Práctica de búsqueda local

IA

2022-2023 Q2

Àlex Domínguez

Manel Murillo

Joan Sales

ÍNDICE

Planteamiento del problema

El problema

El problema que se nos plantea en este trabajo está relacionado con el cambio climático y el calentamiento global. Al ser una amenaza cada vez más grande, la raza humana está obligada a producir menos gases de efecto invernadero. En este caso en concreto, se trata de reducir las emisiones de CO_2 de una ciudad.

Una manera de conseguir este objetivo, es organizar a las personas para reducir el gasto de combustible, por este motivo se ha propuesto una iniciativa peculiar en la ciudad: De todas las personas que tienen que ir trabajar a las 8:00 de la mañana, se elegirán a unas en concreto para que lleven con su coche al trabajo a las demás, minimizando el número de coches de forma muy significativa y por ende emitiendo menos gases invernaderos.

Nuestro objetivo es hacer una IA que sabiendo quienes son los conductores, nos diga cómo organizar los trayectos de cada vehículo para que todo el mundo llegue al trabajo a tiempo (en 1h) y que haya el menor número de coches, que recorran la menor distancia posible, para así contaminar lo mínimo.

Representación y datos del problema

La ciudad se representa como un cuadrado de 10x10 kilómetros dividida en manzanas de 100x100 metros, en resumidas cuentas, una matriz de:

(10000m x 10000m) / (100m x 100m) = 100x100 casillas

Habrán N personas que necesitan ir al trabajo, y de éstas N personas, M (como máximo) serán conductores, siendo M < N. Cada persona se define como una posición X e Y donde está al inicio del problema (Su casa) y una posición X e Y donde tiene que ir a trabajar (Ambas dentro de esta ciudad de 100x100 casillas).

Se nos especifica que la velocidad media del tráfico es de 30km/h. Con esto hemos deducido que:

30 km/h -> 30000 m/h;
(30000 m/h) / (100 m/casilla) = **300 casillas/h**

Un coche puede recorrer un total de **300 casillas** de la matriz en una hora. Este dato nos sirve bastante para nuestro programa.

También se nos dice que un coche tiene tres plazas: una para el conductor y dos más para los pasajeros, nunca un coche puede tener más de tres personas dentro de él.

Una solución válida del problema debe:

- Organizar trayectos para un número igual o inferior a M conductores.
- Asegurarse de que todas las N personas lleguen al trabajo en menos de 1 hora.
- Que ningún coche en ningún momento tenga más de 3 personas dentro.
- Minimizar la distancia recorrida por cada coche en la medida de lo posible.
- Minimizar el número de coches en la medida de lo posible.

Dado que se busca mejorar las asignaciones de personas, número de coches y trayectos de los conductores de la solución, está claro que este problema entra dentro del conjunto de problemas de búsqueda local. Es decir: no buscamos la mejor solución siguiendo un único criterio, buscamos mejorar en la medida de lo posible varios objetivos distintos, haciendo que no sea una tarea trivial destinada a un algoritmo sencillo.

Desarrollo

Definición de “Estado”

Primeramente optamos por almacenar todos los coches en una lista “CarSet”, donde cada uno tendría una lista de instrucciones, la posición actual respecto al tiempo y el tiempo restante del vehículo. Lamentablemente nos topamos con muchos problemas, como por ejemplo que los operadores no estaban del todo claros para una técnica de Hillclimbing con esta representación, así que buscamos otra forma de hacerlo.

Después de meditarlo, decidimos que para nosotros, un estado es una asignación parcial de los usuarios a los coches en el “CarSet”. Para simplificar, supondremos que un coche siempre empezará el trayecto en las coordenadas origen de su conductor, irá a recoger al primer pasajero, irá a recoger al segundo pasajero, dejará en las coordenadas destino al primero, después al segundo y finalmente irá al lugar de trabajo del conductor.

Hemos tomado esta decisión para reducir el coste de los operadores en el futuro, ya que si tuviésemos que decidir cuándo dejar a cada pasajero se añadirían más posibilidades al operador de “swapInCar” que mencionaremos más adelante, y además, teníamos la hipótesis de que las soluciones que nos podía dar esta representación de estado no serían tan malas.
También tuvimos en consideración la posibilidad de hacer que un coche dejase a un pasajero a medio camino para después ser recogido por otro coche que le fuera mejor por el camino que hacía, pero almacenar las posiciones parciales de todos los coches supondría un coste en memoria demasiado elevado.

Así que en resumidas cuentas, nuestro “estado” es representado por la clase “CarSet”, una arrayList con todos los Usuarios del sistema y otra arrayList de coches, donde cada uno tiene tres asientos para los usuarios (El conductor, el primer pasajero y el segundo pasajero). Si el asiento de un coche es *null*, significa que no hay pasajero en ese asiento, de lo contrario estará asignado un usuario, que se representa con las coordenadas de origen y las de destino.

Con esta representación del problema podemos saber las distancias que van a recorrer todos los coches simplemente calculándolas siguiendo el esquema de recogida y dejada estipulado anteriormente. Como sabemos que un coche no puede recorrer más de 300 casillas, es muy sencillo comprobar si un estado es una solución válida o no.

Dicho esto, nuestro espacio de búsqueda son todos aquellos estados donde están todos los usuarios asignados a algún coche de “CarSet” y cumplen con las condiciones de aplicabilidad del enunciado. En nuestros ojos es un espacio de búsqueda completo y adecuado ya que solo se aceptan asignaciones completas de Usuarios.

El tamaño concreto del espacio de búsqueda suele ser del orden de $O(r^p)$, donde “r” es el factor de ramificación promedio y “p” la profundidad promedio hasta llegar a una solución. En este problema no es sencillo saber la profundidad a la que se llegará a una solución ya que depende de cuántos usuarios (N) haya en el sistema. El factor de ramificación promedio depende de los operadores, pero tendrá una pinta aproximada al orden de $O(N^2*M^2)$, ya que huele a que tendremos que hacer cambios de asignaciones entre varios usuarios entre uno o varios coches.

Generación de solución inicial

OrderedInit

El diseño de la inicialización del estado inicial de manera ordenada lo hemos planteado de la siguiente manera: Después de recibir los usuarios que deben ser conductores en el mes que estamos generando, asignamos a los usuarios como pasajeros según el orden que tienen el la lista que los contiene (*allUsers*). Solo se añaden pasajeros válidos, es decir, no se asigna a ningún pasajero que implique que la distancia recorrida por el coche supere las 300 casillas disponibles. En caso de haber pasajeros que no puedan ser colocados en ningún coche bajo esa premisa, se crea un nuevo coche con ese usuario como conductor.
De esta manera, nos aseguramos de que la solución inicial esté dentro del espacio de soluciones válidas, aunque acaben habiendo más coches de los que se especificó en la entrada. Esta decisión la hemos tomado porque quizá con nuestros operadores no podamos entrar durante la búsqueda en un estado solución si empezamos desde fuera.

Como se han de llenar todos los coches comprobando si caben los pasajeros, el coste para generar esta solución en el peor de los casos es del orden de O(M*N), siendo M el número de conductores y N el número de usuarios del sistema.

Los problemas de esta solución inicial son:

- Al empezar con más coches de los estipulados, puede que hayamos ensuciado el camino del algoritmo.
- No se puede randomizar del todo

RandomInit

De manera parecida, la asignación aleatoria la hemos generando con índices aleatorios, de la lista de usuarios, y vamos probando hasta encontrar un pasajero válido. Similarmente a la implementación anterior, con el conjunto de usuarios que no puedan ser asignados como pasajeros, crearemos coches siendo estos los conductores. Al usar la misma estratégia que en la inicialización anterior tienen la misma complejidad en caso peor.

Los problemas de esta solución inicial son:

- Al empezar con más coches de los estipulados, puede que hayamos ensuciado el camino del algoritmo.

GreedyInit

Por último, la inicialización voraz, es bastante más costosa a nivel de complejidad, ya que para cada coche, encontramos el mejor pasajero para añadir al coche. Esto lo hacemos calculando todas las distancias que producirían añadir a un pasajero al coche y quedándonos con el mejor.

Al estudiar la complejidad del algoritmo vemos que, por cada M coches asignar N-M pasajeros, siendo N el número de usuarios, y por cada pasajero calculamos M posibles distancias, nos queda una complejidad total de:

$$O(M*(N-M*(M))) = O(M*(M(N-1))) = O(N*M^2)$$

Los problemas de esta solución inicial son:

- Puede ser demasiado buena y llegar a un mínimo local muy rápido
- No se puede randomizar del todo
- El coste es muy elevado

Operadores

Los operadores de búsqueda local deben permitir movernos dentro del espacio de soluciones y encontrar el siguiente estado con una solución ligeramente mejor que la que ya tenemos. Éstos además deberán cumplir las condiciones de aplicabilidad mencionadas en la representación del problema. En nuestro caso hemos utilizado 3 operadores:

MovePassengerToAnotherCar: Dado un coche, asigna a uno de sus pasajeros a otro coche del sistema (como último pasajero posible). Ya que todos los usuarios del sistema siguen asignados a un coche y se comprueban las condiciones de aplicabilidad, seguimos estando dentro del espacio de soluciones. Su objetivo es reducir la distancia recorrida total entre dos coches.

Este operador es trivialmente útil, lo utilizamos para probar a asignar un usuario (pasajero) a otro coche distinto.

Como tenemos que hacer este operador sobre un coche, elegir a un pasajero y asignarlo a otro coche, el factor de ramificación es del orden de O(M*2*(M-1)), que queda ≈ $O(M^2)$, donde M es el número de coches.

SwapOrderInCar: Dado un coche con dos pasajeros y el conductor, intercambia el orden de los pasajeros, como todos los usuarios del sistema siguen asignados a un coche y se comprueban las condiciones de aplicabilidad, seguimos estando dentro del espacio de soluciones. Su objetivo es reducir la distancia recorrida de un coche.

Este operador sirve por si es mejor dejar a un pasajero u otro primero, pongamos un ejemplo:
Tenemos un conductor que empieza en 0,50 y debe acabar en 0,100.
Este conductor lleva a dos pasajeros:

El primero está en 0,75 y debe ir a 0,99
El segundo está en 0,25 y debe ir a 0,0

Si seguimos el orden de recogida y dejada estipulado, el coche debería recorrer una distancia de 25+50+74+99+100 = 348 (No le da tiempo)

Sin embargo, después de intercambiar los pasajeros la distancia recorrida queda como 25+50+75+99+1 = 250 (Le da tiempo)

Acabamos de comprobar que este operador es útil, y somos conscientes de que este resultado se podría conseguir simplemente con el operador de "*MovePassengerToAnotherCar*" aplicado varias veces con un coche auxiliar, pero optamos por usar también este por ser más compacto y ahorrar acciones para llegar antes a una solución siguiendo este patrón, además, por cómo funciona Hillclimbing, podría ser que este "swap" dentro de un mismo coche nunca se llegase a hacer con el anterior operador.

Ya que el operador se aplica sobre un coche y punto, el factor de ramificación es del orden de O(M), donde M es el número de coches. Nótese que este factor de ramificación es menor que hacer 4 operadores "*MovePassengerToAnotherCar*".

MakeDriverToPassenger: Dado un coche con solamente el conductor, Asigna a ese usuario como pasajero de otro coche y el vehículo original deja de existir en el sistema. Como todos los usuarios del sistema siguen asignados a un coche y se comprueban las condiciones de aplicabilidad, seguimos estando dentro del espacio de soluciones. Su objetivo principal es reducir el número de coches.

La utilidad de este operador es reducir la cantidad de coches del sistema, mientras simultáneamente comprobamos de minimizar la distancia total recorrida entre el coche que va a desaparecer y el que se le va a añadir el usuario.

El operador elige uno de los M coches que solo tenga el conductor y lo asigna a otro de éstos, por tanto el factor de ramificación es del orden de O(M*(M-1)), que es ≈ $O(M^2)$, donde M es el número de coches del sistema.

Somos conscientes que este operador es exactamente el mismo que "MovePassengerToAnotherCar" pero adaptado a eliminar coches del sistema. Escogimos tenerlos por separado para explorar más limpiamente el espacio de soluciones. Pensamos que si tuviéramos incorporado el eliminar coches dentro del otro operador siempre saldría una mejor heurística y podría ser un problema para el hill-climbing ya que se podría llegar a un mínimo local demasiado rápido.

Heurísticas

Las heurísticas sirven para cuantificar cuán buena es una solución dada. Es muy importante (y complicado) elegir una que sea capaz de representar con exactitud lo que queremos así que hemos propuesto varias:

H1: Sum distancias

Como primer heurístico queríamos probar a hacer algo sencillo, simplemente quedarnos con las soluciones donde globalmente los coches recorrieron la menor distancia. También teníamos curiosidad de si esta heurística tan simple daría resultados buenos con lo intuitiva y fácil que es.

H2: (Sum RemDist) / log2(avg distancias conducidas)

Como segundo heurístico propusimos hacer algo más complejo con las distancias y dimos con esta fórmula. La explicación del por qué de todas las variables es la siguiente:

- Queremos maximizar la distancia restante de los coches, por eso está en el numerador (valor alto, heurística alta).
- Queremos minimizar las distancias recorridas de cada coche, por eso está en el denominador (valor bajo, heurística alta), hacemos la media de este valor para saber si los coches están eficientemente organizados.
- Hacemos el logaritmo para "achatar" la media de las distancias, ya que consideramos que el hecho de que los coches estén equilibrados no es tan importante como el total de distancia sobrante por recorrer.

H3: ((Sum RemDist) * N) / (log2(avg distancias conducidas) + M)

Como tercer heurístico queríamos poner más variables en juego para reducir el número de coches, así que dimos con esta fórmula, similar a la anterior pero con dos variables nuevas. El razonamiento, a parte de seguir los puntos de la heurística anterior, es el siguiente:

- Queremos minimizar el número de conductores, por eso está en el denominador (valor bajo, heurística alta).
- Multiplicamos el numerador por el número de usuarios porque consideramos que cuantos más usuarios haya, más difícil es dar con un una solución donde haya menos coches así que contrarrestamos de esta manera los experimentos con un número elevado de usuarios.

Experimentación

Influencia de la solución inicial

Para estudiar cómo la estrategia de solución inicial influencia la resolución del problema, hemos decidido hacer dos experimentos: uno muy sencillo en el cuál medimos el tiempo y estudiamos los resultados de ejecutar las funciones generadoras, y otro donde se ejecuta todo el programa.

Nuestra hipótesis inicial era:

La iniciación aleatoria nos dará los mejores resultados, ya que en promedio, llegaremos a resultados bastante distintos; la inicialización voraz dará buenos estados iniciales, ya que hay un algoritmo detrás encargado de hacer la mejor organización posible basándose en la distancia, pero creemos que caeremos en mínimos locales bastante pronto; y la inicialización ordenada pensamos que será la que peor responderá.

Todos los experimentos que hemos hecho están realizados con semillas aleatorias (según la semilla, aparecen los usuarios en diferentes sitios y deben llegar a destinos diferentes), pero el número de usuarios y conductores mínimos es el mismo, decidimos hacerlo con 200 usuarios y 80 coches porque son valores con suficiente margen de mejora como para aplicar los algoritmos.

El primer experimento lo hemos llevado a cabo haciendo 100 ejecuciones, con semillas aleatorias, con una población de 200 usuarios y 80 coches a llenar, eso nos da una proporción de 2,5 usuarios/coche, por tanto, aproximadamente la mitad de coches tendrán 1 pasajero y la otra mitad 2. En la ordenación aleatoria, cada ejecución de las 100, es un promedio de 5 ejecuciones, también con semillas aleatorias.

Hemos decidido tomar muestras del número de coches, la distancia y el tiempo. Con el número de coches, hemos decidido hacer el cálculo de los coches extra que tiene que crear, ya que es lo que realmente nos interesa.

	Avg Extra Cars	Avg Inicial Distance	Avg Time
Ordered	1,11	18547,52	0,34
Random	0	8252,94	0,26
Greedy	0	8252,94	0,61

Como se puede apreciar en el cuadro, tanto la versión aleatoria, como la voraz, responden muy bien, no han creado coches extra en ninguna ejecución y han conseguido la misma distancia, pero el tiempo de ejecución de la random es casi tres veces más rápida que la voraz. De toda manera estamos hablando de ejecuciones de tiempos inferiores a la milésima de segundo. Por otro lado, la ejecución ordenada ha generado de media 1,11 coches extra y la distancia es mucho más superior que la de los demás.

Cabe destacar que ha tardado casi lo mismo en ejecutarse que en la versión aleatoria.

El segundo experimento constaba de 200 usuarios a repartir entre 80 coches y hemos realizado una ejecución ordenada con semillas aleatorias, 10 con la Greedy y y cada iteración de las otras eran el promedio de cinco aleatorias:

Cars			Distance			Tiempo			Pasos		
Ordered	Random	Greedy	Ordered	Random	Greedy	Ordered	Random	Greedy	Ordered	Random	Greedy
78	71	78	5882	5670	5422	4596	3912	2698	119	126	71
69	68	66	5043	5571	4869	4234	3795	2630	139	137	92
81	79	79	5314	5506	4978	3529	3430	2365	116	117	70
73	69	70	5489	5359	4790	3773	3661	2661	128	133	85
84	79	79	6217	5849	5103	3709	3586	2410	113	115	71
73	71	79	5276	5244	5423	3694	3763	2355	130	134	70
65	67	67	4583	4802	4520	3821	3737	2627	149	143	97
80	79	77	5416	5429	5028	3802	3558	2425	121	121	74
74	73	69	5211	5596	4711	3706	3668	2573	130	127	94
80	76	78	5540	5157	5023	3464	3584	2369	116	125	69

De este experimento hemos extraído cuatro valores, número de coches, distancia tiempo y pasos:

	Ordered		Random		Greedy	
	MED	DEV	MED	DEV	MED	DEV
Cars	75,70	5,93	73,20	4,73	74,20	5,47
Distance	5397,10	444,84	5418,30	296,94	4986,70	287,29
Tiempo	3832,80	337,82	3669,40	138,24	2511,30	138,38
Pasos	126,10	11,44	127,80	8,94	79,30	11,39

Los gráficos que hemos decidido usar para explicar los resultados del siguiente experimento son una tabla con los registros de las 10 ejecute ejecuciones, una tabla con las medias y las tres desviaciones de cada valor respecto al tipo de solución inicial, y por último un diagrama de caja con cada uno de los valores para cada tipo de inicialización.

En líneas generales vemos a través de los gráficos de caja que todos los valores están compactados. Si echamos un ojo por ejemplo a la media de los coches. Podemos ver que entre la más baja y la más alta sólo hay 1,5 coches.

	Ordered		Random		Greedy	
	Diff	%	Diff	%	Diff	%
Cars	5,41	6,67%	6,8	8,50%	5,8	7,25%
Distance	13150,42	70,90%	2834,64	34,35%	3266,24	39,58%

Podemos ver que para el valor de los coches, la máxima optimización que se consigue de media S, es organizar los usuarios con 6,8 coches menos de los que se nos pide.

Llegando a un promedio, en caso de la inicialización aleatoria, de una ocupación de los coches de N, luego iría laborar con una ocupación de M y la ordenada con una ordenación de K. Además, si nos fijamos bien las desviaciones son muy parecidas entre los tres tipos de iniciación inicial por este lado.
El cuadro de las diferencias entre de mejora que ha supuesto la ejecución de Hillclimbing no son muy explicativas para ver cuál de las tres estrategias es mejor.

Por un lado vemos qué la mejora del Greedy en coches es de un 6 %, tampoco es mucho mayor en las demás estrategias.

Lo que sí que es interesante es las mejoras de las distancias ya que por un lado el ordenado ha mejorado un 70 % respecto a la iniciación inicial aportada.

Por otro lado la estrategia Greedy ha conseguido menor la menor distancia, que es lo que buscábamos, y por tanto ha hecho una mejora de casi el 40 %, respecto a lo que la distancia refiere.

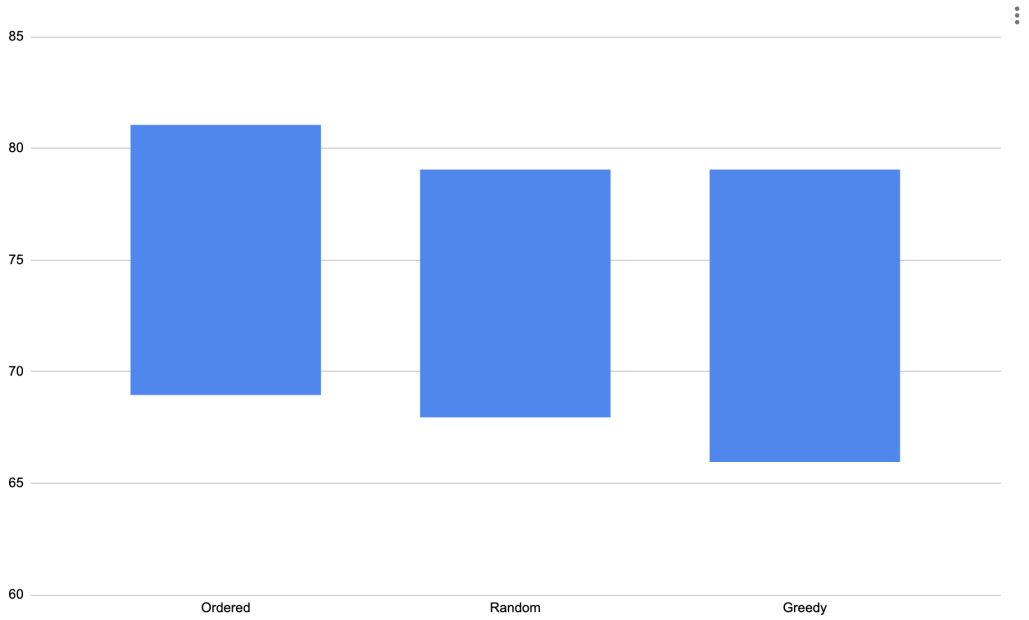
En cuanto a los coches, la mejor hora ha sido del 7,25 %, una diferencia de casi seis coches de media.

Por último, la versión aleatoria ha mejorado casi siete coches y le ha supuesto una mejora del ocho, y medio por ciento y en distancia un poco menos que el Greedy un tren casi un 35 %.

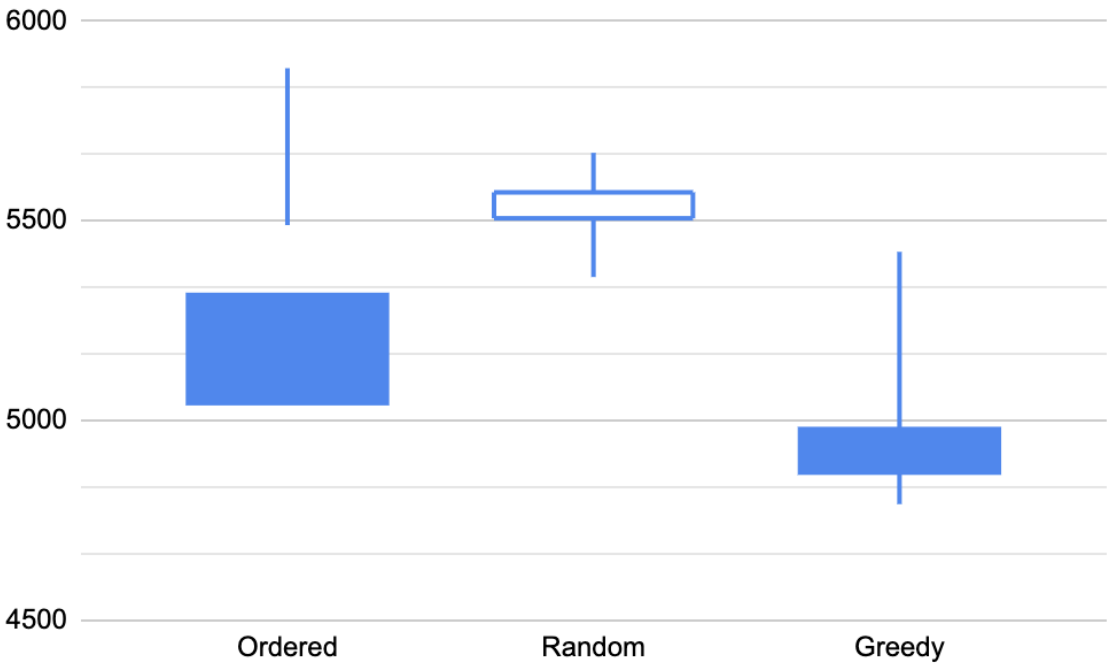
Por último vamos hablar de las otras dos variables que determinan realmente dónde está la diferencia entre usar una estrategia o la otra.

Si miramos tanto el diagrama de cajas del tiempo y de las acciones, ambos, el número de acciones y cantidad de tiempo que requieren usar, la inicialización ordenada y aleatoria, es significativamente superior a la voraz, sobre todo en el tiempo que tarda casi la mitad que las otras:

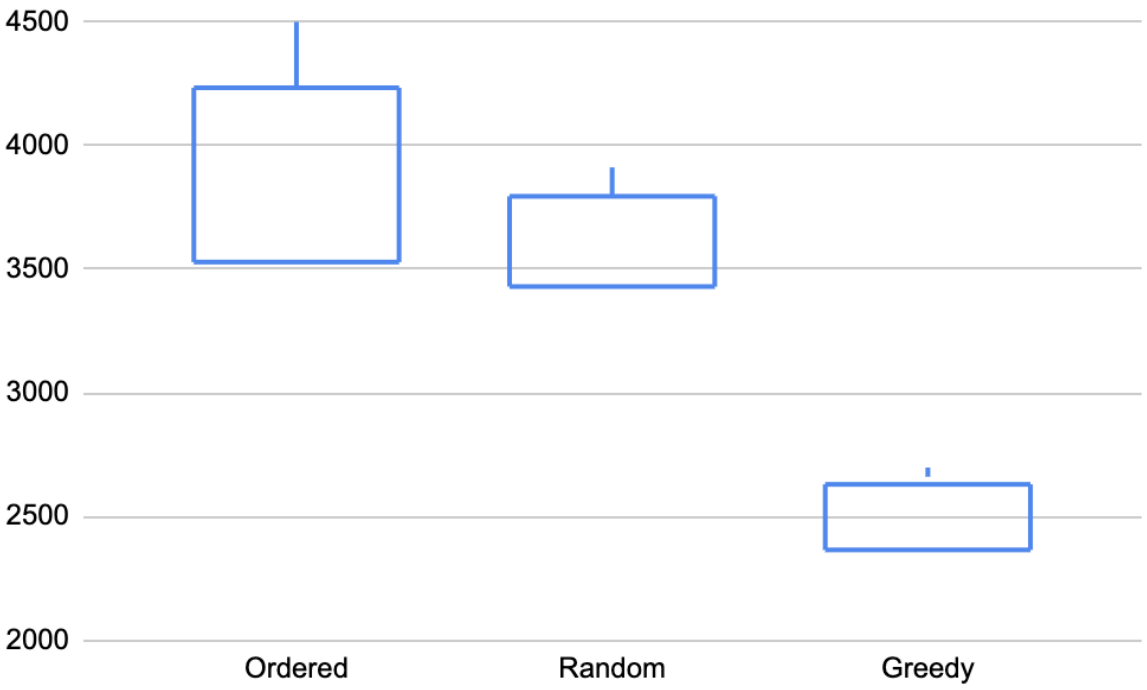
Cars:



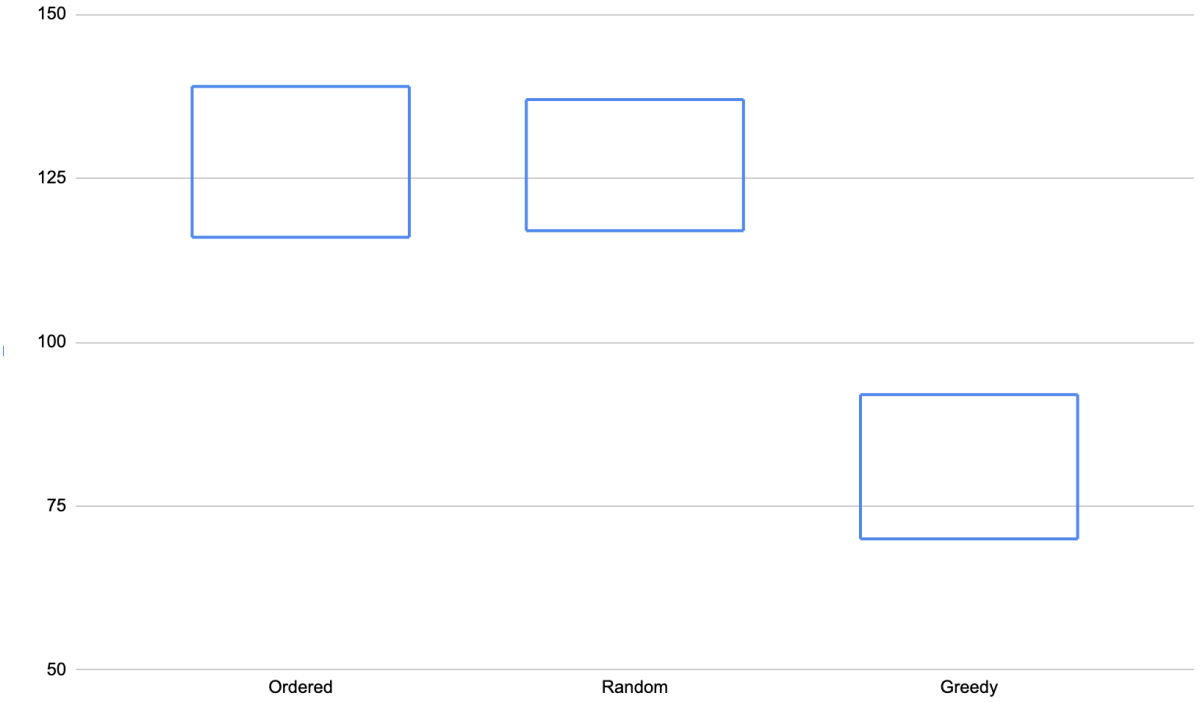
Distance:



Time:



Actions:

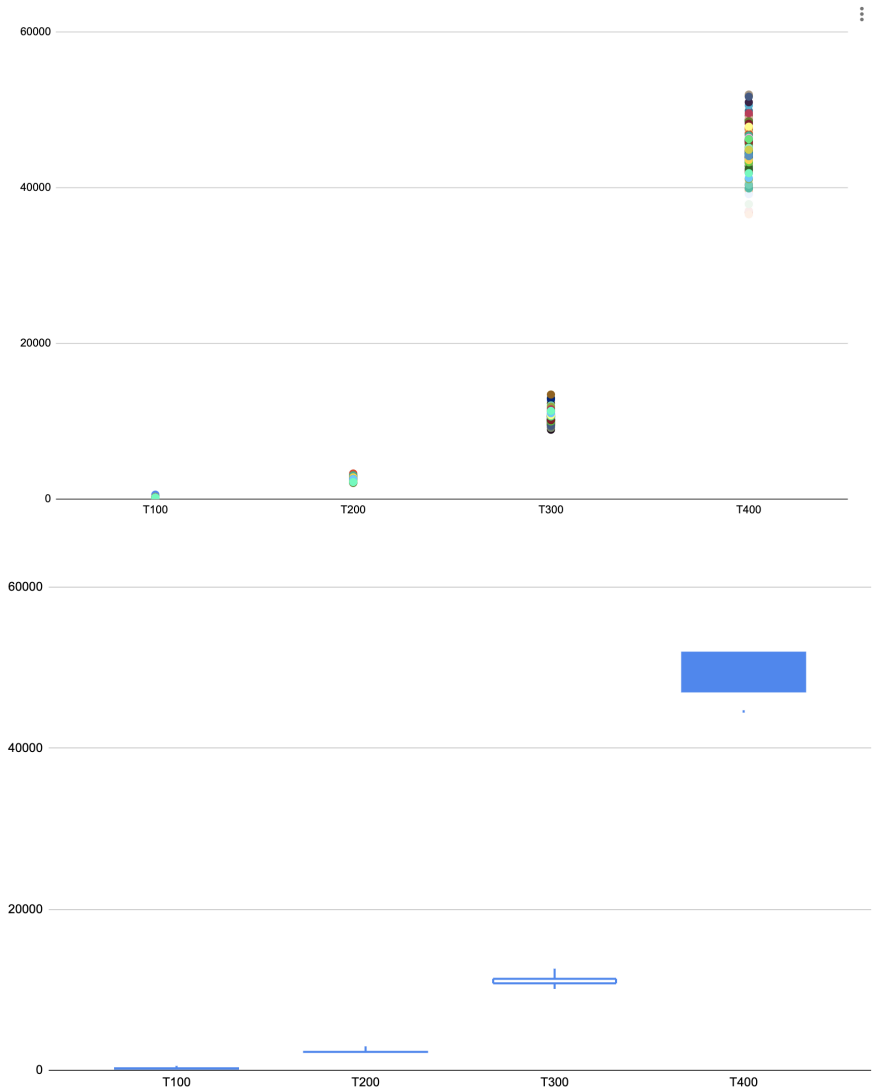


Influencia del tamaño del problema

La influencia del tamaño del problema la hemos decidido estudiar haciendo un experimento en el cual ejecutamos 100 veces el algoritmo con un tamaño de 100,200,300 y 400 usuarios y un ratio de un 40 % de coches respecto usuarios, lo cual no se deja una ocupación de 2,5 usuarios por coche. Para este experimento hemos usado la inicialización voraz, y por supuesto, la estrategia Hillclimbing.

La influencia del tamaño del problema la hemos estudiado únicamente a través de la variable tiempo.

Como podemos ver en el diagrama de cajas el tiempo y en el diagrama de dispersión, y el tiempo de ejecución tiene una tendencia exponencial. Hemos decidido añadir también una tabla con la media y la desviación del tiempo, para cada tamaño de entrada.



Podemos apreciar cómo la media va subiendo de manera vertiginosa respecto al tamaño de la entrada, como veíamos en los gráficos.

Un dato muy interesante que nos da el diagrama de caja es que los tiempos son muy parecidos entre ellos, igual que vemos en el diagrama de dispersión, casi todas las ejecuciones con el mismo tamaño dan tiempos de ejecución muy muy parecidos.

No hay ningún punto del gráfico de dispersión que el tiempo de ejecución de un tamaño de entrada esté a la misma altura del de otro:

Tamaño	T100	T200	T300	T400
Media	184,82	2443,54	10352,57	44582,12
Desviación	41,54	207,61	856,83	3365,40

Influencia de las heurísticas

Lamentablemente, solo hemos tenido tiempo de experimentar con una heurística (la primera de todas, H1), y no podemos saber a ciencia cierta la influencia de ésta sobre las soluciones que nos da el programa respecto a las demás.

Lo que sí podemos decir es que este heurístico es válido para guiarnos sobre una característica de la solución que queremos mejorar: las distancias recorridas de los coches.

Determinar los parámetros para el Simulated Annealing

Debemos parametrizar de forma correcta el Simulated Annealing, por lo tanto, vamos a estudiar las diferentes combinaciones de parámetros de Simulated Annealing. Usaremos la solución inicial creada con inicialización voraz y los operadores de los que disponemos. Las pruebas las haremos con 200 personas y 80 conductores, en la semilla 123.

Hemos hecho diversas combinaciones de los parámetros que actúan en Simulated Annealing. Con el número de iteraciones por paso de temperatura hemos probado con 100, 1000 y 10000. En el caso de la k, hemos trabajado con 10, 100 y 1000. Con lambda hemos usado los valores 0.01, 0.001 y 0.0001.

Cabe recalcar que en las tablas se observa que Simulated Annealing no siempre devuelve una solución válida, pero conociendo el algoritmo, contamos con que hay veces que no devolverá un resultado válido.

La combinación que mejor resultado nos ha dado ha sido: stiter = 1000, k = 10, lambda = 0.01.

Experiment Iteracions		Distancia Reduida (%)	Temps (s)	Cotxes reduits (%)	%goaltest
200persones/80cotxes	1000	41.45	14.79	6.25	100%
1000 stiter	10000	52.87	21.22	19.75	50%
k = 10	100000	56.14	53.83	24.63	30%
lambda = 0.01	500000	55.96	57.85	24.66	30%
Seed = 123	1000000	60.83	52.53	25.79	10%

Tabla con los mejores resultados obtenidos

Número de iteraciones

Lo primero a observar es el número de iteraciones. En la anterior tabla podemos observar cómo a partir de 100.000 iteraciones no existe mejora significativa en cuanto al porcentaje de distancia reducida ni en cuanto al porcentaje de coches reducidos. Por ende, vemos que usar un número mayor de iteraciones solo va a significar mayor tiempo de ejecución.
El único inconveniente que observamos al usar 100.000 iteraciones es que el porcentaje que existe de devolver una solución válida es bajo. 1000 iteraciones si que siempre devuelve un resultado final válido, claro que hay mucho menos porcentaje de distancia reducida y de coches reducidos.

Iteraciones por paso de temperatura

Cuando hemos querido modificar las iteraciones por paso de temperatura hemos observado que el cambio era mínimo en los resultados, y hemos encontrado 1000 como la que mejor media de resultados obtiene.

K

Al aumentar k hemos visto que tanto la distancia reducida como los coches reducidos aumentaban, pero a cambio, bajaba sustancialmente el porcentaje de solución válida.

Experiment Iteracions		Distancia Reduida (%)	Temps (s)	Cotxes reduits (%)	%goaltest
200persones/80cotxes	1000	45.01	13.41	17.85	40%
1000 stiter	10000	63.99	16.61	37.25	20%
k = 100	100000	64.45	50.99	34.5	20%
lambda = 0.01	500000	65.81	52.76	35.62	20%
Seed = 123	1000000	65.52	55.89	36.2	10%

Ejemplo de k = 100

Lambda

Modificar la lambda solo nos ha llevado a un porcentaje de solución valida mucho más bajo y a unos resultados que devuelven menos de 5 coches usados y distancias imposibles. Así que nos hemos quedado con 0.01.

Conclusion

Para la comparación entre los dos algoritmos usaremos:
100.000 iteraciones, stiter = 1000, k = 10, lambda = 0.01.

Comparación entre algoritmos

Vamos a comparar el algoritmo de Hillclimbing y el de Simulated Annealing.
El método que usaremos para generar la solución inicial será la inicialización voraz. Usaremos 200 personas y 80 conductores y lo que usaremos para compararlos es el porcentaje de distancia reducida por segundo de ejecución.

Resultados de Hillclimbing

Distancia inicial	Distancia Final	%Distancia reducida	Coches iniciales	Coches Finales	Tiempo	DistReducida/Tiempo	Goal Test
8353	4568	45.31	80	73	4.99	758.52	TRUE
8353	4568	45.31	80	73	4.1	923.17	TRUE
8353	4568	45.31	80	73	3.93	963.10	TRUE
8353	4568	45.31	80	73	3.83	988.25	TRUE
8353	4568	45.31	80	73	4.57	828.23	TRUE
8353	4568	45.31	80	73	3.97	953.40	TRUE
8353	4568	45.31	80	73	4.05	934.57	TRUE
8353	4568	45.31	80	73	3.81	993.44	TRUE
8353	4568	45.31	80	73	4.09	925.43	TRUE
8353	4568	45.31	80	73	3.97	953.40	TRUE
8353	4568	45.31	80	73	4.131	922.15	100%

Resultados de Simulated Annealing

Distancia inicial	Distancia Final	%Distancia reducida	Coches iniciales	Coches Finales	Tiempo	DistReducida/Tiempo	Goal Test
8353	4344	47.99473243	80	67	66.89	59.93	TRUE
8353	3781	54.73482581	80	63	57.12	80.04	FALSE
8353	4389	47.45600383	80	71	51.26	77.33	TRUE
8353	3545	57.56015803	80	61	55.22	87.07	FALSE
8353	3851	53.89680354	80	60	56.72	79.37	TRUE
8353	4065	51.33484975	80	65	61.25	70.01	TRUE
8353	3748	55.12989345	80	62	68.25	67.47	TRUE
8353	3190	61.8101281	80	54	48.132	107.27	FALSE
8353	3530	57.73973423	80	65	51.94	92.86	TRUE
8353	3890	53.42990542	80	62	57.796	77.22	FALSE
8353	3833.3	54.10870346	80	63	57.4578	79.8574972	60%

Podemos observar que en el caso de Simulated Annealing, la solución media es mejor tanto en el porcentaje de distancia reducida, como en el número de coches eliminados. El problema de este algoritmo es el tiempo medio de ejecución, que roza el minuto.

De forma contraria, el algoritmo de Hillclimbing da peor resultado medio en los dos valores, pero el tiempo medio de ejecución es a penas de 4.1 segundos. Un tiempo mucho menor que el de su competidor.

Por lo tanto, si usamos de comparador la distancia reducida por el tiempo de ejecución, es mucho más alto en el caso de Hillclimbing. Para 200 usuarios y 80 conductores, que eran nuestros parámetros del problema, Hillclimbing es el mejor algoritmo.

Falta recalcar que, en el caso de que el número de usuarios aumentara a un valor mucho mayor y sabiendo que Hillclimbing tiene problemas para entradas muy grandes, Simulated Annealing tendría una clara ventaja, ya que para ese tipo de entradas, Hillclimbing podría tardar horas, incluso días, en devolver un resultado, a diferencia de Simulated Annealing que podría devolver un resultado en un tiempo mucho menor.

¿Hemos resuelto el problema?

Se puede observar que las soluciones que encuentran todos los experimentos minimizan tanto las distancias recorridas de los coches como el número de coches en general, así que podemos confirmar que ésta IA sí resuelve el problema planteado en el enunciado.

Extrapolando estos datos a gran escala, este programa reduce en cierta medida las emisiones de CO_2 de una ciudad, y es capaz de organizar a personas con la iniciativa que se propuso para ello.

Conclusiones

En esta práctica hemos profundizado en dos algoritmos de búsqueda local, Hillclimbing y Simulated Annealing, dos algoritmos de los que hemos descubierto sus ventajas y desventajas.
Hemos de reconocer que al inicio nos costó encontrar el camino adecuado para construir el estado inicial, para crear las funciones sucesoras, los operadores, etc. Pero a medida que nos íbamos adentrando en el trabajo, aprendimos cómo preparar el estado para que se pudiera meter en los algoritmos de búsqueda, y a partir de entonces el trabajo fue siendo cada vez más asequible.
Con la experimentación hemos observado realmente cómo funciona el programa que hemos hecho y sus características. Por ejemplo, hemos visto cuál de nuestras tres formas de crear una solución inicial es más eficaz.
En definitiva, esta práctica nos ha ayudado a entender mejor el cuándo y el cómo del uso de un algoritmo de búsqueda local, con una práctica con un enunciado que es verosímil y que acerca la teoría de las clases al mundo real y sus problemas.

Trabajo de innovación

Avances

Tenemos pensado el tema de lo que vamos a investigar: Un sistema de vigilancia con IA que predice si van a ocurrir desgracias o accidentes en el lugar.

De momento, el plan principal es que cada integrante del equipo busque por su cuenta aplicaciones reales y proyectos donde se está usando para más adelante ponernos en común y empezar a redactar y discutir los métodos.

Enlaces de noticias relacionadas, de momento se quiere hacer en los juegos olímpicos de 2024:

- https://es.euronews.com/next/2023/02/03/paris-quiere-desplegar-videovigilancia-asistida-por-ia-con-motivo-de-los-juegos-olimpicos-

- https://www.seguritecnia.es/actualidad/inteligencia-artificial-vigilando-los-juegos-olimpicos-2024_20230314.html