



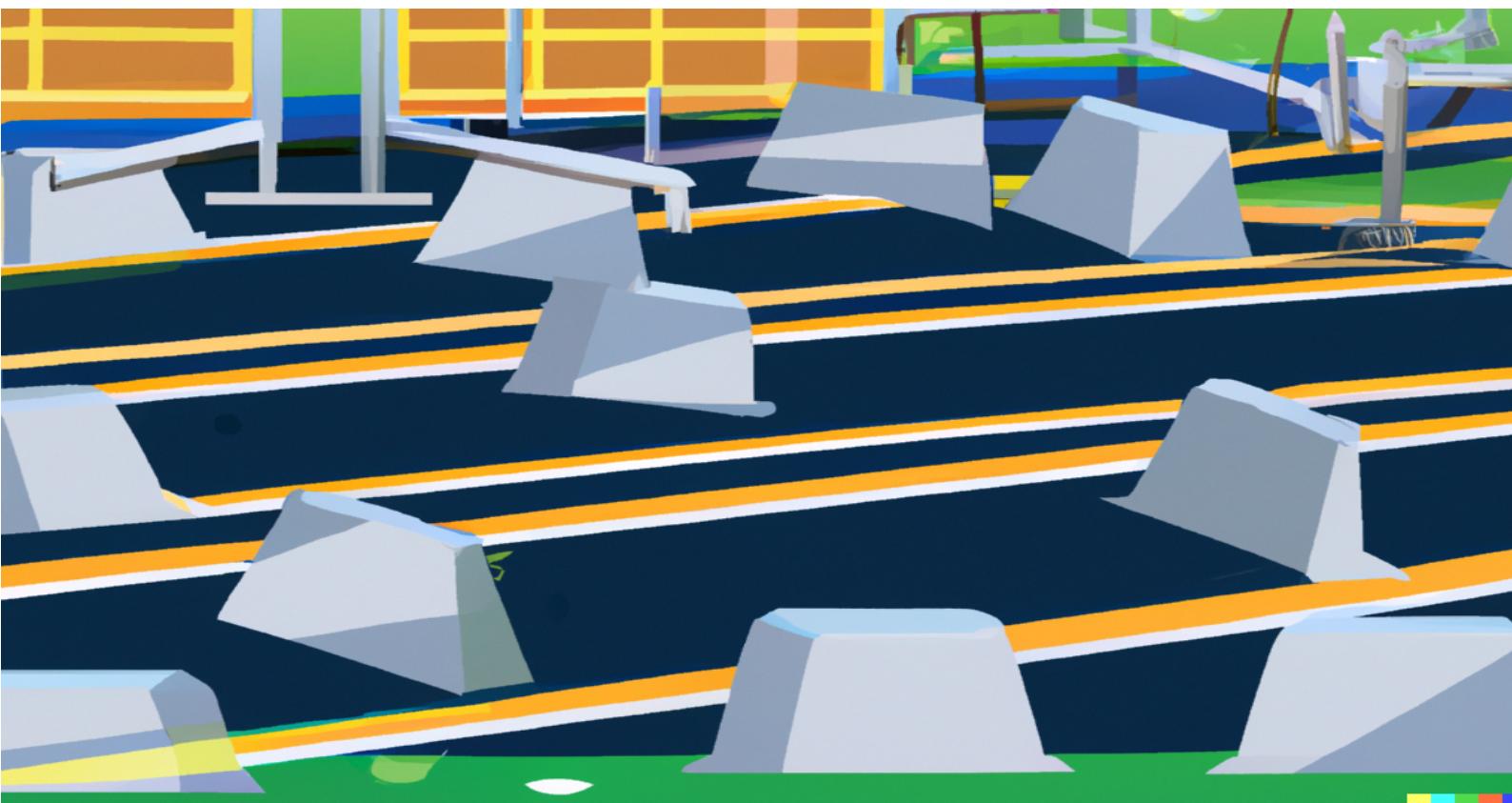
Speed Run

Algoritmos e Estruturas de Dados
3 de Dezembro 2022

Maria Rafaela Alves Abrunhosa, 107658-50%
Matilde Moital Portugal Sampaio Teixeira, 108193-50%

Índice

INTRODUÇÃO	2
METODOLOGIA	3
CONTEXTUALIZAÇÃO DO PROBLEMA	3
CÓDIGO PROFESSOR.....	4
CÓDIGO OTIMIZADO.....	6
TRAVAGEM	8
RESULTADOS OBTIDOS	11
OTIMIZAÇÕES.....	11
GRÁFICOS.....	11
CONCLUSÃO	14
ANEXOS	15
CÓDIGO C.....	15
<i>Solução Otimizada</i>	15
<i>Método Travagem.</i>	20
CÓDIGO MATLAB	25
<i>PDFs Criados</i>	29
BIBLIOGRAFIA	31



Introdução

Desde a sua infância que o ser humano foi sempre instruído a brincar, a fazer raciocínios baseados em lógica para conseguir alcançar o final. Estes poderiam variar, mas tinham um final único, descobrir com o mínimo número de tentativas como chegar ao ponto final com o mínimo de tempo.

O termo “*speed run*” é cunhado pelo *Dictionary.com*, como um “(...) jogo em que o objetivo é terminar (...) ou atingir outro objetivo de jogo, o mais rápido possível” (DICTIONARY.COM, 2022). Ao ser utilizado *speed run*, algumas sequências de caminho serão omitidas de modo a que o tempo seja o mais diminuto possível.

O intuito deste trabalho é assim a otimização de um código, previamente fornecido, para um jogo de *speed run*, tentando alcançar a última posição no mínimo tempo possível. Neste jogo, todas as peças do tabuleiro encontram-se previamente marcados com números aleatórios (dependentes do número mecanográfico inserido), e o objetivo do utilizador será chegar à última posição no menor tempo possível. Porém, o utilizador só terá três hipóteses de velocidade. Além disto, para poder passar com velocidade ‘x’, terá que satisfazer algumas condições ou não conseguirá avançar com velocidade ‘x’.

Para a otimização deste problema, foram implantadas algumas alterações de modo a que este se torne mais rápido que o código previamente fornecido.

Neste relatório serão explicados os vários processos realizados para a otimização deste e discutidas as vantagens e desvantagens de cada um, bem como os resultados obtidos.

Metodologia

Contextualização do Problema

Como apresentado previamente, o objetivo do projeto é reduzir significativamente o tempo de execução do código em C fornecido, otimizando-o, tendo, por isso, o nome de *speed_run*. Sucintamente, pretende-se chegar ao último troço de uma estrada com o menor número de movimentos possíveis, sendo que existem velocidades máximas às quais não podemos exceder, podendo apenas aumentar a velocidade em 1, mantê-la ou diminuí-la em 1.

Uma estrada é dividida em vários segmentos com aproximadamente o mesmo comprimento, onde cada segmento tem o seu limite de velocidade, no mínimo 1 e no máximo 9. Este limite de velocidade de cada casa é gerado aleatoriamente sendo que se introduzirmos o número mecanográfico do aluno como argumento de entrada, existem pequenas alterações nesta escolha aleatória, isto é, cada número mecanográfico tem-lhe associado um conjunto de limites de velocidade em cada casa diferente. Quando nos referimos a velocidade (*speed*), referimo-nos ao número de casas que podemos avançar num único movimento. A cada movimento, o carro pode percorrer uma das três hipóteses para a velocidade: acelerar (*speed+1*), manter (*speed*) ou abrandar (*speed-1*).

Na primeira posição, o carro inicia o seu percurso com uma velocidade de 0 e ao chegar ao final da estrada, deve atingir o último troço com uma velocidade de 1 para que a partir daqui possa reduzir a sua velocidade para 0 e então parar.

Código Professor

Foi-nos fornecido uma resolução pouco eficiente para este problema de *speed_run*, uma implementação recursiva que demora bastante tempo a executar. Na *solution_1_recursion*, são calculadas todas as combinações possíveis das diferentes velocidades que o carro pode adquirir ao percorrer a estrada, cumprindo todas as regras previamente apresentadas. Estas combinações, serão comparadas entre si de forma a descobrir qual delas contém o menor número de movimentos possíveis para percorrer todo o caminho.

A função *solution_1_recursion* admite como argumentos de entrada o número de movimentos, a posição em que o carro se encontra, a velocidade com que chegou a posição atual e a posição final que equivale ao fim da estrada.

Podemos justificar o comportamento da função como uma procura em profundidade (Depth-First Algorithm), ou seja, temos uma árvore onde cada ramo diverge para cada uma das 3 velocidades possíveis. A pesquisa começa, no entanto, pelo ramo mais demorado onde, por todo o seu percurso, a velocidade limita-se a 1, o carro anda de uma em uma casa (ver linha azul, figura nº1). Só depois de testar este caso é que começa a testar a opção de aumentar a velocidade para que o carro possa avançar o maior número de casas possíveis num único movimento, obedecendo às regras. Por fim, depois de todas as combinações possíveis terem sido testadas, estas são submetidas a um processo de comparação onde se escolhe o percurso que obteve o menor número de movimentos.

Esta solução é, deste modo, uma solução ineficaz uma vez que perde tempo a calcular combinações à partida inúteis, como é o caso de se avançar uma casa de cada vez.

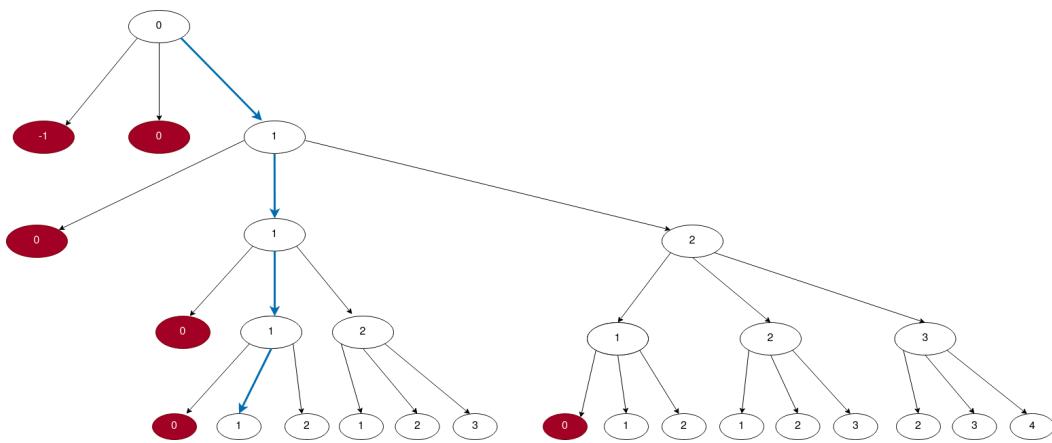


Figura nº1 - primeira pesquisa da “solution_1_recursion”

Código Otimizado

Com o propósito de se ultrapassar as dificuldades do código da função *solution_1_recursion*, removendo os casos improváveis, criou-se a função *solution_1_recursion_optimized*. Apesar de ser uma versão aprimorada, não deixa de ter um raciocínio idêntico ao do algoritmo anterior, implementando, igualmente, a recursividade.

Os argumentos de entrada são, portanto, os mesmos, nomeadamente o número de movimentos, a posição atual, a velocidade com que chegou à posição atual e a posição final.

Da mesma forma que o comportamento da *solution_1_recursion* se fundamenta numa pesquisa em profundidade, o da *solution_1_recursion_optimized* também a institui. Da árvore que dá origem a três nós, três velocidades possíveis, neste algoritmo optamos por pesquisar no ramo em que nos oferece sempre a maior velocidade possível, salvando tempo (ver linha verde, figura nº2). Assim, só se esta velocidade exceder o limite máximo ou ultrapassar a posição final, entre outras regras, é que a pesquisa retorna aos nós anteriores para poder respeitar as normas predefinidas.

Para além desta alteração, faz-se uma nova comparação na qual se verifica onde é que o carro consegue chegar com o mesmo número de nós saltados da árvore, isto é, com três movimentos (1,2,3) e com outros três movimentos (1,2,2) posso chegar às posições seis ou cinco e o algoritmo opta pelo caminho onde com menos movimentos permite a que o carro chegue mais longe (ver setas amarelas, figura nº2), isto se cumprir todas as outras diretrizes.

Além disso, foi implementado uma linha de código para que quando chegar a um nó da árvore este irá ignorar qualquer outro nó da árvore do lado que não foi percorrido até ali, assim salvando memória, tempo e espaço.

Conseguimos, assim, com diferentes e pequenas alterações, melhorar significativamente o tempo de execução e consequentemente a eficiência apesar de a complexidade computacional ser idêntica à do algoritmo anterior visto que, para o pior dos casos, terá de testar todas as combinações possíveis.

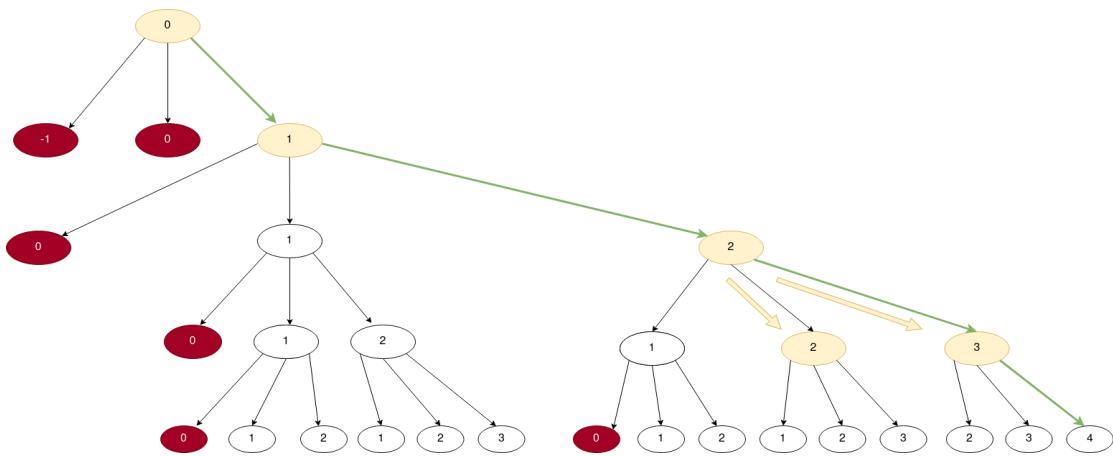


Figura nº2 - primeira pesquisa da “solution_1_recursion_optimized”

Travagem

O método Travagem consiste numa implementação recorrendo a uma função recursiva booleana, ***testar_travagem***, que recebe os parâmetros ***int new_speed, int new_position, int move_number, int final_position*** e se a posição mais a ***new_speed*** for maior que a final position ele retorna ***false***.

Caso isso não aconteça, este irá entrar num ***for***, que inicia com ***pos*** igual à posição, até que a posição mais a velocidade a testar (***new_speed***) seja menor que a ***pos***. Ou seja, este vai percorrer todas as posições de ***position*** (posição inicial) até à posição, mas a nova velocidade, e verificar que, todas as posições entre estas as duas, têm o limite de velocidade menor do que ***new_speed***. Se alguma delas der ***false***, a função para a sua execução, caso nenhuma tenha limite de velocidade maior, ela vai continuar.

Se não for retornado o valor ***false***, este irá verificar se se trata da velocidade 1 (significando que não se pode diminuir a velocidade) ele irá retornar logo ***true***, isto pode acontecer se estiver na verificação da última posição ou se entrar logo com velocidade 1. Caso não tenha ***new_speed=1***, este irá entrar no ***else*** onde irá incrementar o ***positions*** para ***positions+newspeed***, de em seguida, executar novamente a função desta vez com ***newspeed-1***. Assim, com o uso da recursiva, será verificado se é possível executar o movimento (vendo pela distância de travagem total, até chegar a ***newspeed=1***).

Deste modo, iremos verificar se ele consegue fazer a passagem sem chegar à posição final com velocidade maior que 1. O ***count*** foi implementado quando é realizado o ***for***, sendo este fator decidido ser implementado nesta parte do código, devido a ser a zona que apresenta maior “esforço” de execução.

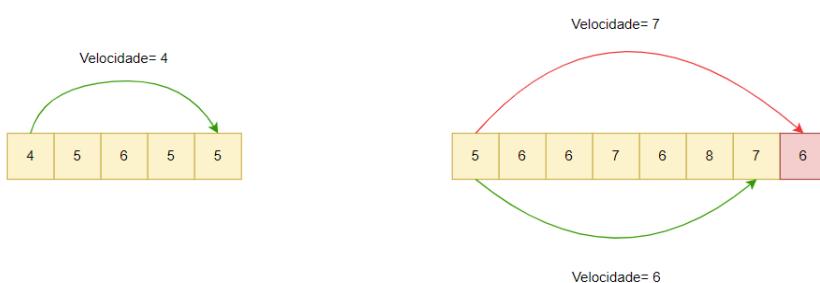
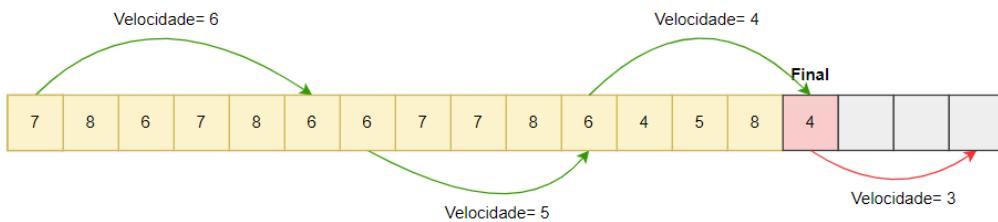
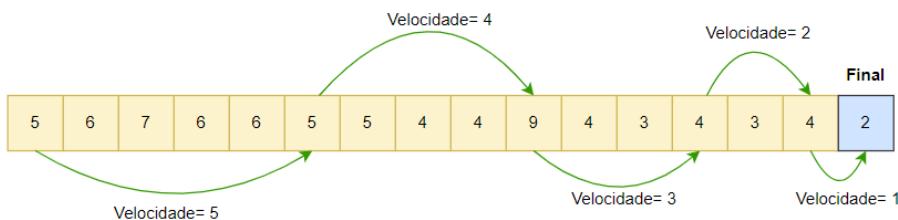


Figura nº3 - possibilidade e não possibilidade de transição de posição



Distância de travagem para velocidade=6 é $6+5+4+3+2+1=21$ ao seja tem de passar 22 posições, não conseguindo chegar à posição final no caso acima com velocidade 1.



Quando a velocidade diminui para 5 a distância de travagem fica $5+4+3+2+1=15$ ao seja tem de passar 16 posições, conseguindo chegar à posição final no caso acima com velocidade 1.

Figura nº4 - exemplo da aplicação da execução do método de travagem

Posteriormente esta função será verificada, quanto ao seu valor booleano, na função **solution_2_travagem**. Esta função funciona com um ciclo *for* dentro de um *while*. O ciclo *while* irá correr enquanto não chegar à última posição. O ciclo foi iniciado baseado no código utilizado para a solução otimizada (Código Otimizado), onde este será inicializado com **new_speed=speed+1**, até **new_speed=>newspeed-1**, decrementando-o.

Dentro deste ciclo *for* irá ser verificada se a velocidade (**new_speed**) é maior ou igual a 1 e se esta é menor ou igual que o limite de velocidade estabelecido nessa posição (**max_road_speed**). Caso isso aconteça, será verificado se a função mencionada anteriormente (**testar_travagem**) tem valor *true*, e se acontecer, será gravado na posição associada ao valor de **move_number** a velocidade (**new_speed**).

Em seguida, o número de movimentos será igual ao número de movimentos+1 (só incrementada na linha seguinte devido aos sinais estarem posicionados após o nome da variável), e após isso, a velocidade (**speed**) vai ser atualizada para o valor de **new_speed**, a posição será incrementada o número a que está associada a **new_speed**, ou seja anda para a frente o número de posições que a velocidade deixa avançar (se **new_speed** for igual a 4 a posição será incrementada mais 4), no final destes, irá ser executado um *break* do ciclo *for*, no entanto, o código continuará a executar o ciclo

while, enquanto a posição for diferente da posição final. Caso não cumpra algumas destas condições o ciclo *for* continuará a correr até que isto seja possível, dentro dos limites deste.

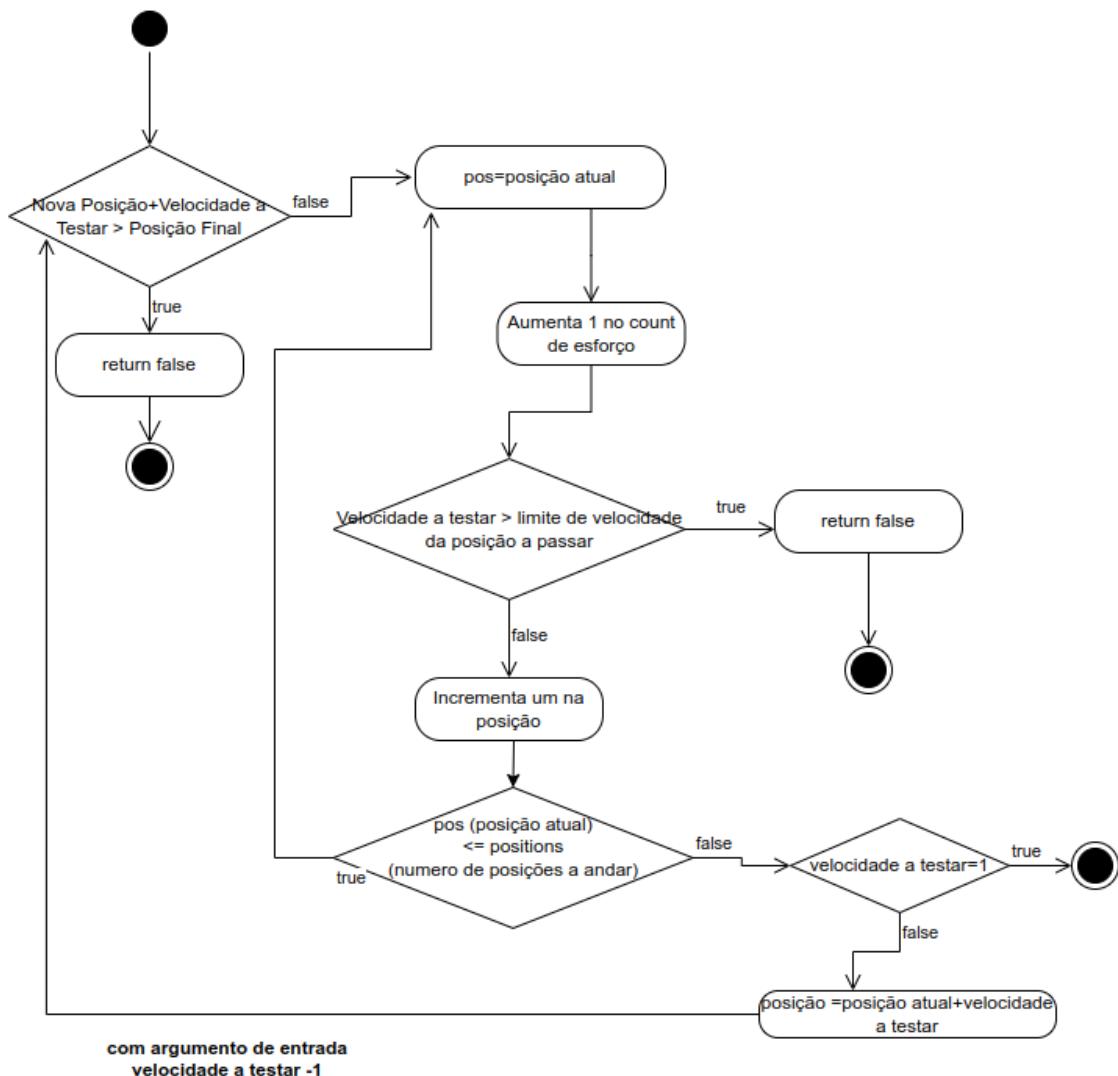


Figura nº5 - diagrama exemplificativo da aplicação da função recursiva *testar_travagem*

Resultados obtidos

Otimizações

Os últimos dois algoritmos explicados são o resultado final de várias otimizações feitas ao longo do seu desenvolvimento. Realçamos a escolha de funções recursivas e a da utilização “do que já foi calculado anteriormente”, isto é, no caso do método da travagem, reduzir o esforço aproveitando o que já foi calculado para as posições anteriores (se vou calcular para a posição 100, aproveito o que já foi calculado para a posição 99, 98, 97, ...).

A travagem vai assim verificar se cada movimento é exequível até ao final. Deste modo este irá confirmar se consegue chegar à posição final (*posição + velocidade*) com velocidade 0 para não ultrapassar a posição final.

No caso da *solution_1_recursion_optimized*, apostamos, logo no início da pesquisa, na remoção dos casos improváveis cujo cálculo dos mesmos será, à partida, desnecessário.

Gráficos

Em cada um dos algoritmos apresentados, será mostrada a performance deste e avaliado o tempo de execução para cada, consoante o tempo de execução. Esta avaliação será executada para todos os tipos de argumentos de entrada, incluindo argumento de entrada nulo e ambos os nossos números mecanográficos.

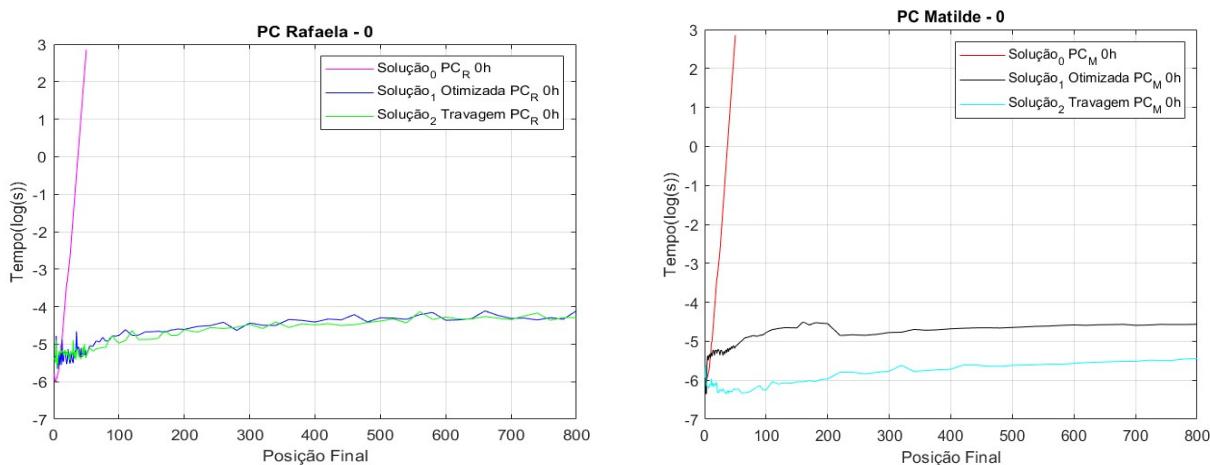


Figura nº7 -Gráfico da performance de PC R e M, sob as 3 soluções, com argumento de entrada nulo.

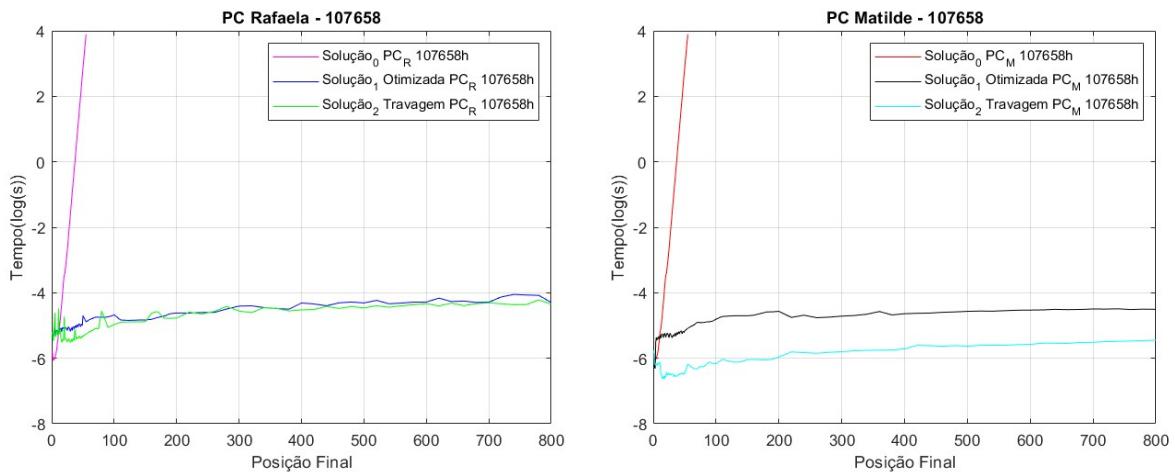


Figura n°8 -Gráfico da performance de PC R e M, sob as 3 soluções, com argumento de entrada 107658

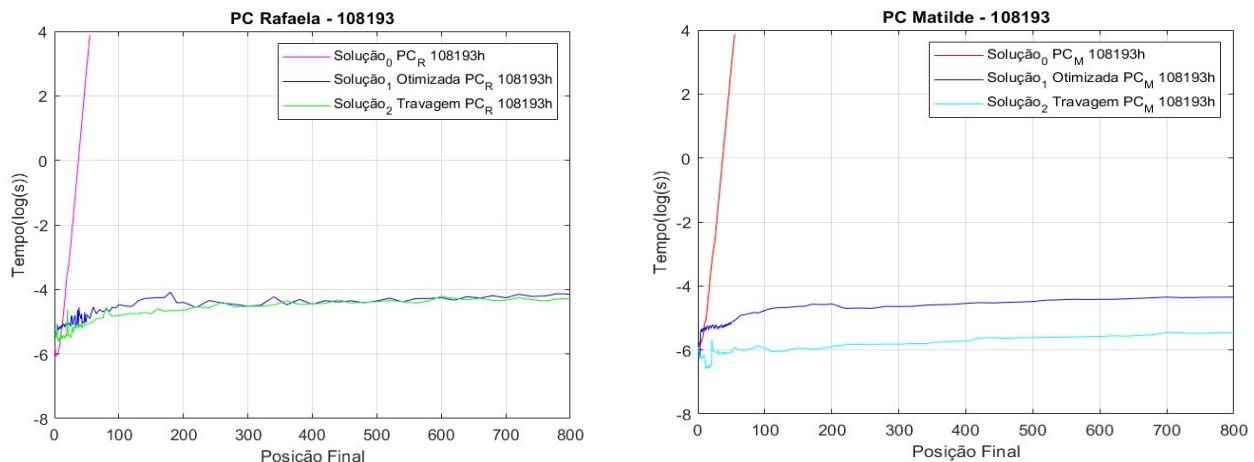


Figura n°9 -Gráfico da performance de PC R e M, sob as 3 soluções, com argumento de entrada 108193

Deste modo, após o estudo destes gráficos é possível inferir algumas conclusões. Enquanto que a Solução 0, solução previamente fornecida, só corre até 55, em todos os casos, apresentando um comportamento linear, com muito tempo despendido, a Solução 1 (a solução previamente fornecida mas otimizada) e a Solução 2 (usando Método Travagem) oferecem um desempenho significativamente mais rápido.

De facto, quando o argumento de entrada é nulo, como exemplificado pelos gráficos, os dois computadores apresentam comportamentos diferenciados. Enquanto que, no PC R, a solução 1 e 2 os gráficos tempo

posição se apresentam muitos próximos com alguns destes picos a convergir, no PC M tal já não acontece e é mostrado que a Solução 1 tem um desempenho que aumenta até à posição 200, após isso baixando um pouco a performance até chegar à posição final (800).

O mesmo acontece com a solução 2, método travagem, apresentando um desempenho mais ao menos contínuo, com alguns picos negativos e positivos, mas que ao comparar com a solução 1, executa o algoritmo em muito menos tempo, não se apresentando cruzado com o gráfico da solução 1, como acontece em PC R.

Quando os argumentos de entrada são números mecanográficos, também irá acontecer o descrito anteriormente, com alguns picos negativos e positivos a mais ou a menos.

Deste modo, concluímos que o algoritmo Travagem apresenta uma performance significativamente melhor em comparação com o algoritmo Otimizado. Estes os dois apresentam uma taxa de execução bem mais acessível do que o algoritmo previamente fornecido, que devido à sua taxa de execução só conseguia executar os movimentos até à posição 55.

Conclusão

Ao longo da resolução deste trabalho, deparámo-nos não só com diferentes problemas como também alguns sucessos. Posto isto, podemos afirmar que melhorou a nossa capacidade de resolução de problemas uma vez que nos permitiu a procura de soluções distintas que melhor satisfizesse o objetivo do trabalho.

Conseguimos compreender a importância de um tempo de execução ajustado, um baixo tempo de execução para a execução completa do programa, sendo que para o ajuste foi necessário entender o código em C para que o pudéssemos, posteriormente, otimizar. Quanto à otimização, adquirimos novas técnicas de como tornar o código mais rápido para além de percebermos que tipo de estruturas de dados, ciclos e *statements* demoram menos tempo a ser executados e que conjugados desses mesmos têm uma menor complexidade computacional.

Este trabalho, motivou-nos a repensar, no futuro, na forma como construímos o nosso código para que seja o mais eficiente possível.

Anexos

Disponibilizamos, aqui, o código desenvolvido em C para as resoluções do problema pretendido, cujas duas soluções desenvolvidas foram feitas em dois scripts diferentes, bem como o código Matlab utilizado para permitir a visualização dos gráficos dos diferentes tempos de execução para que se pudessem retirar as conclusões projetadas. Além disto anexaremos as imagens dos pdfs criados pelo programa de quando o carro faz um percurso de 400 e 800.

Código C

Solução Otimizada

```
#define _max_road_size_ 800 // the maximum problem size
#define _min_road_speed_ 2 // must not be smaller than 1, should not 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 "elapsed_time.h"
#include "make_custom_pdf.c"

// road stuff

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

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

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

```
        max_road_speed[i] = _max_road_speed_;
    }
}

// description of a solution

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

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

static void solution_1_recursion_otimizado(int move_number,int position,int
speed,int final_position)
{
    int i,new_speed;

    if (solution_1_best.n_moves != final_position + 100) return;

    solution_1_count++;
    solution_1.positions[move_number] = position;

    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;
    }

    if(final_position <= 3){
        solution_1_best.n_moves = final_position;
    }else{
        for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--) // testar
primeiro quando a velocidade aumenta
            if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed
<= final_position){
                for(i = 0; new_speed <= max_road_speed[position + i]; i++)
                    ;
            }
    }
}
```

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

        if(i>new_speed)
            solution_1_recursion_otimizado(move_number + 1, position + new_speed,
new_speed, final_position);
    }
}
}

//rest of the code

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_otimizado(0,0,0,final_position);
    // solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;

    solution_1_elapsed_time = cpu_time();
    solution_1_count = 0ul;
    solution_1_best.n_moves = final_position + 100;
    solution_1_recursion_otimizado(0,0,0,final_position);
    solution_1_elapsed_time = cpu_time() - solution_1_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_be
st.n_moves,&solution_1_best.positions[0],solution_1_elapsed_time,solution_1_count,
"Plain recursion");
    printf("mad road speeds:");
    for(i = 0;i <= final_position;i++)

```

```

    printf(" %d",max_road_speed[i]);
    printf("\n");
    printf("positions:");
    for(i = 0;i <= solution_1_best.n_moves;i++)
        printf(" %d",solution_1_best.positions[i]);
    printf("\n");
}

// main program

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

    // generate the example data
    if(argc == 2 && argv[1][0] == '-' && argv[1][1] == 'e' && argv[1][2] == 'x')
    {
        example();
        return 0;
    }
    // initialization
    n_mec = (argc < 2) ? 0xAED2022 : atoi(argv[1]);
    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 == 4 || final_position == 200) ? 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_best.positions[0],solution_1_elapsed_time,solution_1_count,"Plain recursion");
            }
        }
    }
}

```

```
    printf(" %3d %16lu %9.3e
| ",solution_1_best.n_moves,solution_1_count,solution_1_elapsed_time);
}
else
{
    solution_1_best.n_moves = -1;
    printf("                                | ");
}
// second solution method (less bad)
// ...

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

Método Travagem

```
#define _max_road_size_ 800 // the maximum problem size
#define _min_road_speed_ 2 // must not be smaller than 1, should not be smaller
than 2
#define _max_road_speed_ 9 // must not be larger than 9 (only because of the PDF
figure)
#define STACK_EMPTY
#define EMPTY -1
//
// 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 <stdbool.h>
#include "elapsed_time.h"
#include "make_custom_pdf.c"

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

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

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

///////////changing it to match com distancia de
travagem///////////
//



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_2_best;
static double solution_2_elapsed_time; // time it took to solve the problem
static unsigned long solution_2_count; // effort dispended solving the problem

// 5 - 4 - 3 - 2 -1
bool testar_travagem(int new_speed, int positions, int move_number, int
final_position)
{
    if (positions + new_speed > final_position)
    {
        return false;
    }
    else
    {
        for (int pos = positions; pos <= positions + new_speed; pos++)
        {
            solution_2_count++;
            if (new_speed > max_road_speed[pos])
            {
                return false;
            }
        }

        if (new_speed == 1)
        {
            return true;
        }
        else
        {
            positions = positions + new_speed;
            return testar_travagem(new_speed - 1, positions, move_number,
final_position);
        }
    }
}

void solution_2_travagem(int speed, int position, int move_number, int
final_position)
{
    while (position != final_position)
    {
        // try all legal speeds
        for (int new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
        {
            if (new_speed >= 1 && new_speed <= _max_road_speed_)
            {
                if (testar_travagem(new_speed, position, move_number, final_position) ==
true)
```

```

    {
        solution_2_best.positions[move_number++] = new_speed;
        solution_2_best.n_moves = move_number;
        speed = new_speed;
        position += new_speed;
        break;
    }
}
}

}

///////////////////////////////
static void solve_2(int final_position)
{
    if (final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr, "solve_2: bad final_position\n");
        exit(1);
    }
    solution_2_elapsed_time = cpu_time();
    solution_2_count = 0ul;
    solution_2_best.n_moves = final_position + 100;
    solution_2_travagem(0, 0, 0, final_position);
    // solution_2_travagem(last_speed, last_position, last_move_number,
    final_position);
    solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}

//  

// example of the slides  

//  

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

    srand(0xAED2022);
    init_road_speeds();
    final_position = 30;
    solve_2(final_position);
    make_custom_pdf_file("example.pdf", 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("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_2_best.n_moves; i++)
    printf(" %d", solution_2_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_2_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_2_elapsed_time < _time_limit_)
        {
            solve_2(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_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("                                |");
}
// second solution method (less bad)
// ...

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

Código MatLab

```
%% PC Rafaela

%% sem número mecanográfico
% solução 0 s/ n_mec
valores16=load("speed_run_solS.txt");
pos_finais16=valores16(:,1);
tempo16=valores16(:, 4);

% solução 1 otimizada s/ n_mec
valores1=load("speed_run_sol1_ot_R0.txt");
pos_finais1=valores1(:,1);
tempo1=valores1(:, 4);

% solução 2 travagem s/ n_mec
valores2=load("speed_run_sol2_tra_R0.txt");
pos_finais2=valores2(:,1);
tempo2=valores2(:, 4);

figure(1);
plot(pos_finais16, log10(tempo16),"m",pos_finais1, log10(tempo1),'b',
pos_finais2, log10(tempo2), 'g');
legend('Solução_0 PC_R 0h', 'Solução_1 Otimizada PC_R 0h', 'Solução_2
Travagem PC_R 0h');
ylabel("Tempo(log(s)))");
xlabel("Posição Final");
title ("PC Rafaela - 0")
grid on;

%% 107658
% solução 0 107658
valores17=load("speed_run_sols_107658.txt");
pos_finais17=valores17(:,1);
tempo17=valores17(:, 4);

% solução 1 otimizada 107658
valores3=load("speed_run_sol1_ot_R107658.txt");
pos_finais3=valores3(:,1);
tempo3=valores3(:, 4);

% solução 2 travagem 107658
valores4=load("speed_run_sol2_tra_R107658.txt");
pos_finais4=valores4(:,1);
tempo4=valores4(:, 4);

figure(2);
plot(pos_finais17, log10(tempo17), "m",pos_finais3, log10(tempo3), 'b',
pos_finais4, log10(tempo4), 'g');
```

```
legend('Solução_0 PC_R 107658h','Solução_1 Otimizada PC_R 107658h',
'Solução_2 Travagem PC_R 107658h');
ylabel("Tempo(log(s)));
xlabel("Posição Final");
title ("PC Rafaela - 107658")
grid on;

%% 108193
% solução 0 s/ n_mec
valores18=load("speed_run_sols_107658.txt");
pos_finais18=valores18(:,1);
tempo18=valores18(:, 4);

% solução 1 otimizada 108193
valores5=load("speed_run_sol1_ot_R108193.txt");
pos_finais5=valores5(:,1);
tempo5=valores5(:, 4);

% solução 2 travagem 108193
valores6=load("speed_run_sol2_tra_R108193.txt");
pos_finais6=valores6(:,1);
tempo6=valores6(:, 4);

figure(3);
plot(pos_finais18, log10(tempo18),'m',pos_finais5, log10(tempo5), 'b',
pos_finais6, log10(tempo6), 'g');
legend('Solução_0 PC_R 108193h','Solução_1 Otimizada PC_R 108193h',
'Solução_2 Travagem PC_R 108193h');
ylabel("Tempo(log(s)));
xlabel("Posição Final");
title ("PC Rafaela - 108193")
grid on;

%% PC Matilde

%% sem número mecanográfico
% solução 0 s/ n_mec
valores13=load("speed_run_sols.txt");
pos_finais13=valores13(:,1);
tempo13=valores13(:, 4);

% solução 1 otimizada s/ n_mec
valores7=load("speed_run_sol1_ot_M0.txt");
pos_finais7=valores7(:,1);
tempo7=valores7(:, 4);

% solução 2 travagem s/ n_mec
valores8=load("speed_run_sol2_tra_M0.txt");
pos_finais8=valores8(:,1);
tempo8=valores8(:, 4);
```

```
figure(4);
plot(pos_finais13, log10(tempo13), "r", pos_finais7,
log10(tempo7), 'k', pos_finais8, log10(tempo8), 'c');
legend("Solução_0 PC_M 0h", "Solução_1 Otimizada PC_M 0h", "Solução_2
Travagem PC_M 0h");
ylabel("Tempo(log(s))");
xlabel("Posição Final");
title ("PC Matilde - 0")
grid on;

%% 107658
% solução 0 107658
valores14=load("speed_run_sols_107658.txt");
pos_finais14=valores14(:,1);
tempo14=valores14(:, 4);

% solução 1 otimizada 107658
valores9=load("speed_run_sol1_ot_M107658.txt");
pos_finais9=valores9(:,1);
tempo9=valores9(:, 4);

% solução 2 travagem 107658
valores10=load("speed_run_sol2_tra_M107658.txt");
pos_finais10=valores10(:,1);
tempo10=valores10(:, 4);

figure(5);
plot(pos_finais14, log10(tempo14),"r", pos_finais9, log10(tempo9), 'k',
pos_finais10, log10(tempo10), 'c');
ylabel("Tempo(log(s))");
xlabel("Posição Final");
legend("Solução_0 PC_M 107658h", 'Solução_1 Otimizada PC_M 107658h',
'Solução_2 Travagem PC_M 107658h');
title ("PC Matilde - 107658")
grid on;

%% 108193
% solução 0 108193
valores15=load("speed_run_sols_108193.txt");
pos_finais15=valores15(:,1);
tempo15=valores15(:, 4);

% solução 1 otimizada 108193
valores11=load("speed_run_sol1_ot_M108193.txt");
pos_finais11=valores11(:,1);
tempo11=valores11(:, 4);

% solução 2 travagem 108193
valores12=load("speed_run_sol2_tra_M108193.txt");
```

```
pos_finais12=valores12(:,1);
tempo12=valores12(:, 4);

figure(6);
plot(pos_finais15, log10(tempo15),'r', pos_finais11, log10(tempo11), 'b',
pos_finais12, log10(tempo12), 'c');
ylabel("Tempo(log(s))");
xlabel("Posição Final");
legend("Solução_0 PC_M 108193h", 'Solução_1 Otimizada PC_M 108193h',
'Solução_2 Travagem PC_M 108193h');
title ("PC Matilde - 108193")
grid on;

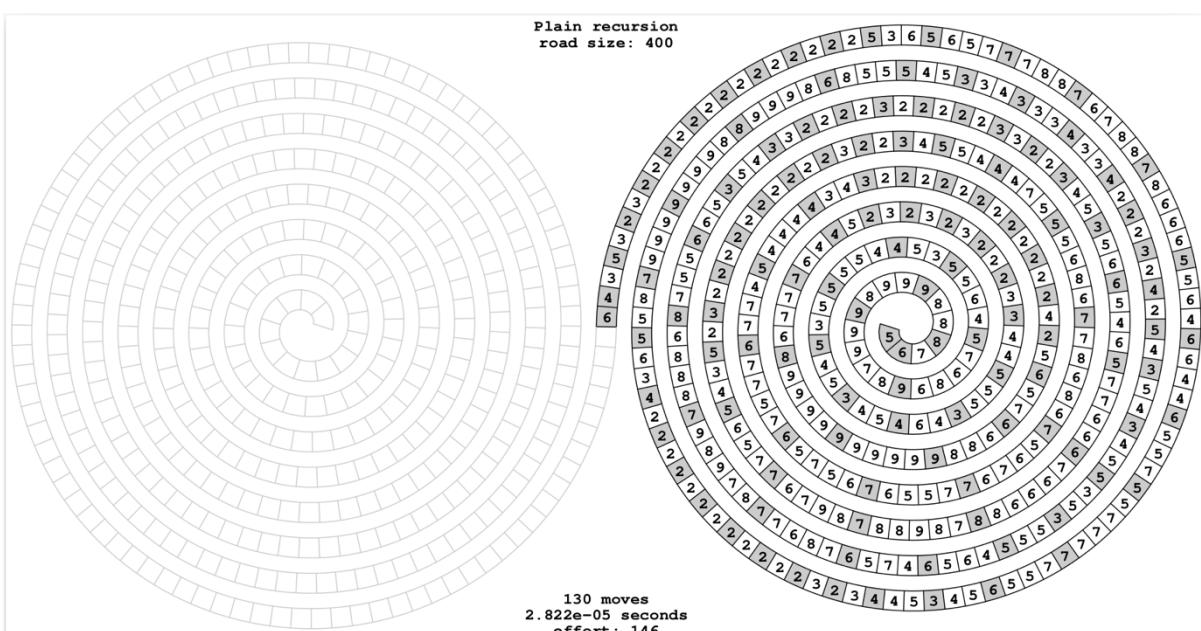
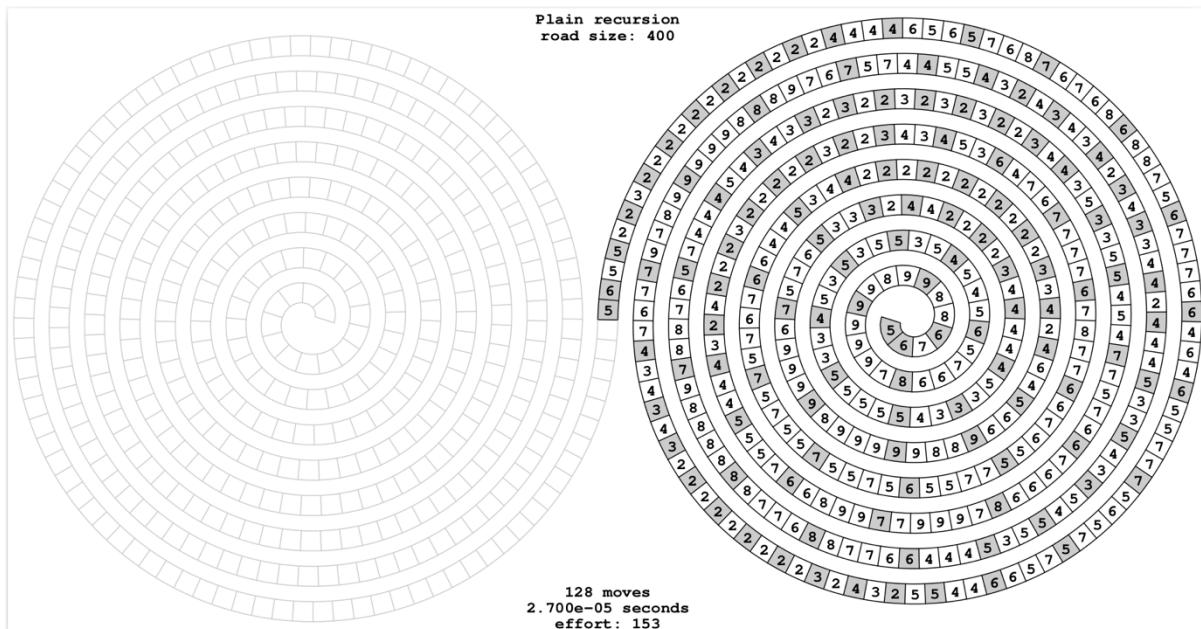
%% outros
%tempo_log=log10(tempo);
%N=[pos_finais(20:end) 1+0*pos_finais(20:end)];
%Coefs=pinv(N)*tempo_log(20:end);
%Ntotal=[pos_finais pos_finais*0+1];

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

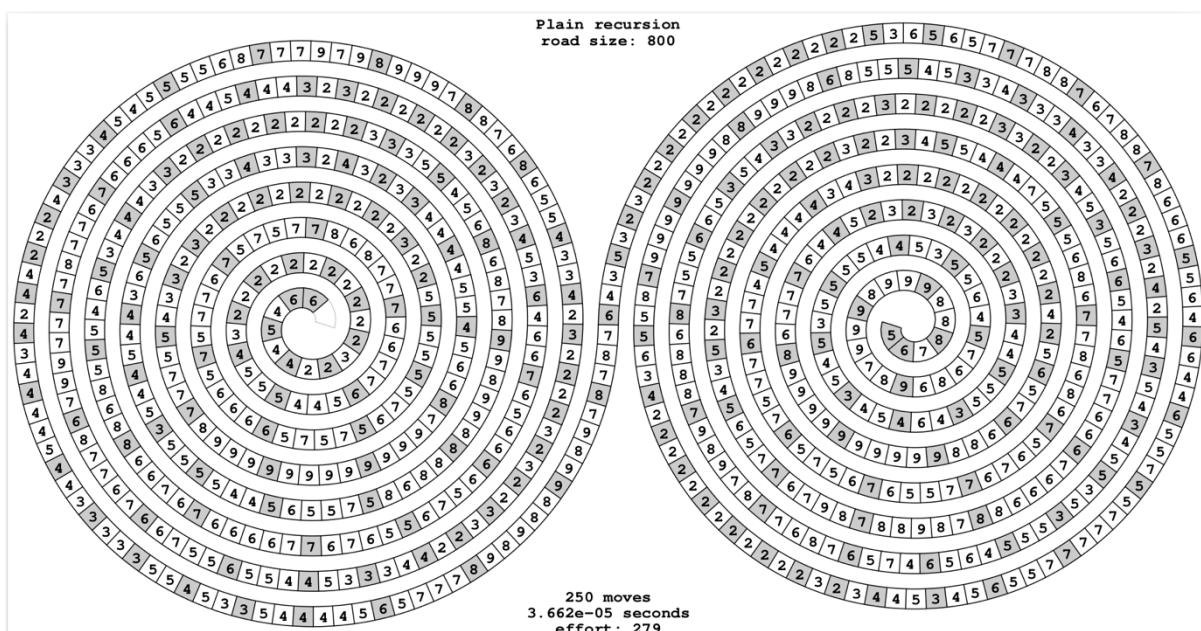
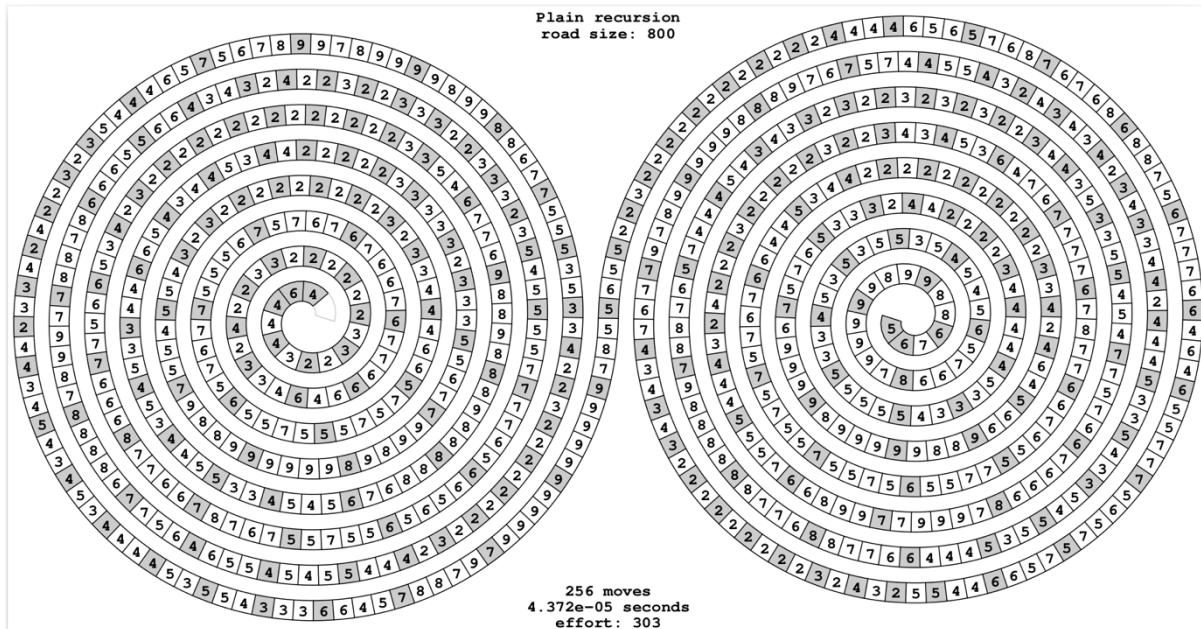
%t800_log=[800 1]*Coefs;
%t800=10^t800_log
```

PDFs Criados

400 posições:



800 posições:



Bibliografia

- Slides Teóricos de Algoritmos e Estruturas de Dados
- Aulas Práticas de Algoritmos e Estruturas de Dados
- DICTIONARY.COM. (2022). speedrun. Obtido de dictionary.com:
<https://www.dictionary.com/browse/speedrun> (Acedido a 03/12)
- <https://www.geeksforgeeks.org/basic-code-optimizations-in-c/> (Acedido a 26/10)
- <https://www.digitalocean.com/community/tutorials/trie-data-structure-in-c-plus-plus> (Acedido a 03/11)
- <https://www.geeksforgeeks.org/data-structures/> (Acedido a 14/11)
- <https://www.geeksforgeeks.org/dynamic-programming/> (Acedido a 16/11)