Fore-and-Back

Chapter 10
Fore-and-Back

10.1 Introduction

Fore-and-Back has been presented in the literature as *Forward&Backward* and also as *F&B*, but the algorithm presented in this chapter differs in some details from the previously published ones. It is an extension of *Beam Search* (BS) that can improve its effectiveness. Fore-and-Back, when run with no limits on computational resources, becomes an exact solution method. However, by design, it is mainly concerned with heuristic solving, trying to quickly get high quality solutions with little attention paid to optimality proofs.

A significant characteristic of this method is that, despite being a primal only method, it is able to compute bounds to the cost of completing partial solutions, therefore to discard partial solutions from expansion and ultimately to reduce the search space.

Beam search (BS) is a variant of standard tree search that limits the number of offsprings that are expanded at each iteration. BS core ideas were originally introduced in artificial intelligence contexts, and only later transposed to optimization. The first problems for which BS was used were scheduling problems, but BS has since proved successful also on many other different combinatorial optimization problems.

BS does not complete the search that would normally be carried out by branch and bound algorithms, therefore it is an approximate method and a matheuristic of its own. BS has, in fact, been proposed as an effective heuristic methodology, and as such it has been enhanced and hybridized with other heuristics, for example, ant colony optimization. Other matheuristics closely related to BS have been proposed. One is the Pilot Method, which consists of a partial enumeration strategy, where the possible expansions of each partial solution are evaluated by means of a pilot heuristic. Another one is the Filter&Fan method, which starts with a feasible solution and builds a search tree, where branches correspond to submoves in the neighborhood of the solution and where each node corresponds to a solution obtained as a result of the sequence of submoves associated with the root-node path. In this algorithm, the

initial candidate list of moves is filtered at each tree level by evaluating each move in the list with respect to all the solutions at that level. The best moves at each level are included in the candidate list of the next level and the corresponding solutions are the nodes of the successive level.

The characterizing idea of BS is to allow the extension of partial solutions into a limited number of offsprings. This is similar to what we have already seen for diving (Chapter 5), for VLSNS (Chapter 6) and for the Corridor Method (Chapter 8), but here the focus is on the result, the number of offspring, and not on the method to limit their number. At each BS iteration, the algorithm extends a partial solution from a set **T**, the *beam*, generating a possibly limited number of offspring. Each offspring is either a complete solution, or it is inserted into the set **T** itself, in case it is a partial solution worth of further analysis.

At the end of the expansions, BS selects from **T** up to δ (a parameter called the *beam width*) solutions. The selection is based on some criterion for ranking the expected usefulness of an expansion, for example, on the basis of bounds to the cost of the completions.

More in detail, BS proceeds as follows. At the first step, the beam **T** is initialized with an empty solution. Then, the algorithm iterates a basic procedure in which a set of promising nodes at a given level of the search tree are expanded to generate their δ offspring, which all become members of the set of the unexpanded nodes of a subsequent tree level. A node can be considered to be promising in accordance with its completion bound cost. When a level has been expanded, two strategies are possible: either expand the subsequent level or expand the nodes of lowest cost completion bound. Intermediate strategies are possible, where expansions proceed depth-first in order to quickly complete good solutions but, when the last level is expanded, it is possible to backtrack to the nodes with the lowest completion bound, which can be high in the tree hierarchy. This process is iterated until a termination condition is met (heuristic) or until all unexpanded nodes have a completion bound cost that is not smaller than the current upper bound, which is, therefore, the optimal cost (exact).

10.2 Fore-and-Back

Algorithm Fore-and-Back builds on this general BS approach, alternating searches following opposite expansion directions, and storing in memory previous partial results that can be used as a lookahead to complete partial solutions when search is performed in the opposite direction. The algorithm works therefore best when the problem suggests a natural direction of partial solution expansions, which can also be reversed.

Actually, there is a vast class of combinatorial optimization problems that fit this schema. These are problems that exhibit a regular substructure that can be decomposed into n subproblems that are linked together by a set of coupling constraints. These problems can often be modeled by defining, for each k-th subproblem, a set S_k

containing all the feasible solutions for the k-th subproblem. The resulting problem consists of choosing, from each set S_k, a single component in such a way that the set of selected components satisfies all constraints. This is, for example, the case for the GAP, where subproblems could refer to the assignments of single clients and the capacity constraints act as linking constraints, or vice-versa (subproblems defined on capacities and linking constraints on assignments).

Fore-and-Back exploits this structure, molding around it an iterative heuristic algorithm, that adopts a memory-based look-ahead strategy exploiting the knowledge gained in its past search history. Algorithm Fore-and-Back iterates a partial exploration of the solution space by generating a sequence of beam search-like search trees of two types, called *forward* and *backward* trees. Each node at level h of the trees represents a partial solution containing h components. At each iteration t, the algorithm generates a forward tree $\mathbf{F}^t$ if t is odd, or a backward tree $\mathbf{B}^t$ if t is even. In generating a tree, each partial solution X is extended to a feasible solution using the partial solutions generated at the previous iteration in the complementary tree, and the cost of the resulting solution is used to bound the quality of the best complete solution that can be obtained from X.

10.2.1 Search trees

During search, Fore-and-Back alternatively builds *forward trees* and *backward trees*.

A forward tree is an n-level tree, if the number of subproblems is n, where each level $h = 1, ..., n$ is associated with a component set S_h and each node at level h corresponds to a partial solution containing one component from each set $S_1, S_2, \ldots, S_h$. Components are considered to be solution elements, for example, one component could be the assignment of a client to a server in the case of the GAP, or an arc to be included in the route in the case of the TSP.

Conversely, in a backward tree, each level h is associated with a set S_{n-h+1}, in case these labels were numbered according to the forward exploration, so that a node at level h, represents a partial solution containing one component of each set $S_n, S_{n-1}, \ldots, S_{n-h+1}$.

A list, denoted by L_h^t, is associated with each level h of the tree built at iteration t. The list contains δ nodes generated but not expanded at level h, where δ is an input control parameter equivalent to the likewise denoted parameter of Beam Search. All nodes so far expanded at level h in all trees at odd iterations are kept in set E^h for forward trees, while those expanded at level h in all trees at even iterations are kept in $\overline{E}^h$ for backward trees. The nodes in the lists L_h^t, $h = 1, \ldots, n$, represent the memory of iteration t, that will be used to guide the exploration of the current tree and of the tree that will be explored in the following iteration $t+1$.

The core idea of Fore-and-Back is to evaluate the completion cost of partial solutions stored at level h of the tree by means of the partial solutions stored in L_{n-h}^{t-1}

in case of forward trees, or, analogously, the completion of partial solutions in L_{n-h}^{t} by means of solutions in L_{h}^{t-1} in case of backward trees.

As an example, suppose we are building the forward tree associated with an odd iteration t. Consider two partial solutions, $X \in L_h^t$ and $\overline{X} \in L_{h+1}^{t-1}$. Since t is odd, X contains one component of each set $S_1, S_2, ..., S_h$, while $\overline{X}$ contains one component of each set $S_n, S_{n-1}, ..., S_{h+1}$. These two solutions can be combined to obtain a (not necessarily feasible) complete solution $X \cup \overline{X}$ of cost $c\left(X \cup \overline{X}\right)$. Clearly, if the resulting solution $X \cup \overline{X}$ satisfies all constraints, the associated cost represents a valid upper bound to the optimal problem solution cost (assuming, here and in the rest of the chapter, to deal with a minimization problem).

In general, at each iteration t, algorithm Fore-and-Back builds the associated tree and computes the cost $c(X)$ of each node X. The cost of an inner node is derived from the cost of completing the partial solution X and from the penalty assigned to the level of infeasibility that the completion shows. Algebraically:

$$c(X) = \min_{\overline{X} \in L_{h+1}^{t-1}} \left\{ c\left(X \cup \overline{X}\right) + c_{inf}\left(X \cup \overline{X}\right) \right\}, \tag{10.1}$$

where $c_{inf}(X \cup \overline{X})$ is an arbitrary function whose value is related to the degree of infeasibility of $X \cup \overline{X}$ and that is equal to 0 if $X \cup \overline{X}$ is a feasible solution. It is important to effectively balance the push toward feasibility against the quest for good solutions, a balance that is problem-dependent when not even instance-dependent.

During the first iteration, there is no backward tree to match partial solutions against. The lists L_h^0 are empty at each level, therefore expression (10.1) gives the cost of the partial solution X.

10.2.2 Fore-and-Back pseudocode

To simplify the code and avoid all duplications needed to account for forward or backward directions, we will denote by **T** alternatively the forward tree **F** or the backward tree **B**, depending on the parity of the iteration counter. Consequently, T_h denotes the level h of the tree **T**, be it forward or backward.

The algorithm is controlled by three user-defined parameters: δ, the number of nodes expanded at each level of both forward and backward trees; *maxn*, the maximum total number of nodes to expand; *maxtnodes*, the limit of the number of nodes of each tree.

In order to expand level h of a tree at iteration t, the algorithm computes the value $c(X)$ for each node $X \in T_h$ and builds the set $L_h^t \subseteq T_h$. The list is defined ordering T_h by increasing cost values and keeping in L_h^t only its δ nodes having the smallest cost, which will be further expanded.

A distinguishing feature of Fore-and-Back is that at each level h of the tree built at iteration t, we store also the cost $\hat{c}_h^t$, which is the least cost of nodes in T_h but not in L_h^t. This is the least cost we are expected to pay in case we want to com-

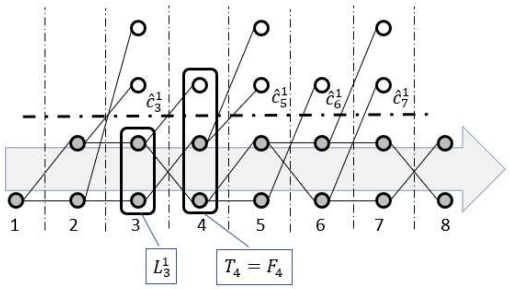

Fig. 10.1 Forward tree, initial (t=1)

plete a complementary partial solution without using the partial solutions that led to defining L_h^t.

In case a feasible solution is found, we possibly update a variable z_{best}, which keeps the cost of the best solution achieved so far: it is initialized to ∞ at the beginning of the algorithm and takes on progressively lower values during search.

In forward trees, each node X included in L_h^t is expanded to create a new node $X \cup \{s\}$ for each component s of the set S_{h+1} of the subproblem associated with level $h+1$, provided that the following conditions hold:

1) $X \cup \{s\}$ does not violate any constraint.
2) $c(X \cup \{s\}) + \hat{c}_{h+2}^{t-1} < z_{best}$, in case $X \cup \{s\}$ cannot be feasibly completed with a partial solution in L_{h+2}^{t-1} or in $\overline{E}^{h+2}$ (pruning).

These conditions hold, with opportune indices update, also for backward trees. In the pseudocode, e make use of two dummy levels, L_0^t for forward trees and L_{n+1}^t for backward trees, in order to initialize their computation.

Node expansions continue in this fashion until the last level is reached ($h = n$) or until a maximum number of nodes have been expended in the current tree. This last condition is typically met in problems, like the GAP, where partial solutions cannot be feasibly expanded in any way, therefore backtracks are in order. Backtracking can be made either in a depth-first way, expanding the last generated unexpanded nodes, or jumping to the stored node of least expected cost $c(X)$.

Algorithm Fore-and-Back terminates after the expansion of *maxn* nodes, or after two consecutive iterations where the value of z_{best} does not improve. This last is a condition that no further improvements would be possible if infeasibilities are properly accounted for.

Figures 10.1 and 10.2 show an example of possible forward and backward trees for the first iteration of the algorithm, expanding $\delta = 2$ nodes per level.

Algorithm 41: Algorithm Fore-and-Back

```
1  function Fore-and-Back(δ, maxn, maxtnodes);
   Input  : δ, beam width, maxn, max total num of nodes, maxtnodes, max num of nodes
            per tree
   Output: A feasible solution x* of value z_best
2  initialize t = 0, noimpr = 0, nNodes = 0, z_best = ∞;
3  while nNodes ≤ maxn do        // Alternate forward and backward trees
4      t = t + 1; noimpr = noimpr + 1;
5      let L_0^t = L_{n+1}^t = {∅}, ntNodes=0;
6      foreach level h = 1,...,n do
7          set T_h = {∅};                              // generate the node set T_h
8          if t is odd then
9              set k = h; k1 = h − 1;
10         else
11             set k = n − h + 1; k1 = n − h + 2;
12         end
13         foreach node X ∈ L_{k−1}^t do
14             foreach component s ∈ S_k do
15                 let X' = X ∪ {s};
16                 if X' meets conditions i) and ii) then
17                     set T_k = T_k ∪ X';
18                     ntNodes = ntNodes + 1;
19                     nNodes = nNodes + 1;
20                     if h = n and c(X') < z_best then        // feasible solution
21                         set z_best = c(X'), noimpr = 0 and x* = X';
22                     end
23                 end
24             end
25             set E^k = E^k ∪ X;                          // node expanded
26         end
                                                  // extract the subset L_k^t ∈ T_k
27         foreach node X ∈ T_k do
28             if |T_k| ≤ δ then
29                 set L_k^t = T_k;
30             else
31                 let L_k^t contain only the δ least cost partial solutions of T_k;
32             end
33         end
34         if ntNodes > maxtnodes then break;
35     end
36     if noimpr = 2 then // best sol. not improved for two iterations
37         stop;
38     end
39 end
```

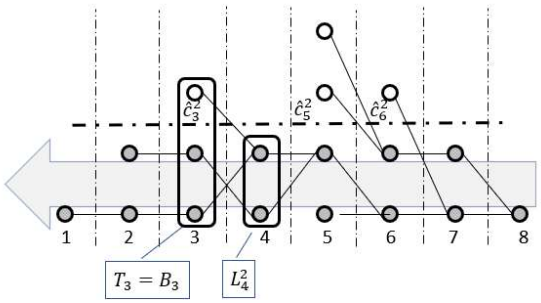

Fig. 10.2 Backward tree (t=2)

10.3 Fore-and-Back for the GAP

Algorithm 41 can be directly applied to the GAP, as this problem enjoys the structural property required in Section 10.2. In the following run, in fact, we assumed the GAP to be composed of subproblems defined by client assignments, one subproblem for each client. The linking constraints are given by the capacity constraints, much alike the decomposition used in the example of Section 7.4. Algorithm 41 is applied to instance *example8x3* of Section 1.1 and run with parameters $\delta = 2$, $maxn = 5000$ and $maxtnodes = maxn/10$.

Initially, step 2, variables and structures are initialized, and a standard Beam Search is run. We saw in Chapter 1 that GAP is strongly NP-hard and that the number of infeasible solutions that can be expressed by the decision variables of formulation GAP (see section 1.1) exceeds by far the number of feasible solutions (Table 1.1). This implies that unguided search is likely to produce partial solutions that cannot be feasibly expanded. Figure 10.3 shows the complete forward tree that was explored during the first forward run. The figure shows all generated nodes and the resulting tree topology, except that nodes corresponding to complete solutions are not stored in the code, being immediately pruned if dominated, and are consequently not shown in the figure. The root node is the dummy empty node.

The condition that actively terminated the first search was on the number of tree nodes, *maxtnodes* and search went through 41 backtrackings. No feasible solution could be found. The bounds for unmatched completions were $\hat{c}_1^1 = 88$, $\hat{c}_2^1 = 172$, $\hat{c}_3^1 = 252$, $\hat{c}_4^1 = 282$, $\hat{c}_5^1 = 221$, $\hat{c}_6^1 = 229$, $\hat{c}_7^1 = 257$.[1]

The stored partial solutions, along with the bounds, proved extremely useful to guide search for the backward tree. Figure 10.4 shows the first backward tree. This time the active terminating condition was the completion of the loop at step 6, as a complete feasible solution could be constructed (of the two longer paths of figure 10.4 one could not be expanded including the last client). However, feasible solu-

[1] We remind that in the computational traces the decision variables are indexed from 0

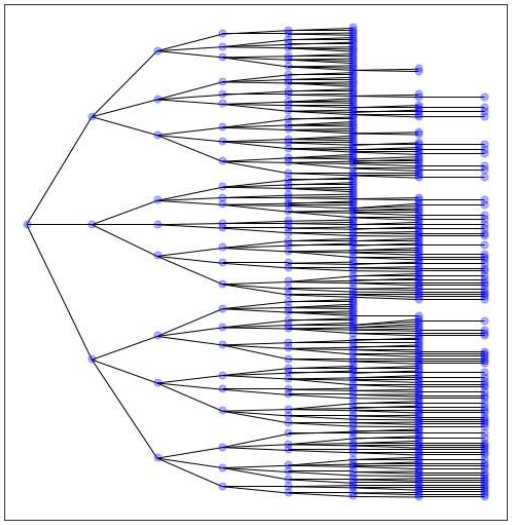

Fig. 10.3 Initial forward tree for instance *example8x3*

tions could be found before that, upon matching partial backward solutions with partial solutions stored by the forward search.

The first feasible matching, thus the first feasible solution, was found at level 4 of the backward tree, where the partial solution $\overline{X} = (\_,\_,\_,\_,2,2,2,1)$ of cost 276 could be matched with the forward partial solution $X = (0,0,0,1,\_,\_,\_,\_)$ of cost 61, thus producing a feasible solution of cost 337.

This upper bound was repeatedly improved during the backward search. After obtaining higher cost matchings, at level 4 the forward partial solution $X = (0,0,0,2,\_,\_,\_,\_)$ of cost 105 could eventually be matched against the backward partial solution $\overline{X} = (\_,\_,\_,\_,1,2,2,1)$ of cost 230, thus producing a feasible solution of cost 335.

Then, an improved matching of cost 334 was found, then one of cost 330, until the forward partial solution $X = (1,0,0,2,\_,\_,\_,\_)$ of cost 117 could be matched with the backward partial solution $\overline{X} = (\_,\_,\_,\_,1,2,2,0)$ of cost 211, thus producing a feasible solution of cost 328. At the end of the backward run, the algorithm expanded 29 nodes, which permitted to identify 36 feasible solutions (one by constructions, the other ones by matching unexpanded offspring with stored partial solutions).

The best found solution of cost 328 is still not optimal. An optimal solution could be found in the subsequent forward run, with $t = 2$, where first an improving solution of cost 327 was found, then one of cost 326, and finally, after backtracking at level 1, the forward partial solution $X = (1,\_,\_,\_,\_,\_,\_,\_)$ of cost 22 could be matched with the backward partial solution $\overline{X} = (\_,1,0,2,0,2,2,0)$ of cost 303, thus producing a feasible solution of cost 325, which is optimal.

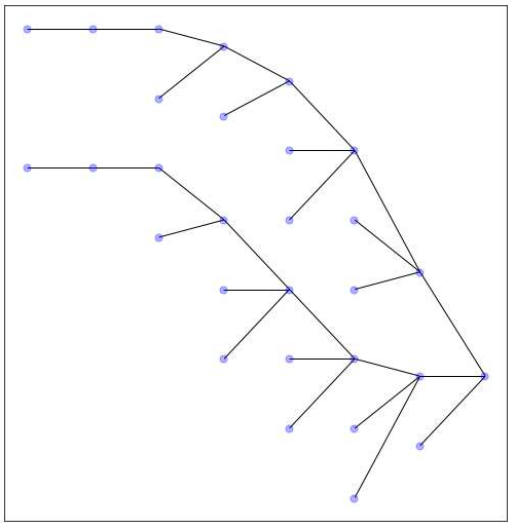

Fig. 10.4 First backward tree for instance *example8x3*

Having no awareness of the reached optimality, search goes on until a termination condition is met. In this case, the condition refers to the number of iterations without best solution improvements, and lets search terminate after 3 main loop iterations.

Figures 10.5 and 10.6 report about the number of nodes explored during search. The left figure shows the number of node expansions at each iteration, along with the number of nodes that were actually stored into memory. It is possible that the number of stored nodes is higher than the number of expansions, as at iteration 1, because when expanding a node more than an offspring is generated. It is also possible that the number of expansions is higher than the number of stored nodes, as at iteration 2, because many infeasible offspring get generated, and these are not stored into memory.

Data is presented separately for forward and for backward trees. It is apparent how the first forward tree, that cannot use completion bounds for pruning, generates a number of nodes much higher than the following ones, that can make use of search memory.

Figure 10.6 presents aggregate data on the total number of stored nodes and on the total number of open nodes per iteration. Again, one can see how most nodes are generated during the first two iterations, mainly during the first, and that the termination condition stops the search when there would still be open nodes to expand, which could however not lead to improving solutions (though the algorithm is unaware of this).

Coming to feasible solutions obtained by matching partial solutions, Figure 10.7 shows, at each level of the search tree, the number of feasible solutions that could be obtained. The distribution is heavily skewed to the right, as a result of the much

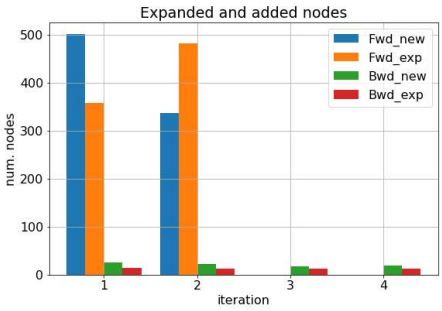

Fig. 10.5 Number of expanded nodes

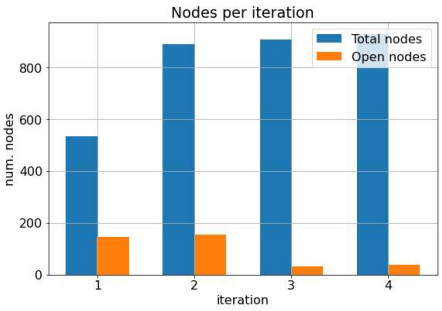

Fig. 10.6 Number of nodes per iteration

higher number of nodes produced by forward search, which permitted the early pruning of backward solutions, when they were generated. No matchings were achieved at level 7.

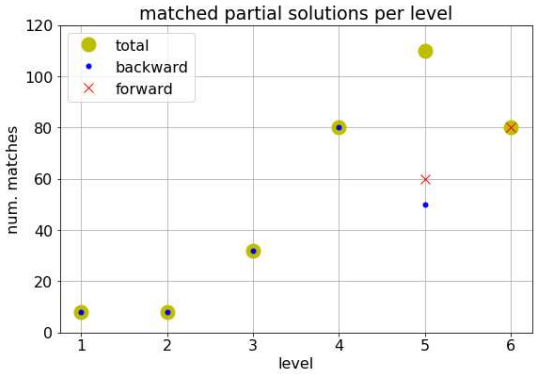

Fig. 10.7 Number of matched solutions

Finally, Figure 10.8 shows, separately for forward and for backward trees, the profile of the completion bounds at each level h of the trees. These are the $\hat{c}_h^2$ and $\hat{c}_h^3$ bounds that could be read at the end of the run, $h = 1, \ldots, n$.

The blue forward bound profile increases monotonically up to level $n-1$ (the last level has no bound as it corresponds to complete solutions), thanks to the sufficient number of nodes that were generated.

The orange backward bound is counter-intuitively non monotonic and very low for the middle levels. This comes from the fact that comparatively few nodes were generated for backward trees, therefore data refers only to the best solutions, which were produced in a depth-first fashion. For example, nodes at level 4 derived from the expansion of the best nodes at level 5, and the node that caused a comparatively

high bound at level 5 was not yet expanded, therefore did not cause a monotonic increase of the backward completion bound.

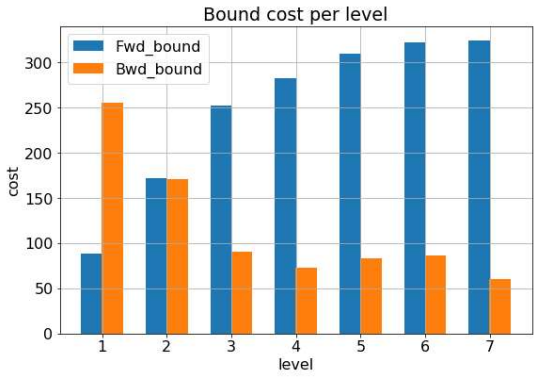

Fig. 10.8 Completion bound cost per level

10.4 Related literature

Fore-and-Back was initially presented in Bartolini et al (2008) and Bartolini and Mingozzi (2009).

AI introduction on beam search context can be found in Lowerre (1976); Reddy (1977). Scheduling applications of beam search are described in Ow and Morton (1988) and Pinedo (1995). Extensions and hybrids can be found for example in Della Croce et al (2004), Blum (2005), Blum (2008), and Maniezzo (1999).

The pilot method was proposed in Duin and Voß (1999), the Filter&Fan in Glover (1998), and Greistorfer and Rego (2006).

References

Bartolini E, Mingozzi A (2009) Algorithms for the non-bifurcated network design problem. Joutnal of Heuristics 15(3):259–281

Bartolini E, Maniezzo V, Mingozzi A (2008) An adaptive memory-based approach based on partial enumeration. In: Maniezzo V, Battiti R, Watson JP (eds) LION 2, LNCS 5313, Springer, pp 12–24

Blum C (2005) Beam-ACO - Hybridizing ant colony optimization with beam search: an application to open shop scheduling. Computers and Operations Research 32(6):1565–1591

Blum C (2008) Beam-ACO for simple assembly line balancing. INFORMS Journal on Computing 20(4):618–627

Della Croce F, Ghirardi M, Tadei R (2004) Recovering beam search: Enhancing the beam search approach for combinatorial optimization problems. Journal of Heuristics 10(1):89–104

Duin C, Voß S (1999) The pilot method: A strategy for heurisic repetition with application problem in graphs. Networks 34:181–191

Glover F (1998) A template for scatter search and path relinking. In: Ronald E, Schoenauer M, Snyers D, Hao JK, Lutton E (eds) Artificial Evolution, Lecture Notes in Computer Science, Vol.1363, pp 3–51

Greistorfer P, Rego C (2006) A simple filter-and-fan approach to the facility location problem. Computers & Operations Research 33:2590–2601

Lowerre B (1976) The HARPY speech recognition system. PhD thesis, Carnegie Mellon University, Pittsburgh, PA

Maniezzo V (1999) Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. INFORMS Journal on Computing 11(4):358–69

Ow P, Morton T (1988) Filtered beam search in scheduling. International Journal of Production Research 26:297–307

Pinedo M (1995) Scheduling: Theory algorithms, and systems. Prentice-Hall

Reddy D (1977) Speech understanding systems: A summary of results of the five-year research effort. Tech. rep., Department of Computer Science, Carnegie-Mellon University