



universidade de aveiro

SPEED RUN

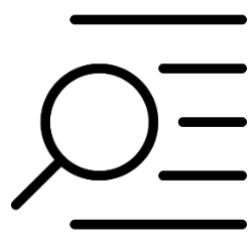
**ALGORITMOS E ESTRUTURAS
DE DADOS 2022**

Prof. Tomás Oliveira e Silva

André Oliveira	/ 107637	33%
Duarte Cruz	/ 107359	33%
Rodrigo Graça	/ 107634	33%



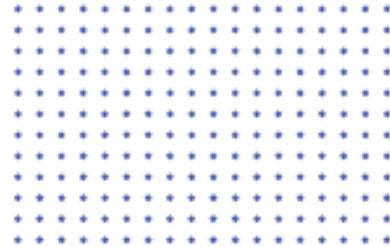
ÍNDICE



Introdução	3
Problema	3
Estimativa da solução Brute Force	4
Métodos.....	5
1 ^a Solução (solution_1_recursion) – Brute Force	5
2 ^a Solução (solution_2_recursion) – Brute Force Otimizada	7
3º Solução (solution_3_recursion)	8
4 ^a Solução (solution_4_non_recursion)	11
5º Solução (solution_5_dynamic)	12
Comparação	15
Brute Forces (soluções 1 e 2)	16
Brute Force Otimizada e recursiva desenvolvida (soluções 2 e 3)	17
Recursiva desenvolvida e não recursiva com mesma ideia (soluções 3 e 4)	18
Recursiva desenvolvida e dinâmica com a mesma base (soluções 3 e 5)	19
Conclusão	20
Apêndice.....	21
Código C	21
Código Matlab	25
Exemplo de Imagens das Soluções	39

SPEED RUN

INTRODUÇÃO



O objetivo deste trabalho prático foi implementar e estudar diferentes técnicas de resolução para o problema proposto “Speed-Run” através de uma proposta de solução apresentada previamente, que era muito ineficiente.

Problema

Este problema consistia numa estrada dividida em diversos segmentos (ruas). Cada um tem um limite de velocidade que o carro não poderá ultrapassar. A velocidade do carro é medida pelo número de segmentos que este avança, num único movimento. A cada movimento, o carro pode acelerar em 1 a sua velocidade, manter a velocidade ou diminuir em 1. Parte da posição inicial com velocidade 0 e terá de chegar à ultima com velocidade 1, obrigando-o o parar.

O objetivo do problema é conseguirmos fazer o carro chegar à posição final (800), com o menor número de movimentos e tempo de execução possível.

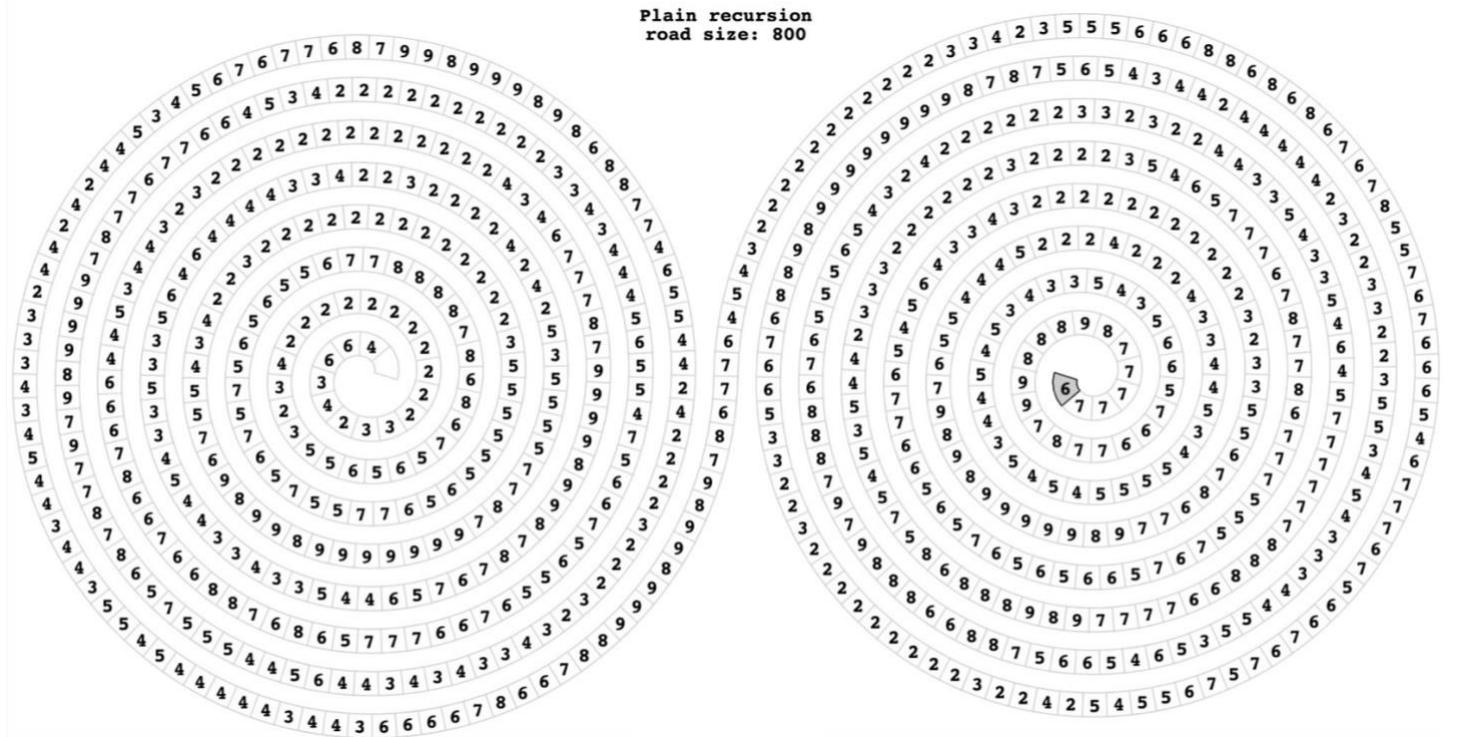


Figura 1 | Exemplo de rua com 800 segmentos

Estimativa da solução Brute Force

%% Solução 1 (PC Duarte)

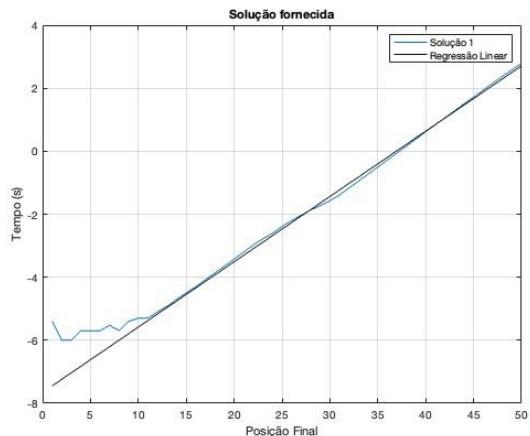
```
valores = load("resultados_1_Duarte.txt");
pos_finais = valores(:,1);
tempos = valores(:,4);

figure(1);
plot(pos_finais,log10(tempos));
ylabel("Tempo (s)");
xlabel("Posição Final");
title("Solução fornecida");
hold on;

tempos_log = log10(tempos);
N = [pos_finais(20:end) 1+0*pos_finais(20:end)];
Coefs = pinv(N)*tempos_log(20:end);
Ntotal = [pos_finais pos_finais*0+1];

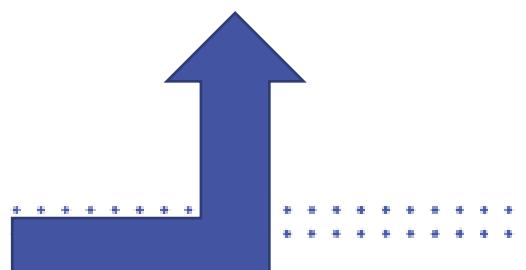
plot(pos_finais, Ntotal*Coefs, "k");
legend("Solução 1", "Regressão Linear");
grid on;
hold off;

t800_log = [800 1]*Coefs;
t800 = 10^t800_log;
fprintf('Tempo previsto até à posição 800 = %i\n',t800);
```



O intuito deste código foi colocar os resultados calculados por 1 hora da solução fornecida inicialmente num gráfico tempo(s)-posição_final.

A partir disto, conseguimos efetuar uma regressão linear do mesmo, representando-a no gráfico, e simultaneamente, conseguindo calcular uma estimativa de tempo que esta solução levaria a chegar à posição final 800.



MÉTODOS

Com a intenção de encontrarmos a solução mais eficiente e rápida para resolver este problema desenvolvemos diferentes algoritmos com diferentes complexidades computacionais exploradas.

1^a Solução (solution_1_recursion) – Brute Force

A primeira solução que está presente no nosso trabalho foi-nos apresentada no início do mesmo de forma a auxiliar-nos na percepção do objetivo do trabalho.

É uma solução recursiva, porém muito pouco eficiente, pois à medida que a posição final aumenta, o número de soluções é muito grande, sendo “impossível” chegar ao fim do problema (800 ruas). Esta solução consiste em calcular todas as sequências possíveis de velocidades para o problema. À medida que as soluções vão sendo calculadas, estas são guardadas (tanto os movimentos necessários para chegar ao fim, como as posições por onde passou), mas sempre que é encontrada uma solução com menor número de movimentos do que a guardada anteriormente, esta é substituída pela melhor. A falta de eficiência da mesma deve-se ao facto de, à medida que o número da posição final do programa aumenta, para este verificar a melhor solução possível vai ter de encontrar todas as soluções possíveis, tornando-se muito lento e praticamente “impossível” de concluir os 800 segmentos.

```
static void solve_1(int final_position)
{
    if (final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr, "solve_1: bad final_position\n");
        exit(1);
    }
    solution_1_elapsed_time = cpu_time();
    solution_1_count = 0ul;
    solution_1_best.n_moves = final_position + 100;
    solution_1_recursion(0, 0, 0, final_position);
    solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;
}
```

Inicialmente, no código que nos foi fornecido, é chamada uma função a `solve_1` onde são capturados alguns dados importantes para a realização do estudo da eficiência do algoritmo, tais como o tempo que demorou a correr cada `final_position`, o número de movimentos que levou até alcançar essa mesma posição e o esforço desenvolvido (número de vezes que a recursiva `solution_1_recursion` foi chamada).

```
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);
        }
}
```

Nesta mesma recursiva é feita a contagem do esforço do programa através de `solution_1_count` (que é incrementado cada vez que esta mesma função é iterada) e o armazenamento de uma nova posição através do número de casas que permitiu avançar.

Para testar se alcançamos o destino verificamos se a nossa posição atual corresponde à posição final imposta pela chamada da função anteriormente descrita (`solve_1`). Se o mesmo se verificar, o programa guardará essa nova solução, caso contrário, será testada uma nova alternativa de resolução do problema, começando por testar as 3 velocidades que se pode avançar (diminuir em 1, manter ou aumentar em 1). É, posteriormente, testado neste `for loop` se a nossa `new_speed` contém todos os requisitos para ser uma solução, testado se é maior que 1, menor que 9 (`max_road_speed`) e se com ela não iremos exceder a posição final. Com os testes casa a casa para verificar se podemos atravessar as casas pretendidas com aquela velocidade, vamos descobrir se a mesma é adequada e, caso “*i*” chegue ao ponto de ser maior que `new_speed`, significa que a mesma é valida para todas as casas chamando, desta maneira, a recursiva novamente com o novo valor de velocidade, posição e incrementando em 1 o número de movimentos.

2^a Solução (solution_2_recursion) – Brute Force Otimizada

A segunda solução consiste apenas na melhoria da solução anterior apresentada previamente como referido.

Como referido, o intuito da solução anterior era gerar todas as soluções possíveis para as diferentes posições finais, logo apenas precisamos de colocar uma condição que verifica se a solução já guardada apresenta uma posição maior num menor número de movimentos do que as possibilidades que iríamos testar a seguir. Logo, após esta verificação, serão apenas testadas opções em que as soluções sejam melhores, evitando a geração de todas as sequências de soluções possíveis, sendo assim possível chegar ao último segmento (800), adicionando apenas esta pequena condição.

```
static void solution_2_recursion(int move_number, int position, int speed, int final_position)
{
    int i, new_speed;

    solution_2_count++;
    solution_2.positions[move_number] = position;

    if (position == final_position && speed == 1)
    {
        if (move_number < solution_2_best.n_moves)
        {
            solution_2_best = solution_2;
            solution_2_best.n_moves = move_number;
        }
        return;
    }

    if (solution_2_best.positions[move_number] > solution_2.positions[move_number])
    {
        return;
    }

    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);
        }
}
```

中 中 中 中 中

SPEED RUN

3º Solução (solution_3_recursion)

A terceira solução já provém de um algoritmo criado por nós.

Esta solução também recorre a uma função recursiva em que o seu primeiro objetivo é tentar acelerar o carro. O método que usámos para testar as diferentes velocidades foi implementado na função `testarVelocidade1()`.

Como já referido, a ideia principal é acelerar o carro, mas antes é testada essa possibilidade. Verificamos essa possibilidade garantindo que, dependendo das velocidades dos segmentos futuros, se o carro tiver de abrandar, seja possível fazê-lo com antecedência.

Recorrendo a um exemplo, se o carro estiver com velocidade 5, vamos testar a próxima velocidade sendo esta 6 (prioridade de tentativa de aumento de velocidade). Para garantir que o carro consegue abrandar até qualquer que seja a velocidade limite de cada segmento, vamos ter que testar os 21 ($6+5+4+3+2+1 = 21$) segmentos seguintes pois, para o carro abrandar até à velocidade 1, terá este número de casas a percorrer antes de o conseguir fazer. Este método é bastante eficaz na redução da velocidade quando chegamos à mais variada posição final que se pretende.

De um modo mais geral, se verificarmos que o carro não consegue abrandar até à sua posição final, teremos que fazer o mesmo teste, mas agora testando apenas o manutenção da velocidade (no exemplo anterior, testamos os $5+4+3+2+1 = 15$ segmentos seguintes) e, se também não passar no teste, no pior dos casos teremos que reduzir a velocidade (no exemplo anterior, testamos os $4+3+2+1 = 20$ segmentos seguintes).

Tal como na solução mencionada nos dois pontos anteriores, esta também possui um campo onde testa e guarda as melhores posições. Começa por criar uma nova variável (*new_speed*) que será incrementada em 1 em relação à velocidade original, de modo a testar a subida do valor de speed. Entrando no *while loop*, vão ser testados os valores de *new_speed* até encontrar algum que retenha todos os requisitos.

```

bool testarVelocidade1(int new_speed, int position, int final_position)
{
    int x, i, casas_a_andar = 0;
    int record_position = 0;

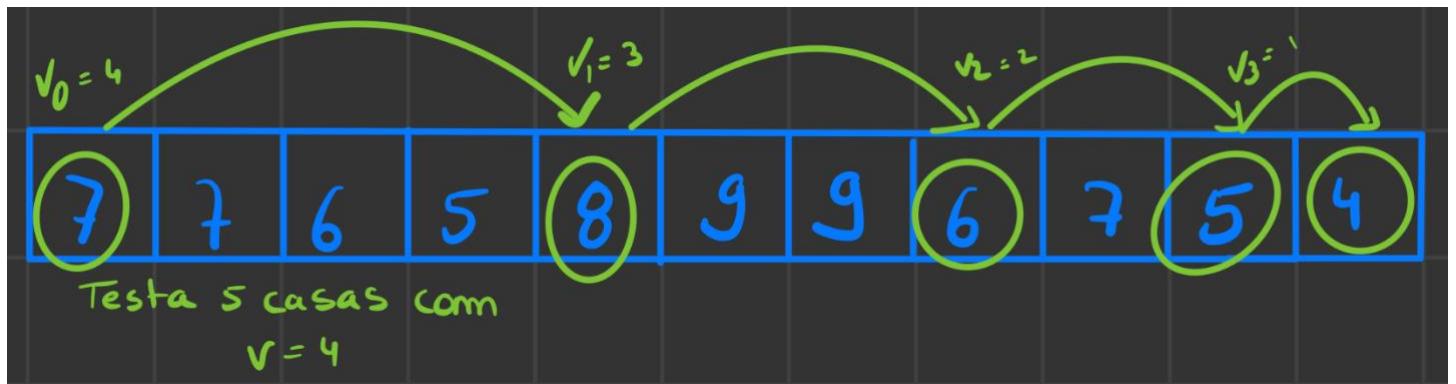
    for (i = 1; i <= new_speed; i++)
    {
        casas_a_andar += i;
    }

    for (x = new_speed; x >= 1; x--)
    {
        // record_position += x;
        for (i = 0; i <= x; i++)
        {
            if (x > max_road_speed[position + record_position + i] || position + casas_a_andar > final_position || new_speed < 1)
            {
                return false;
            }
        }
        record_position += x;
    }
    return true;
}

```

Esta é uma função que retorna um Boolean, de maneira a que consigamos controlar a saída do loop. Foi criada a variável “casas_a_andar” onde é feita a soma do total de casas que queremos avançar, por exemplo se o nosso valor de *new_speed* for 4, *casas_a_andar* será 4+3+2+1.

Dentro do segundo *for loop* é onde começa o teste para todos os valores da velocidade e as respetivas casas que pretende avançar. O valor de *x* corresponde aos diferentes valores da velocidade (do exemplo anterior, se a velocidade for igual a 4 será testada a velocidade de 4 para 5 casas, velocidade 3 para 4 casas, etc) e o valor de “*i*” às diferentes casas que vai ter de percorrer.



Se algum dos valores de *new_speed* não passar nos testes, excendendo a velocidade máxima permitida numa casa, sendo menor que um ou a sua soma com o valor da posição exceder a posição final, a função terminará e retornará “false”, o que permitirá ao *while loop* anteriormente referido diminuir em 1 o valor da velocidade e voltar a chamar a função de testes.

```
while (verificador == false)
{
    verificador = testarVelocidade1(new_speed, position, final_position);
    if (verificador == false)
    {
        new_speed--;
    }
}
```

Caso todos os testes sejam concluídos com sucesso, retornará “true” e, se for uma solução válida, irá guardar os seus valores numa nova chamada da função recursiva.

4^a Solução (solution_4_non_recursion)

A quarta solução é apenas a implementação da solução anterior não recorrendo à recursividade.

Resolvemos fazê-la para verificar a presença de alguma diferenças que pudessem vir a surgir em relação à eficiência ou velocidade da mesma.

```
static void solution_4_non_recursion(int final_position)
{
    int move_number = 0;
    int position = 0;
    int speed = 0;
    int new_speed;
    bool verificador = false;

    solution_3_count++;
    solution_3.positions[move_number] = position;

    while (position != final_position)
    {
        verificador = false;
        new_speed = speed + 1;

        while (verificador == false)
        {
            verificador = testarVelocidade2(new_speed, position, final_position);
            if (verificador == false)
            {
                new_speed--;
            }
        }

        if (new_speed <= _max_road_speed_ && position + new_speed <= final_position && new_speed > 1)
        {
            move_number++;
            position += new_speed;
            speed = new_speed;
            solution_4.positions[move_number] = position;
        }
    }

    if (position == final_position && speed == 1)
    {
        if (move_number < solution_4_best.n_moves)
        {
            solution_4_best = solution_4;
            solution_4_best.n_moves = move_number;
        }
    }
    return;
}
```

Ao contrário da solução anterior que guarda o progresso através da chamada de si mesmo, nesta foram criadas variáveis que irão agir da mesma maneira.

```
if (new_speed <= _max_road_speed_ && position + new_speed <= final_position && new_speed >= 1)
{
    move_number++;
    position += new_speed;
    speed = new_speed;
    solution_4.positions[move_number] = position;
}
```

5º Solução (solution_5_dynamic)

A quinta solução é a implementação de uma resolução dinâmica do problema, sendo eta a mais eficiente de todas as apresentadas.

Esta solução acentou na ideia base da solução 3, tendo sido adicionado um método para guardar posições antigas e a partir delas conseguirar retirar a última posição, velocidade e número de movimentos de um percurso já calculado.

```
static int last_speed = 0, last_position = 0;
static int guardar_listas = 0, listas_guardadas = 0;

bool testarVelocidade3(int new_speed, int position, int final_position)
{
    int x, i, casas_a_andar = 0;
    int record_position = 0;

    for (i = 1; i <= new_speed; i++)
    {
        casas_a_andar += i;
    }

    int teste_velocidade = position;

    for (x = new_speed; x >= 1; x--)
    {
        for (i = 0; i <= x; i++)
        {
            teste_velocidade++;

            if (teste_velocidade > final_position && final_position > 1)
            {
                guardar_listas = 1;
            }

            if (x > max_road_speed[position + record_position + i] || position + casas_a_andar > final_position || new_speed < 1)
            {
                return false;
            }
        }
        record_position += x;
    }
    return true;
}
```

Esta função foi já explicada na secção da solução 3, tendo sido apenas adicionado o método para indicar à função principal quando guardar as listas de posições antigas.

Foi criada uma variável chamada *teste_velocidade* inicializada com o número da posição atual para nos ajudar a percorrer todas as posições que os *for loops* testam. A partir do momento que esta variável é maior que o valor da posição final (apenas são guardadas listas quando o percurso guardado chegou ao fim), iremos colocar a variável estática *guardar_listas* com valor 1, para a função principal saber que vai guardar as respetivas listas. Também foi criada uma variável *listas_guardadas* como indicadora que uma lista anterior já está guardada.

SPEED RUN

```
static void solution_5_dynamic(int move_number, int position, int speed, int final_position)
{
    int new_speed;
    bool verificador = false;

    solution_5_count++;
    solution_5.positions[move_number] = position;

    if (position == final_position && speed == 1)
    {
        solution_5_best = solution_5;
        solution_5_best.n_moves = move_number;
        return;
    }
    else
    {
        new_speed = speed + 1;

        while (verificador == false)
        {
            verificador = testarVelocidade3(new_speed, position, final_position);
            if (verificador == false)
            {
                new_speed--;
            }
        }

        if (guardar_listas == 1 && listas_guardadas == 0)
        {
            listas_guardadas = 1;
            for (int i = 0; i <= final_position; i++)
            {
                if (i < move_number)
                {
                    solution_5_listas_guardadas.positions[i] = solution_5.positions[i];
                }
                else
                {
                    solution_5_listas_guardadas.positions[i] = 0;
                }
            }
            solution_5_listas_guardadas.n_moves = move_number - 1;
            last_position = solution_5_listas_guardadas.positions[move_number - 1];
            last_speed = solution_5_listas_guardadas.positions[move_number - 1] - solution_5_listas_guardadas.positions[move_number - 2];
        }
    }

    solution_5_dynamic(move_number + 1, position + new_speed, new_speed, final_position);
}
}
```

A função recursiva também tem como base a já desenvolvida na solução 3, sendo acrescentado o método de guardar as posições . Através de um `if` vamos verificar se a função anterior mandou “guardar” as posições e se já existe alguma lista guardada. Caso seja para guardar as listas e não exista nenhuma guardada , iremos proceder ao método para as guardar.

Este processo, é feito através de um *for loop* que percorre os índices das posições todas sem exceção até à *final_position*. Com isto, iremos guardar as posições calculadas apenas até ao número do movimento atual, pois ainda não existe cálculo de posições posteriores.

Após guardar a lista de posições, criámos duas variáveis estáticas `last_position` e `last_speed`, onde serão armazenados, respetivamente, a última posição guardada e a última velocidade guardada (calculada a partir da diferença de posições consecutivas).

```
static void solve_5(int final_position)
{
    if (final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr, "solve_5: bad final_position\n");
        exit(1);
    }
    solution_5_elapsed_time = cpu_time();
    solution_5_count = 0ul;
    solution_5_best.n_moves = final_position + 100;
    solution_5.n_moves = solution_5_listas_guardadas.n_moves;
    for (int i = 0; i <= final_position; i++)
    {
        solution_5.positions[i] = solution_5_listas_guardadas.positions[i];
    }
    guardar_listas = 0;
    listas_guardadas = 0;
    solution_5_dynamic(solution_5_listas_guardadas.n_moves, last_position, last_speed, final_position);
    solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;
}
```

Na função principal irão ocorrer diversos processos.

Inicialmente, iremos transportar o número de movimentos guardado, para o novo cálculo. Depois iremos transportar as posições guardadas anteriormente, para as posições atuais e, consequentemente, voltaremos a colocar as variáveis `guardar_listas` e `listas_guardadas` a 0, para podermos voltar a guardar listas futuras.

Por fim, chamaremos a função recursiva, logicamente, apenas calculando os valores a partir da última posição guardada nas listas como já explicado.

COMPARAÇÃO

Para se ter uma melhor noção de como o tempo de execução de cada algoritmo vai depender das especificações da máquina onde está a ser executado, decidimos executar os algoritmos em três computadores diferentes.

	Memória RAM	Processador
PC André	16GB	Apple M2
PC Duarte	16GB	Apple M1
PC Rodrigo	16GB	11th Gen Intel® Core™ i7-1165G7 @ 2.80Ghz

Nas páginas seguintes vão ser apresentados diversos gráficos a demonstrar e a comparar os tempos de execução para as diferentes abordagens ao problema.

Optámos por colocar um gráfico, para cada PC, a comparar:

- **Brute Forces** (soluções 1 e 2);
- **Brute Force Otimizada** e recursiva desenvolvida (soluções 2 e 3);
- Recursiva desenvolvida e não recursiva com mesma ideia (soluções 3 e 4);
- Recursiva desenvolvida e dinâmica com mesma base (soluções 3 e 5);

Por fim, expusemos um último gráfico onde entram todos os algoritmos executados por todos os PC, com o intuito de se visualizar claramente as diferenças significativas entre os métodos brutos (**Brute Forces**) e os restantes desenvolvidos.

Nota: Foram obtidos dados até posições finais de 8000 para melhor percepção e visualização das conclusões a tirar, conseguindo “contrariar”/“esconder” os ruídos existentes nos resultados. Teve como consequência a pequena demora dos computadores no começo da execução da solução, sendo os valores de tempos mais pequenos quando calculados para 800 posições.

Brute Forces (soluções 1 e 2)

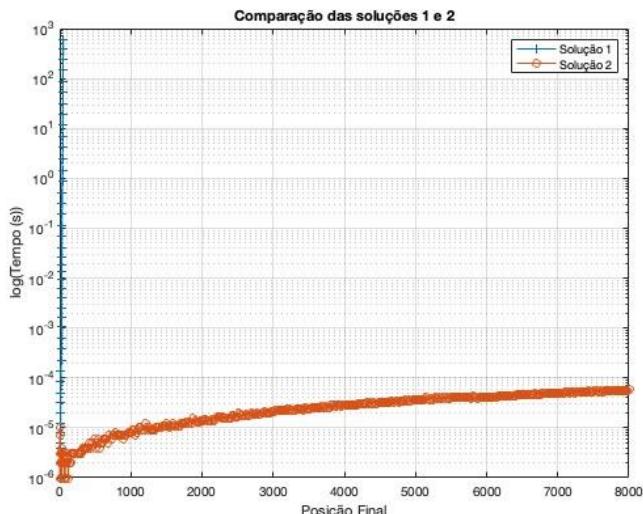


Figura 3 | Gráfico pertencente a PC Duarte

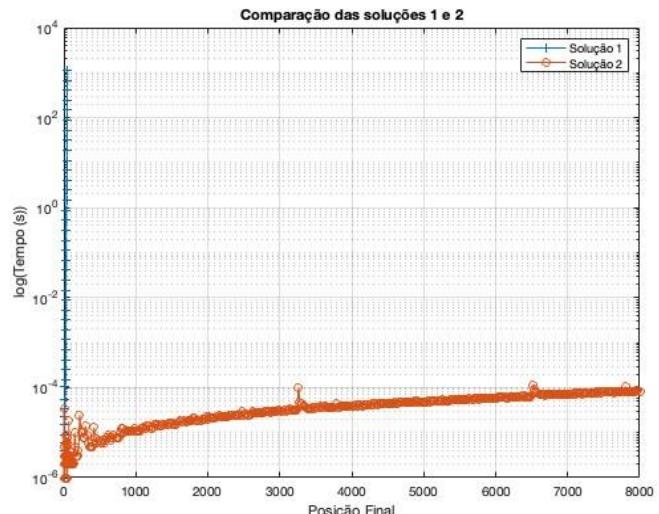


Figura 2 | Gráfico pretendente a PC André

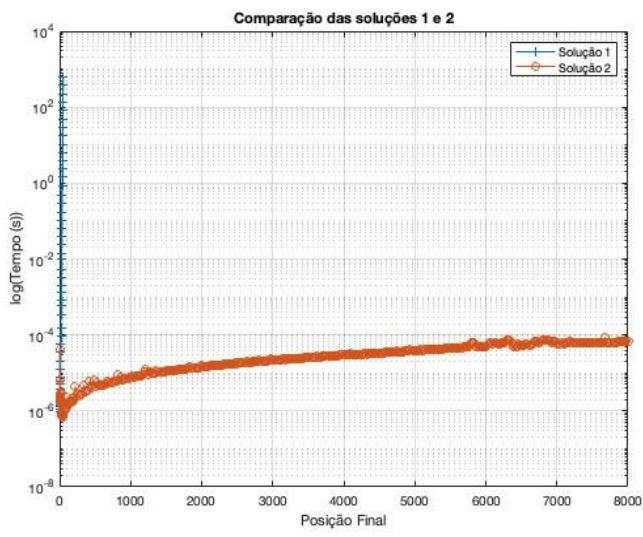


Figura 5 | Gráfico pertencente a PC Rodrigo

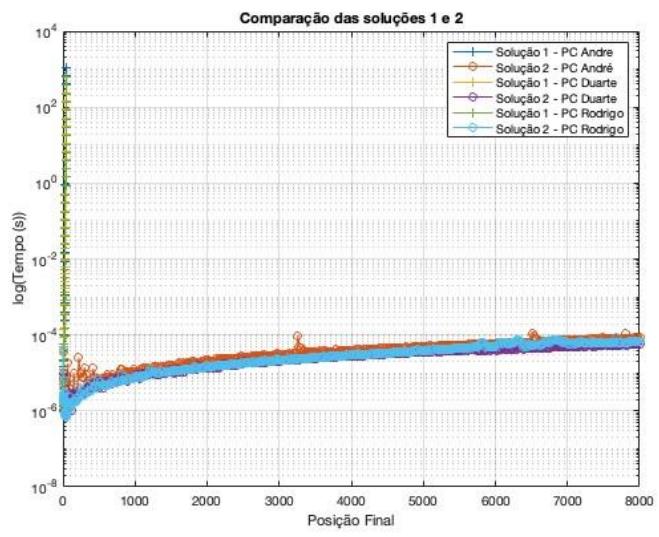


Figura 4 | Gráfico pertencente aos 3 PC's

Como se pode observar nos gráficos acima, a velocidade da solução Brute Force Otimizada é bastante mais elevada que a solução Brute Force normal, assim como o número de posições finais que a primeira consegue calcular é bastante maior que a capacidade de cálculo da segunda no mesmo tempo.

Brute Force Otimizada e recursiva desenvolvida (soluções 2 e 3)

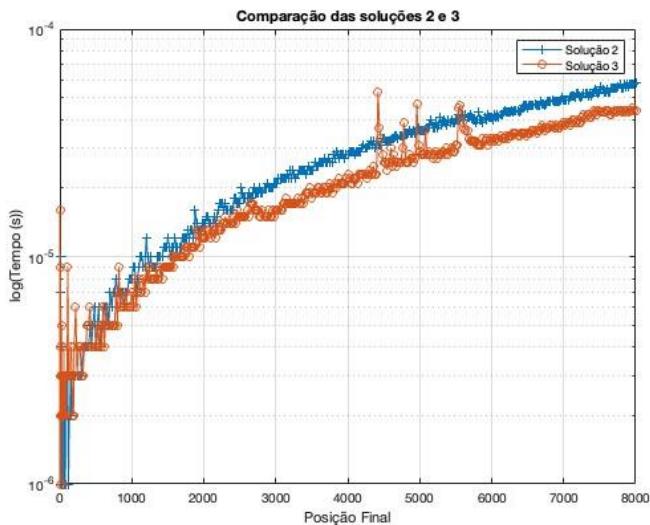


Figura 7 | Gráfico pertencente a PC Duarte

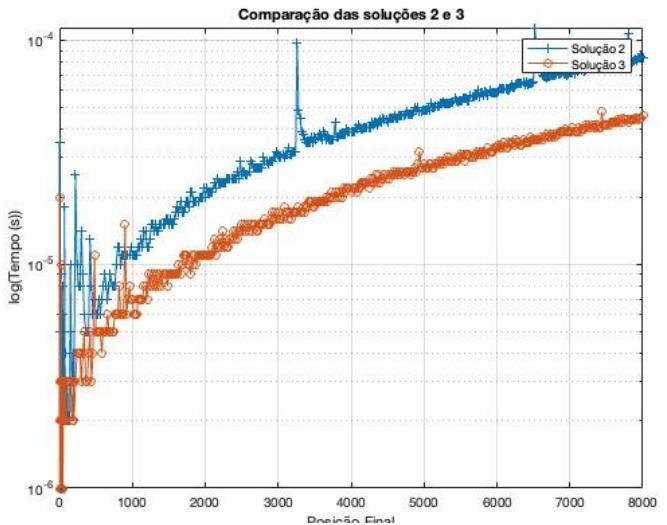


Figura 6 | Gráfico pertencente a PC André

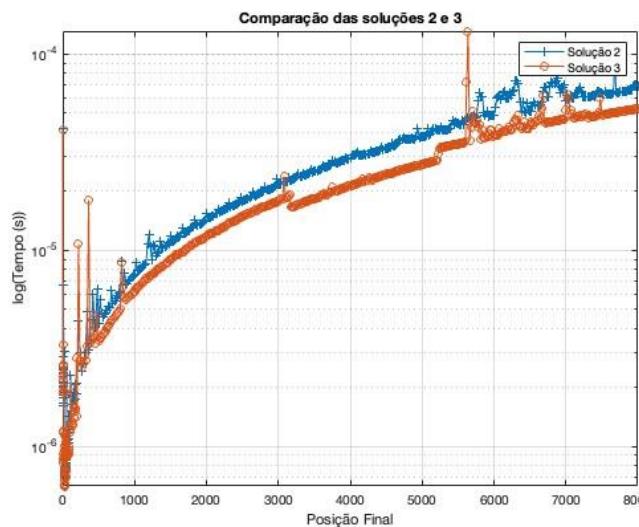


Figura 9 | Gráfico pertencente a PC Rodrigo

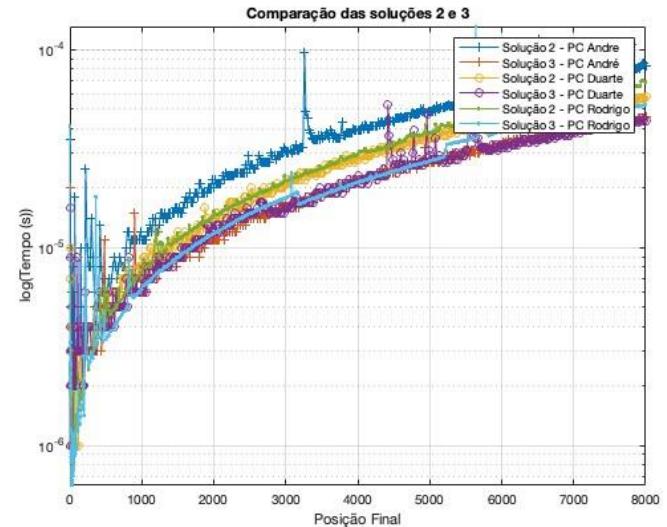


Figura 10 | Gráfico pertencente aos 3 PC's

Como se pode observar nos gráficos acima, conseguimos perceber que a solução recursiva desenvolvida é significativamente mais rápida do que a solução Brute Force Otimizada, no entanto, observamos que as duas são capazes de calcular valores de posições finais bastante altos.

Recursiva desenvolvida e não recursiva com mesma ideia (soluções 3 e 4)

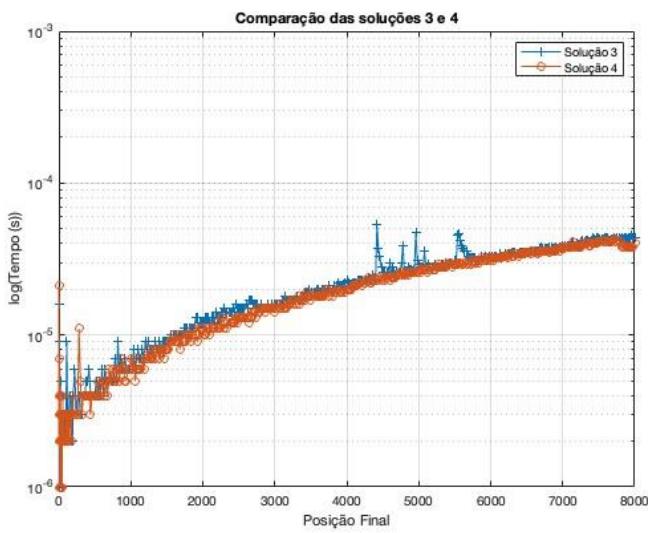


Figura 11 | Gráfico pertencente a PC Duarte

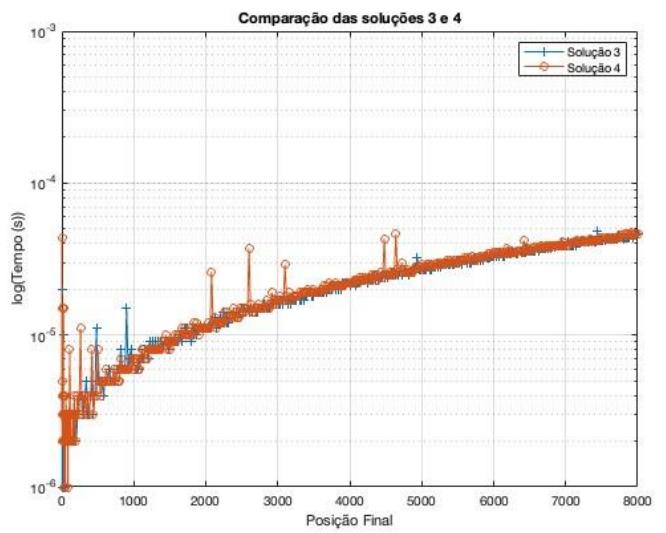


Figura 42 | Gráfico pertencente a Fórmula

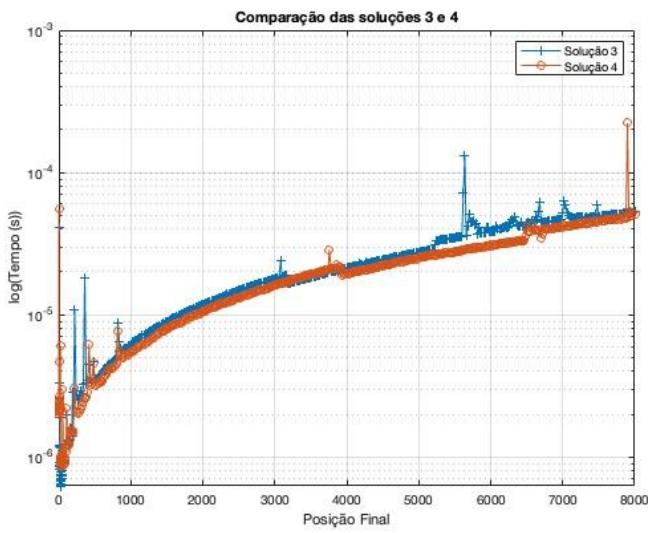


Figura 13 | Gráfico pertencente a PC Rodrigo

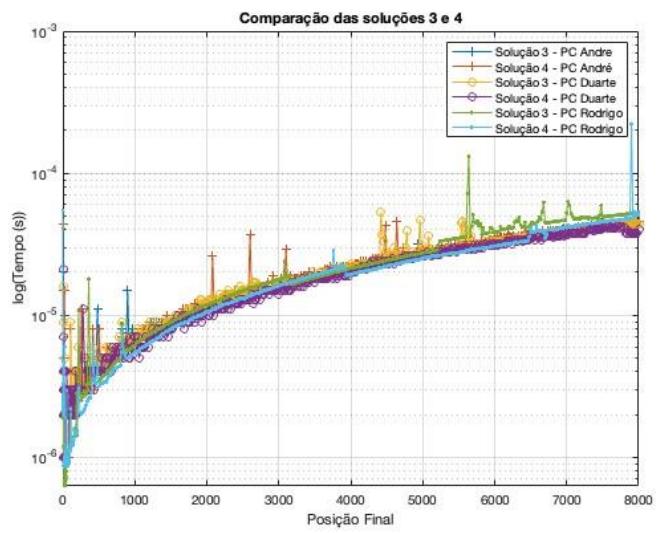


Figura 14 | Gráfico pertencente aos 3 PC's

Os gráficos representados acima foram criados com o intuito de comparar uma solução recursiva com uma não recursiva, porém com a mesma resolução do problema. Estes, esclareceram a nossa curiosidade relativamente à questão de haver, ou não, diferença no tempo de execução de uma função recursiva. Conseguimos concluir que não existe diferença.

Recursiva desenvolvida e dinâmica com a mesma base (soluções 3 e 5)

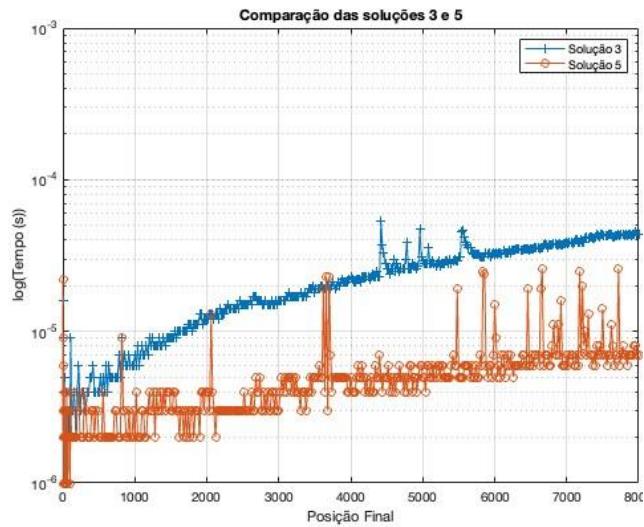


Figura 15 | Gráfico pertencente a PC Duarte

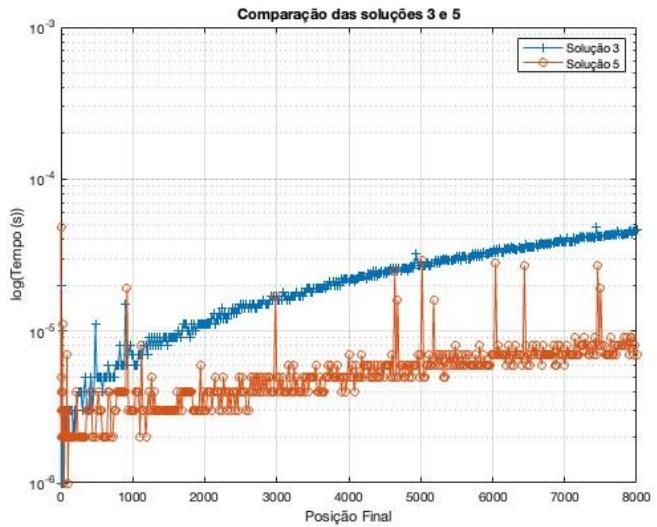


Figura 16 | Gráfico pertencente a PC André

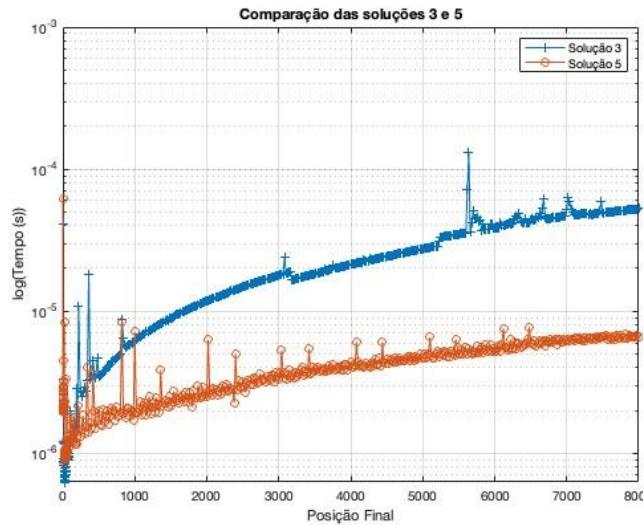


Figura 17 | Gráfico pertencente a Rodrigo

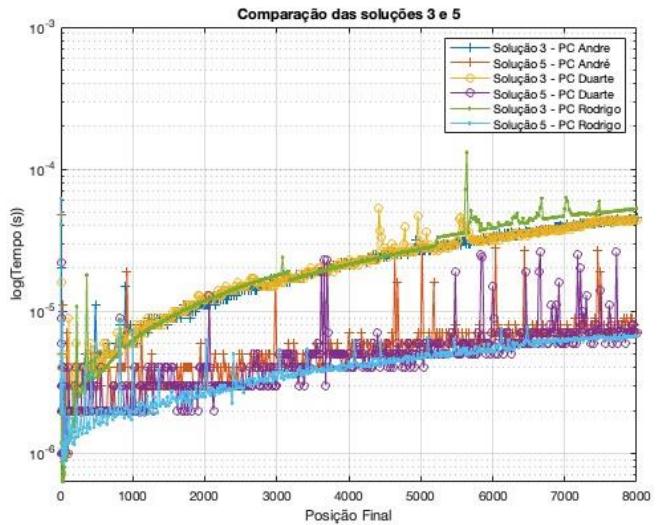
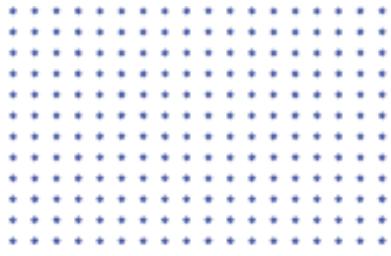


Figura 18 | Gráfico pertencente aos 3 PCs

Como se pode observar nos gráficos acima, a diferença de velocidade de execução da nossa solução recursiva e da dinâmica é bastante evidente, sendo a dinâmica muito mais rápida, obviamente, tendo em conta a escala pequena em que estamos a trabalhar.

SPEED RUN

CONCLUSÃO



Em suma, este trabalho demonstrou que diferentes complexidades computacionais irão originar tempos de execução muito diferentes, o que nos leva a concluir que na resolução de um problema é tão relevante a sua resolução como a decisão sobre a abordagem a seguir.

Abaixo, apresenta-se um gráfico com todas as soluções de todos os PC colocados no mesmo gráfico, onde podemos observar bem todas as conclusões que tirámos.

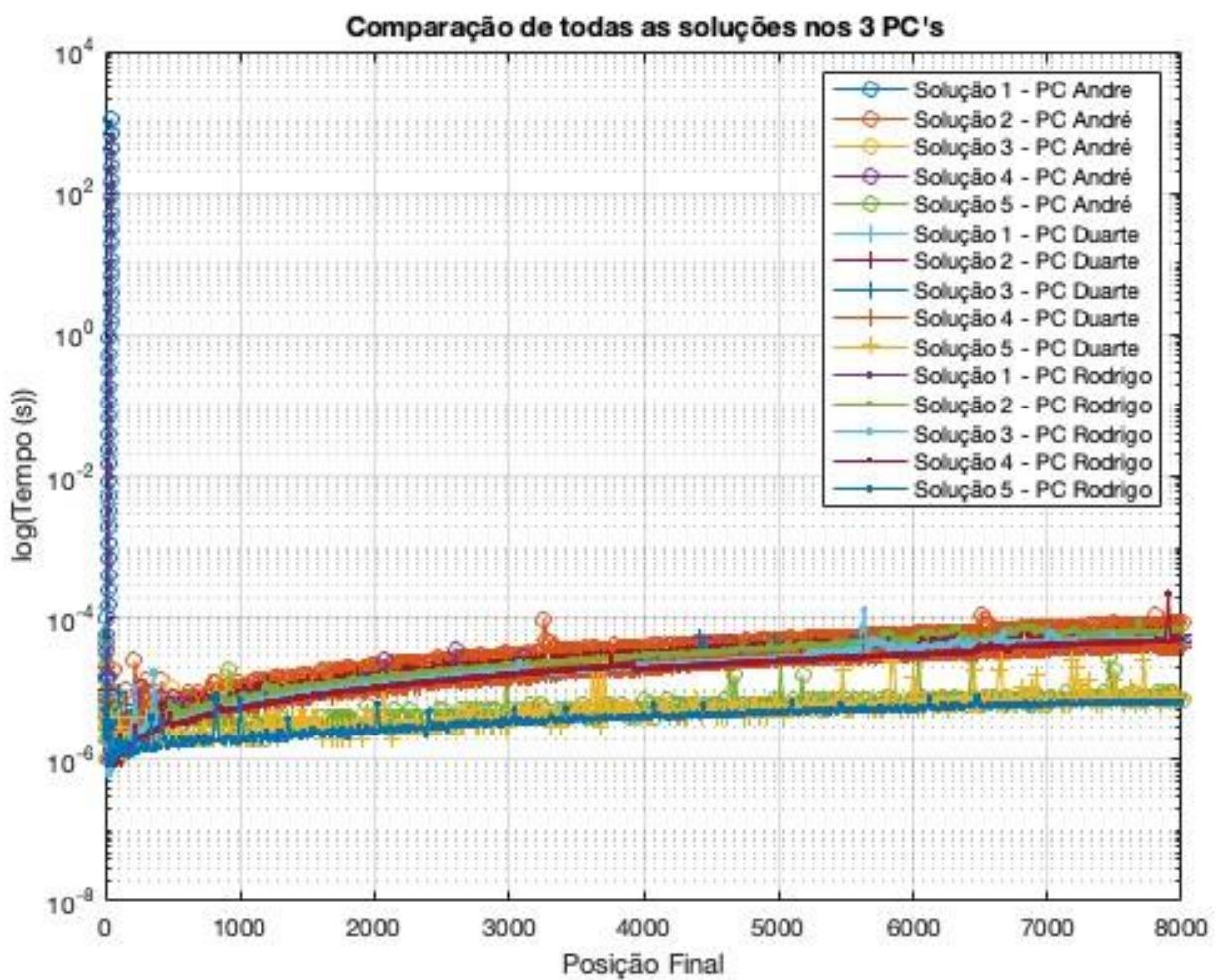


Figura 19 | Gráfico comparação todas as soluções e respetivos PC

APÊNDICE

Código 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. 107637 Name: André Oliveira  
// N.Mec. 107359 Name: Duarte Cruz  
// N.Mec. 107634 Name: Rodrigo Graça  
//  
// 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 ".\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)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;  
static solution_t solution_1, solution_1_best;  
static solution_t solution_2, solution_2_best;  
static solution_t solution_3, solution_3_best;  
static solution_t solution_4, solution_4_best;  
static solution_t solution_5, solution_5_best; // solution_5 listas guardadas;  
static double solution_1_elapsed_time; // time it took to solve the problem 1  
static double solution_2_elapsed_time; // time it took to solve the problem 2  
static double solution_3_elapsed_time; // time it took to solve the problem 3  
static double solution_4_elapsed_time; // time it took to solve the problem 4  
static double solution_5_elapsed_time; // time it took to solve the problem 5  
static unsigned long solution_1_count; // effort dispended solving the problem 1  
static unsigned long solution_2_count; // effort dispended solving the problem 2  
static unsigned long solution_3_count; // effort dispended solving the problem 3  
static unsigned long solution_4_count; // effort dispended solving the problem 4  
static unsigned long solution_5_count; // effort dispended solving the problem 5  
//  
// solution 1 (the very inefficient recursive solution given to the students)  
//  
static void solution_1_recursion(int move_number, int position, int speed, int final_position)  
{  
    int i, new_speed;  
  
    // record move  
    solution_1_count++;  
    solution_1.positions[move_number] = position;  
    // is it a solution?  
    if (position == final_position && speed == 1)  
    {  
        // is it a better solution?  
        if (move_number < solution_1_best.n_moves)  
        {  
            solution_1_best = solution_1;  
            solution_1_best.n_moves = move_number;  
        }  
        return;  
    }  
    // no, try all legal speeds  
    for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)  
        if (new_speed > 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)  
        {  
            for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)  
            {  
                if (i > new_speed)  
                    solution_1_recursion(move_number + 1, position + new_speed, new_speed, final_position);  
            }  
        }  
    }  
//  
// solution 2 (Solução melhorada do prof)  
//  
static void solution_2_recursion(int move_number, int position, int speed, int final_position)  
{  
    int i, new_speed;  
  
    solution_2_count++;  
    solution_2.positions[move_number] = position;  
  
    if (position == final_position && speed == 1)  
    {  
        if (move_number < solution_2_best.n_moves)  
        {  
            solution_2_best = solution_2;  
            solution_2_best.n_moves = move_number;  
        }  
        return;  
    }  
    if (solution_2_best.positions[move_number] > solution_2.positions[move_number])  
    {  
        return;  
    }  
    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++)  
            {  
                #* * * * SPEED RUN * * * *  
                #* * * *  
            }  
        }  
    }  
}
```



```

        guardar_listas = 1;
    }
    if (x > max_road_speed(position + record_position + i) || position + casas_a_andar > final_position || new_speed < 1)
    {
        return false;
    }
    record_position += x;
}
return true;
}

static void solution_5_dynamic(int move_number, int position, int speed, int final_position)
{
    int new_speed;
    bool verificador = false;
    solution_5_count++;
    solution_5.positions[move_number] = position;
    if (position == final_position && speed == 1)
    {
        solution_5_best = solution_5;
        solution_5_best.n_moves = move_number;
        return;
    }
    else
    {
        new_speed = speed + 1;
        while (verificador == false)
        {
            verificador = testarVelocidade3(new_speed, position, final_position);
            if (verificador == false)
            {
                new_speed--;
            }
        }
        if (guardar_listas == 1 && listas_guardadas == 0)
        {
            listas_guardadas = 1;
            for (int i = 0; i <= final_position; i++)
            {
                if (i < move_number)
                {
                    solution_5_listas_guardadas.positions[i] = solution_5.positions[i];
                }
                else
                {
                    solution_5_listas_guardadas.positions[i] = 0;
                }
            }
            solution_5_listas_guardadas.n_moves = move_number - 1;
            last_position = solution_5_listas_guardadas.positions[move_number - 1];
            last_speed = solution_5_listas_guardadas.positions[move_number - 1] - solution_5_listas_guardadas.positions[move_number - 2];
        }
        solution_5_dynamic(move_number + 1, position + new_speed, new_speed, final_position);
    }
}

// define solution5
// solve_1
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 = 0;
    solution_1_best.n_moves = final_position + 100;
    solution_1_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_2: bad final_position\n");
        exit(1);
    }
    solution_2_elapsed_time = cpu_time();
    solution_2_count = 0;
    solution_2_best.n_moves = final_position + 100;
    solution_2_recursion(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 = 0;
    solution_3_best.n_moves = final_position + 100;
    solution_3_recursion(0, 0, 0, final_position);
    solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}

static void solve_4(int final_position)
{
    if (final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr, "solve_4: bad final_position\n");
        exit(1);
    }
    solution_4_elapsed_time = cpu_time();
    solution_4_count = 0;
    solution_4_best.n_moves = final_position + 100;
    solution_4_non_recursion(final_position);
    solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}

static void solve_5(int final_position)
{
    if (final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr, "solve_5: bad final_position\n");
        exit(1);
    }
    solution_5_elapsed_time = cpu_time();
    solution_5_count = 0;
    solution_5_best.n_moves = final_position + 100;
    solution_5.n_moves = solution_5_listas_guardadas.n_moves;
    for (int i = 0; i < final_position; i++)
    {
        solution_5.positions[i] = solution_5_listas_guardadas.positions[i];
    }
    guardar_listas = 0;
    listas_guardadas = 0;
    solution_5_dynamic(solution_5_listas_guardadas.n_moves, last_position, last_speed, final_position);
    solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;
}

// main program
// solve_5

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];

    // initialization
    n_mec = (argc < 2) ? 0xAED2022 : atoi(argv[1]);
    random((unsigned int)n_mec);
    init_road_speeds();
    // all solution methods for all interesting sizes of the problem
    final_position = 1;
    solution_1_elapsed_time = 0.0;
    printf("-----+\n");
    #* * * * *
    #* * * * * SPEED RUN
    #* * * * *

```

```

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
#if defined solution1
    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("                !");
    }
#endif

    // second solution method
#if defined solution2
    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("                !");
    }
#endif

    // third solution method
#if defined solution3
    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("                !");
    }
#endif

    // fourth solution method
#if defined solution4
    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("                !");
    }
#endif

    // fifth solution method
#if defined solution5
    if (solution_5_elapsed_time < _time_limit_)
    {
        solve_5(final_position);
        if (print_this_one != 0)
        {
            sprintf(file_name, "%03d_5.pdf", final_position);
            make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_5_best.n_moves, &solution_5_best.positions[0], solution_5_elapsed_time, solution_5_count, "Plain recursion");
        }
        printf(" %3d %16lu %9.3e !", solution_5_best.n_moves, solution_5_count, solution_5_elapsed_time);
    }
    else
    {
        solution_5_best.n_moves = -1;
        printf("                !");
    }
#endif

    // 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_
}

```

SPEED RUN

Código Matlab

%% Solução 1 (PC Duarte)

```
valores = load("resultados_1_Duarte.txt");
pos_finais = valores(:,1);
tempos = valores(:,4);
```

```
figure(1);  
plot(pos_finais,log10(tempos));  
ylabel("Tempo (s)");  
xlabel("Posição Final");  
title("Solução fornecida");  
hold on;
```

```

tempos_log = log10(tempos);
N = [pos_finais(20:end) 1+0*pos_finais(20:end)];
Coefs = pinv(N)*tempos_log(20:end);
Ntotal = [pos_finais pos_finais*0+1];

```

```
plot(pos_finais, Ntotal*Coefs, "k");
legend("Solução 1", "Regressão Linear");
grid on;
hold off;
```

```
t800_log = [800 1]*Coefs;  
t800 = 10^t800_log;  
fprintf('Tempo previsto até à posição 800 = %i\n',t800);
```

%% Solução 1 e Solução 2 (PC André)

```
valores = load("resultados_1_Aandre.txt");
pos_finais1 = valores(:,1);
tempos1 = valores(:,4);
```

```
valores = load("resultados_2_Angela.txt");
```

SPEED RUN

```

pos_finais2 = valores(:,1);
tempos2 = valores(:,4);

figure(1);
semilogy(pos_finais1,tempos1, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 1 e 2")
hold on;
semilogy(pos_finais2,tempos2, "o-");
legend("Solução 1", "Solução 2");
grid on;
hold off;

```

%% Solução 1 e Solução 2 (PC duarte)

```

valores = load("resultados_1_Duarte.txt");
pos_finais1 = valores(:,1);
tempos1 = valores(:,4);

valores = load("resultados_2_Duarte.txt");
pos_finais2 = valores(:,1);
tempos2 = valores(:,4);

```

```

figure(1);
semilogy(pos_finais1,tempos1, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 1 e 2");
hold on;
semilogy(pos_finais2,tempos2, "o-");
legend("Solução 1", "Solução 2");
grid on;
hold off;

```

%% Solução 1 e Solução 2 (PC Rodrigo)

```

valores = load("resultados_1_Rodrigo.txt");

```

*** * * * * SPEED RUN * * * * ***

```

pos_finais1 = valores(:,1);
tempos1 = valores(:,4);

valores = load("resultados_2_Rodrigo.txt");
pos_finais2 = valores(:,1);
tempos2 = valores(:,4);

figure(1);
semilogy(pos_finais1,tempos1, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 1 e 2");
hold on;
semilogy(pos_finais2,tempos2, "o-");
legend("Solução 1", "Solução 2");
grid on;
hold off;

```

%% Solução 1 e Solução 2 (3 PC's)

```

valores = load("resultados_1_Andre.txt");
pos_finais1A = valores(:,1);
tempos1A = valores(:,4);

valores = load("resultados_2_Andre.txt");
pos_finais2A = valores(:,1);
tempos2A = valores(:,4);

valores = load("resultados_1_Duarte.txt");
pos_finais1D = valores(:,1);
tempos1D = valores(:,4);

valores = load("resultados_2_Duarte.txt");
pos_finais2D = valores(:,1);
tempos2D = valores(:,4);

```

```

valores = load("resultados_1_Rodrigo.txt");
pos_finais1R = valores(:,1);

```

+

SPEED RUN

+

```

tempos1R = valores(:,4);

valores = load("resultados_2_Rodrigo.txt");
pos_finais2R = valores(:,1);
tempos2R = valores(:,4);

figure(1);
semilogy(pos_finais1A,tempos1A, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 1 e 2");
hold on;
semilogy(pos_finais2A,tempos2A, "o-");
semilogy(pos_finais1D,tempos1D, "+-");
semilogy(pos_finais2D,tempos2D, "o-");
semilogy(pos_finais1R,tempos1R, "+-");
semilogy(pos_finais2R,tempos2R, "o-");
legend("Solução 1 - PC Andre", "Solução 2 - PC André", ...
    "Solução 1 - PC Duarte", "Solução 2 - PC Duarte", ...
    "Solução 1 - PC Rodrigo", "Solução 2 - PC Rodrigo");
grid on;
hold off;

```

%% Solução 2 e Solução 3 (PC André)

```

valores = load("resultados_2_Angelo.txt");
pos_finais2 = valores(:,1);
tempos2 = valores(:,4);

valores = load("resultados_3_Angelo.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);

figure(1);
semilogy(pos_finais2,tempos2, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 2 e 3");

```

***** SPEED RUN *****

```

hold on;
semilogy(pos_finais3,tempos3,"o-");
legend("Solução 2", "Solução 3");
grid on;
hold off;

```

%% Solução 2 e Solução 3 (PC Duarte)

```

valores = load("resultados_2_Duarte.txt");
pos_finais2 = valores(:,1);
tempos2 = valores(:,4);

```

```

valores = load("resultados_3_Duarte.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);

```

```

figure(1);
semilogy(pos_finais2,tempos2, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 2 e 3");
hold on;
semilogy(pos_finais3,tempos3,"o-");
legend("Solução 2", "Solução 3");
grid on;
hold off;

```

%% Solução 2 e Solução 3 (PC Rodrigo)

```

valores = load("resultados_2_Rodrigo.txt");
pos_finais2 = valores(:,1);
tempos2 = valores(:,4);

```

```

valores = load("resultados_3_Rodrigo.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);

```

```

figure(1);

```

+

SPEED RUN

+

```

semilogy(pos_finais2,tempos2, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 2 e 3");
hold on;
semilogy(pos_finais3,tempos3,"o-");
legend("Solução 2", "Solução 3");
grid on;
hold off;

```

%% Solução 2 e Solução 3 (3 PC's)

```

valores = load("resultados_2_Aandre.txt");
pos_finais2A = valores(:,1);
tempos2A = valores(:,4);

```

```

valores = load("resultados_3_Aandre.txt");
pos_finais3A = valores(:,1);
tempos3A = valores(:,4);

```

```

valores = load("resultados_2_Duarte.txt");
pos_finais2D = valores(:,1);
tempos2D = valores(:,4);

```

```

valores = load("resultados_3_Duarte.txt");
pos_finais3D = valores(:,1);
tempos3D = valores(:,4);

```

```

valores = load("resultados_2_Rodrigo.txt");
pos_finais2R = valores(:,1);
tempos2R = valores(:,4);

```

```

valores = load("resultados_3_Rodrigo.txt");
pos_finais3R = valores(:,1);
tempos3R = valores(:,4);

```

```

figure(1);
semilogy(pos_finais2A,tempos2A, "+-");

```

+

SPEED RUN

+

```

ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 2 e 3");
hold on;
semilogy(pos_finais3A,tempos3A,"+-");
semilogy(pos_finais2D,tempos2D,"o-");
semilogy(pos_finais3D,tempos3D,"o-");
semilogy(pos_finais2R,tempos2R,".-");
semilogy(pos_finais3R,tempos3R,".-");
legend("Solução 2 - PC Andre", "Solução 3 - PC André", ...
    "Solução 2 - PC Duarte", "Solução 3 - PC Duarte", ...
    "Solução 2 - PC Rodrigo", "Solução 3 - PC Rodrigo");
grid on;
hold off;

```

%% Solução 3 e Solução 4 (PC André)

```

valores = load("resultados_3_Aandre.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);

valores = load("resultados_4_Aandre.txt");
pos_finais4 = valores(:,1);
tempos4 = valores(:,4);

```

```

figure(1);
semilogy(pos_finais3,tempos3, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 4");
ylim([0 1e-3]);
hold on;
semilogy(pos_finais4,tempos4, "o-");
legend("Solução 3", "Solução 4");
grid on;
hold off;

```

%% Solução 3 e Solução 4 (PC Duarte)

***** SPEED RUN *****

```
valores = load("resultados_3_Duarte.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);
```

```
valores = load("resultados_4_Duarte.txt");
pos_finais4 = valores(:,1);
tempos4 = valores(:,4);
```

```
figure(1);
semilogy(pos_finais3,tempos3, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 4");
ylim([0 1e-3]);
hold on;
semilogy(pos_finais4,tempos4, "o-");
legend("Solução 3", "Solução 4");
grid on;
hold off;
```

%% Solução 3 e Solução 4 (PC Rodrigo)

```
valores = load("resultados_3_Rodrigo.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);
```

```
valores = load("resultados_4_Rodrigo.txt");
pos_finais4 = valores(:,1);
tempos4 = valores(:,4);
```

```
figure(1);
semilogy(pos_finais3,tempos3, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 4");
ylim([0 1e-3]);
hold on;
```

+

S P E E D R U N

+

```
semilogy(pos_finais4,tempos4,"o-");  
legend("Solução 3", "Solução 4");  
grid on;  
hold off;
```

%% Solução 3 e Solução 4 (3 PC's)

```
valores = load("resultados_3_Aandre.txt");
pos_finais3A = valores(:,1);
tempos3A = valores(:,4);
```

```
valores = load("resultados_4_Aandre.txt");
pos_finais4A = valores(:,1);
tempos4A = valores(:,4);
```

```
valores = load("resultados_3_Duarte.txt");
pos_finais3D = valores(:,1);
tempos3D = valores(:,4);
```

```
valores = load("resultados_4_Duarte.txt");
pos_finais4D = valores(:,1);
tempos4D = valores(:,4);
```

```
valores = load("resultados_3_Rodrigo.txt");
pos_finais3R = valores(:,1);
tempos3R = valores(:,4);
```

```
valores = load("resultados_4_Rodrigo.txt");
pos_finais4R = valores(:,1);
tempos4R = valores(:,4);
```

```
figure(1);  
semilogy(pos_finais3A,tempos3A, "+-");  
ylabel("log(Tempo (s))");  
xlabel("Posição Final");  
title("Comparação das soluções 3 e 4");  
ylim([0 1e-3]);  
hold on;
```

中 + + + + +

SPEED RUN

```
semilogy(pos_finais4A,tempos4A,"+-");
semilogy(pos_finais3D,tempos3D,"o-");
semilogy(pos_finais4D,tempos4D,"o-");
semilogy(pos_finais3R,tempos3R,".-");
semilogy(pos_finais4R,tempos4R,".-");

legend("Solução 3 - PC Andre", "Solução 4 - PC André", ...
       "Solução 3 - PC Duarte", "Solução 4 - PC Duarte", ...
       "Solução 3 - PC Rodrigo", "Solução 4 - PC Rodrigo");

grid on;
hold off;
```

%% Solução 3 e Solução 5 (PC André)

```
valores = load("resultados_3_Angelo.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);
```

```
valores = load("resultados_5_Aandre.txt");
pos_finais5 = valores(:,1);
tempos5 = valores(:, 4);
```

```
figure(1);
semilogy(pos_finais3,tempo3, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 5");
ylim([0 1e-3]);
hold on;
semilogy(pos_finais5,tempo5, "o-");
legend("Solução 3", "Solução 5");
grid on;
hold off;
```

%% Solução 3 e Solução 5 (PC Duarte)

```
valores = load("resultados_3_Duarte.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);
```

SPEED RUN

```

valores = load("resultados_5_Duarte.txt");
pos_finais5 = valores(:,1);
tempos5 = valores(:,4);

figure(1);
semilogy(pos_finais3,tempos3, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 5");
ylim([0 1e-3]);
hold on;
semilogy(pos_finais5,tempos5, "o-");
legend("Solução 3", "Solução 5");
grid on;
hold off;

```

%% Solução 3 e Solução 5 (PC Rodrigo)

```

valores = load("resultados_3_Rodrigo.txt");
pos_finais3 = valores(:,1);
tempos3 = valores(:,4);

```

```

valores = load("resultados_5_Rodrigo.txt");
pos_finais5 = valores(:,1);
tempos5 = valores(:,4);

```

```

figure(1);
semilogy(pos_finais3,tempos3, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 5");
ylim([0 1e-3]);
hold on;
semilogy(pos_finais5,tempos5, "o-");
legend("Solução 3", "Solução 5");
grid on;
hold off;

```

S P E E D R U N

%% Solução 3 e Solução 5 (3 PC's)

```
valores = load("resultados_3_Aandre.txt");
pos_finais3A = valores(:,1);
tempos3A = valores(:,4);
```

```
valores = load("resultados_5_Aandre.txt");
pos_finais5A = valores(:,1);
tempos5A = valores(:,4);
```

```
valores = load("resultados_3_Duarte.txt");
pos_finais3D = valores(:,1);
tempos3D = valores(:,4);
```

```
valores = load("resultados_5_Duarte.txt");
pos_finais5D = valores(:,1);
tempos5D = valores(:,4);
```

```
valores = load("resultados_3_Rodrigo.txt");
pos_finais3R = valores(:,1);
tempos3R = valores(:,4);
```

```
valores = load("resultados_5_Rodrigo.txt");
pos_finais5R = valores(:,1);
tempos5R = valores(:,4);
```

```
figure(1);
semilogy(pos_finais3A,tempos3A, "+-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação das soluções 3 e 5");
ylim([0 1e-3]);
hold on;
semilogy(pos_finais5A,tempos5A, "+-");
semilogy(pos_finais3D,tempos3D, "o-");
semilogy(pos_finais5D,tempos5D, "o-");
semilogy(pos_finais3R,tempos3R, ".-");
```

S P E E D R U N

```

semilogy(pos_finais5R,tempo5R,".-");
legend("Solução 3 - PC Andre", "Solução 5 - PC André", ...
    "Solução 3 - PC Duarte", "Solução 5 - PC Duarte", ...
    "Solução 3 - PC Rodrigo", "Solução 5 - PC Rodrigo");

grid on;
hold off;

```

%% 5 Soluções (3 PC's)

```

valores = load("resultados_1_Aandre.txt");
pos_finais1A = valores(:,1);
tempo1A = valores(:,4);

```

```

valores = load("resultados_2_Aandre.txt");
pos_finais2A = valores(:,1);
tempo2A = valores(:,4);

```

```

valores = load("resultados_3_Aandre.txt");
pos_finais3A = valores(:,1);
tempo3A = valores(:,4);

```

```

valores = load("resultados_4_Aandre.txt");
pos_finais4A = valores(:,1);
tempo4A = valores(:,4);

```

```

valores = load("resultados_5_Aandre.txt");
pos_finais5A = valores(:,1);
tempo5A = valores(:,4);

```

```

valores = load("resultados_1_Duarte.txt");
pos_finais1D = valores(:,1);
tempo1D = valores(:,4);

```

```

valores = load("resultados_2_Duarte.txt");
pos_finais2D = valores(:,1);
tempo2D = valores(:,4);

```

```

valores = load("resultados_3_Duarte.txt");

```

S P E E D R U N

```

pos_finais3D = valores(:,1);
tempos3D = valores(:,4);

valores = load("resultados_4_Duarte.txt");
pos_finais4D = valores(:,1);
tempos4D = valores(:,4);

valores = load("resultados_5_Duarte.txt");
pos_finais5D = valores(:,1);
tempos5D = valores(:,4);

valores = load("resultados_1_Rodrigo.txt");
pos_finais1R = valores(:,1);
tempos1R = valores(:,4);

valores = load("resultados_2_Rodrigo.txt");
pos_finais2R = valores(:,1);
tempos2R = valores(:,4);

valores = load("resultados_3_Rodrigo.txt");
pos_finais3R = valores(:,1);
tempos3R = valores(:,4);

valores = load("resultados_4_Rodrigo.txt");
pos_finais4R = valores(:,1);
tempos4R = valores(:,4);

valores = load("resultados_5_Rodrigo.txt");
pos_finais5R = valores(:,1);
tempos5R = valores(:,4);

figure(1);
semilogy(pos_finais1A,tempos1A, "o-");
ylabel("log(Tempo (s))");
xlabel("Posição Final");
title("Comparação de todas as soluções nos 3 PC's");
hold on;
semilogy(pos_finais2A,tempos2A, "o-");

* * * * *
      S P E E D   R U N
* * * * *

```

```

semilogy(pos_finais3A,tempo3A,"o-");
semilogy(pos_finais4A,tempo4A,"o-");
semilogy(pos_finais5A,tempo5A,"o-");
semilogy(pos_finais1D,tempo1D,"+-");
semilogy(pos_finais2D,tempo2D,"+-");
semilogy(pos_finais3D,tempo3D,"+-");
semilogy(pos_finais4D,tempo4D,"+-");
semilogy(pos_finais5D,tempo5D,"+-");
semilogy(pos_finais1R,tempo1R,".-");
semilogy(pos_finais2R,tempo2R,".-");
semilogy(pos_finais3R,tempo3R,".-");
semilogy(pos_finais4R,tempo4R,".-");
semilogy(pos_finais5R,tempo5R,".-");

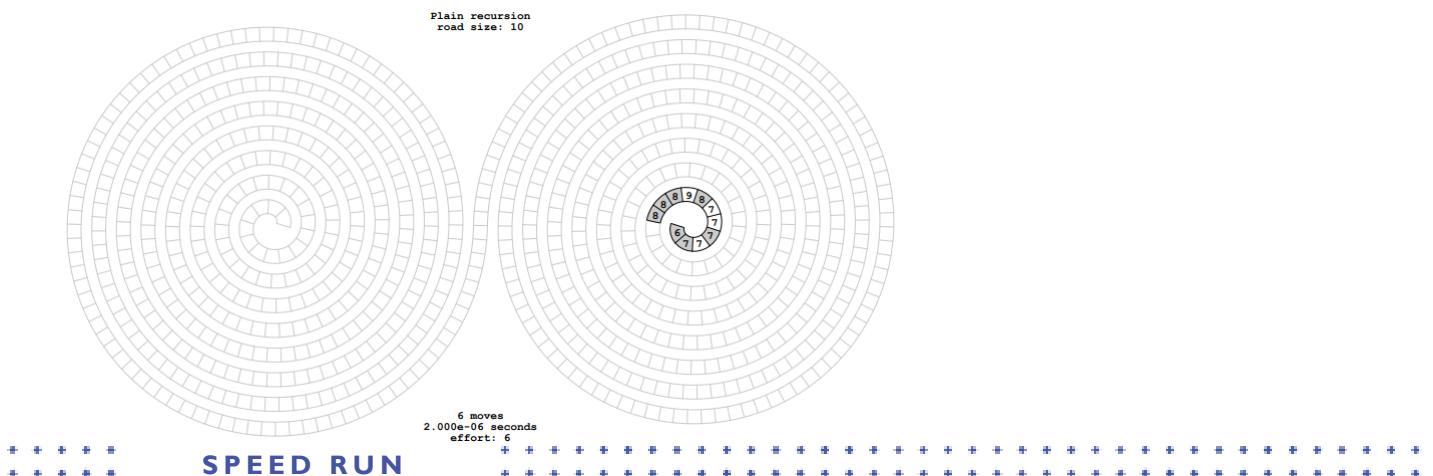
legend("Solução 1 - PC Andre", "Solução 2 - PC André", ...
      "Solução 3 - PC André", "Solução 4 - PC Duarte", ...
      "Solução 5 - PC André", "Solução 1 - PC Duarte", ...
      "Solução 2 - PC Duarte", "Solução 3 - PC Duarte", ...
      "Solução 4 - PC Duarte", "Solução 5 - PC Duarte",...
      "Solução 1 - PC Rodrigo", "Solução 2 - PC Rodrigo", ...
      "Solução 3 - PC Rodrigo", "Solução 4 - PC Rodrigo", ...
      "Solução 5 - PC Rodrigo");

grid on;
hold off;

```

Exemplo de Imagens das Soluções

Solução 1 (NºMec 107359)



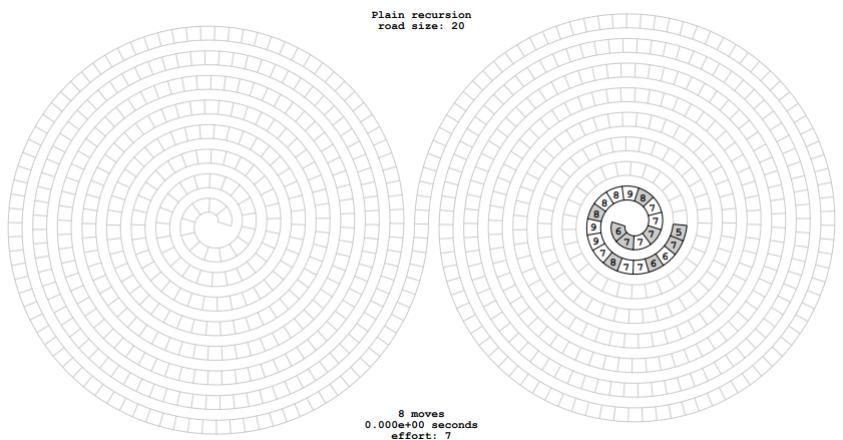


Figura 21 | Percurso pertencente à posição final 20

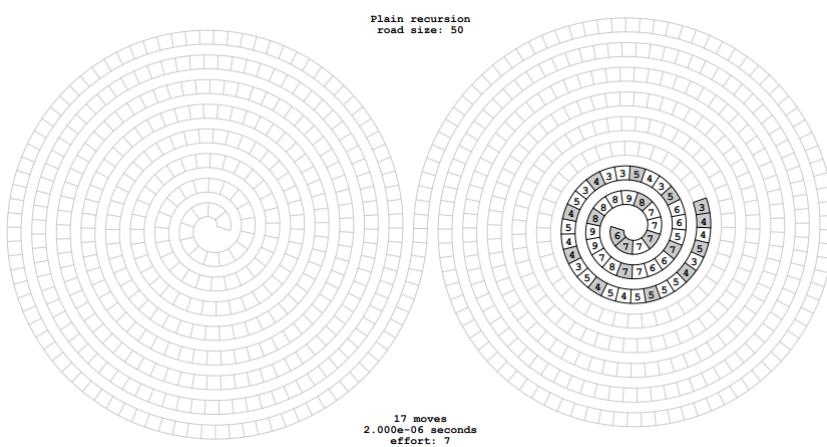


Figura 22 | Percurso pertencente à posição final 50

Solução 2 (NºMec 107637)

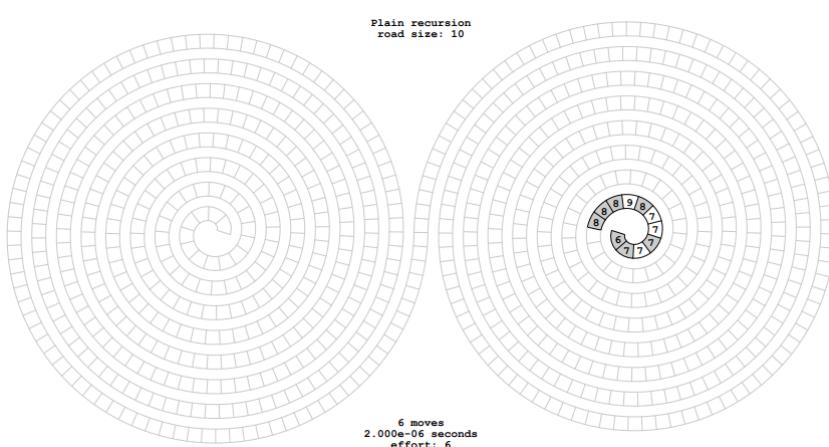


Figura 23 | Percurso pertencente à posição final 10

SPEED RUN

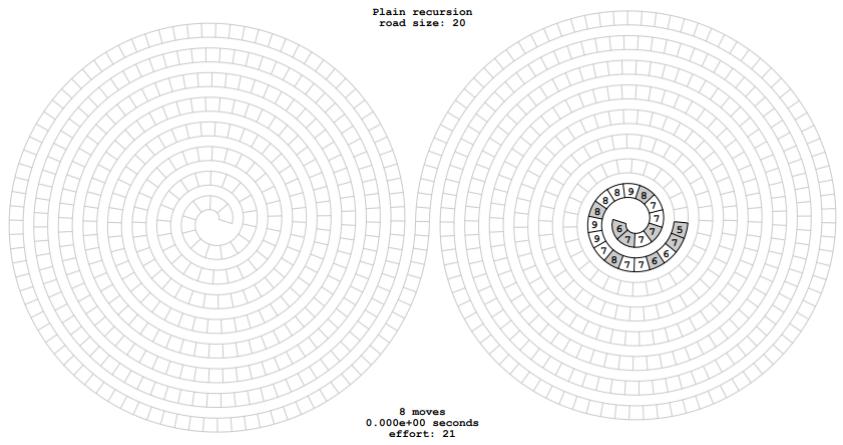


Figura 24 | Percurso pertencente à posição final 20

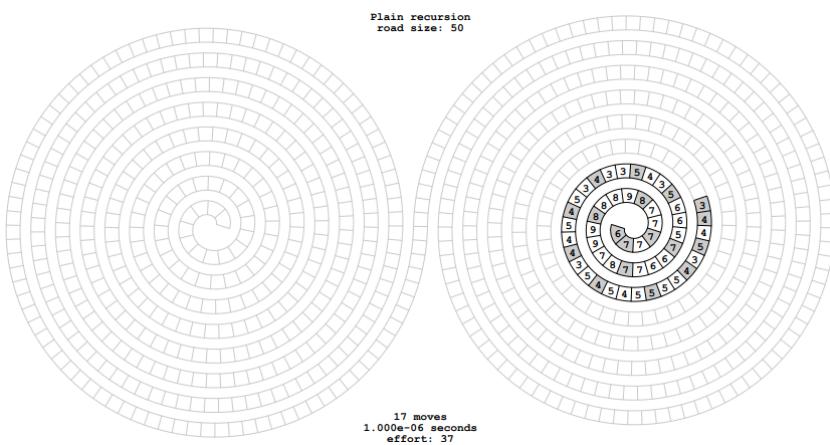


Figura 25 | Percurso pertencente à posição final 50

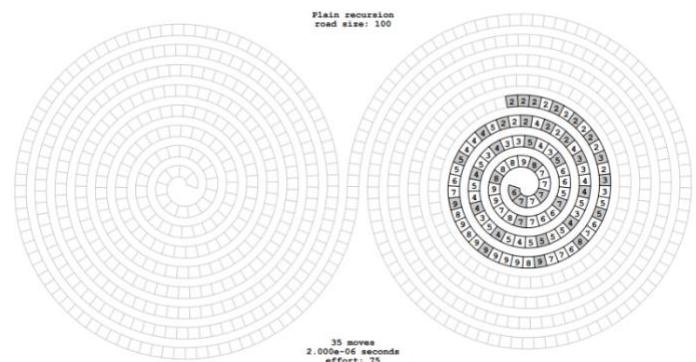


Figura 26 | Percurso pertencente à posição final 100

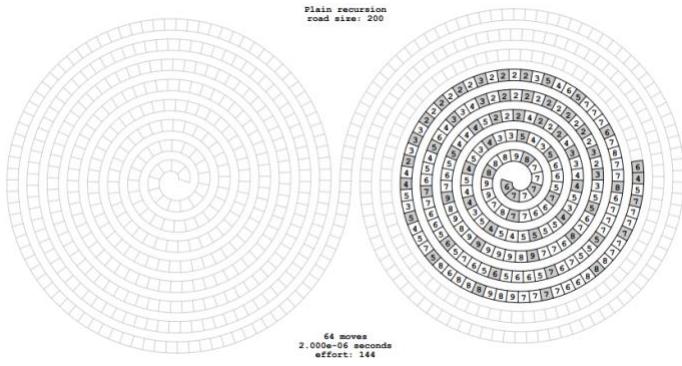


Figura 27 | Percurso pertencente à posição final 200

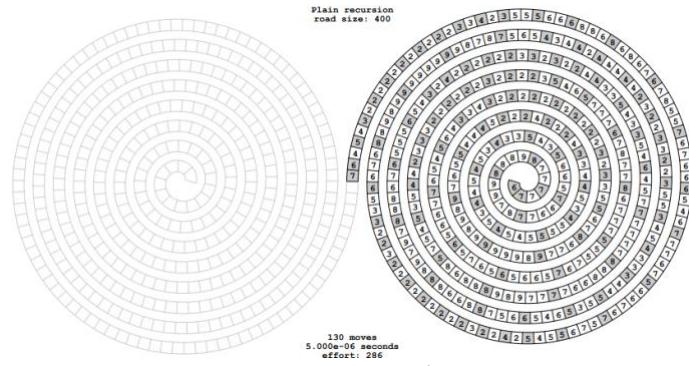


Figura 28 | Percurso pertencente à posição final 400

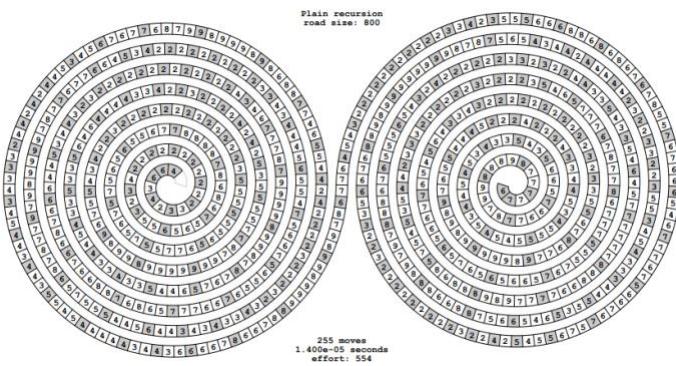


Figura 29 | Percurso pertencente à posição final 800

Solução 3 (NºMec 107634)

SPEED RUN

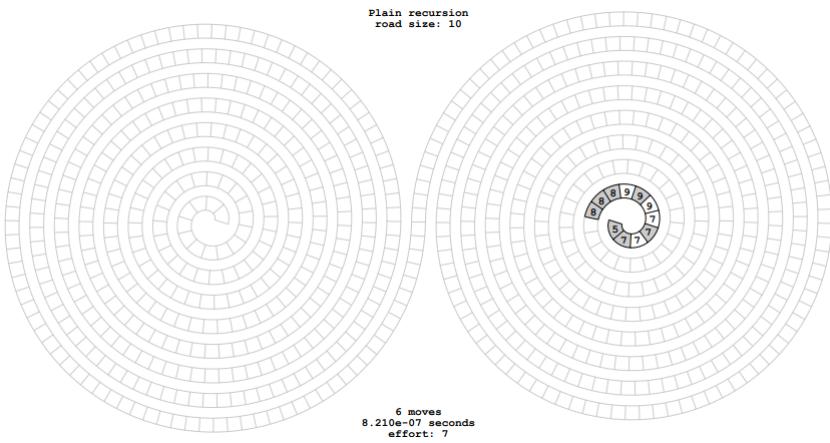


Figura 30 | Percurso pertencente à posição final 10

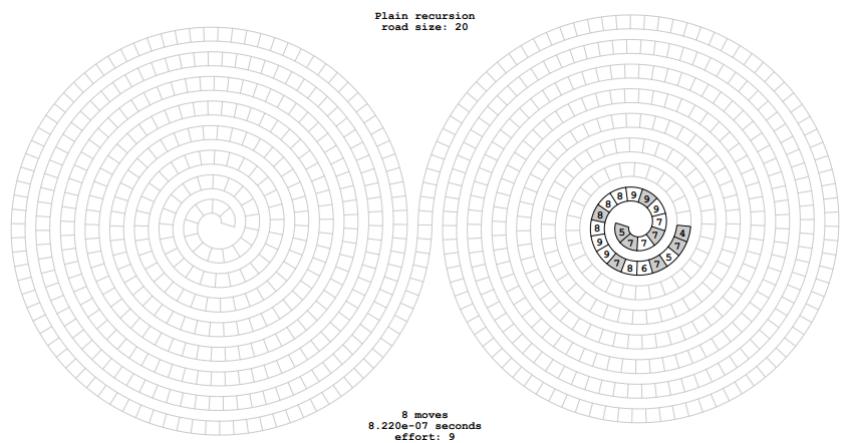


Figura 31 | Percurso pertencente à posição final 20

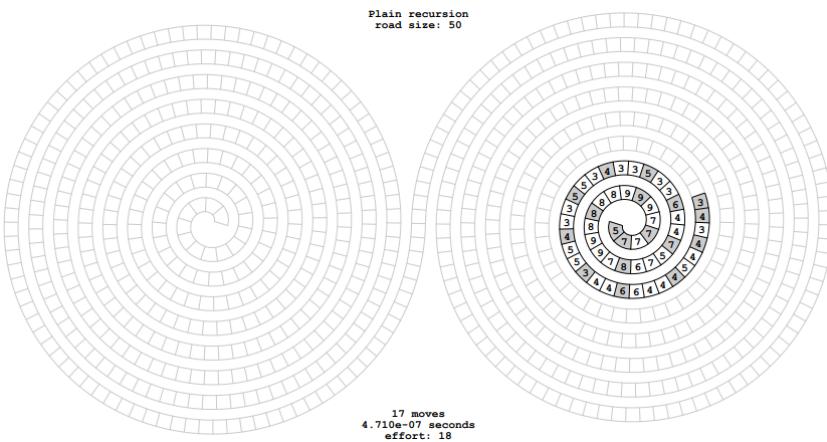


Figura 32 | Percurso pertencente à posição final 50

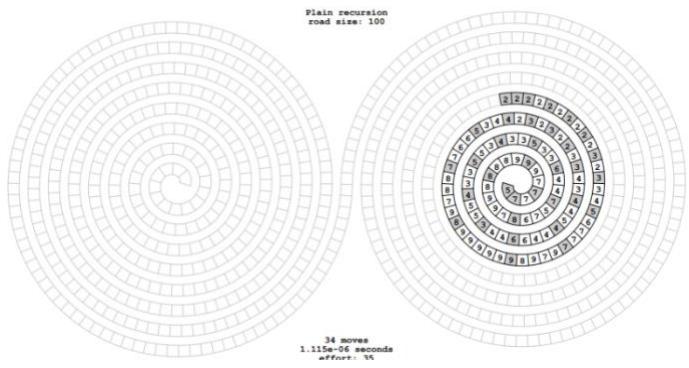


Figura 33 | Percurso pertencente à posição final 100

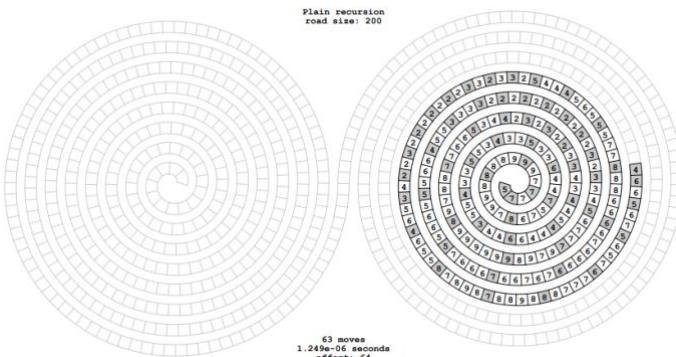


Figura 34 | Percurso pertencente à posição final 200

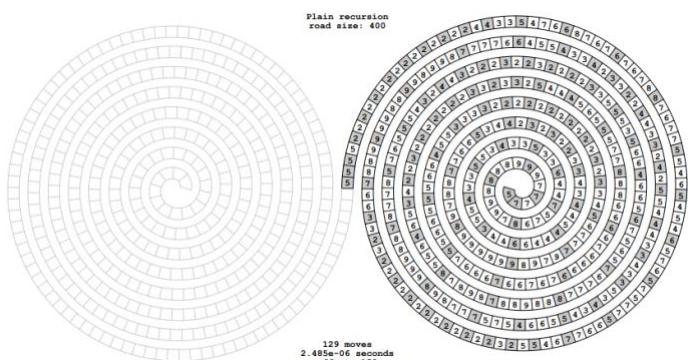


Figura 35 | Percurso pertencente à posição final 400

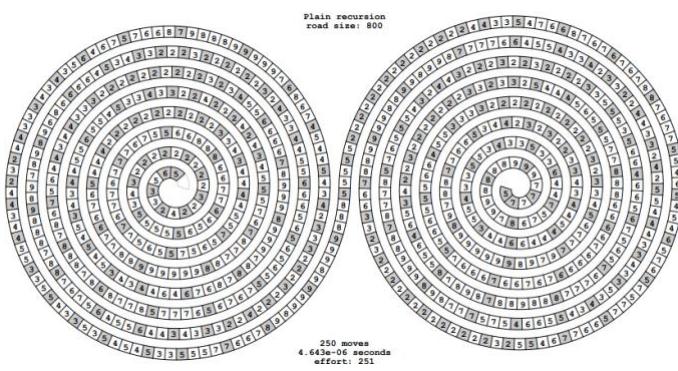


Figura 36 | Percurso pertencente à posição final 800

Solução 4 (NºMec 107359)

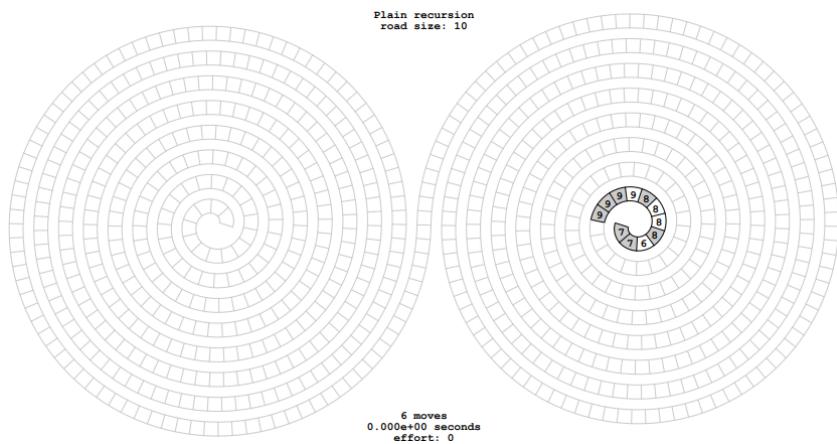


Figura 37 | Percurso pertencente à posição final 10

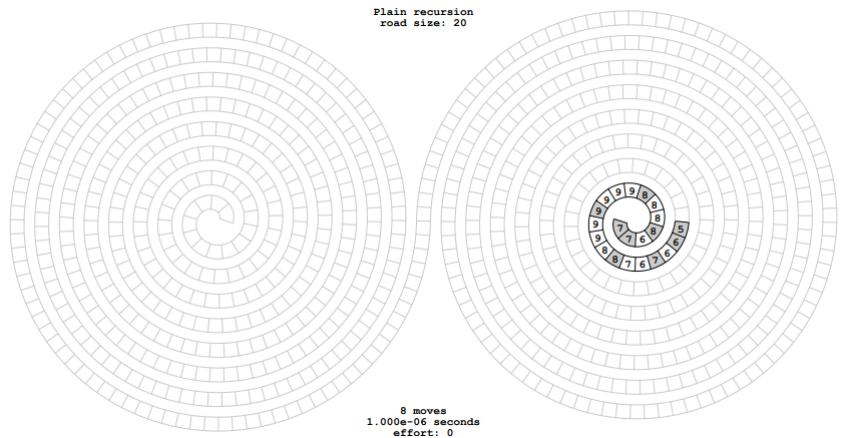


Figura 38 | Percurso pertencente à posição final 20

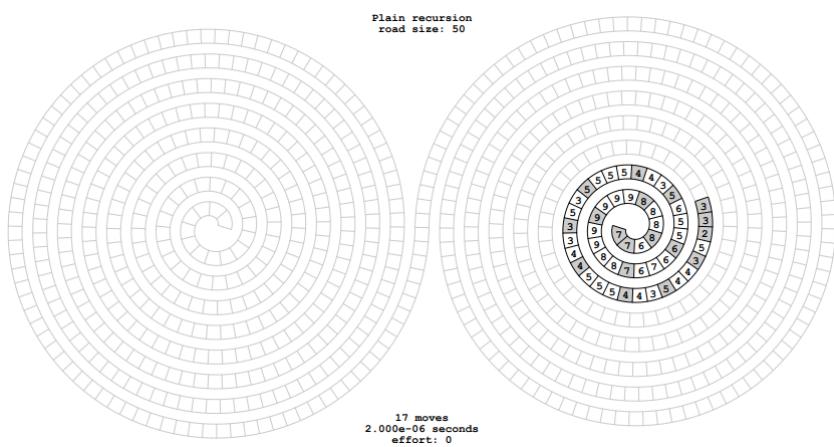


Figura 39 | Percurso pertencente à posição final 50

中華書局影印

SPEED RUN

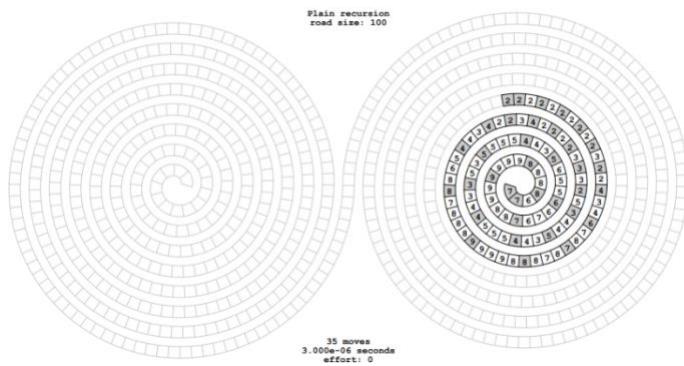


Figura 40 | Percurso pertencente à posição final 100

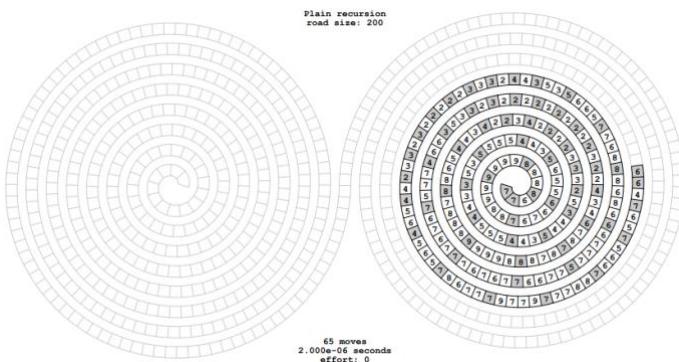


Figura 41 | Percurso pertencente à posição final 200

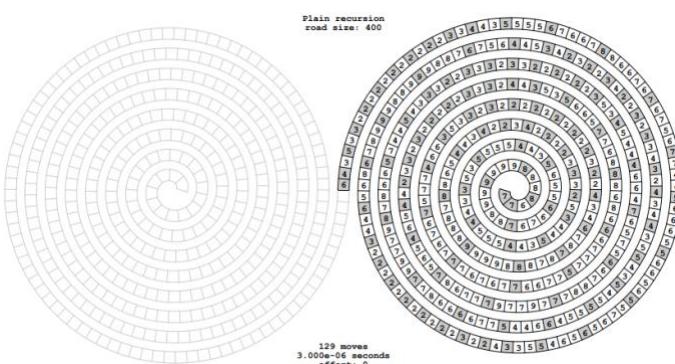


Figura 42 | Percurso pertencente à posição final 400

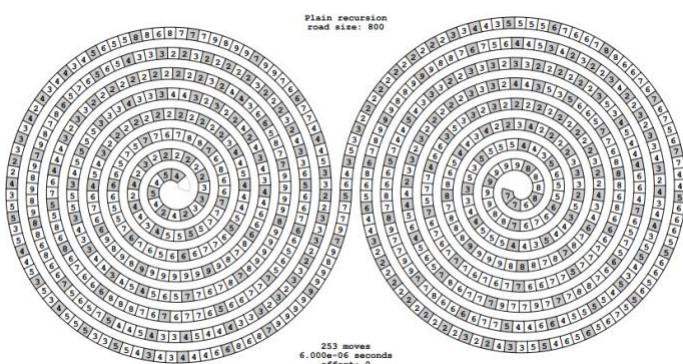


Figura 43 | Percurso pertencente à posição final 800

中 中 中 中 中

SPEED RUN

Solução 5 (NºMec 107637)

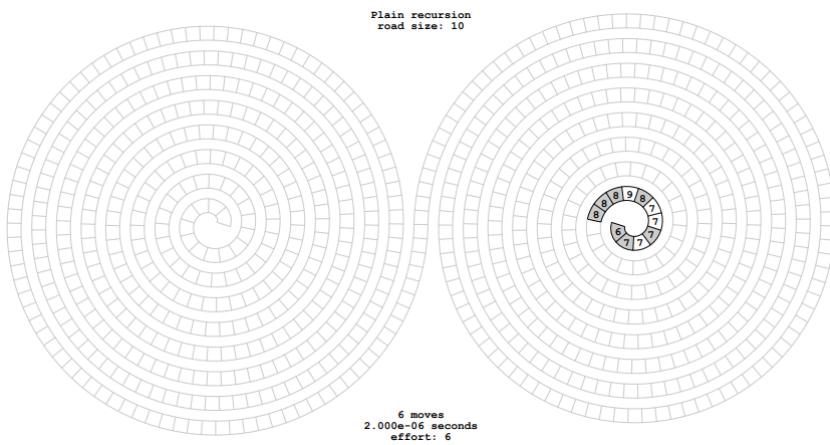


Figura 44 | Percurso pertencente à posição final 10

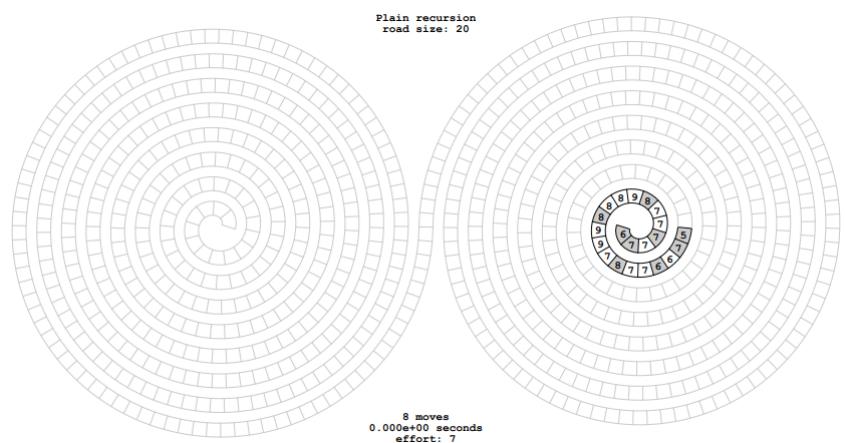


Figura 45 | Percurso pertencente à posição final 20

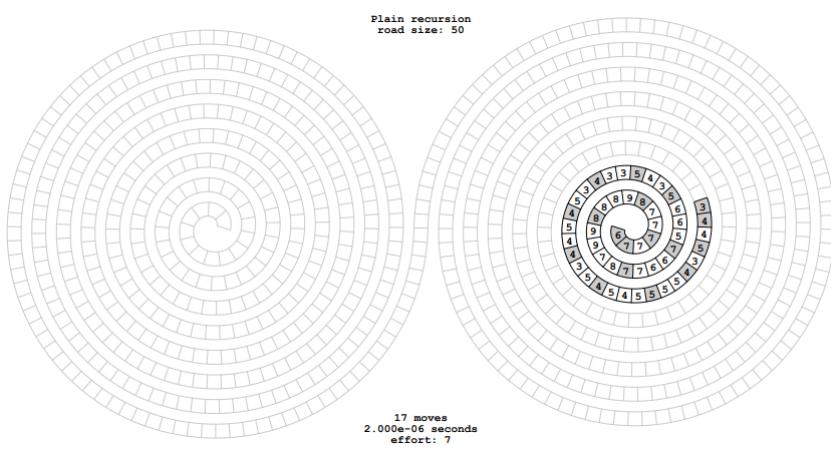


Figura 46 | Percurso pertencente à posição final 50

中 中 中 中 中

SPEED RUN

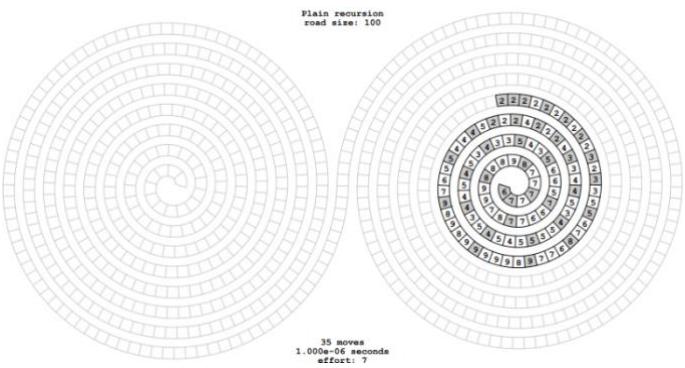


Figura 47 | Percurso pertencente à posição final 100

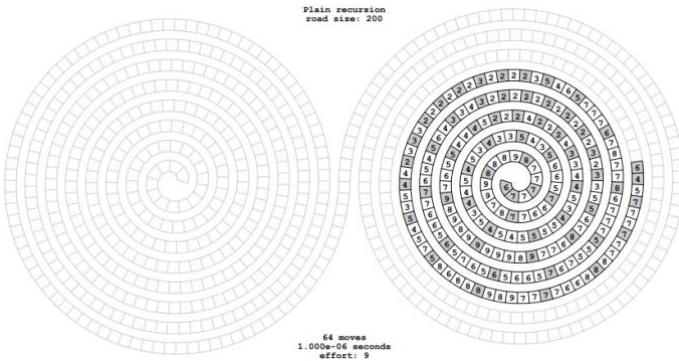


Figura 48 | Percurso pertencente à posição final 200

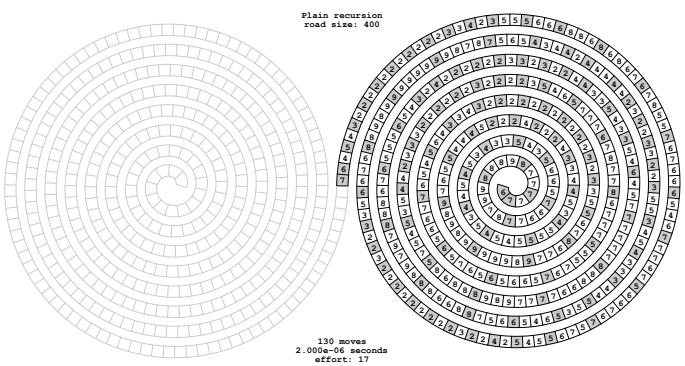


Figura 49 | Percurso pertencente à posição final 400

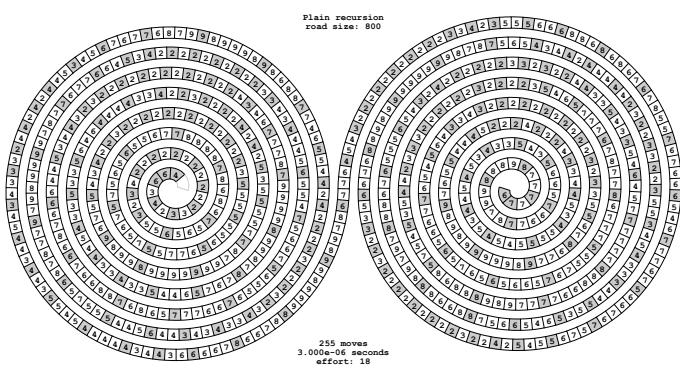


Figura 50 | Percurso pertencente à posição final 800

♦ ♦ ♦ ♦ ♦

SPEED RUN

♦ ♦ ♦ ♦ ♦
♦ ♦ ♦ ♦ ♦