**Anotaciones guía 4**

**Computación evolutiva e inteligencia de enjambre**

**EJERCICIO 1**

<https://drive.google.com/drive/folders/1U0tUQK7EB05OsQpdo2_qdjOlkuqAzrUH>

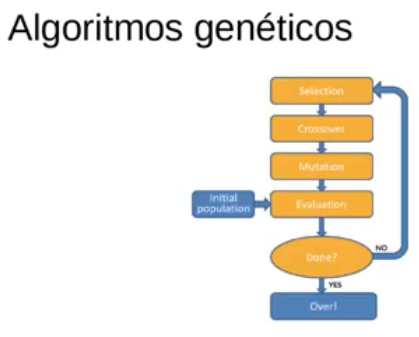
Se pide implementar un algoritmo genético y luego para probar este algoritmo vamos a abordar un método de optimización muy sencillo que consiste en buscar el mínimo global de las funciones que tenemos. En nuestro algoritmo, la función de aptitud o fitness va a estar dada en cada caso por la ecuación de cada función, y las soluciones, es decir, los individuos que componen la solución en nuestro algoritmo genético, van a representar los valores de “x”, es decir que, al decodificar el cromosoma de cada individuo va a representar un valor de “x” en el rango determinado para cada función.

A continuación vamos a hacer un **repaso de algoritmos genéticos**. En los vídeos de teoría esta explicado más en detalle.

Los algoritmos genéticos están inspirados en los principios de selección y evolución natural. Siguiendo la imagen de abajo, tenemos una población de individuos, donde cada individuo representa una solución al problema. Comenzamos con una población inicial (“initial population”, rectángulo azul a la izquierda de la imagen), iniciada al azar, luego esa población se evalúa (Evaluation) mediante la función de aptitud o fitness que depende del problema, y luego vamos iterando mientras el mejor individuo de la población no alcance la aptitud deseada, o podría ser una cantidad de iteraciones máxima, e iteramos aplicando los operadores: primero se aplica el operador de selección (Selection), que selecciona padres entre la población que van a tener hijos a partir de los operadores de cruza (Crossover) y mutación (Mutation).

Recordemos que este operador de selección permite elegir cualquier individuo de la población pero dándoles mayor probabilidad a los individuos más aptos, luego se aplica el operador de cruza que lo que hace es combinar información útil de distintos individuos (padres), para generar nuevos individuos (hijos) más aptos. Y también tenemos el operador de mutación, el cual modifica aleatoriamente el valor de un gen del cromosoma, lo cual le permite al algoritmo explorar distintas zonas del espacio de búsqueda y, a su vez, escapar de mínimos locales.

Una vez que tenemos la nueva población, volvemos a evaluar cada individuo (Evaluation) con la función de aptitud o fitness y así vamos iterando a lo largo de las generaciones hasta encontrar un nivel de aptitud deseado o alcanzar un máximo de iteraciones. De esta forma se espera que a lo largo de las iteraciones, las poblaciones tengan individuos cada vez más aptos.



La principal **ventaja** de este algoritmo es que permite optimizar cualquier tipo de función, no necesita que esa función tenga características como la existencia de derivada o continuidad, incluso ni siquiera necesita una expresión matemática de la función, podría ser por ejemplo un algoritmo, lo único que necesita es que la función nos dé una medida de que tan buena es cada posible solución, y eso lo hace muy flexible ya que nos permite resolver infinidad de problemas de optimización.

Diferencia entre cromosoma y gen:

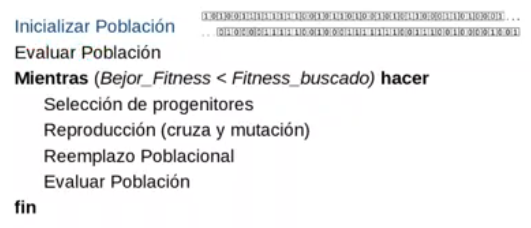
Cromosoma: Toda la cadena de bits.

Gen: Representación de una de las partes de la solución (un bit).

**Pseudocódigo algoritmos genéticos**

Como se ve en la imagen de abajo, comenzamos inicializando la población de forma aleatoria. Cada individuo está representado por un cromosoma que es una cadena binaria, es decir que, para inicializarlo vamos a completar esa cadena binaria con unos y ceros de forma aleatoria.

Como segundo paso vamos a evaluar la población aplicando la función de fitness dada por el problema que queremos optimizar, y previamente tenemos que decodificar el cromosoma, es decir, llevar los valores del genotipo al genotipo en el dominio donde se puede evaluar con la función de fitness y luego, mientras no se cumpla la condición de criterio de corte, vamos a aplicar un método de selección, luego se aplican los operadores de cruza y mutación para la reproducción y generar la descendencia (hijos) y una vez que tenemos la descendencia de eso se pueden aplicar distintas estrategias de reemplazado poblacional, y luego se vuelve a evaluar la población e iterando hasta cumplir el criterio de corte.

****

**Representación del cromosoma**:

En los algoritmos genéticos que vamos a implementar, la representación es binaria, así que *los cromosomas son cadenas binarias.*

Lo que represente cada gen (bit) del cromosoma depende del problema en cuestión, por ejemplo, podría ser que cada gen represente un switch y que indique si esta encendido o apagado. En problemas como los que vamos a resolver nosotros, en el cromosoma representamos valores numéricos, por lo tanto, tenemos que decodificar esa cadena binaria en uno o más valores numéricos (como en la imagen de abajo). Si estamos representando un solo valor, decodificamos todo el cromosoma en un solo valor (real o entero), y si estamos codificando varios valores tendremos una cantidad de bits para cada valor, que no necesariamente debe ser la misma cantidad de bits sino que puede ir variando, y al decodificar iremos teniendo en cuenta los bits que representa cada variable en nuestro problema.



**Operadores de selección: Ruleta**

A cada individuo de la población se le asigna un área en la ruleta, proporcional a su valor de aptitud o fitness, por lo tanto, los individuos más aptos tendrán un área más grande que los menos aptos. Al simular el giro de la ruleta habrá más probabilidad de que caiga en un área grande, es decir, es más probable que se elijan los individuos más aptos.

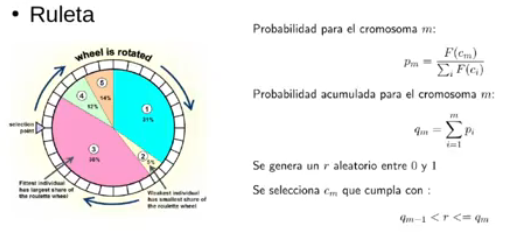
**Implementación de ruleta**:

-Se calcula una probabilidad para cada cromosoma “m” (pm), que se calcula como el valor de fitness para ese cromosoma F(cm) sobre la suma total de los fitness para todos los individuos.

-Luego se calcula una probabilidad acumulada para el cromosoma “m” (qm) que se calcula como la sumatoria de todas las probabilidades pi de los cromosomas desde 1 hasta m, es decir, desde el primer individuo hasta el individuo “m” para el cual estoy calculando su probabilidad acumulada.

-Se genera un valor aleatorio “r” entre 0 y 1.

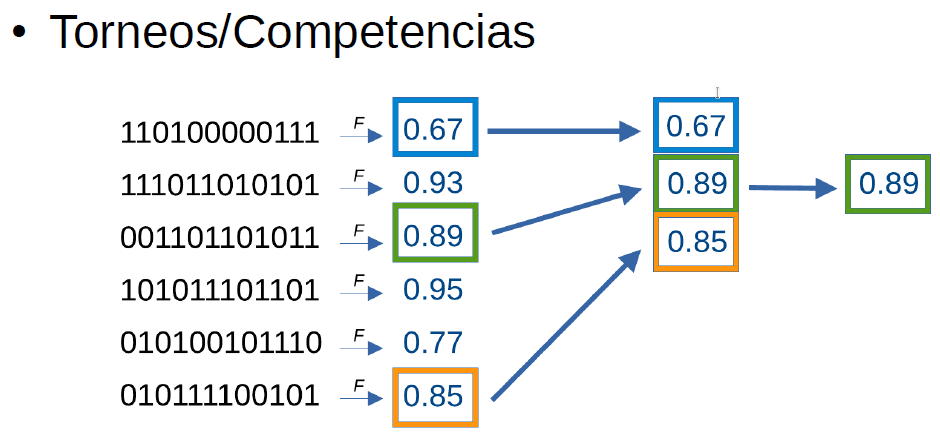
-Se elige el cromosoma cm para el cual se cumpla que “r” esta entre qm-1 y qm, y de esa forma elegimos con el método de la ruleta.



Si bien la ruleta es un método muy usado, tiene **dos problemas**: mar de mediocres y mar de virtuosos, como vimos en teoría. El mar de mediocres significa que si tenemos muchos individuos de baja aptitud y pocos individuos buenos, tenemos mayor chance de elegir un individuo malo que uno bueno, y el mar de virtuosos es que si tenemos todos los individuos con poca diferencia entre ellos, por ejemplo 49.9, 50, 50.1, etc. va a terminar siendo una elección al azar. Por eso existen otros métodos que solucionan estos problemas.

**Operadores de selección: torneos/competencias**.

Consiste en elegir “n” individuos al azar y luego de esos “n” nos quedamos con el de mejor fitness. Ese “n” es un parámetro del método y mientras más grande sea más chances tendremos de elegir los individuos más altos.

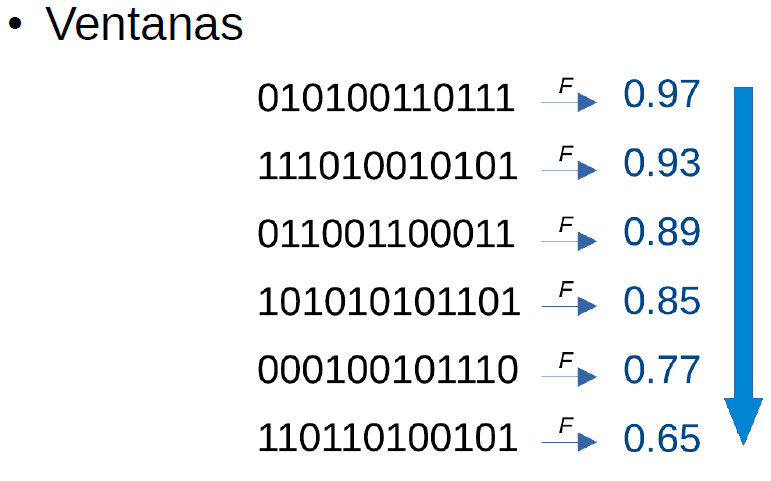
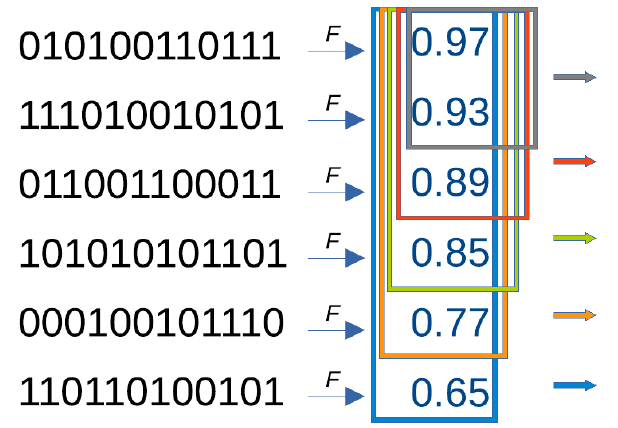


**Operadores de selección: ventanas**.

Consiste en ordenar la población de mayor a menor fitness, luego tomamos una ventana que abarque a todos los individuos y tomamos uno al azar, con la misma probabilidad para todos. Ahí puede salir un individuo con alto fitness o bajo. Luego achicamos esa ventana, por ejemplo, desde el individuo de mejor fitness hasta el anteúltimo (o un 80% o cosas así), y elegimos otro individuo, pero entonces ahora el de peor fitness (el que estaba último) ya no tiene posibilidad de ser elegido porque no entró en la ventana, y así seguimos achicando la ventana y eligiendo nuevos individuos hasta completar la población.

Lo que logramos es que los individuos más aptos estén en mayor cantidad de ventanas y tengan mayor probabilidad de tener descendencia (hijos).

Como vamos achicando la ventana para quedarnos solo con los de mejor fitness (porque están ordenados), si hacemos un listado de los fitness que vamos obteniendo siempre al final tendremos valores más altos (no necesariamente siempre creciente porque se elige al azar) porque vamos a estar eligiendo entre los mejores.

**Operadores de variación: cruza**

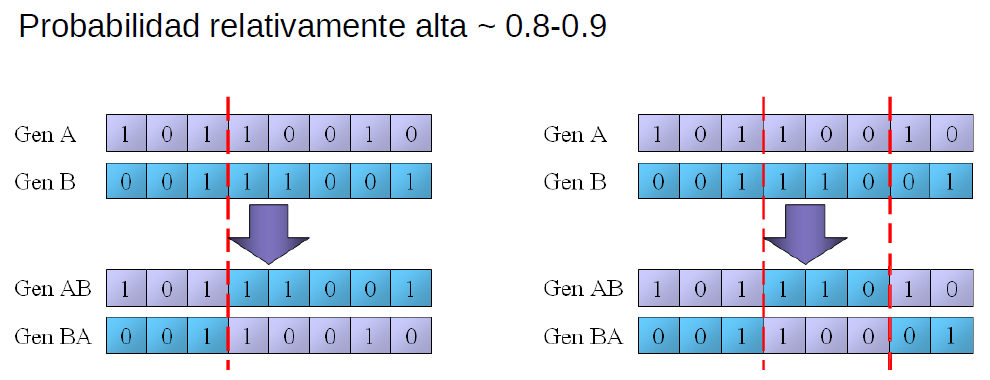
Permite combinar información genética de distintos padres para generar los hijos.

Una vez que tenemos los padres elegidos mediante el método de selección, lo que hacemos es tomarlos de a pares, como muestra la imagen de abajo (Gen A y Gen B), y los vamos a cruzar con cierta probabilidad. En general se usa una probabilidad relativamente alta como 0.8 o 0.9, y entonces lo que hacemos es generar un número aleatorio entre 0 y 1, así si ese número aleatorio es menor que la probabilidad elegida cruzamos los individuos padres, y si es mayor, los individuos hijos son exactamente iguales a los padres.

En el caso de cruzarlos, primero elegimos un punto de cruza al azar (marcado con la línea roja punteada) y a partir de ese punto de cruza intercambiamos la información de los individuos padres, por ejemplo, vemos que el hijo Gen AB tiene la primera parte del padre A (hasta el punto de cruza) y la segunda parte del padre B (desde el punto de cruza hasta el final), mientras que el otro hijo Gen BA, en la primera parte tiene los genes del padre B y en la segunda parte los genes del padre A.

Es decir que, a partir de la cruza de dos padres generamos dos hijos combinando la información genética de esos dos padres.

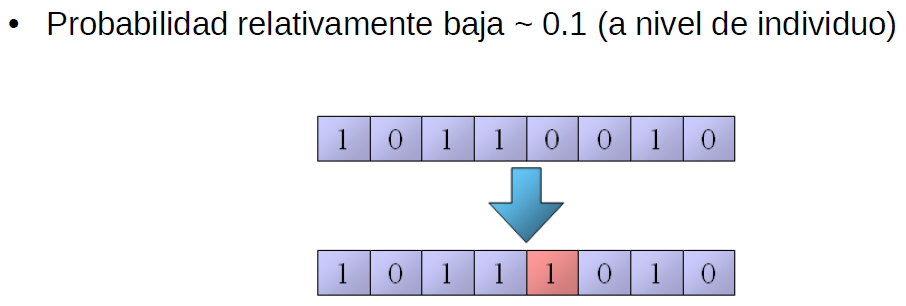
Ese ejemplo sería con un punto de cruza, pero también se puede hacer con varios puntos de cruza, como se muestra a la derecha en la imagen, donde vemos un ejemplo con dos puntos de cruza y el procedimiento es exactamente igual al anterior.



**Operadores de variación: mutación**

Una vez que generamos la descendencia, es decir, los individuos hijos mediante el operador de cruza, vamos a aplicar el operador de mutación a cada hijo.

Vamos a usar una probabilidad baja, alrededor de 0.1, a nivel de individuo (puede ser 0.2 en el caso de que se use elitismo), y consiste en generar un valor al azar entre 0 y 1, si ese valor es menor a la probabilidad de mutación significa que vamos a mutar el cromosoma, para lo cual vamos a elegir de forma aleatoria el gen a mutar, y si ese gen tiene un valor 0 lo cambiamos a 1 y si tiene un valor 1 lo cambiamos a 0.



Esa es la forma más sencilla de implementar la mutación en el caso de cromosomas binarios. Otra forma de implementarla puede ser considerando una probabilidad de mutación individual para cada gen, es decir, consideramos una probabilidad más baja a nivel de cada gen. Entonces en lugar de elegir al azar una vez el gen que se va a mutar, podemos tirar la moneda para cada gen de forma independiente, entonces podría ser que haya más de un gen mutado. De todas formas, para ese caso hay que tener en cuenta que la probabilidad total del cromosoma (toda la cadena de genes o bits) de ser mutado no debería ser superior a 0.1 aproximadamente, es decir, debería ser baja.

Se podría decir que la cruza utiliza información ya existente, mientras que la mutación agrega nueva información.

**Criterios de finalización o corte de los algoritmos genéticos**

-Fitness deseado.

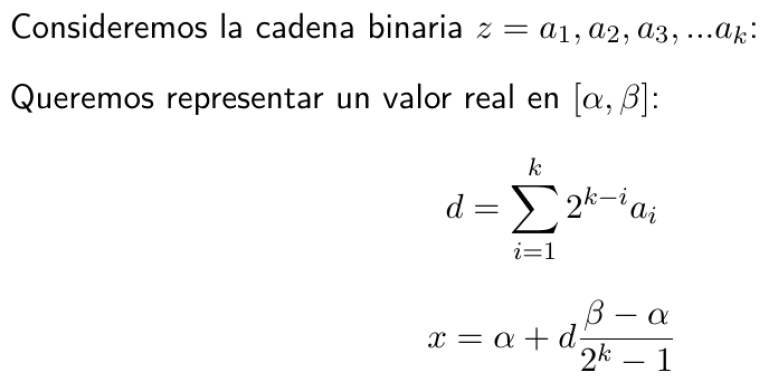
-Cantidad máxima de generaciones.

-Sin mejoras en el fitness por “n” generaciones.

La idea sería que si se alcanza un valor de fitness deseado se detiene la iteración. Pero si no se obtiene el valor de fitness deseado, podemos usar una cantidad máxima de generaciones. El problema es que tampoco sabemos cuánto va a tardar el algoritmo en converger, entonces si seteamos una cantidad máxima de generaciones que sea un poco chica podemos terminar cortando el algoritmo de forma prematura, entonces conviene setear una cantidad máxima alta de generaciones y luego podemos combinarlo con otro criterio donde vamos siguiendo el mejor fitness entre las generaciones y si ese mejor fitness no tiene mejoras a lo largo de “n” generaciones, ahí podemos considerar que el algoritmo convergió y cortamos.

**Decodificación**

Lo que sigue es decodificar nuestros cromosomas en valores enteros o reales. Es decir, tenemos cromosomas que son cadenas binarias, como el cromosoma “z” de la imagen que es una cadena de “k” bits, y los queremos mapear en un intervalo de valores reales entre α y β, para obtener un valor real, entonces primero los mapeamos a un decimal “d”, es decir, convertimos de binario a decimal con la sumatoria indicada, y una vez que tenemos dicho valor decimal usamos la última fórmula para obtener “x” entre α y β, que sería el valor “x” que necesitamos para evaluar el individuo mediante nuestra función de fitness.



*Ver bien cómo explicar esas fórmulas por si las preguntan en el* ***parcial****.*

*Sofi tenía anotado lo siguiente sobre decodificación*:

: 0 ó 1

d: entero

Cuantos más bits se tienen, mayor representación o “granularidad” tenemos

(bits de 0 a 512, codificados con 00, 01, 10 y 11)

0 01 10 512

00 11

🡪 escala

🡪 traslada

🡪 sería como un delta

**Otras consideraciones importantes**

-Mapeo de fitness en problemas de minimización:

Otra cuestión a tener en cuenta es que en los algoritmos genéticos buscamos maximizar el fitness, es decir, queremos obtener el mayor valor posible de fitness. Mientras que, en algunos problemas de optimización como en el caso del ejercicio de la guía donde tenemos que buscar el mínimo global de ciertas funciones, se tratan de problemas de minimización, entonces en esos casos podemos hacer un mapeo entre la función objetivo y la función de fitness, por ejemplo, podríamos considerar que la función de fitness “F” sea F = 1/f, donde “f” sería el valor de la función y de esa forma lo convertimos a un problema de maximización.

Otra opción sería hacer el fitness F = 1 – (f/max(f)), es decir, 1 menos la función sobre el valor máximo de la función en caso de que sepamos cual es, y de esa forma podemos mapear el valor de la función objetivo en un valor de fitness y pasamos de un problema de minimización a uno de maximización.

Por último, hay dos *estrategias para tener en cuenta al momento de reemplazar la población por su descendencia*, que son el elitismo y la brecha generacional.

-**Elitismo**: consiste en copiar de la población actual el mejor individuo que tenemos, y pasarlo a la población siguiente, sin ninguna modificación, es decir, sin aplicarle operadores de variación.

Eso tiene la ventaja de que no vamos a perder la mejor solución encontrada hasta el momento, que de otra forma con los operadores de variación se podría perder. Además, nos permite aumentar la probabilidad de mutación, permitiendo una mejor exploración del espacio de búsqueda sin riesgo de complicar la convergencia del algoritmo.

(Sofi tenía anotado que si no se hace elitismo, el fitness puede oscilar).

-**Brecha generacional**: consiste en seleccionar una cantidad determinada de individuos de la población, empleando el algoritmo de selección, es decir, dándole mayor probabilidad a los individuos más aptos, y también copiar esos individuos sin modificación para pasarlos a la población de la generación siguiente. De esta forma se favorece la estabilidad de convergencia del algoritmo.

**Volviendo al ejercicio 1**

Una vez implementado el algoritmo genético, lo debemos probar buscando el mínimo global de esas funciones “f(x)”, donde la primera es una función de una variable y que tiene varios mínimos locales, y la segunda función f(x, y) es de dos variables entre -100 y 100, que es la graficada en el pdf donde también se ve que tiene muchos mínimos locales y el algoritmo genético debería ser capaz de encontrar el mínimo global de la función.

Por último, para comparar el desempeño del algoritmo genético, se pide implementar el método del gradiente descendente para buscar el mínimo global de las funciones, para lo cual vamos a partir de un punto elegido al azar y hacer la búsqueda en base al gradiente, repitiendo eso “n” veces para que sea comparable al algoritmo genético.

Esa comparación es simplemente hacer una búsqueda del mínimo por el método del gradiente descendente. Tendremos una función f(x) (o f(x, y) en el inciso ii) que queremos minimizar. Partiendo de un punto x\_0 cualquiera (elegido al azar), sabemos que la dirección del gradiente en x\_0 es la dirección donde la función crece más rápido, por lo tanto la dirección opuesta (-gradiente) será la que decrece más rápido. De esa forma, haces una iteración donde x\_t+1 = x\_t - \alfa gradiente con un \alfa pequeño. Eso debe converger al mínimo local más próximo al punto inicial donde hayas empezado.

**EJERCICIO 2**

Sobre este ejercicio no hay video, pero dejo la explicación que el profe nos dio por correo.

Este problema es fácil si ya tenés implementado el algoritmo genético. Tenés un conjunto de datos donde tenés pocos patrones, pero en una dimensión muy grande (es decir, la cantidad de entradas para hacer un clasificador es enorme). Una forma de encarar este problema es reducir el número de entradas, "eligiendo" un subconjunto para el cual puedas tener un buen desempeño del clasificador. Para eso es que vas a usar un algoritmo genético.

Para usar lo que ya implementaste debemos cambiar 2 cosas simplemente: la codificación de los individuos y la función de fitness:

1) individuos: cada individuo es una potencial solución, o sea una selección de qué entradas debemos usar en el clasificador. Vas a usar una cadena binaria, de igual longitud que la cantidad de características (entradas) del dataset, donde cada bit indica si esa característica la vas a usar o no en el entrenamiento. Por ejemplo, suponiendo que hubiera 5 entradas solamente, una cadena 1 0 0 1 0 te diría que tenés que usar sólo las características en la primera y cuarta posición, y el resto descartarlas.

2) lo otro que tenés que modificar es la función de fitness. Y ¿cómo se hace eso? primero tenés que tener claro el objetivo, que en realidad son dos: por un lado querés que el clasificador funcione muy bien (maximizar), y por otro queres que use la menor cantidad de características posibles (minimizar la cantidad de características). Entonces vas a usar una función de fitness con dos términos, uno es directamente el accuracy de un clasificador, y el otro tiene que restar fitness si el número de características usadas es muy alto. Entonces fitness = \alfa\* accuracy - \beta (número de características elegidas del individuo/número de características totales), donde \alfa y \beta son parámetros que controlan qué tanta importancia se le da a estos dos objetivos (si alfa es más grande, el algoritmo tratará de maximizar el accuracy sin importarle tanto usar más cantidad de características, si beta es más alto tratará de reducir la cantidad de características sacrificando un poco de accuracy, y así)

*(Esa fórmula de fitness está más abajo en una imagen del PDF).*

Lo único que falta es definir de dónde sacar el accuracy. Y eso se hace ENTRENANDO UN CLASIFICADOR (el que queramos de los que usamos en la guía 3 sobre de scikit learn).

Resumiendo:

- Codificas en binario los individuos con una posición para cada característica que querés ver si retenes o no.

- Para cada individuo, usas su codificación para elegir qué características vas a usar.

- Recortas el dataset para usar sólo esas características.

- Entrenas un clasificador (el que quieras) con los datos de entrenamiento.

- Obtenés el accuracy del clasificador.

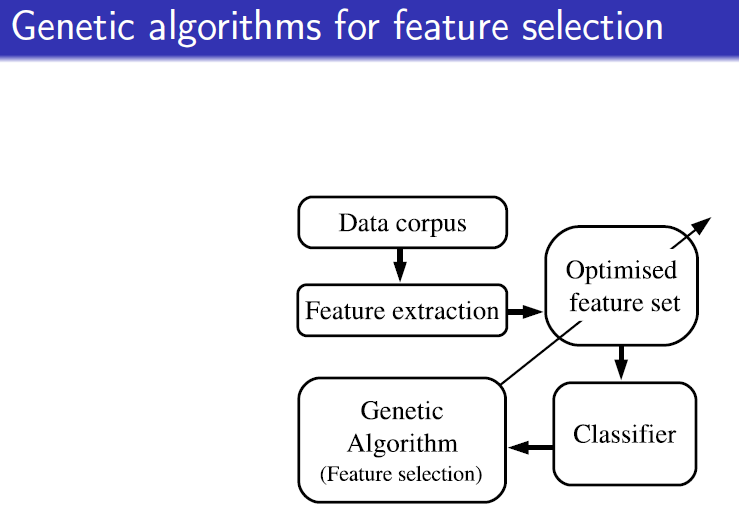
- Con eso calculas el fitness de ese individuo

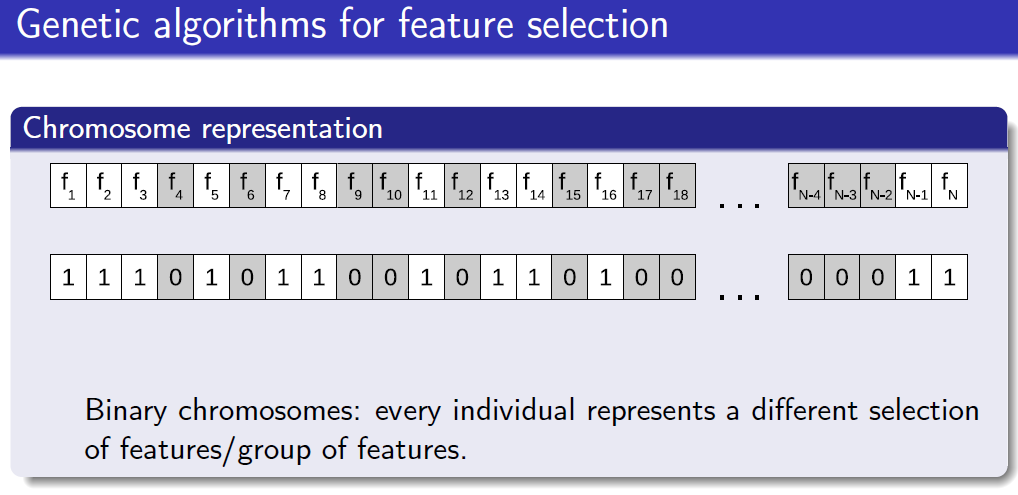
- Repetís esto para todos los individuos

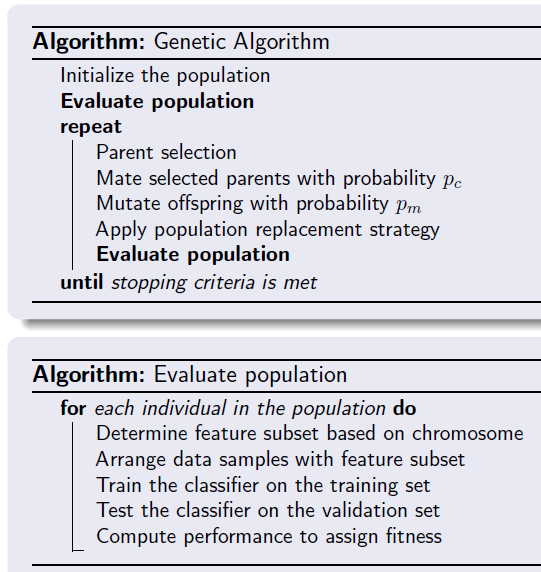
- Y ahí tenés todo lo que necesitas para hacer el algoritmo genético, con el fitness de cada individuo haces selección, cruza, mutación, generas una nueva población, y repetís todo lo anterior, hasta que converja.

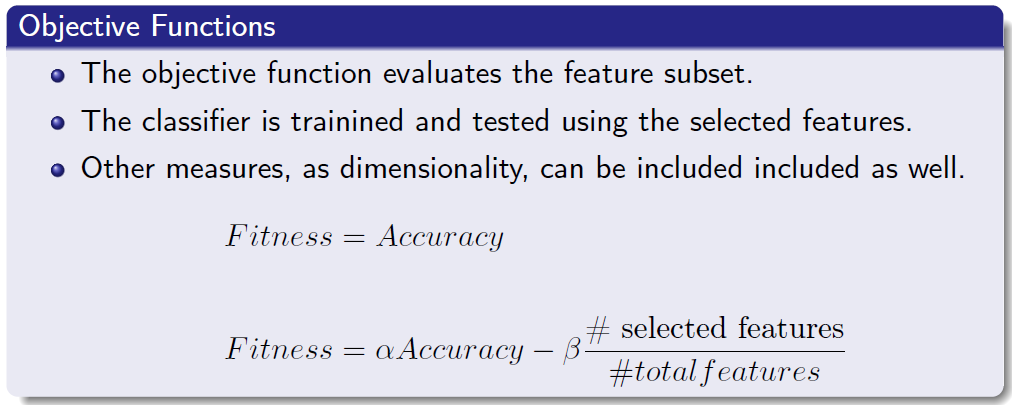
En el **PDF** que subieron los profes (“repasoAG\_introFS” lo tengo en la carpeta) también se explican cosas de este ejercicio a partir de 12 en adelante, y se muestra un algoritmo y ecuaciones que se usan.

Abajo dejo imágenes de pág. 19 en adelante.





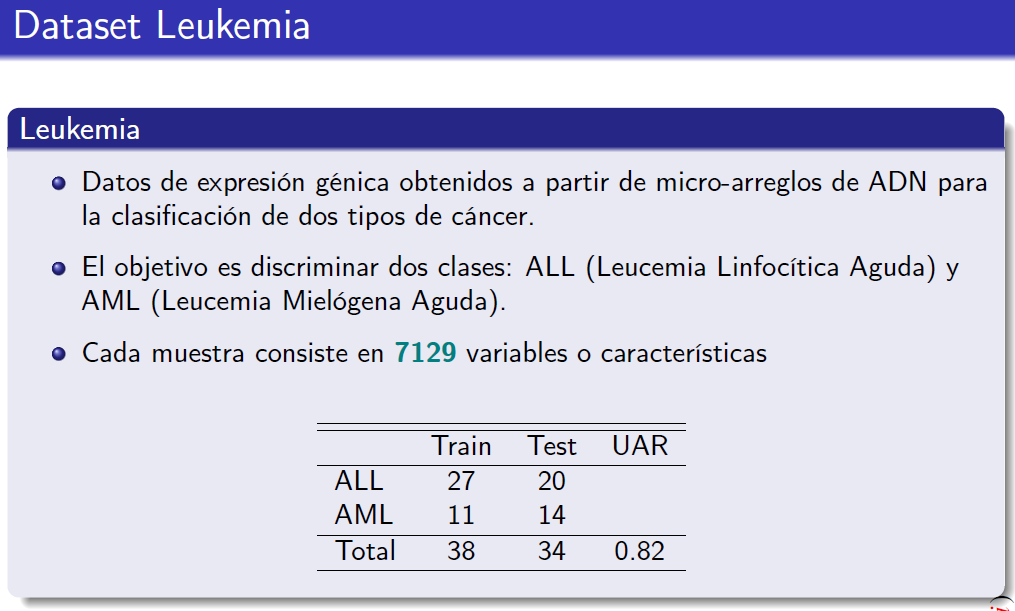




-Mientras más “grande” sea el valor de ‘Accuracy’, mejor.

-Mientras más “chico” sea el valor de ‘total features’, mejor.

Porque así nos dará un fitness más alto.



Sobre lo de “UAR” en la tabla Sofi tenía anotado:

UAR: Referencia. Calcular el accuracy para cada clase, y después calcular el promedio.

**EJERCICIO 2 DEL 2022**

<https://drive.google.com/drive/folders/1U0tUQK7EB05OsQpdo2_qdjOlkuqAzrUH>

**Este ejercicio era del 2022 pero no lo vemos en el 2023**

Se pide implementar el algoritmo de optimización con enjambre de partículas y utilizarlo también para encontrar el mínimo global de las funciones del ejercicio 1. Luego se pide comparar los resultados de éste método con los del algoritmo genético del ejercicio 1, en términos de las soluciones encontradas y la velocidad de convergencia.

**Repaso del método de optimización por enjambre de partículas**

(En la teoría están desarrollados más en profundidad).

-Son métodos de búsqueda inspirados en el comportamiento de bandadas de pájaros. Se basa en el comportamiento de los individuos de imitar el éxito de otros individuos además de considerar su propia experiencia. A partir de esto se logra que, mediante reglas individuales y simples se logren comportamientos sociales complejos.

Entonces en un enjambre cada partícula consiste de:

-Posición actual xi(t), donde “i” indica la partícula del enjambre, “t” es el instante de tiempo, y el vector “x” en cada dimensión tiene una variable del problema, es decir, en este caso no hay una codificación como en los algoritmos genéticos, sino que las variables están en el mismo dominio del problema.

-Velocidad actual vi(t), también es un vector de las mismas dimensiones.

-Mejor posición histórica yi(t), es decir, la posición en la que obtuvo el mejor fitness hasta ese momento.

-Mejor posición histórica global o de su vecindario ^yi(t), es información sobre su enjambre, va a conocer la mejor posición obtenida en todo el enjambre hasta ese momento. (El ^ es un sombrerito en la “y”).

Nota: Cuando la mejor posición global no usa el subíndice “i” sino que es solo ^y, representa la mejor posición histórica global para todo el enjambre, mientras que si es ^yi no va a ser la misma para todo el enjambre, sino que va a depender de la vecindad en la que se encuentre la partícula, es decir, la mejor posición histórica de la vecindad de esa partícula.

**Pseudocódigo del algoritmo de optimización para enjambres de partículas**

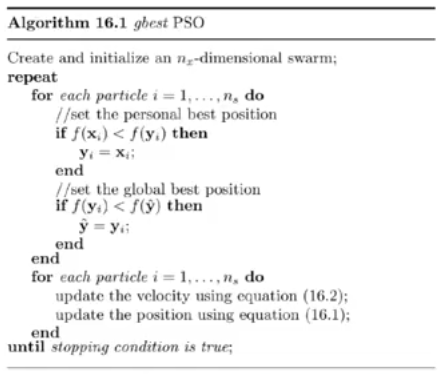
Debajo vemos la imagen del pseudocódigo, en el cual, el primer paso es inicializar los vectores de posición de cada partícula en el enjambre, esto lo hacemos de forma aleatoria y con distribución uniforme. Luego comienza el bucle, donde para cada partícula “i” desde 1 hasta ns que sería el tamaño del enjambre, en el bucle primero vamos a calcular la función objetivo evaluada en la posición actual de la partícula f(xi), y la vamos a comparar con la función objetivo evaluada en la mejor posición histórica de esa partícula f(yi). Si la función objetivo es mejor que la mejor posición obtenida para esa partícula “i” hasta el momento, vamos a actualizar esa mejor posición histórica de la partícula, es decir, yi = xi. Como estamos actualizando solo si la función objetivo en la posición actual es menor, significa que estamos minimizando la función objetivo, a diferencia del algoritmo genético donde maximizamos el fitness.

Para la primera iteración, la mejor posición histórica va a ser la misma que la posición actual, por lo tanto, ese if pasa de largo la primera vez.

Luego en el segundo if vamos a comparar la función objetivo evaluada en la mejor posición historia de la partícula f(yi) contra la función objetivo evaluada en la mejor posición global f(^y), es decir, la mejor posición histórica en el enjambre, y si fuera mejor, actualizamos la mejor posición del enjambre con esa mejor posición de la partícula: ^y = yi.

Luego, en el segundo for, para cada partícula vamos a actualizar la velocidad y después actualizar el vector de posiciones.

Esto lo vamos a iterar hasta que se cumpla la condición de corte, en este caso sería el pseudocódigo para la variante global del algoritmo, donde se considerar esa mejor posición global ^y de todo el enjambre. La diferencia con la variante local es que esa mejor posición en vez de considerarla de todo el enjambre vamos a tener una mejor posición histórica para cada vecindad, es decir, cada partícula va a tener una vecindad y solo va a conocer la mejor posición dentro de esa vecindad y no dentro del todo el enjambre.



Lo que nos faltaba definir en el pseudocódigo es cómo actualizar la posición de la partícula y cómo actualizar la velocidad, como se muestra en la imagen de abajo.

En el caso de la actualización del vector de posición, la posición de la partícula “i" en el instante t+1: xi(t+1) va a ser la posición actual de la partícula más el vector de velocidad.

Mientras que, para actualizar el vector de velocidad (global best), es el corazón de este algoritmo. En la imagen se puede notar un pequeño cambio de notación, donde estamos descomponiendo el vector de velocidad vi en sus distintas componentes, por eso aparece el índice “j” en vij(t+1), donde “j” representa cada una de las dimensiones del vector vi, que son las mismas dimensiones que las del vector “x”.

Entonces, para actualizar la velocidad de la partícula “i” en la dimensión “j” para el instante t+1, es decir, vij(t+1), vamos a calcularlo como se muestra en la imagen, donde sumamos la velocidad actual de la partícula en esa dimensión vij(t), y tenemos luego un término que corresponde a la experiencia personal de la partícula (todo el término con c1r1 y los corchetes), y un segundo término que corresponde a la experiencia del enjambre (la suma del c2r2 y el corchete). Se puede ver que en el primer término consideramos la mejor posición histórica yij de la partícula “i”, y en el segundo término se considera la mejor posición histórica del enjambre ^yj, por lo tanto, ese primer término lo que hace es tratar de llevar esa partícula hacia la dirección donde obtuvo su mejor desempeño, mientras que el otro término trata de llevar la partícula hacia la mejor posición global.

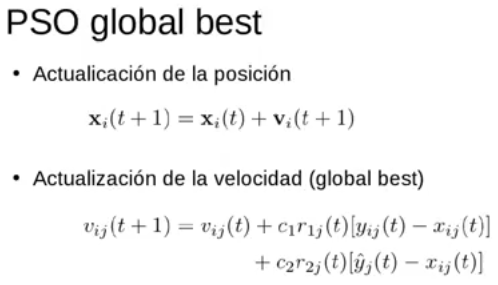
Tenemos un vector **r**1j y un vector **r**2j, que también tienen el subíndice “j”, por lo tanto, son vectores con la misma dimensión que el vector velocidad, y son vectores aleatorios que se generan con distribución uniforme y de esa manera se agrega una componente estocástica al algoritmo, similar a lo que sería el operador de mutación en los algoritmos genéticos. Es importante notar que los vectores r1 y r2 al tener diferentes valores aleatorios en cada una de sus componentes, eso permite que la influencia de la mejor posición de la partícula y la mejor posición global tengan diferente peso en cada una de las dimensiones, lo cual permite a la partícula hacer una exploración más completa del espacio de búsqueda, le permite moverse con mayor libertad. Además esos vectores dependen del tiempo “t”.

Y las dos constantes “c1” y “c2” son dos constantes del algoritmo que nos permiten controlar cuánta importancia le damos a la experiencia personal de la partícula y cuánta a la experiencia del enjambre, en el cálculo de la velocidad.

Si c1 es mayor que c2, estaríamos dando mayor peso a la componente que representa la experiencia personal de la partícula, por lo tanto, en ese caso el algoritmo hace una mayor exploración del espacio de búsqueda, mientras que si c2 es mayor que c1, estaríamos favoreciendo a la convergencia del algoritmo pero a su vez limitando un poco la exploración del espacio de búsqueda.

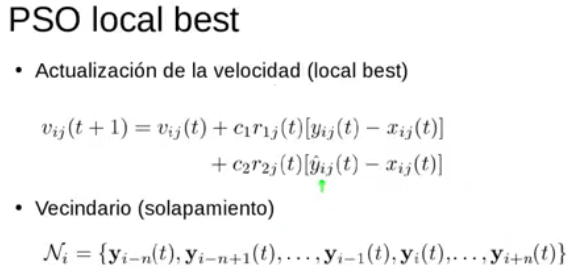
Entonces lo ideal es hacer que esas dos constantes varíen a lo largo de las iteraciones, de esa forma se puede hacer que c1 sea grande al principio para que en las primeras iteraciones se haga una buena exploración del espacio de búsqueda y luego se vaya reduciendo, mientras que para c2 hacemos que sea pequeña al comienzo y luego vaya creciendo a lo largo de las iteraciones, para que en las últimas iteraciones se le vaya dando más importancia a la componente social del enjambre, y de esa forma se favorezca a la convergencia del algoritmo.

En general, para c1 y c2 se suelen utilizar valores entre 0.5 y 2.5.

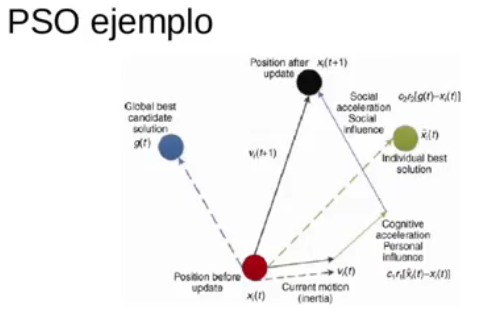


Ya vimos como actualizar la velocidad para el caso de la versión global del algoritmo. Para la versión local la actualización es similar, lo que cambia es que la componente la social (marcada con un flecha en la imagen de abajo), ya no vamos a tener en cuenta la mejor posición de todo el enjambre, sino la mejor posición histórica de la vecindad de esa partícula. Por eso ahora aparece el índice “i” en la mejor posición histórica social, es decir, ^yij porque ya no va a ser la misma para todo el enjambre, sino que va a depender de la vecindad en la que se encuentre la partícula.

Entonces podemos definir una vecindad para la partícula “Ni”, por ejemplo una vecindad lineal de tamaño “n”, entonces esa vecindad va a incluir las partículas desde i-n hasta i+n, y entre esas partículas van a compartir la información social, es decir, la mejor posición histórica de la vecindad.



En la imagen de abajo vemos un ejemplo que ilustra cómo se actualiza la posición en la partícula, teniendo en cuenta la información de la mejor posición en el enjambre y la experiencia individual de la partícula. Tenemos la posición actual roja, la mejor posición del enjambre es la azul, y la mejor posición individual de la partícula es la verde. Entonces según cuánto peso se les dé a cada componente, se actualiza la velocidad y luego se actualiza la posición de la partícula, que sería la negra en la imagen, en este caso más o menos entre medio de la mejor posición global y la mejor posición personal, por lo tanto, se estaría explorando una nueva zona del espacio de búsqueda que no había sido explorada.



**Consideraciones finales:**

-En el caso de la versión global del algoritmo, la mayor interconectividad le permite una convergencia más rápida, pero en el caso del algoritmo local tener vecindarios aislados le permite conservar mayor diversidad, y esto lo hace menos susceptible a caer en mínimos locales.

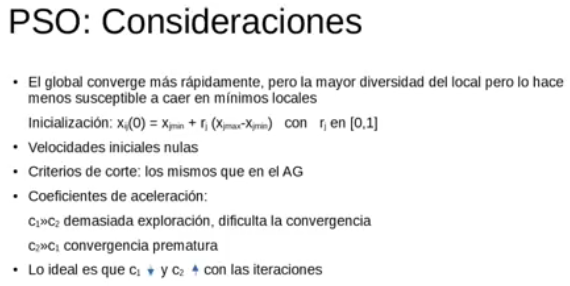
-Para la inicialización, si conocemos los valores máximos y mínimos para cada dimensión del problema, podemos aprovecharlo para inicializar las posiciones de esa manera que se muestra, generando un “r” aleatorio entre 0 y 1 con distribución uniforme y de esa manera nos aseguramos que inicializamos las posiciones dentro de los rangos adecuados.

-Las velocidades iniciales se setean como nulas.

-Como criterios de corte podemos tener los mismos que mencionamos para algoritmos genéticos.

-Para los coeficientes de aceleración, las constantes c1 y c2, si c1 > c2, le damos más importancia a la componente individual que a la social, entonces se realiza una mayor exploración pero se dificulta la convergencia, mientras que si c2 > c1 le damos mayor importancia a la componente social, entonces tenemos riesgo de una convergencia prematura.

Lo ideal dijimos que es que c1 sea grande comparada con c2 al principio, en las primeras iteraciones y luego vaya decreciendo, mientras que c2 sea pequeña al principio y luego vaya incrementándose para que el algoritmo tienda a converger en las últimas iteraciones pero pueda realizar una buena exploración del espacio en las primeras iteraciones.



**EJERCICIO 3**

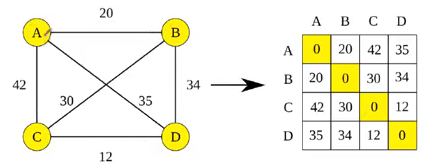
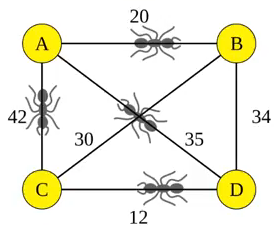
<https://drive.google.com/drive/folders/1U0tUQK7EB05OsQpdo2_qdjOlkuqAzrUH>

En la imagen vemos un ejemplo para este problema del agente viajero, donde tenemos cuatro ciudades (A, B, C, D), conectadas entre sí. El problema consiste en partir desde una ciudad y hacer un recorrido que nos permita pasar por las otras tres ciudades y volver a la de origen, sin repetir ninguna ciudad. Por ejemplo, un recorrido que sea: A – B – D – C – A.

Es un problema simétrico, ya que viajar a “A” a “B” o de “B” a “A” tiene la misma longitud.

También se puede representar con una matriz de conexión como se muestra en la imagen, donde la intersección entre una fila y una columna nos dice la distancia que hay para viajar entre dos ciudades.

La diagonal en esa matriz podemos ver que son todos 0, ya que viajar de una ciudad a la misma no tiene distancia. Y es simétrico como dijimos antes.

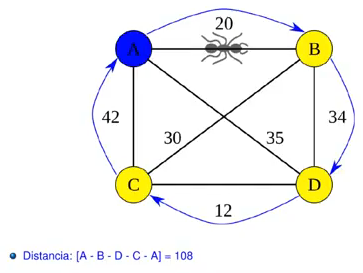
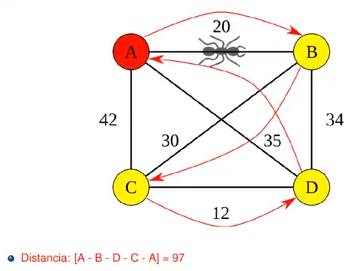
En este contexto, aplicar el problema de hormigas consistirá en poner cada hormiga en un nodo, podría ser todas en el mismo o en nodos diferentes, y permitir que cada hormiga haga un recorrido eligiendo las ciudades y volviendo a la de origen y proponga una solución.

Algo interesante para este problema es que *no importa cuál sea el nodo de origen para comenzar el camino, la solución debería ser la misma, ya que es un recorrido cerrado*. Por ejemplo, si hacemos el recorrido A – B – D – C – A, sería equivalente al recorrido D – C – A – B – D.

Entonces la idea es que cada hormiga parta de un nodo y proponga un recorrido teniendo en cuenta las distancias y matriz de feromonas que veremos luego. En la imagen de abajo a la izquierda tenemos un ejemplo donde la hormiga parte del nodo A y hace el recorrido A – B – C – D – A y el recorrido tiene una distancia de 108 unidades.

Otro recorrido sería el de la imagen de la derecha, donde el recorrido es diferente y la distancia total también, ahora es menor.

*La idea es que las hormigas vayan proponiendo recorridos hasta encontrar el de menor longitud*.

**Repaso del algoritmo de sistema de hormigas**.

Se comienza inicializando la matriz de feromonas. La matriz de feromonas es una estructura donde las hormigas van a ir guardando la información de las soluciones que encuentran, para compartirlas con las demás hormigas. Para inicializar esa matriz se sugiere utilizar valores pequeños entre 0 y un valor cercano a 0, extraídos a partir de una distribución uniforme. En la práctica, también se puede usar una inicialización donde todos los valores sean iguales a 1, total a lo largo del algoritmo ese valor se irá actualizando y no habrá problemas.

Algo interesante a tener en cuenta es que, la matriz de distancia para representar esa ciudad del ejemplo, será de 4x4, ahora bien, la matriz de feromonas también será de 4x4 en este ejemplo, ya que en cada elemento guardamos la cantidad de feromonas que deja la hormiga cuando pasa de una ciudad a otra.

Una vez inicializada la matriz de feromonas, ubicamos “N” hormigas en el nodo de origen. Podríamos poner todas en el mismo nodo o ubicarlas en distintos nodos, total las soluciones que se encuentren serán las mismas al ser un recorrido cerrado. Lo único que si elegimos distintos nodos como nodo inicial, hay que hacer luego un alineamiento para verificar que las soluciones sean iguales (después lo vemos).

Después se comienza con el proceso de búsqueda iterativo. Cada iteración va a consistir en lo siguiente:

-Primero, para cada hormiga definimos una lista vacía donde vamos a almacenar las ciudades en el orden en que se van recorriendo, y el primer elemento de la lista será el nodo o ciudad inicial del que parte la hormiga.

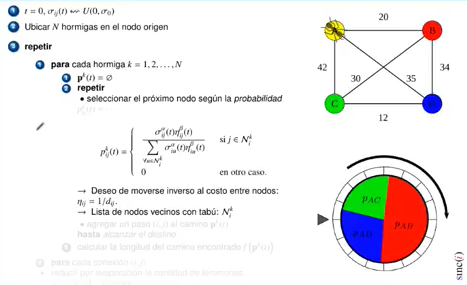
-Luego de forma iterativa, para cada hormiga y dado el nodo en el que estamos parados, vamos a elegir cuál es la siguiente ciudad a la que nos vamos a mover, con una probabilidad que viene dada por la ecuación de la imagen. En el numerador aparece el valor de feromonas que tenemos almacenado para movernos de la ciudad “i” a la ciudad “j”, que sería σij, por ejemplo, para movernos de A hacia B, y también está el valor “ƞ” que se calcula como la inversa de la distancia entre las dos ciudades. Ese valor se podría tener almacenado previamente, lo que sí hay que tener cuidado con los elementos de la diagonal, porque las distancias de un nodo consigo mismo son 0, así que habría divisiones por 0 que debemos evitar.

Mientras que en el denominador tendremos una sumatoria de todos los posibles valores de transición, que serían los productos de “σ” y “ƞ” que se usaban en el numerador, entonces se va a normalizar esos productos del numerador dividiendo por la sumatoria de todos los productos de las transiciones.

Se debe tener en cuenta que en la sumatoria interviene ese parámetro “N” que va a ser una lista de vecinos con taboo. ¿Qué significa eso? Supongamos que del nodo B nos movemos al nodo A, ahora dado A debemos elegir a qué nodo nos vamos a mover, que será C, D o B, pero como venimos de B, esa lista N de posibles nodos para movernos lo que hace es quitar B de las posibilidades para movernos.

-Entonces vamos a calcular una probabilidad “p” de esa manera y luego hacemos lo mismo que en el algoritmo genético, vamos a normalizar esas probabilidades y aplicar una especie de método de ruleta (imagen de la derecha) para elegir a qué ciudad nos vamos a mover. En ese ejemplo, movernos de A hacia B tiene más probabilidad que movernos de A hacia C o hacia D.

Entonces, normalizamos las probabilidades “p” calculadas, tiramos un número al azar entre 0 y 1, y dependiendo de la proporción que esa probabilidad representa en la ruleta nos da la siguiente ciudad donde nos vamos a mover.



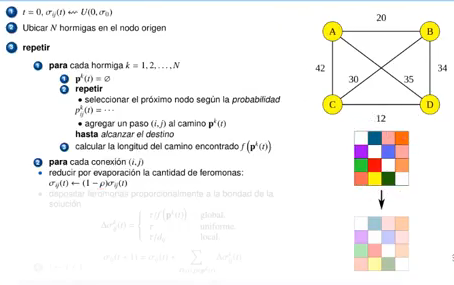
-Una vez elegida la siguiente ciudad, vamos a agregar esa ciudad al camino que estamos construyendo y vamos a repetir esto hasta que no tengamos más ciudades, es decir, hasta que todas las ciudades disponibles formen parte del camino.

Llegados a ese punto, recordar que si partimos de la ciudad A, a mano deberíamos agregar al final de la lista la ciudad A nuevamente, para volver al nodo de origen.

-Luego calculamos la longitud del camino encontrado sumando todas las distancias entre las transiciones de ciudades que fuimos realizando y que forman la solución.

-Una vez completada la búsqueda con cada hormiga, realizamos el proceso de evaporación de feromonas multiplicando la matriz de feromonas por un valor que es la tasa de evaporación. La idea es que las feromonas almacenadas en cada una de las transiciones o caminos, pierda una pequeña proporción debido a la evaporación y la hormiga pierda parte de la información de los caminos que son menos utilizados.

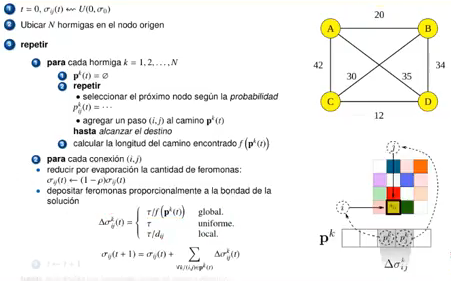
Si bien en la imagen esta expresado como una operación elemento a elemento, el valor “*p*” es constante se aplica a todos los elementos de la matriz, entonces se puede hacer una operación matricial.



-Una vez que se tiene eso listo, se hace el depósito de feromonas teniendo en cuenta las soluciones encontradas. Para eso hace falta calcular cuánta feromona vamos a depositar en cada camino que fuimos recorriendo, por ejemplo, para la transición de A hacia B decidir cuánta feromona vamos a dejar. Para eso, como se ve en la imagen de abajo vamos a considerar tres métodos: el método uniforme supone depositar una cantidad constante τ en cada transición, el método global propone dejar una fracción de ese total τ de feromonas, que va a depender de la longitud total del camino, es decir, divide esa cantidad total de feromonas por la longitud total del camino. Cuanto más corto sea el camino más feromonas vamos a dejar, pero en todas las transiciones va a dejar la misma cantidad. Mientras que en el método local, la cantidad de feromonas τ se divide por la distancia que hay para moverse de la ciudad “i” a la “j” (dij), es decir, esa τ que vamos a depositar en cada transición va a ser distinto, ya que las distancias entre nodos o ciudades son distintas.

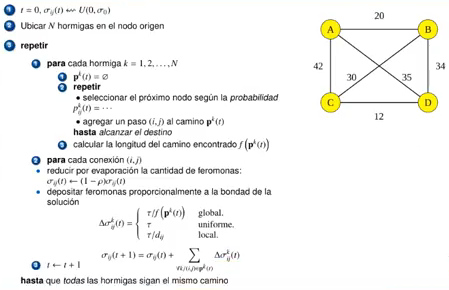
De esa forma vamos a calcular cuánta cantidad de feromona Δσij vamos a dejar en una transición.

-Teniendo en cuenta ese delta de feromonas, vamos a actualizar la matriz de feromonas recorriendo todas las transiciones que realizaron cada una de las hormigas y, de acuerdo al método de depósito de feromonas que elegimos, ir sumándole al valor de feromonas que tenemos en la matriz, el valor Δσij que calculamos antes de acuerdo al método.



-Una vez hecho esto, comenzamos una nueva iteración y se repite hasta alcanzar un criterio de corte.

-En este punto, vamos a hacer dos aclaraciones. La primera es que, como toda meta heurística hace falta elegir un número máximo de iteraciones, para evitar que el algoritmo se ejecute de forma indefinida. Además, vamos a elegir un criterio específico para este problema, y es que todas las hormigas sigan el mismo camino, y acá es importante tener en cuenta si todas las hormigas parten del mismo nodo inicial o no. Si elegimos que todas parten del mismo nodo inicial directamente podemos comparar las secuencias que obtuvimos con cada hormiga y deberían ser iguales. Mientras que, si no partimos con todas del mismo nodo, se deben alinear las secuencias para determinar si efectivamente son las mismas soluciones.

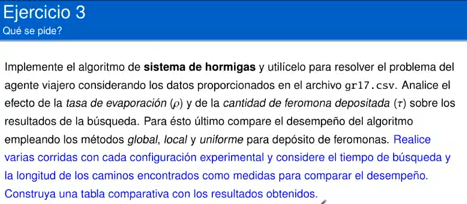


Otras sugerencias:

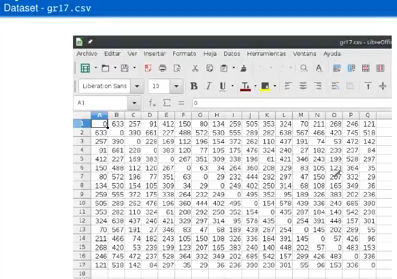
-No nos quedemos con una sola iteración donde se cumpla que todas las hormigas siguen el mismo camino, ya que podría darse aleatoriamente que en una iteración se alinean todas las hormigas y en la siguiente no, entonces conviene tener un contador y, por ejemplo, si todas las hormigas se alinean durante cinco iteraciones consecutivas ahí sí detener el algoritmo.

-Conviene incluir en el algoritmo una hormiga elite, es decir, ir almacenando en cada iteración la mejor solución encontrada hasta el momento, es decir, el mejor recorrido (que tiene la menor longitud), y actualizándolo a lo largo de las iteraciones. Y al final devolvemos la mejor solución.

**Volviendo a lo que se pide en el ejercicio**



Respecto al archivo de datos, tendrá la estructura que se muestra en la imagen, similar a la que vimos en el ejemplo, donde podemos ver que en la diagonal tendrá todos valores 0 (distancias de una ciudad a ella misma), y es simétrico, ya que viajar de A hacia B tiene la misma distancia que viajar de B hacia A.



Respecto a lo que se pide, la idea es que creemos tablas similares a las que se muestran abajo.

Eligiendo un valor de tasa de evaporación “*p*”, la idea es que probemos distintos valores de “τ”, que era la cantidad de feromonas que dejamos en el depósito (por ejemplo, 0.1, 1 y 10), y con esos valores y para cada método de depósito de feromonas que vimos, realicemos “N” recorridos.

Por ejemplo, con *p* = 0.1, τ = 0.1 y el método de depósito de feromonas uniforme, repetir N = 10 veces la corrida y con esas 10 soluciones calcular el tiempo promedio que llevo encontrar la solución, la distancia promedio de la solución encontrada y el número de iteraciones promedio utilizado. Y lo mismo para las demás combinaciones de parámetros.

Luego empleando otra tasa de evaporación “*p*” realizamos lo mismo y comparamos cuál es el efecto de esos parámetros sobre la calidad de las soluciones encontradas.

