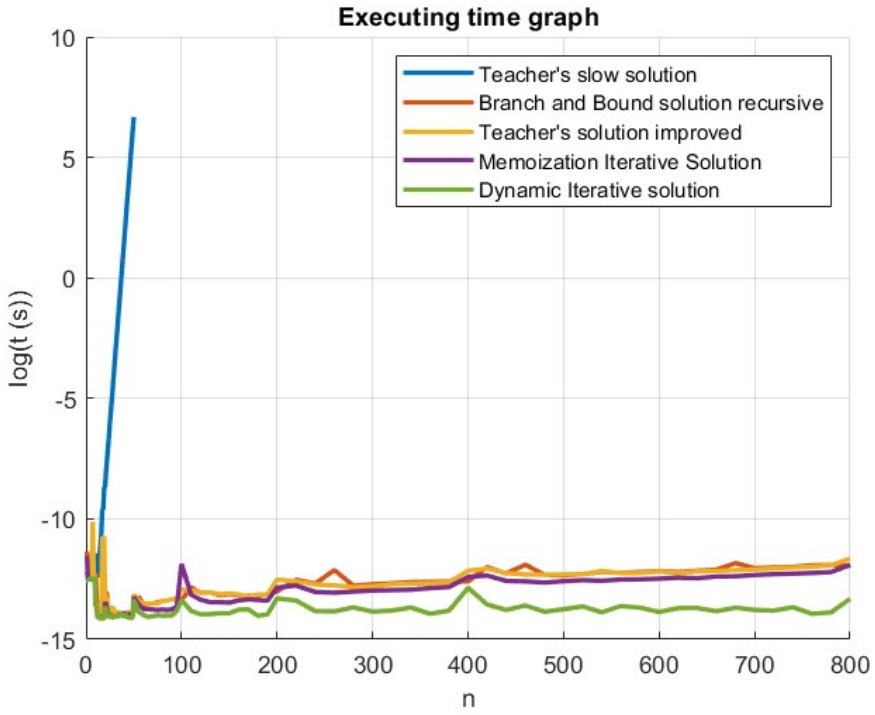


**Fig.16 –** Gráficos do tempo de execução de todas as funções criadas

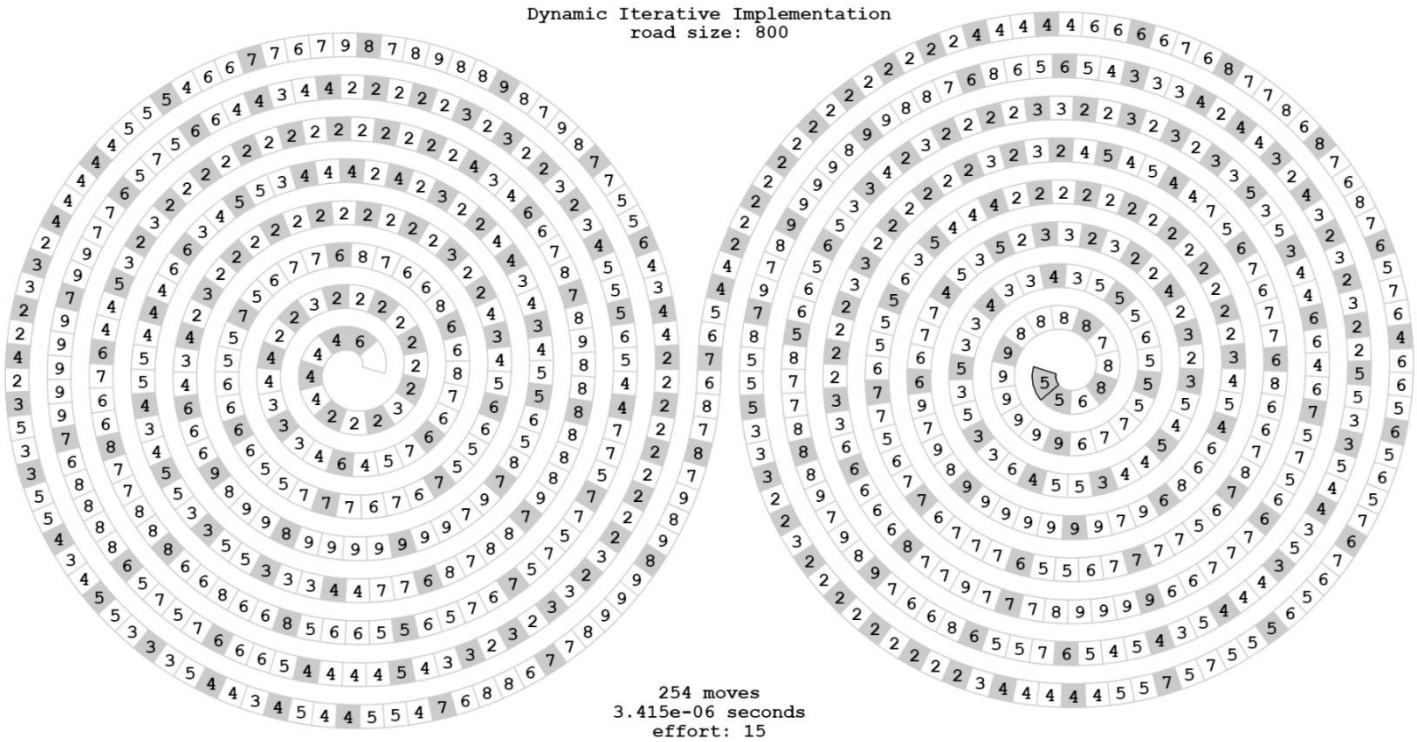


**Fig.17–** Ficheiro pdf gerado pela solução iterativa de programação dinâmica criada

# *3.Código*

## Código em 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. 108712  Name: Diogo Falcão
//   N.Mec. 108840  Name: José Gameiro

//
// 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;

//
// 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 int visited[_max_road_size_][_max_road_size_][_max_road_speed_]; 3 dimension array to store the visited positions and to use and backtracking approach to solve the problem
// But this approach wasn't good because it took more time then the branch and bound approach

static void solution_1_recursion(int move_number, int position, int speed, int final_position)
// position -> posição no momento
// speed -> velocidade no momento
// final_position -> posição final
{
  int i, new_speed;

  // record move -> esforço que faz
  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; // Funciona como um break na função recursiva
  }
  // branch-and-bound
  if (solution_1_best.positions[move_number] > solution_1.positions[move_number])
    return;
  // backtrack
  // if (visited[move_number][position][speed] == 1)
  //   return;

  // visited[move_number][position][speed] = 1;

  // no, try all legal speed
  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 solve_1(int final_position)
{
  // Clear the visited array
  // for (int i = 0; i < final_position; i++)
  //   for (int j = 0 ; j < final_position; j++)
  //     for (int k = 0; k < _max_road_speed_; k++)
  //       visited[i][j][k] = 0;


  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;
  solution_1_recursion(0, 0, 0, final_position);
  solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;
}

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

//------------------------------------------------------------------------------

static void solution_2_recursion(int move_number, int position, int speed, int final_position)
// position -> posição no momento
// speed -> velocidade no momento
// final_position -> posição final
{
  int i, new_speed;
  // already found one solution?
  if (solution_2_best.n_moves != final_position)
  {
    // return because its the best solution
    return;
  }
  // record move -> esforço que faz
  solution_2_count++;
  solution_2.positions[move_number] = position;
  // is it a solution?
  if (position == final_position && speed == 1)
  {
    // is it a better solution?
    if (move_number < solution_2_best.n_moves)
    {
      solution_2_best = solution_2;
      solution_2_best.n_moves = move_number;
    }
    return; // Funciona como um break na função recursiva
  }

  // no, try all legal speed
  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_2_recursion(move_number + 1, position + new_speed, new_speed, final_position);
      }
    }
  }
}

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_best.n_moves = final_position;
  solution_2_recursion(0, 0, 0, final_position);
  solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}

static solution_t solution_3, 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_iterative(int move_number, int position, int speed, int final_position)
{
  int i, new_speed,j;
  int new_pos;
  while (position < final_position)
  {
```

```
    {
      for (new_speed = speed + 1; new_speed >= 1; new_speed--)
      {
        int break_distance = new_speed * (new_speed + 1) / 2;
        if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + break_distance <= final_position)
        {
          // check break distance
          new_pos = position;
          for (i = new_speed ; i >= 1 ; i--) {// de velocidade maxima até 1
            for (j = 0; j <= i && i <= max_road_speed[new_pos + j]; j++);
            if (j <= i) {
              break; // nao respeitou um limite i <= max_road_speed[new_pos + j]
            }
            new_pos = new_pos + i;
          }

          if (i == 0) // se a velocidade for válida
          {
            position += new_speed;
            speed = new_speed;
            solution_3_count++;
            solution_3.positions[move_number++] = position;
            solution_3.n_moves = move_number;
            break;
          }
        }
      }
    }
    solution_3_best = solution_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;
    solution_3_iterative(0, 0, 0, final_position);
    solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}

static solution_t solution_4, solution_4_best;
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 last_move_number = 0;
static int last_position = 0;
static int last_speed = 0;

// Dynamic programming solution using a iterative solution, meaning that it uses the result of the best solution found and it calculates were does it need to start reducing its speed
static void solution_4_iterative(int move_number, int position, int speed, int final_position)
{
    int i, new_speed,j;
    int new_pos;
    int reducing = 0;
    while (position < final_position)
    {
      for (new_speed = speed + 1; new_speed >= 1; new_speed--)
      {
        int break_distance = new_speed * (new_speed + 1) / 2; // Calculates the distance that the car will travel until it needs to break

        if (position + break_distance > final_position) { // If the car is going to break after the final position, it will not be a valid solution
          if (reducing == 0) {
            last_move_number = move_number;
            last_speed = speed;
            last_position = position;
          }
          reducing = 1; // The car is going to break after the final position, so it needs to start reducing its speed
          continue; // Goes to the next speed
        }
        if (new_speed >= 1 && new_speed <= _max_road_speed_) // se a velocidade é válida
        {
          // check break distance
          new_pos = position; // new_pos is the position that the car will be after it travels the break distance
          for (i = new_speed ; i >= 1 ; i--) {// de velocidade maxima até 1
            for (j = 0; j <= i && i <= max_road_speed[new_pos + j]; j++); // checks if the car can travel at speed i
            if (j <= i) {
              break; // nao respeitou um limite i <= max_road_speed[new_pos + j]
            }
            new_pos = new_pos + i;
          }

          if (i == 0) // se a velocidade for válida
          {
            position += new_speed;
            speed = new_speed;
            solution_4_count++;
            solution_4.positions[move_number++] = position;
            solution_4.n_moves = move_number;
            break;
          }
        }
      }
    }
    solution_4_best = solution_4;
}
```

```c
//  solve_4 uses the result of the best solution found and it calculates were does it need to start reducing its speed
static void solve_4(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_) // enquanto a posição final for menor que 1 ou maior que o tamanho maximo da estrada
  {
    fprintf(stderr, "solve_4: bad final_position\n");
    exit(1);
  }

  // If the last solution was not the best solution, it will start from the beginning
  solution_4_elapsed_time = cpu_time();
  solution_4_count = 0ul; // effort dispended solving the problem
  solution_4_best.n_moves = final_position; // best solution found
  solution_4_iterative(last_move_number, last_position, last_speed, final_position); // calculates the best solution
  solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time; // time it took to solve the problem
}

//---------------------------------------------------------------------

//
// 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                  // 1 hora de execução
  int n_mec, final_position, print_this_one; // print_this_one -> imprime o pdf // n_mec -> numero de casas
  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("    |               Branch and Bound |     |                plain recursion |     |                plain iterative |     |              dynamic iterative |\n");
  printf("--- + --- ---------------- --------- + --- + --- ---------------- --------- + --- + --- ---------------- --------- + --- + --- ---------------- --------- +\n");
  printf("  n | sol            count  cpu time |   n | sol            count  cpu time |   n | sol            count  cpu time |   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);
    // first solution method (branch and bound)
    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("                                |");
    }
    // second solution method (return first solution)
    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 (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("                                |");
    }
    // third solution method (iterative with break distance)
    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 (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.n_moves, &solution_3_best.positions[0], solution_3_elapsed_time, solution_3_count, "Plain recursion");
      }
      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("                                |");
    }
    // fourth solution method (iterative with break distance)
    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 (solution_4_elapsed_time < _time_limit_)
    {
      solve_4(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_4.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_4_best.n_moves, &solution_4_best.positions[0], solution_4_elapsed_time, solution_4_count, "Plain recursion");
      }
      printf(" %3d %16lu %9.3e | ", solution_4_best.n_moves, solution_4_count, solution_4_elapsed_time);
    }
    else
    {
      solution_4_best.n_moves = -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 em Matlab

```
clear
clc

exec_temp_1 = load('time_execution_1.txt');
exec_temp_2 = load('time_execution_2.txt');
exec_temp_3 = load('time_execution_3.txt');
exec_temp_4 = load('time_execution_4.txt');
exec_temp_5 = load('time_execution_5.txt');
n = load('n.txt');
n_long = zeros(1,length(exec_temp_1));

for i = 1:length(exec_temp_1)
    n_long(i) = n(i);
end

figure(1)
plot(n_long,log(exec_temp_1),"LineWidth",2,"Color",[0 0.4470 0.7410])
title("Execution time graph")
legend("Teacher's slow soolution")
xlabel("n")
ylabel("log(t (s))")
grid on

figure(2)
hold on
plot(n_long,log(exec_temp_1),"LineWidth",2,"Color",[0 0.4470 0.7410])
plot(n,log(exec_temp_2),"LineWidth",2,"Color",[0.8500 0.3250 0.0980])
title("Execution time graph")
legend("Teacher's slow solution","Branch and Bound solution recursive")
xlabel("n")
ylabel("log(t (s))")
grid on
hold off

figure(3)
hold on
plot(n_long,log(exec_temp_1),"LineWidth",2,"Color",[0 0.4470 0.7410])
plot(n,log(exec_temp_2),"LineWidth",2,"Color",[0.8500 0.3250 0.0980])
plot(n,log(exec_temp_3),"LineWidth",2,"Color",[0.9290 0.6940 0.1250])
title("Execution time graph")
legend("Teacher's slow solution","Branch and Bound solution recursive","Teacher's solution improved")
xlabel("n")
ylabel("log(t (s))")
grid on
hold off
```

```
46
47  figure(4)
48  hold on
49  plot(n_long,log(exec_temp_1),"LineWidth",2,"Color",[0 0.4470 0.7410])
50  plot(n,log(exec_temp_2),"LineWidth",2,"Color",[0.8500 0.3250 0.0980])
51  plot(n,log(exec_temp_3),"LineWidth",2,"Color",[0.9290 0.6940 0.1250])
52  plot(n,log(exec_temp_4),"LineWidth",2,"Color",[0.4940 0.1840 0.5560])
53  title("Executing time graph")
54  legend("Teacher's slow solution","Branch and Bound solution recursive","Teacher's solution improved","Memoization Iterative Solution")
55  xlabel("n")
56  ylabel("log(t (s))")
57  grid on
58
59  figure(5)
60  hold on
61  plot(n_long,log(exec_temp_1),"LineWidth",2,"Color",[0 0.4470 0.7410])
62  plot(n,log(exec_temp_2),"LineWidth",2,"Color",[0.8500 0.3250 0.0980])
63  plot(n,log(exec_temp_3),"LineWidth",2,"Color",[0.9290 0.6940 0.1250])
64  plot(n,log(exec_temp_4),"LineWidth",2,"Color",[0.4940 0.1840 0.5560])
65  plot(n,log(exec_temp_5),"LineWidth",2,"Color",[0.4660 0.6740 0.1880])
66  legend("Teacher's slow solution","Branch and Bound solution recursive","Teacher's solution improved","Memoization Iterative Solution" ...
67                                                                                                      ,"Dynamic Iterative solution")
68  grid on
69  xlabel("n")
70  ylabel("log(t (s))")
71  title("Executing time graph")
72  hold off
```

# *4. Conclusão*

Com a realização deste trabalho concluímos que existem várias maneiras de abordar o problema proposto. Estas soluções utilizam métodos recursivos e iterativos. Constatamos que os métodos iterativos são mais rápidos e eficientes que os recursivos através da análise de gráficos de execução e dos ficheiros gerados pelo programa.

Observámos que, na criação de uma solução usando programação dinâmica, esta foi a melhor solução até à data visto que o gráfico do seu tempo de execução e o esforço obtido foram os menores.

# *5. Webgrafia*

GEEKS (2022), Branch and Bound Algorithm, [visitado em 5 de dezembro de 2022]:
https://www.geeksforgeeks.org/branch-and-bound-algorithm/

GEEKS (2022), Back Traking Algortithm, [visitado em 5 de dezembro de 2022]:
https://www.geeksforgeeks.org/backtracking-algorithms/

GATWIRI (2022), Dynamic Programming In Javascript using Tabulation, [visitado em 6 de dezembro de 2022]:
https://www.section.io/engineering-education/dynamic-programming-in-javascript-using-tabulation/

GEEKS (2022), What is memoization? A Complete tutorial, [visitado em 5 de dezembro de 2022]:
https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

SKIENA (2008), Backtracking, [visitado em 3 de dezembro de 2022]:
https://guides.codepath.com/compsci/Backtracking

HUANG (2021), Is Recursion Really Slower than Iteration?, [visitado em 22 de novembro de 2022]:
https://edward-huang.com/2021/02/17/is-recursion-really-slower-than-iteration/