

# Word Ladder

**Relatório 08/12/2022**

## Tiago Sousa Fonseca (107266) – 37.5%

**Tomás Sousa Fonseca (107245) – 37.5%**

## Beatriz Ferreira (107214) – 25%

**Universidade de Aveiro Algoritmos e Estruturas de Dados**

**Índice**

[Introdução 3](#_TOC_250005)

Funções Modificadas 4

hash\_table\_create [4](#_TOC_250003)

hash\_table\_grow 5

hash\_table\_free 6

find\_word 6

find\_representative 6

add\_edge 6

breadh\_first\_search 6

list\_connected\_component 6

path\_finder 6

connected\_component\_diameter 6

graph\_info 6

Resultados 7

Solução Dada Inicialmente. 7

[Solução Desenvolvida 10](#_TOC_250002)

[Comentários 13](#_TOC_250001)

[Apêndice 14](#_TOC_250000)

# Introdução

Uma “word ladder” é uma sequência de palavras em que duas palavras adjacentes diferem por uma letra. Por exemplo:

* Inglês:

head —> heal —> teal —> tell —> tall —> tail

* Português:

tudo —> todo —> nodo —> nado —> nada

Seguindo o princípio estabelecido para a diferenciação de duas palavras, podemos assim estabelecer um componente conectado, ao qual uma palavra pertence.

A partir de vários componentes conectados, é possível estabelecer um caminho entre duas palavras.

# Funções Modificadas

## hash\_table\_create:

## Este código é uma função em C chamada "hash\_table\_create" que cria e inicializa uma estrutura de dados de tabela hash. A função começa por alocar memória para a tabela hash usando o malloc e atribui o endereço de memória a um ponteiro chamado "hash\_table". Se a alocação de memória falhar, a função irá imprimir uma mensagem de erro "create\_hash\_table: out of memory" para o stream de erro padrão e sair do programa com o status 1. Em seguida, a função define o tamanho da tabela hash para 50, o número de entradas e arestas para 0. Em seguida, a função usa o malloc novamente para alocar memória para um array de ponteiros chamado "heads" com o tamanho da tabela hash. Se a alocação de memória falhar novamente, a função irá imprimir a mesma mensagem de erro e sair do programa. Por fim, ele usa um loop for para inicializar todos os ponteiros no array com NULL, e depois a função retorna a tabela hash criada.

## hash\_table\_grow:

## Este código é uma função em C chamada "hash\_table\_grow" que aumenta o tamanho da tabela hash. A função começa criando um novo array de ponteiros para as cabeças dos nós da tabela hash com o tamanho da tabela atual multiplicado por dois. Se não for possível alocar memória para o novo array, a função imprimirá uma mensagem de erro "create\_hash\_table: out of memory" para o fluxo de erro padrão e sairá do programa com o status 1. Em seguida, o loop for inicializa todos os ponteiros no novo array com NULL. Depois, outro loop for percorre cada nó na tabela antiga e usa uma função de hash (crc32) para calcular o novo índice para cada nó e adicioná-lo ao novo array. Por fim, a função libera a memória do array antigo, atualiza a tabela hash com o novo array e atualiza o tamanho da tabela.

## hash\_table\_free:

## Este código é uma função em C chamada "hash\_table\_free" que tem como objetivo libertar a memória alocada para uma tabela hash. A função usa um loop for para percorrer cada ponteiro no array de cabeças da tabela hash e liberta a memória de cada nó da tabela usando a função free\_hash\_table\_node. Depois, a função usa a função free para libertar a memória do array de cabeças. A função tem como objetivo garantir que toda a memória alocada para a tabela hash seja libertada, evitando vazamentos de memória e garantindo que a memória possa ser reutilizada pelo sistema.

## find\_word:

## Este código é uma função em C chamada "find\_word" que encontra um nó específico em uma tabela hash. A função usa uma função de hash (crc32) para calcular o índice do nó com base na palavra dada. Em seguida, usa um loop para percorrer a lista encadeada no índice calculado e verifica se a palavra no nó atual corresponde à palavra dada. Se encontrar o nó, a função o retorna. Se o parâmetro "insert\_if\_not\_found" for definido como 1 e o nó não for encontrado, a função aloca e inicializa um novo nó com a palavra dada e o insere na tabela hash. A função também verifica se a tabela está quase cheia e, se estiver, chama a função "hash\_table\_grow" para aumentar o tamanho da tabela. Se o nó não for encontrado e o parâmetro "insert\_if\_not\_found" não for definido como 1, a função retorna NULL.

## find\_representative:

Este código é uma função em C chamada "find\_representative" que encontra o representante de um nó em uma tabela hash. O representante é o nó de cima na estrutura de conjunto de cada grupo de palavras relacionadas. A função usa dois loops while para percorrer a estrutura de conjunto até chegar ao representante. O primeiro loop começa com o nó dado e segue o ponteiro "representative" até encontrar o representante. O segundo loop começa com o nó dado e segue o ponteiro "representative" até o representante, alterando cada nó visitado para apontar diretamente para o representante. Isso é conhecido como compressão de caminho e é usado para acelerar a busca de representante em futuras chamadas. A função retorna o representante encontrado.

## add\_edge:

Esta função, chamada "add\_edge", é usada para adicionar uma aresta entre dois nós na tabela hash. O nó "from" e o nó "word" são passados como argumentos, com a tabela hash também sendo passada. A função usa primeiro a função "find\_word" para localizar o nó "to" na tabela hash que corresponde à palavra passada. Se este nó não for encontrado, a função sai. Caso contrário, o número de arestas na tabela hash é incrementado e um nó de adjacência é alocado para ligar os nós "from" e "to". A função, então, realiza uma operação de conjunto-união nos nós representativos do "from" e "to" para combiná-los se eles não estiverem já no mesmo conjunto. A função termina retornando.

## breadh\_first\_search:

Esta função, chamada "breadth\_first\_search", é um algoritmo de busca em largura usado para encontrar o caminho mais curto entre dois nós em um grafo representado por uma tabela hash. A função leva como argumentos o número máximo de vértices, uma lista de todos os vértices, o nó de origem e o nó de destino. A função usa uma fila para controlar os nós a serem visitados e começa inicializando o nó de origem como o primeiro elemento da fila, marcando-o como visitado e definindo o nó anterior como nulo. A função então itera através da fila até que ela esteja vazia, retirando o nó da frente, verificando se é o nó de destino, se for, retorna o número de vértices visitados. Se o nó de destino não for encontrado, a função itera através dos vizinhos do nó retirado e os adiciona à fila se eles ainda não foram visitados. A função termina retornando o número de vértices visitados e redefinindo o status visitado de todos os vértices visitados na busca para 0.

## 

## list\_connected\_component:

Esta função, chamada "list\_connected\_component", é usada para listar todos os nós que estão conectados a um nó específico em uma tabela hash. A função leva como argumentos a tabela hash e uma palavra, e usa a função "find\_word" para localizar o nó específico na tabela hash. Se o nó não for encontrado, a função sai. Caso contrário, a função usa a função "find\_representative" para encontrar o nó representativo do componente conectado ao qual o nó específico pertence. Em seguida, usa a função "breadth\_first\_search" para realizar uma travessia em largura do componente conectado e armazena os nós visitados em uma lista. A função, então, imprime as palavras dos nós na lista e libera a lista de vértices.

## path\_finder:

Esta função, chamada "path\_finder", é usada para encontrar e imprimir o caminho mais curto entre duas palavras em uma tabela hash. A função recebe como argumentos a tabela hash e as duas palavras, "from\_word" e "to\_word". Em primeiro lugar, a função utiliza a função "find\_word" para localizar os nós correspondentes na hash\_table para as palavras "from\_word" e "to\_word" e os atribui às variáveis "from" e "to". Se uma das palavras não for encontrada na hash\_table, a função sai e imprime uma mensagem de erro. Se ambas as palavras forem encontradas, a função utiliza a função "breadth\_first\_search" para encontrar o caminho mais curto entre os nós "from" e "to" e armazena os nós visitados em uma lista. Em seguida, a função itera através dos nós na lista, começando pelo nó "to" e imprime as palavras de cada nó no caminho, com uma seta "->" entre cada palavra e uma nova linha no final.

## Connected\_component\_diameter:

A função, chamada "connected\_component\_diameter", é usada para encontrar o caminho mais longo dentro de um componente conectado de um grafo representado em uma tabela hash. A função leva como argumento um nó da tabela hash. A função primeiro encontra o nó representativo do componente conectado ao qual o nó de entrada pertence. Em seguida, aloca uma lista de vértices e usa a função "breadth\_first\_search" para encontrar todos os nós no componente conectado. A função, então, itera através dos nós na lista e usa a função "breadth\_first\_search" novamente para encontrar o caminho mais longo entre cada nó e todos os outros nós no componente conectado. O caminho mais longo encontrado é armazenado como o diâmetro do componente conectado. A função, então, libera a lista de vértices alocada e retorna o diâmetro do componente conectado.

## Solução Mais Eficiente

Ao longo da determinação de uma solução para o problema dado, são encontradas várias que podem ser usadas, e que, cumprindo todas as regras, permitem-nos chegar ao destino. No entanto, há que ser feita a seleção da melhor solução, tendo em conta a eficiência (dada pelo número de movimentos do carro).

Assim, desde o início da determinação das soluções para o problema dado, e para cada um dos movimentos feitos, é verificada a existência de uma melhor solução que se encontra guardada em ***solution\_1\_best*** (derivada da estrutura ***solution\_t***).

Caso a solução atualmente a ser desenvolvida, e até ao momento da verificação, seja mais eficiente do que aquela guardada, continuamos a desenvolvê-la, de forma a explorar a existência de uma solução ainda melhor.

# Resultados

## Solução Dada Inicialmente

Com uma solução ineficiente como esta, a complexidade revelou ser um verdadeiro problema, tendo aumentado exponencialmente os tempos de execução em função do tamanho da estrada.

Assim, esta solução só permite resolver o problema para tamanhos de estrada menores ou iguais a 55. Na Figura 1 podem ser observados estes dados, assim como uma expressão matemática aproximada, que traduz o tempo de execução em função do tamanho da estrada.

9,000E+03

8,000E+03

7,000E+03

6,000E+03

CPU Time

5,000E+03

4,000E+03

3,000E+03

2,000E+03

1,000E+03

0,000E+00

### Speed Run Complexity

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  | y = 2E-08e | 0,4735x |
|  |  |  |  | R² = 0,99 | 96 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

0 10 20 30 40 50 60

Road Size (n)

Série1 Exponencial (Série1)

*Figura 1. Complexidade Speed Run*

No entanto, uma previsão do tempo de execução para uma estrada com tamanho 800 (Figura 2), revela que a solução mais eficiente levaria imenso tempo a ser encontrada, o que não é ideal para um algoritmo generalizado para todos os tamanhos de estrada.

Em suma, a reformulação do algoritmo revelou ser algo critico para a resolução do problema num tempo de execução significativamente menor.

7,000E+156

6,000E+156

5,000E+156

CPU Time

4,000E+156

3,000E+156

2,000E+156

1,000E+156

0,000E+00

### Speed Run Prediction

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | y = 2E-08 | e0,4735x |  |
|  |  |  |  |  |  | R² = 0, | 9996 |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

0 100 200 300 400 500 600 700 800 900

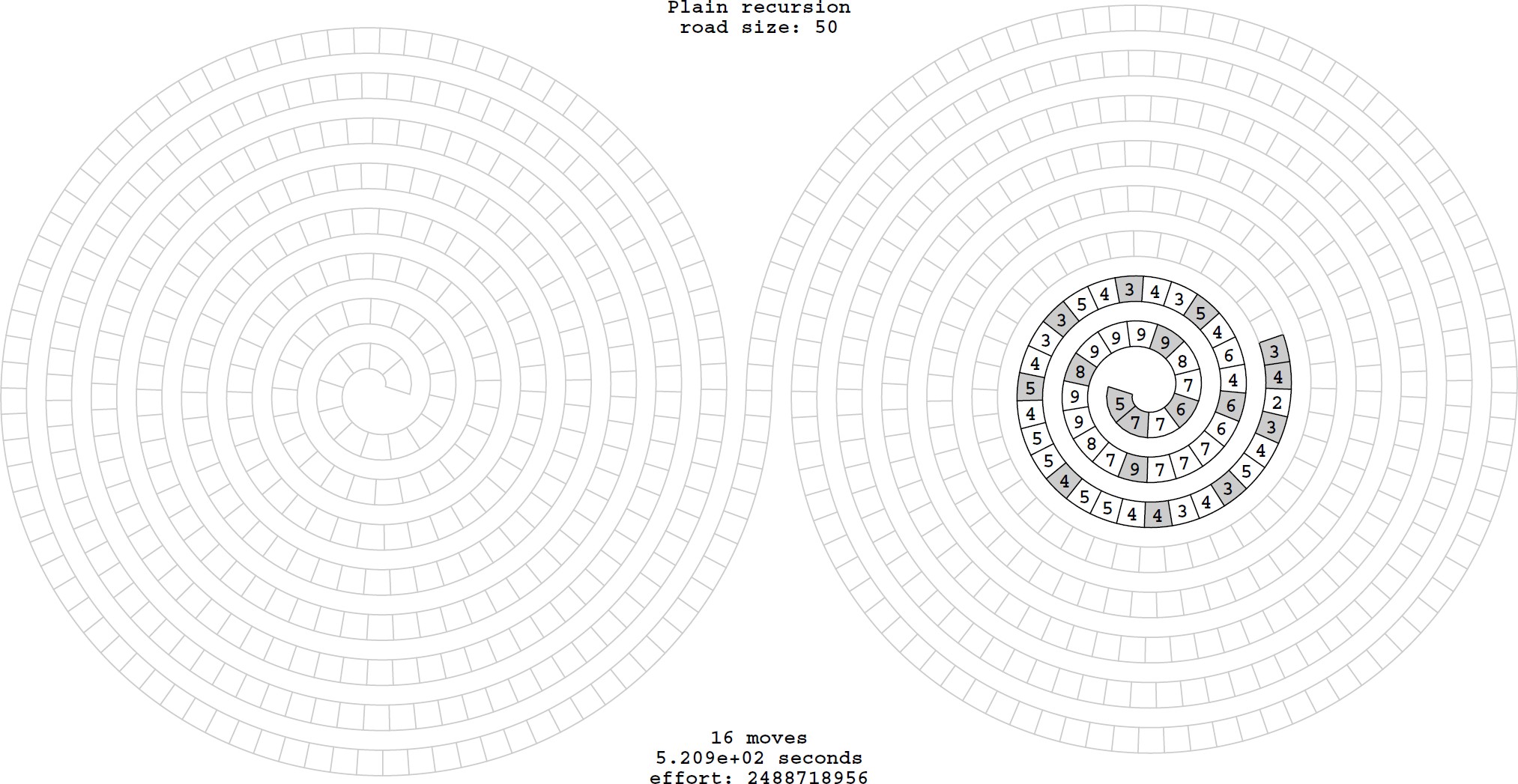
Road Size (n)

Série1 Exponencial (Série1)

*Figura 2. Speed Run Prediction*

Na Figura 3 é apresentado o resultado da solução inicialmente dada, para o número mecanográfico 107266. Como se observa, o tempo de execução para uma estrada de tamanho 50 é menor do que o tempo de execução médio para uma estrada do mesmo tamanho (sendo, no entanto, bastante alto), este tempo está diretamente associado ao esforço necessário para determinar o melhor caminho.





*Figura 3. Resultados Speed Run para o número mecanográfico 107266*

## Solução Desenvolvida

Com base na solução inicialmente dada, foi desenvolvida uma nova solução. Esta solução tem uma complexidade significativamente mais reduzida do que a fornecida inicialmente, aplicando métodos baseados na realidade (métodos mencionados no capítulo Métodos Utilizados), e que revelam ser eficazes na identificação de caminhos inválidos com uma maior antecedência.

Assim, a complexidade passou de exponencial para linear, tendo sido observados tempos de execução notoriamente menores, como também um esforço reduzido na resolução do problema em questão, o que permitiu atingir uma estrada de tamanho 800 em microssegundos (Figura 4).

8,000E-06

7,000E-06

6,000E-06

5,000E-06

CPU Time

4,000E-06

3,000E-06

2,000E-06

1,000E-06

0,000E+00

### Solution Speed Run Complexity

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | y | = 7E-09x | + 2E-07 |
|  |  |  |  |  |  |  | R² = 0,87 | 68 |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

0 100 200 300 400 500 600 700 800 900

Road Size (n)

Série1 Linear (Série1)

*Figura 4. Solution Speed Run Complexity*

Uma comparação da complexidade da solução dada inicialmente (***Speed Run***) com a complexidade da solução desenvolvida (***Solution Speed Run***), demonstra a significante redução do tempo de execução em função do tamanho da estrada (Figura 5), sendo assim demonstrada a eficácia dos métodos aplicados para resolver o problema, independentemente do tamanho da estrada.

10000

8000

CPU Time

6000

4000

2000

0

### Complexity Comparison

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  |  |
|  | y | = 2E-08e0 | ,4735x |  |  |  |  |  |  |
|  |  | R² = 0,999 | 6 |  |  |  |  |  |  |
|  |  |  |  |  |  | y | = 7E-09x | + 2E-07 |  |
|  |  |  |  |  |  |  | R² = 0,87 | 68 |  |

0 100 200 300 400 500 600 700 800 900

Road Size (n)

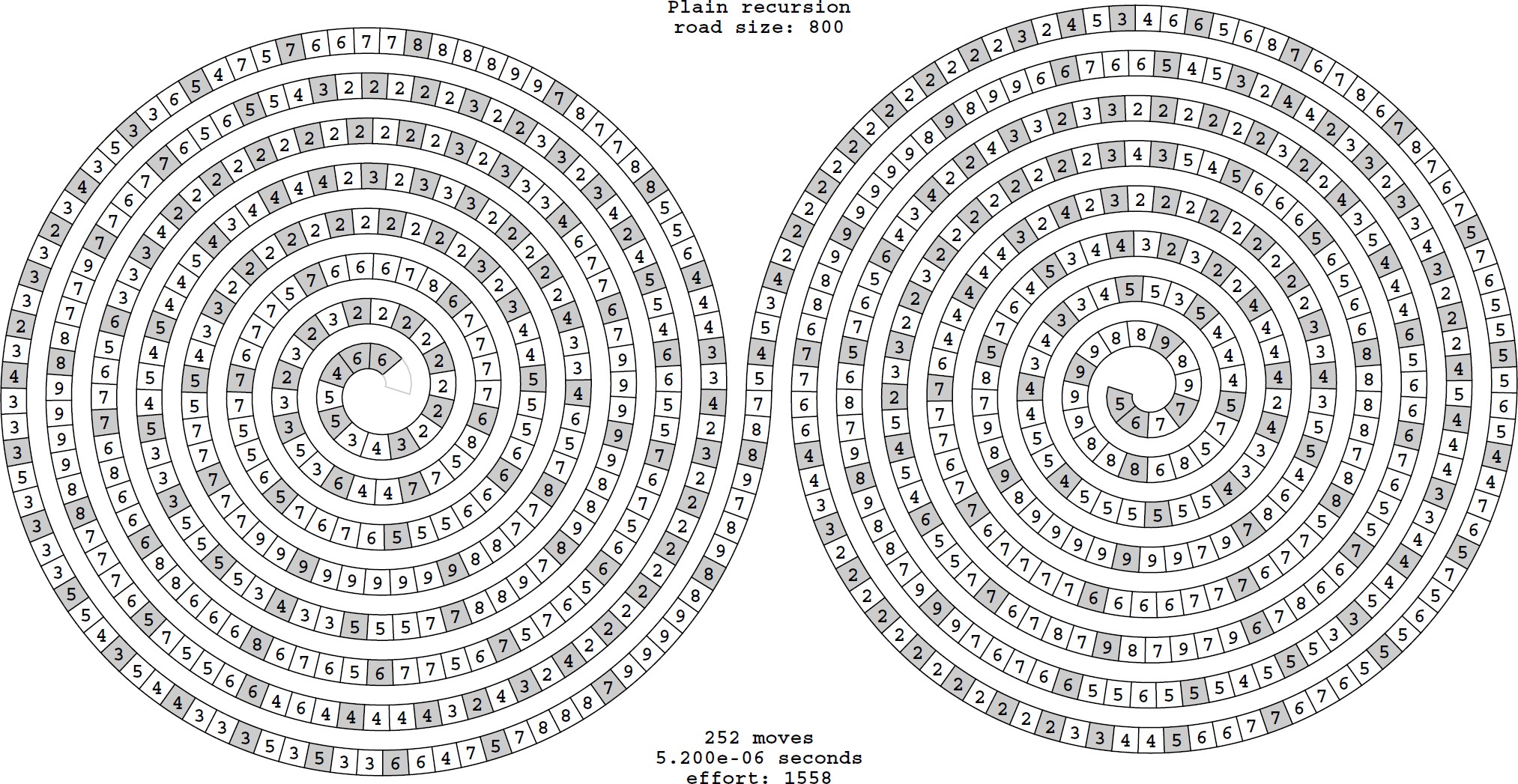
Solution Speed Run Speed Run

Linear (Solution Speed Run) Exponencial (Speed Run)

*Figura 5. Speed Run Complexity vs Solution Speed Run Complexity*

Na Figura 6 é apresentado o resultado da solução desenvolvida, para o número mecanográfico 107245. Como se observa, foi possível obter uma solução para uma estrada de tamanho 800 em microssegundos, tendo sido reduzido o esforço necessário para chegar a uma solução.





*Figura 6. Resultados Solution Speed Run para o número mecanográfico 107245*

# Comentários

Todos os dados apresentados acima são baseados na média dos tempos de execução em função do tamanho da estrada para cada um dos números mecanográficos dos autores.

O esforço apresentado na solução desenvolvida é relativamente alto para o resultado apresentado. Este fenómeno pode ser justificado pela exploração momentânea de caminhos inválidos para uma nova solução, caminhos estes que instantaneamente deixaram de ser explorados, mas que acabam por ser contabilizados.

Assim, e tendo em conta a rápida desistência por parte do algoritmo em prosseguir a exploração destes caminhos erráticos, este esforço não traduz de forma direta a exploração feita, podendo antes ser visto como um número que representa o caminho explorado combinado com desvios breves (é expectável que o verdadeiro esforço seja de 518 para uma estrada de tamanho 800).

Os dados e resultados de cada um dos autores, assim como a sua exaustiva análise, podem ser encontrados aqui.

Todo o código usado pode ser encontrado aqui.

# Apêndice

## solution\_speed\_run.c

//

// AED, December 2022 (107266, 107245, 107214)

//

// 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. 107266 Name: Tiago Fonseca

// N.Mec. 107245 Name: Tomás Fonseca

// N.Mec. 107214 Name: Beatriz Ferreira

//

//

// 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 <stdlib.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)rand() % 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 *boolean* solution\_1\_recursion(int *move\_number*,int *position*,int *speed*,int *final\_position*) // first

arguments (0, 0, 0, final\_position)

{

int speed\_soma = 0;

solution\_1.positions[*move\_number*] = *position*;

solution\_1\_count++;

if (*speed* == 1 && *position* == *final\_position*){

if(*move\_number* < solution\_1\_best.n\_moves)

{

solution\_1\_best = solution\_1;

solution\_1\_best.n\_moves = *move\_number*;

}

return TRUE;

}

if (solution\_1\_best.positions[*move\_number*] > *position*){

return FALSE;

}

if (*position* == *final\_position*){

return FALSE;

}

if(*move\_number* > solution\_1\_best.n\_moves){

return FALSE;

}

if(*speed* > \_max\_road\_speed\_ || *speed* > max\_road\_speed[*position*] || *speed* < 0 || (*speed* < 1 && *position*

>0) ){

return FALSE;

}

int positions\_left = *final\_position* - *position*;

for(int i = *speed*; i >=1; i--){

speed\_soma += i;

if (*speed* > max\_road\_speed[*position*+i]){

return FALSE;

}

}

if(speed\_soma > positions\_left){

return FALSE;

}

solution\_1\_recursion(*move\_number* + 1, *position* + *speed*, *speed*+1, *final\_position*);

solution\_1\_recursion(*move\_number* + 1, *position* + *speed*, *speed*, *final\_position*);

solution\_1\_recursion(*move\_number* + 1, *position* + *speed*, *speed*-1, *final\_position*);

return FALSE;

}

static void solve\_1(int *final\_position*) // we pass the final position of the road as an argument

{

if(*final\_position* < 1 || *final\_position* > \_max\_road\_size\_) // verify that the final position is valid

{

fprintf(stderr,"solve\_1: bad final\_position\n");

exit(1);

}

solution\_1\_elapsed\_time = cpu\_time(); // start the timer

solution\_1\_count = 0ul; // reset the effort counter

solution\_1\_best.n\_moves = *final\_position* + 100; // set the best solution to an impossible value

solution\_1\_recursion(0,0,0,*final\_position*); // call the recursive function

(move\_number,position,speed,final\_position)

solution\_1\_elapsed\_time = cpu\_time() - solution\_1\_elapsed\_time; // stop the timer and calculate the elapsed

time

}

//

// example of the slides

//

static void example(void)

{

int i,final\_position;

srand(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\_bes

t.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]);

srand((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;

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);

// 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\_b

est.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("

|");

}

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\_

}