PRÁCTICA 1

Resolución del problema QAP usando el algoritmo Greedy y el algoritmo de búsqueda local del primer mejor

Información relativa al estudiante que ha realizado la práctica:

- **Nombre completo:** Guillermo Ramblado Carrasco
- **Curso:** 4º DGIIM
- **DNI:** 29628168M
- **Correo de contacto:** guilleramblado@correo.ugr.es
- Prácticas Miércoles (5:30-7:30)

Contenido

1. ¿En qué consiste el problema QAP a resolver?

Suponiendo que tenemos 'n' unidades y 'n' localizaciones, y conocemos el flujo existente entre cada pareja de unidades, y la distancia entre cada pareja de localizaciones, nuestro objetivo es asignar las unidades entre las diferentes localizaciones intentando realizar la mejor asignación posible.

¿A qué me refiero con 'mejor asignación posible'? A aquella cuyo coste total sea el menor posible, que en este caso equivale a conseguir aquella asignación que minimice la función indicada a continuación. Suponiendo que ***fij*** indica el flujo que circula entre la unidad 'i' y la unidad 'j', y ***dS(i)S(j)*** la distancia existente entre la localización a la que hemos decidido asignar la unidad 'i' (S(i)) y la localización a la que hemos asignado la unidad 'j' (S(j)) …

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} \cdot d_{S(i)S(j)}$$

2. Aplicación de los algoritmos a nuestro problema

Antes de nada, es importante indicar que he considerado interpretar la solución de dicho problema a nivel de código como un vector de enteros cuyo tamaño equivale al nº de unidades (o localizaciones) a asignar, es decir: ***vector<int> solución(n)***

El índice de cada componente del vector representará a la unidad 'i' ('n' unidades identificadas mediante enteros en [0,n-1]), tal que el valor almacenado en la componente 'i' será justamente la localización a la que hemos decidido asignar la unidad 'i.' Dichas localizaciones también se identificarán mediante enteros en [0,n-1].

En resumen, la solución no es más que una permutación del vector {0,1,2,3,…,n-1}.

¿Cómo podremos saber si una solución es mejor que otra?

Para ello, haremos uso de la función **fitness** (función objetivo a minimizar), cuyo pseudocódigo indicaré a posteriori, que se encarga de calcular el coste total de la asignación realizada, tal que una solución será de mayor calidad cuanto menor sea su valor fitness.

Pseudocódigo de la función **fitness**:

```
Funcion entero <- fitness (solución: vector de enteros por valor, matriz_flujo: vector de vectores de enteros por valor, matriz_distancia: vector de vectores de enteros por valor)
//Declaraciones
coste_total: entero = 0
distancia: entero = 0
flujo: entero = 0
n: entero = matriz_flujo.size()
//Cuerpo
Para i desde 0 hasta n-1 hacer:
        Para j desde 0 hasta n-1 hacer:
                distancia=matriz_distancia[solución[i]][solución[j]]
                flujo=matriz_flujo[i][j]
                coste_total+=distancia*flujo
Retornar coste_total
Fin Funcion
```

La primera forma de resolver el problema será aplicando el algoritmo Greedy, un algoritmo que nos permite construir la solución paso a paso, escogiendo en cada paso el elemento más prometedor, usando una determinada heurística que nos indica cuál es la mejor opción a nivel local.

Recordemos que Greedy elige siempre la mejor opción en cada paso local con la esperanza de llegar a una solución final óptima, pero no nos garantiza ni mucho menos obtener una solución óptima al problema.

En nuestro caso, la heurística a utilizar con Greedy será la siguiente:

Asociar unidades de gran flujo con localizaciones céntricas en la red y viceversa

Para conocer si una unidad tiene un mayor o menor flujo que otra, nos veremos en la obligación de calcular el potencial de flujo de cada unidad, que se calcula mediante la sumatoria del flujo que circula entre la unidad 'i' fija y el resto de unidades.

$$\hat{f}_i = \sum_{j=1}^{n} f_{ij} \quad ; \quad i = 1, \ldots, n$$

fi -> potencial de flujo de la unidad 'i'

fij -> flujo que circula entre la unidad 'i' y 'j'

n -> número de unidades

Por otro lado, necesitamos calcular también el potencial de distancia de una localización, que nos permitirá conocer si dicha localización es más o menos céntrica que otra, tal que cuanto menor sea su potencial de distancia, más centrada estará dicha localización con respecto a las demás.

$$\hat{d}_k = \sum_{l=1}^{n} d_{kl} \quad ; \quad k = 1, \ldots, n$$

dk -> potencial de distancia de la localización 'k'

dkl -> distancia entre la localización 'k' y 'l'

n -> número de localizaciones (=nº unidades)

Una vez calculado dichos potenciales de flujo y distancia, iremos asociando la unidad con mayor potencial de flujo con la localización más céntrica, hasta obtener la solución final.

La segunda forma que se plantea para resolver el problema es hacer uso del algoritmo de búsqueda local del primer mejor, que a diferencia de Greedy (que construye la solución final paso a paso) tomará una solución inicial e irá generando paso a paso el entorno de dicha solución inicial hasta que ocurra alguno de los siguientes casos:

-Se obtiene una solución vecina que es mejor que la actual: en este caso, la solución vecina sustituye a la actual y se continúa iterando, generando ahora el vecindario de esta nueva solución actual.

-Se construye el vecindario completo sin haber conseguido ninguna mejora: finaliza la ejecución del algoritmo.

Para este problema, implementaremos una técnica específica a la hora de generar los vecinos en la búsqueda local que más adelante detallaré, que nos permitirá reducir significativamente el tiempo de ejecución con una reducción muy pequeña de la eficacia de dicho algoritmo.

Además, la solución inicial a partir de la cual ejecuto mi algoritmo de búsqueda local se reduce a permutar una sola vez y de forma totalmente aleatoria los elementos del vector [0,1,…,n-1].

3. Descripción en pseudocódigo de los algoritmos empleados

3.1 ALGORITMO GREEDY

Primero indicaré el pseudocódigo del algoritmo Greedy y las diferentes funciones y estructuras utilizadas en dicho algoritmo…

-Pseudocódigo del registro que almacena los vectores de potenciales de flujo y distancias…

```
DosVectores: registro
potenciales_flujo: vector de enteros
potenciales_distancia: vector de enteros
Constructor DosVectores(n:entero)
        potenciales_flujo.resize(n)
        potenciales_distancia.resize(n)
Fin Registro
```

-Pseudocódigo de la función que se encarga de calcular los potenciales de flujo y distancias devolviéndolos como vectores de enteros en un registro de tipo DosVectores.

```
Funcion DosVectores <- potenciales (matriz_flujo: vector de vectores de enteros por valor,
matriz_distancia: vector de vectores de enteros por valor)
n: int = matriz_flujo.size()´
salida: DosVectores de tamaño n
flujo: int
distancia: int
Para i desde 0 hasta n-1 hacer
        Flujo=0
        Distancia=0
        Para j desde 0 hasta n-1 hacer
                Flujo+=matriz_flujo[i][j]
```

Distancia+=matriz_distancia[i][j]

Fin Para

Salida.potenciales_flujo[i]=flujo

Salida.potenciales_distancia[i]=distancia

Fin Para

Devolver salida

Fin Funcion

-Pseudocódigo de las funciones de comparación necesarias para ordenar los elementos con la función 'sort'.

Es importante recalcar que dichas funciones de comparación deben definir un orden estricto entre los elementos del vector, devolviendo 'true' cuando el elemento pasado como primer parámetro vaya antes que el segundo, y viceversa. Además, debe cumplir las siguientes propiedades:

-Irreflexiva: devuelve 'false' para dos elementos que sean iguales

-Antisimétrica: si a<=b y b<=a → a=b

La primera función de comparación se usará para ordenar los potenciales de flujo de mayor a menor. En el caso de que tomemos dos potenciales con el mismo valor, colocaremos primero aquel asociado a la unidad de menor índice.

Funcion booleano <- comparar_flujos (a:pareja de enteros,b:pareja de enteros)

Si a.second > b.second Entonces

Devolver true

Sino Si a.second==b.second Entonces

Devolver(a.fisrt>b.first)

Sino

Devolver falso

Fin Si

Fin Funcion

La segunda función de comparación nos permitirá ordenar los potenciales de distancia de menor a mayor, tal que devolverá 'true' si el potencial de distancia de 'a' es menor que el de 'b', colocando por tanto 'a' antes que 'b'.

Funcion booleano <- comparar_distancias (a:pareja de enteros,b:pareja de enteros)

Si a.second < b.second Entonces

Devolver true

Sino Si a.second==b.second Entonces

Devolver(a.fisrt>b.first)

Sino

Devolver falso

Fin Si

Fin Funcion

-Pseudocódigo del algoritmo Greedy…

Funcion Vector de enteros <- algoritmo_greedy(matriz_flujo: vector de vectores de enteros por valor, matriz_distancia: vector de vectores de enteros por valor)

Matrices: DosVectores = potenciales(matriz_flujo,matriz_distancia)

Potenciales_flujo: vector de enteros = matrices.potenciales_flujo

Potenciales_distancia: vector de enteros = matrices.potenciales_distancia

n:entero = potenciales_flujo.size()

pareja: pareja de enteros

flujos: vector de parejas de enteros

distancias: vector de parejas de enteros

Para i desde 0 hasta n-1 hacer

pareja.first=i

pareja.second=potenciales_flujo[i]

Flujos.push_back(pareja)

Pareja.second=potenciales_distancia[i]

Distancias.push_back(pareja)

Fin Para

Sort(flujos.begin(),flujos.end(),comparar_flujos)

Sort(distancias.begin(),flujos.end(),comparar_distancias);

Solución: vector de enteros

Solución.resize(n)

Para i desde 0 hasta n-1 hacer

Solución[flujos[i].first] <- distancias[i].first

Fin para

Devolver solucion

Fin Funcion

3.2 ALGORITMO DE BÚSQUEDA LOCAL DEL PRIMER MEJOR

Ahora procederé a implementar el pseudocódigo del algoritmo de búsqueda local del primero mejor.

Comenzaremos indicando el **pseudocódigo de la función 'diferencia_coste'**, que nos permite calcular la diferencia de coste entre dos soluciones obtenidas mediante la BL, factorizando, para que dicha operación sea lo menos costosa posible.

Funcion entero <- diferencia_coste (solución_actual: vector de enteros, r: entero, s: entero, matriz_flujo: vector de vectores de enteros, matriz_distancia: vector de vectores de enteros)

n: entero <- solucion_actual.size()

assert(r>=0 y r<n y r!=s)

assert(s>=0 y s<n)

diferencia: entero <- 0

Para i desde 0 hasta n-1 hacer

Si i!=r y i!=s Entonces

diferencia+=matriz_flujo[r][i]*(matriz_distancia[solucion_actual[s]][solucion_actual[i]]-matriz_distancia[solucion_actual[r]][solucion_actual[i]])+

matriz_flujo[s][i]*(matriz_distancia[solucion_actual[r]][solucion_actual[i]]-matriz_distancia[solucion_actual[s]][solucion_actual[i]])+

matriz_flujo[i][r]*(matriz_distancia[solucion_actual[i]][solucion_actual[s]]-matriz_distancia[solucion_actual[i]][solucion_actual[r]])+

matriz_flujo[i][s]*(matriz_distancia[solucion_actual[i]][solucion_actual[r]]-matriz_distancia[solucion_actual[i]][solucion_actual[s]]);

Fin Si

Fin Para

Fin Funcion

Ahora indico el **pseudocódigo de 'busqueda_vecindario'**, función de la que haremos uso dentro de la función 'busqueda_local' y que se encarga de generar paso a paso el vecindario de la solución actual que consideremos en cada momento.

Para nuestro problema, el vecino generado equivale a intercambiar entre sí las localizaciones a las que se han asignado dos unidades para la solución actual.

```
Funcion vector de enteros <- búsqueda_vecindario (dlb:vector de booleanos Por Referencia,
solucion_actual: vector de enteros, matriz_flujo: vector de vectores de enteros,
matriz_distancia: vector de vectores de enteros)

n:entero <- dlb.size()

mejora:booleano <- false

Para i desde 0 hasta n-1 Hacer

        Si ¡dlb[i] Entonces

                mejora <- false

                Para j=0 hasta j=n-1 Hacer

                        Si j==i continuar:

                        Si
diferencia_coste(solucion_actual,i,j,matriz_flujo,matriz_distancia)<0 Entonces

                                swap(solucion_actual[i],solucion_actual[j])

                                dlb[i]=false

                                dlb[j]=false

                                mejora=true


                        Fin Si

                Fin Para

                Si (no mejora) Entonces

                        Dlb[i]=true

                Fin Si

Devolver solucion_actual

Fin Funcion
```

Por último, y antes de proceder con la búsqueda local, será necesario disponer de una **función que genera una primera solución inicial de forma aleatoria**, siendo necesario indicar la semilla a usar para generar el generador de números aleatorios que posteriormente utilizaremos para permutar los elementos de un conjunto y obtener así la solución inicial.

Funcion entero <- myrandom(i:entero)

Devolver rand()%i

Fin Funcion

Funcion vector de enteros <- generar_solucion_aleatoria (semilla: entero sin signo, tamanio: entero)

Solucion_inicial: vector de enteros de tamaño 'tamanio'

Iota(solucion_inicial.begin(),solucion_inicial.end(),0)

Srand(semilla)

Random_shuffle(solucion_inicial.begin(),solucion_inicial.end(),myrandom)

Devolver solucion_inicial

Fin Funcion

Ahora procederemos a indicar el **pseudocódigo de lo que es la propia búsqueda local**, haciendo uso de las funciones indicadas anteriormente

Funcion vector de enteros <- buqueda_local (matriz_flujo: vector de vectores de enteros, matriz_distancia: vector de vectores de enteros,semilla: entero sin signo)

Solucion_actual: vector de enteros <- generar_solucion_aleatoria(semilla,matriz_flujo.size())

Dlb: vector de booleanos de tamaño 'matriz_flujo.size()' y elementos a 'false'

Solucion_obtenida: vector de enteros

Seguir_iterando: booleano <- false

Hacer

Solucion_obtenida <- búsqueda_vecindario(dlb,solucion_actual,matriz_flujo,matriz_distancia)

Seguir_iterando<- (solucion_obtenida!=solucion_actual)

Solucion_actual <- solucion_obtenida

Mientras que (seguir_iterando)

Devolver solucion_actual

Fin Funcion

4. Desarrollo de la práctica

La práctica ha sido realizada en C++ usando VisualStudioCode. Se ha creado un Makefile que se encargará de enlazar los archivos correctamente y generar el ejecutable, siendo necesario pasar como parámetro la semilla a utilizar a la hora de generar el generador de números aleatorios, utilizado para obtener una solución inicial a la hora de aplicar BL, permutando de forma aleatoria los elementos del vector {0,1,2,…,n-1}.

5. Experimentos y análisis de resultados

Voy a proceder a solucionar el problema de QAP planteado en cada uno de los 20 ficheros de datos usando tanto Greedy como Búsqueda Local del Primer Mejor.

En el caso de la búsqueda local, he utilizado una semilla de valor 15.

→ En las dos siguientes tablas adjunto los resultados obtenidos, indicando…

1. El coste o valor fitness de dicha solución (recordemos que a menor valor fitness, mayor calidad)
2. Lo próxima que se encuentra la solución obtenida de la mejor solución encontrada hasta el momento para dicho problema
3. El tiempo empleado para conseguir la solución, expresado en milisegundos.

Algoritmo Greedy			
Caso	**Coste obtenido**	**Fitness**	**Tiempo (milisegundos)**
Chr22a	12138	97.17	0.023
Chr22b	13280	114.40	0.014
Chr25a	20414	437.78	0.018
Esc128	154	140.63	0.228
Had20	7512	8.52	0.01
Lipa60b	3218450	27.71	0.02
Lipa80b	10037083	29.28	0.032
Nug28	6348	22.88	0.012
Sko81	105828	16.30	0.031
Sko90	131450	13.78	0.034
Sko100a	172116	13.23	0.038
Sko100f	170472	14.38	0.039
Tai100a	23936546	13.70	0.041
Tai100b	1574846615	32.79	0.039
Tai150b	623469733	24.97	0.147
Tai256c	98685678	120.48	0.401
Tho40	312934	30.11	0.02

Tho150	9527466	17.14	0.125
Wil50	54670	11.99	0.02
Wil100	293152	7.37	0.053

Algoritmo Búsqueda Local del Primero Mejor (semilla=15)			
Caso	**Coste obtenido**	**Fitness**	**Tiempo (milisegundos)**
Chr22a	7062	14.72	2.921
Chr22b	7096	14.56	1.821
Chr25a	6006	58.22	3.476
Esc128	84	31.25	222.221
Had20	7160	3.44	0.718
Lipa60b	3021770	19.91	29.416
Lipa80b	9464956	21.91	74.773
Nug28	5370	3.95	2.126
Sko81	93296	2.53	128.511
Sko90	118324	2.41	198.389
Sko100a	155644	2.40	269.38
Sko100f	152144	2.09	286.017
Tai100a	21893816	4.00	144.287
Tai100b	1248421045	5.26	249.9
Tai150b	522499335	4.73	1079.89
Tai256c	44957496	0.44	4120.56
Tho40	255866	6.38	13.69
Tho150	8339394	2.53	1194.14
Wil50	49538	1.48	25.942
Wil100	276206	1.16	234.353

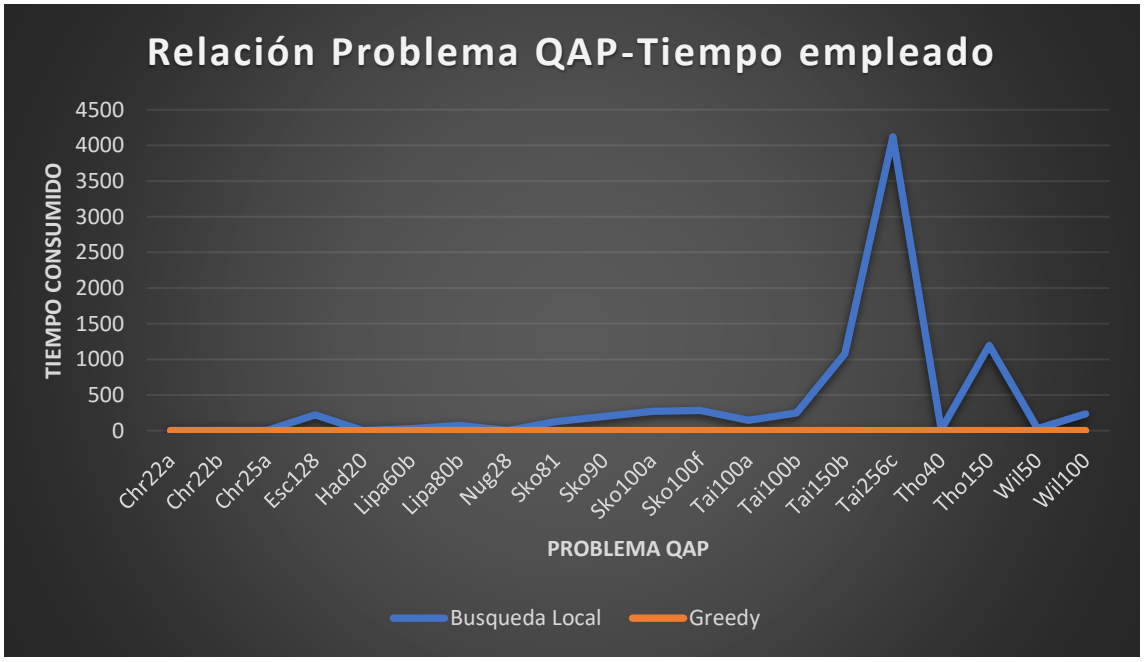
Vamos a calcular el tiempo medio necesario para solventar el problema QAP y el coste medio obtenido para cada algoritmo utilizado.

Algoritmo	Media fitness	Tiempo medio
Greedy	59.73	0.06725
Búsqueda Local del Primero Mejor	10.1685	414.12655

Analizando los resultados obtenidos, se puede apreciar que Greedy es mucho más rápido que BL, aunque la solución obtenida es de mucha peor calidad que la que nos ofrece la búsqueda local.

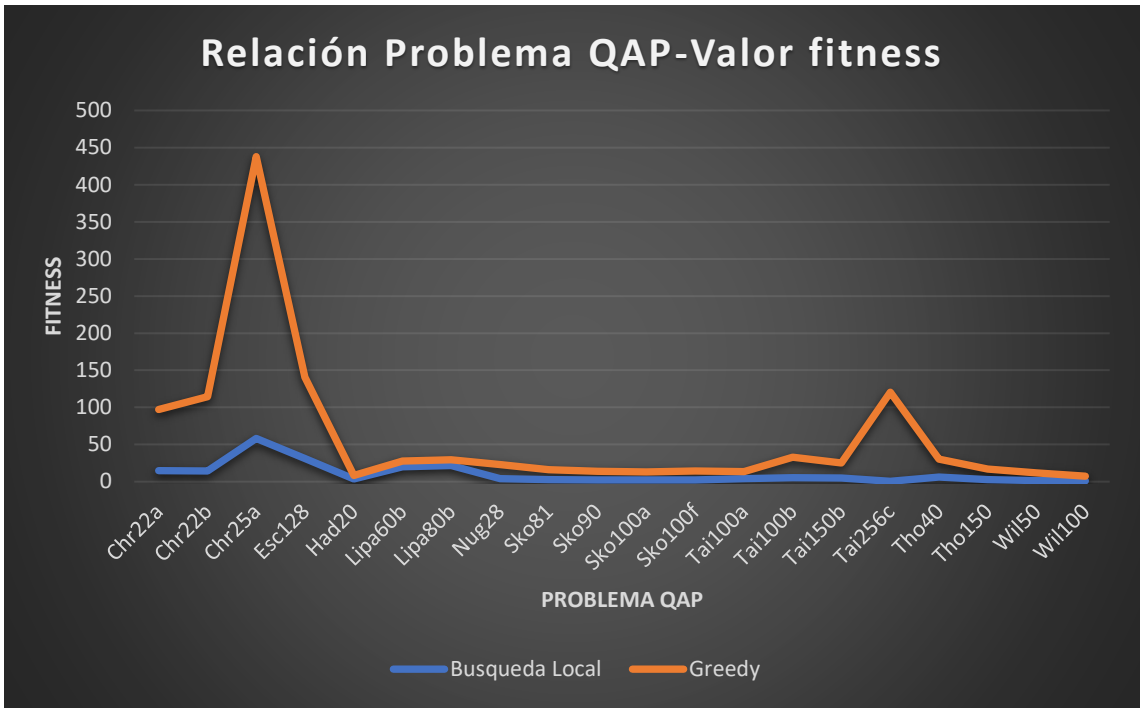
Además, es importante recalcar que los tiempos obtenidos al aplicar búsqueda local hubiesen sido mayores de no ser por la técnica utilizada 'Don´t look bits', que nos ha permitido focalizar la búsqueda en una zona del espacio en la que potencialmente pueda ocurrir algo.

Vamos a apoyarnos en una serie de gráficos que nos permiten relacionar ambos algoritmos de búsqueda en función del tiempo que consumen y del valor fitness de la solución calculada.



El anterior gráfico de líneas nos permita visualizar el tiempo consumido por los dos algoritmos de búsqueda para cada problema QAP propuesto. Se puede observar sin ninguna dificultad que el tiempo consumido por la búsqueda local es muy superior al que consume Greedy, que es muchísimo más veloz.

Sin embargo, la búsqueda local nos permite aproximarnos mucho más a la mejor solución encontrada hasta el momento para cada problema QAP, a diferencia de Greedy, que nos devuelve soluciones lejanas a la óptima, tal y como se visualiza en el siguiente gráfico



En cuanto a la búsqueda local, podríamos haber utilizado diferentes estrategias para obtener esa solución de partida, como por ejemplo realizar varias permutaciones sobre el vector [0,1,…,n-1] y tomar como solución de partida aquella que nos ofrezca un menor valor fitness. Esto claramente supondría un incremento del tiempo necesario para obtener la solución final, por lo que sería vital cuidar el número de soluciones iniciales que queremos generar y comparar entre sí para decidir de cuál partir sin que afecte demasiado al tiempo total consumido.

Otra posible estrategia sería tomar como solución inicial la construida por el algoritmo Greedy, aprovechando el hecho de que dicho algoritmo de búsqueda es muy rápido y nos aseguraría no producir un cambio notable en el tiempo total consumido tras aplicar la búsqueda local.

El utilizar una estrategia u otra es lo que nos permitirá partir de una solución inicial más próxima o más alejada de la solución óptima al problema, y por ende, conseguir una solución de mayor o menor calidad, existiendo el riesgo de caer en un mínimo local (mejor solución pero a nivel local, dentro de su vecindario o entorno).