CS 227 Programming Assignment 3: Job Shop Scheduling

Todd Sullivan
todd.sullivan@cs.stanford.edu

Harry Robertson
harry.robertson@gmail.com

Pavani Vantimitta
pavani@stanford.edu

1 Introduction

Job shop scheduling problems arise in various areas of computing and real-life domains. To formalize a typical job shop scheduling problem, one considers a set of operations, which must be executed in a specified time window. The problem contains precedence constraints (i.e. operation A must occur before operation B) and resource constraints (i.e. operation A needs exclusive use of resource R during its execution). These scheduling problems are NP-complete.

One can translate a JSS problem into a temporal constraint satisfaction problem, for which the variables are the time points for the tasks involved and the constraints are precedence and time-window constraints.

In this paper we consider a set of 60 predefined JSS problems (each with 50 operations and 5 resources), which our algorithms aim to solve by scheduling the given set of operations according to the given set of constraints.

We start off by translating the base problem as a temporal problem and evaluating the earliest and latest start times for each of the operations. We use the Bellman-Ford algorithm to do this initialization. We then perform constraint-based search to order the operations that share resource needs, using Slack-based ordering heuristics to select which variables/values to pick. The resource constraints are satisfied by using a chronological backtracking search procedure to search for a consistent solution. We perform constraint propagation at each search node, either in the distance graph or in the constraint graph.

2 Algorithms

2.1 Bellman-Ford Algorithm

The Bellman-Ford algorithm computes the single-source shortest paths in a weighted distance graph. Unlike Dijkstra's algorithm, Bellman-Ford can be used in the presence of negative cycles, which is a necessary property for the problems at hand. Indeed, a temporal problem is considered to be consistent only if it does not have a cycle of negative weight, and Bellman-Ford recognizes this case.

The Bellman-Ford algorithm takes in a graph, and progressively computes the shortest paths from the given source node to the other nodes. At each iteration of its primary loop, Bellman-Ford attempts to "relax" each edge u-v, i.e. to see if there is a shorter path to v through the edge u-v. Bellman–Ford runs in $O(V.E)$ time, where V and E are the number of vertices and edges respectively (at each iteration it relaxes all edges, and iterates this up to |V| - 1 times). The repetitions allow minimum distances to accurately propagate throughout the graph, since in the absence of negative cycles the shortest path can only visit each node at most once (and negative cycles can be detected after the maximum number of iterations).

2.2 Ordering of Operations

Every task that has to be scheduled has an earliest start time *(est)*, a latest start time *(lst)* and a processing time *(p)*. There are four possible cases for ordering two given operations. When $\boldsymbol{est_i + p_i \leq lst_j}$ and $\boldsymbol{est_j + p_j > lst_i}$, then operation i must necessarily be scheduled before

operation *j* (Case 1). Case 2 is the reverse, when operation *j* necessarily precedes operation *i*. Case 3 is when there is a conflict between the two operations (in which case we need to backtrack). Case 4 is when either ordering is possible.

Constraint based analysis is used to rule out orderings in cases 1, 2 and 3; in these cases we continue by simply choosing the obvious ordering (or by backtracking in case 3). Slack based heuristics are used to decide which ordering to pick in case 4.

2.3 Search Procedure

We use chronological backtracking search to search for a solution satisfying the resource constraints. We first start by initializing using the Bellman-Ford algorithm discussed in Section 2.1. Then there are two functions, label() and unlabel(), that are important.

In the label function we select a value from the domain and propagate it through the graph; if it turns out to be inconsistent with the graph then we undo the propagation, unassign the value and delete it from the domain. We move on to try a different value from the domain until we find a value that is consistent or exhaust the domain. If we find a value that is consistent it is pushed into the assigned stack and the next variable is picked up. If no consistent value is found for that variable then we return false. Each variable is a pair of operations that both require the same resource. The domains of each variable contain a maximum of two values, one for each possible ordering of the pair of operations.

In the unlabel function we remove the inconsistent value for the variable, undo the propagations for that value and update the domain for that variable.

2.4 Variable Ordering Heuristic

A variable is a set of operations *i, j* that share a resource need. We discussed four possible cases in the ordering of two operations in Section 2.2. The variable ordering heuristic helps select what orderings are to be dealt with first. Cases 1 and 2 are picked first and then Case 4. Case 1 and 2 can be resolved directly by choosing the only possible corresponding ordering. Case 4 needs to be handled differently: we use temporal slack to decide among the different possible variable pairs.

The slack when ordering *i* before *j* is defined as $lst_j - (est_i + p_i)$ and when ordering j before i is $lst_i - (est_j + p_j)$. The minimum overall slack between these two decisions is the minimum between the two slack values.

There is another improved version of the slack heuristic called the BSlack heuristic. In this we calculate *a* and *b* as the minimum and maximum values of the slack between the decision to have *i* precede *j* and *j* precede *i*. Then we calculate *S* as (*a/b*). Then $Bslack(i \rightarrow j) = Slack(i \rightarrow j)(S\text{-}1/a + S\text{-}1/b)$ and $Bslack(j \rightarrow i) = Slack(j \rightarrow i)(S\text{-}1/a + S\text{-}1/b)$.

Finally, we also considered a slightly different version of BSlack, which we call B2Slack, which is that given in the original Smith & Cheng paper [1]. Instead of using *a* and *b* as the negative inverse powers of *S*, we simply use constants (we used the values 2 and 3, which Smith & Cheng report to be the best pair).

2.5 Value Ordering Heuristic

In case 4 we need to choose between the two potential orderings, $i \rightarrow j$ and $j \rightarrow i$. We can use the Slack values of the two potential orderings as a heuristic for this, by taking the ordering with the highest slack (or alternatively the lowest). Between the variable and the value ordering heuristics we should be able to solve most problems with few backtracks.

2.6 Constraint Propagation

We do constraint propagation when we choose a particular ordering in the label function. We propagate the choice through the graph to see if it holds consistently through the graph. This can be done in two ways. One is by continuing

the Bellman-Ford algorithm after each new operation ordering is chosen. The second way is to calculate the new est and lst values for the new ordering and then propagate the est forward through the outward edges defined by the ordering constraints and lst backward through the inward edges defined by the ordering constraints. For example consider choosing $i \rightarrow j$ then $estj = max(estj, esti + pi)$ and $lsti = min(lsti, lstj - pi)$. We then propagate *estj* forward through constraints $j \rightarrow k$ and propagate *lsti* backward through constraints $k \rightarrow i$. An inconsistency is detected when $esti > lsti$ for any i.

3 Optimizations

Caching and other optimizations are essential for creating usable job shop scheduling solvers. Like our previous projects' solvers, we developed our algorithms with efficiency in mind. Our optimizations include managing our own memory for most data structures (no STL Vectors), quick lookup structures for variables and operations, intelligently updating slacks and cases, tuning the temporal reasoner, random restarts, and several other memory allocation tweaks. Unless otherwise noted, all results in this section used our experimental method as detailed in Section 4

3.1 Memory Management

As we have shown with our previous solvers, using our own memory management techniques through our superArray class instead of the C++ Standard Template Library (STL) Vector class can result in a measurable performance gain in the actual problem domain. We again chose to use our superArray class, and will only provide a brief overview of the discussion from our previous two projects.

As a reminder, through testing with our SAT and CSP solvers we found that our superArray is 1.4 times faster at reads and 1.7 times faster at writes than the Vector Op method and 5.5 times faster at reads/writes than the Vector At method. (The Vector At method includes out-of-bounds checking, while superArray and Vector Op do not.) Table 3.1a summarizes these results.

Table 3.1.a: TestVector.h Computation Time Relative to superArray

	Reads	Writes
superArray	1.0	1.0
Vector Op	1.4	1.7
Vector At	5.5	5.5

In our previous crossword puzzle domain, we were able to calculate the real-world performance gains of using superArray through a preprocessor command called SUPERARRAY_TYPE in data_structures.h of our source that effectively gave us the ability to decide at compile time whether to use the Vector Op method or our superArray method. In the crossword puzzle domain we showed that our superArray is 1.02 times faster than using Vector Op. Using our read/write results from Table 3.1a we were also able to estimate that our superArray would be 1.18 times faster than the Vector At method. We unfortunately could not use our preprocessor method to obtain exact results of using Vector At because the switch took advantage of the fact that the superArray and Vector Op methods both use brackets [ ] to access the array members. These results are summarized in Table 3.1b.

Table 3.1.b: Crossword CSP Computation Time Relative to superArray

superArray	1.0
Vector Op	1.02
Vector At	1.18*

*Estimated using relative read / write times from Table 3.1.a.

Since we developed our schedulers using the superArray class, we can again measure real-world performance gains over using Vector. To add to the usefulness of this exercise, and to increase our productivity, we also

measured the real-world performance gains over using a method similar to Vector At. We added an option to SUPERARRAY_TYPE that allows us to use the superArray class with out-of-bounds checking. This not only allows us to measure the real-world performance hits brought about by out-of-bounds checking, but it also eliminated many development bugs and increased productivity because we could pause the program in our debugger whenever an out-of-bounds error occurred.

We measured performance by turning all heuristics off, turning random restarts off, setting the max time to ten seconds, and executing the solver with each of the various settings on problem 3 (sadeh3.fol). With these settings, the solver was not be able to find a solution before the timeout. For each setting of SUPERARRAY_TYPE, we executed the solver using ten different seeds and we recorded the amount of times that the solver attempted to label a variable. Since we used the same ten seeds for all three settings of SUPERARRAY_TYPE, the solvers made the same decisions in the same order and the only difference was speed.

Table 3.1.c shows the results of this experiment. superArray is again 1.02 times faster than using the Vector Op method. superArray is 1.16 times faster than using superArray with out-of-bounds checking, which is similar to our estimated 1.18 times faster in the crossword puzzle domain.

Table 3.1.c:
JSS Computation
Time Relative to superArray

superArray	1.0
Vector Op	1.02
Vector At*	1.16

*Actually superArray with out-of-bounds checking.

Since we do not require array resizing, the only advantage of using Vector is the out-of-bounds checking in At. Now that we have included that functionality into our SUPERARRAY_TYPE switch, we have no real reason to use Vector. In fact, we are much better off using superArray because we can use out-of-bounds checking during development and easily strip the out-of-bounds checking from the program when compiling for a release.

3.2 Quick Lookups

Data structures are critical for optimizing any algorithm. In the job shop scheduling domain, it is critical that our algorithms can loop through all of the operations that have an incoming or outgoing edge with respect to some given operation. Our algorithms also require an efficient method for looping over the set of variables that contain a given operation. To handle these needs, our Operation class includes the arrays outEdges, inEdges, and variablesIn, which contain the operations that must happen before the given operation, the operations that must happen after the given operation, and the variables that the given operation exists in. The importance of these data structures will become apparent in the remaining sections of Section 3.

3.3 Updating Slacks and Cases

Inefficient slack and case updates for the variables can have an enormous impact on speed. After each propagation while labeling and after undoing a propagation while unlabeling, one must update the slack values and cases of the variables. Through rough experiments we found that naively recomputing all slacks/cases drastically decreases performance.

While propagating an ordering choice, we keep track of all of the operations that have a change in earliest start time or latest start time. If the propagation did not find an inconsistency, we then need to update all of the slack values. We do this by using the data structures in Section 3.2 and only recomputing the slacks and case values for variables that contain an affected operation. We found that this small change makes the overall program 1.05 times

faster than recomputting the slacks/cases for all unlabeled variables.

We use the slack values to calculate the case values. If we are using an additional heuristic such as Bslack then we only update the Bslacks if the variable is in case 4 because we only use the heuristic when comparing case 4 variables.

If any of the variables are found to be in case 3 then we immediately unpropagate and treat the labeling attempt as a failure. We do not do this by moving to the unlabel function. Instead, we treat it in the same way as if the propagation had failed (we simply remove the value from the variable's domain). This single optimization, coupled with the random restarts in Section 3.6, makes our algorithm fast enough to solve all 60 problems with all heuristics off in a total cumulative time of around 35 seconds. Removing either this optimization or the random restarts makes the solver spend much more time searching for a solution.

3.4 Memory Allocation and Quick Insertions

As we noticed when creating our SAT and CSP solvers, memory allocation can be costly. To reduce the amount of memory allocation required during the search, we allocate all data structures before starting the label/unlabel loop. This includes allocating enough space in the inEdges and outEdges arrays of each operation so that each operation can potentially store all other operations in each array without resizing. When adding an edge to an operation during propagation, we simply treat the edge arrays as stacks and push the new operation onto the top by using the current length of the array as the index into the array. When removing an edge while undoing a propagation, we simply pop the operation off of the stack by decrementing the array's length variable. We also preallocate all arrays that are used by the propagator to store the changed start times during any given propagation. Additionally, we make every attempt to use pointers and not invoke a copy constructor anywhere in the solver's code.

3.5 Temporal Reasoning

3.5.1 Temporal problem transformation

In order to even begin solving the job shop scheduling problem, we first transform it into a temporal scheduling problem and determine the earliest and latest possible start-times for each of the operations by running a shortest-path algorithm on the distance graph. For this initial transformation we consider the *length*, *before*, *release*, and *due* job shop constraints, but not the *needs* constraints (which are the primary consideration of the subsequent search phase).

As the temporal-problem distance-graph based approach uses time points (and not time intervals) as variables, we initially used as variables:

- the start time of each operation
- the end time of each operation
- the time origin

This naïve approach results in 2*n+1 variables (where n is the number of operations). However, since the job shop problems specify a length for each operation, it is clear that having separate variables for the start and end of each operation is redundant (indeed, the arcs between the start and end variables for each operation in our naïve implementation were actually just specifying that end = start + length).

We can therefore safely remove all the end-time variables, practically halving the number of variables and removing the *length*-constraint arcs (of course, the length constraints are implicitly contained in the other values of the other arcs).

3.5.2 Bellman-Ford optimization

As for every graph-based algorithm implementation, a key factor for the performance of our Bellman-Ford implementation is the choice of the data structure used to represent the graph. For obvious efficiency reasons, we represented the nodes as integers indexed from 0, which allows us to index the node directly into C arrays.

To capitalize on this we naively represented the edge set as an n*n 2-dimensional array (where n is the number of nodes). The drawback to this approach is that to iterate over all edges, one actually has to iterate over every cell in the matrix, and check if there is an edge or not for each cell. This is clearly inefficient if the edge matrix is at all sparse.

To improve on this we changed the edge set representation to just be a stack of *edge* structs, which obviously allows us to iterate over all the edges in time linear in the number of edges. As this is the only type of access Bellman-Ford makes to the edges, this data structure is "optimal" in time complexity for the case at hand. This change yielded a 4x speed-up in our over-all system (when using Bellman-Ford for constraint propagation as well), which strongly justifies post-hoc the optimization.

We further optimized the algorithm by stripping out all elements not relevant to our usage: in particular, as we never need to actually generate the shortest paths themselves, the Π array of back-pointers is useless here.

3.5.3 Constraint propagation in the distance graph

After the initial earliest/latest start-time determination step, the Bellman-Ford component can also be used as a sub-routine for constraint propagation in the search phase. The simplest way to do this is simply to add an edge corresponding to the new constraint to the distance graph, and then run Bellman-Ford from scratch to determine the shortest paths. This is obviously extremely slow and inefficient.

We can improve on this by observing that adding a new edge to the distance graph can only decrease shortest path distances, as all the former shortest paths are still valid. Therefore we can add the new edge and just continue where the last run of Bellman-Ford "left off" (i.e. starting off with the previously generated d array). Empirically this yields roughly a 10x speed-up relative to fully re-running Bellman-Ford. However, a brief theoretical analysis reveals that the speed-up is very dependent on the sparsity of the graph and the position of the new edge relative to source. And in the worst case, with a cheap new edge adjacent to the source, this method degrades to fully re-running Bellman-Ford. We suspect that there is a more sophisticated and efficient way of propagating constraints in the distance graph, and will look further into this in the future.

Of course, another problem with this approach is how to efficiently "unpropagate" a formerly propagated constraint. The most basic solution is simply to remove the corresponding edge from the distance graph and fully re-run Bellman-Ford. However, fundamentally we simply want to restore the "state" of Bellman-Ford to what it was before the bad constraint propagation. And the state of Bellman-Ford is nothing more than the *d* shortest-path-distance array. We therefore solved the problem in a simple time-efficient memory-inefficient manner: whenever we propagate a constraint, we back up the former *d* array onto a stack. Then if we want to unpropagate a constraint, we just remove the corresponding edge, pop the former *d* array from the stack and set it back into Bellman-Ford's state.

3.6 Restart Timeouts

We found that many of our algorithms were susceptible to bad luck on their initial seed values for the random number generator. We randomize our choices by randomly shuffling the array of variables before entering the label/unlabel loop and after a restart is triggered.

We found that implementing continual, quick restarts significantly decreases the computation time required to find schedules.

An obvious insight from these observations is that periodically restarting with a reshuffled ordering in which equally good variables are chosen for labeling could drastically improve overall performance. Similar to our restart timeouts for our CSP solvers, we implemented an increasing restart timer that allows the algorithm additional time between restarts as the number of restarts increases. We found that rapid restarts worked best.

Thus all of our solvers have the execution length of the first run set to one second, with a multiplier of 1.025. The first run is allowed to execute for 1.0 seconds, the second run for 1.025, the third for 1.05, and so on. Unlike our CSP solvers, we did not include a method for ensuring deterministic behavior given the same initial seed. We did not include this feature because in our previous project we found it to be hindering in our CSP solver performance.

3.7 Variable Ordering

When not using any heuristic for choosing which variable to label next, one would generally think that choosing all case 1 and case 2 variables before case 4 variables would be an obvious smart selection choice. We found that with our solvers this is not the case. Quite surprisingly, choosing the first variable in the array of unlabeled variables performed better than the previously mentioned selection criterion.

We compared the two selection methods on Problem 3 (sadeh3.fol), which is one of the more difficult problems for our solvers to solve. We ran both selection methods with 10 different seeds, using the random restart settings described in Section 3.6 and using no special heuristics. Neither solver experienced a timeout. In terms of the average CPU time for each solver averaged over all 10 seeds, returning the first variable was 40 times faster than choosing all case 1 and case 2 variables before considering case 4 variables.

The reason for this massive speed difference might be that we do not maintain separate sets for the unlabeled variables. We keep a single array that contains all unlabeled variables. Thus in order to ensure that we choose all case 1 and case 2 variables first, we always have to loop through the entire array of unlabeled variables. This loop could be the cause of the slowdown.

To remedy the situation, we could maintain two sets of unlabeled variables and move the variables amongst the sets when their slacks/cases are updated. If we had separate sets then we would not need the loop. Unfortunately, we could not pursue this further due to time constraints.

4 Experimental Method

Our experimental method is almost identical to our previous projects. We conducted all of our experiments on the Pod cluster. The Pod cluster contains Dell Precision 390s, each with a 2.4 GHz Core 2 Duo and 2 GB of RAM running Ubuntu 7.04. All results in Section 5 are from running each solver on the 60 scheduling problems using 10 different seeds for the random number generator. We used the same seeds as in our CSP solvers. These seeds were chosen by seeding the random number generator with the clock's time and then printing out 10 random numbers. The readme included with the source describes how to specify your own seed or which of our 10 seeds to use.

4.1 Naming Conventions

In all results we refer to our constraint-graph propagator as C and our distance graph propagator as BF. Our variable ordering heuristics include NONE, BSLACKL, B2SLACKL, MOSL, BSLACKG, B2SLACKG, and MOSG. Our value ordering heuristics include NONE, SG, SL, BSG, BSL, B2SG, and B2SL, with SG and SL using the slack values, BSG and BSL using the Bslack values, and B2SG

and B2SL using the Bslack values with parameters of 2.0 and 3.0 as described in the Smith and Cheng 1993 paper. An "L" at the end of a name indicates that we are minimizing the value while a "G" indicates that the heuristic is maximizing the value ("L" and "G" were chosen due to the use of the less than and greater than operators). The naming convention for our solvers is var?-prop?-val? where the question marks are replaced with a propagator or heuristic from the previous lists.

Through all of our experiments we gathered results for all combinations. We imposed a timeout limit of 30 seconds for each scheduling problem. Our best solvers were able to solve all 60 scheduling problems within the 30 second timeout constraint for all seed values. Several solvers were even able to solve all 60 problems in a cumulative time of 30 to 35 seconds. Additionally, we ran all solvers that experienced timeouts from this initial run a second time with the same seeds and a 10 minute timeout limit.

4.2 Our Performance Metrics

We track seven metrics: Die EST Counter, Die LST Counter, Label Counter, Unlabel Counter, CPU Time, Real Time and Restart Counter. Die EST Counter and Die LST Counter record the amount of times that a propagation detects an inconsistency while propagating the earliest start times or latest start times respectively. Label Counter is the amount of times we attempt to label a variable and Unlabel Counter is the amount of times we unlabel a variable. Restart Counter is the number of times the solver restarts the search and reshuffles the ordering of the variables in the unlabeled variables array.

The CPU Time and Real Time are both in seconds. Since our CPU timer is less reliable for runtimes close to zero, we record the Real Time for both CPU Time and Real Time when the reported CPU Time is less than one. CPU Time and Real Time do not include the processing time for reading in the problems, but do include the time spent preallocating the solver's memory. The timing starts when the function Scheduler::scheduleProblem is called.

5 Progressive results analysis

As we progressively improved our algorithms' performance, we juggled several different decision factors and were continually re-evaluating our performance to identify potential improvements. For example, our introduction of a random restart mechanism was a reaction to the extremely high variance in our run-times and success rates. In order to avoid an excessively long narrative, we shall evaluate in this section certain key areas of the performance and behavior of our final implementation.

5.1 Constraint propagation methods

In our earlier naïve implementations, we avoided performing proper constraint propagation by simply running Bellman-Ford from scratch in the distance graph after each variable assignment. Obviously this resulted in atrocious performance. Subsequently, we employed two constraint propagation methods: propagation in the distance graph through continued Bellman-Ford, and propagation in the constraint graph following the rules discussed in class.

While the modified Bellman-Ford approach is better than nothing, and benefits from our optimizations to our Bellman-Ford implementation, it is dominated by the constraint graph propagation approach (referred to as "propC" in the following results). Indeed, the two approaches give the exact same final results, but our evaluation showed that the constraint graph method is approximately 23 times faster than the distance graph method. This explains why in all subsequent results we will primarily use constraint graph propagation.

5.2 Variable ordering heuristics

The most significant question mark in our minds when we started evaluating our algorithms was on the effect of the variable ordering heuristics. Indeed, Dynamic Variable Ordering proved hugely effective for CSP solving, so Case 4 variable ordering with a good heuristic promised to be effective for JSPs as well.

We evaluated three heuristics: regular Slack (referred to as MOS in the results), BSlack, and B2Slack (see Section 2.4). We tried using each of these heuristics in two ways, taking the variable with either the maximal or minimal heuristic value. For comparison purposes we also computed results using no heuristic. Results using Greater Slack value ordering (in blue) and no-heuristic value ordering (in red) with a 30 second time-out follow on the graph below.

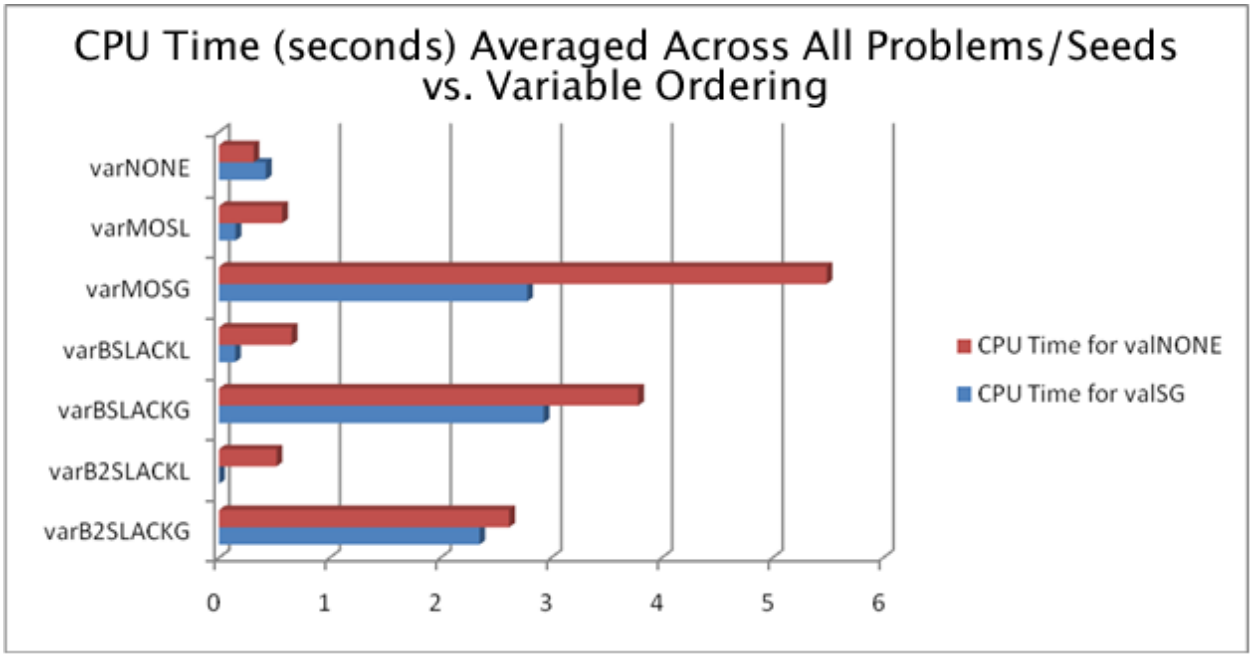
An obvious first conclusion is that all of the "Greatest" heuristic variable choosing methods (MOSG, BSLACKG and B2SLACKG) perform extremely poorly, greatly under-performing the baseline no-heuristic algorithm.

The effect of the "Least" heuristic variable choosing methods is less clear-cut. It is notable that in the results represented in the graph above, all three "intelligent" heuristics (MOSL, BSLACKL, B2SLACK) are actually out-performed by varNONE when no value ordering heuristic is used. However, when the smarter valSG value ordering heuristic is used, all three of these heuristics prove their worth. To better compare the heuristics, let us look at some specific CPU time figures averaged across all problems/seeds:

	Avg CPU Time for valSG	Avg CPU Time for valNONE	Avg CPU Time for valSL
varB2SLACKL	0.001673843	0.51964482	0.381577602
varBSLACKL	0.145988742	0.657223548	0.348550018
varMOSL	0.149682745	0.570096953	0.314295552

The most disappointing result is definitely that of BSlack: it pretty much consistently fails to even perform as well as regular Slack. This could conceivably be an artifact of our results runs or our implementation, but at least for our current implementation BSlack does not provide the performance that we had hoped for.

B2Slack, on the other hand, gives scorching performance when coupled with certain value ordering heuristics (such as valSG). To get a better idea of this performance, let us look at the maximal values across all 60 problems and 10 seeds of certain metrics for these three solvers on the next page.



	Label Counter	Unlabel Counter	CPU Time (seconds)	Restart Counter
varB2SLACKL-propC-valSG	225	0	0.004372	0
varBSLACKL-propC-valSG	920,882	483,735	15.79	13
varMOSL-propC-valB2SG	651,003	381,450	9.99	9

We note that MOS and BSlack can solve every problem in around 10 seconds / restarts. But B2Slack solves every problem in time on the order of the millisecond. This is clearly because the algorithm basically never has to unlabel. It would therefore seem that B2Slack variable ordering, coupled with valSG or valB2SlackG (see results in Annex) value ordering, is at the very least extremely well tailored to this class of job shop problem. The heuristic apparently finds a nigh-on perfect ordering of variables, thus avoiding backtracking.

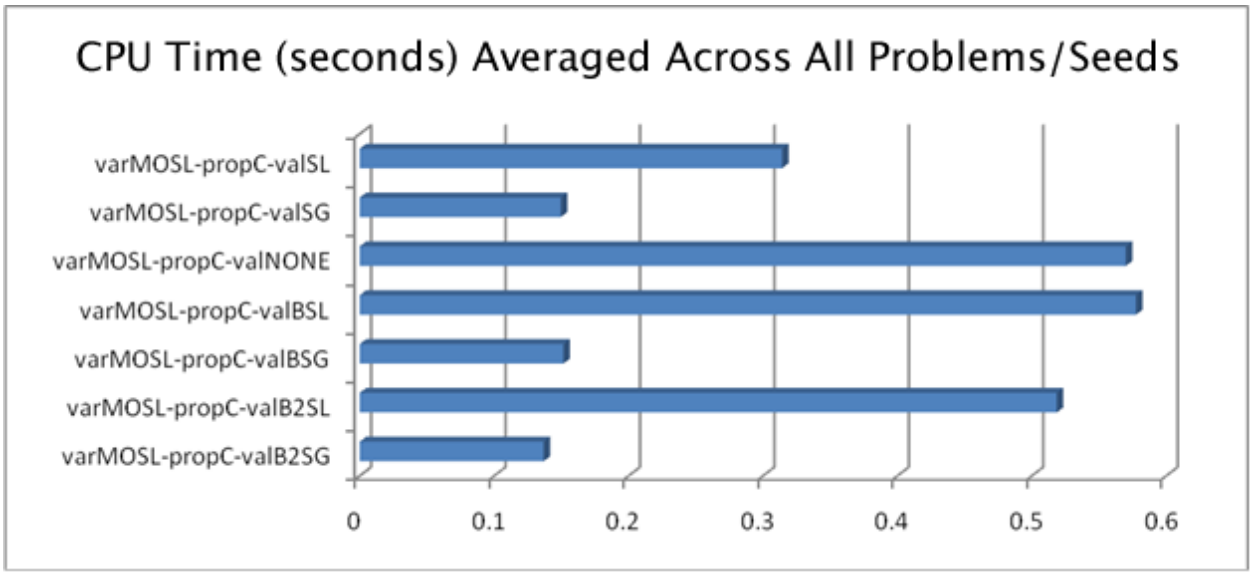
As an aside note, to do full justice to our heuristics we re-ran all the algorithms which had had time-outs with a 30-second time limit with a full 10-minute time limit. The results are included in the Annex, but the primary conclusion is that whatever the value ordering used, all of our algorithms excluding the "Greater" heuristic value variable choosing methods were able to finish every problem in less than 105 seconds. The "Greater" heuristic value variable choosing methods still timed out in most cases on some of the problems.

5.3 Value ordering heuristics

While we were initially primarily concerned with the use of our slack-based heuristics for variable ordering, they can of course equally well serve as heuristics for ordering the two potential values of case 4 variables. To evaluate the performance effects of the heuristics for value ordering, we fixed the variable ordering heuristic (to MOSL) and computed average run-times. The graph below synthesizes the results.

The primary observation is that the effects of the "Greater" and "Lesser" heuristic value choosing methods are reversed relative to the variable ordering case: for values, it is far better to choose the value with highest slack. This is not unsurprising, as having more slack means more available time between the pair of operations of the variable, which intuitively indicates less chance of having to backtrack.

There is actually very little to distinguish performance-wise between the Slack, BSlack and B2Slack value ordering heuristics. They are virtually identical in performance. This is understandable: unlike having to choose a variable from hundreds of possibilities, one has a maximum of two possibilities for picking a value, and the three heuristic functions are all reasonably similar.



5.4 Random restarts

We observed massive variance in run-time for different random seeds in our initial algorithm implementation. Indeed, the exact same algorithm could time out on a problem, and then solve the same problem in seconds on the next run. This was both a serious problem and a great opportunity. We hypothesized that we could leverage this variance by using a restart mechanism to both decrease variance and improve average run-time.

Our random restart method is described in Section 3.6. To isolate and evaluate its impact on over-all average performance, we ran one of the weaker versions of our algorithm (with no variable or value heuristic usage) on the full problem set with 10 seeds, with and without random restarts. Here are the (average) results for a 60 second timeout:

	Label Counter	Unlabel Counter	CPU Time (seconds)	Std Dev of CPU Time Normalize by Avg
varNONE-propC-valNONE-NORESTART	462,050	282,410	1.63	4.21
varNONE-propC-valNONE-RESTART	75,125	44,848	0.319	2.89

These figures clearly show that our restart mechanism has a huge impact on run-time performance, improving speed by almost a factor of 10. This is obviously due to a corresponding proportional decrease in the number of label and unlabel operations.

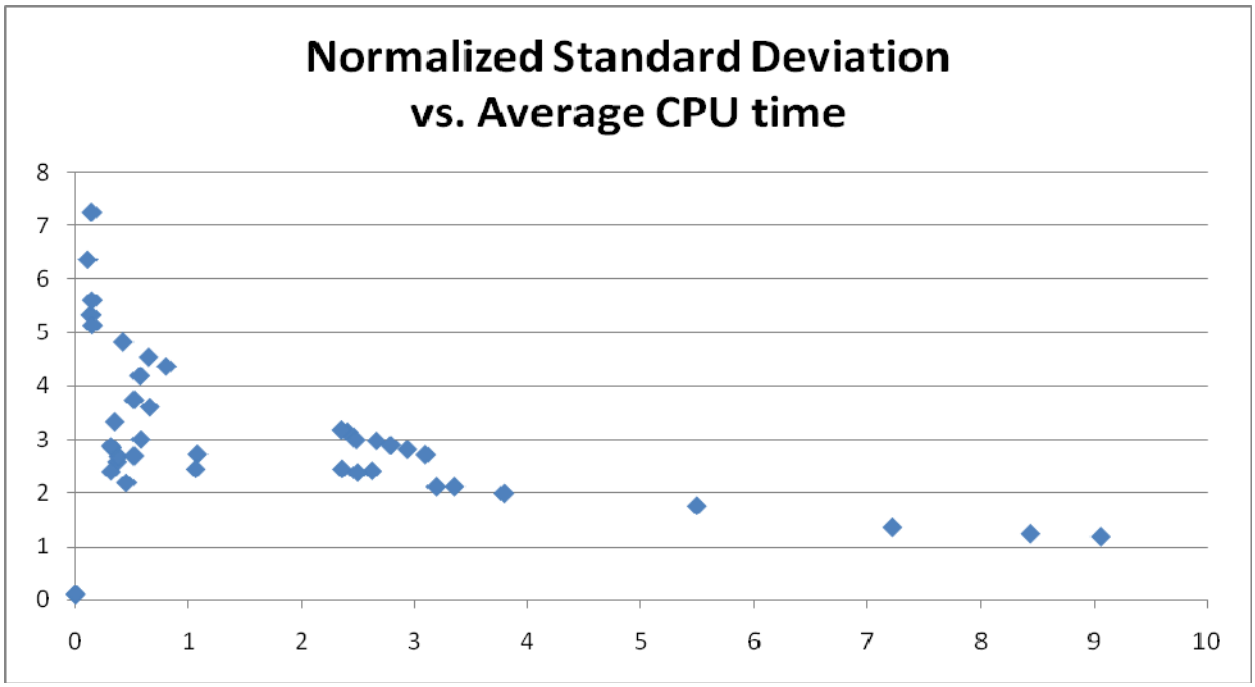
Furthermore, these averages do not show the effect of restarts on the number of problems solved. Indeed, on this particular run the varNONE-propC-valNONE-NORESTART algorithm timed out on 5 of the 60 problems, while varNONE-propC-valNONE-RESTART finished them all. These results are unsurprising given how widely our JSS's performance varies depending on the initial randomization. Our restart mechanism therefore strongly mitigates the "luck" factor in solving the problems.

5.5 Variance

The random restart mechanism clearly has a strong smoothing effect on the variance of our algorithms' run-time performance. To go beyond this basic initial analysis, and given that we now always use the restart mechanism, we computed the standard deviation of CPU time for all our algorithms. To account for the fact that the different algorithms have very different expected run-times, we normalized the standard deviation relative to the expected value.

The full results are given in the Annex, but the analysis is complex. Indeed, it is hard, and probably illusory, to discern "winners" and "losers" in the variance figures, and no heuristics stand out as having a particular consistent effect. To evaluate the over-all trend of variance relative to average performance, consider the scatter plot on the top of the next page, which plots average CPU time in seconds on the x-axis and normalized standard deviation on the y-axis.

Our more pattern-minded readers will note that the Normalized Standard Deviation pretty much follows a hyperbolic curve relative to the average run-time, irrespective of the nature of the algorithm and heuristics. This would seem to indicate that our random restart mechanism has smoothed out all important variance distinctions between our algorithms. The two notable outliers and exceptions to this are visible at the origin of the plot: our two best-performing algorithms (varB2SLACKL-propC-valB2SG and varB2SLACKL-propC-valSG), are so good at all the problems (basically never unlabeling) that they also have very low variance.



6 Future work

6.1 Smarter backtracking

In our current search implementation we use the most primitive form of chronological backtracking. Just as regular CSP solvers can be greatly improved by upgrading chronological backtracking to say Conflict-Directed Backjumping, our job shop solver could potentially also greatly benefit from more sophisticated dependency-directed backtracking.

Advanced backtracking techniques are discussed in Xiong, Sycara & Sadeh [2]. They propose three backtracking related methods:

- Dynamic Consistency Enforcement (DCE): a selective dependency-directed scheme that dynamically focuses its effort on critical resource sub-problems.
- Learning From Failure (LFF): an adaptive scheme that suggests new variable orderings based on earlier conflicts.
- Heuristic Backjumping (HB): a scheme that gives up searching areas of the search space that require too much work.

The authors judge that a complete dependency-directed backtracking scheme would be too expensive computationally, and thus use DCE, which only checks consistency on subgroups of operations which they call "Dangerous Groups". The determination of these Dangerous Groups is done dynamically, and is an integral part of the algorithm.

Given the size of our problems (and the increase in computational power since 1992!), it seems likely that a full dependency-directed backtracking scheme is feasible in our case. By analogy with the regular CSP CBJ case, we need to maintain a *conf-set* for each variable, which is the set of past levels with which the variable x_{ij} conflicts with.

For our unlabel function, we can act just as in regular CBJ: we simply backjump to the the most recently assigned variable *h* in the current variable *i*'s conf-set, empty the conf-sets of the variables in between (and restore their domains), and add the variables in *i*'s conf-set to that of *h*.

For the *label* function the changes are not so obvious, as it is not trivial to determine which previously assigned variable is responsible for an inconsistency with the current variable. It seems that it would be more effi-

cient to do an FC-CBJ type approach to filling in the conf-sets: whenever we perform constraint propagation, we work out which variables are having their domains reduced (from 2 to 1, otherwise we would get an inconsistency anyway), and add the variable currently being assigned to these variables' conf-sets.

There are two possible ways to work out the variables whose domain is affected by the current assignment:

- During the constraint propagation, by considering, at each operation visited by the propagation, the set of variables linked to this operation, and evaluating whether each of these variables is going to go from case 4 to case 1/2 (in which case we add the freshly assigned variable to that variable's conf-set).
- After the end of the constraint propagation, by simply considering every unassigned variable and seeing whether it has switched from case 4 to case 1/2.

Both methods would imply a lot of extra work/time, but they would allow us to identify the root cause of a conflict further down the line. Due to the speed at which our best solvers can solve our given scheduling problems, we believe that any smart backtracking method would spend too much time maintaining data structures and would not make up for the lost time in significant backjumps. Nevertheless, the methods would probably be helpful in more difficult scheduling problems.

6.2 Bslack

Another area which we wish to look into is experimenting with the B2Slack heuristic function. Indeed, the success we had with B2Slack heuristic variable ordering with just the base constants provided in Smith & Cheng's paper indicates that even better performance could potentially be obtained by tuning the function's parameters. A basic hill-climbing approach (or at the very least grid search) could easily determine "optimal" values for the constants a and b.

6.3 Case 1/2 & 4 Sets

As noted in section 3.7, we do not maintain separate arrays for the unlabeled variables of each possible case. Maintaining separate arrays instead of a single array could potentially increase performance by decreasing the amount of time required to determine if a case 1/2 variable exists.

7 References

[1] S. Smith and C. Cheng, "Slack-Based Heuristics for Constraint Satisfaction Scheduling," Proceedings 11th National Conference on Artificial Intelligence, July, 1993.

[2] Y. Xiong, K. Sycara, and N. Sadeh-Koniecpol, "Intelligent Backtracking Techniques for Job Shop Scheduling," Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, October, 1992, pp. 14-23.