Uma imagem com texto

Descrição gerada automaticamente

**Speed Run**

Algoritmos e Estruturas de Dados 2022

Professor Tomás Oliveira e Silva e Professor Pedro Miguel Ribeiro Lavrador

Trabalho realizado por:

João Nuno da Silva Luís (107403) | 50%

Diana Raquel Rodrigues Miranda (107457) | 50%

**Índice**

[Introdução 3](#_Toc120961717)

[Algoritmo fornecido 4](#_Toc120961718)

[Segundo Algoritmo 13](#_Toc120961719)

[Terceiro Algoritmo – Programação Dinâmica 15](#_Toc120961720)

[Resultados 19](#_Toc120961721)

[ **Tempos de execução finais** 22](#_Toc120961722)

[Conclusões finais 23](#_Toc120961723)

[Código C 24](#_Toc120961724)

[ Speed\_run.c 24](#_Toc120961725)

[Código MATLAB 29](#_Toc120961726)

[ Execution\_time.m 29](#_Toc120961727)

# **Introdução**

Este trabalho foi realizado no âmbito da disciplina de Algoritmos e Estruturas de Dados do 2º ano da Licenciatura em Engenharia Informática.

Foi-nos proposto desenvolver um algoritmo em C que determinasse o número mínimo de movimentos necessários para alcançar a posição final. No entanto, têm de ser respeitadas certas regras, que são as seguintes:

- O carro só pode aumentar 1 velocidade, reduzir 1 velocidade ou mantê-la;

- O carro começa no primeiro segmento da estrada com uma velocidade 0 e tem de atingir o último segmento de estrada com uma velocidade de um;

- O carro não pode em nenhum momento passar num segmento de estrada com uma velocidade superior à nela permitida.

Com isto em mente, temos que a finalidade principal do nosso trabalho é conseguir otimizar o algoritmo fornecido pelo professor de modo a tornar possível atingir a posição 800. Para além disto, se for possível, desenvolver um novo algoritmo que resolva o problema com o menor tempo de execução possível.

# **Algoritmo fornecido**

O algoritmo fornecido segue o conceito de depth first search, que é um algoritmo utilizado para realizar uma procura em árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder.

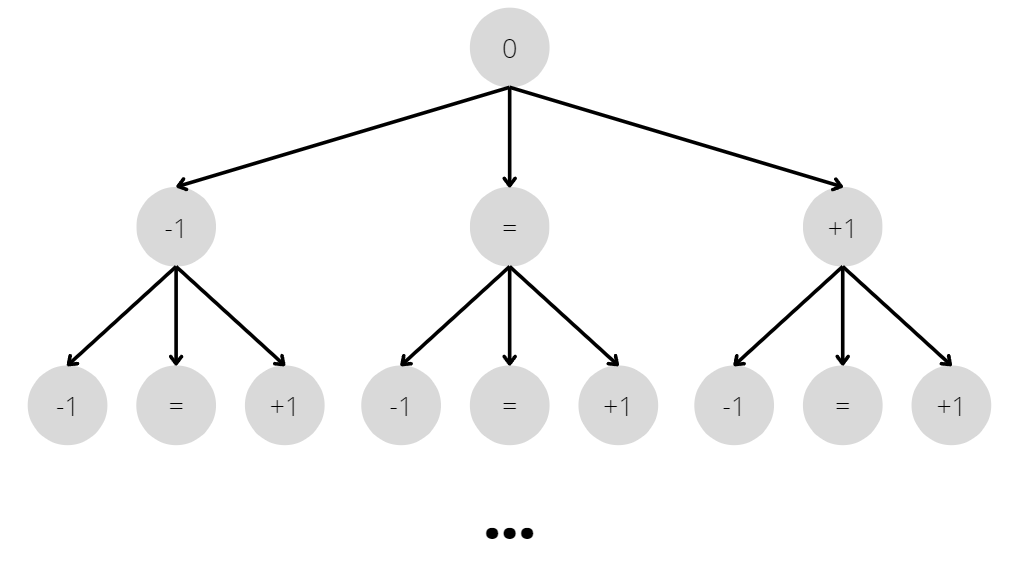
Posto isto, temos no nosso caso de estudo (Speed run) o nó raiz como a primeira posição de onde o carro irá arrancar e desse nó irão sair três novos nós, um com a opção de aumentar a velocidade, um com a opção de a manter e outro com a opção de a diminuir, e só depois de todas as possibilidades terem sido percorridas, é que o algoritmo vai retroceder e escolher o melhor caminho.

Figura 1 – Exemplo gráfico da árvore percorrida pela função fornecida.

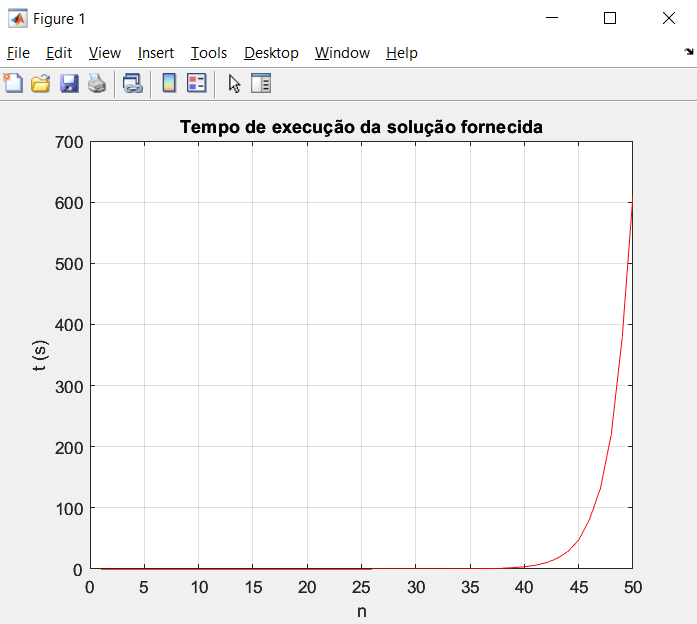
 Fazendo o gráfico do tempo de execução do algoritmo fornecido, antes de fazer qualquer otimização, obtemos um gráfico exponencial a partir da posição 40.

Figura 2- Tempo de execução da solução fornecida

Este tipo de gráfico não é o melhor para visualizarmos os nossos dados, nem calcular a reta de ajuste, pelo que podemos aplicar um logaritmo na base dez ao eixo dos yy (eixo dos tempos), e ao mesmo tempo calcular a reta de ajuste aos dados obtidos, que nos permite fazer uma previsão do tempo de execução deste algoritmo até à posição final.

Através desta resta de ajuste, calculamos que a solução fornecida iria demorar 1.114e+162 segundos a chegar à posição 800.

Figura 3- Reta de ajuste aos dados da solução fornecida

Estes gráficos provam que apesar do algoritmo ser capaz de chegar à solução correta, o mesmo irá demorar muitíssimo tempo a resolver o problema para as 800 posições.

O nosso método para tornar este algoritmo mais eficiente foi pensar numa maneira de otimizar a pesquisa em árvore e diminuir o número de ramos visitados.

**1ª melhoria** – Tentar acelerar primeiro.

Como o nosso objetivo é chegar à posição final o mais rápido possível, vamos visitar primeiro o nó em se verifica o aumento da velocidade e depois visitamos os outros.

Uma imagem com texto

Descrição gerada automaticamentePara isso, alterámos o seguinte pedaço de código:

Uma imagem com texto

Descrição gerada automaticamentePara:

Assim, vai começar a sua pesquisa sempre pelo nó que acelera.

Com o código fornecido e em execução durante uma hora (com o número mecanográfico 107457) é possível chegar à posição 50 em 6.121e+02 segundos. Com esta alteração foi possível, durante o mesmo tempo de execução, chegar à posição 50 em 5.935e+02 segundos. É uma melhoria mínima, pois mesmo a começar a pesquisa pelo nó que acelera o algoritmo vai verificar todos os ramos da árvore possíveis, o que não melhora muito o tempo de execução.

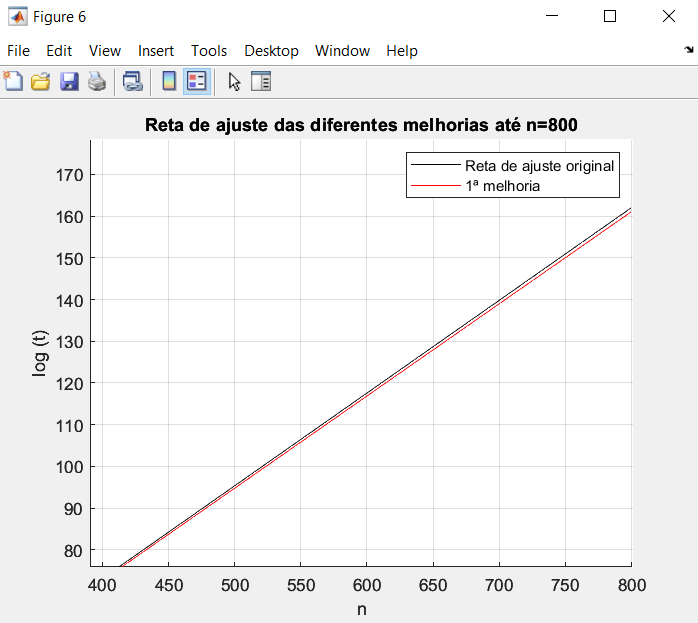
****

Figura 4- Comparação da reta de ajuste até à posição n=800

**2ª melhoria** – Acrescentar um if no código da função fornecida.

Uma imagem com texto

Descrição gerada automaticamente Este if verifica se o número de movimentos da solução que está a ser vista nesse momento já é maior do que o número de movimentos total da “melhor” solução anteriormente guardada. Se for maior, então o algoritmo pode parar essa procura, pois já não nos interessa uma vez que já temos uma solução melhor, o que torna possível cortar alguns ramos da árvore.

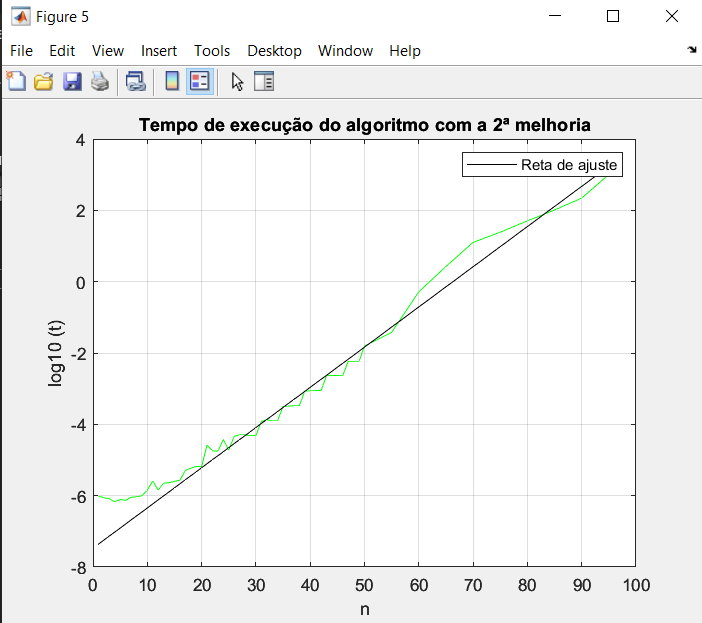
 Apesar de esta melhoria ser bastante simples, pois só acrescentámos duas linhas de código, é uma melhoria que nos permite reduzir o tempo de execução para metade. Com isto, durante uma hora de execução (com o número mecanográfico 107457), já conseguimos chegar à posição 95 em 1.009e+03 segundos.

Figura 5- Reta de ajuste aos dados da 2ª melhoria

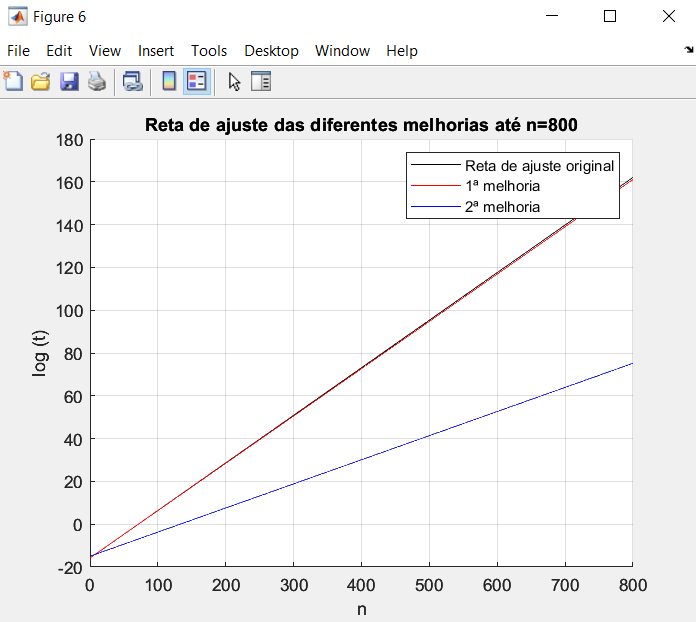


Figura 6- Comparação das diferentes retas de ajuste.

A 2º melhoria permite reduzir o tempo de execução para metade.

**3ª melhoria** – Acrescentar um outro if no código da função fornecida.

Uma imagem com texto

Descrição gerada automaticamente

Neste segundo if, vai ser verificado se numa determinada posição foi possível, na solução já guardada, passar com uma velocidade maior do a que está a ser vista naquele momento. Se tiver sido possível então podemos abandonar a pesquisa desse ramo, pois interessa-nos andar sempre com a velocidade máxima possível.

 Esta melhoria, também sendo simples, torna possível, juntamente com as outras melhorias resolver o problema para as 800 posições, sendo possível agora chegar à posição 800 em 4.739e-06 segundos (com o número mecanográfico 107457).

Figura 7 - Reta de ajuste aos dados da 3ª melhoria

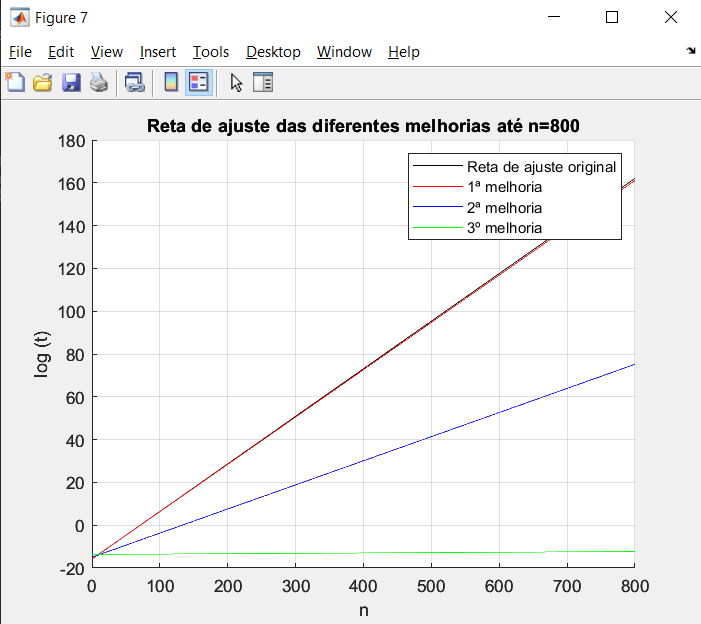


Figura 8- Comparação das diferentes retas de ajuste.

A 2ª e a 3ª melhoria foram baseadas no algoritmo de Branch and Bound, uma vez que a sua função é descartar logo um ramo se essa solução for pior que a “melhor” solução anteriormente encontrada e guardada.

# **Segundo** **Algoritmo**

Este algoritmo foi criado pensando numa maneira de descobrir a melhor solução para o problema percorrendo a estrada toda uma única vez.

Uma imagem com texto

Descrição gerada automaticamentePosto isto, começamos por criar um ciclo while que permitisse percorrer toda a estrada. Dentro desse ciclo temos um if que vai verificar se é possível aumentar, diminuir ou manter a velocidade.

Figura 9 - Função da Solução 2

Para conseguir fazer uma verificação que garantisse que em nenhum momento se iria desrespeitar as regras da estrada criámos uma função (respect\_limits) que tem como parâmetros de entrada a posição onde se encontra, a velocidade a que está a tentar seguir e a posição final da estrada. O objetivo desta é verificar se a velocidade que está a tentar seguir é válida, e para essa avaliação, o algoritmo verifica, se com essa velocidade teria tempo de travar se já se encontrasse perto do fim da estrada. Verifica, também, se essa velocidade respeita a velocidade de todos os segmentos por onde vai passar até chegar à posição seguinte.

Uma imagem com texto

Descrição gerada automaticamenteSe a velocidade cumprir estes dois requisitos a função vai retornar o valor 1, caso contrário retorna o valor 0.

Figura 10 - Função usada na solução 2 para verificar a velocidade

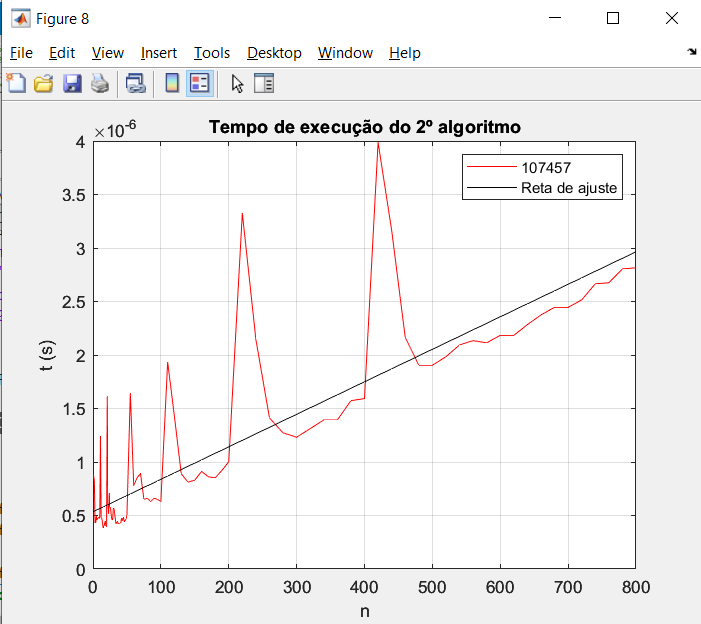
Com esta solução temos que o tempo de execução para a posição 800 é 8.285e-06 segundos (com o número mecanográfico 107457).

Figura 11-Reta de ajuste aos dados do 2º algoritmo

De notar que na Figura 11 não foi necessário aplicar logaritmos à construção da reta de ajuste, pois a mesma não é exponencial. Assim sendo, temos um gráfico do tempo (t(s)) de execução em função do número de segmentos da estrada (n).

# **Terceiro Algoritmo – Programação Dinâmica**

Para este algoritmo utilizámos a programação dinâmica, que consiste em dividir um problema de otimização em subproblemas mais simples e guardar a solução para cada, de modo que cada subproblema seja resolvido só uma vez.

No contexto do problema em estudo, o que começámos por pensar foi, por exemplo, numa estrada com 10 segmentos os saltos que o carro vai dar até começar a travar, por já estar perto do fim da estrada, vão ser os mesmo que numa estrada com 20 segmentos até essa posição, e assim em diante.

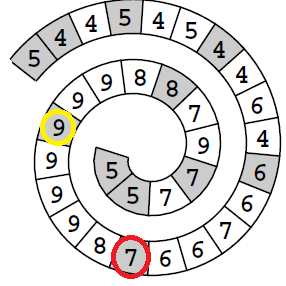
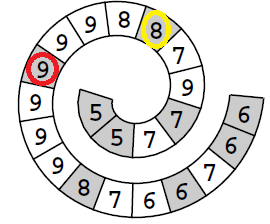
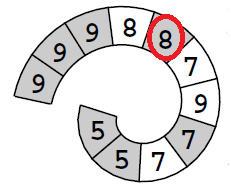


Figura 12 - 10 segmentos

Figura 13 - 20 segmentos

Figura 14 - 30 segmentos

As figuras acima demonstram o que foi descrito anteriormente, a posição marcada a vermelho é a posição onde o carro começa a travar, por já se encontrar perto do fim, e a amarelo está marcado a posição onde o algoritmo vai começar a nova procura, pois, as posições que estão para trás são iguais às já pesquisadas anteriormente.

Com este pensamento, criámos um algoritmo que segue a mesma ideia do segundo algoritmo, mas neste é criado um array com as posições onde o carro passou até à posição onde começa a travar, por se encontrar perto do fim da estrada, e é também guardado a velocidade com que ia nessa posição. Assim, na próxima pesquisa é reaproveitado esse array, e em vez da pesquisa começar no inico da estrada com velocidade 0, a pesquisa é iniciada na última posição desse array e com a velocidade guardada, uma vez que até aí as posições de paragem vão ser sempre as mesmas, o que torna a procura mais rápida e eficiente, evitando assim que faça duas vezes a mesma pesquisa.

Uma imagem com texto

Descrição gerada automaticamentePara esta solução foi criada uma estrutura nova (solution3\_t) para conseguir guardar o número de saltos, a velocidade, a posição em que ficou e o array das posições onde passou, e poder assim usar esses valores quando necessário.

Figura 15 - Estrutura criada para a solução 3

Foi também reaproveitado o código da solução 2, fazendo só algumas alterações, que são:

Uma imagem com texto

Descrição gerada automaticamente- Dentro do while foi acrescentado um if que só é executado na primeira vez em que o carro chega a uma posição com uma velocidade onde tem de começar a reduzir para respeitar os limites até ao fim da estrada.

Figura 16 - Função da Solução 3

Uma imagem com texto

Descrição gerada automaticamente- Na função que verifica os limites de velocidade foram alterados os valores de retorno. Como para esse algoritmo é necessário saber exatamente quando é que a verificação falha por se ultrapassar a posição final da estrada é retornado 1 quando isso acontece. De seguida é verificado se não é ultrapassado o limite de velocidade de nenhum segmento de estrada por onde passa, se for retorna 2. Por fim, se a velocidade passar estas duas verificações o valor de retorno é 0.

Figura 17 - Função usada na solução 3 para verificar a velocidade

Uma imagem com texto

Descrição gerada automaticamente Alterámos também a função que executa o código do terceiro algoritmo, para chamar a solução com o número de saltos, a posição e a velocidade guardadas na estrutura da solução 3.

Figura 18 - Função que chama a função da solução 3

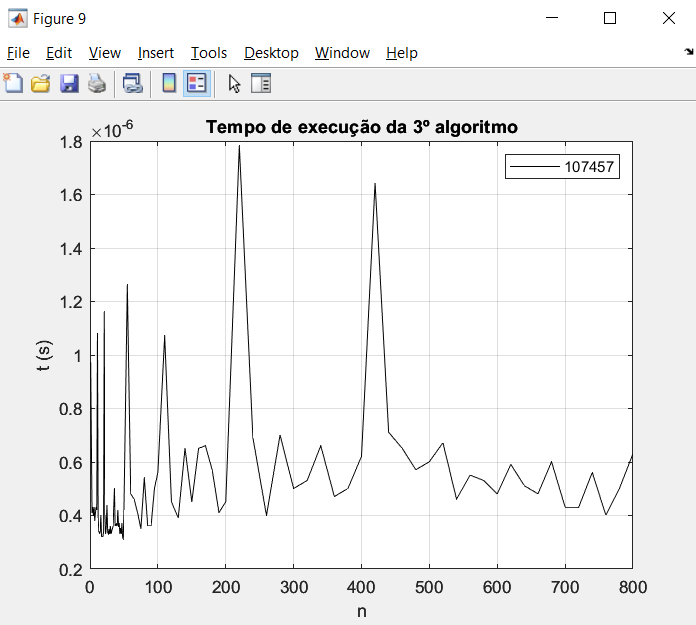
Com este algoritmo temos que o tempo de execução para a posição 800 é 1.472e-06 segundos (com o número mecanográfico 107457).

Figura 19- Dados do 3º algoritmo

Para este algoritmo, não fizemos a resta de ajuste aos dados visto que o gráfico é inconstante e tem bastante ruído. O gráfico é decrescente devido ao facto de que sempre que se inicia uma nova procura com um novo número de posição final o tempo é começado a zero, no entanto as primeiras posições possíveis já foram vistas e guardadas num array logo o tempo que demorou a fazer essa pesquisa anteriormente não vai ser contabilizado. Isto provoca estas inconsistências no gráfico.

# **Resultados**

Figura 20 - Resultados com solução do professor otimizada

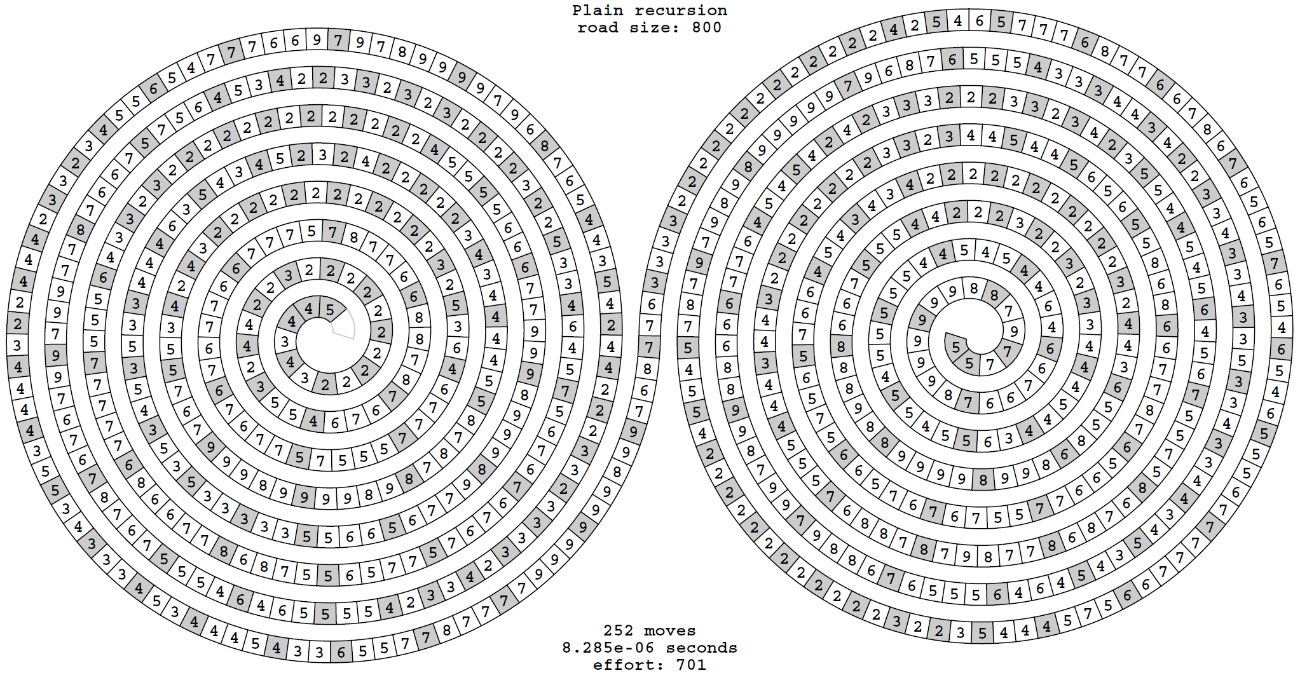


Figura 21 - Resultados com a segunda solução

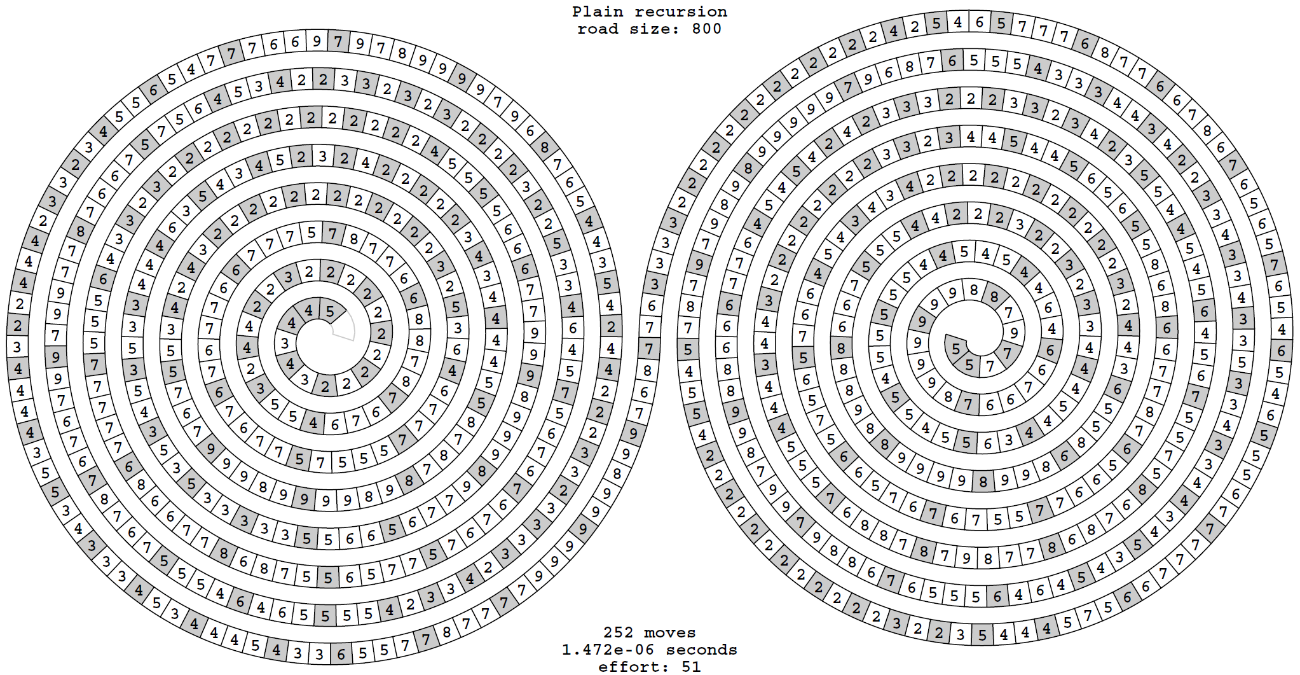
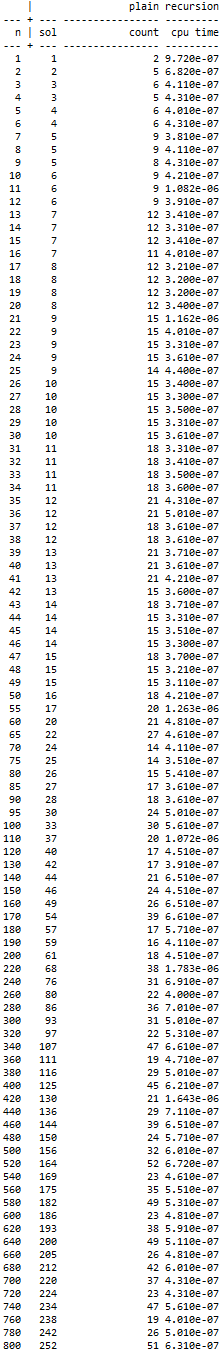


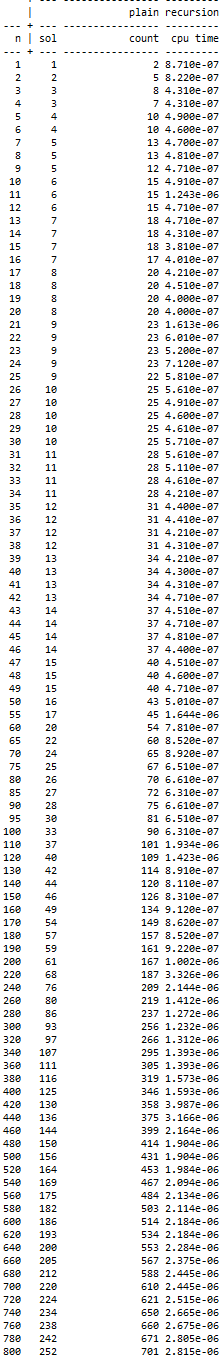
Figura 22 - Resultados com a terceira solução (Programação dinâmica)

Através destas imagens vemos que os resultados das 3 solução são iguais, ou seja, todos dão 252 saltos (com o número mecanográfico 107457). No entanto, os tempos de execução variam, o que nos permite concluir que a terceira solução é a melhor, uma vez que tem um tempo de execução menor. Estes resultados vão de encontro à teoria, pois esta solução usa programação dinâmica, que permite que a melhor solução para o problema seja combinada com a melhor solução pesquisada e guardada anteriormente, evitando assim que se volte a calcular uma coisa já antes vista, enquanto os outros dois algoritmos vão sempre calcular a solução partindo do zero.

## **Tempos de execução finais**

Uma imagem com texto

Descrição gerada automaticamente1º Algoritmo: 2º Algoritmo: 3º Algoritmo



# **Conclusões finais**

Com a realização deste trabalho, conseguimos aprofundar bastante os nossos conhecimentos em linguagem C e também adquirir novos conhecimentos sobre alguns algoritmos como o depth first search, o branch and bound e programação dinâmica. Antes do início da execução do projeto tínhamos pouco conhecimento sobre a otimização de algoritmos e como é que uma simples alteração num código consegue fazer tanta diferença na complexidade computacional e no tempo de execução.

Sentimos alguma dificuldade a entender a solução já fornecida devido à função recursiva e à falta de experiência em trabalhar na linguagem em C. No entanto essas dificuldades foram diminuindo à medida que tínhamos mais aulas e que fazíamos mais pesquisas sobre a matéria, conseguindo assim ultrapassá-las.

Por fim, podemos afirmar que os objetivos propostos foram alcançados com sucesso, visto que conseguimos implementar 2 soluções de forma eficiente e otimizar a solução já fornecida com sucesso, chegando assim a ter 3 algoritmos que atingem a posição final com o menor número de saltos no tempo de execução de microssegundos.

# **Código C**

## Uma imagem com texto Descrição gerada automaticamenteUma imagem com texto Descrição gerada automaticamenteSpeed\_run.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 mesa

Descrição gerada automaticamente

Uma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamente

# **Código MATLAB**

## Execution\_time.m

Uma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamenteUma imagem com texto

Descrição gerada automaticamente