Práctica 1.b:

Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Agrupamiento de Restricciones

CURSO 2019-2020

Algoritmos considerados:

Greedy COPKM y BL (Búsqueda Local)

Pablo Pérez Méndez

54108599H

ccppabloperez@correo.ugr.es

Grupo 3ºB (MH1): Miércoles de 17:30h a 19:30h

Profesor: Óscar Cordón

Índice

Breve descripción del problema:

El problema que se estudia en esta práctica consiste en una generalización del agrupamiento clásico. Teniendo un conjunto de datos X con un número de instancias n, la resolución de este consistirá en agrupar los elementos de X en diferentes clusters de manera que se minimice la desviación general y, a su vez, intentando cumplir el mayor número de restricciones posibles. Las restricciones impuestas para el problema se dividen en dos tipos:

- Restricción Must-Link (ML). Si esta restricción está asociada a una pareja de elementos, quiere decir que ambos elementos deben pertenecer al mismo cluster.

- Restricción Cannot-Link (CL). Si esta restricción está asociada a una pareja de elementos, quiere decir que ambos elementos no pueden pertenecer al mismo cluster

En esta práctica se trabajará con 3 conjuntos de datos distintos:

- El conjunto de datos Iris. Posee 3 tipos de clases.

- El conjunto de datos Ecoli. Posee 8 tipos de clases.

- El conjunto de datos Rand. Posee 3 tipos de clases.

Para cada uno de ellos habrá 2 conjuntos de restricciones diferentes (uno que consistirá en el 10% de restricciones posibles del problema y otro que tendrá el 20%), lo cual quiere decir que se trabajará con 6 instancias distintas.

Como ya se ha explicado anteriormente, el objetivo de la práctica será minimizar lo máximo posible la desviación general de todos los clusters, además de hacer lo mismo con el término *infeasibility*, que no es más que el número de restricciones incumplidas. En el caso de la búsqueda local, la función objetivo a minimizar tendrá presente también un término (λ) que le dará relevancia a la *infeasibility*, quedando de la siguiente manera:

A minimizar $\rightarrow$ $f = C + (\text{infeasibility} * \lambda)$

Representación del problema

Para la representación del problema, se ha decidido crear una clase llamada PAR que tendrá en su interior todo lo necesario para operar y calcular las diferentes operaciones que se llevan a cabo.

En el constructor de la clase es necesario pasarle 2 variables como parámetros: el nombre del archivo del dataset y el nombre del archivo de restricciones del mismo dataset (puede ser el de 10% o el de 20%).

La clase posee atributos para almacenar los siguiente datos:

- La matriz de datos del dataset (*X*)

- El número de elementos (*data_size*)

- El número de clústers (*k*)

- Una lista de listas para los clusters (*clusters*)

- Una matriz de numpy para guardar todo el archivo de restricciones (R)

- Una lista para guardar solamente las restricciones que existen (aquellas que o son 1 o -1), llamada R_List

- El número de características (*dimensions)*

- Un array de booleans con tamaño *data_size* que se utilizará para saber si un elemento ha sido ya introducido en algún cluster (processed_X)

-Un array para los centroides (*centroids*).

A continuación se proporcionará en pseudo código los métodos de la clase que son comunes a ambos algoritmos

read_data se encargará de leer los archivos e introducir los datos leídos en los atributos anteriormente mencionados

read_data

Si archivo_de_datos != alguno de los archivos de la práctica
Levanta excepción
Abrir lector_de_archivos sobre archivo_de_datos
Inicializar X con los datos leídos y guardamos tamaño en data_size

Abrimos lector de archivos sobre archivo_de_restricciones
Para cada fila:
Para cada elemento:
Se guarda en R[fila][elemento]

Si es distinto de 0 y solo para la diagonal superior:
Guardamos elemento en R_Lista

Guardamos el número de características en dimensions

compute_centroid se encarga de calcular el centroide de un cluster dado

compute_centroid

Se crea un vector de tamaño dimensions, inicializado a ceros

Para cada elemento del cluster:

Se le suma el elemento al vector inicial

Se divide el vector por tamaño cluster

Se devuelve el vector

compute_intra_cluster_distance se encarga de calcular la distancia intra-cluster de un cluster dado

compute_intra_cluster_distance

Se crea una variable icd inicializada a 0

Para cada elemento del cluster:

Se le añade a icd la distancia entre cada elemento del cluster y su centroide

Se divide icd por el tamaño cluster

Se devuelve la variable

infeasibility calcula la infeasibilidad que tendría añadir un elemento dado a un cluster dado

infeasibility(element_index, cluster)

En la matriz R, se recorre la fila que corresponde al element_index:
Si la restricción != 0, el elemento ya se ha introducido en algún cluster y se ignora el valor de [element_index][element_index]:
Si la restricción == 1 y el elemento comparado no está en el cluster:
infeasibility += 1
Si la restricción == -1 y el elemento comparado está en el cluster:
infeasibility += 1
Se devuelve el valor de infeasibility

compute_general_infeasibility(S) calcula la infeasibilidad total de un conjunto de elementos asignados a clusters

compute_general_infeasibility(S)

Para cada restricción en R_List:
c1 es el cluster del primer elemento
c2 es el cluster del segundo elemento
si la restricción == 1 y c1 != c2:
infeasibility += 1
si la restricción == -1 y c1 == c2:
infeasibility += 1
Se devuelve infeasibility

initialize_centroids() inicializa los centroides aleatoriamente

initialize_centroids()

```
min = Primera fila de X
max = Primera fila de X
Para cada fila de X:
    Para cada elemento de fila:
        Si elemento < min[j]:
            min[j] = elemento
        Si elemento > max[j]
            max[j] = elemento

Para cada elemento de centroids:
    elemento = random (min, max)
```

initialize_clusters() inicializa los clusters y los deja vacíos

initialize_clusters

clusters = []

for i := 0 to k:

meter en clusters[i] una lista vacía

compute_objective_function(S, lambda) calcula el valor de la función objetivo para una organización de clusters dada y el valor lambda que servirá para dotar de importancia a la infeasibility

compute_objective_function (S, lambda)

desviacion_general := calcular_desviacion_general(S)

infeasibilidad := calcular_infeasibilidad_general(S)

funcion_objetivo := desviacion_general + infeasibilidad*lambda

devolver funcion_objetivo

Algoritmo de Búsqueda Local

compute_local_search()

inicializar_clusters()
S ← vector de int de tamaño data_size inicializado a ceros

Mientras** (true) **hacer
clusters_used ← vector de bool de tamaño k inicializado a False

Para** i:=0 **hasta** longitud(S) **hacer
cluster_index ← int random entre 0 y k
S[i] ← cluster_index

Si** clusters_used[cluster_index] == False **Hacer
clusters_used[cluster_index] = True

Si** todos los elementos de clusters_used == True **Hacer
Salir del bucle

processed_X ← vector de bool de tamaño data_size inicializado a True
EVALUATION_LIMIT ← 100000
mejor_entre_los_vecinos ← False
*lambda ← (máxima distancia del dataset*3.0) / (número de restricciones))*
current_eval ← compute_objective_function (S, lambda)
evaluaciones ← 1

Mientras** (evaluaciones < EVALUATION_LIMIT y mejor_entre_los_vecinos == False) **Hacer
mejor_entre_los_vecinos ← True
vecinos ← lista vacía
Para** i:= 0 hasta longitud(S) **hacer
Para** j:=0 hasta k **hacer
Si** S[i] != j **hacer
n ← lista con i, j
añadir n a vecinos
barajar (vecinos)

Para** i := 0 hasta longitud(vecinos) **hacer
copia_de_S ← S
cambio_cluster (copia_de_S, vecinos[i][0], vecinos[i][1])

```
                cluster_vacios ← hay_clusters_vacios(copia_de_S)

                Si cluster_vacios == False hacer
                    eval_vecino ← compute_objective_function(copia_de_S,
                                                             lambda)
                    evaluaciones += 1
                    Si eval_vecino < eval_actual hacer
                        cambio_cluster(S, vecinos[i][0], vecinos[i][1])
                        eval_actual ← eval_vecino
                        mejor_entre_los_vecinos ← False
                        Salir del bucle for
                    Si evaluaciones == EVALUATION_LIMIT hacer
                        Salir del bucle for

Imprimir por pantalla evaluaciones
Imprimir por pantalla desviación general
Imprimir por pantalla infeasibilidad general
Imprimir por pantalla el agregado final

Devolver S
```

Algoritmo Greedy COPKM

```
RSI ← lista de ints de 0 a data_size
barajar (RSI)
infeasibilidad_general ← 0

copia_centroides ← matriz de k filas y dimensions columnas inicializada a 0

Mientras copia_centroides != centroides hacer
        infeasibilidad_general ← 0
        copia_centroides ← centroides
        inicializar_clusters()
        processed_X ← vector de booleans de tamaño data_size inicializado a False

        Para i := 0 hasta longitud(RSI) hacer
                infeasibilidad_por_cluster ← lista vacía

                Para c := 0 hasta longitud(clusters) hacer
                        inf ← infeasibilidad (RSI[i], clusters[c])
                        añadir inf a infeasibilidad_por_cluster

                min_inf ← min(infeasibilidad_por_cluster)
                infeasibilidad_general += min_inf
                indice ← índice de min_inf en la lista infeasibilidad_por_cluster
                contador ← numero de veces que aparece min_inf en la lista
                                --infeasibilidad_por_cluster

                Si contador == 1 hacer
                        añadir i a clusters[indice]

                Si contador > 1 hacer
                        minima_distancia = 10000.0
                        indice_minima_distancia = indice

                        Para inf := 0 hasta longitud(infeasibilidad_por_cluster) hacer
                                Si infeasibilidad_por_cluster[inf] == min_inf hacer
                                        distancia ← distancia_euclidea(X[i],
                                                                centroides[i])
                                        Si minima_distancia > distancia hacer
                                                minima_distancia = distancia
```

indice_minima_distancia = inf

añadir i a clusters[indice_minima_distancia]
processed_X[i] ← True

Para** i := 0 **hasta** k **hacer
centroids[i] ← calcular_centroide(clusters[i])

Imprimir por pantalla infeasibilidad_general
Impirimir por pantalla desviacion_general()

Procedimiento y manual de usuario

Ambos algoritmos de esta práctica han sido implementados basándose en el pseudo-código proporcionado en las transparencias del seminario 2 además de en el propio guión de la práctica, donde se explican detalladamente los pasos a seguir y las variables a considerar. Aun así, se ha consultado en diversas páginas de Internet la manera de utilizar ciertas funciones importadas de diferentes librerías como leer archivos (la librería importada no reconocía la lectura de rutas relativas, por lo que simplemente se ha decidido poner tanto los archivos de los conjuntos de datos como las restricciones y el propio script todo en la misma carpeta), leer los argumentos a introducir en la consola, etc. O simplemente su explicación matemática, la cual ha servido para poder llevar a cabo su implementación en el código (por ejemplo, euclidean_distance(a, b)).

La manera de utilizar el script de la práctica es sencilla. Simplemente se debe abrir una terminal en la carpeta source_and_data y escribir el siguiente comando:

python PAR.py -d <conjunto de datos[1]> -r <porcentaje de restricciones[2]> -a <algoritmo[3]> [-o <fichero de salida>]

Posibles valores para los argumentos:

1. Conjunto de datos: ecoli / iris / rand

2. Porcentaje de restricciones: 10 / 20

3. Algoritmo: COPKM / BL

Análisis de resultados

Los experimentos con los algoritmos implementados consistían en minimizar lo máximo posible el valor de infeasibilidad y de la función objetivo dados un conjunto de datos y un número de restricciones asociadas a este. Estos conjuntos de datos eran el conjunto de datos Iris, el conjunto de datos Ecoli y el conjunto de datos Rand. A cada uno de ellos se le aplican dos conjuntos de restricciones, que representan el 10% y el 20% del total de restricciones permitidas para cada dataset. Lo cual quiere decir que hemos tenido que estudiar 12 casos distintos, 6 para cada algoritmo. Lo estudiaremos con detalle a continuación comparando por parejas las ejecuciones de los casos con las mismas características.

Comparación 1: Iris con 10% de restricciones

Se puede apreciar que para este conjunto de datos, el Greedy funciona bastante bien, dando casi siempre la mejor solución conjunta y cuando no la da, devuelve una que es muy buena, siempre en un tiempo inferior a 0.2s, lo cual es bastante aceptable teniendo en cuenta la calidad de las soluciones obtenidas. En cambio si nos pasamos al BL, la solución obtenida siempre será la misma, la mejor posible respetando las restricciones (0 infeasibilidad y la misma desviación general en todas las ejecuciones), pero todo esto a cambio de sacrificar un mayor tiempo de ejecución, cuya media se estabiliza sobre los 4,75s. En este caso (y veremos más adelante que con rand pasará algo parecido), pienso que sería bastante más recomendable usar el COPKM, ya que casi siempre encuentra la mejor solución en un tiempo considerablemente menor que el del BL. Eso sí, todo dependerá de lo que queramos sacrificar: posible bajón (bastante pequeño) de calidad en la solución o tiempo de ejecución

Comparación 2: Rand con 10% de restricciones

En este caso ocurre prácticamente lo mismo que en la comparación 1. En el Greedy, de 5 ejecuciones, en 4 se obtiene la mejor solución con un tiempo de ejecución bastante bajo mientras que en el BL siempre se obtiene la mejor solución pero con un tiempo de ejecución mucho peor (exactamente, el Greedy opera 24 veces más rápido que el BL), por lo que una vez más, es recomendable escoger el Greedy.

Comparación 3: Ecoli con 10% de restricciones

Aquí la cosa cambia. El dataset utilizado es bastante más complejo (mayor tamaño y por lo tanto mayor número de restricciones a considerar) por lo que será bastante más difícil encontrar la solución óptima. En este caso el Greedy se comporta bastante mal, dejándonos desviaciones bastante altas e infeasibilidades del orden de 230 aproximadamente, lo cual es bastante malo. Es una solución válida pero existen muchas

muchísimo mejores y, aunque el tiempo de ejecución sea bastante inferior, no nos compensa la diferencia en la calidad de las soluciones.

Se puede observar que, en cambio con el BL, llegamos a un conjunto de soluciones que supera con creces las obtenidas por el COPKM, con una tasa media de 23,0597 y una infeasibilidad media de 38,8, además del valor de la funcion objetivo, que es 26,1827. Aunque el tiempo ronde los 179s de media, como hemos comentado anteriormente, la solución del BL tiene una calidad claramente superior. Es por ello que, para el conjunto de datos Ecoli, el Búsqueda Local es mucho más recomendable.

Comparaciones 4, 5 y 6: Iris, Rand y Ecoli con 20% de restricciones

He decidido juntar estas tres comparaciones porque prácticamente se comportan de la misma manera que las previamente analizadas. Aunque podemos observar que esta vez, de 5 ejecuciones que hemos hecho con el conjunto de datos Iris usando el COPKM, solo 2 (el 40%) han conseguido la solución óptima. En el resto de las ejecuciones se han obtenido unas infeasibilidades que ya empiezan a preocupar en el ámbito de la calidad de la solución. Todo esto, como siempre, en un tiempo de ejecución bastante pequeño. El BL sigue encontrando la solución óptima en cada uno de los intentos, esta vez tardando un poquito más que en la comparación 1.

En el rand usando el COPKM, el 60% de las soluciones obtenidas es la solución óptima, dejándonos una media de infeasibilidad de 20 y una desviación general un poco mayor que la esperada, con un tiempo de ejecución que no sobrepasa los 0,2s nuevamente. En BL vuelve a encontrar para cada una de sus ejecuciones la solución óptima y esperada.

Finalmente, en el Ecoli las diferencias son bastante grandes ya, como en la comparación 3. La desviación típica es muy inferior y por lo tanto mucho mejor en el caso de la BL, dejándonos el Greedy con desviaciones del orden de 36,27 e infeasibilidades con una media de 205,2, por lo que nuevamente, sacrificando tiempo obtendremos mejores soluciones utilizando la BL. Aun así, se puede observar que las infeasibilidades obtenidas en este último se acercan a 100 e, incluso en una ejecución del Greedy se obtiene una infeasibilidad menor que algunas de la BL (89) pero en la media de todas se sigue observando una diferencia abismal.

Como podemos observar, el Greedy, a medida que el conjunto de datos es más complicado, con un mayor número de elementos y restricciones, nos va proporcionando soluciones que siguen siendo factibles pero que se van alejando cada vez más y más del óptimo esperado. Esto se debe a que simplemente se dedica a escoger la asignación que menos restricciones viole o la que más disminuya la desviación general. Entonce, por ejemplo en el Ecoli, llegará un momento que cuando vaya a meter los últimos elementos habiendo metido anteriomente 300 y pico, que haya muchas restricciones que se incumplan ya que no se ha estudiado el que sean lo mayor compatibles posible entre ellas sino que simplemente se coge la más aconsejable en ese momento. En cambio, en BL, se realiza una búsqueda exhaustiva en el entorno de cada solución generada. Esto quiere decir que siempre se supone que, cambiando un solo elemento de cluster (probando alguna combinación) obtendremos una solución mejor y, en cuanto la obtengamos, la cogemos y volvemos a hacer lo mismo. Así hasta encontrar aquella que sea mejor que todas sus posibles variantes. Este tipo de búsqueda es mucho más profunda y más compleja que aquella del COPKM. Es por ello que tarda mucho más pero te da la seguridad de encontrar una solución óptima (aunque esta dependa siempre de donde inicie la búsqueda).

En la siguiente página se pueden observar las tablas con los resultados obtenidos.

Tabla 6.1: Resultados obtenidos por el algoritmo COPKM con 10% restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,6693	0,0000	x	0,1973	38,1039	238,0000	x	19,0900	0,7573	0,0000	x	0,1893
Ejecución 2	0,6693	0,0000	x	0,1407	37,9897	213,0000	x	8,7103	0,7573	0,0000	x	0,1908
Ejecución 3	0,6726	25,0000	x	0,1817	37,5067	209,0000	x	46,7358	0,8161	29,0000	x	0,1399
Ejecución 4	0,6693	0,0000	x	0,1862	38,2893	270,0000	x	13,1059	0,7573	0,0000	x	0,1417
Ejecución 5	0,6640	14,0000	x	0,1924	38,7740	210,0000	x	12,9207	0,7573	0,0000	x	0,1381
Media	0,6689	7,8000	x	0,1797	38,1327	228,0000	x	20,1125	0,7691	5,8000	x	0,1599

Tabla 6.2: Resultados obtenidos por el algoritmo COPKM con 20% restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,6693	67,0000	x	0,2502	35,2416	252,0000	x	13,2560	0,8314	66,0000	x	0,1902
Ejecución 2	0,6764	45,0000	x	0,2517	36,5038	127,0000	x	3,2071	0,7573	0,0000	x	0,1843
Ejecución 3	0,6693	0,0000	x	0,1856	38,0438	205,0000	x	6,4223	0,7573	0,0000	x	0,1872
Ejecución 4	0,6869	78,0000	x	0,2514	36,7905	89,0000	x	6,3925	0,7804	36,0000	x	0,1869
Ejecución 5	0,6693	0,0000	x	0,1832	34,7680	353,0000	x	5,0253	0,7573	0,0000	x	0,1911
Media	0,6742	38,0000	x	0,2244	36,2695	205,2000	x	6,8607	0,7767	20,4000	x	0,1879

Tabla 6.3: Resultados obtenidos por el algoritmo BL en el PAR con 10% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,6693	0,0000	0,6693	4,9156	22,9673	41,0000	26,2674	149,5839	0,7573	0,0000	0,7573	3,3830
Ejecución 2	0,6693	0,0000	0,6693	5,3179	23,0418	36,0000	25,9394	220,9290	0,7573	0,0000	0,7573	4,3654
Ejecución 3	0,6693	0,0000	0,6693	4,8248	22,9603	32,0000	25,5359	190,1435	0,7573	0,0000	0,7573	3,6153
Ejecución 4	0,6693	0,0000	0,6693	4,2640	23,0888	42,0000	26,4693	203,9337	0,7573	0,0000	0,7573	4,5092
Ejecución 5	0,6693	0,0000	0,6693	4,4423	23,2404	43,0000	26,7014	133,0977	0,7573	0,0000	0,7573	3,3178
Media	0,6693	0,0000	0,6693	4,7529	23,0597	38,8000	26,1827	179,5376	0,7573	0,0000	0,7573	3,8381

Tabla 6.4: Resultados obtenidos por el algoritmo BL en el PAR con 20% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	0,6693	0,0000	0,6693	5,1244	22,8783	92,0000	26,5808	237,6340	0,7573	0,0000	0,7573	5,3074
Ejecución 2	0,6693	0,0000	0,6693	5,9844	23,0586	63,0000	25,5940	233,6539	0,7573	0,0000	0,7573	5,4323
Ejecución 3	0,6693	0,0000	0,6693	5,4329	22,8169	108,0000	27,1633	250,0628	0,7573	0,0000	0,7573	3,8605
Ejecución 4	0,6693	0,0000	0,6693	5,1300	23,1086	96,0000	26,9720	234,5912	0,7573	0,0000	0,7573	4,9657
Ejecución 5	0,6693	0,0000	0,6693	6,5873	22,9880	84,0000	26,3685	208,0203	0,7573	0,0000	0,7573	5,2393
Media	0,6693	0,0000	0,6693	5,6518	22,9701	88,6000	26,5357	232,7924	0,7573	0,0000	0,7573	4,9611

Tabla 6.5: Resultados globales en el PAR con 10% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,6689	7,8000	x	0,1797	38,1327	228,0000	x	20,1125	0,7691	5,8000	x	0,1599
BL	0,6693	0,0000	0,6693	4,7529	23,0297	38,8000	26,1827	173,5376	0,7573	0,0000	0,7573	3,8381
Algoritmo 1	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 2	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 3	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 4	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 5	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 6	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 7	x	x	x	x	x	x	x	x	x	x	x	x

Tabla 6.6: Resultados globales en el PAR con 20% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
COPKM	0,6742	38,0000	x	0,2244	36,2695	205,2000	x	6,8607	0,7767	20,4000	x	0,1879
BL	0,6693	0,0000	0,6693	5,6518	22,9701	88,6000	26,5357	232,7924	0,7573	0,0000	0,7573	4,9611
Algoritmo 1	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 2	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 3	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 4	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 5	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 6	x	x	x	x	x	x	x	x	x	x	x	x
Algoritmo 7	x	x	x	x	x	x	x	x	x	x	x	x