MAXIMIZATION OF ORGANIZATIONAL UPTIME USING AN INTERACTIVE GENETIC-FUZZY SCHEDULING AND SUPPORT SYSTEM

Roger M. Stein
Moody's Investors Service
New York

Vasant Dhar
Stern School of Business
New York University
New York

Working Paper Series
STERN IS-93-27

Maximization of Organizational Uptime Using an Interactive Genetic-Fuzzy Scheduling and Support System

Roger M. Stein
Moody's Investors Service
New York, New York

Vasant Dhar
Stern School Of Business
New York University

Abstract

This paper addresses the problem of scheduling multiple time and priority sensitive tasks efficiently in an environment where the number of resources is limited and the resources have varying capabilities and restricted capacities. We use a help desk environment as our working model, however, the methodology could also be adapted to a variety of job shop scheduling problems in general. We introduce a metric called priority time usage *as a measure of task urgency and of schedule efficiency. We also introduce a method of considering user satisfaction in scheduling by utilizing fuzzy monotonic reasoning. We propose a methodology for implementing a heuristic genetic algorithm (GA) to accomplish the scheduling task. We discuss how such a system can use ongoing data about historical schedule performance to adapt and create progressively more accurate schedules in the future. We consider modifications to the scheduling approach which could allow for task inter-dependencies. We present an intuitive user interface which we developed to aid help desk administrators in using the system. In addition to providing a front end to the SOGA system, the interface allows the user of the system to perform "what if" analysis with actual schedules. Lastly, we present preliminary assessments of the utility of both the optimization engine and the user interface.*

Keywords: scheduling, genetic algorithm, fuzzy logic, constraint satisfaction problems, help desk, optimization, heuristics, graphical user interface, hybrid expert system, monotonic reasoning.

1 Introduction

The effectiveness of an organization hinges on the quality of support it is able to provide to its infrastructure of equipment and employees. In particular, as computers, network equipment and other types of hardware and software become increasingly prevalent in organizations, there is an increasing and critical need to support such an infrastructure.

In large organizations, help desks are often implemented by MIS groups within the organization to aid computer users in resolving system problems as well as to provide a central point where users can report troubles. Simple problems are often resolved directly over the phone or in person at the time of reporting. However, more complex technical issues (tasks) must often be dispatched to hardware or software specialists (resources) for more in-depth diagnosis and resolution.

The central objective of the help desk is to maximize organizational productivity by minimizing down time of people, tools, or equipment. Tasks should be scheduled efficiently, according to some meaningful prioritizaton scheme, so that the amount of productive time lost by the community of users is minimized. In order to provide quality service, the prioritiation scheme must also attempt to minimize the total dissatisfaction of the user community. In addition, the scheduler must consider that the availability of the resources and that their ability to perform a given task can vary greatly. These factors depend on issues such as the experience, training, work schedules, and the other non-help-desk related commitments of each resource.

Where support is being provided to employees, a secondary objective may be to maximize user satisfaction by providing prompt service and estimates of expected start and completion times of tasks.

Both objectives can be achieved through judicious use of mathematical modeling and computer technology. In this paper, we use the help desk as an example of a prototypical support function. Such a function involves generating realistic plans and modifying them as new tasks are received and others are completed. While this might appear to be a relatively straightforward problem, it is exceedingly difficult to perform in practice. Inappropriate assignments or miscalculations can result in bad schedules which, in turn, result in longer down times than are

necessary, and where, users are involved, increased dissatisfaction, frustration, and loss of goodwill.

In this paper, we demonstrate one method for addressing the above mentioned objectives and providing a computer-based scheduling system that can be used for scheduling resources and continually updating schedules as the set of outstanding problems changes. The problem faced by help desks in large organizations is one such typical problem. However, we should point out that the approach described herein carries over to a variety of other domains where scarce resources must be continually scheduled to perform pending tasks in as optimal a manner possible.

The paper is divided into seven sections. The first through fifth sections deal largely with the theoretical properties of the methodology, while the remaining two sections discuss and report on a system that has been developed for a help desk environment based on this methodology.

The first section deals with a descriptions of the help-desk environment and a general statement of the scheduling problem. The second section introduces the *priority time usage* metric as a means for measuring schedule efficiency. The basic formulation of the scheduling problem is presented as well. The third section deals with genetic algorithms and the applicability of these techniques to the problem. The fourth section describes modifications that were made to account for the limited or partial availability of resources. The fifth section introduces modifications which, although not implemented, could be made and which might allow the system to be used to find solutions for the multiprocessor type problems where there are inter-task dependencies. The sixth section provides a description of an interactive user interface that the authors designed to facilitate the use of the optimization system. The interface allows a good deal of flexibility and the ability to perform "what if" analysis on schedules. The last section discusses preliminary findings.

1.1 The Help Desk Environment

In many large organizations, technology departments are large enough, and the computer user base is broad enough to warrant a special function within the organization that is responsible for user support and problem resolution. These support units, called help desks, can range in size from one or two specialists to much larger groups.

Typically, in a help desk environment problems are processed in a series of stages. Users contact the help desk with a specific problem. Where possible the problem is resolved at that time. However, more complex or time consuming problems are forwarded to specialists who visit the user site directly. In some cases, multiple visits are required before a problem can be resolved.

The challenge facing the help desk administrator is to schedule tasks in such a manner that the loss of value by the firm due to computer down time is minimized. Another concern is to minimize the dissatisfaction experienced by the user community regarding the quality and timeliness of support.

To minimize the first constraint, the administrator must consider the priority of the various tasks in the queue as well as the time each will take to resolve. The ability of the various technicians to perform the tasks will also impact schedule design. All things being equal, it makes little sense to have a highly experienced technician perform a relatively simple task while a more complex task remains undone because the other (idle) technicians do not have the requisite skill to perform it.

To address the constraint of user satisfaction, the administrator must also consider the amount of time that a given task has been outstanding since it was reported. As this time increases, the user will tend to become more and more disturbed by the slow response time of the help desk. The length of time which passes before the user registers dissatisfaction will vary with the nature of the task.

1.2 Scheduling Constraints

In most cases, not all resources can perform all tasks. Thus schedules must be constructed which only allow tasks to be assigned to resources that have the ability to perform them. In addition, not all resources are available at all times to perform tasks. Moreover, only legal, complete (wherever possible) schedules should be generated.

A legal schedule is one which does not violate any of the above constraints A complete schedule is one in which no assignable task is left un-assigned. A task is assignable if there is a resource which can perform the task available during the scheduling period.

2 Schedule Efficiency and Optimality

In order to optimize a system of schedules of tasks of varying priority, we must first characterize what we mean by an optimal schedule. We start with the far simpler special case in which we seek to optimize a system containing only a single resource. We then proceed to the more complex multiple resource system scheduling problem.

2.1 Efficiency of a Single Resource Schedule

To evaluate the efficiency of a given schedule, it is first useful to define several variables: Let $\mathbf{T}=\{t_1..t_n\}$ be a list of n

tasks to be done. We also define e_i as the estimated time to complete task t_i. We can now define a schedule $S = \{s_1 \ldots s_n\}$ to be a vector of integers where s_j is the *index* of the j^{th} task to be performed in schedule S. (So, for example, if s_2 were equal to 15, it would mean that the fifteenth task, t_{15}, would be performed second on the schedule.)

A schedule can be constructed such that the total time that passes before the completion the a task, c_j is defined recursively as:

$$c_j = \begin{cases} e_{s_j}, \text{ when } j=1 \\ c_{j-1} + e_{s_j}, \text{ otherwise.} \end{cases} \qquad \text{(Eq. 1)}$$

The total waiting time to complete a schedule of n items to is therefore simply equal to the finishing time of the last task, c_n. However, note that the total time lost by the *user community* is:

$$\sum_{j=1}^{n} c_j \ , \qquad \text{(Eq. 2)}$$

since each user must wait while all prior tasks are performed. That wait time is time lost by the user community and should be considered in assessing the efficiency of a schedule.

To appreciate this point, consider the following case: assume that there are two users with tasks, t_1 and t_2, with estimated times to completion e_1= 0:05 and e_2 = 5:00. It can be seen that the schedule S = {1,2} will have an estimated time cost of 5:10 hours, since both users must wait five minutes for the completion of t_1 and one user waits an additional five hours for the completion of t_2 in contrast, the schedule S = {2,1} will result in the much higher cost of 10:05 hours since both users must wait five hours, and one user must wait an additional five minutes.

2.2 Priority Time Usage

If we were concerned with minimizing the useful time lost before the completion all of the tasks in a schedule, we could seek to optimize the schedule by trying to minimize Eq. 2. However, Eq. 2 does not take into account the priority of a given task and is therefore useful only if all tasks are of equal importance or urgency. However, this is not usually the case. To incorporate priority into our scheduling evaluation paradigm we now define *priority time usage*, u_j, as the amount of priority weighted time lost by the community of users while waiting for the j^{th} item in S to be completed. To define u_j, let p_i be the priority of task t_i. We define p_i as the number of productive hours lost by the community of users for every hour that task t_i is unfinished. (p_i is defined more fully below.) For any s_j in S, u_j can be defined as:

$$u_j = c_j p_{s_j}. \qquad \text{(Eq. 3)}$$

where p_{s_j} denotes the priority of task s_j.

We can now calculate the *total* amount of priority time usage, U, for a schedule S as follows:

$$U = \sum_j u_j \ . \qquad \text{(Eq. 4)}$$

Our task in optimizing a schedule now becomes simply to minimize U. In doing so, we minimize the total value of the time lost by the user community while the tasks are performed.

2.3 Task Priority (p_i)

We now return to a more complete definition of p_i , the priority of a given task, t_i.

Let $\bar{h}$ be the value of one hour of the average user's time to the user community. Let h_k be the value of one hour of the k^{th} user's time. (Note that $h_k > \bar{h}$ when worker k's time is more valuable than average.) If we assume that there are m users affected by task t_i, and that the severity of task t_i is such that any user k who is affected by task t_i loses all ability to perform usefully until the task is completed then:

$$p_i = \sum_{k=1}^{m} h_k \qquad \text{(Eq. 5)}$$

A substantially similar, but somewhat more complex definition of p_i can be derived if the above assumption about the severity of time lost is relaxed.

If we define a series of m weights, w_k, representing the degree (0.0 to 1.0) to which task t_i causes a loss of functionality to user k (i.e.: the percentage of user k's job that the user is prevented from performing), we can then weight appropriately the impact of the task on the community of users as follows:

$$p_i = \sum_{k=1}^{m} w_k h_k \qquad \text{(Eq. 6)}$$

2.4 Efficiency of a Multiple Resource Schedule System

As we move from the special case of a single resource system to the more general case of a multiple resource system, there are addition factors to be considered.

Since we are assigning tasks to multiple resources we must assign them in such a way that the over all priority time usage to complete all tasks is minimized. If we have q resources, we create a multiple schedule system by creating a separate schedule $\mathbf{S}_I$ for each resource where I=1..q. (Each $\mathbf{S}_I$ is analogous to $\mathbf{S}$ in Section 2.1.) Tasks in $\mathbf{T}$ are then distributed among the various $\mathbf{S}_I$. Each schedule $\mathbf{S}_I$, is evaluated to yield a priority time usage $\mathbf{U}_I$. (Here again, each $\mathbf{U}_I$ is analogous to $\mathbf{U}$ in Section 2.2.) To determine the priority time usage of the multiple schedule system we calculate:

$$\sum_{I=1}^{q} U_I \qquad \text{(Eq. 7).}$$

To optimize the schedule, we must minimize Eq. 7.

2.5 User Satisfaction and Goodwill: A Fuzzy Set Representation

While minimizing Eq. 7 reduces the total priority hours lost by the user community, it does not consider the level of satisfaction and goodwill experienced by the user community.

Consider the case where there are two tasks t_1 and t_2 of equal priority and duration. Assume further that t_1 was initiated five hours earlier than t_2. From the standpoint of Eq. 7, we are indifferent as to whether we perform t_1 first or t_2 first. Since the five hour difference is time that has already been lost by the user community, it is not considered in our schedule formulation since it will have no impact on the future completion time of any new schedules. It is a sunk cost. Nonetheless, fairness might compel us to consider performing t_1 before t_2 since it was initiated first. Such decisions become more complex when priorities and durations are not equal.

Additionally, we might argue that as user satisfaction declines, so does productivity. A user whose task is continually postponed or placed at the end of task queues will become progressively more dissatisfied and less productive.

Even if this were not the case, we might be willing to compromise overall system efficiency as measured only by total priority hours lost so that we could increase overall user satisfaction with support.

To accommodate the concept of satisfaction, we can introduce two new measures: we define o_i as the amount of time that task t_i has been outstanding at the time of the formulation of the schedule; and we define the function g(k,t,o) which returns the level of *goodwill* or satisfaction ($0 < g(\cdot) \leq 1$) that user k experiences as a result of the fact that task t has been outstanding for time o. We can now offer an alternative formulation of Eq. 6:

$$p_i = \frac{\sum_{k=1}^{m} w_k h_k}{g(k, t_i, [o_i + c_j])}, \qquad \text{(Eq. 6a)}$$

where j is chosen such that $s_j = i$. Note that the outstanding time parameter passed to $g(\cdot)$ is considered to be the sum of both the total time that the task has been outstanding at the point of the creation of the schedule, *and* the total time that the user will have to wait ($o_i + c_j$) while other tasks are being processed.

If we elect to do so, substituting this alternative definition for p_i into Eq. 3 lets us consider the satisfaction of the user community due to unresponsive service as well as the total priority hours lost by the community when we optimize a schedule.

2.5.1 Properties of g(·)

Since g(k,t,o) reflects the level of satisfaction that user k experiences as a result of the status of task t for the time period o, there are several characteristics worth noting.

Firstly, the level of satisfaction will vary depending on the nature of the task. For example, what is considered to be a long time for a certain type of hardware repair, may not be considered to be a long time for a major installation.

Secondly, the degree to which $g(\cdot)$ varies with respect to o is often non-linear. A user may not be concerned if a certain task is not completed within some moderate time frame. However, as that time frame increases beyond some threshold, the rate at which a user's patience becomes exhausted may rapidly increase with time.

2.5.2 Fuzzy Logic

Fuzzy logic offers a natural means to encode $g(\cdot)$. It provides a framework for dealing with uncertainty. One of the premises of fuzzy logic is that most natural phenomena do not fall into crisp categories. In fact, most events and

objects occur in the gray areas between categories or where they overlap.

Fuzzy logic defines the degree to which the value of a variable is (partially) contained in a fuzzy set that describes that variable as the value's *membership* (μ) in the fuzzy set. μ is defined as continuous on [0,1]. μ=0.0 is equivalent to a Boolean value of FALSE, and μ=1.0 is similarly equivalent to a Boolean value of TRUE. All other values indicate partial membership in a set. A fuzzy set is a bi-directional mapping, either functional or pairwise, of values within the domain of the variable space to their fuzzy membership values [Zadeh].

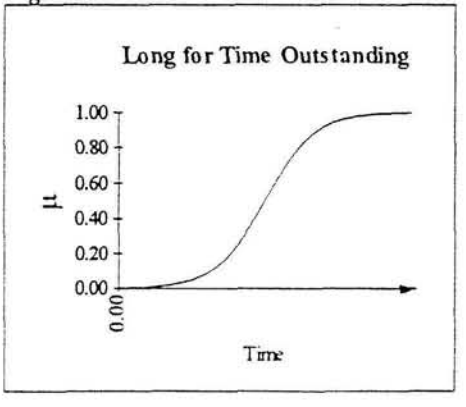
Monotonic reasoning allows the mapping from an antecedent clause of the form "IF $<v_1>$ is (in) $<F_1>$" to consequent clauses of the form "THEN $<v_2>$ is (in) $<F_2>$", where v_1 and v_2 are variables, and F_1 and F_2 are fuzzy sets.

The degree to which the consequent clause is executed (fired) will depend upon the level of membership that the value of $<v_1>$ has in the fuzzy set $<F_1>$. When the value of $<v_1>$ has a membership of μ in the fuzzy set $<F_1>$, the value of $<v_2>$ is derived by mapping μ into $<F_2>$.

2.5.3 A Fuzzy Set Representation of g(·)

To address these factors we chose to define g(·) as an object oriented fuzzy membership function. We define a fuzzy membership surface LOW for g(·) which ranges from zero to 1.00. We then define the fuzzy sets LONG for time outstanding (o) as appropriate for each of the classes of task. Figure 1 shows an example of a fuzzy set representing LONG for o.

Figure 1



Lastly we define the simple rule:

IF o is LONG (for task t)
THEN g(·) is LOW.

Applying monotonic reasoning, we can now infer the level of satisfaction for the time that a given task has been outstanding at any point in time. The shape of the membership surfaces of the fuzzy sets can take any form desired (sigmoid, step, exponential, etc.). These sets, of course, must be contemplated carefully so as to reflect the actual sentiments of the user community.

Here we note again that it is not *necessary* to include this measure in our formulation of U, only that if we elect to, we now have a means of doing so.

3 Genetic Algorithms

Genetic algorithms (GAs) have their basis in the biological metaphor of survival of the fittest. GAs have been found to be useful for finding good solutions for a wide variety of problems, including classes of problems that were previously computationally prohibitive [Davis, Goldberg, 1989a].

A genetic algorithm attempts to solve a problem by creating a range (*population*) of possible solutions. These usually take the form of strings. Each member of the population (an *individual*) is then interpreted and evaluated in the context of the problem and ranked in terms of its *fitness*. Fitness is an assessment of how well a particular individual solves the problem at hand. (In a biological context, the problem specifications can be seen as analogous to the environmental constraints brought to bear on an organism, and fitness as a measure of how well the organism survives in its environment.) The individuals are then matched with other individuals in the population such in a way that those with higher fitness are more likely to be selected (*fitness proportionate reproduction*). The results of this mating form the offspring that make up the population of the next *generation* and the process can be repeated with this new population.

During the reproduction process two operations take place: *mutation* and *crossover*. Mutation involves changing the value of an information unit (an *allele*) in an individual. Crossover involves the exchange of portions information between two individuals.

By mutating and crossing over, the GA is, in effect, experimenting with new solutions while preserving potentially valuable interim results or building blocks [Davis; Goldberg, 1989a; Goldberg, et. al, 1991; Goldberg, et al, 1992; Kargupta, et al]. If an experiment (crossover or mutation) fails (that is, produces a relatively unfit offspring), then the offspring will, in all likelihood, be dropped from the population within a few generations due to its inferior fitness. On the other hand, if the experiment is successful, then these new interim results can be passed on to the future generations for further refinement. Thus

the more promising areas of a solution space are explored, and lower payoff areas are examined more in a more cursory manner.

The genetic algorithm paradigm therefore allows the search of potentially huge problem spaces in a parallel and efficient manner. Because of the constant adjustments due to mutations and crossovers, the risk of converging to a local minima or maxima is low in comparison to many other methods, provided that the problem is coded sufficiently to avoid deceptive lower-order schema. [Goldberg, et al, 1992; Kargupta, et al, 1992].

3.1 Motivation for Using A GA for Optimization

Earlier research has shown that constrained problems, such as scheduling, lend themselves well to heuristic optimization methodologies [Dhar]. In contrast, applying mathematical optimization methods such as integer programming (IP) to such problems can sometimes result in unpredictable and, in some cases, unacceptable, execution time for optimization. This is particularly problematic when a solution must be recomputed frequently, as is the case when a scheduled is updated in response to changing task data.

In addition, the added precision which might be gained from the application of IP or other numerical techniques, is not necessarily required for a job scheduling application such as the one described in this paper. While fully optimal solutions are desirable, they are not essential in order for the application to be successful. In fact, a near optimal solution, if derived in reasonable time, will always suffice. While GAs often do not produce optimal solutions, if designed carefully, they have been shown to produce very good, near optimal, solutions.

Lastly, the nature of the help desk scheduling problem dictates that constraints be added, modified, and deleted with some regularity. In addition, the introduction of factors such as goodwill and the continuous nature of time in this domain make formulation of the problem difficult for some numerical techniques.

3.2 Genetic Scheduling Operators

Permutation problems, such as task scheduling, require a somewhat different set of operators than those found in traditional genetic algorithms.

Since, in these problems, the exact value of the of the individual tasks at a given position in a list is often not as important as their relative ordering [Davis; Goldberg, et al, 1992; Kargupta, et al], the goal of operators in these problems should be to preserve the ordering information within good strings while, at the same time creating strings which are not illegal.

We chose as operators variants of order-based mutation and order-based crossover as described in [Syswerda].

In addition to having the attractive property of operating on the ordering information contained in the string representations, these operators have the added advantage that they do not create illegal or incomplete lists from legal complete ones; that is, no tasks are deleted, no new tasks are created, and no tasks are duplicated.

The order based mutation operator works as follows: two tasks within a list are selected at random and their position in the task list are exchanged.

The order based crossover operator imposes the order of selected tasks in one parent on the other parent. That is, a subset of tasks is selected in one parent A, and those tasks are shuffled so that they are in the same order as in parent B.

A more complete treatment of these operators can be found in [Syswerda].

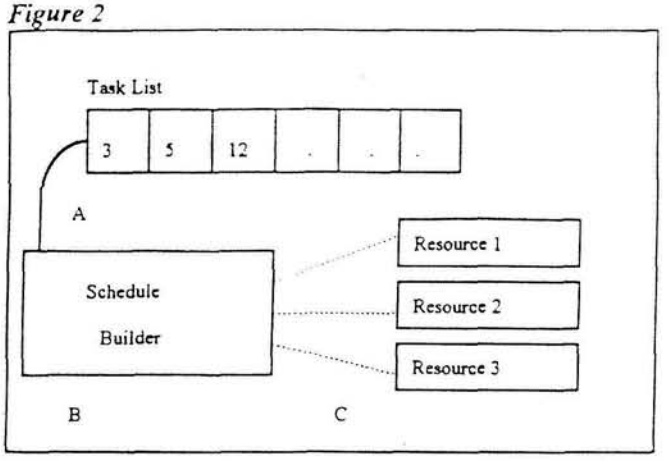
We use scaled fitness proportionate reproduction with partial steady state generational replacement. We always preserve the single individual from the previous generation with the highest fitness, a technique called *elitism*.

4 The Schedule Optimizing GA (SOGA)

We now apply the genetic algorithm approach to the problem of designing good schedules for the resolution of tasks in our environment. We start by defining a structure to hold our task list ($\mathbf{T}$). (All of the information needed for the calculation of u_j, as described above is assumed to be present.)

We also design a "greedy" schedule builder (similar, in some respects, to that used in [Whitley, et al]). The schedule builder reads tasks from the task list and heuristically assign each in order to a resource (see Figure 2). The schedule builder will only assign a task to a resource if the resource meets the constraints required by the task. If there is more than one resource that meets these constraints, the task is assigned to the least utilized resource. This results in a legal complete schedule system for any task list provided there are resources available with the ability to perform each of the tasks. If this is not the case, we add another "virtual" resource. This resource will serves to track all unassigned tasks. An appropriate penalty function is added to the calculation of $\mathbf{U}$ to compensate for this. This, however, is never the case in our environment.

Figure 2



To search for a good schedule, we:

1. Generate a random population of task lists.

2. After building a schedule system from each list, evaluate each schedule system using the objective function described in Eq. 7 (incorporating either Eq. 6 or Eq. 6a).

3. Using fitness based reproduction, perform genetic crossover and mutation on the original population to yield a new population.

4. Repeat steps 2 and 3 until an arbitrary desired fitness is achieved, the population converges, or some maximum number of generations has been created.

5. (Optional) After the global schedule has been optimized, repeat steps 1 through 4 for each sub-schedule, that is, for each resource's schedule. Instead of using all tasks in the task list, use only those assigned to the individual resource. When distributing tasks, only assign tasks to that resource. Repeat this for each resource.

4.1 Time constraints on resources

Due to the simplicity of both the objective function and the schedule building algorithm, additional constraints can often be added without a great amount of effort. For example, we wished to impose time availability constraints on the resources (i.e.: each resource has other obligations besides just resolving the tasks in **T**), so we designed a framework for describing the schedule of available and non-available times for each resource. Then, we modified Eq. 1 to also take into account the time spent waiting while the resource is unable to perform task resolution (i.e.: the resource is either absent or must perform other obligations, not related to task resolution). For example, assume that a resource was going to be absent for one hour between the end of the first task and the start of second tasks in his or her queue. The calculation of c_1, the completion time for the first task, would remain unchanged, but the calculation of c_2, would be extended for one hour to account for the dead time between tasks while the resource is scheduled to be absent.

Alternatively, if we had wished to make such a constraint a hard constraint, the schedule builder could have been modified such that it would only schedule tasks on a resource that had time to complete them fully before being called away. This later constraint, however, results in a schedule builder that is considerably more time intensive. Since this was not a requirement of our environment, we elected to implement SOGA without hard availability constraints.

Parenthetically, by implementing time availability functionality, we get the added benefit of being able to generate new schedules dynamically, as new tasks arrive. Since we have an estimate of the completion time of each task, e_i, we can, at any time during the execution of task t_i, approximate the time at which the resource performing the task will become available again, upon completion of the task. This is done by calculating the difference between the current time and the time at which task t_i was assigned. This value is then subtracted from e_i to give the time remaining until completion of a task. As new tasks arrive, we can thus calculate new schedules as necessary, incorporating the knowledge of the future availability of all resources in our planning.

4.2 Adapting Schedule Planning Over Time

The SOGA system relies upon estimates of task duration (e_i) in order to formulate schedules. The accuracy of these estimates impacts greatly the degree to which proposed schedules reflect the reality of the help desk environment. If these estimates are inaccurate, the resulting schedules will not make efficient use of the resources time.

In addition, as new task types are added to the environment, resources will not be familiar with the nature of these tasks. As a result, good estimates of time to completion are not usually available. Furthermore, the time that tasks take to complete will vary over time based upon such factors as the experience of resources, the introduction of new technologies, and the changing complexity of the user environment.

In short, estimates of task duration may be unstable over time. As a result, it would be useful for a scheduling system to dynamically adjust estimates of task duration in response to changing environments.

By providing SOGA with a link to a database of task case histories, the system will be able to perform simple statistical analysis across the database once it hs been sufficiently populated with historical data. This data can then be used to calculate more accurate estimates of task duration. In doing so, the system uses the historical experiences of a help desk to generate more accurate plans.

5.Similarity to the Multiprocessor Problem

The task scheduling problem can be seen as similar to the problem of efficiently assigning n tasks to q identical multiprocessors. In our case, we assign n tasks to q resources. The multiprocessor problem is *NP*-complete.

The complexity of the problem presented here is different than that of the problem described above. In the traditional multiprocessor problem, all processors are assumed to be identical. In our case, we have stated that the resources are, in fact, not identical. Certain resources can perform tasks that others cannot. On the other hand, in the classic multiprocessor problem, tasks may be dependent upon other tasks. This is currently not so in the formulation of our problem.

5.1 Introducing inter-task dependencies

This last constraint, that of inter-task dependencies, while not currently addressed, could be added to the problem described in Section 2 of this paper without losing the generality of the methodology.

To do this we could adopt a methodology which would allow the coding of precedence in schedules while maintaining our overall framework. One such methodology, a portion of which might suffice, is proposed in [Hou, et al]. In this methodology, each task is assigned a value representing its height in a task graph of dependencies. (A task dependent on no tasks takes a height of 0; a task dependent on one predecessor takes on a height of 1; a task with two predecessors takes on the height of the larger of the two, etc.)

We could then implement SOGA as before except that 1) prior to evaluation of each u_j we sort all tasks on each resource so that they are ordered in terms of height; and 2) when we calculate u_j we consider the additional idle time in cases where a task on one processor must wait for the completion of a task that is on another processor and on which it depends.

In choosing a sorting algorithm for item 2, above, we should take care to choose one that does not change the ordering of elements in the list that are already in the correct order, thus minimizing disruption to good strings.

We note that such an addition could, depending on the implementation, require that we modify or eliminate Step 5 of the algorithm described in Section 4, since we would now need to introduce global relationships among the resources in the system. In addition, the introduction of inter-task dependencies could result in added complexity and considerable overhead in terms of additional calculation time and bookkeeping.

In our environment, the occurrence of inter-task dependencies is rare.

6. The representational user interface

A secondary objective of the system was to provide for help desk administrators (HDAs) a mechanism whereby they could assess more easily the overall load of the help desk and better track problems as they progressed through the resolution queue.

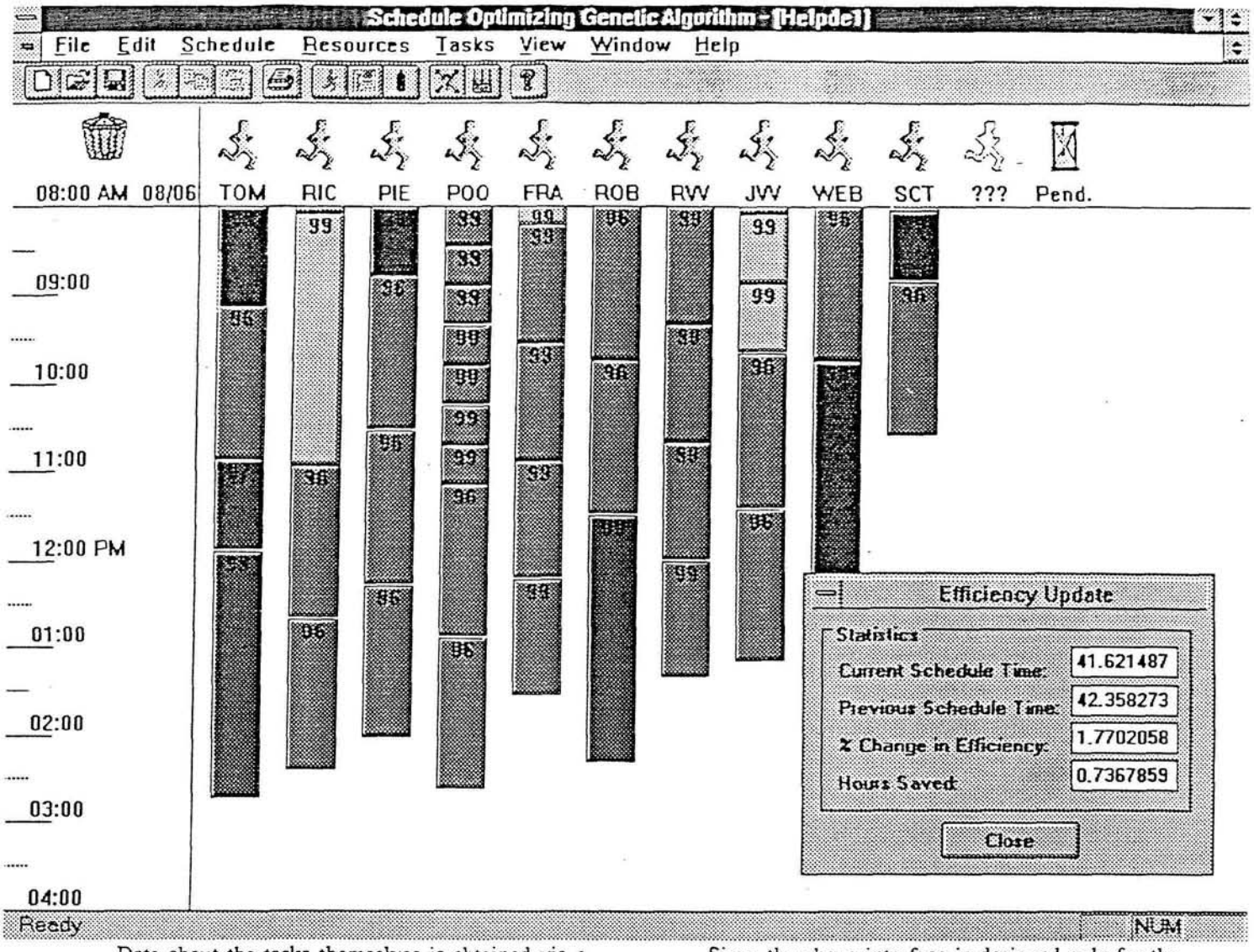
In addition we wished to provide the users with a means for modifying optimized schedules to fit unusual circumstances or non-explicit preferences. Such functionality is also useful in that it de-mystifies the underlying technology by putting the user, who may not be familiar with fuzzy logic or genetic algorithms, back in control of the scheduling process.

We designed a user interface that would allow the user of the SOGA system to manipulate tasks and resources in an interactive and intuitive manner. The interface is useful either in conjunction with the SOGA system, or as a stand alone tool. In practice, however, it is almost always used in conjunction with SOGA, rather than on its own.

The interface represents both tasks and resources as screen objects. The size of the task object is proportional to its estimated length (e_i). A rudimentary coloring scheme indicates the relative priority (p_i) of each task. Detailed information about each task and resource can be accessed by clicking with the mouse on the object of interest.

The user may manipulate the task objects by moving them between resources with the mouse. Doing so updates the data structures that these objects represent. In addition, a status window provides the user with feedback on the effect of a proposed change. This window provides information on constraint violations and on the overall number priority hours gained or lost by making a change. Fig. 3 shows a screen image of this portion of the interface.

This last feature allows the user of the SOGA system to perform "what if" types of analysis with actual schedules. The user can see the results of altering a schedule and determine whether the cost in priority time (and user satisfaction, if applicable) is worth the change.



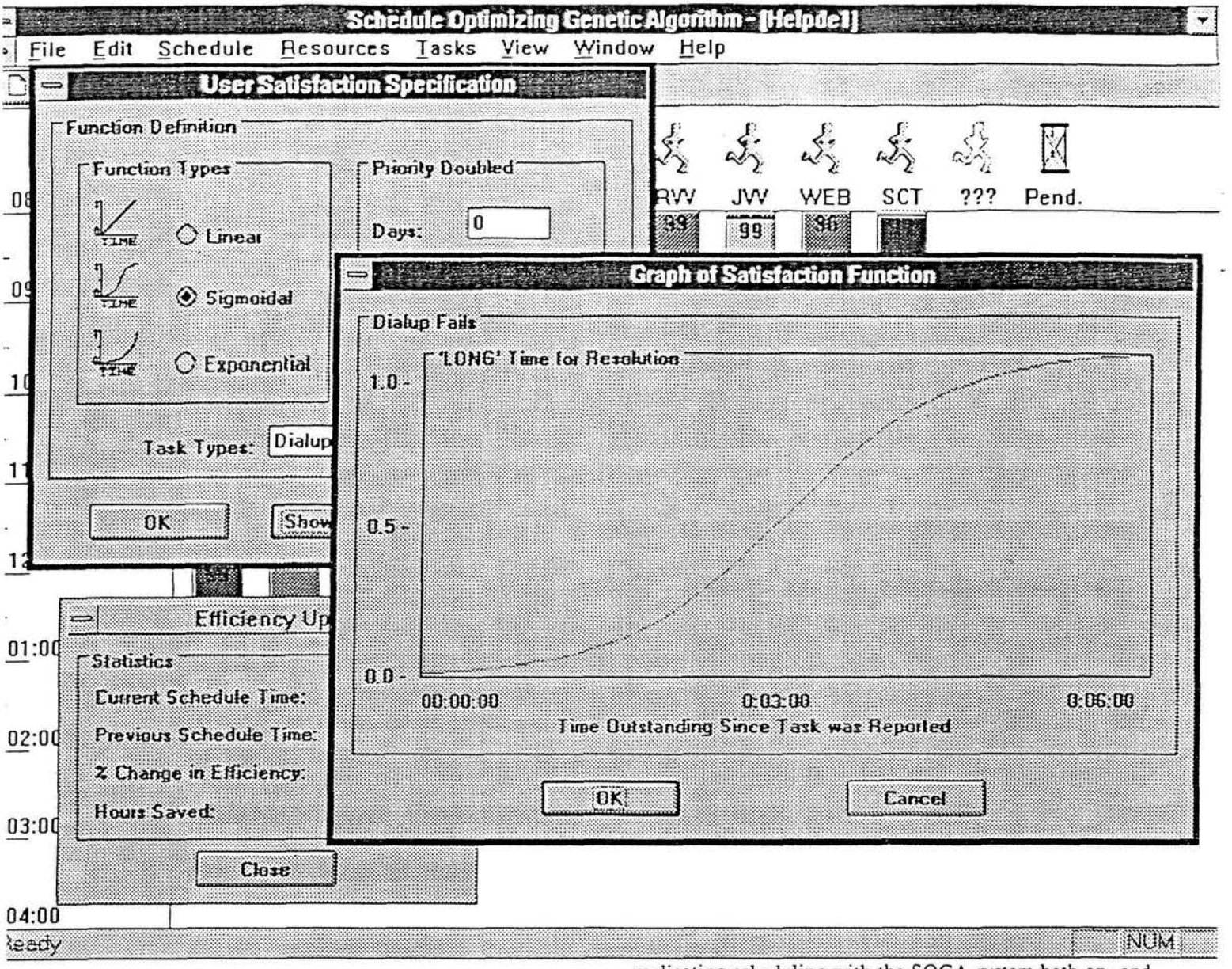
Data about the tasks themselves is obtained via a link to the help desk case database, a component of the existing problem entry and tracking system. However, knowledge about the resource proficiencies, average completion times, etc. is stored in separate proprietary tables used by the SOGA system.

In order to calculate the goodwill function, $g(\cdot)$, knowledge about the acceptable duration of tasks, and the fuzzy surfaces that represent them must be maintained. We have developed an intuitive user interface to allow users to define both the shape and parameters of these fuzzy sets. The user is able, through mouse clicks and movement, to manipulate and transform the shape of the fuzzy surfaces within the fuzzy sets, as well as the boundaries of the sets. Fig. 4 shows a screen image of this portion of the interface.

Other features of the interface include the ability to generate textual representations of schedules for any resource, dynamically locate schedule information about any task in the system, and the ability to manipulate parameters of the GA and fuzzy portions of the SOGA systems optimization engine.

Since the above interface is designed only for the use of the HDA, a separate interface component of the system, not discussed in detail in this paper, was designed for CSR use. It provides each CSR with access to the schedules generated by SOGA. This link allows CSRs to view their own current queues, and to accept, update, reassign, suspend, and close tasks in the queue as they work. Detailed information about tasks is available both on-line and in printed form.

Both components of the interface communicate with the SOGA optimization engine and the help desk case database. Thus the results of changes made by either the help desk administrator or by a CSR are visible dynamically to both parties, and the case database also reflects these changes.



7. Preliminary findings

The system has been very well received by the help desk administrators. Nonetheless, for reasons that will be enumerated below, it is difficult to make rigorous *quantitative* assessments of the system in terms of field performance.

Firstly, the nature of the help desk environment makes it difficult to run controlled experiments. An actual help desk is a dynamic system in which actors and events are constantly changing. Furthermore, at the time that the SOGA system was being developed, the particular help desk environment for which it was designed was undergoing a series of modifications, making isolation of influential factors difficult.

In addition, the primary objective of the help desk is to resolve user problems as quickly as possible, so replicating scheduling with the SOGA system both on- and off-line was not a viable option due to the cost in time. If the system were instead tested statistically on- and off-line over a period of time, the breadth and complexity of various tasks would require such testing to take place over a period of many months before reasonable analysis could be done. The users of SOGA felt that the system was useful enough to warrant on-line usage 100% of the time from a business perspective.

Secondly, historical comparisons are difficult since the schedules available in a historical database represent an audit trail of tasks as they were actually done, not as they were scheduled to be done. It is difficult to say how much more or less efficient a *proposed* schedule (with estimated times to completion, etc.) is in comparison to a schedule that was *actually executed* without also executing the proposed schedule itself.

Compounding this second issue, in evaluating the SOGA system, we faced problems in data sufficiency.

noise, and coarseness, in actual practice. In our specific case, the historical data was determined to be generally of too poor quality to yield statistically meaningful results for our purposes.

Thirdly, even if we were to have perfect historical data, we would still be faced with the problem of determining how to evaluate the successfulness of an historical schedule. One might suggest using the objective functions defined in this paper, but this is clearly unfair since SOGA is designed specifically to optimize **U**, while other methods are not. One would naturally expect SOGA to outperform other methods based on this criterion.

A serious complication arises in comparing historical schedules with those generated by SOGA. It is very difficult to determine whether the historical objective of the help desk was exactly the same as that of the GA optimization engine. Even if we assume that the historical objective was implicitly the same, it is difficult for us to claim that our method of *measuring* it, as described in this paper, is *the* correct method. Thus, while we hypothesize that the objective function that we have developed is robust and addresses the concerns of a help desk administrator, there may be other competing means of measuring the productivity other than maximization of uptime and goodwill of users.

Lastly, as alluded to in the beginning of this section, the users of the system felt that it was useful enough to be brought on-line after minimal testing. The system is currently in the process of being brought on line for field testing. As professionals faced with the job of serving a user community, their concern was and remains resolving the problems of that user community, not demonstrating the validity of the hypotheses set forth in this paper.

7.1 Comparison with original system

Useful insight can be gained by comparing the recording, scheduling, and tracking of tasks under the original system, versus under the new system incorporating SOGA and the interactive interface.

7.1.1 Task Scheduling Without SOGA

The following process describes how the help desk functions without SOGA.

A call is received by the help desk. Where possible, the call is resolved over the phone at the time of contact. Where the call is too complex, a determination is made as to the broad task type category to which the call belongs. Associated with each task type is a ranked (relative) priority. The call is then entered into a computer-based problem tracking system, with priority, description, and timing information. A paper document called a call ticket is generated and this is filed in one of about fifteen different folders in a public area within the help desk.

Customer Service Representatives (CSR) periodically scan the tasks in the folders for the broad types of tasks in which they are proficient and remove the task tickets for the problems that they are able to and elect to resolve. Preference may be given to the higher priority tasks. The CSR then logs the tasks that have been chosen into the tracking system thus, accepting responsibility for the task. Upon completion, the CSR logs completed tasks into the tracking system. The CSR usually logs tasks for an entire day at the same time.

The prior procedures, as described above, have several drawbacks. Firstly, the priority assignment system focuses on *relative* priorities of tasks rather than mapping priorities to an organizational cost. As a as a result, it is difficult to determine how much preference should be given tasks of varying priority when other factors such as duration and time in the queue needed to be considered.

Secondly, CSR personnel tend to focus on high priority tasks, regardless of their duration or time in the queue. As there are almost always many high priority tasks in the queue, often resulting in many of the lower priority, but nonetheless important, tasks being ignored for long periods of time, thus increasing the level of frustration that was felt by the user community.

Thirdly, the HDA, and other CSR personnel have little control over the overall efficiency of the task scheduling. In essence, they cannot "see the forest for the trees." CSRs are unable to consider fully the proficiencies and availabilities of other CSRs when they make their task selections, and it is difficult for the administrator to determine the overall load and characteristics of the task queue. This makes planning difficult and also makes it difficult to estimate when a given task will be started or completed.

Fourthly, CSRs exhibit preferences for some types of tasks over others, and this also impacts their scheduling decisions. Tedious or difficult tasks often get postponed in favor of more interesting ones.

Lastly, since the tasks types are not defined to a very low level of detail, and, since CSRs often do not log the completions of tasks as they occur, but rather in batch form at the end of the day, tracking trends and developing statistical analysis of problem behavior is difficult.

7.1.2 Task Scheduling Under SOGA

With the SOGA system and interface, tasks and resources are scheduled by the following process.

A call is received by the help desk. As before, where possible, the call is resolved over the phone at the time of contact. In cases where the call is too complex, a determination is made as to the *specific* task type category to which the call belongs. Cursory information about the caller is obtained and entered into a modified version of the tracking system. An initial priority is calculated for the task based upon the hourly cost of the downtime associated with the task to the user community (i.e.: priority time usage). Information for this calculation is retrieved from SOGA's proprietary tables. We note that with respect to priority time usage, our current implementation makes a simplifying assumption. Specifically, with respect to p_i in equations 6 and 6a, we currently assume that each task affects only one user.

The SOGA optimization engine runs in the background behind the tracking system. It updates schedules based upon a predefined time threshold (every fifteen minutes, for example). The HDA, through the interactive interface, has the option to change priorities, assign tasks to a resource, or rearrange tasks on the resource queues as needed.

CSRs access their current job queues through an interface in the tracking system, and are only allowed to accept a specified (small) number (two or three) of tasks at any time. If necessary, job tickets are printed at that time. When the jobs are completed, re-assigned, or suspended, CSRs log the status of the tasks either directly or remotely. Since CSRs can only have a limited number of tasks open at any given time, the system requires this logging to take place before assigning new tasks. This encourages CSRs to log tasks in a timely manner.

The SOGA optimization engine can use this historical data to better estimate task lengths going forward.

7.2 Improvements Under SOGA

The new procedures, as described above, address several of the weaknesses of the prior system.

The SOGA enhanced system allows priorities to be assigned automatically, and based on a consistent framework. The scheduling is done in a manner so as to favor global minimization of downtime while giving consideration to user satisfaction. Since duration in the queue and lengths of tasks are both considered in addition to priority when scheduling, the tendency for low priority tasks to be indefinitely postponed is reduced.

Furthermore, the function of determining priorities and ordering of tasks in the queue is now done by the SOGA optimization engine, with oversight from the help desk administrator. Because of the interactive interface, the administrator can get a broader and deeper view of the status of the job queue, as well as exercise better control over how and when the tasks are executed. This facilitates planning and allows some estimation of task start and completion times. This also reduces the tendency of CSRs to act on preferences for different tasks.

Lastly, since the tasks types have been redefined, and since the new system encourages prompt logging of completed tasks, tracking trends and developing statistical analysis of problem behavior at a much higher level of detail and accuracy is now possible.

8. Conclusions

We have presented a methodology for scheduling and tracking tasks that have varying priorities in an environment in which the resources that available to perform the tasks have differing abilities and limited time availability. This methodology has applications to a wide variety of problems. The help desk, which is an important support function in many large organizations, was used to illustrate the workability of our solution. We have introduced a metric called *priority time usage* which allows the ordering and distribution of tasks in a schedule of multiple resources to be evaluated in terms of the value of productive time lost, $\mathbf{U}$, by the user community. We next proposed an alternative formulation of this metric that also incorporates user satisfaction or goodwill in relation to the response speed of task resolution. We have proposed a fuzzy set based implementation of this alternative metric. We have described a system that uses such a heuristic genetic algorithm to minimize $\mathbf{U}$ for a given set of resources and tasks. While the current formulation of our problem does not call for them, we discussed possible modifications which would allow the system to be used to schedule tasks that have inter-task dependencies. This renders the problem similar to the multiprocessor problem. We presented an intuitive graphical user interface which acts as a front end for this system and goes beyond simple schedule optimization to allow the user to experiment and perform "what if" analysis with schedules. Finally, we discussed preliminary findings and user response to the system. We plan report further findings in future publications.

The authors wish to acknowledge the programming efforts of Spencer Kimball of UC Berkley. He was responsible for the implementation of large portions of the C++ code for the user interface.

References

Davis, L., (1991). *Handbook of Genetic Algorithms*, Van Nostrand Reinhold.

Dhar, V. and Ranganathan, N., (1990) Integer Programming vs. Expert Systems: An experimental comparison, *Communications of the ACM*, Vol. 33, No. 3.

Goldberg, D. E., (1989a). *Genetic Algorithms in Search, Optimization, & Machine Learning*, Addison-Wesley.

Goldberg, D. E., Deb, K., and Clark, J.H., (1991). *Genetic Algorithms, Noise, and the Sizing of Populations* (IlliGAL Report No. 91010), University of Illinois at Urbana-Champaign, Illinois Genetic Algorithm Laboratory.

Goldberg, D. E., Deb, K., and Horn, J., (1992). Massive multimodality, deception, and genetic algorithms, *Parallel Problem Solving from Nature, 2*, Manner, R and Manderik B., editors, Elevier Science.

Hou, E. S. H., Ren, H., and Ansari, N., (1992). Efficient Multiprocessor Scheduling Based on Genetic Algorithms, *Dynamic, Genetic, and Chaotic Programming*, Soucek, B and the IRIS Group, John Wiley & Sons.

Kargupta, H., Deb, K., and Goldberg, D. E., (1992). Ordering genetic algorithms and deception, *Parallel Problem Solving from Nature, 2*, Manner, R and Manderik B., editors, Elevier Science.

Syswerda, G., (1991). Schedule Optimization Using Genetic Algorithms, *Handbook of Genetic Algorithms*, Davis, L., Editor, Van Nostrand Reinhold.

Whitley, D. Starkweather, T., and Shaner, D., (1991). The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination, *Handbook of Genetic Algorithms*, Davis, L., Editor, Van Nostrand Reinhold.

Zadeh, L., (1965) Fuzzy Sets, *Information and Control*, Vol. 8.