Job-Shop Scheduling in Localizer

Laurent Michel and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-98-03
April 1998

Job-Shop Scheduling in LOCALIZER[1]

Laurent Michel and Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912
{ldm,pvh}@cs.brown.edu

Abstract

LOCALIZER is a domain-specific language for developing local search procedures. By automating many of the tedious tasks underlying local search algorithms, LOCALIZER was shown to reduce the development time significantly and to preserve most of efficiency of specialized algorithms on applications such as Boolean satisfiability and graph coloring. This paper applies LOCALIZER to job-shop scheduling to provide evidence that it can scale up to more sophisticated applications. It presents two LOCALIZER model: an array-based model and a model based on graph-theoretic concepts. This last model illustrates how advanced data types make the statement even closer to the application, while not degrading performance. Both models find optimal or near optimal solutions to standard scheduling benchmarks quickly (from 20 to 90 seconds) on a modern workstation. The underlying implementation techniques are also described.

1 Introduction

LOCALIZER [1] is a domain-specific language for the implementation of local search algorithms, combining aspects of declarative and imperative programming, since both are important in local search algorithms. LOCALIZER offers support for defining traditional concepts like neighborhoods, acceptance criteria, and restarting states. In addition, LOCALIZER also introduces the declarative concept of *invariants* in order to automate the most tedious and error-prone aspect of local search procedures: incremental data structures. Invariants provide a declarative way to specify what needs to be maintained to define the neighborhood and the objective function. LOCALIZER was shown to produce significant reduction in development time for applications such as satisfiability and graph-coloring, while inducing only a reasonable overhead over special-purpose implementations.

The goal of this paper is to provide some evidence that LOCALIZER can scale up to more sophisticated applications. It considers job-shop scheduling as a case-study and investigate how a good local search algorithm could be implemented in LOCALIZER. The selected algorithm uses a neighborhood that swaps arcs on critical paths and a tabu search strategy. This algorithm quickly produces near-optimal solutions to standard scheduling benchmarks. The paper presents two models for job-shop scheduling based on this algorithm. The first algorithm uses traditional data structures over arrays and sets and maintains recurrence relations incrementally. It is short and compact, yet it compares well in efficiency with specialized algorithms. The second model uses recent extensions of LOCALIZER that supports graphs as a primitive data type. It shows that high-level data structures may improve the expressive power of LOCALIZER, while not degrading performance, indicating a promising direction for LOCALIZER, and constraint programming in general. Both of these models require significant generalizations of the LOCALIZER implementation, which are also discussed.

The rest of this paper is organized as follows. Section 2 is a brief overview of LOCALIZER. Section 3 reviews the job-shop scheduling problem and the local search algorithm considered in this paper. Sections

[1] This research was partly supported in part by an NSF NYI Award.

4 and 5 present the two models. Section 6 discusses the implementation and Section 7 reports preliminary experimental results. Section 8 concludes the paper.

2 A Brief Overview of LOCALIZER

To understand statements in LOCALIZER, it is best to consider first the underlying computational model. Figure 1 depicts the computational model of LOCALIZER for decision problems. The model captures the essence of most local search algorithms. The algorithm performs a number of local searches (up to **MaxSearches** and while a global condition is satisfied). Each local search consists of a number of iterations (up to **MaxTrials** and while a local condition is satisfied). For each iteration, the algorithm first tests if the state is satisfiable, in which case a solution has been found. Otherwise, it selects a candidate move in the neighborhood and moves to this new state if this is acceptable. If no solution is found after **MaxTrials** or when the local condition is false, the algorithm restarts a new local iteration in the state *restartState(s)*. The computation model for optimization problems is similar, except that line 5 needs to update the best solution so far if necessary, e.g. in the case of a minimization,

```
5      if value(s) < bestBound then
5.1        bestBound := value(s);
5.2        best := s;
```

The optimization algorithm of course should initialize f^* properly and return the best solution found at the end of the computation.

```
procedure LOCALIZER
begin
 1  s := startState();
 2  for search := 1  to MaxSearches while Global Condition  do
 3      for trial := 1  to MaxTrials while Local Condition  do
 4          if satisfiable(s)  then
 5              return s;
 6          select n in neighborhood(s);
 7          if acceptable(n)  then
 8              s := n;
 9      s := restartState(s);
end
```

Figure 1: The Computation Model of Localizer

The purpose of a LOCALIZER statement is to express the generic parts of the computational model, e.g., the neighborhood, the acceptance criteria, and the parameters. LOCALIZER offers a variety of constructs to simplify these specifications. Perhaps the most fundamental tool offered by LOCALIZER is the concept of invariants that maintain sophisticated data structures incrementally. Invariants make it possible to specify, in a declarative fashion, the data structures needed to define the neighborhood concisely without having to be concerned with how to maintain them incrementally.

3 A Local Search Algorithm for Job-Shop Scheduling

This section specifies the problem to solve, introduces the main concepts and notations, and presents the local search algorithm used as a case-study in this paper.

3.1 Job-Shop Scheduling

A job-shop scheduling problem consists of a set of jobs. Each job is a sequence of tasks (e.g., the first task in a job must precede the second task and so on) and each task executes on a given machine. Tasks executing on the same machine cannot overlap in time. The job-shop scheduling problem amounts to assigning starting dates to the tasks satisfying the precedence and non-overlapping constraints and minimizing the makespan, i.e., the maximum duration of all jobs.

It is well-known that solving a job-shop problem mostly consists of determining an optimal ordering for the tasks on the various machines. Such an ordering in fact reduces the job-shop application to a PERT problem in a graph where each vertex corresponds to a task and each arc expresses a precedence constraint. In addition, the graph contains a source vertex that precedes the first task of every job and a sink vertex that follows the last task of every job. This graph is called a *solution graph* in the rest of this paper and its arcs are of two types: the *precedence arcs* that express the precedence constraints within a job and the *machine arcs* that express the precedence constraints induced by the chosen ordering. The weight of an arc is simply the duration of the task corresponding to the source of the arc (with the convention that the sink and the source have durations zero). Of course, the precedence edges are fixed. Typically, a PERT algorithm computes, among other things, the earliest starting dates for each task. These earliest starting dates provide a solution to the job-shop scheduling problem.

The local search algorithm described later in this paper applies local transformations to solution graphs. It is thus useful to review a number of notations and concepts on solution graphs. These notations assume that the source and the sink execute (first and last) on all machines and are the first and the last tasks of all jobs, which is not restrictive since they have duration zero. The duration of task t is denoted by $d(t)$ and we assume that the problem has N tasks ($N+2$ when the source and the sink are included). Tasks are identified by integers and the source and the sink are numbered 0 and $N+1$ respectively.

Every task t in a solution graph (except the sink and the source) has two predecessors: a predecessor for the job, denoted by $pj(t)$, and a predecessor for the machine, denoted by $pm(t)$. Similarly, every task t (except the source and the sink) has two successors: a successor for the job, denoted by $sj(t)$, and a successor for the task, denoted by $sm(j)$. The release date of a task t, denoted by $r(t)$, is the longest path from the source to t. The tail of a task t, denoted by $q(t)$, is the longest path from t to the sink. Intuitively, the tail of a task represents what remains to be done, once t is released. The release dates and the tails can be computed by simple recurrences:

$$r(t) = \begin{cases} 0 & \text{if } t = 0 \\ max(r(pj(t)) + d(pj(t)), r(pm(t)) + d(pm(t))) & \text{otherwise} \end{cases}$$

$$q(t) = \begin{cases} 0 & \text{if } t = N+1 \\ max(q(sj(t)) + d(t), q(sm(t)) + d(t)) & \text{otherwise} \end{cases}$$

Of course, the release date of the sink is the makespan of the solution. Critical arcs play a fundamental role in the local search algorithm. An arc is critical if it belongs to a critical path. More precisely, an arc (t, u) is critical if $r(t) + d(t) = r(u)$ and if $r(t) + q(t)$ is equal to the makespan.

3.2 The Local Search Algorithm

This section describes a local search algorithm for the job-shop scheduling based on a neighborhood known as *N1* and a tabu-search strategy. This local search algorithm is the essence of the procedure proposed in [2]. Other neighborhoods and strategies have been proposed and it is not difficult to generalize the results presented here to these other algorithms.

The Neighborhood As mentioned, a solution to the job-shop scheduling problem mostly consists of ordering the tasks of the various machines. The idea underlying neighborhood N1 is to consider all critical arcs (u, v) and to swap the two tasks u and v in the ordering of their machine. In the following, we often abuse language and talk about swapping an arc. Note however that such a swap in fact consists of removing and adding three arcs. If p (resp. s) is the predecessor (resp. successor) of u (resp. v) on the machine, then the arcs (p, u), (u, v), and (v, s) are removed from, and the arcs (p, v), (v, u), and (u, s) are added to, the solution graph. Neighborhood N1 has a number of interesting properties: it preserves feasibility, it is connected (i.e., there exists a sequence of moves from any given state to the optimal solution), and it is minimal in the sense that swapping non-critical arcs cannot improve the makespan.

The Exploration Strategy The exploration strategy is based on tabu search, which appears to be successful for job-shop scheduling. It consists of selecting the swap that results in the state with the best makespan (which, in fact, may be worse that the makespan of the current state). A tabu-list also keeps the inverse of the swaps performed recently (e.g., if (u, v) is swapped, then (v, u) is marked as tabu). The length of the tabu-list varies over time. It decreases (resp. increases) when the makespan decreases (resp. increases).

Incrementality Because running a PERT algorithm to evaluate the makespan of every possible move is relatively expensive, it is generally proposed to approximate its value by considering the paths going through the swapped vertices only. This approximation is of course a lower bound on the actual makespan but it can be computed in constant time. Indeed, for a swap (v, w), it is sufficient to compute

$max(r'(v) + q'(v), r'(w) + q'(w))$

where

$r'(v) = max(r(pj(v)) + d(pj(v)), r'(w) + d(w)))$
$q'(v) = max(d(v) + q(sj(v)), d(v) + q(sm(w))))$
$r'(w) = max(r(pj(w)) + d(pj(w)), r(pm(v)) + d(pm(v)))$
$q'(w) = max(d(w) + q(sj(w)), d(w) + q'(v))$

4 A First LOCALIZER Model

This section presents a first model for the job-shop scheduling model expressed directly in terms of arrays and sets. It starts with the basic model and then discusses some improvements. The model contains several novelties, and requires new implementation techniques, compared to the applications discussed in [1].

4.1 The Basic Model

Figure 2 describes a basic LOCALIZER model for job-shop scheduling.

Optimize

Constant:

nbJ : **int** = ...;
nbM : **int** = ...;
N : **int** := $nbJ \times nbM$;
d : **array**[0..$N+1$] **of int** = ...;
m : **arrray**[1..N] **of int** = ...;
sj : **array**[i **in** 1..N] **of int** = ...;
pj : **array**[i **in** 1..N] **of int** = ...;
F : **{int}** = ...;
L : **{int}** = ...;
$minLen$: **int** = ...;
$maxLen$: **int** = ...;

Variable:

pm : **array**[1..N] **of int**;
sm : **array**[1..N] **of int**;
$tabu$: **array**[1..N,1..N] **of int**;
$tabuLen$: **int** = ...;

Invariant:

r : **array**[i **in** 0..N] **of int** :=
if i=0 **then** 0 **else max**($r[pj[i]] + d[pj[i]], r[pm[i]] + d[pm[i]]$);
q : **array**[i **in** 1..N+1] **of int**:=
if i=N+1 **then** 0 **else max**($d[i] + q[sj[i]], d[i] + q[sm[i]]$);
p : **array**[i **in** 1..N] **of int**:= r[i] + q[i];
$makespan$: **int** := **max**(i in L) $p[i]$;
Ca : **{int}** := { i : **int** |
select i **in** 1..N **where** p[i] = makespan **and** $r[pm[i]] + d[pm[i]] = r[i]$ **and** $pm[i] \neq 0$};

Operator:

void $swap(i$: **int**, j : **int**) {
smj : **int** := $sm[j]$;
$sm[i] := sm[j]$; $sm[j] := i$; $sm[pm[i]] := j$;
$pm[j] := pm[i]$; $pm[i] := j$; $pm[smj] := i$;
}

Objective Function:

minimize makespan

Neighborhood:

best move $swap(pm[u], u)$
where u **from** Ca
such that $tabu[PM[u], u] + tabuLen \leq trials$
accept when
improvement $\rightarrow$ { tabuLen := max(tabuLen-1,minLen); $tabu[u, pm[u]] := trials$; };
always $\rightarrow$ { tabuLen := min(tabuLen+1,maxLen); $tabu[u, pm[u]] := trials$; };

Figure 2: A First Job-shop Scheduling Model

The Constant Section The data in this model specifies the number of jobs nbJ, the number of machines nbM, the total number of tasks N (excluding the source and the sink), the duration d of the tasks, the machines m assigned to each task, the job successors sj and the job predecessors pj of the tasks, the set of tasks F starting the jobs, and the set of tasks L finishing the jobs. The integer *minLen* and *maxLen* also specify bounds on the length of the tabu list.

The State The state is described by specifying the machine predecessor and the machine successor of each task, as well as the tabu list and its length. The tabu list is represented by a matrix that associates with each move (i.e., a pair of tasks) the last "time" its inverse was performed. Note that the "time" is simply the value of the parameter **trials** in the computation model.

Invariants Invariants are the main tool provided by LOCALIZER to simplify the design of local search algorithms. They state, in a declarative way, the data structures that must be maintained incrementally. The invariants in this model essentially specify the concept introduced in Section 3.2: the release dates, the tails, the makespan, and the critical arcs. The release dates and the tails are maintained by the invariants

r : **array**[i **in** 0..N] **of int** :=
  if i=0 **then** 0 **else max**($r[pj[i]] + d[pj[i]], r[pm[i]] + d[pm[i]]$);
q : **array**[i **in** 1..N+1] **of int**:=
  if i=N+1 **then** 0 **else max**($d[i] + q[sj[i]], d[i] + q[sm[i]]$);

which express the recurrence relations presented earlier directly. These invariants are much more difficult to maintain than the invariants occurring in applications such as GSAT and graph-coloring. On the one hand, they are expressed recursively. On the other, they use variables as indices of invariants (e.g., `r[pj[i]]`), which complicates incremental algorithms since the data dependencies vary over time. The invariant

p : **array**[i **in** 1..N] **of int**:= r[i] + q[i];

maintains the length of the longest path between the source and the sink going trough task i, while the invariant

makespan : **int** := **max**(i in L) $p[i]$;

maintains the makespan as the longest path going through all final tasks. Finally, the critical arcs are represented as the sets of their targets

Ca : **{int}** := { i : **int** |
  select i **in** 1..N **where** p[i] = makespan **and** $r[pm[i]] + d[pm[i]] = r[i]$ **and** $pm[i] \neq 0$};

i.e., Ca is the sets of tasks i such that $(pm[i], i)$ is critical. Once again, the invariant mostly follows the definition given earlier.

The Neighborhood The neighborhood is defined in terms of the critical arcs and it expresses that an arc $(pm[u], u)$ must be considered for a swap unless it is tabu. The best such move is selected as the next state. Procedure `swap` performs the move and the move is tabu if its inverse was executed in the last *tabuLen* iterations. Note also that the tabu list and its length are updated in the acceptance clause.

4.2 Improvements

The model can be modified to include the approximation of the makespan discussed in Section 3.2 and a "reasonable" starting point.

Incrementality It is not difficult to change the neighborhood definition to support the makespan approximation. The key idea is to compute the approximation using a simple function and to choose the move minimizing the approximation. The acceptance criteria is also evaluated in the `current state`, using the current makespan and the approximation. The neighborhood then becomes as follows:

move $swap(pm[u], u)$
where u **from** Ca;
$a = appMakespan(u)$
such that $tabu[PM[u], u] + tabuLen \leq trials$
minimizing a
accept in current state when
makespan - a $> 0 \rightarrow$ { tabuLen := max(tabuLen-1,minLen); $tabu[u, pm[u]] := trials$; };
always $\rightarrow$ { tabuLen := min(tabuLen+1,maxLen); $tabu[u, pm[u]] := trials$; };

Starting Point It is important in job-shop scheduling to start from a reasonable solution. Our model uses a greedy procedure to find a starting point uses procedural constructs over sets and arrays. The procedure maintains a frontier which consists of the first unscheduled tasks of each job and it selects the task with the shortest duration first. More advanced procedures such as `Bidir` proposed by Dell'Amico et al. in [2] can be implemented without difficulty, since LOCALIZER includes a complete programming language.

5 A Higher-Level Localizer Model

The previous section presented a model for job-shop scheduling using only simple data structures such as sets and arrays. Many applications however are naturally described in terms of higher-level concepts such as graphs, trees, and circuits to name only a few. Recent research on LOCALIZER has focused on providing invariants over such high-level data types, since they are likely to be used in numerous applications. This section presents a LOCALIZER model based on graph-theoretic concepts.

5.1 Overview of the Main Concepts

We now describe briefly how graphs are supported in LOCALIZER. The aim is not to be exhaustive but to convey the main ideas underlying high-level extensions of LOCALIZER.

Data Types LOCALIZER offers concepts such as nodes, arcs, paths, circuits, and graphs as primitive data types. In addition, these types are polymorphic. For instance, if T is a type, then *Node(T)* is a node type whose nodes are represented by elements of type T and *Arc(T)* is an arc type whose arcs link nodes of type *Node(T)*.

Nodes Nodes of type T are constructed from values of type T. For instance, the declarations

```
Constant:
    n: Node(1..N) = 3;
    sn: {Node(1..N)} = 1..N;
```

declares a integer node whose value is 3 and a set of nodes whose values are integers from 1 to N. There is also an automatic casting operator transforming a node of type *node(T)* into a value of type T.

Arcs Arcs are ordered pairs of nodes. For instance, the declaration

```
Constant:
    a: Arc(1..N) = <3,4>;
```

declares an arc whose source is node 3 and whose target is node 4. If a is an arc, $a.s$ (resp. $a.t$) denotes its source (resp. its target).

Paths Paths are sequences of nodes. For instance, the declaration

```
Constant:
    p: Path(1..5) = (2,3,4,5,1);
```

declares a path whose origin is node 2 and whose destination is node 1. The declaration

```
Variable:
    seqm: Path(1..5);
```

declares a variable of type *Path(1..5)*. These variables may for instance represent the ordering of the tasks of a machine. Paths can be queried in a variety of ways and these queries can be used inside invariants. For instance, if p is a path of type T and n is a node of type T, *p.pred(n)* denotes the predecessor of n in p, *p.succ(n)* denotes the sucessors of n in p, *p.origin()* denotes the origin of p, *p.dest()* the destination of p, and *p.arcs()* the set of arcs in the paths. LOCALIZER maintains these values dynamically in presence of changes. A path can also be updated through a number of primitives procedures. For instance, *p.swap(a)* swaps the source and origin of arc a in path p and *p.toOrigin(n)* (resp. *p.toDest(n)*) removes node n from the path and makes it the origin (resp. destination) of the path.

Graphs Graphs are constructed from a set of nodes and a set of arcs of the same types. These nodes and the arcs may vary dynamically, since the graphs may be defined in terms of nodes and arcs variables. There are many primitives that can be defined on graphs to maintain incrementally shortest and longest paths, reachability, and connectivity. We only mention the primitive *longestPath(G,w,o,d)* which, given a graph G, two nodes o and d of G, and a vector w which associates a weight with every node returns the length of the longest path from o to d with the understanding that every arc whose source is n has weight $w[n]$.

5.2 The Model

Figure 3 depicts a job-shop scheduling model based on graphs.

Optimize

Constant:

nbJ : **int** = ...;
nbM : **int** = ...;
N : **int** := $nbJ \times nbM$;
$Nodes$: **{Node(1..N)}** := {0..N+1};
d : **array**[0..$N+1$] **of int** = ...;
m : **arrray**[1..N] **of int** = ...;
$seqj$: **array**[i **in** 1..J] **of Path**($Nodes$) := ...
$Arcsj$: {**Arc(1..N)**} := **union**(i **in** 1..J) $seqj[i].arcs()$;
$minLen$: **int** = ...;
$maxLen$: **int** = ...;

Variable:

$seqm$: **array**[1..nbM] **of Path**($Nodes$);
$tabu$: **array**[1..N,1..N] **of int**;
$tabuLen$: **int** = ...;

Invariant:

$Arcsm$: {**Arc(1..N)**} := **union**(i **in** 1..M) $seqm[i].arcs()$;
$Arcs$: {**Arc(1..N)**} := $Arcsm$ **union** $Arcsj$;
G: **Graph(1..N)** := **Graph**($Nodes, Arcs$);
r : **array**[i **in** 0..$N+1$] **of int** := **longestPath**($G, d, 0, i$);
q : **array**[i **in** 0..$N+1$] **of int** := **longestPath**($G, d, i, N+1$);
$makespan$: **int** := $r[N+1]$;
Ca : **Arc({int})** := { $< s,t >$: **Arc(1..N)** |
select $< s,t >$ **in** $Arcsm$ **where** r[t]+q[t] = makespan **and** $r[s] + d[s] = r[t]$ **and** $s \neq 0$};

Objective Function:

minimize makespan

Neighborhood:

best move seqm[m[s]].swap($< s,t >$)
where $< s,t >$ **from** Ca
such that $tabu[s,t] + tabuLen \leq trials$
accept when
improvement → { tabuLen := max(tabuLen-1,minLen); $tabu[t,s] := trials$; };
always → { tabuLen := min(tabuLen+1,maxLen); $tabu[t,s] := trials$; };

Figure 3: A Higher-Level Job-shop Scheduling Model

The Constant Section The constant section remains mostly unchanged. However, the model introduces a node for each task (these nodes are used later to specify the graph) and represents jobs as paths of tasks.

The Variable Section The state in this model is described as an array of paths, one for each machine. These paths represent how tasks using the same machine are ordered.

The Invariant Section The invariants are of course most interesting. They define a graph G whose nodes are the tasks and whose arcs are the precedence arcs (which are static) and the machine arcs (which are dynamic). The release dates and the tails can now be specified directly on the graphs as longest paths between nodes, i.e.,

$$r : \textbf{array}[i \textbf{ in } 0..N+1] \textbf{ of int} := \textbf{longestPath}(G, d, 0, i);$$
$$q : \textbf{array}[i \textbf{ in } 0..N+1] \textbf{ of int} := \textbf{longestPath}(G, d, i, N+1);$$

Note also that r and q are now defined for the source and the sink, making the model more uniform. The makespan is simply the release date of the sink. The critical arcs can be specified directly by manipulating arcs, i.e.,

$$Ca : \textbf{Arc(\{int\})} := \{ < s,t > : \textbf{Arc(1..N)} \mid$$
$$\textbf{select} < s,t > \textbf{ in } Arcsm \textbf{ where } \text{r[t]+q[t]} = \text{makespan } \textbf{and } r[s] + d[s] = r[t] \textbf{ and } s \neq 0\};$$

The Neighborhood Section The neighborhood section remains mostly unchanged but it now manipulates arcs directly.

Comparing the Models It is useful to step back and study the benefits of this new model. First, the model is closer to the informal description of the algorithm and it manipulates the underlying concepts directly. Second, the new description is much more flexible and extensible. It is easy to generalize to non-pure job-shop scheduling problems, e.g., problems with additional precedence or distance constraints. No change to the invariants is necessary, since they are expressed directly in terms of the graph. The first model in contrast, would require changing the precedence relation, making the model less compact and less natural. Note also that these extensions do not induce any overhead. LOCALIZER essentially compiles these high-level invariants in terms of traditional invariants over sets and arrays.

6 Implementation

This section reviews the main techniques for implementing the invariants, which are the cornerstone of LOCALIZER. Only a subset of arithmetic invariants is considered, since they capture the essence of the implementation.

6.1 Normalization

The first phase of the implementation consists of rewriting the invariants of LOCALIZER into primitive invariants by flattening expressions and arrays. The primitive invariants are of the form:

$$
\begin{array}{l}
x := \mathrm{c} \\
x := y \oplus z \\
x := \prod(x_1, \ldots, x_n) \\
x := \mathrm{element}(e, x_1, \ldots, x_n)
\end{array}
$$

where c is a constant, $x, y, z, x_1, \ldots, x_n$ are variables, $\oplus$ is an arithmetic operator such as $+$ and $*$ or an arithmetic relation such as $\geq, >$ and $=$, and $\prod$ is an aggregate operator such as **sum, prod, max**, and **min**. Relations return 1 when true and 0 otherwise. An invariant $x := element(e, x_1, \ldots, x_n)$ assigns to x the element in position e in the list $[x_1, \ldots, x_n]$. This last invariant is useful for arrays which are indexed by expressions containing variables.

At any given time, LOCALIZER maintains a set of invariants $\mathcal{I}$ over variables $\mathcal{V}$. Given an invariant $I \in \mathcal{I}$ of the form $x := e$ (where e is an expression), $def(I)$ denotes x while $exp(I)$ denotes e. Given a set of invariants $\mathcal{I}$ over $\mathcal{V}$ and $x \in \mathcal{V}$, $invariants(x,\mathcal{I})$ returns the subset of invariants $\{I_i\} \subseteq \mathcal{I}$ such that x occurs in $exp(I_i)$. The set of variables $\mathcal{V} \setminus \{def(I) | I \subseteq \mathcal{I}\}$ are the variables which are the parameters of the system of invariants. These variables only can be modified in the neighborhood definitions. Note also that a variable x can be defined by at most one invariant. i.e. there exist at most one $I \subseteq \mathcal{I}$ such that $def(I) = x$.

6.2 Overview of the Approach

Informally speaking, invariants are implemented using a planning/execution model. The planning phase determines a specific order for propagating the invariants, while the execution phase actually performs the propagation. The main design decision in LOCALIZER is to impose restrictions on the invariants to ensure that a pair (variable,invariant) is considered at most once by the execution phase. Various such restrictions can be imposed. Some of these restrictions make it possible to order invariants at compile time and are sufficient to execute models for satisfiability and graph-coloring. However, these restrictions preclude many interesting models for scheduling and resource allocation problems. The difficulty lies with invariants of the form $x := element(e, y_1, \ldots, y_n)$ If an ordering must be determined at compile time, it means that the invariant must be evaluated only after e and $y_1, \ldots, y_n$ have been determined. This is a very strong constraint, since it would prevent LOCALIZER from accepting expressions where some elements of an array may depend on some other, unknown at compile time, elements of the same array. As a consequence, the scheduling models presented in this paper would be rejected by this restriction.

The basic idea behind the LOCALIZER implementation is to evaluate the invariants by levels. Each invariant is associated with one level and, inside one level, the invariants can be ordered topologically. Once the propagation of a level is completed, planning of the next level can take place using the values of the previous levels, since lower levels are never reconsidered. The planning phase is thus divided in two steps. A first step, which can be carried out at compile time, partitions the invariants in levels. The second step, which is executed at runtime, topologically sorts the invariants within each level when the invariants at the lower level have been propagated.

The high-level algorithm is shown in Figure 4. It receives as inputs the set of invariants $\mathcal{I}$ and a set of variables $\mathcal{M}$ that have been updated. It first serializes the invariants. It then considers each level, applies a planning phase that produces a topological ordering on the invariants of that level, and propagates these invariants following this ordering.

```
procedure execute(I,M)
begin
    < I_0, ..., I_p > := serialize(I);
    for( i := 0; i <= p; i++) do
        t := plan(I_i);
        execute(I_i,M,t);
    endfor
end
```

Figure 4: The Propagation Algorithm

6.3 The Planning Phase

The planning/serializing phase can be formalized in terms of two assignments $l : \mathcal{I} \rightarrow \mathcal{N}$ and $t : \mathcal{I} \rightarrow \mathcal{N}$ and of two sets of constraints.

Definition 1 The level constraints associated with an invariant I, denoted by $lc(I)$, are defined as follows:

$$\begin{array}{lcl} lc(x := c) & = & \{l(x) = 0\} \\ lc(x := y \oplus z) & = & \{l(x) = max(l(y), l(z))\} \\ lc(x := \prod(x_1, \ldots, x_n) & = & \{l(x) = max(l(x_1), \ldots, l(x_n))\} \\ lc(x := element(e, x_1, \ldots, x_n)) & = & \{l(x) = max(l(e) + 1, l(x_1), \ldots, l(x_n))\} \end{array}$$

The level constraints are not strong except for the invariant **element** where the level of x is strictly greater than the level of e. Informally, it means that e must be evaluated in an earlier phase than x.

Definition 2 The level constraints associated with a set of invariants $\mathcal{I}$ and denoted by $l(\mathcal{I})$ is simply $\bigcup_{I \in \mathcal{I}} lc(I)$.

Definition 3 A set of invariants $\mathcal{I}$ is serializable if there exists an assignment $l : \mathcal{I} \rightarrow \mathcal{N}$ satisfying $lc(\mathcal{I})$.

A serializable set of invariants can be partitioned into a sequence $< \mathcal{I}_0, \ldots, \mathcal{I}_p >$ such the invariants in $\mathcal{I}_i$ have level i. This serialization can be performed at compile-time. The second step consists of ordering the invariants inside each partition. This ordering can only take place at runtime, since it is necessary to know the values of some invariants to simplify the **element** invariants.

Definition 4 Let S be a computation state and let $S(x)$ denote the value of x in S. The static constraints associated with an invariant I wrt S, denoted $sd(I, S)$ are defined as follows:

$$\begin{array}{lcl} sd(x := c, S) & = & \{t(x) = 0\} \\ sd(x := y \oplus z, S) & = & \{t(x) = max(t(y), t(z)) + 1\} \\ sd(x := \prod(x_1, \ldots, x_n, S) & = & \{t(x) = max(t(x_1), \ldots, t(x_n)) + 1\} \\ sd(x := element(e, x_1, \ldots, x_n), S) & = & \{t(x) = t(x_{S(e)}) + 1\} \end{array}$$

Definition 5 The static constraints associated with a set of invariants $\mathcal{I}$ wrt a state S, denoted by $sd(\mathcal{I}, S)$ is simply $\bigcup_{I \in \mathcal{I}} sd(\mathcal{I}, S)$.

```
procedure execute(I,M,t)
begin
    Q := {⟨x, I⟩|x ∈ M ∧ I ∈ invariants(x, I)}
    while Q ≠ ∅ do
        ⟨x, I⟩ := POP(Q,t);
        propagate(x,I,I,Q');
        Q := Q ∪ Q';
    endwhile
end

function POP(Q,t)
    pre: Q is not empty
    post: ⟨x, I⟩ ∈ Q such that ∀⟨x', I'⟩ ∈ Q : t(I') ≥ t(I)
```

Figure 5: The Execution Phase

Definition 6 A set of invariants $\mathcal{I}$ is static wrt to a state S if there exists an assignment $t : \mathcal{V} \rightarrow \mathcal{N}$ satisfying $sd(\mathcal{I}, S)$.

We are now in position to define the admissible invariants, i.e., the set of invariants currently supported by the LOCALIZER implementation.

Definition 7 Let S_0 be a computation state. A set of invariants $\mathcal{I}$ is admissible wrt S_0 if

1. $\mathcal{I}$ is serializable and can be partitioned into a sequence $< \mathcal{I}_0, \ldots, \mathcal{I}_p >$;
2. $\mathcal{I}_i$ is static wrt S_i where S_i $(i > 0)$ is the state obtained by propagating the invariants $\mathcal{I}_{i-1}$ in S_{i-1}.

Admissible invariants cannot be recognized at compile-time in general. It is easy however to recognize them at runtime.

6.4 The Execution Phase

The execution phase is given the set of variables $\mathcal{M}$ that have been updated and a topological assignment t. It then propagates the changes according to the topological ordering. The algorithm uses a queue which contains pairs of the form $\langle x, I \rangle$. Intuitively, such a pair means that invariant I must be reconsidered because variable x has been updated. The main step of the algorithm consists of popping the pair $\langle x, I \rangle$ with the smallest $t(I)$ and to propagate the change, possibly adding new elements to the queue. The algorithm is shown in Figure 5.

6.5 Propagating the Invariants

To complete the description of the implementation, it remains to describe how to propagate the invariants themselves. The basic idea here is to associate two values x^o and x^c with each variable x. The value x^o represents the value of variable x at the beginning of the execution phase, while the value x^c represents the current value of x. At the beginning of the execution phase, $x^o = x^c$ of course. By keeping these two values, it is possible to compute how much a variable has changed and to update the invariants accordingly. For instance, the propagation of the invariant $x := sum(x_1, \ldots, x_n)$ is performed by a procedure

```
procedure propagate(x_i, x := sum(x_1, ..., x_n), I, Q)
begin
    x^c := x^c + (x_i^c - x_i^o);
    if x^c ≠ x^o then
        Q := invariants(I, x);
end
```

The procedure updates x^c according to the change of x_i. Note that, because of the topological ordering, x_i^c has reached its final value. Note also that x^c is not necessarily final after this update, because other pairs $\langle x_j, x := sum(x_1, \ldots, x_n)\rangle$ may need to be propagated.

7 Experimental Results

This section reports some preliminary experimental results of LOCALIZER on a set of classic benchmarks. The goal of this section is to provide evidence that LOCALIZER is a viable tool to implement local search algorithms. Similar evidence was provided in [1] for satisfiability, graph-coloring, and graph-partitioning. We do not aim at comparing local and global search algorithms (which have different functionalities). Similarly, we did not try to produce the fastest implementation possible but rather to demonstrate the potential of LOCALIZER. The experiments were based on the parameters reported in [2]: the maximal number of searches (maxSearches) is 1, the maximal number of iterations for the inner loop (maxTrials) is 12000, the tabu list has a varying length constrained in between 5 and 30 and its length is updated according to the rule of [2]. Contrary to [2], our model does not use a restarting strategy.

Table 1 reports the preliminary results on a Sun Sparc Ultra 1 for the graph model (the first model is about 15% slower in the average). The table reports a coarse histogram that summarizes the frequencies of the solutions (i.e., the value of their makespan). `Loc` is the CPU time in seconds to run until completion, while `LO` is the time to run to completion or to a known optimal solution (this is the time measure used in [2]). Column `Avg. sol.` gives the average makespan, while column *%O* gives the average distance to the optimum in percent. Results in italics indicates that the optimal solution is not known (as of 1993). The results indicate that the LOCALIZER model model obtains near-optimal solutions quickly. For problems where the optimum is known, the solutions are always within 6% of the optimum. In fact, the model finds optimal solutions for 14 benchmarks. The table indicates that these results are also obtained quickly: from 20 seconds to 90 seconds. These results cannot be really compared with the results of [2], since his neighborhood is an extension of the neighborhood presented here and the results are not directly comparable. But a rough comparison suggests that the difference between LOCALIZER and a specialized implementation is once again of the order of a machine generation, confirming the results in [1].

8 Conclusion

This paper explored the application of LOCALIZER to job-shop scheduling. It presented two models for this application: a model based on traditional data structures and a model based on graph-theoretic concepts. The second model illustrates how general data types may improve the modeling power of the language without degrading performance. The underlying implementation was also described and experimental results demonstrate the potential of the approach. The LOCALIZER models find optimal or near-optimal solutions in about 30 seconds on the standard collection of benchmarks. These results provide some

Benchmark			Results (MD=12000,MS=1,Neighborhood=N1)											
Name	*Job/M*	*Opt*	*Ranges*				*Freq.*				Loc	LO	*Avg.sol.*	%O
LA06	15/5	926	926	926	926	926	100	0	0	0	22.1	0.8	926.0	0
LA07	15/5	890	890	890	890	890	100	0	0	0	24.5	3.1	890.0	0
LA08	15/5	863	863	863	863	863	100	0	0	0	23.7	1.5	863.0	0
LA09	15/5	951	951	951	951	951	100	0	0	0	22.7	1.2	951.0	0
LA10	15/5	958	958	958	958	958	100	0	0	0	21.9	1.1	958.0	0
LA11	20/5	1222	1222	1222	1222	1222	100	0	0	0	25.2	3.6	1222.0	0
LA12	20/5	1039	1039	1039	1039	1039	100	0	0	0	24.0	3.1	1039.0	0
LA13	20/5	1150	1150	1150	1150	1150	100	0	0	0	24.0	6.8	1150.0	0
LA14	20/5	1292	1292	1292	1292	1292	100	0	0	0	22.8	2.4	1292.0	0
LA15	20/5	1207	1207	1207	1207	1207	100	0	0	0	27.4	5.5	1207.0	0
LA16	10/10	945	947	961	975	988	1	11	21	67	35.4	35.2	975.1	3
LA17	10/10	784	784	790	796	801	17	74	8	1	36.1	34.2	786.4	0
LA18	10/10	848	848	857	866	873	1	31	56	12	36.4	35.9	860.1	1
LA19	10/10	842	843	850	857	864	1	25	49	25	37.0	36.5	853.9	1
LA20	10/10	902	902	908	914	918	13	24	59	4	36.7	33.1	909.5	1
LA21	15/10	*1048*	1060	1078	1096	1114	1	27	57	15	48.4	49.0	1084.9	*3*
LA22	15/10	927	935	948	961	974	1	28	56	15	47.2	47.9	952.9	3
LA23	15/10	1032	1032	1033	1034	1034	99	0	1	0	49.2	22.0	1032.0	0
LA24	15/10	935	945	959	973	985	2	22	67	9	47.2	47.8	964.3	3
LA25	15/10	977	989	1010	1031	1051	1	39	51	9	46.0	46.7	1015.1	4
ABZ5	10/10	1234	1236	1246	1256	1264	3	34	54	9	36.8	38.4	1248.8	1
ABZ6	10/10	943	943	948	953	958	13	69	16	2	37.4	37.4	946.9	0
ABZ7	20/15	*667*	686	723	760	797	1	58	39	2	79.3	84.7	721.5	*8*
ABZ8	20/15	*678*	688	724	760	796	1	6	70	23	83.3	80.7	747.7	*10*
ABZ9	20/15	*692*	715	735	755	774	2	50	43	5	93.9	88.9	735.2	*6*
MT6	6/6	55	55	55	55	55	100	0	0	0	13.2	1.0	55.0	0
MT10	10/10	930	941	959	977	941	1	35	43	0	36.6	23.6	966.1	4
MT20	20/5	1165	1173	1194	1215	1173	8	67	23	0	29.8	18.7	1186.1	2
ORB1	10/10	1059	1073	1095	1117	1160	1	2	25	72	36.2	36.4	1124.6	6
ORB2	10/10	888	889	896	903	917	3	33	42	22	36.1	36.3	899.6	1
ORB3	10/10	1005	1021	1081	1141	1261	2	89	8	1	37.4	37.6	1060.5	6
ORB4	10/10	1005	1019	1031	1043	1064	1	30	44	25	33.8	34.1	1037.3	3
ORB5	10/10	887	899	911	923	947	1	29	40	30	37.5	37.7	918.7	4
ORB6	10/10	1010	1022	1034	1046	1069	2	26	42	30	35.8	36.1	1041.8	3
ORB7	10/10	397	397	403	409	420	1	5	47	47	38.5	38.3	409.5	3
ORB8	10/10	899	914	932	950	986	1	9	54	36	37.5	37.8	947.8	5
ORB9	10/10	934	934	945	956	976	1	3	35	61	32.4	32.7	959.8	3
ORB10	10/10	944	944	956	968	992	2	24	48	26	37.0	36.6	963.5	2

Table 1: Job-Shop Scheduling: Experimental Results

evidence that Localizer has the potential to scale up for tackling sophisticated local search applications. Future work will aim at applying these ideas to vehicle routing and travelling salesman problems.

References

[1] L. Michel, P. Van Hentenryck. Localizer: A Modeling Language for Local Search. In *Second International Conference on Principles and Practice of Constraint Programming (CP'97)*, Linz, Austria, October 1997. (Extended Version invited to the special of *Constraints* on CP'97).

[2] Marco Trubian Mauro Dell'Amico. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.