

**Trabalho Prático 1**

**Speed Run**

**Relatório**

**Turma P8**

**Realizado por:**

Diogo Falcão Nº108712 - Licenciatura em Engenharia Informática – **50%**

José Gameiro Nº108840 - Licenciatura em Engenharia Informática – **50%**

***Índice:***

[***1.*** ***Introdução*** 3](#_Toc121327904)

[**1.1.** **Conceitos Base** 3](#_Toc121327905)

[***2.*** ***Desenvolvimento*** 4](#_Toc121327906)

[**2.2.** **Solution\_1\_recursion (melhorada)** 7](#_Toc121327907)

[**2.3.** **Solution\_2\_Recursion** 9](#_Toc121327908)

[**2.4.** **Solution\_3\_Iterative** 10](#_Toc121327909)

[**2.5.** **Solution\_4\_Iterative (Programação Dinâmica)** 12](#_Toc121327910)

[***3.Código*** 15](#_Toc121327911)

[***4.*** ***Conclusão*** 22](#_Toc121327912)

[***5.*** ***Webgrafia*** 22](#_Toc121327913)

# ***Introdução***

Para o projeto “Speed Run” da cadeira Algoritmos e Estruturas de Dados, foi-nos proposto otimizar um programa com base no código fornecido pelo professor. O objetivo deste programa é calcular, para uma estrada constituída por blocos com velocidade máximas, o número mínimo de movimentos de um carro desde o início ao fim de toda a estrada.

Existem alguns aspetos a ter em conta: toda os blocos da estrada são aproximadamente da mesma largura e como já referido, têm todos uma velocidade máxima estabelecida aleatória entre um e nove. A velocidade é medida pela quantidade de blocos que o carro irá fazer num único movimento. Em cada movimento, o carro pode diminuir, manter ou aumentar a sua velocidade uma unidade. Por fim, no último bloco, tem de se garantir que a velocidade do carro vai ser 1, para depois descer para 0 na última casa e assim, parar.

Este relatório irá demonstrará que existem soluções recursivas e iterativas mais eficientes através de uma breve explicação de cada solução, gráficos dos tempos de execução e ficheiros PDF gerados por estas.

## **Conceitos Base**

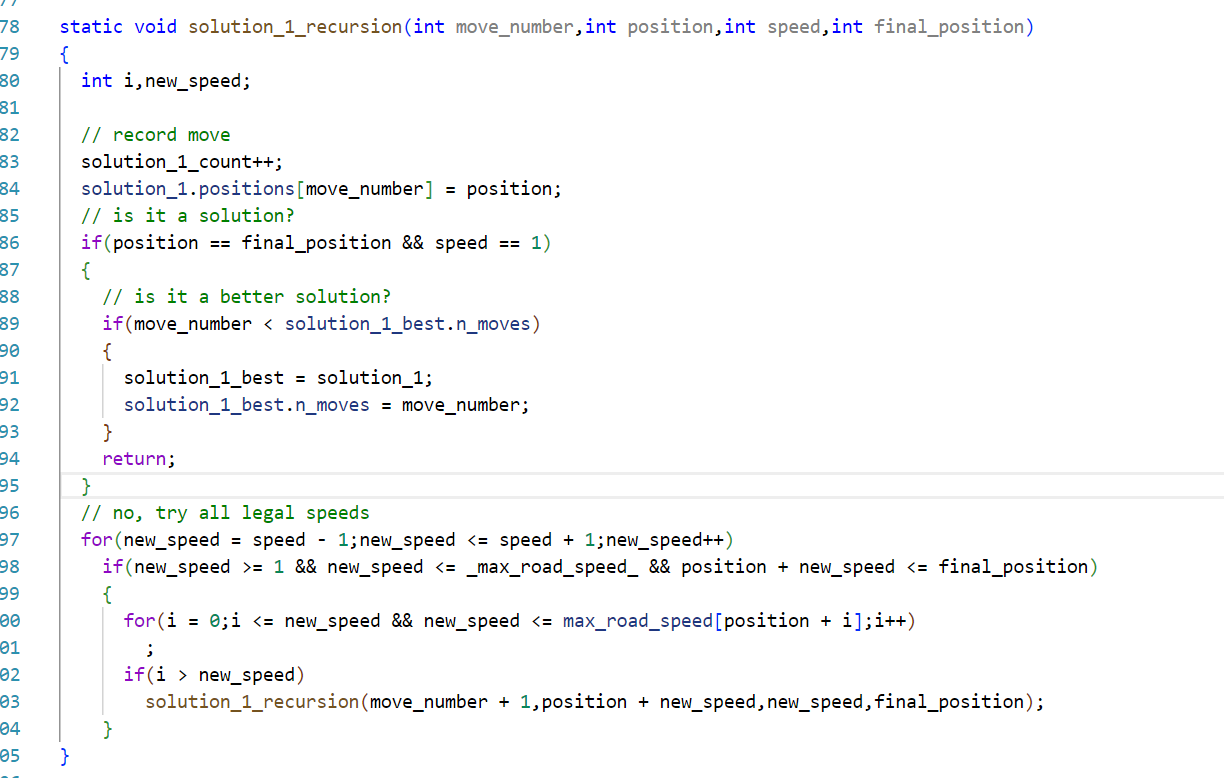
**Branch and Bound** é uma técnica algorítmica que é utilizado muitas vezes para resolver problemas de otimização combinatória. Os problemas em que se aplica esta técnica são do tipo exponencial em termos de complexidade algorítmica que podem necessitar da análise de todas as permutações possíveis. Esta técnica é utilizada frequentemente para problemas deste género pois consegue encontrar uma solução de forma bastante rápida.

A abordagem **Back Tracking** é utilizada para resolver problemas de forma recursiva, onde se tenta construir elaborar uma solução de forma incremental, removendo as soluções que falham nas restrições criadas a qualquer momento da execução da solução.

**Tabulation** é um método utilizado para a resolução de problemas com programação dinâmica. Este método baseia-se em criar uma tabela e ao longo do tempo ir preenchendo-a com dados do problema em questão, para depois se poder encontrar uma solução, com base no resultado da tabela. É considerado um método ascendente, ou seja, começamos por resolver o problema pelos casos mais simples inserindo na tabela os dados e completando com os dados dos casos seguintes até ao topo da tabela. Esta implementação é considerada iterativa.

**Memoization** é uma forma de armazenar os dados de um problema em cache, que será utilizada em programação dinâmica. O objetivo da cache é melhorar o desempenho de programas e manter os dados acessíveis para que possam ser utilizados em futuros algoritmos. Isto faz com que o esforço de calcular novamente a solução para o mesmo problema seja removido.

# ***Desenvolvimento***

* 1. **Solution\_1\_Recursion (Disponibilizada pelo professor)**

**Fig.1 –** Solução recursiva fornecida pelo professor

Esta função dada, é uma função recursiva que usa 4 inteiros como argumentos, “move\_number” – o número de blocos que o carro irá percorrer em um único movimento, “position” – posição em que o carro se encontra no movimento atual, “speed” – velocidade do carro numa determinada posição e “final\_position” – posição final a que o carro terá de chegar. Em cada iteração, a função atualiza todos estes valores (à exceção da final position) e conta o número de movimentos, guardando-os em “solution\_1”.

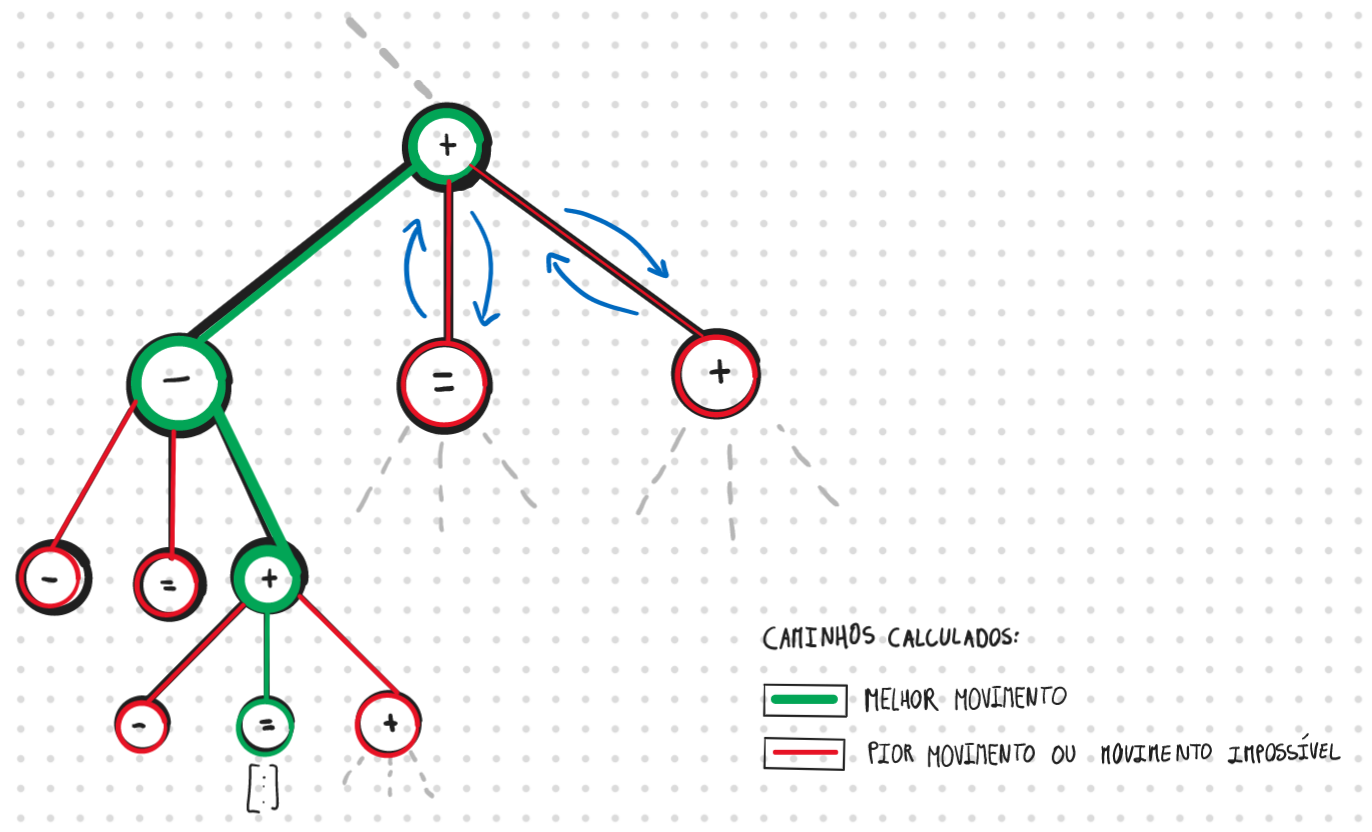
Se a posição não for igual à posição final e a velocidade não for igual a 1, ou seja, enquanto o carro não parar, testa-se qual a melhor opção entre descer velocidade, manter velocidade ou aumentar velocidade. Para cada uma destas opções, segue-se um “if statement” que averigua se a nova velocidade se situa entre o intervalo 0 a 9 e se a nova posição é menor ou igual à posição final.

Em seguida, se a nova velocidade cumprir todos os requisitos anteriormente impostos, existe um ciclo for a percorrer todos os blocos da estrada que a nova velocidade irá avançar. Aqui, certifica-se que em nenhum destes blocos, a velocidade é ultrapassada.

Finalmente, se o último “if statement” se verificar, a função irá ser executada novamente com os argumentos “move\_number”, “position” e “new\_speed” atualizados.

A função recursiva “solution\_1\_recursion”, faz uma pesquisa em árvore, isto é, cada vez que se executa a função, traça-se um caminho possível até à casa final. Se durante este percurso alguma casa não verificar as condições impostas, a função vai ter de voltar ao nó anterior da árvore e calcular a partir daí outras possibilidades de caminho.

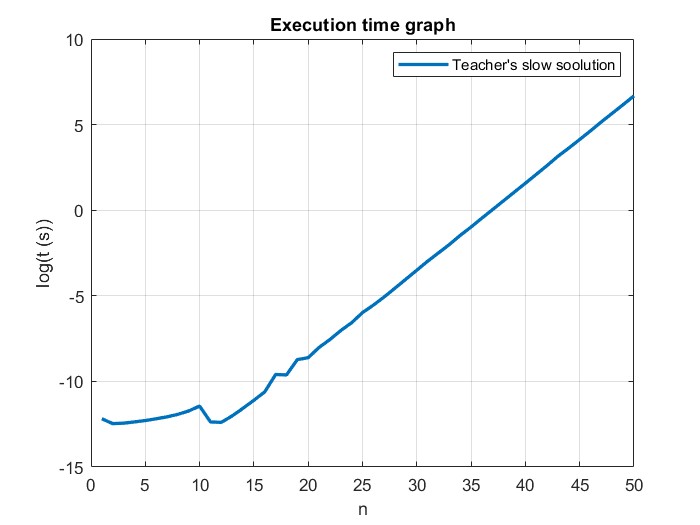
 No início do programa, estes “andar para a frente e para trás” não exigem muitos cálculos e são relativamente rápidos de calcular. No entanto, a cada passo, a árvore vai ficando cada vez mais complexa e a pesquisa torna-se cada vez mais intensiva. Através da imagem abaixo é possível constatar o referido.

****

**Fig.2 –** Ilustração da pesquisa em árvore

Para cada nó são gerados outros três. Estes representam as velocidades possíveis: pelo nó da esquerda diminui-se a velocidade do nó anterior, pelo nó do meio, mantém-se a velocidade e pelo nó da direita, aumenta-se a velocidade.

O algoritmo verifica para o primeiro nó gerado (com a velocidade mais pequena), se o caminho é possível. Se sim, avança para o próximo nó, se não, volta ao nó anterior e testa para outra velocidade.  Na prática, ao correr o programa com a função dada, observamos que o tempo do CPU aumenta exponencialmente.

Mostra-se que esta função tem uma pesquisa lenta visto que imposto o limite de 1 hora, o programa apenas chegou a

**Fig.3 –** Gráfico do tempo de execução da função recursiva fornecida pelo professor

Uma imagem com texto, monitor, eletrónica, captura de ecrã

Descrição gerada automaticamente

**Fig.4 –** Ficheiro pdf gerado pela função recursiva fornecida pelo professor

## **Solution\_1\_recursion (melhorada)**

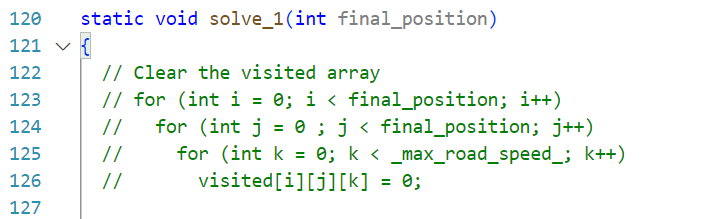
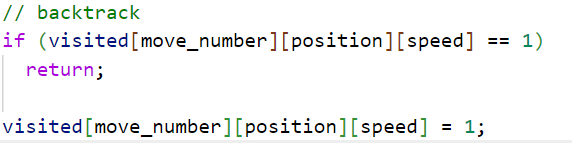
Seguindo o problema de recursividade e instabilidade da função dada, a nossa primeira ideia foi a de mudar a ordem pela qual a função testa a melhor velocidade para o carro. A função calcularia primeiro um possível caminho aumentado a velocidade do carro, depois testaria um caminho quando ficasse com a mesma velocidade do carro e por último, calcularia um caminho se descêssemos a velocidade do carro.

Em teoria, a velocidade de execução deste algoritmo ia aumentar, no entanto, mais tarde, percebemos que independentemente da ordem pela qual a função testa as velocidades, o melhor caminho poderá ser constituído por uma mesma redução ou aumento das velocidades, calculadas apenas por ordens diferentes.

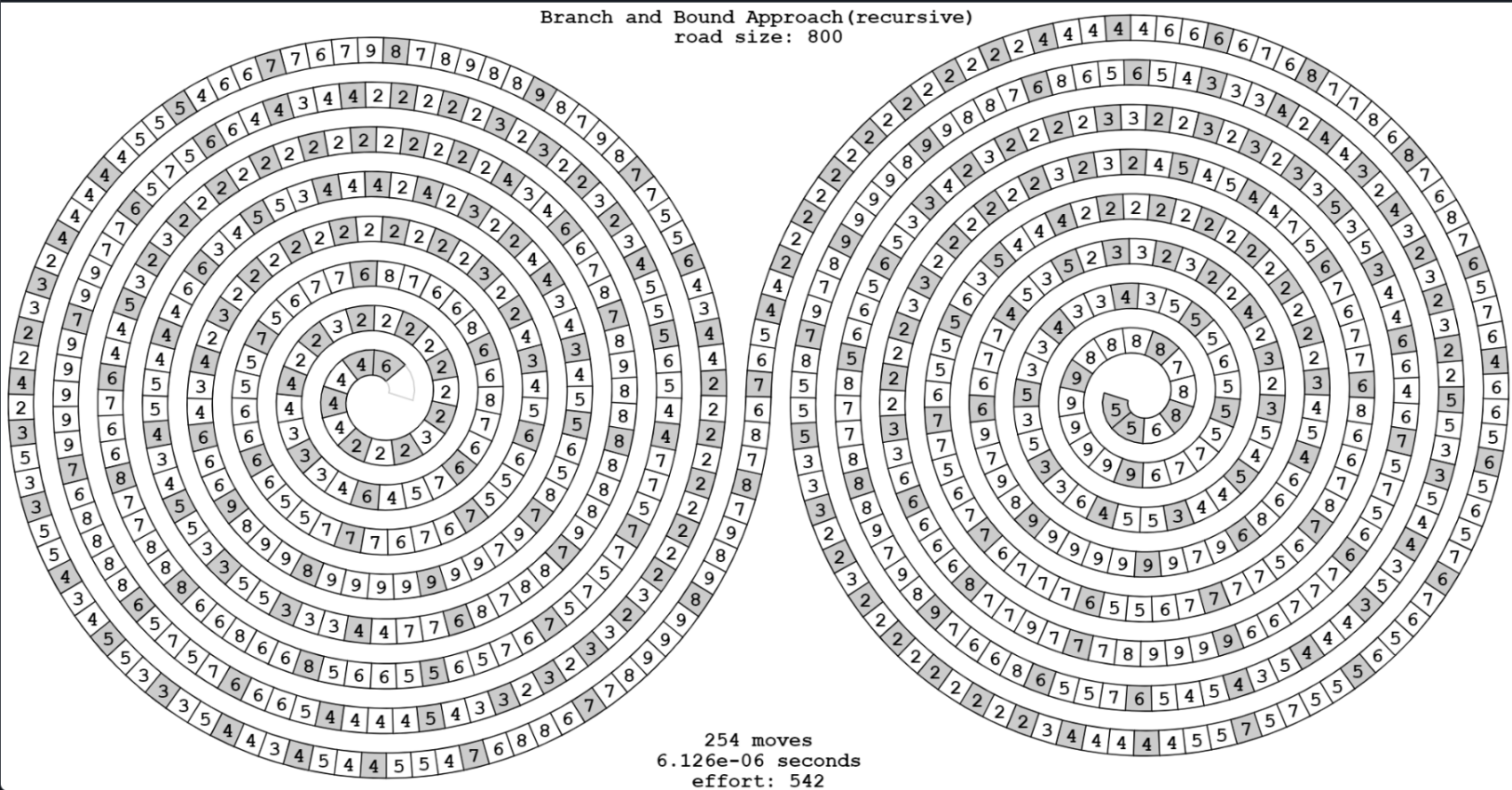
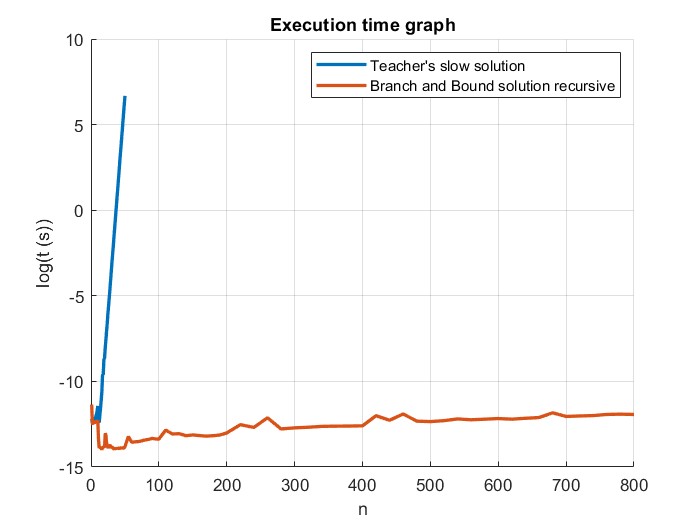
Desta forma, decidimos incluir um novo if statement que retorna assim que possível uma solução de um caminho, sem necessitar de calcular as outras opções. A condição retorna imediatamente o cálculo do primeiro caminho válido.

**Fig.5 –** Condição lógica da função recursiva melhorada

Esta aperfeiçoamento à função resulta no uso de uma perspetiva algorítmica “Branch and Bound”, que resolveu este problema de uma maneira relativamente rápida, diminuindo drasticamente o tempo de execução.

De modo a incluir programação dinâmica, tentámos implementar uma abordagem algorítmica chamada de Back Tracking. Esta abordagem vem tentar tornar ainda mais eficiente o programa. Testámos este método implementando um array booleano de 3 dimensões (move\_number, position e speed) chamado visited. À medida que o programa ia correndo, com esta solução, era guardado no array o move\_number, a posição e a velocidade. Criámos uma condição em que caso a solução já tivesse encontrado uma solução com os valores de move\_number, posição e velocidade, este iria utilizar a solução já encontrada para aqueles valores.  Sempre que esta solução era chamada, seria necessário limpar o array visited visto que na posição inicial, todas as variáveis presentes no array são 0. Ao corrermos esta solução verificámos que o esforço e o tempo de execução aumentaram por isso decidimos não a incluir na solução.

**Fig.6 –** Código implementado para a abordagem Back Tracking

Esta função apresenta uma evolução completa em relação aos tempos de execução com a função dada, como é possível verificar no gráfico abaixo.

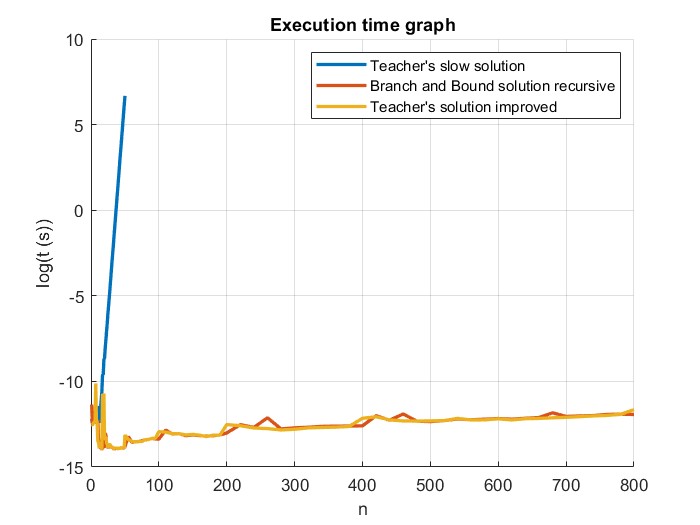
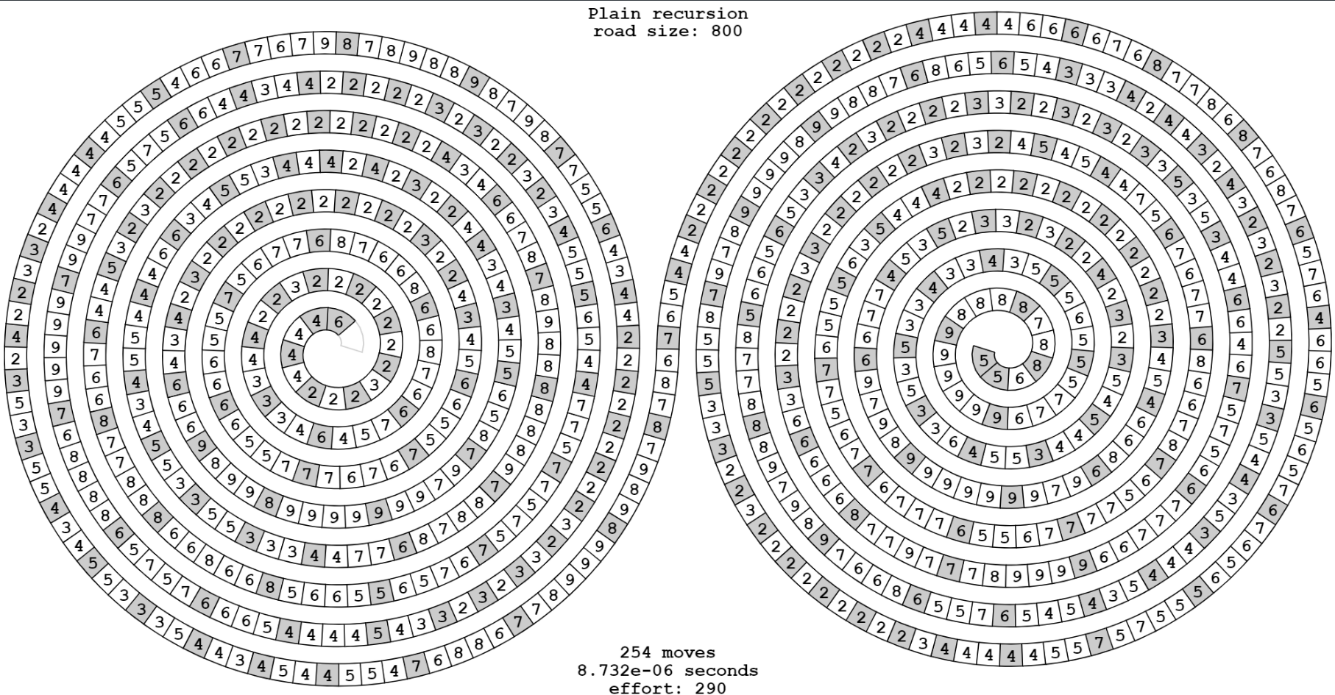
**Fig.7 –** Gráficos do tempo de execução da função recursiva melhorada e lenta

**Fig.8 –** Ficheiro pdf gerado pela função recursiva fornecida melhorada

## **Solution\_2\_Recursion**

Desenvolvemos outra solução recursiva parecida com a anterior. Esta tem um modo de correr diferenciado: pretende atualizar os valores retornados desde o pior até ao melhor. Para isto, adicionámos uma condição if caso esta já tenha encontrado uma solução melhor, ou seja, uma solução com menos esforço irá parar a execução.

Em vez de se calcular todas as diferentes combinações do caminho que o carro tem de percorrer, a função retorna instantaneamente uma solução, porque esta “será a melhor solução”. Desta forma, na primeira execução, o efford situa-se nos 800, depois na segunda execução reduzirá e assim por diante. Quando numa execução, a solução para certos movimentos já foi encontrada, retorna ou seja interrompe a sua execução.

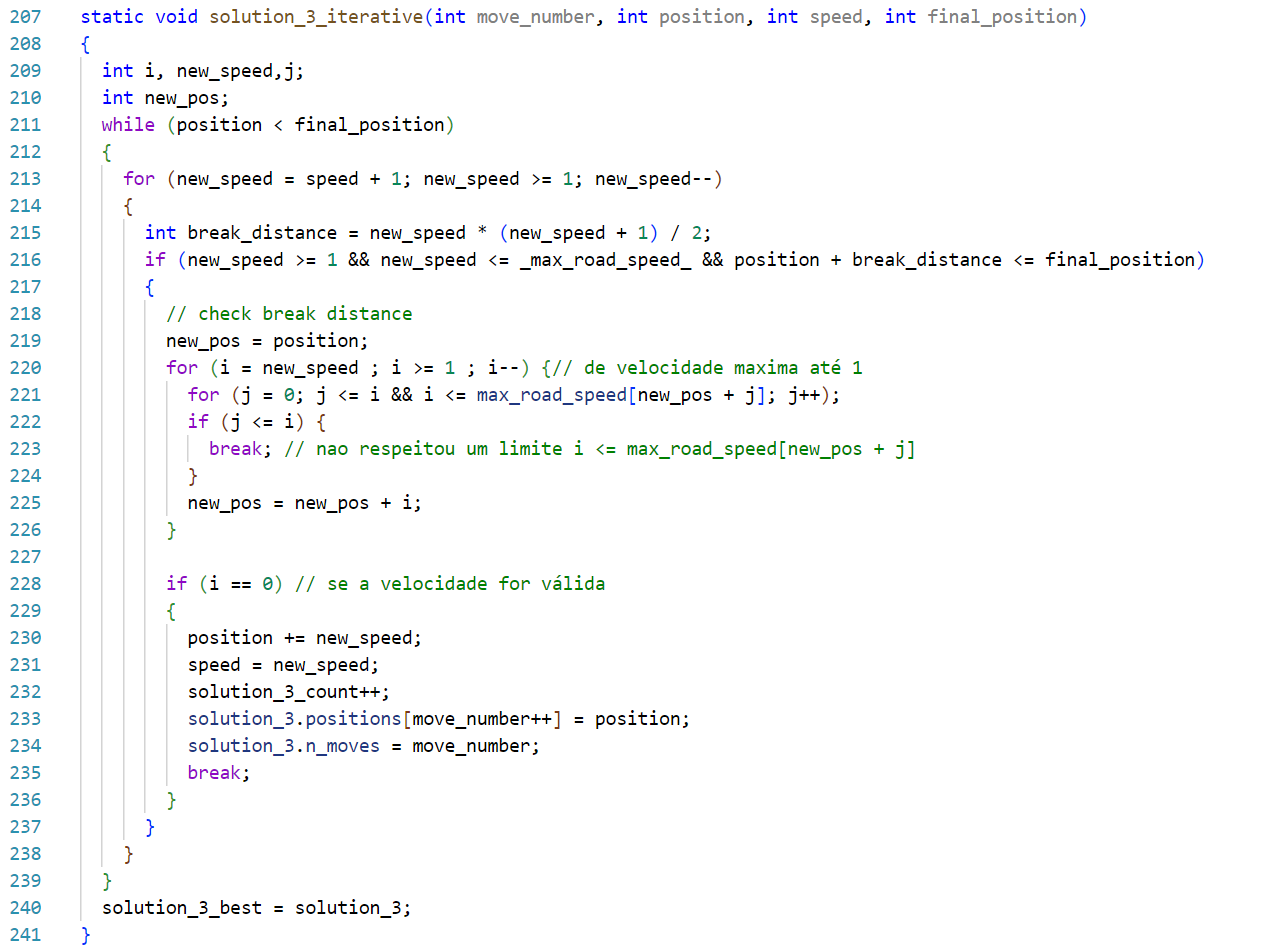
Comparando esta solução com a função dada e com a função recursiva anterior, nota-se que, por muito relativa que seja a diferença, uma ligeira progressão a nível de tempo de execução.

**Fig.9 –** Gráficos do tempo de execução das soluções recursivas criadas

**Fig.10 –** Ficheiro pdf gerado pela segunda função recursiva criada

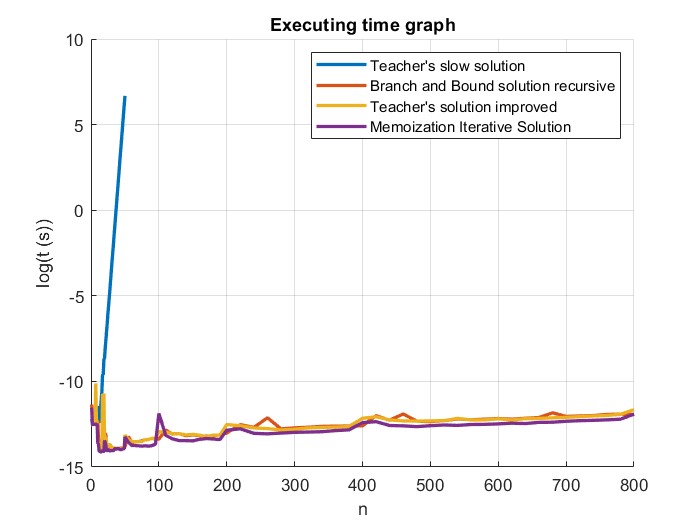
## **Solution\_3\_Iterative**

Antes de criar qualquer solução iterativa, já sabíamos que em norma, as funções iterativas são mais rápido que as recursivas. Isto porque nas funções iterativas, não precisamos de guardar imediatamente um resultado na stack. Significa, deste modo, menos instruções e assim, menos ciclos de CPU.

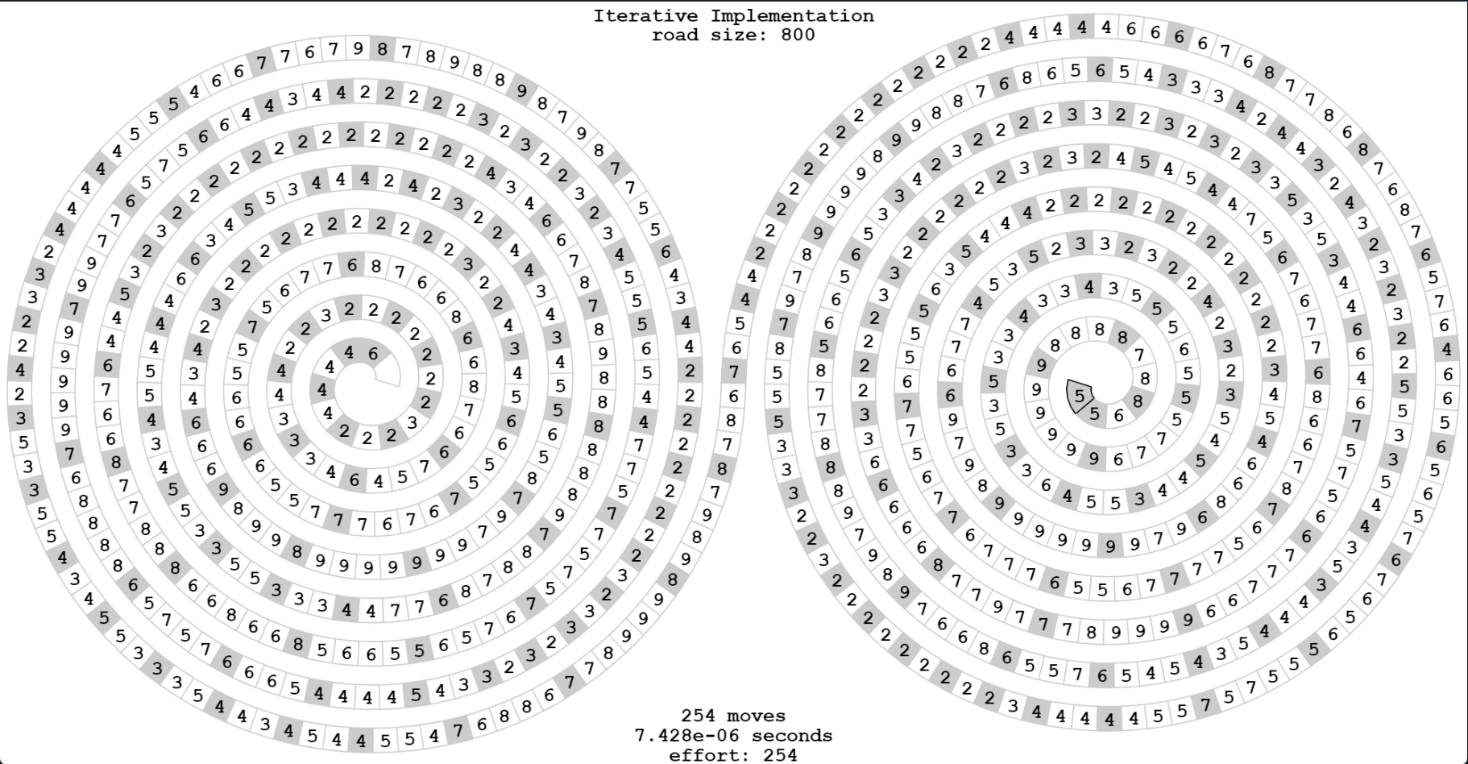
A nossa função iterativa tem por base a distância de travagem, ou seja, este algoritmo, para cada velocidade possível e começando sempre pela velocidade maior, irá calcular a distância de travagem, sendo esta o somatório das velocidades, começando na maior velocidade e ir decrementando uma unidade até esta ser igual a um. Calculada a distância de travagem (a distância que este precisa de percorrer para conseguir parar), é verifica-se se se consegue percorrer a distância de travagem sem exceder nenhum limite de velocidade. Caso exceda algum, irá verificar para outra velocidade, fazendo o mesmo algoritmo para esta. Caso não exceda, avança para a próxima posição com novos valores.

**Fig.11 –** Código da primeira solução iterativa criada

O algoritmo que desenvolvemos tem por base um ciclo while que irá ser repetido enquanto a posição atual for menor do que a posição final. Dentro deste ciclo utiliza-se a possibilidade de a velocidade aumentar uma unidade como primeira tentativa para o algoritmo ser mais rápido e, com esta velocidade, é calculada a distância de travagem. É verificado se a nova velocidade é maior ou igual do que um, se esta também é menor do que a velocidade máxima possível, ou seja, nove e se a soma desta com a distância de travagem é menor ou igual do que a posição final, se esta condição se verificar avança para a próxima instrução, caso não se verifique testa a mesma condição para o próximo caso (a velocidade manter-se). Se a condição anterior se verificar, avança para os dois ciclos for’s presentes nas linhas 220 e 221 em que é avaliado para cada velocidade se é excedido algum limite no número de casas percorridas. Se todos os limites se cumprirem é guardado a posição e a velocidade e executado novamente o ciclo while com os novos valores.

Ao comparamos esta função com todas as outras já apresentadas, observamos que esta é a melhor até à data, dado principalmente à diferença da quantidade de instruções entre funções recursivas e iterativas.

**Fig.12 –** Gráficos do tempo de execução das soluções recursivas mais a solução iterativa

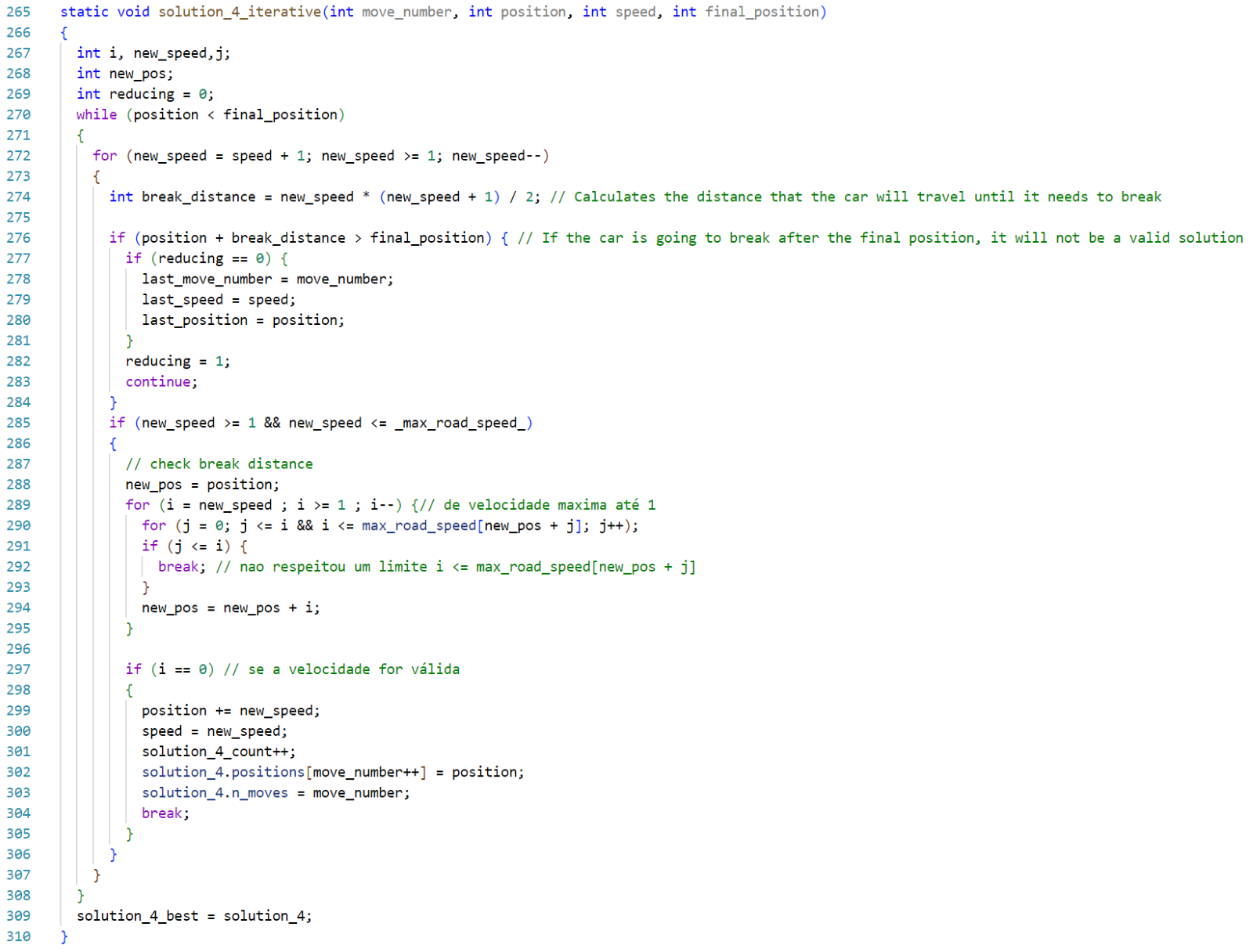


**Fig.13 –** Ficheiro pdf gerado pela primeira solução iterativa criada

## **Solution\_4\_Iterative (Programação Dinâmica)**

 Na última solução desenvolvida, procurámos incluir efetivamente um método de programação dinâmica para levar mais além os tempos de execução e otimização do programa. Com a programação dinâmica, tende-se a guardar soluções de problemas para uso futuro.

Para esta função iterativa, a linha de pensamento é muito parecida à função anterior “solution\_3\_iterative”. Usámos novamente um ciclo while que itera enquanto a posição for menor do que a posição final. Depois, são testadas as velocidades possíveis na linha 272 através de um ciclo for e calcula-se a distância que o carro necessita para travar logo depois. De seguida, conclui-se que as soluções nas quais o carro para só depois da posição final, não são válidas e por isso, é forçada uma redução através da variável “reducing”, que vai ter o valor 1. Após isto, a partir da linha 285, se a velocidade for válida, verifica-se a distância de travagem ao percorrer as velocidades desde a velocidade máxima até 1 e para cada velocidade, se esta respeita o limite de imposto. Se a distância de travagem das velocidades não for correta, o carro não poderá andar com a sua velocidade atual.



**Fig.14 –** Código da segunda função iterativa criada

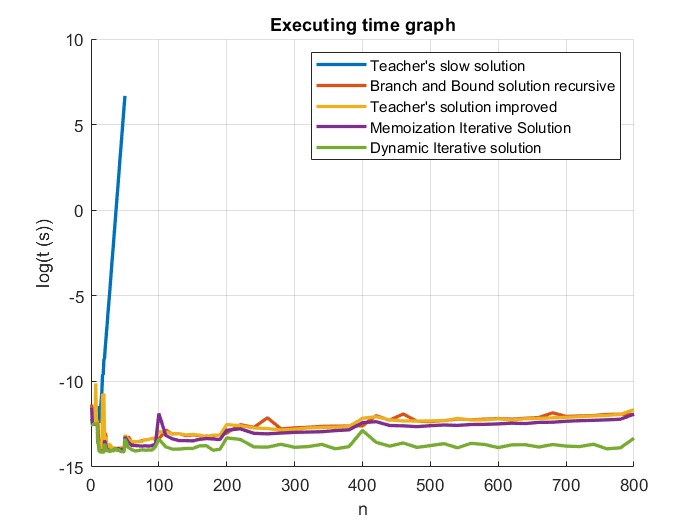
Mas é no método estático “solve\_4”, que mais se difere. Este método usa o resultado da melhor solução da solution\_3\_iterarive e calcula onde é que é necessário começar a reduzir a velocidade. Nota-se que enquanto a posição final for menor do que 1 ou maior que o tamanho máximo da estrada, não é contabilizada.

No anexo observa-se que ao chamar a função “solution\_4\_iterative”, em vez de se inicializar as variáveis com zero de novo, a função irá ser chamada já com o “last\_move\_number”, “last\_position” e “last\_speed”. É desta forma que se reaproveita o código previamente calculado e o tempo de execução diminui.

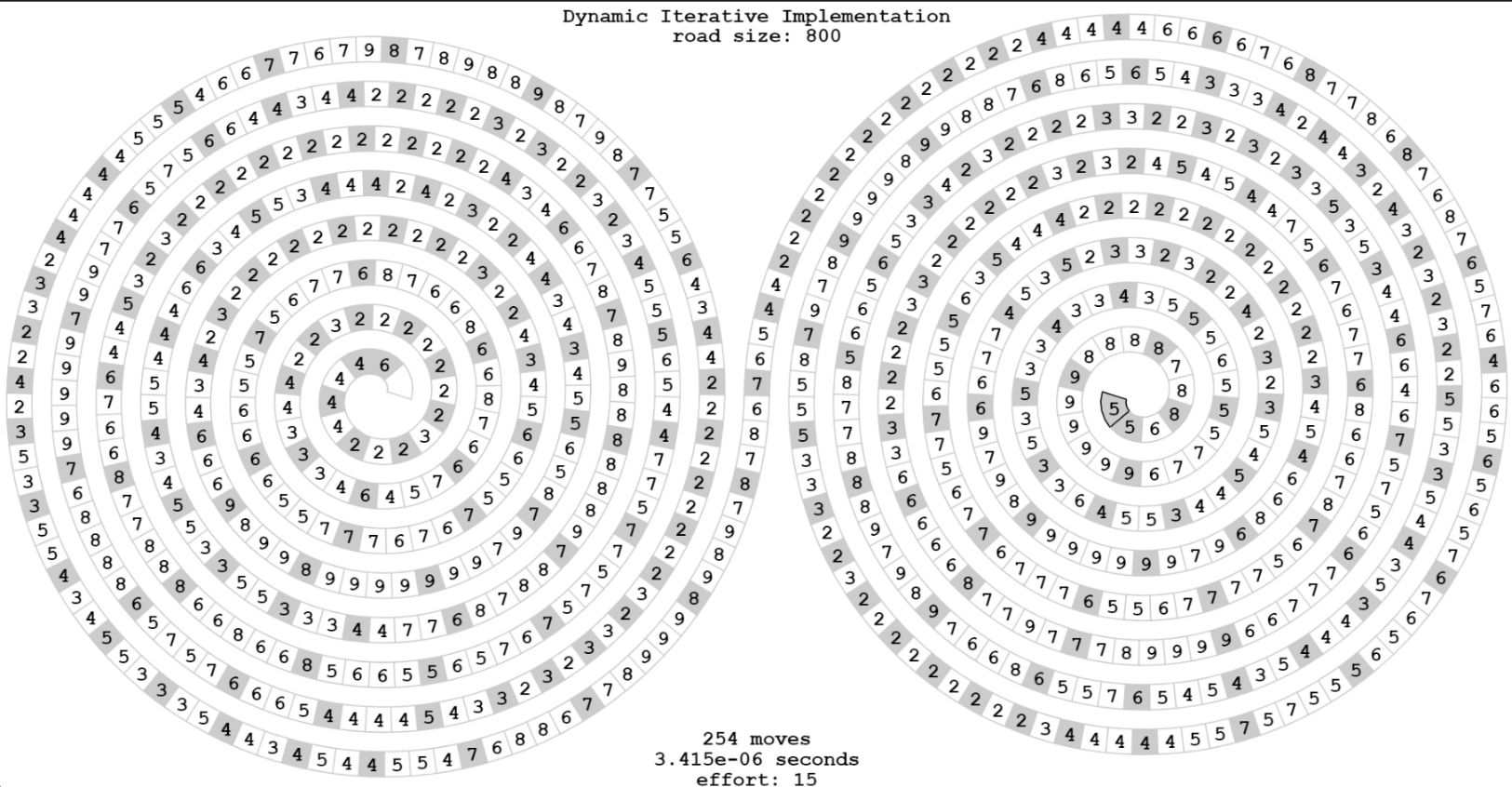
Uma imagem com texto

Descrição gerada automaticamenteAssim, numa última comparação, chegamos à conclusão que juntando funções iterativas com programação dinâmica, a diferença entre todos os outros métodos usados é abismal.

**Fig.15 –** Código da função solve\_4



**Fig.16 –** Gráficos do tempo de execução de todas as funções criadas



**Fig.17–** Ficheiro pdf gerado pela solução iterativa de programação dinâmica criada

# Uma imagem com texto Descrição gerada automaticamente***3.Código***

**Uma imagem com texto

Descrição gerada automaticamenteCódigo em C**

Uma imagem com texto

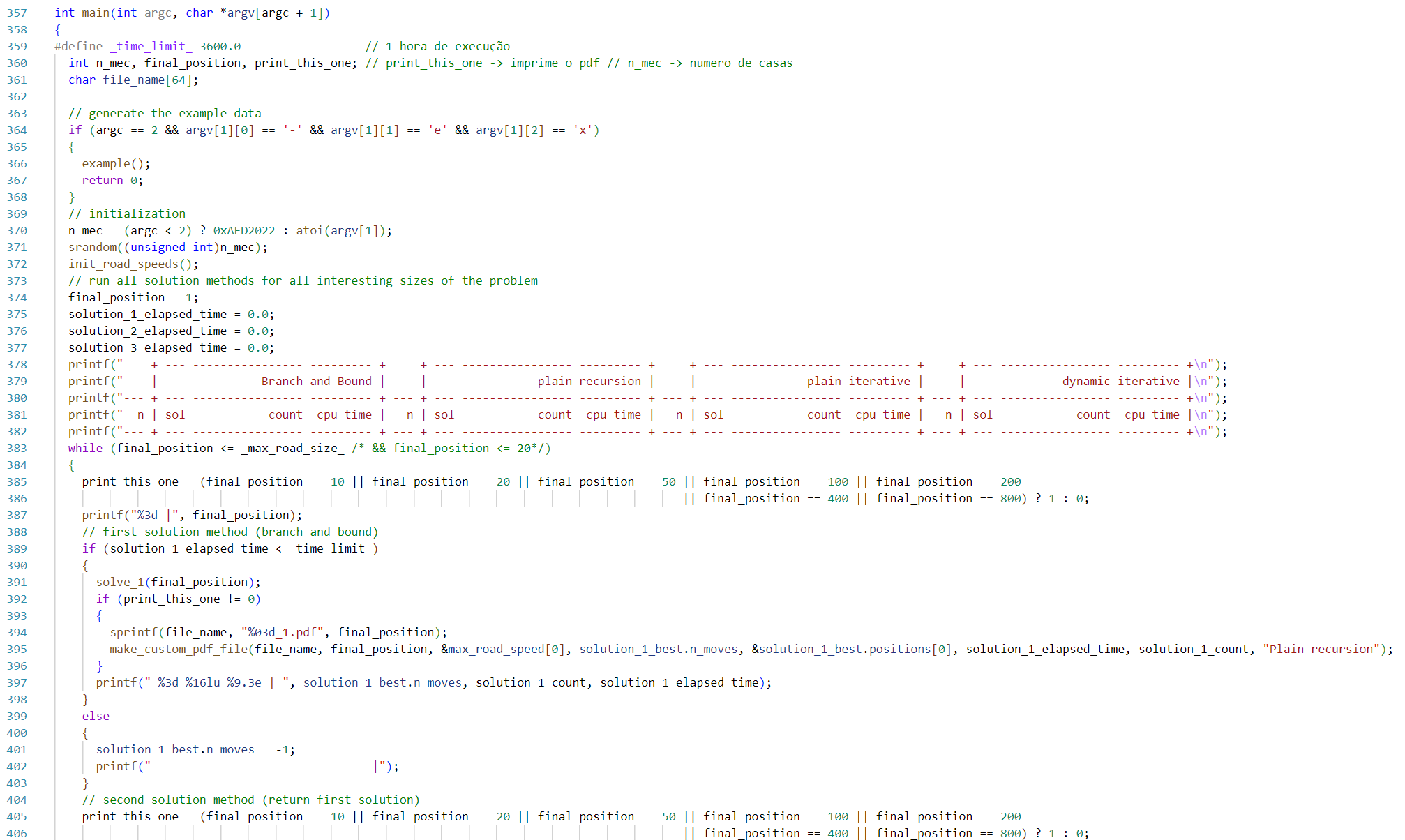
Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamente

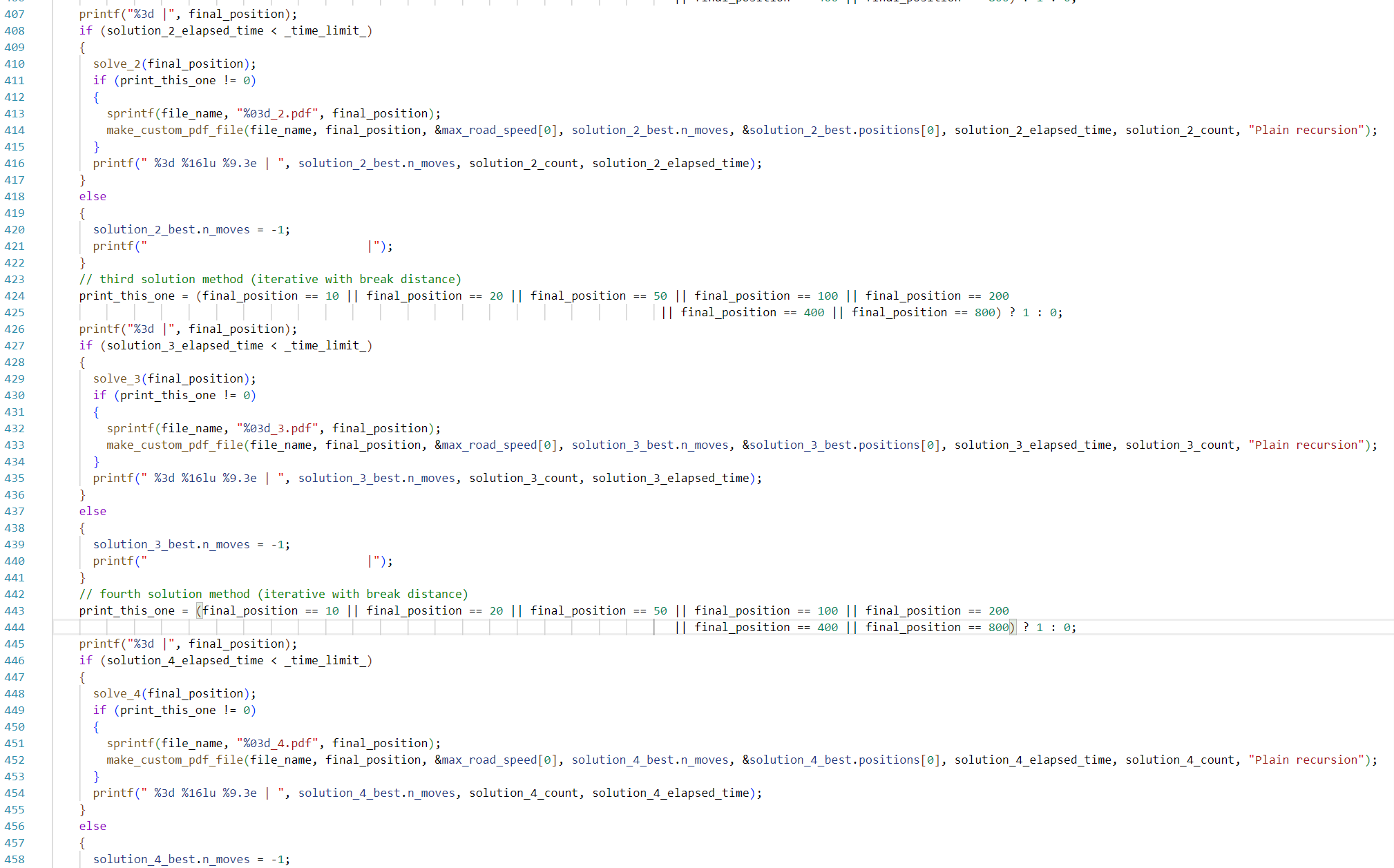
Uma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

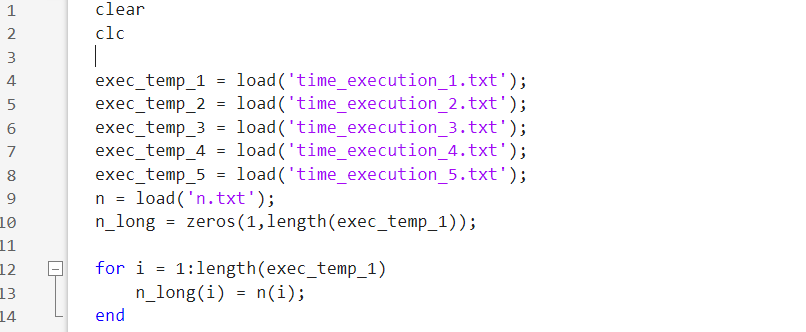
Descrição gerada automaticamente

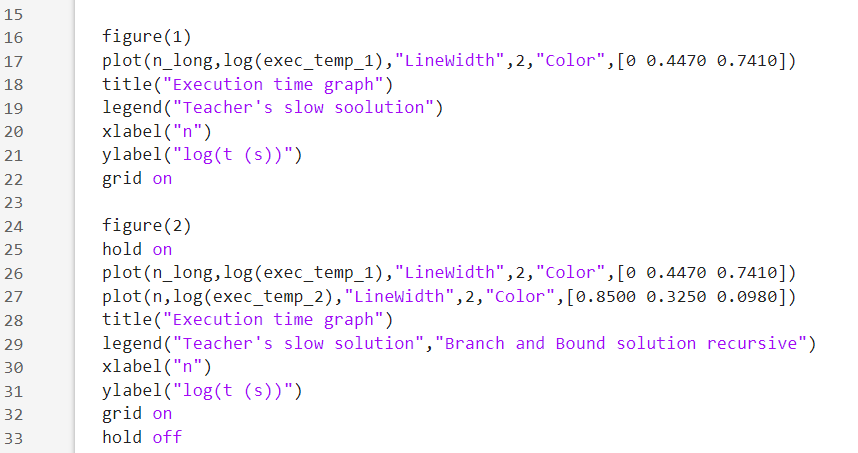
Uma imagem com texto

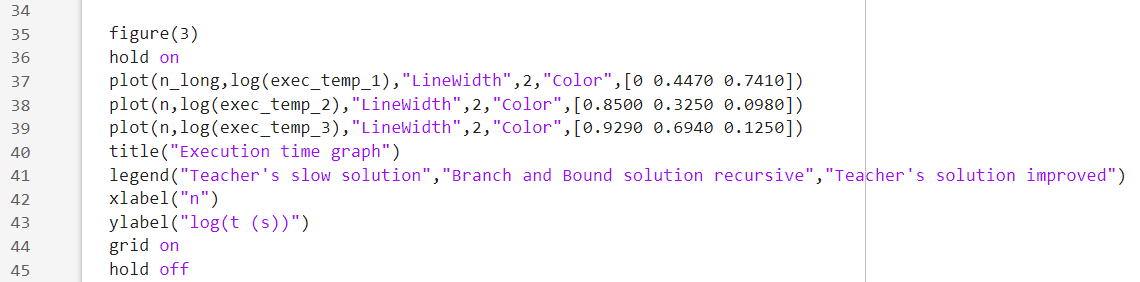
Descrição gerada automaticamente

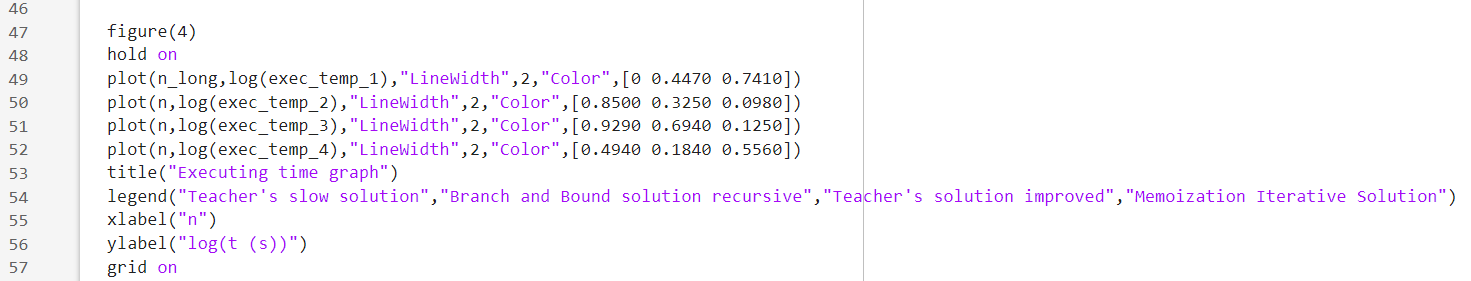


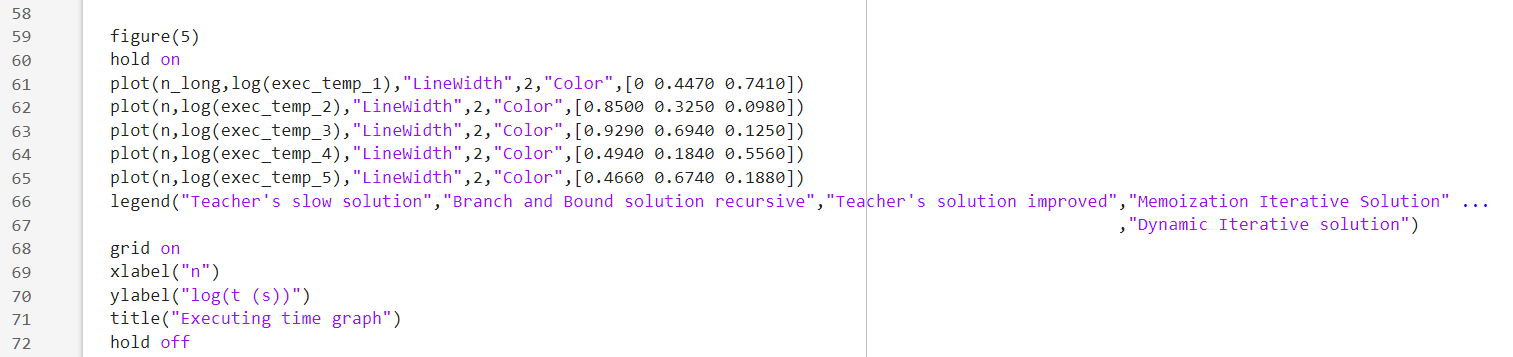
**Código em Matlab**

****

****

****

****

****

# ***Conclusão***

Com a realização deste trabalho concluímos que existem várias maneiras de abordar o problema proposto. Estas soluções utilizam métodos recursivos e iterativos. Constatamos que os métodos iterativos são mais rápidos e eficientes que os recursivos através da análise de gráficos de execução e dos ficheiros gerados pelo programa.

Observámos que, na criação de uma solução usando programação dinâmica, esta foi a melhor solução até à data visto que o gráfico do seu tempo de execução e o esforço obtido foram os menores.

# ***Webgrafia***

GEEKS (2022), Branch and Bound Algorithm, [visitado em 5 de dezembro de 2022]: <https://www.geeksforgeeks.org/branch-and-bound-algorithm/>

GEEKS (2022), Back Traking Algortithm, [visitado em 5 de dezembro de 2022]: <https://www.geeksforgeeks.org/backtracking-algorithms/>

GATWIRI (2022), Dynamic Programming In Javascript using Tabulation, [visitado em 6 de dezembro de 2022]:

<https://www.section.io/engineering-education/dynamic-programming-in-javascript-using-tabulation/>

GEEKS (2022), What is memoization? A Complete tutorial, [visitado em 5 de dezembro de 2022]:

<https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/>

SKIENA (2008), Backtracking, [visitado em 3 de dezembro de 2022]:

<https://guides.codepath.com/compsci/Backtracking>

HUANG (2021), Is Recursion Really Slower than Iteration?, [visitado em 22 de novembro de 2022]:

<https://edward-huang.com/2021/02/17/is-recursion-really-slower-than-iteration/>