

Metaheurística

Práctica 4: Metaheurísticas basadas en Poblaciones

5 de mayo de 2019

Resumen

In the present document we will explain how we have approached the solution to the *Multiple Quadratic Knapsack Problem* using C++ language. Here we will use the metaheuristics based on populations, using the Genetic Algorithm and Ant Colony Operations in order to reach the best solutions.

Índice

1. MKQPSolution	3
2. MQKPMutationOperator::mutate(Solution *sol)	4
3. MQKPCrossoverOperator::cross(Solution *s1, Solution *s2)	5
4. MQKPCrossoverOperator::cross(vector<Solution*>&parents, vector<Solution*>&offspring)	5
5. TournamentSelector::selectOne	6
6. TournamentSelector::select	6
7. MQKPGeneticAlgorithm::initialise	7
8. MQKPGeneticAlgorithm::indexBest	8
9. MQKPGeneticAlgorithm::indexWorst	8
10. MQKPGeneticAlgorithm::selectNewPopulation	9
11. MQKPGeneticAlgorithm::evaluate	9
12. MQKPGeneticAlgorithm::initPopulation	10
13. MQKPGeneticAlgorithm::run	11
14. MQKPAnt::sumSignificances	11
15. MQKPAnt::createAlternatives	12
16. MQKPAnt::selectBestAlternative	13
17. MQKPAnt::resetSolution	14
18. MQKPAnt::chooseOperation	14
19. MQKPAntColonyOpt::localUpdate	15
20. MQKPAntColonyOpt::releaseAnts	15
21. MQKPAntColonyOpt::iterate	16
22. MQKPAntColonyOpt::initialise	17
23. MQKPAntColonyOpt::run	18
24. Experimentos y resultados	18
24.1. Gráficas	18
24.2. Conclusiones	38
25. Problema seleccionado	40

1. MKQPSolution

En la clase *MQKPSolution*, formada por los ficheros del mismo nombre, de tipo *.cpp .hpp*. Estos ficheros han sido proporcionados en el código esqueleto de esta práctica.

En esta clase, no hay que realizar tarea alguna, pero se han realizado una serie de cambios en el código, los cuáles vamos a enumerar y explicar en detalle, para que sea más fácil entender el funcionamiento de la clase a lo largo de la ejecución del código, en las llamadas que haremos a esta.

- Primero, se ha creado la superclase *Solution*, para independizar algunos de los operadores de la codificación/problema utilizado, lo cuál ha generado unos cambios en la clase *MQKPSolution*.

```
1 class MQKPSolution : public Solution
```

Listing 1: Creación de la superclase *Solution* en *MQKPSolution*

- Se ha añadido en la parte *protected* de la clase, la variable *_fitnessAssigned*, para saber qué individuos tienen el fitness asignado.

```
1 int *_sol;
2 int _numObjs;
3 double _fitness;
4 bool _fitnessAssigned;
```

Listing 2: Nueva variable *_fitnessAssigned* en *MQKPSolution*

- Se ha modificado la función *putObjectIn* en el fichero *.cpp* de la clase.

```
1 void MQKPSolution::putObjectIn(int object, int knapsack)
2 {
3     _sol[object] = knapsack;
4     _fitnessAssigned = false;
5 }
```

Listing 3: Método *putObjectIn* de la clase *MQKPSolution*

- Luego, se crean el observador y el modificador del fitness (como métodos virtuales en el fichero *.h*), de los cuáles se mostrarán cómo se ha implementado el código de estos métodos en el fichero *.cpp*.

```
1 double MQKPSolution::getFitness() const
2 {
3     return _fitness;
4 }
5
6 void MQKPSolution::setFitness(double fitness)
7 {
8     _fitness = fitness;
9     _fitnessAssigned = true;
10 }
```

Listing 4: Observador y modificador de la variable *_fitness* en *MQKPSolution*

- A continuación, veremos cómo se ha implementado el método *copy*, que se ha de declarar como método virtual en el fichero *.h*, cuyo cometido es copiar la información de otra solución.

```
1 void MQKPSolution::copy(Solution &solution)
2 {
3
4     MQKPSolution &auxSol = (MQKPSolution &)solution;
5
6     for (int i = 0; i < _numObjs; i++)
```

```

7     _sol[ i ] = auxSol._sol[ i ];
8
9     _fitness = auxSol._fitness;
10    _fitnessAssigned = auxSol._fitnessAssigned;
11 }

```

Listing 5: método *copy* de la clase *MQKPSolution*

- Por último, veremos la función *isValidFitness*, creada en el fichero *.h* de la clase. Está indica si el ‘Fitness’ de la solución es válido.

```

1   bool isValidFitness()
2   {
3       return _fitnessAssigned;
4   }

```

Listing 6: Método *isValidFitness* de la clase *MQKPSolution*

Hemos creado una variable y una función nuevas y hemos modificado varios métodos de la clase, con el motivo de añadir la funcionalidad de determinar si la solución tiene un valor de fitness correctamente asignado.

Esto se ha hecho para ahorrar tiempo en la ejecución del código, ya que es innecesario volver a evaluar la solución si esta ya ha sido evaluada anteriormente, es decir, en el caso de que el hijo copie de forma inalterada la solución del padre, estaríamos ante la misma solución que ya evaluamos cuándo estábamos evaluando al padre, de ahí que fuese algo repetitivo e inútil el evaluar la misma solución de nuevo.

Como se ha podido observar, en este punto hemos abarcado los dos primeros puntos, de las tareas a realizar, pertenecientes al PDF de la práctica 4.

2. MQKPMutationOperator::mutate(Solution *sol)

En la clase *MQKPMutationOperator* (comprendida por un solo fichero, de tipo *.h*), se ha de completar el método *mutate*, cuyo cometido es mutar una solución. En esta función, hemos de implementar el código que recorra todos los objetos, y, en función de la probabilidad de mutar que se tenga, asignarlos a una mochila aleatoria. La mutación podría no modificar gen alguno, modificar uno o más de un gen.

A continuación se mostrará el código implementado para que la función realice lo explicado en el párrafo anterior:

```

1   void mutate( Solution *sol )
2   {
3       MQKPSolution *s = (MQKPSolution *)sol;
4
5       for ( unsigned i = 0; i < _numObjs; i++)
6       {
7
8           if (((double)rand() / RAND_MAX) < _mutProb)
9               s->putObjectIn(i, rand() % (this->_numKnapsacks + 1));
10
11 }

```

Listing 7: Método *mutate* de la clase *MQKPMutationOperator*

3. MQKPCrossoverOperator::cross(Solution *s1, Solution *s2)

Esta función, viene incompleta en el código esqueleto proporcionado, para que sea completada. En dicho método se cruzan dos soluciones en función de la probabilidad de cruce entre estas. En caso de que no se produzca el cruce, la solución que devolveremos en la función será la del primer parent.

El cruce a realizar será uniforme, es decir, para cada gen se elige de forma aleatoria el valor de dicho gen de uno de los padres. En caso de no haberse realizado el cruce, se copia la solución del primer parent.

Todo esto, lo podemos ver implementado en el código de la función:

```
1  MQKPSolution *cross(Solution *s1, Solution *s2)
2  {
3      MQKPSolution *sol = new MQKPSolution(*_instance);
4      MQKPSolution *sol1 = (MQKPSolution *)s1;
5      MQKPSolution *sol2 = (MQKPSolution *)s2;
6
7      double randSample = (((double)rand()) / RAND_MAX);
8
9      if (randSample < _crossProb)
10     {
11         int parent;
12
13         for (unsigned i = 0; i < _numObjs; i++)
14         {
15             parent = rand() % 2;
16
17             if (parent == 0)
18             {
19                 sol->putObjectIn(i, sol1->whereIsObject(i));
20             }
21
22             else
23             {
24                 sol->putObjectIn(i, sol2->whereIsObject(i));
25             }
26         }
27     }
28
29     else
30     {
31
32         sol->copy(*sol1);
33     }
34
35
36     return sol;
37 }
```

Listing 8: Método *MQKPCrossoverOperator::cross(Solution *s1, Solution *s2)* de la clase *MQKPCrossoverOperator*

4. MQKPCrossoverOperator::cross(vector<Solution*>&parents, vector<Solution*>& offspring)

Esta función, tiene el mismo nombre que la del apartado anterior, pero es distinta dado que los parámetros de entrada son distintos a los de la otra función, haciendo que sean funciones comple-

tamente distintas pese a compartir el nombre.

Dicha función recibe una población de padres (mediante un vector de soluciones), los cuáles se emparejarán de dos en dos y se cruzarán, llamando de forma iterativa a la función *cross* explicada en el punto anterior. Cada hijo obtenido, se guardará en el otro vector pasado a la función.

```

1   void cross(vector<Solution *> &parents, vector<Solution *> &offspring)
2   {
3       unsigned numParents = (unsigned) parents.size();
4
5       for (unsigned i = 0; i < numParents; i = i + 2)
6       {
7           MQKPSolution *sol = cross(parents[i], parents[i + 1]);
8           offspring.push_back(sol);
9       }
10      }
11  }
```

Listing 9: Método *MQKPCrossoverOperator::cross*(*vector<Solution*>& parents, vector<Solution*>& offspring*) de la clase *MQKPCrossoverOperator*

5. TournamentSelector::selectOne

Esta función, perteneciente a clase *TournamentSelector*, selecciona una solución de todo el conjunto mediante torneo. Esto consiste en seleccionar una solución como la ganadora del torneo, de forma aleatoria e ir cogiendo una solución aspirante. Ambas soluciones se compararán y nos quedaremos con la mejor de estas. Este procedimiento se realizará k-1 veces, siendo ‘k’, el número de participantes del torneo.

```

1   Solution *selectOne(vector<Solution *> &set)
2   {
3       Solution *best;
4
5       best = set[rand() % set.size()];
6
7       for (unsigned i = 0; i < _k - 1; i++)
8       {
9           Solution *sol;
10          sol = set[rand() % (int)set.size()];
11
12          if (MQKPEvaluator::compare(sol->getFitness(), best->getFitness()) > 0)
13          {
14              best = sol;
15          }
16      }
17
18      return best;
19  }
```

Listing 10: Método *selectOne* de la clase *TournamentSelector*

6. TournamentSelector::select

En este método, someteremos a torneo a tantas parejas de padres como padres tengamos, llamando de forma iterativa a la función *selectOne*. De esta forma, si originalmente tenemos N padres,

tras aplicar este método, generaremos N hijos mediante torneo, cuya información guardaremos en un vector.

```

1   virtual void select(vector<Solution *> &orig , vector<Solution *> &result)
2   {
3
4     for (unsigned i = 0; i < orig.size(); i++)
5     {
6       result.push_back(selectOne(orig));
7       result.push_back(selectOne(orig));
8     }
9   }
```

Listing 11: Método *select* de la clase *TournamentSelector*

7. MQKPGeneticAlgorithm::initialise

En la clase *MQKPGeneticAlgorithm*, hay un método, *initialise*, en el inicializamos el algoritmo. En este método se configuran por defecto unos operadores de selección, cruce y mutación. A continuación, se mostrará el código de la función:

```

1   void initialise (unsigned popSize , MQKPIstance &instance )
2   {
3     _instance = &instance ;
4
5     if (popSize <= 0)
6     {
7       cerr << "The population size must be greater than 0" << endl;
8       exit(1);
9     }
10
11    if (_bestSolution != NULL)
12    {
13      delete _bestSolution;
14      _bestSolution = NULL;
15    }
16
17    bestSolution = new MQKPSolution(*_instance);
18    MQKPSolGenerator::genRandomSol(*_instance , *_bestSolution);
19    double fitness = MQKPEvaluator::computeFitness(*_instance , *_bestSolution);
20    _bestSolution->setFitness(fitness);
21
22    _popSize = popSize;
23
24    if (_crossoverOp == NULL)
25    {
26      _crossoverOp = new MQKPCrossoverOperator(0.8 , *_instance);
27    }
28
29    if (_mutOp == NULL)
30    {
31      _mutOp = new MQKPMutationOperator((0.25 / _instance->getNumObjs()) ,
32                                       *_instance);
33    }
34
35    if (_selector == NULL)
36    {
37      _selector = new TournamentSelector(2);
38    }
```

```
39 }
```

Listing 12: Método *initialise* de la clase *MQKPGeneticAlgorithm*

8. MQKPGeneticAlgorithm::indexBest

El método *indexBest*, busca en el vector de soluciones la mejor solución y devuelve el índice de esta en el vector. Para ello, compara iterativamente con la función *compare*, con la mejor solución encontrada hasta el momento, cogiendo por defecto, la primera solución del vector.

A continuación, mostraremos la función completa:

```
1  unsigned indexBest(vector<Solution *> &set)
2  {
3
4      unsigned indexBest = 0;
5
6      for (unsigned i = 1; i < set.size(); i++)
7      {
8
9          if (MQKPEvaluator::compare(set[i]->getFitness(), set[indexBest]->
10             getFitness()) > 0)
11          {
12              indexBest = i;
13          }
14      }
15
16      return indexBest;
17 }
```

Listing 13: Método *indexBest* de la clase *MQKPGeneticAlgorithm*

9. MQKPGeneticAlgorithm::indexWorst

Este método, *indexWorst*, es similar al método *indexBest*, explicado en el apartado anterior, solo que compara por pares los elementos del vector de soluciones para quedarse con el peor, para así, al final del bucle, obtener el índice de la peor solución del vector.

```
1  unsigned indexWorst(vector<Solution *> &set)
2  {
3
4      unsigned indexWorst = 0;
5
6      for (unsigned i = 1; i < set.size(); i++)
7      {
8
9          if (MQKPEvaluator::compare(set[i]->getFitness(), set[indexWorst]->
10             getFitness()) < 0)
11          {
12              indexWorst = i;
13          }
14      }
15
16      return indexWorst;
17 }
```

Listing 14: Método *indexWorst* de la clase *MQKPGeneticAlgorithm*

10. MQKPGeneticAlgorithm::selectNewPopulation

En la clase *MQKPGeneticAlgorithm*, tenemos un método que hemos de completar, cuyo nombre es: *selectNewPopulation*. El cometido de dicha función es seleccionar la nueva población, que pasará a ser la población actual.

La nueva población de descendientes (offspring), sustituirá a la población actual (padres), con la excepción de que si la mejor solución de la población actual, es mejor que la peor de las soluciones del vector de offspring, esta solución será sustituida en la nueva población por la mejor de la actual, es decir, guardamos el mejor parente y desecharmos al peor hijo, para conformar la nueva población.

```

1 void selectNewPopulation(vector<Solution *> &offspring)
2 {
3
4     unsigned int indexBestPop = indexBest(_population);
5     unsigned int indexBestOff = indexBest(offspring);
6
7     if (MQKPEvaluator::compare(_population[indexBestPop]->getFitness() , offspring
8 [indexBestOff]->getFitness()) > 0)
9     {
10
11         offspring[indexWorst(offspring)]->copy(*_population[indexBestPop]);
12
13     for (unsigned i = 0; i < _popSize; i++)
14     {
15         delete (_population.back());
16         _population.pop_back();
17     }
18
19     unsigned offSize = (unsigned)offspring.size();
20
21     for (unsigned i = 0; i < offSize; i++)
22     {
23         _population.push_back(offspring.back());
24         offspring.pop_back();
25     }
26 }
```

Listing 15: Método *selectNewPopulation* de la clase *MQKPGeneticAlgorithm*

11. MQKPGeneticAlgorithm::evaluate

En la función `evaluate`, se evalúan las soluciones del vector y y se les asigna el *fitness*. Evaluamos las soluciones que no tengan un *fitness* válido, para obtener un nuevo *fitness* y se actualiza la mejor solución, por si esta ha cambiado.

```

1 void evaluate(vector<Solution *> &set)
2 {
3
4     for (Solution *sol : set)
5     {
6         MQKPSolution *s = (MQKPSolution *)sol;
7         double fitness;
8         if (!(s->hasValidFitness())))
9         {
10             fitness = MQKPEvaluator::computeFitness(*_instance, *s);
11         }
12     }
13 }
```

```

12     _results.push_back(fitness);
13     s->setFitness(fitness);
14
15     if (MQKPEvaluator::compare(fitness, _bestSolution->getFitness()) > 0)
16     {
17         _bestSolution->copy(*s);
18     }
19 }
20 }
21 }
```

Listing 16: Método *evaluate* de la clase *MQKPGeneticAlgorithm*

12. MQKPGeneticAlgorithm::initPopulation

En esta función, se inicializa la población del algoritmo genético. Primero, se han de generar soluciones aleatorias y evaluar su fitness, una vez hecho esto guardamos la mejor solución obtenida. Por último almacenaremos en un vector de soluciones todas las soluciones obtenidas.

```

1   void initPopulation(unsigned popSize)
2   {
3
4     if (_instance == NULL)
5     {
6         cerr
7             << "The evolutionary algorithm has not been initialised. At least, its
8 _instance is NULL"
9             << endl;
10            exit(1);
11
12    for (unsigned i = 0; i < popSize; i++)
13    {
14
15        MQKPSolution *sol;
16        sol = new MQKPSolution(*_instance);
17        double fitness;
18
19        MQKPSolGenerator::genRandomSol(*_instance, *sol);
20
21        fitness = MQKPEvaluator::computeFitness(*_instance, *sol);
22
23        sol->setFitness(fitness);
24
25        if (MQKPEvaluator::compare(fitness, _bestSolution->getFitness()) > 0)
26        {
27
28            _bestSolution->copy(*sol);
29        }
30
31        _results.push_back(fitness);
32        _population.push_back(sol);
33    }
34 }
```

Listing 17: Método *initPopulation* de la clase *MQKPGeneticAlgorithm*

13. MQKPGeneticAlgorithm::run

En la función *run* de la clase *MQKPGeneticAlgorithm*, se ejecuta el algoritmo genético hasta que la función *stopCondition* considera que se ha cumplido la condición de parada.

Primero, inicializamos la población. Luego, realizamos el cruce, que da lugar a nuevas soluciones descendientes de las primeras, aunque hay la probabilidad de que puedan ser mutadas con la función *mutate*, tras el cruce. Por último, se evalúan las soluciones obtenidas con la función *evaluate*.

Tras esto, se almacena la media de los descendientes y se almacena junto con la mejor solución obtenida y seleccionamos la nueva solución obtenida como la solución actual.

Todo este proceso es realizado en bucle, hasta cumplir la condición de parada. Esto se puede comprobar en el código de la función, el cuál mostraremos a continuación:

```
1  virtual void run(MQKPStopCondition &stopCondition)
2  {
3
4      initPopulation(_popSize);
5
6      while (stopCondition.reached() == false)
7      {
8
9          _popMeanResults.push_back(computeMeanFitness(_population));
10         _bestPerIterations.push_back(
11             _population.at(indexBest(_population))->getFitness());
12
13         vector<Solution *> parents;
14         _selector->select(_population, parents);
15
16         vector<Solution *> offspring;
17         _crossoverOp->cross(parents, offspring);
18
19         _mutOp->mutate(offspring);
20
21         evaluate(offspring);
22         _offMeanResults.push_back(computeMeanFitness(offspring));
23
24         selectNewPopulation(offspring);
25
26         stopCondition.notifyIteration();
27     }
28
29     _popMeanResults.push_back(computeMeanFitness(_population));
30     _bestPerIterations.push_back(
31         _population.at(indexBest(_population))->getFitness());
32 }
```

Listing 18: Método *run* de la clase *MQKPGeneticAlgorithm*

14. MQKPAnt::sumSignificances

Para la creación de este método, primero tenemos que saber que es una clase dentro de otra, es decir, dentro de la clase *MQKPAntColonyOpt* nos encontramos con la clase *MQKPAnt*, la cuál irá buscando las distintas soluciones que podemos encontrar.

Esta clase recibirá un vector de tipo *double*, en el cuál se almacenará la suma de todas las soluciones. La solución obtenida es observable en el Listing 19.

```

1   double sumSignificances( vector<double> &significances )
2   {
3       double sum = 0;
4
5       for ( unsigned i = 0; i < significances.size(); i++)
6       {
7           sum += significances[ i ];
8       }
9
10      return sum;
11  }
```

Listing 19: Método *sumSignificances* de la clase MQKPAnt

15. MQKPAnt::createAlternatives

Este método, como en el caso anterior, pertenece a una clase que se encuentra dentro de otra clase. Este método se encarga de crear las distintas soluciones alternativas que podemos encontrar teniendo en cuenta la solución anterior. El Listing 20 muestra cómo hemos creado este método.

```

1   void createAlternatives(
2       vector<MQKPObjectAssignmentOperation *> &alternatives ,
3       vector<double> &significances )
4   {
5
6       MQKPIstance *instance = _colony->_instance ;
7       double alpha = _colony->_alpha ;
8       double beta = _colony->_beta ;
9       vector<vector<double> *> &phMatrix = _colony->_phMatrix ;
10
11      unsigned numKnapsacks = instance->getNumKnapsacks() ;
12      unsigned numTries = 0;
13
14      for ( auto indexObj : _objectsLeft )
15      {
16
17          for ( unsigned j = 1;
18              j <= numKnapsacks && numTries < _candidateListSize ;
19              j++)
20          {
21
22              double maxViolation =
23                  instance->getDeltaMaxCapacityViolation(* _sol ,
24                                              indexObj , j );
25
26              if ( maxViolation > 0)
27                  continue ;
28
29              double deltaFitness = instance->getDeltaSumProfits(* _sol ,
30                                              indexObj , j );
31              numTries++;
32
33              if ( deltaFitness <= 0)
34                  continue ;
35
36              MQKPObjectAssignmentOperation *al =
37                  new MQKPObjectAssignmentOperation();
38              double density = deltaFitness / instance->getWeight(indexObj); //Es
39              el valor heuristico
40              double relevance = pow(density , beta) * pow(phMatrix.at(indexObj)->
41              at(j) , alpha);
42              al->setValues(indexObj , j , deltaFitness);
43              alternatives.push_back(al);
```

```

42         significances.push_back(relevance);
43     }
44 }
45 }
```

Listing 20: Método *createAlternatives* de la clase MQKPAnt

16. MQKPAnt::selectBestAlternative

Para la creación de este método, necesitaremos un objeto de la clase *MQKPObjetAssignmentOperation*, el cuál se encargará de comprobar cuál es la mejor alternativa encontrada. Dicha solución será guardada en el objeto de entrada para poder así utilizarla posteriormente. El Listing 21 muestra el código resultante.

```

1   void selectBestAlternative(MQKPObjetAssignmentOperation &op)
2   {
3       MQKPIstance *instance = _colony->_instance;
4       vector<vector<double> *> &phMatrix = _colony->_phMatrix;
5       double beta = _colony->_beta;
6       double alpha = _colony->_alpha;
7
8       unsigned numKnapsacks = instance->getNumKnapsacks();
9       double bestSignificance = -1;
10      unsigned numTries = 0;
11
12      for (auto indexObj : _objectsLeft)
13      {
14
15          for (unsigned j = 1;
16              j <= numKnapsacks && numTries < _candidateListSize;
17              j++)
18      {
19
20          double maxViolation =
21              instance->getDeltaMaxCapacityViolation(*_sol,
22                                              indexObj, j);
23
24          if (maxViolation > 0)
25              continue;
26
27          double deltaFitness = instance->getDeltaSumProfits(*_sol,
28                                              indexObj, j);
29          numTries++;
30
31          if (deltaFitness <= 0)
32              continue;
33
34          double density = deltaFitness / instance->getWeight(indexObj); //Es
35          el valor heuristico
36          double relevance = pow(density, beta) * pow(phMatrix[indexObj]->at(j),
37                                  alpha);
38          if (bestSignificance < relevance)
39          {
40              bestSignificance = relevance;
41              op.setValues(indexObj, j, deltaFitness);
42          }
43      }
44  }
```

Listing 21: Método *selectBestalternative* de la clase MQKPAnt

17. MQKPAnt::resetSolution

Este método, al contrario que los tres anteriores, pertenece a la parte pública de la clase *MQKPAnt*. Este método se encargará de resetear la variable solución de esta clase para poder así volver a crear una colección se soluciones distintas. El resultado es observable en el Listing 22.

```
1     void resetSolution()
2     {
3         unsigned numObjs = _colony->_instance->getNumObjs();
4
5         for (unsigned i = 0; i < numObjs; i++)
6         {
7
8             _sol->putObjectIn(i, 0);
9             _objectsLeft.insert(i);
10        }
11        _sol->setFitness(0);
12    }
```

Listing 22: Método *resetSolution* de la clase MQKPAnt

18. MQKPAnt::chooseOperation

Este método, también perteneciente a la clase *MQKPAnt*, se encargará de seleccionar la operación más oportuna a realizar sobre nuestros individuos. Primero creará las diferentes soluciones alternativas. Seguidamente las sumará y comprobará si son óptimas o no, es decir, mejoran la solución anterior. En caso de hacerlo, se guardará dicha operación para realizarla más a delante. El código resultante es observable en Listing 23.

```
1     void chooseOperation(MQKPOperationAssignmentOperation &operation)
2     {
3
4         double randSample = (((double)rand()) / RAND_MAX);
5
6         if (randSample < _colony->q0)
7         {
8             selectBestAlternative(operation);
9         }
10        else
11        {
12
13            vector<MQKPOperationAssignmentOperation *> alternatives;
14            vector<double> significances;
15            createAlternatives(alternatives, significances);
16
17            if (significances.size() <= 0)
18            {
19                return;
20            }
21
22            double v_sumSignificances = sumSignificances(significances);
23            double randSample = (((double)rand()) / RAND_MAX) * v_sumSignificances;
24            randSample -= significances.at(0);
25            unsigned opSelected = 0;
26
27            while (randSample > 0)
28            {
29                opSelected++;
30                randSample -= significances.at(opSelected);
31            }
32
33            unsigned indexObj = alternatives.at(opSelected)->getObj();
```

```

34     unsigned indexKnapsack =
35         alternatives .at( opSelected )->getKnapsack() ;
36     double deltaFitness =
37         alternatives .at( opSelected )->getDeltaFitness() ;
38     operation .setValues( indexObj , indexKnapsack , deltaFitness );
39
40     freeAlternatives( alternatives );
41 }
42
43 if ( operation .getObj() >= 0 )
44 {
45     operation .apply( getSolution() );
46
47     _objectsLeft .erase( operation .getObj() );
48 }
49 }
```

Listing 23: Método *chooseOperation* de la clase MQKPAnt

19. MQKPAntColonyOpt::localUpdate

Este método pertenece a la clase *MQKPAntColonyOpt* el cuál actualizará de manera local las soluciones encontradas y las guardará en una matriz de tipo *double*. La función resultante es observable en el Listing 24.

```

1   void localUpdate( MQKPObjectAssignmentOperation &op )
2   {
3       _phMatrix[ op .getObj() ]->at( op .getKnapsack() ) = ( 1 - _evaporation ) * _phMatrix
4           [ op .getObj() ]->at( op .getKnapsack() ) + _evaporation * _initTau ;
5   }
```

Listing 24: Método *localUpdate* de la clase MQKPAntColonyOpt

20. MQKPAntColonyOpt::releaseAnts

Este método, también perteneciente a la clase *MQKPAntColonyOpt*, se encarga de simular cómo las hormigas empezarían a buscar las soluciones. Primero inicializamos todas las soluciones, seguidamente, calcularemos el *fitness* de cada una de ellas y finalmente guardaremos las soluciones obtenidas. El Listing 25 muestra cómo hemos abordado la solución.

```

1   void releaseAnts()
2   {
3
4       unordered_set<unsigned> movingAnts;
5       unordered_set<unsigned> stoppedAnts;
6       int i = 0;
7
8       for ( auto ant : _ants )
9       {
10          ant->resetSolution();
11          movingAnts .insert( i );
12          i++;
13      }
14
15      while ( movingAnts .size() > 0 )
16      {
17          stoppedAnts .clear();
18
19          for ( auto iAnt : movingAnts )
20          {
```

```

21 MQKPAnt *ant = _ants[iAnt];
22 MQKPObjectAssignmentOperation op;
23 op.setValues(-1, -1, 0);
24
25 ant->chooseOperation(op);
26 if (op.getObj() != -1)
27 {
28     localUpdate(op);
29 }
30 else
31 {
32     stoppedAnts.insert(iAnt);
33 }
34 }
35
36 for (auto iAnt : stoppedAnts)
37 {
38     movingAnts.erase(iAnt);
39 }
40 }
41
42 double bestFitness = _bestSolution->getFitness();
43
44 for (auto ant : _ants)
45 {
46     MQKPSolution &sol = ant->getSolution();
47     double currentFitness = ant->getSolution().getFitness();
48
49     if (MQKPEvaluator::compare(currentFitness, bestFitness) > 0)
50     {
51         _bestSolution->copy(sol);
52         bestFitness = currentFitness;
53     }
54 }
55 }
```

Listing 25: Método *releaseAnt* de la clase MQKPAntColonyOpt

21. MQKPAntColonyOpt::iterate

Este método se encargará de iterar sobre las diferentes soluciones simulando el camino que tomaría una hormiga para encontrar la solución. Las distintas soluciones se guardarán en una matriz. Esta método también pertenece a la clase *MQKPAntColonyOpt*. La solución obtenida es observable en el Listing 26.

```

1 void iterate()
2 {
3
4     releaseAnts();
5     saveStatistics();
6
7     unsigned numObjs = _instance->getNumObjs();
8     double fitness = _bestSolution->getFitness();
9
10    for (unsigned i = 0; i < numObjs; i++)
11    {
12        _phMatrix.at(i)->at(_bestSolution->whereIsObject(i)) = (1 - _evaporation)
13        * _phMatrix.at(i)->at(_bestSolution->whereIsObject(i)) + fitness;
14    }
}
```

Listing 26: Método *iterate* de la clase MQKPAntColonyOpt

22. MQKPAntColonyOpt::initialise

Este método, que pertenece a la parte pública de la clase *MQKPAntColonyOpt* se encarga de inicializar el algoritmo para poder ir encontrando las posibles soluciones al problema de la mochila. Primero inicializamos todas las variables, calculamos el *fitness* de dicha solución y finalmente la almacenaremos para poder mostrarla al final. El código resultante es observable en el Listing 27.

```

1   void initialise (unsigned numAnts, double q0, double alpha, double beta,
2                     double initTau, double evaporation, unsigned candidateListSize ,
3                     MQKPIstance &instance)
4   {
5     _instance = &instance ;
6     _q0 = q0 ;
7     _alpha = alpha ;
8     _beta = beta ;
9     _initTau = initTau ;
10    _evaporation = evaporation ;
11
12    if (numAnts <= 0)
13    {
14      cerr << "The number of ants must be greater than 0" << endl;
15      exit(1);
16    }
17
18    if (_bestSolution != NULL)
19    {
20      delete _bestSolution ;
21      _bestSolution = NULL;
22    }
23
24    _bestSolution = new MQKPSolution(*_instance) ;
25    MQKPSolGenerator::genRandomSol(*_instance , *_bestSolution) ;
26    double fitness = MQKPEvaluator::computeFitness(*_instance ,
27                                                    *_bestSolution) ;
28    _bestSolution->setFitness(fitness) ;
29
30    for (unsigned i = 0; i < numAnts; i++)
31    {
32      MQKPAnt *hormiga = new MQKPAnt(candidateListSize , this) ;
33      _ants.push_back(hormiga) ;
34    }
35
36    unsigned numObjs = _instance->getNumObjs() ;
37    unsigned numKnapsacks = _instance->getNumKnapsacks() + 1;
38
39    for (unsigned i = 0; i < numObjs; i++)
40    {
41      vector<double> *aVector = new vector<double> ;
42      _phMatrix.push_back(aVector) ;
43
44      for (unsigned j = 0; j < numKnapsacks; j++)
45      {
46        aVector->push_back(_initTau) ;
47      }
48    }
49 }
```

Listing 27: Método *initialise* de la clase MQKPAntColonyOpt

23. MQKPAntColonyOpt::run

Esta función se encarga de ejecutar la metaheurista ACO sobre el problema de la mochila. En el caso de que la condición de parada no se cumpla, este método iterará sobre las distintas soluciones para llegar a la solución óptima. Como los anteriores métodos, esta función pertenece a la clase *MQKPAntColonyOpt*, más específicamente a su parte pública. La solución para esta función es observable en el Listing 28.

```
1   virtual void run(MQKPStopCondition &stopCondition)
2   {
3
4       if (_instance == NULL)
5       {
6           cerr << "The ACO algorithm has not been initialised" << endl;
7           exit(1);
8       }
9
10      //TODO Mientras no se llegue a la condición de parada, iterar
11      while (stopCondition.reached() == false)
12      {
13          iterate();
14          stopCondition.notifyIteration();
15      }
16 }
```

Listing 28: Método *run* de la clase MQKPAntColonyOpt

24. Experimentos y resultados

24.1. Gráficas

1. Conjunto de datos jeu_100_25_4:

- Resultados con tres mochilas:

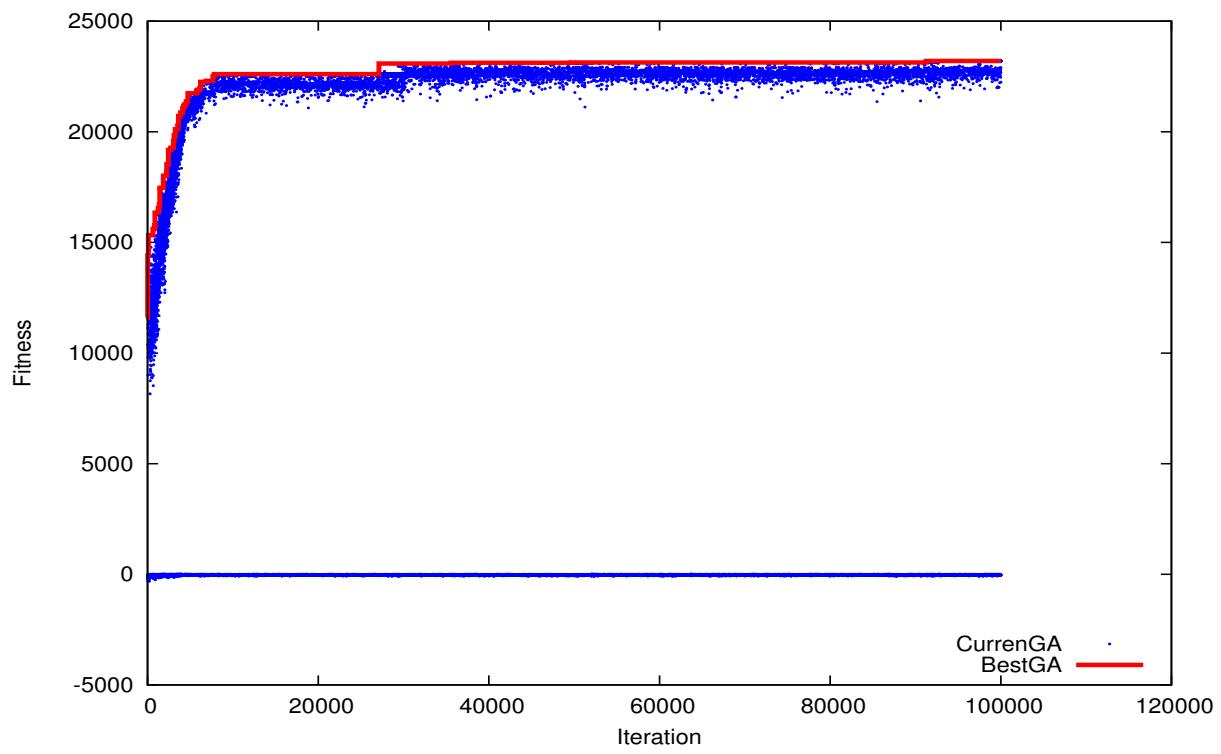


Figura 1: Metaheurísticas generadas por GA

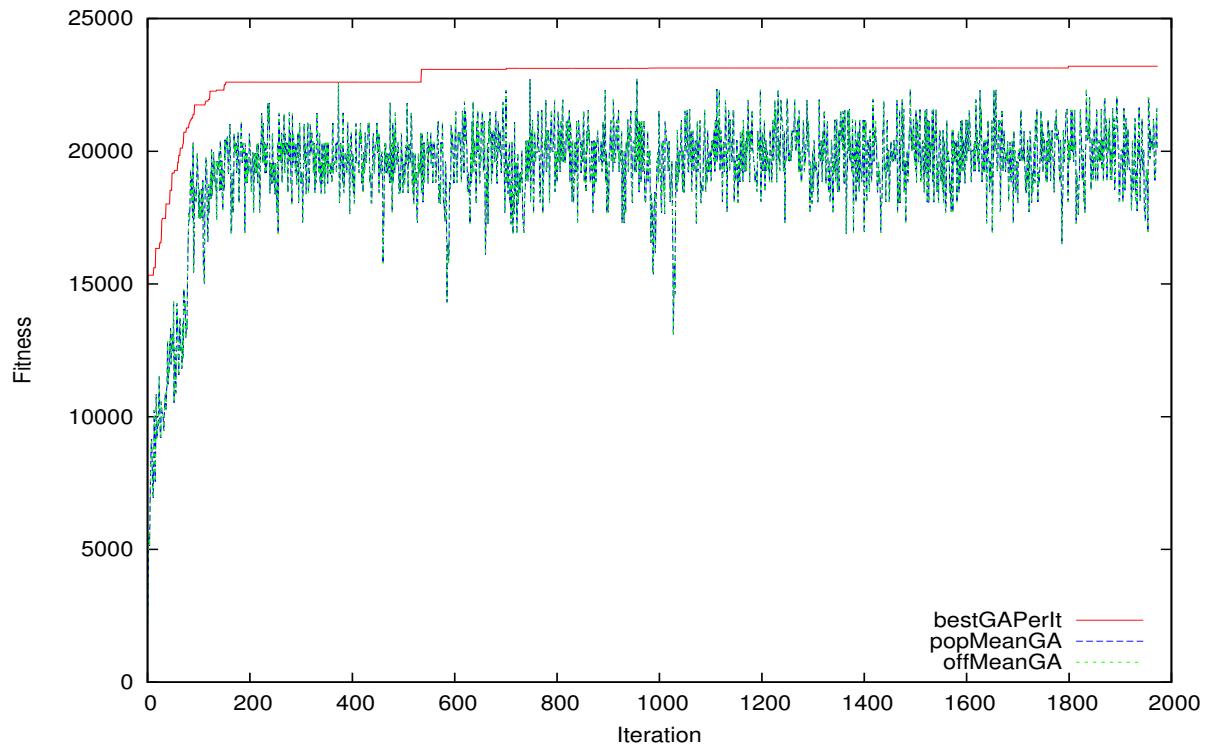


Figura 2: Metaheurísticas generadas por GA

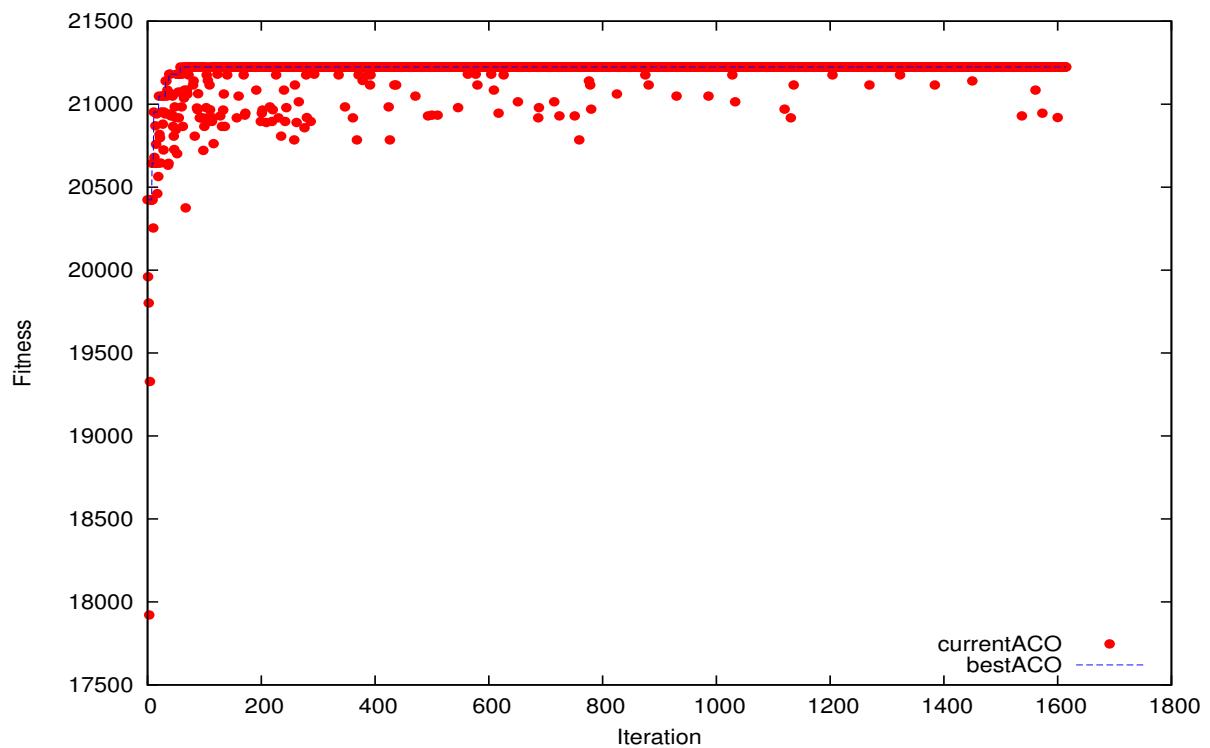


Figura 3: Metaheurísticas generadas por el ACO

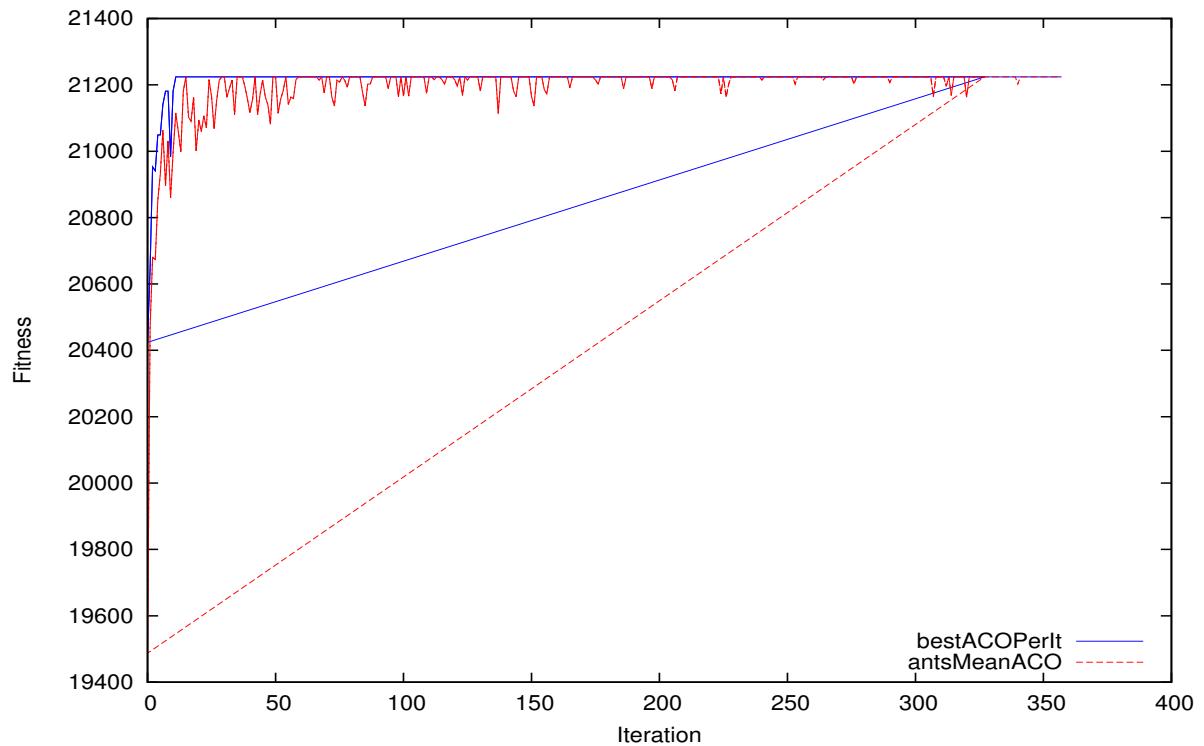


Figura 4: Metaheurísticas generadas por las hormigas

■ Resultados con cinco mochilas:

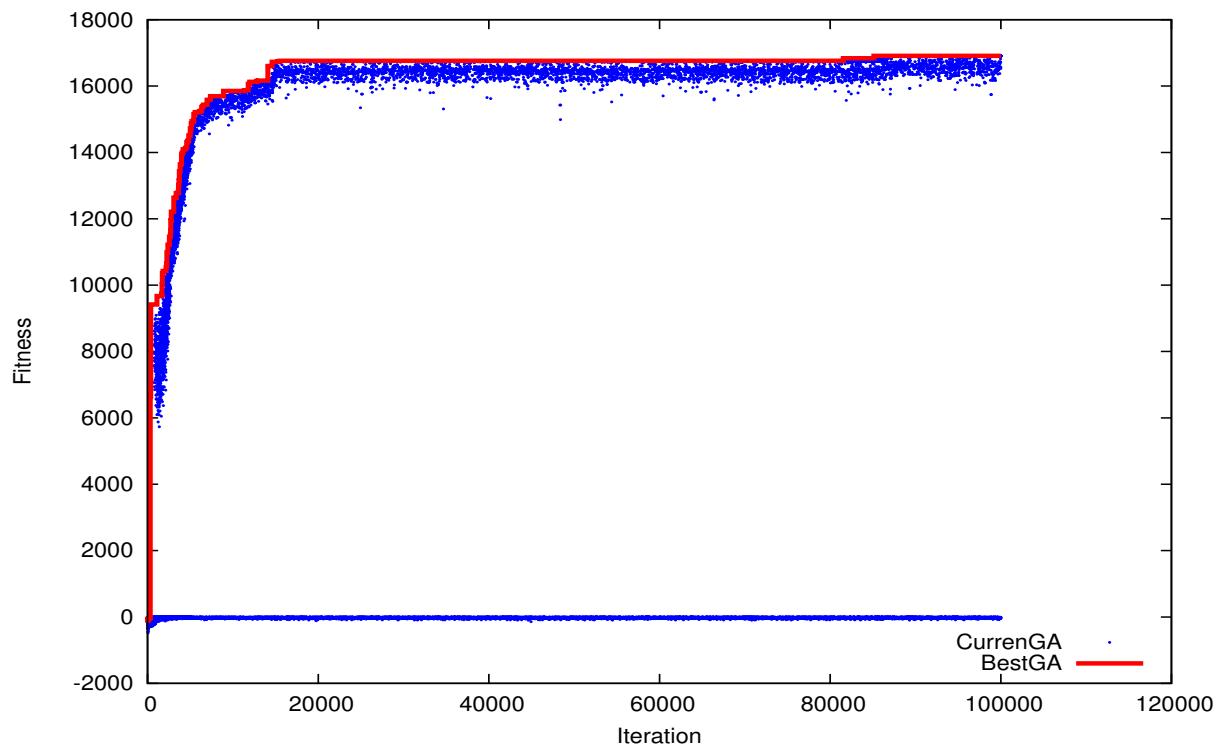


Figura 5: Metaheurísticas generadas por el Genético con cinco mochilas

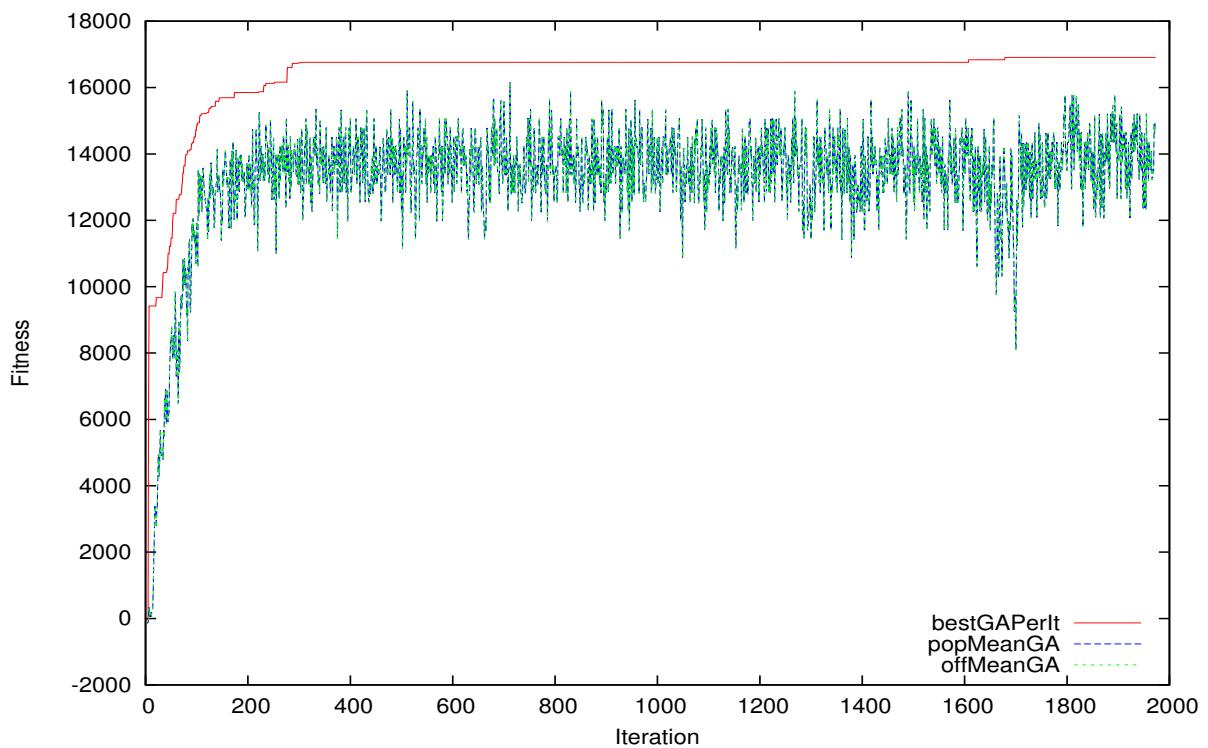


Figura 6: Metaheurísticas de la población con cinco mochilas

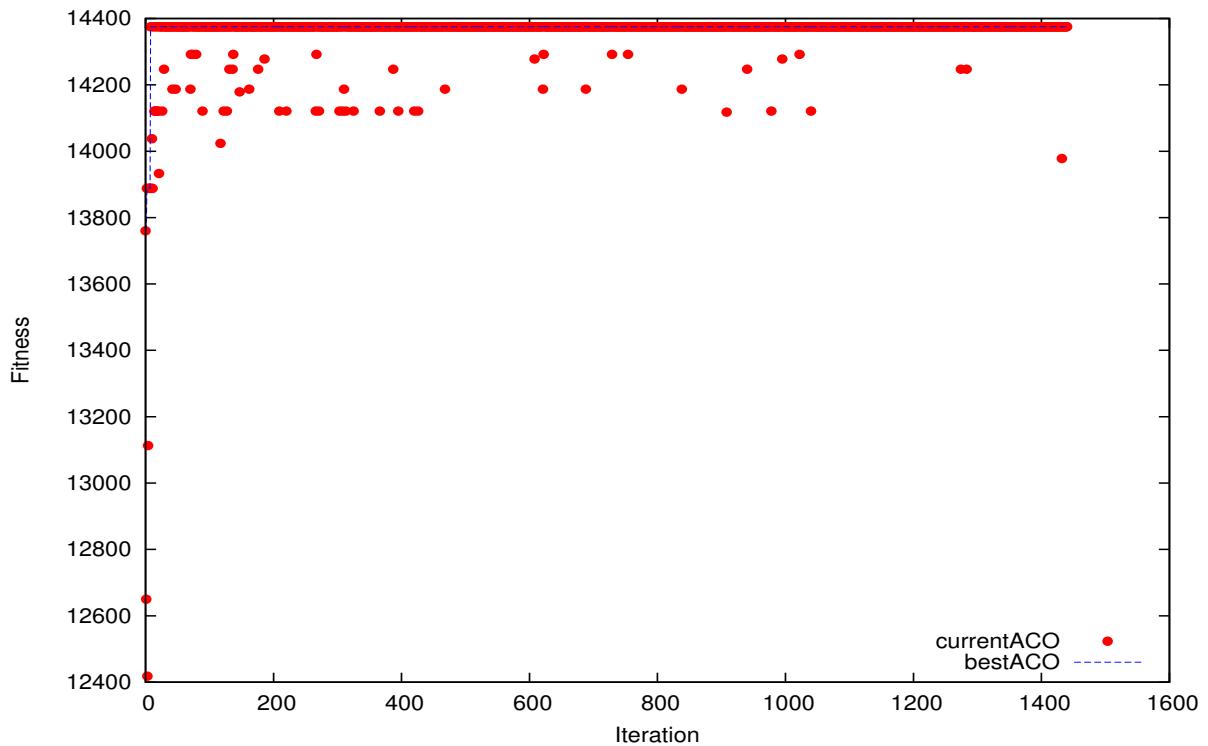


Figura 7: Metaheurísticas generadas por el ACO con cinco mochilas

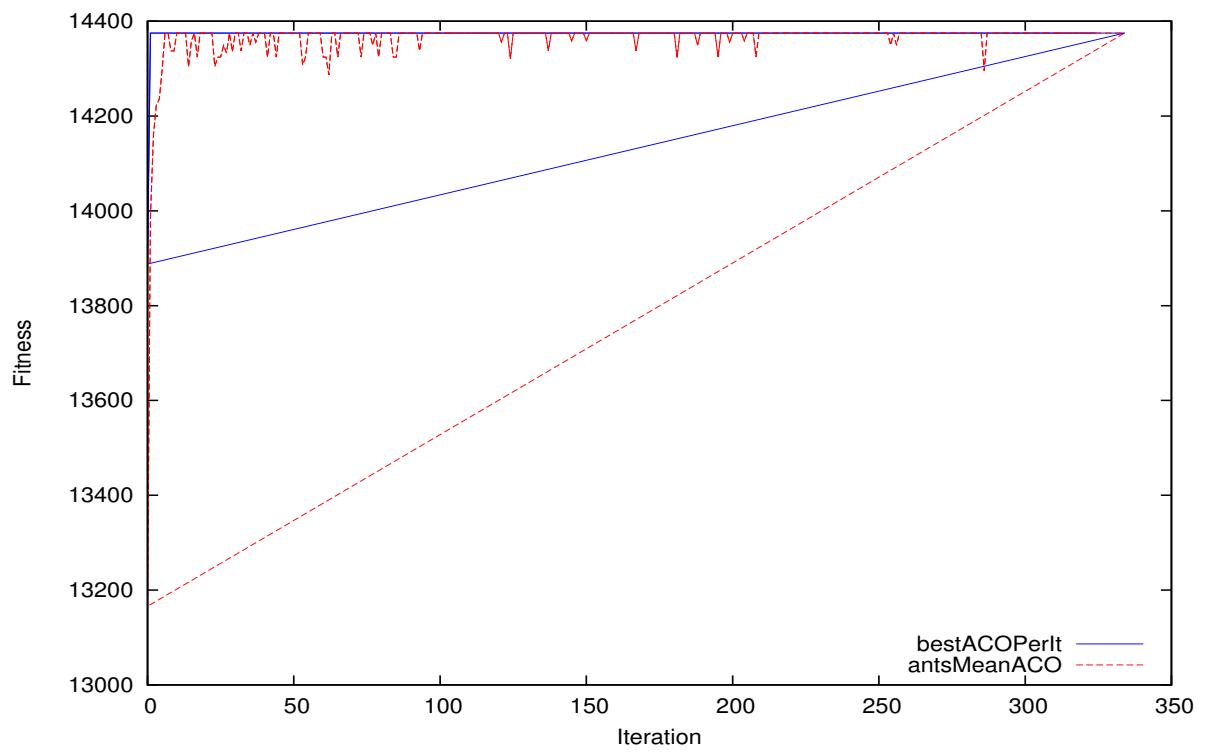


Figura 8: Metaheurísticas generadas por las hormigas con cinco mochilas

2. Conjunto de datos jeu_100_75_2:

- Resultados con tres mochilas:

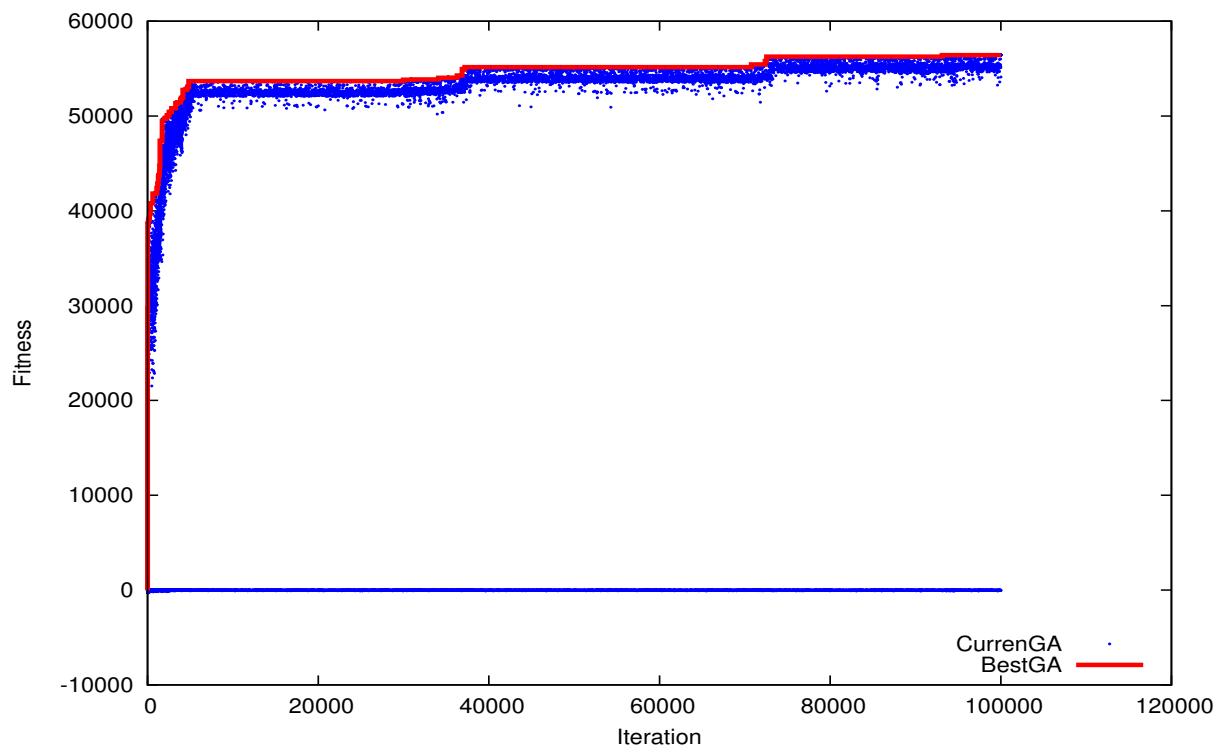


Figura 9: Metaheurísticas generadas por el Genético

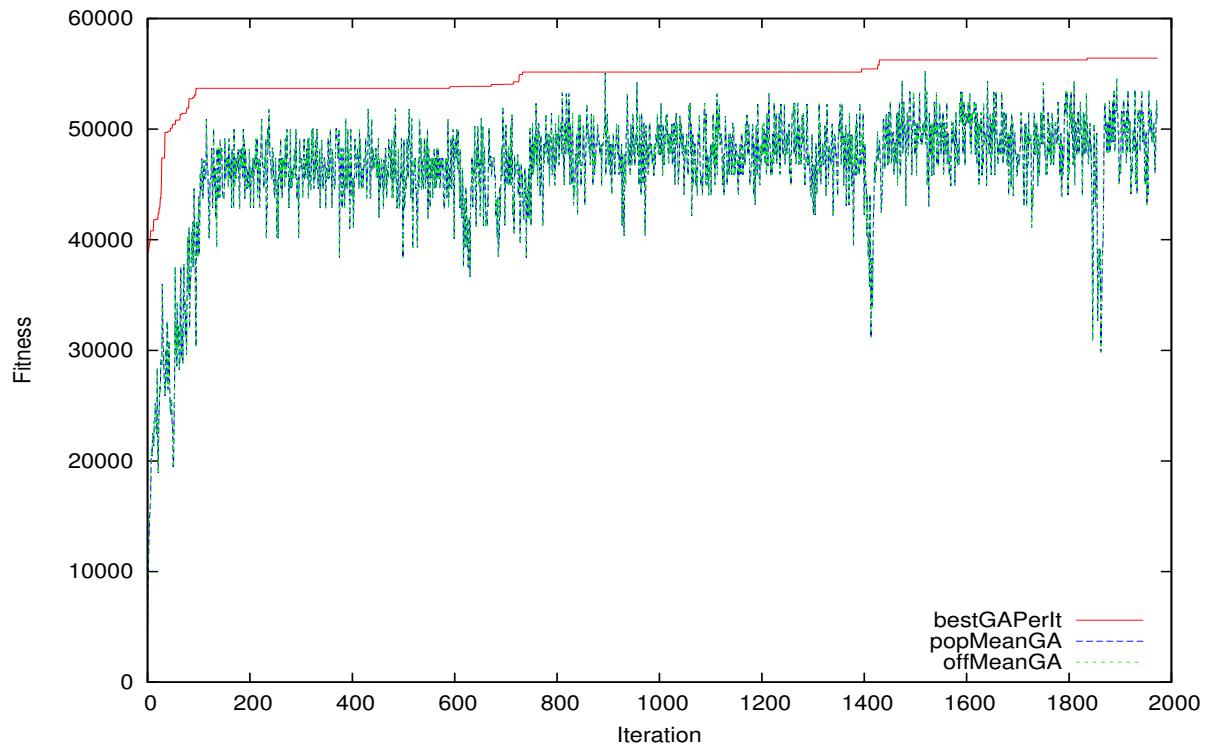


Figura 10: Metaheurísticas de la población

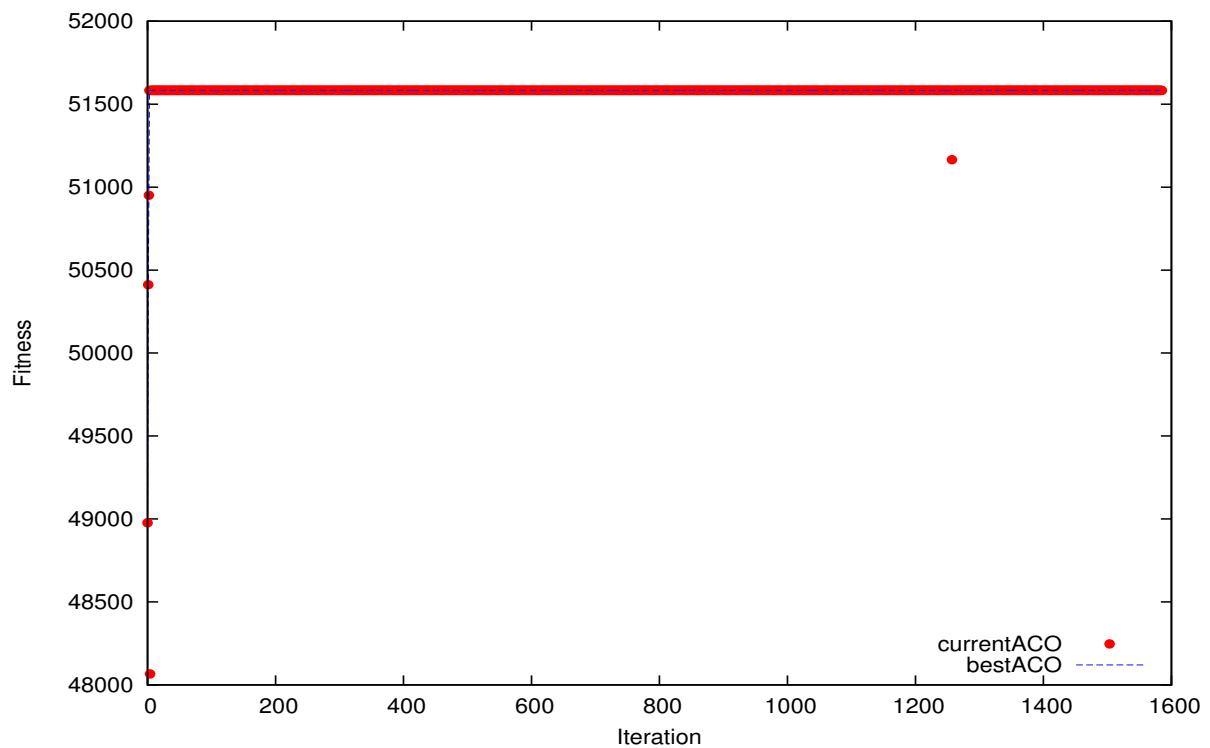


Figura 11: Metaheurísticas generadas por el ACO

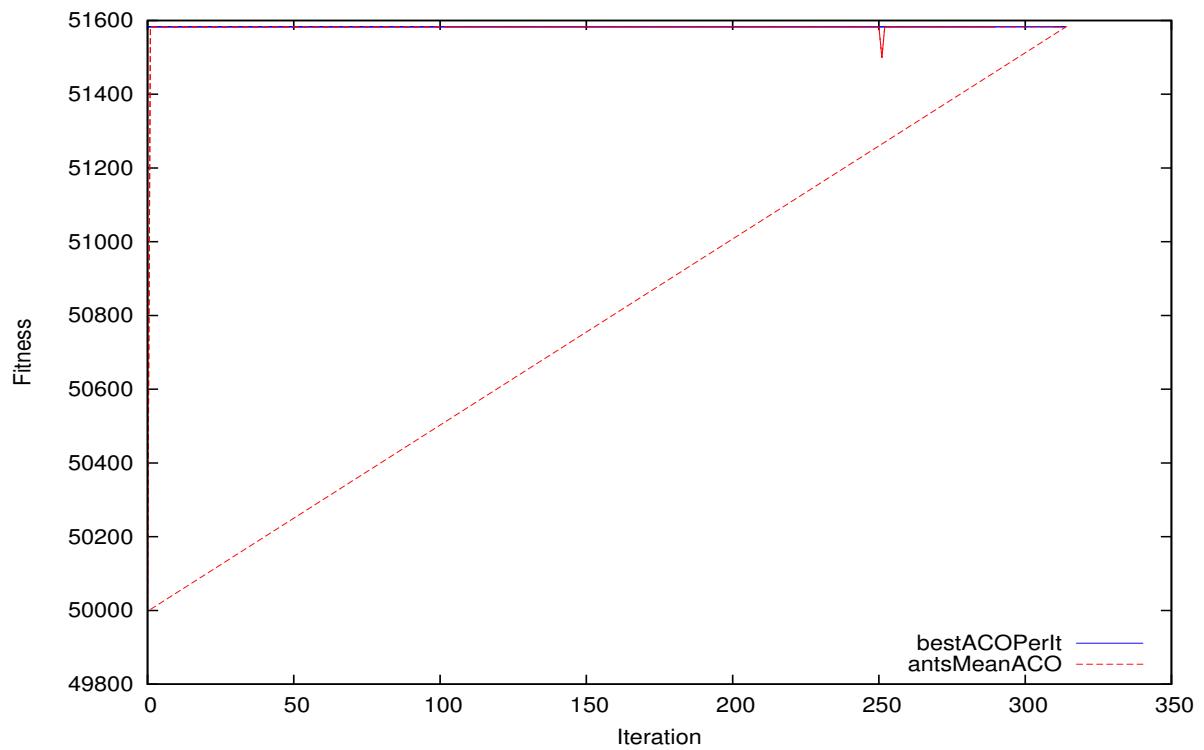


Figura 12: Metaheurísticas generadas por las hormigas

■ Resultados con cinco mochilas:

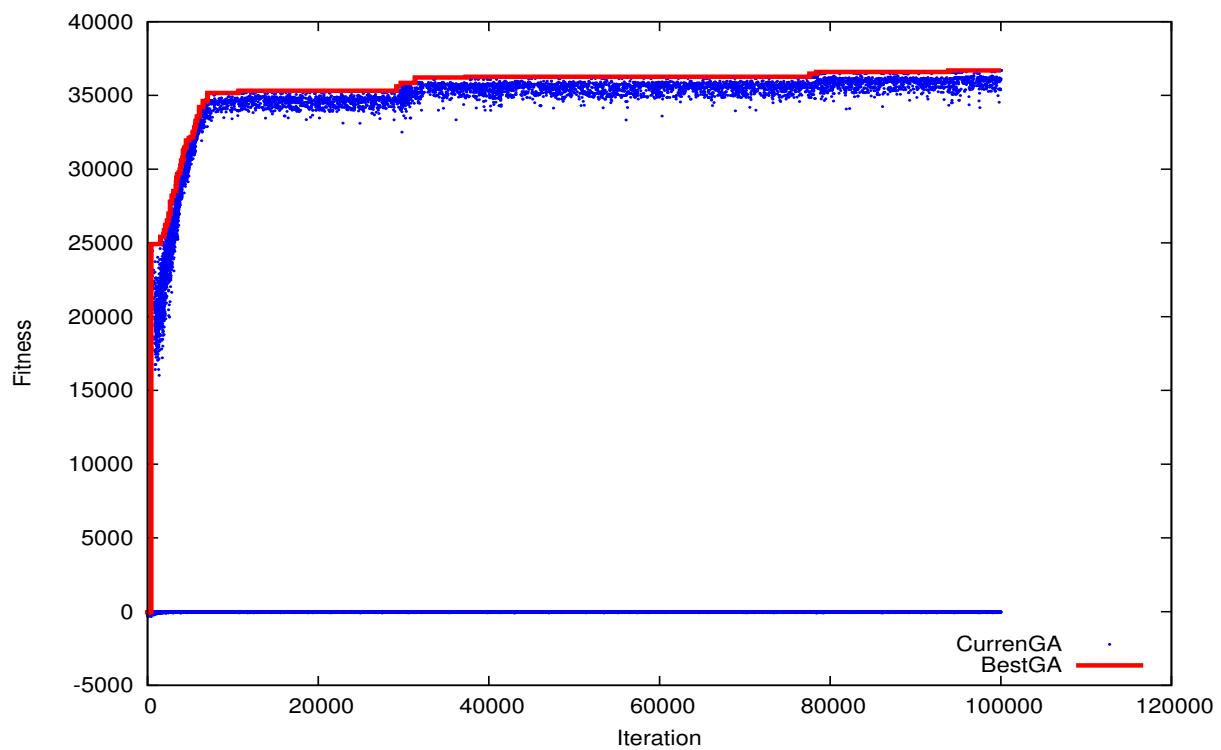


Figura 13: Metaheurísticas generadas por el Genético con cinco mochilas

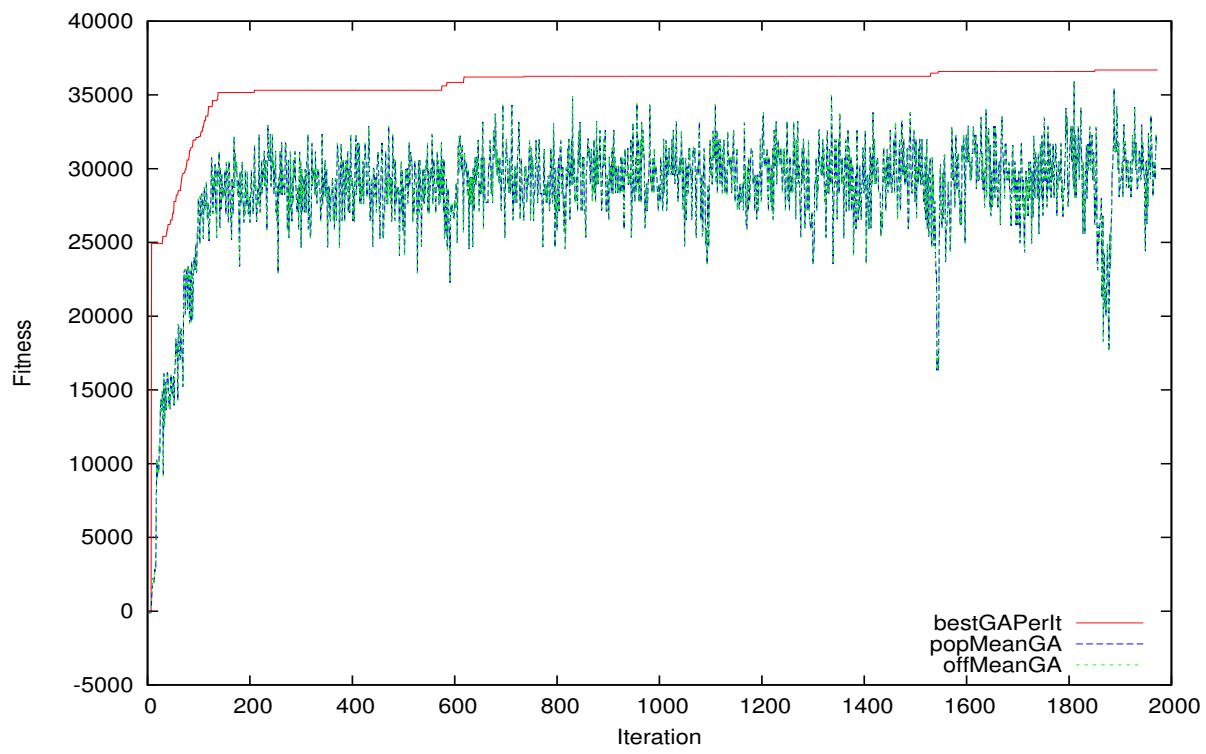


Figura 14: Metaheurísticas de la población con cinco mochilas

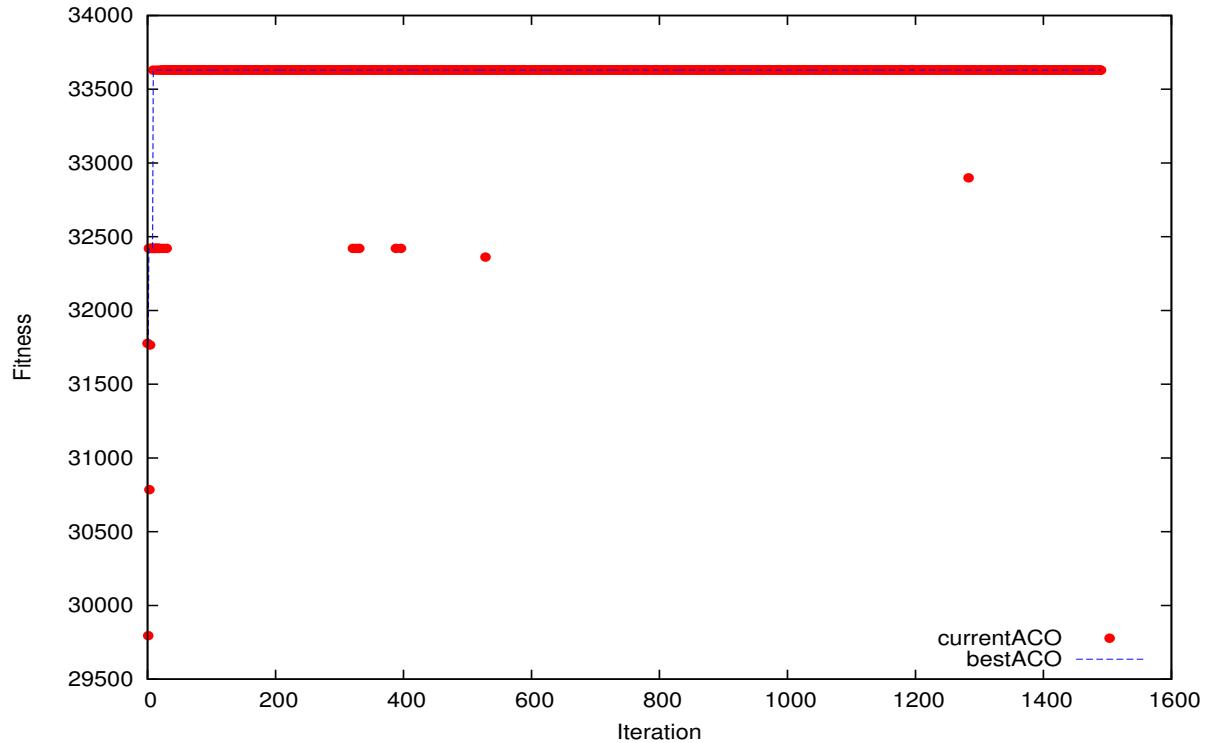


Figura 15: Metaheurísticas generadas por el ACO con cinco mochilas

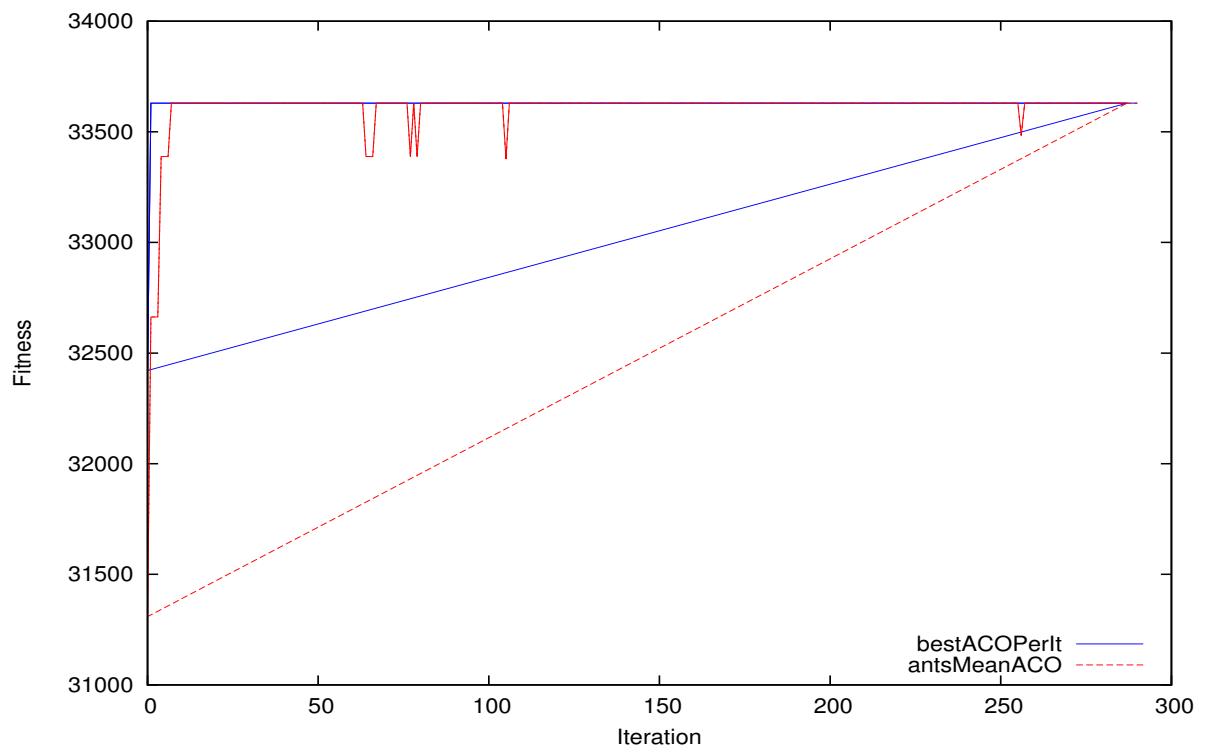


Figura 16: Metaheurísticas generadas por las hormigas con cinco mochilas

3. Conjunto de datos jeu_200_25_8:

- Resultados con tres mochilas:

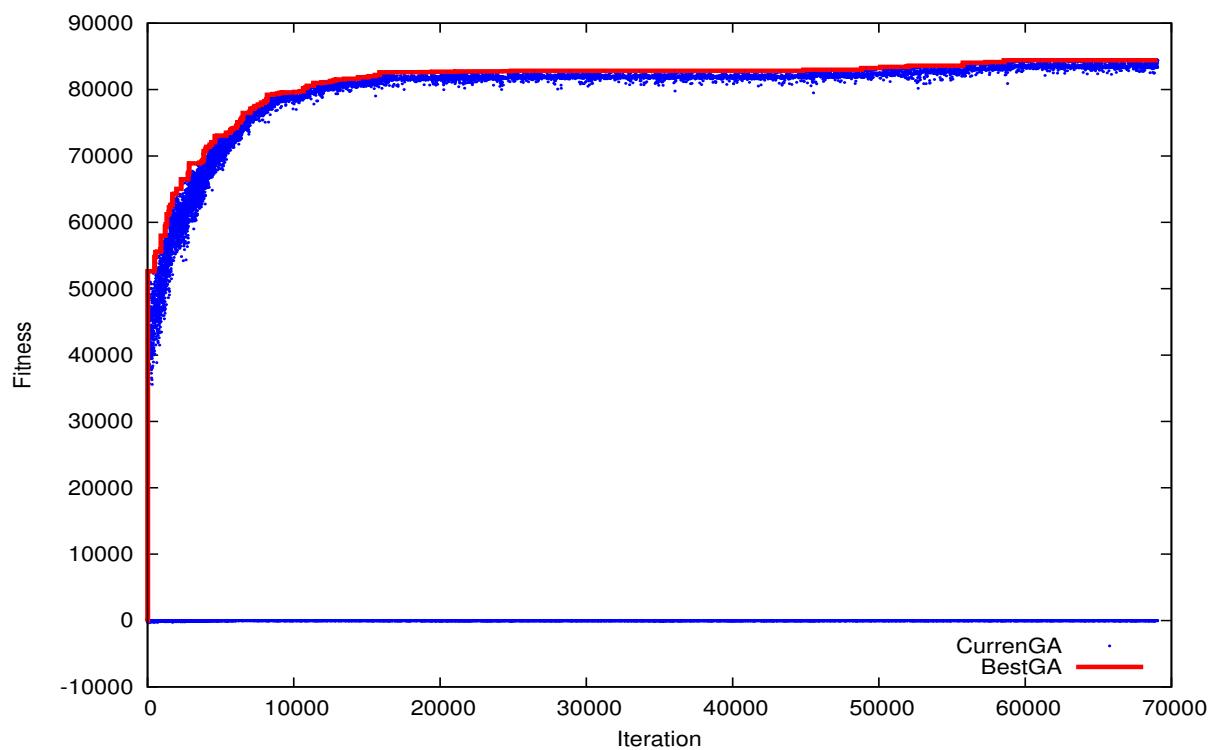


Figura 17: Metaheurísticas generadas por el Genético

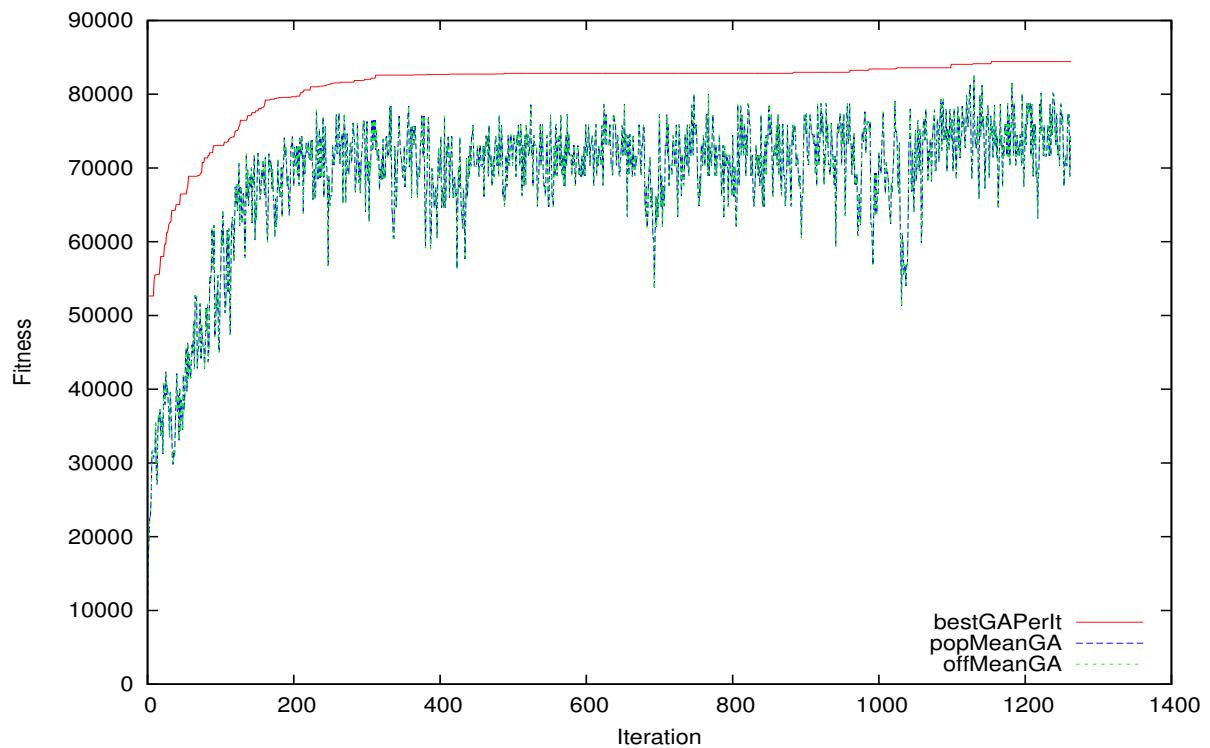


Figura 18: Metaheurísticas de la población

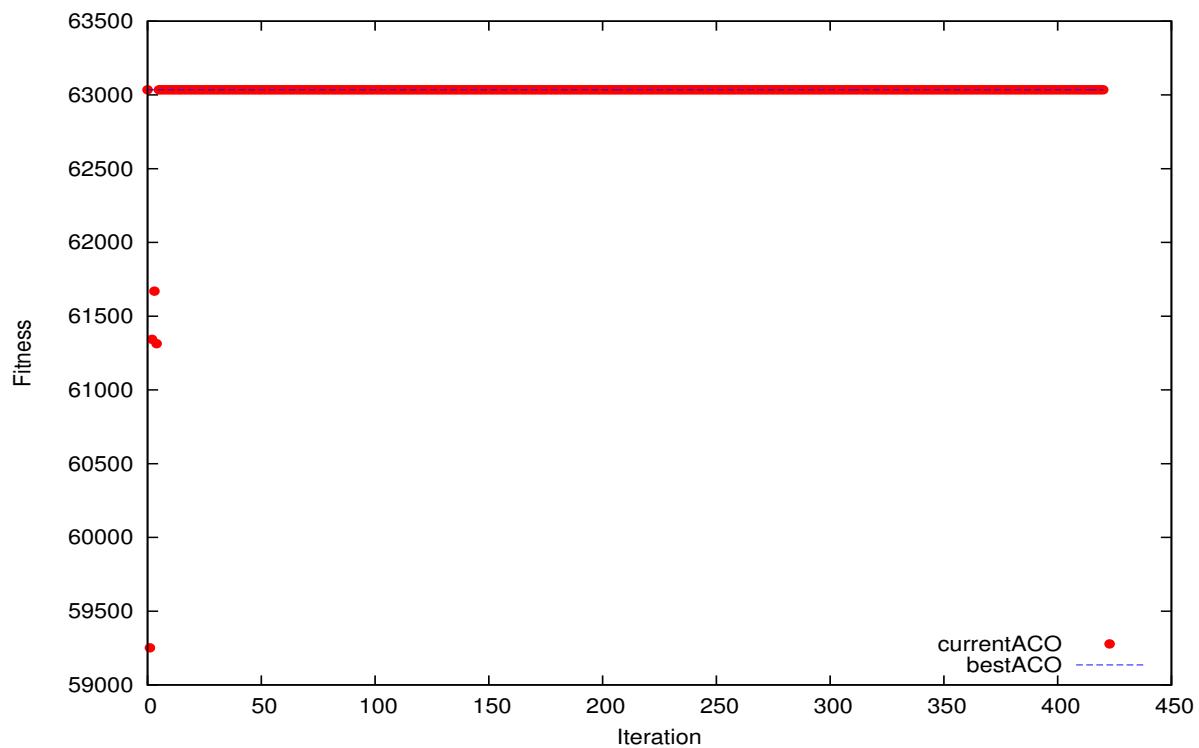


Figura 19: Metaheurísticas generadas por el ACO

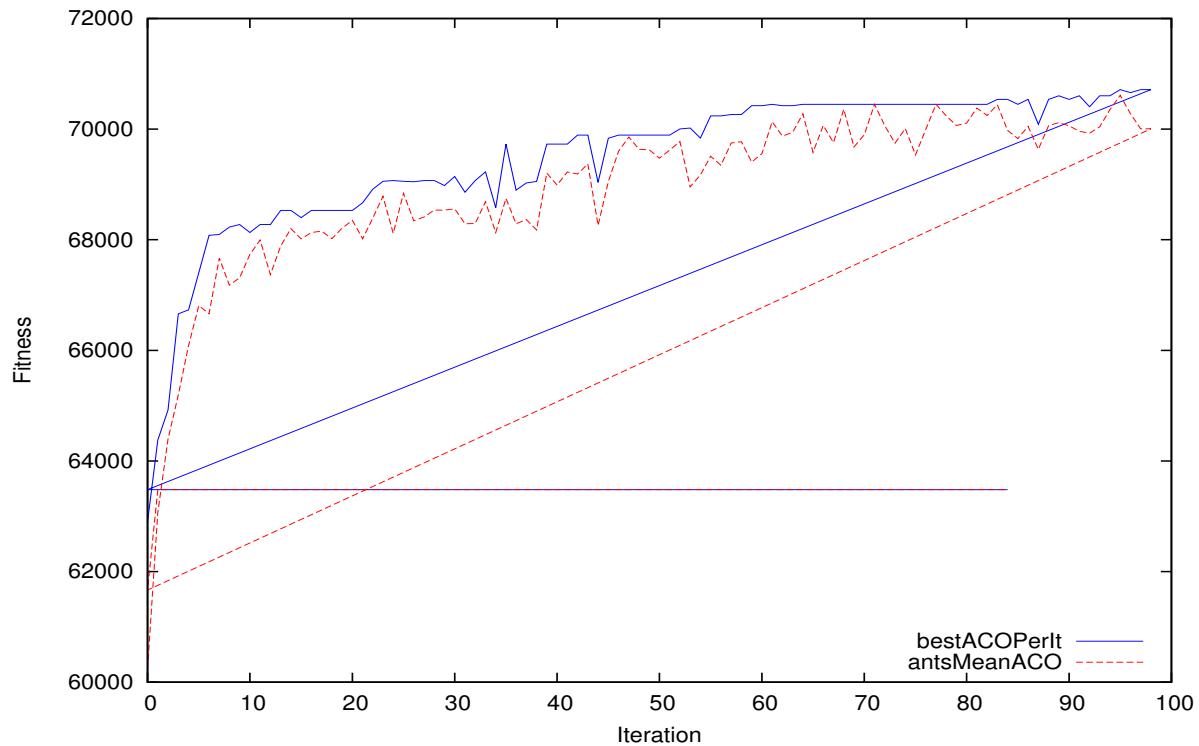


Figura 20: Metaheurísticas generadas por las hormigas

■ Resultados con cinco mochilas:

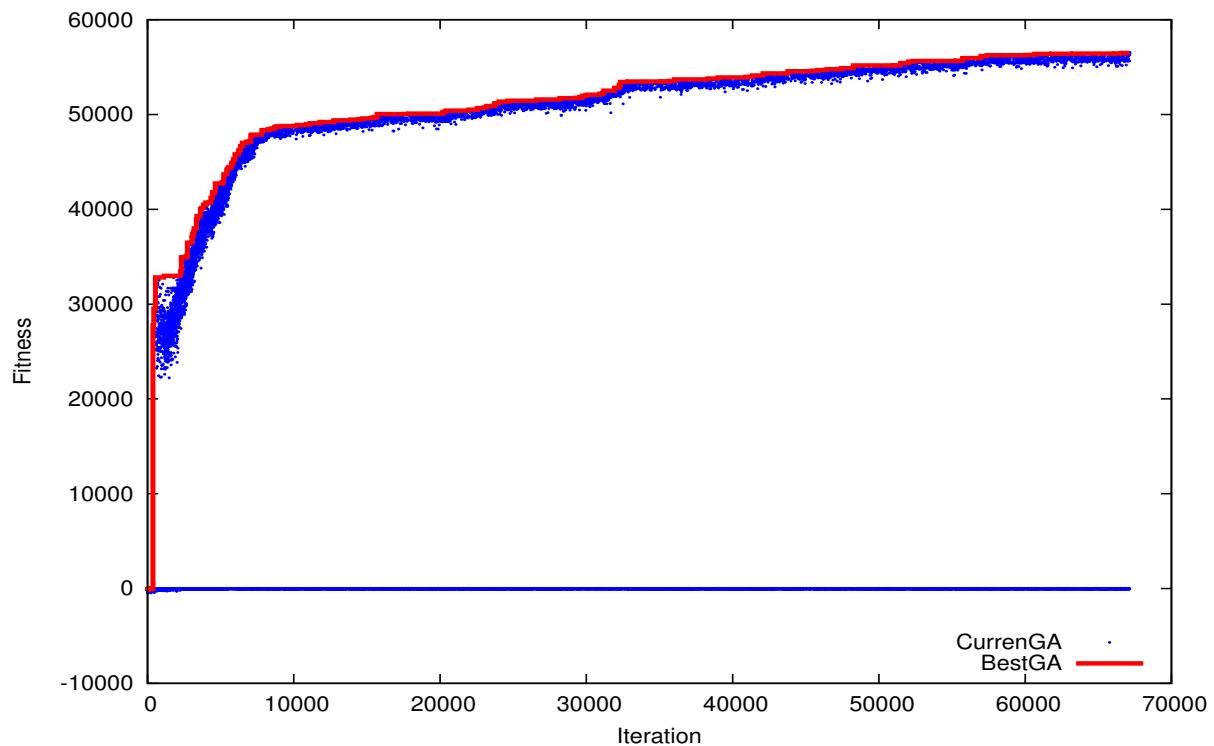


Figura 21: Metaheurísticas generadas por el Genético con cinco mochilas

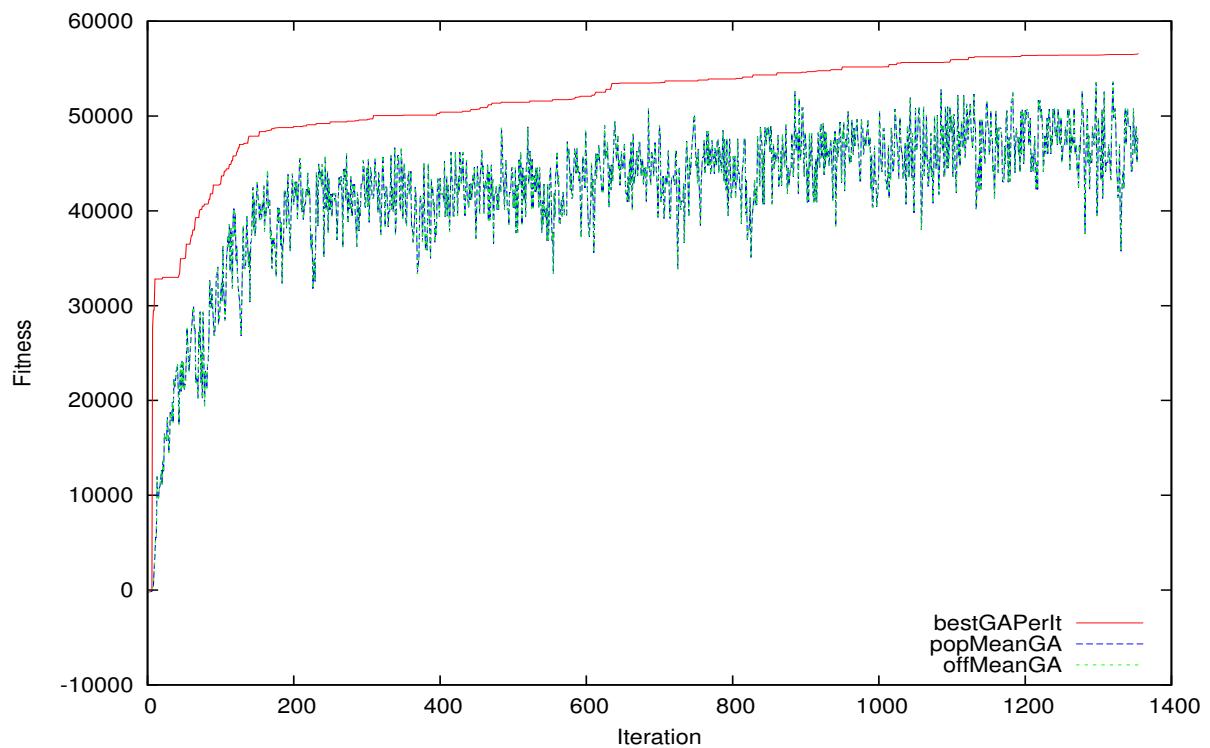


Figura 22: Metaheurísticas de la población con cinco mochilas

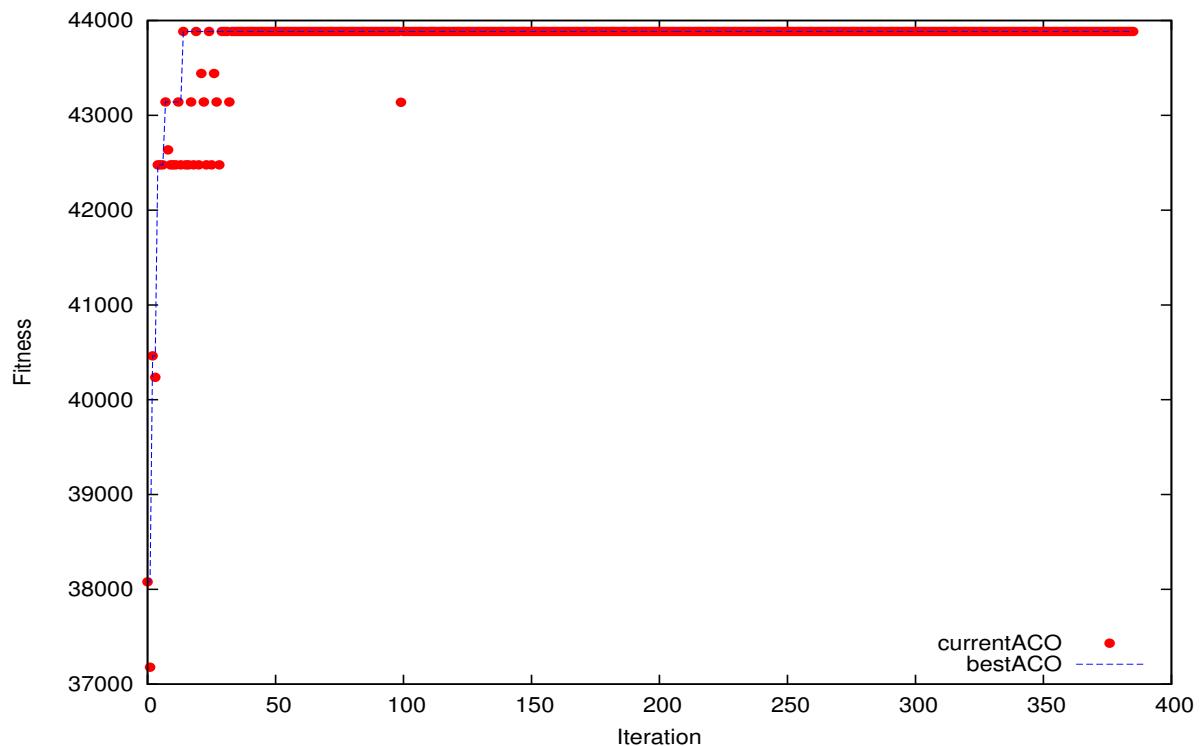


Figura 23: Metaheurísticas generadas por el ACO con cinco mochilas

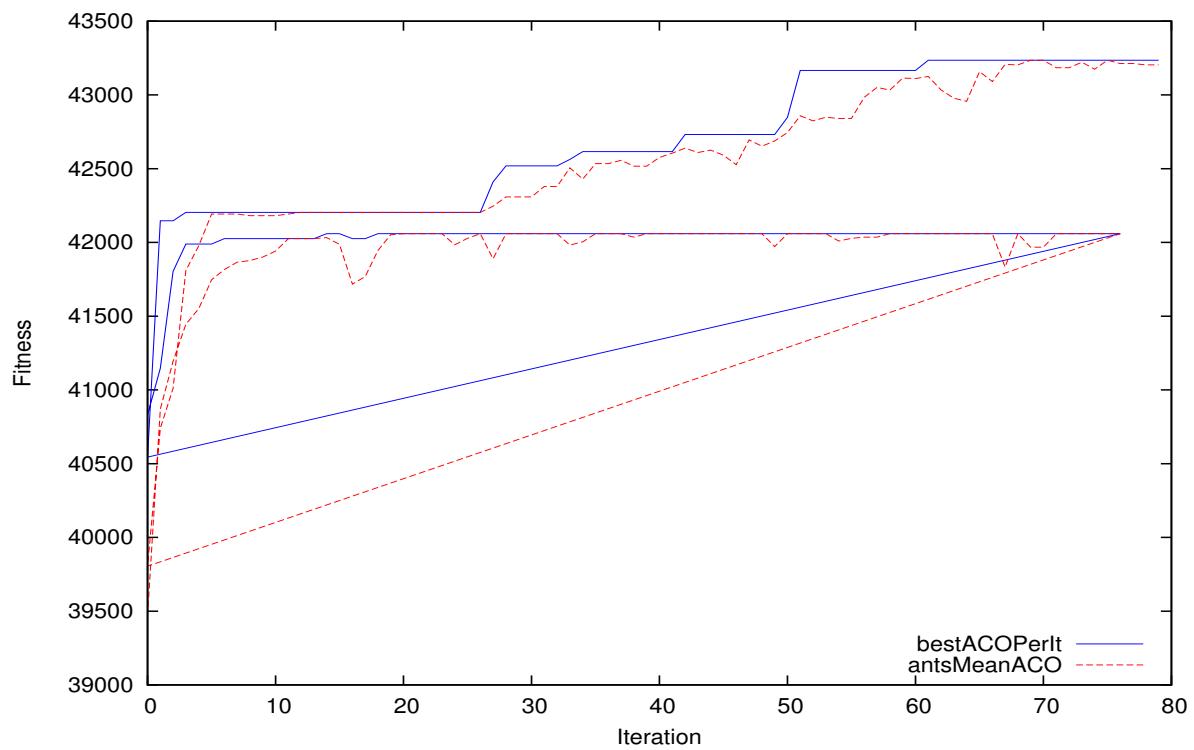


Figura 24: Metaheurísticas generadas por las hormigas con cinco mochilas

4. Conjunto de datos jeu_200_75_5:

- Resultados con tres mochilas:

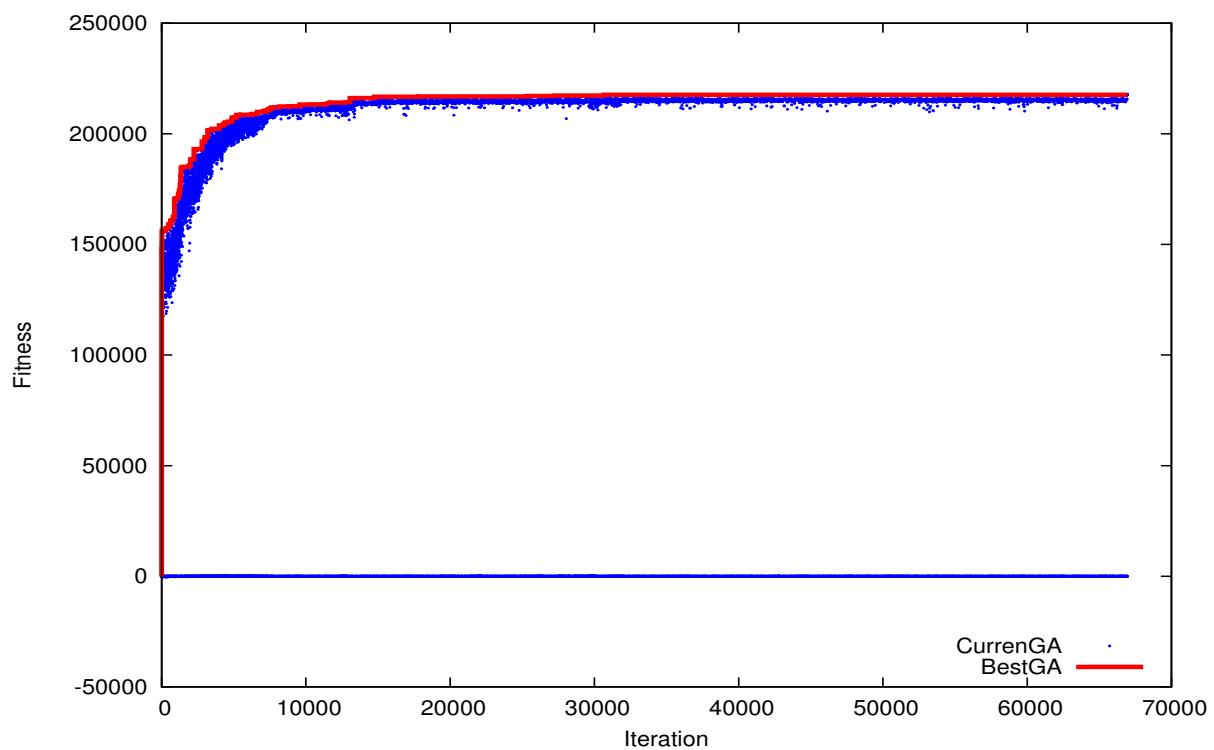


Figura 25: Metaheurísticas generadas por el Genético

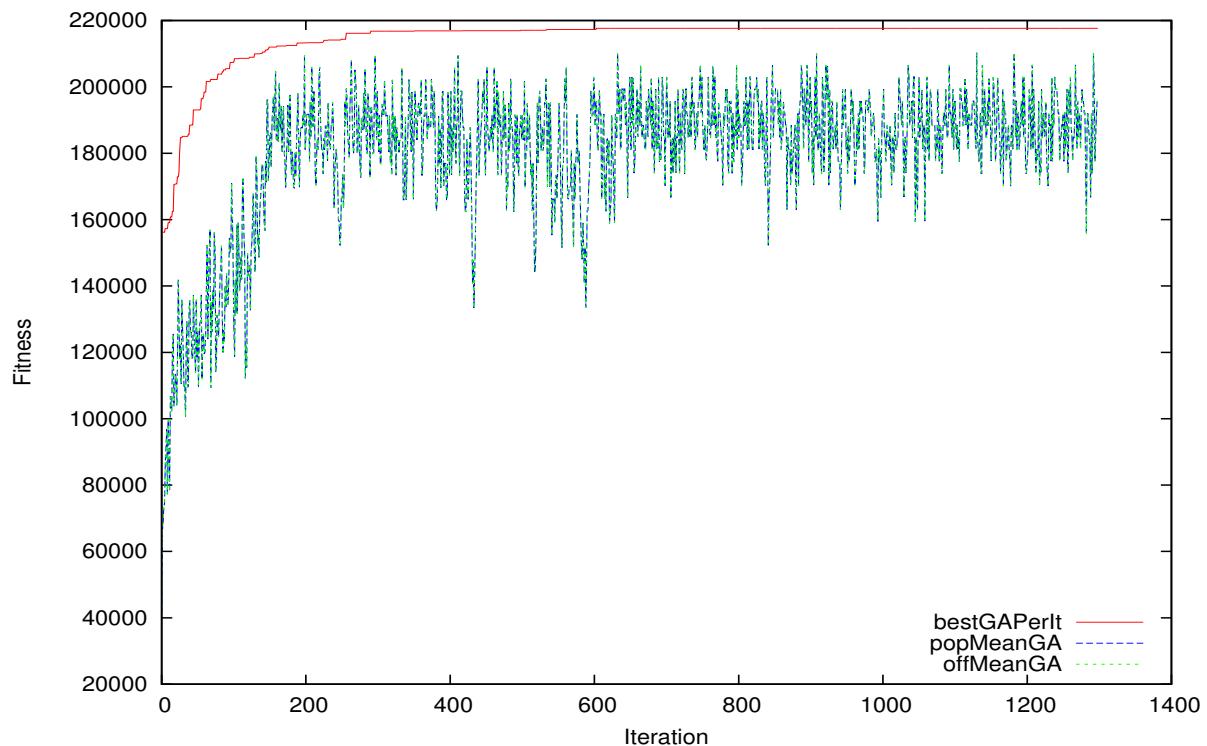


Figura 26: Metaheurísticas de la población

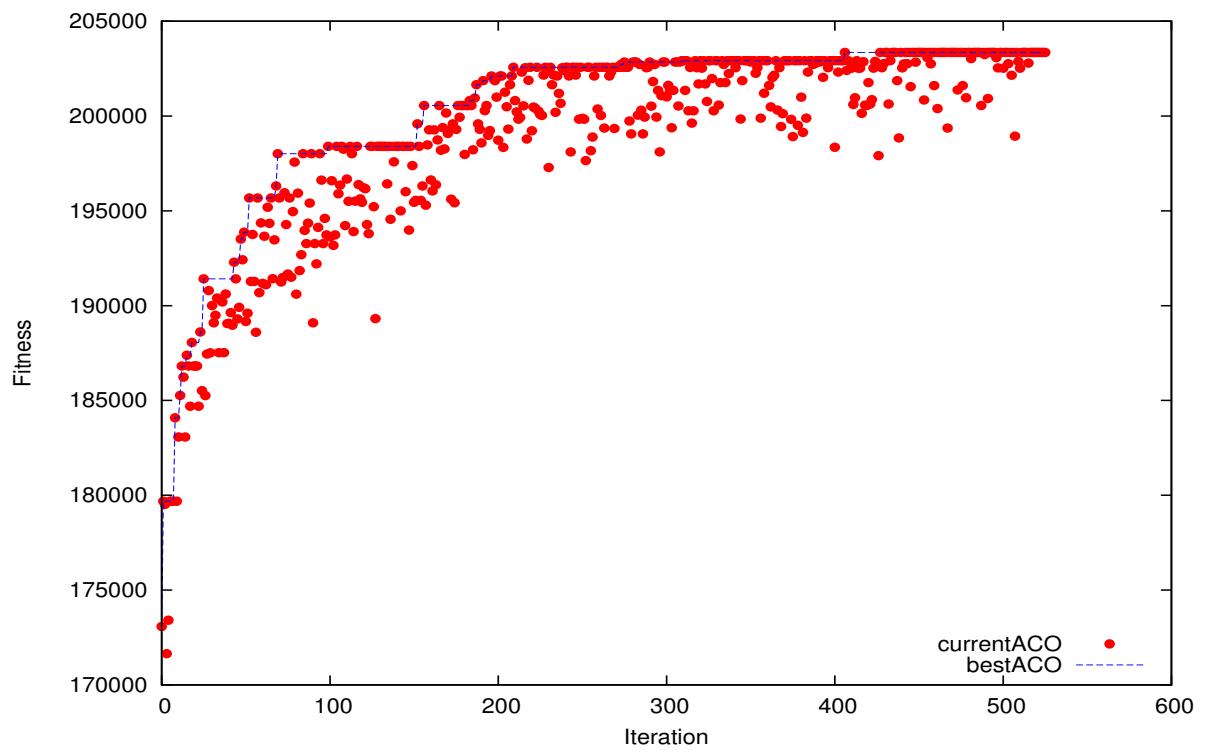


Figura 27: Metaheurísticas generadas por el ACO

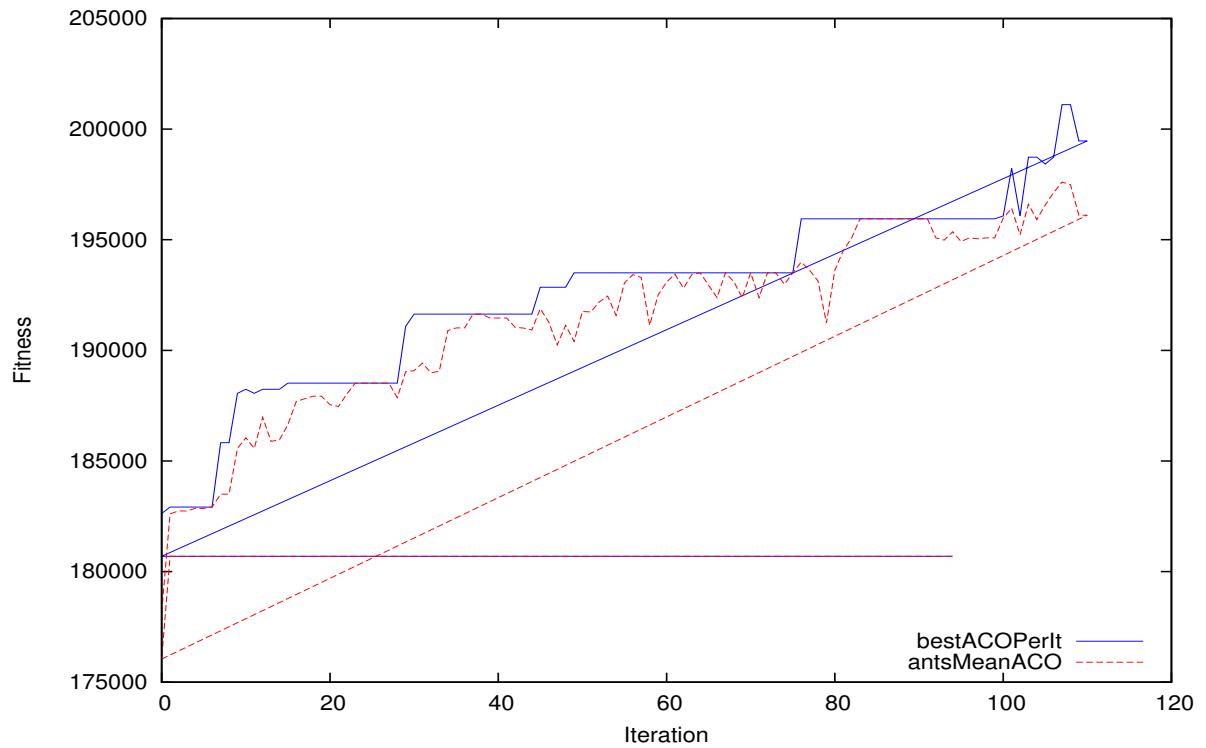


Figura 28: Metaheurísticas generadas por las hormigas

■ Resultados con cinco mochilas:

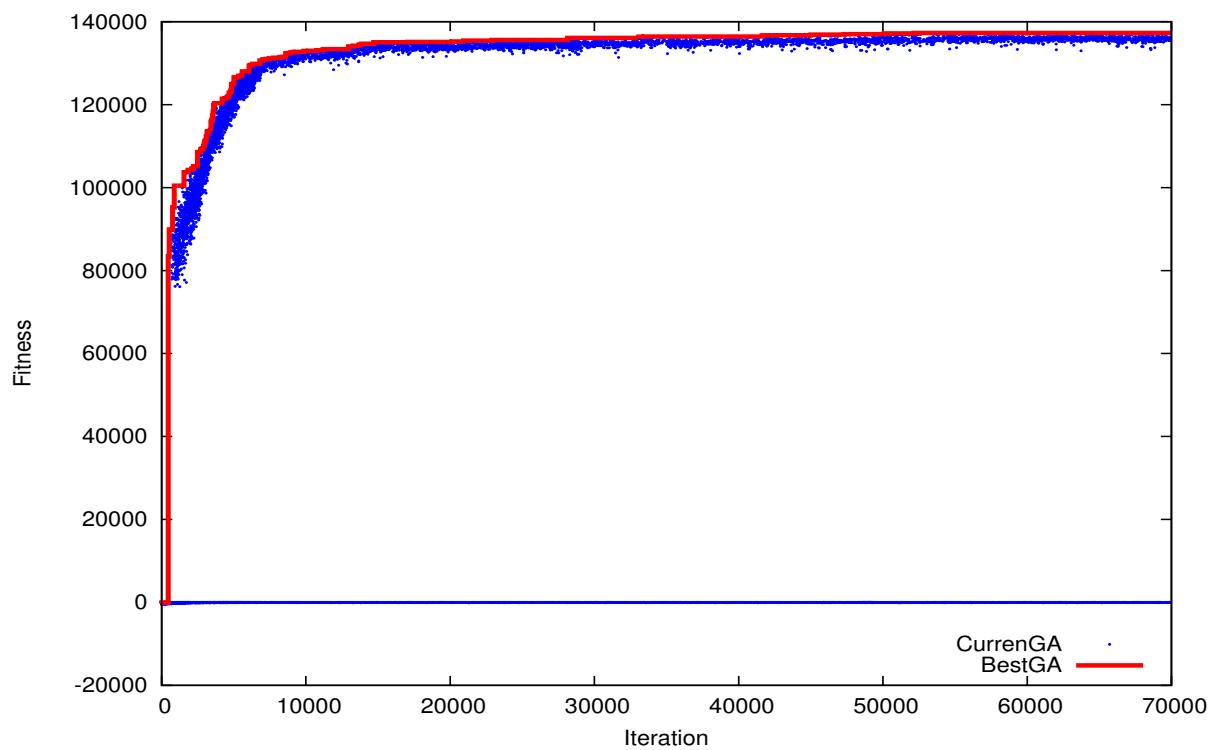


Figura 29: Metaheurísticas generadas por el Genético con cinco mochilas

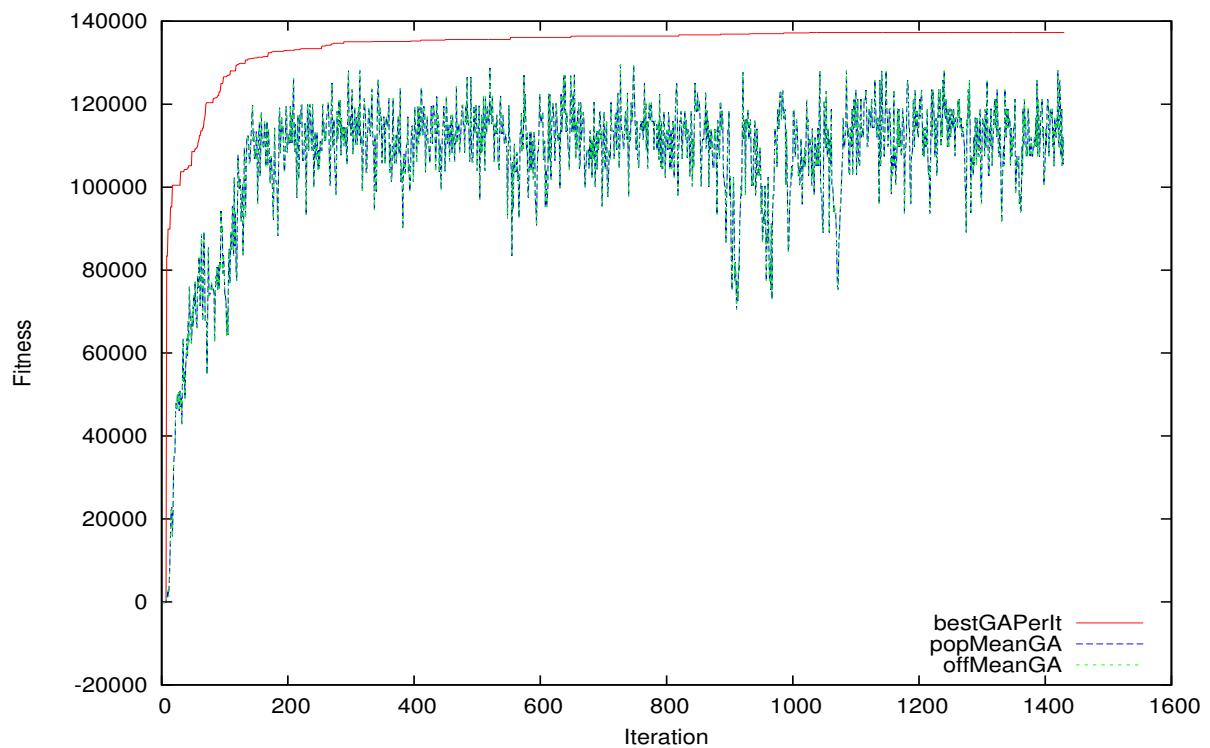


Figura 30: Metaheurísticas de la población con cinco mochilas

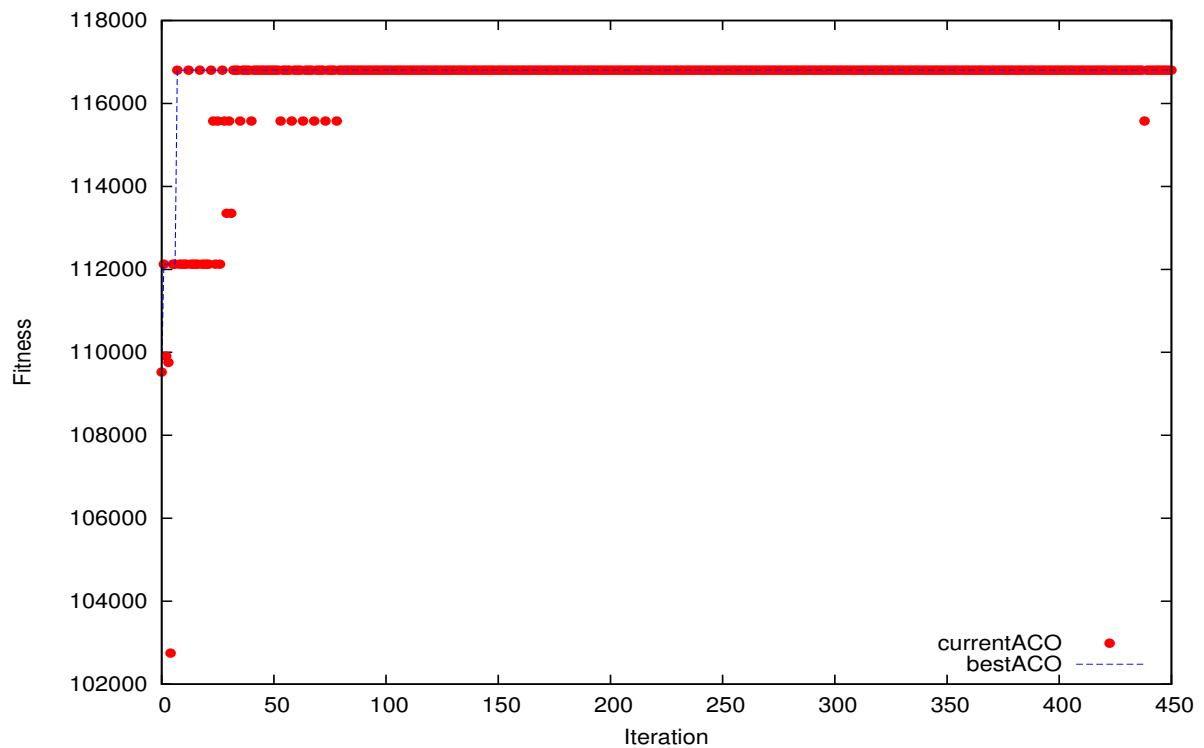


Figura 31: Metaheurísticas generadas por el ACO con cinco mochilas

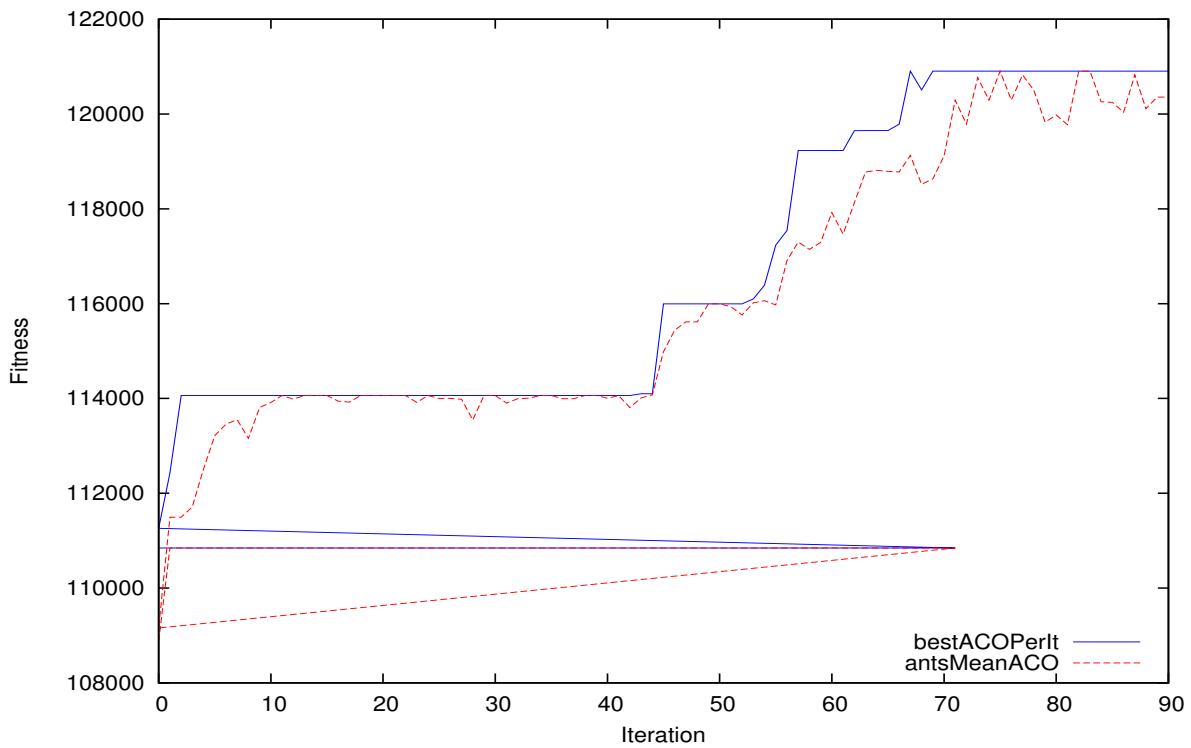


Figura 32: Metaheurísticas generadas por las hormigas con cinco mochilas

24.2. Conclusiones

Una vez han sido mostradas todas las gráficas, cabe destacar que para una mejor conclusión sobre los datos obtenidos en estas, hemos decidido dividirlos por conjunto de datos, y a continuación, dividir estas en el número de mochilas empleado para cada gráfica.

1. Conjunto de datos jeu_100_25_4:

Cómo se puede observar en las Figuras 1 y 2, los datos obtenidos con este algoritmo son más altos que con la metaheurística ACO (ver Figuras 3 y 4). También se puede observar que los resultados obtenidos superan en *fitness* a los resultados obtenidos con el mismo algoritmo pero con dos mochilas más. Por tanto, podemos asegurar, que para este conjunto de datos, los mejores resultados se obtienen con el Algoritmo Genético para tres mochilas.

Por otra parte, analizando las Figuras 3 y 4, se puede observar que mejor resultado es peor respecto al obtenido con el otro algoritmo para las mismas mochilas, pero el crecimiento en las primeras iteraciones es muy superior respecto a este, siendo necesarias muchas menos iteraciones para alcanzar su *fitness* máximo. Si comparamos el resultado obtenido, con su homólogo pero teniendo 5 mochilas en vez de 3, podemos observar que el crecimiento de este último es mucho más rápido, casi forma una vertical hasta alcanzar el máximo, aunque a costa de obtener un *fitness* considerablemente más pobre.

Si nos pasamos analizar los resultados obtenidos para este conjunto de datos teniendo 5 mochilas en el problema, podemos observar algo similar a lo comentado en el apartado de las tres mochilas; se obtiene un mejor *fitness* en las cuatro primeras gráficas pero con cinco mochilas se necesitan menos iteraciones para alcanzar el *fitness* máximo obtenido, a costa de

obtener un peor *fitness* respecto al obtenido con tres mochilas.

Por otro lado, si comparamos el rendimiento de cada algoritmo, podemos ver que la lectura es la misma que la de los resultados con tres mochilas, ya que cuando empleamos la metaheurística del Algoritmo Genético (ver figuras 5 y 6), se obtiene mejor rendimiento que con la metaheurística de la Colonia de Hormigas (ver figuras 7 y 8), pese a que la velocidad de crecimiento en cuanto a número de iteraciones es superior en esta última.

2. Conjunto de datos jeu_100_75_2:

Usando este conjunto de datos para obtener unos resultados, siguiendo el mismo procedimiento que con el anterior conjunto de datos analizado, en líneas generales se puede ver con suma facilidad cómo tanto el crecimiento como el mejor *fitness* de cada gráfica es equivalente a lo explicado en el apartado anterior.

La única diferencia reseñable entre los resultados obtenidos entre un conjunto de datos y otro es el valor del *fitness*; en este conjunto de datos se obtiene un rendimiento considerablemente superior al obtenido con el primer conjunto de datos. De hecho, casi triplica en cada gráfica equivalente el *fitness* obtenido en el primero del conjunto de datos analizado, respectivamente.

3. Conjunto de datos jeu_200_25_8:

La principal diferencia de los resultados obtenidos con este conjunto de datos, es que hay una ligera mejoría en rasgos generales en cuanto al mejor *fitness* obtenido hasta el momento. Es bastante similar al segundo conjunto de datos analizado a excepción de la forma de la gráfica en cuanto al crecimiento se refiere. Aquí el crecimiento es más paulatino que en el segundo conjunto de datos, aunque el primer *fitness* obtenido en las primeras iteraciones sea superior.

4. Conjunto de datos jeu_200_75_5:

En este cuarto y último conjunto de datos, se puede observar como se obtiene un *fitness* de valores desproporcionadamente altos, en comparación con el resto de soluciones obtenidas. Cabe destacar, cómo el algoritmo de la Colonia de Hormigas es capaz de obtener unos niveles de *fitness* altísimos con tan pocas iteraciones. Aquí se acentúa más aún que en el resto de conjuntos de datos, la diferencia del *fitness* en cuanto al número de mochilas, siendo los resultados del rendimiento con tres mochilas muy superiores a los obtenidos con cinco mochilas.

25. Problema seleccionado

En este apartado vamos a nombrar y a explicar las metaheurísticas que hemos acordado implementar en nuestro problema elegido "Multidimensional 2-way Number Partitioning Problem".

- **Búsqueda Aleatoria:** Al igual que hicimos con el problema de la mochila, aplicaremos algoritmos para que de manera aleatoria generemos soluciones aleatorias y guardando así la mejor de las soluciones obtenidas. Esta técnica no garantiza la obtención de una solución óptima, pero nos permite en una primera instancia generar buenas soluciones en periodo corto de tiempo.
- **Búsqueda Local:** Como ocurría con la metaheurística antes usada, en la búsqueda local generaremos una solución aleatoria, para así explorar la totalidad del conjunto. Una vez realizado la búsqueda devuelve una solución que mejore a la anteriormente encontrada.
- **Greedy:** Greedy recibe la instancia que vaya a utilizar, cargando las particiones en un vector. Este vector se ordenará de mayor a menor en función de una columna elegida al azar. Una vez tengamos el vector crearemos unas particiones para evaluarlas. Se elegirá la que mayor fitness tenga.
- **Enfriamiento simulado:** Para la realización de esta metaheurística vamos a basarnos en el algoritmo de Metrópolis, para así obtener los resultados usando el criterio de aceptación de soluciones vecinas. Para calcular la temperatura inicial generaremos una solución aleatoria y calcularemos el vecino. Una vez obtenido lo evaluaremos y calcularemos su valor de fitness.