

Bernardo Pinto, 105926 – 50%
José Mendes, 107188 – 50%

Índice

Índice	1
Introdução	2
Conhecendo o problema	3
Metodologia	4
Soluções	5
1. solution_1_recursion() e solve_1() (previamente fornecido)	5
2. solution_2_recursion() e solve_2()	8
3. solution_3_recursion() e solve_3()	9
4. solution_4_recursion() e solve_4()	11
5. solution_5_recursion() e solve_5()	13
Resultados obtidos	15
Solução 1:	15
Solução 2:	17
Solução 3:	19
Solução 4:	21
Solução 5:	23
Comparações entre soluções:	25
Conclusão	27
Código – C	28
Código – Matlab	42

Introdução

O objetivo deste primeiro trabalho prático foi analisar e implementar a forma mais rápida e eficiente de resolver o problema apresentado, “Speed Run”.

Já é atribuída uma possível solução para este problema, no entanto, apesar de eficaz é bastante demorada a encontrar a solução pretendida, daí ser pedido para implementar novas soluções que consigam resolver este dilema mais rapidamente.

Esta implementação, por sua vez, necessita de uma explicação sobre o método utilizado e a razão para ser melhor ou até pior do que o método original. Podendo ser realizada recorrendo a métodos que foram lecionados nas aulas teóricas e práticas ou por programação dinâmica.

Conhecendo o problema

O primeiro trabalho prático da unidade curricular de Algoritmos e Estruturas de Dados tem como objetivo compreender a solução dada e desenvolver soluções melhores para o problema “Speed Run”.

O problema consiste num carro que inicia o seu percurso numa estrada com velocidade zero e que deve chegar ao final dessa com velocidade umz para que possa reduzir a velocidade e, então, parar. No entanto, o problema contém normas que devem ser seguidas, entre elas:

1. A estrada é subdividida em segmentos com aproximadamente o mesmo comprimento.
2. Cada segmento tem o seu próprio limite máximo de velocidade.
3. A velocidade é medida pelo número de segmentos que o carro pode avançar em um único “movimento”
4. Um movimento pode ser: reduzir a velocidade em um (i.e, travagem), manter a velocidade atual ou aumentar a sua velocidade em um (i.e, acelerar).

Além disso, o objetivo da resolução do problema é determinar o número mínimo de movimentos necessários para alcançar a posição final seguindo as normas impostas.

Metodologia

Código previamente fornecido:

Para a realização desta tarefa, foi fornecido algum material para tornar a compreensão daquilo que é pedido mais simples e fácil de entender.

O ficheiro “**make_costum_pdf.c**” recebe os resultados das soluções implementadas e transforma-as em um **pdf** com a estrada desenhada, já com as velocidades de cada segmento representadas, e com as posições que o carro ocupa ao longo do seu trajeto num tom mais escuro. Exemplo fornecido:

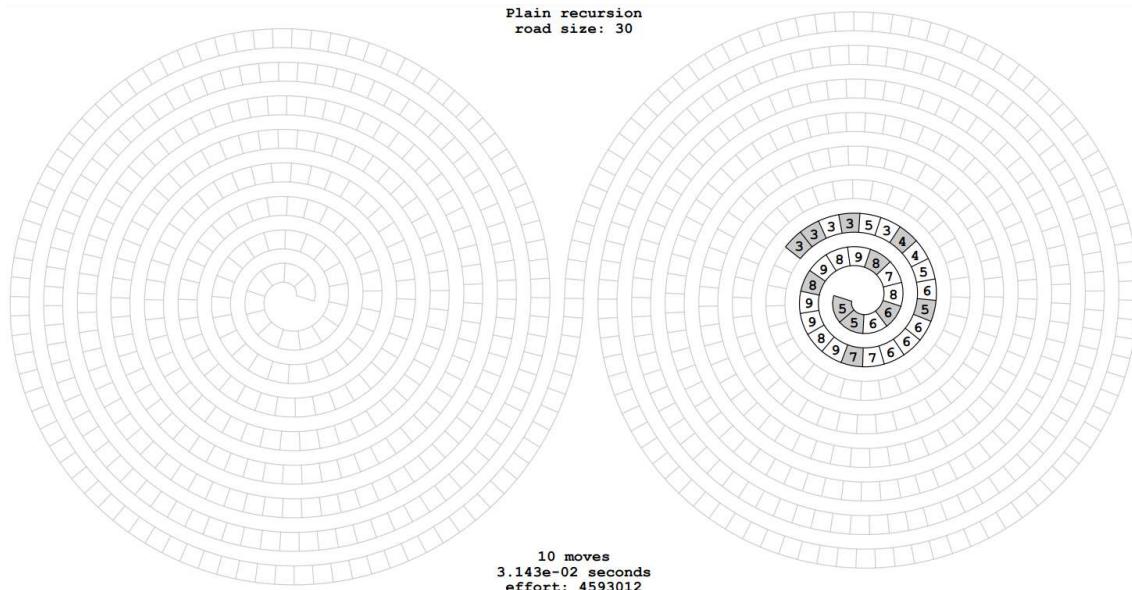


Figura 1- 10 movimentos no programa fornecido

Este exemplo é criado através de uma função que seja capaz de encontrar uma solução. No caso do exemplo de cima, foi utilizada a função já fornecida (a imagem foi obtida ao executar “./speedrun -ex” no terminal).

O ficheiro “**elapsed_time.h**” é um *header* que permite calcular o tempo que uma determinada função demora a executar. Por sua vez, bastante útil para este trabalho prático, pois ajuda a ver o tempo que cada solução encontrada demora na sua execução (quanto menos tempo, melhor a solução).

Por fim, o ficheiro “**speed_run.c**”, que é o principal e o único que necessita de alterações para a realização desta tarefa. Possui 5 funções já fornecidas e que transmitem a ideia principal desta tarefa.

A função “**init_road_speeds**” que vai preencher o *array* que tem como objetivo guardar as velocidades limite de cada segmento da estrada, “**max_road_speed**”.

A função “**solution_1_recursion**” que possui a ideia principal do trabalho prático. A forma como o carro tem de avançar para conseguir chegar ao final da estrada com velocidade um. Esta será explicada mais detalhadamente ao decorrer deste relatório.

A função “**solve_1**” que permite que a função anterior seja realizada como pretendido e para poder calcular o tempo que a função recursiva demora a executar.

A função “**example**” permite a realização da imagem acima apresentada. Este chama a função “**solve_1**” com o tamanho da estrada que pretende, neste caso 30, descobre as posições que o carro deve ocupar e desenvolve um **pdf** com os resultados. Pode ser utilizada executando “**./speed_run -ex**” no terminal.

Finalmente, a função “**main**” que vai utilizar a função “**solve_1**” para descobrir a melhor solução possível para um determinado tamanho da estrada, desde 1 até 800. Apresentando os valores numa tabela e ao mesmo tempo em **pdf** através da função “**make_custom_pdf**” que produz estes ficheiros com vários cumprimentos de estrada, desde 10 até 800.

Soluções

Duas partes essenciais para a realização deste trabalho são: a struct **solution_t** e as variáveis estáticas definidas fora da função. A struct possui uma variável inteira que guarda o número de movimentos de cada solução, **n_moves**, e um array de inteiros que guarda as posições que o carro necessita passar para realizar a solução, **positions[1 + tamanho máximo da estrada]**. As variáveis estáticas são, **solution_n_elapsed_time** (n é o número da solução), para calcular o tempo que cada solução demora a executar, e **solution_n_count** (n é o número da solução), para calcular o effort que cada solução tem de ter até encontrar a melhor solução.

1. **solution_1_recursion()** e **solve_1()** (previamente fornecido)

A função “**solution_1_recursion**” recebe por parâmetros o número de movimentos (**move_number**) realizados até então, a posição (**position**) em que o carro se encontra, a velocidade (**speed**) que o carro se encontra naquela posição, e a posição final (**final_position**).

Inicialmente, a função começa por definir duas variáveis **i** e **new_speed**, para ser mais tarde utilizado num *loop for* e de modo a obter a nova velocidade a ter em conta, respetivamente. É também, incrementada a variável **solution_1_count** e a posição atual é guardada no *array* definido na *struct*.

É criado um bloco condicional *if* para verificar se esta fase da recursão é uma solução (“**position == final_position && speed == 1**”), caso seja, vai verificar se é a melhor solução até agora. Se se verificar, vai substituir a, até então, melhor solução. Acaba retornando ao passo anterior da recursão (ou terminando), **return**.

Após esta condição, é iniciado um *loop for* que vai atribuir uma nova velocidade, **new_speed**, começando inicialmente por tentar reduzir, manter ou aumentar a velocidade por esta ordem de prioridade (caso uma funcione, as outras não serão utilizadas nesta recursão). Dentro deste, vai verificar se a nova velocidade é legal, isto é, se é maior do que a velocidade mínima, menor do que a velocidade máxima permitida e se não iria ultrapassar o último segmento da estrada. Caso tudo se verifique, vai ver se os segmentos entre a posição atual e a suposta futura posição possuem velocidades limites menores do que a velocidade a que o carro

se encontra (*loop for e if*). Se tudo isto ocorrer com sucesso, é então chamada a função novamente, através de recursão, em que o **move_number** é incrementado, **position** vai ser igual á posição anterior mais a nova velocidade, **new_speed**, **speed** vai ser a nova velocidade e **final_position** permanece inalterado (e assim será em todas as recursões).

Apesar de funcionar de forma correta, esta solução é bastante lenta. Uma vez que ao deixar o computador executar o programa durante 1 hora (**_time_limit_** de 3600.0), apenas serão calculadas as melhores soluções até á 50 (o objetivo é 800):

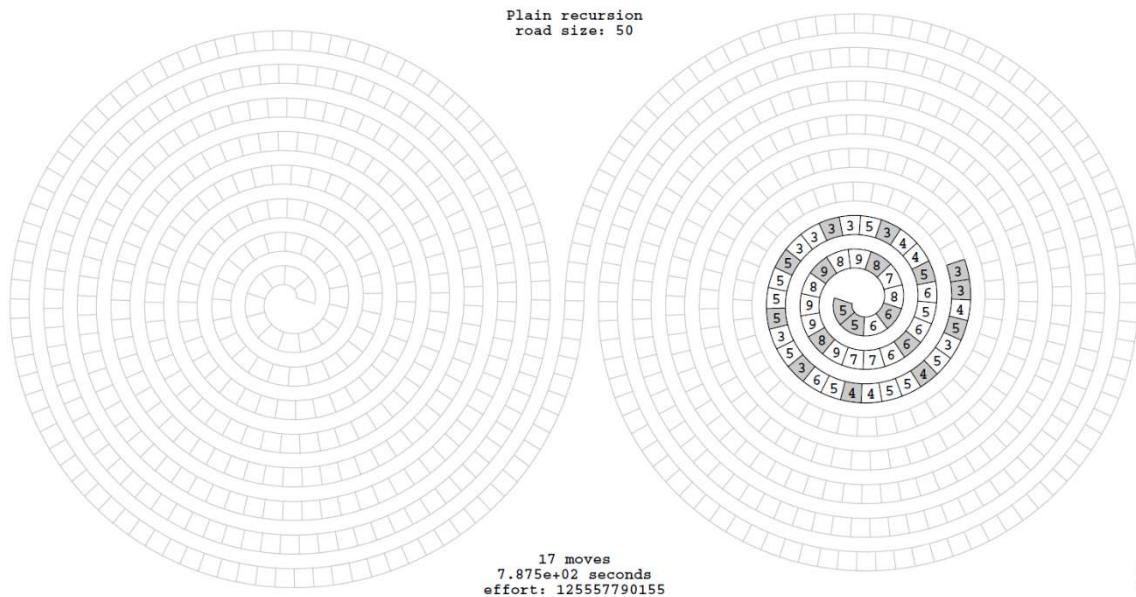


Figura 2 - 17 movimentos na solução 1

Como podemos ver pelo **pdf** gerado, esta solução demorou 7.9e2 segundos e um **effort** (que significa o número de vezes que executou a recursão) superior a 100 mil milhões.

Isto acontece, pois, esta solução vai testar todas as possibilidades. Mesmo que encontre a melhor solução, o programa vai continuar a testar todas as soluções possíveis, daí percorrer a recursão um número tão elevado de vezes. Além disso, como referido anteriormente, esta solução vai começar por testar se pode diminuir a velocidade em primeiro lugar e acelerar em último, logo, como pretendemos a solução em que o carro consiga percorrer a estrada o mais rapidamente possível, a melhor solução vai ser uma das últimas a ser calculada ou mesmo a última. Com o intuito de compreender melhor esta explicação, foi criado a segunda solução.

Neste trabalho prático também foi pedido para tentar encontrar uma fórmula que seja capaz de estimar o tempo de execução que a solução dada possui para uma certa posição final.

Para tal, é elaborado o gráfico da primeira solução, com os dados obtidos através da mesma:

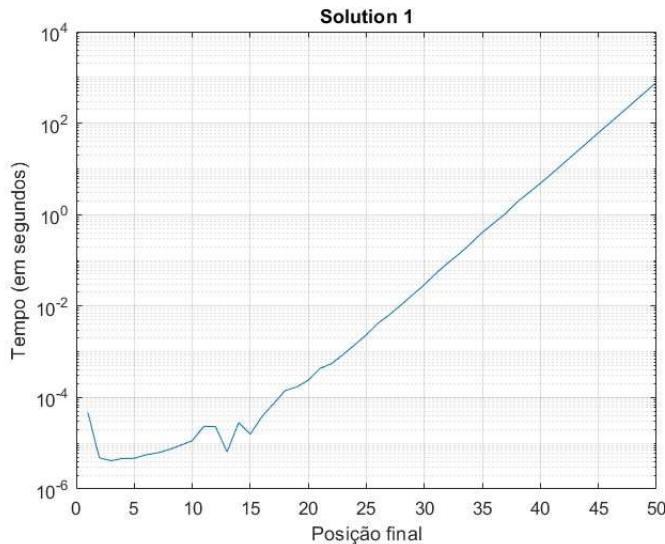


Figura 3 - Gráfico da posição no tempo (solução 1)

Como é possível observar no gráfico gerado, quando a posição final é aproximadamente 20, a linha do gráfico assemelha-se a uma equação linear, logo, com o intuito de obter uma aproximação desta equação, são apenas selecionados os dados de 20 em diante. Obtém-se, para cada ponto, as suas coordenadas do gráfico e, por fim, é elaborada a reta:

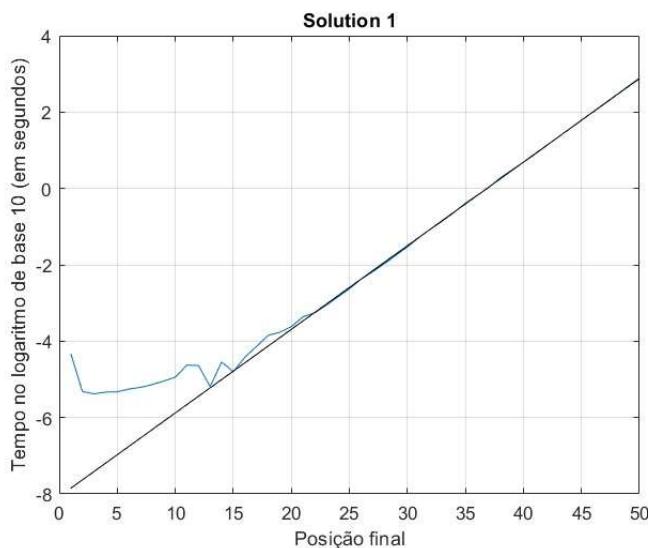


Figura 4 - Aproximação da equação linear

Pretendemos saber quanto tempo demoraria esta primeira solução a calcular a melhor solução quando a posição final corresponde a 800. Através desta reta conseguimos prever que o tempo que esta solução demoraria a realizar este cálculo seria aproximadamente $4.0912e+159$ anos, por isso é necessário criar novos métodos mais eficientes e mais rápidos.

2. solution_2_recursion() e solve_2()

Esta solução foi desenvolvida a partir da primeira, em que apenas é alterada a prioridade da alteração da velocidade do carro. Primeiramente tenta acelerar, se não conseguir, mantém a velocidade, e por fim se nenhuma for possível diminui a velocidade. Deste modo, como é pretendido obter a solução em que o carro chegue á posição final com o menor número de movimentos, uma das primeiras soluções encontradas ou mesmo a primeira será a melhor solução, uma vez que, quanto maior for a velocidade, menor será o número de movimentos.

No entanto, este programa continua a testar todas as soluções possíveis, mesmo encontrando a melhor, como demonstra o pdf:

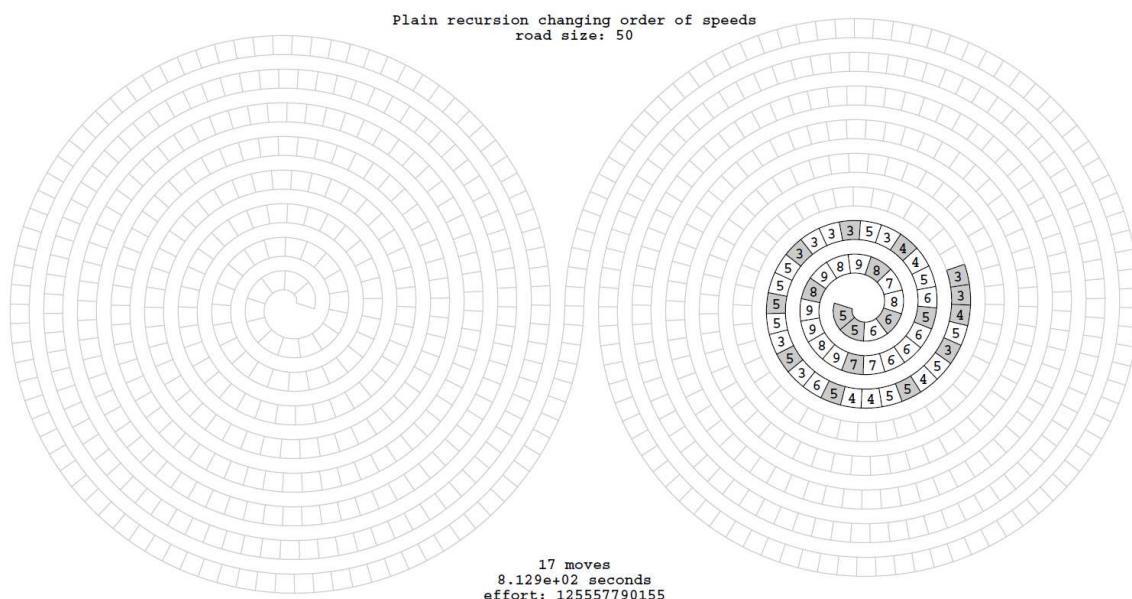


Figura 5 - 17 movimentos na solução 2

Assim, com estas conclusões, foi desenvolvida outra solução, que guardasse num array as velocidades e o número de movimentos em cada posição, de modo a permitir comparar esta com as seguintes. Caso essa nova solução numa determinada posição tenha um maior número de movimentos ou menor velocidade, não há necessidade de a continuar a testar.

3. solution_3_recursion() e solve_3()

Primeiramente, vale ressaltar que nessa solução foi implementado um *array* bidimensional “**best_moves_speed[800][2]**” para que fosse armazenado tanto o número de movimentos como a velocidade de acordo com a sua posição (1 a 800 ou 0 a 799 em índices do *array*). Este *array* é preenchido da função **solve_3()**, em que a posição das velocidades é preenchida com zeros e a do número de movimentos com o **_max_road_size_**, para que seja possível preencher o *array* pela primeira vez, pois este só é preenchido se a velocidade for maior e o número de movimentos for menor numa dada posição.

Esta solução intermédia inicia, assim como as outras, com a declaração de uma variável “**i**”, que será usada posteriormente nos ciclos, e uma variável “**new_speed**”, que também será utilizada no ciclo, mas desta vez para fazer a alteração da velocidade.

Em seguida o valor do contador “**count**” é incrementado em uma unidade, para que, dessa forma, sempre que a função for utilizada seja contado o *effort* da recursão.

O próximo passo consiste em fornecer como índice do *array* “**solution_3.positions**” o valor da variável “**move_number**”, para que nesse índice seja salvo o valor da posição atual “**position**”, que, assim como “**move_number**”, é instanciada nos parametros da função. Para que, assim, a posição atual fique salva e correlacionada ao número de movimentos realizados até o momento.

Em seguida foi construído um bloco condicional que testa se o passo atual da recursão é uma solução, ou seja, se a posição atual “**position**” é igual a posição final, “**final_position**” e se a velocidade atual é 1. Por fim, caso uma solução tenha sido encontrada, é realizado um teste que verifica se esta solução é a melhor solução encontrada até o momento. Este teste consiste em testar se o número de movimentos realizados até o momento “**move_number**” é menor do que o numero de movimentos da melhor solução já encontrada “**solution_3_best.n_moves**”. Caso se verifique, a solução atual torna-se a melhor solução já encontrada “**solution_3_best**”, o número de movimentos até o momento é repassado ao número de movimentos da melhor solução “**solution_3_best.n_moves**” e é efetuado o “**return**”.

Em contra partida, caso o passo atual da recursão não gere uma solução, foi implementado um bloco condicional que testa, através do *array* bidimensional “**best_moves_speed[][]**”, se a posição atual já foi alcançada com menos movimentos e com velocidade menor, detetando, assim, que o passo atual da recursão já é pior do que algum outro encontrado, fazendo com que esta termine (**return**) sem que precise chegar ao final, tornando assim o programa mais rápido e eficiente. No entanto, caso a posição atual ainda não tenha sido alcançada com o número de movimentos e a velocidade atual, essa informação é atribuída da seguinte forma ao *array* bidimensional:

- **best_moves_speed[position][0] = move_number;**
- **best_moves_speed[position][1] = speed;**

Posteriormente, caso o passo da recursão ainda não tenha sido encerrado, a execução do programa entra no loop *for* responsável por decidir o próximo movimento do carro, assim como nas outras soluções desenvolvidas.

Essa função é utilizada na main através da função **solve_3**, que irá testar se a solução chegou realmente ao fim da estrada e irá definir alguns parâmetros, entre eles o tamanho da estrada, o tempo de início e, obviamente, realizar chamada inicial da função **solution_3_recursion()**

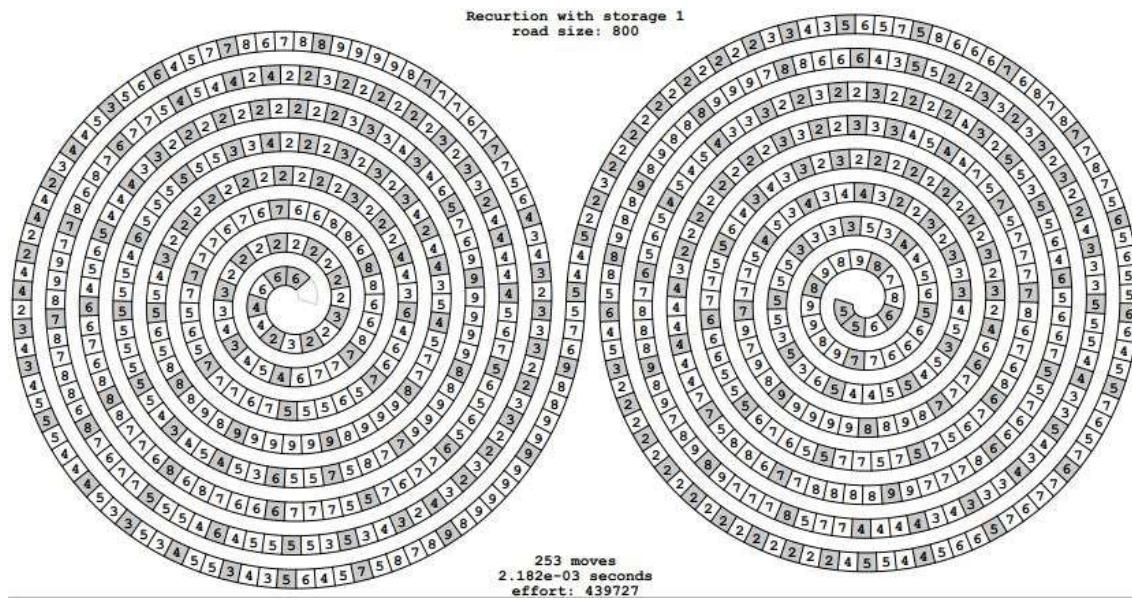


Figura 6 - 253 movimentos na solução 3

Por fim, por mais que esta função tenha resultado em uma solução completa do problema em apenas alguns segundos, o grupo percebeu que algumas alterações ainda podiam ser realizadas para que fosse encontrada, então, outra solução ainda mais eficiente. Então foi desenvolvida a solução 4.

4. solution_4_recursion() e solve_4()

Nessa solução intermedia, além das variáveis declaradas nos parâmetros da função **solution_3_recursion**, foram declaradas as variáveis locais “*i*” e “*k*”, que serão utilizadas posteriormente em loops, e a variável “**new_speed**”, que receberá o valor da velocidade para a recursão seguinte.

A função inicia com um loop *for* que tem a função de, no *array* bidimensional **solution_4.move_speed[][]** (declarado como parâmetro da *struct* acima definida), repassar os valores da quantidade de movimentos e a velocidade na posição atual para as posições que o carro passou mas não parou. Por exemplo:

8	9	9	8	9	7
[10]	[11]	[12]	[13]	[14]	[15]

Figura 7 - Exemplo de posições

Considerando que o carro chegou na primeira posição (i.e, primeira posição cinza) com 10 movimentos e com velocidade igual a 4, e, em seguida, acelera avançando para a segunda posição cinza, chegando com 11 movimentos e velocidade igual a 5. O ciclo *for* serve para atribuir às posições intermedias, representadas pelos números 9,9,8 e 9, a mesma quantidade de movimentos (11) e velocidade atual (5) da última posição cinza, para que desta forma a recursão funcione da melhor maneira possível.

No seguimento da função a quantidade de movimentos, que é igual ao número de chamadas a função, é incrementado em uma unidade e, além disso, no *array* “**solution_4.positions**”, no índice com valor do número de movimentos, é armazenada a posição atual.

Posteriormente, foi implementado um bloco condicional que testa se o passo atual da recursão é uma solução, bloco esse que consiste, assim como na solução 3, em testar se a posição atual é igual à posição final e se a velocidade atual é 1. Então, caso uma solução tenha sido encontrada, é verificada se tal solução é a melhor solução até o momento, testando, portanto, se o numero de movimentos realizados “**move_number**” é menor do que o numero de movimentos da melhor solução “**solution_4_best.n_moves**”. Em seguida, caso se verifique, a solução atual passa a ser a melhor solução já encontrada “**solution_4_best**” e o número de movimentos até o momento passa a ser o número de movimentos da melhor solução “**solution_4_best.n_moves**”. Por fim, mesmo que a solução encontrada não seja a melhor solução, a variável global “**sol_found**” recebe o valor inteiro 1 e é dado, assim, o “**return**” da função. Esta variável, tem como objetivo apenas fazer as verificações para ver se uma solução é melhor que outra assim que uma primeira solução seja encontrada, de modo a ter forma de comparar. Na solução anterior o *array* era previamente preenchido na função **solve3()**, neste caso, tal não é necessário com a ajuda desta variável.

Logo após foi desenvolvido um bloco condicional que testa, caso o valor da variável “**sol_found**” seja igual a um, se a posição seguinte já foi alcançada com um número menor de movimentos ou com uma velocidade maior, garantindo, assim, que já existe uma melhor solução, e, portanto, o ciclo atual já pode ser encerrado efetuando o “**return**” da função.

Por fim, caso a função ainda não tenha sido encerrada, a execução do programa entra no ciclo *for* que irá decidir o próximo movimento do carro, assim como é feito nas outras soluções.

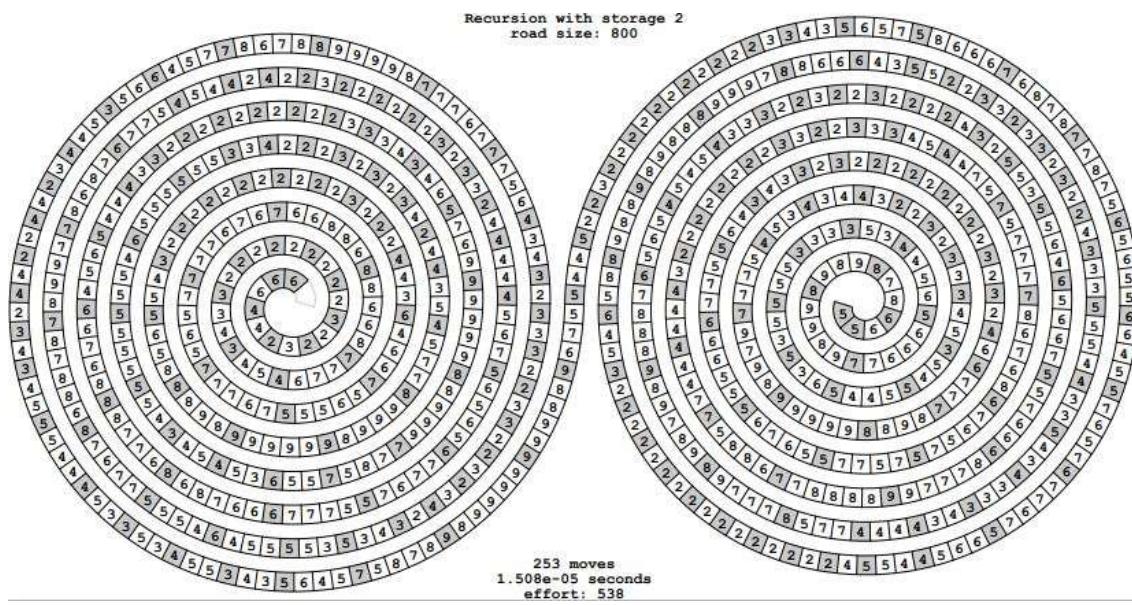


Figura 8 - 253 movimentos na solução 4

Entretanto, ainda que a solução 4 tenha resultado em uma melhora significativa no tempo e no “effort” quando comparada com a solução 3, o grupo notou que alguns ajustes ainda poderiam ser realizados, principalmente no código responsável por perceber que uma provável solução já é pior do que uma solução encontrada anteriormente e no uso da programação dinâmica. Alterações que resultaram na solução 5.

5. solution_5_recursion() e solve_5()

Esta solução é obtida através das informações dadas pelas soluções anteriores, sendo que todas estas foram evoluções passo a passo, com o intuito de obter esta solução final, em programação dinâmica.

A **solution_5_recursion** é bastante similar á anterior, **solution_4_recursion**, em que também é usada uma variável global **sol_found** para saber quando é encontrada uma primeira solução, e um *array* multidimensional que guarda informações para mais tarde descobrir a melhor solução sem ter de fazer a recursão por inteiro.

Inicialmente, começa por declarar as variáveis **i**, **k** e **new_speed**, em que as primeiras duas são usadas em *loops* e blocos condicionais e a segunda guarda o novo valor da velocidade, **speed**.

De seguida, tal como na solução anterior, é iniciado um *loop for*, que tem como objetivo, preencher o *array* bidimensional **solution_5.move_speed[][]** (declarado como parâmetro da *struct* acima definida), com os valores da quantidade de movimentos e a velocidade na posição atual para as posições que o carro passou mas não parou, exemplificado na solução acima apresentada.

A variável **solution5_count** (*effort*) é incrementada e a posição atual é colocada no *array* **solution_5.position**.

Seguidamente, é iniciado um bloco condicional para verificar se a recursão atual é uma solução. Se a posição atual corresponde á posição final e a sua velocidade é igual a 1, então representa uma solução. Ainda dentro desta condição, outro *if*, em que, caso esta solução tenha um menor número movimentos que a melhor solução até então encontrada (**solution_5.best.n_moves**), a melhor solução passa a ser a agora descoberta. É também altera a variável **sol_found**, que recebe agora o valor 1, indicando que já foi encontrada a primeira solução (pode ser a melhor ou não).

Posteriormente, tal como em todas as outras soluções, é iniciado um *loop for* com o valor da nova velocidade, seguido de condições para verificar se esta velocidade pode ou não ser utilizada. Em caso positivo, realiza-se a recursão.

A diferença com a solução anterior é que a verificação feita com o *array* bidimensional iniciado e preenchido anteriormente, **move_speed**, apenas é usado para verificar se a nova solução é melhor ou não depois de a recursão ser chamada.

O grupo concluiu que para a solução ser mais rápida e eficiente que a anterior, as verificações com os blocos condicionais tinham de ser feitas a seguir á chamada da recursão, uma vez que, quando é encontrada a primeira solução, a função dá **return**, ou seja volta para onde aconteceu a recursão.

As verificações só se vão efetuar caso a primeira solução já tenha sido encontrada, ou seja, caso a variável **sol_found** apresente o valor 1. De seguida, usando o *array* que guarda o número de movimentos e a velocidade de cada posição, da melhor solução, **solution_5.move_speed**, são feitas as comparações, em que caso a nova solução tenha um maior número de movimentos ou uma menor velocidade numa determinada posição, a função dá **return**, pois esta solução é pior que a melhor até então encontrada, pelo que, não vale a pena continuar. Se esta condição não se verificar, é feita outra recursão, com os valores desta nova solução.

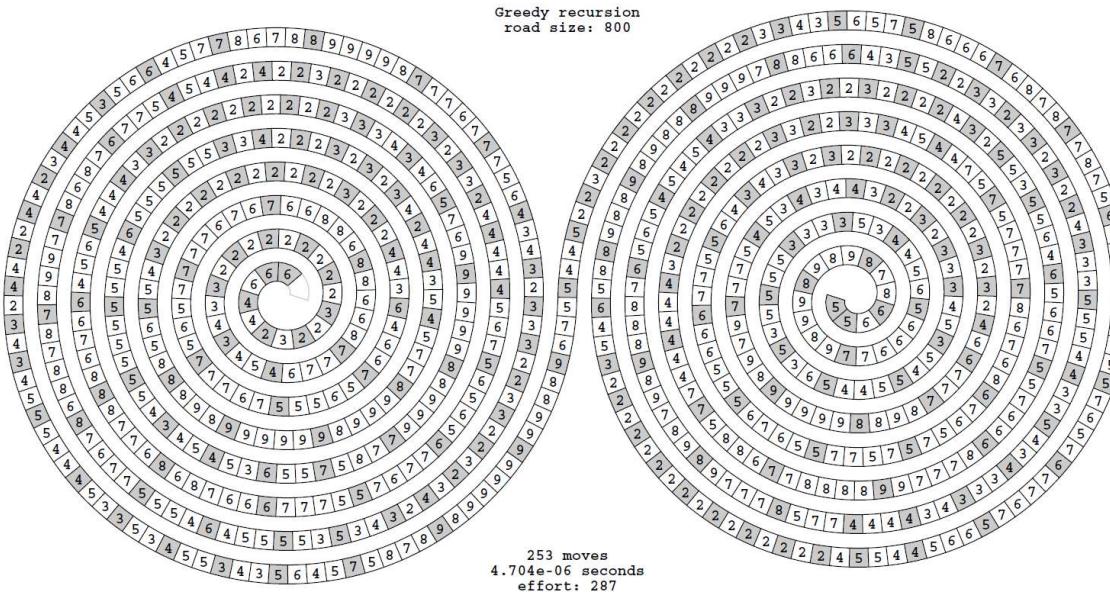


Figura 9 - 253 movimentos na solução 5

Como é possível verificar, o “effort” desta solução é bastante pequeno, apenas 287 quando o carro vai ter 253 movimentos. Esta pequena diferença corresponde aos ajustes que o carro tem de realizar quando a sua velocidade não pode ser usada em determinado ponto da estrada.

Esta solução recursiva realizada em programação dinâmica (uma vez que acontece o *memoization* dos valores intermédios), é, por sua vez, bastante rápida, realizando esta solução com posição final de 800 em apenas 4.404e-06 segundos, sendo, portanto, a solução mais eficiente encontrada.

Resultados obtidos

Os resultados obtidos estão representados em forma de **pdfs** e de gráficos.

Os **pdfs** são apenas os das soluções com os números mecanográficos dos alunos, uma vez que os base estão apresentados ao longo do relatório.

Os gráficos que possuem números mecanográficos foram obtidos nos computadores respetivos de cada aluno, os restantes são das soluções base.

Solução 1:

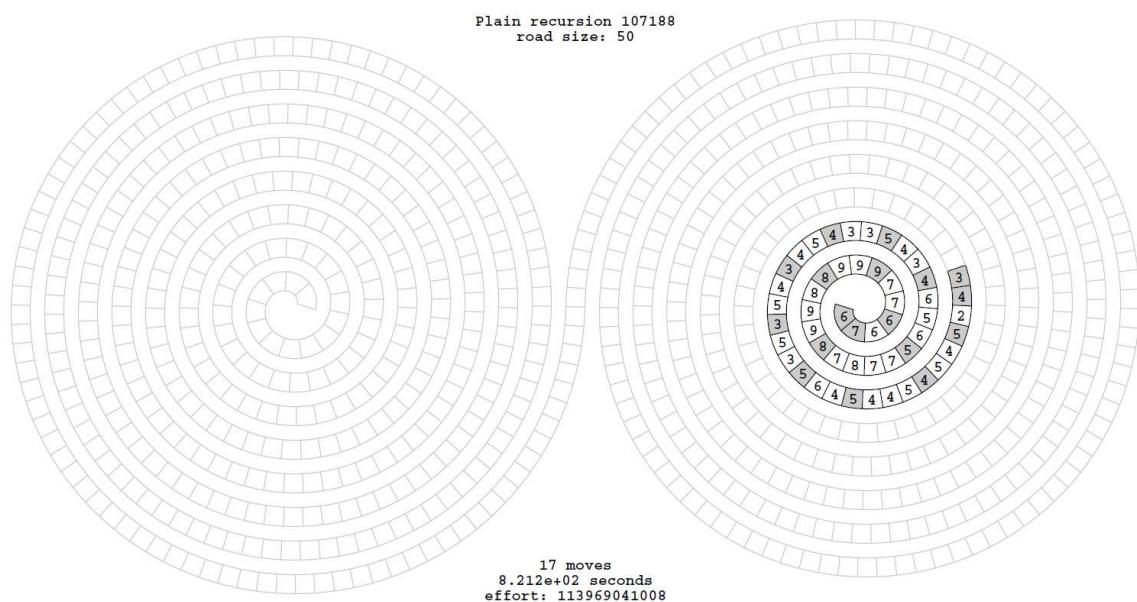


Figura 10 - 17 movimentos na solução 1 com número mecanográfico 107188

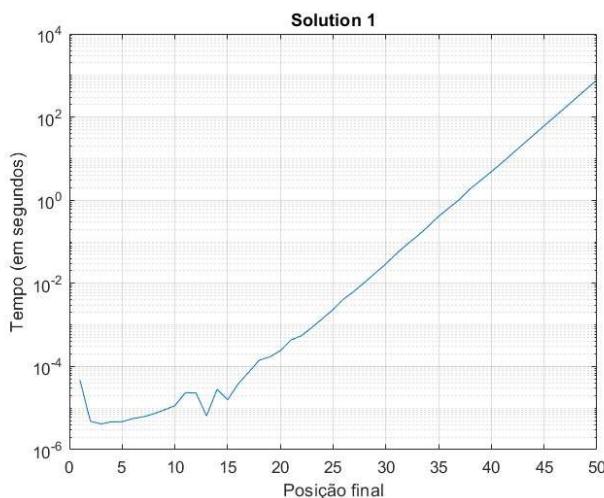


Figura 11 - Posição no tempo na solução 1

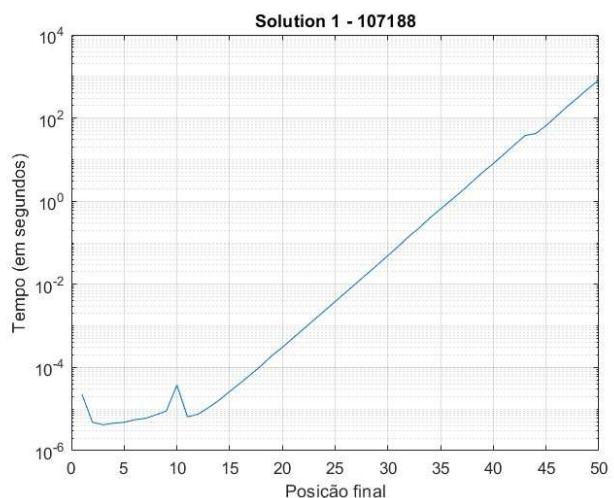


Figura 12 - Posição no tempo na solução 1
Nmec 107188

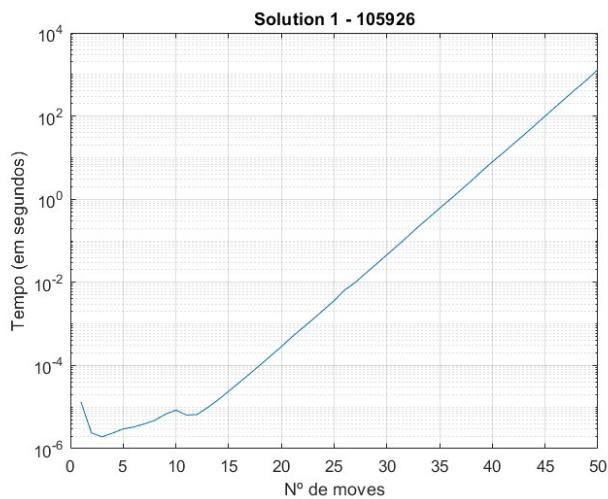


Figura 13 - Posição no tempo na solução 1 Nmec 105926

Solução 2:

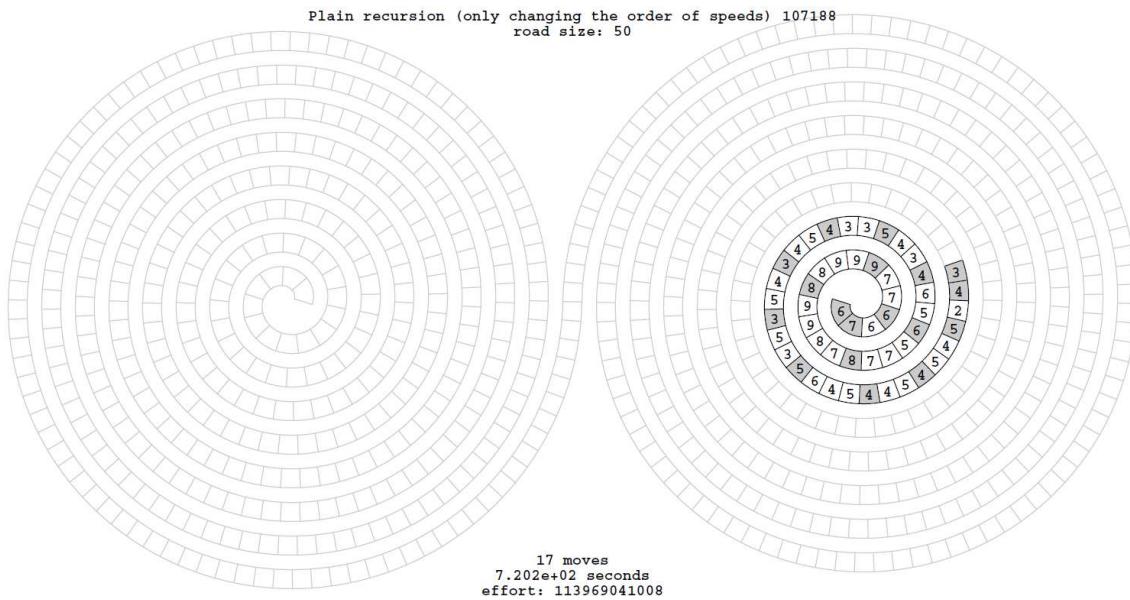


Figura 14 - 17 movimentos na solução 2 com número mecanográfico 107188

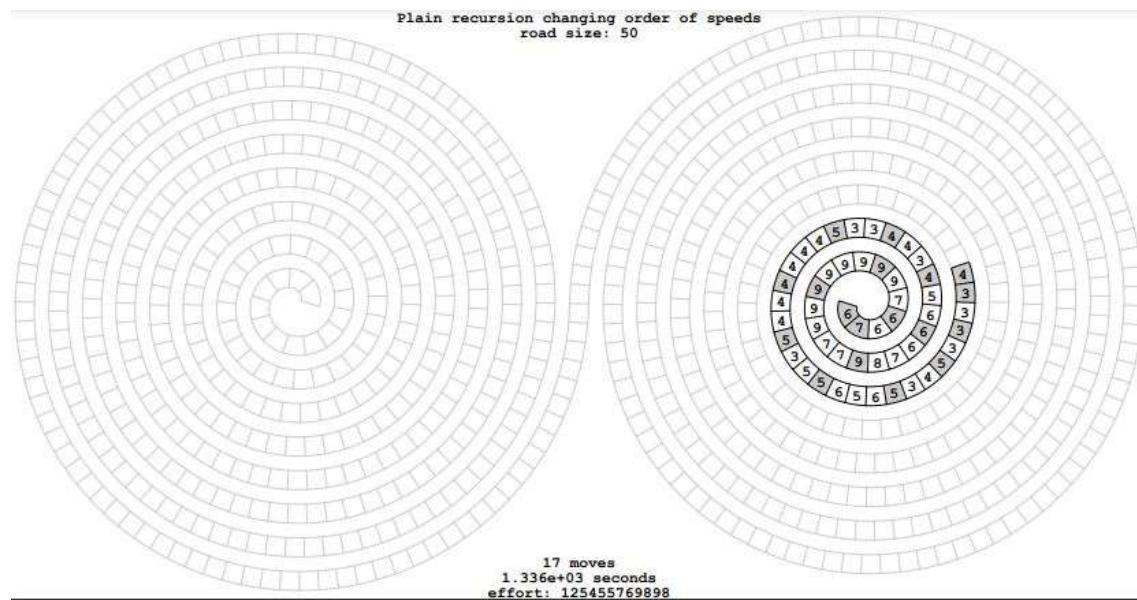


Figura 15 - 17 movimentos na solução 2 com número mecanográfico 105926

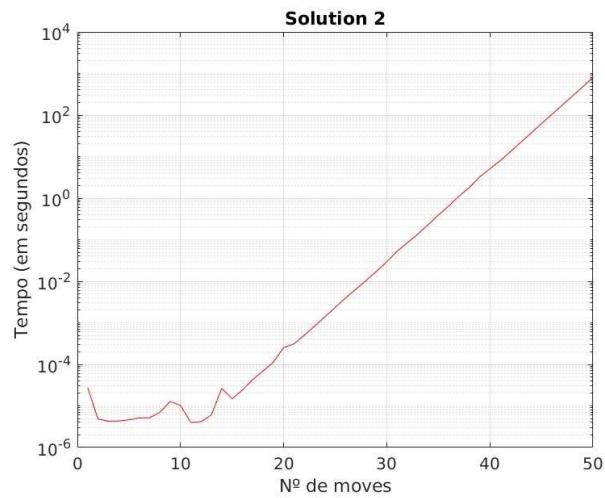


Figura 16 - Posição no tempo na solução 2

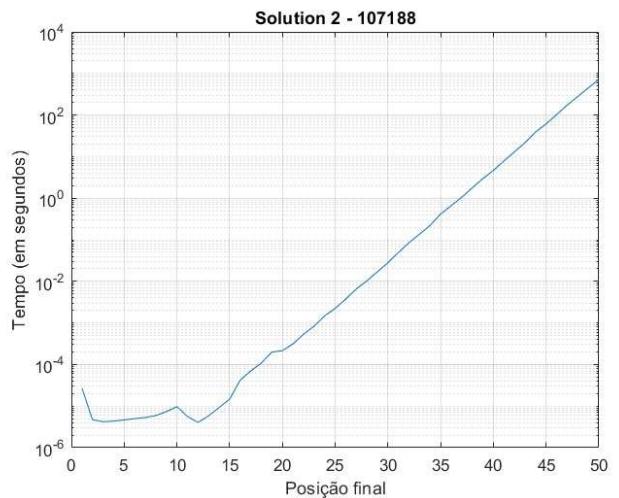


Figura 17- Posição no tempo na solução 2 com Nmec 107188

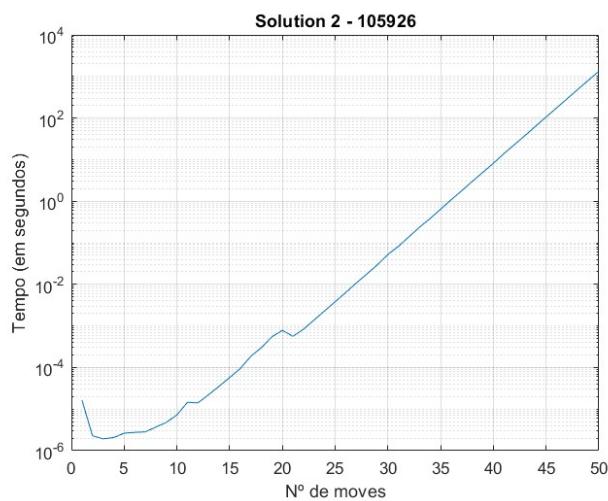


Figura 18 - Posição no tempo na solução 2 com Nmec 105926

Solução 3:

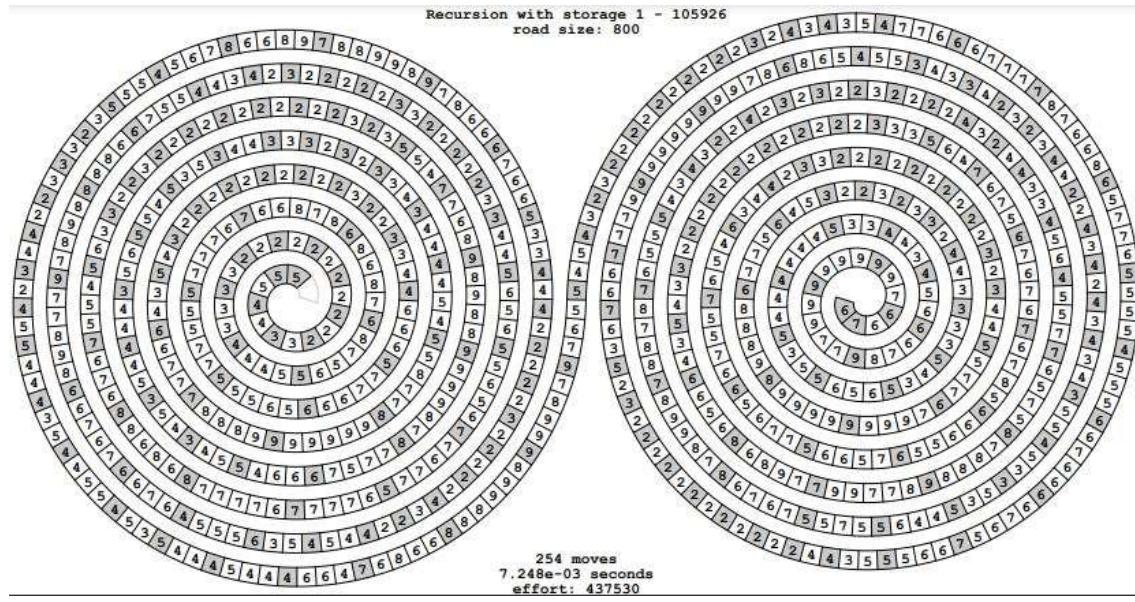


Figura 19 - 254 movimentos na solução 3 com Nmec 105926

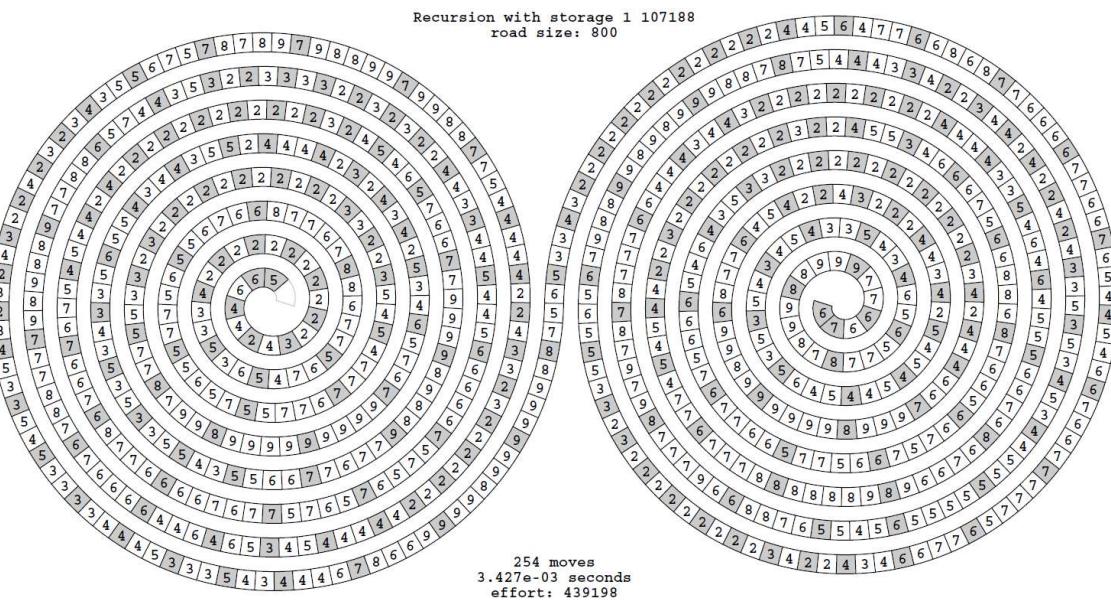


Figura 20 - 254 movimentos na solução 3 com Nmec 107188

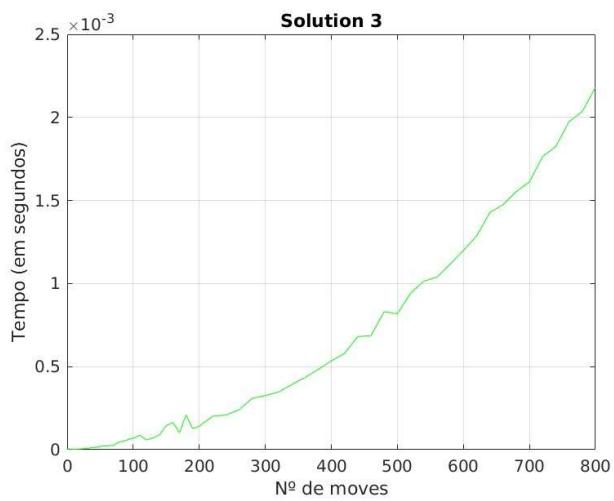


Figura 21 - Posição no tempo na solução 3

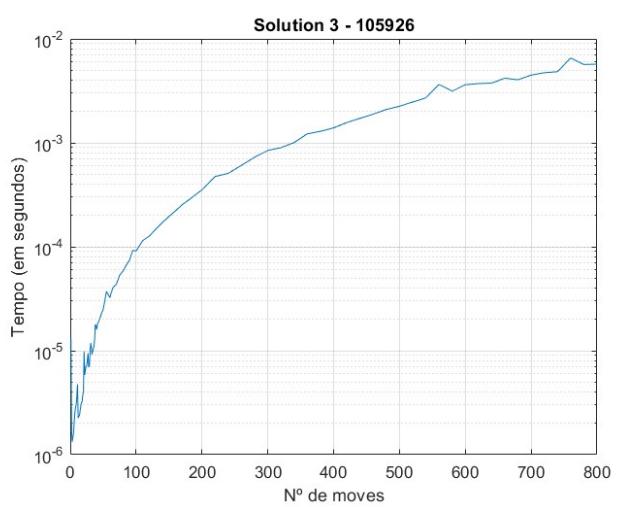


Figura 22 - Posição no tempo na solução 3 com Nmec 105926

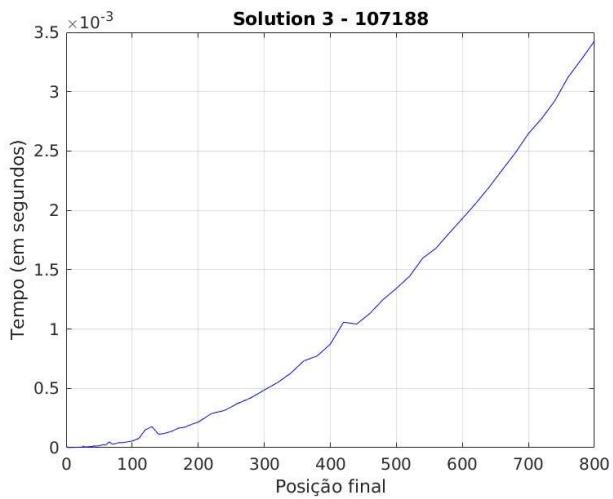


Figura 23 - Posição no tempo na solução 3 com Nmec 107188

Solução 4:

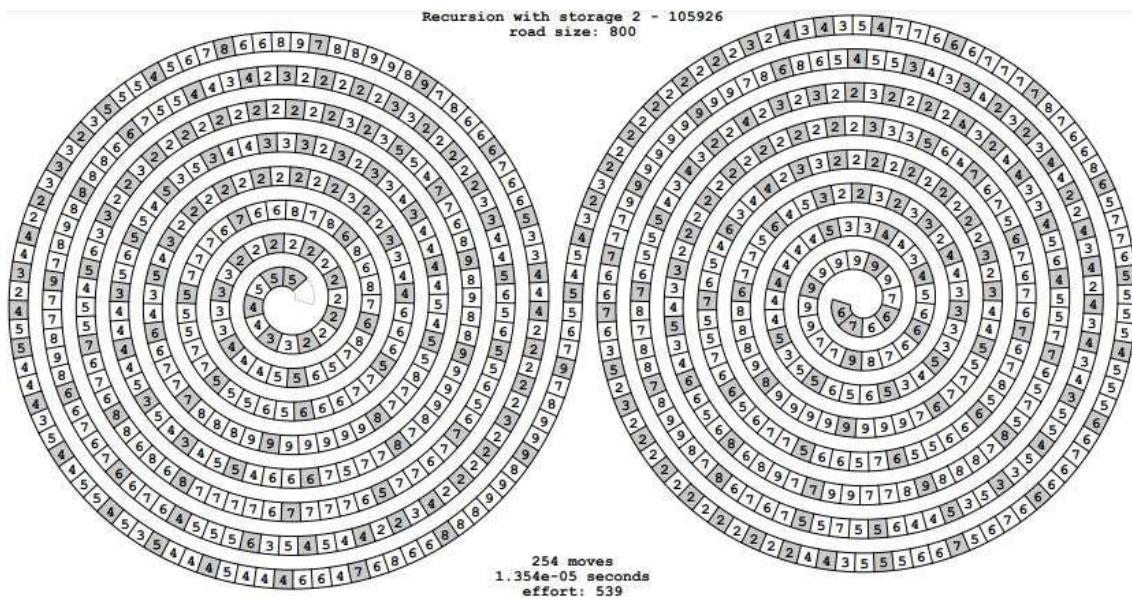


Figura 24 - 254 movimentos na solução 4 com Nmec 105926

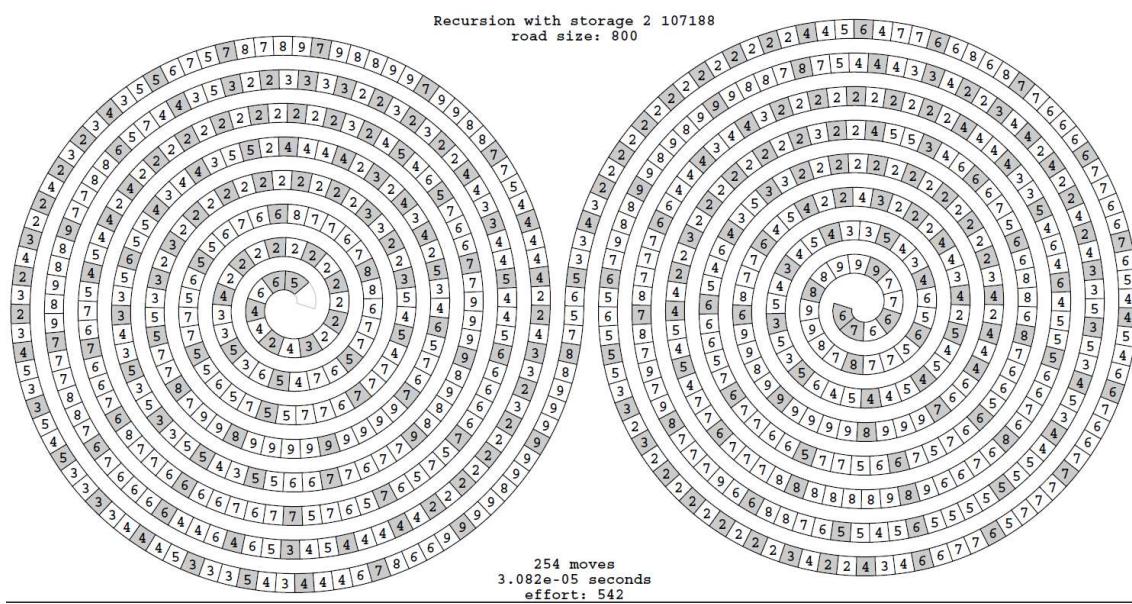


Figura 25 - 254 movimentos na solução 4 com Nmec 107188

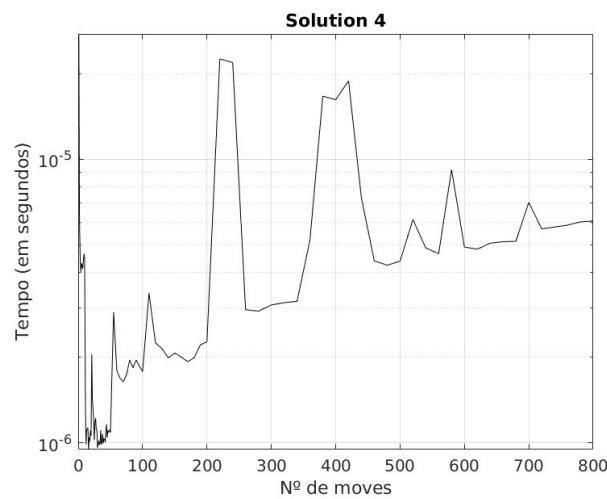


Figura 26- Posição no tempo na solução 4

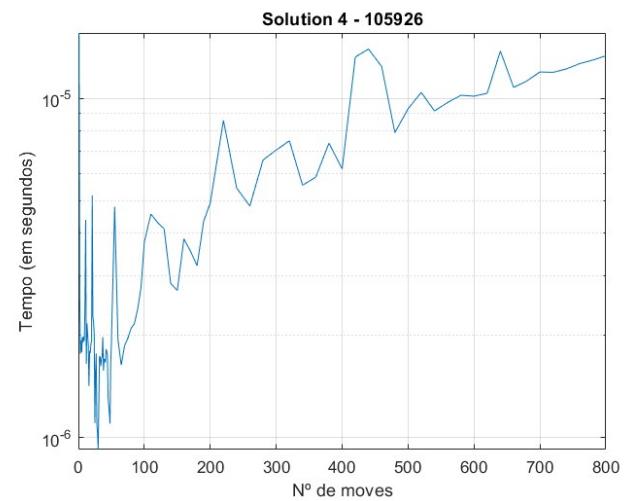


Figura 27 - Posição no tempo na solução 4 com Nmec
105926

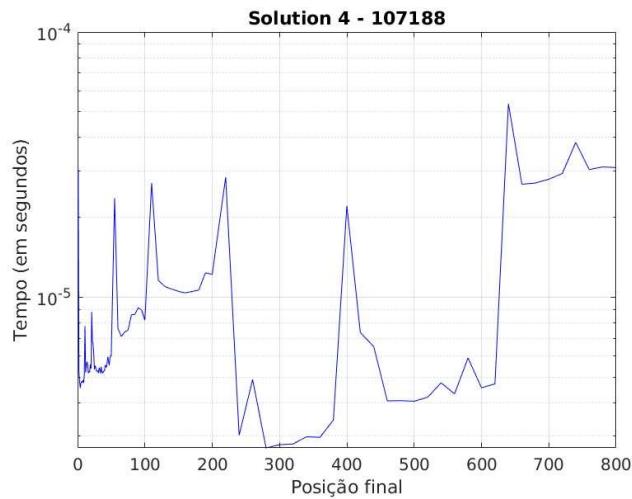


Figura 28 - Posição no tempo na solução 4 com Nmec
107188

Solução 5:

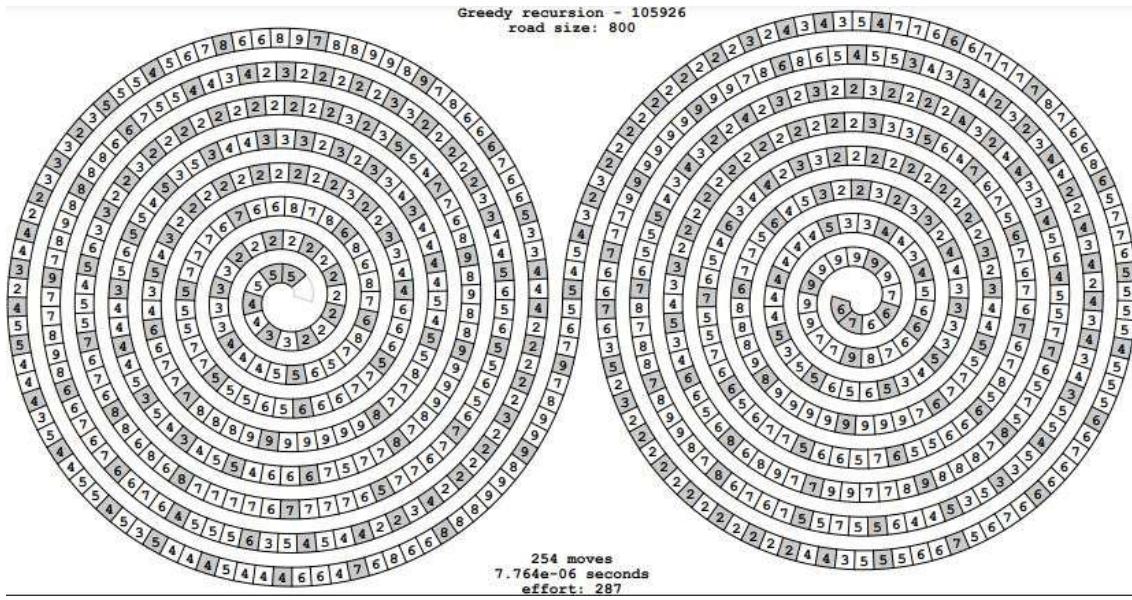


Figura 29 - 254 movimentos na solução 5 com Nmec 105926

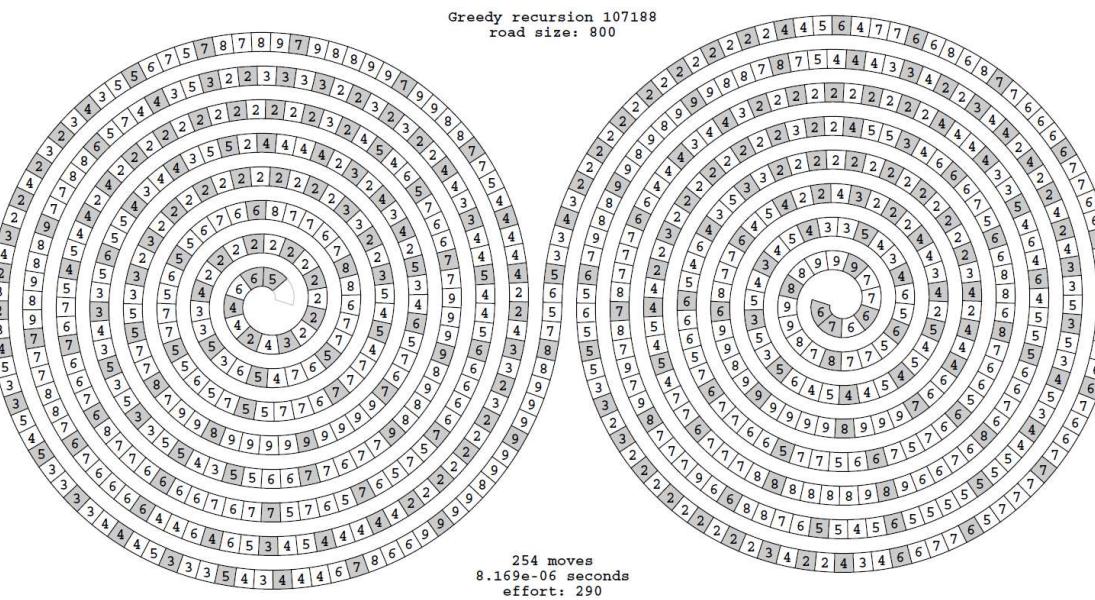


Figura 30 - 254 movimentos na solução 5 com Nmec 107188

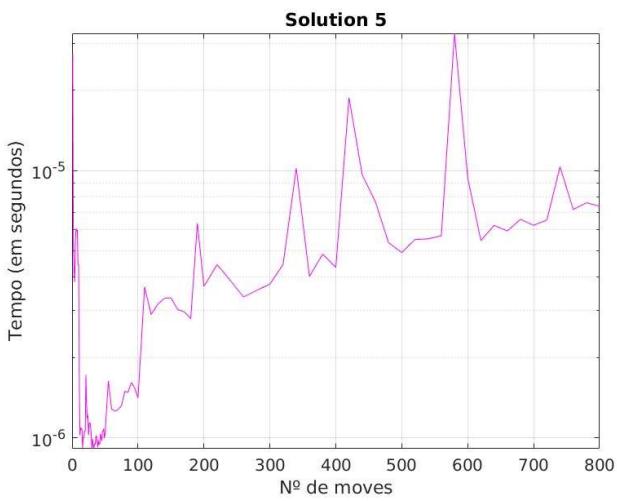


Figura 31 - Posição no tempo na solução 5

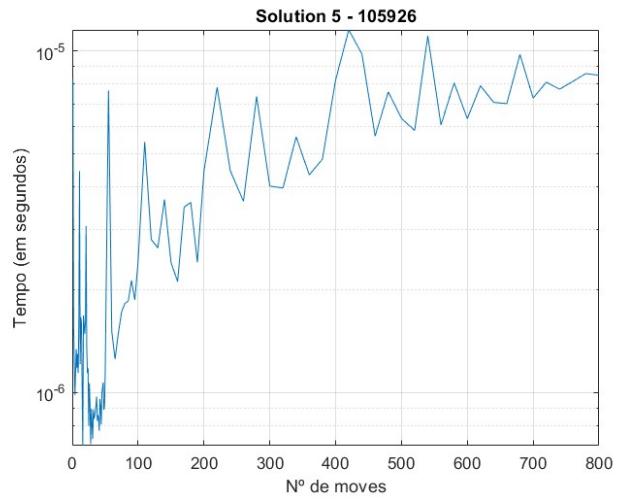


Figura 32 - Posição no tempo na solução 5 com Nmec
105926

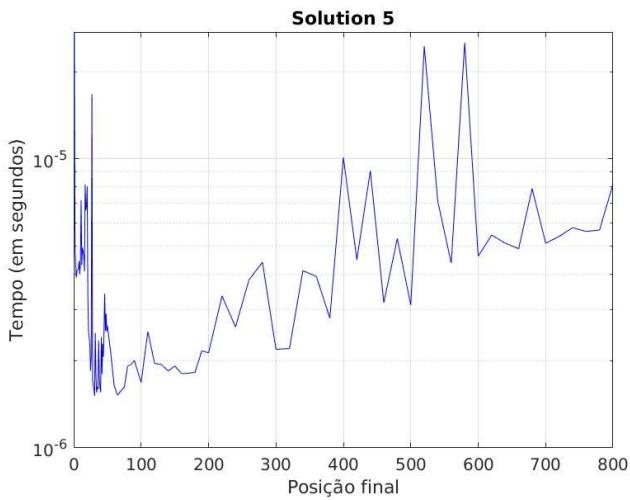


Figura 33 - Posição no tempo na solução 5 com Nmec
107188

Comparações entre soluções:

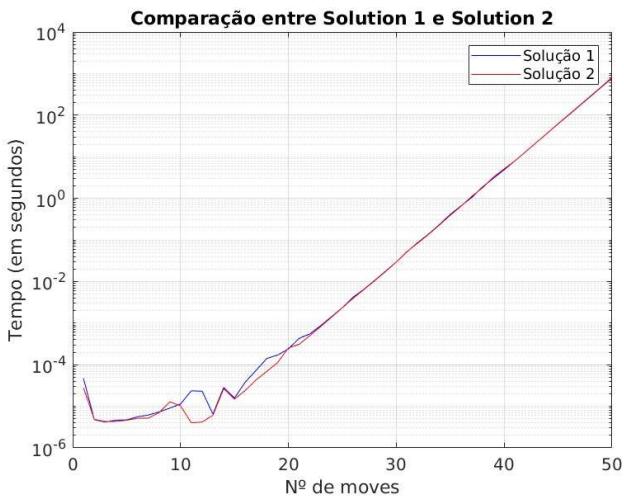


Figura 34 – Comparação entre Solução 1 e 2

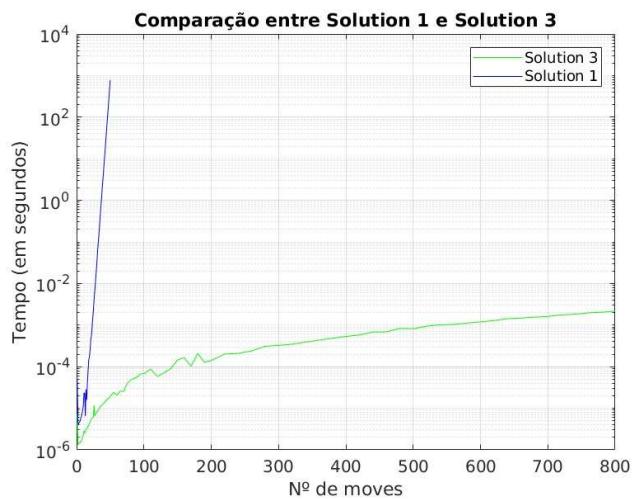


Figura 35 – Comparação entre Solução 1 e 3

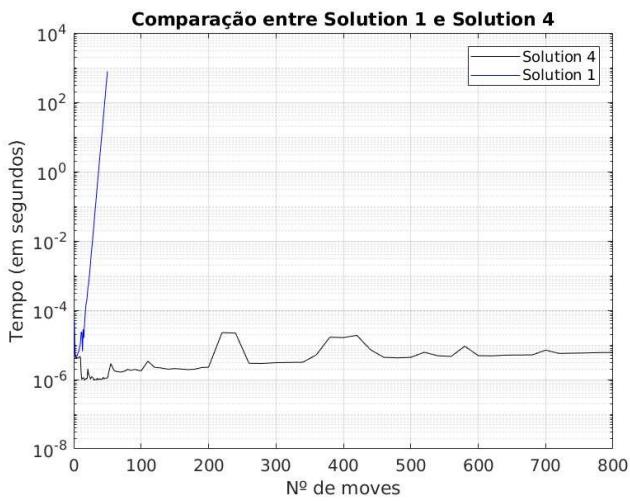


Figura 36 – Comparação entre Solução 1 e 4

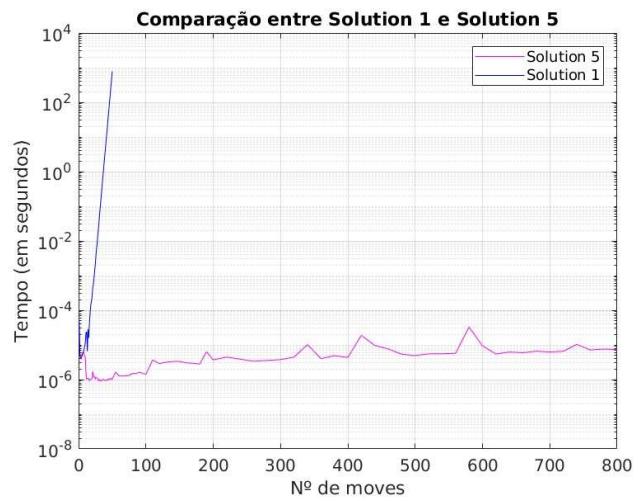


Figura 37 – Comparação entre Solução 1 e 5

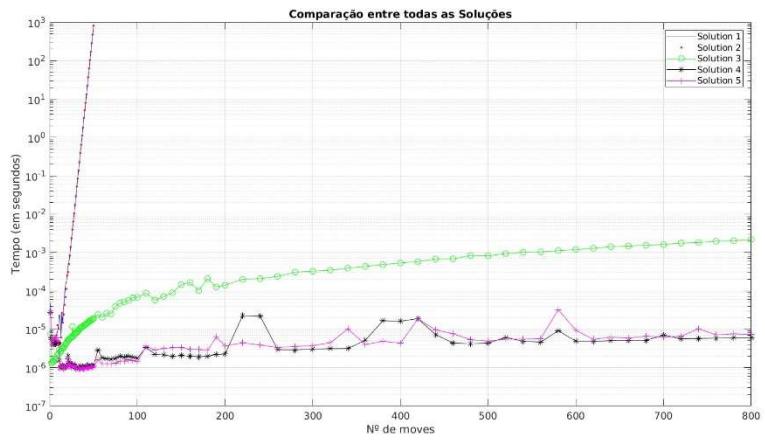


Figura 38 – Comparação entre Todas as Soluções

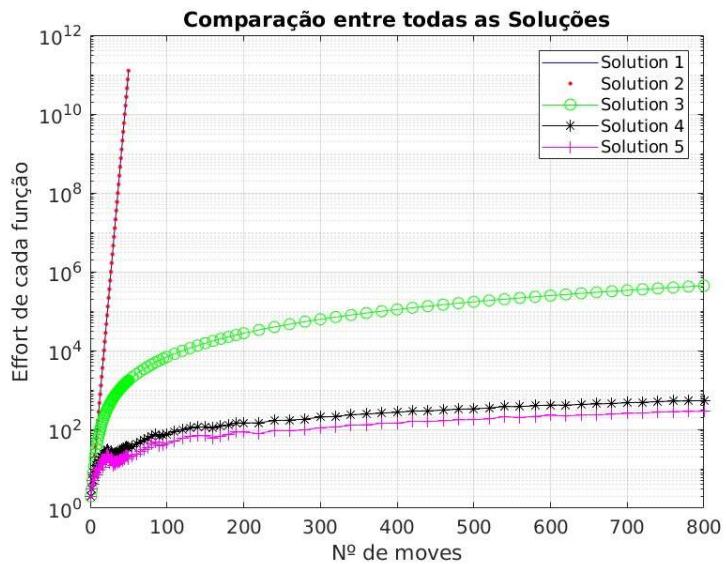


Figura 39 – Comparação entre o Effort de Todas as Soluções

Conclusão

Em suma, este primeiro trabalho prático demonstrou que apesar de um método para a resolução de um problema funcionar como aquilo que é pedido, existem maneiras melhores para o resolver, de forma mais eficiente e muito mais rápida. Este efeito é visível ao longo deste relatório, em que a cada solução é melhorado aos poucos uma solução para o problema “Speed Run”.

Este permitiu perceber melhor a linguagem C, as funções recursivas, e os algoritmos que permitem solucionar o mesmo problema. Inclusive, permitiu aprender sobre programação dinâmica, um conceito bastante útil em programação e que auxiliou na resolução deste problema muito mais rapidamente.

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. XXXXXX Name: XXXXXXXX  
//  
  
//  
// static configuration  
//  
  
#define _max_road_size_ 800 // the maximum problem size  
#define _min_road_speed_ 2 // must not be smaller than 1, shouldnot be  
// smaller than 2  
#define _max_road_speed_ 9 // must not be larger than 9 (only because of  
// the PDF figure)  
  
//  
// include files --- as this is a small project, we include the PDF generation  
// code directly from make_custom_pdf.c  
//  
  
#include <math.h>  
#include <stdio.h>  
#include "../P02/elapsed_time.h"  
#include "make_custom_pdf.c"
```

```

//  

// road stuff  

//  
  

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

0.._max_road_size_  
  

static void init_road_speeds(void)  

{  

    double speed;  

    int i;  
  

    for(i = 0;i <= _max_road_size_;i++)  

    {  

        speed = (double)_max_road_speed_ * (0.55 + 0.30 * sin(0.11 * (double)i) +  

        0.10 * sin(0.17 * (double)i + 1.0) + 0.15 * sin(0.19 * (double)i));  

        max_road_speed[i] = (int)floor(0.5 + speed) + (int)((unsigned int)random()  

        % 3u) - 1;  

        if(max_road_speed[i] < _min_road_speed_)  

            max_road_speed[i] = _min_road_speed_;  

        if(max_road_speed[i] > _max_road_speed_)  

            max_road_speed[i] = _max_road_speed_;  

    }  

}  
  

//  

// description of a solution  

//  
  

typedef struct  

{  

    int n_moves; // the number of moves (the number of  

positions is one more than the number of moves)  

    int positions[1 + _max_road_size_]; // the positions (the first one must be  

zero)  

    int move_speed[1 + _max_road_size_][2]; // to register the move number and  

speed in the position
}

```

```

}

solution_t;

//


// the (very inefficient) recursive solution given to the students
//


static solution_t
solution_1,solution_1_best,solution_2,solution_2_best,solution_3,solution_3_be
st,solution_4,solution_4_best,solution_5,solution_5_best;//solution_6,solutio
n_6_best;

static double solution_1_elapsed_time,
solution_2_elapsed_time,solution_3_elapsed_time,solution_4_elapsed_time,soluti
on_5_elapsed_time; //solution_6_elapsed_time; // time it took to solve the
problem

static unsigned long solution_1_count,
solution_2_count,solution_3_count,solution_4_count,solution_5_count;//solution
_6_count; // effort dispended solving the problem

static int sol_found;

static int best_moves_speed[_max_road_size_][2]; // apenas vai ser usado na
solution_3


static void solution_1_recursion(int move_number,int position,int speed,int
final_position)

{
    int i,new_speed;

    // record move

    solution_1_count++;

    solution_1.positions[move_number] = position;

    // is it a solution?

    if(position == final_position && speed == 1)
    {
        // is it a better solution?

        if(move_number < solution_1_best.n_moves)
        {

            solution_1_best = solution_1;

            solution_1_best.n_moves = move_number;
        }
    }
}

```

```

    return;
}

// no, try all legal speeds

for(new_speed = speed - 1;new_speed <= speed + 1;new_speed++)
{
    if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed
    <= final_position)
    {

        for(i = 0;i <= new_speed && new_speed <= max_road_speed[position +
        i];i++)
        {

            if(i > new_speed)

                solution_1_recursion(move_number + 1,position +
                new_speed,new_speed,final_position);

        }
    }
}

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

    // record move

    solution_2_count++;

    solution_2.positions[move_number] = position;

    // is it a solution?

    if(position == final_position && speed == 1)
    {

        // is it a better solution?

        if(move_number < solution_2_best.n_moves)
        {

            solution_2_best = solution_2;
            solution_2_best.n_moves = move_number;
        }
    }

    return;
}

// no, try all legal speeds

for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--) // apenas é
alterado a velocidade, para testar primeiro a mais rápida (não altera nada na
solução)

```

```

        if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed
<= final_position) // continua a fazer todas as possibilidades (é necessario
cortar os ramos)

    {

        for(i = 0;i <= new_speed && new_speed <= max_road_speed[position +
i];i++)

        {

            if(i > new_speed)

                solution_2_recursion(move_number + 1,position +
new_speed,new_speed,final_position);

        }

    }

static void solution_3_recursion(int move_number,int position,int speed,int
final_position)

{

    int i,new_speed;

    // record move

    solution_3_count++;

    solution_3.positions[move_number] = position;

    // is it a solution?

    if(position == final_position && speed == 1)

    {

        // is it a better solution?

        if(move_number < solution_3_best.n_moves)

        {

            solution_3_best = solution_3;

            solution_3_best.n_moves = move_number;

        }

        return;

    }

    if(best_moves_speed[position][0] >= move_number &&
best_moves_speed[position][1] <= speed){

        return;

    }

    else{

        best_moves_speed[position][0] = move_number;

```

```

    best_moves_speed[position][1] = speed;
}

// no, try all legal speeds

for(new_speed = speed + 1; new_speed >= speed - 1; new_speed--) // apenas é
alterado a velocidade, para testar primeiro a mais rápida (não altera nada na
solução)

if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed
<= final_position) // continua a fazer todas as possibilidades (é necessário
cortar os ramos)

{
    for(i = 0; i <= new_speed && new_speed <= max_road_speed[position +
i]; i++)

    ;
    if(i > new_speed)

        solution_3_recursion(move_number + 1, position +
new_speed, new_speed, final_position);

}
}

// solução 3 --> fazer com array em que as speeds são preenchidas com 0 e os
moves com maxroadsize e fazer as comparações com esse

// assim em princípio não dá erro

static void solution_4_recursion(int move_number, int position, int speed, int
final_position)

{
    int i, k, new_speed, count_speed;

    for(k = position; k > position - speed; k--)
    {
        solution_4.move_speed[k][0] = move_number;
        solution_4.move_speed[k][1] = speed;
    }

    // record move
    solution_4_count++;
    solution_4.positions[move_number] = position;
    // is it a solution?

    if(position == final_position && speed == 1)
    {

```

```

// is it a better solution?

if(move_number < solution_4_best.n_moves)
{
    solution_4_best = solution_4;
    solution_4_best.n_moves = move_number;
}

sol_found = 1;

return;
}

if(sol_found == 1){

    if(solution_4_best.move_speed[position + 1][0] <= move_number + 1 ||
solution_4_best.move_speed[position + 1][1] >= new_speed) // fazer testes para
verificar se os novos valores sao meljore

    return;
}

// no, try all legal speeds

for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--) // apenas é
alterado a velocidade, para testar primeiro a mais rapida (nao altera nada na
solucao)

    if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed
<= final_position) // continua a fazer todas as possibilidades (é necessario
cortar os ramos)

    {

        // count_speed = (new_speed * (new_speed + 1) / 2);

        // // count_speed vai ser o valor da nova velocidade mais as velocidades
menores

        // if(count_speed + position > final_position)

        // {

        //     return;
        // }

        for(i = 0;i <= new_speed && new_speed <= max_road_speed[position +
i];i++)
        ;
        if(i > new_speed)

            solution_4_recursion(move_number + 1,position +
new_speed,new_speed,final_position);
}

```

```

        }

    }

static void solution_5_recursion(int move_number,int position,int speed,int
final_position)

{
    int i,k,new_speed;

    for(k = position; k > position-speed; k--)
    {
        solution_5.move_speed[k][0] = move_number;
        solution_5.move_speed[k][1] = speed;
    }

    // record move
    solution_5_count++;
    solution_5.positions[move_number] = position;

    // is it a solution?
    if(position == final_position && speed == 1)
    {
        // is it a better solution?
        if(move_number < solution_5_best.n_moves)
        {
            solution_5_best = solution_5;
            solution_5_best.n_moves = move_number;
        }
        sol_found = 1;
    }
    return;
}

// no, try all legal speeds
for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--)
{

```



```

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;
}

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_recursion(0,0,0,final_position);
    solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}

static void solve_3(int final_position)
{
    int i;
    if(final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr,"solve_3: bad final_position\n");
        exit(1);
    }
}

```

```

for(i = 0; i < _max_road_size_; i++)
{
    best_moves_speed[i][0] = 0; // speed
    best_moves_speed[i][1] = _max_road_size_; // moves
}

solution_3_elapsed_time = cpu_time();
solution_3_count = 0ul;
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 = 0ul;
    solution_4_best.n_moves = final_position + 100;
    sol_found = 0;
    solution_4_recursion(0,0,0,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 = 0ul;
    solution_5_best.n_moves = final_position + 100;
}

```

```

sol_found = 0;

solution_5_recursion(0,0,0,final_position);

// for(int l = 0; l < _max_road_size_ + 1; l++)
//   printf("%d-%d ", solution_5.move_speed[l][0],
solution_5.move_speed[l][1]); // movimentos-speed

solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;

}

//


// example of the slides
//


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

    srand(0xAED2022);

    init_road_speeds();

    final_position = 30;

    solve_1(final_position);

    make_custom_pdf_file("example.pdf",final_position,&max_road_speed[0],solution_1_best.n_moves,&solution_1_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_4_elapsed_time = 0.0;  

    printf("      + ----- +\n");  

    printf("      |           greedy 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_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_be
st.n_moves,&solution_4_best.positions[0],solution_4_elapsed_time,solution_4_co
unt,"Recursion with storage 2");

    }

    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("                                |");

}

// 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

Apenas serão introduzidos os códigos das soluções base (sem número mecanográfico) e comparações, uma vez que os restantes são apenas alterações no ficheiro introduzido.

```
A = load("data_solution1.txt");

n = A(:,1);
t = A(:,4);

plot(n,t)
grid on;

semilogy(n,t);
title("Solution 1")
xlabel("Posição final")
ylabel("Tempo (em segundos)")
grid;
figure
plot(n,log10(t))

t_log = log10(t);

N = [n(20:end) 1+0*n(20:end)];
Coefs = pinv(N)*t_log(20:end);

hold on
grid;
Ntotal = [n n*0+1];
plot(n,Ntotal*Coefs, 'k');
title("Solution 1")
xlabel("Posição final")
ylabel("Tempo no logaritmo de base 10 (em segundos)")

t800_log = [800 1] * Coefs;
t800 = 10^t800_log / 3600 / 24 / 365
```

Figura 40 – Gráfico da solução 1 e calculo da equação que permite calcular aproximadamente o tempo numa dada posição final

```

A = load("data_solution2.txt");
n = A(:,1);
t = A(:,4);

plot(n,t)
grid on;

semilogy(n,t, 'r-');
title("Solution 2")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")
grid on;



---


%% comparação entre a solution 1 e a 2

B = load("data_solution1.txt");
n1 = B(:,1);
t1 = B(:,4);

figure(2)
semilogy(n1,t1, 'b-');
hold on;
semilogy(n,t, 'r-')
legend('Solução 1', 'Solução 2')
title("Comparação entre Solution 1 e Solution 2")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")
grid on;

```

Figura 41 – Gráfico da solução 2 e comparação com o gráfico 1

```

sol3 = load("data_solution3.txt");

n_sol3 = sol3(:,1);
t_sol3 = sol3(:,4);

plot(n_sol3,t_sol3, 'g-')
title("Solution 3")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")
grid on;



---


%% comparação com a solution 1

sol1 = load("data_solution1.txt");

n_sol1 = sol1(:,1);
t_sol1 = sol1(:,4);
figure(2)
semilogy(n_sol3,t_sol3, 'g-')
hold on;
semilogy(n_sol1, t_sol1, 'b-');
grid on;
legend('Solution 3', 'Solution 1')
title("Comparação entre Solution 1 e Solution 3")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")

```

Figura 42 – Gráfico da solução 3 e comparação com o gráfico 1

```

sol4 = load("data_solution4.txt");

n_sol4 = sol4(:,1);
t_sol4 = sol4(:,4);

semilogy(n_sol4,t_sol4, 'k-')
title("Solution 4")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")
grid on;

%% comparação com a solution 1

sol1 = load("data_solution1.txt");

n_sol1 = sol1(:,1);
t_sol1 = sol1(:,4);
figure(2)
semilogy(n_sol4,t_sol4, 'k-')
hold on;
semilogy(n_sol1, t_sol1, 'b-');
grid on;
legend('Solution 4', 'Solution 1')
title("Comparação entre Solution 1 e Solution 4")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")

```

Figura 43 – Gráfico da solução 4 e comparação com o gráfico 1

```

sol5 = load("data_solution5.txt");

n_sol5 = sol5(:,1);
t_sol5 = sol5(:,4);

semilogy(n_sol5,t_sol5, 'm-')
title("Solution 5")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")
grid on;

%% comparação com a solution 1

sol1 = load("data_solution1.txt");

n_sol1 = sol1(:,1);
t_sol1 = sol1(:,4);
figure(2)
semilogy(n_sol5,t_sol5, 'm-')
hold on;
semilogy(n_sol1, t_sol1, 'b-');
grid on;
legend('Solution 5', 'Solution 1')
title("Comparação entre Solution 1 e Solution 5")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")

```

Figura 44 – Gráfico da solução 5 e comparação com o gráfico 1

```

sol1 = load("data_solution1.txt");
sol2 = load("data_solution2.txt");
sol3 = load("data_solution3.txt");
sol4 = load("data_solution4.txt");
sol5 = load("data_solution5.txt");

n_sol1 = sol1(:,1);
n_sol2 = sol2(:,1);
n_sol3 = sol3(:,1);
n_sol4 = sol4(:,1);
n_sol5 = sol5(:,1);

t_sol1 = sol1(:,4);
t_sol2 = sol2(:,4);
t_sol3 = sol3(:,4);
t_sol4 = sol4(:,4);
t_sol5 = sol5(:,4);

semilogy(n_sol1,t_sol1, 'b-')
hold on;
semilogy(n_sol2,t_sol2, 'r.')
semilogy(n_sol3,t_sol3, 'go-')
semilogy(n_sol4,t_sol4, 'k*-')
semilogy(n_sol5,t_sol5, 'm+-')
hold off;

legend("Solution 1", "Solution 2", "Solution 3", "Solution 4", "Solution 5");
title("Comparação entre todas as Soluções")
xlabel("Nº de moves")
ylabel("Tempo (em segundos)")
grid on

```

Figura 45 – Gráfico da comparação entre soluções

```

%% com o n moves e o effort

e_sol1 = sol1(:,3);
e_sol2 = sol2(:,3);
e_sol3 = sol3(:,3);
e_sol4 = sol4(:,3);
e_sol5 = sol5(:,3);

figure(2)
semilogy(n_sol1,e_sol1, 'b-')
hold on;
semilogy(n_sol2,e_sol2, 'r.')
semilogy(n_sol3,e_sol3, 'go-')
semilogy(n_sol4,e_sol4, 'k*-')
semilogy(n_sol5,e_sol5, 'm+-')
hold off;

legend("Solution 1", "Solution 2", "Solution 3", "Solution 4", "Solution 5");
title("Comparação entre todas as Soluções")
xlabel("Nº de moves")
ylabel("Effort de cada função")
grid on

```

Figura 45 – Gráfico da comparação entre soluções – effort