A mathematical model and a metaheuristic approach for a memory allocation problem

AMMM - Q1 2014

Constantino Gómez
Cristóbal Ortega
David Trilla

Table of Contents

About the Problem....3
Introduction....3
Definition of the problem....4

Mixed Integer Linear Programing Model....5
Data variables used in CPLEX:....5
Constraints & Equations:....6

Meta-heuristics Model 1: Tabu Search VNS....8
Constructive phase (GreedyMemEx)....9
Tabu Local Search (TabuMemEx)....9
Neighborhood N0 and N1....11
VNS-TS MemEx procedure....12
About the implementation of the procedures....13

Meta-heuristics Model 2: BRKGA....14
Structure of the chromosome....14
Algorithm for the Decoder....15
Parameters in the algorithm....17

Evaluation of the implementation....19
CPLEX against our Metaheuristics....19
Metaheuristics performance....21

Conclusions....23

About the Problem

Introduction

The paper presents a really common problem in embedded computing, that heavily affects the efficiency and performance of the microprocessors. The access to the memory in order to retrieve data incurs in heavy penalties and waste of cycles, different optimizations have been applied to reduce this cost, and memory banking is one of the most common ones.

Since you can access simultaneously the internal memory banks, distributing the memory accesses between memory banks allows faster executions, but to do so we require some knowledge about how the data is accessed. The paper presents a solution to map in the best efficient way the data structures to the memory banks, so minimum penalty for the data access is applied and performance is enhanced.

Definition of the problem

The problem is abstracted by defining a set of internal banks, each of these has a limited capacity to store data structures. This allows us to define what data structures fit in the internal memory. Those structures that do not fit, have to go to the external memory, this external memory is defined to have unlimited capacity. This defines where we can put structures and where we can not put them.

The true problem arises , when at a given point in a code, two memory structures are accessed at the same time. If this happens, we state that those two structures have a conflict, this means that if both structures are mapped on the same memory bank, the program will perform worse, since it will have to send two memory request, and sequentialize the accesses.

```
a[i] = a[i+1]
```

This is an example of the same memory structure, having a conflict with itself.

Every structure has a different base access cost, this base cost, is increased by a penalty multiplier if that structure is allocated in the external memory, since it's located outside the chip and it's more expensive to access.

Finally there are different costs that can be applied to the conflicts, the most obvious one, is cost 0, this means that the conflict does not arise, because the two memory structures involved are mapped in different internal memory banks. A conflict can have its default conflict cost, if the two structures are mapped in the same memory bank. When one of the structures is mapped to the external memory, we must apply the penalty cost for accessing data outside the chip, and therefore the default conflict cost is multiplied by that penalty. Finally, if both structures end in the external memory, we apply the penalty twice, and this means multiplying the default cost by 2p.

Mixed Integer Linear Programing Model

Once we understood how the model has been set, it's time to implement it using a mixed integer linear programing model.

Data variables used in CPLEX:

In order to program the model in CPLEX we made an abstraction from the constraints, and we have used these variables:

We have as input data:

- N: Number of structures
- M: Number of memory banks
- O: Number of conflicts between structures
- p: Penalization

- s[N size]: Size of each structure
- c[M size]: Capacity of each bank
- e[N size]: Access cost to each structure
- conf[O size][3 size]: For each conflict, cost of the conflict and structure A and B producing that conflict.

And as decision variables:

- x[N size][M+1 size]: For each structure. (the '+1' represents the external memory)
- y[O size]: For each conflict, values 0 , 1 , p or 2p depending on the cost assumed.

Constraints & Equations:

The MILP is defined by the following equations:

$$\text{Min} \sum_{k=1}^{o} y_k d_k + \sum_{i=1}^{n} \sum_{j=1}^{m} \left(e_i x_{i,j}\right) + p \sum_{i=1}^{n} \left(e_i x_{i,m+1}\right) \tag{2}$$

$$\sum_{j=1}^{m+1} x_{i,j} = 1, \quad \forall i \in \{1, \ldots, n\} \tag{3}$$

$$\sum_{i=1}^{n} x_{i,j} s_i \leq c_j, \quad \forall j \in \{1, \ldots, m\} \tag{4}$$

$$x_{a_k,j} + x_{b_k,j} \leq 1 + y_k, \quad \forall j \in \{1, \ldots, m\}, \forall k \in \{1, \ldots, o\} \tag{5}$$

$$x_{a_k,j} + x_{b_k,m+1} \leq 1 + \frac{1}{p} y_k, \quad \forall j \in \{1, \ldots, m\}, \ \forall k \in \{1, \ldots, o\} \tag{6}$$

$$x_{a_k,m+1} + x_{b_k,j} \leq 1 + \frac{1}{p} y_k, \quad \forall j \in \{1, \ldots, m\}, \ \forall k \in \{1, \ldots, o\} \tag{7}$$

$$x_{a_k,m+1} + x_{b_k,m+1} \leq 1 + \frac{1}{2p} y_k, \quad \forall k \in \{1, \ldots, o\} \tag{8}$$

$$x_{i,j} \in \{0, 1\} \quad \forall (i, j) \in \{1, \ldots, n\} \times \{1, \ldots, m\} \tag{9}$$

$$y_k \geq 0 \quad \forall k \in \{1, \ldots, o\} \tag{10}$$

(2): This is the objective function, it defines the cost of the accesses, it is divided into three adding parts. The first one computes the cost of the conflicts between data structures, the second one, adds the cost of accessing data structures in the internal memory banks, and the last part, computes the cost multiplied by a penalty, for those structures located in the external memory bank.

(3): This constraint disables the possibility of having the same data structure in more than one bank.

(4): This constraint prevents the location of data structures on memory banks where their capacity has reached it's limit.

(5) ,(6), (7), (8): This constraints set the value of variable y, for every conflict to the correct value, (5) sets y to 0 (for no conflict) or to the default conflict cost (two data structures mapped in the same internal bank), (6) and (7) set it to value p, (one of the structures having a conflict is allocated on the external memory. And finally (8) sets the value to 2p,(both data structures are in the external memory).

(9): Setting the decision variable x to a boolean state.

(10): Non-negativity constraint.

Meta-heuristics Model 1: Tabu Search VNS

This metaheuristic is introduced by the authors of the paper to find solutions for big input problems within a reasonable amount of time.

The key features of the metaheuristic are:

Greedy constructive phase, allows to generate an initial feasible solution in time cost O(mn).

Randomization of the initial solution, before performing any search, destroys 60% of the solution and its regenerated with the same greedy algorithm. Allows for a (potential) different base solution each time so the search explores a different region of the solution space.

Tabu local search, during the search keeps track of the previous moves of structures to avoid being stuck in a local optimal. Uses the TabuCol variation of the Tabu search, presented by Herz and de Werra (1987), the tabulist size is dynamically changed during the local search.

Variable Neighborhood Search, alternates the method of searching between repetitions of the local search. Two different Neighborhoods are presented. This is used to achieve higher level of diversification and again scape a local optimal.

Constructive phase (GreedyMemEx)

```
Input: A ← {a_1, ..., a_n}
Output: [X*, f*]
Initialization:
Capacity used: u_j ← 0, ∀j ∈ {1, ..., m+1}
Allocation: x*_ij ← 0, ∀i ∈ {1, ..., n}, ∀j ∈ {1, ..., m+1}
f* ← 0
Assignment:
for i ← 1 to n do
    h* ← ∞        // (auxiliary variable for the partial
       greedy solution)
    for j = 1 to m + 1 do
        if u_j + s_ai < c_j then
            Compute g_ij, the cost for allocating data a_i to memory bank j
            if g_ij < h* then
                b ← j
                h* ← g_ij
            end
        end
    end
    x*_ai,b ← 1
    u_b ← u_b + s_ai
    f* ← f* + h*          //(total cost of the solution)
end
```

It doesn't introduce any special feature, it's a classical greedy algorithm, that **given an order for the data structures, assigns each one to the memory bank that implies less cost.**

The solution obtained could be far from optimal but obtained in a very small amount of time.

Tabu Local Search (TabuMemEx)

This procedure in conjunction with the two neighborhoods describe how the local search is done

```
Input: Initial solution X and number of neighborhood k
Output: [X*, f*]
Initialization:
Capacity used u_j ← 0   ∀j ∈ {1, ..., m}
NT ← NTmax
f* ← ∞
Iterative phase:
Iter ← 0
while Iter < Niter and f(X) > 0 do
    [X', (i, h, j)] ← Explore-Neighborhood-N_k(X)
    X ← X'
    if f(X') < f* then
        f* ← f(X')
        X* ← X'
    end
    Update the tabu list with pairs (i, j) and (i, h)
    Update the size of tabu list NT
    Iter ← Iter + 1
end
```

everytime we call the TabuMemEx function.

Things to consider here:

In each iteration we will obtain the best feasible solution X' from the neighborhood k of the input solution X.

That solution will be used in the next iteration whether improves or not our current solution.

Every time we generate a new solution, we add the last move we did to the tabu list, the pairs `<i,h>` and `<i,j>`.

Every fixed number number of iterations (NTMax) of the local search, we will update the size of the tabu list (FIFO). To compute the new size we will use the formula NT = a + NTMax*rand[0,2].

NTMax and a are arbitrary integers.

As a side note, In the paper handed at raco, we found a problem with the update of the tabu lists with pairs. It made no sense that the list was updated outside the while loop, making it useless, that lead us to look for other implementations of this method and finally we found a book edited by the same people that had a correct version of the algorithm (figure below). The wrong and original pseucode is the following:

while $Iter < Niter$ **and** $f(X) > 0$ **do**
  $[X', (i, h, j)] \leftarrow$ `Explore-Neighborhood-`$\mathcal{N}_k(X)$
  $X \leftarrow X'$
  if $f(X') < f^*$ **then**
    $f^* \leftarrow f(X')$
    $X^* \leftarrow X'$
  end
  Update the size of tabu list NT
  $Iter \leftarrow Iter + 1$
end
Update the tabu list with pairs (i, j) and (i, h)

Neighborhood N0 and N1

In the first method N0, the neighborhood of a solution are all the feasible solutions that can be reached by changing one data structure to another bank. As always we pick the best one.

Input: X
Output: $[X', (i, h, j)]$
Find non tabu min cost move (i, h, j), such that $h \neq j$ and $u_j + s_i \leq c_j$
Build the new solution X' as follows:
$X' \leftarrow X$
$x'_{i,h} \leftarrow 0$
$x'_{i,j} \leftarrow 1$
$u_j \leftarrow u_j + s_i$
$u_h \leftarrow u_h - s_i$

Note how we don't allow to move any data structure to a memory bank if that move is registered in the tabu list.

Second method is more complicated, now, we allow unfeasible moves (but still tabu moves aren't allowed). If we obtained a unfeasible solution we will then fix it by moving other data structures to other banks trying to make space. This works in the sense that maybe you're stuck in a local optimal and you want to scape hoping you're gonna find the global optimal. This Neighborhood is more useful when the capacity of the banks is very limited compared to the size of structures, in those cases a lot of unfeasible moves will prevent you from moving around the solution space so especial measures have to be taken, hence Neighborhood 1.

Input: X
Output: $[X', (i, h, j)]$
`First phase: considering a potentially unfeasible move`
Find non tabu min cost move (i, h, j), such that $h \neq j$
Build the new solution X' as follows:
$X' \leftarrow X$
$x'_{i,h} \leftarrow 0$
$x'_{i,j} \leftarrow 1$
$u_j \leftarrow u_j + s_i$
$u_h \leftarrow u_h - s_i$
`Second phase: repairing the solution`
while $u_j > c_j$ **do**
  Find non tabu min cost move (l, j, b), such that $l \neq i$, $j \neq b$ and $u_b + t_l \leq c_b$
  Update solution X' as follows:
    $x'_{l,j} \leftarrow 0$
    $x'_{l,b} \leftarrow 1$
    $u_b \leftarrow u_b + s_l$
    $u_j \leftarrow u_j - s_l$
end

VNS-TS MemEx procedure

The top level of the pseudocode shows how the whole metaheuristic is implemented. The initialMemex part corresponds to the GreedyMemex procedure to generate a initial first solution. The paper proposes to also implement a completely random initial solution but we decided to pick the greedy one.

```
output: X* and f*
Initialization:
Generate A
(X*, f*) ←InitialMemex(A)
k ← 0
Iterative phase:
i ← 0
while i < Nrepet do
    // (Make a new initial solution X from X*)
    X ← 60%X* complete the solution with GreedyMemex
    Apply (X', f') ←TabuMemex(X, k) using Explore-Neighborhood-N_k
    if f' < f* then
        X* ← X'
        f* ← f'
        i ← 0
        k ← 0
    else
        if k = kmax then
            k ← 0
        else
            k ← k + 1
        end
        i ← i + 1.
    end
end
```

As for the randomization part where we destroy the solution will help us to find the global optimal.

Everytime we perform a local search and we obtain a better solution than the one so far, we reset the Nrepet counter. This works both ways, allowing to keep going while we are improving the results and stop if we've been not obtaining anything better for a long time.

After each local search we swap the k variable varying between neighboorhoods each iteration.

About the implementation of the procedures

Its important to consider how the implementation of a metaheuristic is done, we know that, for a big enough combinatorial problem ILP will perform way worse but, being careful about the metaheuristics implementation can lead easily to a 10-30x speed improvement over a base metaheuristics implementation. That improvement can be used to allow more deep exploration of the solution space using the same amount of time or obtain the same solution but faster.

To produce an optimal implementation of the procedures shown, you have to consider using efficient data structures and design carefully the part that corresponds to the computational efforts. One thing missing in this paper is some hints about how to better implement the heavy computational recurrent parts. Also there's a vague and sometimes confusing way to describe the parts of the algorithm that are not properly specified in the pseudocode.

Anyhow from the experience, we found that for the incremental cost of solution function its important to keep copy of decision matrixes at minimum so everytime we want to check the cost of a solution, we modify the X decision matrix, compute the cost, and then undo the changes. Everytime a solution has to be returned it's done by reference. Also its recommended to update the <Y vector> of status of the conflicts the same way we do with X.

To keep the computing of the new cost under lineal time a structure that allows direct access avg. O(1) to the conflicts of each data structure is required, this can be done with an unordered map in C++. Only part of all the improvements mentioned here were included since a fully optimized implementation was not required for the assignment and implementing (and debugging) all the features of the VNS-TS algorithm was time consuming enough.

Meta-heuristics Model 2: BRKGA

For the second meta-heuristics we chose Biased random-key genetic algorithm for the next reasons:

- We think it is a very interesting algorithm because we only have the decoder as dependent-problem part. We can use an existing interface for the rest of the algorithm and just implement our decoder to start to have results.
- Just coding the decoder has another advantage: we can test different approaches to the algorithm for the decoder easily.

Structure of the chromosome

As we have the inputs:

- N: number of structures to be placed in banks or memory
- M: number of banks
- O: number of conflicts (and who is producing it)

and we have a vector of N random keys that are produced by the algorithm BRKGA:

←---------- N ---------->

0.21	0.54	0.12	..	..	..	..	..	..	0.973

Then, we need to treat this input of the algorithm to create a good solution for our problem:
We need to sort this random keys vector in some way:

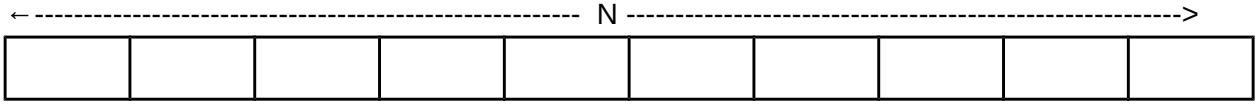
- Sort by size of each structure to be placed
- Sort by number of conflicts

we choose the first one, sort by size, because if we place first the structures that have a bigger size we can fill up a single memory bank, and if we have less structures in a single memory bank it is more probably that we do not have a conflict with structures in the same memory bank. Also, doing this makes the decoder easier, so we can compute an input vector of random keys faster.

Algorithm for the Decoder

Our data is represented in this way:

Vector of size of our structures to place:



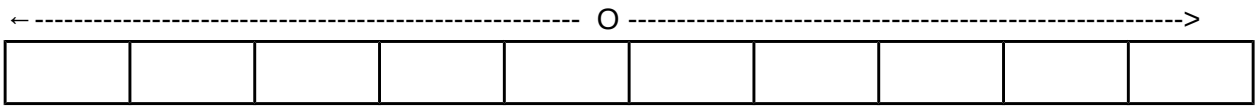
Position 0 represents size of structure 1
Position 1 represents size of structure 2
…
Position N-1 represents size of structure N

Vector of capacity of each bank:

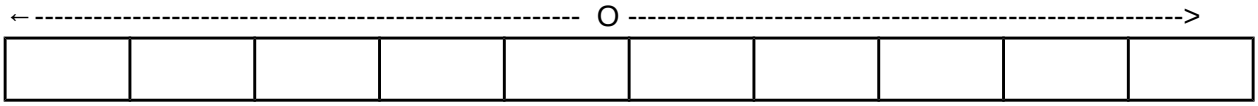
Position 0 represents capacity of bank 1
Position 1 represents capacity of bank 2
…
Position M-1 represents capacity of bank M

Vectors of conflicts:

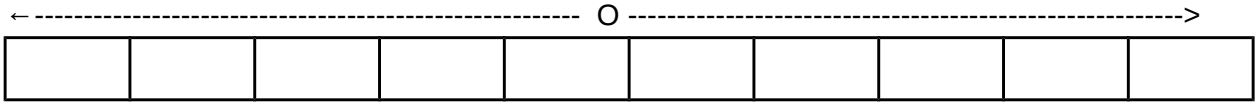
Vector of penalties:



Vector of first structure with conflict:



Vector of second structure with conflict:



Position 0 represents conflict 1, with its penalty and its 2 structures in conflict
Position 1 represents conflict 2, with its penalty and its 2 structures in conflict
…
Position O-1 represents conflict O, with its penalty and its 2 structures in conflict

1. Treat our input:
   In BRKGA's decoder we receive a vector of random keys that we have to treat to give a solution, for this reason, we are going to create a structure with all the information that has to be sorted:
   - Random keys
   - Size of our structures

   So we do a direct relation between the position in the vector of the random keys and the position in the vector of size of structures.
   Then, we apply the sort, and we recompute the vector of conflicts because the original position probably has changed. As we explained before, we assume that structure 1 is in the position 1 in the vector of structures, so in the vector of conflicts we refer to that structure by position in the vector.

2. Compute a solution
   When we have the vectors sorted and recalculated we do a simple greedy algorithm seen in the section **Constructive phase (GreedyMemEx).** This constructive phase will give us a first solution for our vectors giving priority to the biggest structure that we have.

3. Return objective function value

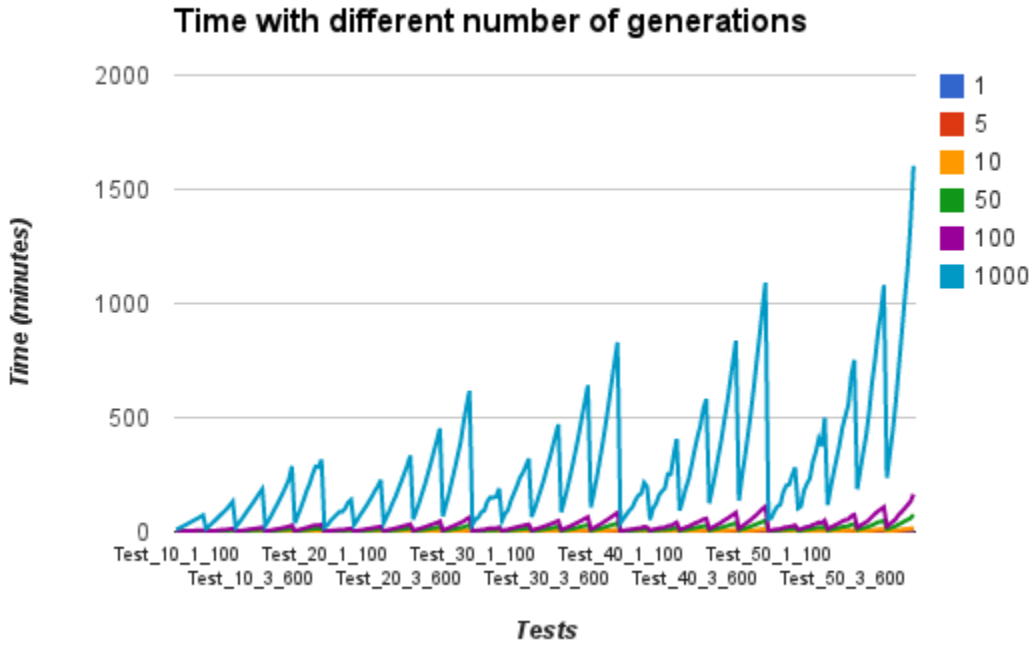
Parameters in the algorithm

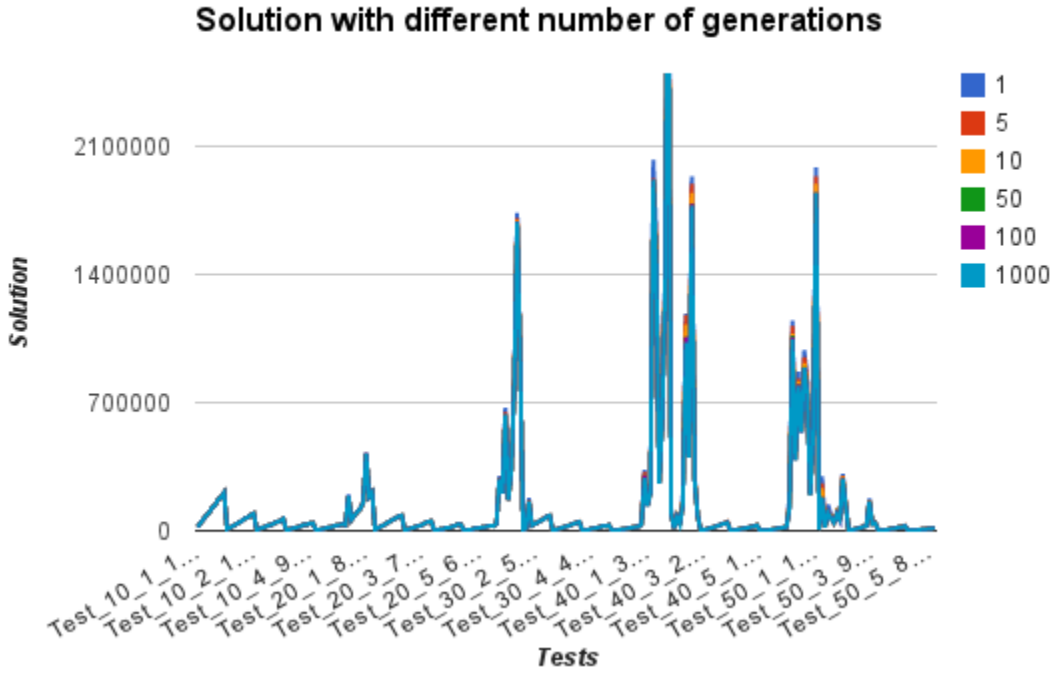
The parameters we use are:

Size of population: 100
Fraction of population to be the elite-set: 0.20
Fraction of population to be replaced by mutants: 0.10
Probability that offspring inherit an allele from elite parent: 0.70
Number of independent populations: 3

We did some tests to check when we have a reasonable execution time and a reasonable good solution to determine the size of population.

Also, we checked the number of generations that we should use for our final tests.
As we see in the next plots in the solution that they found there is almost no difference (only when the time grows) with more generations better the solution but slower we find it. In the final tests we will choose a number of generations that takes the same time that our another metaheuristic so we can compare them properly.

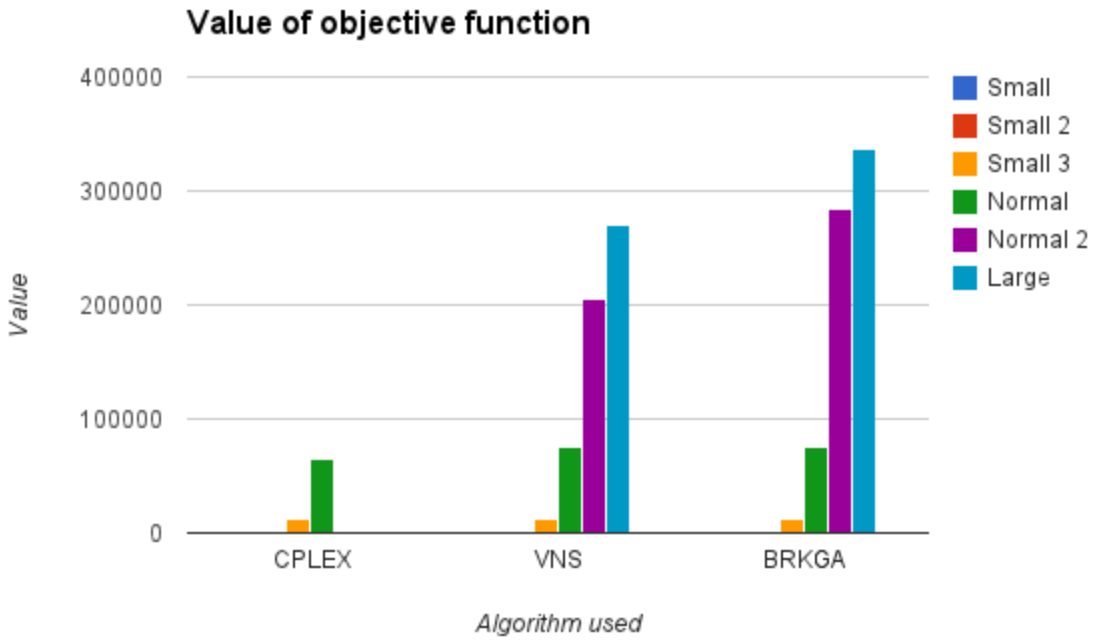

Time with different number of generations
2000
1500
1000
500
0
Time (minutes)
1
5
10
50
100
1000
Test_10_1_100
Test_20_1_100
Test_30_1_100
Test_40_1_100
Test_50_1_100
Test_10_3_600
Test_20_3_600
Test_30_3_600
Test_40_3_600
Test_50_3_600
Tests


Solution with different number of generations
2100000
1400000
700000
0
Solution
1
5
10
50
100
1000
Test_10_1_1...
Test_10_2_1...
Test_10_4_9...
Test_20_1_8...
Test_20_3_7...
Test_20_5_6...
Test_30_2_5...
Test_30_4_4...
Test_40_1_3...
Test_40_3_2...
Test_40_5_1...
Test_50_1_1...
Test_50_3_9...
Test_50_5_8...
Tests

Evaluation of the implementation

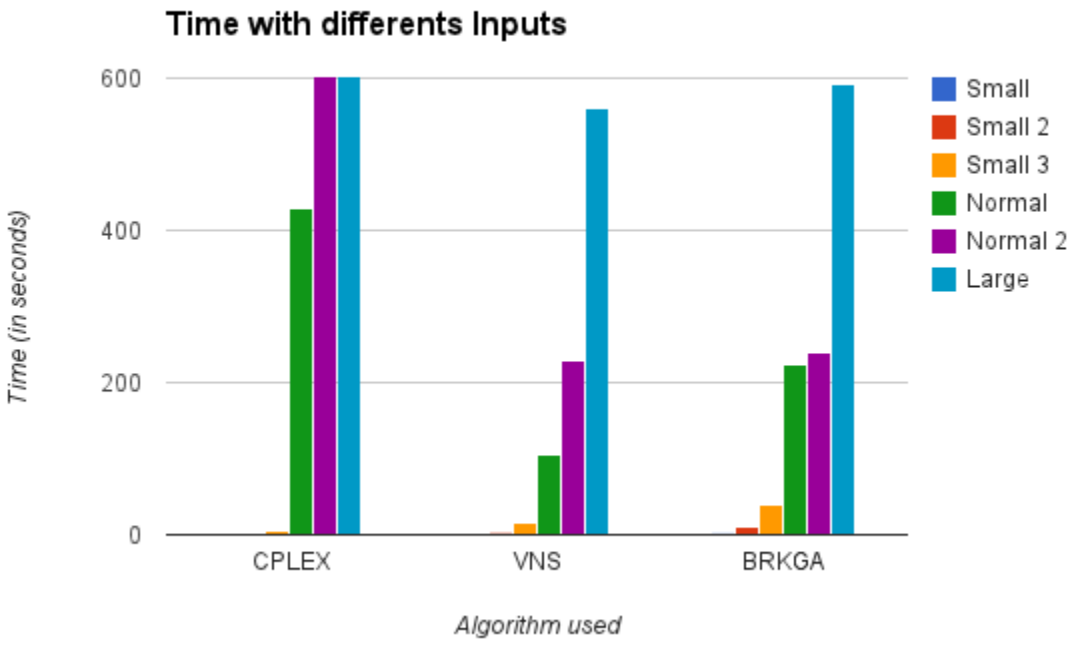
CPLEX against our Metaheuristics

We first evaluate the value of the objective function to see who gives the better solution to our problem:



In this plot we see that CPLEX always offers the better objective function (less is better), but for the tests Normal 2 and Large we had to stop it because it was running for a long time. We expect that the solution that CPLEX would offered us for those tests would have been better than the Metaheuristics used as in the other 4 tests.

We then check the time they spend computing the solution obtained:

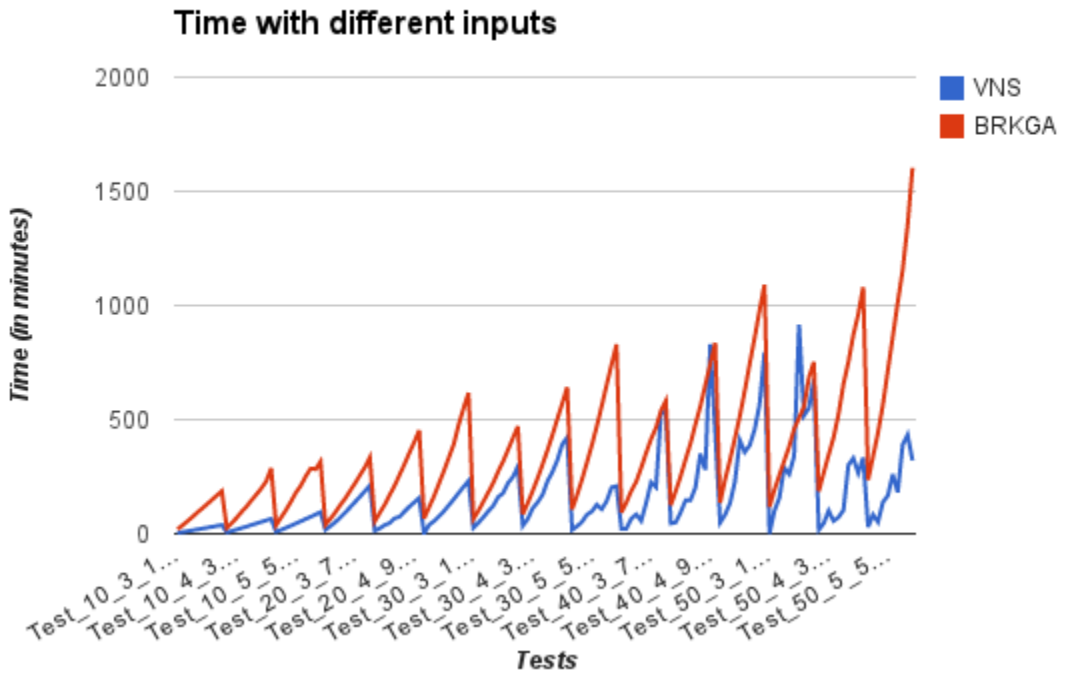


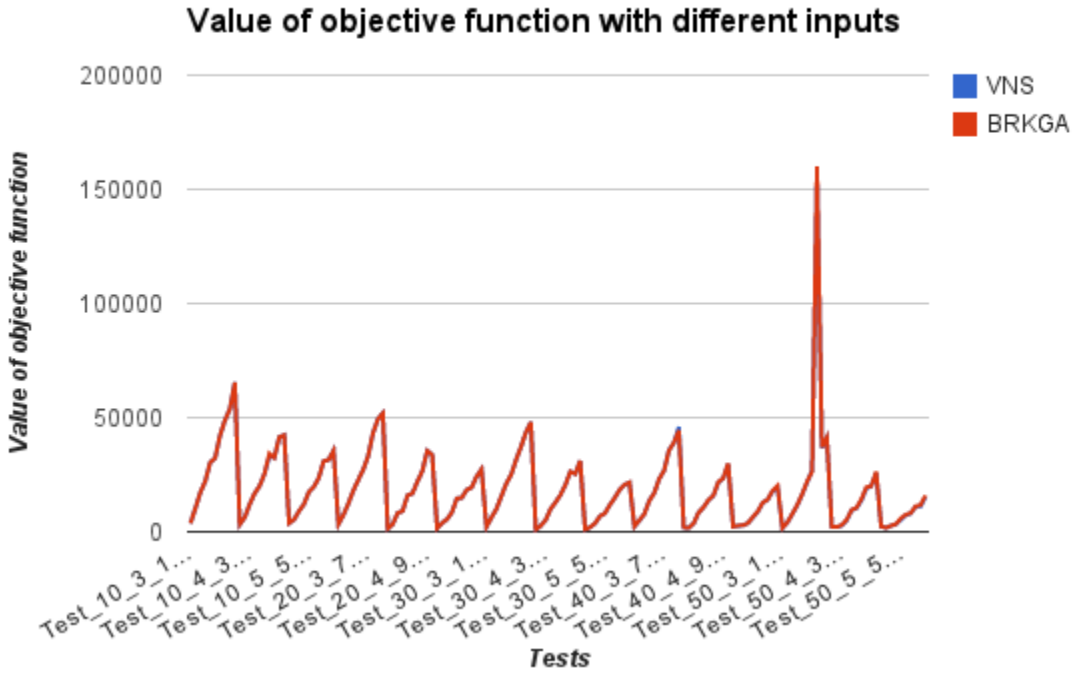
We can see that CPLEX grows exponentially, the 2 largests tests did not finish so we represent it as the top of our max value in the plot.

We can also observe that our first Metaheuristic takes usually less time than BRKGA but not always and the solutions in BRKGA are a bit worse than in the VNS

Metaheuristics performance

The next plots have been generated with 250 different inputs, increasing first the number of structures, then number of memory banks and finally the number of conflicts.
We can see that the real problem is when we increase the number of conflicts.





In this plot we see that we got the same solution in both cases (or almost all, there are visible some blue points over and below the BRKGA line)

Conclusions

We have seen that metaheuristics are a huge save of time when computing a problem, and, specially in this case and the metaheuristics we used (VNS and BRKGA) we should use VNS instead of BRKGA because we have almost the same solutions and less time.

Of course, if we want the best solution we need to run our problem in CPLEX, but we also saw that time in CPLEX grows exponentially and maybe it is a cost that we cannot afford. For this reason, depending in our needs and constraints (our time and our computing resources) we should use the more appropriate algorithm.