Speed Run

Algoritmos e Estruturas de Dados

2022/2023

Alexandre Cotorobai – 107849 (55%)

Vitalie Bologa – 107854 (45%)

universidade
de aveiro

Índice

Introdução 3
Abordagem 4
Sol 1 - Teacher's Solution (Unoptimized) 4
Sol 2 - Our solution from scratch 4
Sol 3 - Teacher Optimized Solution 8
Sol 4 - Our solution in Dynamic Programing 10
Analise em MatLab 12
Comparação entre Soluções nos PC1 e PC2 12
Análise ao desempenho da solução 1 14
Aproximação do elapsed_time da solução 1 para n=800 15
Resultados e algumas conclusões 16
Solução 1 diferente 16
Conclusão 17
Apêndice 18

Introdução

No âmbito da unidade curricular de Algoritmos e Estruturas de Dados, foi nos proposto um trabalho prático que consiste num segmento, que está dividido em 800 casas, em que cada casa tem um *speed_limit*, podendo variar entre 2 e 9 unidades. Essas casas serão percorridas por um carro, com velocidade não acima do limite imposto, que sai da primeira casa com velocidade 0 e chega à última com velocidade 1. O número de movimentos consiste nos avanços que o carro faz, com uma certa velocidade.

Objetivo deste trabalho será fazer o carro chegar ao final do percurso, com o menor número de passos possível, sem quebrar nenhuma das regras impostas. A primeira solução já nos é dada pelo professor, mas, devido a problemas mais à frente descritos, teremos de encontrar forma de a otimizar, assim como desenvolver novas soluções para resolver o problema.

Neste relatório será explicado a proposta de resolução dada pelo professor, as suas falhas e possíveis otimizações, assim como outras criadas com métodos de resolução diferentes, explicando todo o raciocínio que nos levou a tomar cada decisão. No final iremos comparar as várias propostas de resolução, os seus tempos de execução e eficiências, e apresentá-los graficamente para facilitar formar conclusões.

Abordagem

Sol 1 - Teacher's Solution (Unoptimized)

Depois da leitura do enunciado, fomos ler e executar o código do professor. Após algum tempo a correr verificamos que o tempo de execução é muito elevado, demorando muito tempo a encontrar cada solução.

Analisando o código, verificamos que o processo de procurar a melhor solução forma uma árvore ternária. Em que cada nó (que representa posição) dá origem a 3 ramos, correspondentes a cada uma das 3 possíveis mudanças de velocidade.

Dessa análise, vimos que a ordem com que se testa a mudança de velocidade (diminuir, manter ou aumentar) não é a maneira mais eficiente de resolver o problema. Pois, ao percorrer as velocidades de baixo para cima, a primeira solução obtida será o pior caso possível, que será percorrer todo o caminho com velocidade um. Partindo desta solução, serão encontrados novos caminhos para outras soluções, que possivelmente serão melhores, obrigando com que a melhor solução seja das últimas a ser encontrada. E assim sendo, a best_solution, vai mudar constantemente, ao longo do programa até que encontre todas as soluções possíveis, obrigando a percorrer todos (ou quase todos) os caminhos, o que torna a função mais lenta.

Tendo em vista, tratar-se de uma árvore ternária, podemos calcular o nível de complexidade desta função, sendo ela 3^n (visto que para cada nó existem 3 ramos, referentes à mudança de velocidade possíveis),

Decidimos deixar a solução do professor rodar, para descobrir em quanto tempo acabaria de executar. Após horas de execução, e longos períodos de análise, a função dá timeout e termina na posição 55. A causa será da ordem de validação de mudança de velocidade (e não só), o que pode justificar o motivo do número de chamadas à função ser tão elevado. Este ponto será analisado na solução 3 onde iremos alterar valores e adicionar condições para melhorar o desempenho da função.

Sol 2 - Our solution from scratch

Nesta resolução tentou-se encontrar o método mais eficiente que conseguíssemos imaginar. Começamos por analisar bem o intuito do trabalho, "percorrer o caminho no menor número de passos possível", à partida não haveria grande dificuldade, bastava subir a velocidade ao máximo possível e a solução seria óbvia, mas claro que não era assim tão simples, existiam limitações, cada casa possui um limite máximo de velocidade sobre o qual o carro não pode ultrapassar.

Visto isto, pensámos em simplesmente averiguar se para um carro subir para uma velocidade "v", nenhuma das "v" casas seguintes teria limite de velocidade inferior a "v".

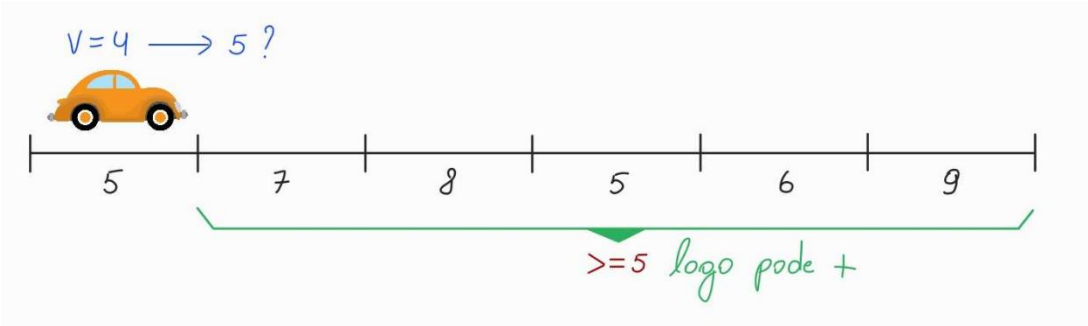


Figura 1 - Exemplo

Apesar deste raciocínio não ser algo inovador, visto que o mesmo está presente na própria solução dada pelo professor, achamos este seria um bom ponto de partida sobre o qual devíamos trabalhar e aprofundar.

À primeira vista este método parece satisfazer as condições impostas, mas quando se corre o programa vemos rapidamente que mesmo possui enormes falhas, isto, porque existe mais uma restrição no que toca a alterar velocidades, o carro apenas pode alterar no máximo em 1 a sua velocidade, ou seja, estando o carro numa dada casa a velocidade "v" o mesmo só poderá de lá sair com velocidades "v+1", "v" ou "v-1".

Em seguida iremos perceber em que é que essa restrição implica, usando parte do exemplo anterior.

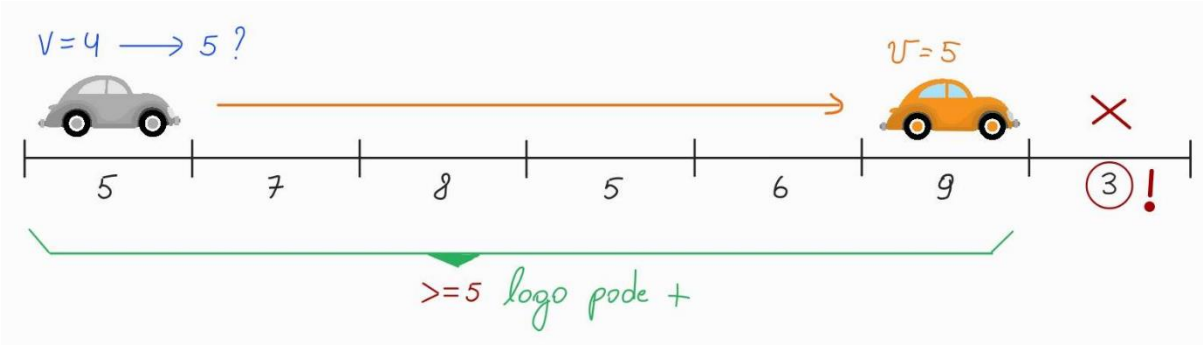


Figura 2 - Exemplo

Com base na figura, após aquela primeira validação ir-nos-ia dar como possível aumentar a velocidade para 5 pois nenhuma das 5 casas seguintes teria um limite inferior a esse número, mas ocorre o problema da 6ª casa conter limite 3 (neste exemplo), deste modo o carro estando na quinta casa não teria como diminuir a velocidade de modo a não quebrar o limite de velocidade pois só pode descer uma velocidade de cada vez.

De modo a prevenir que este caso acontecesse, e se agora também verificássemos as casas seguintes? E após verificar essas também verificássemos as próximas? E as depois dessas? E assim por diante.

Agora a única questão que não nos saía da cabeça era: E quantas casas precisaremos de analisar? Para termos certeza que não corríamos o risco de acontecer um caso em que o carro não conseguiria travar a tempo, como o descrito no esboço acima.

Foi neste momento que percebemos que havia um padrão no número de casas que teriam de ser obrigatoriamente analisadas. O grande entrave ficava no facto de, em cada move, o carro poder sair com 3 velocidades diferentes, mas descobrimos que afinal não era necessário analisar todas as possibilidades a cada move (subir, manter ou descer velocidade), mas sim apenas analisar no caso de descer a velocidade. Assim iamo-nos essencialmente preocupar, no pior caso, se o carro conseguiria abrandar a tempo sem infringir nenhum limite de velocidade.

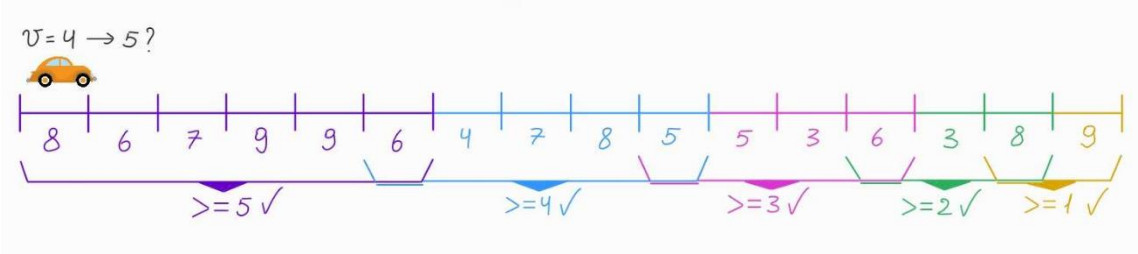


Figura 3 - Exemplo

Retomando o exemplo do carro em velocidade 4, para que saber se seria possível aumentar a velocidade para 5 sem nenhum impedimento, teríamos de primeiramente ver se nas 5 casas seguintes os limites são acima de 5, caso sejam teremos de ver as 4 casas seguintes (limites >=4), as 3 casas a seguir a essas (limites >=3), as duas depois dessas e a última casa (Esta última casa pode parecer desnecessária mas mais para a frente iremos explicar a sua importância na deteção do final do percurso).

Então e se em algum instante este esquema não se verificar? Fácil, visto que começamos sempre por verificar o caso de aumentar a velocidade, caso este não dê basta tentarmos fazer o mesmo para o caso do carro manter a velocidade.

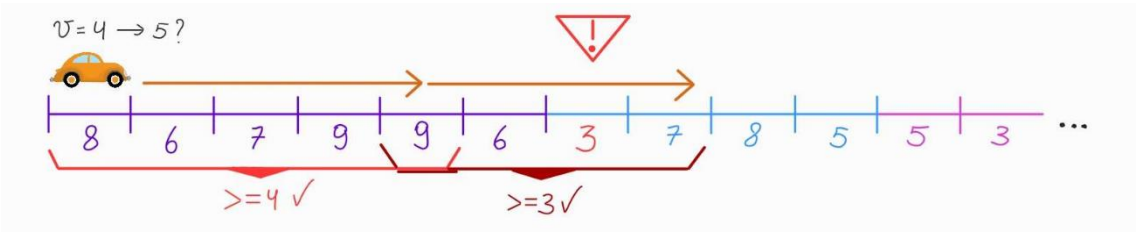


Figura 4 - Exemplo

Aqui neste caso, se alterarmos um dos valores do caminho (o 3 no exemplo) de modo a impedir o aumento de velocidade, iriamos concluir que o correto seria manter. Deixariamos de analisar as próximas 5+4+3+2+1 casas seguintes, para analisar as 4+3+2+1 casas, e o mesmo se verifica caso também não fosse possível manter a velocidade, passávamos a analisar as 3+2+1 casas seguintes. Pesquisando um pouco chegou-se a conclusão que a expressão geral do número de casas a frente a analisar é v*(v+1)/2, sendo “v” o valor da velocidade que queremos testar.

E assim, através de uma análise logo a partir da primeira casa, é possível tomar decisões sem nunca ter que voltar atrás para refazer o percurso, cada movimento é dado com a plena certeza que mais à frente não haverá nenhum constrangimento que impeça o carro de continuar.

Com recurso a este algoritmo teremos sempre uma rota assegurada, a rota de ir diminuindo a velocidade até 1 (a importância de ser até 1 será explicada mais à frente), conseguindo garantir esta rota, o resto é um bónus, sempre que o carro parar numa casa aplicamos este algoritmo para tentar aumentar, caso não dê, manter e em último caso diminuir, porque diminuir dará sempre, por já terá sido calculado na posição anterior onde passou. É óbvio que das três alterações de velocidade possíveis a mais benéfica será aumentar, mas como neste algoritmo apenas tomamos uma decisão após ter certeza que a mesma não terá conflitos, muitas vezes vamos ter que manter ou até mesmo abrandar.

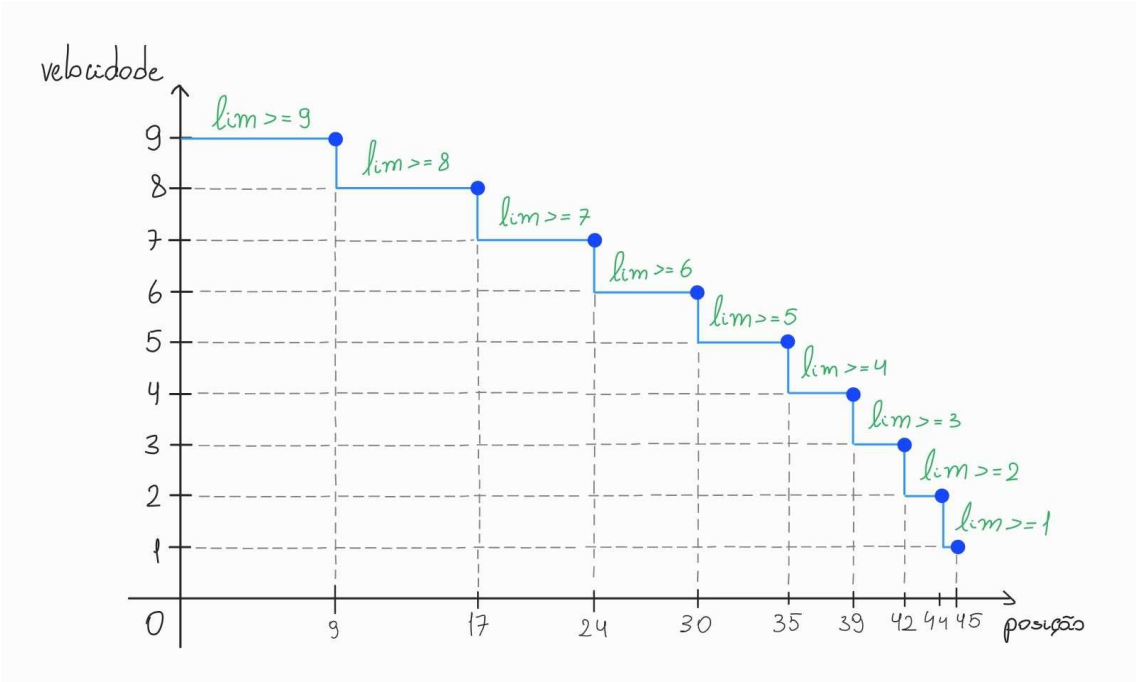


Figura 5 – representação da análise das v(v+1)/2 casas*

Aqui com este esboço pretendemos mostrar o caso hipotético do carro ter chegado a uma dada casa p (posição relativa 0) com velocidade 8, o mesmo só poderá aumentar para velocidade 9 caso as condições da figura se cumpram, desta ótica o esquema tem um aspeto em escadas onde o eixo yy será das velocidades e o eixo xx das posições.

Agora para realçar a ideia de que existe sempre uma rota que está garantida vamos pegar na casa 17 (posição absoluta p+17) onde o carro chega a 8 de velocidade.

Neste ponto é aplicado o algoritmo para testar se é possível aumentar a velocidade, imaginemos que não é possível, e manter também não, não há problema, a rota de diminuir velocidade já foi assegurada quando se tomou a decisão de aumentar a velocidade na casa 0 (ou p) para 9. Assim nunca corrermos riscos, este método é o mais defensivo que se pode ter, há sempre uma rota garantida e nunca se toma nenhuma decisão sem antes termos certeza que não teremos nenhum entrave no futuro.

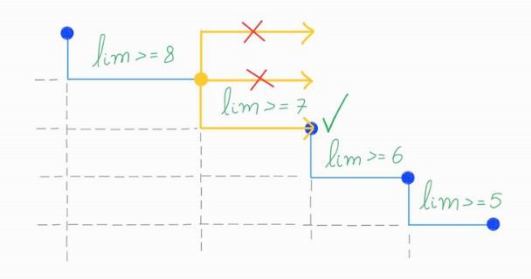


Figura 6 – representação do caminho seguro

Já vimos como o carro progride ao longo da estrada, agora falta-nos saber como é que ele chega ao final nas condições impostas. Simples, graças à análise que é feita sempre que o carro dá um movimento podemos muito facilmente detetar se o fim da estrada está ao nosso alcance. Basicamente, como este algoritmo nos garante sempre a existência de uma rota segura (consecutivamente a abrandar) até à velocidade 1 (e é daqui que vem a tal importância de se calcular sempre o numero de casas até a velocidade ser 1) basta que, quando a posição final estiver dentro do intervalo de casas que estamos a analisar (dentro das v*(v+1)/2 casas à frente), o carro comece logo a abrandar, independentemente dos limites de velocidade, pois o mesmo precisa de acabar com velocidade 1 e é precisamente isso que este algoritmo nos proporciona, um caminho seguro de qualquer que seja a velocidade até 1.

Ao contrário da solução dada pelo professor que forma uma árvore ternária cuja complexidade da mesma será 3^n, esta solução apenas formará um "ramo", visto que não se anda "para a frente e para trás", a complexidade desta será apenas n.

Sol 3 - Teacher Optimized Solution

Nesta solução, será tido em conta o código da solução 1 e os pontos negativos mencionados na mesma.

Tendo em conta o código do professor, foi analisado que a ordem de validação das velocidades (diminuir, manter e aumentar) era desfavorável ao nosso pensamento inicial ("passar pelos segmentos com a maior velocidade possível, respeitando sempre os limites de velocidade"). Logo alteramos a ordem de validação para aumentar, manter ou diminuir, pois, caso válida, a opção de avançar, é uma mais-valia para chegar ao objetivo, embora possa nem sempre ser a opção correta.

No entanto, essa alteração não mudava nada quanto ao desempenho do programa. Logo decidimos tentar entender a resolução da função como uma árvore ternária. Cada iteração contém três hipóteses possíveis, assim sendo, 3^N hipóteses, em que N é o número de segmentos. Recorrendo a esse raciocínio, decidimos traçar um exemplo de uma árvore ternária:

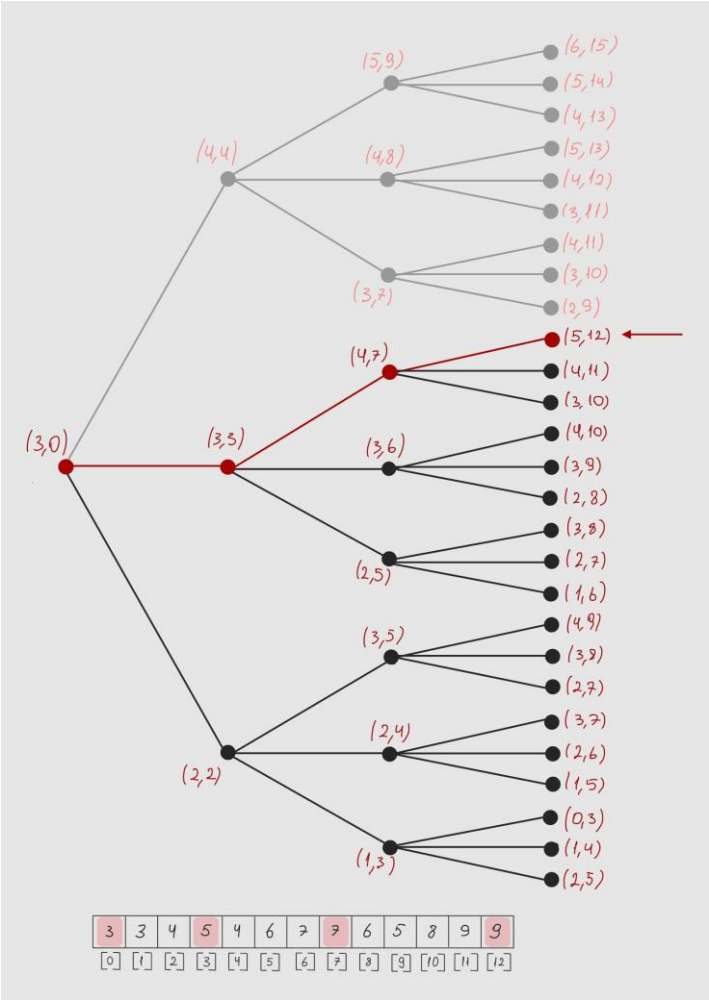


Figura 7 – exemplo de uma árvore ternária e o seu respetivo segmento;

Legenda dos nós : (velocidade, posição)

Neste exemplo prático, criou-se uma parte de um caminho com 12 segmentos, com limites de velocidade aleatórios, para ilustrar um trecho de um caminho possível. Em seguida é apresentada a sua árvore ternária.

Legenda: Os ramos a cinzento representam os caminhos impossíveis perante as regras impostas pelo programa. O ramo destacado a vermelho representa a primeira solução verificada. O resto dos ramos são os restantes possíveis caminhos.

Atendendo ao fluxo da árvore:

- o carro encontra-se na posição 0 (do trecho, não implicando que realmente seja o início do caminho original) com 3 de velocidade, (3,0); é analisada a opção de aumentar (para o nó (4,4)), no entanto não é validado, pois, as casas seguintes, da posição 0 à posição 3, não cumprem o requisito da nova velocidade ser menor ou igual ao limite de velocidade das casas que irá percorrer. Logo o ramo e o resto da sua ramificação serão ignorados e nunca mais analisados.
- testa-se agora a opção de manter a velocidade (para o nó (3,3)), é possível e por isso é avançado um nó;

- no nó (3, 3) a opção de subida de velocidade é validada com sucesso e avançado um nó (4,7);
- entre a posição 7 e 12 (do nó (4,7) para o nó (5,12)) a opção de subida de velocidade é validade e concluída a análise do caminho;
- sendo que chegou à primeira solução, o programa termina;

Analisando a árvore e o fluxo, a primeira solução distingue se das restantes, pelo motivo de chegar a uma posição maior e com a maior velocidade, o que é uma mais-valia para o objetivo principal do programa. As casas selecionadas a vermelho no segmento, representam o trajeto do carro.

Assim sendo, conseguimos concluir que, ao analisar a mudança de velocidade pela ordem indicada é uma vantagem no funcionamento da função, permite o encontro da melhor solução possível, pois a primeira solução identificada será sempre a melhor, o que vai diminuir o número de chamadas à recursiva e consequentemente melhorar a performance do código.

Por fim, de acordo com este pensamento teórico conseguimos provar na prática que a sua solução está correta e funciona para qualquer valor de n, comprimento do caminho, e assim criou-se uma nova proposta de solução que visa melhorar o desempenho do código do professor.

Sol 4 - Our solution in Dynamic Programing

Esta solução tem por base a solução 2. Logo à partida percebemos que a solução 2 era das soluções mais eficientes para resolver o problema (ordem dos 10^-6 para a posição 800), os cálculos eram mínimos e os resultados satisfatórios.

Analisando os próprios caminhos gerados, vimos que cada execução continha parte do caminho da execução anterior, esse caminho comum será a parte do caminho da execução anterior antes deste começar a desacelerar por estar a chegar ao fim, o que tem lógica visto que, desde que o carro não esteja a aproximar-se do final, as regras que regem cada movimento são sempre as mesmas, acelerar o máximo possível para chegar à posição final com o menor número de moves.

Sendo o caminho o mesmo, o único fator que vai diferir na movimentação do carro será o quão longe está do fim, ou seja, enquanto este algoritmo não detetar a casa final na nas v*(v+1)/2 casas a frente, o caminho será exatamente o mesmo.

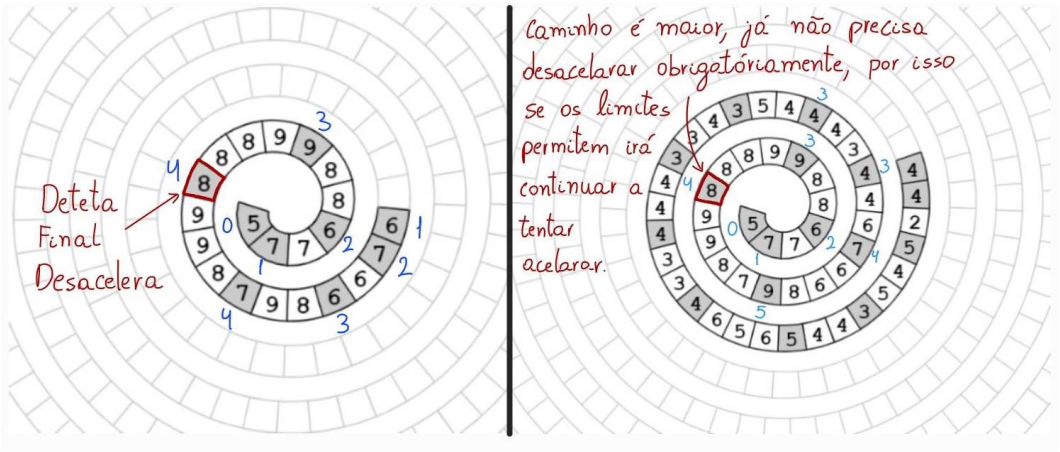


Figura 8 – Deteção do Final

Como se vê pela figura, até à casa vermelha a tendência é de acelerar sempre que possível, sendo que só nessa casa é que o algoritmo (no caso da esquerda) vai detetar que o final está próximo e obrigar o carro a desacelerar, enquanto que à direita como tem ainda um caminho maior pela frente não precisa de se preocupar (ainda) com o chegar ao fim com velocidade 1. Aplicando a expressão antes referida, estando a casa assinalada a 10 de distância do final, com velocidade v=4 teremos de ver se o final faz parte das v*(v+1)/2 casas à frente, como 4*5/2=10, provamos que faz, daí desacelerar.

Como provamos que há segmentos do caminho que são comuns, agora é arranjar maneira de os guardar para não termos que os calcular todos novamente de forma desnecessária.

Para tal efeito criamos uma struct nova “solution_4_copy” para guardar a parte do caminho que será comum ao anterior, assim como variáveis para a velocidade e a sua posição. Na próxima iteração, em vez de termos que calcular o caminho todo desde o zero, vamos simplesmente recomeçar do ponto onde guardamos anteriormente e assim poupamos tempo e recursos.

Analise em MatLab

	CPU	RAM
PC1 – Alexandre Cotorobai	AMD Ryzen 7 4800H (8C/16T, 2,9 ~ 4,2 GHz, 4 MB L2 / L3 8 MB)	2x 8GB DDR4-3200
PC2 – Vitalie Bologa	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz	2x 8GB DDR4-2933

Tabela 1 – Especificações dos computadores utilizados

Comparação entre Soluções nos PC1 e PC2

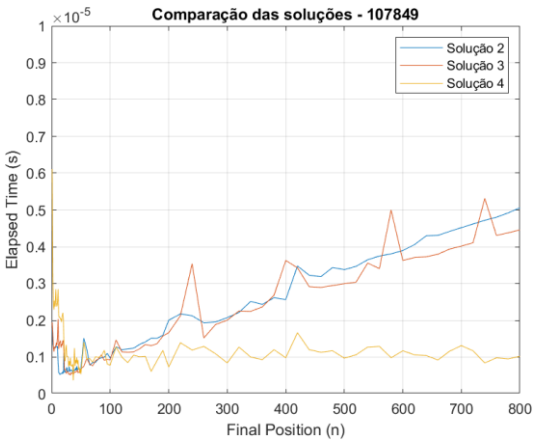


Figura 9 – gráfico da comparação de soluções - 107849

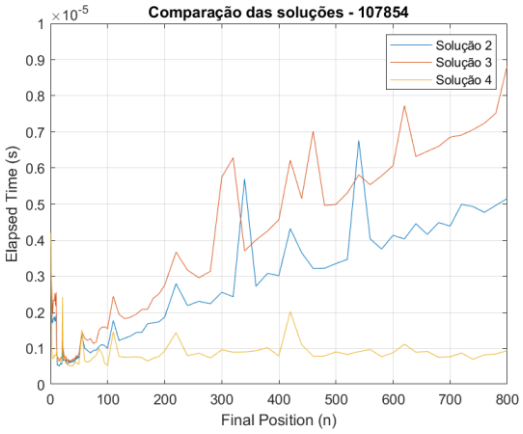


Figura 10 – gráfico da comparação de soluções - 107854

Com recurso a estes dois gráficos quisemos ver as diferenças entre o desempenho de cada uma das soluções que desenvolvemos. Comparando as nossas soluções nos gráficos do tempo em função da posição final, podemos ver uma grande diferença no desempenho da solução 4. O gráfico desta aproxima-se de uma função horizontal enquanto nas soluções 2 e 3 aproximam-se de uma função linear crescente, isto é devido a ter-se recorrido de programação dinâmica na solução 4, onde a mesma usa resultados anterior para calcular os próximo, o que não vai interferir muito com desempenho a cada iteração. Na solução 2 e 3, por terem que constantemente calcular todo o caminho do início a cada iteração, o tempo vai aumentando conforme o percurso se estende.

Podemos ver que os gráficos refletem algumas semelhanças com a complexidade que possuem.

Relação move_number - final_position

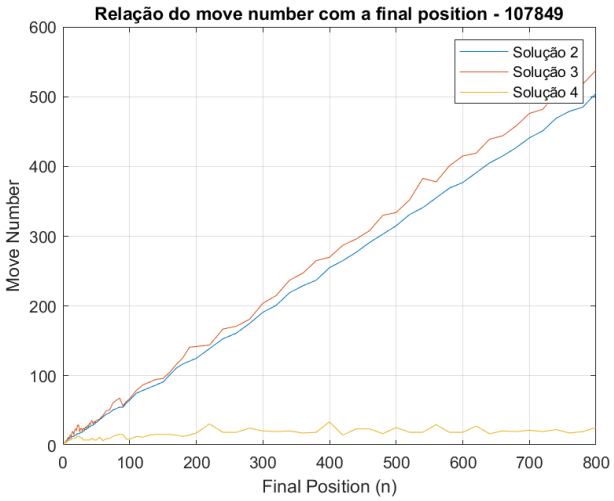


Figura 12 – gráfico do move_number com a final position - 107849

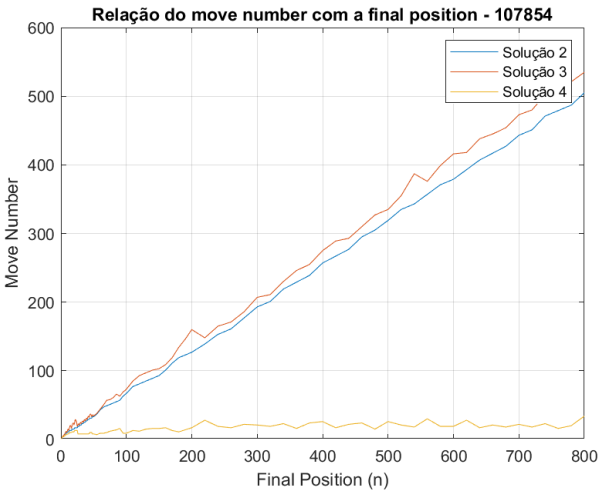


Figura 11 - gráfico do move_number com a final position - 107854

Também é possível comparar o número de moves com a posição final e assim podemos notar uma enorme relação entre esse número com o elapsed_time dos gráficos anteriores, realçando a ideia de que quantos mais moves forem necessários até ao fim, maior o tempo dispendido para encontrar o final. Já a solução 4 mantém-se horizontal pois só são contados o move_number daquela iteração, o resto já tendo sido calculado antes, daí manter-se razoavelmente estável.

Análise ao desempenho da solução 1

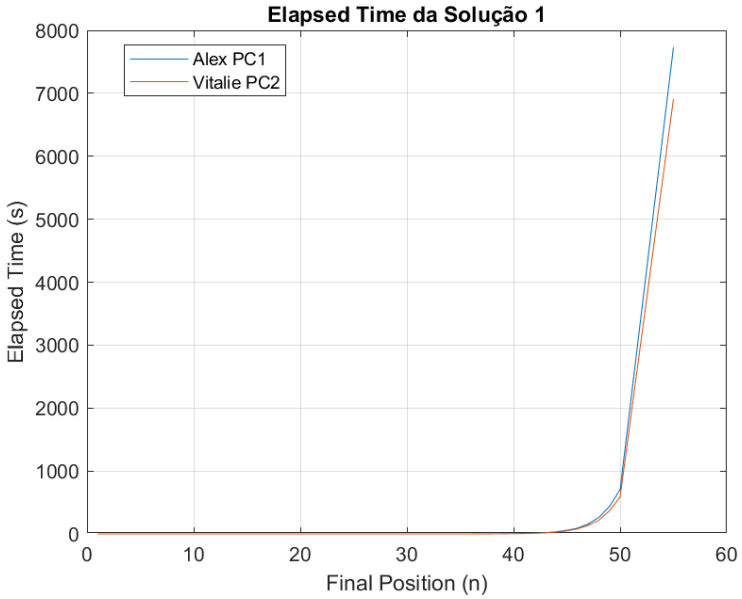


Figura 13 – gráfico Elapsed Time da Solução 1

Quanto à solução dada pelo professor, que apenas chegou a 55, podemos ver o aumento exponencial do que demora a encontrar solução, pois como são explorados todos os possíveis caminhos até encontrar o melhor, o número de chamadas que a função tem que fazer é enorme.

Novamente podemos observar semelhanças entre a complexidade da função, 3^n (exponencial), e o tipo de gráfico que esta forma, que também é exponencial.

Aproximação do elapsed_time da solução 1 para n=800

Para calcular a aproximação do tempo de execução da solução 1 para a posição 800, recorremos a uma regressão linear do logaritmo da função, onde através da reta de regressão conseguimos estimar o tempo que levaria a encontrar-se a melhor solução para n=800.

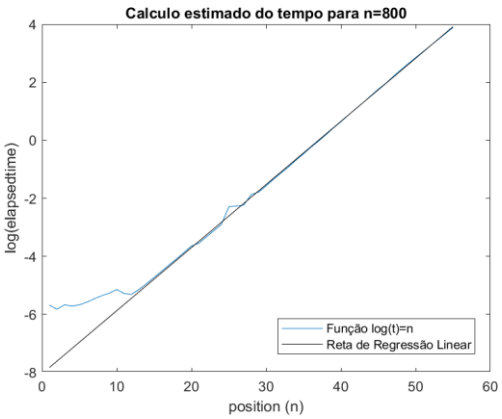


Figura 14 – regressão linear

O tempo estimado é 9.1038e+165 segundos.

O número mecanográfico utilizado foi o 107849.

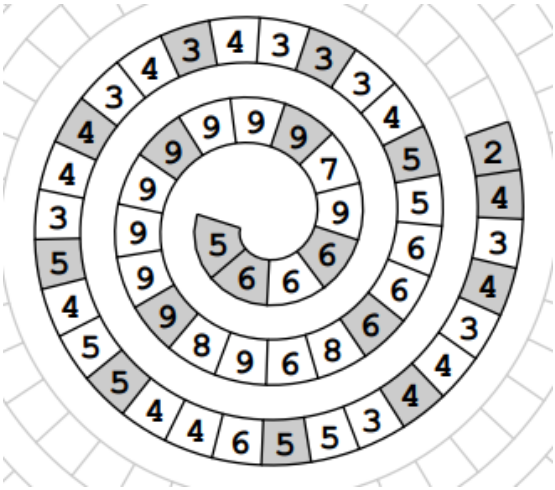
```
load AlexResults\Teachers_solution_107849_results.txt

n = Teachers_solution_107849_results(:,1);

t = Teachers_solution_107849_results(:,4);

figure(1)

plot(n,log10(t), "DisplayName", "Função log(t)=n")
xlabel("position (n)")
ylabel("log(elapsedtime)")

t_log = log10(t);

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

hold on
title("Calculo estimado do tempo para n=800")
plot(n, Ntotal*Coefs, 'k', "DisplayName","Reta de Regressão Linear")
legend()

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

Figura 15 – código em MatLab da regressão linear

Resultados e algumas conclusões

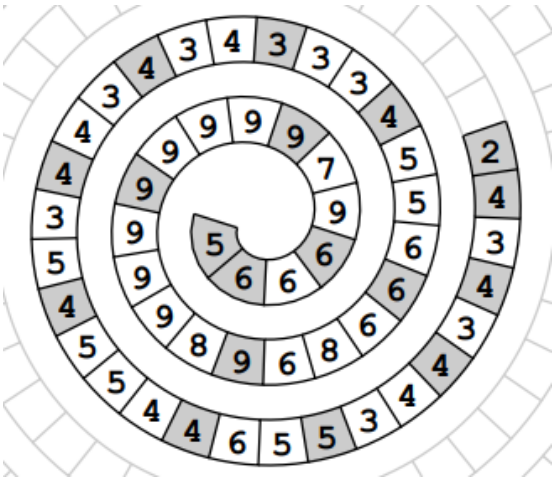
Solução 1 diferente

Após a execução de todas as soluções, ao compararmos as 4, até a casa 50 (pois foi o ponto mais longe que a solução 1 chegou, tendo de comparar em até a mesma distância) verificamos que a solução 1 é diferente das demais. Embora apresente o número de movimentos igual, apresenta uma solução diferente.



```
17 moves
6.966e+02 seconds
effort: 120499035176
```

Figura 16 – Solução 1 - 107849



```
17 moves
6.010e-07 seconds
effort: 33
```

Figura 17 – Solução 2 - 107849

Tendo em conta essa questão, pensamos o que podia levar a essa alteração, e então pensamos na ordem de validação de velocidades, pois é a única que tem a ordem: diminuir, manter ou aumentar, inversa às demais, e isso pode influenciar, pois começa por analisar o pior caso possível até ao melhor.

Na primeira solução, o programa vai guardando a best_solution encontrada até ao momento, e essa solução apenas será alterada caso encontre outra com um move number menor, ou seja, se encontrar outra solução com o mesmo move number, não irá atualizar. O que aconteceu é que como (imaginando uma arvore ternária parecida à da figura 7) a solução 1 começa a procurar de “baixo para cima” e todas as restantes procuram na ordem inversa. O mais provável é terem encontrado 2 soluções ótimas ao problema em sítios diferentes da árvore mas igualmente válidos.

Conclusão

Por fim, atendendo a todo o estudo e pesquisas feitas, conseguimos atingir os objetivos propostos no enunciado, cumprindo todas as regras de circulação impostas.

Apresentamos 3 soluções possíveis, sendo uma a melhoria da solução inicialmente dada, e as restantes de autoria própria, com diferentes modos de resolução, sendo uma aplicando *dynamic programing*. Demonstramos todos os procedimentos, pensamentos estruturados e cálculos efetuados, ao longo da criação das respetivas soluções, sendo posteriormente comprovado os seus resultados e analisados.

Neste trabalho, aprendemos a trabalhar em duplas, na organização e desenvolvimento de pensamentos, complementados com pesquisa e estudo, para a criação de soluções eficientes e otimizadas.

Apêndice

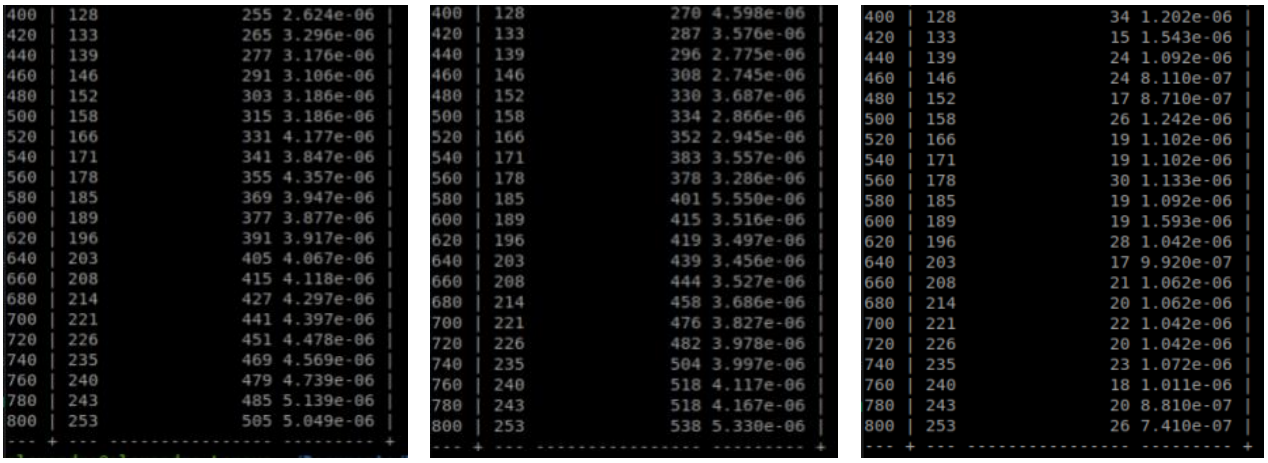
Outputs

		plain recursion	
n	sol	count	cpu time
1	1	2	3.106e-06
2	2	3	3.430e-06
3	3	5	3.178e-06
4	3	8	3.229e-06
5	4	13	3.582e-06
6	4	22	4.430e-06
7	5	36	5.083e-06
8	5	60	6.560e-06
9	5	100	8.597e-06
10	6	167	1.079e-05
11	6	279	4.807e-06
12	6	465	6.472e-06
13	7	777	1.480e-05
14	7	1297	1.446e-05
15	7	2165	2.072e-05
16	7	3614	3.135e-05
17	8	6031	4.872e-05
18	8	10065	7.757e-05
19	8	16795	1.318e-04
20	8	28024	2.082e-04
21	9	46737	3.498e-04
22	9	77953	5.547e-04
23	9	129246	9.270e-04
24	9	214812	1.519e-03
25	9	357550	2.521e-03
26	10	594882	4.184e-03
27	10	990792	6.899e-03
28	10	1650459	1.115e-02
29	11	2749337	1.761e-02
30	11	4580894	2.839e-02
31	11	7632348	4.724e-02
32	12	12717238	7.917e-02
33	12	21190351	1.310e-01
34	12	35308584	2.171e-01
35	13	58834488	3.629e-01
36	13	98035291	5.787e-01
37	13	163355664	8.057e-01
38	14	272199621	1.295e+00
39	14	453565737	2.154e+00
40	14	755777147	3.585e+00
41	15	1265839422	6.017e+00
42	15	2115757244	1.016e+01
43	15	3531976979	1.710e+01
44	16	5868299958	3.134e+01
45	16	9761320467	5.535e+01
46	16	16254754458	9.253e+01
47	16	27057736996	1.440e+02
48	17	43277181715	2.365e+02
49	17	70299608972	3.983e+02
50	17	113541480948	6.476e+02
55	19	1324254012628	7.260e+03

Figura 19 - Solução 1 - 107854

		plain recursion	
n	sol	count	cpu time
1	1	2	1.743e-06
2	2	3	1.402e-06
3	3	5	1.393e-06
4	3	8	1.383e-06
5	4	13	1.513e-06
6	4	22	1.874e-06
7	5	36	2.365e-06
8	5	60	2.686e-06
9	5	100	3.747e-06
10	6	167	5.019e-06
11	6	279	7.615e-06
12	6	465	9.147e-06
13	7	777	1.384e-05
14	7	1297	2.217e-05
15	7	2165	3.582e-05
16	7	3614	5.778e-05
17	8	6031	9.566e-05
18	8	10065	1.582e-04
19	8	16795	2.737e-04
20	8	28024	4.374e-04
21	9	46758	4.939e-04
22	9	78011	8.117e-04
23	9	130089	1.379e-03
24	9	215655	2.305e-03
25	9	358393	3.746e-03
26	10	596510	6.189e-03
27	10	992420	1.052e-02
28	10	1652872	1.774e-02
29	10	2753320	2.763e-02
30	11	4586447	4.589e-02
31	11	7641826	7.620e-02
32	11	12732211	1.275e-01
33	12	21214744	2.188e-01
34	12	35349462	3.836e-01
35	12	58901271	6.128e-01
36	13	98146819	1.011e+00
37	13	163540982	1.695e+00
38	13	273911933	2.912e+00
39	14	457822811	3.797e+00
40	14	764273024	4.874e+00
41	14	1276316808	8.121e+00
42	15	2130938311	1.319e+01
43	15	3536027692	2.190e+01
44	15	5880139571	3.690e+01
45	15	9790346764	5.960e+01
46	16	16292581614	9.877e+01
47	16	27138497290	1.669e+02
48	16	45210347786	2.722e+02
49	17	75315052234	4.592e+02
50	17	120499035176	7.412e+02
55	19	1340251164450	7.751e+03
60			

Figura 18 - Solução 1 - 107849

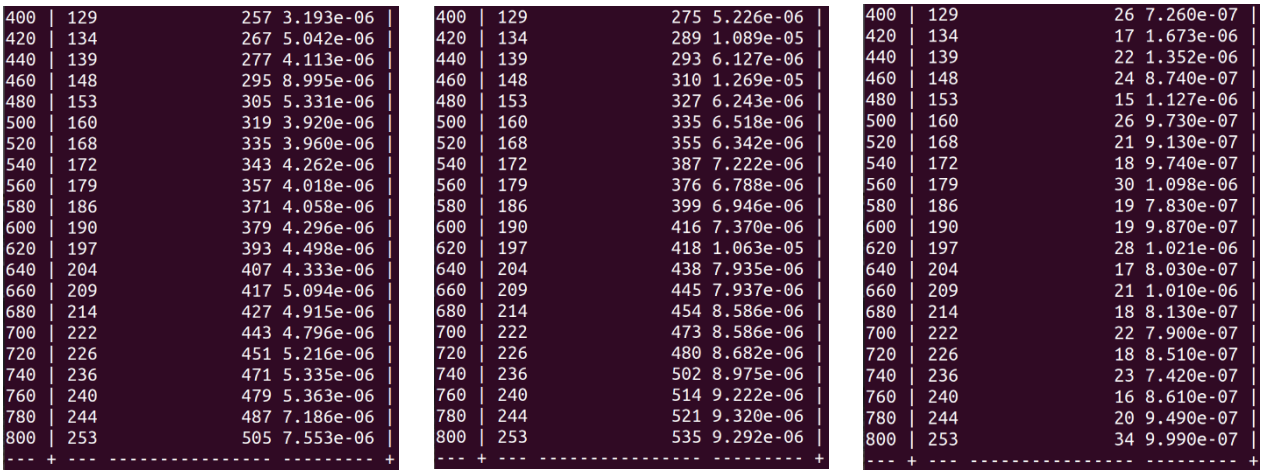


400	128	255	2.624e-06
420	133	265	3.296e-06
440	139	277	3.176e-06
460	146	291	3.106e-06
480	152	303	3.186e-06
500	158	315	3.186e-06
520	166	331	4.177e-06
540	171	341	3.847e-06
560	178	355	4.357e-06
580	185	369	3.947e-06
600	189	377	3.877e-06
620	196	391	3.917e-06
640	203	405	4.067e-06
660	208	415	4.118e-06
680	214	427	4.297e-06
700	221	441	4.397e-06
720	226	451	4.478e-06
740	235	469	4.569e-06
760	240	479	4.739e-06
780	243	485	5.139e-06
800	253	505	5.049e-06

400	128	270	4.598e-06
420	133	287	3.576e-06
440	139	296	2.775e-06
460	146	308	2.745e-06
480	152	330	3.687e-06
500	158	334	2.866e-06
520	166	352	2.945e-06
540	171	383	3.557e-06
560	178	378	3.286e-06
580	185	401	5.550e-06
600	189	415	3.516e-06
620	196	419	3.497e-06
640	203	439	3.456e-06
660	208	444	3.527e-06
680	214	458	3.686e-06
700	221	476	3.827e-06
720	226	482	3.978e-06
740	235	504	3.997e-06
760	240	518	4.117e-06
780	243	518	4.167e-06
800	253	538	5.330e-06

400	128	34	1.202e-06
420	133	15	1.543e-06
440	139	24	1.092e-06
460	146	24	8.110e-07
480	152	17	8.710e-07
500	158	26	1.242e-06
520	166	19	1.102e-06
540	171	19	1.102e-06
560	178	30	1.133e-06
580	185	19	1.092e-06
600	189	19	1.593e-06
620	196	28	1.042e-06
640	203	17	9.920e-07
660	208	21	1.062e-06
680	214	20	1.062e-06
700	221	22	1.042e-06
720	226	20	1.042e-06
740	235	23	1.072e-06
760	240	18	1.011e-06
780	243	20	8.810e-07
800	253	26	7.410e-07

Figura 20 - Soluções 2, 3 e 4, respetivamente - 107849



400	129	257	3.193e-06
420	134	267	5.042e-06
440	139	277	4.113e-06
460	148	295	8.995e-06
480	153	305	5.331e-06
500	160	319	3.920e-06
520	168	335	3.960e-06
540	172	343	4.262e-06
560	179	357	4.018e-06
580	186	371	4.058e-06
600	190	379	4.296e-06
620	197	393	4.498e-06
640	204	407	4.333e-06
660	209	417	5.094e-06
680	214	427	4.915e-06
700	222	443	4.796e-06
720	226	451	5.216e-06
740	236	471	5.335e-06
760	240	479	5.363e-06
780	244	487	7.186e-06
800	253	505	7.553e-06

400	129	275	5.226e-06
420	134	289	1.089e-05
440	139	293	6.127e-06
460	148	310	1.269e-05
480	153	327	6.243e-06
500	160	335	6.518e-06
520	168	355	6.342e-06
540	172	387	7.222e-06
560	179	376	6.788e-06
580	186	399	6.946e-06
600	190	416	7.370e-06
620	197	418	1.063e-05
640	204	438	7.935e-06
660	209	445	7.937e-06
680	214	454	8.586e-06
700	222	473	8.586e-06
720	226	480	8.682e-06
740	236	502	8.975e-06
760	240	514	9.222e-06
780	244	521	9.320e-06
800	253	535	9.292e-06

400	129	26	7.260e-07
420	134	17	1.673e-06
440	139	22	1.352e-06
460	148	24	8.740e-07
480	153	15	1.127e-06
500	160	26	9.730e-07
520	168	21	9.130e-07
540	172	18	9.740e-07
560	179	30	1.098e-06
580	186	19	7.830e-07
600	190	19	9.870e-07
620	197	28	1.021e-06
640	204	17	8.030e-07
660	209	21	1.010e-06
680	214	18	8.130e-07
700	222	22	7.900e-07
720	226	18	8.510e-07
740	236	23	7.420e-07
760	240	16	8.610e-07
780	244	20	9.490e-07
800	253	34	9.990e-07

Figura 21 - Soluções 2, 3 e 4, respetivamente - 107854

Código da Solução 2

```
static solution_t solution_2, solution_2_best;
static double solution_2_elapsed_time; // time it took to solve the
problem
static unsigned long solution_2_count; // effort dispended solving the
problem

static void solution_2_new(int move_number, int position, int speed,
int final_position)
{
  int new_speed = 0;
  int value;
  int validIncrementation;

  solution_2.positions[move_number] = position;

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

  for (value = 1; value >= -1; value--)
  {
    int newskip = 0;
    int oldskip = 0;
    int s;
    validIncrementation = 1;

    solution_2_count++;

    for (s = speed + value; s > 0; s--)
    {
      oldskip = newskip;
      newskip = s + newskip;
      for (int p = position + oldskip; p <= position + newskip; p++)
      {

        if (p > final_position || s > max_road_speed[p])
        {
          validIncrementation = 0;
          break;
        }
      }
      if (validIncrementation == 0)
        break;
    }
    if (validIncrementation == 1)
    {
      new_speed = speed + value;
      break;
    }
  }
```

```
  solution_2_new(move_number + 1, position + new_speed, new_speed,
final_position);
}

static void solve_2(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_1: 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_new(0, 0, 0, final_position);
  solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}
```

Código da Solução 3

```
static solution_t solution_3, solution_3_best;
static double solution_3_elapsed_time; // time it took to solve the
problem
static unsigned long solution_3_count; // effort dispended solving the
problem

static void solution_3_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 (solution_3_best.n_moves != final_position + 100)
  {
    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_3_recursion(move_number + 1, position + new_speed,
new_speed, final_position);

      }
    }
}

static void solve_3(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_1: bad final_position\n");
    exit(1);
  }
  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;
}
```

Código da Solução 4

```
static solution_t solution_4, solution_4_best, solution_4_copy;
static double solution_4_elapsed_time; // time it took to solve the
problem
static unsigned long solution_4_count; // effort dispended solving the
problem
int save = 0;
int saveSpeed = 0;
int savePosition = 0;

static void solution_4_new(int move_number, int position, int speed,
int final_position)
{
  int new_speed = 0;
  int value, validIncrementation;

  solution_4.positions[move_number] = position;
```

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

  for (value = 1; value >= -1; value--)
  {
    int newskip = 0;
    int oldskip = 0;
    int s;
    validIncrementation = 1;

    solution_4_count++;

    for (s = speed + value; s > 0; s--)
    {
      oldskip = newskip;
      newskip = s + newskip;
      for (int p = position + oldskip; p <= position + newskip; p++)
      {
        if (p > final_position && save == 0)
        {
          solution_4_copy = solution_4;
          solution_4_copy.n_moves = move_number;
          saveSpeed = speed;
          savePosition = position;

          save = 1;
        }
        if (p > final_position || s > max_road_speed[p])
        {
          validIncrementation = 0;
          break;
        }
      }
      if (validIncrementation == 0)
        break;
    }
    if (validIncrementation == 1)
    {
      new_speed = speed + value;
      break;
    }
  }

  solution_4_new(move_number + 1, position + new_speed, new_speed,
final_position);
}

static void solve_4(int final_position)
{
  int speed = 0;
  int position = 0;
  if (final_position < 1 || final_position > _max_road_size_)
  {
```

```
    fprintf(stderr, "solve_1: bad final_position\n");
    exit(1);
  }
  solution_4_elapsed_time = cpu_time();
  solution_4_count = 0ul;
  solution_4_best.n_moves = final_position + 100;
  if (save == 1)
  {
    solution_4 = solution_4_copy;
    solution_4_count = 0;
  }
  save = 0;
  solution_4_new(solution_4.n_moves, savePosition, saveSpeed,
final_position);
  solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}
```

Código usado para escrever nos ficheiros

```
// Writing results on file

    sprintf(file_name, "%s_%d_results.txt", solution_name, n_mec);
    fptr = fopen(file_name, "a");
    if (fptr == NULL)
    {
      printf("Error!");
      exit(1);
    }
    fprintf(fptr, "%d,%d,%d,%9.3e\n", final_position,
solution_2_best.n_moves, solution_2_count, solution_2_elapsed_time);
```

MATLAB – Figura 13

```
clear;
clc;
load AlexResults\Teachers_solution_107849_results.txt

n_t = Teachers_solution_107849_results(:,1);

t_t = Teachers_solution_107849_results(:,3);


load VitalieResults\Teachers_solution_107854_results.txt

n_t2 = Teachers_solution_107854_results(:,1);

t_t2 = Teachers_solution_107854_results(:,3);

plot(n_t,abs(t_t),"DisplayName", "Alex PC1")
hold on
plot(n_t2,abs(t_t2), "DisplayName", "Vitalie PC2")
grid()
title("Elapsed Time da Solução 1")
legend()
xlabel("Final Position (n)")
ylabel("Elapsed Time (s)")
```

MATLAB – Figuras 9 e 10

```
clear;
clc;
load AlexResults\Our_solution_107849_results.txt

n2 = Our_solution_107849_results(:,1);

t2 = Our_solution_107849_results(:,4);

load AlexResults\Teachers_solved_107849_results.txt

n3 = Teachers_solved_107849_results(:,1);

t3 = Teachers_solved_107849_results(:,4);

load AlexResults\Our_Dynamic_107849_results.txt

n4 = Our_Dynamic_107849_results(:,1);

t4 = Our_Dynamic_107849_results(:,4);

figure(1)

plot(n2,t2,"DisplayName", "Solução 2")
hold on
plot(n3,t3, "DisplayName", "Solução 3")
hold on
```

```
plot(n4,t4, "DisplayName", "Solução 4")
grid()
title("Comparação das soluções - 107849")
legend()
xlabel("Final Position (n)")
ylabel("Elapsed Time (s)")
xlim([0,800])
ylim([0, 10e-6])

clear;
load VitalieResults\Our_solution_107854_results.txt

n2 = Our_solution_107854_results(:,1);

t2 = Our_solution_107854_results(:,4);

load VitalieResults\Teachers_solved_107854_results.txt
n3 = Teachers_solved_107854_results(:,1);

t3 = Teachers_solved_107854_results(:,4);

load VitalieResults\Our_Dynamic_107854_results.txt
n4 = Our_Dynamic_107854_results(:,1);

t4 = Our_Dynamic_107854_results(:,4);

figure(2)
plot(n2,t2,"DisplayName", "Solução 2")
hold on
plot(n3,t3, "DisplayName", "Solução 3")
hold on
plot(n4,t4, "DisplayName", "Solução 4")
grid()
title("Comparação das soluções - 107854")
legend()
xlabel("Final Position (n)")
ylabel("Elapsed Time (s)")
xlim([0,800])
ylim([0, 10e-6])
```

MATLAB – Figuras 11 e 12

```
clear;
clc;
load AlexResults\Our_solution_107849_results.txt

n2 = Our_solution_107849_results(:,1);

t2 = Our_solution_107849_results(:,3);

load AlexResults\Teachers_solved_107849_results.txt

n3 = Teachers_solved_107849_results(:,1);

t3 = Teachers_solved_107849_results(:,3);
```

```
load AlexResults\Our_Dynamic_107849_results.txt

n4 = Our_Dynamic_107849_results(:,1);

t4 = Our_Dynamic_107849_results(:,3);

figure(1)

plot(n2,t2,"DisplayName", "Solução 2")
hold on
plot(n3,t3, "DisplayName", "Solução 3")
hold on
plot(n4,t4, "DisplayName", "Solução 4")
grid()
title("Relação do move number com a final position - 107849")
legend()
xlabel("Final Position (n)")
ylabel("Move Number")

clear;
load VitalieResults\Our_solution_107854_results.txt

n2 = Our_solution_107854_results(:,1);

t2 = Our_solution_107854_results(:,3);

load VitalieResults\Teachers_solved_107854_results.txt
n3 = Teachers_solved_107854_results(:,1);

t3 = Teachers_solved_107854_results(:,3);

load VitalieResults\Our_Dynamic_107854_results.txt
n4 = Our_Dynamic_107854_results(:,1);

t4 = Our_Dynamic_107854_results(:,3);

figure(2)
plot(n2,t2,"DisplayName", "Solução 2")
hold on
plot(n3,t3, "DisplayName", "Solução 3")
hold on
plot(n4,t4, "DisplayName", "Solução 4")
grid()
title("Relação do move number com a final position - 107854")
legend()
xlabel("Final Position (n)")
ylabel("Move Number")
```

PDF's gerados

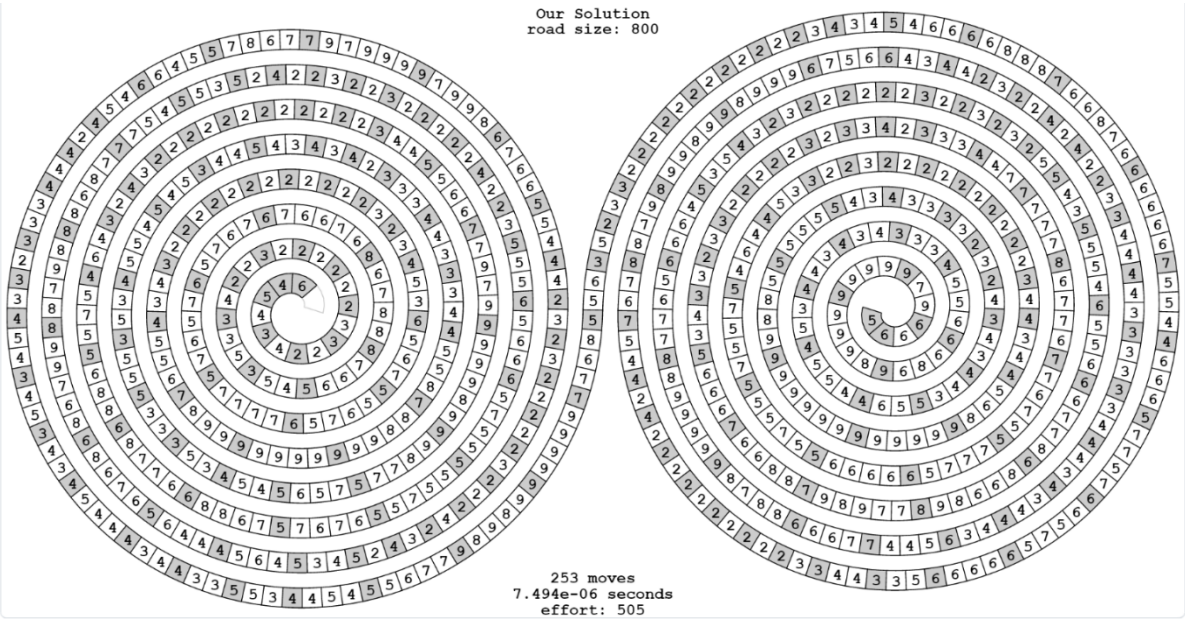


Figura 22 - Solução 2 - 107849

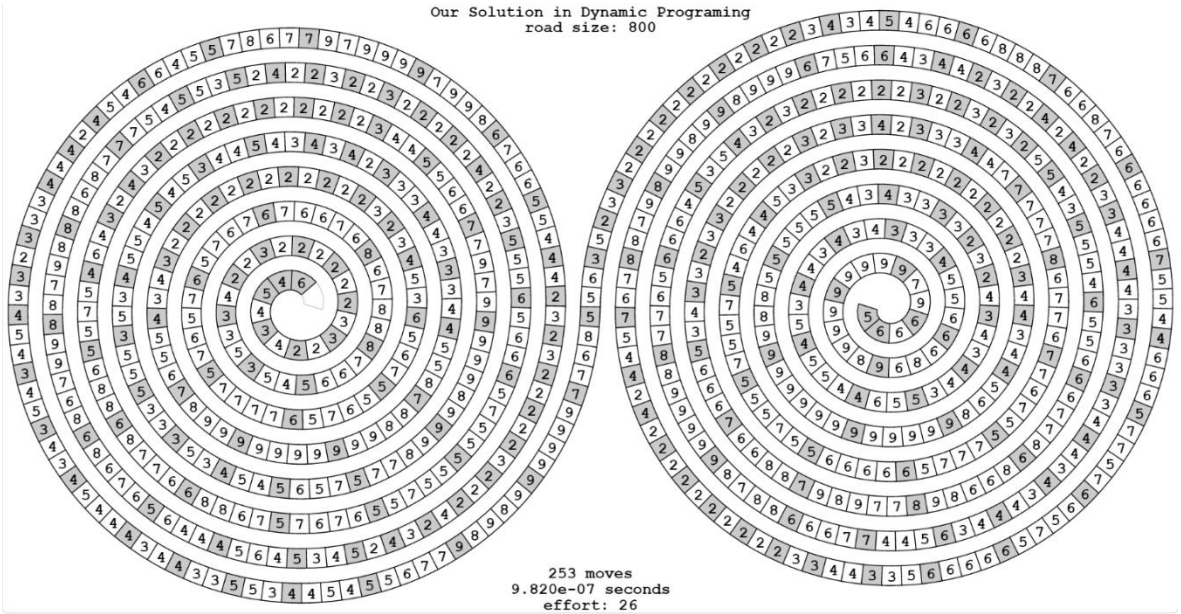


Figura 23 - Solução 4 - 107849

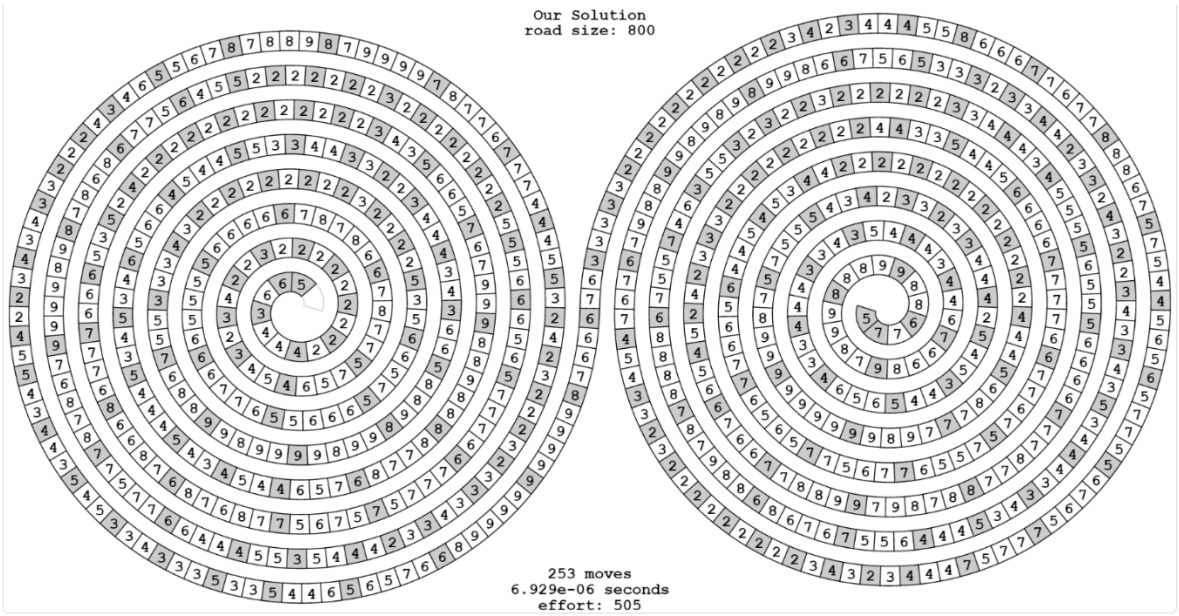


Figura 23 - Solução 2 - 107854

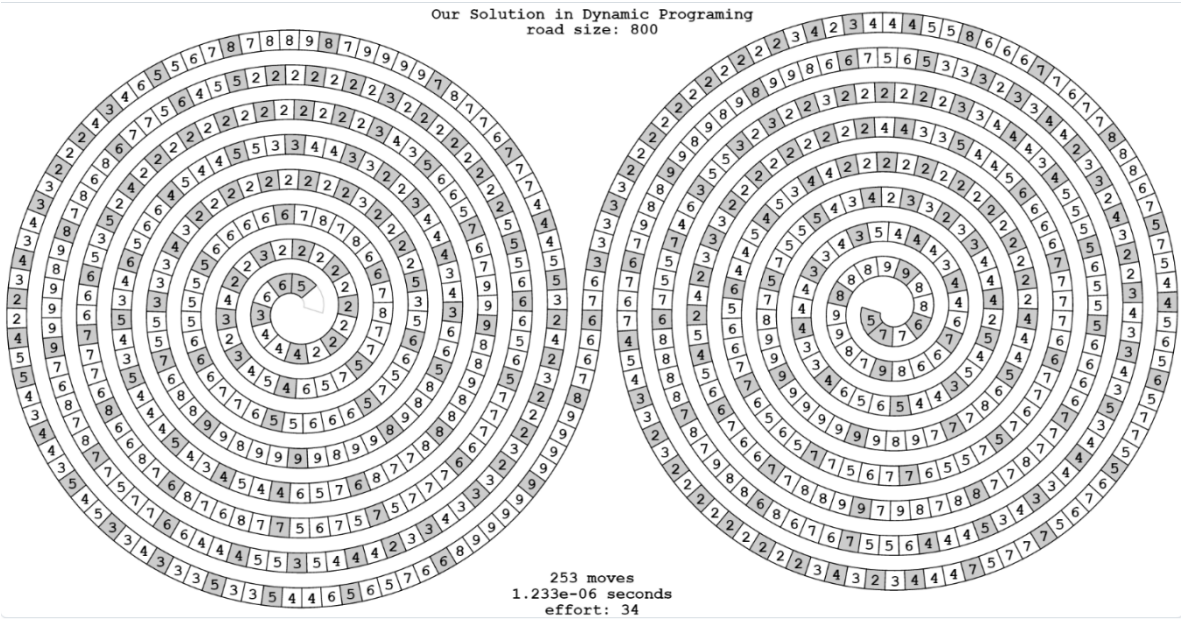


Figura 25 - Solução 4 - 107854