Metaheurística

Práctica 2: Optimización Local de Soluciones

24 de marzo 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

El objetivo de esta práctica de la asignatura **Metaheurísticas** del tercer curso del Grado en Ingeniería Informática con mención en Computación, por la Universidad de Córdoba (UCO), es la de iniciar a los alumnos en el análisis y la programación de metaheurísticas, particularmente en el caso de métodos de búsqueda local. Para ello, el alumnado dispondrá de un código esqueleto, proporcionado por el profesorado, así cómo una guía sobre la realización de la práctica, especificando los pasos que el alumnado ha de seguir para realizar esta práctica.

El ejercicio que se presenta es ta basado en la búsqueda local de soluciones al problema de la mochila múltiple cuadrática. No sólo se ha de completar el código, si no que también se ha de depurar y explicar este y redactar un documento sobre lo realizado en la práctica así cómo una introducción a la práctica final de la asignatura, al final del documento.

2. Clase MQKPSolution

La clase **MQKPSolution**, es la misma que utilizamos en la práctica anterior, a la que se le añaden tres métodos nuevos:

- **getFitness**: método que devuelve el "fitness" de la solución.
- **setFitness**: método que modifica el "fitness" de la solución.
- **copy**: esta función copia la información de otra solución a la solución objeto.

Esta clase dispone de dos archivos; un *.h* y un *.cpp*. En el primero, como podemos ver, declara las cabeceras de los métodos y las variables de la clase.

```
1  /**
2   * MQKPSolution.h
3   *
4   * Fichero que define la clase MQKPSolution. Forma parte del c digo esqueleto para
        el problema de las m ltiples mochilas cuadr ticas, ofrecido para las
       pr cticas de la asignatura Metaheur sticas del Grado de Ingenier a
       Inform tica de la Universidad de C rdoba
5   *
6   * @author Carlos Garc a cgarcia@uco.es
7   */
8
9  #ifndef __MQKPSOLUTION_H__
10 #define __MQKPSOLUTION_H__
11
12 #ifndef __MQKPINSTANCE_H__
13 #include "MQKPInstance.h"
14 #else
15 class MQKPInstance;
16 #endif
17
18 /**
19  * Clase que representa una soluci n al problema
20  *
21  * Sobre la representaci n de soluciones:
22  * La representaci n de las soluciones ser  un vector de n mero enteros: de 1 a
       M para objetos que est n en alguna de las M mochilas y 0 para objetos que no
       est n en ninguna mochilas
23  */
24 class MQKPSolution
25 {
26   protected:
```

```
27      /* Definir las variables miembro
28       * _sol Vector de enteros que ser  la representaci n interna de la soluci n
         al problema
29       * _numObjs Entero donde se almacenar  el n mero de objetos del problema
30       * _fitness valor double que almacena la calidad de la soluci n
31       */
32      int *_sol;
33      int _numObjs;
34      double _fitness;
35
36    public:
37      /**
38       * Constructor
39       * @param[in] instance Referencia a un objeto con la informaci n de la
         instancia del problema MQKP
40       */
41      MQKPSolution(MQKPInstance &instance);
42
43      /**
44       * Destructor
45       */
46      ~MQKPSolution();
47
48      /**
49       * Funci n que asigna un objeto a la mochila indicada
50       * @param[in] object  ndice  del objeto a insertar en la mochila indicada
51       * @param[in] knapsack  ndice  de la mochila donde insertar el objeto
52       */
53      void putObjectIn(int object, int knapsack);
54
55      /**
56       * Funci n que devuelve la mochila en la que est  insertado un objeto
57       * @param[in] object  ndice  del objeto consultado
58       * @return  ndice  de la mochila en la que est  insertado el objeto
59       */
60      int whereIsObject(int object);
61
62      /**
63       * Funci n que devuelve el fitness de la soluci n
64       *
65       * @return fitness de la soluci n
66       */
67      double getFitness() const;
68
69      /**
70       * Funci n que asigna el fitness de la soluci n
71       * @param[in] fitness Fitness de la soluci n
72       */
73      void setFitness(double fitness);
74
75      /**
76       * Funci n que copia la informaci n de otra soluci n
77       * @param[in] solution La soluci n de donde copiar la informaci n
78       */
79      void copy(MQKPSolution &solution);
80  };
```

Listing 1: fichero MQKPSolution.h de la clase MQKPSolution

En el fichero *MQKPSolution.cpp*, se codifican los métodos de la clase. A continuación, mostraremos una captura con el fichero en cuestión:

```
1  /*
2   * MQKPSolution.cpp
3   *
```

```
4  * Fichero que define los m todos de la clase MQKPSolution. Forma parte del
       c digo esqueleto para el problema de las m ltiples mochilas cuadr ticas,
       ofrecido para las pr cticas de la asignatura Metaheur sticas del Grado de
       Ingenier a Inform tica de la Universidad de C rdoba
5  *
6  * @author Carlos Garc a cgarcia@uco.es
7  */
8
9  #include "MQKPSolution.h"
10 #include "MQKPInstance.h"
11 #include <iostream>
12
13 using namespace std;
14
15 MQKPSolution::MQKPSolution(MQKPInstance &instance)
16 {
17    // inicializando las variables miembro. Inicialmente, todos los objetos estar n
        fuera de las mochilas ( = 0)
18    _numObjs = instance.getNumObjs();
19    _fitness = 0;
20    _sol = new int[_numObjs];
21    if (!_sol)
22    {
23       cerr << "No se ha reservado memoria correctamente para _sol" << endl;
24       exit(-1);
25    }
26    int i;
27    for (i = 0; i < _numObjs; i++)
28    {
29       _sol[i] = 0;
30    }
31 }
32
33 MQKPSolution::~MQKPSolution()
34 {
35    //
36    _numObjs = 0;
37    _fitness = 0;
38    delete[] _sol;
39 }
40
41 void MQKPSolution::putObjectIn(int object, int knapsack)
42 {
43    _sol[object] = knapsack;
44 }
45
46 int MQKPSolution::whereIsObject(int object)
47 {
48    return _sol[object];
49 }
50
51 double MQKPSolution::getFitness() const
52 {
53    return _fitness;
54 }
55
56 void MQKPSolution::setFitness(double fitness)
57 {
58    _fitness = fitness;
59 }
60
61 void MQKPSolution::copy(MQKPSolution &solution)
62 {
63
```

```
64      /* TODO
65       * 1. Copiar las asignaciones de objetos a mochilas
66       * 2. copiar el fitness
67       */
68
69      // Copy object assignation into bags
70      for (int i = 0; i < this->_numObjs; i++)
71      {
72          this->_sol[i] = solution.whereIsObject(i);
73      }
74
75      // Copy fitness
76      this->setFitness(solution.getFitness());
77  }
```

Listing 2: fichero que tienen codificados los métodos de la clase MQKPSolution

3. Clase MQKPChangeOperation

La clase **MQKPChangeOperation**, es una clase abstracta, para representar cualquier operación de modificación sobre una solución. Está formada por un fichero .h solamente y en este se encuentran el destructor de la clase y la función *void apply*.

Dicha función aplica el cambio que define el objeto sobre la solución que recibe como argumento. A continuación se mostrará una figura con una imagen del fichero:

```
1  /*
2   * MQKPChangeOperation.h
3   *
4   * Fichero que declara la clase MQKPChangeOperation. Forma parte del c digo
       esqueleto para el problema de las m ltiples mochilas cuadr ticas, ofrecido
       para las pr cticas de la asignatura Metaheur sticas del Grado de Ingenier a
       Inform tica de la Universidad de C rdoba
5   *
6   * @author Carlos Garc a cgarcia@uco.es
7   */
8
9  #ifndef __MQKPCHANGEOPERATION_H__
10 #define __MQKPCHANGEOPERATION_H__
11
12 #include "MQKPSolution.h"
13
14 /**
15  * Clase abstracta para representar cualquier operaci n de modificaci n sobre una
        soluci n.
16  */
17 class MQKPChangeOperation
18 {
19   public:
20     /**
21      * Destructor
22      */
23     virtual ~MQKPChangeOperation()
24     {
25     }
26
27     /**
28      * Funci n que aplica el cambio que define el objeto sobre la soluci n que
        recibe como argumento
29      * @param[in,out] solution Objeto soluci n sobre el que se aplicar  el cambio
30      */
```

```
31      virtual void apply(MQKPSolution &solution) = 0;
32  };
33
34  #endif
```

Listing 3: clase MQKPChangeOperation

4. Clase MQKPObjectAssignmentOperation

La clase **MQKPObjectAssignmentOperation**, está formada por dos ficheros: *MQKPObjectAssignmentOperation.h* y *MQKPObjectAssignmentOperation.cpp*. Esta clase es, a su vez, una subclase de la clase **MQKPChangeOperation**, explicada en la sección 3. Esta clase codifica una operación de asignación de un objeto a una mochila, incluyendo en esta operación de asignación, la acción de sacar dicho onjeto de la mochila en la que se encontraba.

En el fichero *MQKPObjectAssignmentOperation.h* vienen definidas las cabeceras del constructor y del destructor de la clase, también vienen definidas las cabeceras de las dos funciones de esta clase: *void apply* y *void setValues*. En este archivo se nos pide que declaremos las tres variables de tipo *protected*, que usaremos en la clase. Estas son:

- **indexObj**: variable de tipo entero que señala el objeto sobre el que se realizará la acción de meter en una mochila, sacarlo de esta o cambiarlo a otra mochila distinta.
- **indexKnapSack**: variable de tipo entero que indica el destino del objeto referenciado, siendo este una mochila (entre ellas la 'mochila 0' que indica que esta fuera de las mochilas).
- **deltaFitness**: variable de tipo "double" que indica la diferencia en el *fitness* de la solución una vez aplicado el cambio.

A continuación se mostrará una imagen que nos muestra el código del archivo anteriormente explicado, *MQKPObjectAssignmentOperation.h*:

```
1  /* MQKPObjectAssignmentOperation.h
2   *
3   * Fichero que declara la clase MQKPObjectAssignmentOperation. Forma parte del
       c digo esqueleto para el problema de las m ltiples mochilas cuadr ticas,
       ofrecido para las pr cticas de la asignatura Metaheur sticas del Grado de
       Ingenier a Inform tica de la Universidad de C rdoba
4   *
5   * @author Carlos Garc a cgarcia@uco.es
6   */
7
8  #ifndef __MQKPOBJECTASSIGNMENTOPERATION_H__
9  #define __MQKPOBJECTASSIGNMENTOPERATION_H__
10
11 #include "MQKPChangeOperation.h"
12 #include "MQKPSolution.h"
13
14 /**
15  * Clase que codifica una operaci n de asignaci n de un objeto a una mochila,
       pudiendo ser  sta  la mochila 0, es decir, sacarlo de la mochila en la que se
       encuentre
16  */
17 class MQKPObjectAssignmentOperation : public MQKPChangeOperation
18 {
19
20   protected:
21     /*
22      * TODO
```

```
23      * Crea las variables miembro de la clase seg n lo indicado en el gui n de
        pr cticas (_indexObj, _indexKnapsack, _deltaFitness)
24      */
25     int _indexKnapsack;
26     int _indexObj;
27     double _deltaFitness;
28
29   public:
30     /**
31      * Constructor
32      */
33     MQKPObjectAssignmentOperation();
34
35     /**
36      * Destructor
37      */
38     virtual ~MQKPObjectAssignmentOperation();
39
40     /**
41      * Funci n que aplica el cambio que define el propio objeto sobre la soluci n
        que recibe como argumento.
42      * @param[in, out] solution Objeto soluci n sobre el que se aplicar  el cambio
43      */
44     virtual void apply(MQKPSolution &solution);
45
46     /**
47      * Funci n que asigna los valores la operaci n
48      * @param[in] indexObject El  ndice  del objeto que se va a cambiar de mochila
49      * @param[in] indexKnapsack El  ndice  de la mochila donde se insertar  el
        objeto
50      * @param[in] deltaFitness La diferencia en fitness de aplicar dicha operaci n
        de asignaci n de un objeto a una mochila (siempre que la soluci n actual no
        se hubiese cambiado cuando se calculo dicha diferencia)
51      */
52     void setValues(int indexObject, int indexKnapsack, double deltaFitness);
53 };
54
55 #endif
```

Listing 4: archivo .h de la clase MQKPObjectAssignmentOperation

En el fichero *MQKPObjectAssignmentOperation.cpp*, se encuentran codificados el constructor y el destructor, así como las otras dos funciones de la clase.

```
1  /*
2   * MQKPObjectAssignmentOperation.cpp
3   *
4   * Fichero que define las funciones de la clase MQKPObjectAssignmentOperation.
       Forma parte del c digo esqueleto para el problema de las m ltiples mochilas
       cuadr ticas, ofrecido para las pr cticas de la asignatura Metaheur sticas
       del Grado de Ingenier a Inform tica de la Universidad de C rdoba
5   *
6   * @author Carlos Garc a cgarcia@uco.es
7   */
8
9  #include "MQKPObjectAssignmentOperation.h"
10
11 MQKPObjectAssignmentOperation::MQKPObjectAssignmentOperation()
12 {
13     this->_indexKnapsack = 0;
14     this->_indexObj = 0;
15     this->_deltaFitness = 0;
16 }
17
18 MQKPObjectAssignmentOperation::~MQKPObjectAssignmentOperation()
```

```
19 {
20 }
```

Listing 5: Constructor y destructor de la clase MQKPObjectAssignmentOperation

Las otras dos funciones son:

- **apply**: Función que aporta a la superclase, descrita en la sección 3, los datos necesarios.
- **setValues**: función que guarda los valores pasados como argumentos a las variables miembro.

Estas se pueden ver codificadas en la captura mostrada a continuación:

```
1
2  void MQKPObjectAssignmentOperation::apply(MQKPSolution &solution)
3  {
4      /* TODO
5       * 1. Asigna el objeto de  ndice  _indexObj a la mochila _indexKnapsack en
        solution
6       * 2. Actualiza el fitness de solution sum ndole _deltaFitness
7       */
8      solution.putObjectIn(this->_indexObj, this->_indexKnapsack);      //Asignamos el
        objeto
9      solution.setFitness(this->_deltaFitness + solution.getFitness()); //Asignamos
        fitness
10 }
11
12 void MQKPObjectAssignmentOperation::setValues(int indexObject,
13                                               int indexKnapsack, double deltaFitness)
14 {
15     /* TODO
16      * Guarda los valores pasados como argumentos en las variables miembro
17      */
18
19     this->_indexObj = indexObject;
20     this->_indexKnapsack = indexKnapsack;
21     this->_deltaFitness = deltaFitness;
22 }
```

Listing 6: funciones apply y setValues de la clase MQKPObjectAssignmentOperation

5. MQKPInstance

La clase **MQKPSInstance**, esta clase también fue utilizada en la práctica anterior. En esta práctica, se añadirán tres nuevos métodos:

- **randomPermutation**: este método, genera permutaciones pseudo-aleatorias.
- **getDeltaSumProfits**: en esta función, se calcula la diferencia en la suma de los beneficios si a la solución se le aplicase la asignación del objeto "indexObject" a la mochila "indexKnapsack".
- **getDeltaMaxCapacityViolation**: esta función que calcula la diferencia en la máxima violación de alguna de las capacidades de las mochilas si a la solución se le aplicase la asignación del objeto "indexObject" a la mochila "indexKnapsack".

Con método "randomPermutation", que se mostrará a continuación en una captura de pantalla, se genera una nueva permutación aleatoria.

```
1 void MQKPInstance::randomPermutation(int size, vector<int> &perm)
2 {
3
4     /** TODO
```

```
     * 1. Vac a el vector perm
     * 2. Ll nalo con la permutaci n identidad
     * 3. Rec rrelo intercambiando cada elemento con otro escogido de forma
     aleatoria.
     */

    perm.clear(); // Clear vector

    // Fill vector with permutations
    for (int i = 0; i < size; i++)
    {
        perm.push_back(i);
    }

    // Random shuffle
    int new_pos, aux;

    for (int i = 0; i < size; i++)
    {
        aux = perm[i];
        new_pos = rand() % size;
        perm[i] = perm[new_pos];
        perm[new_pos] = aux;
    }
}
```

Listing 7: funcion randomPermutation de la clase MQKPInstance

La función "getDeltaSumProfits" calcula la diferencia en la suma de los beneficios si se le añade el objeto a la mochila. Aquí está el código:

```
    _numObjs = 0;
    this->_capacities = NULL;
    this->_profits = NULL;
    this->_weights = NULL;
}

MQKPInstance::~MQKPInstance()
{
    //
    int i;
    for (i = 0; i < getNumObjs(); i++)
    {
        delete[] _profits[i];
    }
    delete[] _profits;
    delete[] _weights;
    delete[] _capacities;
    _numKnapsacks = _numObjs = 0;
}

double MQKPInstance::getMaxCapacityViolation(MQKPSolution &solution)
{
```

Listing 8: función getDeltaSumProfits de la clase MQKPInstance

La función "getDeltaMaxCapacityViolation", calcula la diferencia en la máxima violación de las capacidades de las mochilas. La codificación del método será expuesto a continuación en la siguiente imagen:

```
double MQKPInstance::getDeltaMaxCapacityViolation(MQKPSolution &solution,
                                                  int indexObject, int indexKnapsack)
{

    /** TODO
     * 1. Obten la mochila donde est  el objeto
```

```
7       * 2. Obten la m xima violaci n actual de la soluci n
8       * 3. Asigna el objeto a la nueva mochila en soluci n
9       * 4. Obten la nueva violaci n de la soluci n
10      * 5. Deshaz el cambio anterior, volviendo a poner el objeto en la mochila en la
         que estaba
11      * 6. Devuelve la diferencia (nueva violaci n - violaci n actual)
12      */
13     int aux_bag;
14     double delta_cap_violation;
15
16     aux_bag = solution.whereIsObject(indexObject);                    //
        Get the object's bag
17     delta_cap_violation = this->getMaxCapacityViolation(solution);
        // Get max violation
18     solution.putObjectIn(indexObject, indexKnapsack);                 //
        Set the object in the new bag
19     delta_cap_violation = this->getMaxCapacityViolation(solution) -
        delta_cap_violation; // Get the difference
20     solution.putObjectIn(indexObject, aux_bag);                       //
        Set the object again in the old bag
21
22     return delta_cap_violation;
23 }
```

Listing 9: función getDeltaMaxCapacityViolation de la clase MQKPInstance

La clase "MQKPEvaluator" ya fue usada en la práctica anterior.En la nueva versión, se añaden nuevas variables y métodos. Las variable nueva es:

- **numEvaluations**: variable dónde se contabiliza el número de soluciones que se evalúan a través de "computeFitness" o "computeDeltaFitness".

```
1   protected:
2     /**
3      * Variable donde se contabiliza el n mero de soluciones que se eval an a
        trav s de  computeFitness o computeDeltaFitness
4      */
5     static unsigned _numEvaluations;
```

Listing 10: variable numEvaluations de la clase MQKPEvaluator

Las tres funciones nuevas implementadas en el código de la clase son:

- **computeDeltaFitness**: función que calcula la diferencia en fitness si a la solución que se le pasa como argumento se le aplicase una operación de asignación de un objeto a una mochila determinada.
- **resetNumEvaluations**: función que resetea la variable interna que contabiliza el número de evaluaciones (pone a cero la variable numEvaluations).
- **getNumEvaluations**: función que devuelve el número de veces que se ha evaluado alguna solución.

Tras haber explicado las modificaciones que se implementaran, en cuánto a métodos se refiere en esta clase, se mostrarán las capturas con el código de estos métodos (dos de los tres métodos están desarrollados en el archivo .cpp, mientras que el restante se encuentra codificado en el fichero .h).

```
1     static unsigned getNumEvaluations()
2     {
3        return _numEvaluations;
4     }
```

Listing 11: función getNumEvaluations de la clase MQKPEvaluator

```
1  double MQKPEvaluator::computeDeltaFitness(MQKPInstance &instance,
2                                            MQKPSolution &solution, int indexObject, int
       indexKnapsack)
3  {
4
5      _numEvaluations++;
6
7      /**
8       * TODO
9       * Dado que el fitness depende de si se violan las capacidades de alguna mochila
         o no,
10      * deben calcularse las violaciones actuales y las posibles nuevas, adem s del
        posible
11      * cambio en la suma de beneficios
12      *
13      * 1. Obten la m xima violaci n actual
14      * 2. Invoca a MQKPInstance.getDeltaMaxCapacityViolation para que devuelva como
        se modifica la m xima violaci n tras la operaci n
15      * 3. Suma las medidas anteriores para obtener la m xima violaci n si se
        aplica la operaci n
16      * 4. Obten la suma de beneficios actual
17      * 5. Invoca a MQKPInstance.getDeltaSumProfits para que devuelva c mo se
        modifica la suma de beneficios si se aplica la operaci n
18      * 6. Suma las dos medidas anteriores para obtener la suma de beneficios si se
        aplica la operaci n
19      *
20      * Finalmente
21      * Si las violaciones actuales y nuevas son positivas, devuelve el negativo de
        deltaMaxCapacityViolation
22      * Si ambas violaciones son 0, devuelve el deltaSumProfits
23      * Si s lo la violaci n actual es positiva y la nueva es 0, devuelve la suma
        de newSumProfits + el negativo de deltaMaxCapacityViolation
24      * Si s lo la violaci n nueva es positiva, devuelve el negativo de (la suma de
         la nueva violaci n + la nueva violaci n de capacidades)
25      */
26
27     double maxViolation = instance.getMaxCapacityViolation(solution);
                             //Obtenemos la m xima violacion actual
28     double deltaMaxCapacityViolation = instance.getDeltaMaxCapacityViolation(
        solution, indexObject, indexKnapsack); //Obtenemos la capacidad
29     double MaxViolationOperation = maxViolation + deltaMaxCapacityViolation;
                               //Suma de las medidas anteriores
30     double actualSumProfits = instance.getSumProfits(solution);
                             //Obtenemos la suma de beneficios actual
31     double deltaSumProfits = instance.getDeltaSumProfits(solution, indexObject,
        indexKnapsack);                    //Obtenemos la suma de beneficios
32     double newSumProfits = actualSumProfits + deltaSumProfits;
                             //Suma de las medidas anteriores
33
34     if (maxViolation > 0 and MaxViolationOperation > 0)
35     {
36        return (deltaMaxCapacityViolation * -1);
37     }
38
39     else if (maxViolation == 0 and MaxViolationOperation == 0)
40     {
41
42        return deltaSumProfits;
43     }
44
45     else if (maxViolation > 0 and MaxViolationOperation <= 0)
46     {
47
48        return (newSumProfits + (deltaMaxCapacityViolation * -1));
```

```
49      }
50      return ((actualSumProfits * -1) + (MaxViolationOperation * -1));
51  }
52
53  void MQKPEvaluator::resetNumEvaluations()
54  {
55      _numEvaluations = 0;
56  }
```

Listing 12: funciones computeDeltaFitness y resetNumEvaluations de la clase MQKPEvaluator

6. MQKPSimpleFirstImprovementNO

La clase **MQKPSimpleFirstImprovementNO**, es una subclase de la clase *MQKPNeighExplorer*, que se encarga de explorar el vecindario de una solución dada, devolviendo la primera operación de asignación de un objeto a una mochila que encuentre que mejore la calidad de la solución recibida. En caso de que no exista ninguna operación que mejore la calidad de la solución recibida, entonces devolverá *false*.

La clase **MQKPNeighExplorer**, es una clase abstracta que define las operaciones de cualquier operador que explora la vecindad de una solución dada. Es la súper clase de la clase **MQKPSimpleBestImprovementNO**.

De la clase **MQKPSimpleFirstImprovementNO** se ha de completar la función *findOperation*, proporcionada en el fichero *MQKPSimpleFirstImprovementNO.cpp* en el código esqueleto de la práctica. A continuación mostraremos los pasos que se han seguido para completar dicha función de la clase:

```
1  bool MQKPSimpleFirstImprovementNO::findOperation(MQKPInstance &instance,
       MQKPSolution &solution, MQKPChangeOperation &operation)
2  {
3
4      MQKPObjectAssignmentOperation *oaOperation = dynamic_cast<
       MQKPObjectAssignmentOperation *>(&operation);
5      if (oaOperation == NULL)
6      {
7          cerr << "MQKPSimpleFirstImprovementNO::findOperation recibi  un objeto
       operation que no es de la clase MQKPObjectAssignmentOperation" << endl;
8          exit(1);
9      }
10
11     //Crear una permutaci n de los  ndices  de los objetos e inicializar algunas
       variables
12     vector<int> perm;
13     int numObjs = instance.getNumObjs();
14     int numKnapsacks = instance.getNumKnapsacks();
15     MQKPInstance::randomPermutation(numObjs, perm);
16
17     /* TODO
18      * 1. Para todo objeto del problema (accediendo en el orden indicado en perm)
19      *    a. Para toda mochila del problema (Nota: no te olvides de ninguna)
20      *       i. Obtener el deltaFitness de asignar dicho objeto a dicha mochila en
       solution
21      *
22      *       ii. Si el deltaFitness es positivo
23      *          . actualizar los valores de la operaci n en oaOperation
24      *          . Salir devolviendo true
25      *
26      * 2. Si se llega a este punto, no se encontr  ning n deltaPositivo y se
       devuelve false
```

```
27      *
28      */
29
30      MQKPEvaluator evaluator;
31      double evalFitness;
32
33      // Recorremos todas los objetos de las mochilas
34      for (int i = 0; i < numObjs; i++)
35      {
36          for (int j = 0; j <= numKnapsacks; j++)
37          {
38              evalFitness = evaluator.computeDeltaFitness(instance, solution, perm[i], j
        );
39              if (evalFitness > 0)
40              {
41                  oaOperation->setValues(perm[i], j, evalFitness);
42
43                  return true;
44              }
45          }
46      }
47
48      return false;
49  }
```

Listing 13: función findOperation de la clase MQKPSimpleFirstImprovementNO

7. MQKPSimpleBestImprovementNO

La clase **MQKPSimpleBestImprovementNO**, es una subclase de la clase *MQKPNeighExplorer*, que se encarga de explorar el vecindario de una solución dada, devolviendo *true* y la operación de asignación de un objeto que mejora en mayor medida la calidad de la solución recibida. En caso de que no exista ninguna operación que mejore la calidad de la solución recibida, entonces devolverá *false* y la operación que produzca el menor detrimento de esta.

La clase **MQKPNeighExplorer**, es una clase abstracta que define las operaciones de cualquier operador que explora la vecindad de una solución dada. Es la súper clase de la clase **MQKPSimpleBestImprovementNO**.

De la clase **MQKPSimpleBestImprovementNO** se ha de completar la función *findOperation*, proporcionada en el fichero *MQKPSimpleBestImprovementNO.cpp* en el código esqueleto de la práctica. A continuación mostraremos los pasos que se han seguido para completar dicha función de la clase.

```
1   bool MQKPSimpleBestImprovementNO::findOperation(MQKPInstance &instance,
2                                                   MQKPSolution &solution, MQKPChangeOperation &
        operation)
3   {
4
5       MQKPObjectAssignmentOperation *oaOperation = dynamic_cast<
        MQKPObjectAssignmentOperation *>(&operation);
6       if (oaOperation == NULL)
7       {
8           cerr << "MQKPSimpleBestImprovementNO::findOperation recibi  un objeto
        operation que no es de la clase MQKPObjectAssignmentOperation" << endl;
9           exit(1);
10      }
11
12      //Crear una permutaci n de los  ndices  de los objetos e inicializar algunas
        variables
```

```
13      vector<int> perm;
14      int numObjs = instance.getNumObjs();
15      MQKPInstance::randomPermutation(numObjs, perm);
16      int numKnapsacks = instance.getNumKnapsacks();
17      bool initialised = false;
18      double bestDeltaFitness = 0;
19
20      /* TODO
21       * 1. Para todo objeto del problema (accediendo en el orden indicado en perm)
22       *   a. Para toda mochila del problema (Nota: no te olvides de ninguna)
23       *     i. Obtener el deltaFitness de asignar dicho objeto a dicha mochila en
         solution
24       *
25       *     ii. Si el deltaFitness es mejor que bestDeltaFitness o si no se hab a
         inicializado bestDeltaFitness (initialised == false)
26       *       . Poner initialised a true
27       *       . actualizar bestDeltaFitness
28       *       . actualizar los valores de la operaci n en oaOperation
29       *
30       * 2. Finalmente, devolver true si bestDeltaFitness es positivo y falso en otro
         caso
31       *
32       */
33
34      MQKPEvaluator evaluator;
35      double fitness_ev = 0.0;
36
37      for (int i = 0; i < numObjs; i++)
38      {
39          for (int j = 0; j <= numKnapsacks; j++)
40          {
41              fitness_ev = evaluator.computeDeltaFitness(instance, solution, perm[i], j)
         ;
42              if (fitness_ev > bestDeltaFitness || initialised == false)
43              {
44                  initialised = true;
45                  bestDeltaFitness = fitness_ev;
46                  oaOperation->setValues(perm[i], j, bestDeltaFitness);
47              }
48          }
49      }
50
51      if (bestDeltaFitness > 0)
52      {
53          return true;
54      }
55      else
56      {
57          return false;
58      }
59  }
```

Listing 14: función findOperation de la clase MQKPSimpleBestImprovementNO

8. MQKPLocalSearch

La clase **MQKPLocalSearch**, recibiendo una solución al problema, la optimiza localmente e iterativamente hasta alcanzar un óptimo local. Esta clase viene implementada en el código esqueleto de la práctica, que esta disponible en la plataforma Moodle de la asignatura. La clase viene incompleta, ya que hay que codificar una serie de variables y métodos de esta clase, especificados en la guía de la práctica.

Las variable que viene declarada en el fichero *MQKPLocalSearch.h* es la siguiente:

- **results**: vector de doubles dónde se almacena la calidad de la última solución aceptada. A continuación mostraremos el trozo de código del archivo en el que viene declarada.

```
class MQKPLocalSearch
{

    /**
     * vector de doubles donde almacena la calidad de la  ltima  soluci n aceptada
     */
    vector<double> _results;
```

Listing 15: variable results de la clase MQKPLocalSearch

Las función a completar es:

- **optimise**: función que optimiza una solución aplicando repetidamente la exploración de su vecindario hasta alcanzar un óptimo local. Los pasos seguidos para la codificación de esta función pueden ser observados en la siguiente imagen.

```
void MQKPLocalSearch::optimise(MQKPInstance &instance,
                               MQKPNeighExplorer &explorer, MQKPSolution &solution)
{

    _results.clear();
    _results.push_back(solution.getFitness());
    MQKPObjectAssignmentOperation operation;

    /** TODO
     * 1. Aplica una vez la exploraci n del vecindario y almacena si se ha
     conseguido o no mejorar la soluci n
     *
     * 2. Mientras se haya conseguido mejorar la soluci n
     *    a. Aplica el cambio indicado en la exploraci n anterior
     *    b. Almacena en _results el nuevo fitness de la soluci n
     *    c. Aplica una nueva exploraci n del vecindario
     */

    // If a better solution is found within the neighbour
    while (explorer.findOperation(instance, solution, operation))
    {
        operation.apply(solution);                  // Apply changes
        _results.push_back(solution.getFitness()); // Set the new fitness
    }
}
```

Listing 16: función optimise de la clase MQKPLocalSearch

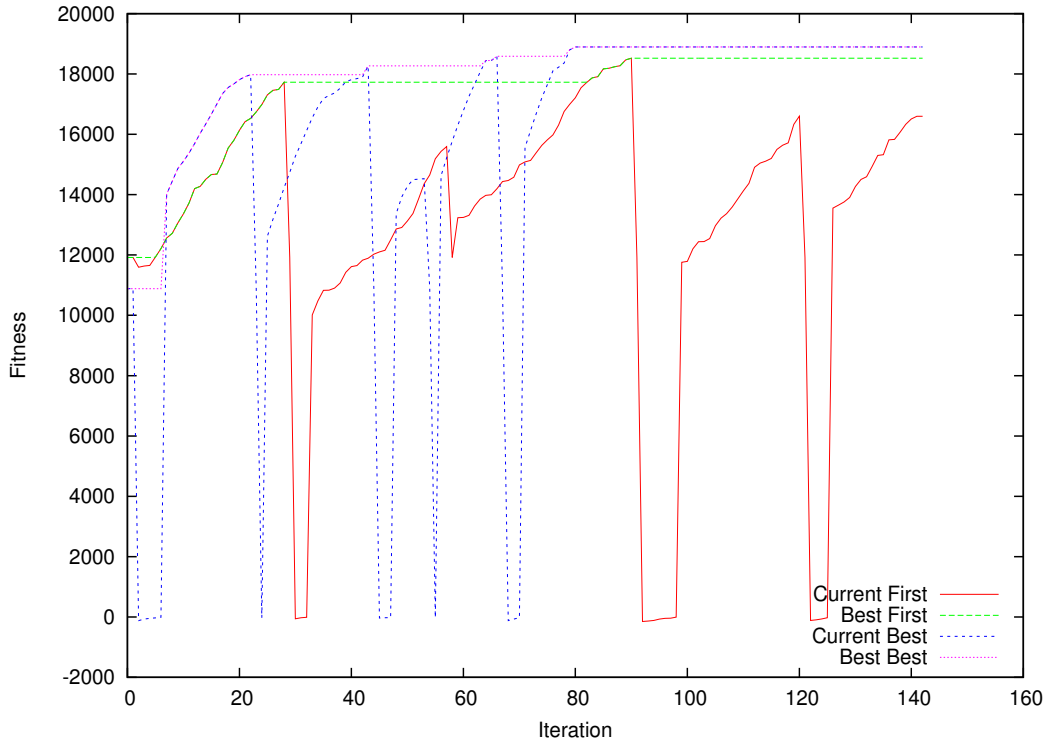
9. Experimentos

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.

9.1. Fichero jeu_100_25_4.txt

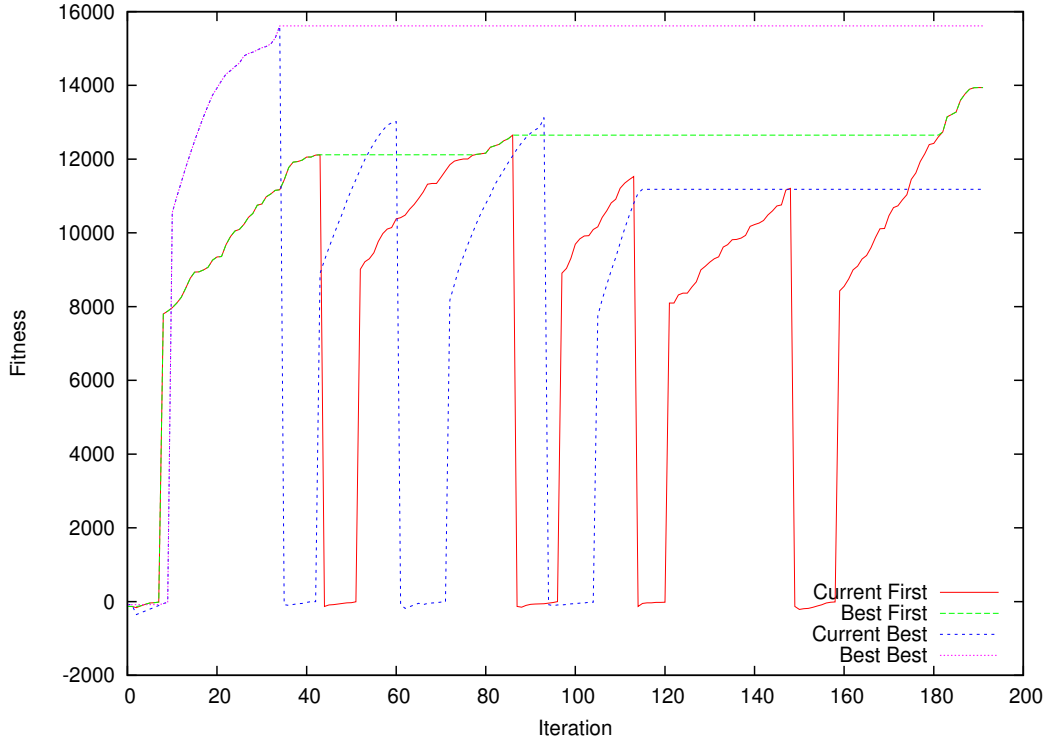
- **Solución con 3 mochilas:** Como vemos en la Figura 1, se puede observar los resultados tras ejecutar el programa con 100 objetos y 3 mochilas. Podemos observar que la gráfica es

correcta ya que el *Best First* y el *Current First* son iguales las primeras iteracciones, para luego con más iteracciones mejorar. Se puede observar también en la gráfica el momento de no mejora y el posterior intento de nuevo.



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

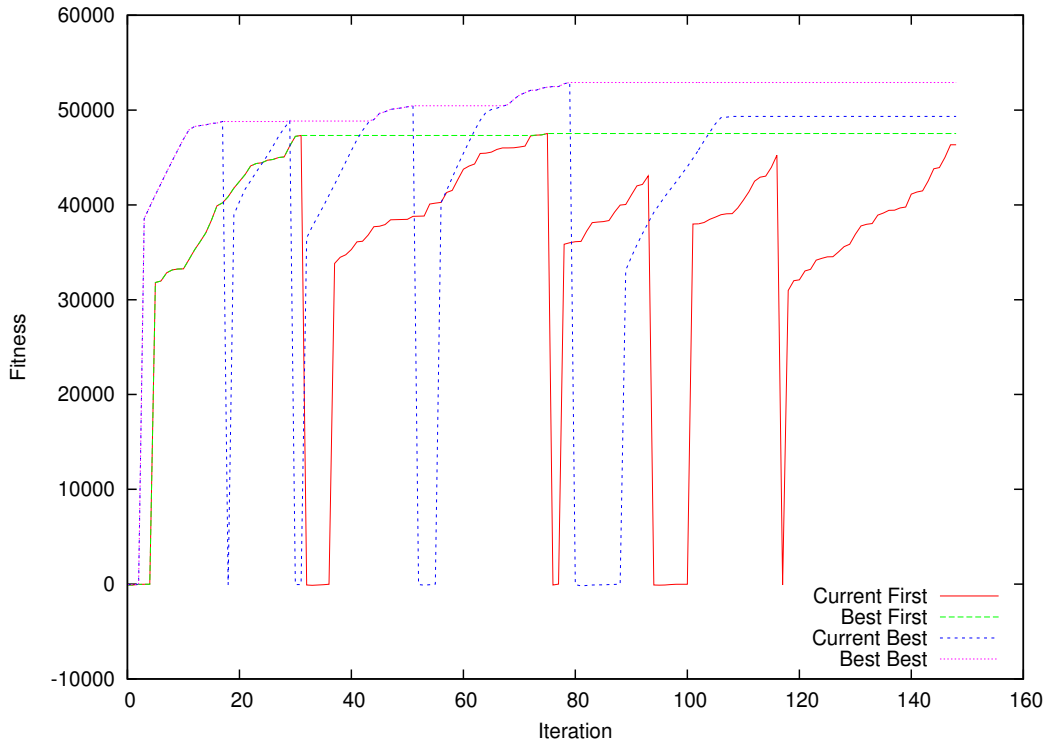
- **Solución con 5 mochilas:** A diferencia del resultado obtenido con 3 mochilas, al utilizar 5 el resultado obtenido es peor. También podemos observar una mayor diferencia de resultados entre el *Current* y el *Best*, cosa que no ocurría con la gráfica anterior.



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

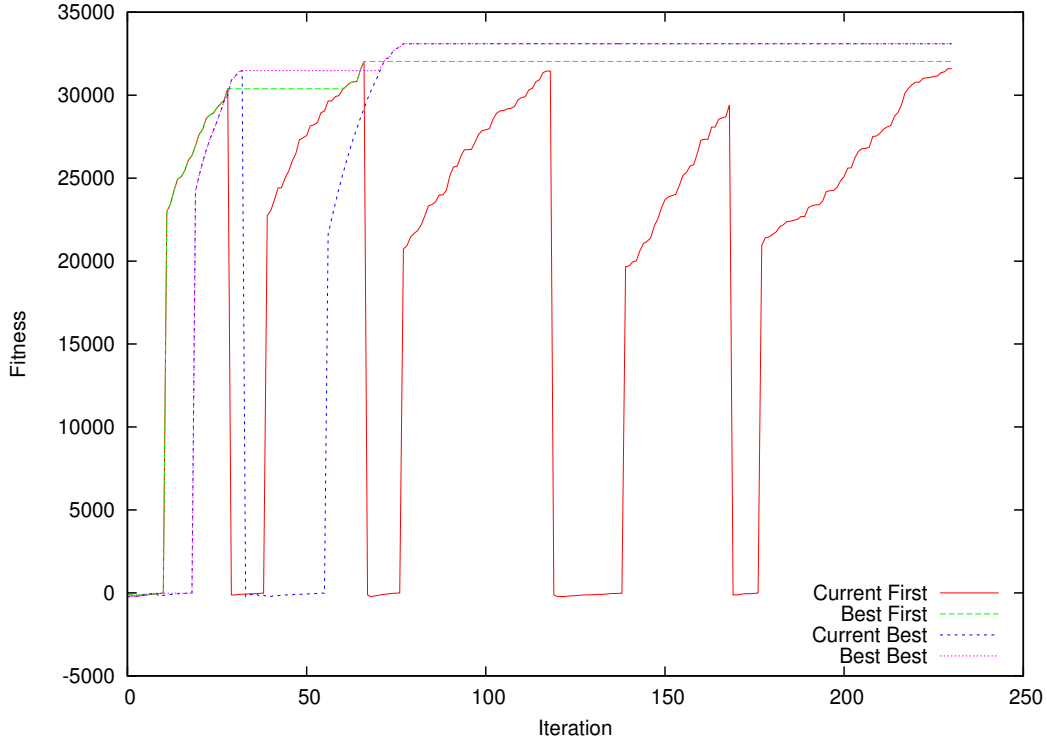
9.2. Fichero jeu_100_75_2.txt

- **Solución con 3 mochilas:** Como podemos observar en la Figura 3, los resultados obtenidos son muy similares a los obtenidos en la Figura 1. Además, como en los dos casos anteriores, el valor de *Fitness* se todas las variables comienzan en el mismo punto. Junto con esto, podemos observar cómo los valores de las variables *Current* comienzan a aumentar hasta llegado un punto en el que encuentran un óptimo local en el cuál el programa comienza desde cero a buscar otra solución. Finalmente, los valores máximos conseguidos con esta configuración son muchísimo mejores que los conseguidos en los dos casos anteiores.



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

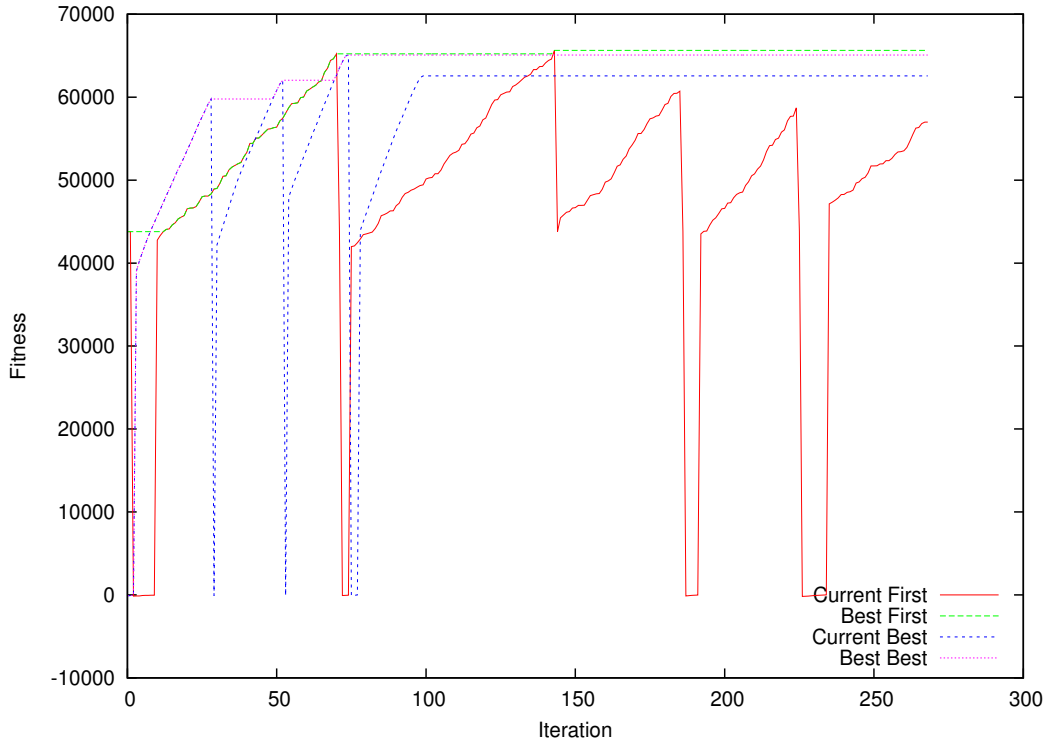
- **Solución con 5 mochilas:** Seguidamente, en la Figura 4, podremos observar los resultados obtenidos utilizando una configuración de cinco mochilas. Como observamos, los resultados obtenidos son un poco peores que los obtenidos anteriormente con tres mochilas. Igualmente, pasa como antes, los valores de las variables *Current* aumentan hasta tal punto en el que el algoritmo encuentra un óptimo local en el que las soluciones obtenidas seguidamente, no mejoran al padre, con lo que el algoritmo vuelve a un valor *Fitness* de cero, explorando de nuevo todas las soluciones hasta encontrar la solución óptima.



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

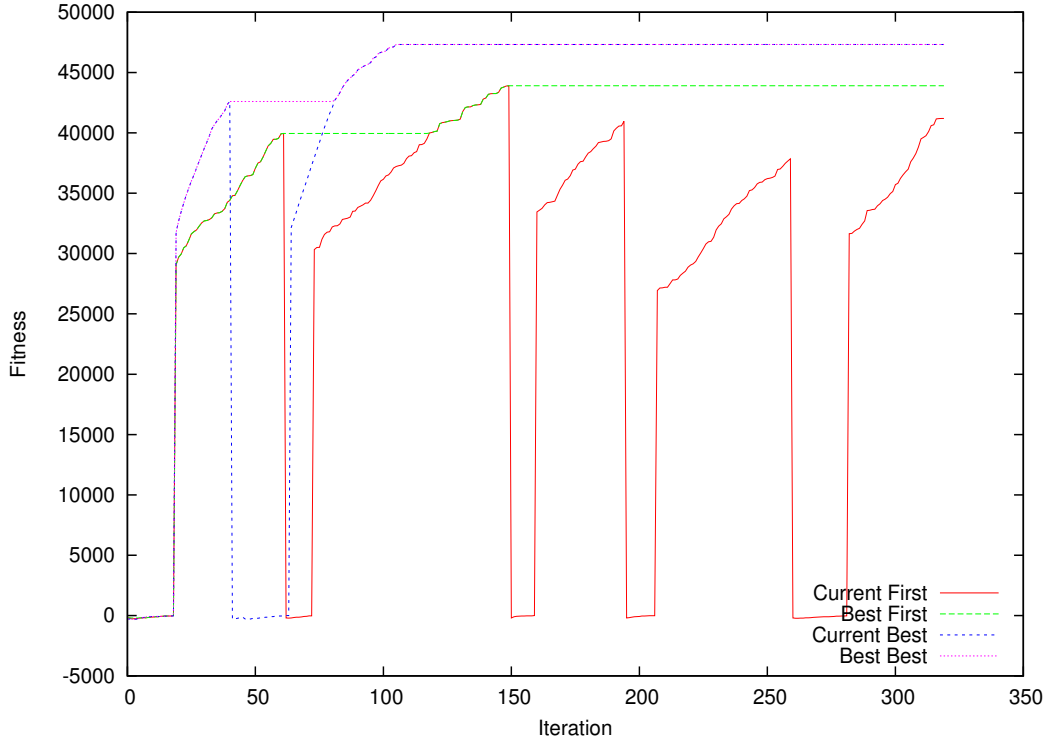
9.3. Fichero jeu_200_25_8.txt

- **Solución con 3 mochilas:** Como vemos en la Figura 5, los resultados obtenidos para este conjunto de datos es muy diferente a los obtenidos hasta ahora. Como vemos, el valor de *Fitness* de la variable *Current First* comienza a aumentar hasta llegar, a lo que, gracias a la Figura, denominaríamos como óptimo global. A diferencia de esta variable, *Current Best* comienza con a encontrar soluciones locales, con lo que el algoritmo vuelve a comenzar dezde cero haciendo que este valor vaya mejorando con el avance de las iteraciones. Finalmente, los valores de las variables comienzan a estabilizarse ya que el programa no es capaz de encontrar una mejor solución.



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

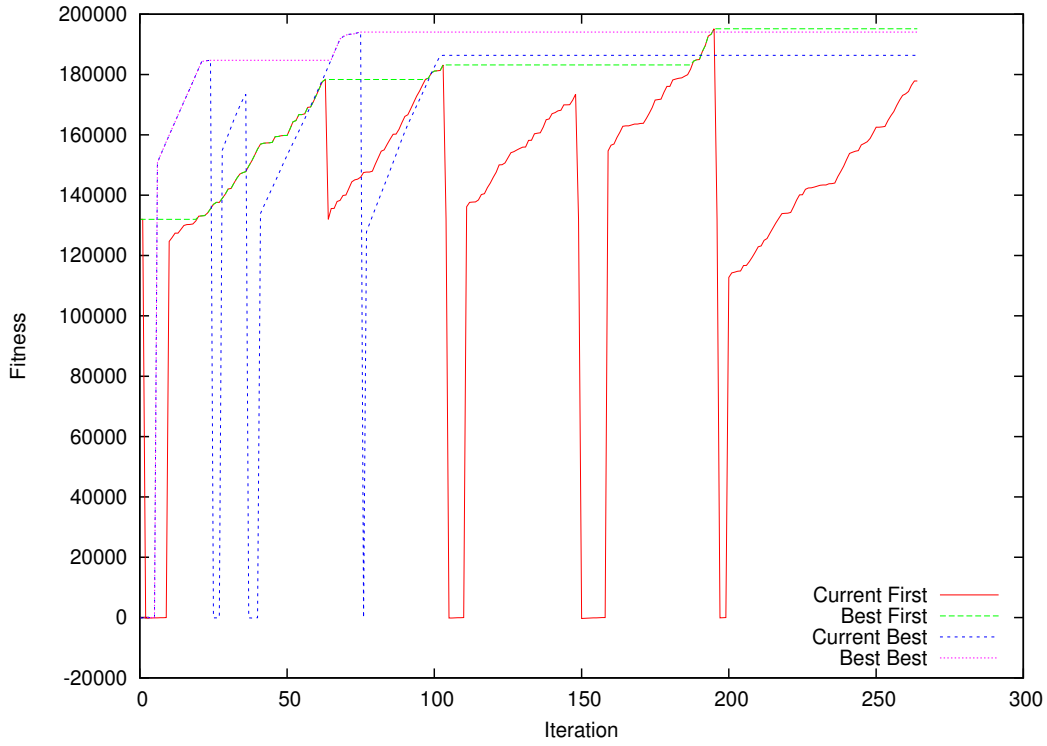
- **Solución con 5 mochilas:** Como podemos observar en la Figura 6, los resultados obtenidos son muy parecido a los obtenidos con tres mochilas. En esta ocasión, el valor de *Fitness*, de la variable *Current First* no encuentra un óptimo global, sino que encuentra un óptimo local, teniendo que volver a comenzar desde cero. Finalmente, como nota tenemos que decir que los valores de *Fitness* que obtenemos con esta configuración son mucho peores que los obtenidos utilizando tres mochilas.



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

9.4. Fichero jeu_200_75_5.txt

- **Solución con 3 mochilas:** Seguidamente, en la Figura 7 podemos observar cómo los resultados obtenidos son los mejores hasta ahora. Además, como hasta ahora, los valores obtenidos al principio son los correspondientes a los resultados de óptimos locales hasta que, en la iteración 200 encuentra el óptimo global de la solución, aumentando el valor de la variable *Best Best*. Como obsrvación, vemos que el valor de la variable *Current Best* encuentra la mejor solución en las primeras 100 iteraciones, de ahí la forma de las funciones en la Figura 7.



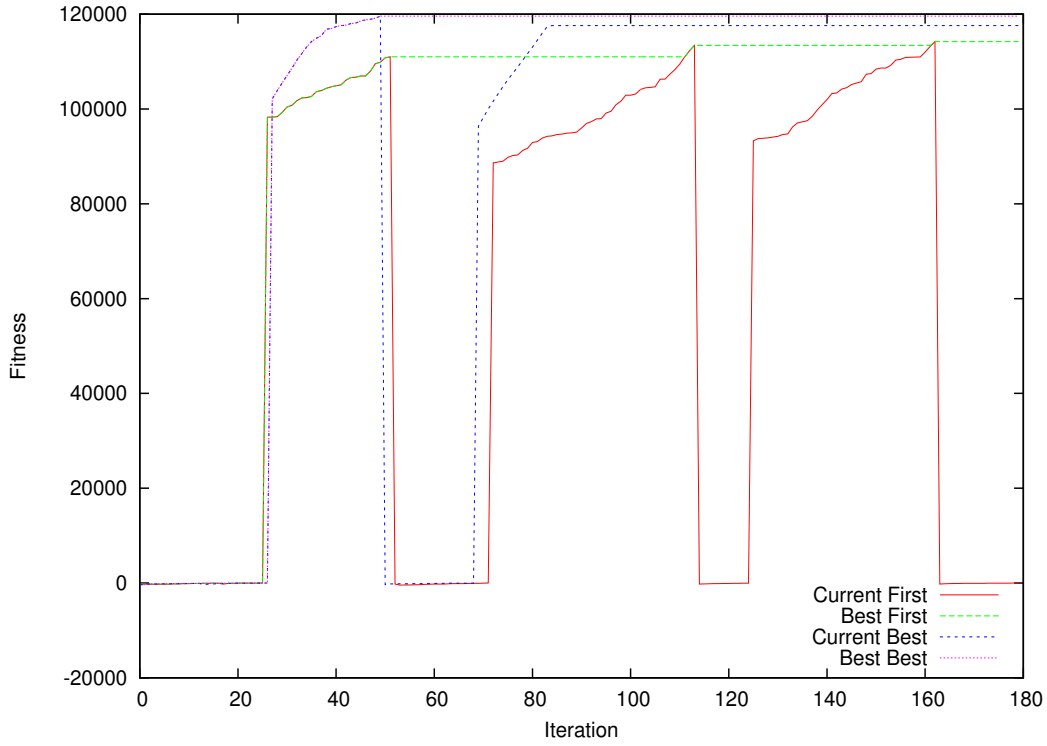
Cuadro 7: Convergencia del algoritmo con 100 objetos y 3 mochilas

- **Solución con 5 mochilas:** Como vemos en la siguiente Figura 8, podemos observar que estos resultados son distintos a la figura anterior, Figura 7. Las distancias de iteraciones es mucho mayor que las vistas hasta hora para obtener una solución mejor que las anteriores. En esta solución también podemos observar que una "simetría"de los valores *Current First*, mejorando con el paso de iteracciones y así incrementando el valor de *Best First*. Por último podemos observar que la mejor solución se ha obtenido muy temprana y no ha precisado muchas iteracciones.

10. Trabajo Final

El trabajo elegido ha sido el de codificar metahurísticas para el problema conocido como *"Multidimensional 2-way Number Partitioning Problem (M2-NPP)"*. El objetivo de dicho trabajo es el de distribuir N vectores de dimensión D en dos conjuntos, de manera que la máxima diferencia entre los conjuntos sea mínima.

Para ello vamos a crear varias instancias con N números de vectores y con distintas dimensiones de tamaño. Con el fin de ir probando las metaheurísticas con las diferentes instancias para así una vez obtenido los resultados poder redactar una conclusión y ver si los resultados varían.



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