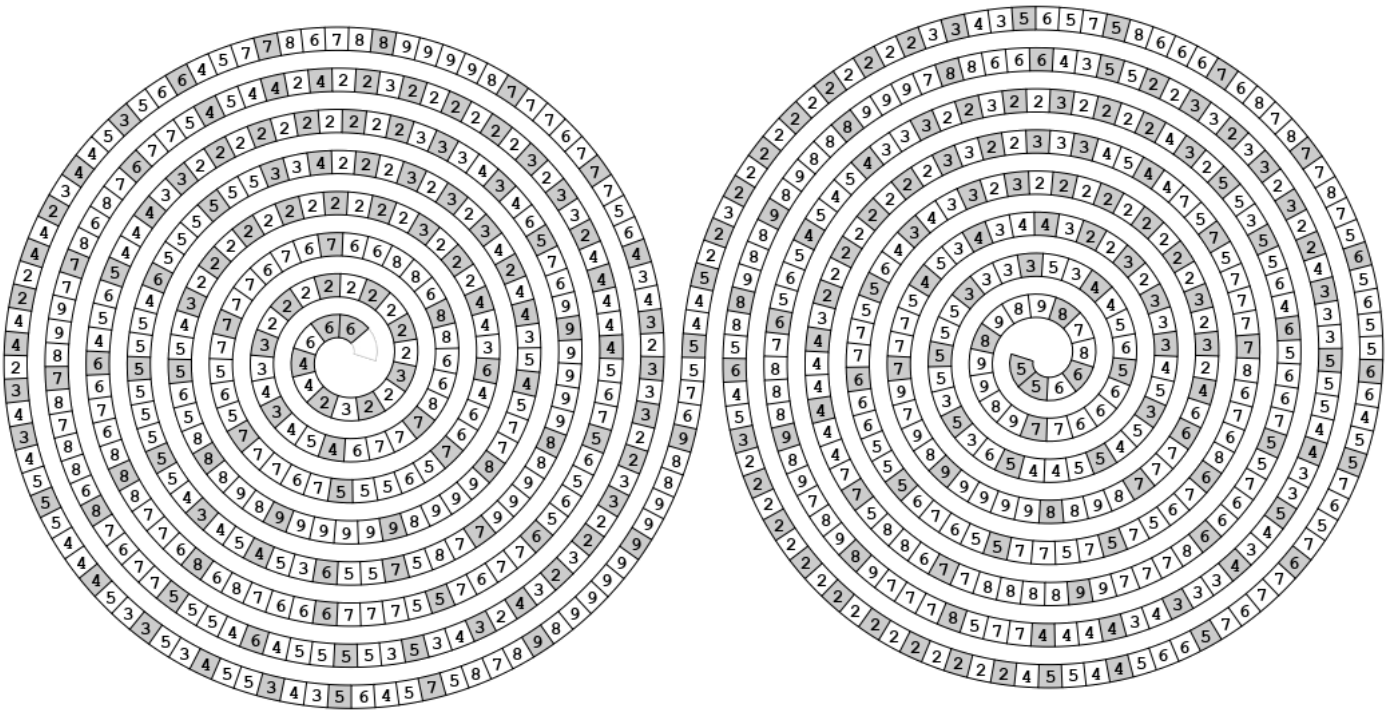
universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática



Speed run

Algoritmos e Estruturas de Dados

108122 – Alexandre Pedro Ribeiro – 50%

110056 – Ricardo Manuel Quintaneiro Almeida – 50%

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

dez 8, 2022

Índice

Introdução

O objetivo deste trabalho consiste em determinar o número mínimo de ações que um carro tem que realizar para percorrer uma estrada.

No contexto deste trabalho, uma estrada encontra-se dividida em segmentos de igual comprimento, cada um com um limite de velocidade. A velocidade corresponde ao número de segmentos que um carro consegue avançar numa única ação. Em cada ação, o carro pode reduzir, manter ou aumentar a sua velocidade, desde que avance e não transgrida os limites de velocidade dos segmentos por onde passa. O carro começa no primeiro segmento com velocidade 0, e tem que chegar ao fim com velocidade 1 (onde irá desacelerar e parar).

Ao longo deste relatório estão descritos os métodos que utilizámos no desenvolvimento das soluções a este problema, acompanhados de estatísticas de demonstram a sua eficácia.

Métodos

No começo deste trabalho, tornou-se claro que havia duas formas de resolver este problema:

- ❖ Recursiva – Cada iteração corresponde a uma ação feita pelo carro. A solução dada no enunciado recorre a este método para testar todos os caminhos possíveis.
- ❖ Não-recursiva – Evita-se o uso de chamadas recursivas à função. Pode-se fazer através de armazenamento e manipulação de dados, ou de outro método cíclico.

Após a testagem de vários conceitos e aplicações diferentes, acabámos com estes 6 métodos:

- *Very Bad Recursion*
- *Slightly Better Recursion*
- *Fast Recursion*
- *Fast Dynamic Non-Recursion*
- *Unitary Cost Dijkstra Recursion*
- *Speed-Based A* Recursion*

Alguns destes utilizam o mesmo raciocínio na procura de solução, mas todos têm algo que os distingue.

Very Bad Recursion

Raciocínio

Este método é dado no enunciado como uma solução bastante ineficaz do problema.

Baseia-se no uso da recursão para testar todos os caminhos possíveis, guardando o mais curto numa estrutura. Quando encontra um caminho que não consegue seguir, seja por violar limites de velocidade ou por falta de estrada, o programa volta atrás um passo e testa uma velocidade diferente. Se encontrar uma nova solução, compara com a que tem armazenada e guarda a mais rápida.

O programa acaba quando todos os caminhos forem testados.

Vantagens e Desvantagens

Testar todos os caminhos possíveis garante que o resultado é, de facto, o caminho mais curto, porém, isto não resulta tão bem para estradas com muitos segmentos.

Cada vez que o número de segmentos aumenta, o número de caminhos sobe exponencialmente e torna-se mais demorado encontrar uma solução.

Outra particularidade é que existem momentos em que a velocidade do carro é demasiado alta para poder desacelerar e acabar no último segmento, e este método não aproveita a oportunidade para descartar esses caminhos.

Resultados

		very bad recursion	
n	sol	count	cpu time
1	1	2	1.871e-06
2	2	3	9.460e-07
3	3	5	1.316e-06
4	3	8	1.362e-06
5	4	13	1.491e-06
6	4	22	1.728e-06
7	5	36	2.043e-06
8	5	60	2.659e-06
9	5	100	3.807e-06
10	6	167	5.235e-06
11	6	279	4.173e-06
12	6	465	4.637e-06
13	7	777	7.088e-06
14	7	1297	1.108e-05
15	7	2165	1.852e-05
16	7	3614	3.080e-05
17	8	6031	4.888e-05
18	8	10065	8.039e-05
19	8	16795	1.339e-04
20	8	28024	2.276e-04
21	9	46758	3.779e-04
22	9	78011	6.532e-04
23	9	130089	1.042e-03
24	9	216968	1.726e-03
25	9	359706	2.933e-03
26	10	597823	4.901e-03
27	10	995046	7.928e-03
28	10	1655498	1.265e-02
29	10	2757259	2.119e-02
30	10	4593012	3.575e-02
31	11	7651017	5.598e-02
32	11	12747967	7.909e-02
33	11	21239691	1.245e-01
34	12	35390165	1.962e-01
35	12	58969547	3.232e-01
36	12	98258424	5.294e-01
37	13	163727428	8.602e-01
38	13	272817267	1.421e+00
39	13	454593881	2.361e+00
40	14	757489987	3.900e+00
41	14	1262204160	6.566e+00
42	14	2114047092	1.089e+01
43	15	3533472456	1.813e+01
44	15	5898663878	3.167e+01
45	15	9850621736	5.256e+01
46	16	16352251531	8.453e+01
47	16	27196767437	1.397e+02
48	16	45288671397	2.334e+02
49	16	75373390362	3.872e+02
50	17	125557790155	6.521e+02
51	17	209172462741	1.116e+03
52	17	334621156464	1.756e+03
53	18	543684522773	2.847e+03
54	18	878196582805	4.554e+03

*Figura 1 - Feito **sem número mecanográfico***

```
Very bad recursion
road size: 50
```

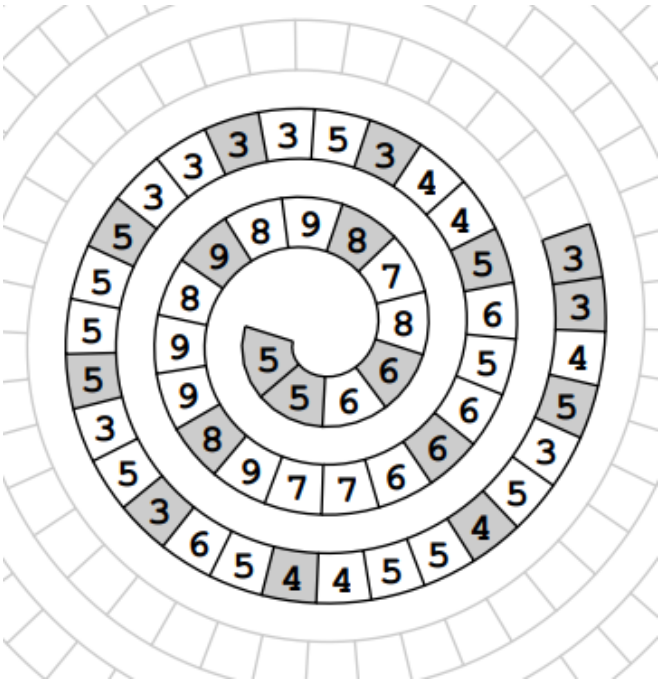


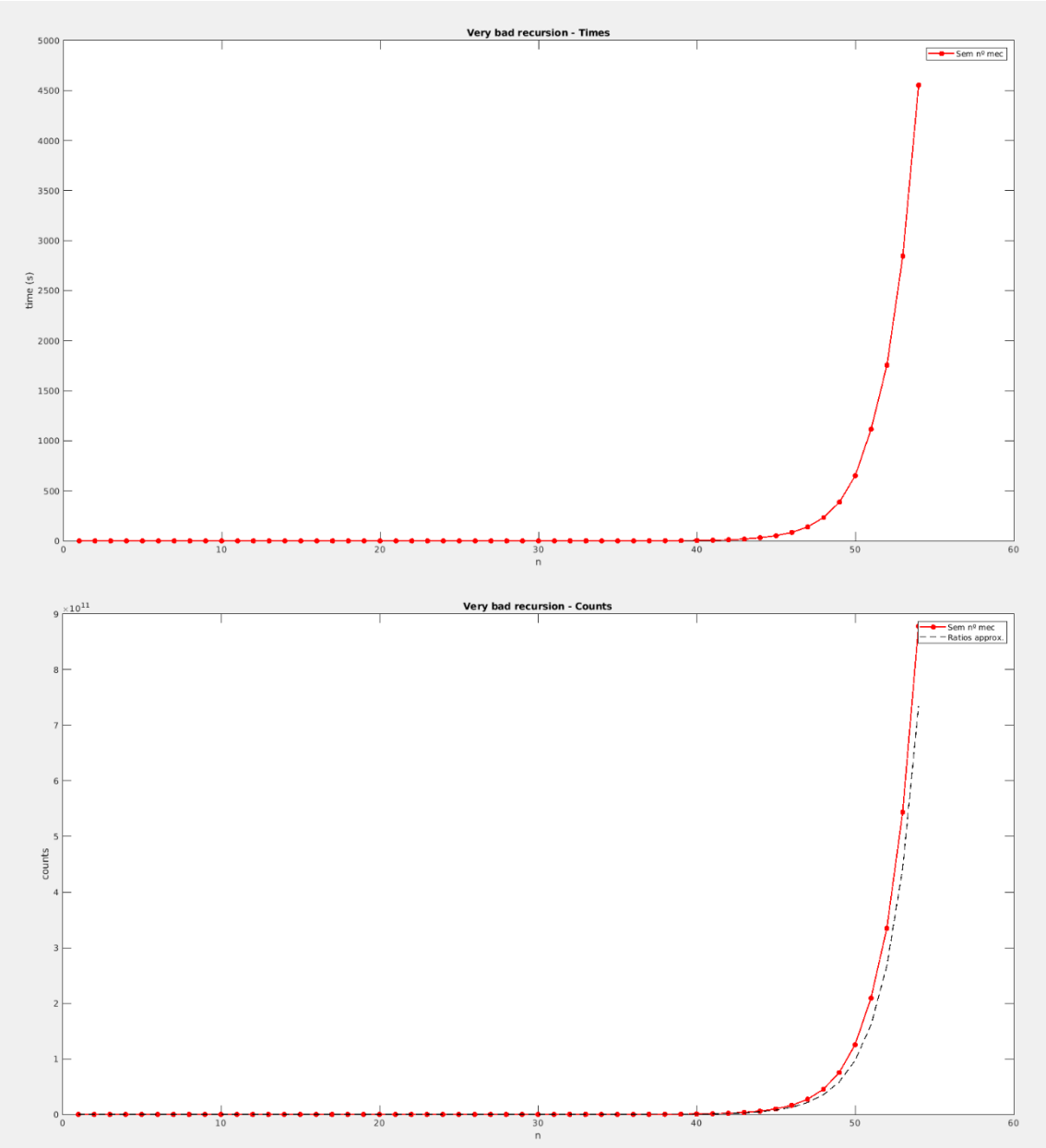
*Figura 2 - Feito **sem número mecanográfico***

```
17 moves
6.016e+02 seconds
effort: 125557790155
```

Análise de dados

Como era esperado, o tempo de execução (**cpu time**) e o número de chamadas à função (**count**) evoluem de forma exponencial à medida que o tamanho da estrada (**n**) aumenta. Para **n** maior do que 54, o programa começa a demorar mais do que 1 hora para determinar a solução.

Com a ajuda do MatLab, conseguimos mostrar estes dados num gráfico, para percebermos melhor esta evolução:



A aproximação da função dos **counts** é **O(n)** = 1,667^**n**.

Slightly Better Recursion

Raciocínio

Este método é uma tentativa de melhorar o anterior, mantendo as ideias fundamentais.

Podemos pensar no problema como uma árvore recursiva, em que cada movimento gera 3 novos ramos, cada um representando uma velocidade válida diferente. O objetivo deste algoritmo é tentar podar alguns desses ramos, pelo menos aqueles que temos a certeza de que não darão origem a uma solução.

Para atingir esse objetivo, utilizamos as seguintes ideias:

- Se o número de movimentos de um certo caminho é maior do que o valor correspondente da melhor solução guardada, esse ramo é cortado.
- Se a velocidade atual não permite desacelerar o suficiente para acabar no último segmento com velocidade 1, esse ramo é cortado.
- A velocidade que começa por testar é speed+1 (prioridade em acelerar).

Vantagens e Desvantagens

Tal como no primeiro, este método permite verificar todos os caminhos possíveis, dando a garantia de uma boa solução, com a vantagem adicional de ser mais rápido.

No entanto, também herda o mesmo problema, em que para números elevados de segmentos de estrada o programa começa a demorar muito tempo.

Resultados

	Slightly better recursion	
n	sol	count cpu time
1	1	2 5.920e-07
2	2	3 2.650e-07
3	3	4 2.600e-07
4	3	5 3.640e-07
5	4	8 3.740e-07
6	4	9 3.790e-07
7	5	16 4.730e-07
8	5	17 4.590e-07
9	5	19 5.050e-07
10	6	38 7.590e-07
11	6	42 1.631e-06
12	6	46 1.065e-06
13	7	98 1.639e-06
14	7	107 1.749e-06
15	7	113 1.754e-06
16	7	120 1.837e-06
17	8	275 3.581e-06
18	8	291 3.758e-06
19	8	308 4.004e-06
20	8	323 4.274e-06
21	9	765 1.091e-05
22	9	807 1.027e-05
23	9	846 1.048e-05
24	9	881 1.108e-05
25	9	913 1.618e-05
26	10	2238 2.615e-05
27	10	2332 2.729e-05
28	10	2418 2.833e-05
29	10	2497 2.917e-05
30	10	2574 2.958e-05
31	11	6451 7.287e-05
32	11	6482 7.561e-05
33	11	6504 7.778e-05
34	12	15728 1.871e-04
35	12	15781 1.640e-04
36	12	15816 1.554e-04
37	13	37964 4.044e-04
38	13	38053 4.437e-04
39	13	38104 4.495e-04
40	14	91232 9.471e-04
41	14	91373 1.039e-03
42	14	91444 1.044e-03
43	15	219127 2.459e-03
44	15	219340 2.479e-03
45	15	219436 2.437e-03
46	16	527041 5.198e-03
47	16	527351 5.715e-03
48	16	528661 5.993e-03
49	16	529455 5.666e-03
50	17	1280874 1.325e-02
55	19	7492103 7.947e-02
60	21	42353906 3.858e-01
65	23	223357998 1.934e+00
70	25	1086070749 9.238e+00
75	26	2330972599 1.930e+01
80	27	4934602616 4.037e+01
85	28	10351839542 8.587e+01
90	29	21646248278 1.800e+02

*Figura 3 – Número mecanográfico **108122***

	Slightly better recursion	
n	sol	count cpu time
1	1	2 1.210e-06
2	2	3 5.500e-07
3	3	4 5.820e-07
4	3	5 6.910e-07
5	4	8 7.320e-07
6	4	9 6.920e-07
7	5	16 9.150e-07
8	5	17 9.290e-07
9	5	19 1.049e-06
10	6	38 1.474e-06
11	6	42 3.198e-06
12	6	46 2.155e-06
13	7	98 3.585e-06
14	7	107 3.578e-06
15	7	113 3.683e-06
16	7	120 4.209e-06
17	8	275 7.873e-06
18	8	291 8.414e-06
19	8	308 8.972e-06
20	8	323 9.303e-06
21	9	765 2.072e-05
22	9	807 1.855e-05
23	9	846 1.957e-05
24	9	881 2.090e-05
25	9	913 2.115e-05
26	10	2238 5.029e-05
27	10	2332 5.120e-05
28	10	2418 5.490e-05
29	11	5774 1.322e-04
30	11	5848 1.378e-04
31	11	5910 1.375e-04
32	12	14030 3.098e-04
33	12	14134 3.085e-04
34	12	14191 3.125e-04
35	13	33681 7.294e-04
36	13	33845 7.363e-04
37	13	33926 7.319e-04
38	14	80989 1.704e-03
39	14	81237 1.743e-03
40	14	81354 1.771e-03
41	15	195140 3.877e-03
42	15	195508 3.600e-03
43	15	195676 3.617e-03
44	16	470822 6.018e-03
45	16	471361 5.286e-03
46	16	473292 5.304e-03
47	16	474464 4.326e-03
48	17	1155391 1.164e-02
49	17	1158838 1.168e-02
50	17	1162552 1.174e-02
55	19	7078431 7.020e-02
60	22	100598776 8.399e-01
65	23	234395976 1.909e+00
70	25	1182108305 9.258e+00
75	26	2566871418 2.005e+01
80	27	5483541065 4.155e+01
85	28	11616284400 8.696e+01
90	29	24655937744 1.839e+02

*Figura 4 - Número mecanográfico **110056***

```
Slightly better recursion
road size: 90
```

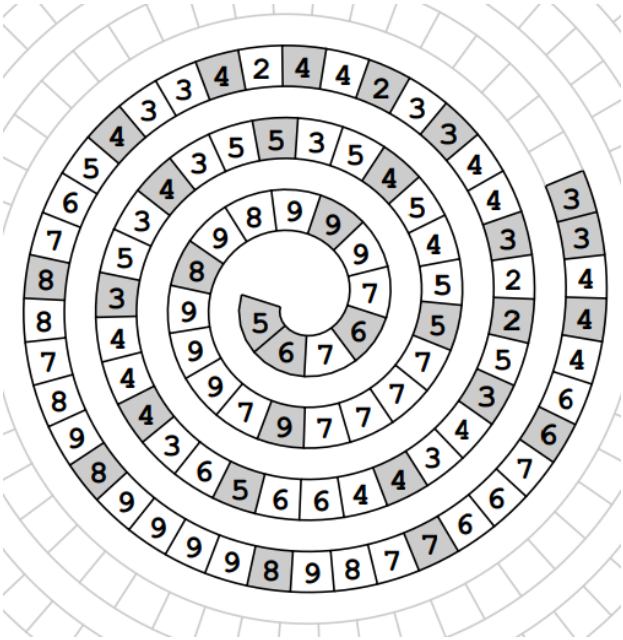


*Figura 5 - Número mecanográfico **108122***

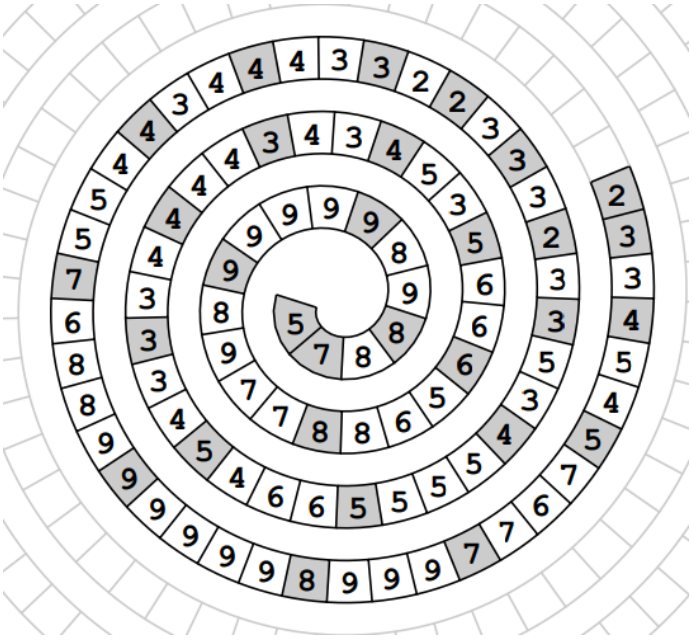


*Figura 6 - Número mecanográfico **110056***

```
29 moves
1.800e+02 seconds
effort: 21646248278
```

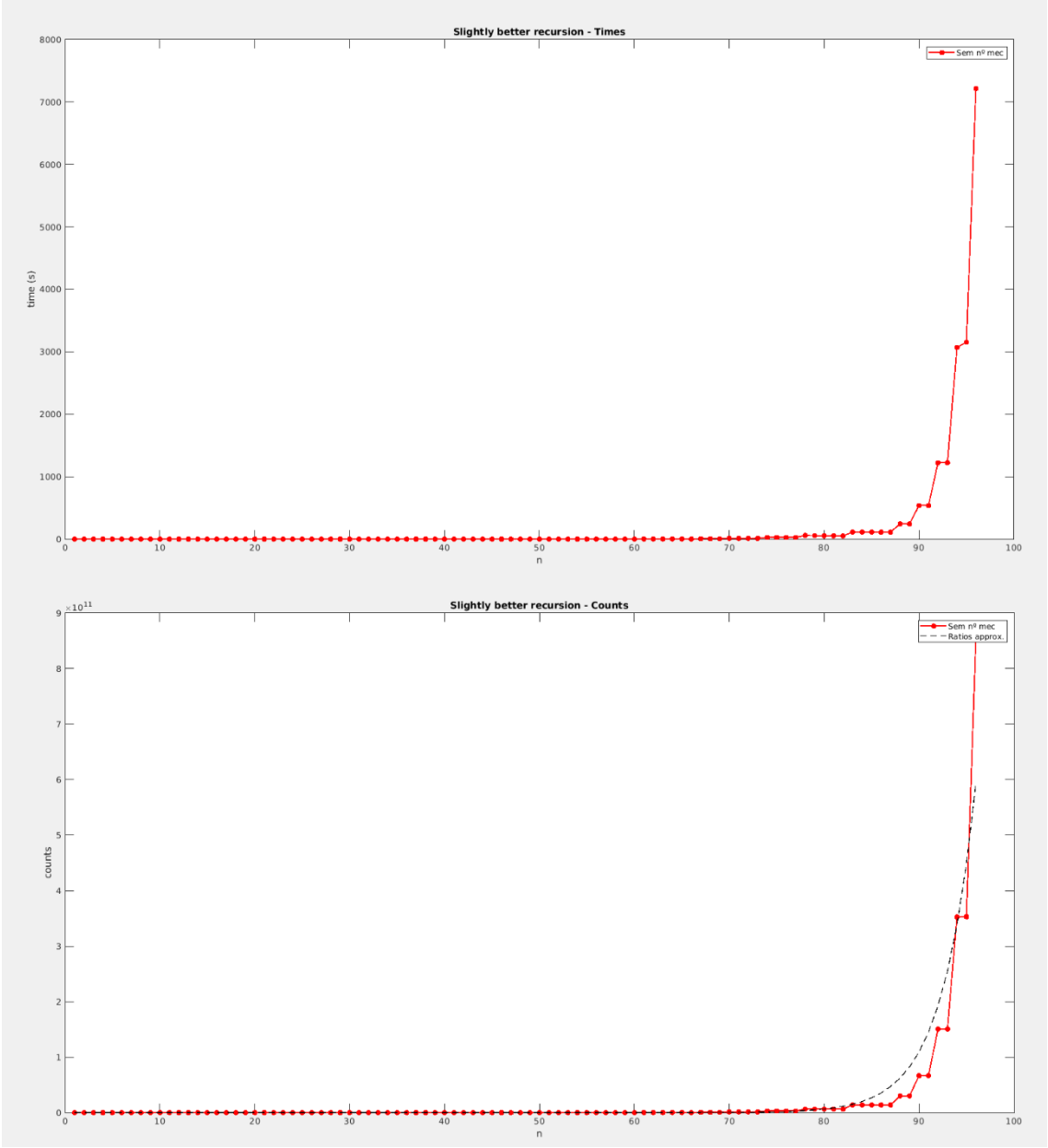
```
29 moves
1.839e+02 seconds
effort: 24655937744
```

Análise de dados

Como podemos observar, os tempos de execução são relativamente mais rápidos, tornando possível a resolução de problemas com **n** mais elevado sem ter que esperar mais de uma hora.

Há uma particularidade, no entanto, com este algoritmo. Se olharmos para os tempos e **counts**, e associarmos às **sols** (número de movimentos do caminho mais rápido) correspondentes, observamos um padrão interessante: para números de **n** diferentes, se o número de **sol** é igual, o tempo de execução e o número de operações são bastante semelhantes.

Para ajudar a visualizar este padrão, podemos recorrer aos seguintes gráficos:



Mas então, qual é a razão para a existência destes “degraus”?

A resposta encontra-se numa das ideias que utilizamos para melhorar o programa: “se o número de movimentos de um certo caminho é maior do que o valor correspondente da melhor solução guardada, esse ramo é cortado”.

O programa, quando encontra e armazena a solução mais rápida, ainda continua a testar outros caminhos, porém, uma grande porção desses caminhos são cortados porque chegam a uma altura em que o número de movimentos que realizaram é maior do que o número de movimentos do caminho armazenado.

No entanto, quando o nº de movimentos da solução aumenta, esse corte é adiado, e o programa permite que esses caminhos inválidos “cresçam” mais até serem inevitavelmente descartados, justificando o aumento em “degraus” dos tempos e dos **counts**.

A aproximação da função dos **counts** é **O(n)** = 1,326^**n**.

Fast Recursion

Raciocínio

Este método tenta encontrar o caminho mais rápido sem testar todos os caminhos possíveis, utilizando um algoritmo que prioriza a aceleração do carro.

O algoritmo segue o mesmo princípio que os anteriores: começa na primeira posição, testa uma velocidade legal e avança para a próxima posição. No entanto, a primeira velocidade testada é sempre a maior possível.

O programa acaba quando encontrar o primeiro caminho possível, que assume ser o mais rápido.

Vantagens e Desvantagens

Ao contrário dos anteriores, o programa não testa todos os caminhos possíveis, o que é tanto uma vantagem como uma desvantagem.

Teoricamente, como para no primeiro caminho válido que encontrar, deve ser mais rápido, mas isso também significa que não temos garantia de que a solução está correta, a não ser que comparemos com os resultados dos outros programas.

Resultados

Fast recursion							
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	1.123e-06	55	19	20	9.410e-07
2	2	3	7.730e-07	60	21	22	7.210e-07
3	3	4	7.890e-07	65	23	24	7.740e-07
4	3	4	8.300e-07	70	25	26	8.180e-07
5	4	5	8.500e-07	75	26	27	7.000e-07
6	4	5	8.500e-07	80	27	28	7.610e-07
7	5	6	8.920e-07	85	28	29	8.400e-07
8	5	6	8.570e-07	90	29	30	8.320e-07
9	5	6	8.830e-07	95	31	32	9.270e-07
10	6	7	8.390e-07	100	33	35	1.147e-06
11	6	7	1.669e-06	110	38	40	1.828e-06
12	6	7	8.500e-07	120	41	43	1.511e-06
13	7	8	7.900e-07	130	43	45	1.377e-06
14	7	8	8.250e-07	140	45	47	1.492e-06
15	7	8	8.940e-07	150	47	49	1.391e-06
16	7	8	8.640e-07	160	50	54	1.705e-06
17	8	9	8.540e-07	170	55	59	1.970e-06
18	8	9	8.410e-07	180	58	62	1.764e-06
19	8	9	8.510e-07	190	60	64	1.774e-06
20	8	9	8.790e-07	200	62	66	1.793e-06
21	9	10	6.280e-07	220	68	72	2.622e-06
22	9	10	4.670e-07	240	76	80	2.446e-06
23	9	10	4.640e-07	260	80	84	3.917e-06
24	9	10	4.250e-07	280	87	94	2.598e-06
25	9	10	4.500e-07	300	95	102	2.462e-06
26	10	11	4.170e-07	320	100	108	2.962e-06
27	10	11	4.280e-07	340	109	121	3.001e-06
28	10	11	4.160e-07	360	113	125	3.031e-06
29	10	11	4.150e-07	380	118	131	3.148e-06
30	10	11	4.520e-07	400	126	140	3.390e-06
31	11	12	4.320e-07	420	132	146	5.505e-06
32	11	12	4.280e-07	440	136	150	3.418e-06
33	11	12	4.250e-07	460	145	159	3.854e-06
34	12	13	4.760e-07	480	150	164	3.362e-06
35	12	13	4.930e-07	500	156	170	3.857e-06
36	12	13	5.640e-07	520	164	178	4.040e-06
37	13	14	4.900e-07	540	169	183	3.700e-06
38	13	14	4.810e-07	560	176	193	3.992e-06
39	13	14	5.560e-07	580	183	200	5.220e-06
40	14	15	5.490e-07	600	187	204	4.195e-06
41	14	15	5.530e-07	620	193	212	4.081e-06
42	14	15	5.020e-07	640	201	220	4.417e-06
43	15	16	4.630e-07	660	206	228	4.409e-06
44	15	16	4.920e-07	680	212	234	4.514e-06
45	15	16	5.270e-07	700	221	243	4.508e-06
46	16	17	5.440e-07	720	226	248	4.785e-06
47	16	17	5.590e-07	740	235	261	5.982e-06
48	16	17	5.610e-07	760	240	266	4.889e-06
49	16	17	6.090e-07	780	243	269	5.050e-06
50	17	18	5.720e-07	800	253	280	5.143e-06

*Figura 7 - Número mecanográfico **108122***

	Fast recursion						
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	1.157e-06	55	19	21	2.065e-06
2	2	3	4.870e-07	60	22	24	1.129e-06
3	3	4	4.660e-07	65	23	25	9.070e-07
4	3	4	5.000e-07	70	25	27	8.710e-07
5	4	5	4.970e-07	75	26	28	9.280e-07
6	4	5	4.880e-07	80	27	29	9.370e-07
7	5	6	4.840e-07	85	28	30	9.540e-07
8	5	6	5.180e-07	90	29	31	9.630e-07
9	5	6	5.320e-07	95	31	33	1.126e-06
10	6	7	5.110e-07	100	33	36	1.407e-06
11	6	7	2.243e-06	110	38	41	2.328e-06
12	6	7	8.970e-07	120	41	44	1.473e-06
13	7	8	8.610e-07	130	43	46	1.328e-06
14	7	8	8.940e-07	140	45	48	1.327e-06
15	7	8	8.880e-07	150	47	50	1.487e-06
16	7	8	9.610e-07	160	50	54	1.611e-06
17	8	9	9.090e-07	170	55	59	1.886e-06
18	8	9	9.030e-07	180	58	62	1.712e-06
19	8	9	8.960e-07	190	61	65	1.742e-06
20	8	9	9.820e-07	200	62	66	1.671e-06
21	9	10	1.344e-06	220	69	73	3.953e-06
22	9	10	7.890e-07	240	76	80	2.707e-06
23	9	10	8.570e-07	260	80	84	2.306e-06
24	9	10	6.770e-07	280	87	93	2.874e-06
25	9	10	9.330e-07	300	95	101	3.151e-06
26	10	11	7.640e-07	320	99	105	2.916e-06
27	10	11	7.970e-07	340	108	115	5.773e-06
28	10	11	8.600e-07	360	112	119	3.074e-06
29	11	12	9.990e-07	380	117	124	3.228e-06
30	11	13	1.085e-06	400	126	141	3.829e-06
31	11	13	8.100e-07	420	131	146	3.169e-06
32	12	14	1.103e-06	440	135	150	2.546e-06
33	12	14	9.080e-07	460	143	158	2.512e-06
34	12	14	1.125e-06	480	149	164	2.437e-06
35	13	15	1.016e-06	500	155	171	3.959e-06
36	13	15	9.540e-07	520	163	179	2.602e-06
37	13	15	7.610e-07	540	168	184	2.685e-06
38	14	16	9.210e-07	560	174	190	2.986e-06
39	14	16	1.051e-06	580	181	197	2.992e-06
40	14	16	8.500e-07	600	185	201	3.004e-06
41	15	17	1.269e-06	620	191	207	3.168e-06
42	15	17	1.106e-06	640	198	214	3.335e-06
43	15	17	9.330e-07	660	202	218	3.175e-06
44	16	18	9.710e-07	680	209	225	4.114e-06
45	16	18	7.890e-07	700	216	232	3.450e-06
46	16	18	1.154e-06	720	221	240	3.716e-06
47	16	18	8.630e-07	740	230	254	3.667e-06
48	17	19	1.132e-06	760	235	259	3.887e-06
49	17	19	1.078e-06	780	239	263	3.865e-06
50	17	19	1.119e-06	800	248	275	4.366e-06

*Figura 8 - Número mecanográfico **110056***

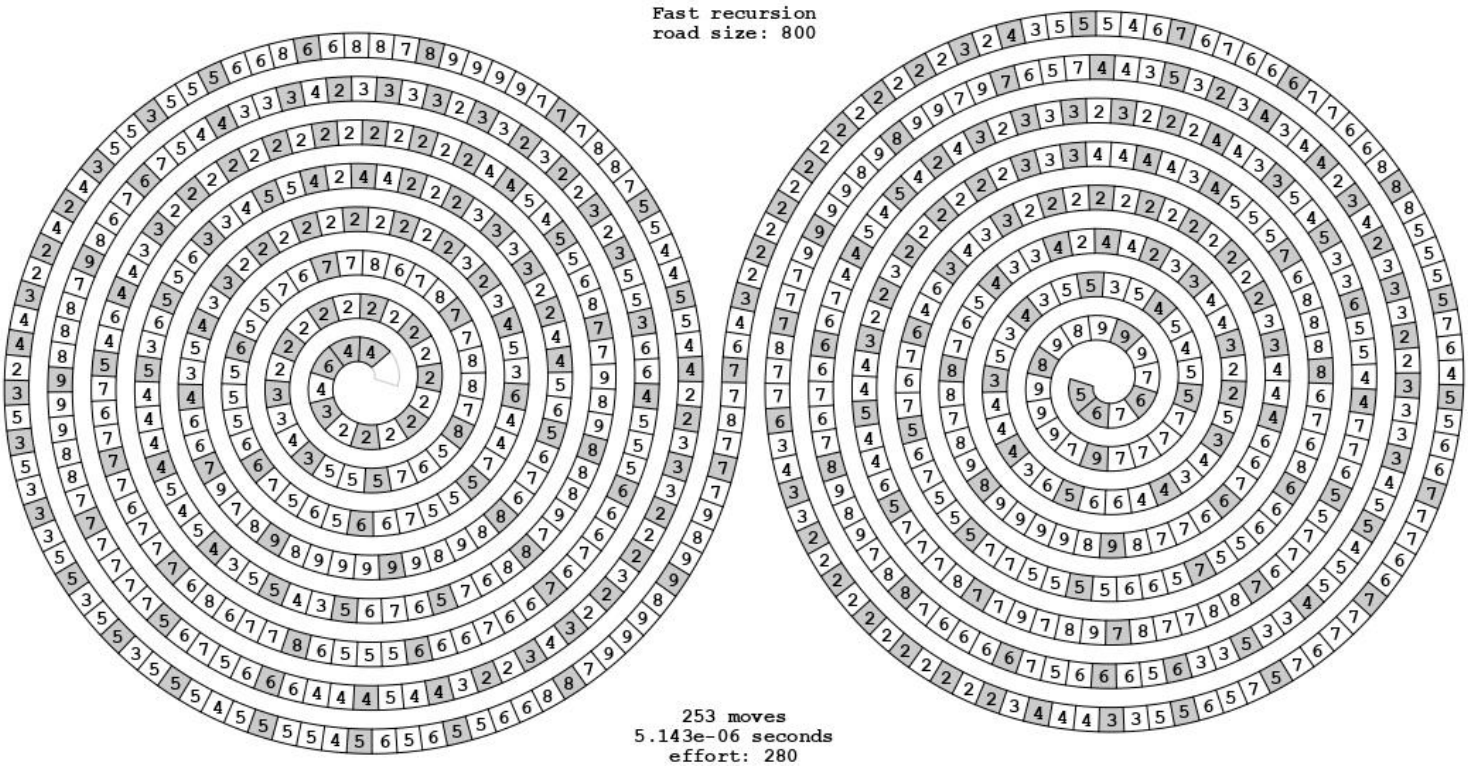


*Figura 9 - Número mecanográfico **108122***

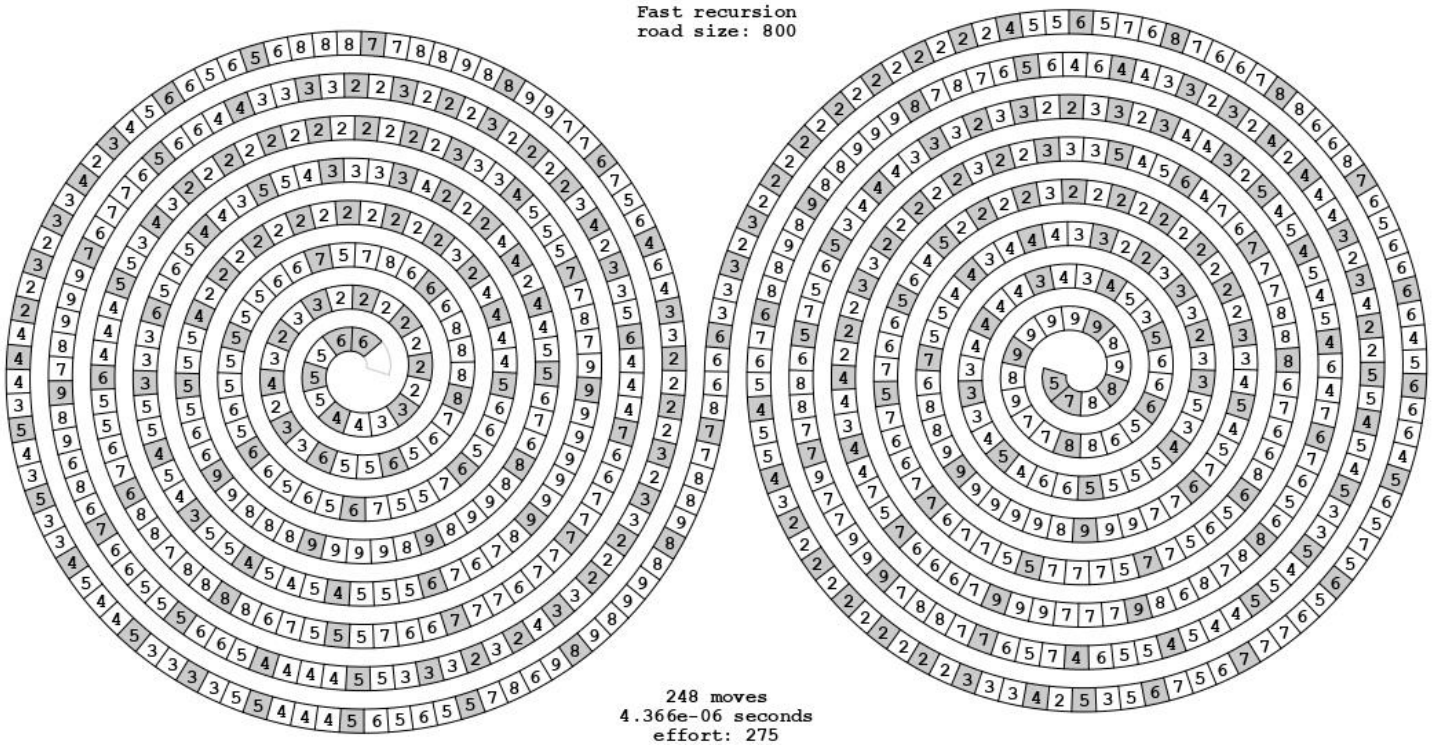


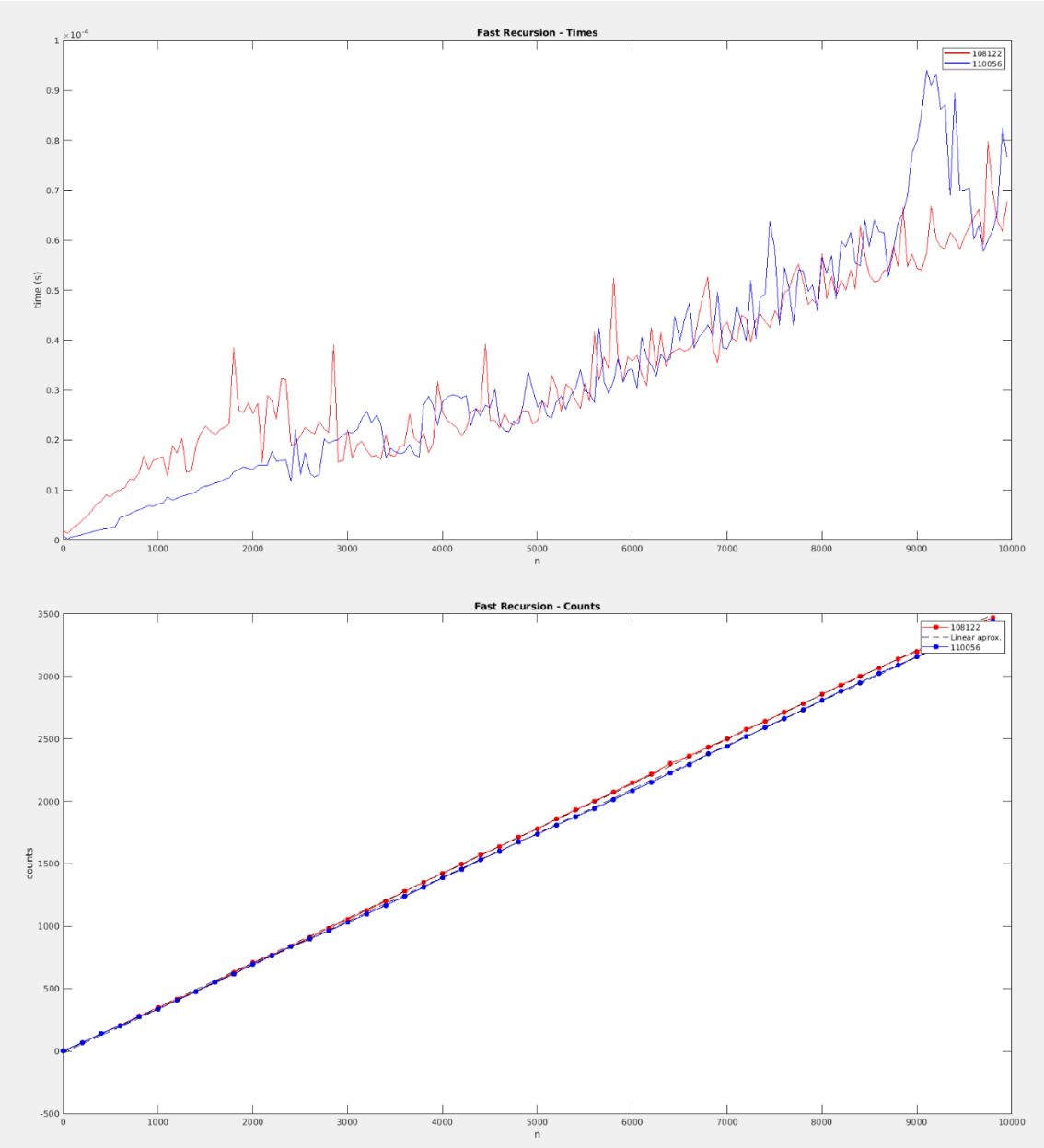
*Figura 10 - Número mecanográfico **110056***

Análise de dados

Como podemos observar, os tempos de execução são extremamente rápidos, demorando apenas alguns microssegundos para resolver o problema com **n** igual a 800.

Se olharmos para cada **n** e o seu respetivo **count**, notamos que o número de operações parece variar linearmente com o tamanho do problema, sendo essa a razão para o pequeno tamanho dos tempos.

Podemos confirmar essa observação com os seguintes gráficos:



Verifica-se que, apesar da função correspondente aos **counts** ser claramente linear, os tempos parecem variar bastante mais do que previsto. A justificação para este comportamento é que o tempo é tão pequeno que qualquer minúscula interrupção ou distração do processador fazem-se notar no gráfico desenhado.

A aproximação da função dos **counts** difere ligeiramente entre os dois números mecanográficos, e são:

- Nº mec. 108122 - **O(n)** = 0.3576***n** – 8.6093
- Nº mec. 110056 - **O(n)** = 0.3523***n** – 17.0746

Contradição

No momento da criação deste algoritmo surgiu logo a questão: "Mas então e se houver uma situação em para obtermos o caminho mais curto temos que manter/abrandar a velocidade numa posição em que podemos acelerar?".

Essa contradição foi algo que nos preocupou bastante, até acabarmos de escrever o programa e observarmos que os resultados estavam de acordo com os do professor. Porém, essa questão manteve-se ao longo do resto do trabalho.

Por isso, decidimos tentar encontrar um caso em que este método não funciona.

Fast Dynamic Non-Recursion

Raciocínio

Este método aplica os conceitos aprendidos sobre *dynamic programming* ao método anterior, criando um programa que chega à mesma solução, mas não utiliza recursão.

Para isso, encapsulamos o código criado no terceiro método num loop **for**, considerando cada iteração como um novo movimento. As chamadas recursivas da função e os respetivos **returns** são substituídos por outras formas de manipulação de dados, nomeadamente armazenamento de informação em **arrays** e uso de uma variável de indexação que representa cada movimento.

O algoritmo aplicado é, na sua essência, o mesmo que foi utilizado para criar o terceiro método, por isso podemos prever que terá resultados semelhantes.

Vantagens e Desvantagens

Tal como no anterior, o programa vai encontrar uma solução quase de imediato, no entanto, os tempos de execução deverão ser ligeiramente maiores, porque as velocidades e as posições estão armazenadas em **arrays**, tornando o seu acesso mais demorado.

Resultados

Fast Dynamic Non-Recursion							
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	9.200e-07	55	19	20	1.048e-06
2	2	3	5.140e-07	60	21	22	7.960e-07
3	3	4	5.230e-07	65	23	24	8.980e-07
4	3	4	5.180e-07	70	25	26	8.190e-07
5	4	5	4.830e-07	75	26	27	8.440e-07
6	4	5	5.020e-07	80	27	28	8.420e-07
7	5	6	5.130e-07	85	28	29	1.032e-06
8	5	6	5.150e-07	90	29	30	1.148e-06
9	5	6	4.850e-07	95	31	32	9.610e-07
10	6	7	5.220e-07	100	33	35	1.214e-06
11	6	7	1.009e-06	110	38	40	1.916e-06
12	6	7	4.360e-07	120	41	43	1.425e-06
13	7	8	3.730e-07	130	43	45	1.264e-06
14	7	8	4.690e-07	140	45	47	1.261e-06
15	7	8	3.820e-07	150	47	49	1.265e-06
16	7	8	4.680e-07	160	50	54	1.566e-06
17	8	9	4.170e-07	170	55	59	1.768e-06
18	8	9	4.210e-07	180	58	62	1.944e-06
19	8	9	3.720e-07	190	60	64	1.973e-06
20	8	9	3.680e-07	200	62	66	1.898e-06
21	9	10	6.180e-07	220	68	72	3.571e-06
22	9	10	3.870e-07	240	76	80	2.857e-06
23	9	10	3.680e-07	260	80	84	2.936e-06
24	9	10	4.430e-07	280	87	94	3.510e-06
25	9	10	4.000e-07	300	95	102	3.850e-06
26	10	11	3.920e-07	320	100	108	4.220e-06
27	10	11	4.170e-07	340	109	121	6.434e-06
28	10	11	4.550e-07	360	113	125	6.032e-06
29	10	11	4.040e-07	380	118	131	6.683e-06
30	10	11	4.050e-07	400	126	140	7.227e-06
31	11	12	5.440e-07	420	132	146	9.241e-06
32	11	12	4.240e-07	440	136	150	9.652e-06
33	11	12	5.460e-07	460	145	159	1.038e-05
34	12	13	4.400e-07	480	150	164	1.067e-05
35	12	13	5.780e-07	500	156	170	1.108e-05
36	12	13	5.690e-07	520	164	178	1.211e-05
37	13	14	5.310e-07	540	169	183	1.279e-05
38	13	14	6.230e-07	560	176	193	1.187e-05
39	13	14	5.810e-07	580	183	200	1.395e-05
40	14	15	5.370e-07	600	187	204	1.517e-05
41	14	15	5.330e-07	620	193	212	1.560e-05
42	14	15	5.060e-07	640	201	220	1.695e-05
43	15	16	6.710e-07	660	206	228	1.767e-05
44	15	16	5.590e-07	680	212	234	1.868e-05
45	15	16	5.350e-07	700	221	243	1.989e-05
46	16	17	5.370e-07	720	226	248	2.039e-05
47	16	17	6.320e-07	740	235	261	2.228e-05
48	16	17	5.480e-07	760	240	266	2.314e-05
49	16	17	5.890e-07	780	243	269	2.335e-05
50	17	18	6.460e-07	800	253	280	2.502e-05

*Figura 11 - Número mecanográfico **108122***

Fast Dynamic Non-Recursion							
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	1.414e-06	55	19	21	2.882e-06
2	2	3	7.940e-07	60	22	24	1.996e-06
3	3	4	8.420e-07	65	23	25	1.613e-06
4	3	4	7.400e-07	70	25	27	1.547e-06
5	4	5	7.200e-07	75	26	28	1.641e-06
6	4	5	7.110e-07	80	27	29	1.627e-06
7	5	6	7.340e-07	85	28	30	1.667e-06
8	5	6	8.140e-07	90	29	31	1.661e-06
9	5	6	8.490e-07	95	31	33	1.875e-06
10	6	7	7.830e-07	100	33	36	2.261e-06
11	6	7	1.966e-06	110	38	41	2.571e-06
12	6	7	1.019e-06	120	41	44	1.937e-06
13	7	8	8.230e-07	130	43	46	2.017e-06
14	7	8	8.960e-07	140	45	48	1.931e-06
15	7	8	7.910e-07	150	47	50	2.279e-06
16	7	8	8.330e-07	160	50	54	2.429e-06
17	8	9	9.010e-07	170	55	59	2.781e-06
18	8	9	1.006e-06	180	58	62	2.842e-06
19	8	9	8.050e-07	190	61	65	3.167e-06
20	8	9	8.150e-07	200	62	66	3.241e-06
21	9	10	1.360e-06	220	69	73	4.459e-06
22	9	10	8.030e-07	240	76	80	3.971e-06
23	9	10	7.360e-07	260	80	84	3.639e-06
24	9	10	9.920e-07	280	87	93	4.168e-06
25	9	10	7.740e-07	300	95	101	4.776e-06
26	10	11	7.660e-07	320	99	105	5.333e-06
27	10	11	8.070e-07	340	108	115	6.005e-06
28	10	11	8.070e-07	360	112	119	5.379e-06
29	11	12	8.250e-07	380	117	124	5.936e-06
30	11	13	1.136e-06	400	126	141	6.633e-06
31	11	13	8.880e-07	420	131	146	1.054e-05
32	12	14	8.750e-07	440	135	150	9.570e-06
33	12	14	9.650e-07	460	143	158	9.860e-06
34	12	14	9.470e-07	480	149	164	1.056e-05
35	13	15	9.710e-07	500	155	171	1.113e-05
36	13	15	1.193e-06	520	163	179	1.173e-05
37	13	15	9.660e-07	540	168	184	1.196e-05
38	14	16	9.640e-07	560	174	190	1.275e-05
39	14	16	1.075e-06	580	181	197	1.340e-05
40	14	16	8.460e-07	600	185	201	1.389e-05
41	15	17	1.328e-06	620	191	207	1.444e-05
42	15	17	1.029e-06	640	198	214	1.564e-05
43	15	17	8.750e-07	660	202	218	1.610e-05
44	16	18	1.175e-06	680	209	225	1.659e-05
45	16	18	1.018e-06	700	216	232	1.762e-05
46	16	18	1.195e-06	720	221	240	1.861e-05
47	16	18	1.014e-06	740	230	254	1.994e-05
48	17	19	1.145e-06	760	235	259	2.062e-05
49	17	19	1.083e-06	780	239	263	2.121e-05
50	17	19	9.930e-07	800	248	275	2.278e-05

*Figura 12 - Número mecanográfico **110056***

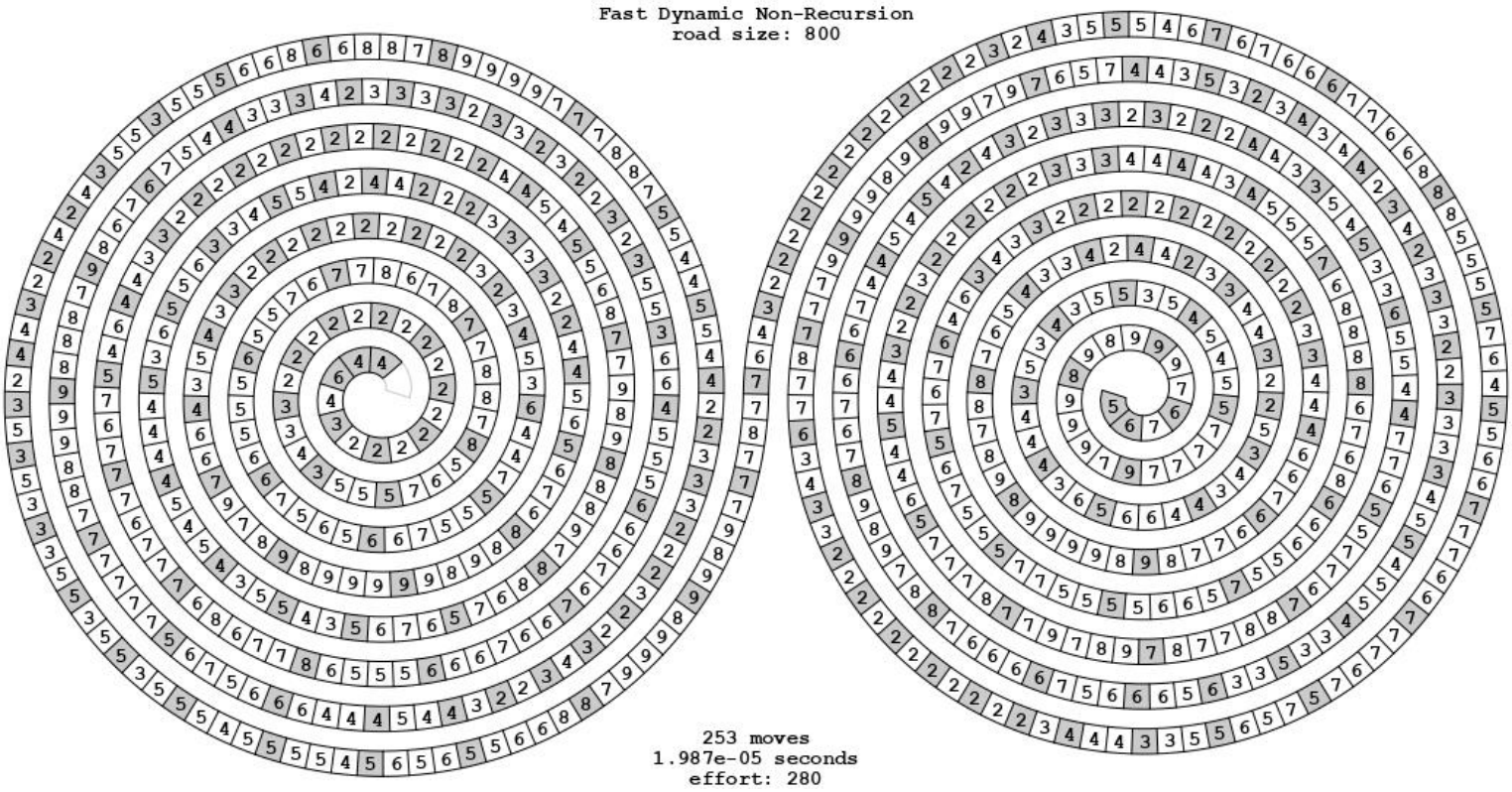


*Figura 13 - Número mecanográfico **108122***

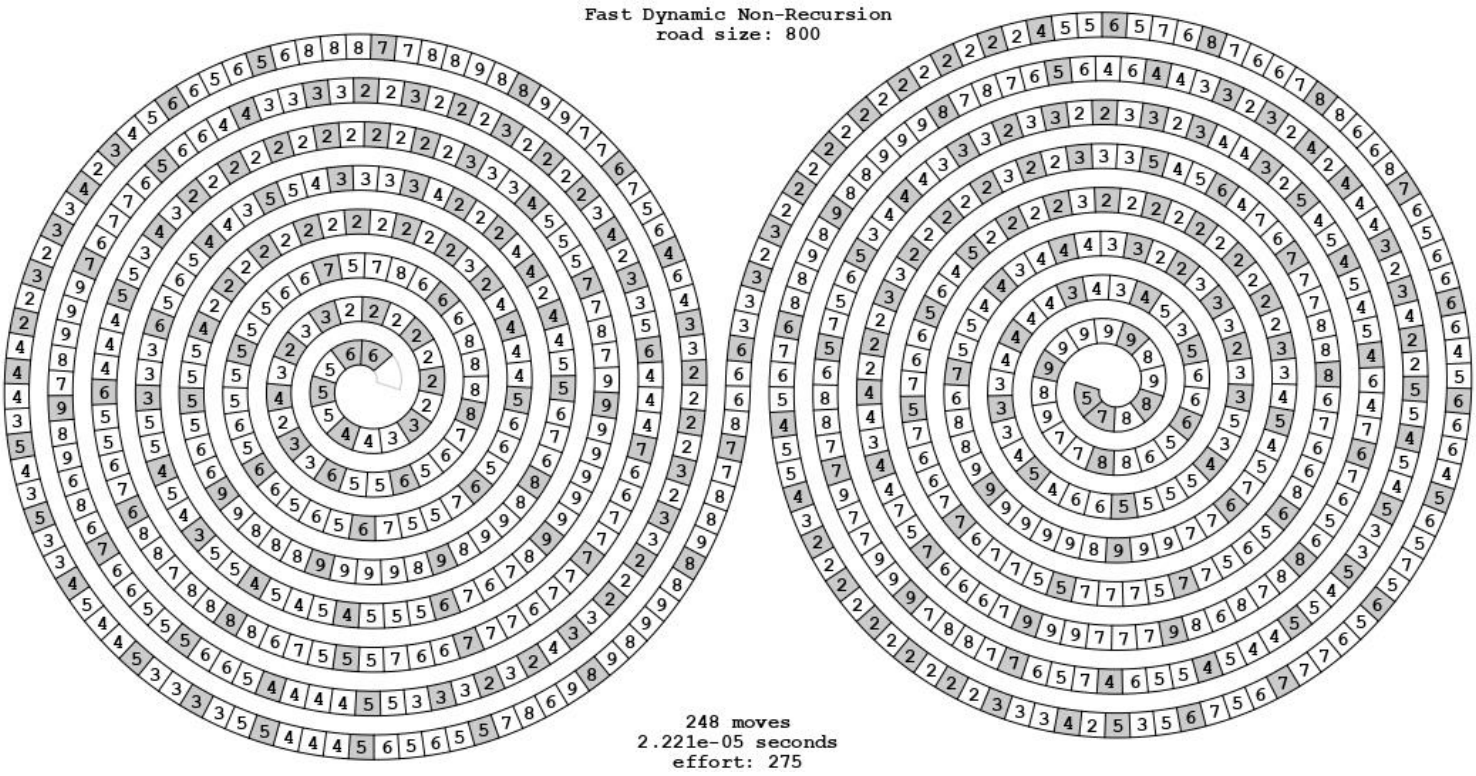
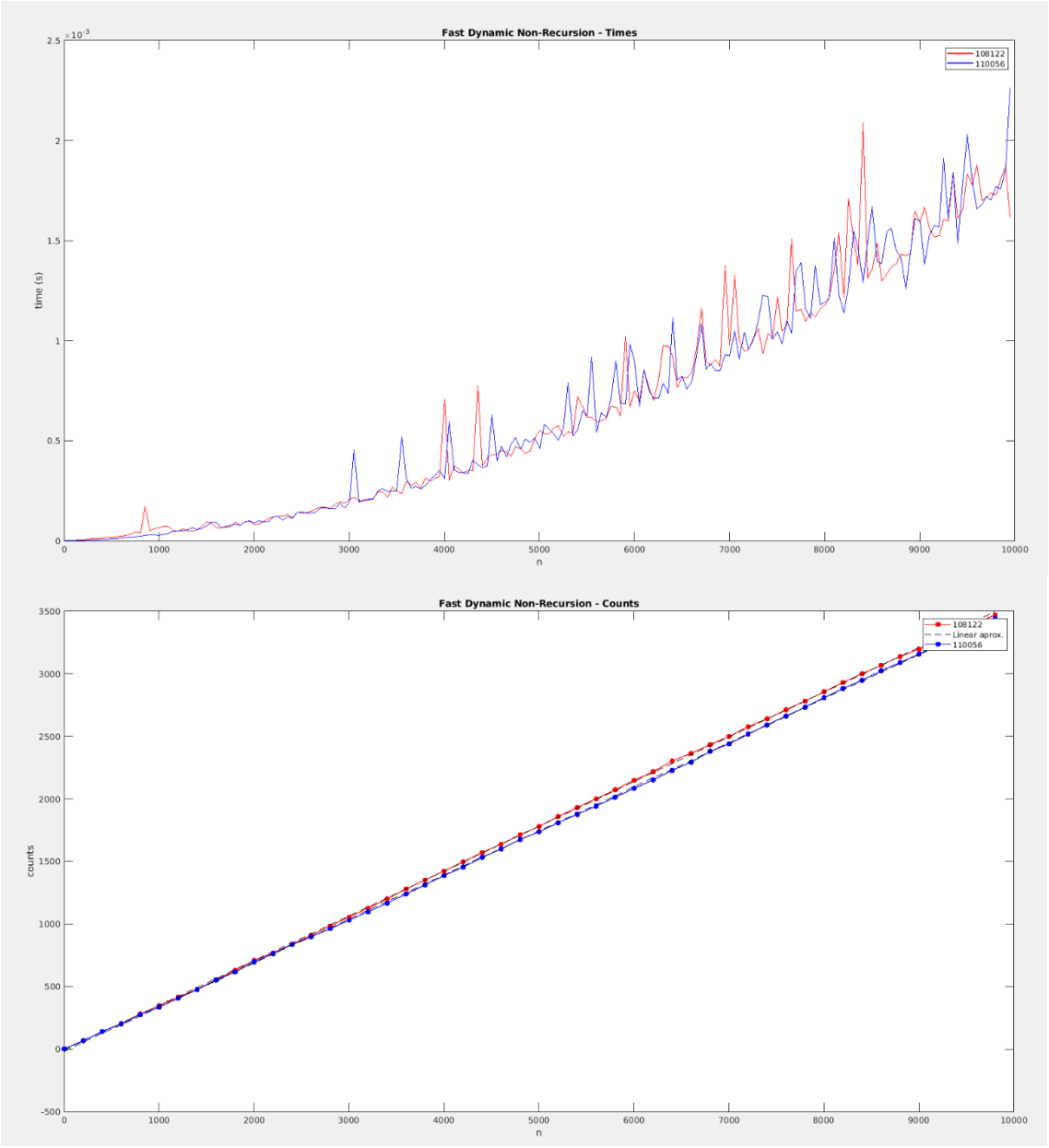


*Figura 14 - Número mecanográfico **110056***

Análise de dados

Como era previsto, o número de **counts** é igual ao método anterior, e o tempo de execução é ligeiramente mais alto, se bem que permanece em valores relativamente baixos.



As aproximações da função dos **counts** são exatamente iguais às obtidas no método anterior, que são:

- Nº mec. 108122 - **O(n)** = 0.3576***n** – 8.6093
- Nº mec. 110056 - **O(n)** = 0.3523***n** – 17.0746

Unitary-Cost Dijsktra Recursion

Raciocínio

Este método tenta adaptar o algoritmo de Dijkstra para encontrar o caminho mais curto, no entanto, o problema não torna fácil a sua implementação, uma vez que as restrições de velocidade não permitem que o grafo seja pré-definido.

Na nossa aplicação deste algoritmo, cada vértice do grafo representa uma posição na estrada, e cada aresta corresponde a uma ligação possível entre duas posições. Todas as arestas têm um custo unitário associado, igual a 1, e os vértices irão guardar a quantidade mínima de arestas que o ligam ao vértice inicial. Para além desse número, é também armazenado o valor da velocidade e da posição anterior.

O programa percorre todos os vértices e, para cada um deles, vai atualizar os dados dos vértices para os quais pode saltar. Se não conseguir saltar para nenhum, testando todas as velocidades possíveis, vai tentar encontrar um vértice à sua frente que tenha um custo menor que si mesmo. Caso encontre, salta para essa posição ou, caso contrário, retorna para o vértice anterior.

Quando atinge a última posição, o programa lê as posições anteriores de menor custo, guardadas em cada vértice, obtendo assim o caminho de menor custo.

Vantagens e Desvantagens

Por um lado, ao contrário dos dois primeiros, este método não tenta encontrar soluções diferentes para a posição final, pois quando lá chega já tem o caminho mais curto traçado.

Por outro lado, irá demorar mais em comparação com os dois métodos anteriores, pois percorre cada posição da estrada pelo menos uma vez, para além de também gastar mais recursos de memória para guardar dados que podem ou não ser úteis para a busca da solução.

Resultados

Unitary-Cost Dijkstra Recursion							
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	1.204e-05	55	19	56	2.732e-06
2	2	3	4.860e-07	60	21	61	2.067e-06
3	3	4	4.570e-07	65	23	66	2.177e-06
4	3	5	4.820e-07	70	25	71	2.344e-06
5	4	6	5.280e-07	75	26	76	2.140e-06
6	4	7	4.660e-07	80	27	81	4.029e-06
7	5	8	4.860e-07	85	28	86	2.410e-06
8	5	9	4.650e-07	90	29	91	2.631e-06
9	5	10	5.300e-07	95	31	96	2.904e-06
10	6	11	6.170e-07	100	33	320	1.161e-05
11	6	12	8.710e-07	110	38	330	1.517e-05
12	6	13	6.740e-07	120	41	340	1.652e-05
13	7	14	7.590e-07	130	43	350	1.659e-05
14	7	15	6.100e-07	140	45	360	2.096e-05
15	7	16	8.190e-07	150	47	370	2.334e-05
16	7	17	9.190e-07	160	50	12439	3.120e-04
17	8	18	1.192e-06	170	55	12449	4.446e-04
18	8	19	8.810e-07	180	58	12459	7.172e-04
19	8	20	9.110e-07	190	60	12469	8.319e-04
20	8	21	9.410e-07	200	62	12479	8.623e-04
21	9	22	1.427e-06	220	68	12499	1.024e-03
22	9	23	8.180e-07	240	76	12519	1.088e-03
23	9	24	1.127e-06	260	80	12539	8.175e-04
24	9	25	1.251e-06	280	87	21706	1.152e-03
25	9	26	1.590e-06	300	95	21726	1.363e-03
26	10	27	1.129e-06	320	100	21874	1.482e-03
27	10	28	1.023e-06	340	109	83137	3.648e-03
28	10	29	1.328e-06	360	113	83157	4.605e-03
29	10	30	1.337e-06	380	118	83364	5.215e-03
30	10	31	1.579e-06	400	126	83407	5.106e-03
31	11	32	1.575e-06	420	132	83427	4.653e-03
32	11	33	1.868e-06	440	136	83447	6.686e-03
33	11	34	1.010e-06	460	145	83467	8.311e-03
34	12	35	1.229e-06	480	150	83487	7.226e-03
35	12	36	1.194e-06	500	156	83507	7.223e-03
36	12	37	1.340e-06	520	164	83527	1.036e-02
37	13	38	1.392e-06	540	169	83547	9.125e-03
38	13	39	1.222e-06	560	176	103181	1.157e-02
39	13	40	1.321e-06	580	183	103201	1.142e-02
40	14	41	1.395e-06	600	187	103221	1.191e-02
41	14	42	1.405e-06	620	193	103426	1.402e-02
42	14	43	1.362e-06	640	201	103446	1.335e-02
43	15	44	1.416e-06	660	206	113012	1.571e-02
44	15	45	1.333e-06	680	212	113032	1.542e-02
45	15	46	1.378e-06	700	221	113052	1.734e-02
46	16	47	1.448e-06	720	226	113072	1.953e-02
47	16	48	1.461e-06	740	235	127852	2.090e-02
48	16	49	1.390e-06	760	240	127872	2.045e-02
49	16	50	4.958e-06	780	243	127892	2.159e-02
50	17	51	2.390e-06	800	253	127925	2.171e-02

*Figura 15 - Número mecanográfico **108122***

		Unitary-Cost Dijkstra Recursion					
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	5.640e-07	55	19	189	5.026e-06
2	2	3	2.970e-07	60	22	194	5.467e-06
3	3	4	2.770e-07	65	23	199	5.394e-06
4	3	5	3.090e-07	70	25	204	5.732e-06
5	4	6	3.250e-07	75	26	209	6.924e-06
6	4	7	2.870e-07	80	27	214	5.980e-06
7	5	8	3.150e-07	85	28	219	6.248e-06
8	5	9	3.050e-07	90	29	224	1.978e-05
9	5	10	3.250e-07	95	31	228	6.875e-06
10	6	11	3.650e-07	100	33	344	9.981e-06
11	6	12	3.320e-07	110	38	354	1.179e-05
12	6	13	3.900e-07	120	41	364	1.313e-05
13	7	14	3.710e-07	130	43	374	1.477e-05
14	7	15	3.290e-07	140	45	384	1.452e-05
15	7	16	3.530e-07	150	47	394	1.376e-05
16	7	17	4.690e-07	160	50	623	2.047e-05
17	8	18	4.780e-07	170	55	633	2.177e-05
18	8	19	4.200e-07	180	58	643	2.366e-05
19	8	20	4.220e-07	190	61	653	2.522e-05
20	8	21	5.250e-07	200	62	663	2.558e-05
21	9	22	5.480e-07	220	69	682	3.027e-05
22	9	23	4.280e-07	240	76	702	3.276e-05
23	9	24	4.550e-07	260	80	722	3.479e-05
24	9	25	4.850e-07	280	87	856	4.186e-05
25	9	26	7.160e-07	300	95	876	4.469e-05
26	10	27	6.680e-07	320	99	896	4.643e-05
27	10	28	6.440e-07	340	108	1135	5.933e-05
28	10	29	6.400e-07	360	112	1155	6.452e-05
29	11	30	6.920e-07	380	117	1175	6.600e-05
30	11	70	1.986e-06	400	126	1371845	2.876e-02
31	11	165	3.847e-06	420	131	1371865	4.833e-02
32	12	166	3.619e-06	440	135	1371885	6.238e-02
33	12	167	3.559e-06	460	143	1371905	7.312e-02
34	12	168	3.947e-06	480	149	1371925	8.191e-02
35	13	169	3.998e-06	500	155	1372056	8.547e-02
36	13	170	4.115e-06	520	163	1372076	9.398e-02
37	13	171	4.155e-06	540	168	1372096	9.926e-02
38	14	172	3.923e-06	560	174	1372116	1.060e-01
39	14	173	4.083e-06	580	181	1372136	1.121e-01
40	14	174	4.089e-06	600	185	1372156	1.156e-01
41	15	175	4.531e-06	620	191	1372173	1.258e-01
42	15	176	4.500e-06	640	198	1372193	1.314e-01
43	15	177	4.305e-06	660	202	1372213	1.397e-01
44	16	178	4.584e-06	680	209	1372233	1.464e-01
45	16	179	4.366e-06	700	216	1372253	1.503e-01
46	16	180	4.351e-06	720	221	1683232	1.614e-01
47	16	181	5.468e-06	740	230	2742943	1.852e-01
48	17	182	4.311e-06	760	235	2742963	2.064e-01
49	17	183	4.488e-06	780	239	2742983	2.184e-01
50	17	184	4.368e-06	800	248	2752150	2.356e-01

*Figura 16 - Número mecanográfico **110056***

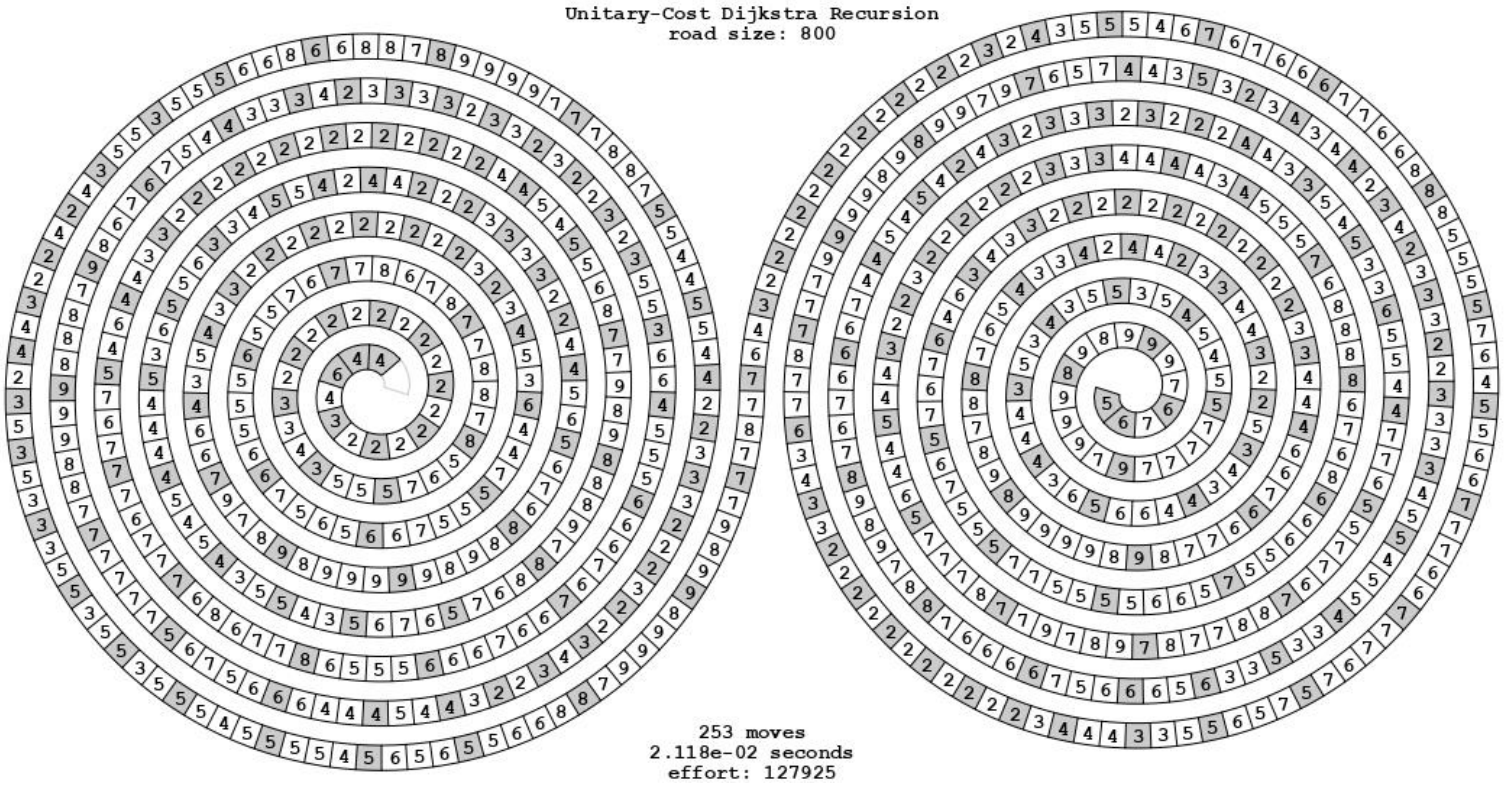


*Figura 17 - Número mecanográfico **108122***

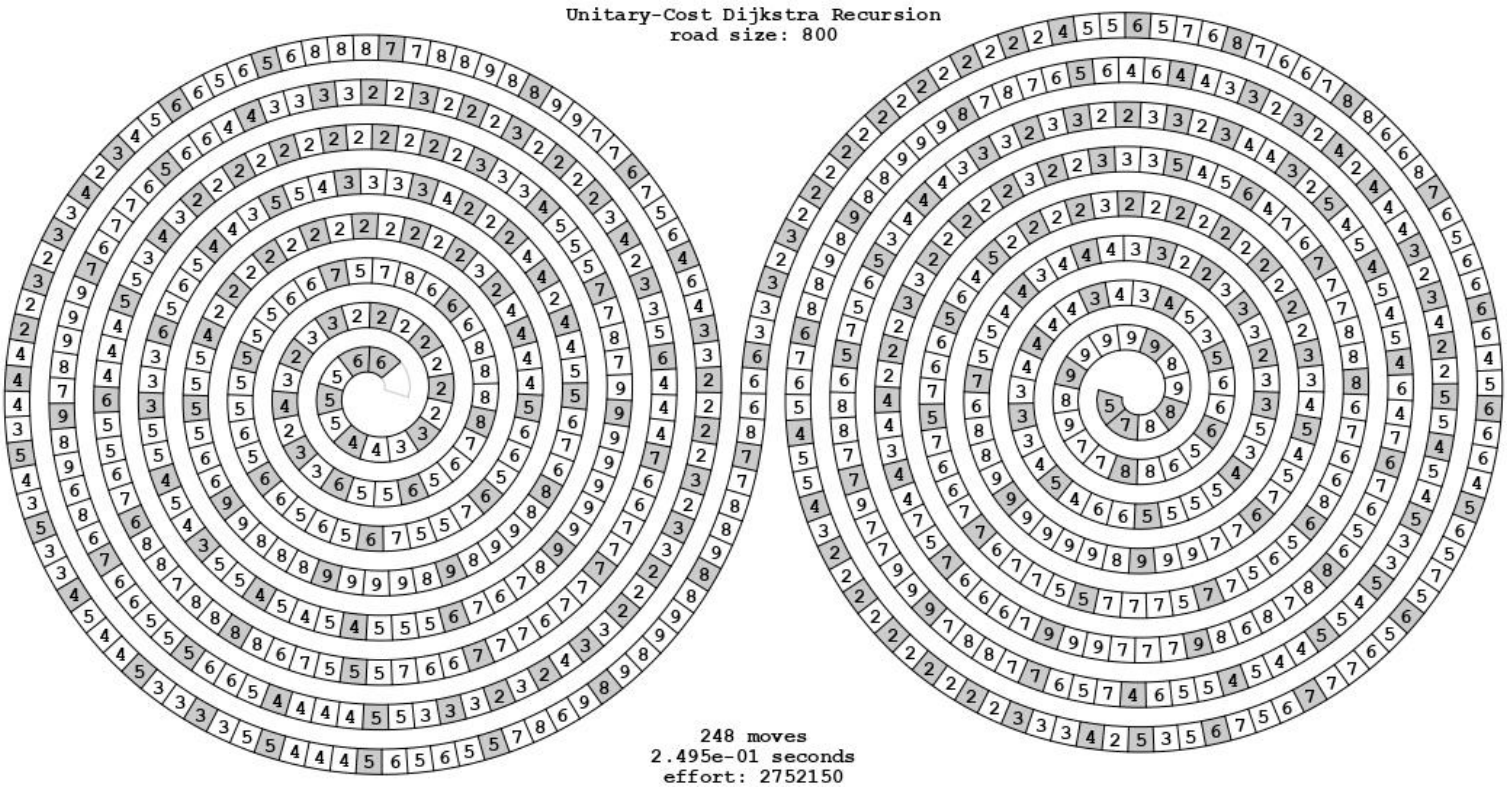
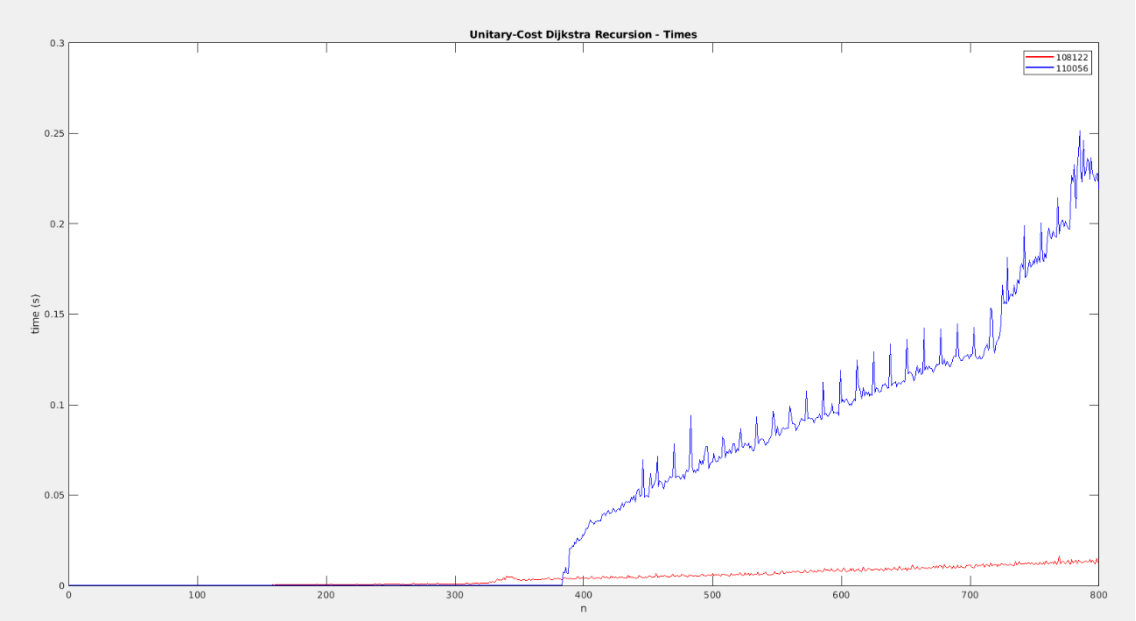


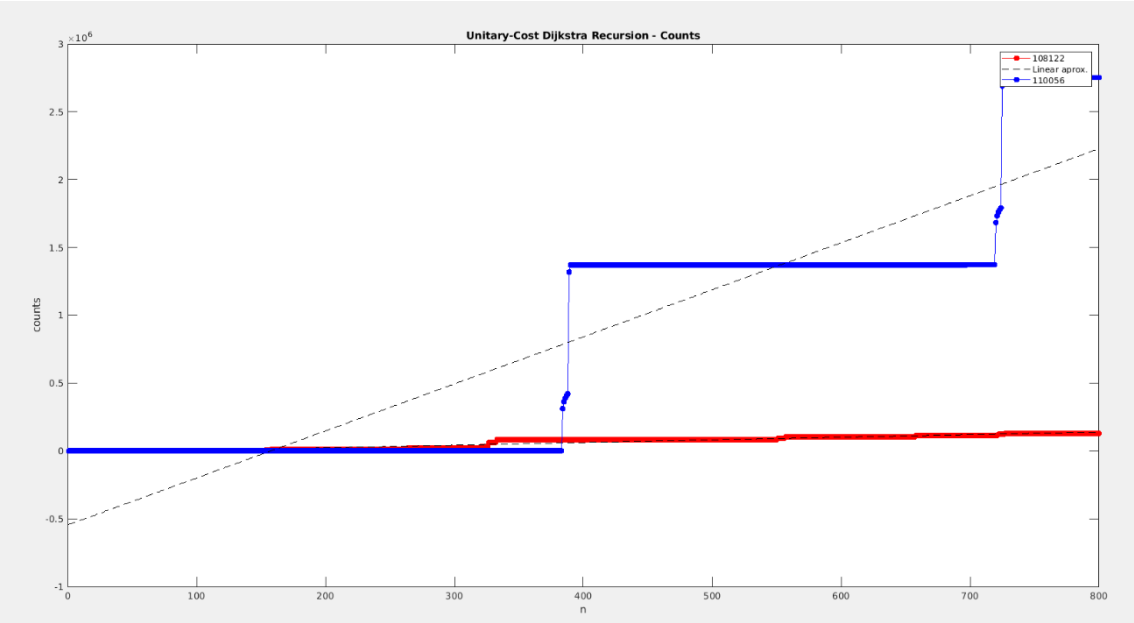
*Figura 18 - Número mecanográfico **110056***

Análise de dados

Portanto, como podemos observar, os resultados diferem **muito** de um número mecanográfico para o outro, e a razão para tal acontecer não nos é aparente. É possível que a implementação não tenha sido feita da melhor forma, e que haja alguma inconsistência no código.

Porém, após compararmos ao 3º método em vários casos, com números mecanográficos diferentes, concluímos que as soluções geradas estão corretas, ou pelo menos têm o mesmo número de movimentos.





Dada a natureza dos gráficos obtidos, que contêm uma grande discrepância entre valores diferentes de **n**, não podemos concluir que a progressão da função **count** seja aproximável a uma função polinomial previsível.

Speed-Based A Recursion*

Raciocínio

Este método recorre ao algoritmo A* (A-star), derivado do de Dijsktra, que implementa a estrutura de dados ***priority queue*** (fila com prioridade).

Ao contrário do método anterior, o custo das arestas é baseado na velocidade do vértice origem e numa métrica adicional que, neste caso, é a distância entre esse vértice e o da posição final. É esta métrica que distingue o algoritmo A* do algoritmo de Dijkstra.

Para cada vértice, o programa testa as velocidades e insere as posições possíveis na **queue**, que vai ser ordenada pelo menor custo. Se nenhuma velocidade for válida, é feito um retorno para a última posição que visitou.

No entanto, se pelo menos uma for válida, o programa testa as posições que estão no topo da **queue** e usa os seus dados para fazer novas iterações, dependendo do número de velocidades válidas.

O programa acaba quando chegar à posição final.

Vantagens e Desvantagens

Ao contrário do método anterior, esta implementação não requer que percorremos todos os vértices, o que resulta em tempos de execução mais rápidos.

Aliás, a execução do programa vai acabar por ser quase idêntica à do método *Fast Recursion*, com a desvantagem de gastar mais recursos de memória, no uso da ***priority queue***.

Resultados

Speed-Based A* Recursion							
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	1.192e-06	55	19	20	1.818e-06
2	2	3	5.990e-07	60	21	22	1.735e-06
3	3	4	4.950e-07	65	23	24	3.547e-06
4	3	4	5.380e-07	70	25	26	1.932e-06
5	4	5	4.130e-07	75	26	27	2.193e-06
6	4	5	4.650e-07	80	27	28	1.853e-06
7	5	6	4.500e-07	85	28	29	2.044e-06
8	5	6	4.900e-07	90	29	30	3.993e-06
9	5	6	8.970e-07	95	31	32	3.855e-06
10	6	7	7.020e-07	100	33	35	3.198e-06
11	6	7	8.070e-07	110	38	40	3.553e-06
12	6	7	7.480e-07	120	41	43	5.039e-06
13	7	8	6.370e-07	130	43	45	3.553e-06
14	7	8	7.320e-07	140	45	47	3.536e-06
15	7	8	8.210e-07	150	47	49	6.381e-06
16	7	8	9.490e-07	160	50	54	4.518e-06
17	8	9	1.036e-06	170	55	59	7.807e-06
18	8	9	1.092e-06	180	58	62	6.140e-06
19	8	9	2.051e-06	190	60	64	7.009e-06
20	8	9	6.920e-07	200	62	66	8.733e-06
21	9	10	6.030e-07	220	68	72	7.399e-06
22	9	10	6.410e-07	240	76	80	1.405e-05
23	9	10	6.260e-07	260	80	84	1.109e-05
24	9	10	6.900e-07	280	87	94	1.077e-05
25	9	10	7.460e-07	300	95	102	1.394e-05
26	10	11	6.640e-07	320	100	108	1.171e-05
27	10	11	6.980e-07	340	109	121	1.429e-05
28	10	11	7.030e-07	360	113	125	1.689e-05
29	10	11	7.250e-07	380	118	131	1.804e-05
30	10	11	8.010e-07	400	126	140	1.661e-05
31	11	12	7.120e-07	420	132	146	1.776e-05
32	11	12	7.080e-07	440	136	150	1.725e-05
33	11	12	1.149e-06	460	145	159	1.997e-05
34	12	13	2.430e-06	480	150	164	1.893e-05
35	12	13	9.820e-07	500	156	170	2.123e-05
36	12	13	1.076e-06	520	164	178	2.772e-05
37	13	14	1.194e-06	540	169	183	2.425e-05
38	13	14	1.130e-06	560	176	193	2.342e-05
39	13	14	1.650e-06	580	183	200	2.309e-05
40	14	15	1.512e-06	600	187	204	2.158e-05
41	14	15	1.613e-06	620	193	212	1.793e-05
42	14	15	1.261e-06	640	201	220	2.593e-05
43	15	16	9.320e-07	660	206	228	2.703e-05
44	15	16	9.120e-07	680	212	234	2.435e-05
45	15	16	2.280e-06	700	221	243	3.352e-05
46	16	17	8.850e-07	720	226	248	2.057e-05
47	16	17	9.290e-07	740	235	261	2.023e-05
48	16	17	9.070e-07	760	240	266	2.227e-05
49	16	17	9.770e-07	780	243	269	2.180e-05
50	17	18	1.559e-06	800	253	280	2.384e-05

*Figura 19 - Número mecanográfico **108122***

Speed-Based A* Recursion							
n	sol	count	cpu time	n	sol	count	cpu time
1	1	2	1.373e-06	55	19	21	2.822e-06
2	2	3	8.780e-07	60	22	24	4.487e-06
3	3	4	9.670e-07	65	23	25	2.780e-06
4	3	4	9.830e-07	70	25	27	2.935e-06
5	4	5	7.920e-07	75	26	28	2.468e-06
6	4	5	1.055e-06	80	27	29	2.551e-06
7	5	6	9.100e-07	85	28	30	4.723e-06
8	5	6	1.110e-06	90	29	31	2.882e-06
9	5	6	1.276e-06	95	31	33	3.421e-06
10	6	7	1.240e-06	100	33	36	4.142e-06
11	6	7	1.217e-06	110	38	41	5.929e-06
12	6	7	1.276e-06	120	41	44	4.614e-06
13	7	8	1.248e-06	130	43	46	4.169e-06
14	7	8	1.331e-06	140	45	48	6.405e-06
15	7	8	1.332e-06	150	47	50	4.874e-06
16	7	8	1.497e-06	160	50	54	5.794e-06
17	8	9	1.400e-06	170	55	59	8.098e-06
18	8	9	1.412e-06	180	58	62	6.348e-06
19	8	9	3.167e-06	190	61	65	1.676e-05
20	8	9	1.454e-06	200	62	66	9.577e-06
21	9	10	1.480e-06	220	69	73	1.224e-05
22	9	10	1.493e-06	240	76	80	8.456e-06
23	9	10	1.528e-06	260	80	84	1.105e-05
24	9	10	1.548e-06	280	87	93	9.026e-06
25	9	10	1.362e-06	300	95	101	1.238e-05
26	10	11	1.362e-06	320	99	105	8.998e-06
27	10	11	1.091e-06	340	108	115	1.336e-05
28	10	11	1.393e-06	360	112	119	1.151e-05
29	11	12	1.239e-06	380	117	124	1.163e-05
30	11	13	1.516e-06	400	126	141	1.431e-05
31	11	13	1.569e-06	420	131	146	1.109e-05
32	12	14	1.486e-06	440	135	150	1.083e-05
33	12	14	1.308e-06	460	143	158	1.100e-05
34	12	14	3.816e-06	480	149	164	1.265e-05
35	13	15	1.833e-06	500	155	171	1.154e-05
36	13	15	1.703e-06	520	163	179	1.357e-05
37	13	15	1.479e-06	540	168	184	1.337e-05
38	14	16	1.714e-06	560	174	190	1.277e-05
39	14	16	1.654e-06	580	181	197	1.332e-05
40	14	16	1.456e-06	600	185	201	1.690e-05
41	15	17	1.790e-06	620	191	207	1.773e-05
42	15	17	1.604e-06	640	198	214	1.701e-05
43	15	17	1.716e-06	660	202	218	1.878e-05
44	16	18	3.860e-06	680	209	225	1.160e-05
45	16	18	2.601e-06	700	216	232	1.311e-05
46	16	18	2.330e-06	720	221	240	1.183e-05
47	16	18	1.668e-06	740	230	254	1.638e-05
48	17	19	1.727e-06	760	235	259	1.669e-05
49	17	19	1.734e-06	780	239	263	1.708e-05
50	17	19	2.016e-06	800	248	275	1.684e-05

*Figura 20 - Número mecanográfico **110056***

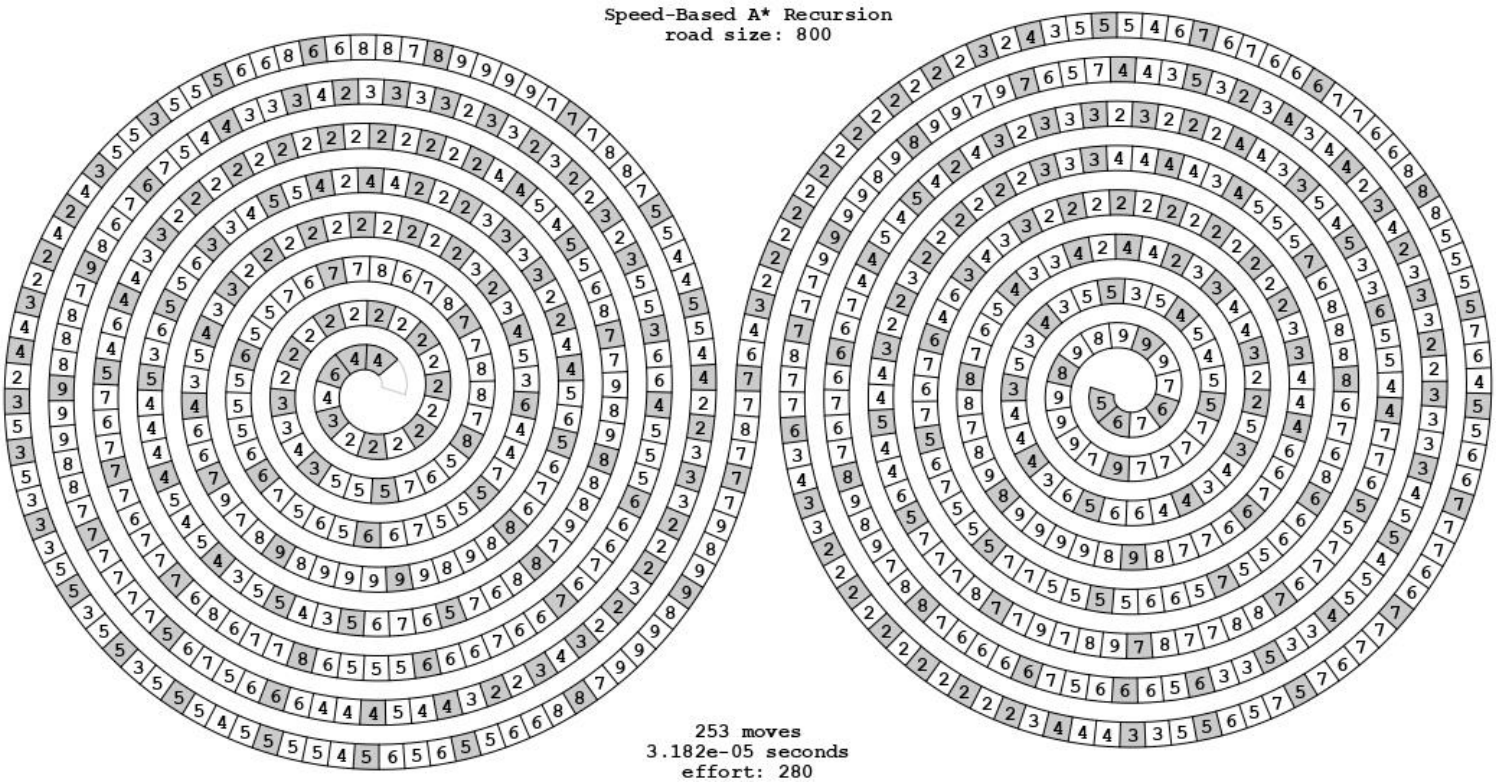


*Figura 23 - Número mecanográfico **108122***

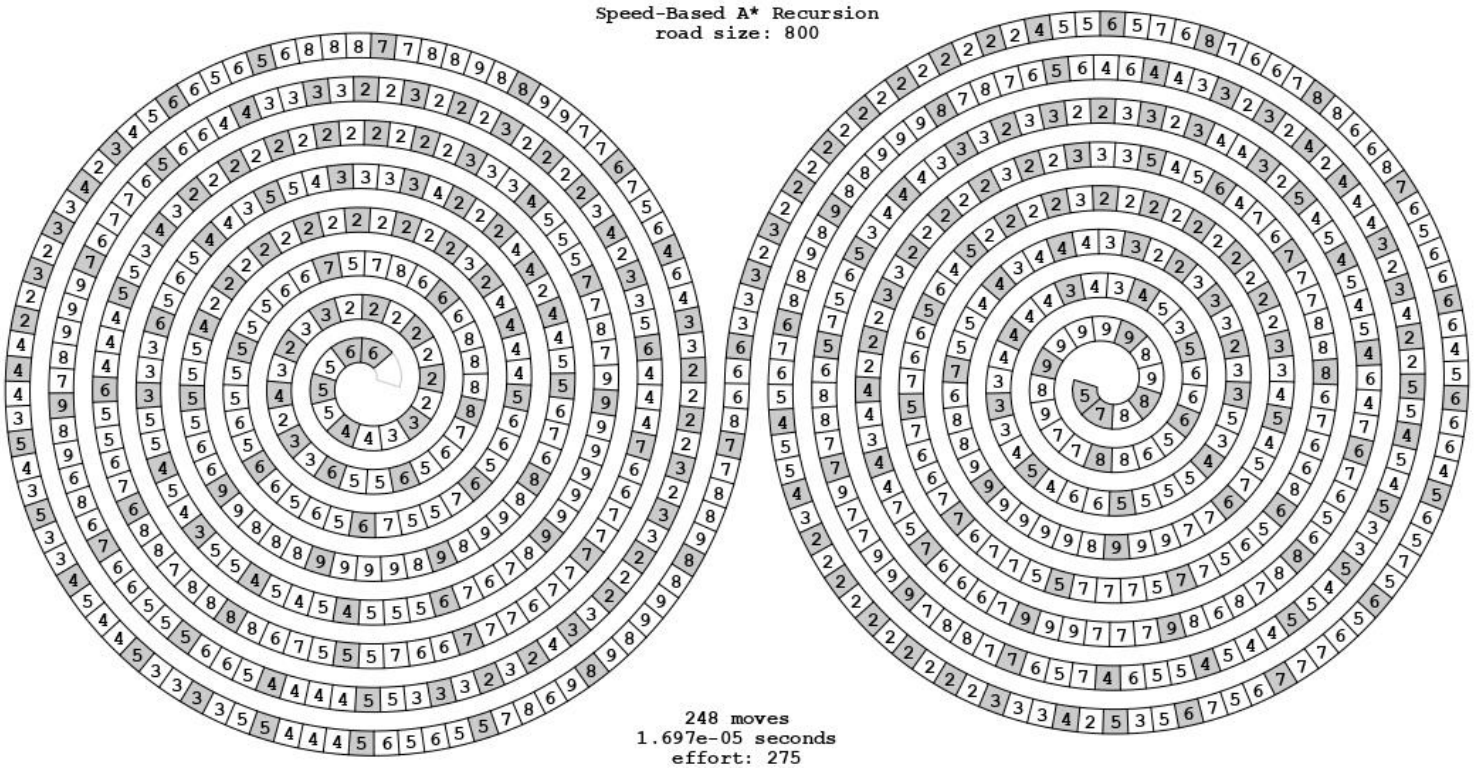
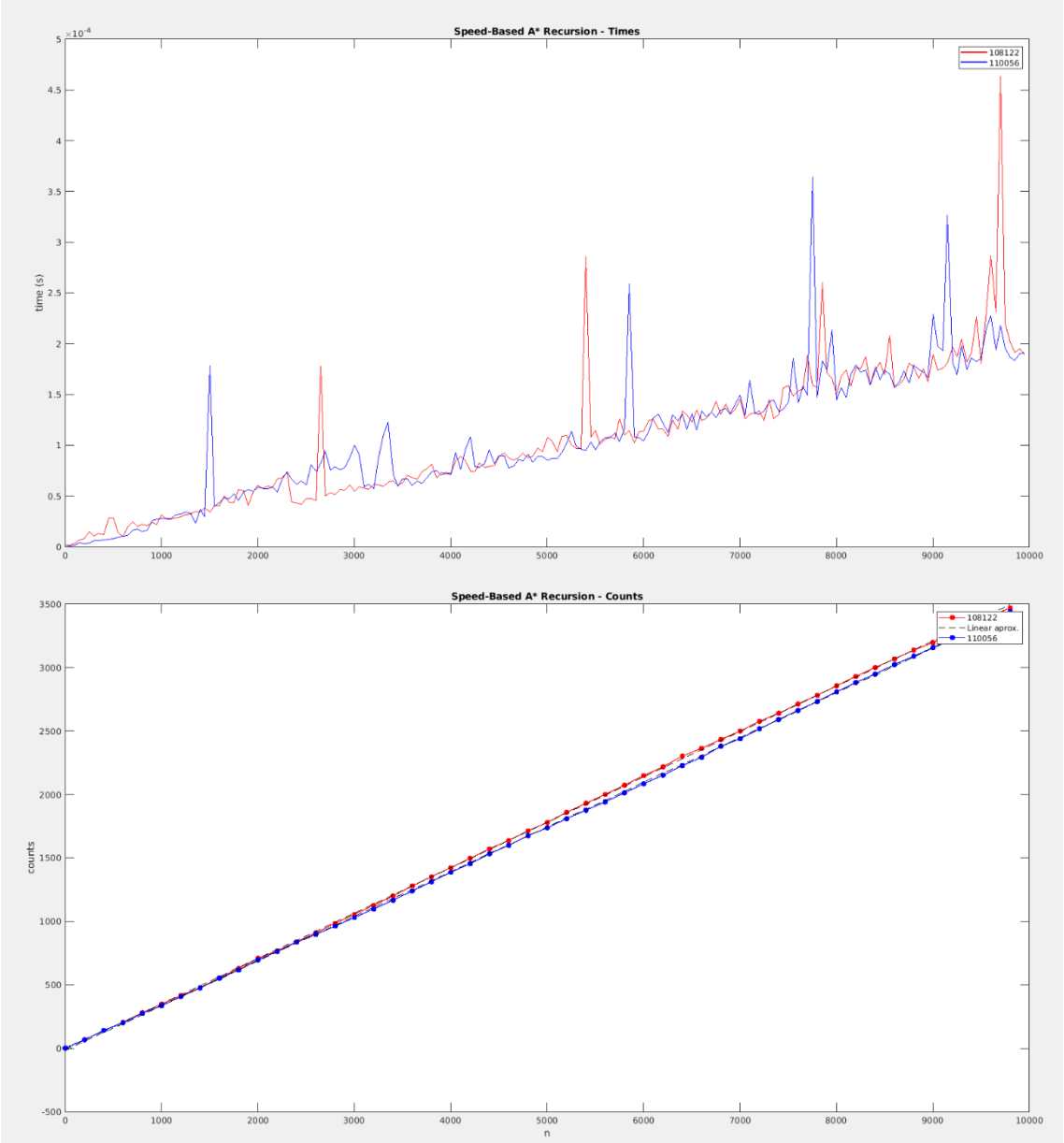


*Figura 24 - Número mecanográfico **110056***

Análise de dados

Tal como previsto, os resultados são bastante semelhantes aos obtidos no método *Fast Recursion*, sendo o gráfico dos **counts** exatamente igual. Isto leva-nos a concluir que este programa segue o mesmo algoritmo que o método referido, mas com uma abordagem diferente, notável nos tempos de execução.



As aproximações da função dos **counts** são exatamente iguais às obtidas no método anterior, que são:

- Nº mec. 108122 - **O(n)** = 0.3576***n** – 8.6093
- Nº mec. 110056 - **O(n)** = 0.3523***n** – 17.0746

Fast Recursion – Contraditório?

Este capítulo tem como objetivo tentar encontrar alguma razão para desaprovar o método *Fast Recursion*. Cada subcapítulo representa uma maneira que utilizamos para cumprir este propósito.

Comparação com outros métodos

Ao longo deste trabalho, utilizámos este método como forma de aprovar os outros que fomos criando, porém, não tínhamos termo de comparação para verificar a veracidade do mesmo.

Por isso, desenvolvemos um script em BASH que permite executar o ficheiro **speed_run.c** várias vezes para um intervalo de números mecanográficos diferentes. Este ficheiro, dentro deste contexto, irá comparar os resultados dos métodos *Fast Recursion* e *Slightly Better Recursion* (este método tem garantia de uma solução correta) até um tamanho máximo razoável da estrada, e dependendo do resultado retorna 1 ou 0.

Ao executarmos o script, e o deixarmos correr durante um tempo que achámos pertinente, chegámos à conclusão que, para os tamanhos de estrada até 70, o método *Fast Recursion* **está sempre correto.**

Percurso da estrada ao contrário

Outra forma que utilizámos para provar a contradição foi utilizar o mesmo método, na mesma estrada, mas começando em pontas diferentes. Para isso, criámos um método diferente que utiliza o algoritmo da *Fast Recursion* para resolver a estrada de trás para a frente. Adicionalmente, para garantir que o número de movimentos é sempre igual, utilizámos o script BASH desenvolvido anteriormente.

Os resultados estão apresentados na página seguinte, e podemos deles concluir que o número de movimentos é sempre igual, mas o caminho escolhido pode ser diferente. De qualquer forma, isto não serviu para desaprovar o método.

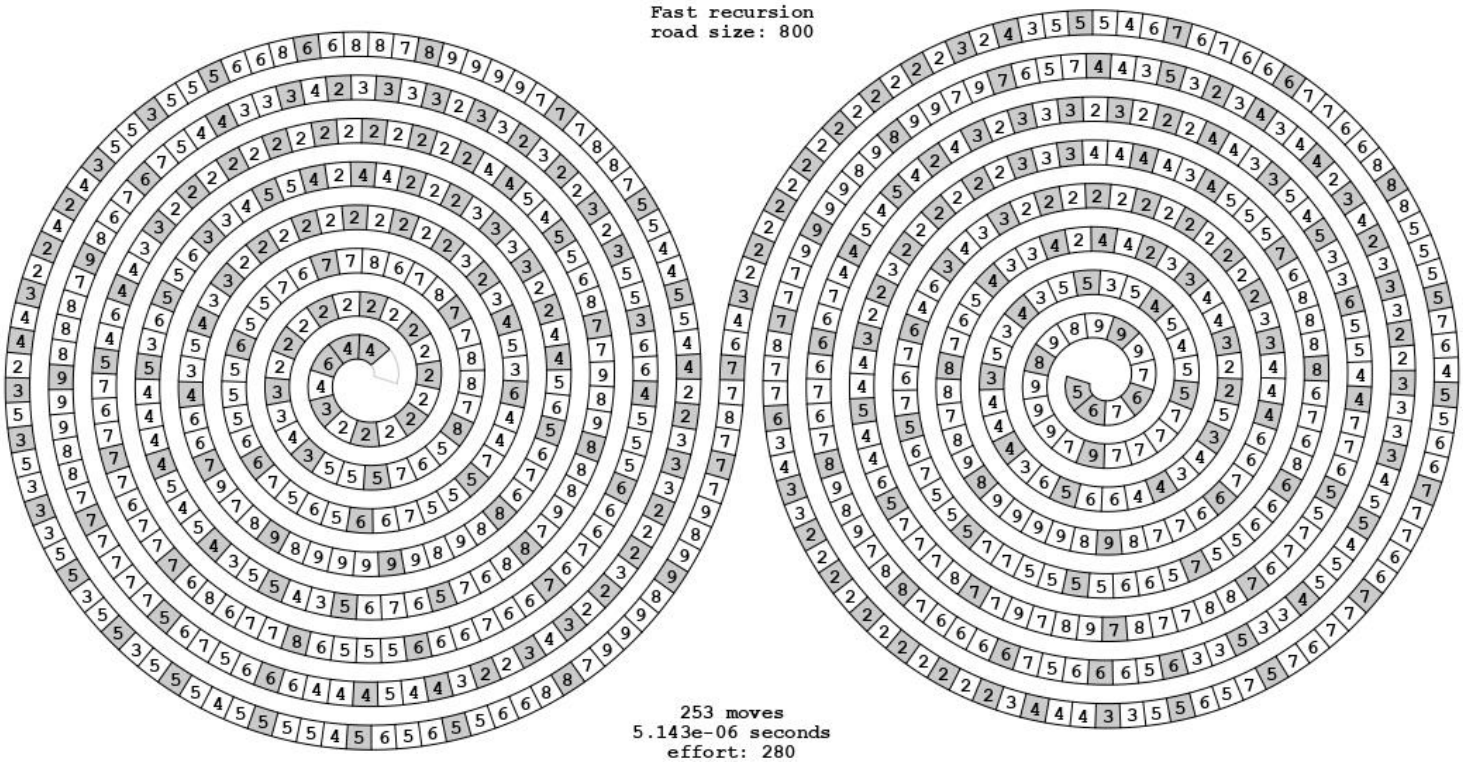


*Figura 15 - Número mecanográfico **108122***

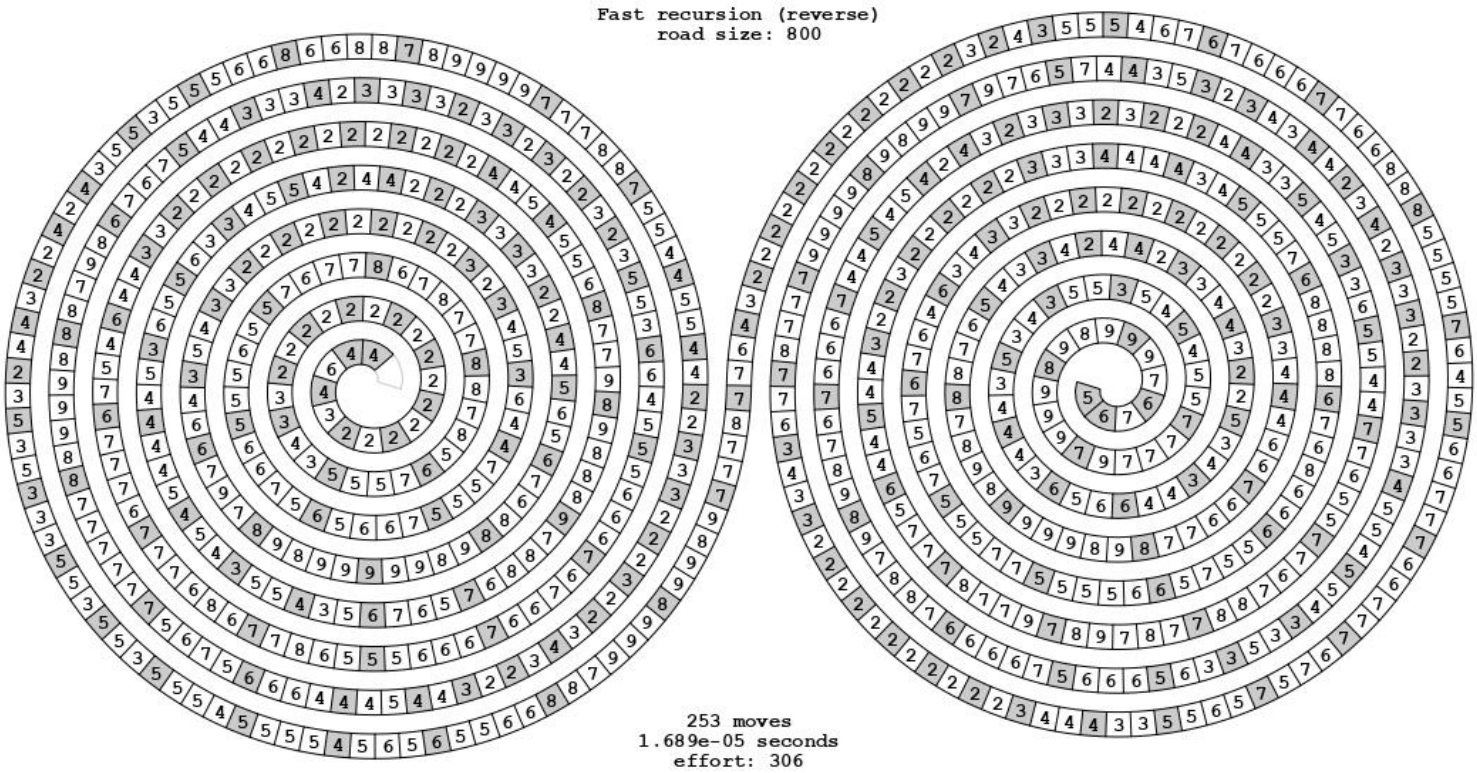


*Figura 26 - Número mecanográfico **108122***

Conclusão

O objetivo principal deste trabalho foi cumprido, visto que arranjámos um método que consegue percorrer a estrada de comprimento 800, nas condições impostas, em apenas alguns microssegundos, nomeado de *Fast Recursion*.

Para além desse, apresentámos uma grande variedade de métodos que obtém os mesmos resultados utilizando abordagens diferentes, e aplicando estruturas de dados diferentes.

Por fim, concluímos que este trabalho contribuiu bastante para o nosso conhecimento da linguagem C e de algoritmos de custo mínimo, e também nos ajudou a pensar de forma mais crítica na abordagem de certos problemas e na implementação de soluções eficientes.

Webgrafia

https://www.geeksforgeeks.org/priority-queue-using-linked-list/

https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra

https://pt.wikipedia.org/wiki/Algoritmo_A*

Código

Linguagem C

speed_run.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. 108122  Name: Alexandre Pedro Ribeiro
//   N.Mec. 110056  Name: Ricardo Manuel Quintaneiro Almeida
//


//
// 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"
#include "priorityQueue.h"      // required for Solution 6


//
// road stuff
//

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

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

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

}


//
```

```
// description of a solution
//

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


//
// Solution 1 - Very bad recursion --- the (very inefficient) recursive solution given to the students
//

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

static void solution_1_recursion(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 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;
}


//
// Solution 2 - Slightly better recursion
//

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_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)
  {
    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--)
  {
    if(new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed*(new_speed+1)/2 <=
final_position)
    {
      for(i = 0;i <= new_speed && new_speed <= max_road_speed[position + i];i++)
        ;
      if(i > new_speed)
      {
        if ((move_number + 1) >= solution_2_best.n_moves) return;
        solution_2_recursion(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_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;
}

//
// Solution 3 - Fast Recursion
//

static solution_t 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 int solution_3_test(int move_number, int position, int speed, int final_position)
{
  // register count
  int new_speed, k;
  solution_3_count++;

  // check if it is the solution
  if(position == final_position && speed == 1) {
      solution_3_best.positions[move_number] = position;
      return 0;
  }

  // test possible speeds, starting from fastest one
  for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--){
        if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed*(new_speed+1)/2 <=
final_position) {
          for (k=0;k<=new_speed && new_speed <= max_road_speed[position + k];k++) {
              ;
          }
          if (k > new_speed) {
              if (solution_3_test(move_number + 1,position + new_speed,new_speed,final_position) == 0){
                  solution_3_best.n_moves++;
                  solution_3_best.positions[move_number] = position;
```

```
                return 0;
            }
        }
    }
  }
  return 1;
}

static void solve_3(int final_position)
{
    if(final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr,"solve_3: bad final_position\n");
        exit(1);
    }
    solution_3_best.n_moves = 0ul;
    solution_3_elapsed_time = cpu_time();
    solution_3_count = 0ul;
    solution_3_test(0,0,0,final_position);
    solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}

//
// Solution 4 - Fast Dynamic Non-Recursion
//

static solution_t solution_4_best;
static double solution_4_elapsed_time; // time it took to solve the problem
static unsigned long solution_4_count; // effort dispended solving the problem

static void solution_4_no_recursion(int move_number, int position, int speed, int final_position)
{
    int new_speed, returned, i, k;
    // declare arrays to store move data
    int positions[final_position];
    int speeds[final_position];

    // register count
    solution_4_count++;

    // record move
    positions[0] = position;
    speeds[0] = speed;

    for (i = 0; i <= final_position; i++)
    {
      returned = 1;
      if (positions[i] == final_position && speeds[i] == 1)
      {
        solution_4_best.n_moves = i;
        for (int j = 0; j <= i; j++)
          solution_4_best.positions[j] = positions[j];
        break;
      }
      for (new_speed = speeds[i] + 1; new_speed >= speeds[i] - 1; new_speed--)
      {
        if (new_speed >= 1 && new_speed <= _max_road_speed_ && positions[i] + new_speed * (new_speed + 1) / 2
<= final_position)
        {
          for (k = 0; k <= new_speed && new_speed <= max_road_speed[positions[i] + k]; k++)
            ;
          if (k > new_speed)
          {
            solution_4_count++;
            speeds[i + 1] = new_speed;
            positions[i + 1] = positions[i] + new_speed;
            returned = 0;
            break;
          }
        }
      }
      for (k = 0; k < i; k++)
      {
        if (returned == 1 && speeds[i - 1] >= 2 && speeds[i - k] >= speeds[i - k - 1])
        {
```

```
          solution_4_count++;
          speeds[i - k] = speeds[i - k] - 1;
          positions[i - k] = positions[i - k] - 1;
          i = i - k - 1;
          break;
        }
      }
    }
}

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_best.n_moves = 0ul;
    solution_4_elapsed_time = cpu_time();
    solution_4_count = 0ul;
    solution_4_no_recursion(0,0,0,final_position);
    solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}

//
// Solution 5 - Unitary-Cost Dijkstra Recursion
//

static solution_t solution_5_best;
static double solution_5_elapsed_time; // time it took to solve the problem
static unsigned long solution_5_count; // effort dispended solving the problem

static int node[_max_road_size_ + 1][3];

#define node_cost 0
#define node_lastPosition 1
#define node_speed 2

static int solution_5_recursion(int position, int final_position)
{
    int i, new_speed;
    int cost = node[position][node_cost];
    int speed = node[position][node_speed];
    solution_5_count++;
    if (position == 0)
    {
        for (int i = 1; i <= final_position; i++)
        {
            node[i][node_cost] = i;
            node[i][node_lastPosition] = i - 1;
            node[i][node_speed] = 1;
        }
        solution_5_recursion(1, final_position);
        return 0;
    }

    if (position == final_position && speed == 1)
    {
        int positions[final_position + 1];
        solution_5_best.n_moves = 0;
        while (position != 0)
        {
          positions[solution_5_best.n_moves] = position;
          position = node[position][node_lastPosition];
          solution_5_best.n_moves++;
        }
        int m = 0;
        for (int j = solution_5_best.n_moves; j > 0; j--)
        {
            solution_5_best.positions[j] = positions[m++];
        }
        return 0;
    }

    int new_position, temp_cost, temp_pos, temp_ns;
```

```
    for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    {
        if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed * (new_speed + 1) / 2 <=
final_position)
        {
          new_position = position + new_speed;
          for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
            ;
          if (i > new_speed)
          {
            temp_cost = node[new_position][node_cost];
            temp_pos = node[new_position][node_lastPosition];
            temp_ns = node[new_position][node_speed];
            if (temp_cost > cost + 1)
            {
                node[new_position][node_cost] = cost + 1;
                node[new_position][node_lastPosition] = position;
                node[new_position][node_speed] = new_speed;
            }
            if (solution_5_recursion(position + 1, final_position) == 0)
                return 0;
            else
            {
              node[new_position][node_cost] = temp_cost;
              node[new_position][node_lastPosition] = temp_pos;
              node[new_position][node_speed] = temp_ns;
            }
          }
        }
    }
    for (i = position + 1; i <= final_position; i++)
    {
        if (node[i][node_cost] < node[position][node_cost])
          break;
    }
    if (i < final_position + 1)
    {
        if (solution_5_recursion(i, final_position) == 0)
          return 0;
    }
    return 1;
}

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;
  solution_5_recursion(0,final_position);
  solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;
}

//
// Solution 6 - Speed-Based A* Recursion
//

static solution_t solution_6_best;
static double solution_6_elapsed_time; // time it took to solve the problem
static unsigned long solution_6_count; // effort dispended solving the problem
static Node* priorityQueue;

static int solution_6_Astar(int move_number, int position, int speed, Node* priorityQueue, int final_position)
{
  int i, new_speed;
  solution_6_count++;

  // record move
  solution_6_best.positions[move_number] = position;
  // is it a solution?
```

```
  if(position == final_position && speed == 1)
  {
    solution_6_best.n_moves = move_number;
    return 0;
  }

  // no, try all legal speeds
  int count = 0;
  for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--)
  {
    if(new_speed >=1 && new_speed <= _max_road_speed_ && position + new_speed*(new_speed+1)/2 <=
final_position)
    {
      for(i = 0;i <= new_speed && new_speed <= max_road_speed[position + i];i++)
        ;
      if(i > new_speed)
      {
        push(&priorityQueue, position+new_speed, _max_road_speed_+1-new_speed+final_position-position,
new_speed);
        count++;
      }
    }
  }
  if (count == 0)
    return 1;
         int a = peek(&priorityQueue);
  int b = peek_Speed(&priorityQueue);
  pop(&priorityQueue);
  while(solution_6_Astar(move_number + 1, a, b, &priorityQueue, final_position) == 1)
  {
    count--;
    if (count == 0) return 1;
    a = peek(&priorityQueue);
    b = peek_Speed(&priorityQueue);
    pop(&priorityQueue);
  }
        return 0;
}

static void solve_6(int final_position)
{
  if(final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr,"solve_6: bad final_position\n");
    exit(1);
  }
  solution_6_elapsed_time = cpu_time();
  solution_6_count = 0ul;
  solution_6_best.n_moves = final_position + 100;
  priorityQueue = newNode(0, final_position+1, 0);
  solution_6_Astar(0,0,0,&priorityQueue,final_position);
  solution_6_elapsed_time = cpu_time() - solution_6_elapsed_time;
}

//
//  Solution 7 - Fast Recursion (reverse)
//

static solution_t solution_7_best;
static double solution_7_elapsed_time; // time it took to solve the problem
static unsigned long solution_7_count; // effort dispended solving the problem

static int solution_7_test(int move_number, int position, int speed, int final_position)
{
  // register count
  int new_speed, k;
  solution_7_count++;

  // check if it is the solution
  if(position == 0 && speed == 1) {
    solution_7_best.positions[move_number] = position;
    solution_7_best.n_moves = move_number;
    return 1;
  }
```

```
  // test possible speeds, starting from fastest one
  for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--){
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed*(new_speed+1)/2 >= 1) {
      for (k=0;k<=new_speed && new_speed <= max_road_speed[position - k];k++) {
        ;
      }
      if (k > new_speed) {
          if (solution_7_test(move_number + 1,position - new_speed,new_speed,final_position) == 1){
              solution_7_best.positions[move_number] = position;
              if (position == final_position)
              {
                int n = solution_7_best.n_moves + 1;
                int aux[_max_road_size_+1], i;

                        for (i = 0; i < n; i++) {
                            aux[n - 1 - i] = solution_7_best.positions[i];
                        }

                        for (i = 0; i < n; i++) {
                            solution_7_best.positions[i] = aux[i];
                        }
              }
              return 1;
          }
      }
    }
  }
  return 0;
}

static void solve_7(int final_position)
{
    if(final_position < 1 || final_position > _max_road_size_)
    {
        fprintf(stderr,"solve_7: bad final_position\n");
        exit(1);
    }
    solution_7_best.n_moves = 0ul;
    solution_7_elapsed_time = cpu_time();
    solution_7_count = 0ul;
    solution_7_test(0,final_position,0,final_position);
    solution_7_elapsed_time = cpu_time() - solution_7_elapsed_time;
}

//
// example of the slides
//

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

  srandom(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]);
  srandom((unsigned int)n_mec);
  init_road_speeds();
  // run all solution methods for all interesting sizes of the problem
  final_position = 1;
  solution_1_elapsed_time = 0.0;
  solution_2_elapsed_time = 0.0;
  solution_3_elapsed_time = 0.0;
  solution_4_elapsed_time = 0.0;
  solution_5_elapsed_time = 0.0;
  solution_6_elapsed_time = 0.0;
  solution_7_elapsed_time = 0.0;
  // printf("    + --- ---------------- --------- + --- ---------------- --------- + --- ---------------- ----
----- +\n");
  // printf("    |             Very bad recursion |      Slightly better recursion |                 Fast
recursion |\n");
  // printf("--- + --- ---------------- --------- + --- ---------------- --------- + --- ---------------- ----
----- +\n");
  // printf("  n | sol            count  cpu time | sol            count  cpu time | sol            count  cpu
time |\n");
  // printf("--- + --- ---------------- --------- + --- ---------------- --------- + --- ---------------- ----
----- +\n");
  printf("    + --- ---------------- --------- + --- ---------------- --------- + --- ---------------- -------
-- + --- ---------------- --------- +\n");
  printf("    |     Fast Dynamic Non-Recursion | Unitary-Cost Dijkstra Recursion|       Speed-Based A*
Recursion |       Fast Recursion (reverse) |\n");
  printf("--- + --- ---------------- --------- + --- ---------------- --------- + --- ---------------- -------
-- + --- ---------------- --------- +\n");
  printf("  n | sol            count  cpu time | sol            count  cpu time | sol            count  cpu
time | 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_1_elapsed_time < _time_limit_)
    // {
    //   solve_1(final_position);
    //   if(print_this_one != 0)
    //   {
    //     sprintf(file_name,"%03d_1.pdf",final_position);
    //
make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_1_best.n_moves,&solution_1_best.posi
tions[0],solution_1_elapsed_time,solution_1_count,"Very bad recursion");
    //   }
    //   printf(" %3d %16lu %9.3e |",solution_1_best.n_moves,solution_1_count,solution_1_elapsed_time);
    // }
    // else
    // {
    //   solution_1_best.n_moves = -1;
    //   printf("                                |");
    // }
    // second solution method (less bad)
    // // ...
    // if(solution_2_elapsed_time < _time_limit_)
    // {
    //   solve_2(final_position);
    //   // if(print_this_one != 0)
    //   // {
    //   //   sprintf(file_name,"%03d_2.pdf",final_position);
```

```
    //   //
make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_2_best.n_moves,&solution_2_best.posi
tions[0],solution_2_elapsed_time,solution_2_count,"Slightly better recursion");
    //   // }
    //   printf(" %3d %16lu %9.3e |",solution_2_best.n_moves,solution_2_count,solution_2_elapsed_time);
    // }
    // else
    // {
    //   solution_2_best.n_moves = -1;
    //   printf("                                |");
    // }
    // // done
    // if(solution_3_elapsed_time < _time_limit_)
    // {
    //   solve_3(final_position);
    //   // if(print_this_one != 0)
    //   // {
    //   //   sprintf(file_name,"%03d_3.pdf",final_position);
    //   //
make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_3_best.n_moves,&solution_3_best.posi
tions[0],solution_3_elapsed_time,solution_3_count,"Fast recursion");
    //   // }
    //   printf(" %3d %16lu %9.3e |",solution_3_best.n_moves,solution_3_count,solution_3_elapsed_time);
    // }
    // else
    // {
    //   solution_3_best.n_moves = -1;
    //   printf("                                |");
    // }
    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_best.n_moves,&solution_4_best.posi
tions[0],solution_4_elapsed_time,solution_4_count,"Fast Dynamic Non-Recursion");
      }
      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("                                |");
    }
    if(solution_5_elapsed_time < _time_limit_)
    {
      solve_5(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_5.pdf",final_position);

make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_5_best.n_moves,&solution_5_best.posi
tions[0],solution_5_elapsed_time,solution_5_count,"Unitary-Cost Dijkstra Recursion");
      }
      printf(" %3d %16lu %9.3e |",solution_5_best.n_moves,solution_5_count,solution_5_elapsed_time);
    }
    else
    {
      solution_5_best.n_moves = -1;
      printf("                                |");
    }
    if(solution_6_elapsed_time < _time_limit_)
    {
      solve_6(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_6.pdf",final_position);

make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_6_best.n_moves,&solution_6_best.posi
tions[0],solution_6_elapsed_time,solution_6_count,"Speed-Based A* Recursion");
      }
      printf(" %3d %16lu %9.3e |",solution_6_best.n_moves,solution_6_count,solution_6_elapsed_time);
    }
    else
```

```
    {
      solution_6_best.n_moves = -1;
      printf("                                |");
    }
    if(solution_7_elapsed_time < _time_limit_)
    {
      solve_7(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_7_%d.pdf",final_position,n_mec);

make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution_7_best.n_moves,&solution_7_best.posi
tions[0],solution_7_elapsed_time,solution_7_count,"Fast recursion (reverse)");
      }
      printf(" %3d %16lu %9.3e |",solution_7_best.n_moves,solution_7_count,solution_7_elapsed_time);
    }
    else
    {
      solution_7_best.n_moves = -1;
      printf("                                |");
    }
    printf("\n");
    // if (solution_3_best.n_moves != solution_5_best.n_moves ) {
    //   printf("ERRO NO 3º ou 5º ALGORITMO!\n");
    //   return EXIT_FAILURE;
    // }
    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");
  printf("--- + --- ---------------- --------- + --- ---------------- --------- + --- ---------------- -------
-- + --- ---------------- --------- +\n");
  return 0;
# undef _time_limit_
}
```

priorityQueue.h

```
// C code to implement Priority Queue
// using Linked List
#include <stdio.h>
#include <stdlib.h>

// Node
typedef struct node {
         int data;

         // Lower values indicate higher priority
         int priority;

         int speed;

         struct node* next;

} Node;

// Function to Create A New Node
Node* newNode(int d, int p, int speed)
{
         Node* temp = (Node*)malloc(sizeof(Node));
         temp->data = d;
         temp->priority = p;
         temp->speed = speed;
         temp->next = NULL;
```

```
        return temp;
}

// Return the value at head
int peek(Node** head)
{
        return (*head)->data;
}


int peek_Speed(Node** head)
{
        return (*head)->speed;
}

// Removes the element with the
// highest priority from the list
void pop(Node** head)
{
        Node* temp = *head;
        if ((*head)->next != NULL)
                (*head) = (*head)->next;
        free(temp);
}

// Function to push according to priority
void push(Node** head, int d, int p, int speed)
{
        Node* start = (*head);

        // Create new Node
        Node* temp = newNode(d, p, speed);

        if ((*head) == NULL)
        {
                printf("HUH");
                Node* temp = (Node*)malloc(sizeof(Node));
                temp->data = d;
                temp->priority = p;
                temp->speed = speed;
                temp->next = NULL;
                (*head) = temp;
        }
        // Special Case: The head of list has lesser
        // priority than new node. So insert new
        // node before head node and change head node.
        if ((*head)->priority > p) {

                // Insert New Node before head
                temp->next = *head;
                (*head) = temp;
        }
        else {

                // Traverse the list and find a
                // position to insert new node
                while (start->next != NULL &&
                        start->next->priority < p) {
                        start = start->next;
                }

                // Either at the ends of the list
                // or at required position
                temp->next = start->next;
                start->next = temp;
        }
}

// Function to check is list is empty
int isEmpty(Node** head)
{
        return (*head) == NULL;
}
```

Linguagem BASH

test_nmecs.sh

```
#!/bin/bash

for n in {1..200000}
do
    echo -ne "\rA testar para $n..." | sed -e "s/12345678/${replace}/g"
    ./speed_run $n
    if [ $? == 1 ]; then
        echo "ERRO para $n"
        exit 1
    fi
done
echo ""
```

Linguagem MATLAB

graphs_sol_1and2.m

```
tempos = load("tempos_sol1.txt");
racios = zeros(1,length(tempos) - 1);

for i=1:length(tempos)-1
    racios(i) = tempos(i+1)/tempos(i);
end
media_tempos = 0;
for i=1:length(tempos)-1
    media_tempos = media_tempos + racios(i);
end
media_tempos = media_tempos/(i);

format long;
counts = load("counts_sol1.txt");
format long;

racios = zeros(1,length(counts) - 1);

for i=1:length(counts)-1
    racios(i) = counts(i+1)/counts(i);
end
media_counts = 0;
for i=1:length(counts)-1
    media_counts = media_counts + racios(i);
end
media_counts = media_counts/(i);

n = 1:1:length(counts);
n2 = 1:1:length(counts);

f1 = media_tempos.^n/100000000;
f2 = media_counts.^n2;
figure(1);
plot(n2,tempos,".-r",'MarkerSize',15,'LineWidth',1.4)
title("Very bad recursion - Times");
xlabel("n");
ylabel("time (s)");
legend("Sem nº mec");
figure(2);
plot(n2,counts,".-r",'MarkerSize',15,'LineWidth',1.4)
hold on
plot(n2,f2,"--k",'MarkerSize',15,'LineWidth',1)
hold off
title("Very bad recursion - Counts");
xlabel("n");
ylabel("counts");
legend("Sem nº mec","Ratios approx.");
```

graphs_sol_3to6.m

```
% Para os diferentes métodos, apenas mudamos o número do ficheiro (por exemplo: "sol_3_..." -> "sol_4_...") e
o título do gráfico
times108122 = load("sol_3_times_108122.txt");
times110056 = load("sol_3_times_110056.txt");
counts108122 = load("sol_3_counts_108122.txt");
counts110056 = load("sol_3_counts_110056.txt");

%% Times
figure(1);

skip = 50;
x = 1:skip:length(counts108122);

plot(x,times108122(x),".-r",'MarkerSize',15,'LineWidth',1)
hold on

plot(x,times110056(x),".-b",'MarkerSize',15,'LineWidth',1)

title("Fast Recursion - Times");
xlabel("n");
ylabel("time (s)");
legend("108122","110056");

%% Counts
figure(2);

skip = 200;
x = 1:skip:length(counts108122);

plot(x,counts108122(x),".-r",'MarkerSize',15,'LineWidth',1)
hold on
linearAprox = polyfit(x,counts108122(x),1);
f = polyval(linearAprox,x);
plot(x,f,"--k",'LineWidth',0.5)

plot(x,counts110056(x),".-b",'MarkerSize',15,'LineWidth',1)
hold on
linearAprox = polyfit(x,counts110056(x),1);
f = polyval(linearAprox,x);
plot(x,f,"--k",'LineWidth',0.5)

title("Fast Recursion - Counts");
xlabel("n");
ylabel("counts");
legend("108122","Linear aprox.","110056");
```