Exploiting the Power of Local Search in a Branch and Bound Algorithm for Job Shop Scheduling

Matthew J. Streeter[1] **and Stephen F. Smith**[2]
Computer Science Department
and Center for the Neural Basis of Cognition[1] and
The Robotics Institute[2]
Carnegie Mellon University
Pittsburgh, PA 15213
{matts, sfs}@cs.cmu.edu

Abstract

This paper presents three techniques for using an iterated local search algorithm to improve the performance of a state-of-the-art branch and bound algorithm for job shop scheduling. We use iterated local search to obtain (*i*) sharpened upper bounds, (*ii*) an improved branch-ordering heuristic, and (*iii*) and improved variable-selection heuristic. On randomly-generated instances, our hybrid of iterated local search and branch and bound outperforms either algorithm in isolation by more than an order of magnitude, where performance is measured by the median amount of time required to find a globally optimal schedule. We also demonstrate performance gains on benchmark instances from the OR library.

1. Introduction

Iterated local search and chronological backtracking have complementary strengths and weaknesses. The strength of backtracking is its systematicity: it is guaranteed to find a global optimum in a bounded amount of time. Its weakness is the fact that it performs a depth-first search: once it visits a region of the search space, it must explore that region exhaustively before moving onward, potentially wasting a lot of time. Iterated local search moves more freely about the search space and often finds a near-optimal solution relatively quickly in practice. The downside is that the amount of time it searches before finding a global optimum is unbounded.

Previous work has addressed the weaknesses of chronological backtracking in a number of ways: through randomization and restart (Gomes 2003), tree search strategies such as limited discrepancy search (Harvey & Ginsberg 1995) and depth-bounded discrepancy search (Walsh 1997), intelligent backtracking techniques such as dynamic and partial-order dynamic backtracking (Ginsberg 1993; Ginsberg & McAllester 1994), and other methods. There have also been a number of attempts to exploit the power of local search within chronological backtracking (Zhang & Zhang 1996; Kamarainen & Sakkout 2002; Nareyek, Smith, & Ohler 2003). The latter techniques are discussed more fully in §8.

In this paper we explore three simple ways of exploiting the power of iterated local search within branch and bound:

1. **Upper bounds.** Near-optimal solutions provide sharpened upper bounds that can be used to prune additional nodes in the branch and bound search tree.
2. **Branch ordering.** In practice, a near-optimal solution to an optimization problem will often have many attributes (i.e., assignments of values to variables) in common with an optimal solution. In this case, the heuristic ordering of branches can be improved by giving preference to a branch that is consistent with the near-optimal solution.
3. **Variable selection.** The heuristic solutions used for variable selection at each node of the search tree can be improved by local search.

We demonstrate the power of these techniques on the job shop scheduling problem, by hybridizing the $I\text{-}JAR$ iterated local search algorithm of Watson el al. (2003a) with the branch and bound algorithm of Brucker et al. (1994).

1.1. Contributions

The primary contributions of this work are as follows.

- We quantitatively compare the performance of iterated local search and branch and bound on random JSP instances. The results of this comparison nicely illustrate the complementary strengths of these two classes of algorithms: the branch and bound algorithm is orders of magnitude more efficient at finding a globally optimal schedule, while iterated local search is orders of magnitude more efficient at finding a near-optimal schedule.
- We show experimentally that for random square JSP instances, the near-optimal schedules that are found quickly by iterated local search are typically only a short distance away from the nearest globally optimal schedule. This suggests using the results of iterated local search to guide the branch ordering decisions made by branch and bound, and we show that such an approach does in fact lead to a more accurate branch ordering heuristic.
- Motivated by these observations, we show how to use iterated local search to improve the upper bounds, branch ordering heuristic, and variable selection heuristic used in branch and bound. On random instances, we find that each of these techniques improves performance by a factor that increases with problem size, where performance is measured by the median time required to find a globally optimal schedule.

2. Background

2.1. The Job Shop Scheduling Problem

We consider the widely studied makespan-minimization version of the job shop scheduling problem ($J\|C_{max}$), which we refer to simply as the JSP.

An N by M instance of the JSP is a set of N jobs, each of which is a sequence of M operations. Each operation must be performed on a designated machine for a specified duration, without interruption. There are M machines, and each job uses each machine exactly once. A schedule assigns start times to each operation such that

1. no machine is scheduled to process more than one operation at the same time, and
2. the ordering of operations within each job is respected.

The *makespan* of a schedule is equal to the maximum completion time (i.e., start time plus duration) of any operation. We consider the makespan-minimization version of the job shop scheduling problem, in which the objective is to find a schedule that minimizes the makespan.

It is convenient to represent a schedule by its *disjunctive graph* (Roy & Sussmann 1964). In a disjunctive graph, there is a vertex corresponding to each operation in the problem instance, as well as two special vertices called the *source* and the *sink*. There are directed edges pointing from the source into (the vertex corresponding to) the first operation of each job, from the last operation of each job into the sink, and from each operation into the next operation (if any) in the job. A directed edge from operation o_1 to operation o_2 indicates that o_1 completes before o_2 starts. The orientation of all the directed edges just discussed is dictated by the problem instance, and these directed edges are called *conjunctive arcs*. The remaining edges connect fixed pairs of operations, but their orientation is defined by a particular schedule. These *disjunctive edges* connect all pairs of operations performed on the same machine. A disjunctive edge with an assigned direction is called a *disjunctive arc*. The weight of each arc is given by the duration of the operation that the edge points out of (or zero if the edge points out of the source). A *critical path* is a longest (weighted) path from source to sink in the (directed) disjunctive graph that represents some particular schedule. It can be shown that the makespan of the schedule is equal to the length of its critical path.

Figure 1 illustrates (A) a JSP instance, (B) a schedule for that instance, and (C) the corresponding disjunctive graph.

2.1.1. Distance Between Schedules. Given two schedules s and s', we define the distance $\|s - s'\|$ between them as the proportion of disjunctive edges that point in opposite directions in the two schedules (Mattfeld, Bierwirth, & Kopfer 1999).

2.2. Algorithms for Job Shop Scheduling

We focus our efforts on a single iterated local search algorithm (I-JAR) and a single branch and bound algorithm ($Brucker$). We chose I-JAR because of its simplicity and its demonstrated performance on benchmark instances of the JSP. We chose $Brucker$ because of its performance and because code for it is freely available online.

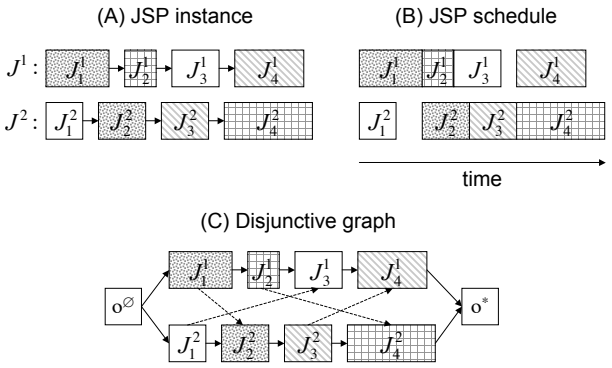

Figure 1: (A) A 2x4 instance of the JSP with jobs J^1 and J^2. Job J^1 consists of the sequence of operations $(J^1_1, J^1_2, J^1_3, J^1_4)$, and J_2 consists of the sequence $(J^2_1, J^2_2, J^2_3, J^2_4)$. Each operation is represented by a rectangle whose texture represents the machine on which the operation is performed and whose width is proportional to the operation's duration. (B) A JSP schedule assigns a start time to each operation. (C) A disjunctive graph represents start times indirectly by defining precedence relations between each pair of operations performed on the same machine. Here $o^{\emptyset}$ is source and o^* is the sink.

2.2.1. I-JAR. Watson et al. (2003a) present a simple iterated local search algorithm called I-JAR whose performance is competitive with the tabu search algorithm of Taillard (1994) on benchmark instances from the OR library. I-JAR performs a series of iterations. In each iteration, it first descends to a local optimum, then escapes from the local optimum by making a number of random moves.

Formally, let $\mathcal{N}_1(s)$ denote the set of all schedules that can be obtained from s by reversing the orientation of a single disjunctive arc that belongs to a critical path (van Laarhoven, Aarts, & Lenstra 1992). I-JAR is defined as follows.

Procedure I-JAR:

1. Initialize $cur \leftarrow$ a randomly-generated schedule.
2. Do:
   (a) *(Descent to local optimum).* For each schedule $s \in \mathcal{N}_1(cur)$, in random order:
      i. If $makespan(s) < makespan(cur)$ then set $cur \leftarrow s$ and go to 2 (a).
   (b) With probability $\frac{1}{100}$, set $l \leftarrow 5$; otherwise set $l \leftarrow 2$.
   (c) *(Random walk).* For i from 1 to l:
      i. Let s be a random element of $\mathcal{N}_1(cur)$, and set $cur \leftarrow s$.

2.2.2. Brucker's branch and bound algorithm. Brucker et al. (1994) present a branch and bound algorithm for job shop scheduling, hereafter referred to as $Brucker$. As in other branch and bound algorithms for job shop scheduling, each search tree node in $Brucker$ represents a set of disjunctive arcs (the "fixed arcs") that must be present in all descendents of that node. Branches are generated by constructing a schedule consistent with the fixed arcs and examining (one of) the schedule's critical path(s). Formally, if G is a set of disjunctive arcs, the procedure $Brucker(G)$ is as follows ($upper\_bound$ is a global variable initialized to ∞).

Procedure $Brucker(G)$:

1. *(Constraint propagation).* Add to G disjunctive arcs that must be present in any schedule with makespan $< upper\_bound$.
2. *(Pruning).* If $lower\_bound(G) \geq upper\_bound$ return.
3. *(Heuristic scheduling).* Using a priority dispatching rule, generate a schedule s that is consistent with G. Set $upper\_bound \leftarrow \min(upper\_bound, makespan(s))$.
4. *(Branching).* Find a critical path, P, in s. P is used to define branches G_1, G_2, ..., G_n (the order of the branches is determined heuristically using data obtained during the computation of lower bounds). For i from 1 to n:
   (a) Call $Brucker(G_i)$.

We will refer to steps 1-3 in the above code as an *iteration* of $Brucker$. The code for $Brucker$ is freely available via ORSEP (Brucker, Jurisch, & Sievers 1992).

Among systematic search algorithms, the performance of $Brucker$ is state-of-the-art for smaller benchmark instances from the OR library and among the best for larger instances (Brinkkötter & Brucker 2001).

We consider two additional algorithms, DDS and LDS, that are identical to $Brucker$ except that they do not use a depth-first tree search strategy. DDS instead uses depth-bounded discrepancy search (Walsh 1997), while LDS uses limited discrepancy search (Harvey & Ginsberg 1995). As compared to depth-first search, these two tree search strategies have been found to reduce the number of nodes that must be explored before finding an optimal or near-optimal solution.

3. Methodology

This section describes how we evaluate the performance of algorithms for job shop scheduling.

3.1. Test Instances

To generate a random N by M JSP instance we let the order in which the machines are used by each job be a random permutation of $\{1, 2, ..., M\}$, and draw each operation duration uniformly at random from $\{1, 2, ..., 100\}$.

For each $N \in \{6, 7, 8, 9, 10, 11, 12, 13\}$ we generate a set, $\mathcal{I}_{N,N}$, of random N by N JSP instances. For $N \leq 12$, $|\mathcal{I}_{N,N}| = 1000$, while $|\mathcal{I}_{13,13}| = 150$. For each random instance I, we used $Brucker$ to determine its optimal makespan, denoted $opt\_makespan(I)$.

Our evaluation focuses on square JSP instances (i.e., those with $N = M$) because they have been found to be the most difficult in practice (Fisher & Thompson 1963).

3.2. Performance Metric

In quantifying the performance of an algorithm $\mathcal{A}$, we focus on the amount of time required to find an optimal or near-optimal schedule with a specified minimum probability. Specifically, for real numbers $q \in (0, 1)$ and $\rho \geq 1$ and integers N and M, we determine the minimum t such that, when A is run on a random N by M JSP instance for t seconds, with probability at least q it finds a schedule whose makespan is at most ρ times the optimal makespan.

Given a set $\mathcal{I}_{N,M}$ of N by M JSP instances with known optimal makespans, our procedure for determining t is straightforward. We run $\mathcal{A}$ on each instance $I \in \mathcal{I}_{N,M}$ with a time limit of T = 1 second, terminating the run immediately if it finds a ρ-optimal schedule (i.e., a schedule whose makespan is at most ρ times the optimal makespan). For each instance, we record the amount of time that $\mathcal{A}$ ran and whether or not it found a ρ-optimal schedule. If the proportion of runs that found a ρ-optimal schedule by time T is at least q, we find the smallest $t \leq T$ such that the proportion of runs that found a ρ-optimal schedule by time t is at least q. Otherwise, we double T and try again. Each run of $\mathcal{A}$ on a particular instance I uses the same random number generator seed, so that our results would be exactly the same (though it would take longer to compute them) if we had initialized T to infinity.

All experiments reported in this paper were performed on a 2.4 GHz Pentium IV with 512 MB of memory.

4. Motivations

In this section we present some experiments that motivate this work.

4.1. Comparing Local and Exhaustive Search

4.1.1. Methodology. For each $N \in \{6, 7, 8, 9, 10, 11\}$, each $\rho \in \{1, 1.05\}$, and each $\mathcal{A} \in \{I\text{-}JAR, Brucker\}$, we used the procedure described in §3.2 to determine the number of iterations required by $\mathcal{A}$ to find a schedule whose makespan is within a factor ρ of optimal with probability at least $q = 0.9$.

4.1.2. Results Figure 2 shows the time (in seconds) required by $I\text{-}JAR$ and $Brucker$ to find either (A) an optimal schedule or (B) a near-optimal schedule with probability at least $q = 0.9$ when run on a random N by N JSP instance. The key observations are that

- the time required by $I\text{-}JAR$ to find a globally optimal schedule exceeds the time required by $Brucker$ by a factor that increases with problem size, ranging from 5.23

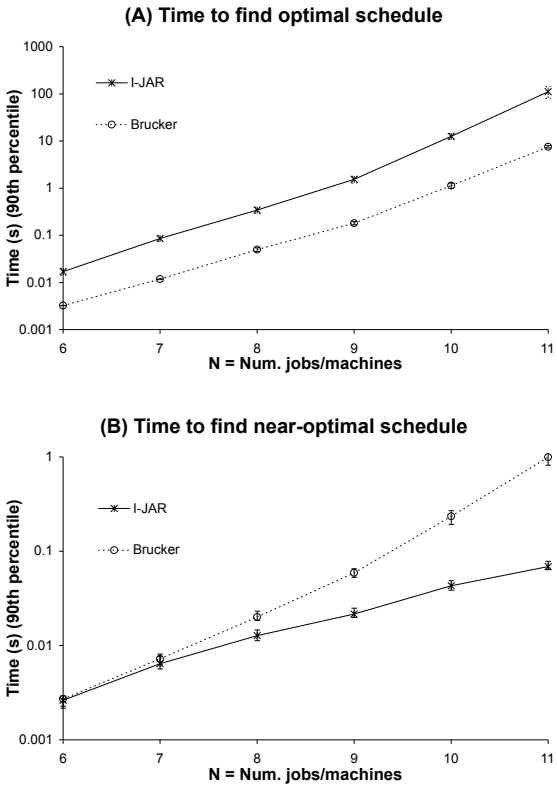

Figure 2: Amount of time (90^{th} percentile) required by $I\text{-}JAR$ and $Brucker$ find (A) a globally optimal schedule and (B) a schedule whose makespan is within a factor 1.05 of optimal, when run on a random N by N JSP instance. Error bars are 95% confidence intervals.

(for 6x6 instances) to 14.7 (for 11x11 instances); however,

- the amount of time required by $Brucker$ to find a *near-optimal* schedule exceeds the time required by $I\text{-}JAR$ by a factor that increases with problem size, ranging from 1.03 (for 6x6 instances) to 14.4 (for 11x11 instances).

These results suggest that a short run of $I\text{-}JAR$ often yields a near-optimal schedule, thus providing an upper bound that may improve the performance $Brucker$.

4.2. Mean Distance to Nearest Optimal Schedule

In this section we estimate the mean distance between schedules found by $I\text{-}JAR$ and the nearest globally optimal schedule (recall that distance between schedules is measured by the proportion of disjunctive edges whose orientations differ).

4.2.1. Methodology For each instance $I \in \mathcal{I}_{6,6} \cup \mathcal{I}_{7,7} \cup \mathcal{I}_{8,8}$, we run $I\text{-}JAR$ until it evaluates a globally optimal schedule. For $\rho \geq 1$, let s_ρ be the first schedule evaluated by $I\text{-}JAR$ whose makespan is within a factor ρ of optimal. For each $\rho \in \{1, 1.01, \ldots, 1.25\}$, we determine the distance from s_ρ to the nearest optimal schedule. The distance

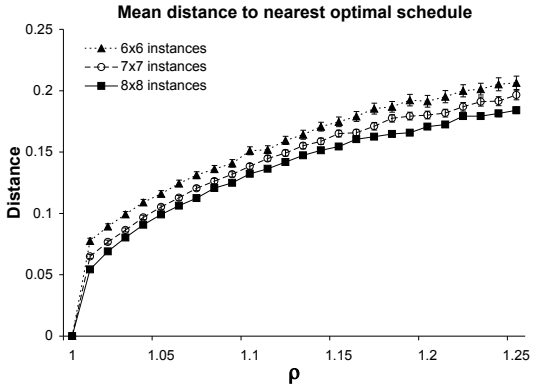

Figure 3: Mean distance from a schedule within a factor ρ of optimal to the nearest optimal schedule. Error bars are 95% confidence intervals.

is computed using a "radius-limited" version of $Brucker$ which determines, for a given radius r and center schedule s_c, whether there exists an optimal schedule in the set $\{s : \|s - s_c\| \leq r\}$. The smallest r for which this set contains an optimal schedule is determined using binary search.

4.2.2. Results Figure 3 presents our results. For each $N \in \{6, 7, 8\}$, Figure 3 shows that the mean distance from s_ρ to the nearest optimal schedule is an increasing function of ρ. For $N = 8$, for example, the mean distance for $\rho = 1.01$ is 0.054, while the mean distance for $\rho = 1.25$ is 0.184.

Suppose that a search tree node G has branches $G_1, G_2, \ldots, G_n$, and let s^* be a known near-optimal schedule that is consistent with G (i.e., the disjunctive graph of s^* contains all the disjunctive arcs in the set G). Based on Figure 3, it is likely that there exists a globally optimal schedule $\bar{s}$ that has many attributes in common with s^*. This suggests that if a branch G_i is consistent with s^* it may also be consistent with $\bar{s}$, so if we want to minimize the work that branch and bound must perform before finding a global optimum it makes sense to move G_i to the front of the list of branches.

The results in Figure 3 are not that surprising in light of previous work. For example, the Markov model of Watson et al. (2003b) shows that as a tabu search run progresses the mean distance to the nearest global optimum decreases. Our results are also consistent with the "big valley" picture of JSP landscapes, for which there is empirical (Nowicki & Smutnicki 2001) as well as theoretical evidence (Streeter & Smith 2005), especially for random N by M JSP instances with $\frac{N}{M} \approxeq 1$.

5. Improving Branch and Bound

5.1. Upper Bounds

The data presented in §4.1 show that I-JAR finds a near-optimal schedule (i.e., one whose makespan is within a factor 1.05 of optimal) more quickly than Brucker's branch and bound algorithm. This suggests running the two algorithms in parallel, each at 50% strength, and using the schedules discovered by $I\text{-}JAR$ to update the upper bound used by

$Brucker$. We refer to this hybrid algorithm as UB.

Specifically, for an algorithm $\mathcal{A} \in \{I\text{-}JAR, Brucker\}$, let $T_{\mathcal{A}}(N, M)$ denote the mean CPU time per iteration that $\mathcal{A}$ requires when run on a random N by M JSP instance. Let $I_{equiv}(N, M) = \left\lfloor \frac{T_{Brucker}(N,M)}{T_{I\text{-}JAR}(N,M)} \right\rfloor$. UB is defined as follows.

> Procedure UB:
>
> 1. Do:
>   (a) Perform $I_{equiv}(N, M)$ iterations of $I\text{-}JAR$. Let ℓ be the makespan of the best schedule found by $I\text{-}JAR$. Set $upper\_bound \leftarrow \min(upper\_bound, \ell)$.
>   (b) Perform a single iteration of $Brucker$.

5.2. Branch Ordering

The results of §4.2 show that near-optimal schedules tend to have many disjunctive arcs in common with globally optimal schedules. We define a hybrid algorithm UB+BR that is designed to take advantage of this fact. Like UB, UB+BR runs $Brucker$ and $I\text{-}JAR$ in parallel and uses the schedules found by $I\text{-}JAR$ to update upper bounds. Additionally, UB+BR alters the branch ordering heuristic used by $Brucker$ as follows:

- Let $(G_1, G_2, \ldots, G_n)$ be the branches as ordered by $Brucker$. Let s^* be the best schedule found so far by $I\text{-}JAR$. If for some i, G_i is consistent with s^* (i.e., s^* contains all the disjunctive arcs in G_i) then move G_i to the front of the list of branches.

Note that when s^* changes, the branch ordering heuristic changes as well, and we may want to backtrack to revisit branching decisions that were made earlier in the search. To facilitate this, we maintain in memory a tree $\mathcal{T}$ of search tree nodes. Each node in $\mathcal{T}$ is initially marked as "unexpanded", then after constraint propagation, lower bounding, and branching have been performed as per §2.2.2, the node is marked as "expanded". Nodes are deleted from the search tree when all descendents have either been pruned or exhaustively explored. Once $\mathcal{T}$ is empty, the algorithm terminates. Formally, we have the following.

> Procedure used by UB+BR to select a search tree node to expand:
>
> 1. Starting at the root of $\mathcal{T}$, follow first-ranked branches until reaching an unexpanded node G. Process G as per the pseudo-code in §2.2.2, and mark G as "expanded".
> 2. If G is pruned, remove G from the tree and delete from $\mathcal{T}$ any "expanded" nodes that no longer have any children.
> 3. Otherwise, add $G_1, G_2, \ldots, G_n$ as "unexpanded" children of G.

In effect, in between updates to s^* UB+BR simply performs a depth-first search. When s^* changes, UB+BR backtracks to the root node of the search tree and continues the search using the new branch ordering (but without discarding the work that has already been done).

If s^* is updated n times, the memory overhead (as compared to depth-first search) is at most a factor of n. When each operation duration is drawn from $\{1, 2, \ldots, 100\}$, $n \leq 100NM$, so the memory required by UB+BR is at most $100NM$ times that required by $Brucker$ in the worst case. In practice, the actual memory requirement is much less.

5.3. Variable Selection

Our third hybrid algorithm, UB+BR+VS, is identical to UB+BR except that it uses local search to generate heuristic schedules (step (3) of the pseudo-code for $Brucker$ presented in §2.2.2), rather than using a priority dispatching rule.

To generate a heuristic schedule that is consistent with a set of disjunctive arcs G, we start $I\text{-}JAR$ at a schedule that is consistent with G and use a restricted move operator $\mathcal{N}_1^{(G)}$ where $\mathcal{N}_1^{(G)}(s) = \{s \in \mathcal{N}_1(s) : s \text{ is consistent with } G\}$. Formally, the procedure is as follows.

> Procedure used by UB+BR+VS to generate heuristic schedules:
>
> 1. Let s be the heuristic schedule generated by $Brucker$.
> 2. Starting at s, run $I\text{-}JAR$ for $I_{equiv}(N, M)$ iterations, using the move operator $\mathcal{N}_1^{(G)}$. Let $\hat{s}$ denote the best schedule found, and let $\hat{\ell}$ be its makespan.
> 3. Set $upper\_bound \leftarrow \min(upper\_bound, \hat{\ell})$.
> 4. Return $\hat{s}$.

Note that the per-search-tree-node runs of $I\text{-}JAR$ used to perform heuristic scheduling are independent of the parallel run of $I\text{-}JAR$ used (as per UB+BR) to establish upper bounds. Also note that in altering step (3) of $Brucker$, we are changing the *variable selection* heuristic (which picks out variables on which to branch), which is distinct from the *branch ordering* heuristic (which determines the order in which branches of the search tree are explored).

6. Evaluation on Random JSP Instances

In this section we compare the performance of the algorithms $Brucker$, DDS, and LDS (described in §2.2.2), with that of UB, UB+BR, and UB+BR+VS (described in §5) on random JSP instances.

6.1. Median Run Length

Using the methodology of §3.2, we determine the median number of iterations required to find a globally optimal schedule for a random N by N JSP instance for each $N \in \{6, 7, 8, 9, 10, 11, 12, 13\}$ and each algorithm $\mathcal{A} \in \{Brucker, DDS, LDS, UB, UB\text{+}BR, UB\text{+}BR\text{+}VS\}$. Figure 4 presents the results. The key observation is that

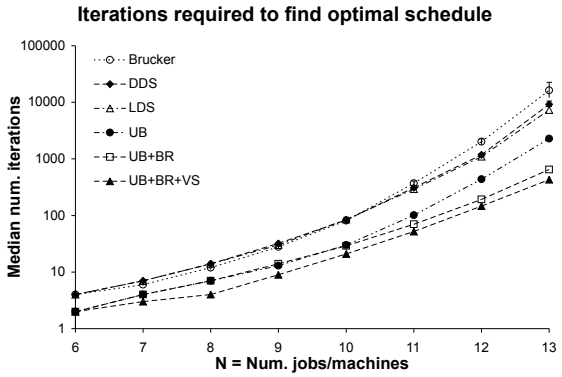

Figure 4: Median number of iterations required by various branch and bound algorithms to find a globally optimal schedule in a random N by N JSP instance. Error bars are 95% confidence intervals.

Table 1: Performance on random 13x13 instances.

Algorithm	**Med. iter.**	**Med. equiv. iter.**
$Brucker$	16300	16300
DDS	9220	9220
LDS	7300	7300
UB	2291	4582
UB+BR	650	1300
UB+BR+VS	429	1287

- Compared to $Brucker$, each of the three techniques (UB, UB+BR, and UB+BR+VS, respectively) reduces the median number of iterations required to find a global optimum by a factor that increases with problem size.

The iterations of these four algorithms are not equivalent in terms of CPU time. By design, an iteration of either UB or UB+BR takes approximately twice as long as an iteration of $Brucker$, while an iteration of UB+BR+VS takes approximately three times as long. Table 1 compares the performance of the four algorithms on the largest instance size, both in terms of raw iterations and "equivalent iterations".

As judged by this table, the performance of UB+BR+VS is not significantly better than that of UB+BR on the largest instance size. However, the trend in the data suggests that UB+BR+VS will significantly outperform UB+BR on larger instance sizes.

Both DDS and LDS outperformed the depth-first version of $Brucker$. It seems likely that the performance of the three hybrid algorithms could be further improved by using these tree search strategies.

6.2. Comparison of $\mathcal{B}_{Brucker}$ and $\mathcal{B}_{UB+BR}$

To understand the difference between the median run lengths of UB and UB+BR we examine the behavior of their branch ordering heuristics, referred to respectively as $\mathcal{B}_{Brucker}$ and $\mathcal{B}_{UB+BR}$.

Formally, let $\mathcal{B}$ be a branch ordering heuristic and let I be a random N by M JSP instance. Call a search tree node with disjunctive arc set G *optimal* if there exists a globally optimal schedule that contains all the arcs in G. Let G_d be an optimal search tree node at depth d in the search tree for I, and let $G_1, G_2, \ldots, G_n$ be the children of G_d, as ordered by some branch ordering heuristic $\mathcal{B}$. The *accuracy* of $\mathcal{B}$ at at depth d, which we denote by $a(\mathcal{B}, d, N, M)$, is the probability that the first-ranked branch (G_1) is optimal.

Given a branch ordering heuristic $\mathcal{B}$, we estimate $a(\mathcal{B}, d, N, M)$ (as a function of d) as follows.

Procedure for estimating $a(\mathcal{B}, d, N, M)$:

1. For each instance $I \in \mathcal{I}_{N,M}$:
   (a) Initialize $G \leftarrow \emptyset$, $upper\_bound \leftarrow 1.05 * opt\_makespan(I)$, and $d \leftarrow 0$.
   (b) Let $G_1, G_2, \ldots, G_n$ be the children of G, as ordered by $\mathcal{B}$.
   (c) For each $i \in \{1, 2, \ldots, n\}$, use an independent run of $Brucker$ to determine whether G_i is optimal. If G_1 is optimal record the pair $(d, true)$; otherwise record the pair $(d, false)$.
   (d) Let G_i be a random optimal element of $\{G_1, G_2, \ldots, G_n\}$; set $G \leftarrow G_i$; set $d \leftarrow d + 1$; and go to (b).
2. For each integer $d \geq 0$ for which some ordered pair of the form (d, V) was recorded, take as an estimate of $a(\mathcal{B}, d, N, M)$ the proportion of recorded pairs of the form (d, V) for which $V = true$.

Figure 5 plots $a(\mathcal{B}, d, N, N)$ as a function of d for each $N \in \{6, 7, 8, 9, 10, 11, 12\}$ and for each $\mathcal{B} \in \{\mathcal{B}_{Brucker}, \mathcal{B}_{UB+BR}\}$.

Examining Figure 5, we make four observations:

1. For all N, the trend is that $a(\mathcal{B}_{Brucker}, d, N, N)$ increases as a function of d.
2. For all d, the trend is that $a(\mathcal{B}_{Brucker}, d, N, N)$ decreases as a function of N.
3. For every combination of N and d, $a(\mathcal{B}_{UB+BR}, d, N, N) > a(\mathcal{B}_{Brucker}, d, N, N)$.
4. For all N, the trend is that $a(\mathcal{B}_{UB+BR}, d, N, N) - a(\mathcal{B}_{Brucker}, d, N, N)$ decreases with d.

Observation (1) is consistent with the conventional wisdom that branch ordering heuristics become more accurate at deeper nodes in the search tree (e.g., Walsh 1997), while observations (2) and (3) are consistent with our expectations. (4) deserves special explanation. The reason for (4) is that when applied to a node G_d, $\mathcal{B}_{UB+BR}$ only makes a decision that differs from that of $\mathcal{B}_{Brucker}$ if s^* is consistent with G_d. The probability that s^* is consistent with G_d decreases with d, and so the benefit of $\mathcal{B}_{UB+BR}$ decreases with d as well.

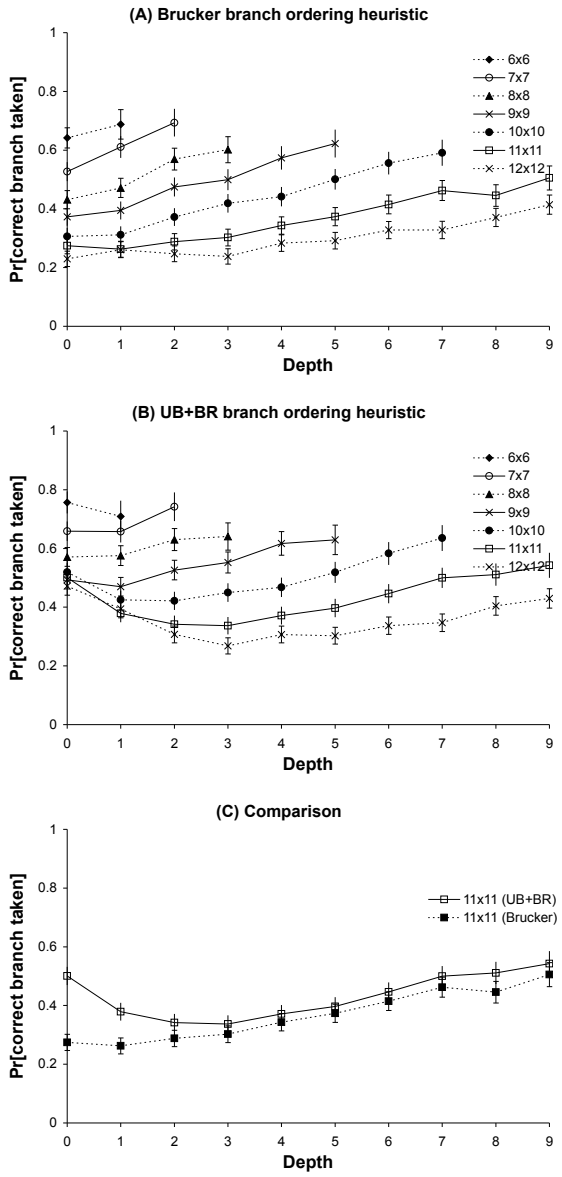

Figure 5: Accuracy of (A) $\mathcal{B}_{Brucker}$ and (B) $\mathcal{B}_{UB+BR}$ as a function of depth for random N by N JSP instances. For ease of comparison (C) superimposes the curves from (A) and (B) for $N = 11$.

Table 2: Mean CPU seconds required by various algorithms to find a globally optimal schedule. For stochastic algorithms, 95% confidence intervals are given.

Inst.	**Size**	**Brucker**	**I-JAR**	**UB+BR**
abz5	10x10	6.3	33.4 ± 10.7	4.3 ± 0.7
abz6	10x10	0.4	1.3 ± 0.3	0.3 ± 0.1
ft06	6x6	< 0.1	< 0.1	< 0.1
ft10	10x10	12.8	179.5 ± 60.5	9.5 ± 1.9
la01	10x5	< 0.1	< 0.1	< 0.1
la02	10x5	< 0.1	0.1 ± 0	0.1 ± 0
la03	10x5	< 0.1	0.1 ± 0	< 0.1
la04	10x5	0.1	0.1 ± 0	< 0.1
la05	10x5	< 0.1	< 0.1	< 0.1
la06	15x5	< 0.1	< 0.1	< 0.1
la07	15x5	< 0.1	< 0.1	< 0.1
la08	15x5	< 0.1	< 0.1	< 0.1
la09	15x5	< 0.1	< 0.1	< 0.1
la10	15x5	< 0.1	< 0.1	< 0.1
la11	20x5	< 0.1	< 0.1	< 0.1
la12	20x5	< 0.1	< 0.1	< 0.1
la13	20x5	< 0.1	< 0.1	< 0.1
la14	20x5	< 0.1	< 0.1	< 0.1
la15	20x5	0.1	< 0.1	0.1 ± 0
la16	10x10	0.8	11.9 ± 3.3	0.3 ± 0.1
la17	10x10	0.2	0.2 ± 0.1	0.1 ± 0
la18	10x10	1.0	0.3 ± 0.1	0.2 ± 0.1
la19	10x10	4.2	0.9 ± 0.2	0.8 ± 0.2
la20	10x10	4.2	1.0 ± 0.3	0.3 ± 0.1
la22	15x10	82.5	329.2 ± 84.0	4.5 ± 1.0
la23	15x10	44.6	0.1 ± 0	0.1 ± 0
la26	20x10	547.0	0.5 ± 0.1	1.1 ± 0.2
la30	20x10	3.8	0.2 ± 0	0.6 ± 0.1
la31	30x10	0.2	0.2 ± 0	0.2 ± 0
la32	30x10	< 0.1	0.1 ± 0	< 0.1
la33	30x10	1.8	0.1 ± 0	0.4 ± 0.1
la34	30x10	0.3	0.3 ± 0	0.8 ± 0.1
la35	30x10	0.6	0.2 ± 0	0.4 ± 0.1
orb01	10x10	80.3	81.1 ± 25.4	29.6 ± 8.6
orb02	10x10	6.7	20.1 ± 5.7	2.3 ± 0.4
orb03	10x10	180	49.5 ± 12.6	33.8 ± 10.4
orb04	10x10	23.8	191.5 ± 44.4	25.7 ± 2.3
orb05	10x10	8.6	194.9 ± 56.0	4.1 ± 0.7
orb06	10x10	38.5	13.7 ± 3.3	3.3 ± 0.7
orb08	10x10	14.7	150.5 ± 35.6	6.6 ± 1.7
orb09	10x10	3.4	12.7 ± 3.1	2.9 ± 0.6
orb10	10x10	2.7	1.7 ± 0.4	0.4 ± 0.1
swv16	50x10	0.1	0.2 ± 0	0.1 ± 0
swv17	50x10	0.1	0.2 ± 0	0.1 ± 0
swv18	50x10	0.1	0.2 ± 0	0.1 ± 0
swv19	50x10	0.2	0.3 ± 0	0.7 ± 0.1
swv20	50x10	0.1	0.2 ± 0	0.1 ± 0
Total		1070	1277 ± 145	134 ± 14.5

7. OR Library Evaluation

In this section we compare the performance of $Brucker$, $I\text{-}JAR$, $UB{+}BR$, and $UB{+}BR{+}VS$ on instances from the OR library. As stated in §3.2, all runs were performed on a 2.4 GHz Pentium IV with 512 MB of memory. First, we ran $Brucker$ with a time limit of 15 minutes on each of the 82 instances in the OR library. $Brucker$ proved that it found the optimal schedule on 47 instances. For each such instance, we recorded the amount of time that elapsed before $Brucker$ first evaluated a globally optimal schedule (in general this is much less than the amount of time required to *prove* that the schedule is optimal). Then for each of these 47 instances, we ran $I\text{-}JAR$, $UB{+}BR$, and $UB{+}BR{+}VS$ 50 times each, continuing each run until it found a globally optimal schedule. Table 2 presents the mean run lengths for $Brucker$, $I\text{-}JAR$, and $UB{+}BR$. The performance of $UB{+}BR{+}VS$ was about a factor 1.5 worse than that of $UB{+}BR$ on most instances, and is not shown.

Averaged over these 47 instances, the performance of $UB{+}BR$ is approximately 9.5 times better than that of $I\text{-}JAR$, 8 times better than that of $Brucker$, and 1.5 times better than that of $UB{+}BR{+}VS$ (not shown). We conjecture that the advantages of $UB{+}BR$ and $UB{+}BR{+}VS$ over $Brucker$ and $I\text{-}JAR$ would increase if we ran them on larger instances from the OR library.

8. Related Work

In this section we discuss previous hybrids of local and systematic search, and indicate the relationship between our techniques and previous work.

8.1. Local Search over Partial Assignments

A number of algorithms have been proposed in which a local search is performed on partial assignments (i.e., assignments of values to a subset of the problem variables), where for each partial assignment, the unassigned variables are assigned values using a systematic search. An early example is the shifting bottleneck algorithm for job shop scheduling (Adams, Balas, & Zawack 1988), in which the partial assignment includes a full set of disjunctive arcs for all machines except one. More recently, Büdenbender et al. (2000) apply a technique of this form to a transportation network design problem. Focacci et al. (2003) give a survey of instances of this general technique.

A related but distinct approach is to use local search to find the largest possible *consistent* partial assignment (i.e., a partial assignment that cannot be trivially pruned due to lower bounds or constraint violations). Instances of this approach include the constraint satisfaction algorithm of Zhang and Zhang (1996) and that of Prestwich (2002).

8.2. Local Search Probing

A number of recent papers have used local search "probes" to guide chronological backtracking. For each node in the search tree, a local search is performed on the subproblem defined by that node. Information from the local search is then used for variable and value selection.

Kamarainen and Sakkout (2002) apply this approach to the kernel resource feasibility problem. At each search tree node, they relax the subproblem by removing resource constraints, then solve the relaxed problem with local search. A resource constraint that is violated by the solution to the relaxation forms the basis for further branching. Nareyek et al. (2003) use a similar approach to solve the decision version of the JSP (the decision version asks whether a schedule with makespan $\leq k$ exists). They perform a local search on each subproblem, and branch by examining a constraint that is frequently violated by local search.

8.3. Discussion

Our upper bounding technique (using upper bounds from an incomplete search to reduce the work that must be performed by a systematic search) has no doubt been used without fanfare many times in practice. Our variable selection technique is an instance of local search probing. Our branch ordering heuristic differs from the two techniques just discussed in that we use an independent run of iterated local search (i.e., one that explores the entire search space, independent of the subspace currently being examined by the backtracking search) as guidance for a backtracking algorithm that remains complete.

9. Future Directions

9.1. Nogood Learning

In this paper we have focused on using information gained from iterated local search to improve the performance of branch and bound. But information can flow in the other direction as well. In particular, once a search tree node G is no longer open (i.e., branch and bound has proved that no schedule containing all the arcs in G can be optimal) it is wasteful for iterated local search to explore schedules that are consistent with G.

Previous work has showed how learned nogoods can be used to create *complete* local search algorithms for satisfiability (Fang & Ruml 2004) and job shop scheduling (Dilkina, Duan, & Havens 2005). Both of these approaches use resolution as a source of nogoods, but could be adapted to instead use branch and bound.

9.2. Improving $\mathcal{B}_{UB+BR}$

Figure 5 (C) shows that the improvement in accuracy obtained by using $\mathcal{B}_{UB+BR}$ in place of $\mathcal{B}_{Brucker}$ decreases with depth. It seems that it should be possible to obtain a more uniform improvement over $\mathcal{B}_{Brucker}$ by performing additional runs of $I\text{-}JAR$ and/or extracting more information from the existing run.

10. Conclusions

We have presented three techniques for incorporating iterated local search into branch and bound. Iterated local search can be used to establish upper bounds and to augment the branching ordering and variable selection heuristics. Empirically, we showed that each of these techniques

reduces the median number of iterations required to find a globally optimal schedule in a random N by N JSP instance. As N increases, each of the three techniques provides a larger boost in performance.

References

Adams, J.; Balas, E.; and Zawack, D. 1988. The shifting bottleneck procedure for job shop scheduling. *Management Science* 34(3):391–401.

Brinkkötter, W., and Brucker, P. 2001. Solving open instances for the job-shop problem by parallel head-tail adjustments. *Journal of Scheduling* 4:53–64.

Brucker, P.; Jurisch, B.; and Sievers, B. 1992. Job-shop (C-codes). *European Journal of Operational Research* 57:132–133. Code available at `http://optimierung.mathematik.uni-kl.de/ORSEP/contents.html`.

Brucker, P.; Jurisch, B.; and Sievers, B. 1994. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics* 49(1-3):107–127.

Büdenbender, K.; Grünert, T.; and Sebastian, H.-J. 2000. A hybrid tabu search/branch-and-bound algorithm for the direct flight network design problem. *Transportation Science* 34(4):364–380.

Dilkina, B.; Duan, L.; and Havens, W. 2005. Extending Systematic Local Search for Job Shop Scheduling Problems. In *Eleventh International Conference on Principles and Practice of Constraint Programming*.

Fang, H., and Ruml, W. 2004. Complete local search for propositional satisfiability. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, 161–166.

Fisher, H., and Thompson, G. L. 1963. Probabilistic learning combinations of local job-shop scheduling rules. In Muth, J. F., and Thompson, G. L., eds., *Industrial Scheduling*. Englewood Cliffs, NJ: Prentice-Hall. 225–251.

Focacci, F.; Laburthe, F.; and Lodi, A. 2003. Local search and constraint programming. In Glover, F., and Kochenberger, G. A., eds., *Handbook of Metaheuristics*. Boston, MA: Kluwer. 369–404.

Ginsberg, M. L., and McAllester, D. A. 1994. GSAT and Dynamic Backtracking. In Torasso, P.; Doyle, J.; and Sandewall, E., eds., *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, 226–237. Morgan Kaufmann.

Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

Gomes, C. 2003. Complete randomized backtrack search. In Milano, M., ed., *Constraint and Integer Programming: Toward a Unified Methodology*. Boston, MA: Kluwer. 233–283.

Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In Mellish, C. S., ed., *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95); Vol. 1*, 607–615. Montréal, Québec, Canada: Morgan Kaufmann, 1995.

Kamarainen, O., and Sakkout, H. E. 2002. Local probing applied to scheduling. In *Principles and Practice of Constraint Programming*, 155–171.

Mattfeld, D. C.; Bierwirth, C.; and Kopfer, H. 1999. A search space analysis of the job shop scheduling problem. *Annals of Operations Research* 86:441–453.

Nareyek, A.; Smith, S.; and Ohler, C. 2003. Integrating local search advice into a refinment search solver (or not). In *Proceedings of the CP-03 Workshop on Cooperative Constraint Problem Solvers*, 29–43.

Nowicki, E., and Smutnicki, C. 2001. Some new ideas in TS for job shop scheduling. Technical Report 50/2001, University of Wroclaw.

Prestwich, S. 2002. Combining the scalability of local search with the pruning techniques of systematic search. *Annals of Operations Research* 115:51–72.

Roy, B., and Sussmann, B. 1964. Les problèmes dordonnancement avec contraintes disjonctives. Note D.S. no. 9 bis, SEMA, Paris, France, Décembre.

Streeter, M. J., and Smith, S. F. 2005. Characterizing the distribution of low-makespan schedules in the job shop scheduling problem. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 61–70.

Taillard, E. 1994. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing* 6:108–117.

van Laarhoven, P.; Aarts, E.; and Lenstra, J. 1992. Job shop scheduling by simulated annealing. *Operations Research* 40(1):113–125.

Walsh, T. 1997. Depth-bounded discrepancy search. In *IJCAI*, 1388–1395.

Watson, J.-P.; Howe, A. E.; and Whitley, L. D. 2003a. An analysis of iterated local search for job shop scheduling. In *Proceedings of the Fifth Metaheuristics International Conference*.

Watson, J.-P.; Whitley, L. D.; and Howe, A. E. 2003b. A dynamic model of tabu search for the job shop scheduling problem. In *Multidisciplinary International Conference on Scheduling*.

Zhang, J., and Zhang, H. 1996. Combining local search and backtracking techniques for constraint satisfaction. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*, 369–374.