Modelos Bioinspirados y Heurísticas de Búsqueda

Practica 1

“Algoritmos basados en Entornos y Trayectorias”

Antonio José Morano Moriña

Modelos Bioinspirados

Índice

El objetivo a cumplir con la práctica es la codificación de distintos algoritmos con la finalidad de optimizar los beneficios que va a obtener una instalación solar, ajustando un array que contendrá la estrategia que seguiremos durante las 24 horas del día.

Base

La base de la practica consiste en implementar:

1. **Función de evaluación,** que será la que nos permita comprobar el beneficio que obtendrán cada uno de los arrays con diferentes soluciones que iremos comprobando con los algoritmos.

```python
# FUNCIÓN DE EVALUACIÓN
#No se puedo comprar y vender a la vez,por eso, en caso de que se vaya a comprar mas de lo
#que cabe en la bateria , directamente no se compra y sigue a la siguiente
#posicion del array
def fEvaluacion(sActual):
    beneficio = 0
    bateria = 0
    for i in range(0, len(sActual)):
        #Comprobamos antes que nada que no haya desbordes de energia perdida
        energia = generacion[i] + bateria
        if energia > capacidad:
            venta = capacidad - energia
            beneficio += pVenta[i] * venta * -1
            bateria = capacidad
        else:
            bateria = energia
        if sActual[i] > 0: # Venta de energía
            venta = energia * sActual[i] / 100
            beneficio += pVenta[i] * venta
            bateria = energia - venta
        elif sActual[i] < 0: # Compra de energía
            espacio = capacidad - bateria
            compra = espacio * -1 * sActual[i] / 100
            if espacio < compra: #Si vamos a comprar mas de lo que nos cabe, cancelamos la compra
                compra = 0
            else:#Si cabe compramos y añadimos esa energia a la bateria
                bateria += compra
            beneficio -=  pCompra[i] * compra

    #Al terminar, si queda bateria la vendemos al precio de la ultima hora
    beneficio += pVenta[-1] * bateria
    return beneficio
```

La estrategia que se realiza con esta función es la siguiente:
Lo primero que se hace cada iteración es añadir a la batería la energía que se generara durante dicha hora de sol, y en caso de que con esa energía se rebase la capacidad de la batería pues la vendemos.
Tras añadir la energía, hay dos opciones:
Se vende energía: En caso de que el valor de la posición actual del array sea mayor que 0, vamos a vender dicho porcentaje de la capacidad de nuestra batería al precio que se venda a esa hora, por lo que cogemos la energía actual y le sacamos el porcentaje equivalente a la cantidad a vender, para saber cuanto vendemos y multiplicamos la cantidad vendida por el valor que tenga en dicha hora y restamos la energía vendida al total de la batería.
Se compra energía: En caso de comprar primero he comprobado cuanto queda para que la batería este al límite, y sacamos cuanto vamos a comprar, en caso de comprar más de lo que cabe, se cancela la compra poniendo su valor a 0 y se pasa a la siguiente hora, si tenemos espacio para almacenar dicha energía pues la sumamos.
Restamos al beneficio actual el dinero invertido en comprar la energía.

Al acabar, antes de devolver el beneficio resultante, le suma la venta de la cantidad de energía que quedara en la batería por el precio de venta de la ultima hora, dejando así la batería vacía al terminar de iterar y no se queda ahí malgastada.

2. **Generador de solución inicial,** Para el problema a resolver he generado una función que crea un array de 24 posiciones (una por cada hora del día) con números aleatorios entre -100 y 100 en cada una de estas.

```
# GENERACIÓN DE LA SOLUCIÓN INICIAL ALEATORIA
def genSolucion() :
    sAleatoria = [rnd.randint(-100, 100) for _ in range(0, len(radiacion))]
    return sAleatoria
```

3. **Operador de movimiento,** he creado una función que genera un vecino, pasándole un array, la posición a modificar y el valor de la granularidad con la que se creará el vecino.
   Copia el array pasado, luego coge el valor de la posición pasada sumándole la granularidad , comprueba que este dentro de los valores válidos para el problema, y lo vuelve a almacenar en la posición pasada. En caso de que salga de estos valores, los deja en los límites permitidos. (+-100)

```
# OPERADOR DE MOVIMIENTO
g = 1  # Granularidad
def genVecino(sActual, pos, g):
    sVecina = sActual.copy()
    valorPos = sVecina[pos] + g
    if valorPos <= -100:  # No puede pasar de 100 ni de -100
        sVecina[pos] = -100
    elif valorPos >= 100:
        sVecina[pos] = 100
    else:
        sVecina[pos] = valorPos
    return sVecina
```

Algoritmo Greedy

Para la codificación de este algoritmo, he creado una función que se encargue de generar el array con la heurística que se pide, y evaluando esta solución con la función de evaluación. (cuyo beneficio siempre será el misma a no ser que sé que cambie la función de evaluación)

```
def solGreedy(): #Solo se llama a la funcion de coste una vez ya que genera un array y s
    maxPrecio = np.argmax(pVenta)
    sGreedy = [0 for _ in range(0, len(radiacion))]
    sGreedy[maxPrecio:len(radiacion)] = [100 for _ in range(maxPrecio, len(radiacion))]
    beneficio = fEvaluacion(sGreedy)
    return beneficio ,sGreedy , 1
```

Primero guardo el máximo valor de venta en una variable y creo un array formado de 24 ceros, luego a ese array le cambio los 0 a partir de la posición del máximo precio de venta(esta incluida) por 100.

La función devuelve el beneficio, la solución que genera ese beneficio y el numero de llamadas a la función de coste, que en este caso es una solamente ya que se evalúa un array generado siguiendo una heurística y se devuelve su beneficio.

Búsqueda Aleatoria

Este algoritmo es básicamente crear 100 soluciones aleatorias y quedarse con la mejor, como ya tenemos una función que nos genera una solución aleatoria válida para el problema, solo tendremos que realizar 100 llamadas a esta y quedarnos con la mejor de las resultantes.

```
#BUSQUEDA ALEATORIA
def busquedaAleatoria(): #Se llama a la f
    bMejor = 0
    for i in range(0, 100):
        sActual = genSolucion()
        bActual = fEvaluacion(sActual)
        if (bActual > bMejor): #Si la sol
            sMejor = sActual
            bMejor = bActual
    return bMejor, sMejor , 100
```

Al principio el mejor beneficio es 0, ya que aun no hemos visto ninguna solución, creamos un bucle de 100 iteraciones, y en cada una de ellas comprueba si su beneficio supera al de la mejor, si lo supera esa solución se convierte en la mejor.

```
Solucion Busqueda Aleatoria  (519.5844017263203, [-69, 86, -97, 78, -60,
-30, 62, -26, 86, 64, -35, -57, 9, 66, -12, -47, 37, -73, 9, 12, 80, 89,
8, -72], 100)
```

Ejemplo de ejecución para la búsqueda aleatoria con semilla 50.

Búsquedas locales

Primer Mejor

```
def BL_VecinoPrimerMejor(sIni):
    sMejor = sIni  #La primera que leemos es la mejor po
    bMejor = fEvaluacion(sMejor)
    llamadasCoste = 1
    enc = False
    while enc == False and llamadasCoste < 3000: # Nos q
        for i in range (len(sMejor)):
            sAux = genVecino(sMejor,i,g)
            bAux = fEvaluacion(sAux)
            llamadasCoste+=1
            if bAux > bMejor:
                sMejor = sAux
                enc = True

    return fEvaluacion(sMejor) , sMejor , llamadasCoste
```

Primero ponemos como mejor solución la pasada por parámetro, para que todos los algoritmos trabajen con el mismo array y poder comparar como recorre cada uno el espacio de búsqueda.

Creamos un bucle en el que estaremos hasta recorriendo el espacio de búsqueda hasta que uno de los vecinos generados obtenga mejor beneficio que el inicial, o hasta que se supere el limite de 3000 llamadas a la función de evaluación sin ninguna mejora.

En cada iteración crearemos un vecino para +g y otro para cada -g si alguno de ellos mejora, saldremos del bucle, si no pasaremos a la siguiente posición del array para generar el siguiente vecino (la posición que cambia es la de la iteración en la que nos encontremos).

```
Solucion inicial generada a pasar a los algoritmos->  [27, -32, -7, 63,
-38, 77, 21, 96, -16, -79, 37, -19, -43, 73, 42, -79, -61, -12, -75, -12,
-19, -44, -52, -83]
--------------------------------------------------
Solucion PRIMER vecino :  (298.75649562966925, [27, -31, -6, 64, -37, 78,
22, 97, -15, -79, 38, -18, -42, 74, 42, -78, -60, -11, -74, -11, -18, -43,
-51, -82], 25)
```

Ejemplo de ejecución del primer vecino con semilla 50.

Mejor Vecino

```python
def BL_VecinoMejor(sIni): # Busca la mejor solución del entorno
    sActual = sIni # Generación solución inicial
    bActual = fEvaluacion(sActual)
    llamadasCoste = 1 #la de arriba
    vecesMejora =0
    while True:
        sMejor = sActual # La actual será la mejor si no se encuentran mejores
        bMejor = bActual
        for i in range(len(sActual)): # Buscamos el mejor vecino del entorno
            for j in range(2): #+ y -
                if j == 0: # +g
                    sVecina = genVecino(sActual,i,g)
                    bVecina = fEvaluacion(sVecina)
                    llamadasCoste+=1
                    if bVecina > bMejor: #Si el vecino supera al mejor d
                        sMejor = sVecina
                        bMejor = bVecina
                else: # -g
                    sVecina = genVecino(sActual,i,-g)
                    bVecina = fEvaluacion(sVecina)
                    llamadasCoste+=1
                    if bVecina > bMejor:
                        sMejor = sVecina
                        bMejor = bVecina
        # Si el mejor vecino del bucle mejora al actual se sustituye y se vuelve a iterar.
        if bMejor > bActual:
            vecesMejora+=1
            sActual = sMejor
            bActual = bMejor
        else: # Si no mejora, salimos del algoritmo.
            break

    return fEvaluacion(sActual) , sActual , llamadasCoste+1 , vecesMejora
```

Primero ponemos como mejor solución la pasada por parámetro, para que todos los algoritmos trabajen con el mismo array y poder comparar como recorre cada uno el espacio de búsqueda.

Esta vez la lógica es similar al primer mejor, pero creamos una solución mejor (sMejor) para poder guardarla ya que aquí no salimos del algoritmo tras la primera mejora, consta de un bucle infinito donde, empezamos generando el espacio de búsqueda al completo, creando un

vecino con +g y otro con -g, son guardados en sVecina y comparados con la sMejor, nos quedaremos con el mejor del espacio de búsqueda actual y luego comprobaremos si ese mejor vecino del bucle es mejor que la solución actual, si lo es volvemos a iterar pero esta vez generaremos el espacio de búsqueda con esta nueva solución, y así hasta que el mejor vecino del bucle sea peor, que será cuando acabe el algoritmo que para salir hace un break.

La función devuelve el beneficio, la solución que lo genera, las llamadas a la función de evaluación y el numero de veces que el mejor vecino del bucle era mejor que el actual.

```
Solucion inicial generada a pasar a los algoritmos->  [27, -32, -7, 63,
-38, 77, 21, 96, -16, -79, 37, -19, -43, 73, 42, -79, -61, -12, -75, -12,
-19, -44, -52, -83]
```

```
Solucion MEJOR vecino :  (374.78920380000005, [27, 0, 0, 63, 0, 79, 21,
96, 0, -100, 1, -19, -43, 1, 1, -79, -61, -12, -75, -12, -19, -44, -52,
-83], 12770, 265)
```

Ejemplo de ejecución con semilla 50.

Estudio de la granularidad del operador de movimiento para búsqueda local.

Para la experimentación que pide la práctica con la granularidad del operador de movimiento, he seleccionado tres valores distintos, aquí están los resultados obtenidos para la ejecución de cada uno de los algoritmos de búsqueda local en función de sus diferentes valores:

-g = 1

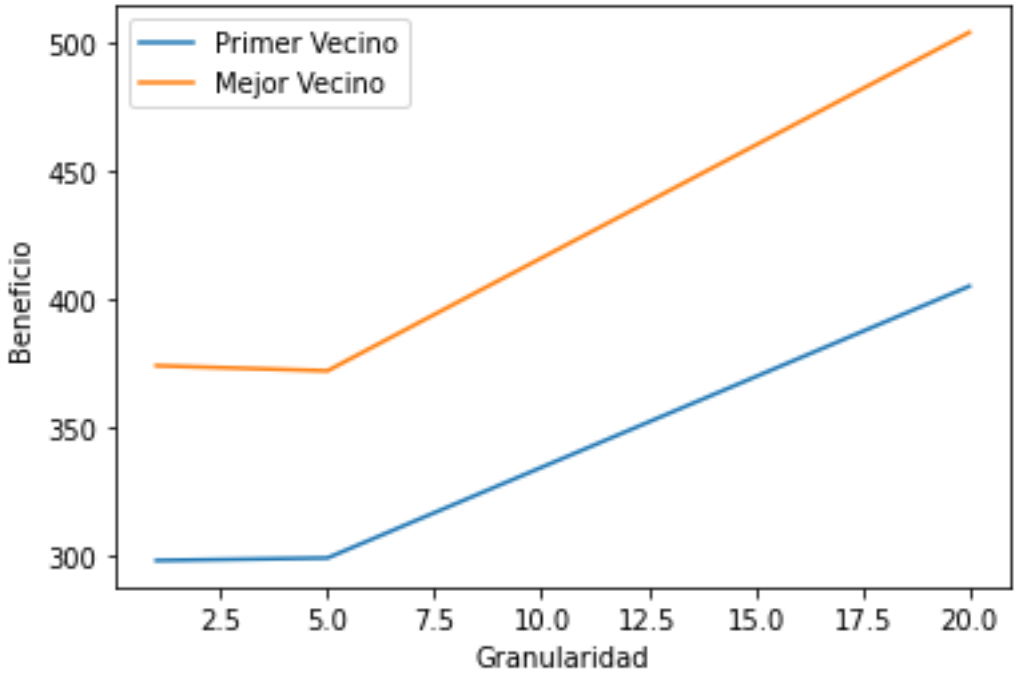
```
Solucion inicial generada a pasar a los algoritmos->  [27, -32, -7, 63,
-38, 77, 21, 96, -16, -79, 37, -19, -43, 73, 42, -79, -61, -12, -75, -12,
-19, -44, -52, -83]
--------------------------------------------------
Solucion PRIMER vecino :  (298.75649562966925, [27, -31, -6, 64, -37, 78,
22, 97, -15, -79, 38, -18, -42, 74, 42, -78, -60, -11, -74, -11, -18, -43,
-51, -82], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (374.78920380000005, [27, 0, 0, 63, 0, 79, 21,
96, 0, -100, 1, -19, -43, 1, 1, -79, -61, -12, -75, -12, -19, -44, -52,
-83], 12770, 265)
--------------------------------------------------
```

-g = 5

```
Solucion inicial generada a pasar a los algoritmos->  [27, -32, -7, 63,
-38, 77, 21, 96, -16, -79, 37, -19, -43, 73, 42, -79, -61, -12, -75, -12,
-19, -44, -52, -83]
--------------------------------------------------
Solucion PRIMER vecino :  (298.9283774048, [27, -27, -2, 68, -33, 82, 26,
100, -11, -79, 42, -14, -38, 78, 42, -74, -56, -7, -70, -7, -14, -39, -47
-78], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (372.97401194014407, [27, 3, 3, 63, -3, 77, 26,
96, -1, -100, 2, -19, -43, 3, 2, -79, -61, -12, -75, -12, -19, -44, -52,
-83], 2642, 54)
```

-g = 20

```
Solucion inicial generada a pasar a los algoritmos->  [27, -32, -7, 63,
-38, 77, 21, 96, -16, -79, 37, -19, -43, 73, 42, -79, -61, -12, -75, -12,
-19, -44, -52, -83]
--------------------------------------------------
Solucion PRIMER vecino :  (405.5161632060032, [27, -12, 13, 83, -18, 97,
41, 100, 4, -79, 57, 1, -23, 93, 42, -59, -41, 8, -55, 8, 1, -24, -32,
-63], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (504.2455476000001, [27, 8, 13, 63, 2, 77, 21,
96, 4, -100, 17, 61, -43, 13, 2, -79, -61, 8, -75, 8, 1, -44, -52, -83],
1058, 21)
```



Observando los resultados obtenidos podemos decir que conforme aumenta la granularidad los beneficios han sido mejores en ambos algoritmos, ya que al realizar cambios mas “bruscos” en los valores, explora otras regiones del espacio a las que antes no llegaba ya que se estancaba en mínimos locales de los que no salía con valores menores de granularidad. Aunque estos cambios también pueden hacer que pase por alto regiones del espacio donde puede haber soluciones mejores.

VND

```
#Array que contiene los 5 niveles de velocidad variable
gVariable = [1,5,10,15,25]
def VND(sIni):
    sActual = sIni
    bActual = fEvaluacion(sActual)
    llamadasCoste = 1
    k=0
    while k < len(gVariable):
        sMejor = sActual #La actual será la mejor si no se encuentran mejores
        bMejor = bActual
        for i in range(len(sActual)): # Obtención del mejor vecino del entorno
            for j in range(2):
                if j == 0: # +g
                    sVecina = genVecino(sActual,i,gVariable[k])
                    bVecina = fEvaluacion(sVecina)
                    llamadasCoste+=1
                    if bVecina > bMejor:
                        sMejor = sVecina
                        bMejor = bVecina
                else: # -g
                    sVecina = genVecino(sActual,i,-g)
                    bVecina = fEvaluacion(sVecina)
                    llamadasCoste+=1
                    if bVecina > bMejor:
                        sMejor = sVecina
                        bMejor = bVecina
        # Si la mejor de las vecinas mejora a la actual se sustituye y se vuelve a iterar
        if bMejor > bActual:
            sActual = sMejor
            bActual = bMejor
            k=0 #Si hay una mejor reiniciamos el valor de k
        else: #Si no, aumentamos k en 1 y variamos el nivel de granularidad de la siguiente iteracion
            k = k +1

    return fEvaluacion(sActual) , sActual , llamadasCoste+1
```

Para realizar el VND he usado la idea dada en clase, de crear diferentes niveles de granularidad, guardados en gVariable, e ir usándolos en la ejecución.

El algoritmo es el mismo que la búsqueda del MEJOR vecino, solo que en caso de no encontrar un vecino mejor no sale del bucle, si no que aumenta el valor de k haciendo que cambie la granularidad y volviendo a iterarse. En el código es, cambiar la condición de parada por que k sea menor que los valores de granularidad de los que dispongamos (len(gVariable)), y en el else donde antes había un break para terminar el algoritmo, lo sustituimos por un incremento del valor de k.

Cada vez que mejora en algún valor de k, reinicia el valor de k a 0 y vuelve a recorrer el espacio con todas las granularidades, el algoritmo finaliza si se han realizados tantas iteraciones como valores de granularidad tengamos, sin mejora en ninguna es estas, así que para salir tendrá que fallar al encontrar un vecino peor en cada espacio de búsqueda generado para cada uno de los valores de granularidad.

```
Solucion inicial generada a pasar a los algoritmos->  [27, -32, -7, 63,
-38, 77, 21, 96, -16, -79, 37, -19, -43, 73, 42, -79, -61, -12, -75, -12,
-19, -44, -52, -83]
```

```
--------------------------------------------------
Solucion VND  (541.8729174, [27, 0, 0, 63, 0, 79, 21, 96, 0, -100, 1, 72,
-43, 1, 1, -79, -61, 1, -75, 1, 1, -44, -52, -83], 17426)
```

Ejemplo de ejecucion para la semilla 50.

Enfriamiento simulado

Esquema enfriamiento

```python
def Cauchy(t0,k):
    Tnueva = t0 / (1 + k)
    return Tnueva
```

Esta función se encarga de enfriar la temperatura en cada iteración, siguiendo el esquema de Cauchy

T0

```python
prob = 0.15
mu = 0.25
def CalculoT0 ():
    bGreedy , _ , _ = solGreedy() #Coste devuelto por el greedy
    T0 = mu / -(np.log(prob)) * bGreedy
    return T0
```

Se encarga de genera el valor de la temperatura inicial, en función de los parámetros Φ,μ y del coste de la solución obtenido con el algoritmo greedy.

Algoritmo

```python
kMax = 100 #numero max de iteraciones
lt= 10 #L(T) vecinos por iteracion
def EnfriamientoSimulado(sIni,Tf):
    llamadasCoste = 0
    sActual = sIni
    sMejor = sActual
    tIni= CalculoT0()
    k=0
    temp = tIni
    while k < kMax:
        solAceptada=0
        solNoAceptada=0
        k+=1
        for i in range (0,lt):
            if(rnd.random() > 0.5):    #+-g de forma aleatoria y vecino con
                sCand= genVecino(sActual,rnd.randint(0, 23),g)
            else:
                sCand= genVecino(sActual,rnd.randint(0, 23),-g)
            difCoste = fEvaluacion(sCand) - fEvaluacion(sActual)
            llamadasCoste+=2
            if (difCoste < 0) or (rnd.random() < np.exp(-difCoste/temp)):
                solAceptada+=1
                sActual = sCand
                bActual = fEvaluacion(sCand)
                if fEvaluacion(sMejor) < bActual:
                    sMejor = sActual #Guardamos la mejor solucion por si nos
                    llamadasCoste+=2
            else:
                solNoAceptada+=1
        temp = Cauchy(tIni,k) #enfriamos tras generar 10 vecinos
    #Porcentaje de soluciones no aceptadadas
    porcentajeRechazo = (solNoAceptada/(solAceptada+solNoAceptada)* 100)
    return fEvaluacion(sMejor) , sMejor ,llamadasCoste+1 , porcentajeRechazo
```

Primero ponemos como mejor solución la pasada por parámetro, para que todos los algoritmos trabajen con el mismo array y poder comparar como recorre cada uno el espacio de búsqueda.

Luego calculamos la temperatura inicial llamando a la función CalculoT0, la cual devuelve la temperatura resultante de la formula propuesta en la práctica, y igualamos la temperatura actual (temp) a la misma. Se ejecuta un bucle para el número máximo de iteraciones que pasemos, en este caso y teniendo en cuenta la experimentación posterior, fui probando y decidí dejarla en 100 y para cada iteración de este, se itera otro bucle donde se crean L(T) vecinos una cantidad determinada de vecinos, fui probando con diferentes valores y los he dejado en 10.

Seleccionamos de forma aleatoria si elegir la granularidad positiva o negativa (ambas con la misma probabilidad) y al candidato generado lo evaluamos, en difCoste guardamos el resultado de restar el beneficio del candidato con el mejor. Si es mayor que 0 estamos ante una solución mejor por lo que la cambiamos, y si es menor depende de la diferencia de costes entre la solución actual y la vecina, y de la temperatura T que se acepte la probabilidad de aceptación de una solución peor.

Una vez terminemos de generar los 10 vecinos por iteración, llamaremos a la función Cauchy(x,y) que se encarga de cambiar la temperatura actual por el resultado de la fórmula de Cauchy.

Antes tenía que se podía salir del bucle cuando temp fuera menor que Tf, que la paso por parámetro, pero tendía a finalizar muy pronto y se estancaba en mínimos locales, dando así peores resultados por falta de exploración del entorno.

La función devuelve el beneficio, la solución que lo genera, las llamadas a la función de evaluación y el porcentaje de rechazo, que corresponde a los 10 vecinos que se generan en la última iteración y cuales de estos son rechazados

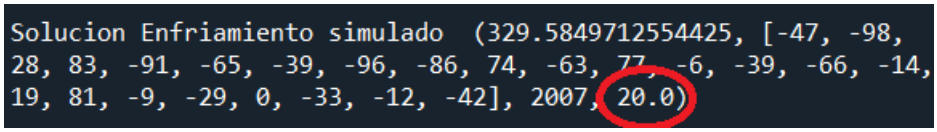
Experimentación Φ(prob) y μ (mu)

En la práctica se pide realizar una experimentación para estos valores que influyen en el calculo de la temperatura inicial y por tanto en la probabilidad de que aceptemos o no diferentes soluciones, la experimentación la he realizado en la semilla 15.

Como se vio en la función de arriba, las variables solAceptada y solNoAceptada, se encargan de controlar si se ha cambiado la solución o no durante la iteración en cuestión.

He devuelto el porcentaje de soluciones no aceptadas con la función del algoritmo, y he ido probando para los diferentes valores de cada una de las variables, se pide experimentar ambas variables entre los valores 0.1 y 0.3, los resultados obtenidos durante la experimentación para los valores elegidos han sido:

Para el valor Φ = 0.1 y μ = 0.1:



Para el valor Φ = 0.1 y μ = 0.3:

```
Solucion Enfriamiento simulado  (350.79425488616005, [-47, -80,
38, 100, -86, -65, -49, -85, -76, 74, -63, 77, -6, -29, -56, -19
19, 76, -9, -44, 10, -28, -32, -42], 2031, 30.0)
```

Para el valor Φ = 0.2 y μ = 0.3:

```
Solucion Enfriamiento simulado  (351.64319778156005, [-47, -75,
38, 100, -86, -65, -44, -85, -71, 79, -58, 77, -11, -34, -56,
-19, 14, 76, -9, -44, 10, -28, -37, -42], 2033, 10.0)
```

Para el valor Φ = 0.15 y μ = 0.25:

```
Solucion Enfriamiento simulado  (350.79425488616005, [-47, -80,
38, 100, -86, -65, -49, -85, -76, 74, -63, 77, -6, -29, -56, -19
19, 76, -9, -44, 10, -28, -32, -42], 2031, 20.0)
--------------------------------------------------
```

Observando los resultados obtenidos durante la experimentación, podemos ver que estos parámetros al afectar al porcentaje de elegir una opción peor, hacen que varie levemente el valor de las soluciones encontradas, aunque el mejor resultado del algoritmo se obtiene con unos valores con los que el porcentaje de rechazos es del 10% y en el enunciado de la practica pide que se obtenga los valores de estos parámetros para que el rechazo sea del 20%(Solo teniendo en cuenta las ultimas 10 soluciones), por lo que he dejado los valores 0,15 y 0,25, ya que aunque con ambas en 0,1 también se obtiene el mismo porcentaje de rechazo pero se queda estancado en un mínimo local, que sin embargo para los demás valores si es capaz de superar.

Búsqueda Tabú

Primero definir las funciones auxiliares que han sido necesarias para el algoritmo tabú,que son sobre todo funciones para poder crear y utilizar las matrices de probabilidad y de frecuencia:

```python
#BUSQUEDA TABU
#Funcion que crea la matriz de frecuencias, horas * valores posibles
def iniMatrizFr():
    rango = len(range(-100, 100 + g, g))
    horas = len(radiacion)
    matrizFr = np.ones((horas, rango))
    return matrizFr
```

La función se encarga de crear la matriz de frecuencias de tamaño horas * valores posibles, que registra el número de veces que cada valor de un atributo ha pertenecido a soluciones visitadas en la búsqueda.

```python
#Funcion que devuelve la matriz de frecuencias actualizada
def incMatrizFr(matrizFr, sol):
    matrizFrNueva = matrizFr.copy()
    for h, v in enumerate(sol): # Por cada par de hora,valo
        indiceV = int(10 + v / g)
        matrizFrNueva[h][indiceV] += 1 #Incrementamos la ca
    return matrizFrNueva
```

Primero creamos una matrizFr nueva, y para cada par hora valor, vamos a calcular el índice que tendría el valor del par en la matriz y después vamos a incrementar la casilla que corresponde al par de la hora con el indiceValor que acabamos de calcular

```python
def genSolucionGreedy(matrizProb):
    horas = len(radiacion)
    sGreedy = [0 for _ in range(horas)]
    for h in range(horas):
        num = rnd.random()
        suma = 0
        for v in range(len(sGreedy)):
            suma += matrizProb[h][v] #Incrementamos
            if num < suma:
                sGreedy[h] = v
                break # Se pasa a la siguiente hora
    return sGreedy
```

Esta función esta sacada de la dada en la practica de ejemplo, en ella, para cada hora generamos un numero aleatorio, y vamos incrementando la suma con cada valor que tenga en la matriz de probabilidad cada uno de los pares (hora,valor) (Estos valores ya están normalizados con respecto a cómo se pide en la practica,entre 0 y 1) , y si el random es menor se incluye el valor resultante en la hora correspondiente de la solucionGreedy.

```python
def genMatrizProb(matrizFr):
    valores = len(matrizFr[0, :])
    horas = [i for i in range(len(radiacion))]
    nhoras = len(horas)
    mInv = np.zeros((nhoras, valores))
    for h in horas: # Recorremos de todas las horas y todos los valores por hora
        for v in range(valores):
            frec = matrizFr[h][v] #Obtenemos frecuencia
            if frec == 0: # prob max en 0
                prob = float('inf')
            else: # Si no,prob sera la inversa de la frecuencia
                prob = 1 / frec
            mInv[h][v] = prob # Completamos la matriz de inversas
    sumInv = [] # Normalizamos,para lo que calculamos el sumatorio de las inversas por hora
    for h in horas:
        sumInv.append(sum(mInv[h]))

    matrizProb = np.zeros((nhoras, valores))
    for h in horas: # Recorrido de todas las horas y todos los valores por hora
        for v in range(valores): #Por cada valor calculamos la casilla y actualizamos la matriz
            norm = mInv[h][v] / sumInv[h]
            matrizProb[h][v] = norm

    return matrizProb
```

Generamos la matriz de probabilidades que usaremos en caso de que tengamos que generar una solución greedy tras la reinicializacion del algoritmo, y en este guardaremos los valores normalizados (la probabilidad de aparición pasado a valores entre 0 y 1)

Para generar la matriz de probabilidades primero creamos una matriz inversa de tamaño horas * valores posibles(mInv), vamos recorriendo todas las horas y sus respectivos valores, para cada par hora, valor obtenemos su frecuencia y según el resultado de esta, si es 0 la probabilidad de aparicion será máxima, por lo que la ponemos en infinito, si no, será la inversa de la frecuencia, ahora para cada hora se deberá calcular las inversas de los valores acumulados y luego normalizar la solución para que tengo valores entre 0 y 1.

Con esto completamos la matriz de inversas, luego necesitamos normalizar para lo que vamos a sumar el valor de todas las inversas por hora y guardarlo(sumInv) para después de crear la matriz de probabilidades(matrizProb), añadiéndole para cada par hora valor el resultado de dividir el valor de ese par en la matriz inversa partido por la que contiene el sumatorio de las

inversas,que será la probabilidad de aparición normalizada a un valor entre 0 y 1. (mInv/sumInv).

```python
def busquedaTabu(sInicial, maxIteraciones, tamTabu, nVecinos):
    sActual = sInicial
    bActual = fEvaluacion(sActual)
    llamadasCoste=1
    sMejor = sActual
    bMejor = bActual
    matrizFr = iniMatrizFr() # Inicializar matriz de frecuencias
    lTabu = [] # Inicializar lista tabú
    for i in range(maxIteraciones): # Ejecutamos las iteraciones
        hora = np.nan
        valor = np.nan
        for j in range(nVecinos):
            #Vecino aleatorio, tanto en pos como en +-g
            pos = rnd.randint(0, 23)
            valor = sActual[pos]
            if(rnd.random() > 0.5):    #+-g de forma aleatoria y vecino con pos aleatoria
                sVecina= genVecino(sActual,pos,g)
                bVecina = fEvaluacion(sVecina)
                llamadasCoste += 1
            else:
                sVecina= genVecino(sActual,pos,-g)
                bVecina = fEvaluacion(sVecina)
                llamadasCoste += 1
            # Si sVecina no está en la lista tabú y supera a la mejor
            if ((pos, sVecina[pos]) not in lTabu) and (bVecina > bMejor):
                sMejor = sVecina
                bMejor = bVecina
        sActual = sMejor
        bActual = bMejor
        #Añadimos el par hora-valor a la lista tabú
        lTabu.append((hora, valor))
        if len(lTabu) > tamTabu: #Lista tabú pasa su tamaño máximo
            lTabu.pop(0) # Eliminamos el mas antiguo
        #Actualizamos matrizFr
        matrizFr = incMatrizFr(matrizFr, sActual, g)
        if i % (maxIteraciones // 4) == 0:
            # Reestablecer lt
            lTabu = []
            if rnd.random() > 0.5: #Aumentar o disminuir 50%
                tamTabu += tamTabu / 2
            else:
                tamTabu -= tamTabu / 2
            # Implementacion estrategia reinicialización
            eReinicio = rnd.random()
            if eReinicio < 0.25: # Solución aleatoria
                sActual = genSolucion()
                bActual = fEvaluacion(sActual)
                llamadasCoste += 1
            elif 0.25 < eReinicio <= 0.75: # Solución Greedy
                matrizProb = genMatrizProb(matrizFr)
                sActual = genSolucionGreedy(matrizProb, g)
                bActual = fEvaluacion(sActual)
                llamadasCoste += 1
            else: # Mejor solución
                sActual = sMejor
                bActual = bMejor
    return bMejor ,sMejor, llamadasCoste
```

Primero ponemos como mejor solución la pasada por parámetro, para que todos los algoritmos trabajen con el mismo array y poder comparar como recorre cada uno el espacio de búsqueda.

Para este algoritmo los parámetros variables los he pasado en la propia llamada de la función y no los he dejado como atributos globales, estos son:

maxIteraciones, es el número máximo de ejecuciones del bucle, lo he dejado en 100.

tamTabu, es el tamaño de la lista tabú que se dice en la practica que inicialmente tiene que ser.

numVecinos , es el numero de vecinos que se generara en cada iteración.

Inicializamos la matriz de frecuencia junto con la lista tabú, para cada iteración inicializamos hora y valor al valor nulo, para cada vecino que tenemos que crear (nVecinos) crearemos un vecino aleatorio en posición y en +-g, al igual que en el enfriamiento simulado, solo que aquí tenemos que guardar la posición aleatoria, ya que ahora tenemos que usarla ya que si el vecino generado no esta en la lista tabú y tiene mejor beneficio, va a pasar a ser la mejor solución (sMejor), la solución mejor va a ser la actual ahora.

Añadimos el par hora valor correspondiente a la lista tabú, y se comprueba que quepa, en caso de que no lo haga sacamos el elemento mas antiguo de la lista. Una vez hemos hecho esto debemos actualizar la matriz de frecuencias llamando a incMatrizFrecuencia().

Tras esto debemos tener en cuenta las 4 reinicializaciones que debemos hacer, en cada una de estas vamos a vaciar la lista tabú y aleatoriamente vamos a aumentar o dividir su tamaño en un 50% de su valor, y también calcularemos como se hará, para la cual tenemos que generar un numero aleatorio entre 0 y 1, si es menor que 0,25 se usara una solución aleatoria si es entre 0,25 y 0,75 lo hará mediante el uso de la memoria a largo plazo, generando la matriz de probabilidades con genMatrizProb() y pasándola como parámetro en la función genSolucionGreedy() para la generación de la solución greedy que se usara tras la reinicializacion; y del 0,75 al 1 hará la reinicializacion desde la mejor solución(sMejor).

Metodología de Comparación

Para las ejecuciones he ido usando diferentes semillas (10,11,15,20 y 25). Antes de rellenar la tabla voy a dejar cada uno de los resultados obtenidos por cada algoritmo en cada iteración para que se pueda comprobar que el contenido de la tabla corresponde con la experimentación realizada. He hecho esto para cada uno de los dos problemas en la práctica.

Para el problema 1 (Datos reales):

SEMILLA 10:

```
Solucion inicial generada a pasar a los algoritmos->  [46, -92, 9, 23, 47, -97,
-48, 18, 25, -29, 67, -59, -92, 33, 25, -17, -81, -37, 90, -8, -89, 7, -65, 54]
--------------------------------------------------
Solucion PRIMER vecino :  (347.94478829154286, [46, -87, 14, 28, 52, -97, -43,
18, 25, -29, 67, -54, -87, 33, 30, -12, -76, -32, 90, -8, -89, 12, -60, 59], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (455.5552262000001, [46, 3, 9, 23, 47, -87, -48, 18, 5,
-100, 2, -59, -92, 3, 5, -17, -81, -37, 5, -8, -89, 100, 0, 54], 5426, 112)
--------------------------------------------------
Solucion VND  (586.22447835, [46, 0, 0, 0, -100, -97, -48, 18, 10, -29, 2, -59,
-92, 3, 5, 55, -81, -37, 5, 2, -89, 100, 0, 54], 18866)
--------------------------------------------------
Solucion Busqueda Aleatoria  (474.4188812713859, [45, -17, -20, 25, 49, 60, -56,
20, -38, -4, 2, 18, 30, -73, 7, 54, 32, 86, -77, -37, -85, -73, 85, -84], 100)
--------------------------------------------------
Solucion Greedy  (338.63000000000005, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (355.823423312915, [46, -92, 9, 28, 47, -87,
-48, 13, 15, -29, 72, -64, -82, 28, 15, -17, -81, -32, 90, -3, -89, 12, -70,
49], 2015, 10.0)
--------------------------------------------------
Solucion Busqueda Tabu  (2185.5856577935283, [-5, 7, -10, 18, -11, 5,
-10, -15, 2, -20, 7, 1, 5, 5, 3, 2, 5, 5, 5, 5, 5, 15, 5, 23], 205)
```

SEMILLA 11:

```
Solucion inicial generada a pasar a los algoritmos->  [15, 43, 99, 19, 15, 30,
50, -52, -53, 31, 21, 61, 57, -53, -76, 14, -23, -64, -77, 37, 77, 62, -90, 52]
--------------------------------------------------
Solucion PRIMER vecino :  (323.83186906629254, [15, 43, 99, 19, 15, 30, 50, -52,
-53, 31, 21, 61, 57, -48, -71, 19, -18, -59, -72, 37, 82, 67, -85, 57], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (636.5689222225201, [15, 43, 99, 19, 15, 30, 50, -97,
-53, 1, 1, 1, 2, -53, -76, 29, -3, -64, -77, 2, 5, 100, 0, 52], 5138, 106)
--------------------------------------------------
Solucion VND  (661.0205326225201, [15, 43, 99, 19, 15, 30, 50, -94, -53, 1, 1,
1, 2, -53, -76, 2, 2, -64, -77, 2, 2, 100, 0, 52], 12434)
--------------------------------------------------
Solucion Busqueda Aleatoria  (481.3511604473371, [-2, 20, -57, 26, 38, 56, 52,
-86, 9, 26, 7, -28, 34, 2, 53, -21, -7, 35, -27, 23, 67, -31, 43, -28], 100)
--------------------------------------------------
Solucion Greedy  (338.63000000000005, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (368.9011082558542, [30, 53, 94, 29, 20, 15, 50,
-42, -38, 16, 11, 46, 52, -53, -51, 9, -23, -79, -77, 27, 62, 77, -95, 57],
2037, 20.0)
```

```
Solucion Busqueda Tabu  (2010.3665243316452, [0, 0, 9, 0, 0, 20, 0, 5,
20, 0, 0, -10, 2, 2, 4, 5, 5, 5, 5, 2, 1, 1, 5, 2], 204)
```

SEMILLA 15:

```
Solucion inicial generada a pasar a los algoritmos->  [-47, -98, 33, 88, -91,
-60, -39, -96, -86, 74, -63, 77, -6, -39, -71, -14, 19, 81, -9, -29, 0, -33,
-12, -42]
--------------------------------------------------
Solucion PRIMER vecino :  (349.85652870447706, [-42, -93, 38, 93, -91, -55, -34,
-91, -81, 74, -63, 77, -1, -34, -66, -9, 19, 81, -4, -24, 5, -28, -7, -37], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (404.66400220000014, [3, 2, 33, 88, -100, -60, -39,
-96, -86, 44, -63, 67, -1, -39, -71, -14, 4, 1, -9, -29, 5, -33, -12, -42],
2978, 61)
--------------------------------------------------
Solucion VND  (560.2243983328001, [0, 0, 0, 88, -100, -60, -39, -96, -86, 44,
-63, 67, 0, -39, -71, 1, 4, 1, 1, -29, 1, -33, 100, 0], 16562)
--------------------------------------------------
Solucion Busqueda Aleatoria  (512.5409516469294, [-48, -10, -20, -43, -22, 86,
30, 7, -41, 47, 16, 7, 25, -80, 18, 44, -8, 12, 45, -20, 77, 65, 15, 3], 100)
--------------------------------------------------
Solucion Greedy  (338.63000000000005, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (350.79425488616005, [-47, -80, 38, 100, -86,
-65, -49, -85, -76, 74, -63, 77, -6, -29, -56, -19, 19, 76, -9, -44, 10, -28,
-32, -42], 2031, 20.0)
```

```
Solucion Busqueda Tabu  (1263.2889569981317, [17, 3, 23, -20, -10, -25,
0, 14, 0, -5, 0, 2, 3, 5, 16, 5, 12, 4, 4, 5, 0, 20, 41, 5], 205)
```

SEMILLA 20

```
Solucion inicial generada a pasar a los algoritmos->  [85, 75, 96, -62, -34, 72,
62, -75, -17, 46, -57, -94, 5, 4, -81, -74, -68, -19, 21, 48, 15, 5, -47, -49]
--------------------------------------------------
Solucion PRIMER vecino :  (425.91163351435125, [85, 75, 96, -57, -29, 77, 62,
-75, -17, 46, -52, -89, 5, 4, -76, -69, -63, -14, 21, 48, 20, 10, -42, -44], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (686.0261336800002, [85, 75, 96, 3, 1, 72, 62, -95,
-17, 41, -57, -94, 5, 4, -81, -74, -68, -19, 1, 3, 5, 100, 3, 1], 3842, 79)
--------------------------------------------------
Solucion VND  (915.8157517620003, [85, 75, 96, 0, -100, -3, -3, -95, 3, 41, -57,
-94, 5, 4, -81, -74, -68, 1, 1, 3, 5, 100, 0, 0], 15938)
--------------------------------------------------
Solucion Busqueda Aleatoria  (540.3787784695774, [-42, -65, 83, 0, -73, 17, -36,
43, -48, -88, -19, 0, -82, 9, 13, 8, 6, 40, -69, 61, 88, -44, 45, 76], 100)
--------------------------------------------------
Solucion Greedy  (338.63000000000005, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (424.4663350821679, [85, 75, 96, -62, -34, 72,
62, -75, -17, 46, -57, -94, 5, 4, -81, -74, -68, -19, 21, 48, 15, 5, -47, -49],
2001, 10.0)
```

```
Solucion Busqueda Tabu  (2038.5676446511052, [2, 15, -5, -4, -5, -14, 1,
-20, 18, -19, 5, 5, 5, 5, 4, 4, 5, 5, 5, 2, 18, 5, 11, 5], 205)
```

SEMILLA 50

```
Solucion inicial generada a pasar a los algoritmos->  [-63, 17, 16, 97, -56, 80, 0,
87, -11, 10, 29, -72, 36, -69, -80, 88, 16, -33, -88, 68, 65, -48, -15, -42]
--------------------------------------------------
Solucion PRIMER vecino :  (340.8970136836208, [-58, 22, 21, 100, -51, 85, 5, 92, -6,
15, 29, -67, 36, -64, -75, 88, 16, -28, -83, 68, 70, -43, -10, -37], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (501.3664420000001, [2, 17, 16, 97, 4, 80, 0, 87, 39, 0, 4,
-72, 1, -69, -80, 63, 1, -33, -88, 3, 5, 100, 0, 3], 6674, 138)
--------------------------------------------------
Solucion VND  (501.36782600000015, [0, 17, 16, 97, 0, 80, 0, 87, 39, 0, 4, -72, 1,
-69, -80, 67, 1, -33, -88, 3, 5, 100, 0, 0], 22658)
--------------------------------------------------
Solucion Busqueda Aleatoria  (489.3616215823971, [54, -52, -17, 61, -23, -57, -19, 7,
16, -69, 15, 24, 3, 6, -58, -51, 21, -84, -12, -48, 90, 7, 24, 27], 100)
--------------------------------------------------
Solucion Greedy  (338.63000000000005, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (338.9349548617271, [-58, 17, 16, 97, -56, 80, 5, 82,
-11, 10, 29, -72, 36, -69, -80, 88, 16, -33, -88, 68, 70, -48, -15, -42], 2007, 20.0)
--------------------------------------------------
```

```
Solucion Busqueda Tabu  (1898.2719706923792, [-5, 16, 0, 24, 10, 0, 9,
5, 26, 18, 0, 0, 5, 5, 3, 3, 5, 5, 5, 5, 5, 5, 5, 15], 204)
```

ALGORITMO	Ev.Medias	Ev.Mejor	Ev.Desv	Mejor €	Media €	Desv €
GREEDY	1	1	0	338,63	338,63	0
ALEATORIA	100	100	0	540,38	499,46	26,90
BL MEJOR	4811	2978	1436,50	686,02	539,824	120,11
BL PRIMER	24	24	0,447	425,90	337,20	39,85
VND	17291	12434	3782,79	915,81	647,95	161,70
ENF.SIMULADO	2018	2001	15,40	424,50	373,60	31,89
TABÚ	204	204	0,55	2185,58	1878,8	359,13

-Observando los resultados obtenidos podemos ver que los algoritmos suelen quedarse con soluciones entre los 400-500, sin embargo, las llamadas a la función de coste varían mas dependiendo de los algoritmos.

-Los resultados del greedy no varían ni en llamadas a la función de evaluación ni en resultado, ya que se genera el array con la heurística greedy y se evalúa, por lo que no le afecta el cambio de semilla.

-En cuanto a la búsqueda aleatoria, no encuentra valores realmente malos teniendo en cuenta que son 100 soluciones aleatorias, por tanto, solo 100 llamadas y aun así encuentra valores mejores que otros algoritmos probabilísticos como el enfriamiento simulado.

-Analizando los algoritmos de búsqueda local entre sí, los resultados son los esperados, ya que realmente el mejor y VND es normal que encuentren mejor solución que el primero, ya que no salen tras el primer vecino que mejore, pero realizan muchas mas llamadas a la función de evaluación, comparando el mejor y el VND podemos ver que pasa lo mismo, el VND encuentra resultados mejores que el mejor pero en contraparte realiza muchas mas llamadas a la función de evaluación ya que explora mucho mas en profundidad el espacio de búsqueda, aun así podemos ver que el coste computacional de mejorar una solución es exponencial, ya que entre estos algoritmos la diferencia de valor medio es de mas o menos 100 mientras que la diferencia de llamadas es de más o menos 12.000.

-El algoritmo de enfriamiento simulado obtiene unos resultados que en comparación de los demás son un poco bajos, es el tercero con peores resultados, pero comparado con los algoritmos de búsqueda local, es capaz de encontrar resultados razonables con una pequeña cantidad de iteraciones.

-Para el algoritmo tabú los resultados de beneficio se disparan al igual que se disminuyen en gran parte la cantidad de llamadas a la función de coste, lo único que podemos decir es que es el algoritmo con mayor desviación en sus resultados, aun así viendo los resultados obtenidos podemos afirmar que el algoritmo de búsqueda tabú es bastante más eficaz que los demás, por lo menos para este problema en cuestión, ya que no hay ningún algoritmo que sea el mejor para todo.

Para el problema 2 (Aleatorizado):

Semilla 10

```
Solucion inicial generada a pasar a los algoritmos->  [46, -92, 9, 23, 47, -97, -48,
18, 25, -29, 67, -59, -92, 33, 25, -17, -81, -37, 90, -8, -89, 7, -65, 54]
--------------------------------------------------
Solucion PRIMER vecino :  (442.2490601265307, [46, -92, 9, 23, 47, -97, -48, 23, 30,
-24, 72, -54, -87, 33, 30, -12, -76, -32, 95, -3, -84, 12, -60, 59], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (778.1615742199999, [-4, -100, 14, 3, 47, -97, -48, 100,
-45, -29, 100, -100, -92, 3, 100, 3, -51, -37, 5, 77, -89, 100, -100, 54], 7490, 155
--------------------------------------------------
Solucion VND  (780.89893992, [-4, -100, 9, 3, 50, -97, -48, 100, -43, -29, 100, -100,
-92, 3, 100, 0, -45, 0, 1, 79, -89, 100, -100, 55], 27362)
--------------------------------------------------
Solucion Busqueda Aleatoria  (596.9251783469643, [25, 72, 18, 2, -66, 7, 38, 49, -11,
37, -1, 25, 79, -58, 43, 12, -75, 6, -91, -100, 12, 37, -84, -88], 100)
--------------------------------------------------
Solucion Greedy  (482.8579999999999, [0, 0, 0, 0, 0, 0, 100, 100, 100, 100, 100, 100,
100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (429.0157881265307, [46, -92, 9, 23, 47, -97, -48,
18, 25, -29, 67, -59, -92, 33, 25, -17, -81, -37, 90, -8, -89, 7, -65, 54], 2001,
10.0)
```

```
Solucion Busqueda Tabu  (1736.3177803321823, [-10, -10, -15, 5, 5, 5, 5,
8, 5, 3, 2, 3, 9, 5, 28, 6, 4, 14, 5, 24, 5, 41, 12, 5], 204)
```

Semilla 11

```
Solucion inicial generada a pasar a los algoritmos->  [15, 43, 99, 19, 15, 30, 50,
-52, -53, 31, 21, 61, 57, -53, -76, 14, -23, -64, -77, 37, 77, 62, -90, 52]
--------------------------------------------------
Solucion PRIMER vecino :  (371.1033577788726, [15, 43, 99, 19, 15, 30, 55, -47, -48,
31, 26, 66, 62, -48, -71, 19, -18, -59, -72, 37, 77, 67, -85, 57], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (1364.7822055039999, [0, -100, 4, 4, 5, 5, 100, 100, -100,
1, 1, 1, 100, -100, -76, 4, -23, -64, -77, 2, 2, 100, -100, 52], 9506, 197)
--------------------------------------------------
Solucion VND  (1229.318340064, [0, -100, 67, -1, 1, 1, 100, 100, -100, 1, 1, 1, 100,
-100, -76, 4, 2, -64, -77, 2, 2, 100, -100, 52], 21362)
--------------------------------------------------
Solucion Busqueda Aleatoria  (634.1537838781676, [84, -82, 81, 58, -89, 5, 93, 70,
-11, -100, -4, -78, 18, 38, 63, -94, 34, -8, 81, -98, -72, 7, 7, -63], 100)
--------------------------------------------------
Solucion Greedy  (482.8579999999999, [0, 0, 0, 0, 0, 0, 100, 100, 100, 100, 100, 100,
100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (372.2785921185983, [30, 38, 100, 24, 10, 35, 45,
-57, -48, 31, 26, 66, 57, -53, -71, 19, -18, -79, -87, 37, 67, 77, -100, 47], 2005,
30.0)
```

```
Solucion Busqueda Tabu  (1856.773702614343, [0, -20, 21, 1, 5, 5, 1, 3,
3, 4, 5, 5, 11, 5, 19, 5, 5, 5, 5, 5, 5, 10, 2, 5], 203)
```

Semilla 15

```
Solucion inicial generada a pasar a los algoritmos->  [-47, -98, 33, 88, -91, -60,
-39, -96, -86, 74, -63, 77, -6, -39, -71, -14, 19, 81, -9, -29, 0, -33, -12, -42]
--------------------------------------------------
Solucion PRIMER vecino :  (306.6811772442437, [-47, -98, 38, 93, -91, -55, -34, -91,
-81, 74, -58, 82, -1, -34, -66, -9, 24, 86, -4, -24, 5, -28, -7, -37], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (658.17947593856, [-47, -100, 100, -2, -91, -60, -39, -96,
-86, 4, -63, 2, 100, -100, 39, -4, 4, 81, 1, -29, 5, -33, -12, -42], 6242, 129)
--------------------------------------------------
Solucion VND  (660.547395008, [-47, -100, 100, 0, -88, -60, -39, -96, -86, 4, -63, 2
100, -100, -71, 1, 4, 84, 0, -29, 1, -33, 3, -42], 12866)
--------------------------------------------------
Solucion Busqueda Aleatoria  (592.1641602642322, [9, -98, 67, 32, -49, -88, 24, 100,
1, -30, 0, 6, -73, -78, 47, -2, 83, 64, 61, 58, -64, -11, 41, -42], 100)
--------------------------------------------------
Solucion Greedy  (482.8579999999999, [0, 0, 0, 0, 0, 0, 100, 100, 100, 100, 100, 100
100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (260.08948444578505, [-47, -98, 33, 88, -91, -60,
-39, -96, -86, 74, -63, 77, -6, -39, -71, -14, 19, 81, -9, -29, 0, -33, -12, -42],
20001, 30.0)
--------------------------------------------------
Solucion Busqueda Tabu  (1325.1905213819355, [-5, 16, -13, 4, 3, 3, 4,
6, 5, 3, 32, 5, 5, 11, 29, 0, 10, 5, 3, 5, 5, 15, 5, 10], 204)
```

Semilla 20

```
Solucion inicial generada a pasar a los algoritmos->  [85, 75, 96, -62, -34, 72, 62,
-75, -17, 46, -57, -94, 5, 4, -81, -74, -68, -19, 21, 48, 15, 5, -47, -49]
--------------------------------------------------
Solucion PRIMER vecino :  (388.85037726269525, [85, 75, 96, -57, -29, 72, 67, -70,
-12, 46, -52, -89, 10, 4, -76, -69, -63, -14, 21, 48, 15, 10, -42, -44], 25)
--------------------------------------------------
Solucion MEJOR vecino :  (989.87939714, [65, -100, 96, -62, -34, 2, 100, 100, -100,
1, -57, -94, 5, 4, -81, -74, -68, -19, 1, 3, 5, 100, -100, -49], 8066, 167)
--------------------------------------------------
Solucion VND  (1002.3603761167999, [65, -100, 96, -59, -34, 2, 100, 100, -100, 1,
-57, -94, 5, 4, -81, -74, -68, 1, 1, 3, 5, 100, -100, -49], 20354)
--------------------------------------------------
Solucion Busqueda Aleatoria  (589.4776033097601, [-79, -95, 30, -22, 7, -29, 17, 59,
-29, -79, 64, 9, 80, -82, -94, 0, 94, -96, -34, 88, -97, 99, 74, 34], 100)
--------------------------------------------------
Solucion Greedy  (482.8579999999999, [0, 0, 0, 0, 0, 0, 100, 100, 100, 100, 100, 100
100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100], 1)
--------------------------------------------------
Solucion Enfriamiento simulado  (384.91649411631835, [85, 75, 96, -62, -44, 67, 67,
-85, -32, 51, -57, -94, 10, 4, -91, -69, -63, -14, 21, 48, 10, 10, -52, -49], 2009,
20.0)
--------------------------------------------------
Solucion Busqueda Tabu  (999.4989864129574, [14, 12, 10, 5, 5, 0, 2, 5,
5, 2, 0, 7, 19, 5, 1, 5, 7, 18, 23, 12, 5, 9, 12, 0], 204)
```

Semilla 50

```
Solucion inicial generada a pasar a los algoritmos->  [-63, 17, 16, 97, -56, 80, 0,
87, -11, 10, 29, -72, 36, -69, -80, 88, 16, -33, -88, 68, 65, -48, -15, -42]
-------------------------------------------------
Solucion PRIMER vecino :  (298.12827146102944, [-63, 17, 21, 100, -51, 80, 5, 92, -6
15, 34, -72, 41, -64, -75, 93, 16, -28, -83, 68, 65, -43, -10, -37], 25)
-------------------------------------------------
Solucion MEJOR vecino :  (905.3198694, [-100, 2, 100, 2, -91, 0, 5, 100, -46, 0, 100
-100, 100, -100, -80, 3, 1, -33, -88, 3, 5, -48, -15, -42], 8354, 173)
-------------------------------------------------
Solucion VND  (971.1896754, [-100, 2, 82, -2, -56, 1, 1, 100, -100, 1, 100, -100,
100, -100, -80, 3, 1, -33, -88, 3, 5, -48, 5, -42], 16658)
-------------------------------------------------
Solucion Busqueda Aleatoria  (582.5940026592002, [-64, -16, 52, -30, -11, -5, 9, -70
48, -54, 89, -68, 81, -70, -80, 19, -61, -49, -57, -85, 43, 82, 94, 40], 100)
-------------------------------------------------
Solucion Greedy  (482.8579999999999, [0, 0, 0, 0, 0, 0, 100, 100, 100, 100, 100, 100
100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100], 1)
-------------------------------------------------
Solucion Enfriamiento simulado  (304.00060829793915, [-58, 7, 16, 95, -61, 70, -5,
92, -31, 20, 24, -67, 36, -79, -85, 83, 21, -58, -68, 73, 70, -43, -15, -27], 2013,
20.0)
Solucion Busqueda Tabu  (564.3223719908084, [-75, -26, 14, 1, 49, -100,
100, 100, 26, -56, 29, 99, 13, 64, 86, -70, -77, 81, 14, 80, 46, 5, -75,
35], 205)
```

ALGORITMO	Ev.Medias	Ev.Mejor	Ev.Desv	Mejor €	Media €	Desv €
GREEDY	1	1	0	482,85	482,85	0
ALEATORIA	100	100	0	634	598,61	20,44
BL MEJOR	7932,4	6246	1194,93	1364	938,83	268,87
BL PRIMER	24	24	0	442	359,1	59,97
VND	19720,4	12866	5427,97	1229	932,42	218,9
ENF.SIMULADO	2005,8	2001	5,21	430,2	363,44	69,19
TABÚ	204	203	0,71	1856,77	1296,13	531,97

Los algoritmos se comportan de forma muy similar para unos valores distintos del problema, el algoritmo greedy encuentra una solución mejor, y también genera un array distinto ya que la hora donde está el precio máximo de venta varia con respecto al problema anterior.

La búsqueda aleatoria sigue encontrando buenos resultados en comparación con el enfriamiento simulado que es el único que ha empeorado significativamente al cambiar los valores del problema, ya que ahora esta casi a la par en beneficio con el primer mejor y itera bastante veces más que este.

La diferencia entre el beneficio medio del mejor y del VND se ha eliminado y de hecho ahora el mejor encuentra soluciones algo mas optimas que el VND, y sigue iterando muchísimas veces menos que este.

El algoritmo tabú sigue encontrando máximos muy grandes comparados con el resto de los algoritmos en una cantidad mucho menor de llamadas a la función de evaluación, por lo que sigue siendo el algoritmo mas optimo con diferencia con respecto a los demás.