



UNIVERSIDAD DE CÓRDOBA

ESCUELA POLITÉCNICA  
SUPERIOR DE CÓRDOBA  
Universidad de Córdoba



# Metaheurística

---

## Práctica 3: Metaheurísticas basadas en Trayectorias

7 de abril de 2019

### Resumen

In the present document we will use C++ language in order to solve the *Multiple Quadratic Knapsack Problem*. Here, we will explain how we have solved this problem in different ways, such as local search, iterated greedy, tabu search and simulated annealing. At the end, we will show all the result that we have reached using these metaheuristics.

# Índice

1. MQKPEvaluator::compare	3
2. MQKPStopCondition	3
3. MQKPSimulatedAnnealing::accept	4
4. MQKPSimulatedAnnealing::run	4
5. MQKPTabuSearch::run	5
6. MQKPGrasp::chooseOperation	7
7. MQKPGrasp::buildInitialSolution	7
8. MQKPGrasp::run	8
9. MQKPIteratedGreedy::chooseOperation	9
10. MQKPIteratedGreedy::rebuild	10
11. MQKPIteratedGreedy::destroy	10
12. MQKPIteratedGreedy::run	11
13. Resultados	13
13.1. Gráficas . . . . .	13
13.1.1. Conjunto de datos jeu_100_25_4: . . . . .	13
13.1.2. Conjunto de datos jeu_100_75_2: . . . . .	15
13.1.3. Conjunto de datos jeu_200_25_8: . . . . .	18
13.1.4. Conjunto de datos jeu_200_75_5: . . . . .	20
13.2. Conclusiones . . . . .	23
14. Problema seleccionado	24

## 1. MQKPEvaluator::compare

La clase **MQKPEvaluator**, que ya fue utilizada en las prácticas anteriores, posee un nuevo método, *compare*. Es una función que realiza la comparación entre 2 valores de *Fitness*, para decidir si son distintos y si es así, cuál es mejor de los dos.

Los dos valores de *Fitness* a comparar, se introducen como parámetros de entrada de la función. Dicha comparación se puede apreciar en el Listing 1.

```
1  static double compare(double f1 , double f2)
2  {
3      if ( f1 > f2 )
4      {
5          return 1;
6      }
7      else if ( f1 < f2 )
8      {
9          return -1;
10     }
11     else
12     {
13         return 0;
14     }
15 }
```

Listing 1: Método *compare* de la clase *MQKPEvaluator*

## 2. MQKPStopCondition

La clase **MQKPStopCondition**, tiene solo un fichero *.h*, en el que se realizan todos los métodos y, se nos pide que resetemos la variable miembro *\_time* en dos ocasiones dentro del texto.

La primera aparición se encuentra en el constructor de la clase, en el que debemos resetear la variable con la función *reset()*, tal y como podemos observar en el Listing 2.

```
1  MQKPStopCondition()
2  {
3      _maxEvaluations = 0;
4      _maxIterations = 0;
5      _maxTime = 0;
6      _numIterations = 0;
7      _time.reset();
8  }
```

Listing 2: Constructor de la clase *MQKPStopCondition*

La segunda aparición, se encuentra más abajo en el código de la clase, al inicio del método *setConditions*. En el Listing 3 mostramos cómo hemos reseteado la variable (se ha hecho de la misma manera que en la primera aparición).

```
1  void setConditions(unsigned maxEvaluations , unsigned maxIterations , double
    maxTime)
2  {
3      _maxEvaluations = maxEvaluations;
4      _maxIterations = maxIterations;
5      _maxTime = maxTime;
6      _time.reset();
7  }
```

Listing 3: Método *setConditions* de la clase *MQKPStopCondition*

### 3. MQKPSimulatedAnnealing::accept

La clase **MQKPSimulatedAnnealing** está formada por un fichero *.cpp* y un fichero *.h*. Aquí hay dos funciones que se han de completar en el fichero *.cpp*: la función *accept* y la función *run*.

La función *run* la completaremos en la siguiente sección del documento, a continuación, pasaremos a completar y explicar cómo lo hemos hecho el método *accept* de esta clase.

En este método calcularemos la probabilidad de aceptar el cambio, éste se calcula de la siguiente forma:

- Primero, hacemos la diferencia del *Fitness*.
- A continuación la dividimos por la temperatura.
- Por último, calculamos la exponencial de esto.

Lo que hemos de hacer es generar un número aleatorio entre 0 y 1 y compararlo con la probabilidad de aceptación. Si el número generado es menor que dicha probabilidad la función devolverá *True*. También a la hora de calcular el *deltaFitness* se ha de tener en cuenta si el problema es de minimización o no, ya que el signo influye en función del tipo de problema. A continuación, en el Listing 4 podremos observar la solución codificada.

```
1 bool MQKPSimulatedAnnealing::accept(double deltaFitness)
2 {
3
4     double auxDeltaFitness = deltaFitness;
5
6     if (MQKPEvaluator::isToBeMinimised())
7     {
8         auxDeltaFitness = (-deltaFitness);
9     }
10
11     double prob = exp(auxDeltaFitness / this->_T);
12     double randSample = (((double)rand()) / RAND_MAX);
13     bool isAccepted = randSample < prob;
14
15     return isAccepted;
16 }
```

Listing 4: Método *accept* de la clase *MQKPSimulatedAnnealing*

### 4. MQKPSimulatedAnnealing::run

Una vez completado el método *accept* de la clase *MQKPSimulatedAnnealing*, pasamos a completar el método *run*.

La función *run* de la clase *MQKPSimulatedAnnealing*, ejecutamos la metaheurística hasta alcanzar la condición de parada. Dicha solución es observable en el Listing 5.

```
1 void MQKPSimulatedAnnealing::run(MQKPStopCondition &stopCondition)
2 {
3
4     if (_T <= 0 || _annealingFactor <= 0)
5     {
6         cerr << "Simulated annealing has not been initialised" << endl;
7         exit(-1);
8     }
9
10    if (_solution == NULL)
```

```

11 {
12     cerr << "Simulated annealing has not been given an initial solution" << endl;
13     exit(-1);
14 }
15
16 _results.clear();
17 unsigned numObjs = _instance->getNumObjs();
18 unsigned numKnapsacks = _instance->getNumKnapsacks();
19 unsigned numIterations = 0;
20
21 while (stopCondition.reached() == false)
22 {
23     int indexObject = rand() % numObjs;
24     int indexKnapsack = rand() % (numKnapsacks + 1);
25     double deltaFitness = MQKPEvaluator::computeDeltaFitness(*_instance, *
        _solution, indexObject, indexKnapsack);
26
27     if (accept(deltaFitness))
28     {
29         _solution->putObjectIn(indexObject, indexKnapsack);
30         _solution->setFitness(_solution->getFitness() + deltaFitness);
31
32         if (MQKPEvaluator::compare(_solution->getFitness(), _bestSolution->
            getFitness()) > 0)
33         {
34             _bestSolution->copy(*_solution);
35         }
36     }
37     numIterations++;
38     _results.push_back(_solution->getFitness());
39
40     if ((numIterations % this->_itsPerAnnealing) == 0)
41     {
42         _T *= _annealingFactor;
43     }
44
45     stopCondition.notifyIteration();
46 }
47 }

```

Listing 5: Método *run* de la clase *MQKPSimulatedAnnealing*

## 5. MQKPTabuSearch::run

La clase *MQKPTabuSearch*, está formada por dos ficheros uno de tipo *.h* y otro de tipo *.cpp*. En el fichero *.cpp* se ha de completar la función *run*, ya que se encuentra incompleta en el código esqueleto de la práctica.

La función *run*, se encarga de ejecutar la metaheurística hasta alcanzar la condición de parada. En ella se genera una permutación de objetos y se realiza la asignación no tabú de la mejor operación posible de forma que se intente alcanzar una mejor solución en cada iteración. A continuación, en el Listing 6 se mostrará el código del método en el que hemos resuelto esta problemática.

```

1 void MQKPTabuSearch::run(MQKPStopCondition &stopCondition)
2 {
3     if (_solution == NULL)
4     {
5         cerr << "Tabu search has not been given an initial solution" << endl;
6         exit(-1);
7     }
8
9     _results.clear();

```

```

10 unsigned numObjs = _instance->getNumObjs();
11 unsigned numKnapsacks = _instance->getNumKnapsacks();
12 unsigned numIterations = 0;
13
14 while (!stopCondition.reached())
15 {
16
17     vector<int> perm;
18     MQKPInstance::randomPermutation(numObjs, perm);
19     double bestDeltaFitness = 0;
20     bool initialisedDeltaFitness = false;
21     MQKPObjectAssignmentOperation bestOperation;
22
23     for (unsigned i = 0; i < numObjs; i++)
24     {
25         unsigned indexObj = perm[i];
26
27         if (this->_shortTermMem_aux.end() == this->_shortTermMem_aux.find(indexObj
28     ))
29     {
30         for (unsigned j = 0; j <= numKnapsacks; j++)
31         {
32
33             if (_solution->whereIsObject(indexObj) == ((int)j))
34                 continue;
35
36             double deltaFitness = MQKPEvaluator::computeDeltaFitness(*this->
37 _instance, *this->_solution, indexObj, (int)j);
38
39             if (deltaFitness > bestDeltaFitness || initialisedDeltaFitness ==
40 false)
41             {
42                 initialisedDeltaFitness = true;
43                 bestDeltaFitness = deltaFitness;
44                 bestOperation.setValues(indexObj, j, bestDeltaFitness);
45             }
46         }
47     }
48
49     bestOperation.apply(*this->_solution);
50     this->_shortTermMem.push(bestOperation.getObj());
51     _shortTermMem_aux.insert(bestOperation.getObj());
52
53     if (this->_shortTermMem.size() > this->_tabuTennure)
54     {
55         unsigned value = this->_shortTermMem.front();
56         this->_shortTermMem.pop();
57         this->_shortTermMem_aux.erase(value);
58     }
59
60     if (MQKPEvaluator::compare(_solution->getFitness(),
61 _bestSolution->getFitness()) > 0)
62     {
63         _bestSolution->copy(*_solution);
64     }
65
66     numIterations++;
67     _results.push_back(_solution->getFitness());
68
69     stopCondition.notifyIteration();
70 }

```

70 }

Listing 6: Método *run* de la clase *MQKPTabuSearch*

## 6. MQKPGrasp::chooseOperation

La clase *MQKPGrasp* está formada por dos archivos: uno de formato *.h* y otro de formato *.cpp*. Se ha de completar varios métodos incompletos de esta clase, en este apartado se explicará cómo se ha completado el método *chooseOperation* de esta clase.

En este método, se generan soluciones aleatorias y se calcula el *deltaFitness* y la densidad de cada solución y actualizamos el estado de la solución guardada con la solución que resulte ser la mejor. El Listing 7 muestra el código del método una vez completado.

```
1 void MQKPGrasp::chooseOperation(MQKPObjectAssignmentOperation &operation)
2 {
3
4     int bestObj = 0;
5     int bestKnapsack = 0;
6     double bestDensity = 0;
7     double bestDeltaFitness = 0;
8     bool initialisedBestDensity = false;
9     unsigned numObjs = _instance->getNumObjs();
10    unsigned numKnapsacks = _instance->getNumKnapsacks();
11
12    unsigned numTries = ((unsigned)(numObjs * numKnapsacks * _alpha));
13
14    for (unsigned i = 0; i < numTries; i++)
15    {
16        int indexObj = rand() % numObjs;
17        int indexKnapsack = (rand() % numKnapsacks) + 1;
18
19        double deltaFitness = MQKPEvaluator::computeDeltaFitness(*this->_instance, *
20        this->_sol, indexObj, indexKnapsack);
21        double density = deltaFitness / this->_instance->getWeight(indexObj);
22
23        if (density > bestDensity || initialisedBestDensity == false)
24        {
25            initialisedBestDensity = true;
26            bestDensity = density;
27            bestObj = indexObj;
28            bestKnapsack = indexKnapsack;
29            bestDeltaFitness = deltaFitness;
30        }
31    }
32    operation.setValues(bestObj, bestKnapsack,
33                        bestDeltaFitness);
34 }
```

Listing 7: Método *chooseOperation* de la clase *MQKPGrasp*

## 7. MQKPGrasp::buildInitialSolution

Otro método a completar de la clase *MQKPGrasp*, es el método *buildInitialSolution*. Este método pone todas las mochilas a 0, es decir vacía todas las mochilas e inserta los objetos fuera (mochila 0). También se deja a cero el *Fitness* de la solución ya que hemos reiniciado el problema.

A continuación, se llama a la función completada en el apartado anterior para generar nuevas soluciones, de las que se almacenará el *Fitness* obtenido en cada una de las soluciones en un vector de resultados para más adelante generar gráficas que muestren la mejora del algoritmo. El código resultante para este método es observable en el Listing 8.

```

1 void MQKPGrasp::buildInitialSolution()
2 {
3
4     unsigned numObjs = _instance->getNumObjs();
5
6     _sol->setFitness(0);
7     for (unsigned i = 0; i < numObjs; i++)
8     {
9         this->_sol->putObjectIn(i, 0); // Set all sacks to 0
10    }
11
12    MQKPObjectAssignmentOperation operation;
13    chooseOperation(operation);
14
15    while (operation.getDeltaFitness() > 0)
16    {
17        operation.apply(*this->_sol); // Apply the operation into the
18        private variable _sol
19        this->_results.push_back(this->_sol->getFitness()); // Keep track of the
20        solutions fitness
21        chooseOperation(operation);
22    }
23 }
```

Listing 8: Método *buildInitialSolution* de la clase *MQKPGrasp*

## 8. MQKPGrasp::run

El último método que nos quedaría por completar de la clase *MQKPGrasp*, sería el método *run*, que ejecuta la metaheurística hasta que se haya alcanzado la condición de parada. En el Listing 9 se muestra la solución codificada de este método.

```

1 void MQKPGrasp::run(MQKPStopCondition &stopCondition)
2 {
3
4     if (_sol == NULL)
5     {
6         cerr << "GRASP was not initialised" << endl;
7         exit(-1);
8     }
9
10    while (stopCondition.reached() == false)
11    {
12        buildInitialSolution();
13        this->_results.push_back(_sol->getFitness());
14        this->_ls.optimise(*this->_instance, this->_no, *this->_sol);
15
16        vector<double> &auxResults = _ls.getResults();
17
18        for (auto result : auxResults)
19        {
20            _results.push_back(result);
21        }
22
23        if (MQKPEvaluator::compare(_sol->getFitness(), _bestSolution->getFitness()) >
24            0)
```



```

24         _bestSolution->copy(*_sol);
25
26         stopCondition.notifyIteration();
27     }
28 }

```

Listing 9: Método *run* de la clase *MQKPGrasp*

En este método primero generamos una solución inicial con uno de los métodos completados anteriormente y a continuación se almacena el resultado de *Fitness* obtenido y se optimiza con el método de búsqueda local y se actualiza con la mejor solución. Esto se hace en bucle hasta que se alcanza la condición de parada.

## 9. MQKPIteratedGreedy::chooseOperation

La clase *MQKPIteratedGreedy* que está formada por dos archivos, uno de formato *.cpp* y el otro de formato *.h*. Estos vienen dados en el código esqueleto de la práctica que ha sido proporcionado desde la plataforma *Moodle* de la asignatura.

En el archivo de formato *.cpp* hay cuatro funciones que vienen incompletas y que hemos de completar. El primer método a completar es el método *chooseOperation*. Este método hace la mejor asignación posible, de un objeto sin asignar (se encontrará en la mochila cero) a una mochila. El Listing 10 muestra el código de este método.

```

1 void MQKPIteratedGreedy::chooseOperation(
2     MQKPObjectAssignmentOperation &operation)
3 {
4
5     int bestObj = 0;
6     int bestKnapsack = 0;
7     double bestDensity = 0;
8     double bestDeltaFitness = 0;
9     bool initialisedBestDensity = false;
10    unsigned numObjs = _instance->getNumObjs();
11    unsigned numKnapsacks = _instance->getNumKnapsacks();
12
13    for (unsigned i = 0; i < numObjs; i++)
14    {
15
16        int indexObj = i;
17
18        if (_sol->whereIsObject(indexObj) == 0)
19        {
20
21            for (unsigned j = 1; j <= numKnapsacks; j++)
22            {
23
24                int indexKnapsack = j;
25
26                double deltaFitness = MQKPEvaluator::computeDeltaFitness(*this->
27                    _instance, *this->_sol, indexObj, indexKnapsack);
28                double density = deltaFitness / this->_instance->getWeight(indexObj);
29
30                if (density > bestDensity || initialisedBestDensity == false)
31                {
32                    initialisedBestDensity = true;
33                    bestDensity = density;
34                    bestObj = indexObj;
35                    bestKnapsack = indexKnapsack;
36                    bestDeltaFitness = deltaFitness;
37                }
38            }
39        }
40    }
41
42    operation.execute(bestObj, bestKnapsack);
43}

```

```

37         }
38     }
39 }
40
41 operation.setValues(bestObj, bestKnapsack, bestDeltaFitness);
42 }

```

Listing 10: Método *chooseOperation* de la clase *MQKPIteratedGreedy*

Primero, se recorren todos los objetos sin asignar (están asignados a la mochila cero). A continuación se recorre cada mochila y se asigna el objeto a dicha mochila, calculando el *Fitness* y la densidad de la nueva disposición del problema. En caso de que la solución sea la mejor obtenida hasta el momento (esto se dilucidará realizando el cálculo del *deltaFitness* y el peso de la nueva solución y comparándolo con el de la solución vigente), se actualiza el resultado con el obtenido.

## 10. MQKPIteratedGreedy::rebuild

En el método *rebuild* de la clase *MQKPIteratedGreedy*, se reconstruye la solución llamando repetidamente al método *chooseOperation*, explicado en el apartado anterior. Una vez hallada la mejor solución posible esta se almacena en la variable miembro *\_sol* de esta misma clase.

Durante el desarrollo del método, se van almacenando en un vector de soluciones todas las soluciones obtenidas debido a las varias llamadas al método del apartado anterior *chooseOperation*, ya mencionado, para más tarde realizar gráficas. El Listing 11 muestra el código resultante.

```

1 void MQKPIteratedGreedy::rebuild()
2 {
3
4     MQKPObjectAssignmentOperation operation;
5     chooseOperation(operation);
6
7     while (operation.getDeltaFitness() > 0)
8     {
9         operation.apply(*this->_sol);
10        this->_results.push_back(this->_sol->getFitness());
11        chooseOperation(operation);
12    }
13 }
14 }

```

Listing 11: Método *rebuild* de la clase *MQKPIteratedGreedy*

## 11. MQKPIteratedGreedy::destroy

En el método *destroy* de la clase *MQKPIteratedGreedy*, se destruye parcialmente la solución guardada, sacando objetos de sus mochilas correspondientes con la probabilidad *alpha*. El resultado es observable en el Listing 12.

```

1 void MQKPIteratedGreedy::destroy()
2 {
3
4     unsigned numObjs = _instance->getNumObjs();
5
6     for (unsigned i = 0; i < numObjs; i++)
7     {
8
9         double randSample = ((double)(rand())) / RAND_MAX;
10
11         if (randSample >= this->_alpha)

```

```

12     {
13         _sol->putObjectIn(i, 0);
14     }
15 }
16
17 double fitness = MQKPEvaluator::computeFitness(*_instance, *_sol);
18 _sol->setFitness(fitness);
19 _results.push_back(_sol->getFitness());
20 }

```

Listing 12: Método *destroy* de la clase *MQKPIteratedGreedy*

De hecho, es lo explicado en el párrafo anterior, simple y llanamente lo que hace el método en sí, sacar objetos con probabilidad *alpha* de las mochilas a las que están asignadas y almacenarlos en la mochila cero, que equivale a que no estén asignados a mochila alguna.

## 12. MQKPIteratedGreedy::run

En el método *run* de la clase *MQKPIteratedGreedy*, se ejecuta la metaheurística hasta que se alcance la condición de parada, cómo en el resto de funciones homónimas de otras clases que hemos explicado con anterioridad en el documento. El código resultante se puede observar en el Listing 13.

```

1 void MQKPIteratedGreedy::run(MQKPStopCondition &stopCondition)
2 {
3
4     if (_sol == NULL)
5     {
6         cerr << "IG was not initialised" << endl;
7         exit(-1);
8     }
9
10    rebuild();
11
12    if (MQKPEvaluator::compare(_sol->getFitness(), _bestSolution->getFitness()) > 0)
13        _bestSolution->copy(*_sol);
14
15    while (stopCondition.reached() == false)
16    {
17        destroy();
18        rebuild();
19        this->_results.push_back(_sol->getFitness());
20
21        if (MQKPEvaluator::compare(_sol->getFitness(), _bestSolution->getFitness()) >
22            0)
23        {
24            this->_bestSolution->copy(*this->_sol);
25            this->_bestSolution->setFitness(this->_sol->getFitness());
26        }
27        else
28        {
29            this->_sol->copy(*this->_bestSolution);
30        }
31        stopCondition.notifyIteration();
32    }
33 }

```

Listing 13: Método *run* de la clase *MQKPIteratedGreedy*

Primero, se llama al método *rebuild*, que se encargará de generar una solución y se almacenará como la mejor hasta el momento. Luego, se entrará en el bucle y no se saldrá de este hasta que sea alcanzada la condición de parada. Dentro del bucle se llamará a la función *destroy* y *rebuild*, en

este orden, para generar una nueva solución y se comparará con la mejor solución obtenida hasta el momento. Si esta última obtenida es mejor que la mejor solución hasta el momento, dicha función pasa a ser la nueva *mejor solución*.

## 13. Resultados

### 13.1. Gráficas

#### 13.1.1. Conjunto de datos jeu\_100\_25\_4:

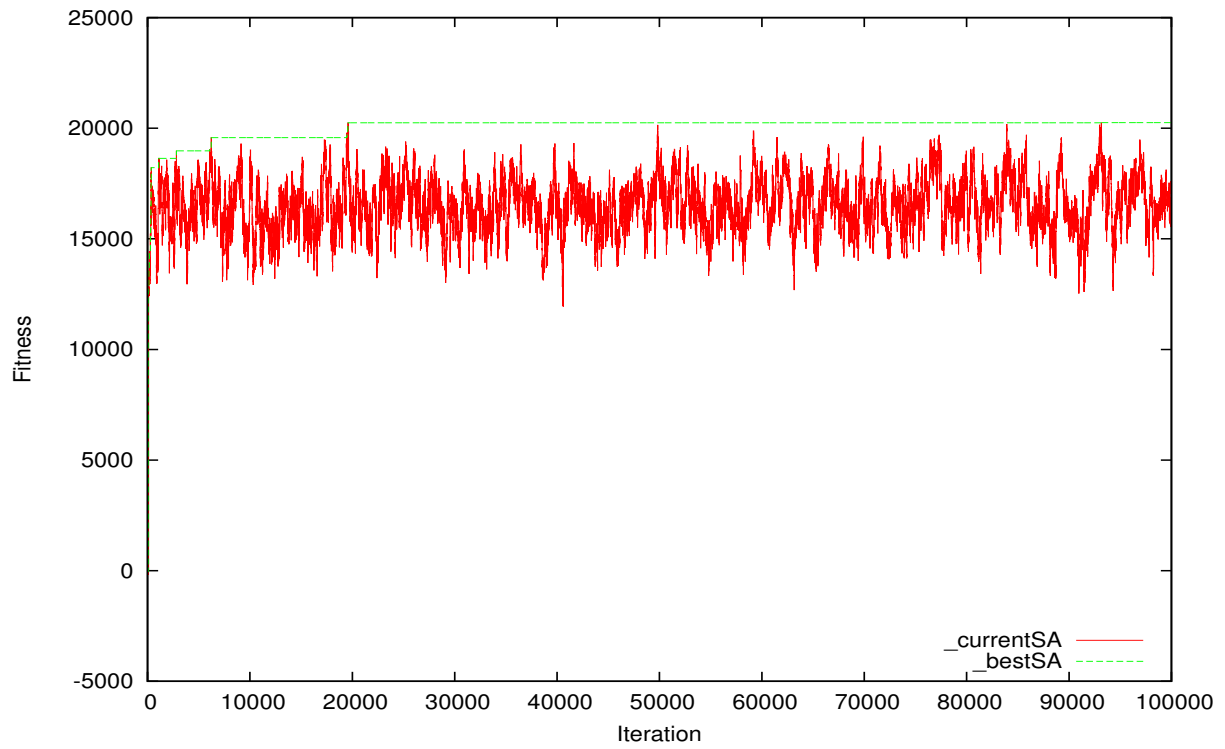


Figura 1: Metaheurística enfriamiento simulado con tres mochilas

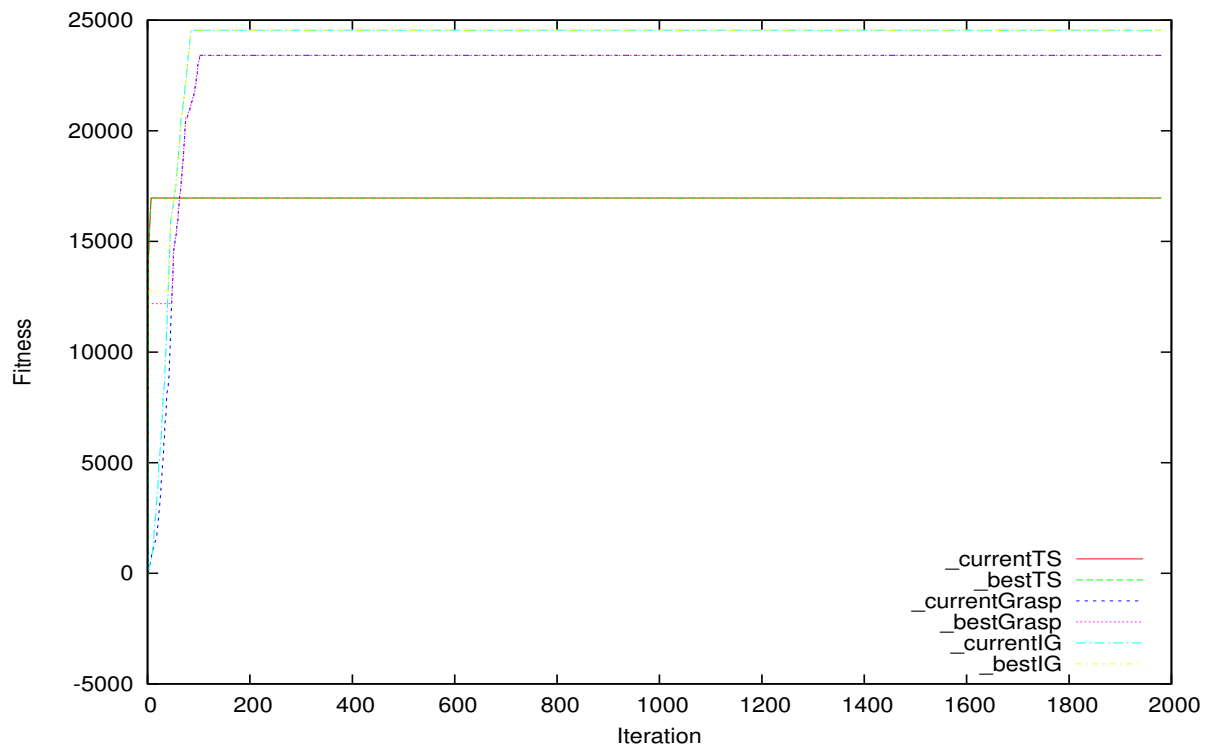


Figura 2: Metaheurísticas con tres mochilas

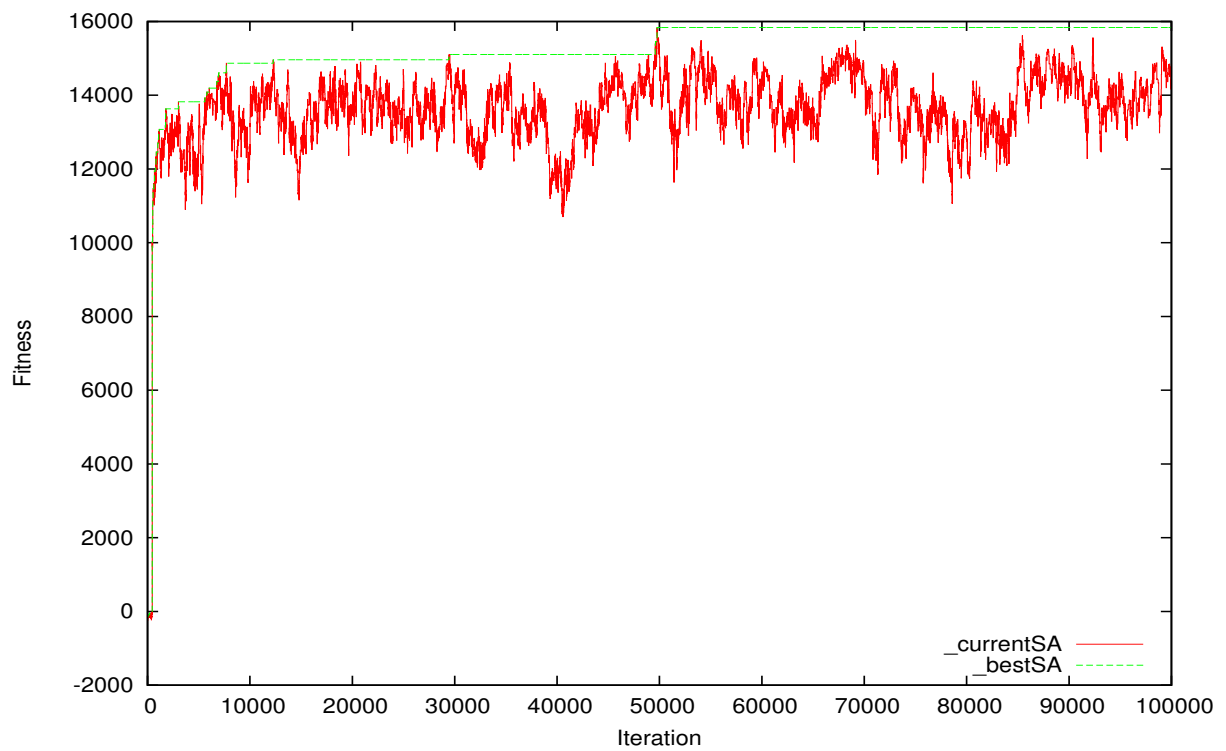


Figura 3: Metaheurística enfriamiento simulado con cinco mochilas

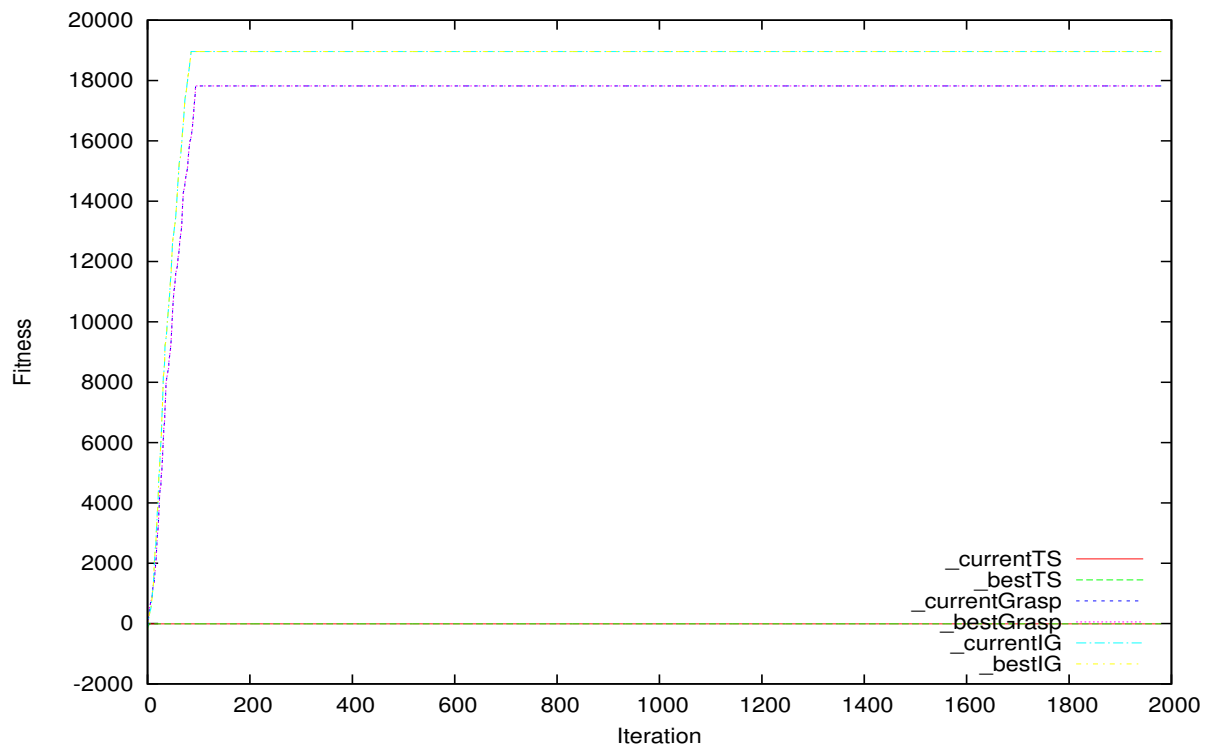


Figura 4: Metaheurísticas con cinco mochilas

#### 13.1.2. Conjunto de datos `jeu_100_75_2`:

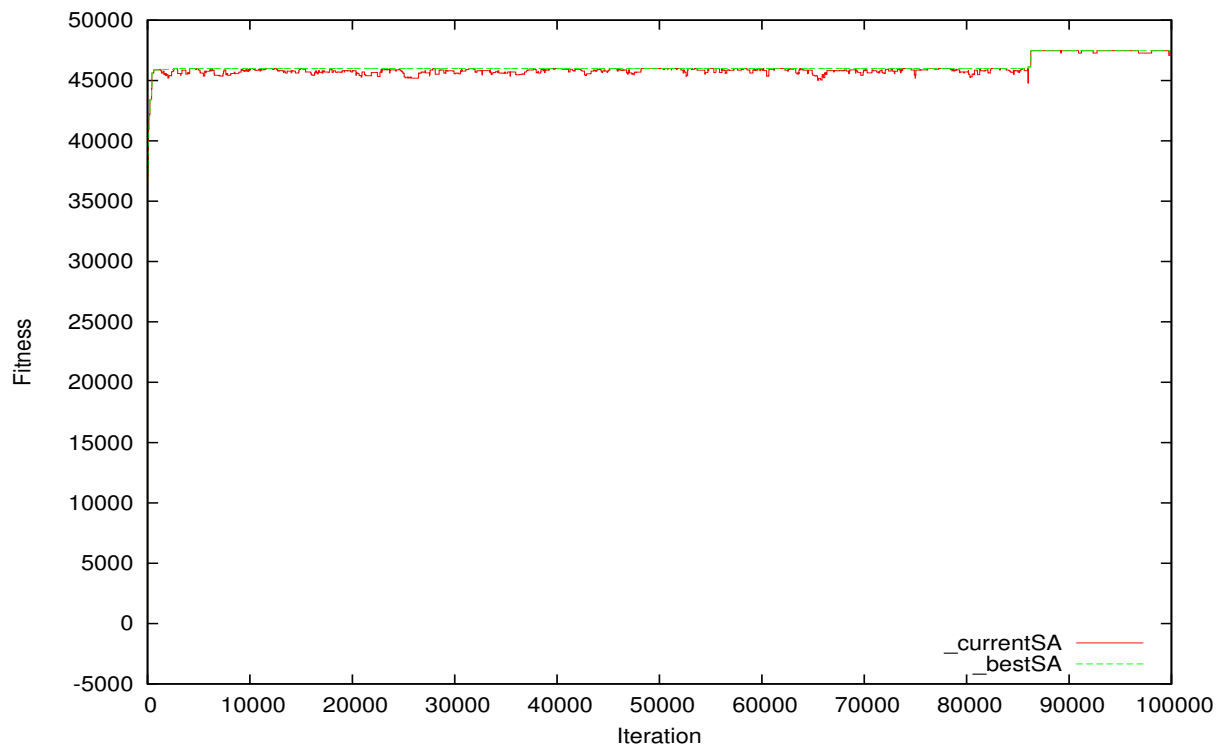


Figura 5: Metaheurística enfriamiento simulado con tres mochilas

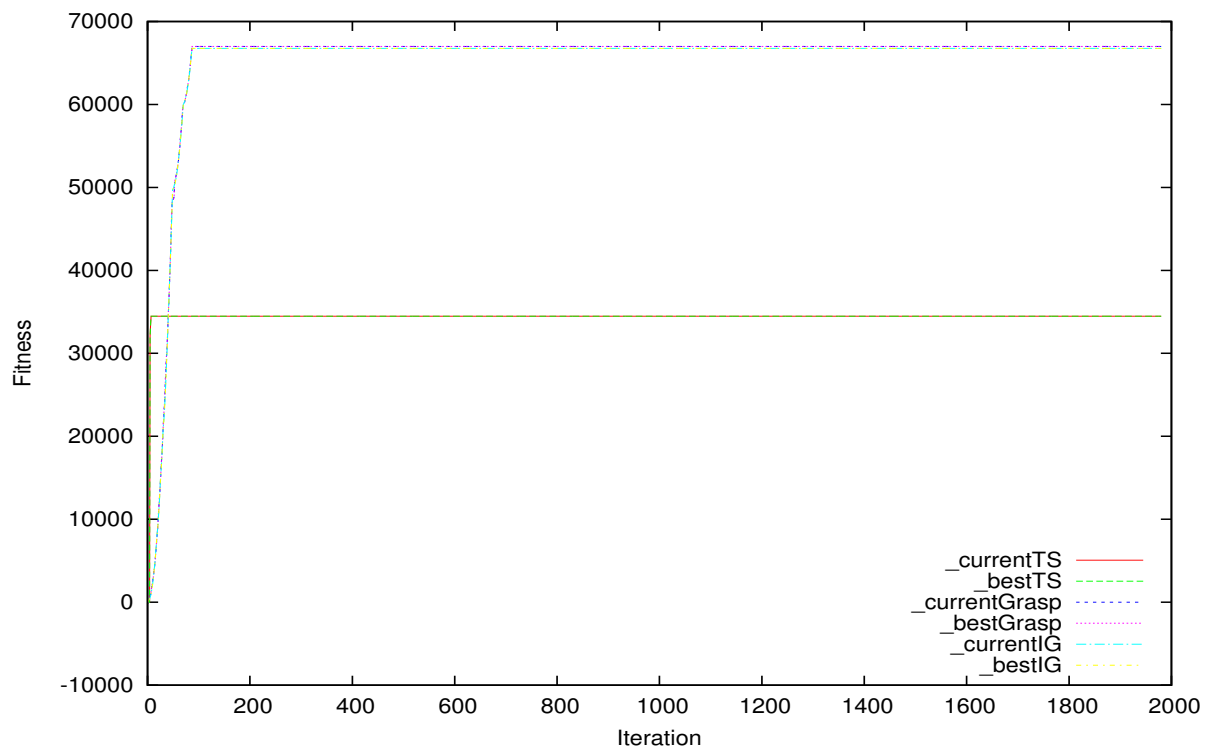


Figura 6: Metaheurísticas con tres mochilas



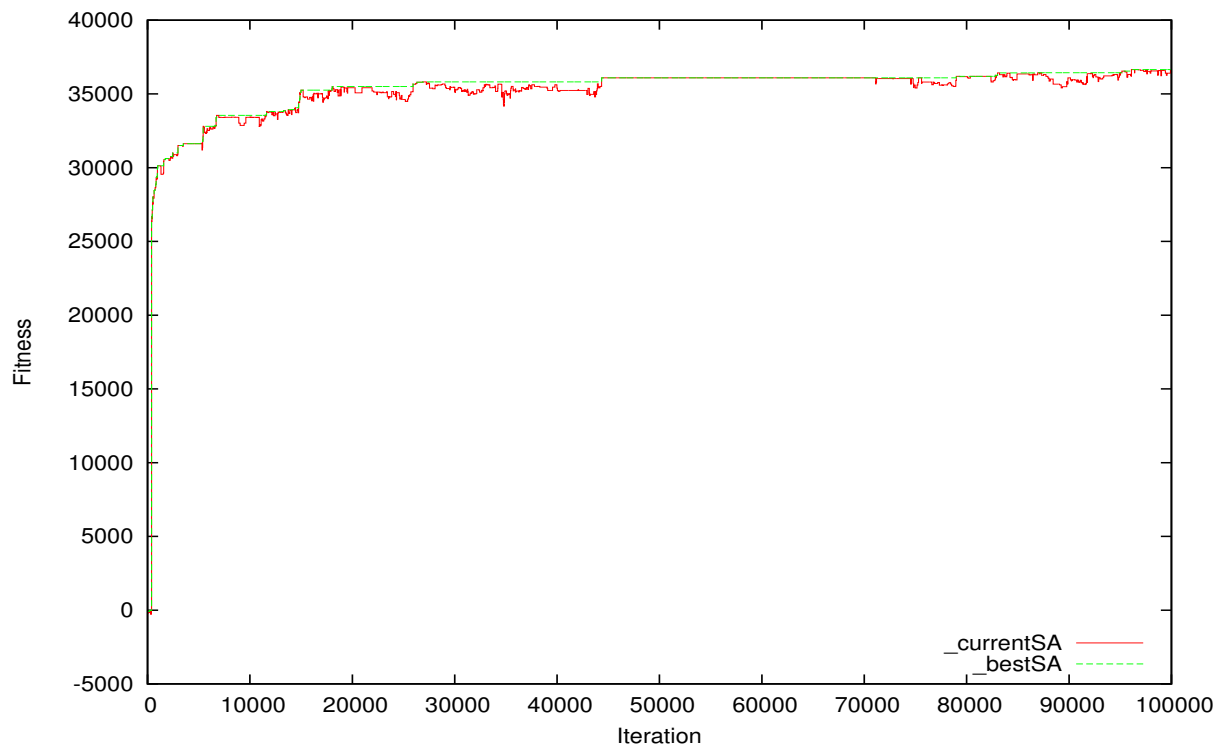


Figura 7: Metaheurística enfriamiento simulado con cinco mochilas

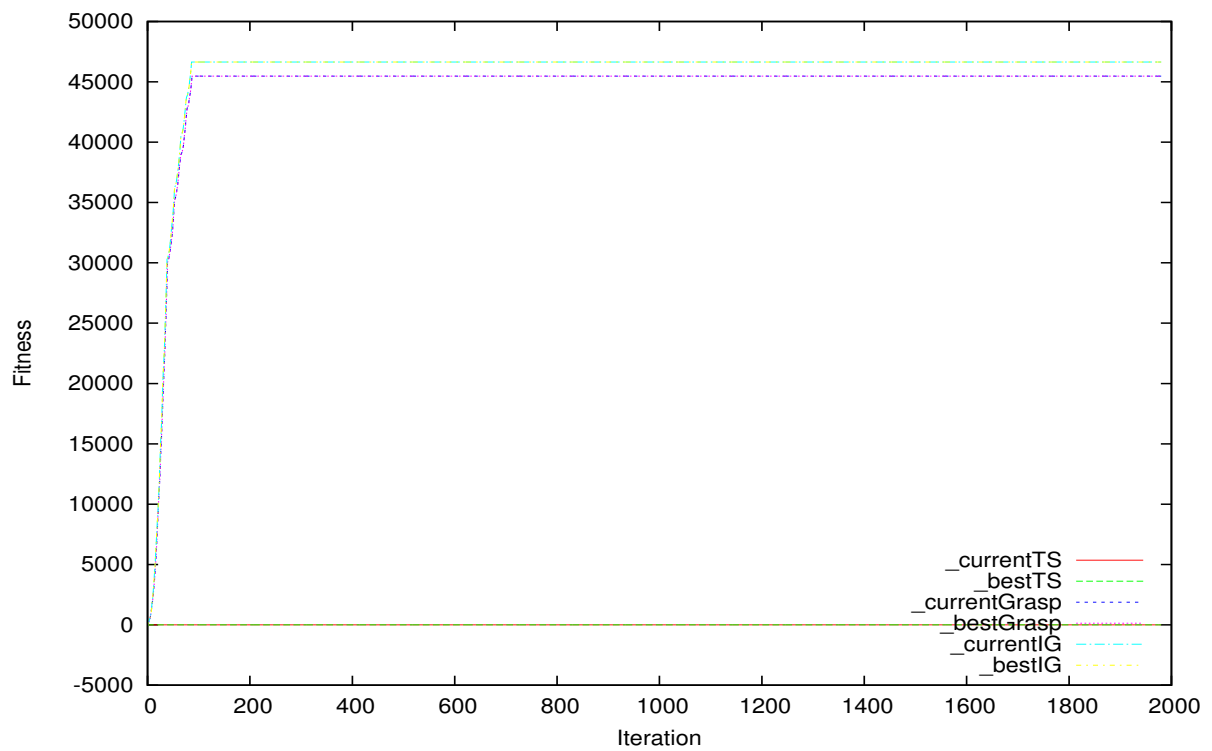


Figura 8: Metaheurísticas con cinco mochilas

### 13.1.3. Conjunto de datos jeu\_200\_25\_8:

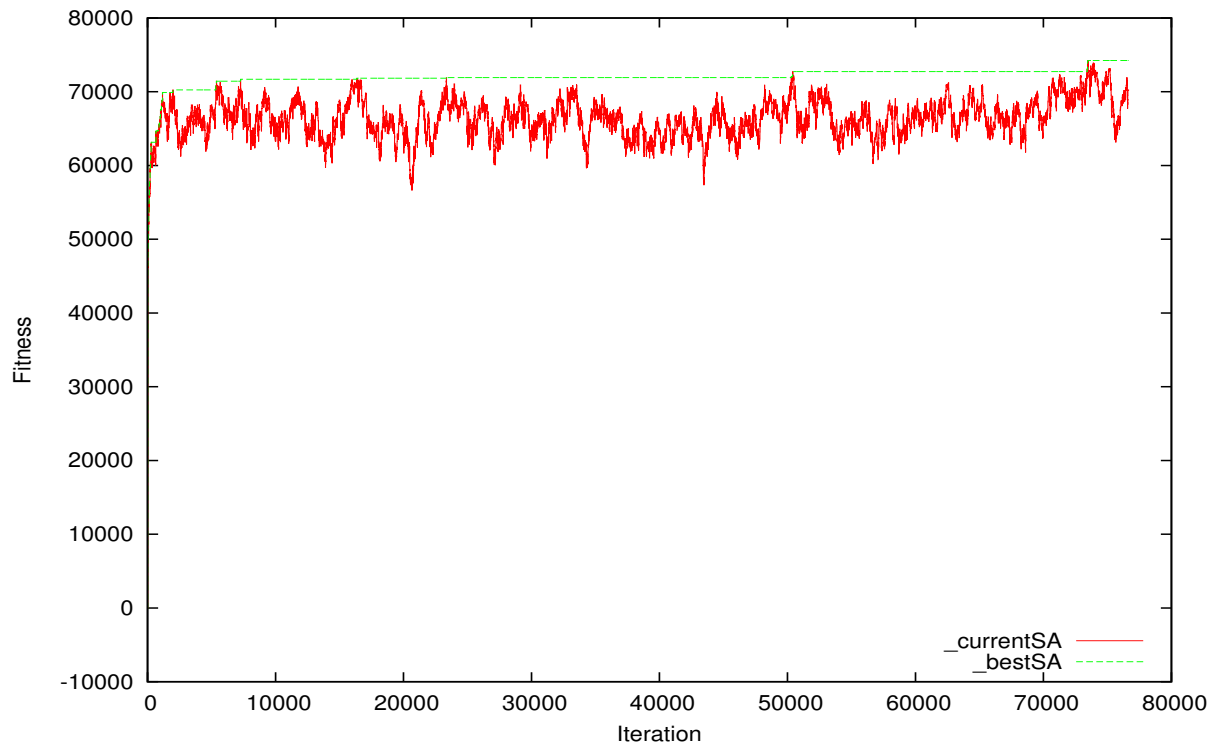


Figura 9: Metaheurística enfriamiento simulado con tres mochilas

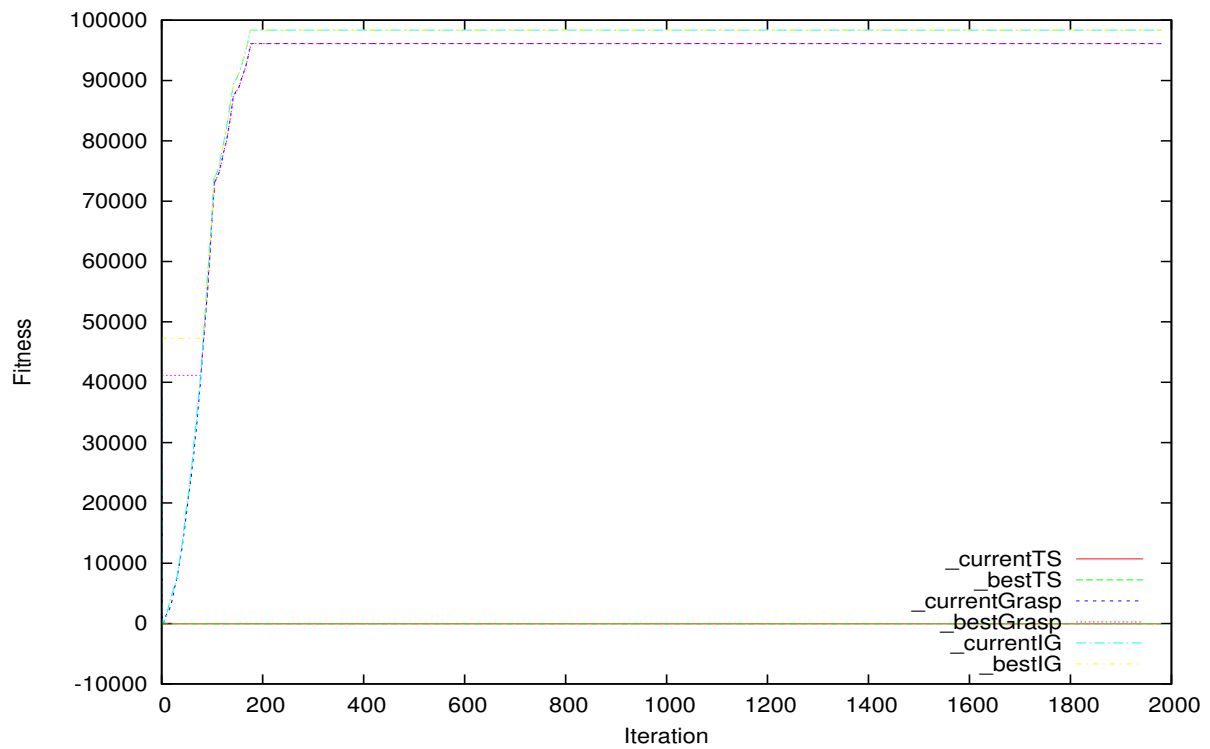


Figura 10: Metaheurísticas con tres mochilas

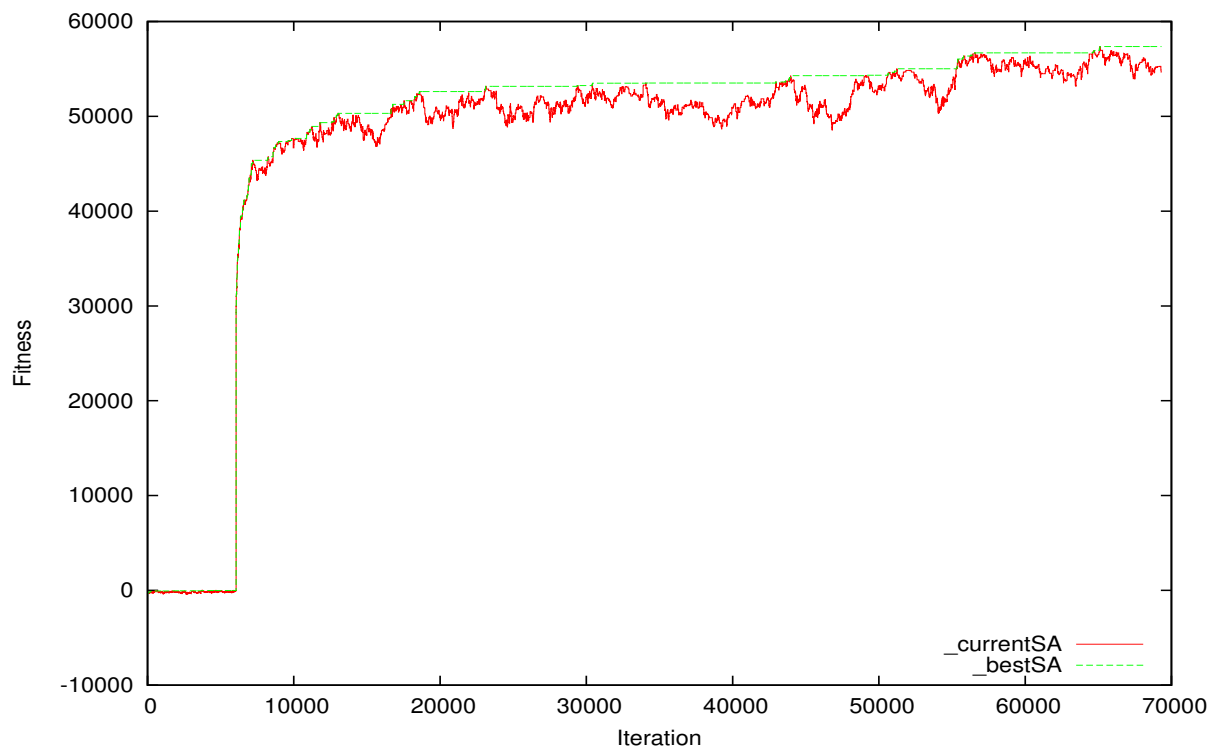


Figura 11: Metaheurística enfriamiento simulado con cinco mochilas

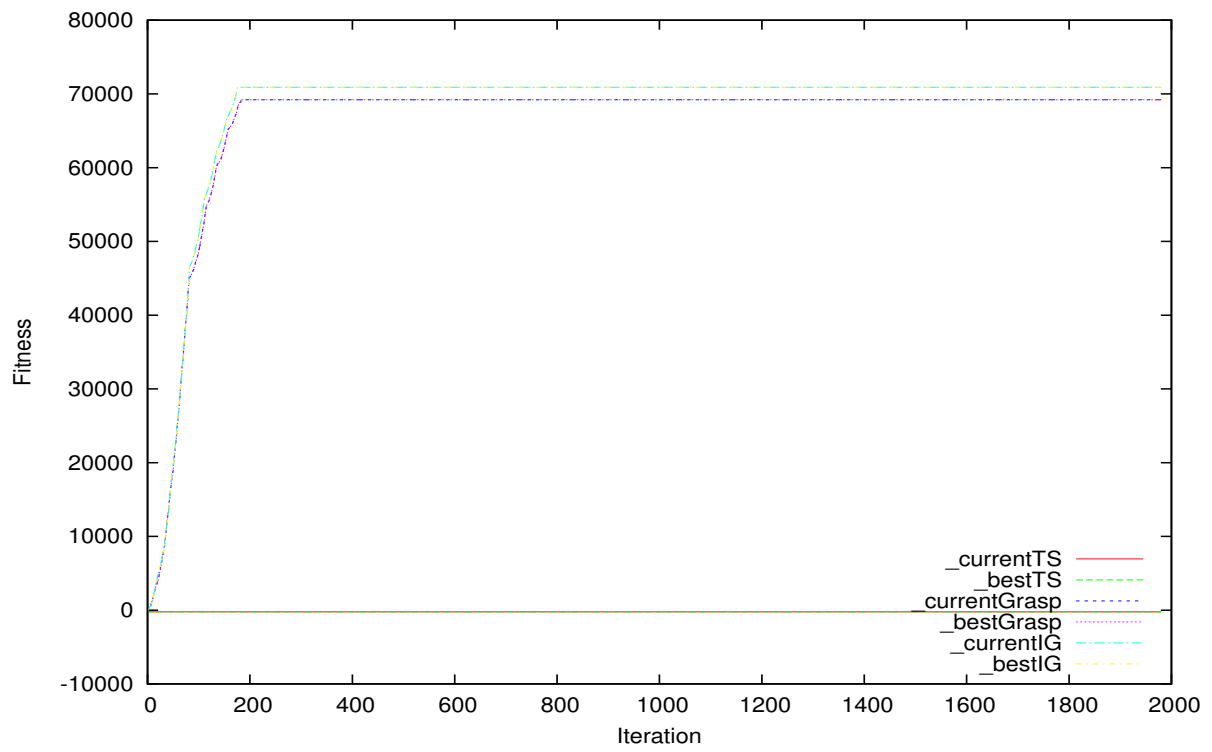


Figura 12: Metaheurísticas con cinco mochilas

#### 13.1.4. Conjunto de datos `jeu_200_75_5`:

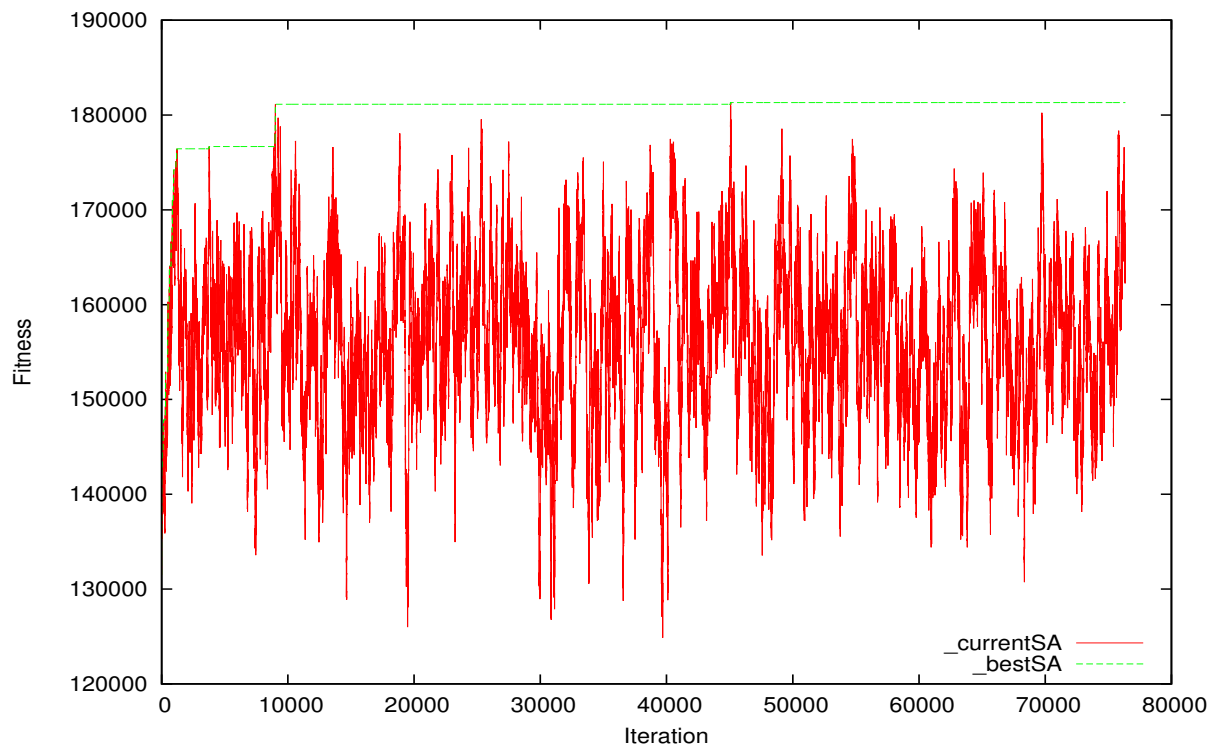


Figura 13: Metaheurística enfriamiento simulado con tres mochilas

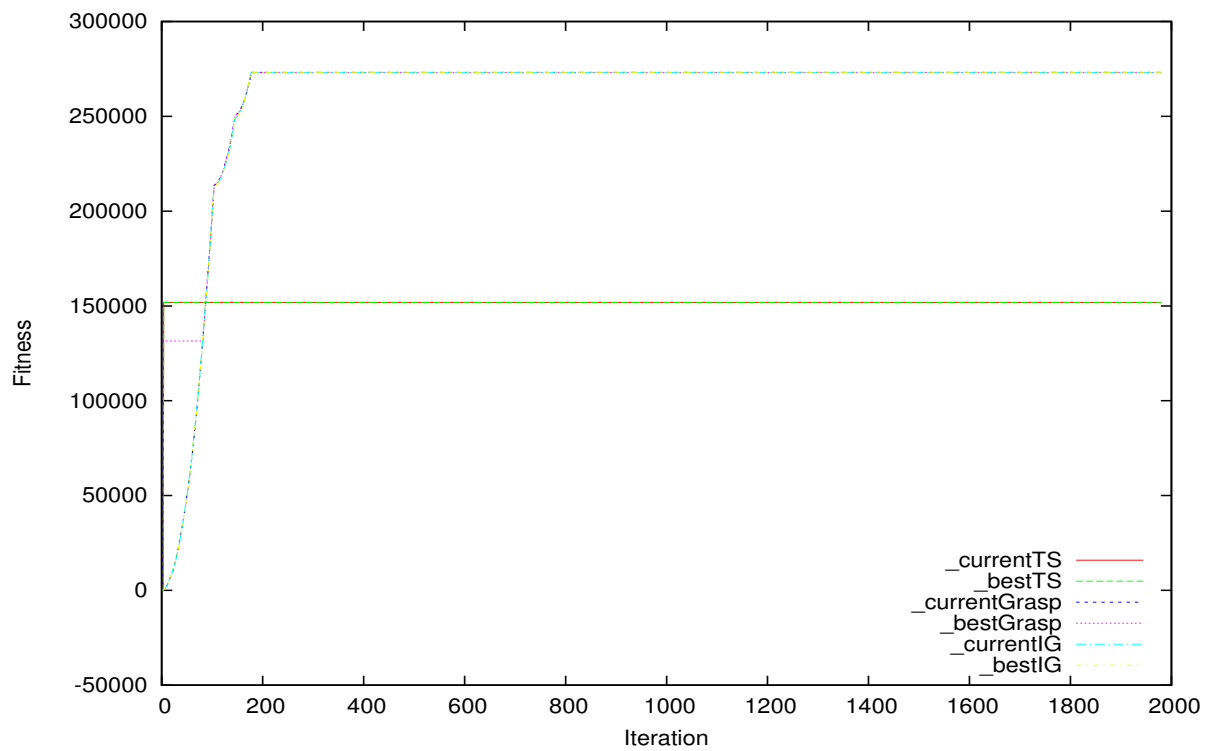


Figura 14: Metaheurísticas con tres mochilas

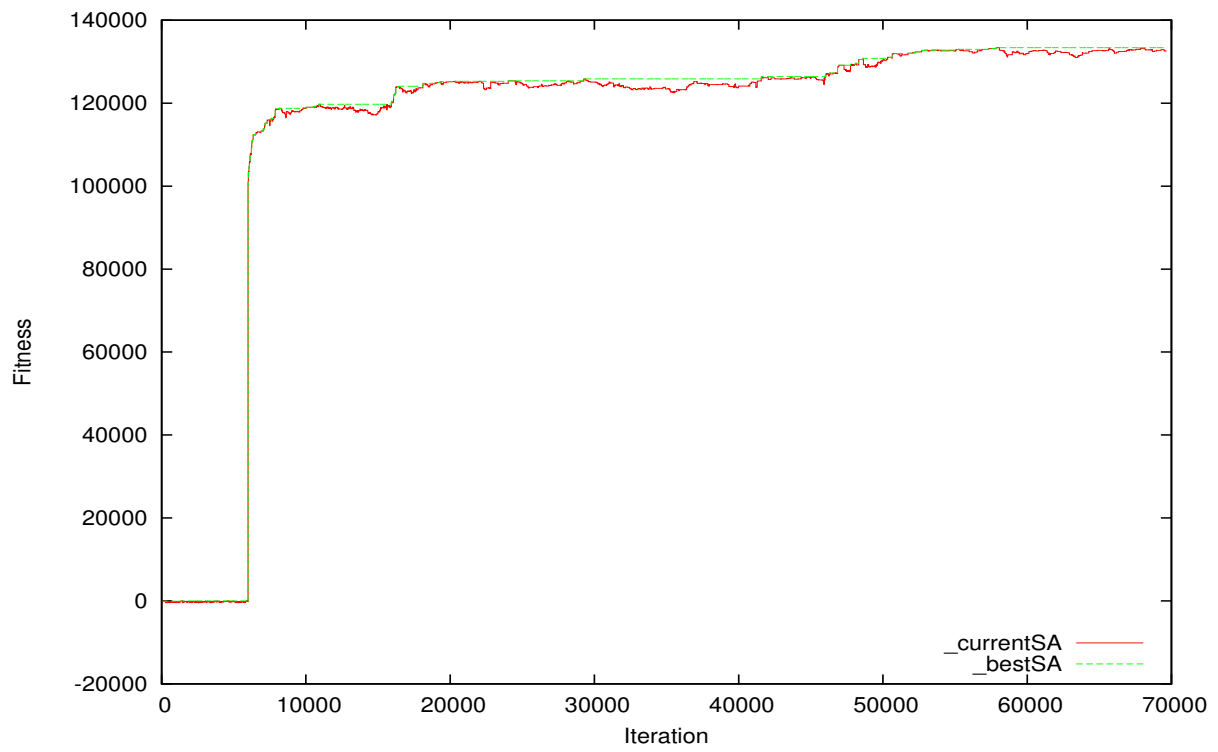


Figura 15: Metaheurística enfriamiento simulado con cinco mochilas

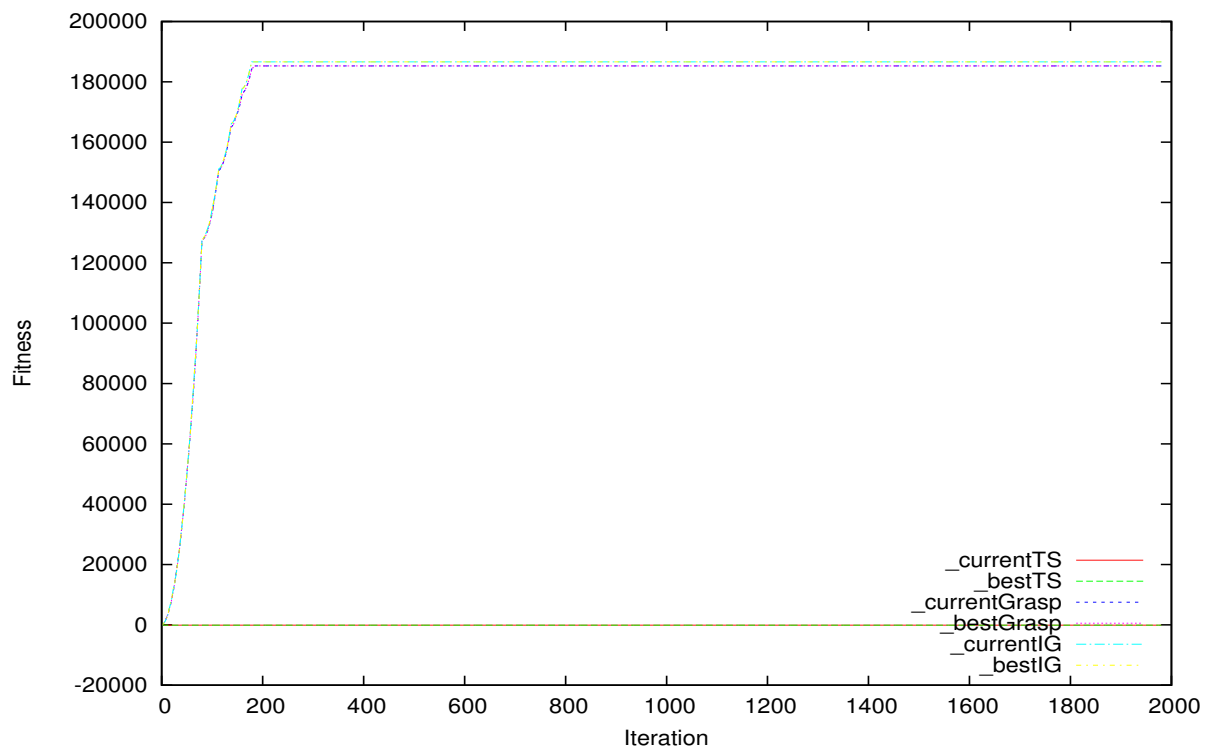


Figura 16: Metaheurísticas con cinco mochilas

## 13.2. Conclusiones

Una vez mostradas todas las gráficas resultantes, hay que decir que hemos decidido dividir los resultados en dos. Una gráfica que incluye los valores obtenidos usando la metaheurística de enfriamiento simulado, y otra gráfica incluyendo el resultado obtenido usando el resto de metaheurísticas. Además, hemos decidido mostrarlas según el conjunto de datos al que pertenecen.

- **Conjunto jeu\_100\_25\_4:** Como se puede observar en las Figuras 1, 2, 3 y 4 los valores de *Fitness* obtenidos son los más bajos, esto puede ser debido a la cantidad de datos que contiene el conjunto. Además podemos observar que para la metaheurística de enfriamiento simulado se necesitan más iteraciones que con las demás. En las figuras 2 y 4, los valores *\_current* y *\_best* de las demás metaheurísticas son los mismos, es decir en cada iteración el valor de *Fitness* es el mismo para ambas variables. Con respecto al enfriamiento simulado podemos observar que el valor actual no es el mismo que el mejor, esto es debido a que el algoritmo permite seleccionar soluciones que no mejoran al anterior. Esto provoca una gran cantidad de picos en las Figuras 1 y 3. Al realizar el mismo estudio con 5 mochilas se han obtenido peores resultados.
- **Conjunto jeu\_100\_75\_2:** Como se puede observar en las Figuras 5, 6, 7 y 8 los resultados obtenidos son mejores que los anteriormente expuestos, usando las mismas metaheurísticas. Como ocurría anteriormente con las demás metaheurísticas, excepto el enfriamiento simulado, los valores de *\_current* y *\_best* son iguales. Al contrario que con estas metaheurísticas, el enfriamiento simulado mejora ya que presenta menos picos en la gráfica. Al igual que ocurría antes el valor de *Fitness* obtenido con 5 mochilas es peor que el obtenido con 3 mochilas.
- **Conjunto jeu\_200\_25\_8:** En las Figuras 9, 10, 11 y 12 se puede observar que los valores de *Fitness* obtenidos son mejores que los del apartado anterior y que al contrario de lo que pasaba con el enfriamiento simulado en el apartado anterior esta vez el algoritmo tiene la necesidad de volver a valores de *Fitness* peores para, finalmente, encontrar la mejor solución. En comparación con el apartado anterior, Figuras 9, 11 presentan muchas diferencias con respecto a como la metaheurísticas llega a la solución óptima.
- **Conjunto jeu\_200\_75\_5:** Este conjunto está compuesto por las Figuras 13, 14, 15 y 16. Como se puede observar en la Figura 13 los valores de *\_current* varían entre un mayor rango que los anteriores, de ahí que podamos observar picos tan grandes en la gráfica. Con respecto a la misma solución pero usando 5 mochilas los valores de *Fitness* presentan un rango mucho más pequeño, lo que conlleva que el valor de *\_current* y *\_best* sean prácticamente el mismo en cada iteración. Como ha ocurrido hasta ahora, los valores de *\_current* y *\_best* de las demás metaheurísticas son los mismos en cada iteración.

## 14. Problema seleccionado

Nuestro grupo ha decidido elegir el problema *Multidimensional 2-way Number Partitioning Problem* (M2NPP). Este problema, como explicamos en la práctica anterior, consiste en, dado  $N$  vectores de una dimensión  $D$ , distribuir dichos vectores en dos conjuntos, de manera que la diferencia máxima entre esos dos vectores sea mínima. Para obtener una mejor idea de cómo resolver este problema, se mostrará a continuación una serie de figuras ilustrativas.

Inicialmente, contamos con un vector de dimensión 15 filas por 10 columnas, como vemos en la Figura 1.


5	7	4	3	9	8	5	1	4	2
3	6	5	4	7	8	6	5	9	8
7	0	6	5	3	0	8	5	6	4
3	7	8	9	6	5	3	8	7	5
6	4	3	0	5	3	5	2	8	5
6	4	1	3	2	4	0	8	6	9
6	4	7	6	5	4	9	7	8	3
4	5	2	9	6	7	4	0	1	2
3	5	4	6	7	0	8	9	7	5
5	5	6	2	4	3	5	9	6	8
7	7	6	4	1	2	3	5	6	7
4	1	3	6	9	5	1	4	7	2
5	6	3	2	1	4	7	8	9	6
2	3	1	7	5	8	9	5	1	4
7	5	6	5	2	5	4	7	8	6

Cuadro 1: Conjunto de vectores inicial

Seguidamente, lo que tenemos que hacer es dividir el conjunto de vectores anterior en dos conjuntos nuevos. La manera de hacerlo se ilustra en las Figuras 2 y 3.

5	5	6	2	4	3	5	9	6	8
7	7	6	4	1	2	3	5	6	7
4	1	3	6	9	5	1	4	7	2
5	6	3	2	1	4	7	8	9	6
2	3	1	7	5	8	9	5	1	4
7	5	6	5	2	5	4	7	8	6

SET 1



30	27	25	26	22	27	29	38	37	33
----	----	----	----	----	----	----	----	----	----

SUM1

Cuadro 2: Primer conjunto



5	7	4	3	9	8	5	1	4	2	SET 2
3	6	5	4	7	8	6	5	9	8	
7	0	6	5	3	0	8	5	6	4	
3	7	8	9	6	5	3	8	7	5	
6	4	3	0	5	3	5	2	8	5	
6	4	1	3	2	4	0	8	6	9	
6	4	7	6	5	4	9	7	8	3	
4	5	2	9	6	7	4	0	1	2	
3	5	4	6	7	0	8	9	7	5	
37	42	40	45	42	39	48	45	48	43	SUM2

Cuadro 3: Segundo conjunto

Como podemos apreciar en las Figuras 2 y 3, una vez dividido el conjunto inicial se realiza la suma de todos los vectores de cada subconjunto, para finalmente realizar la diferencia entre ambos, como se puede apreciar en la Figura 4. Como vemos, la diferencia máxima entre los dos conjuntos es 20, la cuál, hemos conseguido hacerla mínima.

30	27	25	26	22	27	29	38	37	33	SUM1
37	42	40	45	42	39	48	45	48	43	SUM2
7	15	15	19	20	12	19	7	11	10	DIF

Cuadro 4: Conjunto diferencia

Con respecto a las instancias del problema, lo que hemos realizado son tres instancias distintas (pequeña, mediana, grande), cada una con un número de vectores diferentes con dichos tamaños también diferentes. Para la creación de estos conjuntos, hemos decidido que los valores de los vectores estén en el rango  $[0,100]$ , los cuales serán generados de manera aleatoria. Con esto, conseguiremos ver si el tamaño y el número de vectores influye en la solución al aplicar las distintas metaheurísticas.

Para la realización de la solución hemos decidido añadir tres nuevas clases, en las cuales se realizarán las particiones de los correspondientes conjuntos de vectores.

1. **Clase Partition:** Clase que guardará un vector de tipo entero que representará el vector en cuestión, junto con dos variables que corresponderán a la dimensión y al *Fitness* de dicho vector.
2. **Clase Set:** Clase que guardará una variable de tipo Partition, una variable de tipo entero

que representará la dimensión de este conjunto, es decir, el número de vectores incluidos en el conjunto, y finalmente otra variable de tipo Partition que guardará la suma resultante de todos los vectores de dicho conjunto.

3. **Clase Solution:** Esta clase constará de tres variables, dos de tipo Set, las cuales conforman una solución, una variable de tipo Partition, que guardará la diferencia de las dos variables Set anteriores y una variable de tipo entero en la que se guardará el *Fitness* conseguido en esa diferencia.

Una vez descritas las clases adicionales para conseguir la solución a este problema, pasamos a explicar los pasos necesarios para aplicar la Búsqueda Local a nuestro problema. Inicialmente, recibiremos una solución aleatoria, a partir de la cual, el algoritmo irá explorando sus vecinos en busca de una solución que mejore la actual. Para dicha búsqueda, cambiaremos una Partición de un Set a otro, es decir, trataremos la Partición como si fuera un bit. En el instante en el que hemos encontrado un vecino que mejora la solución actual, se devuelve dicha solución y se empezaría de nuevo la búsqueda pero, esta vez, partiendo desde la mejor solución. Esta metaheurística acabaría en el instante en el que no se encuentre ninguna mejor solución vecina.