A proposal of an hybrid meta-strategy for Combinatorial Optimization Problems

Jose M. Framinan *
Industrial Management, School of Engineering, University of Seville
Rafael Pastor
Universidad Politecnica de Catalunya

August 16, 2004

1 Introduction

In this paper, we propose a procedure for dealing with Combinatorial Optimization Problems (COP). This procedure combines features of exact enumeration methods (i.e. branch and bound) with local search procedures. Regarding the combination of exact and approximate methods for COPs, we are aware of the papers by [5] and [1], but not of similar designs to the one that we present here.

In this paper, we employ as local search procedure the so-called Complete Local Search (CLM) metaheuristic [2]. CLM handles three lists of solutions. The first one, called `LIVE`, stores solutions that are available for the heuristic for future exploration. A second list, called `DEAD`, contains solutions that were in `LIVE` at some stage, and have been already explored. The third list, called `NEWGEN` is a temporary store for new solutions being generated by the heuristic during the current iteration. CLM starts with a given input solution(s) stored at `LIVE`, while `DEAD` and `NEWGEN` are initially empty. On each iteration of CLM, a solution is picked from `LIVE` and explored, i.e. all its neighbors are generated and the solution is sent to `DEAD`. From the set of neighbors, all of them whose objective function is better than a threshold value τ are checked for membership in `LIVE`, `DEAD` and `NEWGEN`. If not in these lists, the neighbor is put in `NEWGEN`. At the end of the iteration, all solutions in `NEWGEN` are transferred to `LIVE` for the next iteration of CLM. By proper parameter setting, CLM can be transformed into exhaustive search, generic local search with steepest descent or random descent, simulated annealing, or a memory-based metaheuristic in a slightly differ manner than tabu search. Its performance is controlled by a number of parameters such as the threshold value τ, the memory size of the lists, etc. For further details on CLM, the reader is referred to [2]. The remainder of the paper is as follows: first we describe the algorithm and their main features. In section 3 we apply the algorithm to a well-known COP problem: the permutation flow shop scheduling with makespan criterion. Some preliminary computational experience is presented, and section 4 is devoted to some comments and lines for future research.

2 Description of the algorithm

The high-level sketch of the proposed algorithm is the following:

```
procedure Algorithm()
 S_best ← ObtainInitialSolution()
 do
      for K_b iterations: ExploreTree()
      for K_c iterations: PerformCLM()
      ClearList(LIVE)
 while StoppingCondition() = FALSE
```

With respect to the procedures employed, procedure `ClearList(LIVE)` deletes all solutions stored in the list `LIVE` employed by CLM. Procedure `StoppingCondition()` establishes the termination criteria of the algorithm. Finally, `PerformCLM()` performs one iteration of CLM, i.e. takes the first solution in list `LIVE`, examines all neighbors that have not been explored (they are not in `LIVE`, `DEAD`, and `NEWGEN`), and add those passing a threshold to list `LIVE` for further exploration. The only difference with respect to the original CLM is that in our procedure, neighbors are checked not to belong to list `FORBIDDEN` (discussed in next subsection). Finally, `ExploreTree()` is described in the next subsection.

*Present address: Escuela Superior de Ingenieros, Camino de los Descubrimientos, s/n E 41092 Seville, Spain. Email: jose@esi.us.es

2.1 Procedure ExploreTree()

An iteration of `ExploreTree()` opens a promising node (constituted by a partial solution stored in a list `TREE`) into a number of nodes of the next level. A lower bound for the nodes is calculated in order to branch the fathomed nodes, who are included in the list `FORBIDDEN`. The purpose is to prevent CLM to explore solutions that are 'children' of these nodes, as mentioned before. Among the non fathomed nodes, the most promising one is selected so it can guide the CLM procedure to a region where good solutions can be found (this is done by employing a weighting function that will be discussed later). As the CLM procedure requires a complete solution, a complete solution should be generated from the partial solution represented by this specific node (this procedure is discussed later in subsection 2.1.3). Once a complete solution is generated, it is added to `LIVE` so it can be explored by the CLM algorithm. In detail, the algorithm operates as follows:

```
procedure ExploreTree()
 begin
      S ← GetNodeFromTree()
      delete selected node from list TREE
      if adding a component c to S produces a complete solution:
           S_c ← by adding solution component c to S.
           if UB(S_c) < f(S_best), then S_best ← S_c
           add S_c to LIVE
      else
           for all components i that can be appended to the partial solution S:
                S_i ← by adding solution component i to S.
                LB(S_i) ← CalculateLowerBound(S_i).
                if LB(S_i) < f(S_best)
                     add S_i to TREE.
                else
                     add S_i to FORBIDDEN.
                end
           end
           select S_r := {S_r : LB(S_r) ≤ LB(S_i)}.
           S ← ObtainCompleteSolFromNode(S_r).
           if f(S) < f(S_best), then S_best ← S
           add S to LIVE
      end
      if S_best has been improved, then UpdateTree()
end of procedure ExploreTree()
```

Basically, the procedure `ExploreTree()` selects one node from the list `TREE` according to a weighting function -if the nodes in `TREE` are sorted according to this weighting function, then it simply picks the first node from the list. If adding a new component to the node makes a complete solution (the maximum level in the depth of the branch and bound has been reached), then this solution is compared to the best-so-far solution. If it is better, then a new best-so-far solution is found. The solution is added to `LIVE` as a solution whose neighborhood is worth to be investigated by the CLM procedure. In case that adding a new component to the node does not make a complete solution, then a good complete solution is obtained by making use of procedure `ObtainCompleteSolFromNode()`, which is explained in section 2.1.3.

2.1.1 Procedure GetNodeFromTree()

This procedure basically picks a node from the node list in `TREE`. If the nodes in `TREE` are sorted in ascending order of the weighting function, then this procedure takes the most promising node of `TREE` regardless the length of the partial solution (or equivalently, the level of the node in the branch and bound terminology). In contrast, if the nodes in `TREE` are sorted according to their level (length of the partial solution) and ties are broken according to the weighting function, then the procedure performs a 'first depth search' in the node tree.

If, when invoking this procedure, there is no more nodes in `TREE`, then the algorithm stops as the optimal solution has been found.

Finally, with respect to the weighting function, it is clear that it has a crucial role in the performance of the algorithm. Some judicious choices for the weighting function include:

- The lower bound (LB) of the node.
- The average between the lower bound and an upper bound (UB) of the node, i.e. $(UB + LB)/2$

2.1.2 Procedure UpdateTree(S)

This procedure deletes from list `TREE` all nodes whose lower bound is greater than $C_{max}(S)$. This procedure is invoked whenever the current best solution is improved. Or, in the branch and bound terminology, whenever a new upper bound is found.

2.1.3 Procedure ObtainCompleteSolFromNode(S)

There are several sensible ways to obtain a complete solution from the partial solution represented by the most promising node. These include:

- `LIVE` **based- procedure**. This procedure consists in searching in list `LIVE` for complete solutions in the vertex of this node and place them as next solutions to be explored by CLM. When using this option, it should be taken into account the possibility that no solution in `LIVE` corresponds to the node selected.
- `TREE` **based- procedure**. In this option, a complete solution solely based in the list `TREE` should be obtained. Therefore, the node selected should be expanded into child nodes until a complete solution is obtained.
- **Heuristic based- procedure**. This procedure consists in completing the partial solution by means of some heuristic. A sensible way to do it is using ad-hoc constructive heuristics for the problem under consideration in order to obtain a complete solution.

This third option is selected for the implementation of the algorithm to the $F|prmu|C_{max}$ problem and will be discussed in section 3, as it is problem-specific.

3 Implementation: the $F|prmu|C_{max}$ problem

In order to test the proposed approach, we have implemented a version of the algorithm for the $F|prmu|C_{max}$ problem. This problem is a well-known combinatorial optimization problem, and we refer the reader to references [3] and [4] for the latest advances on this problem. In our implementation of the proposed algorithm, the following decisions have been adopted:

- The solutions for the problem are coded employing the most common codification, i.e. a solution for a n-jobs, m-machines problem is defined by a vector $S = (\sigma_1, \ldots, \sigma_n)$ where σ_i denotes the job being sequenced in order ith. An insertion-based neighborhood has been defined for CLM, consisting in removing a job from position i and inserting it into position j. A node of level kth is represented by a partial sequence where the first k jobs are already fixed. The set of components i that can be appended to a partial solution (node) is constituted by the set of non-scheduled jobs.
- As procedure `ObtainCompleteSolFromNode()`, a heuristic-based procedure has been designed. This procedure is based on the NEH heuristic [6], which is considered the best constructive heuristic for the problem under consideration. According to the NEH heuristic, jobs are arranged in descending order of the sum of their processing times. Then, each job kth is inserted on each of the $(k+1)$th possible slots of the partial sequence, and the so-obtained partial sequence yielding the lowest (partial) makespan is retained as partial sequence for the next step. In our implementation of `ObtainCompleteSolFromNode()`, already k jobs have been scheduled (in the first kth positions). Therefore, the same steps in the NEH heuristic are applied for the non-scheduled jobs, i.e. they are arranged in descending order of the sum of their processing times, and then job ith is inserted in positions from $(k+1)$ to $(i+1)$. The best partial sequence is retained for the next step.
- The selection of the next node to be processed –procedure `GetNodeFromTree()`– is done such as nodes in `TREE` are sorted according to their lower bounds regardless the length of the partial schedule (level of the node). Thus the weighting function employed is a lower bound of the solution. In this implementation, a fast, simple, lower bound for the makespan that can be computed in $O(nm)$ has been selected. This lower bound is denoted as $LB1$ in the review by [4].
- Parameter setting of the algorithm:
  - K_b has been set to 10. As one new solution is added to `LIVE` on each iteration of `ExploreTree()`, this means that 10 solutions are stored in `LIVE` for CLM.
  - K_c has been set to $K_c = K_b \cdot |\texttt{LIVE}|$. This ensures that all relevant solutions in `LIVE` are to be explored before `LIVE` is cleared.
  - As stopping criterion, it has been set to stop after 100 iterations without improvement. This number of iterations have been proved to be more than sufficient to obtain accurate solutions of the problem. In view of the results shown in table 1, we conjecture that a similar performance could have been obtained with a lower number of iterations.

Problem	Size		Makespan		PRD	
Instance	n	m	UB_T	$C_{max}(ALG)$	Indiv.	Avg.
ta001	20	5	1278	1278	0.000	
ta002	20	5	1359	1359	0.000	
ta003	20	5	1081	1081	0.000	
ta004	20	5	1293	1293	0.000	
ta005	20	5	1235	1235	0.000	
ta006	20	5	1195	1195	0.000	
ta007	20	5	1234	1239	0.405	
ta008	20	5	1206	1206	0.000	
ta009	20	5	1230	1240	0.813	
ta010	20	5	1108	1108	0.000	**0.122**
ta011	20	10	1582	1582	0.000	
ta012	20	10	1659	1659	0.000	
ta013	20	10	1496	1496	0.000	
ta014	20	10	1377	1379	0.145	
ta015	20	10	1419	1419	0.000	
ta016	20	10	1397	1397	0.000	
ta017	20	10	1484	1484	0.000	
ta018	20	10	1538	1543	0.325	
ta019	20	10	1593	1593	0.000	
ta020	20	10	1591	1591	0.000	**0.047**
ta021	20	20	2297	2298	0.044	
ta022	20	20	2099	2099	0.000	
ta023	20	20	2326	2326	0.000	
ta024	20	20	2223	2223	0.000	
ta025	20	20	2291	2296	0.218	
ta026	20	20	2226	2230	0.180	
ta027	20	20	2273	2276	0.132	
ta028	20	20	2200	2200	0.000	
ta029	20	20	2237	2237	0.000	
ta030	20	20	2178	2178	0.000	**0.057**
ta031	50	5	2724	2724	0.000	
ta032	50	5	2834	2834	0.000	
ta033	50	5	2621	2621	0.000	
ta034	50	5	2751	2751	0.000	
ta035	50	5	2863	2863	0.000	
ta036	50	5	2829	2829	0.000	
ta037	50	5	2725	2725	0.000	
ta038	50	5	2683	2683	0.000	
ta039	50	5	2552	2561	0.353	
ta040	50	5	2782	2782	0.000	**0.035**
Average					**0.065**	

Table 1: Results obtained in a subset of the testbed by [9]

– Regarding CLM parameters, K has been set to 1, and α_0 and β have set set to zero. This implies that the threshold is the makespan of the current solution, that is: the procedure includes in `LIVE` these solutions whose makespan improves the makespan obtained for the current solution (see [2] for details on the performance of the different parameters for CLM).

Note that we have not employed in our implementation neither sophisticated (and accurate) lower bounds such as the ones described in [4], nor problem-specific neighborhoods, such as those described in [3]. On one hand, such lower bounds are computationally very expensive and their inclusion might distort the balance between the branch and bound and the local search algorithm. On the other hand, using tailored neighborhoods might hide the general capabilities of the algorithm. However, we believe that the current results can be improved by the introduction of these two elements, being this an interesting area of future research (see section 4 with respect to this issue).

To test the performance of the implementation, part of the testbed of problem instances built by Taillard [9] has been selected. The best-so-far results obtained for these instances are available in the OR Library [7] and are denoted in this paper as UB_T. For each instance, $C_{max}(ALG)$ the makespan obtained by the implementation has been measured in terms of the Percentage Relative Difference(PRD), defined as follows:

$$PRD = \frac{C_{max}(ALG) - UB_T}{UB_T} \cdot 100$$

The results of the experiments are shown in table 1. The quality of the solutions is extremely good, reaching the best known solution in 31 out of the 40 instances. Regarding the computation times, they grow rather fast with the problem size.

4 Comments and final remarks

In this paper we present the outline of a hybrid metaheuristic for COPs. The algorithm combines features of exact enumeration methods (i.e. branch and bound with some features borrowed from beam search) with local search strategies. The communication between local search and enumeration procedure is ensured so both algorithms efficiently cooperate. On one hand, the local search provides the enumeration procedure with accurate lower bounds. On the

other hand, the enumeration procedure drives the local search by guiding the latter to new regions where good solutions can be found and providing a diversification mechanism for avoiding local optima. Finally, it has to be noted that the procedure yields exact –optimal– solutions if allowed sufficient computation time.

This procedure has been implemented for a well-known COP, the the $F|prmu|C_{max}$ problem. The preliminary results are encouraging, despite the implementation does not use tailored neighborhoods nor sophisticated lower bounds.

At this stage of the research, our effort concentrates in the following lines:

- Additional features from tree search methods (particularly beam-search [8]) can be incorporated, including the use of a beam width to reduce the computational burden. However, these new features would imply the loss of guarantee of optimality.
- Different local search strategies can be incorporated instead of CLM. The most obvious choice is tabu search, which is also expected to reduce the computational burden of the local search part as it does not need to handle a list of all explored solutions.
- The procedure should be applied to different COPs. We believe that this could be particularly fruitful in its application to problems with severe feasibility restrictions (i.e. problems where their natural codification contemplates unfeasible solutions).
- Regarding the specific implementation that has been presented in this paper, a lot of work has yet to be done. On one hand, it has been already mentioned that using specific (reduced) neighborhoods and accurate lower bound is expected to improve the algorithm. On the other hand, a fine tuning of the parameters and decisions involved in the algorithm should be carried out in order to increase its performance.

References

[1] French, A.P., Robinson, A.C., Wilson, J.M., (2001) Using a hybird genetic-algorithm/branch and bound approach to solve feasibility and optimization integer programming problems, *Journal of Heuristics*, 7, 551-564.

[2] Ghosh, D. and Sierksma, G. (2002) Complete Local Search with Memory, *Journal of Heuristics*, 8, 571-584.

[3] Grabowski, J., Wodecki, M. (2004) A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion, *Computers & Operations Research*, 31, 1891-1909.

[4] Ladhari, T., Haourari, M. (2004) A computational study of the permutation flow shop problem based on a tight lower bound, *Computers & Operations Research*, In press, available online.

[5] Nagar, A., Heragu S.S., Haddock, J. (1996) A combined branch and bound and genetic algorithm based for a flowshop scheduling algorithm, Annals of Operations Research, 63, 397-414.

[6] Nawaz, M., Enscore, E.E., Ham, I. (1983) A heuristic algorithm for the n-job, m-machine flow shop sequencing problem, *Omega*, 11, 91-95.

[7] OR Library: http://mscmga.ms.ic.ac.uk/info.html.

[8] Ow, P.S., Morton, T.E. (1988) Filtered beam search in scheduling, *International Journal of Production Research*, 26, 35-62.

[9] Taillard, E. (1993) Benchmark for basic scheduling problems, *European Journal of Operational Research*, 64, 278-285.