METAHEURÍSTICAS

Informe Práctica 1

Algoritmos implementados

- Búsqueda Local del Mejor
- Búsqueda Tabú
- VNS

Grupo 5

Pablo Morillas Plaza - 26526616A
Salvador Perfectti Martin - 75943965N

Indice

Enlace a Drive con todos los logs

Detalles del problema

Se nos presentan una serie de funciones matemáticas acotadas entre cierto intervalo de números. La meta que se nos presenta es hallar el óptimo global de la función, en este caso **hallar el menor valor** que devuelvan dichas funciones f(x) en los mencionados intervalos.
Las funciones han sido extraídas de la siguiente librería de funciones:

https://www.sfu.ca/~ssurjano/optimization.html

Para la resolución del problema, nos hemos apoyado en algoritmos basados en la búsqueda local y en metaheurísticas basadas en trayectorias.
Antes de detallar los algoritmos usados, se van a detallar las restricciones y las consideraciones utilizadas para la creación de las metaheurísticas:

Los algoritmos usarán un vector de **d** dimensiones (en este caso 10) donde cada x_i representa un valor entre el intervalo a evaluar:
$\mathbf{(x_1, x_2, \dots, x_n), x_n \in [a, b]}$ (donde a y b representan el intervalo entre los que están acotados)

Para evaluar cada solución, usaremos el resultado que nos devuelve dicha función matemática y calcularemos la diferencia entre el valor a buscar, el óptimo y el valor de f(x):
$\mathbf{Valor = |\Omega - f(x_1, x_2, \dots, x_n)|}$ (donde Ω representa el óptimo global de dicha función)

Consideraciones

Para cada uno de los dos algoritmos implementados, hemos tenido en cuenta una serie de restricciones propias de cada problema:

- **BLK**: Cuando realizamos una comparación entre la solución actual y un vecino, usamos $\geqslant$ para evitar que el algoritmo se pueda quedar atrancado en un óptimo local.
- **MA**: En este algoritmo hemos implementado una lista circular con tenencia tabú *t* donde iremos guardando los vecinos cuyo margen de modificación de algún x_i se encuentre entre el 1% de dicho valor, de manera que el algoritmo aspire a escoger soluciones cuyo margen de mejora sea mayor.
  También se ha implementado una memoria a corto plazo la cual almacena las posiciones modificadas en cada iteración, intentado evitar así que siempre se modifiquen las mismas posiciones, permitiendo abrir el espacio de búsqueda aún más.
  Por último, se ha implementado una tabla a modo de memoria a largo plazo como registro histórico. Esta tabla tiene como filas cada x_i de la solución, y como columnas el rango entre el mínimo y el máximo de la función dividida en 10 intervalos. El objetivo de esta tabla es

permitir explorar las zonas menos visitadas o hacer una búsqueda más profunda en las más exploradas (diversificación e intensificación con oscilación estratégica).

Operadores comunes

Parametrización

Dentro del programa hay un **archivo de configuración** *config.txt* en el que se parametrizan operadores comunes a ambos algoritmos, he aquí una lista de estos operadores y de las estructuras de datos usadas para almacenarlos:

- **ArrayList<String>** archivos: Aquí se guardan los archivos con la información de las funciones matemáticas, además indica que objeto de tipo función crear. Los archivos de las funciones guardan el intervalo y el óptimo global.
- **ArrayList<String>** algoritmos: Aquí se guardan los algoritmos a usar, en nuestro caso BLK o MA.
- **ArrayList<long>** semillas: Array con semillas para la generación de los aleatorios. Las semillas usadas son {75943965, 76943595, 95943765, 95953764, 45935769}
- **Integer** dimensión: Este valor representa el tamaño del vector de la solución.
- **Double** mod: Es la probabilidad de cambiar un valor x_i de cada vecino con respecto al de la solución.
- **Integer** tenencia: Tenencia tabú y tamaño de la lista circular.
- **Double** probOE: Probabilidad para cambiar la estrategia a intensificación o diversificación.
- **Integer** porCientoEst: Porcentaje de iteraciones con las que el algoritmo considera que está en estado de estancamiento.

Métodos comunes

Los algoritmos tienen una serie de métodos comunes, pues el MA se basa en el BLK. Los métodos devuelven el mismo resultado pero el procedimiento puede variar un poco, el listado es este:

- **double[] ejecutar()**: ejecuta el algoritmo, devuelve la mejor solución encontrada.

La función de evaluación devuelve un valor binario en BLK mientras que MA no devuelve nada, el funcionamiento es similar, evalúan los vecinos con el criterio dicho anteriormente.

- **double[][] generar_vecinos()**: generan *k* vecinos, el algoritmo MA añade los vecinos no válidos en la lista explícita.

Pseudocódigo

Búsqueda local:

El algoritmo de búsqueda local que hemos utilizado para la resolución del problema ha sido el BLK, se han hecho pruebas generando únicamente 3 vecinos por cada nueva iteración (**BL3**) y generando aleatoriamente una nueva cantidad de vecinos acotada entre 4 y 10 (**BLK**).

```
algoritmo BLK:
        generar_vecinos()

        mientras (iteraciones_actuales < iteraciones_maximas) & (evalua(vecinos)):
                si(modo_blk): k := aleatorio entre 4 y 10

                generar_vecinos()
        final mientras

        return solucion
final algoritmo
```

```
funcion generar_vecinos():
        para i = 1 hasta k
                para j = 1 hasta dimension
                        si (aleatorio entre 0 y 1 < probabilidad_cambio):
                                vecino[i][j] := aleatorio entre minimo y maximo
                        si no:
                                vecino[i][j] := solucion_actual[j]
                        final si
                final para j
        final para i

        return vecino[][]
final funcion

funcion evalua(vecinos[][]):
        i := 0
        mientras i < k:
                si (abs(optimo - vecinos[i]) >= abs(optimo - solucion_actual)):
                        i := i + 1
                si no:
                        iteraciones_actuales := iteraciones_actuales + 1
                        solucion_actual := vecinos[i]
                        return true
                final si
        final mientras

        return false
final funcion
```

Funcionamiento

El algoritmo al iniciar genera un **k** aleatorio y forma un array de k vecinos basados en la semilla introducida. Al empezar las iteraciones se elige otro k aleatorio entre 4 y 10 (en el caso de BL3, k = 3 siempre) y se vuelven a generar k vecinos, así hasta que se cumplan el número de iteraciones o hasta que no haya mejora.

Al generar vecinos, para cada valor hasta **d** lanzamos un aleatorio y comprobamos si está por debajo o no de nuestra probabilidad de cambio (esto es, cada vecino tiene una probabilidad determinada de cambiar, que se introduce como parámetro). Si no se cumple esta condición el valor se mantiene igual pero si se cumple se escoge un valor aleatorio entre el mínimo y el máximo del intervalo de la función que estemos evaluando.

En cada iteración se evalúa el array de vecinos generado. Calculamos cuál de los dos está más cerca del óptimo de la función (ya que no todas las funciones tiene un óptimo global en 0) y si alguno de los k vecinos mejora la solución actual, reemplazamos la solución guardada por ese array y detenemos la comprobación devolviendo "verdadero". En caso de que ningún vecino generado mejore, devolvemos "falso" para parar la ejecución del algoritmo, siendo la última solución guardada la solución que devolvemos.

Búsqueda tabú:

El algoritmo de búsqueda tabú está basado en el BLK. La implementación del algoritmo se puede resumir en:

```
algoritmo MA:
        mientras iteraciones_actuales < iteraciones_maximas:
                generar_vecinos()
                actualizar_lista()
                evaluar(vecinos)

                iteraciones_actuales := iteraciones_actuales + 1

                actualizar_tiempos()

                k := aleatorio entre 4 y 10
        fin mientras

        return solucion_actual
final algoritmo
```

```
funcion generar_vecinos():
        para i = 1 hasta k:
                para j = 1 hasta dimension:
                        si (aleatorio entre 0 y 1 < probabilidad_cambio):
                                vecinos[i][j] := aleatorio entre minimo y maximo
                        si no:
                                vecinos[i][j] := solucion_actual[j]
                        fin si
                fin para
        fin para

        return vecinos[][]
final funcion
```

La función **evaluar()**, al ser extensa, irá incluida en la entrega de platea en un fichero .txt al igual que el resto del pseudocódigo.

```
funcion actualizar_lista(vecinos):
        para i = 1 hasta k:
                para j = 1 hasta dimension:
                        // GENERAR INTERVALO USANDO EL % PARAMETRIZADO
                        si intervalo_minimo >= vecinos[i][j] || vecinos[i][j] <= intervalo_maximo:
                                actualizar_lista_explicita(vecinos[i])

                                si i < k - 1:
                                        i := i + 1
                                        j := 0
                                si no:
                                        break
                                fin si
                        fin si
                fin para
        fin para
final funcion

funcion actualizar_lista_explicita(vecino):
        si lista_explicita.size != tenencia:
                lista_explicita.add(vecino)
                lista_tiempos.add(tenencia)
        fin si
final funcion
```

```
funcion actualizar_tabla():
        para i = 1 hasta dimension:
                para j = 1 hasta dimension:
                        si solucion_actual[i] >= intervalos[j] & solucion_actual[] <
intervalos[j + 1]:
                                memoria_largo_plazo[i][j] := memoria_largo_plazo[i][j] + 1
                        fin si
                fin para
        fin para
final funcion
```

```
funcion actualizar_lista_cambios(mejor_momento):
        si lista_cambios.size != tenencia:
                para i = 1 hasta dimension:
                        si solucion_actual[i] != mejor_momento[i]
                                lista_cambios.add(i)
                                lista_tiempos_cambios.add(tenencia)
                        fin si
                fin para
        fin si
final funcion
```

```
funcion es_tabu(vecino):
        si lista_explicita.contains(vecino):
                return true
        si no:
                para i = 1 hasta lista_cambios.size:
                        si vecinos[lista_cambios[i]] = solucion_actual[lista_cambios[i]]:
                                return false
                        fin si
                fin para
        fin si

        return true
final funcion
```

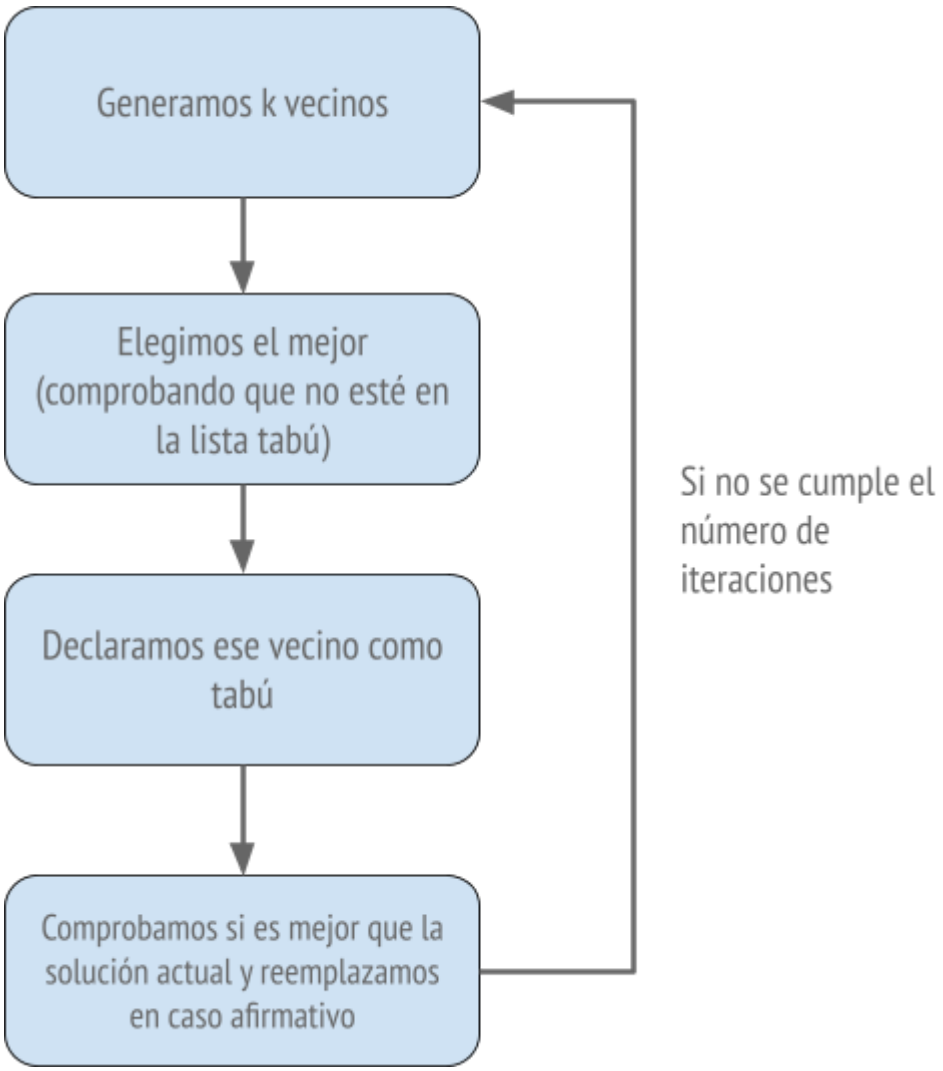
Funcionamiento

El algoritmo MA es parecido al BLK pero añadiendo varias estructuras para evitar caer en óptimos locales y explorar zonas de búsqueda menos visitadas o más prometedoras.

Memoria a corto plazo

La memoria a corto plazo mantiene una lista de soluciones que no se pueden volver a visitar. Estas soluciones se mantienen en una lista circular durante un tiempo determinado **t**, el cual llamamos **tenencia**. Esta técnica es buena para evitar caer en óptimos locales.

Esquema básico



Siguiendo este esquema, ahora después de generar vecinos llamamos a **actualizar_lista()**. En esta implementación haremos una pequeña modificación al esquema general: Comprobamos todos los valores de todos los vecinos generados y para cada vecino, si un valor x_i está dentro de un intervalo definido por el mismo x_i de la solución (en nuestro caso, si está dentro del ±1%), ese vecino se considera tabú y se actualiza la lista con **actualizar_lista_explicita()**.

Esta función actualiza dos listas: La lista tabú en sí misma añadiendo el vecino tabú, y la de tiempos añadiendo la tenencia para luego ir restando en cada iteración hasta que llegue a 0 y se saque de la lista explícita.

Además de estás funciones también tenemos **actualizar_lista_cambios()**. Lo que hará esta función es comprobar en el mejor vecino que valores han cambiado e introducir la posición que ha cambiado en una lista tabú de posiciones (también actualiza una lista de tiempos igual a la función explicada anteriormente). Esto obliga al algoritmo a cambiar posiciones que no han cambiado en iteraciones anteriores para aumentar la búsqueda.

Por último tendremos la función **es_tabu()** que comprueba si los vecinos generados están en estas dos listas para no considerarlos.

Memoria a largo plazo

La memoria a largo plazo almacena o bien una lista de vecinos "élite" para explorar zonas buenas visitadas anteriormente o bien una lista de frecuencias para explorar los valores menos visitados y extender el espacio de búsqueda. Estas listas se usarán cuando el algoritmo entre en estado de estancamiento, es decir, cuando durante un número determinado de iteraciones no haya mejora

En nuestro caso almacenaremos una tabla para cada valor de 1 hasta d y cada columna será un intervalo del rango entre el mínimo y el máximo de la función, dividido en 10 partes.
En cada casilla tendremos un contador de veces en las que cada valor x_i ha estado en el intervalo y_j.
Esta tabla se usará en el siguiente apartado implementando las dos técnicas explicadas anteriormente.

Oscilación estratégica

```
funcion generar_vecinos_oscilando(vecinos , intervalo):
        para i = 1 hasta k:
                para j = 1 hasta dimension:
                        vecinos[i][j] := aleatorio entre intervalo.minimo e intervalo.maximo
                fin para
        fin para
final funcion
```

```
funcion diversificacion(vecinos):
        para i = 1 hasta dimension:
                para j = 1 hasta dimension:
                        si memoria_largo_plazo < menor:
                                menor := memoria_largo_plazo[i][j]
                                intervalo[i] := j
                        fin si
                fin para
        fin para

        generar_vecinos_oscilando(vecinos , intervalo)
final funcion

funcion intensificacion(vecinos):
        para i = 1 hasta dimension:
                para j = 1 hasta dimension:
                        si memoria_largo_plazo > mayor:
                                mayor := memoria_largo_plazo[i][j]
                                intervalo[i] := j
                        fin si
                fin para
        fin para

        generar_vecinos_oscilando(vecinos , intervalo)
final funcion
```

Con la oscilación estratégica podemos aplicar las dos técnicas de memoria a largo plazo: **intensificación** y **diversificación**. Cuando el algoritmo haga un cierto número de iteraciones sin mejora (un 5% en esta práctica), consideraremos que se ha entrado en estado de estancamiento y con una probabilidad, en este caso del 50%, entrará o en intensificación o en diversificación.

Intensificación

Con intensificación recorremos la tabla buscando para cada variable el intervalo en el que **más veces** ha estado su valor y guardamos cada índice en un array. Una vez guardados todos, llamamos a una nueva función **generar_vecinos_oscilando()**, la cual genera nuevos vecinos pero con cada variable dentro del intervalo elegido anteriormente.
Con esta técnica visitamos zonas muy exploradas pero por consecuencia, prometedoras, evitando que el algoritmo quede durante muchas iteraciones en zonas donde no va a ser posible acercarse al óptimo global.

Diversificación

El funcionamiento es prácticamente el mismo que en intensificación, sólo que ahora escogemos los intervalos **menos visitados**, buscando por tanto en zonas poco visitadas para aumentar la zona de exploración.

VNS

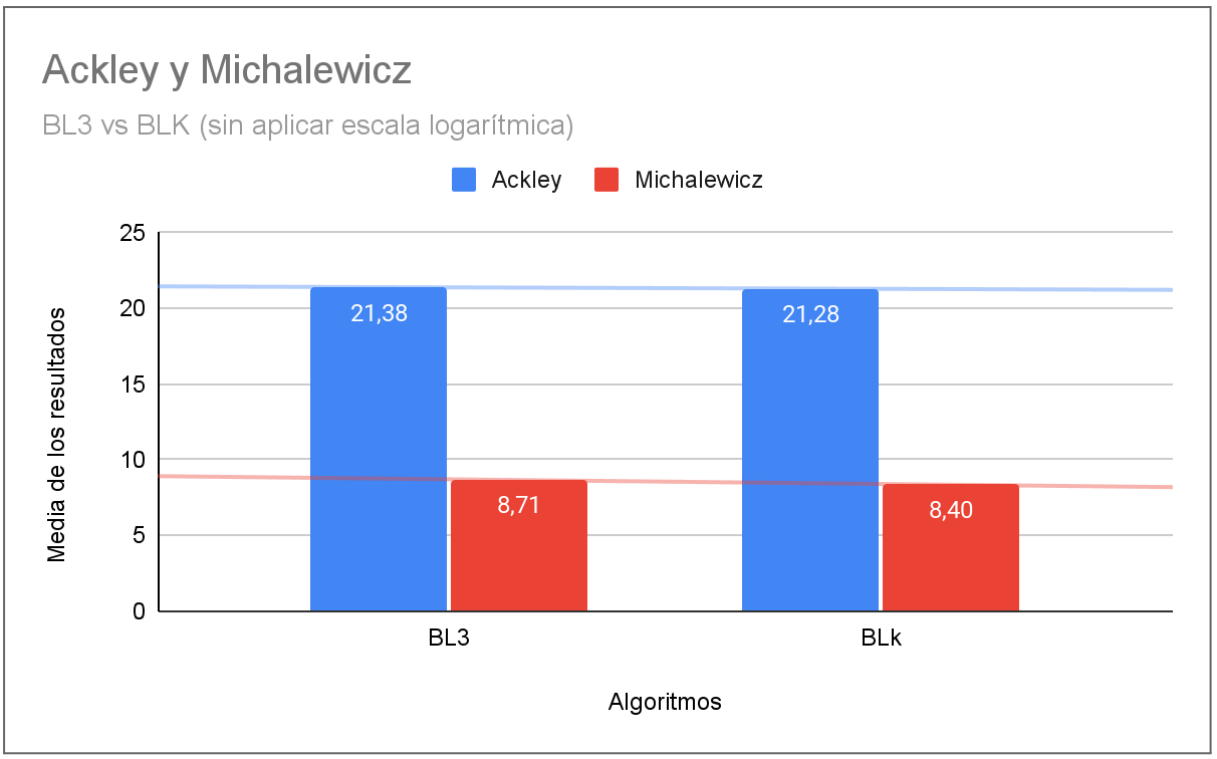
Por último añadimos un algoritmo de multiarranque, el VNS o Variable Neighborhood Search. Lo que se busca con VNS es aumentar aún más la exploración generando entornos completamente diferentes al actual cuando haya empeoramiento.
Hasta ahora, cuando en alguna iteración el algoritmo no mejoraba se elegía el mejor de los vecinos que se habían mejorado y se adoptaba como solución, es decir, empeorábamos la solución con el objetivo de no caer en óptimos locales. Ahora sustituiremos esta técnica por dos reinicializaciones del entorno: Aleatorio y cambio de signo.

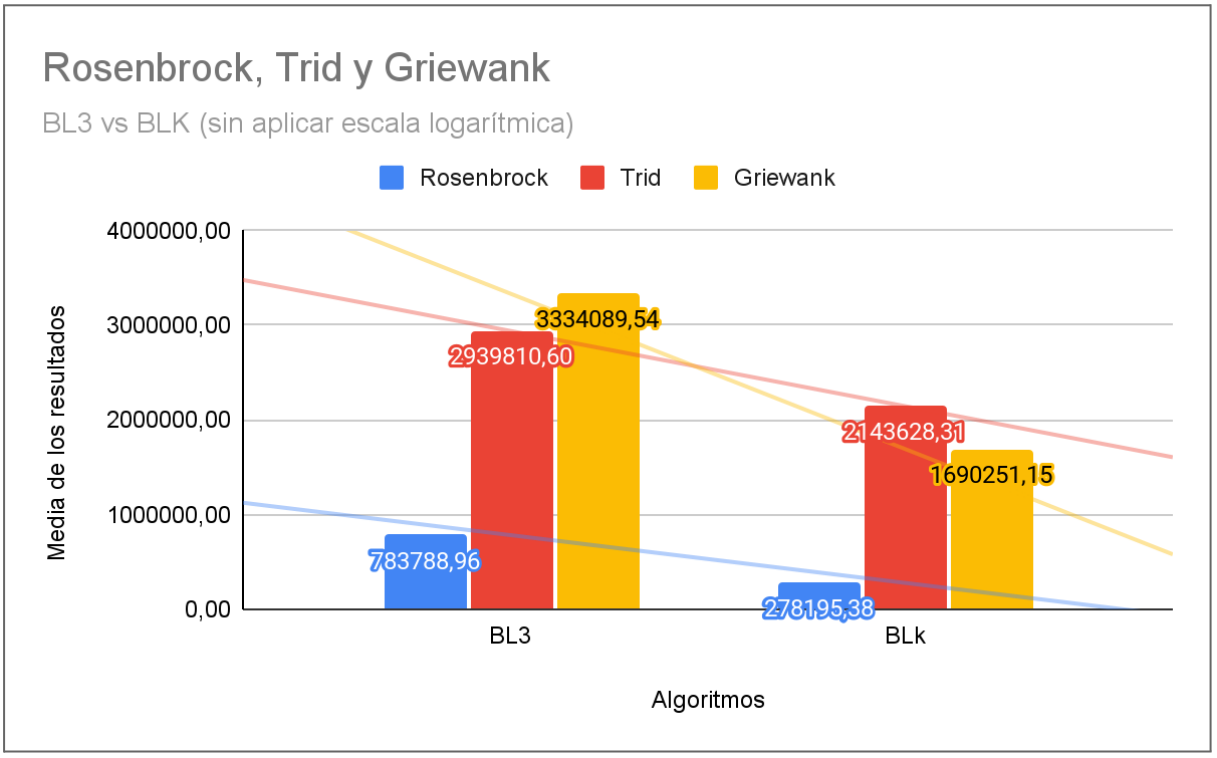
En la función evaluar, si hay un movimiento de empeoramiento tendremos un contador oscilando entre 1 y 2: Si está a 1 reinicializa el entorno aleatoriamente y si está en 2 a cambio de signo. Luego sólo hay que llamar a la función **generar_vecinos()** de la técnica elegida; Para la primera, elegimos para cada valor x_i de cada vecino un número completamente aleatorio entre el mínimo y el máximo de la función a evaluar y para el segundo, cambiamos de signo todas los valores x_i (en Michalewicz hacemos la inversa).

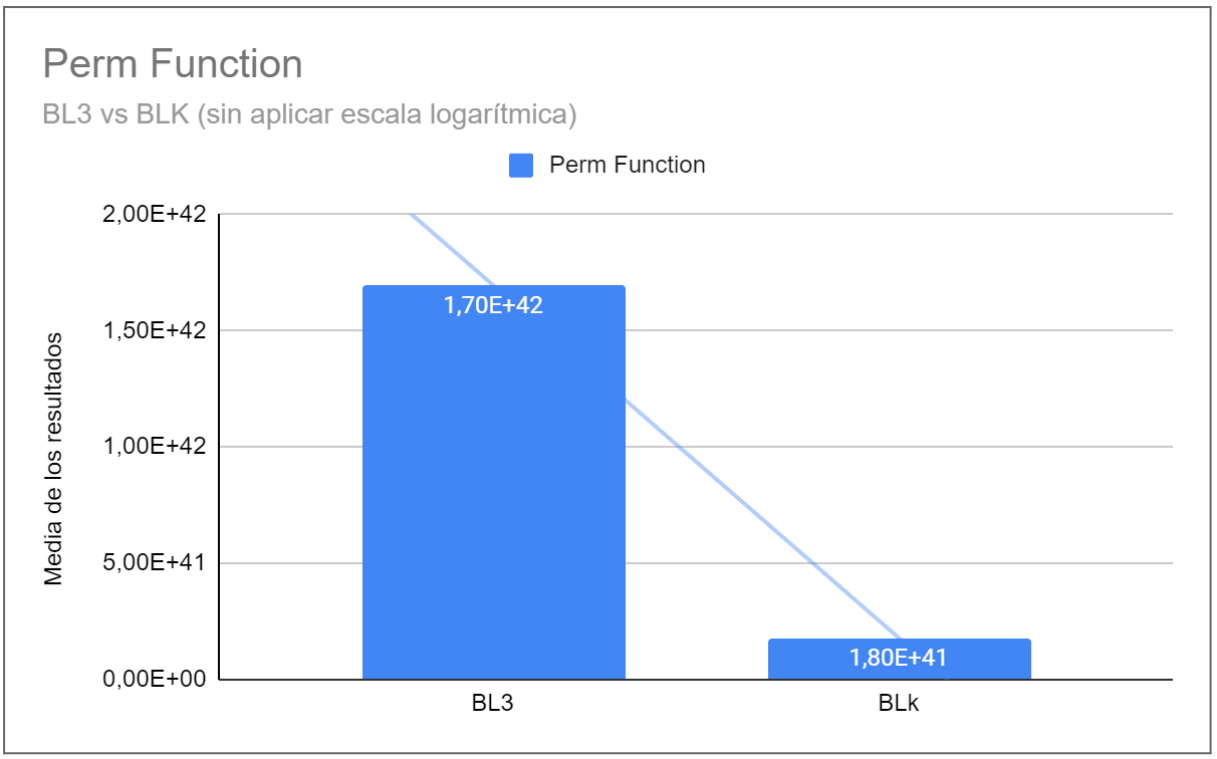
Análisis y experimentos

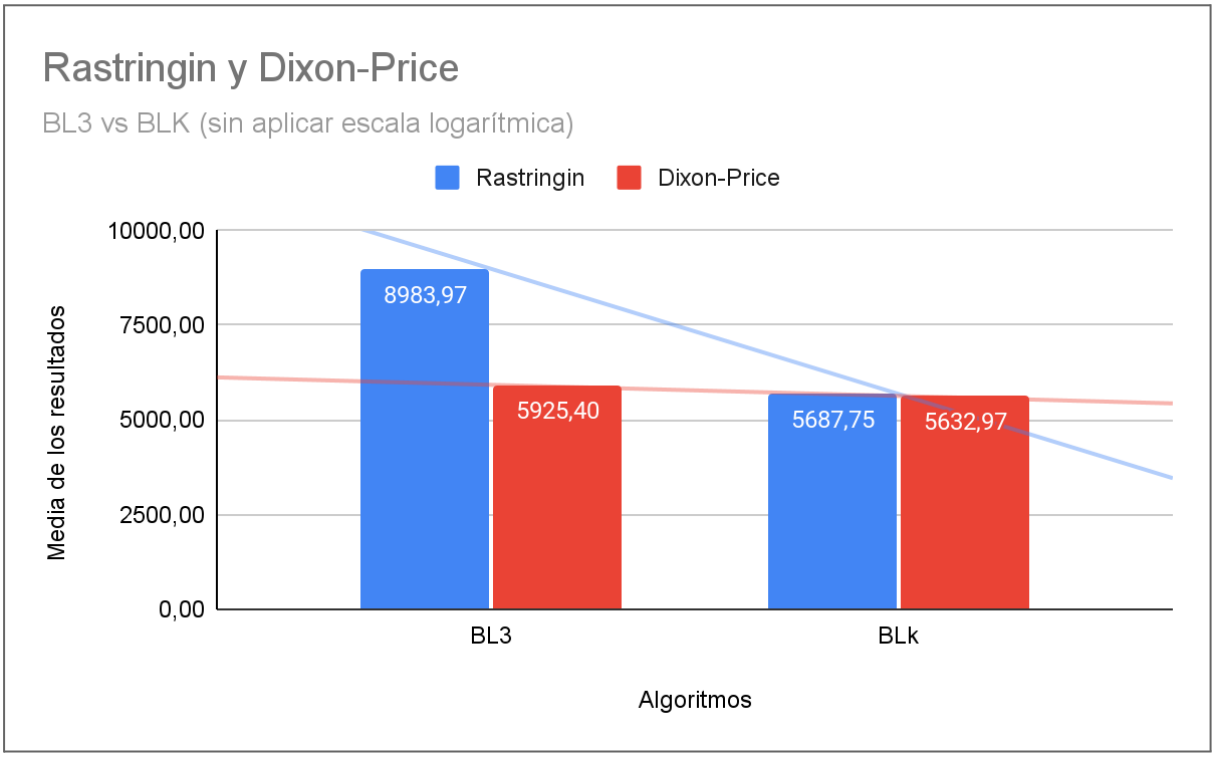
BL3 y BLK

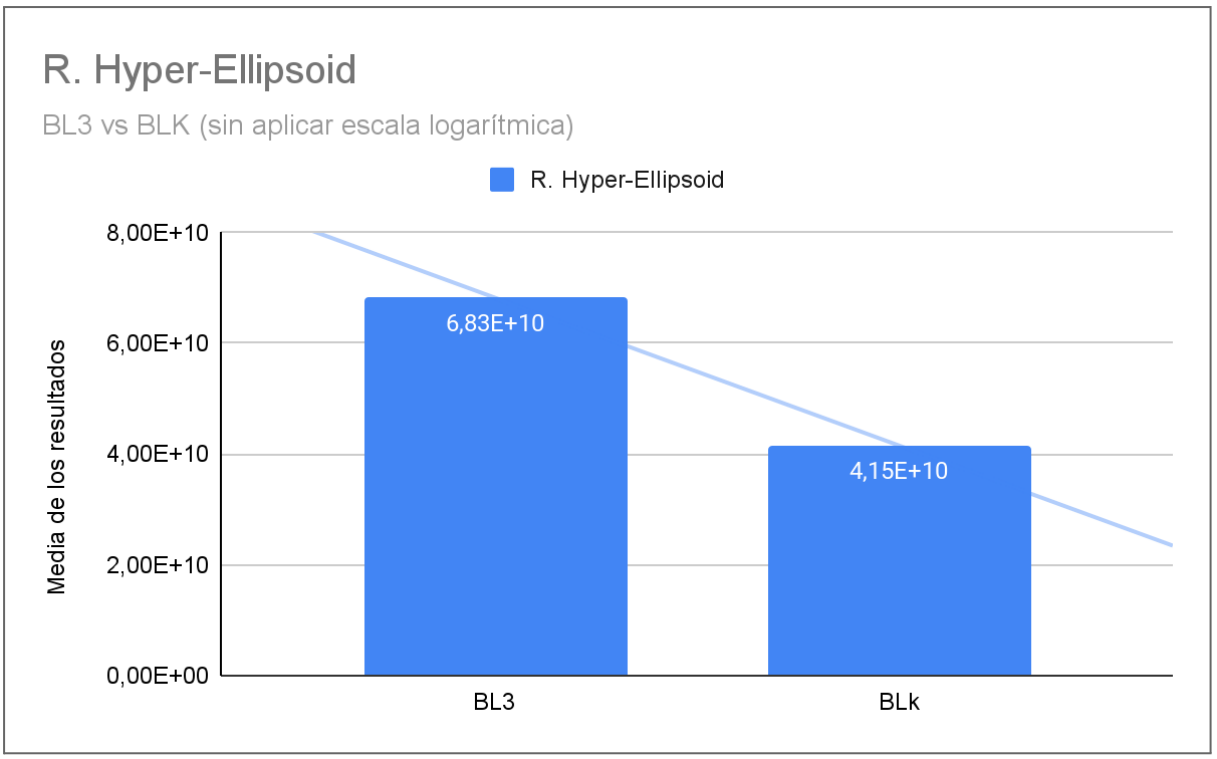
Dado que tenemos dos algoritmos con la misma naturaleza, el BL3 y el BLK; vamos a realizar una comparación entre ambos algoritmos para ver cual de estos nos ofrece mejores resultados.
Es muy difícil comparar todas las soluciones juntas ya que existen algunas diferencias notables entre los resultados de distintas funciones, hay algunas muy cercanas al óptimo global, como Ackley y otras que obtienen resultados muy grandes, como el hiperelipsoide; esto nos dificulta la creación de una comparativa mediante una gráfica que envuelva a todas las funciones; por ello, las dividiremos por grupos y no aplicaremos el escalado logarítmico pues la diferencia es aún demasiado grande, además de contar con una función con medias negativas:

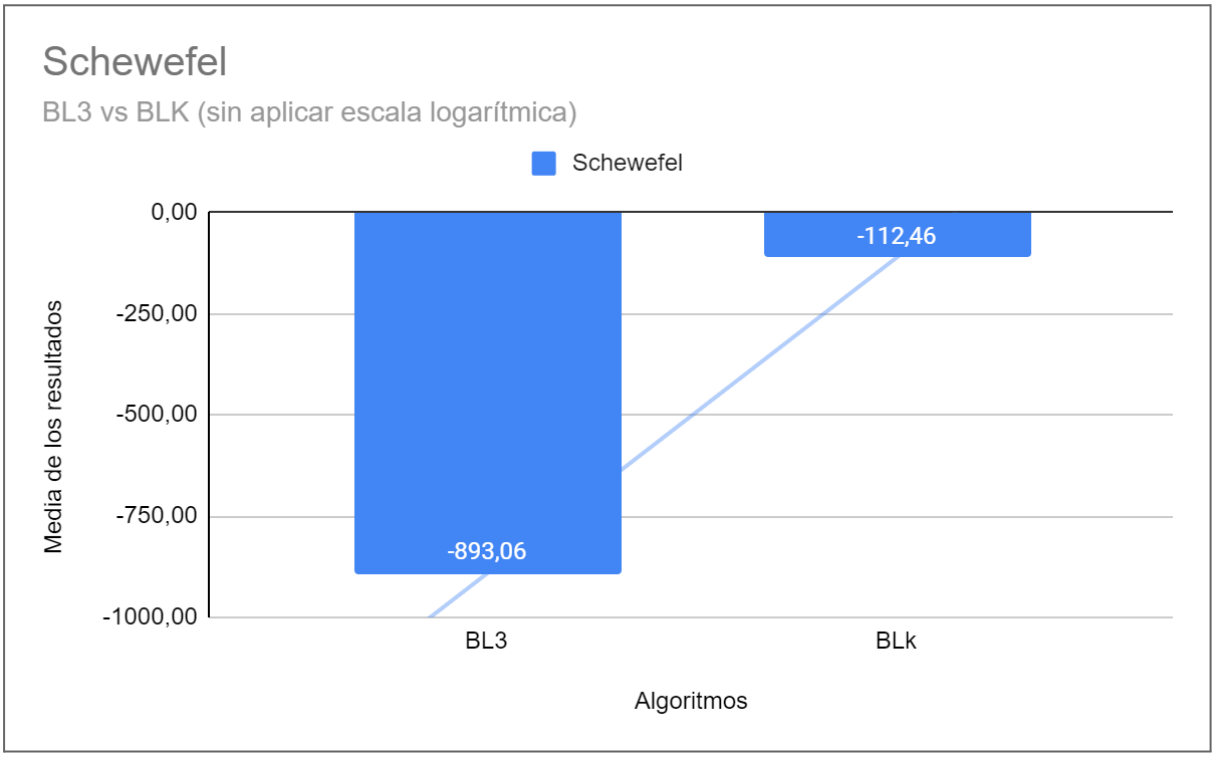

Ackley y Michalewicz
BL3 vs BLK (sin aplicar escala logarítmica)
Ackley
Michalewicz
Media de los resultados
25
20
15
10
5
0
21,38
8,71
21,28
8,40
BL3
BLk
Algoritmos


Rosenbrock, Trid y Griewank
BL3 vs BLK (sin aplicar escala logarítmica)
Rosenbrock
Trid
Griewank
Media de los resultados
4000000,00
3000000,00
2000000,00
1000000,00
0,00
3334089,54
2939810,60
2143628,31
1690251,15
783788,96
278195,38
BL3
BLk
Algoritmos

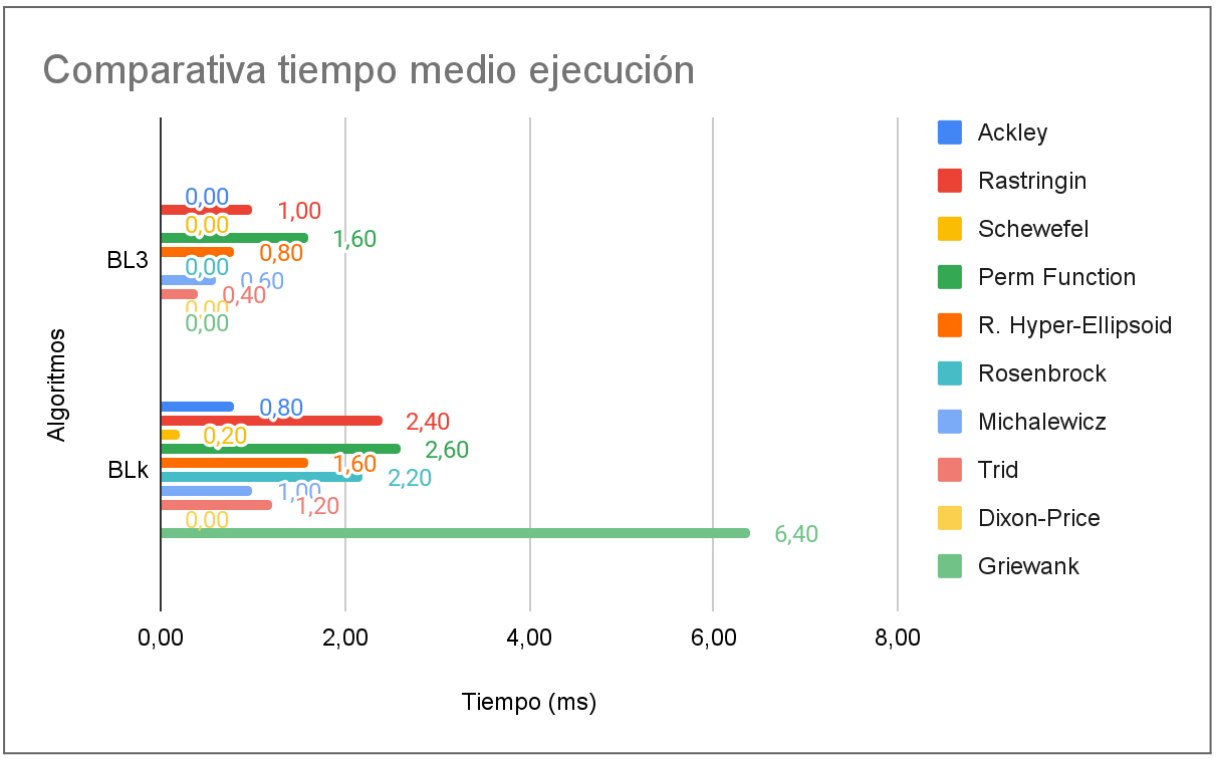

Perm Function
BL3 vs BLK (sin aplicar escala logarítmica)
Perm Function
2,00E+42
1,50E+42
1,00E+42
5,00E+41
0,00E+00
Media de los resultados
1,70E+42
1,80E+41
BL3
BLk


Rastringin y Dixon-Price
BL3 vs BLK (sin aplicar escala logarítmica)
Rastringin
Dixon-Price
10000,00
7500,00
5000,00
2500,00
0,00
Media de los resultados
8983,97
5925,40
5687,75
5632,97
BL3
BLk
Algoritmos





Por último, si echamos un vistazo a los tiempos de ejecución medios de cada algoritmo por cada función, vemos lo siguiente:

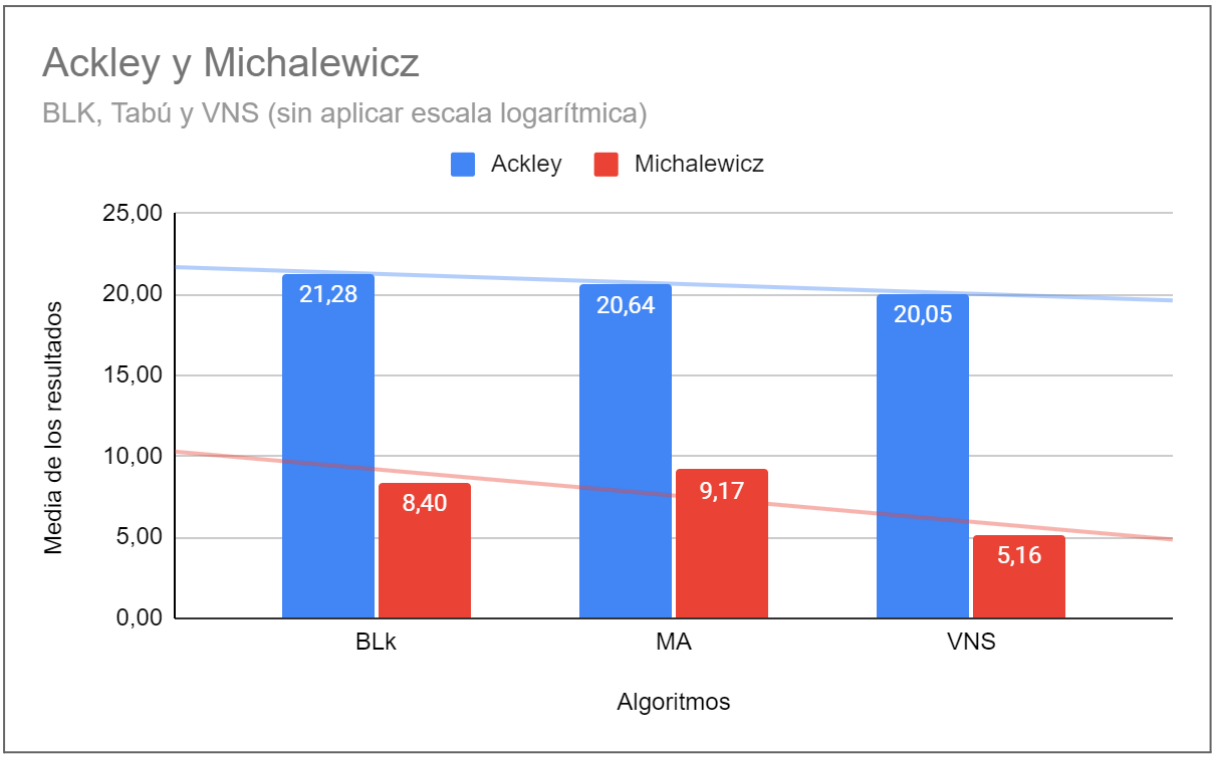


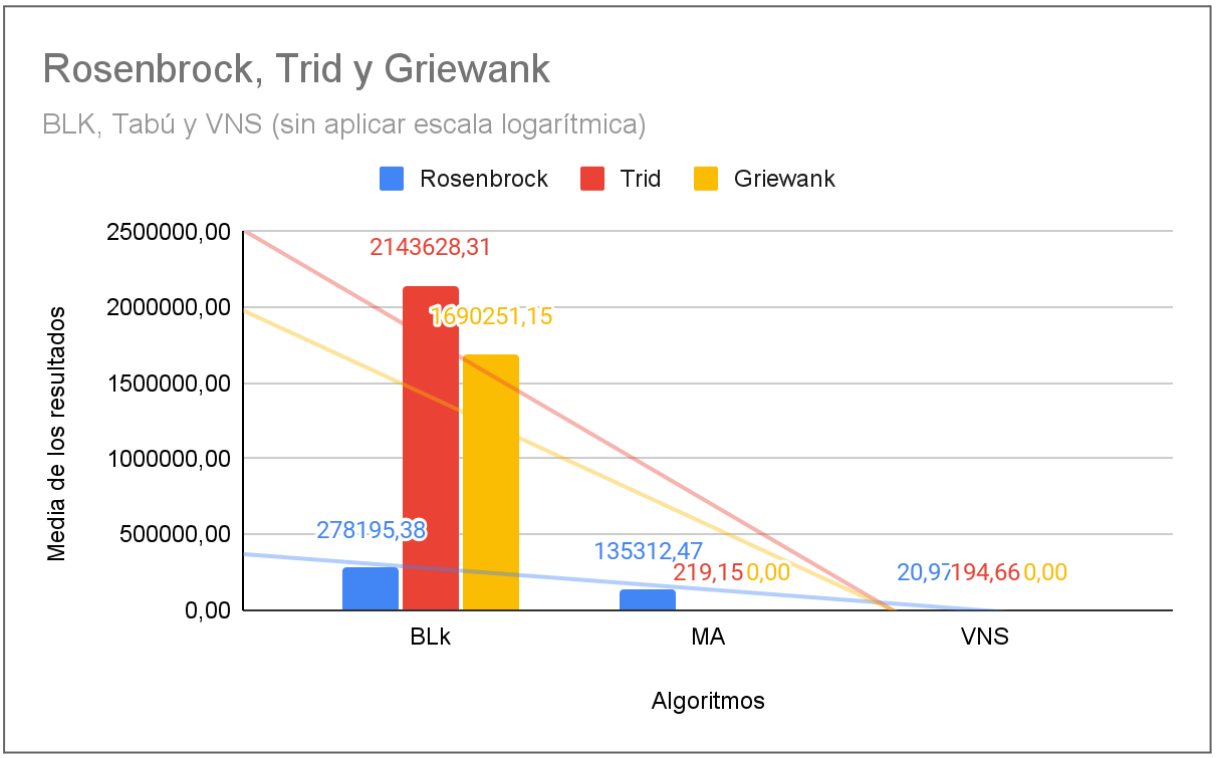
Podemos observar que en todas las medias de resultados, BLK es siempre mejor que BL3; esto se puede llegar a intuir debido a que se generan más vecinos por cada iteración, lo que permite una exploración más amplia. En cambio, los tiempos medios de ejecución son superiores en el BLK, pero esta diferencia es casi despreciable. Aunque en algunos casos en los que las funciones rondan el óptimo local, como en la función Ackley, los dos algoritmos se comportan prácticamente igual. Al ser un tiempo tan ínfimo concluimos que el algoritmo BLK es superior al BL3.

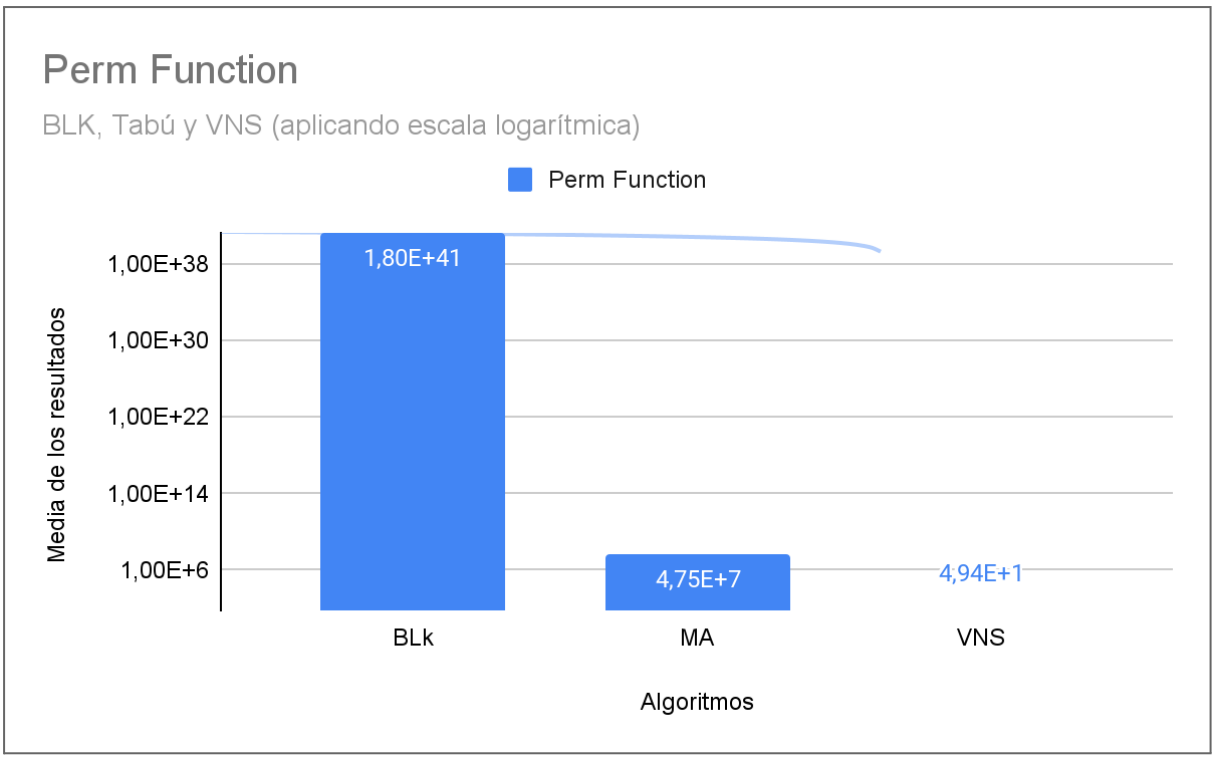
BLK, Tabú y VNS

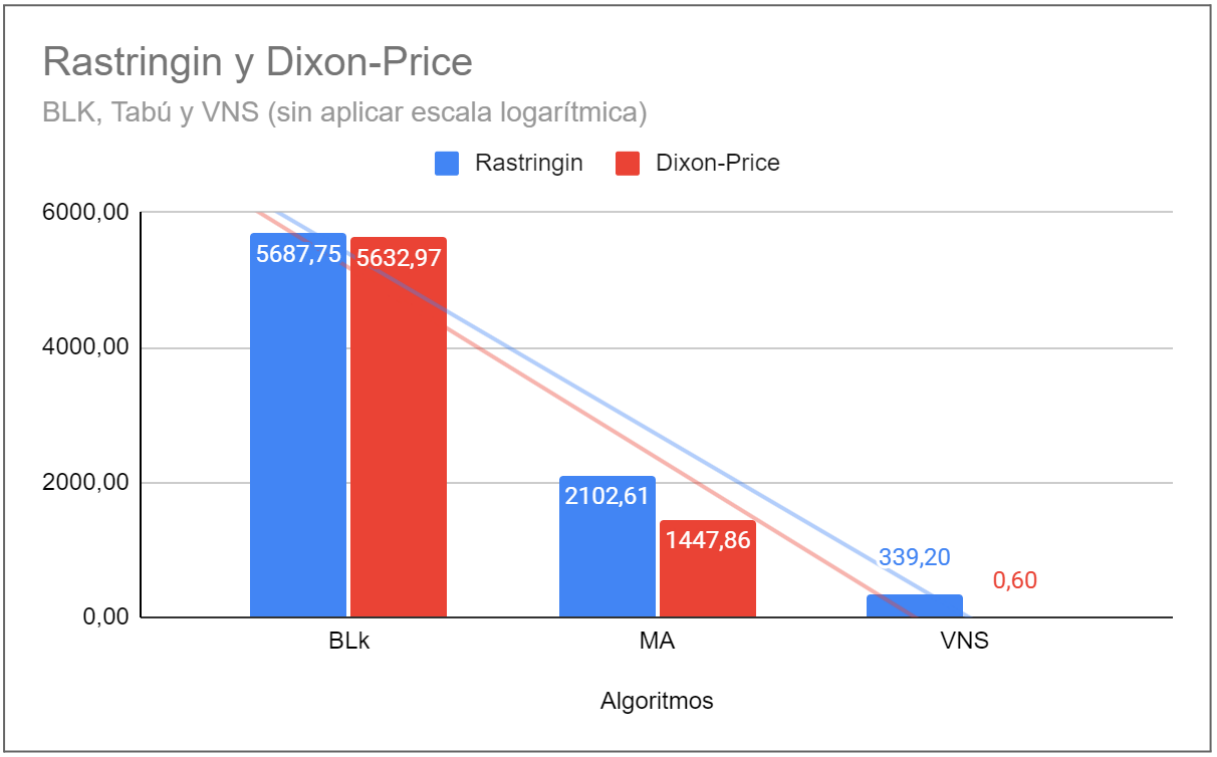
Realizaremos el mismo estudio, los 3 algoritmos que vamos a comparar generan los vecinos de una manera similar, los cambios que hay son en las restricciones a la hora de generar los propios valores de cada x_i o en la selección de la mejor solución en cada iteración.

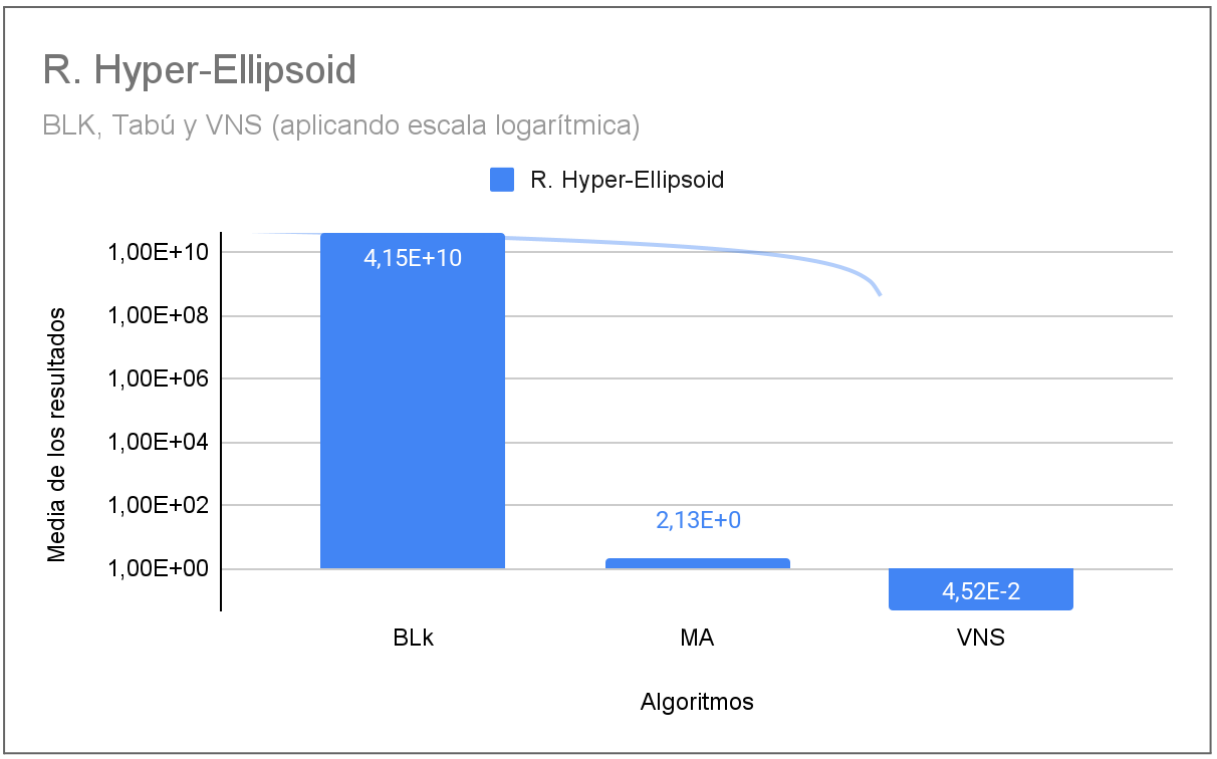
Vamos a ver los resultados medios para cada función aplicando cada algoritmo:

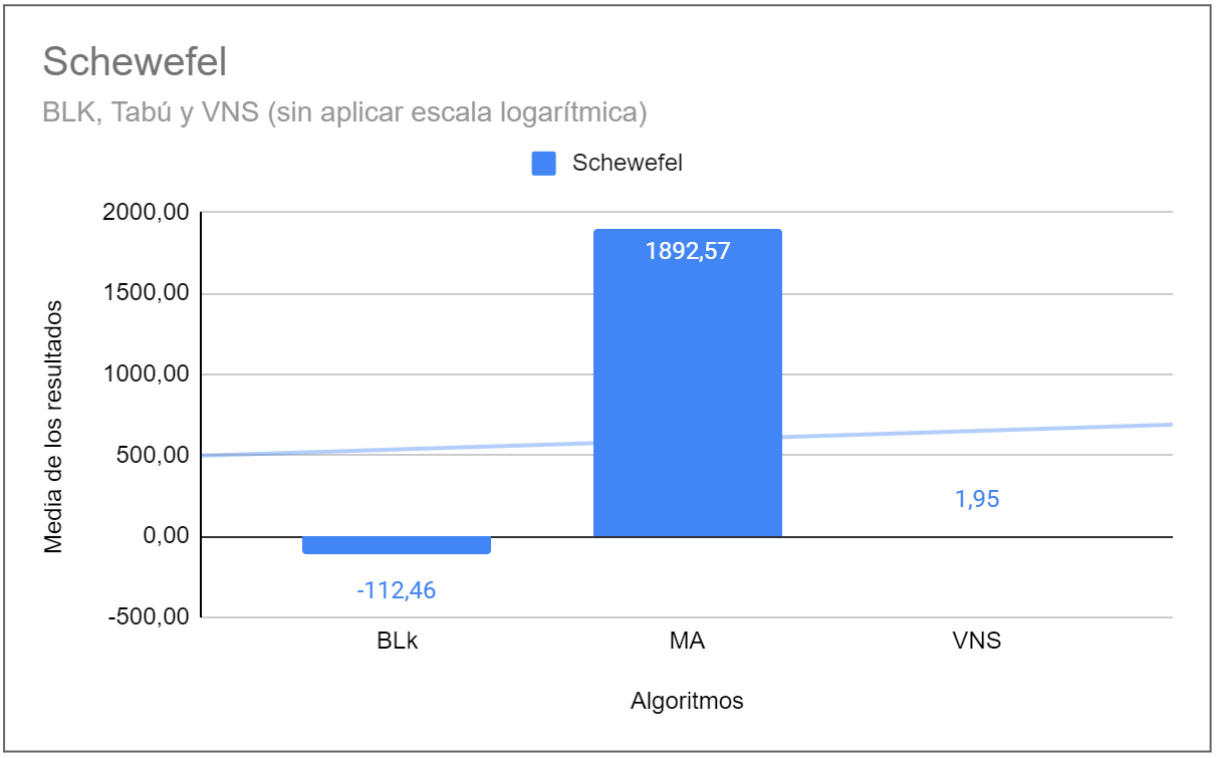

Ackley y Michalewicz
BLK, Tabú y VNS (sin aplicar escala logarítmica)
Ackley
Michalewicz
25,00
20,00
15,00
10,00
5,00
0,00
Media de los resultados
21,28
8,40
20,64
9,17
20,05
5,16
BLk
MA
VNS
Algoritmos


Rosenbrock, Trid y Griewank
BLK, Tabú y VNS (sin aplicar escala logarítmica)
Rosenbrock
Trid
Griewank
2500000,00
2000000,00
1500000,00
1000000,00
500000,00
0,00
Media de los resultados
2143628,31
1690251,15
278195,38
135312,47
219,15
0,00
20,97
194,66
0,00
BLk
MA
VNS
Algoritmos

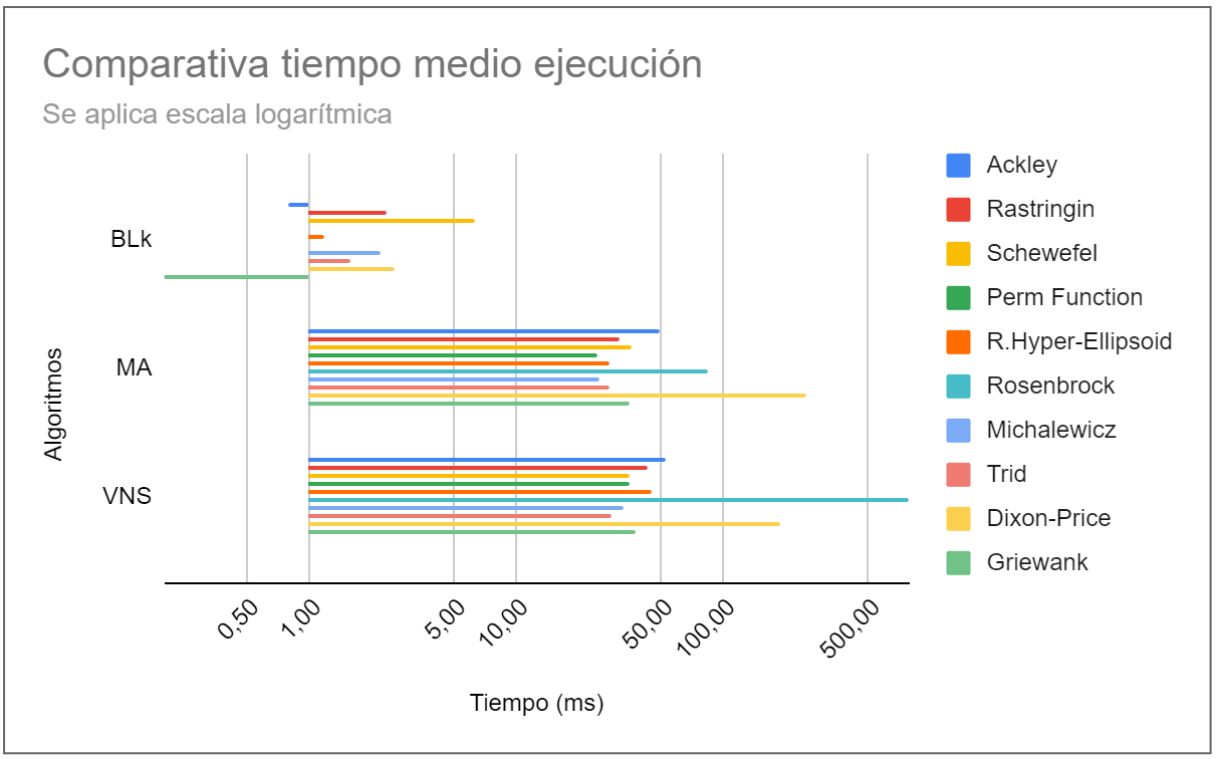

Perm Function
BLK, Tabú y VNS (aplicando escala logarítmica)
Perm Function
1,00E+38
1,00E+30
1,00E+22
1,00E+14
1,00E+6
Media de los resultados
1,80E+41
4,75E+7
4,94E+1
BLk
MA
VNS
Algoritmos


Rastringin y Dixon-Price
BLK, Tabú y VNS (sin aplicar escala logarítmica)
Rastringin
Dixon-Price
6000,00
4000,00
2000,00
0,00
5687,75
5632,97
2102,61
1447,86
339,20
0,60
BLk
MA
VNS
Algoritmos





Podemos hacer una comparativa de los tiempos de ejecución de los algoritmos, podemos observar lo siguiente:

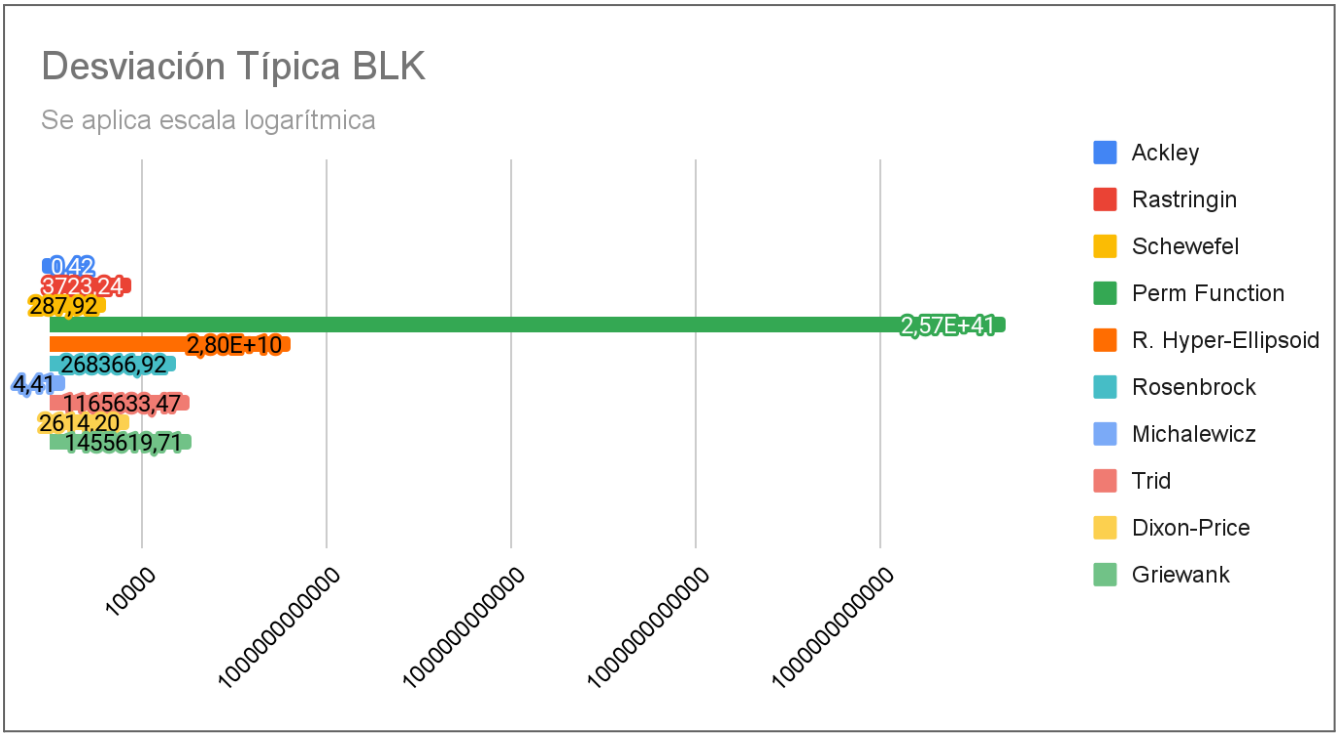


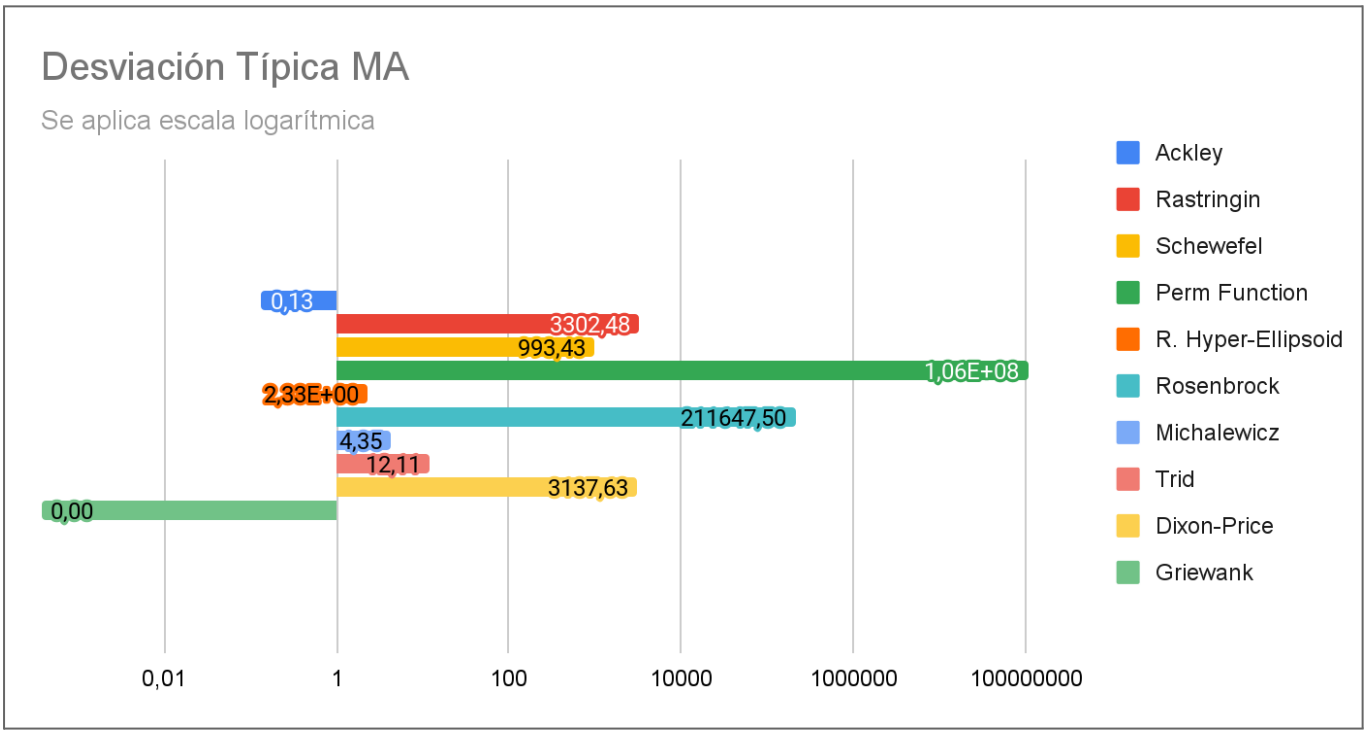
Podemos observar una similitud con el caso anterior, cada nuevo algoritmo mejora los resultados, a cambio el tiempo de ejecución medio aumenta.

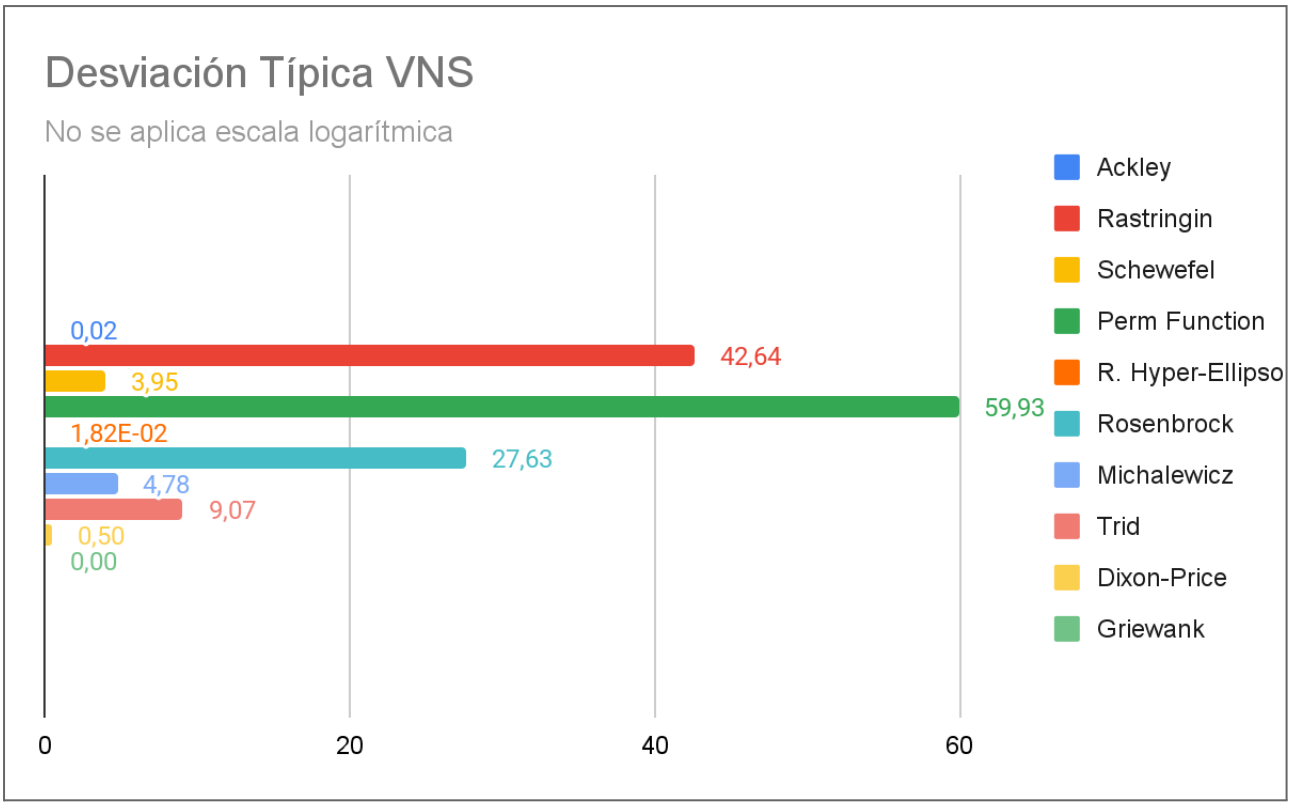
Si analizamos las tablas de las búsqueda tabú, podemos ver como los movimientos de empeoramiento (el aplicar diversificación) hacen que las medias se puedan ver un poco alteradas pero en líneas generales el algoritmo tabú es mejor que el BLK; a su vez, la estrategia VNS mejora significativamente a la búsqueda tabú, por lo que en conclusión merece más la pena usar el VNS ya que los resultados obtenidos son bastante mejores que con la búsqueda tabú.

Desviaciones típicas

Por último, vamos a hacer una comparación de las desviaciones típicas de las 3 técnicas para observar cómo de dispersos están los resultados obtenidos de la media:


Desviación Típica BLK
Se aplica escala logarítmica
0,42
3723,24
287,92
2,57E+41
2,80E+10
268366,92
4,41
1165633,47
2614,20
1455619,71
10000
100000000000
100000000000
100000000000
100000000000
Ackley
Rastringin
Schewefel
Perm Function
R. Hyper-Ellipsoid
Rosenbrock
Michalewicz
Trid
Dixon-Price
Griewank


Desviación Típica MA
Se aplica escala logarítmica
0,13
3302,48
993,43
1,06E+08
2,33E+00
211647,50
4,35
12,11
3137,63
0,00
0,01
1
100
10000
1000000
100000000
Ackley
Rastringin
Schewefel
Perm Function
R. Hyper-Ellipsoid
Rosenbrock
Michalewicz
Trid
Dixon-Price
Griewank



En líneas generales observamos que añadiendo búsqueda tabú y sobretodo VNS las desviaciones típicas van siendo menores en todos las funciones, por lo que vamos obteniendo resultados parecidos en todas las ejecuciones.

Análisis de la mejor solución y conclusiones finales

Después de comparar todas las técnicas y algoritmos podemos sacar algunas conclusiones:

- **BLK es superior a BL3:** Ambos son el mismo algoritmo, cambiando sólo un parámetro. Con BLK en cada iteración generamos más vecinos que con BL3 por lo que era de esperar que hubiera un poco de mejora en todas las funciones evaluadas. El empeoramiento del tiempo de ejecución es prácticamente inapreciable por lo que BLK es claramente mejor que BL3.
- **MA + VNS, la mejor solución:** Si añadimos búsqueda tabú a BLK, obtenemos una mejora significativa en casi todos las soluciones (habiendo algunas bastantes peores que con BLK) teniendo un incremento significativo en el tiempo de ejecución. Ahora bien, si añadimos también VNS y analizamos los resultados nos damos cuenta de que hay mejora en **todas** las

funciones sin prácticamente ninguna penalización en el tiempo. Por tanto, la mejor solución sería aplicar estas dos técnicas juntas.

- **Datos menos dispersos:** Conforme añadimos técnicas nos damos cuenta de que las desviaciones típicas disminuyen, habiendo gran diferencia entre MA y MA+VNS. Aparte de ser la mejor solución es la que menos dispersos están sus resultados, dando en cada ejecución soluciones no muy dispares entre sí y por lo tanto, más sólidas.