UNIVERSITAT POLITÈCNICA
DE CATALUNYA

A variable neighbourhood search algorithm for the constrained task allocation problem

Amaia Lusa, Chris N. Potts

EOLI: Enginyeria d'Organització i Logística Industrial

IOC-DT-P-2006-5
Gener 2006

Institut d'Organització i Control
de Sistemes Industrials

A Variable Neighbourhood Search Algorithm for the Constrained Task Allocation Problem

Amaia Lusa[*1]*; Chris N. Potts*[2]

[1]*Research Institute (IOC) / Engineering School (ETSEIB)*

Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

amaia.lusa@upc.edu

[2]*School of Mathematics, University of Southampton,, Southampton, UK*

C.N.Potts@maths.soton.ac.uk

ABSTRACT

A Variable Neighbourhood Search algorithm is proposed for solving a task allocation problem whose main characteristics are: (*i*) each task requires a certain amount of resources and each processor has a finite capacity to be shared between the tasks it is assigned; (*ii*) the cost of solution includes fixed costs when using processors, assigning costs and communication costs between tasks assigned to different processors. A computational experiment shows that the algorithm is satisfactory in terms of time and solution quality.

Keywords: task allocation problem, variable neighbourhood search, local search, heuristics.

Introduction

The task allocation problem (TAP) consists in assigning a set of tasks to a set of processors (or machines) so that the overall cost is minimised. This cost may include a fixed cost for using a processor, a task assignment cost (which may depend on the task and processor), and a communication cost between tasks that are assigned to different processors. The problem can be constrained (CTAP) or unconstrained (UTAP), depending on whether or not the processors have a limited capacity to be shared between the tasks they are assigned.

The problem arises in distributed computing systems[1], where a number of tasks (programs, editing files, managing data, etc.) are to be assigned to a set of processors (computers, disks, etc.) to guarantee that all tasks are executed within a certain cycle time. The aim is to minimise the cost of the processors and the interprocessor data communication bandwidth installed. The problem also has many industrial applications. For example, Rao[2] introduces a specific constrained task allocation problem belonging to the automobile manufacturing industry: in the modern automobile, many tasks such as integrated

[*] Corresponding author: Amaia Lusa, Research Institute IOC, Av. Diagonal 647 (edif. ETSEIB), p.11, 08028 Barcelona, Spain; Tlf. + 34 93 401 17 05; Fax. + 34 93 401 66 05 ; e-mail: amaia.lusa@upc.edu

chassis and active suspension monitoring, fuel injection monitoring, etc., are performed by a subsystem consisting of micro-computers linked by high-speed and/or low-speed communication lines. The cost of the subsystem is the sum of costs of the micro-computers (or processors) and the installation costs of the data links that provide interprocessor communication bandwidth. Each task deals with the processing of data coming from sensors, actuators, signal processors, digital filters, etc., and has a throughput requirement in KOP (thousand operations per second). Several types of processors are available, and for each one the purchase cost and throughput capacity in terms of the KOP it can handle are known. The tasks are interdependent: a task may need data from other tasks to be completed. Hence, if two tasks are assigned to different processors, they may need a communication link with a certain capacity. The communication load between two tasks is independent of the processors to which they are assigned.

Since its introduction by Stone[1], many authors have tackled different versions of the problem by applying exact algorithms, heuristic procedures and meta-heuristics. However, only a few studies have dealt with the constrained version[3,4,5,6] and, due to the complexity of the problem, none of them are capable of solving some real-world applications optimally. To date, the best of the known approaches for the CTAP is the hybrid method developed by Chen and Lin[3], whose algorithm combines a tabu search and a noise method.

Variable neighbourhood search (VNS) is a relatively recent meta-heuristic for obtaining near-optimal solutions to combinatorial optimisation problems, its main feature being the systematic change of neighbourhood within a local search procedure[7]. Different versions of VNS have been successfully applied to a variety of problems such as *bin-packing*, the *p-median problem*, the *quadratic assignment problem*, the *travelling salesman problem* and the *vehicle routing problem*[8].

In this paper we propose an algorithm based on a VNS scheme for solving the CTAP. The results of a computational experiment show that our procedure outperforms the hybrid method developed by Chen and Lin[3]. The paper is organised as follows: first section introduces the problem, second section describes the VNS approach. Forth section describes our computational experiments and reports the main results. Finally, we present our conclusions in last section.

The constrained task allocation problem

The problem consists in assigning tasks to processors, whilst respecting their capacity. The objective is to minimise the total allocation cost, which may include assigning, fixed and communication costs.

We make use of the following notation.

Data:

n	number of tasks
m	number of processors

a_i requirement of task i (i=1..n)

b_k capacity of processor k (k=1..m)

s_k fixed cost of using processor k (k=1..m)

c_{ij} communication cost if tasks i and j are assigned to different processors (i=1,...,n; j=1,…,n). It is assumed to be independent of the processor.

d_{ik} cost of assigning task i to processor k (i=1,...,n; k=1..m)

Variables:

$x_{ik} \in \{0,1\}$ indicates whether task i is assigned to processor k (i=1..n; k=1..m)

$y_k \in \{0,1\}$ indicates whether any task is assigned to processor k (k=1..m)

The problem can be formulated as follows as a quadratic integer program:

$$[MIN] z = C \text{ (communication cost)} + A \text{ (assigning cost)} + F \text{ (fixed cost)} =$$

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n} c_{ij} \cdot \left(1 - \sum_{k=1}^{m} x_{ik} \cdot x_{jk}\right) + \sum_{i=1}^{n}\sum_{k=1}^{m} d_{ik} \cdot x_{ik} + \sum_{k=1}^{m} s_k \cdot y_k \qquad (1)$$

$$\sum_{k=1}^{m} x_{ik} = 1 \qquad i = 1,..,n \qquad (2)$$

$$x_{ik} \le y_k \qquad i = 1,..,n;\ k = 1,..,m \qquad (3)$$

$$\sum_{i=1}^{n} a_i \cdot x_{ik} \le b_k \cdot y_k \qquad k = 1,..,m \qquad (4)$$

(1) is the allocation cost to minimise (communication, assigning and fixed costs); (2) imposes that each task is to be assigned to one and only one processor; (3) imposes that the binary variable y_k takes value 1 if any task is assigned to processor k; and (4) imposes the capacity constraint.

When the number of processors is equal to 2, the problem can be transformed into a minimum cost cut problem[1] and optimally solved using network flow techniques. However, the problem has been shown to be NP-hard when the number of processors is equal to or greater than three[2].

Since Stone[1], great progress has been made in both computational power and computational technology. Ernst et al.[4] explore the potential of mathematical programming approaches and try different formulations for UTAP and TAP. Nevertheless, the results for the constrained problem cannot be considered to be fully satisfactory. Hence, some kind of heuristic or meta-heuristic procedure seems appropriate for dealing with the problem and finding near-optimal solutions.

Some authors propose local search procedures for solving different versions of the constrained problem. Hadj-Alouane et al.[5] develop a hybrid of the Lagrangian relaxation and genetic algorithm that is shown to be not very efficient when compared to other procedures[3]. Hamam and Hindi[6] propose

a simulated annealing algorithm. Their computational experiment is very limited and there are no results allowing one to see how good their algorithm is in terms of quality solution. Finally, Chen and Lin[3] propose a hybrid method, which combines a tabu search and a noise method algorithm. Essentially, there are three major steps in their approach: first, a relaxed initial solution is created, which consists in assigning all tasks to the cheapest processor (lower fixed cost); second, a local search is undertaken, which combines a tabu search and a noising method. Finally, a processor substitution technique is applied to improve the solutions. Each of the local search methods (tabu search and noising method) is run in two phases: the first uses as a neighbourhood those solutions in which a task is reallocated in another processor; the second uses solutions in which two tasks that are allocated in different processors are exchanged. The results of a computational experiment with a set of randomly generated instances lead them to conclude that their algorithm is better than the random method, the tabu search, the noise method and the genetic algorithm of Hadj-Alouane et al.[5], in terms of both quality and solving time. All the aforementioned algorithms allow non-feasible solutions. Constraint violations are handled by adding appropriate penalties and the authors obtain feasible solutions, but using their procedures offers no guarantee of this.

Our major concern about previous local search procedures is the neighbourhoods that they consider. These algorithms consider as a solution the processor in which each task is allocated, and try the following moves: (1) reallocating a task to another processor and (2) exchanging two tasks assigned to different processors. Although, theoretically speaking, it is possible to achieve any solution by combining these moves, some of them, when considered individually, are too bad to be performed and hence some solutions may remain unexplored. For example, to assign only one task to an empty processor is a very bad move (recall that there are fixed costs), but a good move could consist in allocating a group of high-communicated tasks to an empty processor. Thus, other kinds of moves should be considered (reallocating a group of tasks, for example).

We add the three following types of neighbourhoods to the ones traditionally used (reallocating a task and exchanging two tasks) when solving TAP, which allows us to explore solution spaces of interest: (1) reallocating a cluster of tasks from one processor to another; (2) reallocating a cluster of tasks from different processors to another processor; and (3) emptying a processor by reallocating its assigned tasks to other processors. The results obtained, including these neighbourhood structures, in a VNS algorithm are very satisfactory.

The variable neighbourhood search algorithm

One of the most successful versions of the VNS is the General Variable Neighbourhood Search, GVNS[8], which is detailed in Figure 1. The final condition can be either a maximum CPU time or a maximum number of iterations between two consecutive improvements. One of the steps of GVNS is a descendent local search using different neighbourhoods, VND (see Figure 2). VND finishes when no improvement is obtained, which yields a solution that is a local optimum in all the neighbourhoods that are used.

We make use of the following notation: x is the initial solution; $f(x)$ is the cost of solution x; $umax$ is the number of neighbourhood structures applied; and $N_u(x)$ is the neighbourhood of type u of solution x (u=1,…,$umax$). For the sake of efficiency, f(x) is updated in each step (not evaluated).

General Variable Neighbourghood Search (GVNS)

Generate an initial solution, x, and evaluate (f(x))
While (no final condition) *do*
    u = 1
    While (u ≤ *umax*) *do*
        Choose, at random, a solution of $N_u(x)$, x'
        x'' is the result of applying **VND** to x'
        If f(x'')<f(x) *then*
            $x := x''$ and u = 1
        else
            u:=u+1
        end if
    end while
end while
Return best found solution

Figure 1. General Variable Neighbourhood Search Algorithm, GVNS

Descendent Variable Neighbourghood Search (VND)

x is the initial solution for VND
While (no final condition) *do*
    u = 1
    While (u ≤ *umax*) *do*
        x' is the best solution in $N_u(x)$
        If f(x')<f(x) *then*
            $x := x'$ and u = 1
        else
            u:=u+1
        end if
    end while
end while
Return best found solution

Figure 2. Descendent Variable Neighbourhood Search Algorithm, VND

Neighbourhoods

Five neighbourhood structures were used to allow the algorithm to explore any kind of solution. None of the following moves allow non-feasible solutions. Hence, it is guaranteed that the algorithm will always yield a feasible solution (in ^{5} and ^{3} non-feasible solutions are allowed to be explored).

$N_1(x)$ reallocate a task *i* from processor *k* to processor *l*.

$N_2(x)$ exchange two tasks (task *i* from processor *k* to processor *l* and task *j* from processor *l* to processor *k*).

$N_3(x)$ reallocate a cluster of tasks from processor *k* to processor *l*.

$N_4(x)$ reallocate a cluster of tasks from different processors to processor *l*.

$N_5(x)$ empty processor *k*.

Communication and assigning costs are considered when determining the cluster of tasks to be reallocated. The three new types of neighbourhood proposed in our GVNS are described below.

We make use of the following notation:

x_u is a solution belonging to $N_u(x)$ (u=1,…,5)

P_k set of tasks currently assigned to processor *k* (k=1,…,*m*)

b_k' remaining capacity of processor *k* (k=1,…,*m*)

T_{kl} cluster (set of tasks) currently assigned to processor *k* that can be assigned to processor *l* (k=1,…*m*; l=1,…,*m* | *l* ? *k*)

T_l cluster (set of tasks) that can be assigned to processor *l* (l=1,…,*m*)

In Figure 3 an algorithm for finding a neighbour in $N_3(x)$, x_3, is detailed. The costs added in C_j are "attracting" task *j* to processor *l*, while the costs substracted in C_j are "attracting" task *j* to the processor in which the task is currently assigned. If an initial task *s* were not selected to begin a cluster, there might be set of tasks with high communication costs among them that would not be selected to be in the cluster. This would happen because each of these high-communicated tasks would be attracted to the others or, that is to say, attracted to the processor in which the task is currently assigned. Neighbourhood $N_3(x)$ is obtained by selecting all pairs of processors *k*-*l* and, for each of them, choosing at random *s* different tasks to begin a cluster, and finally generating different values for parameter α. The same ideas were used to design algorithms to find x_4 and x_5, which are detailed in Figures 4 and 5 respectively.

Reallocate a cluster of tasks from processor *k* to processor *l*, $N_3(x)$. Determine x_3

α = random number $\in$ [0-1)
Select a task s ($s \in P_k$ and $a_s = b_l'$)
Initially, $T_{kl} = \{s\}$; $b_l' := b_l' - a_s$
J is the set of tasks j that: $j \in (P_k - T_{kl})$ and $a_j = b'_l$

compute $C_j = \alpha \cdot \left(\sum_{\forall i \in (P_l \cup T_{kl})} c_{ji} - \sum_{\forall t \in (P_k - T_{kl})} c_{jt} \right) + (1-\alpha) \cdot (d_{jk} - d_{jl})$, $\forall j \in J$

While (J ? $\{\varnothing\}$ and $\max(C_j)>0$) *do*
t is the task that maximises C_j ; add t to cluster: $T_{kl} = T_{kl} + t$ and $b'_l = b'_l - a_t$
and $b'_k = b'_k + a_t$
Determine J (set of tasks j that: $j \in (P_k - T_{kl})$ and $a_j = b'_l$)
Update $C_j := C_j + 2 \cdot \alpha \cdot c_{jt} \quad \forall j \in J$
end while
x_3 is the result of reallocating tasks from T_{kl} to processor l.

Figure 3. Algorithm to find x_3

Reallocate a cluster of tasks to processor *l*, $N_4(x)$. Determine x_4

α = random number $\in$ [0-1)
Select a task s ($s \notin P_l$ and $a_s = b_l'$)
Initially, $T_l = \{s\}$; $b_l' := b_l' - a_s$
J is the set of tasks j that: $j \notin (P_l \cup T_l)$ and $a_j = b'_l$

compute $C_j = \alpha \cdot \left(\sum_{\forall i \in (P_l \cup T_l)} c_{ji} - \sum_{\forall t \in (P_k - T_l)} c_{jt} \right) + (1-\alpha) \cdot (d_{jk} - d_{jl})$, $\forall j \in J$ (where k is the processor to which j is currently assigned)

While (J ? $\{\varnothing\}$ and $\max(C_j)>0$) *do*
t is the task that maximises C_j ; add t to cluster: $T_l = T_l + t$ and $b'_l = b'_l - a_t$
Determine J (set of tasks j that: $j \notin (P_l \cup T_l)$ and $a_j = b'_l$)
Update $C_j = \begin{cases} C_j + 2 \cdot \alpha \cdot c_{jt} & \forall j \in J \cap P_k \\ C_j + \alpha \cdot c_{jt} & \forall j \in J - P_k \end{cases}$, where k is the processor to which task t was assigned before adding it to cluster T_l
end while
x_4 is the result of reallocating tasks from T_l to processor l.

Figure 4. Algorithm to find x_4

Empty a processor *k*, $N_5(x)$. Determine x_5

α = random number $\in$ [0-1)

Initially, $T_{kl} = \{\varnothing\}, \forall l \neq k$

J_l is the set of tasks j that: $j \in P_k$ and $a_j = b'_l$, $\forall l \neq k$

compute $C_{jl} = a \cdot \left(\sum_{\forall i \in P_l} c_{ji} - \sum_{\forall t \in P_k} c_{jt} \right) - (1-a) \cdot (d_{jl})$, $\forall l \neq k, \forall j \in J_l$

While ($J_l \neq \{\varnothing\}$) *do*

(t-p) is the pair task-processor that maximises C_{jl} ; add t to cluster: $T_{kp} = T_{kp} + t$ and $b'_p = b'_p - a_t$

Determine J_l (set of tasks j that: $j \in (P_k - \bigcup_{l \neq k} (T_{kl}))$ and $a_j = b'_l$, $\forall l \neq k$)

Update $C_{jl} = \begin{cases} C_{jl} + 2 \cdot a \cdot c_{jt} & \forall j \in J_p \\ C_{jl} + a \cdot c_{jt} & \forall l \neq (p,k); \forall j \in J_l \end{cases}$

end while

x_5 is the result of reallocating tasks from T_{kl} to processor l, $\forall l \neq k$.

Figure 5. Algorithm to find x_5

Size of neighbourhoods

To compute the size of the neighbours it must be considered that, if all processors were used, each of them would have, on average, n/m tasks allocated. The size of each neighbour used in the GVNS algorithm is as follows:

N1(x) $Size_1 = O(n \cdot (m-1))$

N2(x) each task could be exchanged with the n/m tasks allocated in each of the other processors. This gives a $Size_2 = O\left(\frac{n}{m} \cdot \frac{n}{m} \cdot \frac{m(m+1)}{2} \right) = O\left(\frac{n^2 \cdot (m+1)}{2 \cdot m} \right)$

N3(x) There are $m \cdot (m-1)$ combinations of pairs of processors and, for each of them, different clusters can be found by selecting different tasks to begin. To avoid repeating too many clusters, each task of the origin processor (k) is selected with a probability of 0.7. Hence, for each pair of processors (k,l), ($0.7 \cdot n/m$) clusters, on average, are determined.This gives a $Size_3 = O\left(m \cdot (m-1) \cdot \left(0.7 \cdot \frac{n}{m} \right) \right) = O(0.7 \cdot n \cdot (m-1))$

N4(x) For each processor l, different clusters can be found by selecting different tasks allocated in the other processors in order to begin the cluster. To avoid repeating too many clusters, each of these tasks is selected with a probability of 0.7. Hence, for each

processor l, $0.7\cdot(n\text{-}n/m)$ clusters, on average, are determined. This gives a

$$Size_4 = O\left(m \cdot 0.7 \cdot \left(n - \frac{n}{m}\right)\right) = O\left(0.7 \cdot n\left(m-1\right)\right)$$

N5(x) For each processor k different ways of emptying it could be found by generating different random values for the parameter α (if there are assigning costs). In the experiments this parameter was generated once, which yielded $Size_5 = O(m)$

Initial solution

The same basic ideas included in clustering procedures were used to obtain initial solutions. Although random solutions give good results, a short experiment showed that on average the following procedure is better.

Initial solution, *x*

α = random number $\in$ [0-1)
Initially, $P_k = \{\varnothing\}$, k=1,…,m
Sort processors by increasing fixed cost (break ties at random), k is the first processor
While (there are non-assigned tasks) *do*
  J is the set of non-assigned tasks j that: $a_j = b'_k$
  While (J ? $\{\varnothing\}$) *do*
    compute $C_j = a \cdot \left(\sum_{\forall i \in P_k} c_{ji}\right) - (1-a)\cdot(d_{jk})$, $\forall j \in J$
    t is the task that maximises C_j ; add t to processor k: $P_k = P_k + t$ and $b'_k = b'_k - a_t$
    Determine J (set of non-assigned tasks j that: $a_j = b'_k$)
  end while
  Go to next processor, k
end while
Initial solution, x, is determined by P_k, k=1,…,m

Figure 6. Algorithm to find initial solution

Computational experiment

The objective of the computational experiment is to evaluate the operativeness of the GVNS algorithm (that is to say, the algorithm gives good solutions in a reasonable time even for large instances) and to compare the quality of the solutions obtained with the best known procedure, which is the hybrid method developed by Chen and Lin[3].

In [4] and [5] the results of 8 real-world instances from an automobile microcomputer system and a Hughes air-defence system are reported. Chen and Lin[3] describe the way the data is randomly generated. Assigning costs (d_{ik}) are not considered in any of these papers. We programmed the hybrid method (HYBRID) of Chen and Lin[3], including assigning costs (d_{ik}), and we ran three experiments: (1) to solve using GVNS and the HYBRID the 8 real-world instances provided by Hadj-Alouane, Bean and Murty, and to compare these results with the ones obtained in [4] and [5]; (2) to solve using GVNS and the HYBRID a set of 108 randomly generated instances, without considering assigning costs (so the HYBRID is exactly the algorithm described in [3]); and (3) to solve using GVNS and the HYBRID a set of 54 randomly generated instances, including assigning costs.

Each algorithm is run 50 times and, to get a fair comparison, the maximum solving time of HYBRID is kept and GVNS is solved in two stages: (1) set as a final condition a maximum number of iterations between two consecutive improvements, which is set to n, and a maximum solving time equal to maximum HYBRID solving time (GVNS1) and (2) if solving time is equal to the HYBRID, continue solving GVNS with the same final condition as (1) but with a maximum solving time equal to 50 seconds, which can be considered a reasonable solving time for large instances (GVNS2).

Real-world instances

The main data used in Experiment 1 are as follows:

- Problems A, B, C, D, E and F: there are three instances with 20 tasks and 6 processors and three with 40 tasks and 12 processors; task requirements (a_i) range from a few up to approximately 50 units; processors capacities (b_k) range from 100 to 250 units; fixed costs (s_k) range from 1,000 to 5,000 units; communication cost matrices are very dense, with c_{ij} ranging from a few to 50 units; and assigning costs, $d_{ik} = 0$.
- Problem G: 15 tasks and 5 processors; $a_i = 1$; b_k range from 3 to 5 units; $s_k = 0$; communication cost matrices are very sparse, with c_{ij} equal to 0 or 1; and $d_{ik} = 0$.
- Problem H: 41 tasks and 4 processors; a_i range from a few up to 950 units; b_k range from 800 to 1600 units; $s_k = 0$; communication cost matrices are very sparse, with c_{ij} ranging from a few to 70 units; and $d_{ik} = 0$.

Generated data

The data used in Experiments 2 and 3 were generated as follows:

- Experiment 2: n = 20, 40, 60, 80 and 100 number of tasks; Experiment 3: n = 20, 40 and 60.
- m = 5, 10 (only for $n \geq 40$), 20 (only for $n \geq 60$) and 30 (only for n=100) number of processors
- $a_i \in \bigcup [50, 100]$; $A = \sum_{i=1}^{n} a_i$
- $b_k \in \bigcup [bmin, bmax]$:

- *loose* case: $bmin = \frac{3 \cdot A}{m}$, $bmax = \frac{5 \cdot A}{m}$ (on average, only a quarter of the processors may be necessary)
- *medium* case: $bmin = \frac{A}{m}$, $bmax = \frac{3 \cdot A}{m}$ (on average, half the processors may be necessary)
- *tight* case: $bmin = \frac{A}{3 \cdot m}$, $bmax = \frac{7 \cdot A}{3 \cdot m}$ (on average, more than the available processors would be necessary, but in practise feasible solutions are often obtained)

- $sk \in \bigcup [b_k , S \cdot bk]$, with S = 10, 50 and 100
- The communication cost between task i and task j is greater than 0 with a probability of 0.25. This rule gives sparse communication cost matrices, which are good for algorithm testing. Then, $c_{ij} \in \bigcup$ [50 , 100] (with a probability of 25%) or $c_{ij} = 0$ (with a probability of 75%)
- $d_{ik} = 0$ (Experiment 2) and $d_{ik} \in \bigcup [50 , 100]$ (Experiment 3)

Hardware and Software

The algorithms (GVNS and HYBRID) were programmed using C language and run on a PC Pentium IV at 2.6 GHz with 1024 Mb RAM. The computational experiment reported in [5] was performed on a IBM RS/6000-320H (in C language), and the algorithm was run 10 times with different seeds. Ernst et al.[4] implemented their approaches in C/C++ (using CPLEX for solving integer linear programming formulations) and ran the code on a computer using a 500MHz alpha processor.

Experimental results

The following tables (Tables 1 to 7) and figures (Figures 7 to 9) summarise the results of Experiments 1, 2 and 3. In Table 1, EJK (best lower bound and best found solution) stands for Ernst et al.[4] and HBM (best, average and worst found solutions) stands for Hadj-Alouane et al.[5]. For each instance, the best solutions are shown in bold.

Table 1 shows that, for most of the 8 real-world instances, the GVNS algorithm outperforms, in a very short solving time, the results obtained by the hybrid genetic algorithm (HBM), the column generation models in [4] and the HYBRID. Although in Experiment 1 HYBRID does not seem to outperform HBM, Chen and Lin[3] carry out a wide computational experiment and show in their paper how their hybrid method gives better results than HBM in terms of both quality solution and solving time. Hence, if in Experiments 2 and 3 GVNS outperformed the HYBRID results, it could be concluded that GVNS is also better than HBM.

	SOLUTION (min, average, max)						TIME (min, average, max)				
Problem (*n*-*m*)	**Best Low EJK**	**Best EJK**	**HBM**	**HYBRID**	**GVNS1**	**GVNS2**	**EJK**	**HBM**	**HYBRID**	**GVNS1**	**GVNS2**
A (20-6)	13,310.37	**13,450**	13,804 13,866 13,903	13,519 15,508 15,558	**13,450** 13,940 14,263	**13,450** **13,832** 14,120	35,120	3.43 25.93 87.48	0.015 0.022 0.063	0.015 0.021 0.047	0.109 0.153 0.266
B (20-6)	11,946	**11,946**	**11,946** **11,946** 11,946	**11,946** 12,018 12,320	**11,946** 11,998 12,397	**11,946** **11,946** 11,946	671.46	10.56 28.74 73.53	0.015 0.019 0.032	0.015 0.019 0.047	0.109 0.139 0.218
C (20-6)	11,120	**11,120**	**11,120** 11,228 11,864	11,156 11,268 11,315	11,126 11,285 12,039	11,126 **11,204** 11,431	14,589.12	6.94 18.95 46.45	0.015 0.020 0.032	0.015 0.019 0.031	0.109 0.184 0.453
D (40-12)	37,662.39	39,738	39,680 39,869 41,149	41,557 41,753 41,850	39,293 39,591 40,051	**39,214** **39,385** 39,833	2,440	205.2 274.9 395.9	0.374 0.409 0.515	0.172 0.250 0.359	1.875 3.331 7.859
E (40-12)	33,438.86	38,602	36,575 37,214 38,767	37,731 38,052 38,518	35,674 36,481 38,203	**35,671** **35,901** 37,953	3,436	52.79 307.6 389.5	0.375 0.411 0.468	0.172 0.250 0.390	2.047 2.950 6.890
F (40-12)	32,126.36	35,016	35,821 36,427 36,568	36,410 36,570 36,707	**34,674** 35,575 36,360	**34,674** **34,950** 35,890	5,809.13	44.8 346.8 394.9	0.422 0.481 0.532	0.204 0.305 0.453	2.578 4.952 11.187
G (15-5)	16	**16**	**16** **16** 17	no feas	**16** 17 19	**16** **16** 17	181.1	1.31 2.73 6.87	_	0.015 0.016 0.016	0.015 0.029 0.078
H (41-4)	40	**40**	_	**40** 45 52	**40** **40** 48	**40** **40** 44	0.29	_	0.281 0.313 0.625	0.109 0.154 0.188	1.125 1.485 2.375

Table 1. Results of experiment 1 (8 real-world instances)

Table 2 summarises the main results of Experiment 2 regarding the objective function, showing that with a Variable Neighbourhood Search algorithm better solutions are obtained than with the hybrid method. On average, our algorithm outperforms the HYBRID 72.22% of the times, and in these situations the percentage of improvement is quite high (5.52% on average). Only for 27.77% of the instances are the results of the Chen and Lin algorithm[3] better than ours, and in these cases the percentage of improvement is not very high (1.39%). The improvement of GVNS2 compared with GVNS1 is not very great and it needs longer solving times (see Table 5). This led us to conclude that the final condition of *n* iterations between two consecutive improvements may be too much and a shorter number of non-improvement iterations could be used instead of *n*.

In Table 3, the percentage of improvement (average) of the GVNS and HYBRID algorithms is detailed by capacity case (loose, medium or tight) and *S* (related to fixed costs). The improvement offered by our algorithm is greater in situations in which the number of required processors (on average) is greater than the number available (loose and medium cases). This is not surprising, as these are exactly the cases in which it is possible to take greater advantage of the new neighbourhoods. In most solutions of the *tight* case, the remaining capacity of the processors may be very low, and it may very difficult, or even impossible, to reallocate a cluster of tasks to a processor or to empty a processor, which is exactly what is done in moves 3, 4 and 5. Hence, there may not be a great difference between the results of GVNS and those of HYBRID. On the other hand, the improvements offered by both algorithms are approximately the same for the different values of *S* (fixed costs). GVNS takes advantage of emptying a processor because this move allows it to lower fixed costs, but the HYBRID method begins with a solution in which all tasks are allocated to a cheapest processor, so the final solution is also good in terms of fixed cost.

Final condition GVNS	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	72.22	27.77	5.52	1.39
GVNS2	74.1	25.9	6.03	0.87

Table 2. Results of experiment 2 (generated data set without assigning costs)

* % Improvement G (only if $f_G < f_H$) $= 100 \cdot (f_H - f_G) / f_H$

** % Improvement H (only if $f_H < f_G$) $= 100 \cdot (f_G - f_H) / f_G$

Final condition GVNS	capacity case	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	lose	77.77	22.22	2.96	0.77
	medium	75	25	11.70	1.74
	tight	63.88	36.11	1.39	1.52
GVNS2	lose	80.56	19.44	3.42	0.45
	medium	75.00	25.00	12.77	0.32
	tight	66.67	33.33	1.61	1.54

Table 3. Results of experiment 2 (generated data set without assigning costs) by capacity case

* % Improvement G (only if $f_G < f_H$) $= 100 \cdot (f_H - f_G) / f_H$

** % Improvement H (only if $f_H < f_G$) $= 100 \cdot (f_G - f_H) / f_G$

Final condition GVNS	*S* (fixed cost)	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	10	77.78	22.22	4.67	1.39
	50	66.67	33.33	6.04	1.48
	100	72.22	27.78	5.96	1.27
GVNS2	10	77.78	22.22	5.27	0.99
	50	69.44	30.56	6.49	0.87
	100	75.00	25.00	6.40	0.78

Table 4. Results of experiment 2 (generated data set without assigning costs) by *S*

* % Improvement G (only if $f_G < f_H$) $= 100 \cdot (f_H - f_G) / f_H$

** % Improvement H (only if $f_H < f_G$) $= 100 \cdot (f_G - f_H) / f_G$

The final condition set for GVNS1 ensures that its solving time is always equal to or shorter than the maximum HYBRID solving time. Obviously, both algorithms need more time when the number of tasks (*n*) and the number of processors (*m*) grow (see Table 5 and Figures 7 and 8), but the results confirm that the GVNS algorithm is very efficient and can be used even for large instances.

	Solving times (min, average, max)						
n	**HYBRID**	**GVNS1**	**GVNS2**	***m***	**HYBRID**	**GVNS1**	**GVNS2**
20	0.01	0.02	0.14	5	1.93	2.40	12.41
	0.01	0.03	0.21		2.05	2.51	15.57
	0.02	0.05	0.36		2.37	2.77	22.42
40	0.29	0.37	2.89	10	9.96	12.06	23.59
	0.32	0.43	4.03		10.56	13.26	29.02
	0.37	0.66	7.76		14.06	15.20	37.81
60	2.36	3.07	15.97	20	15.70	21.09	34.20
	2.59	3.21	23.83		16.76	23.48	42.75
	3.03	3.54	38.32		23.76	25.14	50.65
80	8.72	11.41	34.89	30	34.52	49.14	50.00
	9.28	11.89	42.11		39.45	56.61	59.20
	11.55	12.73	49.43		56.50	58.98	62.10
100	31.79	42.02	46.03				
	34.64	47.98	53.24				
	49.66	52.04	58.61				

Table 5. Experiment 2 (generated data set without assigning costs). Solving times by *n* and *m* (final condition for GVNS2 includes a maximum solving time of 50)

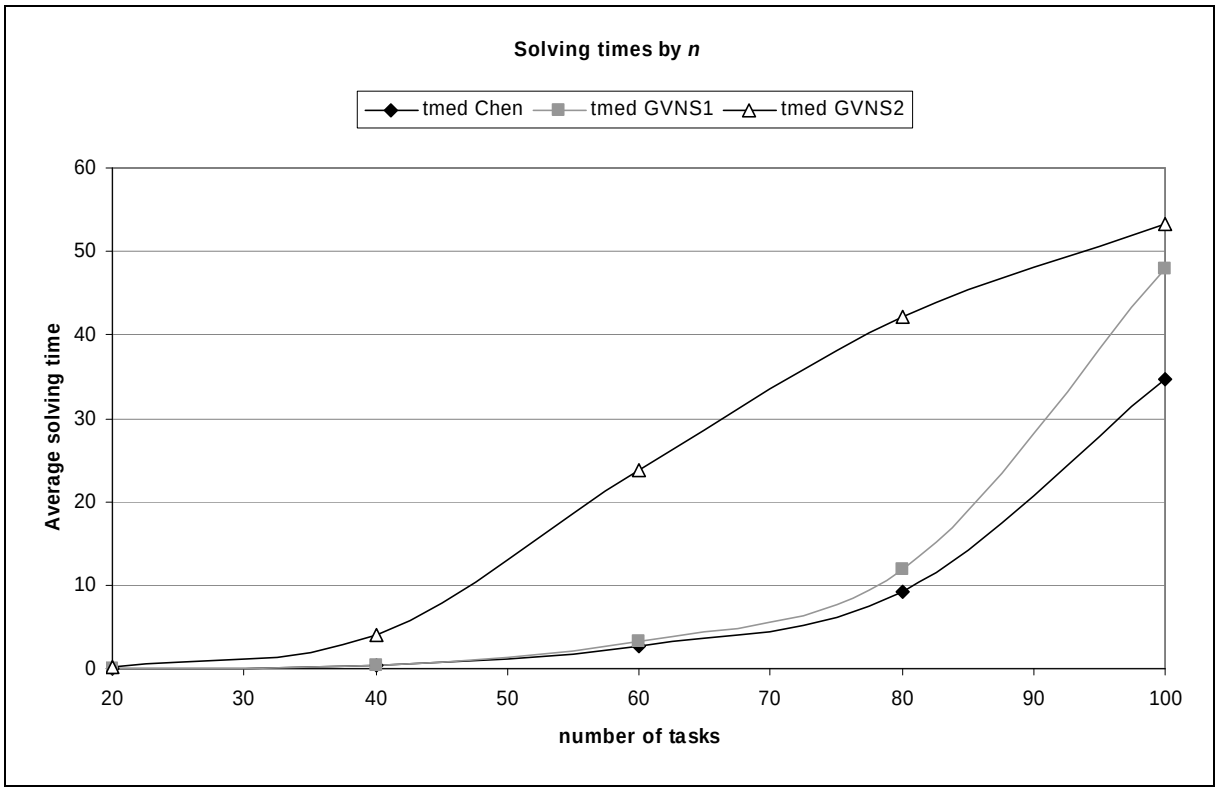

Figure 7. Experiment 2 (generated data set without assigning costs). Solving times by *n*

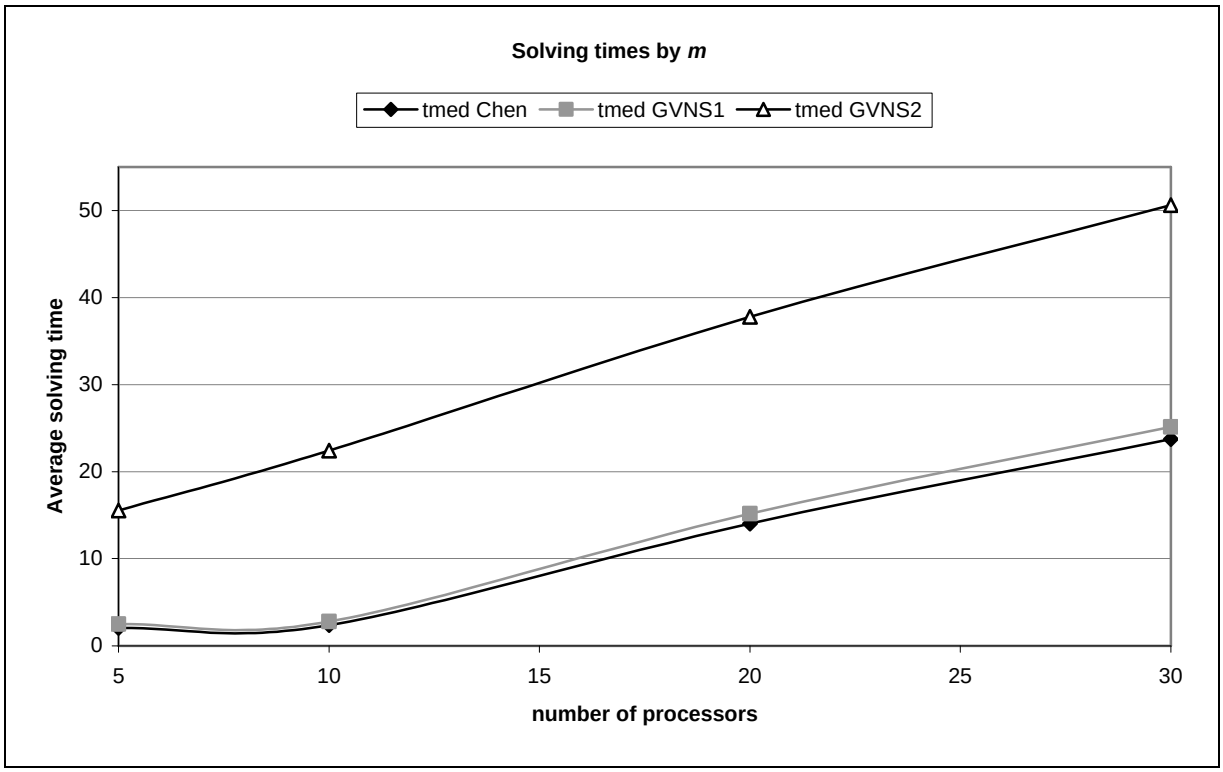

Figure 8. Experiment 2 (generated data set without assigning costs). Solving times by *m*

Finally, Tables 6 and 7 and Figure 9 summarise the main results of Experiment 3 regarding the objective function and solving times. Again, the GVNS algorithm gives better results than the HYBRID method in comparable solving times.

Final condition GVNS	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	62.96	37.04	5.99	1.03
GVNS2	62.96	37.03	7.09	0.77

Table 6. Results of experiment 3 (generated data set with assigning costs)

* % Improvement G (only if $f_G < f_H$) $= 100 \cdot (f_H - f_G) / f_H$

** % Improvement H (only if $f_H < f_G$) $= 100 \cdot (f_G - f_H) / f_G$

	Solving times (min, average, max)		
n	HYBRID	GVNS1	GVNS2
20	0.01	0.03	0.15
	0.01	0.04	0.20
	0.03	0.06	0.33
40	0.29	0.41	3.10
	0.32	0.47	4.27
	0.41	0.60	8.03
60	2.32	3.14	17.14
	2.53	3.29	26.60
	3.10	3.77	41.37

Table 7. Solving times of experiment 3 (generated data set with assigning costs)

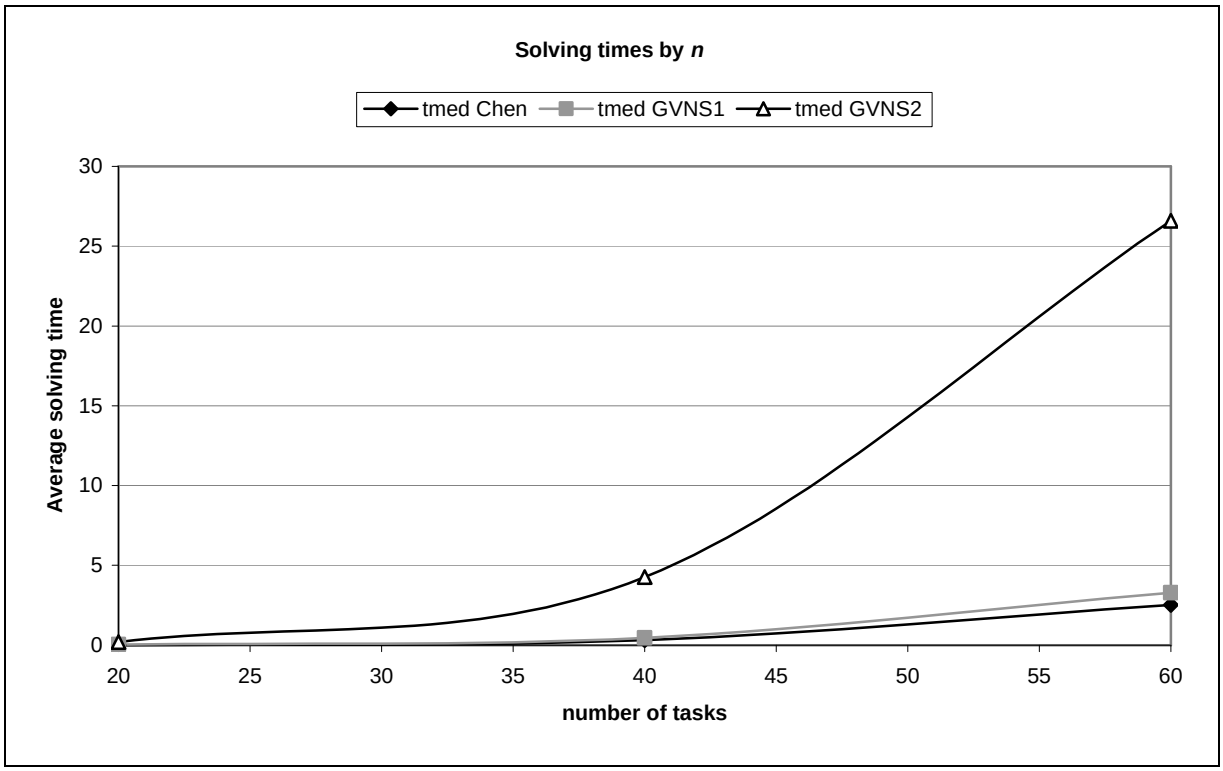

Figure 9. Experiment 3 (generated data set with assigning costs). Solving times by *n*

Conclusions

The Constrained Task Allocation Problem (CTAP), which has been shown to be NP-hard, consists in assigning a set of tasks to a set of processors so that the overall cost is minimised. This cost includes a fixed cost of using a processor, a task assigning cost (which may depend on the task and processor) and a communication cost between tasks that are assigned to different processors.

In this paper, a Variable Neighbourhood Search algorithm for dealing with the CTAP is proposed. Three new neighbourhoods are added to the neighbourhoods traditionally used (reallocating a task and exchanging two tasks): (1) reallocating a cluster of tasks from one processor to another; (2) reallocating a cluster of tasks from different processors to another processor; and (3) emptying a processor by reallocating its assigned tasks to other processors. Three clustering algorithms, which include assigning and communication costs, were designed to find the three new neighbourhoods.

An extensive computational experiment showed that the VNS algorithm outperforms the existing algorithms both in terms of quality solution and computing times.

Acknowledgements

The authors are grateful to James Bean and Atidel Hadj-Alouane for providing test data.

References

Chen W-H., Lin, C-S. (2000). A hybrid heuristic to solve a task allocation problem. *Computers & Operations Research* **27**: 287-303.

Ernst, A., Jiang, H., Krishnamoorthy, M. (2003). Exact Solutions to Task Allocation Problems. *Working Paper*.

Hadj-Alouane, A.B., Bean, J.C., Murty, K.G. (1999). A Hybrid Genetic/Optimization Algorithm for a Task Allocation Problem. *Journal of Scheduling* **2**: 189-201.

Hamam, Y., Hindi, K.S. (2000). Assignment of program modules to processors: A simulated annealing approach. *Eur J Opl Res* **122**: 509-513.

Hansen, P., Mladenovic, N. (1997). Variable neighbourhood search for the p-median. *Location Science* **5**: 207-226.

Hansen, P., Mladenovic, N., Moreno J.A. (2003). Variable neighbourhood search. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial* **19**: 77-92.

Rao, G.S., Stone, H. S., Hu, T.C. (1979). Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers* **C-28:** 291-299.

Rao, K (1992). Optimal synthesis of microcomputers for gm vehicles. *Technical Report*.

Stone,H.S.(1977).Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* **3**: 85-93.