Tabu Search Candidate List Strategies in Scheduling[‡]

BALASUBRAMANIAN RANGASWAMY

Graduate School of Business and Administration

University of Colorado at Boulder

Campus Box 419, Boulder, CO 80309

rangaswa@eddie.Colorado.EDU

ANANT SINGH JAIN[†]

Department of Applied Physics, Electrical & Mechanical Engineering

University of Dundee

Scotland, UK, DD1 4HN

A.Z.Jain@dundee.ac.uk

FRED GLOVER

US West Chair in Systems Science

University of Colorado at Boulder

Campus Box 419, Boulder, CO 80309

Fred.Glover@Colorado.EDU

Abstract – Candidate list strategies form an important cornerstone of Tabu Search (TS), yet are often neglected in TS research. In this paper, we review candidate list construction principles and illustrate basic concepts through simple numerical examples from the resource-constrained scheduling domain. We also provide computational results which document that significant gains are made possible by intelligent implementations of candidate list strategies, even where other Tabu Search components are restricted to relatively simple levels.

key words: candidate list strategies, meta-heuristics, tabu search, resource-constrained scheduling

[‡] Accepted for publication at the 6th *INFORMS Advances in Computational and Stochastic Optimization, Logic Programming and Heuristic Search: Interfaces in Computer Science and Operations Research Conference* in Monterrey Bay, California, January 1998. Proceedings edited by David L. Woodruff, Kluwer Academic Publishers.

[†] This work has been sponsored by the Royal Society of Edinburgh and the Carnegie Trust for the Universities of Scotland while the author was a visiting researcher at the University of Colorado, Boulder.

1. Introduction

Tabu Search (TS) is increasingly being used as an effective method to get good solutions to difficult optimization problems (see e.g., the survey of applications in Glover and Laguna, 1997). The approach relies on adaptive neighborhood modification using memory-based strategies, whose components include multilevel candidate lists, short term and long term memory structures, and blending of neighborhoods to guide the search. The notion of adaptive neighborhood modification by means of memory-based designs has led to the development of many interesting approaches over the past few years. However, there are some fundamental tabu search strategies that are often not applied effectively. A plausible explanation for this is that the rationale underlying these strategies is not completely understood, as described in the following (Glover, 1996; Glover and Laguna, 1997 and the collection of papers in Glover, 1997). The current paper amplifies this theme with special focus on the domain of candidate list strategies. Although candidate list strategies have been successfully implemented in some applications (Woodruff and Spearman (1992), Lokketangen and Glover, (1997), Minghe Sun *et al.* (1997), James and Buchanan (1997)), there is still a broad scope for better implementations especially in the area of scheduling. We discuss some candidate list strategies in general, and comment on their applicability to the scheduling domain. In order to provide a unified context for numerical illustrations, all examples used in this paper come from the resource-constrained scheduling area.

The remainder of the paper is organized as follows. In Section 2, we review candidate list strategies that have notable potential for more efficient and effective implementations, but that have received inadequate attention in the literature. In Section 3, we describe the problem context used for the numerical illustrations. Section 4 provides concrete examples to describe the process of creating and implementing various types of intelligent candidate lists, together with an elaboration of their underlying rationale. Section 5 provides preliminary computational results that disclose the sometimes surprising gains that can be obtained by using intelligent candidate lists. Finally, Section 6 offers useful pointers for further work.

2. 0 Candidate List Strategies in Tabu Search

The first motivation for building candidate lists is the observation that both efficiency and efficacy of the search can be greatly influenced by isolating good candidate moves – in contrast, for example, to evaluating all possible moves in a current neighborhood. (The accent here is on generating the candidate moves by some *intelligent* process rather than by a random or *naïve* process.) The second motivation comes from the need to reduce the time required to evaluate moves, especially where each move may be expensive to generate or analyze, or where the current neighborhood may contain a large number of moves. A third motivation comes from the goal of exploiting problem structure, where particular problem domains give a basis for special constructions to create intelligent candidate lists (giving rise to *context related rules* as described in Glover and Laguna, 1997).

This paper stresses the importance of accounting for multiple factors whose appropriate balance can vary at different points in the search. We find it useful to differentiate among candidate list strategies of five fundamental types — the successive filtration strategy, the elite candidate list strategy, the aspiration plus strategy, the sequential fan strategy and the bounded change strategy. These strategies are first reviewed here in a general sense, following the lines of Glover and Laguna, 1997.

2. 1 Successive Filtration Strategy

This candidate list strategy has a particularly strong influence on search quality in the context of scheduling. In many combinatorial optimization problems, the outcome of a move can be thought of as the combined effect of several fundamental processes. By separating these processes and restricting attention to one process at a time, effective candidate lists can be constructed. For example, in many graph problems, a commonly used move is to replace an edge in the current solution with another edge not in the solution. Such moves can therefore naturally be broken down into two components — an “add edge” component that introduces an edge and a “drop edge” component that deletes an edge. We can isolate the top few outcomes for the “add edge” component to create a set of “best add edge” selections and similarly, isolate the top “drop edge” component to create a set of “best drop edge” selections. If there were 100 possible individual “add edge” and “drop edge” components, for example, a complete examination strategy would require consideration of 10,000 moves. Instead, by considering the 10 to 20 best individual components each from the “best add edge” and “best drop edge” components, only 100 to 400 complete moves need to be evaluated – a reduction of nearly two orders of magnitude in effort, even after considering the work of identifying the component move. The basic premise here is that although the evaluation of the separate components is only an approximate indicator of the evaluation of the move that results by their combination, this approximation is good enough for most applications.

Sometimes the evaluation of the component processes cannot be treated independently i.e., the evaluation of one component is strongly conditional upon the prior choice of another. For example, feasibility requirements may require the coupling of two different variables selected by successive filters. (Such situations occur frequently in scheduling applications where two tasks linked by a precedence relation cannot be executed during the same time window.) A simple way of implementing this is to perform sequential evaluations, ensuring feasibility at each step. Section 4.0 illustrates how this is done in the context of project scheduling.

2. 2 Elite Candidate List Strategy

The elite candidate list strategy records the best (elite) solutions or their attributes and uses the recorded information in the subsequent iterations. Fundamental to the use of this strategy is a Master List that is built by recording the best *k* moves encountered in the examination of alternative moves on a given iteration. The Master List is periodically constructed, and is based on examining a relatively large number of moves. Then at each subsequent iteration until a new list is created, the best move currently available from the Master List is chosen and executed. The process continues to select moves from the master list until either a prescribed number of moves have been made, or the quality of the best available move falls below a threshold. At that point the Master List is reconstituted and the process repeats.

The assumption here is that a collection of best moves is likely to contain a subcollection that will continue to be good for a number of iterations, although we cannot predict in advance precisely what this subcollection will be. Proper monitoring of the evaluation and identity of the moves in the Master List is essential since the execution of each current move can change not only the evaluation but in some cases the character of remaining moves.

2. 3 Aspiration Plus Strategy

In the Aspiration Plus strategy, thresholds for the quality of a move can be established dynamically, based on the search history. The examination of current moves continues until encountering the first move that meets the threshold quality. After that, the examination is continued for a further *Plus* iterations (where *Plus* is a parameter of the process) and the best move found overall is selected. If *Plus* is defined to be zero, the procedure reduces to an approach that always selects the first move that meets the threshold quality. To maintain the flexibility of the process to adapt dynamically to search requirements, the total number of moves examined is allowed to change between two limiting values, *Min* and *Max*. (This ensures that at least *Min* moves and at most Max moves are considered.) If *Max* moves are examined without finding a move that attains at least the threshold quality, then the best move found so far is selected. The strategy is explained for a minimization problem in *Figure 1* and is adapted from Glover and Laguna (1997). In the figure, *Plus* is defined to be 5, *Min* is 7 and *Max* 15. The first move that meets the Aspiration level is move number 6, and this is called *First.* As *Plus* is 5, the value of *First* + *Plus* is 11, and this value falls within the limits imposed by *Min* and *Max*. So a maximum of 11 moves will be examined in this case, and the best move overall is selected. If *First* +

Plus is less than *Min*, then at least *Min* moves are examined; else if *First* + *Plus* is greater than *Max*, a maximum of *Max* moves is examined.

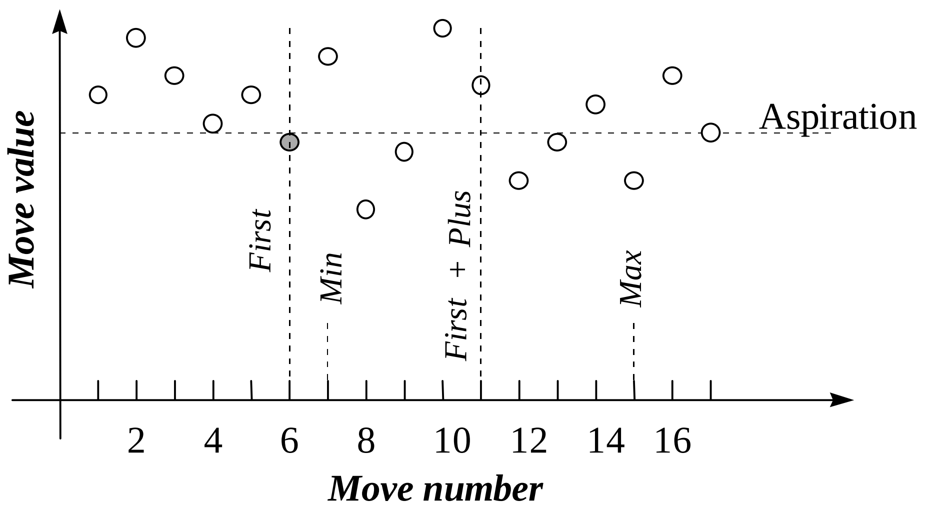

Figure 1. Illustrating Aspiration Plus Strategy

2. 4 Sequential Fan Candidate List Strategy

This strategy accommodates itself very well to parallel processing, and has an interesting connection with the *beam search* approach used in tree search methods (see Morton and Pentico, 1993). At an initial step of this approach, multiple parallel solution streams are generated from the *p* best alternative moves (thereby identifying *p* associated solutions). The *q* best available moves for each solution stream are then examined, where *q* is generally somewhat smaller than *p*, and the *p* best solutions from these *pq* alternatives form the seeds for the next step in generating the streams.

As the depth of the streams increases, *q* typically diminishes. In some variants, *q* quickly drops to 1, so that each solution stream simply selects its best move to perpetuate a single continuation. The streams continue to some cutoff depth (which can be adaptively determined, and need not be the same for each stream), while keeping track of the best solution found on each stream, subject to lying at least a minimum depth from the initiating solution. (It is possible, and often likely, that fewer than *p* distinct solutions will be recorded, because two or more streams may share a common “ancestor solution” that is best for each of them.)

The *r* best of these solutions (for $r < p$) are then selected, whereupon the *r* partial streams that terminate at these *r* solutions become the starting points for repeating the process. Then, for a new value of *q* (restored to be larger than at the points where these *r* solutions were found), *rq* options are now examined, from which *p* are selected. This re-establishes the standard examination of *pq* alternatives at each subsequent step. (Fewer than *pq* alternatives of course result where some streams are terminated at smaller depths than others. A more conservative strategy is also possible that does not resume from the *r* best solutions identified, but from somewhat earlier solutions on the streams that led to these *r* solutions.) Since some of the alternatives examined upon restarting the streams coincide with alternatives examined on the previous pass, appropriate bookkeeping can be useful to accelerate the process.

2. 5 Bounded Change Candidate List

The use of this strategy is indicated in those situations where the domain of choices for each solution component is restricted at each iteration. The degrees of permitted change are defined by a distance measure that depends on the problem context. By varying the extent of change permitted across different dimensions, controlled amounts of intensification can be obtained. In the context of generalized resource-constrained project scheduling, such a strategy has been implemented using the notion of *shift vectors* by Sampson and Weiss (1993). This idea can be exploited in other problem contexts as well.

It is not essential, or even usually a good idea, to use all the foregoing candidate list strategies in any one implementation of tabu search. Rather, the choice of the strategies to use should be dictated by the problem characteristics and the purposes to be achieved during particular phases of search. We now provide illustrations of how such strategies can be applied in the context of scheduling.

3. Resource-Constrained Scheduling

Resource-Constrained Scheduling is a generic term applied to a range of problems that includes the resource-constrained project scheduling problem (RCPSP), the job-shop scheduling problem (JSP), and the Multi-Processor Task Scheduling Problem (MPTSP). The general framework for these problems is as follows: A set of activities (also called operations or tasks) $N = \{1,\ldots,n\}$ is accompanied by precedence relations of the finish-start type with zero time lags. These precedence relations are defined by a set H of ordered pairs, where $(i, j) \in H$ indicates that activity i is an immediate predecessor of activity j. (This implies that activity j cannot be processed until activity i is processed completely.) Without loss of generality, we specify that activity 1 is a unique dummy start node and n is a unique dummy finish node. (By definition, activities 1 and n have zero duration and consume no resources. Further, we assume that activity 1 is available for processing at time $t = 0$.)

Associated with each activity $j \in N$ is a release date r_j and a due date d_j. An activity is not available for processing before its release date even though the precedence and resource constraints may permit this. Each activity demands resources at a constant rate equal to K_r during each period it is in progress drawing from $r \in R$ resource types. The cap on the maximum availability (again assumed constant throughout the time horizon) of each resource type $r \in R$ is B_r. We further assume that each activity can be done in only a single mode (i.e., the duration of the activity cannot be crashed by pumping in more resources) and that each activity once started is put in progress continuously without interruption until it is completed (called the "no-preemption" condition).

We consider two specific problems from this class of problems to provide the context for our numerical examples - the RCPSP and the JSP. In the classical version of the RCPSP (see Demeulemeester *et al.*, 1992) , there are no release and due date constraints and the objective is to minimize the project makespan (which equals the completion time of the unique dummy finish activity). In the JSP, the objective again is to minimize the completion time of the last operation, but in this case only the precedence and resource constraints apply (i.e., there are no release and due date constraints). A particular difference between the two problems lies in the patterns of resource availability and use. In the JSP each resource type is called a machine, and only one machine of each type is available. Moreover, each operation requires the use of only one machine, in contrast to an activity in RCPSP that may require multiple units of many resource types for its processing. (In RCPSP, the basic work unit is called an activity, and in the JSP it is called an operation.) It is easy to see that the RCPSP is a generalization of the job-shop scheduling problem, and that any method for the RCPSP can be applied to the JSP.

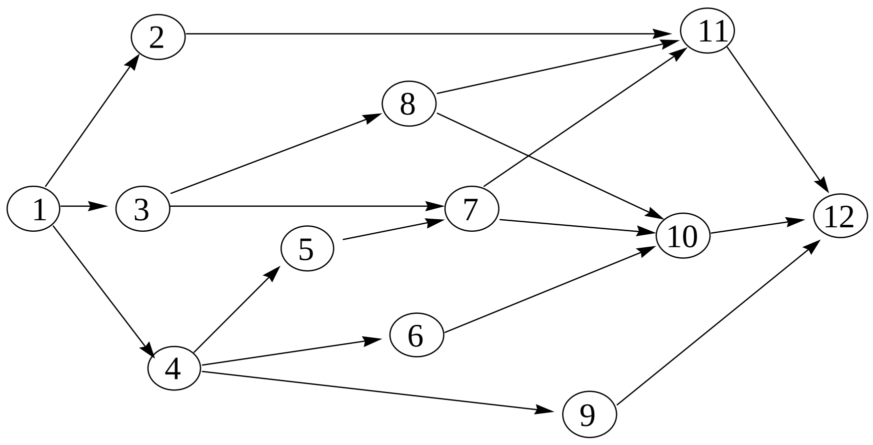

Figure 2. RCPSP Project Network for Example 1.

<u>*Example 1*</u>: Consider the RCPSP project network shown in *Figure 2*, which consists of 12 activities, with activities *1* and *12* representing the unique dummy start and finish activities. The accompanying data are provided in *Table 1*. There are four resource types and the resource demands of each activity

for each type of resource are shown in Columns 3-6 of *Table 1.* The maximum resource availability for each resource type is given by the vector 〈 13, 15, 16, 12 〉 which specifies the limits on the maximum simultaneous use of the resources. The classical RCPSP objective is to find a schedule that minimizes the completion time of activity *12,* subject to the precedence constraints (as shown by the arcs in *Figure 2*) and resource requirements.

Activity	Duration	Res Req (1)	Res Req (2)	Res Req (3)	Res Req (4)
1	0	0	0	0	0
2	4	3	9	8	0
3	7	0	0	2	0
4	2	0	0	4	0
5	1	6	0	0	0
6	10	3	0	10	0
7	1	0	1	0	7
8	6	7	0	3	0
9	9	7	10	2	8
10	1	2	0	1	8
11	1	0	3	0	0
12	0	0	0	0	0

Table 1. Problem Data for Example 1.

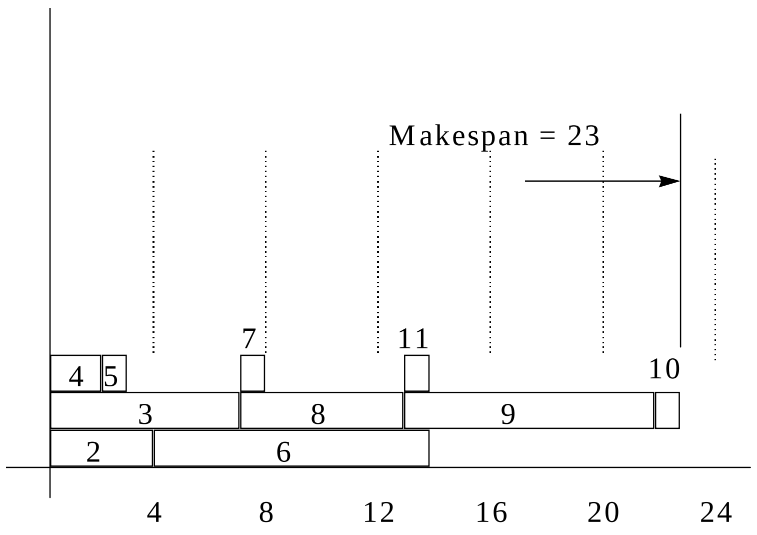

Figure 3. Gantt Chart Solution for Example 1

To find a feasible starting solution to initiate the search, we use a precedence feasible starting sequence and a simple list-scheduling algorithm. Because of the way the activities are indexed, a precedence feasible starting sequence can be obtained by using a lexicographic ordering of the activities. We use the sequence 〈 *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12*〉 as the starting solution. To find a unique schedule that corresponds to this sequence, we schedule activities as early as the precedence and resource constraints will allow. (Note, each activity is put in process until it is completed without any interruption.) Thus activity 5 starts at time $t = 2$, since the constraints do not permit this activity to be put in process earlier (as a result of the precedence relation $4 \rightarrow 5$). Activity *9* cannot start before t = 13 (even though the precedence constraints allow it to start as early as t = 2) because of the resource constraints. A Gantt chart showing the full schedule is shown in *Figure 3*.

4. Illustrating Candidate List Construction Procedures

We first assume that each solution to the RCPSP is represented as a precedence feasible permutation (also called a sequence) of the activities, so that we can operate in the sequence space for finding improved solutions. A schedule is a list of starting times for each activity that respects the precedence and resource constraints. Since there are many possible schedules for any particular sequence, we must additionally specify a list-scheduling algorithm that generates a unique schedule for a given sequence. In our treatment here, we use a list-scheduling algorithm that schedules an activity from the sequence at the earliest precedence and resource feasible time. Note that this algorithm always generates a semi-active schedule (i.e., no local left shift is possible). We also make use of only simple eject and insert moves for the purpose of illustration. Given a sequence, this type of move ejects an element from the current sequence and inserts it after another element to generate a new solution. We use the notation (eject i, insert j) to denote the ejection of activity i from the current sequence and its insertion after activity j. Note that the index of activity j could be either higher or lower than the index of activity i, leading to two different types of moves.

4. 1 Multilevel Candidate Lists

Building multilevel candidate lists is a recommended first step in creating good TS implementations for precedence-constrained scheduling problems. The gains in efficiency by the use of this strategy can be considerable. At every iteration we have to make two decisions which consist of selecting the eject activity and the insert activity. The exhaustive examination strategy that considers all pairs (eject i, insert j) as a basis for such a move produces a very large neighborhood. A better strategy is to build candidate lists at multiple levels. Here we offer an example of a simple bi-level candidate list. This strategy produces solutions of very nearly the same quality as the exhaustive examination strategy but at substantially less computational cost. Our prescription constitutes an operative illustration of the *successive filtration strategy* discussed in a general context in Section 2.1. For the types of moves we consider, the outcome of a move typically can be viewed as the combined effect of two separate processes – one that decreases the start time of a certain activity (causing the activity to be "advanced") and the other that increases the start time of certain other activities (causing the activity to be "delayed"). In the example discussed here, we use two separate filters to control the selections.

This strategy aids in the construction of candidate lists of different sizes and characteristics at two levels. At the top level, there is a smaller list composed only of the activities that need to be processed earlier than their starting times in the current schedule. At the bottom level, there is a set of candidate lists each of whose member lists corresponds to a particular selection from the higher level list. The situation is shown schematically in *Figure 4*. For every selection from the higher level list, we have a set of possible candidates to select from the lower level list. This design can be generalized by declaring the lower level candidate list to contain pointers to possible candidate activity *sets* (rather than elements) that correspond to each selection from the higher level list. The management of these sets created under this generalized stipulation requires careful attention, and we suggest a "pause and project" strategy (see Section 4.2) to treat this general case.

The driving force behind the construction of the higher level candidate list usually derives from the objective function. For example, if the objective is to minimize the makespan of the project, as in the RCPSP, we would choose only those activities on the current critical paths that have been delayed as a consequence of a resource requirement. In general, during the improvement phase of the search, we want to maintain the size of this list as small as possible. Also we recommend a simple sorting scheme to reflect priorities concerning the order in which candidates from the candidate list are examined. (In *Example 2*, described below, once the higher level candidate list is generated we sort the list so that activities with higher indexes are examined earlier than those with lower indexes.)

The lower level candidate list is constructed with reference to a particular selection from the higher level candidate list. A strategy of constructing this list progressively is useful. The progressive construction strategy makes it possible to incorporate more search information dynamically into the list construction process. Established dominance rules from the literature can also be incorporated into the list construction process. Dominance rules to constrain the neighborhoods can be advantageously employed in the improvement phase of a TS algorithm. On the other hand, to drive the search away from powerful local optima the reverse strategy of allowing broad neighborhood definitions is preferable.

Example 2: Consider the problem instance in *Example 1*. The construction of higher and lower level candidate lists is illustrated for a lexicographic initial solution, whose schedule is shown in *Figure 3* (see *Example 1*). There is only one critical path in this solution, defined by the activity sequence *1-3-8-9-10-12*. Activities *1* and *12* have zero duration, and therefore do not appear in the Gantt chart. Comparing the starting times of these activities with those that would result if only the precedence constraints were considered (i.e., if the resource constraints were relaxed), it can be seen that only activities *9* and *10* are delayed in the current schedule. Instead, if all activities (including those not on a current critical path) are considered for a similar comparison, activities *6*, *9*, and *10* are all found delayed in the current schedule. Therefore, the higher level candidate list consists of these three activities. (However restructuring the selection of activities to only those on current critical paths provides better search intensification.) This is shown schematically in *Figure 4*.

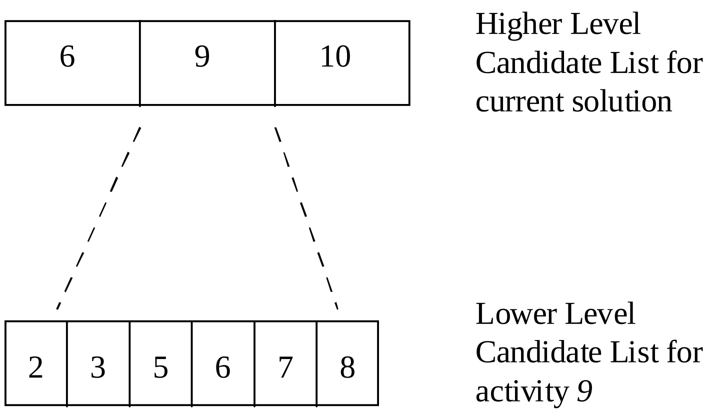
6
9
10
Higher Level Candidate List for current solution
2
3
5
6
7
8
Lower Level Candidate List for activity *9*

Figure 4. Bi-level Candidate List used in Example 2.

There are different ways of constructing the lower level candidate list. For illustrative purposes, in this example, we put on the lower level candidate list all those activities that can be delayed with respect to a current selection from the higher level list. Thus, corresponding to the selection of activity *9* from the higher level candidate list, we have activities *2*, *3*, *5*, *6*, *7*, and *8* in the lower level candidate list. Activity *4* is not a candidate for this list since the precedence constraints require that activity *4* be processed before activity *9*. A similar reasoning indicates that activity *10* on the higher level list gives rise to only activities *2* and *9* as candidates on the lower level list. With the selections above, a total of 11 moves will be evaluated for the current solution (three candidates for activity *6*, six candidates for activity *9*, and two candidates for activity *10*), assuming that the lower level candidates are ejected from their current positions and inserted immediately after the selection from the higher level list.

The neighborhood defined by these selections can be restricted further by one or more of the following choice rules:

a) only activities on the critical paths of the current solution are considered for higher level selections;
b) only activities on a current critical path are considered for higher level selections. (In cases where there are multiple critical paths in the current solution, this choice rule provides a smaller neighborhood than the choice rule in a).
c) selections for the lower level candidate lists are made by considering only those activities that are contiguous to the selection from the higher level list in the current schedule. (In *Example 2,* for selection *10* from the higher level list, we select activity *9*, as the only lower level candidate

as it is the unique activity contiguous with activity *10* in the corresponding schedule shown in *Figure 3*.)

Further choice rules can be constructed along similar lines.

4. 2 Pause and Project Strategies

First, we motivate the need for a pause and project strategy before integrating this strategy with the *sequential fan candidate list strategy*. Frequently in scheduling problems that contain precedence and resource constraints, problem influences that affect the move evaluation process are not fully accounted for when making a "myopic" type of move such as an (eject *i*, insert *j*). However, upon executing a few additional moves, such latent influences can sometimes surface quite clearly, permitting a better evaluation of the quality of the move that leads to such consequences. Hence at every stage of the process, we propose a *pause and project* strategy with the goal of uncovering relevant factors for evaluating outcomes that are not visible by considering only the immediate effects of a current prospective move. This approach is illustrated in *Example 3*.

Example 3: Consider again the resource-constrained project scheduling problem instance whose problem data are given in *Figure 2* and *Table 1*. Also, as in our earlier illustration, we begin with a current solution represented by the lexicographic sequence < *1*, *2*, *3*, *4*, *5*, *6*, *7*, *8*, *9*, *10*, *11*, *12*>, which gives rise to the Gantt Chart in *Figure 3*. Obviously, activity *9* is a member of the current higher level candidate list (see *Example 2*), and assume that we choose this activity to generate a move on the current step. Specifically, we select the move represented by the sequence <*1*, *2*, *3*, *4*, *5*, *6*, *7*, *9*, *8*, *10*, *11*, *12*>, whose corresponding schedule is shown in *Figure 5*.[*] Now, since this move causes a deterioration by increasing the makespan (from 23 to 24), in the absence of other considerations, it would not be chosen as the best move in the current neighborhood.[**] However it is easy to see that the increased makespan results from a resource conflict between activities *7* and *9* that prevents activity *9* from being advanced further in the current schedule. Consequently, we may be able to improve the current schedule if the current move (eject *9*, insert *7*) is combined with the next move (called the *associated* move) given by (eject *9*, insert 6) (see *Figure 6*). Since it cannot be known in advance that such improvement will occur, we "temporarily" accept the current move, and perform the associated

[*] This move is equivalent to the one that results from the "first move definition" in the neighborhood used by Baar *et al*. (1997).

[**] This move can also be considered equivalent to a swap move that exchanges two internal operations of a block in job-shop scheduling. While such a swap will not change the makespan value in the job-shop problem, it typically causes an increase in the makespan in RCPSP. Such "immersed internal block activities" show up only when the activities in question become "localized" in the schedule to compete for resources at the same time. When such activities are too far apart in the current schedule for the RCPSP, it is usually difficult to predict if one activity would temporarily block the advancement of another.

move (eject *9*, inert 6) immediately. The “pause” phase consists of tentatively accepting the current move while the “project” phase consists of performing the associated move and saving the combined move if it is better than the current best move considered.

Note that a progressive construction of the lower level candidate list permits us to adapt the search trajectory dynamically, based on the current search information. This strategy, built into the lower level candidate list, can be considered to be a special and very intense form of the *sequential fan candidate list strategy* discussed in Section 2.4 i.e., it is equivalent to a controlled version of a sequential fan candidate list strategy implementation in which the various parallel solution streams are spun off dynamically, with restricted depths. In spite of the restricted depth of each of the solution streams, substantial search intensification is obtained. This process represents a simple version of an ejection chain approach (see, e.g. Glover 1992, Rego 1997, Glover and Laguna 1997).

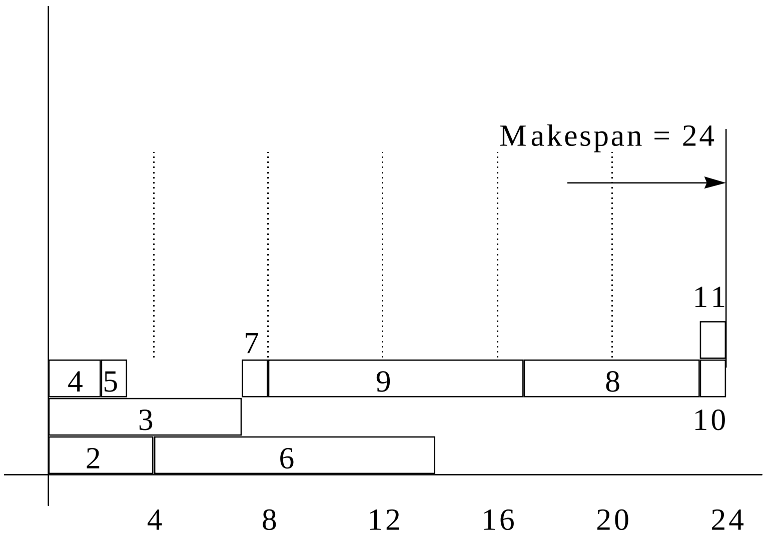

Figure 5. After the Pause Phase of a “Pause and Project” Move

In describing the general tenets of constructing the lower level candidate lists, we have left the details largely unspecified. This is to allow the construction of the lower level candidate list to incorporate as much problem-specific structure as possible. For example, in an improving phase of a standard job-shop scheduling algorithm, we know that a swap of the first two operations of the first block in a current solution can never improve the current makespan (Nowicki and Smutnicki, 1996). Such problem-specific information should be exploited in the lower level list construction process, especially in the improvement phases of the algorithm.

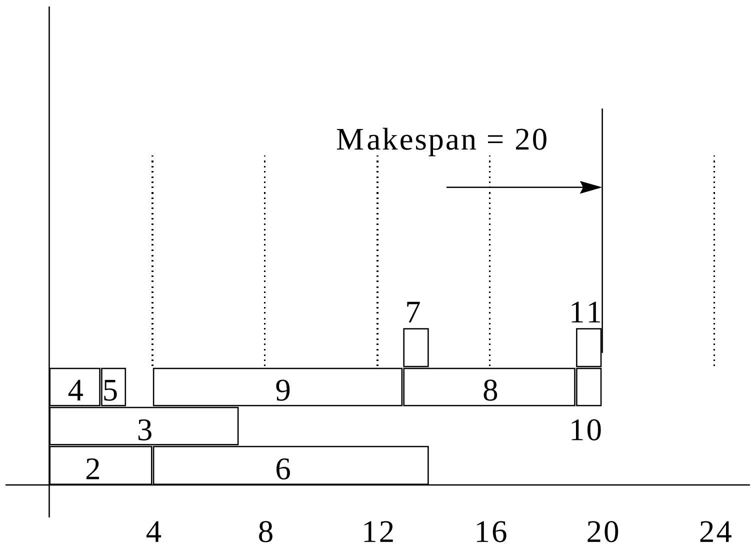

Figure 6. After the "Pause and Project" Move

4. 3 Aspiration Plus Strategy

To effectively use this strategy, we must first define the "quality threshold" for moves. To this end, we first motivate the notion of Aspiration by Quality. The aggressive orientation of TS contrasts with a greedy orientation by allowing the notion of *best* to embrace more than a simple objective function evaluation – and specifically by seeking a balance between the quality of a move and the amount of effort required to produce it. The *influence* of a move, as determined by the search history and problem context, is one of the important determinants of the meaning of best.

In the present context, the specific list scheduling algorithm we use to translate a given precedence feasible sequence into a schedule generally causes resource constraints at the beginning of the schedule to be tighter (i.e., to have less slack) than those at the end of the schedule. (Dell'Amico and Trubian (1993) make a similar observation, and compensate for this disparity by using a bi-directional algorithm to generate initial solutions for the JSP.) Given this empirical observation, a move may be conceived to be more attractive in the sense of exerting a positive influence if it temporarily increases the project makespan but reduces the maximum delay in a partial schedule, provided any associated delay occurs only in an activity that is scheduled toward the end of the project. (We assume here that the size of the partial schedule is not more than approximately half the size of a full schedule.) In such cases, the choice rules may benefit from a design that overrides other move evaluation criteria to accept such moves.

Example 4: Consider the problem instance described in *Example 1*. Suppose that the current solution is defined by the sequence <*1, 2, 3, 4, 9, 5, 8, 7, 11, 6, 10, 12* >. The makespan for this sequence is 20, and the schedule is the same as shown in *Figure 6*. (The schedules produced by <*1, 2, 3, 4, 5, 6, 9, 7, 8, 10, 11, 12* > and <*1, 2, 3, 4, 9, 5, 8, 7, 11, 6, 10, 12* > are the same). The delays for the delayed activities in this solution are 2 each for activities 9 and 6, and 6 each for activities 7, 8, 10, 11, and 12. (These values are obtained by relaxing the resource constraints and solving the resultant problem as a simple CPM problem.) Evaluating the move (eject 2, insert 7), we get the schedule shown in *Figure 7*. Although this solution has increased the makespan from 20 to 26, this move has a positive influence on the search since the delays of the delayed activities now are: 4 each for activities 7, 8, and 11; 11 for activity 2; 13 for activity 6; and 12 each for activities 10 and 12. Note that the maximum delay occurs for activity 6, and this activity is scheduled towards the end.

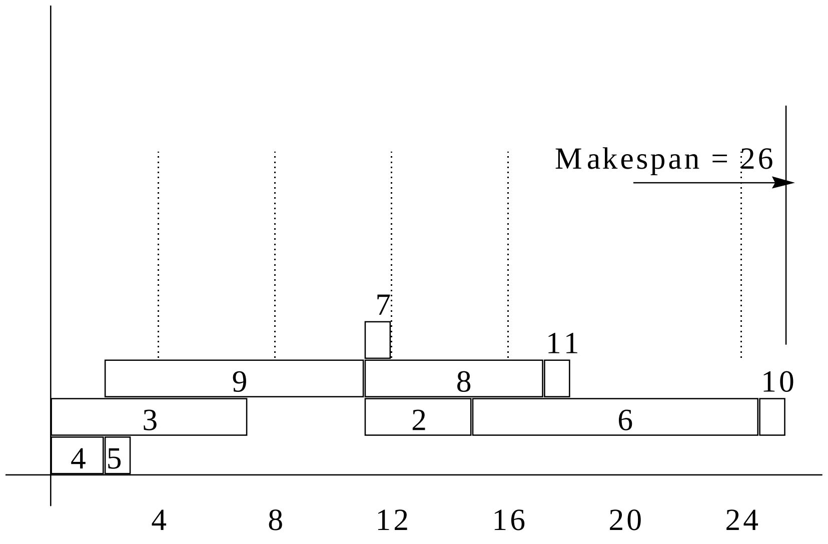

Figure 7. Gantt Chart Solution for Example 4.

We discuss next the use of Aspiration by Quality to construct a special type of candidate list. An empirical observation in the context of RCPSP indicates that on the average, most of the moves in the neighborhood of a current solution are non-improving while only a very few are improving. Thus, in line with the goal of achieving a balance between the effort expended on evaluating moves and the quality of the best move found, the *Aspiration Plus Strategy* (discussed in Section 2.3) becomes a useful basis for creating a candidate list. Here we define the aspiration level to be the first move that meets any one of the following three criteria:

a) produces a makespan better than the current best makespan (or in general the current best objective value);
b) creates a positive influence on the search trajectory (as in *Example 4*);
c) improves a secondary objective criterion subject to the limits imposed by the primary objective criterion.

Criterion c) is an important choice rule for defining aspiration levels since in scheduling applications, multiple critical paths are encountered frequently and a secondary objective in addition to the usual makespan is necessary to indicate good move evaluations. Depending on the stage of the search, the values of *Min* and *Max* for the Aspiration Plus Strategy may be adjusted accordingly. A more advanced variant also changes the Plus parameter values dynamically.

5. Computational Results

To provide a simple preliminary demonstration of the computational utility of a candidate list strategy, we report the results of a study that uses only the successive filtration strategy. The outline of the implementation in the form of a pseudo-code is shown in *Figure 8*. Again for simplicity, we elect to begin the search from a lexicographic initial solution. A bi-level candidate list, constructed progressively, is used to create candidate moves. All delayed activities on all the critical paths of the current solution are used to build the higher level candidate list. The lower level candidate list is composed of those activities which (a) are sequenced earlier than the candidate selected from the higher level list, and (b) are in progress just before the current start time of this candidate from the higher level list. The resultant neighborhood is relatively restricted in size.

Moves are defined by simple eject and insert operations on the current sequence. (For each possible move, sequence adjustments to maintain precedence feasibility are made, if necessary.) To illustrate, in *Example 1* (*Figure 3*), upon choosing activity *9* as the higher level candidate, we select activities *6* and *8* to be the associated lower level candidates. The moves in the neighborhood corresponding to this selection are: (eject *8*, insert *9*), and (eject *6*, insert *9*). Note that another move defined by (eject *9*, insert *4*) is also possible. Therefore we extend the options above to include the moves that result by inserting each element of the higher level list as early in the current sequence as possible. Thus element 9 of the higher level list also gives rise to the move (eject *9*, insert *4*) which causes activity 9 to be sequenced immediately after its predecessor, activity 4. The entire neighborhood (composed of (eject *9*, insert *4*), (eject *8*, insert *9*), and (eject *6*, insert *9*) for the example here) is evaluated and the best non-tabu move is chosen. To escape from local optimality, a simple tabu short-term memory with a fixed tabu tenure of 8 is used. This tabu tenure has been chosen to approximately match the average number of activities in the higher level candidate list, across all the problem instances. No diversification or intensification strategies are applied other than the ones implicit in the use of the candidate list approach. A lower bounding procedure based on the work of Mingozzi *et al*. (1994) is used to provide the lower bounds, so that the search can be terminated if a solution that matches the lower bound value is found. The results of this implementation found with static tabu tenure values held fixed at three levels of 2, 5 and 8 respectively are summarized in *Table 2*.

```
read data;
find resource-relaxed solution;
set best_solution = lexicographic solution;
find Mingozzi lower bound;
iter = 0;
if (best_solution > lower bound)
    do {
        initialize tabu data structures;
        flag = TRUE;
        while flag is TRUE
            do {
                best move value = bignum;
                build higher level candidate list;
                while there is an unexamined higher level candidate
                    do  {
                        select higher level candidate;
                        build lower level candidate lists;
                        while there is an unexamined lower level candidate
                            do {
                                select lower level candidate;
                                evaluate move;
                                if move value better than best move value
                                    {
                                    if move value + current solution better than best solution
                                        {
                                        override tabu status;
                                        update best move;
                                        }
                                    else
                                        {
                                        check tabu status;
                                        if not tabu, update best move;
                                        }
                                    }
                                }
                        }
                make best move;
                update current solution;
                if(current solution better than best solution)
                    {
                    store best solution;
                    }
                update tabu data structures;
                iter = iter + 1;
                if best solution equals lower bound or iter equals maxiter
                    flag = FALSE;
                }
        }
print best solution;
```

Figure 8. Pseudocode of algorithm

Three problem sets, generated using the ProGen software of Kolisch *et al.* (1995), each consisting of 480 instances are considered to be the current benchmarks. Optimal solutions are known for all the j30 problem set instances; however, optimality has not been established for all the j60 and j90 problem instances. The best solutions reported in the literature are used for the comparisons reported in *Table 2*. Overall, one new best solution for the j60 problem set and six new best solutions for the j90 problem set were found.

From *Table 2*, it is clear that the procedure described in *Figure 8* is able to produce solutions of high quality in spite of the simplicity of the implementation. Further, the procedure produces results that are relatively insensitive to tabu tenure values, in the range tested, in comparatively short times. The significant gains in solution quality were made possible by the inclusion of the candidate list construction. Note that in a more complete tabu search implementation for this application, the search would normally be initiated from the best starting solution produced by a set of heuristic dispatching rules. Also, more advanced strategies based on long-term memory, blending of neighborhoods, elite solution recovery, vocabulary building and so forth would typically be used to obtain better results.

TT	Set	N	N_best	N_new_best	D_Start (%)	D_alg (%)	Time
2	j30	480	302	0	9.45	1.8	0.4
	j60	480	272	1	9.7	2.1	1.1
	j90	480	278	6	7.9	1.3	2.1
5	j30	480	304	0	9.45	1.5	0.4
	j60	480	272	1	9.7	1.7	1.1
	j90	480	281	5	7.9	1.1	2.1
8	j30	480	314	0	9.45	1.4	0.4
	j60	480	271	0	9.7	1.7	1.1
	j90	480	280	4	7.9	1.1	2.1

TT	= Tabu tenure
Set	= Problem Set
N	= Number of instances
N_{best}	= Number of solutions matching the best reported upper bound
$N_{new\ best}$	= Number of new best solutions found
D_{start}	= Average deviation of the starting solutions from the best known solutions
D_{alg}	= Average deviation of the final solutions from the best known solutions
Time	= Average time per instance in seconds on a DEC ALPHA 2000 machine

Table 2. Summary of results

6. Conclusions

In this paper, we have discussed some important but often neglected candidate list strategies that deserve fuller consideration in implementing tabu search methods. To clarify the application of such strategies, we have introduced numerical examples in the resource constrained scheduling domain, and have illustrated how the general form of the indicated strategies can be specialized to achieve specific purposes (for goals such as intensification and diversification) within this setting. Empirical verification of the potential value of employing such candidate list strategies in scheduling is demonstrated by preliminary experimentation with the successive filtration strategy, which constitutes one of the simpler candidate list approaches. Combining this procedure with an elementary (naïve) tabu search procedure yields surprisingly good results, matching best known solutions on many problems in the literature, and obtaining a few solutions better than those previously recorded as best. In addition, these outcomes were obtained with a very small investment of computer time, generally about one second per instance. A more comprehensive computational study that examines additional candidate list strategies will be reported in a sequel.

References

Baar, T., P. Brucker and S. Knust, (1997). “Tabu-Search Algorithms for the Resource-Constrained Project Scheduling Problem,” *Technical Report*, Universität Osnabrück.

Dell’Amico M. and M. Trubian, (1993). “Applying tabu search to the job-shop scheduling problem,” *Annals of Operations Research 41,* pp.231-252.

Demeulemeester E. and W. Herroelen, (1992). “A branch-and-bound procedure for the multiple resource-constrained project scheduling problem,” *Management Science 38,* pp. 1803-1818.

James, R. J. W. and J. T. Buchanan (1997) “Performance Enhancements to Tabu Search for the Early/Tardy Scheduling Problem” *to appear in the European Journal of Operational Research Special Issue on Tabu Search*.

Glover, F. (1992). “Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems,” University of Colorado. Shortened version published in *Discrete Applied Mathematics 65*, 223-253, 1996.

Glover, F. (1996). “Tabu Search and Adaptive Memory Programming – Advances, Applications and Challenges,” *Interfaces in Computer Science and Operations Research.* Barr, Helgason and Kennington, eds., Kluwer Academic Publishers.

Glover, F. and M. Laguna, (1997). Tabu Search, Kluwer Academic Publishers.

Glover, F. (1997). Tabu Search, To appear in the *Special Issue of the European Journal of Operational Research*.

Kolisch, R., A. Sprecher, A. Drexl, (1995). “Characterization and generation of a general class of resource-constrained project scheduling problems,” *Management Science 41,* pp.1693-1703.

Lokketangen, A. and F. Glover, (1997). “Candidate List and Exploration Strategies for Solving 0\1 MIP Problems using a Pivot Neighborhood,” *MIC’97 2nd International Conference on Metaheuristics,* Versailles, France, July 21-24.

Minghe Sun, J. E. Aronson, P. G. McKeown, D. Drinka (1997) “A Tabu Search Heuristic Procedure for the Fixed Charge Transportation Problem” *to appear in the European Journal of Operational Research Special Issue on Tabu Search*.

Mingozzi, A., V. Maniezzo, S. Ricciardelli, L. Bianco, (1994). “An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation,” *Technical Report n.32*, Department of Mathematics, University of Bologna, Italy.

Morton, T. E. and Pentico, D. W. (1993) “Heuristic Scheduling Systems,” Wiley Series in Engineering and Technology Management, Wiley, New York.

Nowicki, E. and C. Smutnicki, (1996). “A Fast Taboo Search Algorithm for the job shop problem,” *Management Science* 42, 797-813.

Rego, C. (1997) “Relaxed Tours and Path Ejections for the Traveling Salesman Problems,” Universidade Portucalense, Porto, Portugal, *to appear in the European Journal of Operational Research*.

Sampson, S. E. and E. N. Weiss, (1993). “Local Search Techniques for the Generalized Resource Constrained Project Scheduling Problem,” *Naval Research Logistics 40 (5)*, 665-676.

Woodruff, D. L. and M. L. Spearman (1992). “Sequencing and Batching for Two Classes of Jobs with Deadlines and Setup Times,” *Production and Operations Management 1\1*, 87-102.