UNIVERSIDAD
DE GRANADA

METAHEURÍSTICAS

GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO FINAL

Social Engineering Optimization(SEO)

Autor
Fernando Vallecillos Ruiz

DNI
77558520J

E-Mail
nandovallec@correo.ugr.es

Grupo de prácticas
MH1 Miércoles 17:30-19:30

Rama
Computación y Sistemas Inteligentes

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

CURSO 2019-2020

Índice

1. Descripción del problema

El problema que vamos a analizar en esta práctica se trata del Agrupamiento con Restricciones (PAR). Este es una variación del problema de agrupamiento o *clustering*, el cual persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos. De esta forma, el agrupamiento es una técnica de aprendizaje no supervisado ya que permite descubrir grupos (no conocidos) en un conjunto de datos y agrupar los datos similares. Cabe destacar que no es un algoritmo en específico, sino un problema pendiente de solución. Existen multitud de algoritmos que resuelven este problema los cuales difieren en su definición de *cluster* y su método de búsqueda.

El problema de Agrupamiento con Restricciones es una generalización del problema de agrupamiento. Incorpora al proceso nueva información: las restricciones. Esto provoca un cambio en el tipo de tarea que pasa a ser semi-supervisada. Intentaremos encontrar una partición $C = \{c_1, c_2, \ldots, c_k\}$ del conjunto de datos X con n instancias que minimice la desviación general y cumpla con las restricciones en el conjunto R.

1.1. Variantes del problema

Variantes según tipo de restricciones[1]:

- *Cluster-level constraints*: se definen requerimientos específicos a nivel de *clusters* como:
  - Número mínimo/máximo de elementos
  - Distancia mínima/máxima de elementos
- *Instance-level constraints*: se definen propiedades entre pares de objetos tales como la pertenencia o no de elementos al mismo cluster

Variantes según la interpretación de las restricciones:

- Basados en métricas(*metric-based*): El algoritmo de *clustering* utiliza una distancia métrica. Esta métrica sera diseñada para que cumpla con todas las restricciones.
- Basados en restricciones(*constraint-based*): El algoritmo es modificados de manera que las restricciones se utilizan para guiar el algoritmo hacia una partición C más apropiada. Se lleva a cabo mediante la modificación de la función objetivo del *clustering*.

1.2. Formalización del problema

El conjunto de datos es una matriz X de $n \times d$ valores reales. El conjunto de datos esta formado por n instancias en un espacio de d dimensiones notadas como:

$$\vec{x_i} = \{x_{[i,1]}, \cdots, x_{[i,d]}\}$$

Un *cluster* c_i consiste en un subconjunto de instancias de X. Cada *cluster* c_i tiene asociada una etiqueta (nombre de *cluster*) l_i.

Para cada *cluster* c_i se puede calcular su centroide asociado:

$$\vec{\mu_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \vec{x_j}$$

A partir de estos, se puede calcular la distancia media intra-cluster:

$$\overline{c_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \|\vec{x_j} - \vec{\mu_i}\|$$

Para calcular la desviación general de la partición C podemos:

$$\overline{C} = \frac{1}{k} \sum_{c_i \in C} \overline{c_i}$$

Dada una partición C y el conjunto de restricciones, definimos *infeasibility* como el número de restricciones que C incumple. Definimos $V(\vec{x_i}, \vec{x_j})$ como la función que, dada una pareja de instancias, devuelve 1 si viola una restricción y 0 en otro caso:

$$infeasibility = \sum_{i=0}^{n} \sum_{j=0}^{n} V(\vec{x_i}, \vec{x_j})$$

Entonces, podemos formularlo como:

$$\text{Minimizar } f = \overline{C} + (infeasibility * \lambda)$$

donde λ es un parámetro de escalado para dar relevancia a *infeasibility*. Si se establece correctamente, el algoritmo optimiza simultáneamente el número de restricciones incumplidas y la desviación general.

2. Descripción de los algoritmos

2.1. Consideraciones previas

Antes de realizar una descripción formal de cada algoritmo, necesitamos describir aspectos comunes entre estos: esquema de representación de soluciones, pseudocódigo de operadores comunes o la función objetivo.

Comenzaremos hablando de las estructuras de datos utilizadas. Se ha utilizado una matriz X de $n \times d$ para guardar los datos. Se ha querido aprovechar al máximo la capacidad de paralelización de operaciones, por ello se optó por añadir las columnas necesarias a X para concentrar en una sola estructura todos los datos necesarios para calcular nuestros parámetros. Por ello, en cada algoritmo se añadirán columnas diferentes dependiendo de las necesidades de este. No obstante, existen dos columnas añadidas comunes a cualquier algoritmo. Estas son la columna *cluster*, la cual representa a que *cluster* pertenece la instancia. Y la columna *Distance to Cluster (DC)* que representa la distancia de la instancia al centroide de su cluster. De aquí en adelante, serán referidas como $x_{cluster}$ y x_{DC} respectivamente.

Para los centroides se ha escogido una matriz M de $k \times d$ siendo k el número de *clusters*. De esta forma, para cualquier instancia x, $x_{cluster}$ indicara el índice en la matriz M. Por ello, M puede ser referido como vector de centroides. Para calcularlos, se han calculado los vectores promedios de las instancias.

Algorithm 1: Calcular centroides

```
1 CalcularCentroides (X)
      input  : Matriz de datos X
      output: Vector de centroides M
2     foreach x ∈ X do
3         sum_{l_{x_i}} ← sum_{l_{x_i}} + x
4         count_{l_{x_i}} ← count_{l_{x_i}} + 1
5     centroids = sum/count
6     return centroids
```

También se ha optado por representar el conjunto R de restricciones en una matriz de $n \times n$. Vamos a contar con solo 2 tipos de restricciones *Must-Link (ML)* y *Cannot-Link (CL)*. Dada una pareja de instancias, se establece una restricción *ML* si deben pertenecer al mismo *cluster* y de tipo *CL* si no pueden pertenecer al mismo *cluster*. Se representa una restricción *ML* entre las instancias x_i e x_j si $R[i][j] == 1$. Por otra parte, se representa una restricción *CL* entre las instancias x_i e x_j si $R[i][j] == -1$. Como se puede ver, la operación para calcular *infeasibility* dado

una matriz de datos X es trivial pero explicaremos brevemente su implementación y modularización.

Para calcular *infeasibility* dado una matriz de datos X, podríamos mirarlo como la suma de *infeasibility* de cada instancia $x \in X$.

Algorithm 2: Calcular Infeasibility

```
1  CalcularInfeasibility (X)
       input  : Matriz de datos X
       output: Valor de infeasibility
2      total ← 0
3      for i ← 0 to N − 1 do
4          total ← total + CalcularInfeasibilityRowPartial(X, i)
5      return total
```

Se necesita entonces, calcular el valor de *infeasibility* para cualquier $x \in X$. Paro ello, se recorren las restricciones asociadas a la instancia seleccionada. Se descarta la restricción de una instancia consigo misma. Si la restricción es de tipo *ML* o *CL* se comprueba si se cumple o no, acumulando puntos si fuese necesario.

Algorithm 3: Calcular Infeasibility Row

```
1  CalcularInfeasibilityRow (X, r)
       input  : Matriz de datos X, Índice r
       output: Valor de infeasibility asociado a x_r
2      total ← 0
3      for i ← 0 to N − 1 do
4          if R_r[i] == 1 then
5              if X[i]_cluster ≠ X[r]_cluster then
6                  total ← total + 1
7          else if R_r[i] == −1 then
8              if X[i]_cluster == X[r]_cluster then
9                  total ← total + 1
10     return total
```

Sin embargo, si se llamase a esta función para calcular *infeasibility* de toda la matriz X, se acabaría con el doble de puntuación. Este problema se puede solucionar de forma fácil devolviendo solo la mitad del valor acumulado. Subyace un problema de eficiencia, al ser una matriz, se calculan todas las inversas de las restricciones. Por ello, se puede crear una función para calcular de forma parcial el valor de una fila. Esta función sera llamada solo cuando queramos calcular

infeasibility de todo el conjunto.

Algorithm 4: Calcular Infeasibility Row Partial

```
1  CalcularInfeasibilityRowPartial (X, r)
       input  : Matriz de datos X, Índice r
       output: Valor de infeasibility asociado a x_r
2      total ← 0
3      for i ← r + 1 to N − 1 do
4          if R_r[i] == 1 then
5              if X[i]_cluster ≠ X[r]_cluster then
6                  total ← total + 1
7          else if R_r[i] == −1 then
8              if X[i]_cluster == X[r]_cluster then
9                  total ← total + 1
10     return total
```

Se describen a continuación las funciones necesarias para calcular los parámetros de la función objetivo. Como se ha descrito antes, esta tendrá la siguiente forma:

$$f = \overline{C} + (infeasibility * \lambda)$$

El valor de λ se ha calculado como el cociente entre la mayor distancia entre dos instancias del conjunto y el número de restricciones:

$$\lambda = \frac{\lceil D \rceil}{|R|}$$

El último parámetro es $\overline{C}$. Para calcularlo, primero se asignará la distancia desde cada instancia $x \in X$ a su *cluster*. Para ello, simplemente se calcula la

distancia euclídea entre ambos puntos y se asigna a su columna correspondiente.

Algorithm 5: Calcular Distancia a Centroides

```
1 CalcularDistanceCluster (X, M)
      input : Matriz de datos X, Vector de Centroides M
      output: Matriz de datos X
2     foreach x ∈ X do
3         a ← x
4         b ← M [x_cluster]
5         x_DC ← sqrt(Σ_{i=1}^{d} (a_i − b_i)^2)
6     return X
```

Con las distancias a mano, se puede calcular cada $\overline{c_i}$ y por tanto $\overline{C}$ como la media de estos.

Algorithm 6: Calcular Desviación General

```
1 CalcularDesviacion (X)
      input : Matriz de datos X
      output: Valor de Desviación General average
2     sum_{l_{x_i}} ← [k]
3     count_{l_{x_i}} ← [k]
4     foreach x ∈ X do
5         sum_{l_{x_i}} ← sum_{l_{x_i}} + x_DC
6         count_{l_{x_i}} ← count_{l_{x_i}} + 1
7     average = mean(sum/count)
8     return average
```

Como se puede ver, los dos últimos algoritmos están muy entrelazados entre si. Han sido escritos de forma separada para comprender su comportamiento de una forma fácil y simple. Sin embargo, han sido implementado de forma conjunta para mejorar la eficiencia temporal.

Con esto, se terminan las consideraciones comunes de los algoritmos usados. Se pasará entonces a explicar cada algoritmo en mayor profundidad.

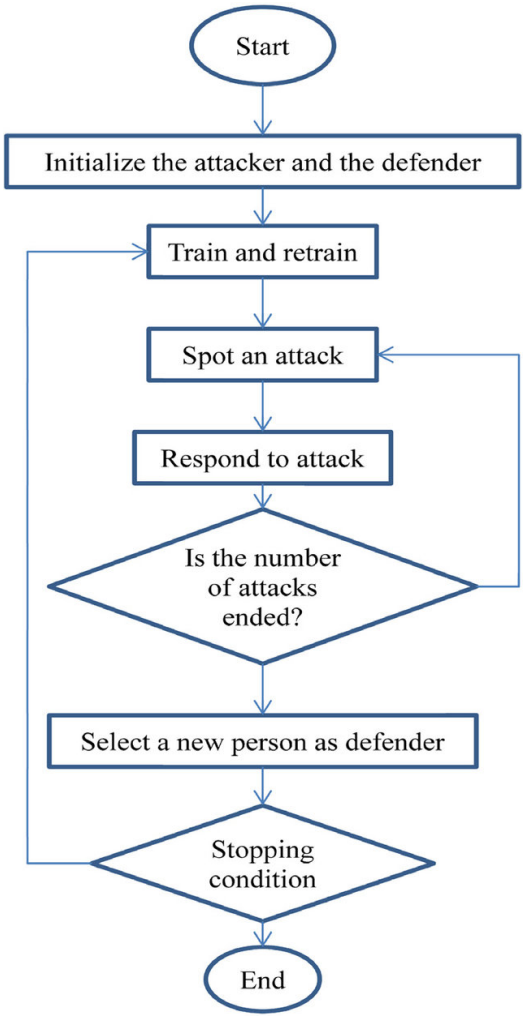
3. *Social Engineering Optimizer*

El algoritmo usado será *Social Engineering Optimizer*(SEO). Este algoritmo se basa en los fundamentos de la ingeniería social para idear un algoritmo basado en

individuos. La ingeniería social se define como ataques indirectos con el objetivo de conseguir cierta información a través de múltitud de técnicas. Este algoritmo se destaca por:

- Basado en individuos y no en poblaciones.
- Simple de comprender con solo 4 fases y 3 parámetros.
- Da la oportunidad de escoger entre 4 diferentes técnicas.

Como se puede apreciar este algoritmo se puede adaptar a multitud de problemas gracias a sus parámetros y diferentes técnicas. También permite añadir nuevas técnicas con facilidad gracias a su simple estructura. En este algoritmo cada solución será identificada como una persona y sus diferentes rasgos. El algoritmo se compone de dos soluciones (generadas de forma aleatoria inicialmente) las cuales serán denominadas **defensor** y **atacante**. Primero se simula una fase de entrenamiento en la cuál el defensor se prepara para una amenaza por el atacante. Esto se realiza mediante una búsqueda local la cual copiará rasgos del atacante. En la siguiente fase, el defensor intercepta los diferentes ataques moviéndose en espacio de búsqueda. Si en algún momento el defensor fuese mejor que el atacante, se intercambiarían los papeles. Tras un número determinado de ataques el defensor será destruido, y un nuevo defensor será generado de forma aleatoria.



Se realiza una búsqueda local/explotación en la fase de entrenamiento. Además, interceptando los ataques se intensifica la búsqueda. La fase de exploración se realiza creando nuevos defensores. Se pasa a discutir las fases y técnicas con mayor detalle.

3.1. Discretización

Antes de comenzar a discutir el las fases se debe indicar el proceso realizado para discretizar los valores. La metaheurística ha sido diseñada para usarse con variables con dominios reales. Sin embargo, una solución se compone de valores en un dominio finito. Cada punto debe pertenecer a uno de los k *clusters* posibles. Para este objetivo, se eligen los dominios reales en el rango $[0, 1]$. Se divide entonces este rango en k subrangos los cuales se mapean a los valores indicados en orden

ascendiente. Se mantendrá un nuevo vector con los valores reales en cada momento del defensor y atacante. Estos podrán ser transformados en soluciones factibles del problema, a partir de las cuáles se generarán las soluciones factibles.

3.2. Inicializar atacante y defensor

En esta fase se generan dos soluciones aleatorias. Una para el defensor y otra para el atacante. La mejor de estas dos soluciones será asignada como atacante.

3.3. Entrenamiento

En la fase de entrenamiento se realiza una búsqueda local basándose en los rasgos del atacante. Para esto se escoge un rasgo aleatorio el cuál será sustiuido por el mismo valor del atacante. Está búsqueda sigue un esquema primero el mejor. Por lo que se generan un total de N_T soluciones nuevas. De estas soluciones se escoge aquella con mejor función objetivo. El valor N_T será calculado de la siguiente forma:

$$N_T = \alpha * nVar$$

Siendo α uno de los parámetros de entrada $\in [0, 1]$ y $nVar$ el número de rasgos en una persona.

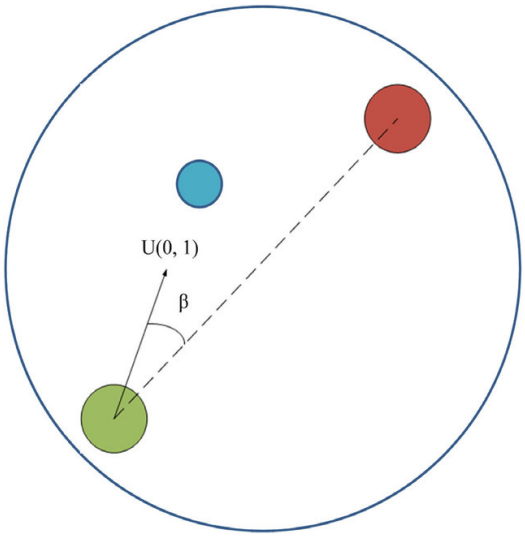
3.4. Interceptar ataque

Se consideran cuatro diferentes técnicas. Estas serán utilizadas con variables aleatorias para explorar el espacio de búsqueda. Todas tienen utilizan el parámetro β el cual será definido como parámetro de entrada $\in [0, 1]$. También se utiliza la notación $U(0, 1)$ para indicar un número aleatorio en una distribución uniforme. Para visualizar los efectos se añaden imágenes que muestran el movimiento del defensor en el espacio. El defensor se indica como un círculo verde, el atacante como un círculo rojo y el nuevo defensor (tras interceptar el ataque) como un círculo azul. Se describen las técnicas a continuación:

3.4.1. *Obtaining*

En esta técnica, el atacante utiliza el defensor de forma directa como guía. Se genera una nueva solución de la siguiente forma:

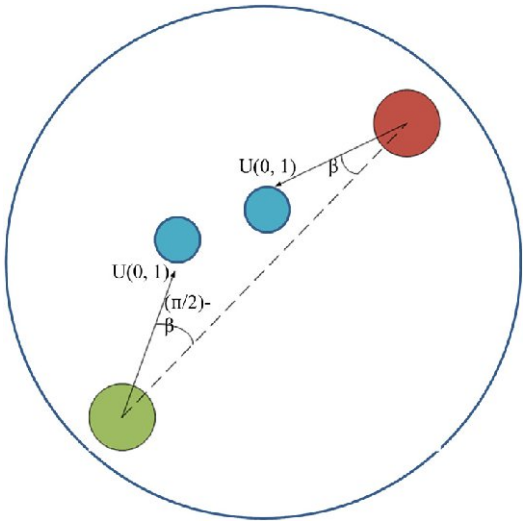
$$def_{new} = def_{ori} * (1 - sin\beta * U(0,1)) + \frac{def_{old}+att}{2} * sin\beta * U(0,1)$$



3.4.2. *Phising*

En esta técnica, el atacante finge acercarse al defensor para que este se mueva a otra posición. De está forma, la técnica genera dos posibles soluciones:

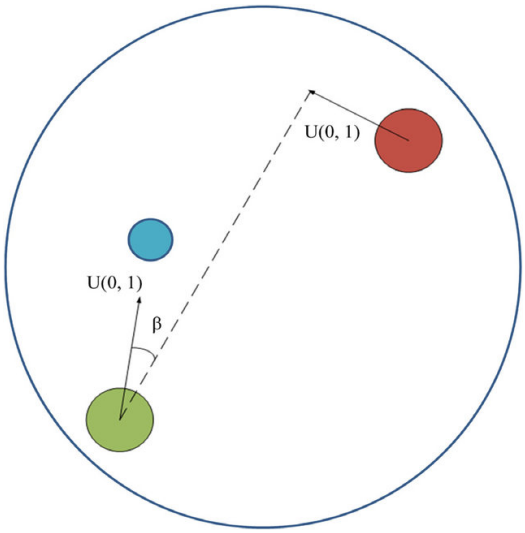
$$def_{new1} = att * (1 - sin\beta * U(0,1)) + \frac{def_{old}+att}{2} * sin\beta * U(0,1)$$
$$def_{new2} = def_{old} * (1 - sin(\frac{\pi}{2} - \beta) * U(0,1)) + \frac{def_{old}+att}{2} * sin(\frac{\pi}{2} - \beta) * U(0,1)$$



3.4.3. *Diversion Theft*

En esta técnica, el atacante guía al defensor a una posición favorable para el ataque. Se genera una nueva solución de la siguiente forma:

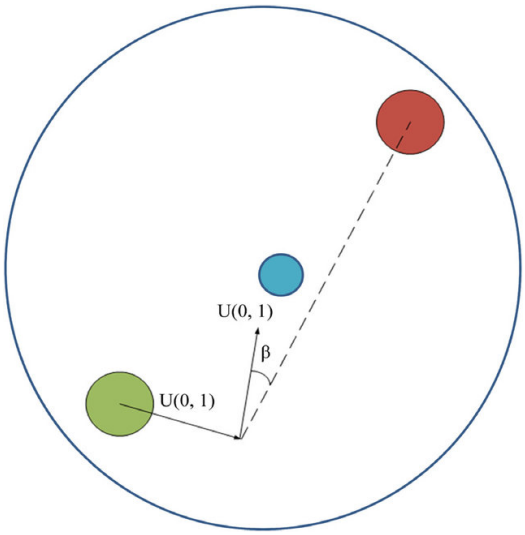
$$def_{new} = def_{ori} * (1 - sin\beta * U(0,1)) + \frac{def_{old} + att * sin(\frac{\pi}{2} - \beta) * U(0,1)}{2} * sin\beta * U(0,1)$$



3.4.4. *Pretext*

En esta técnica, el atacante engaña al defensor con un señuelo dependiendo de los rasgos de este para guiarlo a una nueva posición. Se genera una nueva solución de la siguiente forma:

$$def_{new} = (def_{ori} * sin(\frac{\pi}{2} - \beta) * U(0,1)) * (1 - sin\beta * U(0,1)) + \frac{att + def_{ori} * sin(\frac{\pi}{2} - \beta) * U(0,1)}{2} * sin\beta * U(0,1)$$



3.5. Responder al ataque

En esta fase simplemente se comparan los nuevos defensores con el original. Si el nuevo defensor (o defensores) es mejor que el original, se sustituye antes de realizar el siguiente ataque. Además, si el defensor es mejor que el atacante, se intercambian los papeles.

3.6. Nuevo defensor y condición de parada

Tras realizar los ataques se genera un nuevo defensor de forma aleatoria desechando el antiguo. Se comprueba entonces la condición de parada. Esta puede ser tiempo consumido, número de iteraciones, número de evaluaciones, etc. La mejor solución siempre habrá sido guardada en el atacante.

4. Desarrollo

La práctica se ha implementado en **Python3** y ha sido probada en la versión 3.6.9. Por tanto, se recomienda encarecidamente utilizar un intérprete de Python3. Se han utilizado diferentes paquetes con funciones de utilidad general: el módulo **time** para la medición de tiempos, el módulo **random** para generar números pseudoaleatorios, el módulo **scipy** para calcular λ, el módulo **math** para el cálculo de distancias euclidianas y el módulo **numpy** para gestionar matrices de forma eficiente.

En cuanto a la implementación, se ha creado un solo programa **main.py** el cual contiene todo el código necesario para realizar la búsqueda SEO. Se realizan todos las operaciones comunes y no relevantes (carga de datos, restricciones, semillas para números aleatorios) fuera del código en el cual se realizan medidas temporales. Se ha creado otro programa **comparison.py** para realizar las comparaciones. Este programa lanzará *main.py* múltiples veces y escribirá las medidas en un fichero '.csv'.

El programa ha sido implementado según las guías descritas en el artículo. Sin embargo, existen varios puntos dónde no el artículo puede no ser suficiente. Las diferentes técnicas son explicadas brevemente y se pueden ampliar de forma individual. Sin embargo, no viene ningún tipo de relación o descripción de cómo llegar a las fórmulas para cada técnica. También ha de comentarse que si se adhiere solo a la explicación del artículo, habría diferentes formas de realizar la implementación las cuáles llegarían a diferentes resultados. Sin embargo, los autores publicaron una implementación del programa en *MATLAB*. Esto sirve como guía para resolver cualquier duda de la implementación. Aunque para la implementación base se ha usado como guía, se ha de notar que existe cierta discrepancia entre las técnicas. Algunas técnicas utilizan el mismo número aleatorio $U(0, 1)$ en todas las apariciones de la fórmula. Produciendo un cambio más uniforme e intensificado. Mientras que otras técnicas utilizan diferentes números aleatorios para cada aparición. Esto provoca una mayor exploración y diversificación de la nueva solución.

5. Manual de usuario

Para poder ejecutar el programa, se necesita un intérprete de **Python3**, como se ha mencionado. Además, para instalar los módulos se necesita el gestor de paquetes **pip** (o **pip3**).

Para ejecutar el programa basta con ejecutar el siguiente comando:

```
$ python3 main.py
```

El programa se puede lanzar con o sin argumentos. Al lanzarse sin argumentos se realiza una ejecución por defecto. Para especificar una ejecución se lanza con los siguientes 5 argumentos:

```
$ python3 main.py [dataset] [Restr.%] [Seed]
                  [LambdaMod] [Alpha] [Beta] [nAtaques]
```

- **dataset** $\in$ { **ecoli**, **iris**, **rand**}
- **Restr. %**: nivel de restricción $\in$ {**10**, **20**}
- **Seed**: semilla $\in \mathbb{Z}$
- **LambdaMod**: modificador de $\lambda \in \mathbb{Q}$ (default = 1)
- **Alpha**: parámetro relacionado con la búsqueda local $\in [0, 1]$
- **Beta**: parámetro relacionado con la interceptación de ataques $\in [0, 1]$
- **nAtaques**: parámetro relacionado con la interceptación de ataques $\in \mathbb{N}$

Un ejemplo en una ejecución normal sería:

```
$ python3 main.py rand 10 123 1 0.3 0.5 50
```

Por otra parte, se puede comentar brevemente **comparison.py**. Será utilizado para iterar ejecuciones de **main.py** y escribirlas en archivos. Se ha usado para lanzar el programa con diferentes listas de parámetros e intentar optimizar los resultados.

6. Experimentación y análisis

Como se ha descrito anteriormente la metaheurística no es complicada de implementar. Aprovechando las capacidades de Python, todas las técnicas se pueden aplicar de forma directa sobre los vectores de solución. El entrenamiento, como se ha descrito anteriormente, es una modificación de la búsqueda local implementada anteriormente.

Sin embargo, este algoritmo no sirve para el problema PAR. Se muestra a continuación los resultados obtenidos con 36 posibles combinaciones de parámetros de entrada. Entre estas se encuentran las mencionadas en el trabajo las cuales mantienen α y $nAtt$. Al aumentar α se aumenta la explotación al aumentar las

iteraciones realizadas en la búsqueda local. Mientras que $nAtt$ permite aumentar la exploración para intentar salir de los óptimos locales.

Parámetros			Rand			
α	β	nAtt	TasaC	Inf	Agr	T
0.3	0.05	50	2.52	519	6.5	470
0.3	0.05	100	2.60	510	6.5	524
0.3	0.05	150	2.51	526	6.6	538
0.3	0.25	50	2.69	418	5.9	434
0.3	0.25	100	2.66	410	5.8	468
0.3	0.25	150	2.57	415	5.8	507
0.3	0.5	50	2.71	389	5.7	419
0.3	0.5	100	2.49	404	5.6	486
0.3	0.5	150	2.27	445	5.7	508
0.3	0.75	50	2.58	432	5.9	407
0.3	0.75	100	2.55	429	5.9	474
0.3	0.75	150	2.63	414	5.8	474

Cuadro 1: Rand 10 % Restr. Alfa = 0.3

Parámetros			Rand			
α	β	nAtt	TasaC	Inf	Agr	T
0.5	0.05	50	2.65	505	6.6	378
0.5	0.05	100	2.49	524	6.5	441
0.5	0.05	150	2.66	511	6.6	483
0.5	0.25	50	2.43	452	5.9	347
0.5	0.25	100	2.70	415	5.9	409
0.5	0.25	150	2.66	409	5.8	428
0.5	0.5	50	2.52	440	5.9	334
0.5	0.5	100	2.50	421	5.7	395
0.5	0.5	150	1.30	569	5.7	425
0.5	0.75	50	2.64	418	5.9	326
0.5	0.75	100	2.70	426	6.0	412
0.5	0.75	150	1.97	486	5.7	434

Cuadro 2: Rand 10 % Restr. Alfa = 0.5

Parámetros			Rand			
α	β	nAtt	TasaC	Inf	Agr	T
0.75	0.05	50	2.39	539	6.6	349
0.75	0.05	100	2.60	519	6.6	408
0.75	0.05	150	2.72	504	6.6	440
0.75	0.25	50	2.65	424	5.9	294
0.75	0.25	100	2.47	419	5.7	379
0.75	0.25	150	2.63	393	5.7	431
0.75	0.5	50	2.57	424	5.8	282
0.75	0.5	100	2.35	437	5.7	355
0.75	0.5	150	1.59	506	5.5	395
0.75	0.75	50	2.62	436	6.0	276
0.75	0.75	100	2.26	462	5.8	354
0.75	0.75	150	2.64	423	5.9	394

Cuadro 3: Rand 10 % Restr. Alfa = 0.5

Se ha de notar que el algoritmo *greedy* llega a un valor de función objetivo = 0.76. Se aprecia que no se puede realizar ningún tipo de comparación ya que SEO ni si quiera se acerca a un resultado óptimo. El artículo establece los parámetros de entrada $\alpha = 0.2$ y $\alpha = 0.3$ en la implementación. En las diferentes tablas se intenta combatir la falta de intensificación aumentando este valor. Aunque el mejor valor se alcance con el mayor valor de α, por lo general, la mejoría es leve. También establece $nAtaques = 50$. Ya que los resultados se mantienen altos al no poder salir de óptimos locales, se aumenta este valor para poder salir de estos. Se nota cierta mejoría cuanto mayor haya sido la intensificación previa. Por último, se prueban 4 valores para el parámetro β, siendo los tres primeros recomendados y usados en el estudio del artículo. Este parámetro controla la distancia del nuevo defensor al interceptar un ataque. De esta forma, a mayor valor mayor distancia con respecto a la solución original. También se ha añadido un nuevo valor mayor para aumentar la distancia con respecto a la solución original e intentar evitar óptimos locales. En los los casos testeados, el valor $\beta = 0.5$ da por lo general mejores resultados.

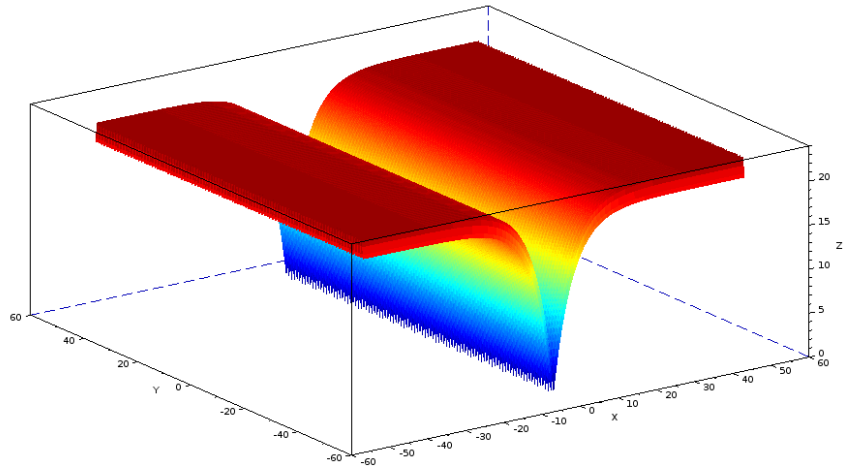
A pesar de haber modificado los parámetros, no se llega a ninguna solución remotamente óptima. Esto puede deberse a varios motivos. A continuación se realiza un estudio para intentar deducirlos.

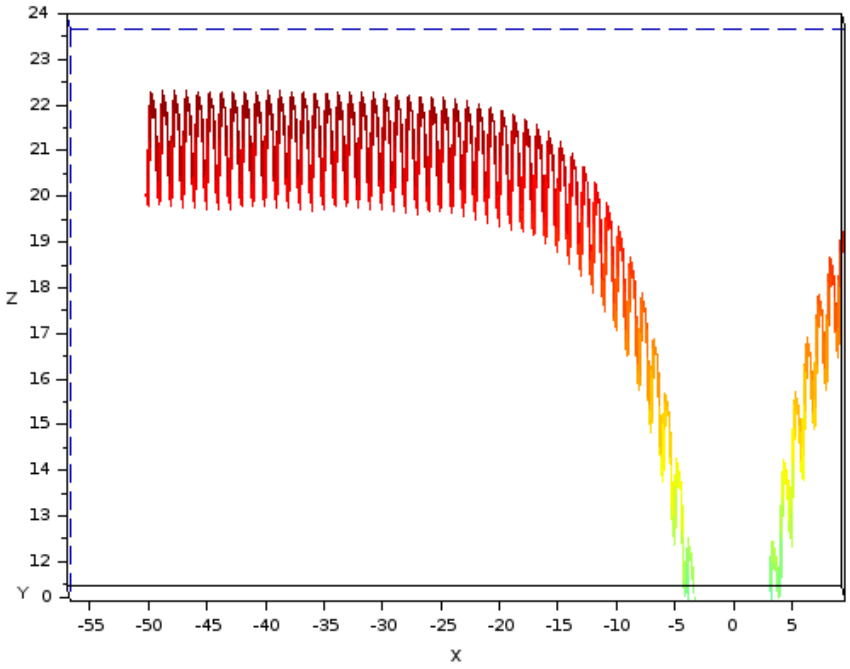
6.1. Estudio de espacio de búsqueda

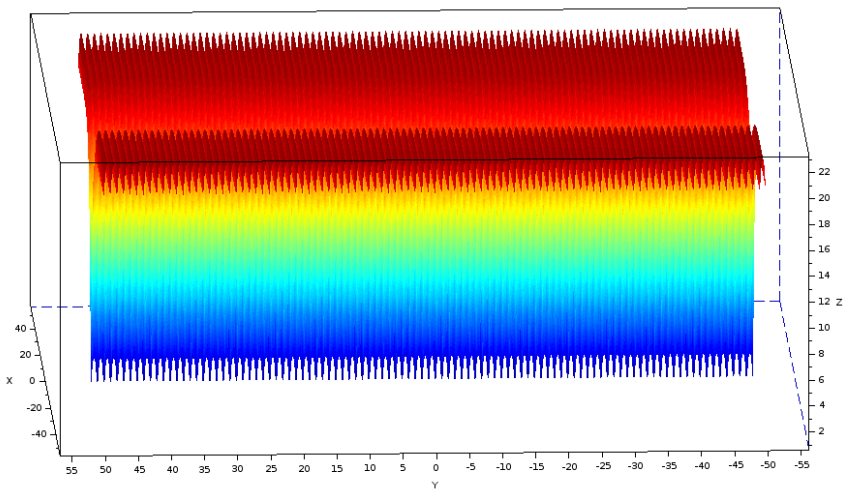
En el artículo no se menciona para que tipo de problemas o espacios de búsqueda este algoritmo es adecuado. Si que se menciona que los casos donde se realiza el

estudio tienen dimensionalidad 30 o 100 (luego se realizan otros experimentos con dimensionalidad menor a 10). En las tablas proporcionadas ya se puede apreciar que el incremento de costo entre ambas dimensionalidades no es lineal si no exponencial. Y una diferencia mucho mayor que cualquiera de las otras metaheurísticas con las que se compara. Ya que la menor dimensionalidad de cualquiera de los conjuntos de datos que se han usado en clase es 150, se puede deducir que tendrá un efecto negativo en los resultados obtenidos.

A pesar de esto, las soluciones proporcionadas no son cercanas a las óptimas a pesar de haber ofrecido más iteraciones de las necesarias. Por ello se pasa a analizar el espacio de búsqueda. En la experimentación explica usar 12 funciones matemáticas. Se comienza a analizar las función de Griewank y la función de Ackley. Ambas han sido implementadas con fórmulas ligeramente distintas a las originales citadas en el estudio de Ghorbani y Babaei (2014)[3]. Ya que son funciones bastante complejas se crea un gráfico para representar el espacio de búsqueda. Para esto se ha elegido la función de Ackley. Esta función cambia dependiendo de la dimensionalidad elegida pero, gracias a la implementación, se conoce que se ha elegido dimensionalidad 1. Se obtiene el siguiente gráfico como resultado:







Se muestra la gráfica desde diferentes ángulos para apreciar los óptimos locales existentes. Esto muestra que los espacios de búsqueda utilizados en el estudio no solo tienen menor dimensionalidad, sino que además los óptimos locales son bastante leves y cercanos unos de otros.

7. Optimizaciones

En esta sección se pasa a discutir algunas de las optimizaciones implementadas. Cabe decir que estas no van a permitir a la heurística competir con las otras estudiadas para este problema. Esto se debe a los problemas anteriormente descritos con la dimensionalidad y el espacio de búsqueda. Para conseguir que resultados aceptables en el problema se deberían hacer demasiados cambios lo que llevaría a una metaheurística diferente. Sin embargo, se puede intentar remediar algunos problemas para conseguir resultados ligeramente mejores.

La primera solución es intensificar la búsqueda en la generación de nuevos defensores. En un principio, estos se generan de forma totalmente aleatoria. Favorece mucho la exploración, pero también impide que buenas soluciones sean exploradas profundamente. Para esto, se escogerá un porcentaje de rasgos del atacante de forma aleatoria los cuales serán insertados en el nuevo defensor. Esto crea una continuidad en las soluciones permitiendo una mayor convergencia y reduciendo ligeramente la exploración. Se ha añadido un nuevo parámetro *optimizacion* al programa para activar este modo.

Se puede añadir todavía más convergencia cambiando el tipo de búsqueda local realizada. Como se ha descrito anteriormente, se ha usado la técnica 'primero el mejor'. Aunque esta técnica puede ayudar a encontrar mejores óptimos, se usan muchas evaluaciones y se realiza gran cantidad de exploración para una pequeña modificación. Por ello, se puede intercambiar esta búsqueda por la implementada anteriormente. La búsqueda local implementada se trata de un proceso iterativo donde, aunque se usen el mismo número de evaluaciones, se sustituye la solución actual por la primera que la mejore. Esto provoca una gran intensificación comparado con la modificación anterior. Aunque esta modificación se podría utilizar sin la optimización anterior, no produciría mejores resultados. Significaría que se realiza una búsqueda multiarranque con un proceso de exploración (interceptación de ataque). Igual que en el anterior caso, se añade un nuevo parámetro *optimizacion2* al programa para activar este modo.

Estas modificaciones cambian de forma drástica el comportamiento del programa. Al activar estas, se deberán modificar de forma concorde los parámetros. Ya que se está aumentando la intensidad y reduciendo la exploración, se puede afirmar que se necesita un α menor y $\beta, nAtt$ mayores. Los resultados mejoran entre medio punto y un punto con respecto a los anteriores. Sin embargo, estas puntuaciones siguen siendo demasiado bajas para cualquier tipo de comparación con otras metaheurísticas.

8. Reducción de Dimensiones

Durante toda la experimentación se describe la incapacidad de comparación entre la nueva heurística y otras vistas en clase. La heurística expuesta no se adapta a las necesidades del problema lo que provoca soluciones no óptimas. Sin embargo, se puede realizar una pequeña comparación con el mismo *dataset* usado anteriormente pero, esta vez, reduciendo la dimensionalidad de este. En un caso real se podría haber usado diferentes técnicas como el análisis de componentes principales para este fin. Sin embargo, se decide tomar un camino más simple y remover las dimensiones extras hasta reducirlas a 50. Es decir, reducimos el *dataset* a un tercio de su tamaño original (manteniendo las restricciones entre los diferentes puntos). Este nuevo *dataset* será usado con las diferentes técnicas vistas en clase:

	Small Rand			
	TasaC	Inf	Agr	T
COPKM	0.23	7	X	0.02
BL	0.23	8	0.51	0.28
BMB	0.23	3	0.33	2.53
ES Cauchy	0.23	3	0.33	8.12
ILS	0.23	4	0.37	1.29
ILS Cauchy	0.23	3	0.33	5.31
AM	0.43	29	1.44	100
AGG	1.56	32	1.57	103
AGE	0.23	7	0.48	115
SEO	0.23	3	0.34	207
SEO1	0.22	4	0.36	197
SEO2	0.22	3	0.32	212

Cuadro 4: Comparación final

Como se había sospechado, la heurística alcanza resultados notablemente mejores al reducir la dimensionalidad. Sorprendentemente alcanza el mejor resultado entre todos las heurísticas, aunque la mayoría llegan a soluciones muy parecidas. Esto confirmaría la sospecha sobre la causa de las pésimas soluciones encontradas anteriormente. Añadiendo los experimentos anteriores, se podría concluir que está técnica es apropiada para problemas con poca dimensionalidad y con óptimos locales no demasiado profundos.

Referencias

[1] TFM: Clustering de documentos con restricciones de tamaño
https://pdfs.semanticscholar.org/812b/edf1c055c37d03b5a2acd655fead801bd215.pdf

[2] Constrained K-means Clustering with Background Knowledge
https://www.cs.cmu.edu/~./dgovinda/pdf/icml-2001.pdf

[3] Exchange market algorithm
https://www.sciencedirect.com/science/article/pii/S156849461400074X