Metaheurística

Práctica 1: Codificación y evaluación de soluciones

24 de febrero de 2019

Resumen

In the present document we will explain how we have solved the Multiple Quadratic Knapsack Problem using C++ language. Furthermore,we will explain all the methods used to solved it and all the variables needed in order to solve it. Finally, we will show plots with the convergence of the algorithm to see how it works.

Índice

1. Introducción

Como ya sabemos, el problema de la mochila múltiple cuadrática consiste en averiguar la cantidad óptima de objetos que caben en varias mochilas, las cuales tienen un límite de *peso*, con lo que la mochila no podría superar ese *peso* al introducir un determinado número de objetos. Como aclaración, ningún objeto puede estar en dos mochilas a la vez. Además, cada objeto, junto con su *peso*, cuenta con una variable llamada *beneficio*, que consiste en el *beneficio* que ofrece ese objeto al introducirlo en cualquiera de las mochilas. Este *beneficio* puede ser de dos formas, primero, el *beneficio* individual, que como ya hemos explicado, es el *beneficio* de introducirlo, sólamente a ese objeto, en cualquier mochila. El segundo *beneficio* es el que nos proporciona un objeto al ser introducido junto con otro en una determinada mochila. Una vez explicado esto, el problema consiste en conseguir el mayor *beneficio* posible llenando las mochilas sin sobrepasar sus límites.

Para dividir mejor este documento, vamos a crear las siguientes secciones, que serán acordes a las clases usadas en el programa. Estas secciones son las siguientes:

1. **MQKPIntance**: Es la clase en la que se crearán los objetos junto con las mochilas.
2. **MQKPEvaluator**: Clase encargada de evaluar el beneficio(fitness) proporcionado por el algoritmo.
3. **MQKPSolution**: Clase que creará las variables necesarias para guardar la solución final.
4. **MQKPSolGenerator**: Es la clase que creará una solución aleatoria del problema.

Al final de la explicación de todas las clases, mostraremos los resultados obtenidos junto con gráficas de convergencia para poder apreciar mejor cómo funciona nuestro algoritmo.

2. Clase MQKPInstance

Esta clase es la encargada de guardar los siguientes datos: número de mochilas, número de objetos, beneficios de cada uno de los objetos, el peso de los objetos y la capacidad de cada una de las mochilas. Para ello, deberemos de crear las variables privadas de esta clase de la siguiente manera:

```
1    int _numKnapsacks;
2    int _numObjs;
3    vector<vector<float> > _profits;
4    vector<float> _weights;
5    vector<float> _capacities;
```

Listing 1: Variables privadas

Como vemos, las variables privadas de esta clase son o números enteros o vectores. Lo más importante de esta clase es el vector de vectores *_profits*, que guarda los beneficios tanto individuales como por parejas de cada uno de los objetos.

Seguidamente, habría que crear los métodos observadores y modificadores de cada una de las variables, quedando de la siguiente forma:

```
1   inline int getNumSacks()
2   {
3       return this->_numKnapsacks;
4   }
5
6   inline int getNumObjs()
7   {
8       return this->_numObjs;
9   }
10
11  inline float getProfits(int row, int col)
12  {
13      return this->_profits[row][col];
14  }
15
16  inline float getWeights(int i)
17  {
18      return this->_weights[i];
19  }
20
21  inline double getCapacity(int pos)
22  {
23      return this->_capacities[pos];
24  }
```

Listing 2: Observadores de las variables

```
1   inline void setNumObj(double numObj)
2   {
3       this->_numObjs = numObj;
4   }
5
6   inline void setNumSacks(int numSacks)
7   {
8       this->_numKnapsacks = numSacks;
9   }
10
11  inline void setProfits(int row, int col, int value)
12  {
13      this->_profits[row][col] = value;
14  }
15
16  inline void setWeight(int pos, float value)
17  {
18      this->_weights[pos] = value;
19  }
20
21  inline void setCapacity(int pos, float value)
22  {
23      this->_capacities[pos] = value;
24  }
```

Listing 3: Modificadores de las variables

Una vez creadas las variables de la clase junto con los métodos correspondientes para su uso y modificación, pasaremos a inicializar el constructor y el destructor de

la clase. Quedando de la siguiente manera:

```
MQKPInstance::MQKPInstance()
{
    _numKnapsacks = 0;
    _numObjs = 0;
    _profits.resize(0);
    _weights.resize(0);
    _capacities.resize(0);
}

MQKPInstance::~MQKPInstance()
{
    for (int i = 0; i < _numObjs; i++)
    {
        _profits[i].clear();
    }
    _profits.clear();
    _weights.clear();
    _capacities.clear();
}
```

Listing 4: Constructor y destructor de la clase

Una vez mostrados las funciones más importantes, comenzaremos con nuestro algoritmo. Lo primero que debemos de hacer es leer la información de los objetos y mochilas de un fichero, para ello, hemos utilizado la siguiente función:

```
void MQKPInstance::readInstance(char *filename, int numKnapsacks)
{
    this->setNumSacks(numKnapsacks);

    ifstream file;
    file.open(filename);

    // Leemos la primera l nea
    string aux;
    file >> aux;

    // Leemos la segunda linea(n mero objetos)
    double numObj;
    file >> numObj;
    this->setNumObj(numObj);

    // Reservamos memoria
    this->_profits.resize(this->getNumObjs());
    for (int i = 0; i < this->getNumObjs(); i++)
    {
        this->_profits[i].resize(this->getNumObjs());
    }
    this->_weights.resize(this->getNumObjs());
    this->_capacities.resize(this->getNumSacks());

    // Leemos los beneficios y pesos
    // Primero leemos la diagonal
    double profits;
    for (int i = 0; i < this->getNumObjs(); i++)
    {
        file >> profits;
        this->setProfits(i, i, profits);
    }
    // Leemos la mitad superior derecha
```

```
35      for (int i = 0; i < this->getNumObjs() - 1; i++)
36      {
37          for (int j = i + 1; j < this->getNumObjs(); j++)
38          {
39              file >> profits;
40              this->setProfits(i, j, profits);
41              this->setProfits(j, i, profits);
42          }
43      }
44      file >> profits;
45      file >> profits;
46      for (int i = 0; i < this->getNumObjs(); i++)
47      {
48          file >> profits;
49          this->setWeight(i, profits);
50      }
51      file.close();
52
53      // Calculamos la capacidad de la mochila
54      int sum = 0;
55      for (int i = 0; i < this->getNumObjs(); i++)
56      {
57          sum += this->getWeights(i);
58      }
59      sum *= 0.8;
60      sum /= this->getNumSacks();
61      this->setCapacity(0, -1);
62      for (int i = 1; i <= this->getNumSacks(); i++)
63      {
64          this->setCapacity(i, sum);
65      }
```

Listing 5: Función de lectura de un fichero

Una vez recogidas todas las variables de un fichero, hay que crear las funciones para averiguar la violación máxima de una mochila al introducir un nuevo objeto, y además, hay que calcular el beneficio obtenido por la inserción de éste. La siguiente función, es la que calcula la máxima violación:

```
1  double MQKPInstance::getMaxCapacityViolation(MQKPSolution &solution)
2  {
3
4      double *sumWeights = new double[this->_numKnapsacks + 1];
5
6      for (int j = 1; j <= this->_numKnapsacks; j++)
7      {
8          sumWeights[j] = 0;
9      }
10
11     for (int i = 0; i < this->_numObjs; i++)
12     {
13         if (solution.whereIsObject(i) > 0)
14         {
15             sumWeights[solution.whereIsObject(i)] = sumWeights[solution.whereIsObject(
       i)] + this->getWeights(i);
16         }
17     }
18
19     double maxCapacityViolation = 0; //Inicializamos la m xima violaci n de alguna
         mochila a 0, que significa que no hay ninguna violaci n
20     int object = 0;
21     for (int j = 1; j <= this->_numKnapsacks; j++)
22     {
```

```
23          if ((sumWeights[j] - this->getCapacity(j)) > maxCapacityViolation)
24          {
25              maxCapacityViolation = sumWeights[j] - this->getCapacity(j);
26              object = j;
27          }
28      }
29
30      delete[] sumWeights;
31      return maxCapacityViolation;
```

Listing 6: Función para calcular la máxima violación de capacidad

Finalmente, hemos creado la función que calcula el beneficio obtenido al conseguir una solución, es decir, en el caso de que no se produzca una violación de la capacidad.

```
1  double MQKPInstance::getSumProfits(MQKPSolution &solution)
2  {
3
4      double sumProfits = 0.;
5
6      for (int i = 0; i < this->getNumObjs() - 1; i++)
7      {
8          if (solution.whereIsObject(i) != 0)
9          {
10             sumProfits += this->getProfits(i, i);
11             for (int j = i + 1; j < this->getNumObjs(); j++)
12             { //i+1 para que salte la diagonal
13                 if (solution.whereIsObject(j) == solution.whereIsObject(i))
14                 {
15                     sumProfits += this->getProfits(i, j);
16                 }
17             }
18         }
19     }
20
21     if (solution.whereIsObject(this->getNumObjs() - 1) != 0)
22     {
23         sumProfits += this->getProfits(this->getNumObjs() - 1, this->getNumObjs() -
       1);
24     }
25     return sumProfits;
26 }
```

Listing 7: Función que calcula el beneficio total

3. Clase MQKPSolution

Esta clase es la encargada de administrar la solución obtenida. Para ello, primero debemos de declarar las bibliotecas que usaremos en esta clase, así como crear las variables *protected* de esta clase, tal y cómo vemos en la siguiente Listing:

```
1      vector<int> _sol;
2      int _numObjs;
3      double _fitness;
```

Listing 8: Variables clase MQKPSolution

Como vemos, las variables que hemos de definir en esta clase son tres: un vector de enteros que será la representación interna de la solución **sol** , un entero en el que almacenaremos el número de objetos que se tiene en el problema **numObjs** y una

variable flotante doble en la que se almacenará la calidad de la solución **fitness**.

Seguidamente, inicializaremos las variables privadas en el constructor de la clase y además, completaremos el destructor, tal y como podemos observar en el siguiente Listing:

```
1  MQKPSolution::MQKPSolution(MQKPInstance &instance)
2  {
3      this->_numObjs = instance.getNumObjs();
4      this->_fitness = 0.0;
5      this->_sol.resize(_numObjs);
6  }
7
8  MQKPSolution::~MQKPSolution()
9  {
10     this->_sol.clear();
11 }
```

Listing 9: Constructor y destructor de la clase MQKPSolution

Además, crearemos los métodos de observación y modificación de la variable _sol, ya que es un vector y al estar protegido necesitaremos una función para poder rellenarlo. La solución se puede observar en el siguiente Listing:

```
1  void MQKPSolution::putObjectIn(int object, int knapsack)
2  {
3      this->_sol[object] = knapsack;
4  }
5
6  int MQKPSolution::whereIsObject(int object)
7  {
8      return this->_sol[object];
9  }
```

Listing 10: Métodos de la clase MQKPSolution

Cómo se puede observar en la imagen anterior, la función **putObjectIn** asigna un objeto a la mochila indicada. Esta, recibe dos parámetros de entrada:

- **object**: Índice del objeto a insertar en la mochila deseada.
- **knapsack**: Índice de la mochila donde insertar el objeto.

El objetivo de la función **whereIsObject**, es decir en qué mochila está insertado un objeto. Dicha función recibirá como parámetro de entrada la variable de tipo entero **object**, qué es un índice del objeto del que queremos saber su localización.

4. Clase MQKPSolGenerator

Esta clase creará una solución aleatoria según los datos iniciales. Como en la clase **MQKPEvaluator**, no tendremos que crear ninguna variable para esta clase, sino que sólamente habrá que crear la función necesaria que nos devolverá una solución al azar. En el siguiente Listing podremos ver una posible solución:

```
1  void MQKPSolGenerator::genRandomSol(MQKPInstance &instance, MQKPSolution &solution)
2  {
3      int numObjs = instance.getNumObjs();
4      int numKnapsacks = instance.getNumSacks();
5
6      for (int i = 0; i < numObjs; i++)
7      {
8          int randomKnapsack = rand() %(1 + numKnapsacks);
9          solution.putObjectIn(i, randomKnapsack);
10     }
11 }
```

Listing 11: Función de la clase MQKPSolGenerator

Como vemos, lo primero que hace esta función será saber cuántos objetos y cuántas mochilas hay en el problema. Seguidamente, escogerá una mochila al azar e insertará el objeto en esa mochila conforme avanza leyendo los objetos disponibles.

5. Clase MQKPEvaluator

Esta clase es la encargada de evaluar la solución proporcionada por las demás clases. Esta clase no tiene ninguna variable, con lo que directamente pasaremos a crear la función para evaluar la solución. La solución se puede observar en el siguiente Listing:

```
1  double MQKPEvaluator::computeFitness(MQKPInstance &instance, MQKPSolution &solution
       )
2  {
3
4      double fitness = 0;
5
6      if (instance.getMaxCapacityViolation(solution) > 0)
7      {
8          fitness = instance.getMaxCapacityViolation(solution) * -1;
9      }
10     else
11     {
12         fitness = instance.getSumProfits(solution);
13     }
14     return fitness;
15 }
```

Listing 12: Función de la clase MQKPEvaluator

Como vemos, la función **computeFitness** se encarga de calcular el beneficio final de la solución. Si la solución obtenida devuelve una violación de la capacidad máxima, esta función la multiplicará por -1 para poder así diferenciar mejor cuándo una solución es válida y cuándo no. Finalmente, como vemos desde la línea 11 hasta la 13 del Listing anterior, esta función llama a la función **getSumProfits** de la clase **MQKPInstance**, la cuál calculará el beneficio conseguido al crear la solución.

6. Experimentos

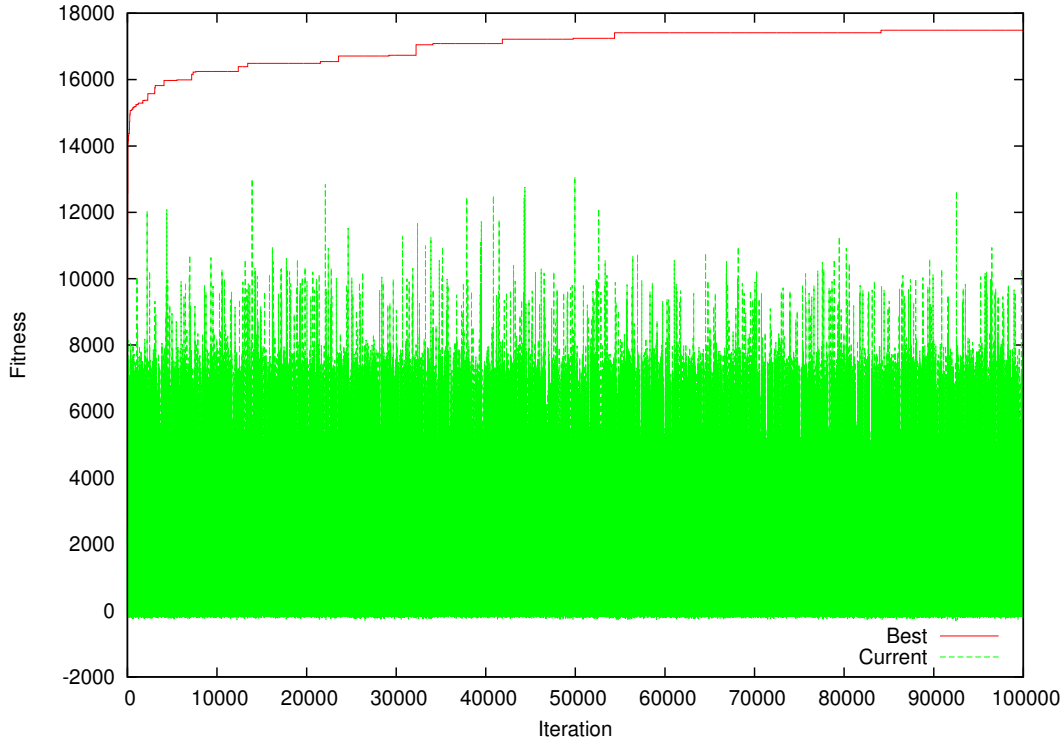
Una vez creadas todas las clases necesarias, pasaremos a la realización de los experimentos y a su correspondiente explicación. Para ello, esta sección la dividiremos

en cuatro, ya que son cuatro ficheros con distintos datos. Además, cada sección estará dividida en dos, una solución obtenida para tres mochilas y otra solución para cinco mochilas.

6.1. Fichero jeu_100_25_4.txt

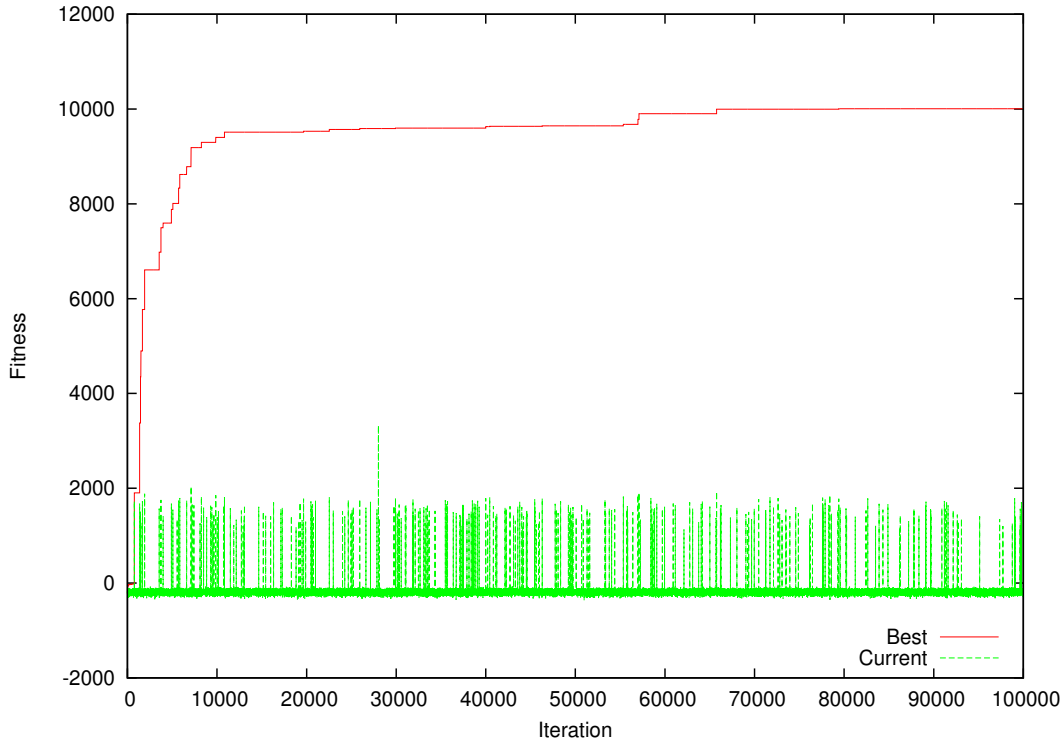
Este fichero contiene un cierto valor de objetos, que en este caso son 100 objetos a poder insertar en las mochilas.

- **Solución con 3 mochilas:** Como vemos en la Figura 1, el beneficio obtenido en cada iteración aumenta, llegando a un punto en el que la mejora conseguida en la siguiente iteración no hace mejorar la media del mejor, de ahí podemos observar que la función de mejora es escalonada y creciente.



Cuadro 1: Convergencia del algoritmo con 100 objetos y 3 mochilas

- **Solución con 5 mochilas:** A diferencia de la solución obtenida con 3 mochilas, al utilizar 5 mochilas vemos como el beneficio obtenido es peor. En la Figura 2 podemos observar la convergencia de esta mejora. Como vemos, al igual que antes, la función que representa el mejor beneficio de cada iteración, es escalonada y creciente. La mayor diferencia es que al comenzar el algoritmo este beneficio es mínimo, a diferencia de la conseguida en la Figura 1, que en el primer instante el beneficio conseguido es positivo.

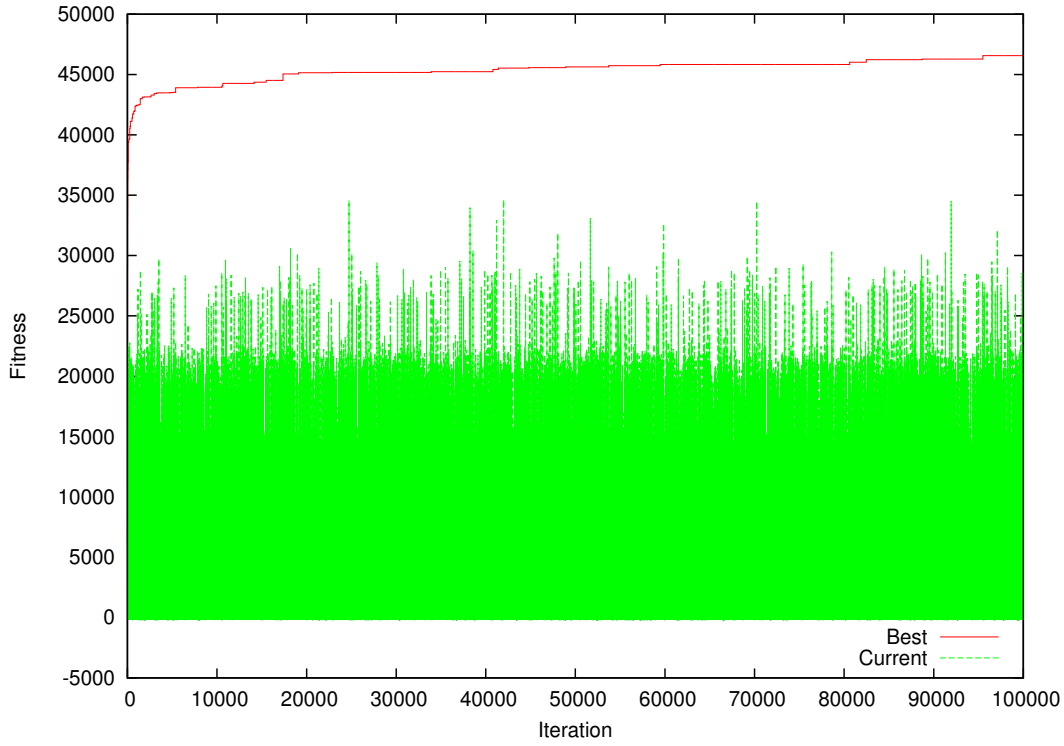


Cuadro 2: Convergencia del algoritmo con 100 objetos y 5 mochilas

6.2. Fichero jeu_100_75_2.txt

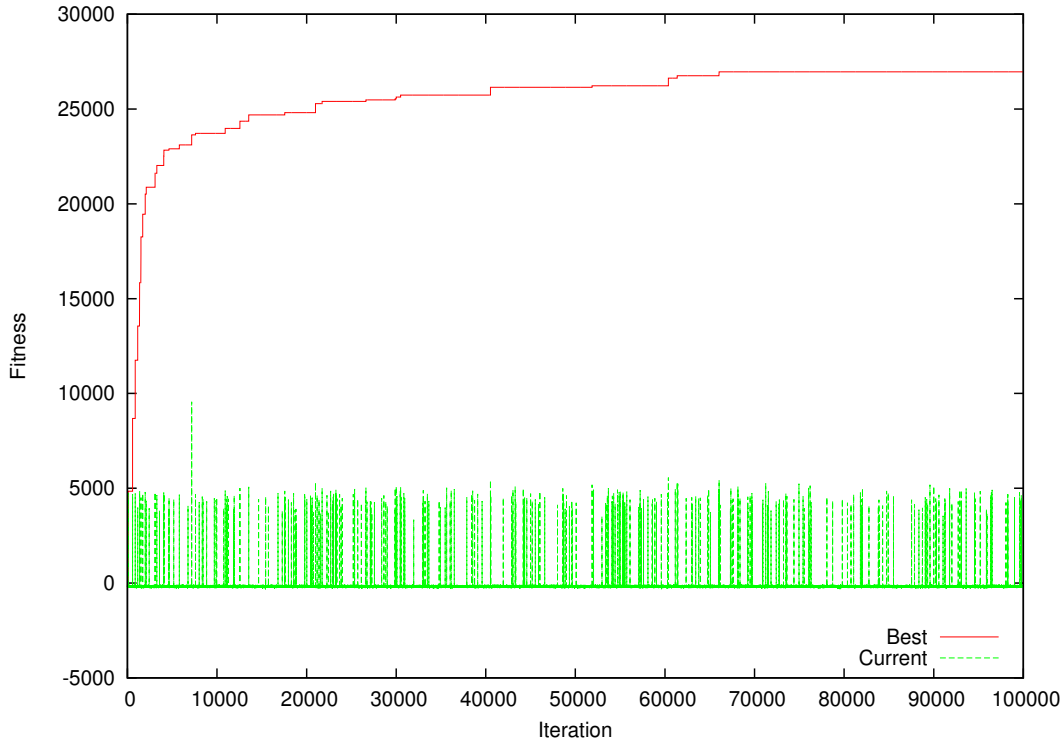
Este fichero contiene un cierto valor de objetos, que en este caso son 100 objetos a poder insertar en las mochilas.

- **Solución con 3 mochilas:** Como vemos en la Figura 3, pasa como en la Figura 1, el beneficio al principio es positivo, y además la función del mejor beneficio obtenido en cada iteración es escalonada y creciente. La diferencia con la Figura 1, es que no es tan escalonada, esto quiere decir que la diferencia de beneficio en cada iteración es casi la misma, de ahí que no presente escalones tan pronunciados.



Cuadro 3: Convergencia del algoritmo con 100 objetos y 3 mochilas

- **Solución con 5 mochilas:** Como vemos en la Figura 4, el beneficio obtenido con cinco mochilas es menor que el conseguido con 3, y a diferencia con la Figura 2 esta solución presenta muchos más escalones que la anterior. Esto quiere decir que la diferencia entre el beneficio conseguido en una iteración y la siguiente es muy grande, produciendo así más escalones.

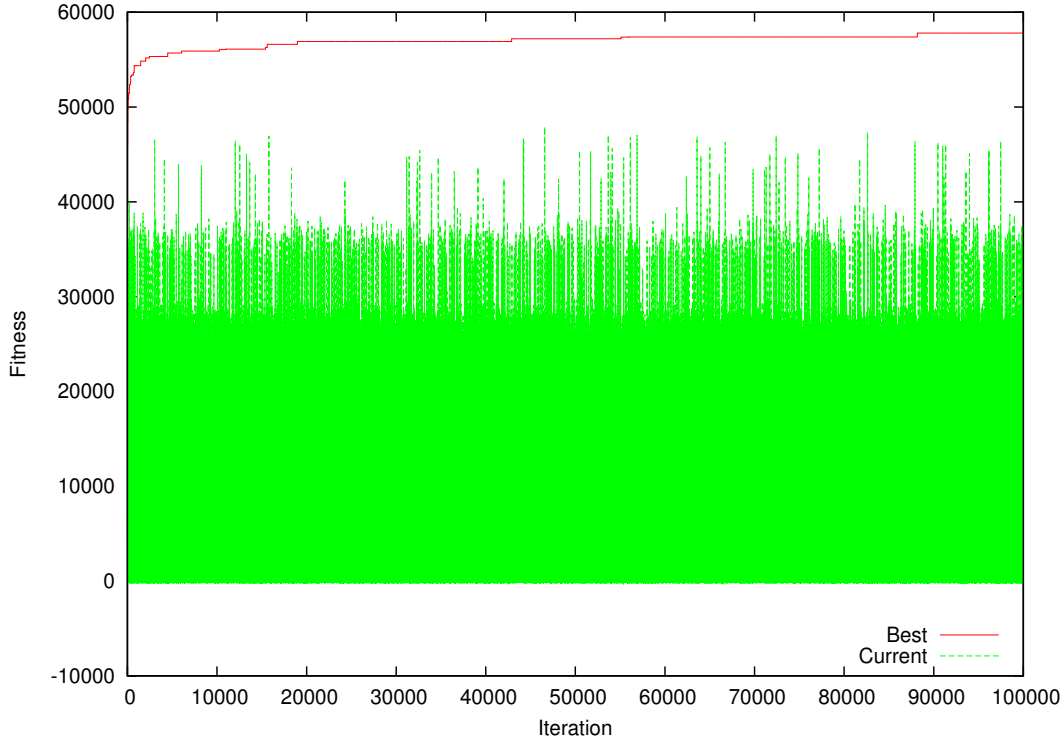


Cuadro 4: Convergencia del algoritmo con 100 objetos y 5 mochilas

6.3. Fichero jeu_200_25_8.txt

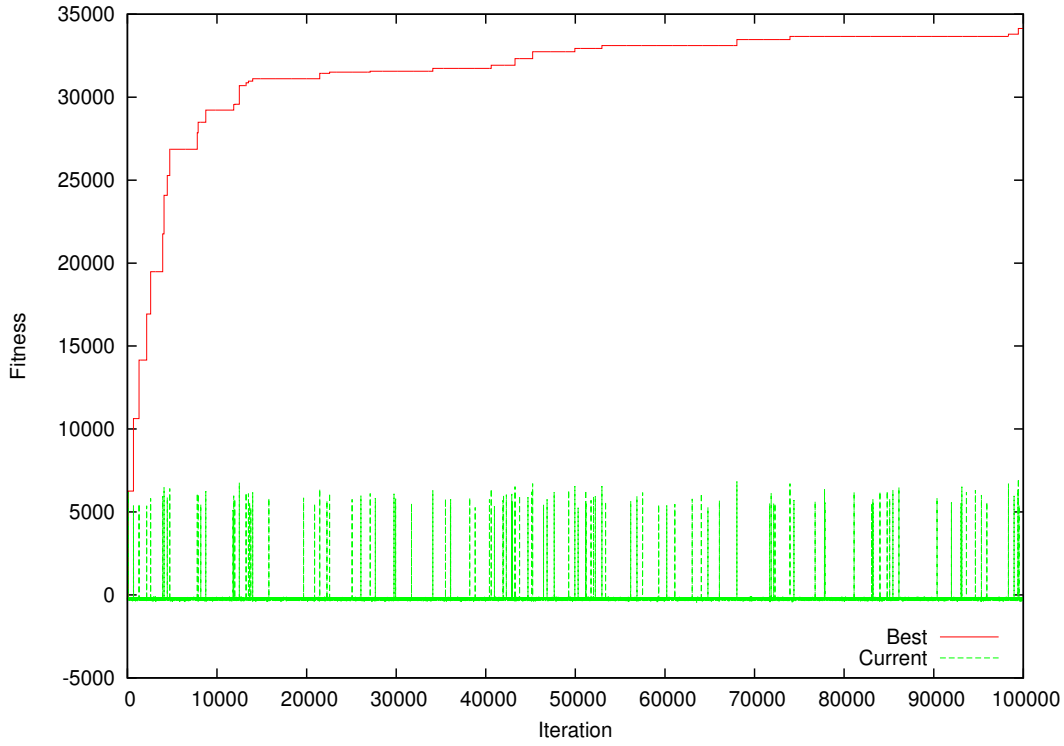
Este fichero contiene un cierto valor de objetos, que en este caso son 200 objetos a poder insertar en las mochilas.

- **Solución con 3 mochilas:** Como vemos en la Figura 5, pasa como en la Figura 1, el beneficio al principio es positivo, y además la función del mejor beneficio obtenido en cada iteración es creciente y progresiva aumentando en cada iteración. Hay similud con la Figura 1, esto quiere decir que la diferencia de beneficio en cada iteración es casi la misma, de ahí que no presente escalones tan pronunciados.



Cuadro 5: Convergencia del algoritmo con 200 objetos y 3 mochilas

- **Solución con 5 mochilas:** Como vemos en la Figura 6, el beneficio obtenido con cinco mochilas es menor que el conseguido con 3, y a diferencia con la Figura 2 esta solución presenta una función más escalonada, lo que indica que hay una mayor diferencia entre el beneficio de una interacción con el beneficio de la interacción siguiente.

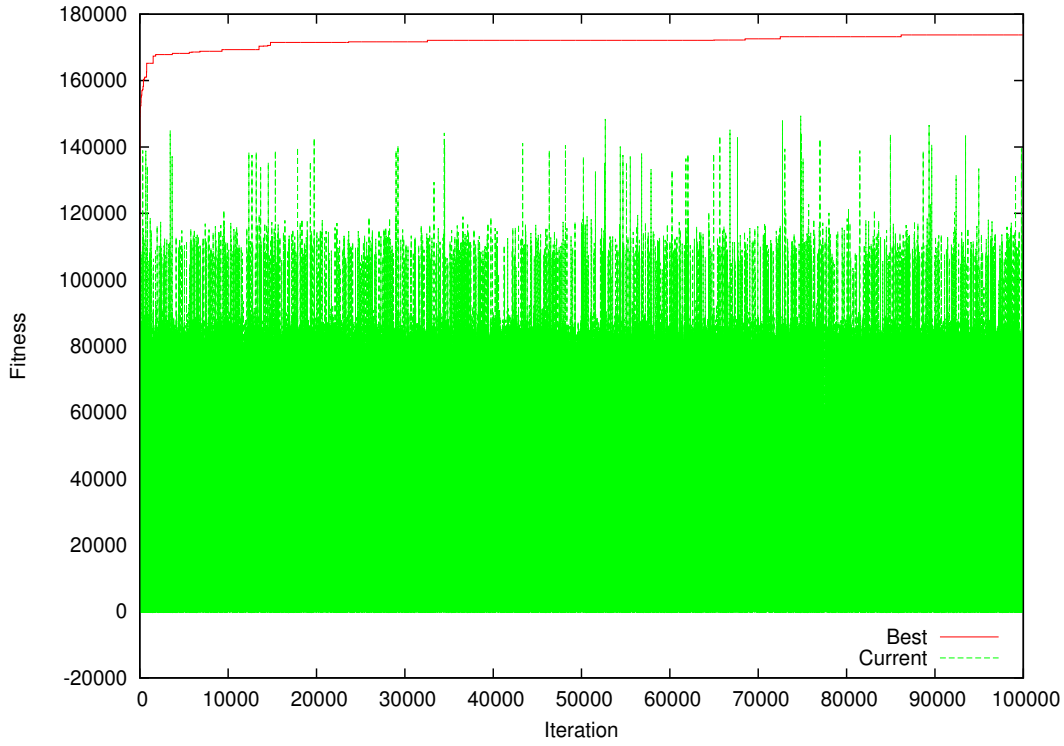


Cuadro 6: Convergencia del algoritmo con 200 objetos y 5 mochilas

6.4. Fichero jeu_200_75_5.txt

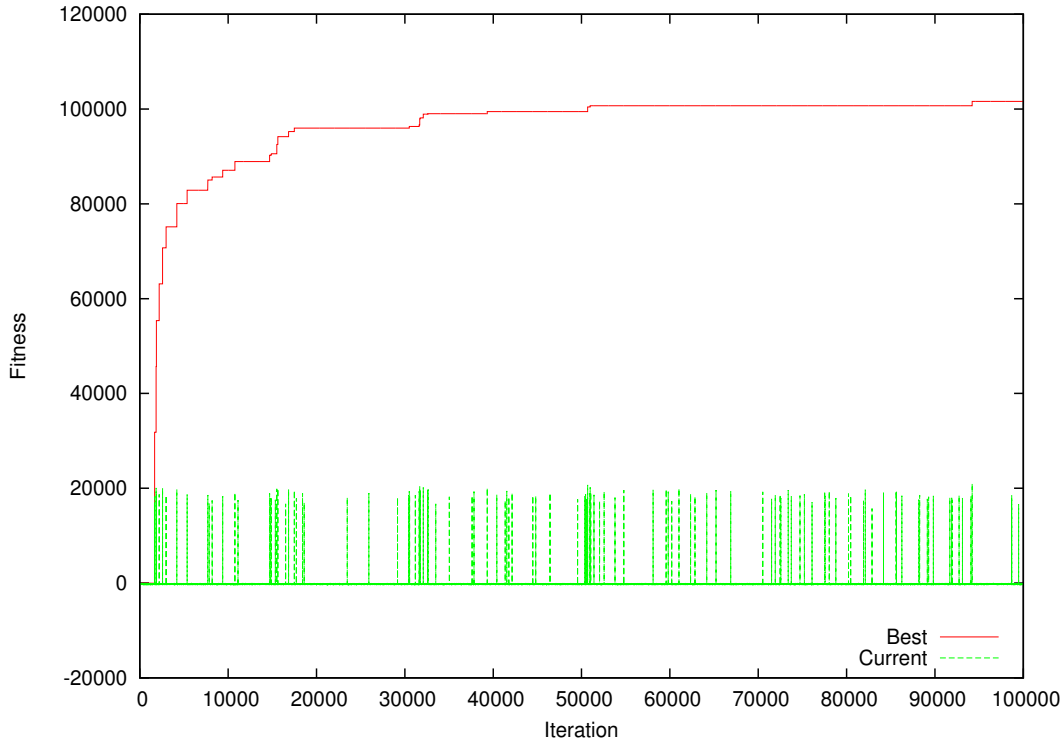
Este fichero contiene un cierto valor de objetos, que en este caso son 200 objetos a poder insertar en las mochilas.

- **Solución con 3 mochilas:** Como vemos en la Figura 7, tenemos una función creciente, alcista, indicando un beneficio positivo. Este beneficio mejora con cada iteración.



Cuadro 7: Convergencia del algoritmo con 100 objetos y 3 mochilas

- **Solución con 5 mochilas:** Como vemos en la Figura 8, el beneficio obtenido con cinco mochilas es menor que el conseguido con 3, aún así tenemos una función escalonada, muy similar a la Figura 2.



Cuadro 8: Convergencia del algoritmo con 100 objetos y 5 mochilas

6.5. Resumen general

En definitiva, como hemos podido observar en las figuras anteriores, conseguimos mejores resultado con tres mochilas que con cinco. Esto puede ser debido a la cantidad de soluciones distintas que pueden darse. Además, hemos visto que el beneficio siempre aumenta, por muy poco que sea, y esto es debido, como hemos explicado antes, a que el mejor beneficio conseguido en la siguiente iteración hace aumentar bastante la media de beneficio, creando así escalones.

7. Bibliografía

1. Explicación del problema
2. Documentación de Latex