# A Reconfigurable Optimizing Scheduler

**David J. Montana**
BBN Technologies
10 Moulton Street, Cambridge, MA 02138
dmontana@bbn.com

## Abstract

We have created a framework that provides a way to represent a wide range of scheduling and assignment problems across many domains. We have also created an optimizing scheduler that can, without modification, solve any problem represented using this framework. The three components of a problem representation are the metadata, the data, and the scheduling semantics. The scheduler performs the optimization using an order-based genetic algorithm to feed different task orderings to a greedy schedule/assignment builder. The scheduler obeys the hard and soft constraints specified in the scheduling semantics. We have applied this reconfigurable scheduler to a variety of scheduling and assignment problems including the job shop, traveling salesman, vehicle routing, and generalized assignment problems. The results demonstrate that the optimizer can provide not only easy reconfigurability but also competitive performance.

## 1 Introduction

Optimizing schedulers have traditionally targeted a single problem or narrow class of problems. Changing a scheduler to handle a new problem or domain has required redesigning the scheduler and rewriting portions of its software. This introduces an expense that makes optimized scheduling impractical for most applications that could benefit from it. Only applications with large amounts of money tied to the quality of the schedules can justify the costs of developing custom software and algorithms.

Our work aims to change this. We provide a simple yet effective way for a user to configure our optimizing scheduler for a particular problem/domain. Configuring our scheduler does not require recoding or detailed knowledge of how the scheduler works. This can potentially make optimized scheduling sufficiently inexpensive to be practical for a far greater range of problems than it is currently.

Other researchers have recognized the benefits of a unified or reconfigurable approach to scheduling. [Smith and Becker, 1997] creates a unified scheduling ontology, but this ontology is not well suited to simple representation of a problem and is not in a form easily used by an optimizing scheduler. [Davis and Fox, 1994] and [McIlhagga, 1997] both make initial attempts at a reconfigurable scheduler, but they fall short in terms of the generality and flexibility required. The work on AMPL [Fourer *et al.*, 1993] does emphasize easy reconfigurability. It is similar to our approach in its use of algebraic expressions to define the problem as well as its separation of the problem specification from the solver. It is different from our approach because it is targeted at mathematical programming applications and not well suited to many symbolically oriented scheduling problems.

The two key innovations that have allowed us to create a truly reconfigurable optimizing scheduler are in the problem representation. The first is letting the user define the metadata, i.e. the formats for all the data sent to the scheduler. Hence, for any problem the user can define a data representation that is natural for that problem. The second innovation is allowing the user to specify the scheduling semantics using formulas. This allows the scheduler to compute problem-specific information such as whether a resource can perform a task or how much time a resource takes to perform a task.

Our scheduler uses an approach that was introduced by [Whitley *et al.*, 1989] and refined by [Syswerda, 1991]. An order-based genetic algorithm generates task orderings to feed to a greedy schedule builder. What is novel about our scheduler is the way that it can solve any scheduling problem represented using our problem representation framework. Hence, the scheduler is truly reconfigurable.

In the remainder of the paper, we start with an overview of

| **Constraint** | **Return Type** | **Defined Variables** | **Default Value** | **Description** |
|---|---|---|---|---|
| Optimization Criterion | number | | 0 | Numerical measure of quality of the current full schedule |
| Optimization Direction | multiple choice | N/A | minimize | Must be either minimize or maximize |
| Delta Criterion | number | task, resource | 0 | Incremental contribution to optimization criterion introduced by having resource perform task |
| Best Time | datetime | task, resource | starttime | Optimal time for the task to begin |
| Capability | boolean | task, resource | true | Whether resource has the required skills to perform task |
| Task Duration | number | task, resource | 0 | How many seconds it takes resource to perform task |
| Setup Duration | number | task, previous, resource | 0 | How many seconds it takes resource to prepare to perform task if it last performed previous |
| Wrapup Duration | number | task, next, resource | 0 | How many seconds it takes resource to clean up after doing task if it will be performing next |
| Prerequisites | list of strings | task | empty list | Names of all the tasks that must be scheduled before scheduling task |
| Task Unavailability | list of intervals | task, resource, prerequisites | empty list | All intervals of time when task cannot be scheduled (label1 and label2 fields ignored) |
| Resource Unavailability | list of intervals | resource | empty list | All intervals of time when resource is busy (label1 and label2 can be used for text and color) |
| Capacity Contribution | list of numbers | task | 0 | How much task contributes towards filling each type of capacity |
| Capacity Threshold | list of numbers | resource | 0 | How much capacity of each type that resource has |
| Multitasking | multiple choice | N/A | none | Ability of resources to perform more than one task at a time (none, ungrouped, or grouped) |
| Groupable | boolean | task1, task2 | false | Whether task1 and task2 can be placed in the same group |

Table 1: List of the various constraints that can be specified

the problem representation framework. We then describe how our scheduler utilizes the information in a problem representation in order to find an optimized schedule for that problem. We conclude with some results on the performance of the scheduler.

## 2 The Problem Representation Framework

A problem representation consists of three components: the metadata, the data, and the scheduling semantics. We now provide an abbreviated discussion of what each of these involves. More details on the problem representation framework are available in [Montana, 2001].

**Metadata -** Our scheduling system provides a small number of atomic data types (string, number, boolean, datetime, and list) and predefined composite data types (interval, xy-coord, latlong, and matrix). The user builds new composite data types (also called *object types*) from these atomic and predefined types. The data type for a field can itself be another user-defined object, and hence the user can potentially build complex objects. The user must specify a single object type for tasks and a single object type for resources.

**Data -** Most of the data are instances of object types specified by the metadata. There must be some task instances to schedule and some resource instances to which to assign these tasks. There can also be other data, such as business rules or distance matrices, not associated with a particular task or resource but used as part of the scheduling logic. Two pieces of data that are not object instances are the start and end times of the “scheduling window”, which define the earliest and latest time that an assignment can occur. Other non-object data is that specifying which set of assignments from a previously produced schedule should remain frozen in the current scheduler run. (This concept of freezing is important for dynamic rescheduling.)

**Scheduling Semantics -** We have defined a set of general constraints that define what constitutes a legal and optimized schedule. For most of these constraints, the user specifies a formula that tells how to compute the value of the constraint in a given context. For example, the Task Duration constraint tells how many seconds it takes a particular resource to perform a particular task. If this value is obtained by dividing the distance field of the task object by the speed field of the resource object, then the formula to specify for this constraint is task.distance / resource.speed. A description of the mini-language for specifying formulas in given in [Montana, 2001]; the examples in Section 3 should provide an idea of how these formulas work and what they can express.

Table 1 lists all the different constraints for which the user can specify a formula. If the user does not specify a formula, the default value is used. The context in which the constraint is evaluated is given by the value of the variables that are defined. While some variables (tasks, resources, starttime, and endtime) are defined for all constraints, some variables (e.g., task and resource) are defined only for certain constraints. The descriptions provided are brief; Section 4 provides a better understanding of some of these constraints by describing how they are actually used.

## 3 Examples of Problem Specifications

We now describe four examples of problem specifications. These well-known problems from the operations research literature are the problems we used for the experiments described in Section 5. (The OR-Library [Beasley, 1990] is a good source of such classic problems.) We have specified and solved problems much more algorithmically complex than those given here, but these highly idealized problems provide a good introduction to how to specify a problem.

### 3.1 Traveling Salesman Problem (TSP)

There is a salesman who needs to start at a given city, travel to a set of other cities visiting each city once, and then return to the starting city. The distance from any city to any other city is provided. The objective is to minimize the total distance traveled.

The task object, *city*, and resource object, *salesman*, are defined to have the fields:

- **city** - id (string) and index (number)
- **salesman** - id (string)

There is one salesman with arbitrary id; $N$ cities with index = i and id = "City i" for $i = 1, ..., N$; and an $N$x$N$ matrix named *distances* that contains all the intercity distances. For the scheduling semantics, the constraints with non-default values are shown in Table 2.

| **Constraint** | **Formula** |
|---|---|
| Optimization Criterion | maxover (resources, "r", complete (r)) - starttime |
| Setup Duration | matentry (distances, task.index, if (hasvalue (previous), previous.index, 1)) |
| Prerequisites | if (task.id = "City 1", mapover (tasks, "t2", if (t2.id != "City 1", t2.id))) |

Table 2: Constraints for Traveling Salesman Problem

### 3.2 Vehicle Routing Problem with Time Windows

This problem is described in [Solomon, 1987]. There are M vehicles and N customers from whom to pick up cargo. Each vehicle has a limited capacity for cargo, and each piece of cargo contributes a different amount towards this capacity. There is a certain window of time in which each pickup must be initiated, and the pickups require a certain non-zero time. Each vehicle that is utilized starts at a central depot, makes a circuit of all its customers, and then returns to the depot. The objective is to minimize the total distance traveled by the vehicles.

| **Constraint** | **Formula** |
|---|---|
| Optimization Criterion | sumover (resources, "r", preptime (r)) + sumover (tasks, "t", if (hasvalue (resourcefor (t)), 0, 1000)) |
| Delta Criterion | preptime (resource) - previousdelta (resource) |
| Task Duration | extra.servicetime |
| Setup Duration | dist (task.location, if (hasvalue (previous), previous.location, extra.depotlocation)) |
| Wrapup Duration | if (hasvalue (next), 0, dist (task.location, extra.depotlocation)) |
| Task Unavailability | list (interval (starttime, starttime + task.earliest), interval (starttime + task.latest + extra.servicetime, endtime)) |
| Capacity Contributions | list (task.load) |
| Capacity Thresholds | list (resource.capacity) |

Table 3: Constraints for Vehicle Routing Problem

The problem-specific objects are:

- **customer** - id (string), load (number), earliest (number), latest (number), and location (xycoord)
- **vehicle** - id (string) and capacity (number)
- **extradata** - servicetime (number) and depotlocation (xycoord)

The single object of type extradata is named *extra*. For the scheduling semantics, the constraints with non-default values are shown in Table 3.

### 3.3 Generalized Assignment Problem (GAP)

This problem is describe in [Osman, 1995]. There are N jobs to be assigned to M agents. There are defined assignment costs, one associated with each pairing of a job and

| Constraint | Formula |
|---|---|
| Optimization Criterion | sumover (tasks, “t”, entry (t.costs, resourcefor (t).index)) |
| Optimization Direction | maximize |
| Delta Criterion | entry (task.costs, resource.index) |
| Capacity Contributions | task.loads |
| Capacity Thresholds | loop (length (resources), “i”, if (i = resource.index, resource.capacity, 100000)) |

Table 4: Constraints for Generalized Assignment Problem

an agent. Each agent has a defined capacity, and each job contributes a defined amount towards the capacity of each agent, with this amount depending on the agent. The objective is to maximize the total costs.

The problem-specific objects are:

- **job** - id (string), index (number), costs (list of numbers), and loads (list of numbers)
- **agent** - id (string), index (number), and capacity (number)

The costs field of each job contains one cost for each agent, which can be accessed from the list using the index of the agent. The same applies to the loads field of each job. For the scheduling semantics, the constraints with non-default values are shown in Table 4.

### 3.4 Job-Shop Scheduling Problem (JSSP)

This problem was originally proposed by [Muth and Thompson, 1963]. There are M machines and N manufacturing jobs to be completed. Each job has M steps, with each step corresponding to a different specified machine. There is a specified order in which the steps for a certain job must be performed, with one step not able to start until the previous step has ended. The objective is to minimize the end time of the last step completed.

The problem-specific objects are:

- **step** - id (string), duration (number), machine (string), and preceedingstep (string)
- **machine** - id (string)

For the scheduling semantics, the constraints with non-default values are shown in Table 5.

## 4 The Reconfigurable Scheduler

We have created a scheduler that is capable of finding an optimized solution for any scheduling problem specified using the framework described above. A “greedy”, i.e. locally optimal, scheduler builder takes a particular ordering of tasks and assigns them one at a time to the best resource for that task. A genetic algorithm generates different task orderings to feed the greedy schedule builder, searching for an optimal ordering. The overall control flow of the scheduler is shown in Figure 1.

| Constraint | Formula |
|---|---|
| Optimization Criterion | maxover (resources, “r”, complete (r)) - starttime |
| Capability | task.machine = resource.id |
| Task Duration | task.duration |
| Prerequisites | if (task.preceedingstep != “”, list (preceedingstep)) |
| Task Unavailability | mapover (prerequisites, “t”, interval (starttime, taskendtime (t))) |

Table 5: Constraints for Job-shop Scheduling Problem

```
Greedy initialization;
Genetic loop:
  Determine new task ordering;
  Task (greedy) loop:
    Find next task to schedule;
    Resource (greedy) loop:
      Find next capable resource;
      Time (greedy) loop:
        Search to find best interval
         for resource to perform task;
      end Time loop;
      Check whether this
       resource/interval best so far;
    end Resource loop;
    Assign task to best resource
     during best interval;
  end Task loop;
  Evaluate fitness of schedule;
end Genetic loop;
```

Figure 1: Control flow of the scheduler

### 4.1 The Genetic Algorithm

The genetic algorithm is a fairly standard order-based one. We number each task from 1 to N, where N is the number of tasks, and a chromosome is some permutation of the numbers 1 through N. The crossover operator we use is position-based crossover, which is described in [Syswerda, 1991]. The mutation operator is a variation on Syswerda’s order-based mutation except that, instead of selecting only two positions whose order to exchange, our mutation selects between 2 and N positions whose order is randomly generated while the other positions remain the same. The population is initialized by choosing random orderings.

The replacement scheme is steady-state rather than generational, i.e. a single child enters the population and the worst individual leaves the population in a single "generational cycle". Duplicate individuals are not allowed in the population. The parent selection probabilities are exponentially distributed. The parameter parent-scalar is defined as the ratio of the probabilities of selecting the $i^{th}$ best individual and of selecting the $(i-1)^{st}$ best individual.

There are four conditions under which the genetic algorithm can terminate. First, it will stop if the elapsed wall time of its current run exceeds a parameter (max-time). Second, it will terminate if the total number of evaluations (i.e., individuals generated) exceeds a parameter (max-evals). Third, it will stop if the best score has not improved for a consecutive number of evaluations exceeding a parameter (max-top-dog-age). Fourth, it will terminate if the number of duplicate individuals generated exceeds a parameter (max-duplicates).

Evaluation of an individual is done by first feeding the ordering of the tasks to the greedy schedule builder and letting it build a schedule. The formula given by the Optimization Criterion constraint is then executed on this schedule. The number returned by the formula is the chromosome's fitness.

## 4.2 The Greedy Schedule Builder

The algorithm of the greedy schedule builder, although simple in concept, is complicated by the need to consider so many different factors. For the special case of the job-shop scheduling problem, our greedy scheduler is equivalent to the active schedule generation algorithm presented in [Giffler and Thompson, 1960]. However, to handle problems other than the job-shop problem, our greedy scheduler must consider a variety of other factors including:

- resource selection - Many scheduling problems allow a choice between different qualified resources for each task.
- time selection - For many scheduling problems finishing a task earlier is not always better, such as is the case with just-in-time scheduling.
- multitasking - Some scheduling problems allow resources to perform more than one task simultaneously.

As shown in Figure 1 there are different components of the greedy schedule builder. We now discuss each of these.

**Initialization -** There are certain results that the greedy schedule builder needs but that do not vary based on what assignments are made. For the sake of efficiency, these are computed once before the genetic algorithm even starts. These results include:

- **Lists of capable resources -** For each task, it creates a list of all those resources that have the skills/capabilities to perform that task. It determines whether a resource has the required skills by executing the Capability formula with the *task* variable set to the task and the *resource* variable set to the resource.
- **Resource unavailable times -** For each resource, it computes a set of nonoverlapping intervals of time for which that resource is not available to be assigned to a task due to other commitments (e.g., time off or maintenance). To do this, it executes the Resource Unavailable Times formula with the *resource* variable set appropriately to obtain a preliminary set of intervals. It adds to this list the intervals that represent the constraint that resources should not be scheduled before the start or after the end of the scheduling window of the window. Then, it resolves these into a set of nonoverlapping intervals.
- **Capacity contributions -** For each task, it computes the task's contribution towards each of the capacities by executing the Capacity Contributions formula with the *task* variable set appropriately. The $i^{th}$ element of the list is the contribution to the $i^{th}$ capacity.
- **Capacity thresholds -** For each resource, it computes the resource's threshold for each of the capacities using the Capacity Thresholds formula.
- **Prerequisites -** For each task, it computes the set of other tasks that must be scheduled prior to this task regardless of the ordering of tasks provided by the genetic algorithm. The Prerequisites formula provides a list of task names, which are used to look up the task objects.

**Task Loop -** The greedy schedule builder assigns one task at a time. It attempts to adhere as much as possible to the order in the chromosome, but it will not schedule a task before its prerequisites have been scheduled. So, each time through the loop it picks the task earliest in the chromosome that has not yet been scheduled but all of whose prerequisite tasks have been scheduled. After executing the resource loop in order to find the best resource and time, it assigns the task to that resource at that time. If there is no resource that is capable and available to perform the task, then the task is marked as unassigned.

The assignment process involves the following steps. First, the task must be inserted into the resource's schedule. If the Multitasking selection is grouped and the resource loop has specified a particular group for the task, then the task is placed in this group. Otherwise, a new schedule entry is made for this task and resource with setup start time, task start time, task end time, and wrapup end time as specified from the resource loop. (The time interval associated with a task assignment is divided into three consecutive intervals: the setup interval when the resource is preparing to perform the task, the task interval when the resource performs the task, and the wrapup interval when the resource cleans up. The four times represent the boundaries of these three intervals.) The wrapup end time of the previous task in the resource's schedule and the setup start time of the next task are also updated if necessary as specified by the resource loop. If there is grouped multitasking, then a new entry is also a new group.

Next, the capacities are updated. If the Multitasking selection is none, the capacities are single aggregates summed over time, and the capacities used by the resource are updated by adding the capacity contributions from the task. Otherwise, the capacities are time histories, and they are updated accordingly.

**Resource Loop -** To find the best resource and interval of time to which to assign a given task, the greedy schedule builder examines each resource on the task's list of capable resources. For a given resource, it starts by computing, using the Best Time formula, the ideal time for the task start time. This is a soft constraint that tells the time loop where to start its search. It also computes two hard constraints on time, the task duration and the task unavailable times, using the corresponding formulas. It then uses the time loop to search forward from the best time for the nearest legal task start time, where a time is legal if

- the resource is available for the entire interval between the corresponding setup start time and wrapup end time, and the task is available between the task start time and the corresponding task end time
- the setup start time for the task is not earlier than the wrapup end time from the previous task for that resource, and the wrapup end time of the task is not later than the setup start time of the next task
- none of the aggregate capacity contributions exceed their corresponding capacity thresholds

Alternatively, if there is grouped multitasking, then a task start time is legal if it is the task start time for an existing group such that

- its task duration is no longer than the task duration of the group
- the aggregate capacity contributions of the group after adding the task do not exceed any capacity thresholds
- executing the Groupable formula for this task and a task already in the group returns true

If the forward search yields a legal time, then it makes a temporary assignment of the task to the resource at the specified time, and evaluates the Delta Criterion formula to obtain a fast measure of how good that assignment would be. If the forward search yields no time or a time which is not the best time, then it repeats the process, this time searching backward from the best time for the closest legal start time. If neither the forward or backward search yields a time, then the task cannot be assigned to this resource. If the forward and backward search both yield times, then it picks the one with the best delta criterion. The resource (and time) with the best delta criterion is selected for assignment.

**Time Loop -** When performing the search for the legal task start time closest to the best time, there are a few items about which to be careful. First, the setup and wraup durations depend respectively on the previous and next task in the resource's schedule. Hence, they can only be computed in the context of a proposed position of the task in the resource's schedule. Additionally, the previous task's wrapup time and next task's setup time (if these tasks exist) are potentially altered by the placement of the new task and must therefore be recomputed. All these quantities are stored along with the task start time to allow the task loop to make the assignment. A second item to be careful about is that this is the innermost loop and hence is executed the most frequently. Therefore, it needs to be particularly efficient.

## 5 Experimental Results

The data for which we have executed our experiments are instances of the problems given in Section 3. These are commonly studied problems that we use because they allow comparison with other algorithms. We cannot hope to match the performance of the best algorithms developed for these problems for two reasons. First, we do not tune our algorithm to any particular problem and therefore will generally not achieve optimal performance for a particular problem. Second, the formulas are not compiled directly into machine code but rather are interpreted, and hence they execute less efficiently than compiled code. However, the benefit of our approach is the wide range of problems it can handle and the ease with which it can handle new problems, so we only need to prove reasonably good, not optimal, performance.

For each experiment, we have selected a particular data set and a particular set of genetic algorithm parameters, and we have made ten genetic scheduler runs. Table 6 summarizes the results of these experiments. Note that for each experiment, Table 6 tells the key genetic algorithm parameters: population size, parent-scalar, and either max-evals or max-top-dog-age (depending on which actually caused all the terminations). The table also gives the following results from the experiments:

- Best Known Score - the score of either the provably best solution or the best solution found by any algorithm to date (used as a reference)
- Best Score - the score of the best solution from all ten runs
- Median Score - the median of the scores of the ten solutions found by the ten runs
- Average Score - the mean of the scores from the ten runs

- Average Number of Evaluations - the average number of individuals evaluated in a run before the run terminated (because the genetic algorithm is steady-state, this is a better measure than the number of generations)
- Average Time Per Run - the average amount of time it required a run to execute to completion
- Time Per Evaluation - the average number of milliseconds required to perform a single evaluation

All the runs were made on a 200 MHz UltraSparc processor.

For the traveling salesman problem, we have so far used a single instance, bays29, which is a 29-city symmetric problem available at the TSPLIB web site. The first two rows in Table 6 correspond to two sets of runs for this data with different genetic algorithm parameters. The first row has a larger population, proportionately lower fitness pressure from parent-scalar and a larger max-top-dog-age. It does well at finding nearly optimal solution. The second row runs faster but does not do as well. This illustrates the tradeoff between search time and quality of solution. (A third factor in the tradeoff is computational power and its cost, particularly with an inherently parallelizable algorithm such as a genetic algorithm.) This is a relatively small traveling salesman problem, and while we could practically do significantly bigger problems, this algorithm cannot compete with specially designed algorithms such as [Lin and Kernighan, 1973].

For the job-shop scheduling problem, we have so far used only the Muth-Thompson 6x6 data [Muth and Thompson, 1963], referred to as ft06 at the OR-Library web site. It contains 36 tasks and 6 resources. Despite the fact that this is larger than the traveling salesman problem, the scheduler clearly has an easier time with the job-shop problem. The time per evaluation is roughly the same even though the job-shop problem has more resources because the job-shop problem has only one capable resource per task, and that is a better measure of the computation required. The jobshop problem requires less evaluations to find the optimal solution because the search space is in practice smaller. This is because the constraints in the job-shop problem, particularly the prerequisites constraint, make it so that many different chromosomes decode to the same schedule. One lesson is that one cannot predict the search time required purely based on the number of tasks and resources.

The generalized assignment problem is so far the only problem for which we have experimented with multiple instances. From the OR-Library web site, we have used c515-1 (5 resources and 15 tasks), c530-1 (5 resources and 30 tasks), and c1030-1 (10 resources and 30 tasks). This has allowed a very preliminary examination of the scaling properties of our algorithm. We would expect the time per evaluation to be roughly proportional to the product of the number of tasks and the number of capable resources per task (which in this case is the number of resources), and this is the case for this data. We would also expect an increase in the number of evaluations required with an increase in the number of tasks due to the larger search space, and this is also borne out by the data. Overall, these problems are solved quickly because the greedy algorithm does most of the work. One interesting result is that while the algorithm can get close to the optimal solution for c530-1 quickly, it requires a long search to find the best solution.

The next logical step for the experimentation process is to perform the same experiments for larger search problem such as the Muth-Thompson 10x10 job-shop problem or the Solomon vehicle routing problems.

## 6 Conclusions and Future Work

We have developed a powerful framework for representing scheduling problems, and we have built a reconfigurable scheduler that can find an optimized solution for any problem specified in this framework. The optimization performance of this scheduler is good, even though the generality of our approach does mean that, for certain problems, we cannot acheive the performance a scheduler designed specifically for that problem. The major benefit of reconfigurability is that it makes development of optimized scheduling for a wide range of problems simple and inexpensive. There is a vast array of scheduling problems that are currently solved using manual or non-optimized scheduling, and for most of these problems making optimized scheduling practical requires a simple and inexpensive solution rather than the best possible performance.

Further enhancing the ease of use of our reconfigurable scheduler is a web-based system we have built to allow the user to interact with the scheduler. The details of this interface are beyond the scope of this paper, but in general terms the browser-based interface allows the user to fully specify a problem (metadata, data, and scheduling semantics), start a new scheduler run and check on its progress, and graphically view the schedules. Using display constraints similar to the scheduling constraints described in Section 2 allows the user to select the colors and text to display with each assignment.

Also beyond the scope of this paper but illustrating the advantages of reconfigurability, we have integrated our reconfigurable scheduler into the same multiagent infrastructure as described in [Montana *et al.*, 2000]. This has allowed us to build multiagent societies that have included multiple interacting reconfigurable scheduling agents as well as other types of agents.

There are two directions in which to extend our work on the reconfigurable scheduler. First, as we expand the prob-

| **Problem Name** | **Pop Size** | **Parent Scalar** | **Max Evals** | **Max Top Dog** | **Best Known Score** | **Best Score** | **Median Score** | **Avg Score** | **Avg Num Evals** | **Avg Time (M:S)** | **Msecs Per Eval** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TSP-bays29 | 5000 | 0.998 | N/A | 20000 | 2020 | 2028 | 2028 | 2042 | 134,429 | 13:25 | 5.99 |
| TSP-bays29 | 1000 | 0.99 | N/A | 4000 | 2020 | 2058 | 2204 | 2191 | 26,680 | 2:41 | 6.02 |
| JSSP-mt06 | 1000 | 0.99 | 5000 | N/A | 55 | 55 | 55 | 55 | 5000 | 0:54 | 10.9 |
| GAP-c515-1 | 500 | 0.98 | 2500 | N/A | 336 | 336 | 336 | 336 | 2500 | 0:09 | 3.48 |
| GAP-c1030-1 | 1000 | 0.99 | 8000 | N/A | 709 | 709 | 709 | 708.8 | 8000 | 1:31 | 11.4 |
| GAP-c530-1 | 1000 | 0.99 | 5000 | N/A | 656 | 655 | 653 | 653.3 | 5000 | 0:39 | 7.88 |
| GAP-c530-1 | 20000 | 0.9995 | 100000 | N/A | 656 | 656 | 656 | 655.3 | 100000 | 14:10 | 8.50 |

Table 6: Summary of experimental results

lem representation, we need to extend the scheduler capabilities to match. Currently, the problem representation framework does not allow certain concepts such as resettable capacities (e.g., the ability to empty a load) or multiple resources per task. When we put these into the problem representation, the scheduler algorithm needs to handle them. Second, we should make the scheduler smarter about handling special cases. If the scheduler could recognize special cases, then it could apply special-purpose, higher-performance algorithms for these cases. This would improve the performance of the scheduler without sacrificing its generality.

## Acknowledgments

This work was supported by DARPA contract MDA972-97-C-0800 under the Advanced Logistics Program. Thanks to Gordon Vidaver for his ideas and his help coding and to Todd Carrico for challenging us to build a generic scheduler.

## References

[Beasley, 1990] J. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[Davis and Fox, 1994] G. Davis and M. Fox. ODO: A constraint-based architecture for representing and reasoning about scheduling problems. In *Proceedings of the 3rd Industrial Engineering Research Conference*, 1994.

[Fourer *et al.*, 1993] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 1993.

[Giffler and Thompson, 1960] B. Giffler and G. Thompson. Algorithms for solving production-scheduling problems. *Operations Research*, 8(4):487–503, 1960.

[Lin and Kernighan, 1973] S. Lin and B. Kernighan. An effective heuristic algorithm for the TSP. *Operations Research*, 21(2):498–516, 1973.

[McIlhagga, 1997] M. McIlhagga. Solving generic scheduling problems with a distributed genetic algorithm. In *Proceedings of the AISB Workshop on Evolutionary Computing*, pages 85–90, 1997.

[Montana *et al.*, 2000] D. Montana, J. Herrero, G. Vidaver, and G. Bidwell. A multiagent society for military transporation scheduling. *Journal of Scheduling*, 3(4):225–246, 2000.

[Montana, 2001] D. Montana. A problem representation framework for a reconfigurable scheduler, 2001. Currently unpublished.

[Muth and Thompson, 1963] J. Muth and G. Thompson. *Industrial Scheduling*. Prentice Hall, 1963.

[Osman, 1995] I. Osman. Heuristics for the generalised assignment problem: Simulated annealing and tabu search approaches. *OR Spektrum*, 17:211–225, 1995.

[Smith and Becker, 1997] S. Smith and M. Becker. An ontology for constructing scheduling systems. In *Working Notes of 1997 AAAI Symposium on Ontological Engineering*, 1997.

[Solomon, 1987] M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations Research*, 35:254–165, 1987.

[Syswerda, 1991] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[Whitley *et al.*, 1989] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.