PROBLEMA DE LA ESTACION DE ITV Y SOLUCION PROPUESTA

Para empezar, se recuerda el planteamiento del problema a resolver. Suponemos una estación de ITV que consta de *m* líneas de inspección de vehículos iguales. Hay un total de n vehículos que necesitan inspección que, en función de sus características, tardarán en ser inspeccionado un tiempo t_i donde i = 1, . . . , n. El objetivo es encontrar la manera de atender a los n vehículos y acabar en el menor tiempo posible.

Se pide que se utilice un algoritmo de vuelta atrás y lo primero es decidir qué mostrar y cómo. Obviamente, algo a satisfacer obligatoriamente es lo principal que se nos pide: cómo atender los coches para que ocupen el menor tiempo posible; pero para esto hay diferentes alternativas. Por ejemplo:

- Mostrar un vector que contenga los índices de los coches ordenados según su acceso a una línea libre.

	0	**1**	**2**	**3**	**…**	**n**
Coches	5	3	2	0	…	7

Que representaría que el coche con índice 5 entra primero, el 3 entra el siguiente, seguido del 2, etc.

- Mostrar un vector que contenga la posición en el que van a ir entrando vehículos en las líneas

	0	**1**	**2**	**3**	**…**	**n**
Orden	5	3	2	0	…	7

Que representa que el coche 0 será el quinto en entrar en una línea libre, que el coche 1 iría tercero y el coche 2 pasaría a ser atendido en la segunda línea libre que se encuentre, etc.

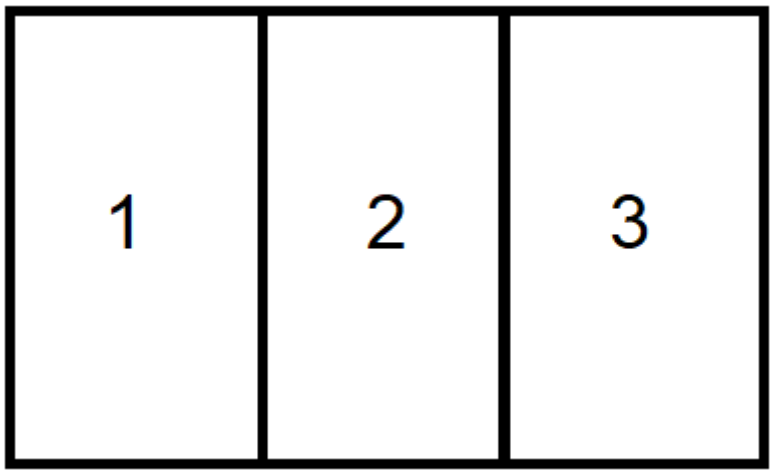
- Línea en la que será atendido cada coche

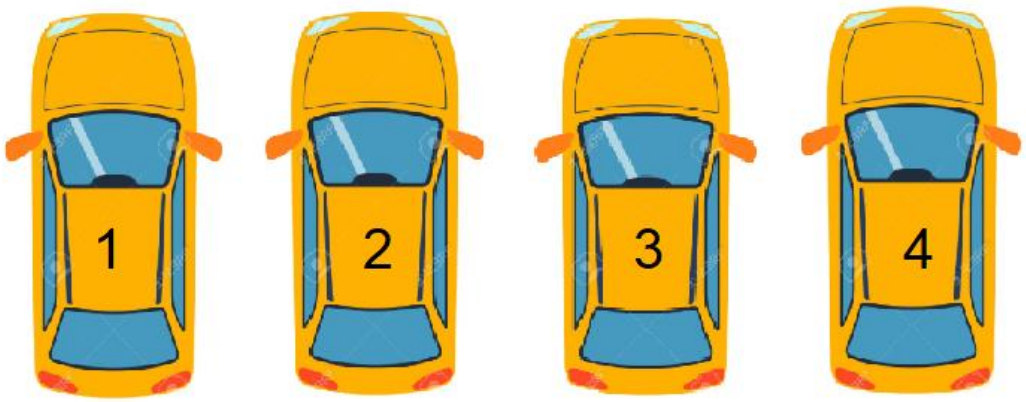
	0	**1**	**2**	**3**	**…**	**n**
Línea	1	0	0	2	…	1

Donde se daría a entender que el coche 0 entraría en la línea 1, el coche 1 en la línea 0, etc.

De todas estas representaciones planteadas para la solución al problema (ni mucho menos son todas las que hay), se ha escogido aquella que muestra el orden en el que accederá cada coche a la línea vacía que esté disponible antes. Este vector es el que se mostrará por pantalla, y para explicar mejor lo que se pretende representar, ahí va un ejemplo gráfico.

Suponiendo una estación de ITV con 3 líneas y 4 coches, tendríamos esta situación:





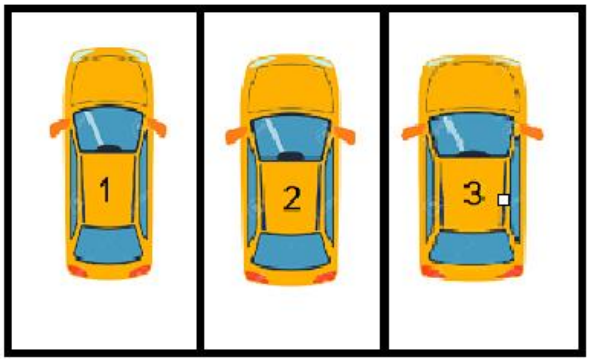
Existen 24 combinaciones diferentes en las que ordenar los coches, pero esta técnica seguiría el siguiente procedimiento. Supongamos que los tiempos que tardan en ser inspeccionados cada uno son los siguientes:

COCHES	1	2	3	4
TIEMPOS	2	1	3	4

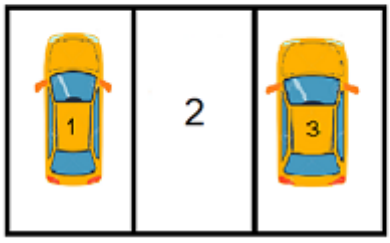
Una posible combinación (y la primera que comprobaría) sería meter el siguiente orden:

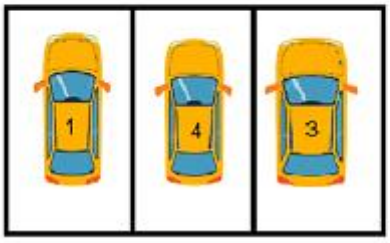
{1, 2, 3, 4}

Lo cual llevaría meter en las 3 líneas vacías del principio los coches 1, 2 y 3 y dejar el 4 para cuando una línea esté libre. Esa representación es la que se ha pretendido mostrar con el vector seleccionado como solución. Ese ejemplo se vería así:

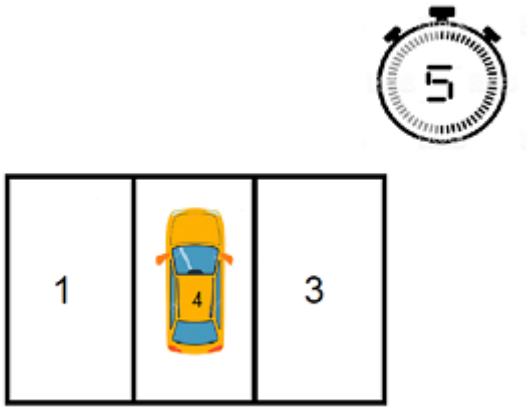


Tras una unidad de tiempo, el coche 2 terminaría y deja el hueco libre, entrando el 4 en el mismo momento:

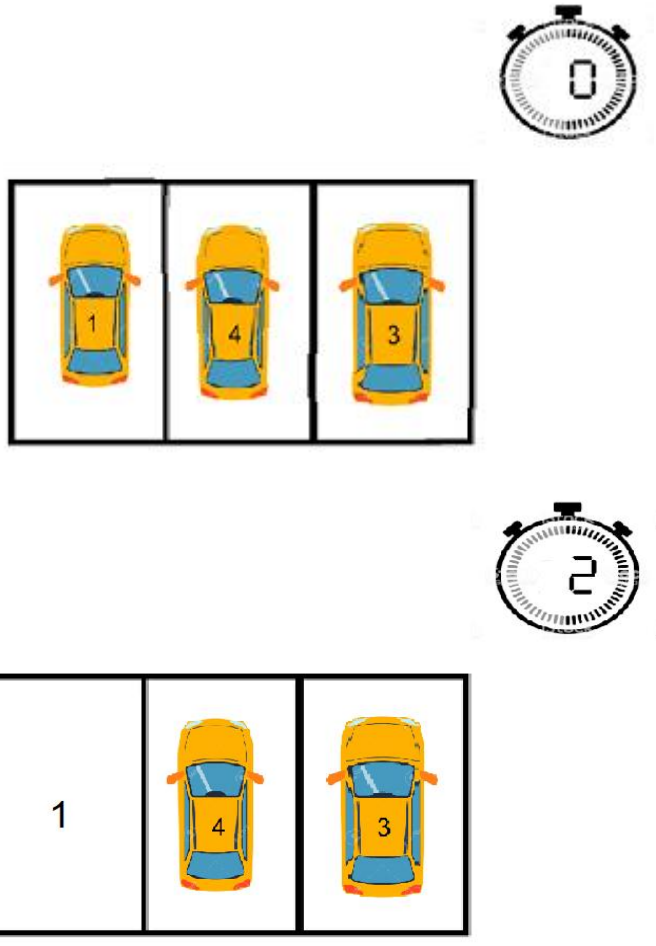


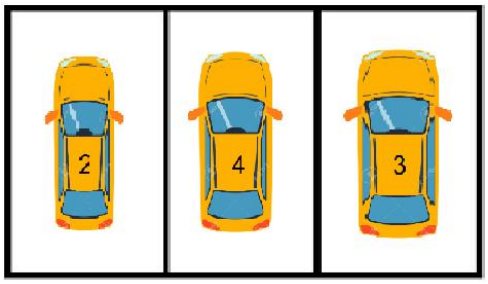


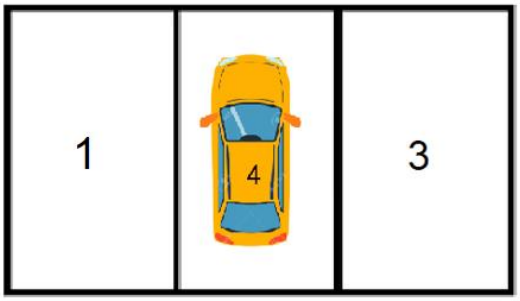
El coche 1 se iría en el instante 2 (que es lo que dura su inspección) y ya en el instante 3 abandona el coche 3. De esta forma, el 4 queda hasta que se cumplen 4 instantes desde que entró:

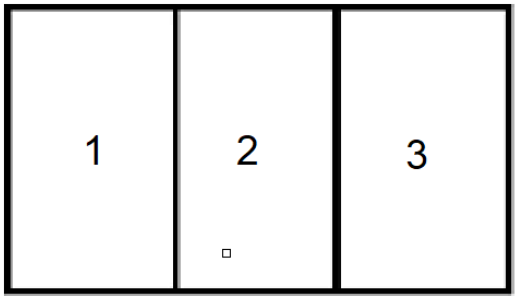


Este orden hace que, en 5 instantes, todos los coches hayan sido atendidos, pero si seguimos comprobando vemos que existen algunas combinaciones que hacen que se consiga la inspección de todos los vehículos en menos tiempo. Para hacerlo más rápido se muestra la secuencia de imagen únicamente:

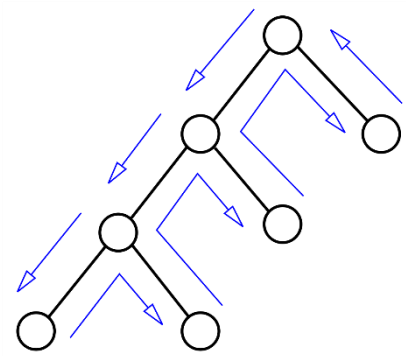








Como se ha podido observar fácilmente, el orden {1, 4, 3, 2} tarda menos (4 unidades de tiempo) en realizar la misma tarea. Estas comprobaciones se realizan con backtracking, siguiendo el algoritmo que se explica a continuación. La facilidad con la que se ve la solución conociendo el orden y lo irrelevante que es conocer y almacenar la línea en la que entra un vehículo concreto, es lo que nos ha llevado a elegir esta representación de la solución.



Nuestra solución propuesta pasa por partir de una etapa 0, que muestra la profundidad del árbol de soluciones por la que vamos comprobando, e ir llamando recursivamente a etapas posteriores hasta haber probado todas las combinaciones válidas. Se recalca que un camino tiene que ser válido por la existencia de una cota que evita continuar haciendo comprobaciones sobre una rama que se sabe que ya es peor opción que la que tenemos hasta ahora. De esta forma, la cota se podría obtener de muchas maneras, incluso utilizando greedy para una primera solución y, a partir de esta, usarla como comparación para el resto. Sin embargo, en nuestro código, este valor es obtenido del cálculo de la primera combinación completa. Se establece un valor insuperable, el valor máximo que puede contener un int, para asegurarnos de que, al comparar este valor con el tiempo obtenido con el primer camino, si o si el del camino va a ser menor o igual, por lo que pasa a ser nueva cota. A raíz de esto, por cada etapa en la que avancemos (avanzar una etapa es asignar un orden a un coche nuevo hasta tenerlos todos asignados) vamos comprobando si es válido continuar por aquí o no. Partiendo de la etapa 0 y sin combinaciones aún calculadas, se probaría si meter el coche 0 como primero de esta lista es válido. Inevitablemente será correcto pues la inspección de un solo coche no puede ser mayor que la cota inicial (MAX_INT) ni que ningún camino completo que hayamos obtenido. Al ser correcto, “desbloquea” la siguiente etapa y se pasa a probar si meter primero el 0 y luego el 1 es válido, calculando el tiempo que tardarían estos dos vehículos en ese orden concreto y comparándolo con la cota. Cuando se llega al caso base en el que la etapa por la que voy comprobando es igual al número de coches (se habrán ordenado todos y ningún tiempo habrá dado superior a la cota) se comprueba si el tiempo obtenido con esa combinación ha llegado hasta ahí por ser menor o por ser igual (de haber sido mayor no habría llegado hasta este caso). Si es menor que el que tenemos, esta solución nueva pasa a ser la “definitiva” por ahora y si no, tardarán lo mismo y no renta perder tiempo reasignando la solución.

Si ya se tiene un camino seleccionado temporalmente (supongamos de un total de 5 coches, haber ordenado correctamente los 5), y comprobando otra solución por la 3º etapa (ordenar 3 de ellos), ya vemos que el tiempo es peor que el que se tarda con ese reparto de 5 coches, se evita comprobar todas y cada una de las combinaciones que empezaban por estos 3 vehículos ordenados de esta forma. Esto reduce considerablemente la cantidad de comprobaciones que se hacía con PD.

La comprobación de la validez de un camino se realiza de la siguiente manera. Se crea un vector que contendrá tantas posiciones como líneas de inspección haya y se inicializan a un número concreto (0 o algún negativo que representen una posición vacía). Por cada etapa, se asigna el coche a comprobar en la línea que sea mejor opción que el resto. Mientras la etapa por la que vamos sea menor que el número de líneas de inspección, obviamente nos limitaremos a introducir estos coches en las líneas vacías y almacenar en el vector **el tiempo que tardaría ese coche que estamos asignando.** De esta forma, cuando el coche que se está comprobando es el siguiente a aquel que ocupó la última línea, se busca de entre todas las posiciones del vector, aquella que tiene el número más bajo (lo cual significará que el coche que será atendido en esa línea será el que menos tarde en salir y, por tanto, la mejor opción

donde meter el siguiente para no hacerle esperar y dejar perder instantes de tiempo vacíos). Al encontrar la línea en la que se atendería este coche, sumas el tiempo que tardaría este en completar su inspección al valor que ya tenía la línea. Tras esto, buscas el valor más grande del vector y comparas este con la cota. Ese valor más alto no significa otra cosa más que el tiempo que tardaría la estación en dar por atendidos a todos los coches, pues es la línea que más va a tardar en estar vacía. La función devuelve un booleano que te indica si seguir por ese nodo o todos sus hijos han dejado de ser una opción viable.

CASOS DE EJECUCION

En la carpeta de códigos se encuentran varios .cpp destinados a objetivos variados. El estándar llamado “ITV.cpp” es el que muestra por pantalla soluciones al problema, con toda la información necesaria limpia y ordenada; el archivo “ITV_tiempos.cpp” es aquel al que se le han suprimido todas las salidas por pantalla y se ha mantenido únicamente la muestra del número de coches y el tiempo de ejecución (facilita el uso de scripts que saquen .dat para la eficiencia empírica en gnuplot y es más rápido sin los *cout*) y por último, el código “ITV_generador.cpp” que genera datos adecuados para el uso de estos en cualquier problema que siga la estructura de este, sin tener en cuenta funciones de cálculo ni reparto. La compilación de todos estos es automática con la sentencia *make*, pero la forma manual de llevarla a cabo es:

g++ -std=c++11 <*ARCHIVO*>.cpp -o <*EJECUTABLE*>

Y su ejecución:

./<EJECUTABLE> [nº coches] [nº lineas]

*

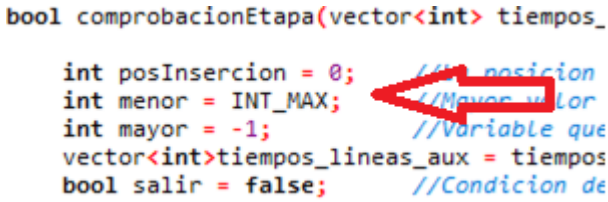
ACLARACION: EL VALOR INT_MAX DE LA LINEA:

78: ITV.cpp

260: ITV_tiempos.cpp

DEBE SER CAMBIADO POR INT8_MAX EN UBUNTU

*

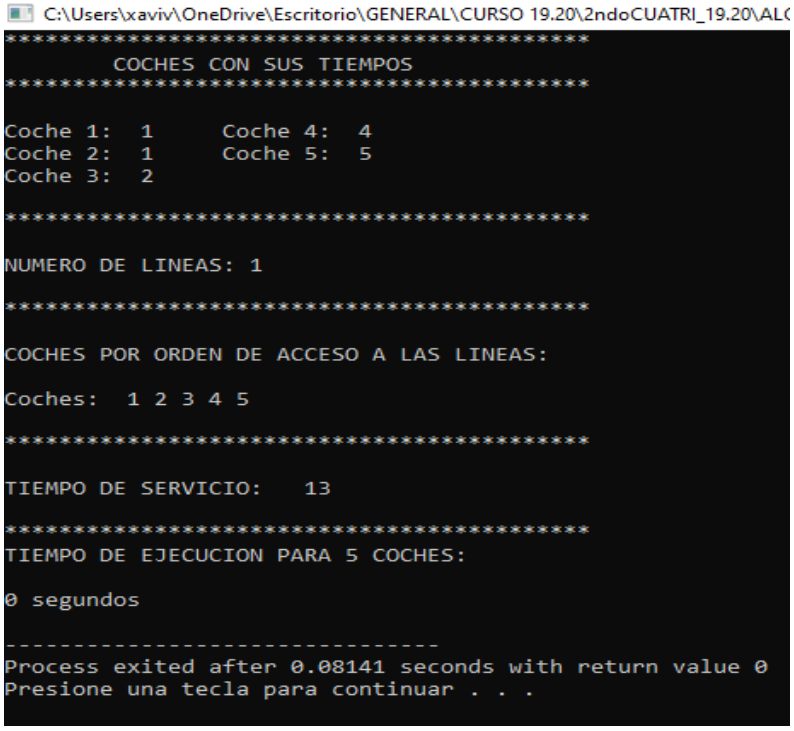


A continuación, se comprueban varios casos que simulen situaciones lo más variadas posible y observar el comportamiento del programa ante la mayoría de contextos que puedan darse.

El apartado de las ejecuciones que se refiere al orden de acceso de los coches está referido a qué vehículo será el siguiente que ocupará la primera línea libre que encuentre. Resuelve así el problema planteado de conocer cómo atender a todos los vehículos para acabar en el menor tiempo posible.

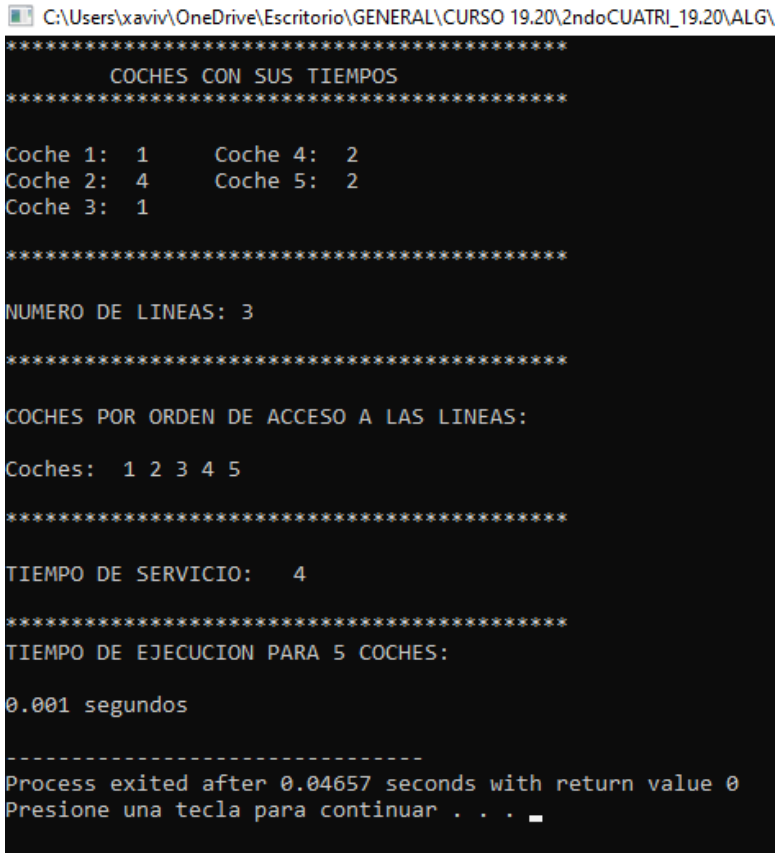
EJEMPLO 1: Nº COCHES: 5 // Nº LINEAS: 1

En este caso vamos a probar a introducir 5 coches a repartir en una única línea de inspección. Este caso es algo extremo, pero es una situación que puede darse en la vida cotidiana. Para estos casos, obviamente el tiempo que tardarán todos los coches en ser atendidos será la suma de cada uno de sus tiempos, pues al haber una única línea estos irán pasando uno tras otro en la misma línea hasta terminar. El orden para introducir los coches es siempre el mismo, uno secuencial y ordenado, pues se queda con la primera mejor solución que se encuentra y no hay opción de obtener menor tiempo que haciendo pasar a todos los coches por la misma línea.



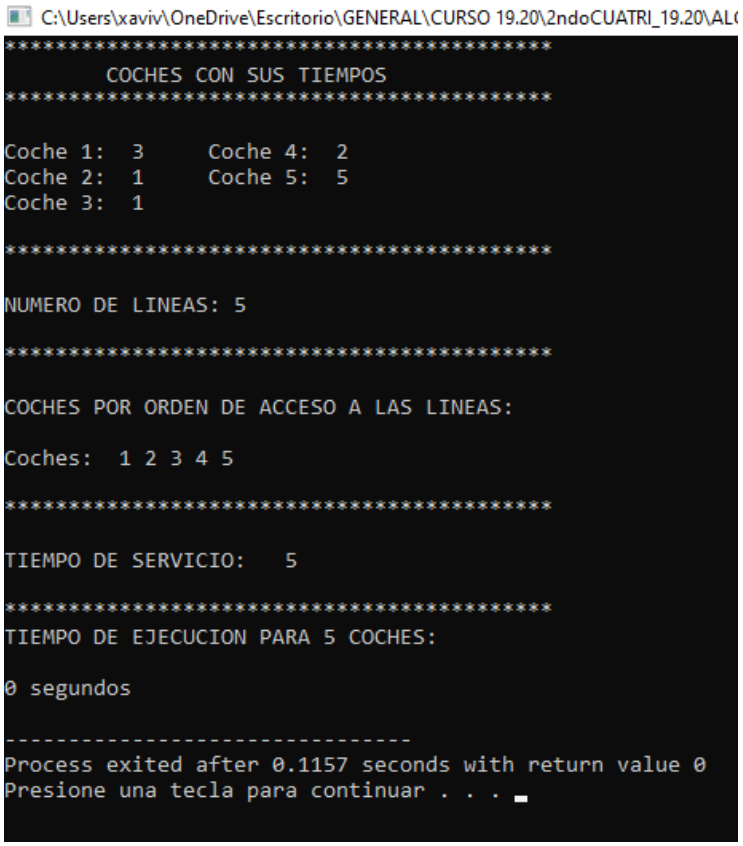
EJEMPLO 2: Nº COCHES: 5 // Nº LINEAS: 3

Esta vez el ejemplo simula una situación algo mas común. Hay 5 coches a repartir en 3 líneas de inspección. En este caso, el tiempo y el orden dependerá del factor aleatorio que determina las características del coche.



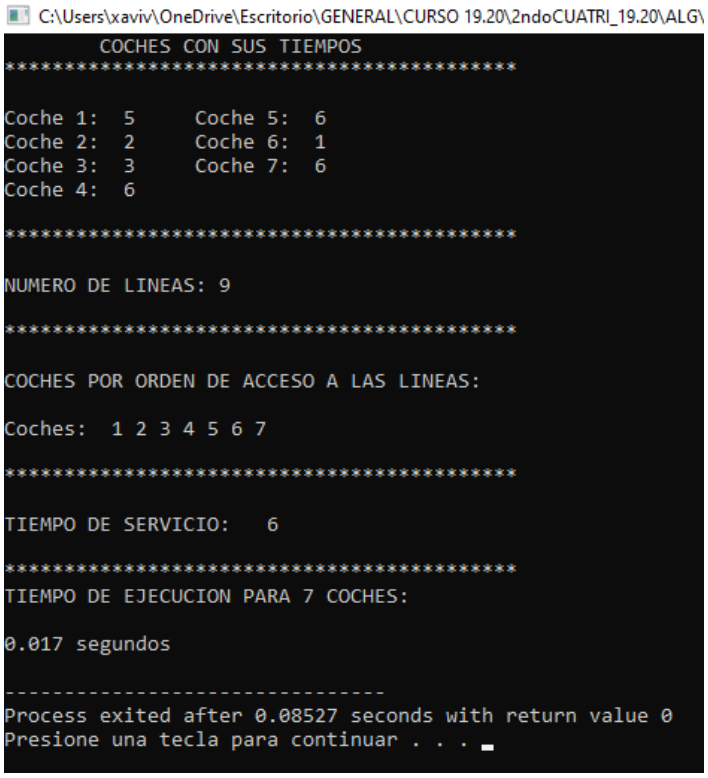
EJEMPLO 3: Nº COCHES: 5 // Nº LINEAS: 5

Aquí observamos otra de esas situaciones “especiales” en la que el número de líneas y de vehículos es la misma. Al igual que si hubiese una sola línea, el tiempo siempre es el mismo, y corresponde al tiempo que tarde el coche más lento. Se debe a que estos datos permiten atender a todos los coches a la vez, uno en cada línea, pero hasta que no termine el más lento no se podrán dar todos por atendidos. En el caso siguiente, el coche más lento tarda 5 unidades, y precisamente el mejor reparto (también en orden porque entran todos a la vez) obtiene un tiempo total de 5 unidades.



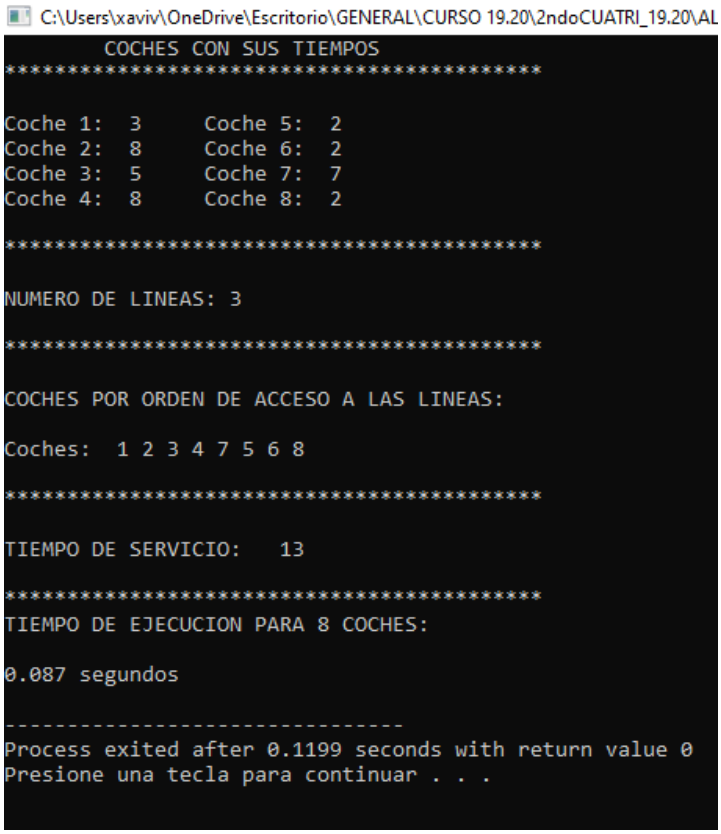
EJEMPLO 4: Nº COCHES: 7 // Nº LINEAS: 9

En este otro ejemplo se plantea una situación que no se ha dado hasta ahora: el caso en el que hay más líneas que vehículos a atender. A efectos prácticos ocurre lo mismo que si coinciden el número de vehículos y de líneas. Todos los coches van a ser inspeccionados a la vez (y aún habrá líneas por ocupar) y no se darán todos por terminados hasta que no termine el más lento de todos. En el ejemplo de la imagen, ocurre con los coches 4, 5 y 7 en los que el tiempo de inspección es 6, el más alto de ellos.



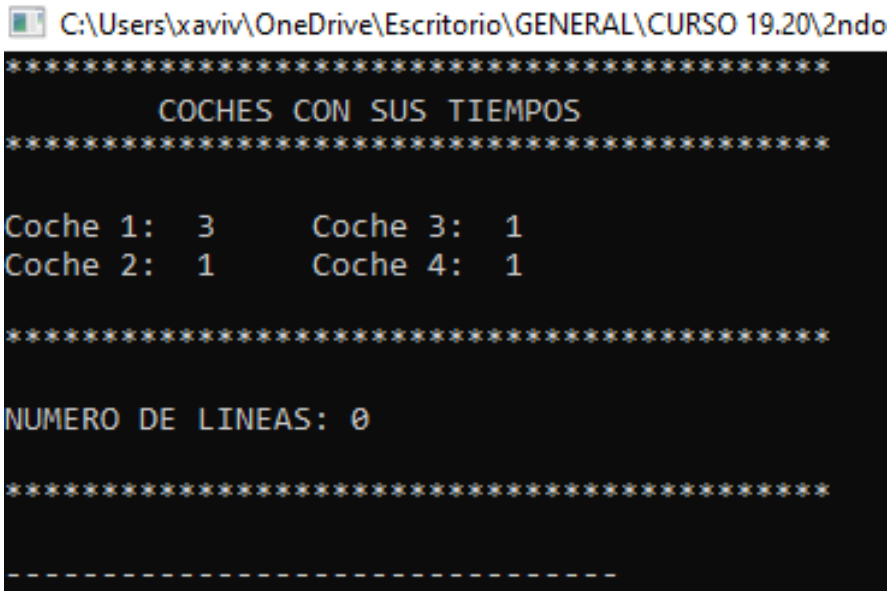
EJEMPLO 5: Nº COCHES: 8 // Nº LINEAS: 3

Otro caso estándar en el que tenemos varios coches y un número de líneas de inspección menor.



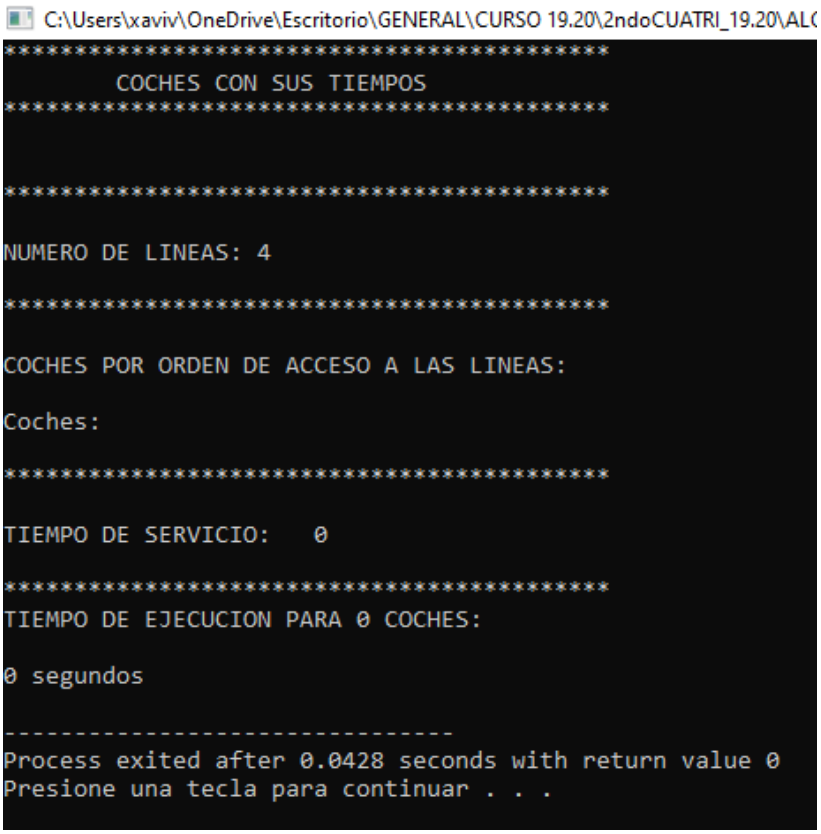
EJEMPLO 6: Nº COCHES: 4 // Nº LINEAS: 0

Por último, dos casos que serían casi imposibles o lo suficientemente básicos como para no necesitar algoritmo alguno, pero que permite ver mejor el funcionamiento de nuestro algoritmo en todos los casos. En este, suponer que no existen líneas de inspección.



EJEMPLO 7: Nº COCHES: 0 // Nº LINEAS: 4

Y otro en el que no hay coches a los que atender



Estudio empírico de la eficiencia

Antes de entrar a explicar el estudio empírico de la eficiencia de este código, hay que tener en cuenta que se trata de un problema que cuenta con m líneas de inspección y un total de n vehículos que tardan un tiempo t_i en realizar la inspección. Los dos primeros valores representan los parámetros del programa, pues el primer parámetro es el número de coches para inspeccionar y el segundo el número de líneas para ello, mientras que el tiempo que se le otorga a esos vehículos es un número entero aleatorio que puede ser 1..n, por lo que si hay 5 coches, podemos encontrar los siguientes valores: 1,2,3,4,5 como tiempo que tardan en realizar la inspección. Es por ello por lo que es necesario contrastar diferentes tiempos para un número diferente de coches, así como de líneas.

En nuestro caso, hemos realizado las siguientes comprobaciones de tiempo:

·6 coches fijos y el número de líneas que incrementa su valor de 1 a 11

·7 coches fijos y el número de líneas que incrementa su valor en cada iteración de 1 a 11

·Mismo casos que los anteriores, pero con 8, 9 y 10 coches fijos

·3 líneas de inspección fijas y el número de coches que incrementa de 1 a 11 en cada iteración

·5 líneas de inspección fijas y el número de coches que incrementa de 1 a 11 en cada iteración

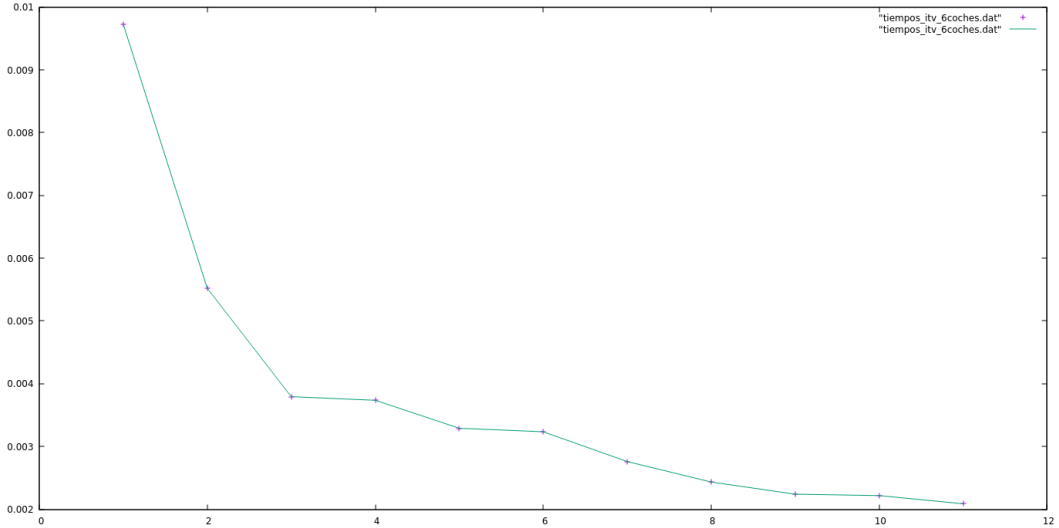
·Mismos casos que los anteriores, pero con 7, 9 10 y 11 líneas de inspección fijas

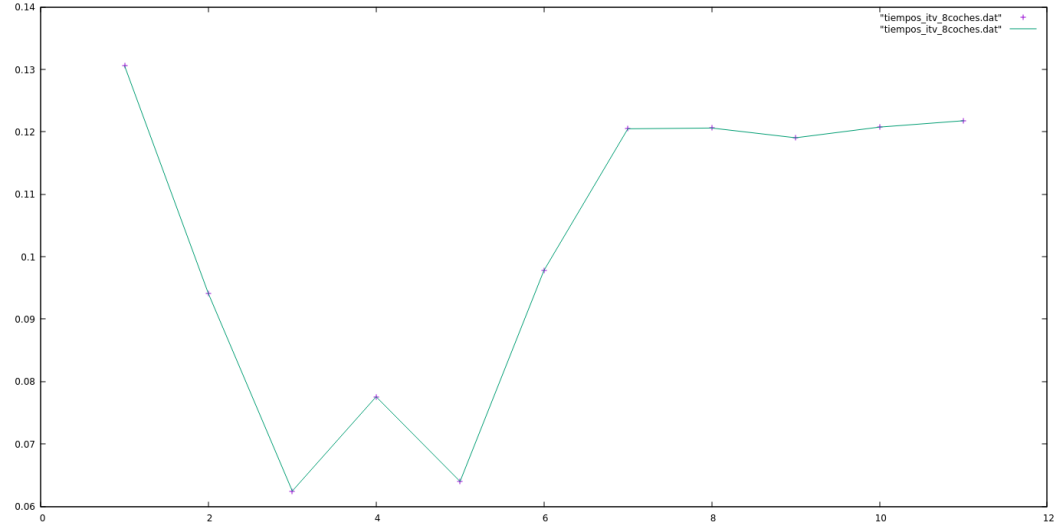
Para realizar un ajuste de los datos con una función teórica, hemos tenido en cuenta que nuestro algoritmo tiene una eficiencia $\in O(n!)$ ya que es un código realizado por medio de la técnica de backtracking y al realizarse una búsqueda exhaustiva y sistemática en el espacio de soluciones suele resultar muy ineficiente. En el mejor de los casos, backtracking podría generar sólo O(n) nodos, aunque podría llegar a tener que explorar el árbol completo de espacio de estados del problema. Sin embargo, en el peor caso, si el número de nodos es 2n o n! (como es nuestro caso) el tiempo de ejecución del algoritmo backtracking será generalmente de orden $O(p(n)2^n)$ u O(q(n)n!) respectivamente, con p y q polinomios en n.

Este ajuste teórico solamente se ha podido realizar en los casos en los que hemos dejado un número fijo de líneas ya que dependemos del número de coches y estos entran al número de líneas que existe, pero en el caso de que el número de coches es fijo, depende de la aleatoriedad de sus valores de tiempo lo que nos determina el tiempo que va a durar esa ejecución para un número de líneas cambiantes.

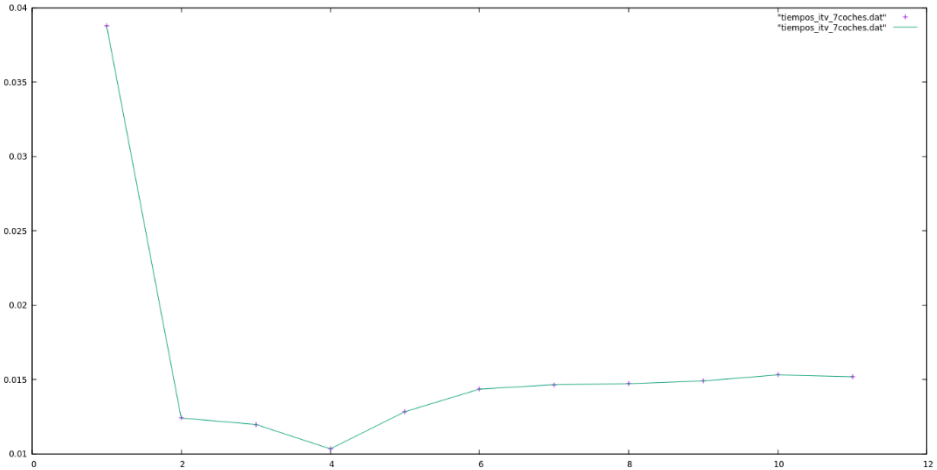
Además, hemos compilado el código con las optimizaciones -O1, -O2 y -O3 y hemos comprobado qué mejora puede existir para los distintos casos de ejecución, aunque para esta ocasión y para no hacer un estudio tan largo, hemos reducido los ejemplos para 7 y 10 coches fijos y para 7 y 10 líneas de inspección fijas, teniendo para cada caso el otro parámetro incrementando su valor de 1 a 11 en cada iteración de la ejecución.

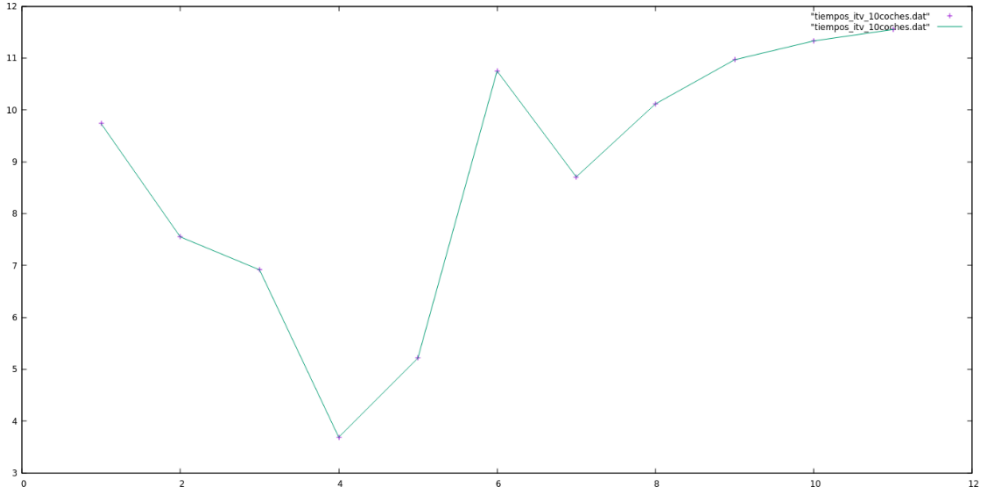
6 coches fijos y líneas cambiantes (8 para la foto de debajo)

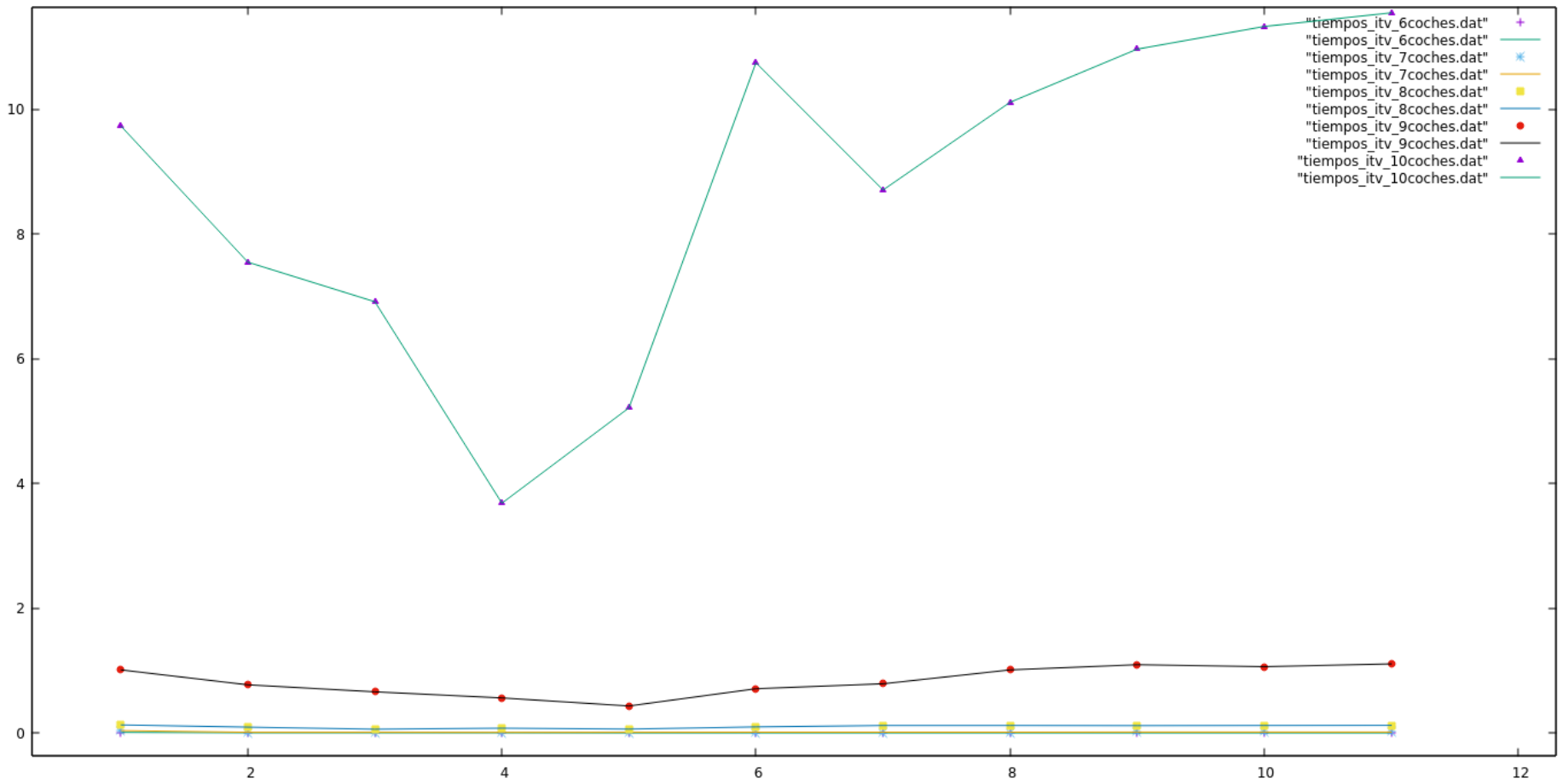




7 coches cambiantes y líneas cambiantes (10 para la foto de debajo)

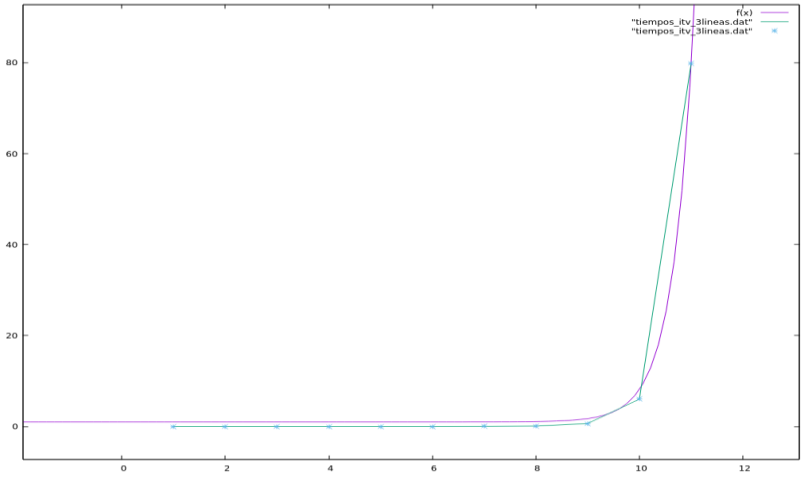


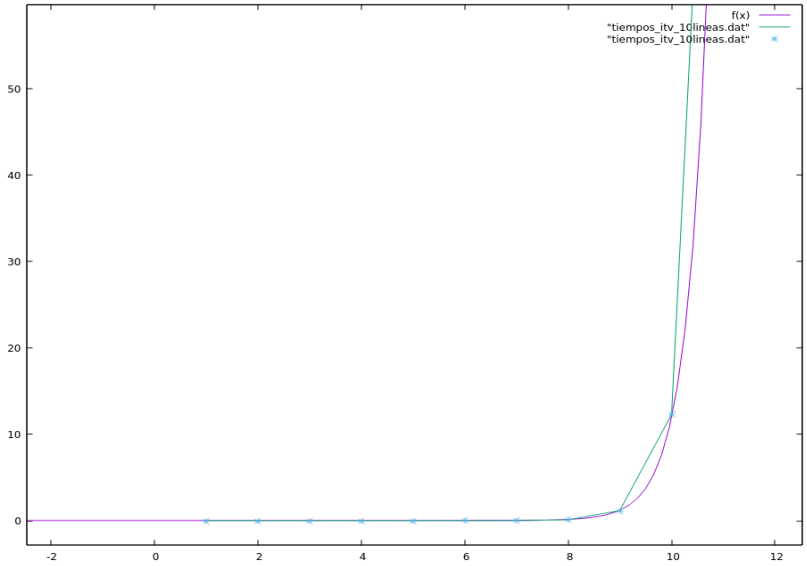




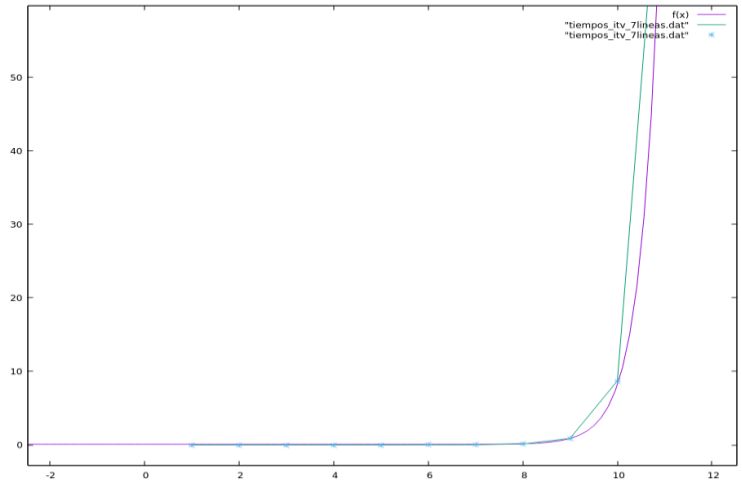
En esta foto vemos todos los casos posibles que comentamos anteriormente, dejando un número fijo de coches y el número de líneas de inspección incrementa de 1 a 11, es por lo que cuando tenemos 10 coches independientemente del número de líneas que haya, el número de comprobaciones que se realiza por medio de backtracking es exponencial a diferencia de lo que ocurre cuando tenemos 6, 7, 8 o incluso 9 coches fijos.

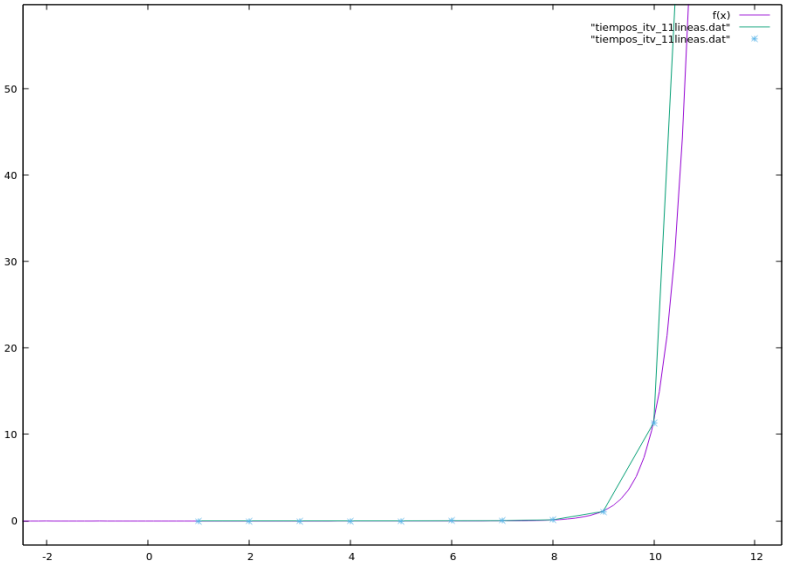
3 líneas fijas y número de coches cambiantes (10 para la foto de debajo)

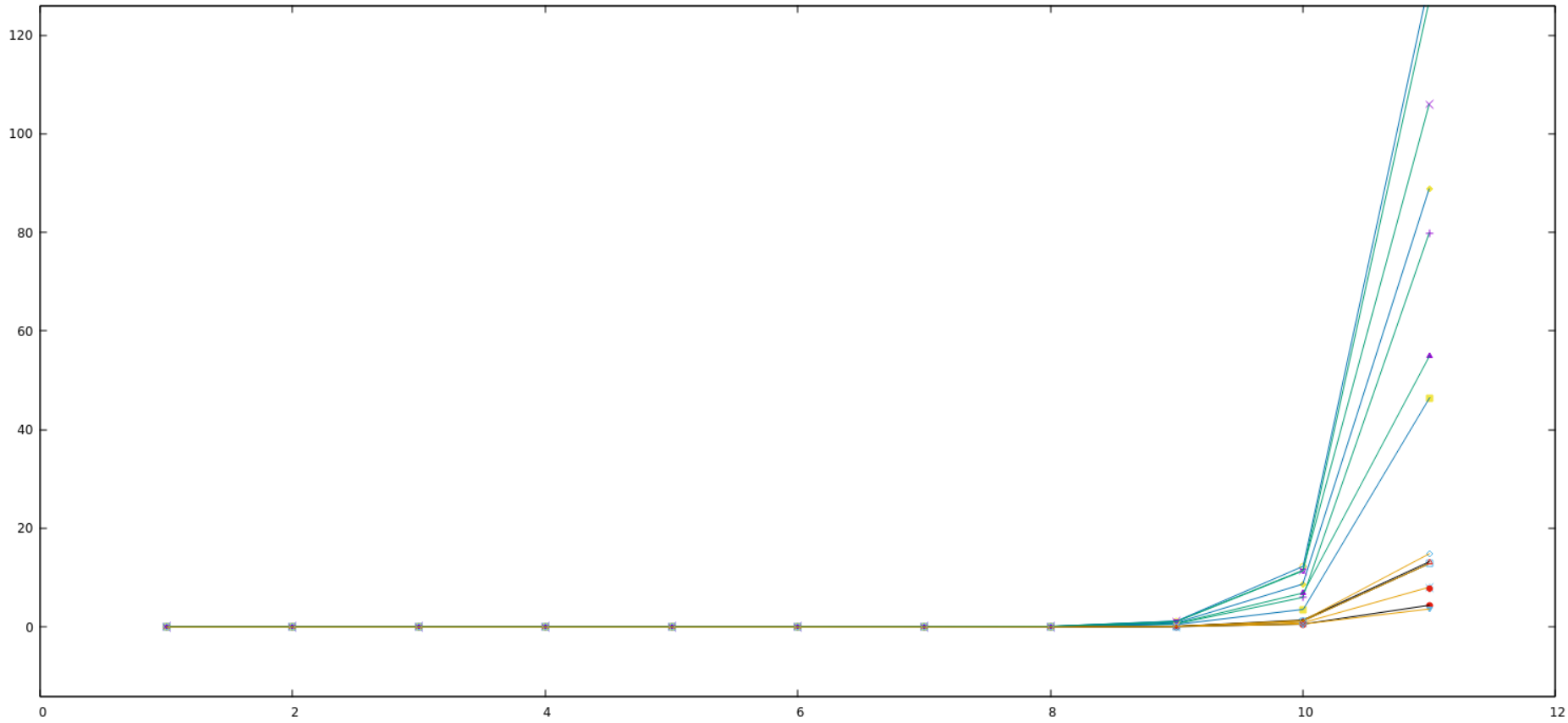




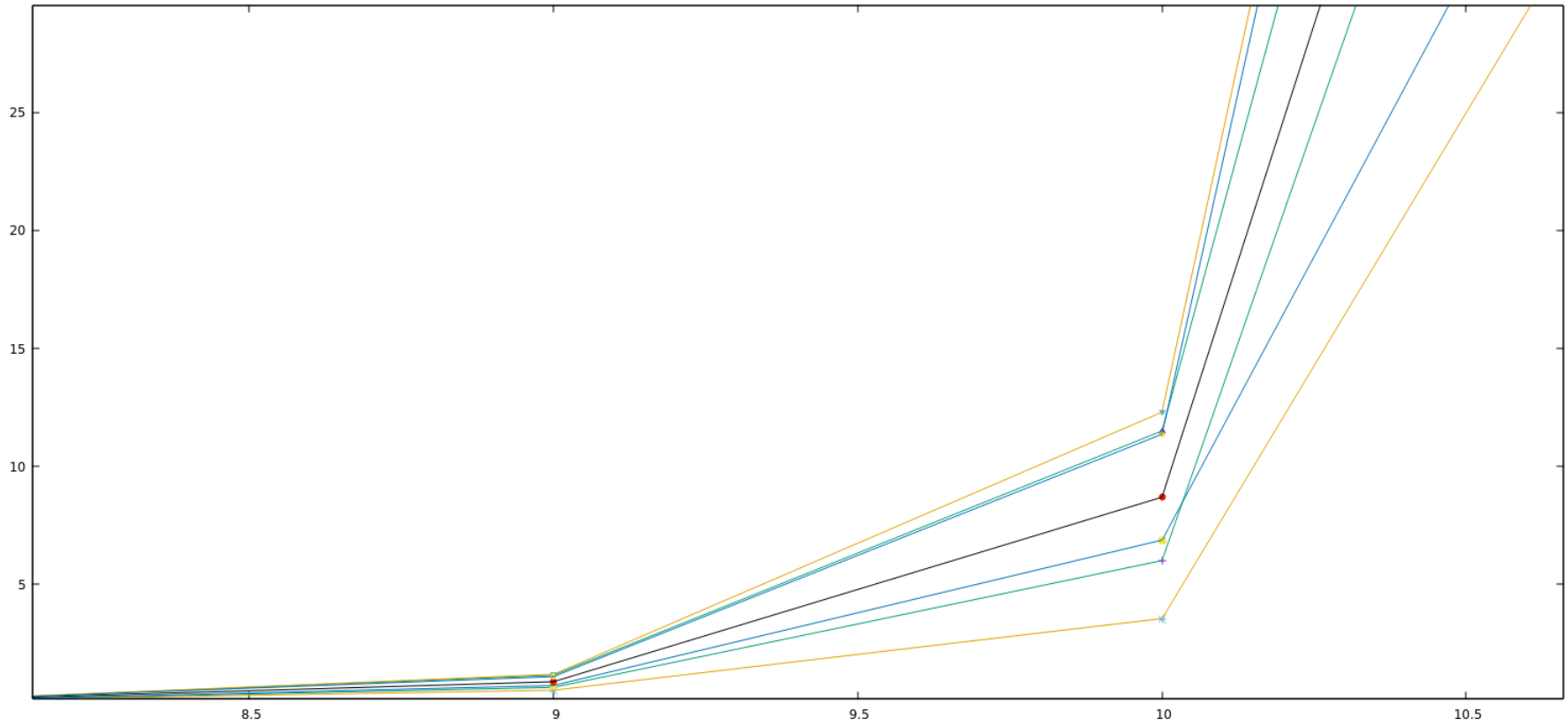
7 líneas fijas para la foto de arriba y 11 para la foto de debajo





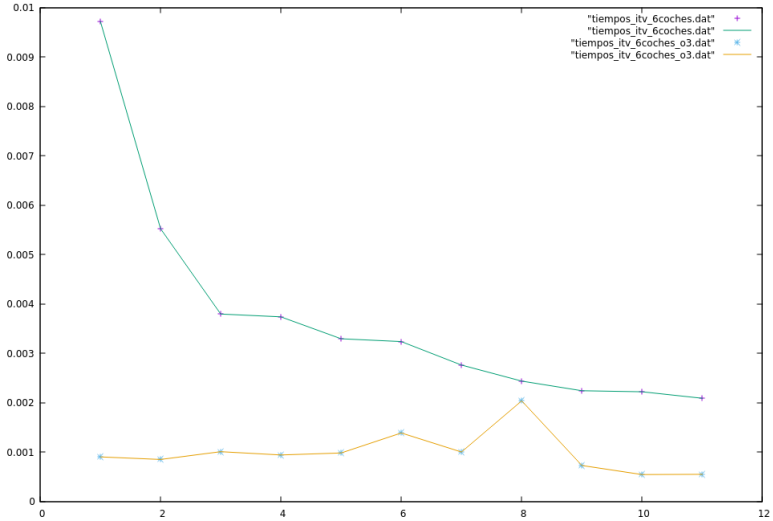


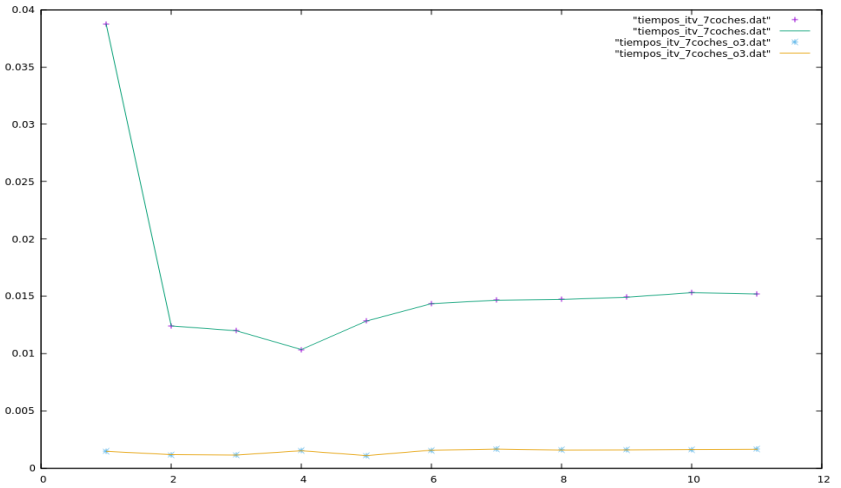
Como hemos visto en la foto anterior, la función teórica de a*n!+b ofrece un gran ajuste para nuestro resultados y a medida que agranda el número de líneas fijo obtenemos puntos más cercanos entre sí. Para el caso de dejar el número de líneas de inspección fijo, a medida que el número de coches aumenta (obteniendo valores aleatorios) podemos notar que el número de comprobaciones aumenta de forma que el tiempo de ejecución también lo hace generando gran diferencia entre un caso y otro. De abajo hacia arriba este parámetro aumenta, pero para verlo aún más claro vamos a agrandar esta zona y explicar qué ocurre.

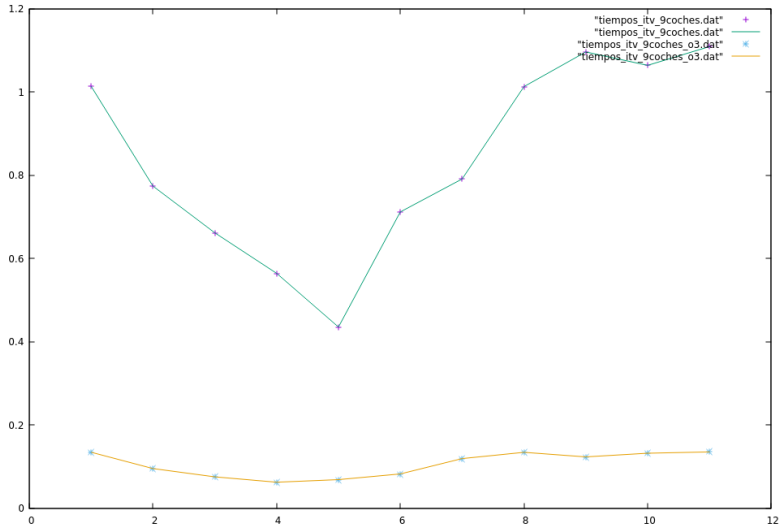


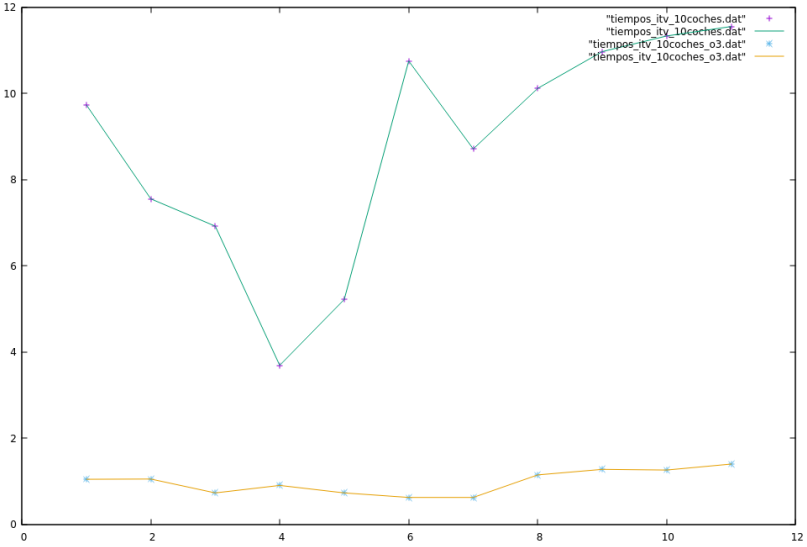
Para ver qué está ocurriendo es necesario agrandar la zona en la que se unen las líneas para poder observar que cuando tenemos un número fijo de líneas a medida que agranda el número de coches (esto ocurre en la gráfica de abajo hacia arriba) el tiempo de la ejecución crece muy rápido. Vemos como la línea verde (tercera empezando por abajo en la gráfica) que tiene 4 líneas de inspección fijas y en ese punto 10 coches, su tiempo para la última iteración supera al caso en el que existen 5 líneas y 10 coches a pesar de que de forma general podemos pensar que debería ser al revés. Esto ocurre por la aleatoriedad que se genera en el tiempo que se le asocia a cada coche.

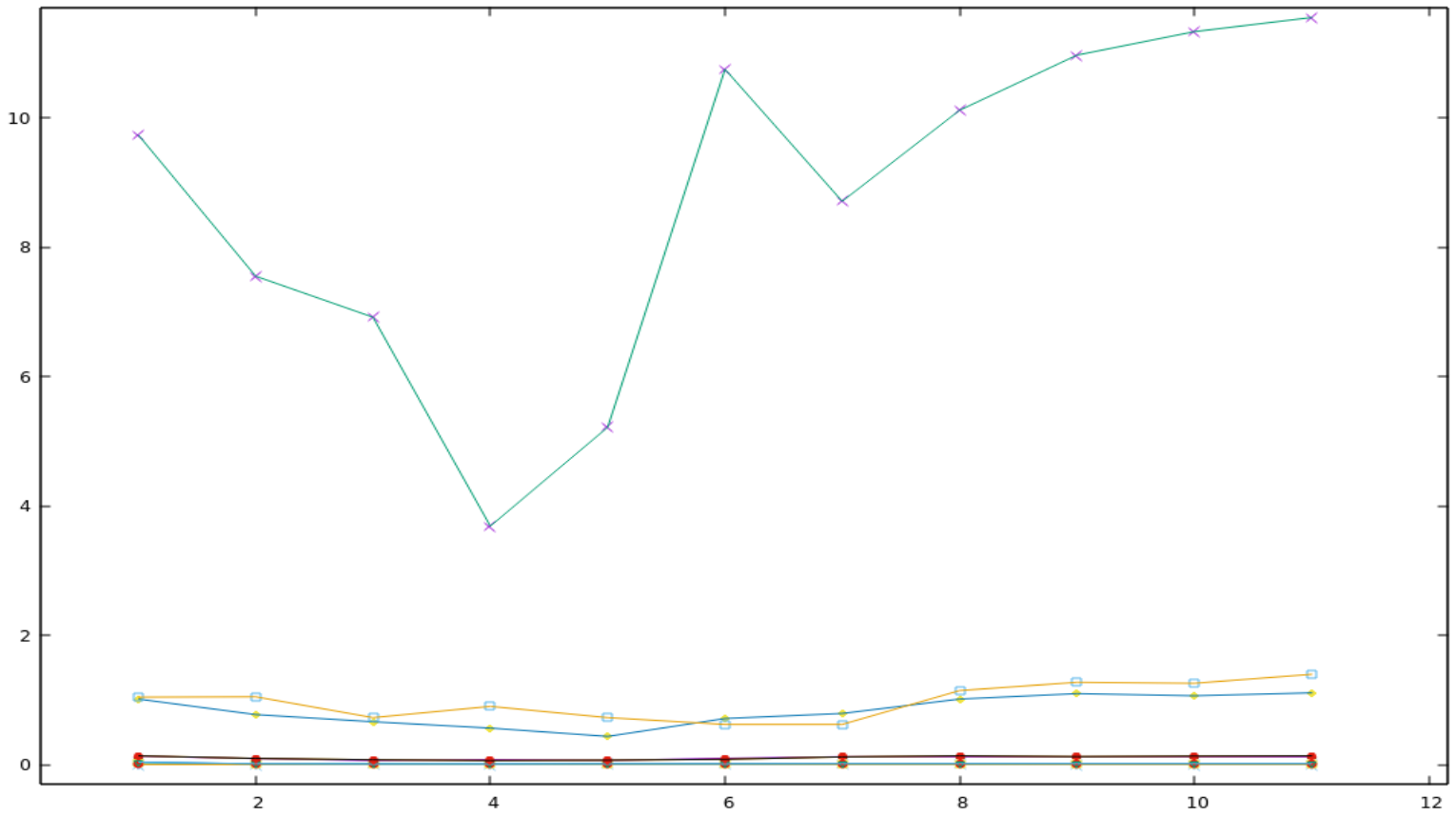
6, 7, 9 y 10 coches fijos (de izquierda a derecha respectivamente) comparando para cada caso su resultado con compilación normal y optimizada con -O3



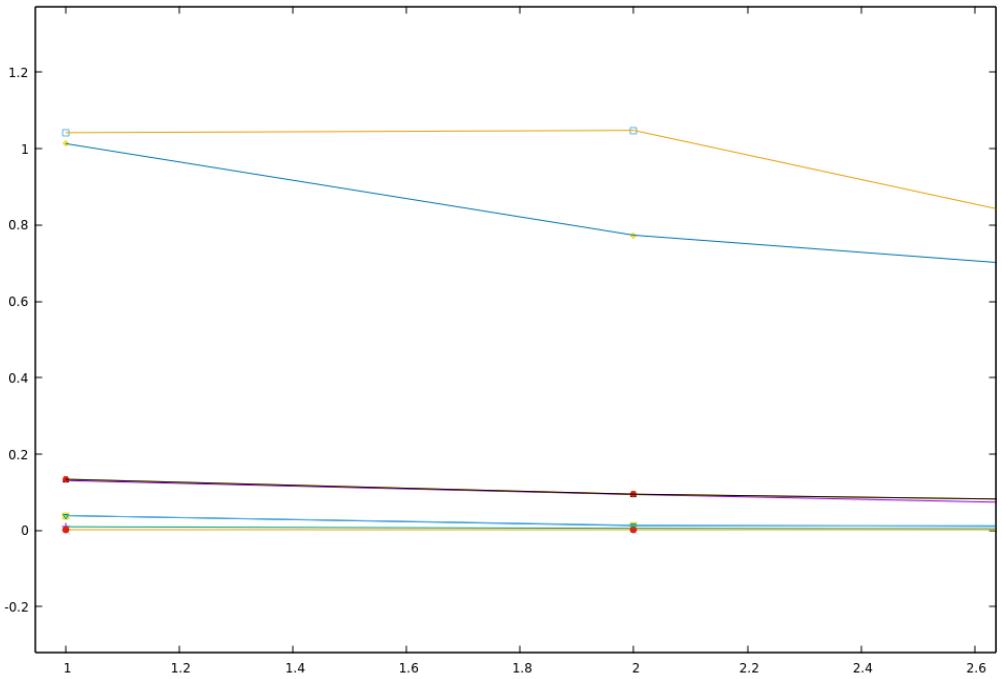


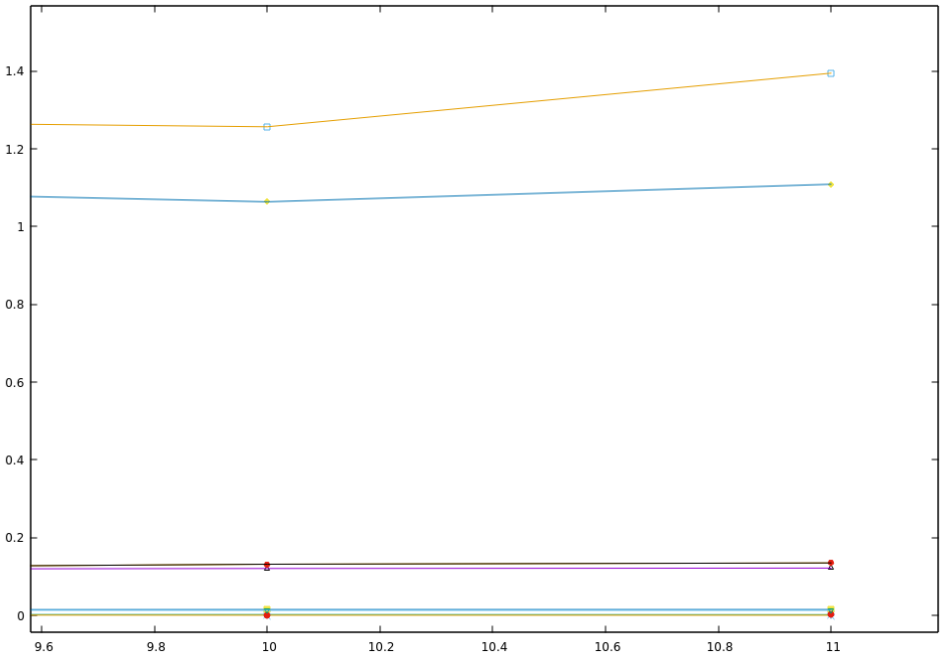






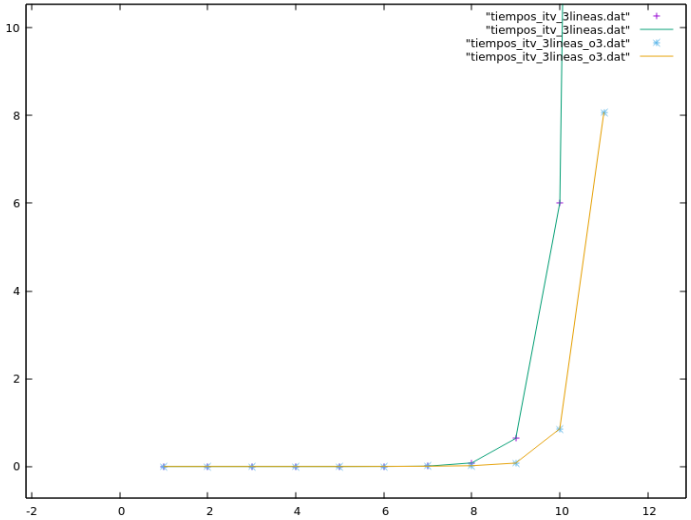
En esta imagen podemos ver juntas las gráficas que han generado las ejecuciones cuando hemos compilado con la opción de optimización -O3 junto a la gráfica que generó tener 10 coches fijos y líneas de inspección cambiantes, para ver la gran diferencia de tiempos que se ha generado. Para ver lo que está ocurriendo entre las líneas de abajo vamos a agrandar esa zona para inspeccionar con más detalle:

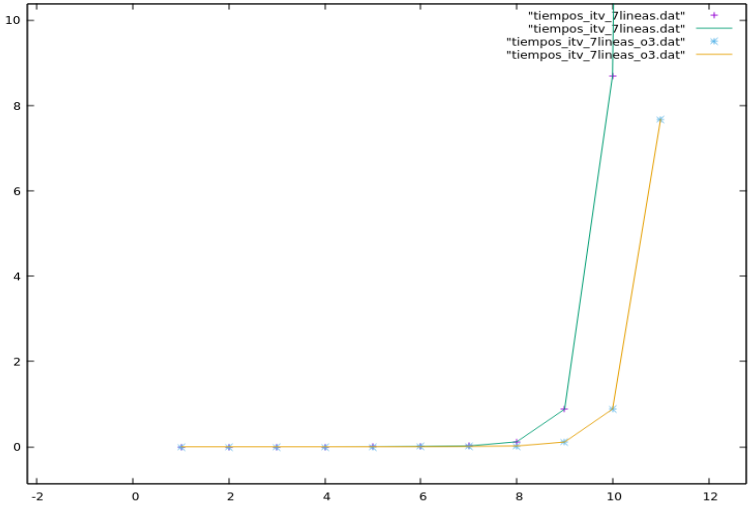


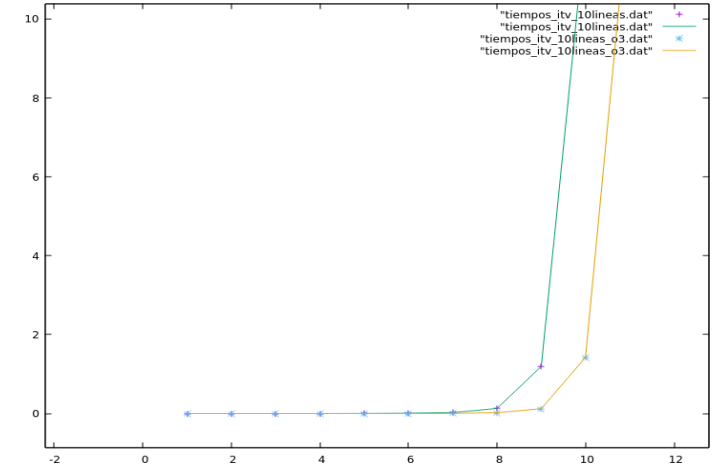


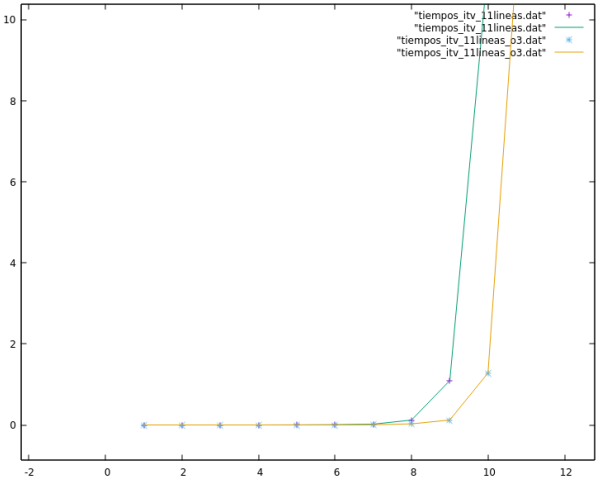
En la imagen de la izquierda vemos la parte inicial, en la que sigue ocurriendo que para un número mayor de coches fijos y líneas cambiantes el tiempo aumenta ya que también lo hace el número de comprobaciones el programa. En la foto de la derecha vemos el final de las gráficas para diferenciar cómo acaba cada ejecución.

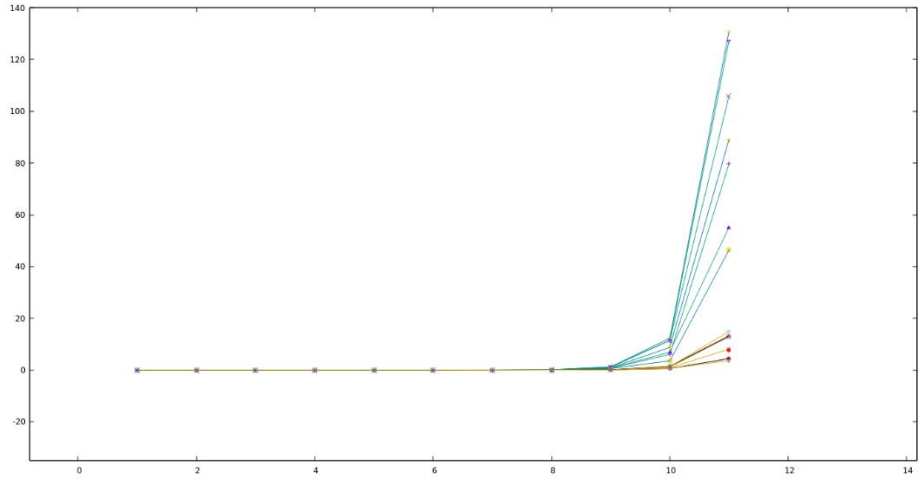
3, 7, 9 y 11 líneas (de izquierda a derecha respectivamente) comparando para cada caso su resultado con compilación normal y optimizada con -O3

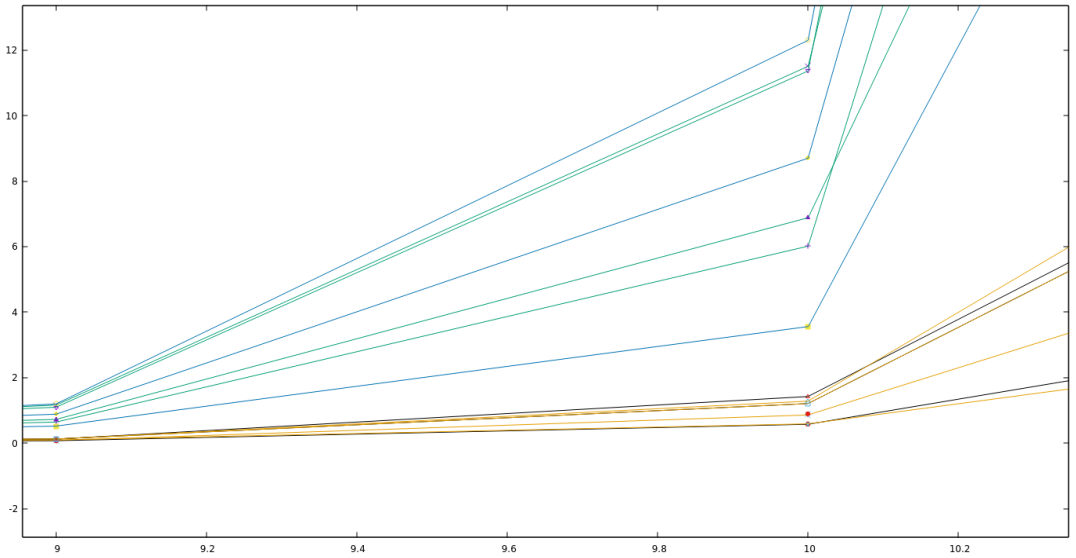


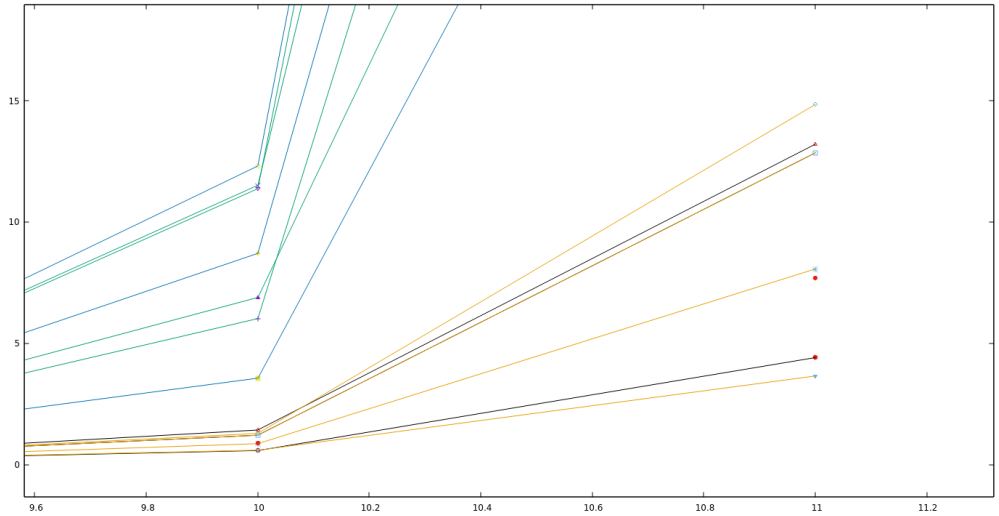












Si comparamos todas las ejecuciones normales sin optimizar con la opción -O3 del compilador podemos comprobar en estas gráficas como obtenemos unos tiempos mucho mejores para cada caso por individual, y junto con las imágenes en las que agrandamos esas zonas conflictivas vemos como se mantienen las diferencias que existían en las ejecuciones normales, pero en esta ocasión conseguimos tiempos mucho más reducidos.

Para no hacer repetitiva esta parte, hemos reducido el número de ejemplos que se muestran en la memoria, pero todas estas imágenes se encuentran en las carpetas correspondientes de las fotos ordenadas para cada caso.

Además, también contamos con la comparación de cuatro casos concretos (7 y 10 líneas fijas y 7 y 10 coches fijos) en los que mostramos qué diferencia de tiempo ocurre si se ejecuta ese mismo caso con opciones de compilación -O1, -O2 y -O3, en la que por normal general vamos a obtener mejores tiempos para -O1, -O2 y -O3 respectivamente. Esta parte no la hemos incluido porque es algo que se puede obviar pero que queríamos realizarlo por si ocurría algo inesperado e interesante para comentarlo en la memoria. Se encuentra en su carpeta correspondiente.

El hecho de que el tiempo que tarda cada coche en realizar la inspección se genera de forma aleatoria ha ocasionado unas pequeñas modificaciones en los tiempos esperados en las ejecuciones que han sido comentadas y estudiadas al detalle.

Como conclusión final, decir que, en su gran mayoría, hemos obtenido unos resultados esperados ya que al tratarse de la técnica de backtracking la utilizada en la resolución de este problema, cuando tenemos un número mayor de líneas de inspección como de coches que necesitan realizar la inspección, el número de comprobaciones aumenta de forma considerable, por lo que los tiempos para esas ocasiones es mayor que cuando tenemos un valor bajo para estas variables.

Pseudocódigo

La función booleana que soluciona el problema asignado recibe el nombre de ITV cuyo pseudocódigo requiere a su vez del mismo de la función comprobacionEtapa la cual nos dice para cada etapa si esta es correcta o no.

Elementos a tener en cuenta en el pseudocódigo:

·vector<int> que tiene el tiempo de cada coche = tiempos_coches

·vector<int> que tiene los índices de los coches que aún quedan por asignar = nRestantes

·vector<int> que tiene soluciones temporales = sol

·vector<int> que tiene la solución definitiva del problema = sol_def

·vector<int> que sirve para calcular el tiempo completo y las asignaciones de los coches = tiempo_lineas

·tiempos_lineas_mod es un vector auxiliar para hacer modificaciones y no perder tiempos_lineas

```
bool ITV(numLineas, tiempos_coches, etapa, nRestantes, sol, sol_def, tiempo_lineas)

INICIO

Si etapa es igual tiempos_coches.tamaño entonces

        Si tiempoParcial menor que tiempoSolucion entonces
                sol_def <- sol
                tiempoSolucion <- tiempoParcial
        FinSi

Devuelve falso
FinSi

for i desde 0 hasta nRestantes.tamaño y NO correcto hacer
        tiempos_lineas_mod <- tiempos_lineas
        solucion[etapa] <- nRestantes[i]

        Si (comprobacionEtapa(tiempos_coches, tiempoParcial, numLineas, tiempoSolucion,
tiempos_lineas_mod, nRestantes[i], etapa) es VERDADERO) entonces
                Si etapa es diferente tiempos_coches.tamaño entonces
                        elementoBorrado <- nRestantes[i]
                        nRestantes.borrar(elemento en posición i);
                        correcto <- ITV (con etapa+1)

                        for j desde etapa hasta solucion.tamaño hacer
                                solucion[j] <- (-1)

                        nRestantes.insertar(elementoBorrado en la posición de la que se borro);
                Si no
                        correcto <- Verdadero
                FinSi
        FinSi
FinFor
Devuelve correcto

FINAL// fin de la función ITV
```

Pseudocódigo para la función comprobacionEtapa

Elementos a tener en cuenta en el pseudocódigo:

·menor tiempo obtenido hasta el momento de un camino completo = menorTiempo
·índice del nuevo coche a comprobar = nuevoCoche
·la variable “menor” estará inicializada a INT_MAX para que cualquier valor sea menor que ella
·la variable “mayor” estará inicializada a -1 para que cualquier valor sea mayor que ella

```
comprobacionEtapa(tiempos_coches, tiempoParcial, numLineas, menorTiempo, tiempos_lineas,
nuevoCoche, etapa)

INICIO
for i desde 0 hasta tiempos_lineas_aux.tamaño y NO salir hacer
        Si etapa es menor que numLineas y tiempos_lineas_aux[i] igual a -1 entonces
                tiempos_lineas_aux[i] <- tiempos_coches[nuevoCoche]
                salir <- Verdadero
        Si no, si tiempos_lineas_aux[i] es menor que menor entonces
                menor <- tiempos_lineas_aux[i]
                posInsercion <- i
        FinSi
FinFor

Si etapa mayor o igual que numLineas entonces
        tiempos_lineas_aux[posInsercion] += tiempos_coches[nuevoCoche]

for w desde 0 hasta tiempos_lineas_aux.tiempo hacer
        Si tiempos_lineas_aux[w] mayor que mayor entonces
                mayor <- tiempos_lineas_aux[w]

                Si mayor es mayor que menorTiempo entonces
                        Devuevle Falso

        FinSi
FinFor

tiempoParcial <- mayor
tiempos_lineas <- tiempos_lineas_aux

devuelve Verdadero

FINAL
```