A Parallel Tabu Search Algorithm For The 0-1 Multidimensional Knapsack Problem

Smail Niar, Arnaud Freville
Université de Valenciennes , LIMAV
Le Mont Houy BP 311 - 59304 Valenciennes Cedex-France
niar, freville@univ-valenciennes.fr

Abstract

The 0_1 Multidimensional Knapsack Problem (0_1 MKP) is a NP-complete problem. Its resolution for large scale instances requires a prohibitive processing time. In this paper we propose a new parallel meta_heuristic algorithm based on the Tabu Search (TS) for the resolution of the 0-1 MKP. We show that, in addition to reducing execution time, parallel processing can perform automatically and dynamically the settings of some TS parameters . This last point is realized by analyzing the information given by cooperative parallel search processes, and by modifying the search processes during the execution. This approach introduces a new level where balancing between intensification and diversification can be realized.

1 Introduction

Since many years, parallel computers have been used to solve hard problems in different areas. In the field of combinatorial optimization,parallel computers offer the advantage of reducing the execution time and give the opportunity to solve new problems which we could not attack by sequential computers which are limited in execution speed and memory capacity. The 0-1 MKP is a combinatorial optimization problem which is often used for modeling different applications like capital budgeting or resource allocation [8]. It can be defined as follows:

$$MAX \ \sum_{j=1}^{n} c_j * x_j$$

$$subject\ to\ \sum_{j=1}^{n} a_{ij} * x_j \leq b_i,\ \ i = 1..m.$$

$$x_j \in \{0,1\},\ \ j = 1..n$$

where a_{ij}, $b_i$and c_j are all positive real_valued constants. Solving 0-1 MKP of large dimensions (n> 500 and m>25) by an exact approach (like Branch and Bound or Dynamic Programming) requires a great amount of time. Here the proposed solution consists in using the Tabu Search (TS) meta-heuristic as a resolution algorithm and performing a parallel exploration of the solution domain. The aim of our work is to show that parallel processing can :
- reduce the execution time of the TS algorithm
- allow the resolution of large size 0-1 MKP
- improve the quality of the final solution
- allow the implementation of new TS strategies.
- Tune automatically and dynamically some parameters of the TS algorithm.

2 Parallelism and Tabu search

Different sources of parallelism exist in TS algorithm. Four of these sources are often pointed out.
- Parallelism in cost function evaluation.
- Parallelism in neighborhood examination and evaluation
- Parallelism in problem decomposition
- Parallelism in solution domain exploration by maintaining different independent or dependent search paths.

The first two sources of parallelism represent low level approaches to parallelism. In this case, phases of the algorithm which are time consuming are delegated to a specialized parallel computer[2]. The third source of parallelism in TS has been used by Taillard [10] to solve the vehicle routing problem.
In the last type of parallelism, different independent parallel processes, called *parallel search threads*, are created. Each search thread consists in executing a TS algorithm from an initial solution and using a set of parameters. This set of parameters determines the behavior of the TS processes and specifies for each processor a search *Strategy* to be executed. For example, a strategy is represented by : the length of the Tabu list, the number of neighbor solutions evaluated

at each move, the move realized at each iteration, ...etc.
The present paper is concerned with this last type of parallelism applied to 0-1 MKP of large size. Regarding to other types, the parallelization of TS by maintaining parallel threads has the advantage of minimizing the communication overhead between threads by implementing coarse grain parallelism. This kind of parallelism is well suited to MIMD parallel computer with distributed memory[11].
The parallel search threads have the possibility to exchange informations. In this case, parallel search threads are called *Synchronous* threads if communications between threads are realized at predetermined moments (rendez_vous). Otherwise, if these information exchanges are realized at different moments, determined by the internal state of the thread, the communication scheme is called *Asynchronous*.
Although it is very easy to build search algorithm which consists in several independent search threads, this approach has not a great interest and is equivalent to execute several times the same sequential TS algorithm with different starting solutions and/or different search parameters. Despite of its poor performances this approach has been greatly used with more or less good results. When independent search threads are executed, it is easy to make the following remarks:
1- After a certain number of TS iterations, it is interesting to stop a search thread when no improvement of the best solution is encountered for a great number of iterations. This may be due either because the search is done in a "non_interesting" region or because the search parameter values are not adapted to the problem instance under test. The stopped process may be reinitialized with a new "interesting" solution and/or a new set of parameters.
2- It is useful to stop a search thread when the region, where the search is done, have been explored (or is under exploration) by another thread. The stopped process will be reoriented to a new not yet explored region.

3 The Search Algorithm

In this paragraph we will present the sequential part of the TS algorithm which is executed by a processor. The figure 1 presents the general statement of this algorithm.

3.1 The Local Search Phase

The basis of the TS algorithm are those proposed by Glover [5]. Each slave processor k starts with an initial solution X_{init} and uses a strategy St_k. During each iteration of the process a neighborhood N(X) of the current solution X is examined in order to select the best solution X' as a new solution.
The selected solution X' must not be "Tabu". A Tabu status of a solution avoid cycling when non monotonic exploration of the domain is realized, i.e when solutions of less quality

```
PROCEDURE Tabu_search ($X_{init}$, Nb_div, Nb_int, Nb_local,
        Nb_Drop, Lt_length, BestSol_array)
/* X is the actual solution */
1- X=$X_{init}$; $Lt_{Lt\_length}$={};
2- For i=0 to Nb_div do
   3- For j=0 to Nb_int do
         4- $X^*_{local}$=X;
         /* move */
         5- go from X to X' by a sequence of Nb_Drop/Add;
         6- If F(X')> F($X^*$) then $X^*$=X' ;$X^*_{local}$=X';
            Else If F(X')>F($X^*_{local}$) Then $X^*_{local}$=X';
         7- If X' is apart of the *B* Best solutions
            Then insert X' in the BestSol_array;
         8- X = X' ; update *History*;
         9- $Lt_{Lt\_length}$=$Lt_{Lt\_length}$+ X;/* X is Tabu*/
         10- If no improvement of F($X^*$) during Nb-local
         iterations go to 10, Else go to 4 ;
      11- Intensification($X^*_{local}$, $X^*$);
   12- Diversification(*History, X*);
```

Figure 1. The general form of the TS algorithm

are accepted. Nevertheless, this Tabu state "Barrier" may be left for a solution X', if the corresponding objective function value F(X') is better than the best solution cost F(X^*) found so far. This mechanism is called the aspiration criteria. Once the new solution X' is selected, a transformation (a move) is applied in order to transform X in X'.
Like in [3] a move consists of 2 steps:
1)Drop: This step selects *Nb_drop* components fixed at 1 in the actual solution X and resets these components to 0. The choice of the components to drop is made as following:
i^*= Arg Min $\{(\sum_{j=1}^{N} a_{ij} * x_j) - b_i|$i= 1,2 ... m$\}$
i^* is the index of the most saturated constraint
j^* = Arg max$\{a_{i^*j}/c_j | x_j = 1$, j=1, ..., n$\}$ and j $\notin Lt_{Lt\_length}$;
2)Add: Here one or several components fixed at 0 are chosen and set to 1. The selected component must not be Tabu except if the aspiration criteria is realized. Adding object to the knapsack is realized until no object can be added.
The intensification and diversification phases complete the local search. The role of the intensification is to drive the search towards interesting region (promising region) by taking into account characteristics found in high quality solutions in the selection of future solutions to visit. The diversification phase, on the other hand, forces the search to explore regions containing unvisited solutions.

3.2 The Intensification Phase

In our project, two intensification procedures have been used.

Intensification by swapping components

The first intensification procedure tested in our project consists in taking the best solution found during the last local search loop X^*_{local}, selecting a component X^*_{local}[i] having the value 1 and performing exchange with another component X^*_{local}[j] which has the value 0 with C[j]>C[i]. This exchange is realized for each couple (i, j) satisfying the previous conditions.

Intensification by strategic oscillation.

Strategic oscillation, consists in crossing the feasible domain boundary by accepting infeasible solutions during a fixed number of iterations. Then the infeasible solution is projected onto the feasible domain to generate a new feasible solution. This projection is performed by excluding from the knapsack the less interesting objects (i.e those with large $\sum_i a_{ij}/c_j$ ratio). This method has been successfully applied to the 0-1 MKP [6] [7]. Yet the main drawback is its extra computing time due to the exploration of a great number of infeasible solutions. In order to reduce this execution time, we have limited the number of explored infeasible solutions by limiting the depth of the search path in the infeasible domain[9].

3.3 The Diversification Phase

The diversification step uses a memory containing informations relative to visited solutions since the beginning of the search. This memory, called *long term memory*, is used in order to favoring the exploration of these new regions.
The diversification phase starts by generating a new starting solution X_{diver}. Rather than taking this new solution randomly, X_{diver} is generated by taking into account the most frequently components set to 0 or 1.
We use an array called *History*. The value of History[i] represents the number of iterations where the component i of the current solution is set to 1. By this way, if History[i] has a value greater (rep. less) than a threshold, X_{diver}[i] is set to 0 (rep. to 1), and the component i is set Tabu. By this manner, search is forced to go into region which was neglected in the previous searches. The search is limited to this new region during a fixed number of iterations before continuing in normal conditions.

4 The parallel TS algorithm for the 0_1 MKP

4.1 Dynamic TS parameter setting

Our TS approach has the advantage to be a flexible method and easily adaptable for solving different problems in combinatorial optimization. But to be efficient, it requires to tune parameter values controlling the search. In the previous section, the term Strategy has been used to name the set of parameters which must be fixed before starting the execution of the TS. For instance, the Tabu list length value (tl_length) must be large in order to avoid cycling and loosing time in reevaluating several times the same solution. Conversely, if tl_length is very large no movement will be allowed after an initial phase. Indeed, the process does not allow to accede to solutions which have never been encountered and may reduce the number of accessible neighbor solutions. The tl_length has also an influence on the progress of the TS. In fact, if this value is small, the last best visited solutions becomes quickly accessible, allowing by this way, to come back to regions containing solutions of good quality. In this way, a small value of tl_length allows the search to realize an intensification in a region containing good solutions. In counterpart, when tl_length is large, the search will be forced to accept less good solutions. This will give an opportunity to the search to go out from regions containing a great number of good solutions and will diversify the search.
In our opinion, the proposed methods for controlling dynamically the tl_length involve a great time and memory space overheads. Among these methods, we quote the Reverse Elimination Methods (REM) [3]. This method is based on the building of list containing all the moves executed from the initial configuration (the running list). In spite of its good performances for a set of problems, this method has the drawback of having a time overhead proportional to the number of executed iterations. Battiti and Tecchioli [1] propose another method called Reactive Tabu search. it consists in using aside the classic Tabu list another data structure (hashing table) which contains objective function values of all visited solutions. The using of hashing function for MKP of great size will produce a great number of collisions and this will lead to an important overhead.
Another parameter in the strategy permit also to balance between intensification and diversification. Experimental tests [9] have shown that, when the number of consecutive drops (nb_drop) done in a move is small (less than 3), the objective function changes less rapidly and the visited solutions are close ones another. When the value of nb_drop becomes high, the variations in the objective function are more important and the visited solution are distant ones another. Here the proposed solution to the problem of setting these two parameters, consists in using parallel computer architecture with multiple independent processors. In this architecture, one processor has the status of *Master*. The process executed by the master (the master process) consists in controlling the search threads (*Slave* processes) by modifying dynamically the parameters which governs the search thread.

```
Procedure Master_Process(P, Nb_search_it)
  Read and send to slaves problem data
  For i=1 to Nb_search_it do
    Call SGP(P, Data_struc, $\alpha$) and ISP(P, Data_struc)
    Send Initial solutions and strategies to slaves
    Receive from each slave its B best solutions
```

Figure 2. The master process

4.2 The master process

The master process executes an iterative program. At each (search) iteration, P slave processes are launched. To start a search, the master process must give to each slave a starting solution (an initial solution) and a search strategy (synchronous centralized communication scheme). From a functional point of view, the master process is composed of three procedures and uses one data structure. Those procedures are : the main procedure, the strategy generation procedure (SGP) and the initial solution generation procedure (ISP). The data structure used by the master process is an array of P entries. The entry i corresponds to informations given to or by the slave processor i and contains four items:
- The search strategy (three values) (St_i)
- The initial solution used by the slave (S_i)
- The B best solutions found by the slave i ($best_i$)
- The score of the slave i ($score_i$)
The figure 2 presents the master process. It is noteworthy that slaves processors must terminate their search (approximately) at the same time. Indeed, when the elapsed time between two ends of search is important, this will penalize the shortest search. In order to get the new starting solution and the new search strategy, each slave must wait until all other slaves terminate their search thread in the previous search iteration. The parameter Nb_drop has a great influence on the execution time. Indeed, more this parameter is high more long will be the search time. By consequent, one way to balance the execution times of the different slave processors, is to give a value to Nb_it which is proportional to Nb_drop conversely.

- The initial solution generation procedure (ISP)
  The purpose of this procedure is to provide to slaves P initial solutions for the next search iteration . As a first step, for each entry i in the data structure, the next initial solution (S_i) will be the best solution found by the processor i (S_i^*). Nevertheless, this solution will be substituted by another solution if one of the following conditions happens:
  1)Its cost C(S_i^*) is less than a fraction (α) of the best cost found by all processors since the beginning of the search (c(S^*)). In this case, S^* will assigned to S_i. Consequently solutions having smaller cost are eliminated from the initial solution pool and replaced by S^* [11].
  2)An initial solution S_i has not been modified during a fixed number of iterations, it will be substituted by a new randomly generated solution.
  By changing dynamically (in the main procedure) the value of the parameter α, it is possible to force or to forbid threads to realize search in the same region. This may be considered as a form of (macro) intensification. A small value of α coupled with introduction of new random solutions will give to threads the ability to explore different regions at each iteration. This simulate a (macro) diversification of the search.

- The Strategy generation procedure (SGP)
  The role of this procedure is to produce P strategies at each search iteration. In the actual version of the program, a strategy is characterized by three parameters:
  - The Tabu list size (Lt_length)
  - The maximum number of consecutive drops (Nb_drop)
  - The number of iterations in local search before starting an intensification (Nb_local).
  As mentioned in the previous section, a score is affected to each strategy to reflect its performance. This performance is measured by the capacity of the strategy to find solutions of high quality.
  Initially, the parameter $score_i$, is set to a predetermined value (four in the actual version). At each search iteration, $score_i$ is incremented if the final solution cost returned by the slave i (C_i^*) is higher than the initial solution cost (C_i). Otherwise, (i.e if $C_i^* < C_i$) $score_i$ is decremented. Once $score_i$ reaches the value 0, st_i is removed and new values are affected to each parameter for st_i.
  These new values may be chosen randomly or in a clever manner by using the B best solutions returned by the slave i. If the B best solutions found by a slave are in close areas, we conclude that this slave has not visiting a lot of areas. In this case, it is interesting to increment lt_size_i and nb_drop_i and to reduce the nb_it_i parameter. The hamming distance is used to compute the distance between solutions. In the opposite, if the B best solutions are very far ones another, the master processor (by executing the SGP) will force slave processors to do intensification around one (the best solution found so far) or several solutions. This operation is realized by reducing the values of the lt_size_i and nb_drop_i parameters and incrementing the value of nb_it_i parameter.

5 Computation results

The developed program, written in C language, has been tested on 2 sets of 0_1MKP. The parallel architecture used during tests is the Farm of 16 Alpha processors. These processors have a pick performance of 500 MIPS and are connected by a high speed optic fiber crossbar (16X16 links of 200Mb/sec each). Communication between processors are realized by using the PVM library. The first set of problems is proposed in Fréville and Plateau [4]. This first benchmark is composed of 57 problems. The number of variables varies from 6 up to 105 and the number of constraints from 2 up to 30. The optimal solution is reached for all these problems in less execution time than to those in [3]. The second set of problems has been proposed in Glover and Kochenberger [6]. This set consists in 24 0-1MKP of size ranging from 3*10 up to 25*500. Numerical results are reported in Table 1 (see [9] for more details).

Prob_nbr	m*n	Max.Exec.Time	Dev. in %
1 to 4	3*10	0.14	0.0
5 to 8	5*15	0.63	0.0
9 to 14	10*30	1.4	0.0
15 to 17	15*50	45.2	0.1
1 to 22	25*100	300.2	0.1
23	15*200	4683	0.3
24	25*500	9065	0.91

Table 1 : Computational results for Glover-Kochenberger Problems

The execution times for these two benchmarks are very short comparing to those given in [7] which do not include the time needed to find the suitable parameter values for the TS algorithm. In order to show the benefits of our approach, we present in Table 2 the cost of the best solution found by four approaches of the TS for a fixed execution time :
-SEQ : One sequential TS. The strategy parameters and the initial solution are chosen randomly.
-ITS : P parallel independent TS threads with neither communication nor strategy parameter modification.
- CTS1 : P parallel cooperative TS threads with communication but with no modifying strategy parameters.
-CTS2 : P parallel cooperative TS threads with communication and dynamically strategy parameter settings.

Prob	SEQ	ITS	CTS1	CTS2	Exec Time
MK15	2264	2273	2286	2294	50
MK16	1818	1862	1878	1880	50
MK17	1336	1350	1350	1361	50
MK23	9038	9089	9130	9179	100
MK24	8763	8823	8846	8903	600

Table 2 : Comparison of the four approaches

6 Conclusion and future extension

In this paper we present a parallel implementation of TS for the 0_1 MKP allowing both reducing execution time and setting dynamically strategy parameters. In fact, we have shown that parallel cooperative TS threads allow better search orientation in the different slave processors by doing either an intensification in a promising zone or diversification in order to explore a neglected zone. We have also shown that parallel cooperative search may be used in order to unload the user from the task of finding the efficient TS parameters for each problem instance. Moreover, modifying TS parameters dynamically create another level of balancing between intensification and diversification phases.
In future work, we project to replace the centralized synchronous communication scheme (master_slave model), by a decentralized asynchronous communication scheme.

References

[1] Battiti.R and G.Tecchiolli. The reactive tabu search. *ORSA Jour. on Comp.*, 6(2), 1994.

[2] Chakrapani.j and J.Skorin-Kapov. Massively parallel tabu search for the quadratic assignment problem. *Annals of Op. res.*, (41), 1993.

[3] Demmeyer.F and S.Voss. Dynamic tabu list management using the reverse elimination method. *Annals of OR*, (41):31–46, 1993.

[4] Fréville.A and G.Plateau. Hard 0-1 test problems for size reduction methods. *Investigacion Operativa*, (1):251–270, 1990.

[5] Glover.F. Tabu search part 1. *ORSA Journal on Computing*, 1(3), 1989.

[6] Glover.F and G. Kochenberger. Critical event tabu search for multidimensional knapsack problems. In Osman and Kelly, editors, *Meta-Heuristics : Theory and applications*. Kluwer Academic Pub, 1996.

[7] Hanafi.S and A.Freville. An efficient tabu search approach for the 0-1 mkp. To appear in European Journal of OR, 1996.

[8] Martello.S and P.Toth. *Knapsack Problems:Algorithms and computer implementation*. Wiley & Sons, 1990.

[9] Niar.S and A.Freville. Parallel resolution of the 0-1 MKP. Technical Report 1997_01, LIMAV University of Valenciennes, 1997.

[10] Taillard.E. Parallel iterative search methods for vehicle routing problems. *Networks*, (23):661–673, 1993.

[11] Toulouse.M, T.G.Crainic, and M.Gendreau. Communication issues in designing cooperative multithread parallel searches. In Osman and Kelly, editors, *Meta-Heuristics : Theory and applications*. Kluwer Academic Pub, 1996.