Informe de la práctica nº5

Computación evolutiva

UNIVERSIDAD DE BURGOS

Por: Adrián Zamora Sánchez

Índice

Introducción **3**
Elementos que se mantienen de la práctica anterior **4**
Cambios en el programa respecto al anterior **5**
Asignación de videos 5
Función de viabilidad 6
Función de distancia 7
Optimización multi-objetivo 8
Pruebas 10
Selección 10
Generaciones 14
Resultados obtenidos **19**
Conclusiones **20**

Introducción

En esta práctica se incluye una mejora a la solución propuesta anteriormente, donde se tratará de implementar una medida del buen uso de recursos, el recurso que se desea optimizar es la memoria/capacidad de los servidores caché, mientras se mantiene el objetivo principal de mejorar la latencia mediante la asignación de videos.

Para realizar esta mejora se modificará el algoritmo incluyendo un nuevo parámetro de evaluación y unas funciones de viabilidad (feasibility function) y distancia respecto de las soluciones válidas (distance function). Este enfoque permite generar a los individuos de formas incorrectas y posteriormente penalizar los casos inviables (que exceden la capacidad de algún caché) y corregir estos defectos durante la ejecución del programa.

Para la obtención de los resultados será especialmente interesante analizar cómo se adaptan los individuos al uso de memoria que utilizan, por lo tanto se incluirá en el resultado final el porcentaje de memoria que utilizan y una gráfica con la media de uso de memoria de la población a lo largo de las diferentes generaciones.

El código completo se puede encontrar adjuntado al informe y en el siguiente repositorio:
Enlace a GitHub

Elementos que se mantienen de la práctica anterior

El programa mantiene la generación de individuos representados como arrays donde cada posición representa un servidor caché, además cada uno de estos servidores caché contienen un array con los videos que se le han asignado.

Respecto a la configuración se ha mantenido la función de mutación por permutación shuffleIndex, la cual reordena la asignación de vídeos de una caché a otra, lo cual es óptimo puesto que todas las cachés tienen la misma capacidad.

Como función de cruce se mantiene cxUniform, donde los descendientes contienen un genoma compuesto de forma aleatoria por los genes de los padres. Este método funciona correctamente puesto que se utilizan los datos de los padres sin alterar y dará como resultado hijos donde se aplican configuraciones de cachés iguales a las de dos de los padres.

También se ha decidido mantener el elitismo, donde los tres mejores individuos de cada generación pasan de forma asegurada a la siguiente generación, permitiendo tener una buena base de individuos bien adaptados dentro de la población sin importar los cruces o mutaciones que hayan ocurrido en el cambio de generación.

El programa se ejecuta en la consola con python ./main.py y se debe cambiar el fichero de datos en la siguiente variable:

```
254   # Lectura del archivo especificado
255   filename = "videos_worth_spreading.in"
256   data = read_input(filename)
257
258   if __name__ == "__main__":
```

Cambios en el programa respecto al anterior

Asignación de videos

En el programa anterior mantuve un enfoque anticonceptivo con respecto a la restricción sobre los individuos no válidos, de forma que nunca se podría generar un individuo que superarse su propia capacidad por exceso de videos asignados.

```python
def inicializar_genotipo():
    """
    Genera los genotipos iniciales, colocando videos aleatorios en cada caché
    esta asignación puede exceder la capacidad de las cachés
    """
    # Se define el individuo
    individuo = [[] for _ in range(data["num_caches"])]

    # Lista de videos disponibles
    videos_disponibles = list(range(data["num_videos"]))

    # Mezclar los videos para que cada individuo sea aleatorio
    # Este método de asig
    random.shuffle(videos_disponibles)

    # Se asignan como mucho num videos / 2 para evitar que el algoritmo
    # tarde demasiado en encontrar individuos válidos por exceso de asignaciones
    videos = round(data["num_videos"]/2)

    for cacheID in range(data["num_caches"]):
        # Número de videos a añadir
        max_videos_por_cache = random.randint(1, videos)

        while len(videos_disponibles) > 0 and len(individuo[cacheID]) < max_videos_por_cache:
            # Seleccionar el último video disponible
            videoID = videos_disponibles.pop()
            individuo[cacheID].append(videoID)

    return individuo
```

Para permitir que esto ocurra como se especifica en el enunciado de la práctica he decidido asignar un video aleatorio de videos entre 1 y el máximo de vídeos / 2, de forma que en muchos casos se generan individuos que exceden su capacidad y estos deben ser penalizados por las siguientes funciones.

Función de viabilidad

La función de viabilidad es encargada de evaluar si cada individuo es válido respecto a la restricción de la capacidad de cada caché que conforma el genotipo del individuo. La función recorre las cachés del individuo y suman el uso de cada video que tiene asignado, en caso de exceder la capacidad de alguna de ellas devuelve False (es invalido) y en el caso contrario devuelve True (es válido).

```python
def feasible(ind):
    """
    Función de factibilidad para el individuo.
    Devuelve True si el individuo es válido (ninguna caché excede la capacidad)
    """
    # Inicializar la capacidad usada por caché
    cache_usage = [0] * data["num_caches"]

    # Convertir las cachés del individuo a conjuntos
    cache_sets = [set(cache) for cache in ind]

    for video_id, endpoint_id, num_requests in data["requests"]:
        endpoint_data = data["endpoints"][endpoint_id]
        datacenter_latency = endpoint_data["datacenter_latency"]
        cache_connections = endpoint_data["cache_connections"]

        # Acumula el uso de la capacidad
        for cache_id in cache_connections:
            if video_id in cache_sets[cache_id]:
                cache_usage[cache_id] += data["video_sizes"][video_id]

    # Comprueba si el uso de alguna cache sobrepasa la capacidad
    for usage in cache_usage:
        if usage > data["cache_capacity"]:
            return False

    # El individuo es válido
    return True
```

Función de distancia

La función de distancia se encarga de penalizar dependiendo de cómo de mala sea la adaptación de los individuos al objetivo de la restricción (no exceder la capacidad de cada caché). Para realizar esto, se emplea la siguiente función de distancia:

```python
def distance(ind):
    """
    Función de distancia respecto de los resultados válidos
    Devuelve cuánto excede la capacidad el individuo
    """
    # Inicializar la capacidad usada por caché
    cache_usage = [0] * data["num_caches"]

    # Convertir las asignaciones del individuo a conjuntos
    cache_sets = [set(cache) for cache in ind]

    for video_id, endpoint_id, _ in data["requests"]:
        endpoint_data = data["endpoints"][endpoint_id]
        cache_connections = endpoint_data["cache_connections"]

        # Acumula el uso de la capacidad
        for cache_id in cache_connections:
            if video_id in cache_sets[cache_id]:
                cache_usage[cache_id] += data["video_sizes"][video_id]

    # Calcular el exceso de capacidad total
    total_excess = sum(max(0, usage - data["cache_capacity"]) for usage in cache_usage)
    return total_excess
```

Como se puede apreciar en la función, solo penaliza en caso de exceder la capacidad de alguna caché, donde la penalización es igual al exceso acumulado por todas las cachés, sin embargo, si no hay exceso de asignación no se penaliza (devuelve 0)

Se aplican ambos cambios como un extra sobre la función de evaluación en el siguiente fragmento de código:

```python
# Registrar funciones en DEAP
toolbox.register("individual", tools.initIterate, creator.Individual, inicializar_genotipo)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", eval_sol)

# Comprobación de la validez y distancia de los individuos invalidos respecto de la región válida
toolbox.decorate("evaluate", tools.DeltaPenalty(feasible, 1.0, distance))
```

Optimización multi-objetivo

La función de evaluación se redefine con algunos cambios:

```python
def eval_sol(ind):
    """
    Evalúa la mejora de latencia y el uso de la capacidad de las cachés.
    """
    total_latencia = 0
    latency_data_center = 0

    # Convertir las cachés del individuo a conjuntos
    cache_sets = [set(cache) for cache in ind]

    # Inicializar la capacidad usada por caché
    cache_usage = [0] * data["num_caches"]

    for video_id, endpoint_id, num_requests in data["requests"]:
        endpoint_data = data["endpoints"][endpoint_id]
        datacenter_latency = endpoint_data["datacenter_latency"]
        cache_connections = endpoint_data["cache_connections"]

        # Suma latencia directa al datacenter
        latency_data_center += datacenter_latency * num_requests

        # Calcula la latencia mínima con cachés
```

```python
        # Calcula la latencia mínima con cachés
        min_latency = datacenter_latency
        for cache_id, cache_latency in cache_connections.items():
            if video_id in cache_sets[cache_id]:
                min_latency = min(min_latency, cache_latency)

        # Suma la latencia mínima para esta solicitud
        total_latencia += min_latency * num_requests

        # Acumula el uso de la capacidad
        for cache_id in cache_connections:
            if video_id in cache_sets[cache_id]:
                cache_usage[cache_id] += data["video_sizes"][video_id]

    # Calculo del uso de este individuo
    total_cache_usage = 0
    for cache in cache_usage:
        total_cache_usage += cache

    # Calcular la capacidad total
    total_capacity = data["cache_capacity"] * data["num_caches"]

    # Espacio libre disponible que se quiere maximizar
    # pues to que el espacio libre capacidad - espacio usado crece cuanto
    # más se ahorra espacio utilizado
    free_capacity = (total_capacity - total_cache_usage)
    if(free_capacity < 0):
        free_capacity = 0

    # Calcula la mejora de latencia, se quiere maximizar el
    # ahorro de tiempo en las requests, es decir minimizar
    # lo bueno que es el uso del data center frente al uso de las caches
    latency_gained = latency_data_center - total_latencia

    # Devolver la latencia mejorada y la capacidad ajustada
    return (0.6*latency_gained, 0.4*free_capacity)
```

El principal cambio de esta función consiste en calcular cuanto espacio libre queda en cada caché del individuo y devolver este valor como uno de los objetivos a optimizar, de forma que se maximiza el espacio libre, de forma que las cachés no se llenan con videos los cuales no aportan a minimizar la mejora de latencia.

Se devuelven como una dupla donde se le da más valor la optimización de la latencia (un 60% del valor del fitness) que al ahorro de espacio (solo un 40% del fitness depende de esto), de forma que el objetivo principal sigue siendo mejorar la latencia.

Además se define como un problema multi fitness con ambos objetivos en la maximización:

```
# Configuración de DEAP para maximizar
toolbox = base.Toolbox()
creator.create("FitnessMulti", base.Fitness, weights=(1.0, 1.0))
creator.create("Individual", list, fitness=creator.FitnessMulti)
```

Pruebas

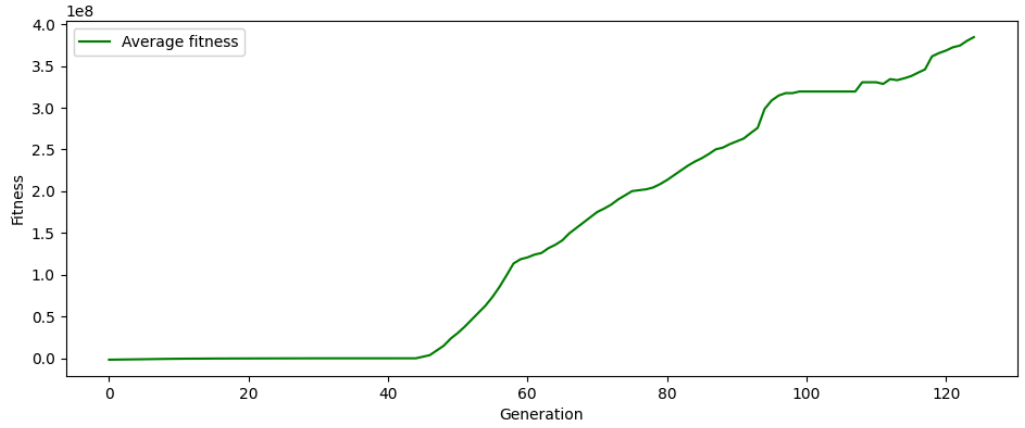
Los parámetros más relevantes en la optimización de este programa han sido el número de generaciones y el método de selección. Si no se emplean suficientes generaciones los resultados obtenidos pueden ser inválidos, puesto que pueden no haber individuos válidos al no haber tenido iteraciones suficientes que eliminen a estos de la población.

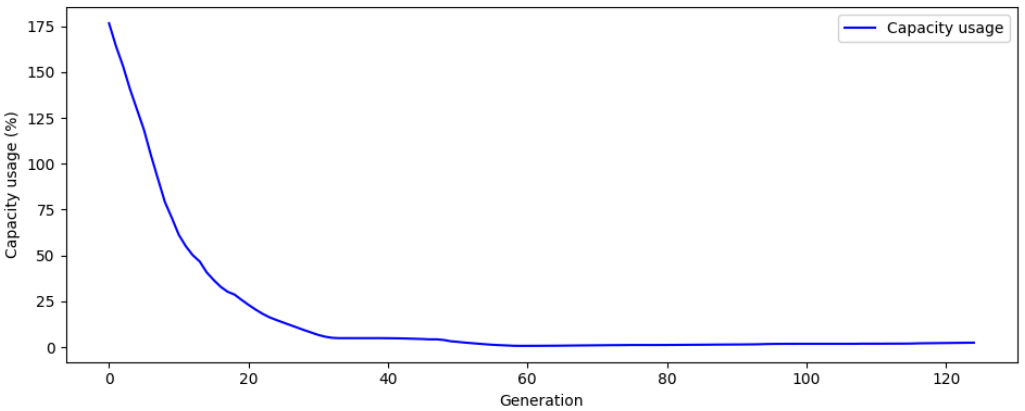
Por otro lado, es un requisito emplear un método de selección capaz de optimizar múltiples objetivos, puesto que de otra forma obtendremos malos resultados, los dos métodos de selección probados son selNSGA2 y selSPEA2.

Las pruebas las voy a realizar con los ficheros me_at_the_zoo.in y videos_worth_spreading.in, puesto que estos son los únicos con los que el tiempo de ejecución no es excesivamente alto.

Selección

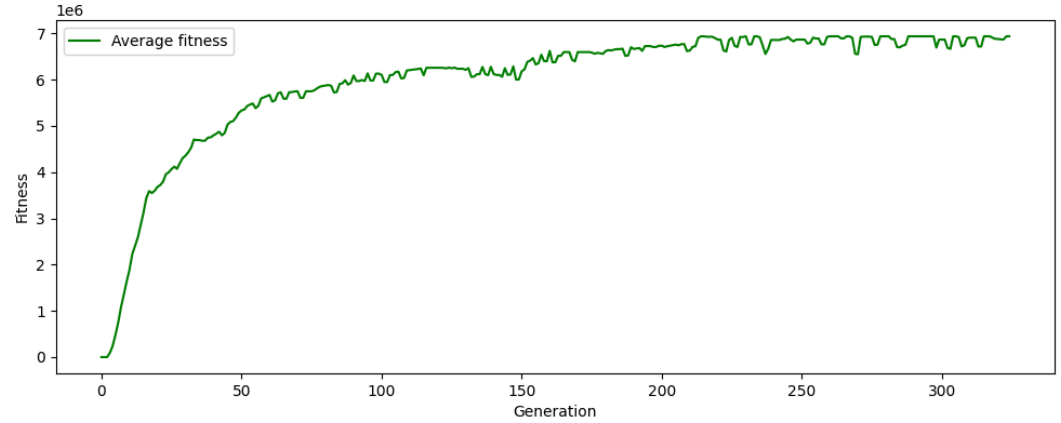
Con **selNSGA2** he conseguido buenos resultados, un tiempo de ejecución aceptable para el fichero videos_worth_spreading, donde ha podido obtener un buen resultado final, aunque ligeramente peor que el algoritmo de selección selSPEA2. Se puede apreciar esto en las siguientes capturas:

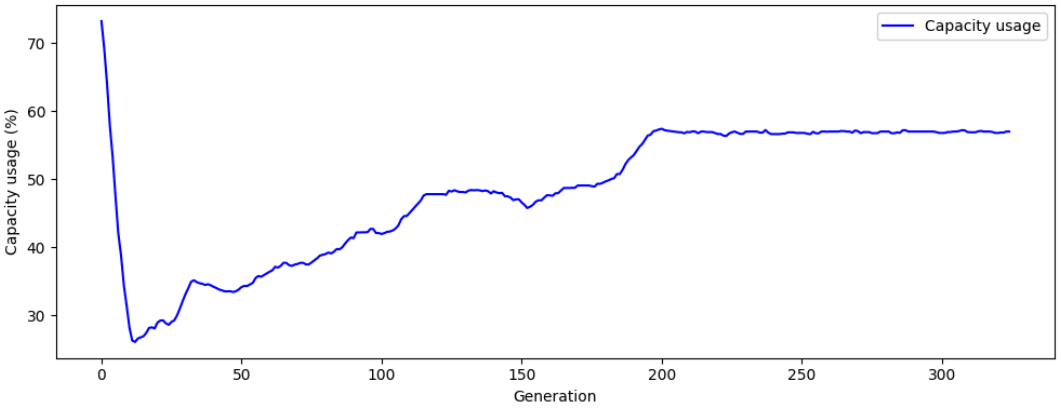





```
123     3.801e+08       8.96365e+08     0               2.35411
124     3.84634e+08     8.96365e+08     0               2.39466
Tiempo de procesamiento: 273.529442 segundos
La mejor solución encontrada es el fenotipo: 896364838
El mejor individuo utiliza una capacidad del 2.860800%
```

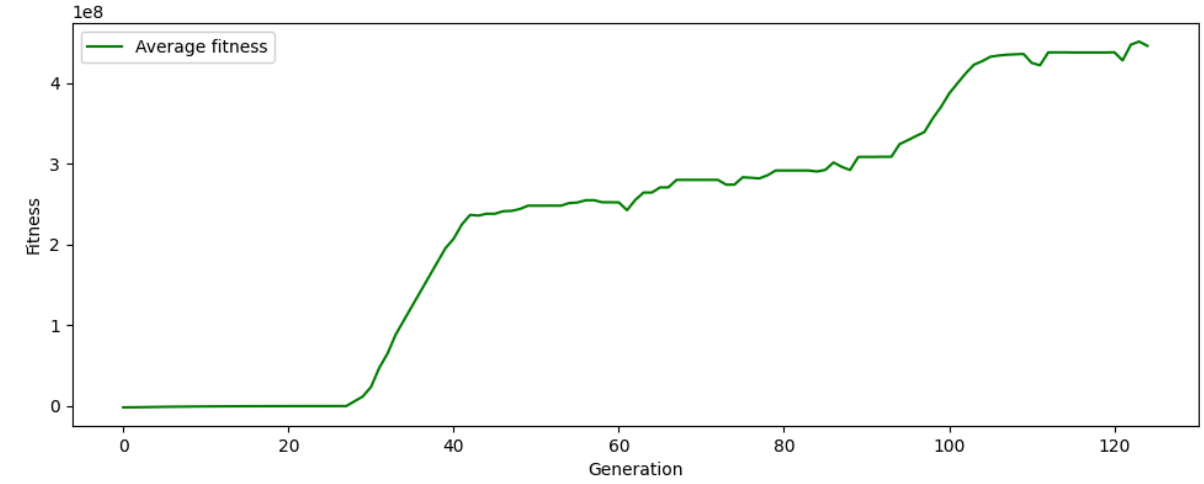
Para el fichero me_at_the_zoo ha conseguido resultados similares al algoritmo selSPEA2, aunque obteniendo un resultado ligeramente peor en cuanto a fitness. Esto se puede ver en el siguiente gráfico y la salida del programa:

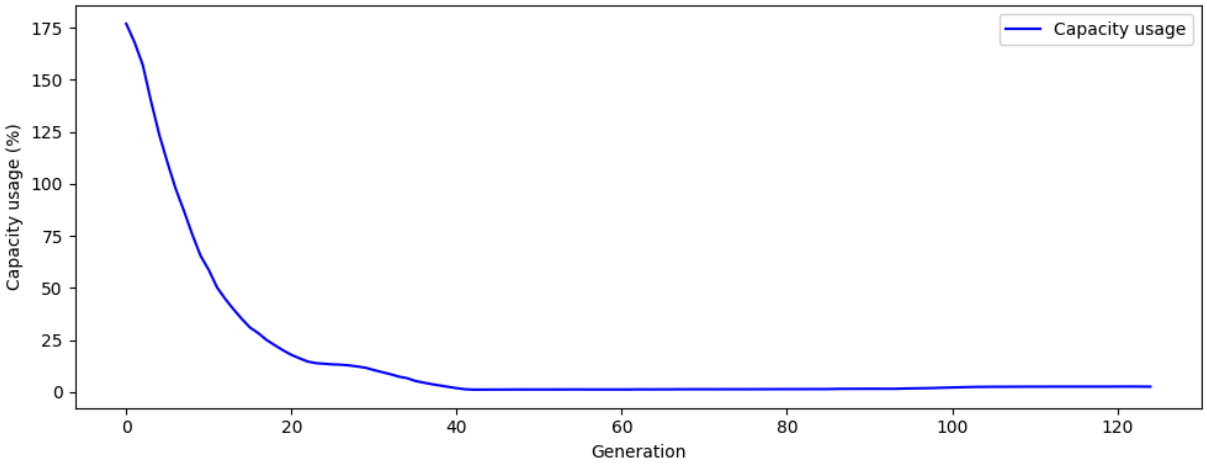





```
321     6.87162e+06     1.38787e+07     204             56.8525
322     6.87162e+06     1.38787e+07     204             56.8525
323     6.93946e+06     1.38787e+07     215             57
324     6.93946e+06     1.38787e+07     215             57
Tiempo de procesamiento: 1.311720 segundos
La mejor solución encontrada es el fenotipo: 13878698
El mejor individuo utiliza una capacidad del 57.000000%
```

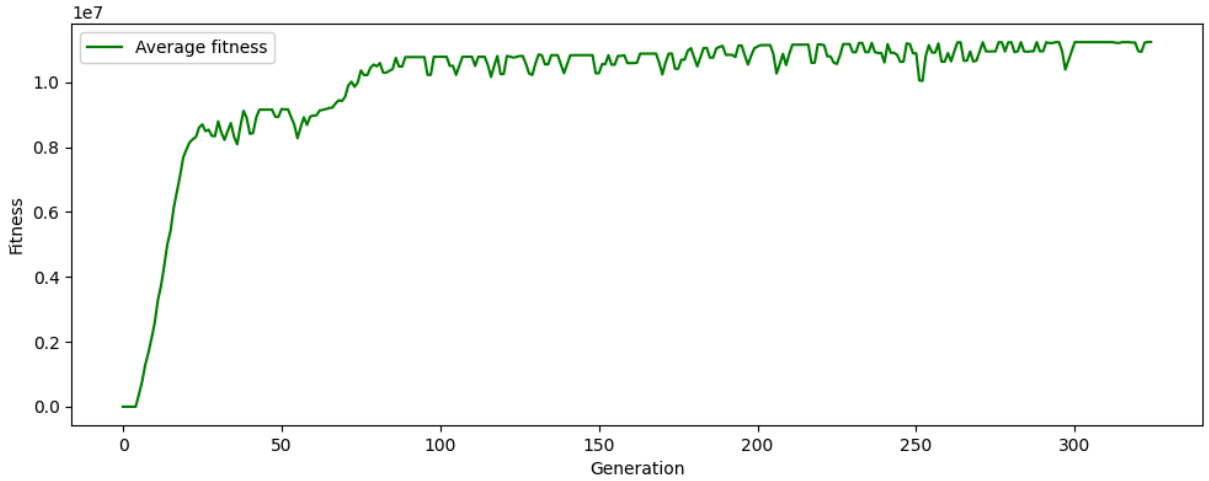
Con **selSPEA2** los resultados han sido muy similares a los obtenidos con selNSGA2, cabe destacar que el algoritmo ha conseguido encontrar soluciones válidas mucho más rápido y por lo tanto ha conseguido optimizar mejor a la población y por lo tanto ha conseguido un mejor resultado. Además el tiempo de ejecución ha sido ligeramente superior. Se puede apreciar en los siguientes resultados:

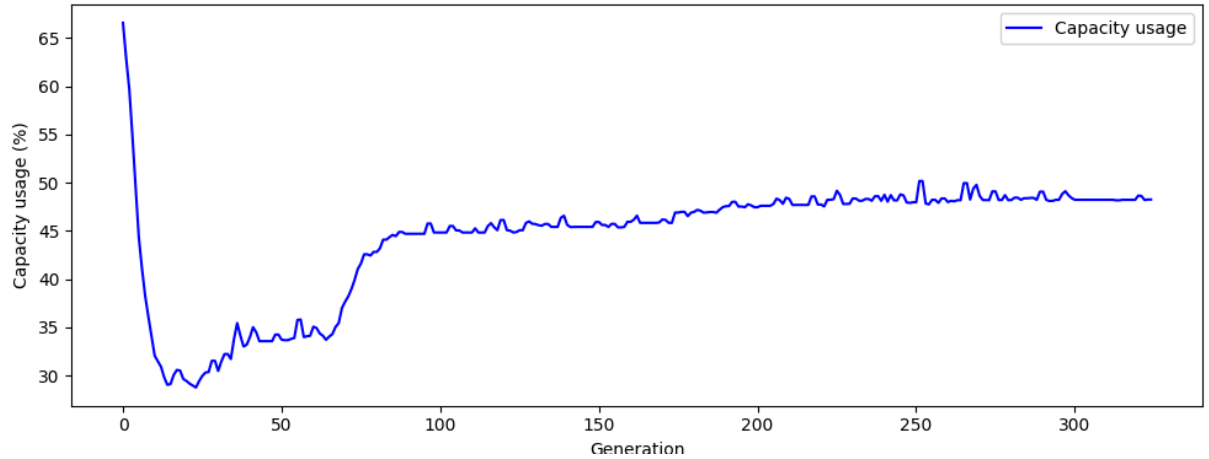





```
122     4.47608e+08     1.19417e+09     482069          2.64096
123     4.51452e+08     1.19417e+09     482069          2.60788
124     4.46131e+08     1.19417e+09     -4618           2.57909
Tiempo de procesamiento: 291.059846 segundos
La mejor solución encontrada es el fenotipo: 1194172308
El mejor individuo utiliza una capacidad del 2.924700%
```

Para el fichero me_at_the_zoo ambos han conseguido resultados muy similares, consiguiendo también converger a un resultado donde el uso de memoria y fitness son muy similares, aunque ligeramente mejores para este algoritmo. Los resultados para este fichero son:



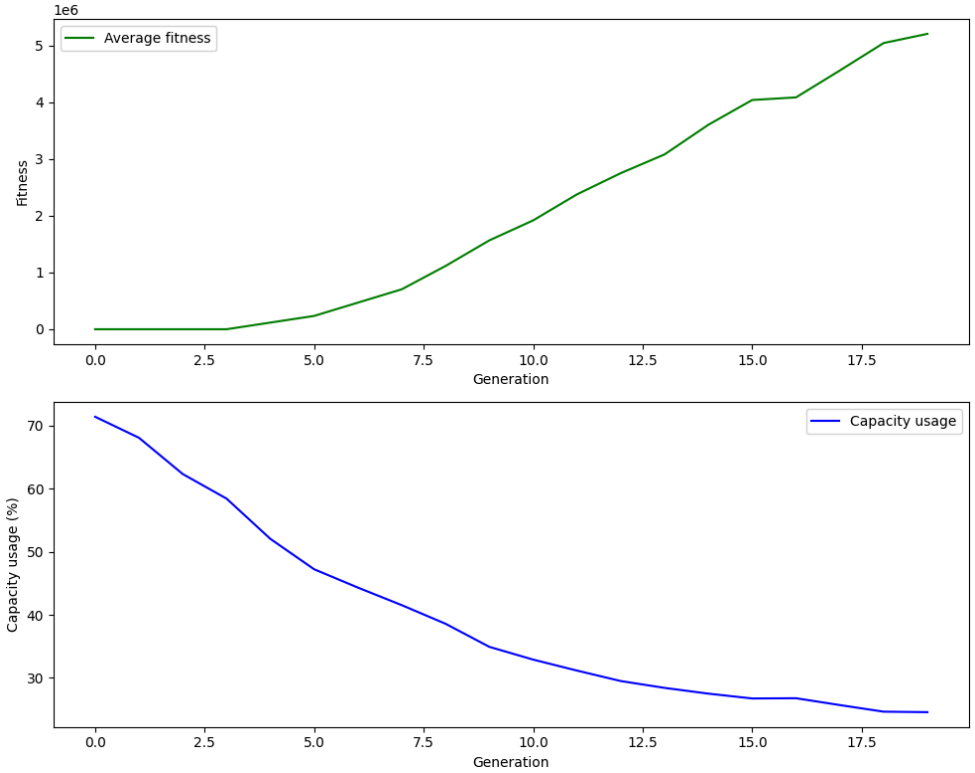



```
320     1.09534e+07     2.25328e+07     -158            48.675
321     1.09318e+07     2.25328e+07     -158            48.625
322     1.12144e+07     2.25328e+07     257.5           48.215
323     1.12351e+07     2.25328e+07     257.5           48.265
324     1.12351e+07     2.25328e+07     257.5           48.265
Tiempo de procesamiento: 1.694206 segundos
La mejor solución encontrada es el fenotipo: 22532804
El mejor individuo utiliza una capacidad del 48.500000%
```

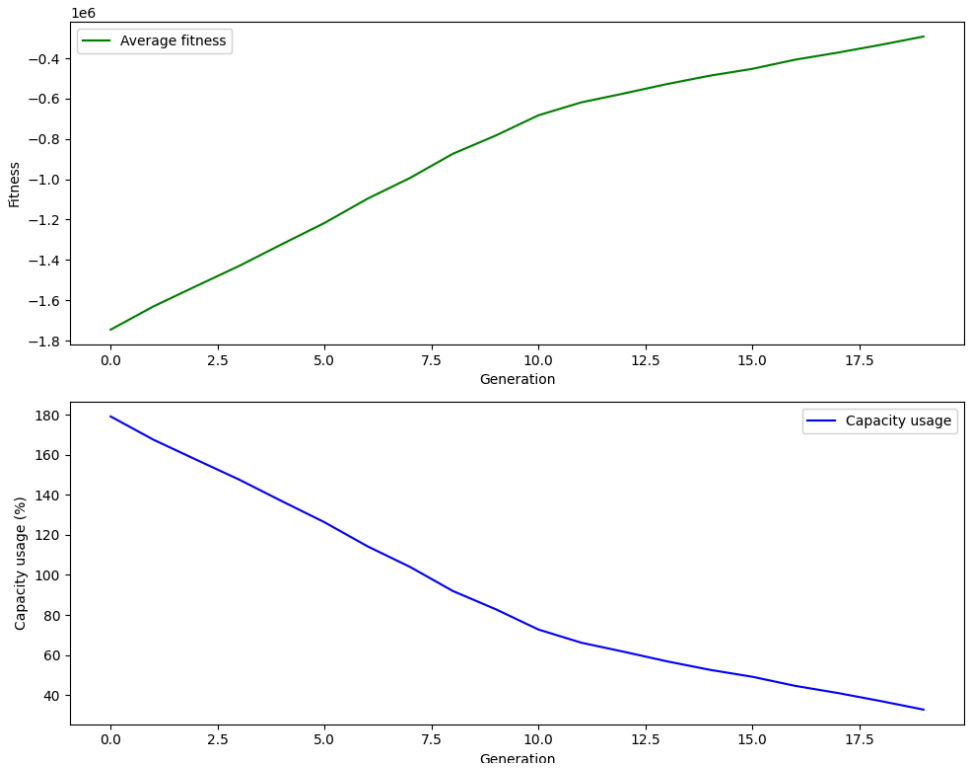
Generaciones

Con **20 generaciones** es un número inadecuado de generaciones, no se obtienen buenos resultados, con ficheros de datos grandes solo consigue resultados negativos (inválidos).

me_at_the_zoo:



videos_worth_spreading:

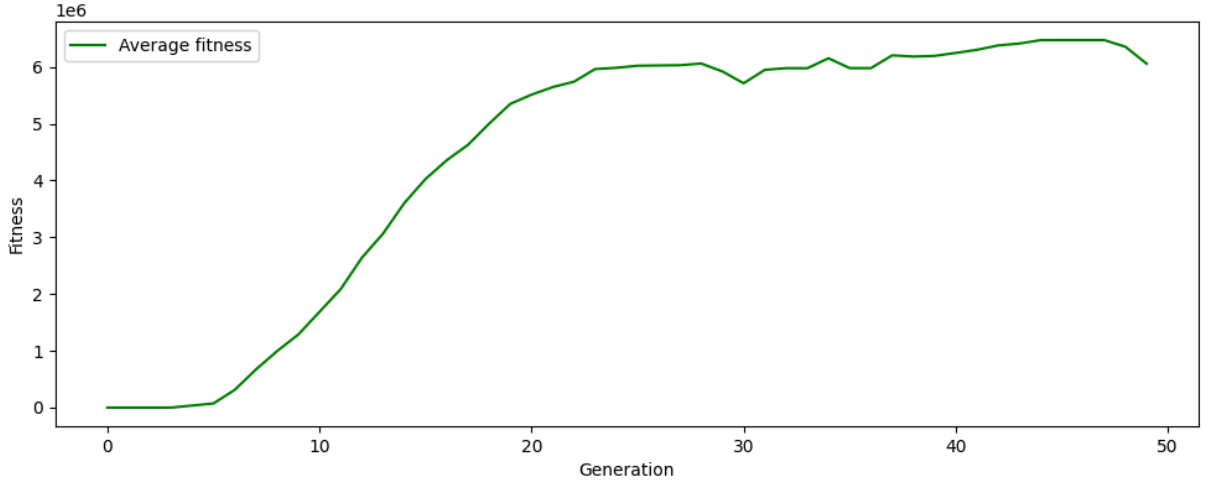


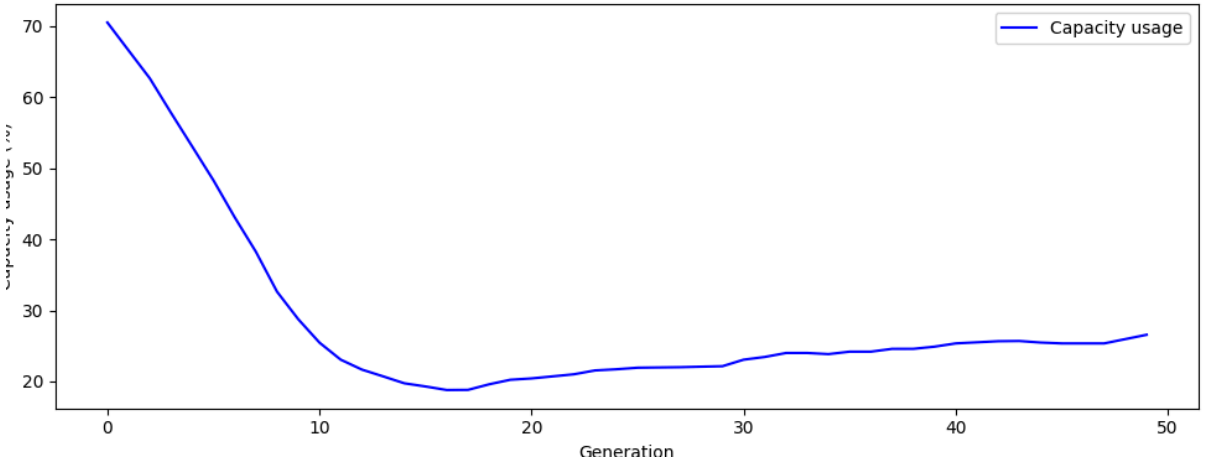
Como se puede ver en esta captura asociada a la gráfica anterior, los resultados aún son penalizados por la función de distancia de forma que son negativos.

```
17      -371725         -82762          -732746         41.0342
18      -333283         -82762          -813947         37.0129
19      -292025         -82762          -671054         32.7634
Tiempo de procesamiento: 39.775101 segundos
La mejor solución encontrada es el fenotipo: 4742621972
El mejor individuo utiliza una capacidad del 11.536200%
```

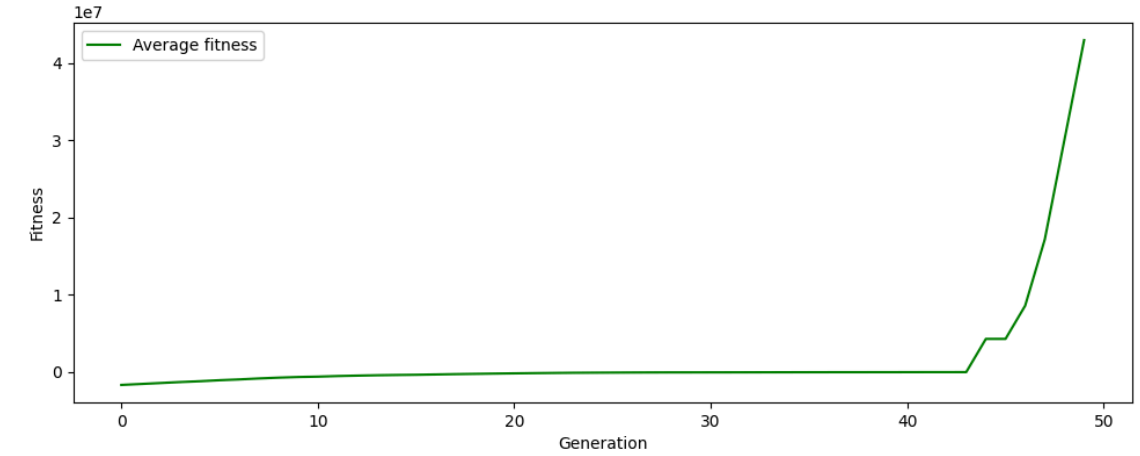
Con **50 generaciones** normalmente el programa comienza a obtener individuos válidos, con lo que, al menos, se obtiene una configuración válida del problema, sin embargo, faltan generaciones para conseguir unos resultados aceptables. Para problemas más complejos como el planteado por videos_worth_spreading se puede llegar a conseguir resultados viables, aunque muy lejos de ser estos óptimos.

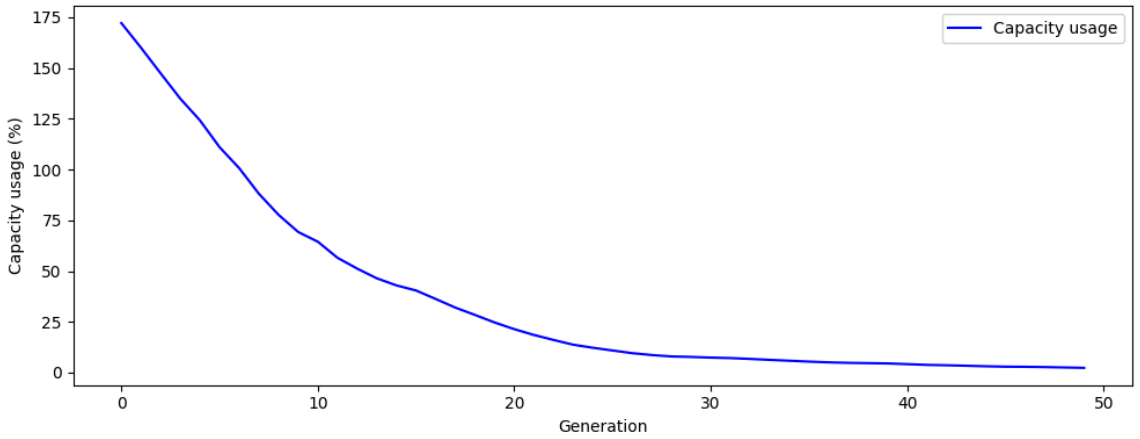
me_at_the_zoo:





videos_worth_spreading:



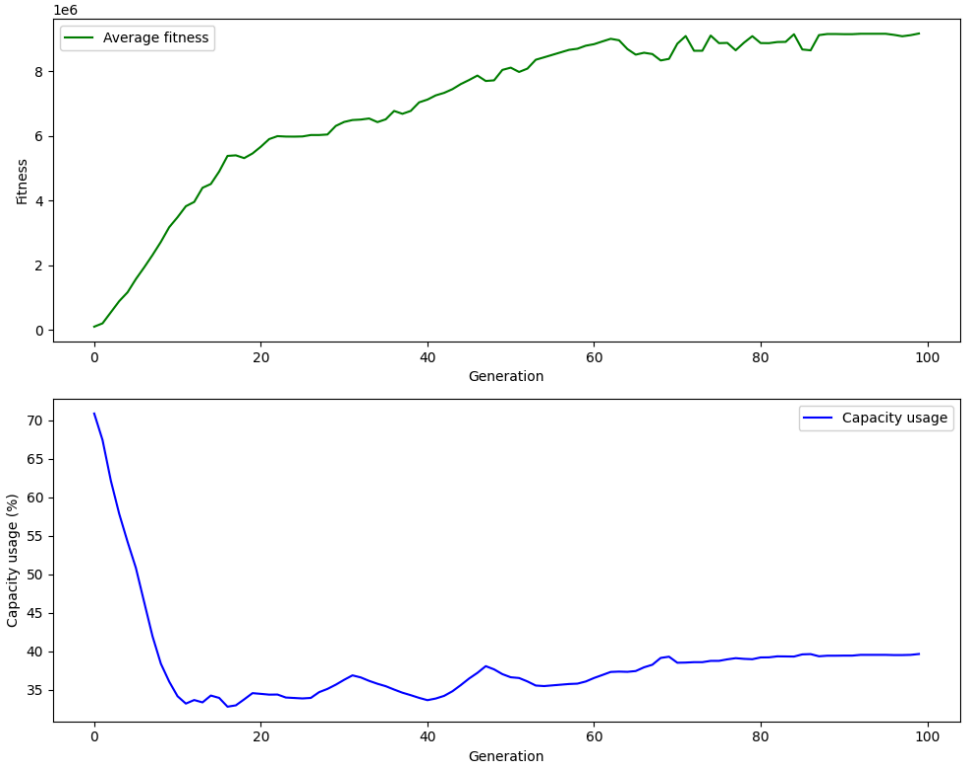


En la siguiente captura asociada a la salida de procesar videos_worth_spreading se puede ver el salto del fitness cuando se encuentran soluciones válidas y la función de penalización deja de restar al fitness:

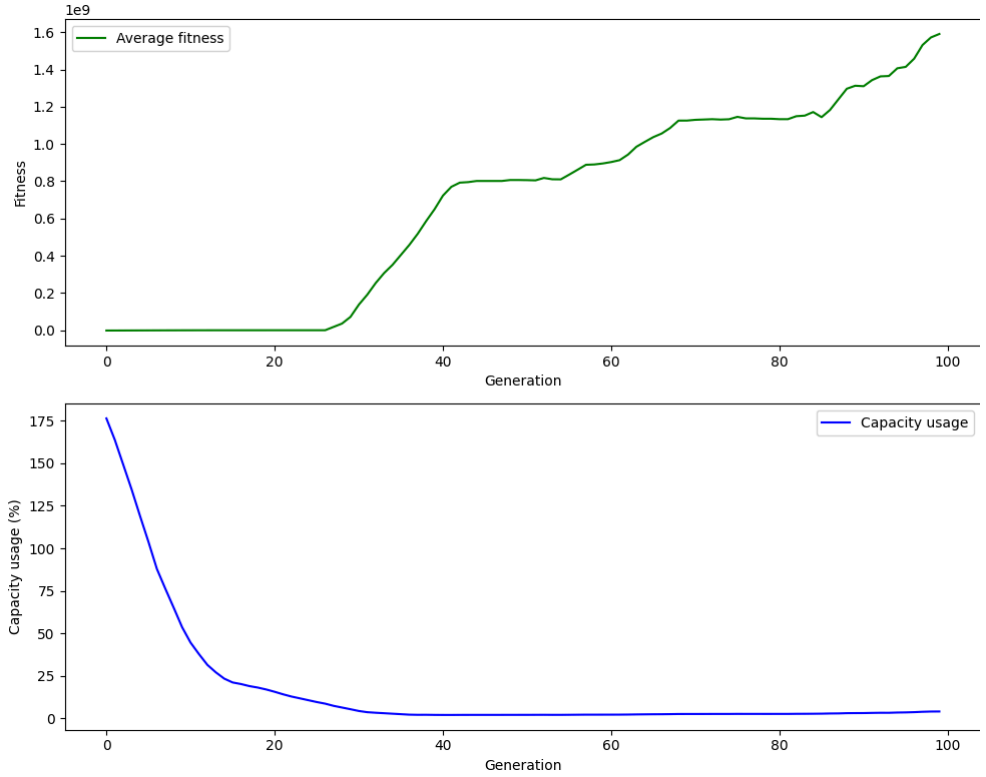
39	-26750	-15535	-101109	4.5085
40	-23709.6	-15535	-97944	4.15318
41	-20748.2	-15535	-41325	3.7939
42	-19384.5	-15535	-62399	3.60624
43	-18061.7	-15535	-53891	3.36146
44	4.28056e+06	3.43262e+08	-67170	3.10305
45	4.28185e+06	3.43262e+08	-15535	2.90186
46	8.57923e+06	3.43262e+08	-15535	2.83802
47	1.7174e+07	3.43262e+08	-15535	2.71034

Con **100 generaciones** se pueden conseguir resultados bastante más optimizados, aunque aún existe margen de mejora en el fitness. Para ficheros de problemas muy sencillos como me_at_the_zoo se llegan a converger a óptimos locales.

me_at_the_zoo:

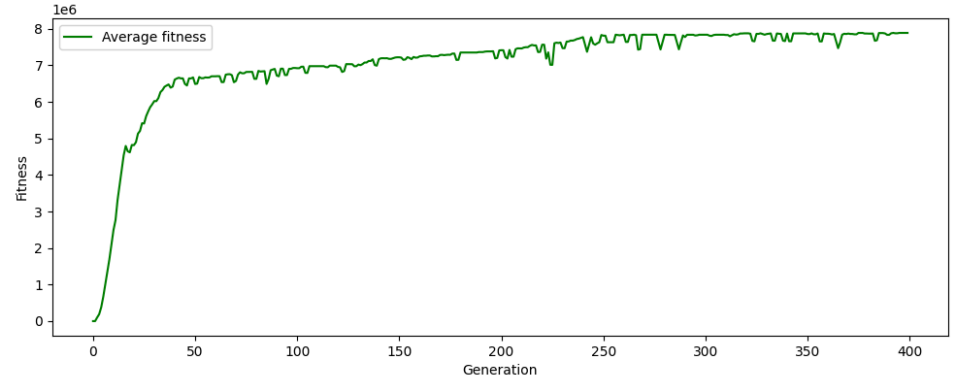


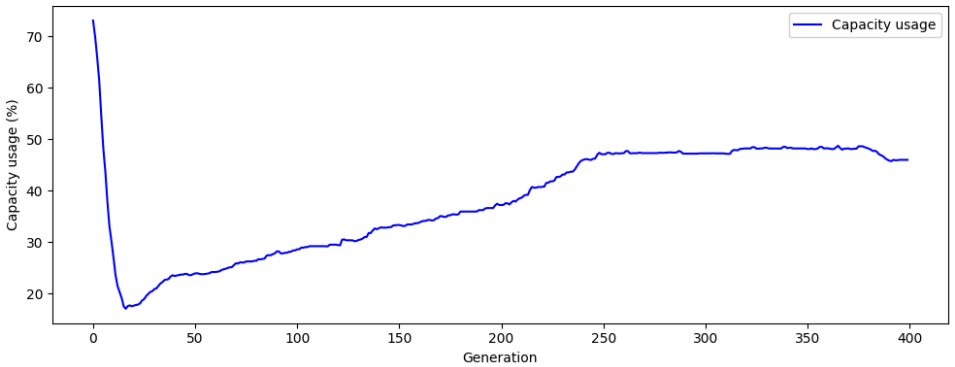
videos_worth_spreading:



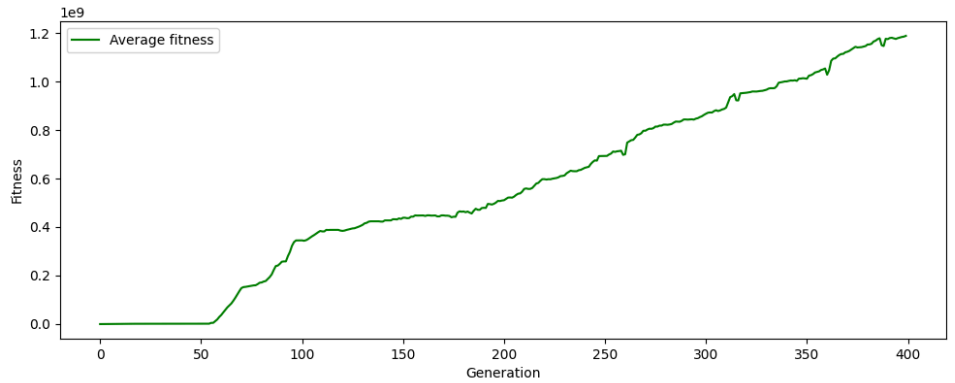
Con **300+ generaciones** se obtienen muy buenos resultados, aunque a cambio de un tiempo de ejecución extremadamente elevado en los casos más complejos. Para estos casos extremadamente complejos pueden ser insuficientes generaciones para alcanzar una solución óptima, aunque si se consiguen soluciones suficientemente buenas.

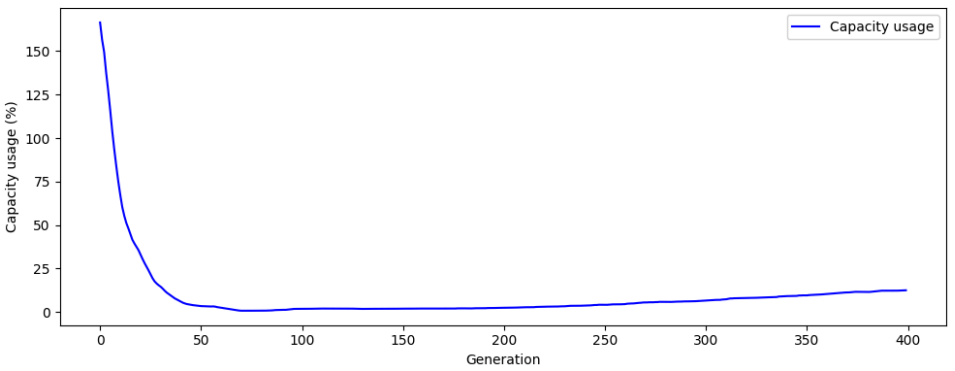
me_at_the_zoo:





videos_worth_spreading:





Resultados obtenidos

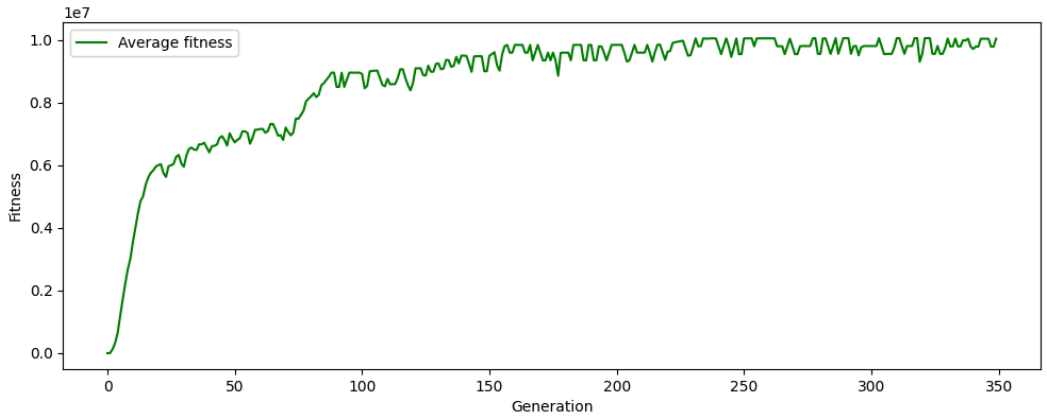
Los resultados obtenidos han sido buenos, aunque probablemente mejorables con una mejor definición de las funciones de distancia y viabilidad.

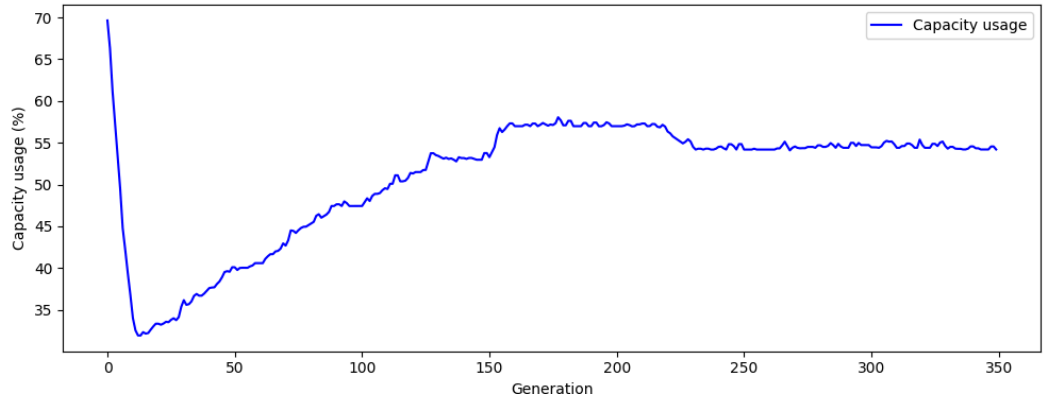
En general el programa presenta un comportamiento bastante predecible, donde los individuos comienzan utilizado mucho más capacidad de la permitida, por consiguiente el rendimiento en cuanto a fitness es muy bajo a causa de la fuerte penalización de la función de distancia.

Seguido de esto el programa llega a un punto donde la capacidad utilizada es viable y el programa presenta un crecimiento muy brusco en su fitness, dado que se deja de penalizar a los individuos de las próximas generaciones, los cuales tienden a tener configuraciones válidas.

Finalmente el programa llega a valores donde el uso de la capacidad es muy bajo y solo se mantienen unos pocos videos en cada caché, pero en algunos casos se reasignan los videos o se asignan más videos a algunos de los servidores caché, reduciendo el espacio disponible pero mejorando aún el fitness, probablemente porque estos son videos que son pedidos por los endpoints en las requests.

Lo descrito anteriormente se puede distinguir gracias a las siguientes gráficas:





El resultado final del programa devuelve el tiempo de ejecución, junto al resultado obtenido en cuanto a mejora de latencia y finalmente el % de uso de las cachés:

```
347     9.79564e+06     2.01322e+07     -60             54.555
348     9.79564e+06     2.01322e+07     -100            54.5625
349     1.00473e+07     2.01322e+07     228             54.2
Tiempo de procesamiento: 1.890968 segundos
La mejor solución encontrada es el fenotipo: 20132210
El mejor individuo utiliza una capacidad del 54.400000%
```

Conclusiones

Tras experimentar y probar diferentes formas de implementar este nuevo método de optimizar los resultados obtenidos para el problema, he tenido algunas dificultades consiguiendo que el programa no penalice demasiado con la función de distancia.

También he experimentado algunas dificultades consiguiendo que el fitness no dependiera excesivamente de ninguno de los dos parámetros a optimizar. Una buena solución a este problema hubiese sido normalizar ambos valores a un % de mejora de latencia y de capacidad utilizada, de forma que ambos valores hubiesen sido valores entre 0 y 1. Por falta de tiempo me ha sido imposible realizar esta mejora.

El uso de dos métricas como en este caso con la mejora de latencia y el uso de memoria de las cachés me han dado problemas a la hora de recoger los datos en el logbook de DEAP, esta dificultad no ha sido fácil de resolver mediante la documentación de la librería donde se proponen diferentes soluciones pero con malos resultados en mi caso.

Como última dificultad me gustaría resaltar los graves problemas de rendimiento que he tenido que afrontar, puesto que mi implementación multiproceso del programa de la práctica anterior no ha funcionado para este caso, al parecer por la adición de las funciones de distancia y viabilidad en la parte de la evaluación de los individuos. El método propuesto por DEAP para resolver este problema (la librería SCOOP) tampoco me ha resuelto dicho problema, por lo que hacer las pruebas me ha tomado un extra de tiempo.

Aún con todas estas dificultades y mejoras que me han sido imposible realizar, considero que he experimentado ampliamente con este tipo de problema y los métodos con los que se puede resolver y obtener los resultados.