NASA-TM-108120

IN-63-TM

135320

p-30

Scheduling and Rescheduling with Iterative Repair

MONTE ZWEBEN
EUGENE DAVIS
BRIAN DAUN
MICHAEL DEALE
AI RESEARCH BRANCH, MAIL STOP 269-2
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94025, USA

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-92-16

April, 1992

Scheduling and Rescheduling with Iterative Repair

Monte Zweben
Eugene Davis*
Brian Daun†
Michael Deale‡
NASA Ames Research Center
M.S. 269-2
Moffett Field, California 94035
zweben@ksc.arc.nasa.gov

Abstract

This paper describes the GERRY scheduling and rescheduling system being applied to coordinate Space Shuttle Ground Processing. The system uses *constraint-based iterative repair,* a technique that starts with a complete but possibly flawed schedule and iteratively improves it by using constraint knowledge within repair heuristics. In this paper we explore the tradeoff between the informedness and the computational cost of several repair heuristics. We show empirically that some knowledge can greatly improve the convergence speed of a repair-based system, but that too much knowledge, such as the knowledge embodied within the MIN-CONFLICTS lookahead heuristic, can overwhelm a system and result in degraded performance.

*Recom Technologies
†Recom Technologies
‡Lockheed Space Operations Company

Introduction

Space Shuttle ground processing encompasses the inspection, repair, and refurbishment of space shuttles in preparation for launch. The Kennedy Space Center (KSC) ground processing team frequently modifies the schedule in order to accommodate unanticipated events, such as unavailability of specialized personnel, unexpected delays, and the need to repair newly discovered problems. If the Space Shuttle ground processing turnaround time could be shortened, even by a small percentage, millions of dollars would be saved. This paper presents GERRY, a general scheduling system being applied to the Space Shuttle ground processing problem.

GERRY is a novel approach to scheduling that uses *constraint-based iterative repair* [1]. *Iterative repair* methods such as [1, 2, 3, 4, 5, 6, 7] differ from *constructive* scheduling methods in that they begin with a complete, but possibly flawed, set of assignments and then iteratively modify or repair those assignments to improve the overall schedule. Constructive scheduling methods [8, 9] incrementally extend valid, partial schedules until a complete schedule is synthesized or until backtracking is required.

In this paper we explore the tradeoff between the informedness and the computational cost of repair heuristics. We show empirically that knowledge can greatly improve the convergence speed of a repair-based system, but we also show that too much knowledge can overwhelm a system and result in degraded performance.

Problem Class: Fixed Preemptive Scheduling

Scheduling is the process of assigning times and resources to the tasks of a plan. Scheduling assignments must satisfy a set of domain constraints. Generally, these include temporal constraints, milestone constraints, and resource requirements. Temporal constraints relate tasks to other tasks; e.g., $end(T1) \leq start(T2)$. Milestone constraints relate tasks to fixed metric times; e.g., $end(T1) \leq$ 11/23/90 12 00 00. A resource requirement consists of a type and quantity of a resource; e.g., this task needs 4 mechanical technicians, 3 cranes. Each resource requirement has a corresponding capacity constraint which states that the resource must not be overallocated.

The Space Shuttle domain also requires the modeling of state variables.

State variables are conditions that can change over time; examples include the positions of switches, the configuration of mechanical parts, and the status of orbiter sub-systems. Tasks might be constrained by the state conditions (a *state requirement*) and they might cause a change in the state conditions (a *state effect*). A state requirement asserts that a state variable must have a certain value during a task's scheduled time. For example, in the Space Shuttle domain, certain tasks cannot be performed unless the payload bay doors are in designated positions. State effects are changes that tasks impose upon state variables such as the opening of the payload bay doors. The persistence of each effect is specified by the user.

Preemption is an additional complicating factor introduced by the Space Shuttle problem. In preemptive scheduling, each task is associated with a calendar of legal work periods that determine when the task must be performed. For example, suppose a task has a duration of 16 hours and a calendar indicating that only the first shift of each non-weekend day is legal. Given that the first shift of the day extends from 8:00 am to 4:00 pm, if the task is started on Monday at 8:00 am, then it will be suspended at the end of the shift (at 4:00 pm). It would restart on Tuesday at 8:00 am and would complete the same day at 4:00 pm. If the task had been started on Friday, however, it would not complete until the following Monday at 4:00 pm.

Preemption effectively splits a task into a set of subtasks. Resource and state constraints are annotated as to whether they should be enforced for each individual subtask (and not during the suspended periods between subtasks) or during the entire time spanning from the first subtask until the last (including suspended periods). Labor is a resource type that is not typically required during the suspended periods; in contrast, heavy machinery is difficult to relocate and thus may remain allocated during the suspended periods.

Preemptive scheduling requires additional computational overhead since for each task-time assignment, the preemption times must be computed and the appropriate constraint manipulation must be performed.

In summary, the input to a scheduling problem is a set of tasks, each with a work duration, a work calendar, a set of temporal constraints, a set of resource requirements, and a set of state requirements and effects. A solution to the problem is a decomposition of each task into its preempted subtasks, where each subtask is assigned a start time, an end time, and a resource pool for each resource request. A solution is satisfied if each subtask is consistent with its preemption work calendar and if all temporal, resource, and state constraints are satisfied.

Iterative Repair versus Constructive Methods

In previous work we concentrated on constructive methods [10, 11] which were difficult to adapt to the Space Shuttle problem. Space Shuttle ground processing is predominately a rescheduling problem; to reschedule with a constructive method, the system must remove some tasks from the schedule and then restart the scheduling process. Unfortunately, determining which tasks to *unschedule* is not straightforward. As a result we opted for a repair method that patches an existing schedule when exogenous events occur, thus circumventing the need to unschedule tasks. Further, even though a task is unaffected by an exogenous event, it may be possible to provide a better schedule by reconsidering its assignments. For example, placing an unaffected task much later in the schedule might cause little perturbation and allow many tasks (which are affected by the exogenous events) to fit in its place. Unfortunately, this opportunity would be missed if the unaffected tasks are not considered in the rescheduling process.

We also found it difficult to use constructive methods since Space Shuttle problems are over-constrained. For instance, milestones may be overly ambitious and impossible to meet when considering the other constraints. Additionally, the other constraints may be too conservative. For example, suppose there are two tasks each requiring a quality assurance officer. If these two tasks are physically proximate, then one of the officers might be able to handle both jobs, thus relieving the other officer for other procedures. When the problem is over-constrained, a constructive method must exhaust all possibilities before it can infer that constraints must be relaxed. Repair-based methods attempt to iteratively improve solutions regardless of whether the problem is over-constrained or not and terminate with a set of assignments that is as close to a solution as could be derived in the time allotted.

Since repair methods search through a space of complete schedules, "global" constraints and optimization criteria can be cheaply evaluated. If one only has a partial schedule, then the evaluation of global criteria can only be approximated. For example, suppose that in a particular domain it is desirable to minimize the use of labor resources on the weekend (which is a global optimization criterion) and that a particular machine is not allowed to change configuration more than a certain number of times per month (which is a global constraint). The evaluation of this criterion and constraint is easily calculated with a complete schedule but can only be estimated (based on the remaining tasks and possible times) with a partial schedule.

Repair methods also have their shortcomings. One problem is that repair methods could suffer from local minima in the sense that they can cycle indefinitely through a set of unsatisfactory solutions. Another problem is that repair methods are usually not complete and therefore not guaranteed to encounter the best possible solutions.

Constraint-Based Iterative Repair

Constraint-based iterative repair begins with a complete schedule of unacceptable quality and iteratively modifies it until its quality is found to be satisfactory. The quality of a schedule is measured by the *cost* function: $cost(s) = \sum_{c_i \epsilon constraints} penalty_{c_i}(s) * weight_{c_i}$, which is a weighted sum of constraint violations. The *penalty* function of a constraint returns a non-negative number reflecting the degree to which the constraint is violated. The *weight* function of a constraint returns a non-negative number representing the importance or utility of a constraint.

In GERRY, repairs are associated with constraints. Local repair heuristics that are likely to satisfy the violated constraint can then be encoded without concern for how these repairs would interact with other constraints. Of course, local repairs do occasionally yield globally undesirable states, but these states, if accepted (see below), are generally improved upon after multiple iterations.

Repairing any violation generally involves moving a set of tasks to different times; at least one task participating in the constraint violation is moved, along with any other tasks whose temporal constraints would be violated by the move. In other words, all temporal constraints are preserved after the repair. We use the Waltz constraint propagation algorithm over time intervals [12, 13] to carry this out (thus enforcing a form of arc-consistency [14, 15]). The algorithm recursively enforces temporal constraints until there are no outstanding temporal violations. This scheme can be computationally expensive, since moving tasks involves checking resource constraints, calculating preemption intervals, etc.

At the end of each iteration, the system re-evaluates the cost function to determine whether the new schedule resulting from the repairs is better than the current solution. If the new schedule is an improvement, it becomes the current schedule for the next iteration; if it is also better than any previous solution, it is cached as the best solution so far. If it is not an improvement, it is either accepted anyway with some probability described below, or it is rejected and the changes are not kept. When the changes are not kept, it is hoped that repairs in the next iteration will select a different set of tasks to move and the cost function will improve.

The system sometimes accepts a new solution that is worse than the current solution in order to escape local minima and cycles. This stochastic technique is referred to as simulated annealing [16]. The escape function for accepting inferior solutions is: $Escape(s, s', T) = e^{-|Cost(s)-Cost(s')|/T}$ where T is a "temperature" parameter that is gradually reduced (i.e. *cooled* during the search process. When a random number between 0 and 1 exceeds the value of the escape function, the system accepts the worse solution. Note that escape becomes less probable as the temperature is lowered.

In GERRY, the types of constraints that can contribute to the cost function include the resource and state constraints.[1]

Resource Constraints

The penalty of a resource capacity constraint is 1 if the resource is overallocated. If K simultaneous tasks overallocate the resource, then all K tasks are considered violated. One of these tasks will be selected in an attempt to repair as many of the K violations as possible. The heuristic used to select this task considers the following information.

Fitness: *Move the task whose resource requirement most closely matches the amount of overallocation.* A task using a significantly smaller amount

[1]We have also experimented with a number of other optimization constraints, the description of which are beyond the scope of this paper. In [17], we demonstrate the ability to reduce perturbation in rescheduling problems. We have also demonstrated the ability to reduce the number of weekends in a schedule resulting in lower overtime labor costs.

is not likely to have a large enough impact on the current violation being repaired. A task using a far greater amount is more likely to be in violation wherever it is moved.

Temporal Dependents: *Move the task with the fewest number of temporal dependents.* Moving a task with many dependents is likely to cause temporal constraint violations and result in many task moves.

Distance of Move: *Move the task that does not need to be shifted significantly from its current time.* A task that is moved a greater distance is more likely to cause other tasks to move as well, increasing perturbation and potentially causing more constraint violations.

For each of the tasks contributing to the violation, the system considers moving the task to its *next earlier* and *next later* times such that the resource is available, rather than exploring many possible times. This reduces the computational complexity of the repair to be linear in the number of tasks and, like the "distance to move" criterion above, tends to minimize perturbation.

Each candidate move is scored using a linear combination of the *fitness*, *temporal dependents*, and *distance to move* heuristic values. This calculation is evaluable in time proportional to the number of tasks. The repair then chooses the move stochastically by converting each score into a probability, and a method is selected based on these probabilities. After the repair is performed, the Waltz algorithm moves other tasks in order to preserve temporal constraints.

In summary, this repair strategy only considers two possible moves for a task participating in a violation: one earlier and one later. The evaluation criterion used to select a repair is based upon three computationally inexpensive heuristic criteria: degree of fitness, number of temporal dependents, and distance to move.

State Constraints

When a required state is not set correctly, the penalty of the associated state constraint returns 1. To repair a state constraint, either the task with the violated state requirement is reassigned to a time when the state variable takes on the desired value, or a new task that achieves the correct state at the appropriate time is added to the schedule. The insertion of new tasks is analogous to the operations performed by traditional planning systems [18, 19, 20].

Specifically, the system selects from the following possible repairs:

1. Insert a new task that sets the state correctly from the start-time to the end-time of the violated task.

2. Move the violated task forward to a time where the constraint is satisfied.

3. Move the violated task forward to a time where the state can be changed (by a new task) without causing additional state violations. Then insert the new task, thus changing the state for at least the duration of the violated task.[2]

4. Move the violated task backward to a time where the constraint is satisfied.

5. Move the violated task backward to a time where the state can be changed without causing additional state violations. Then insert the new task with an effect that will change the state for the violated task.

If the first method is successful, it is selected. If it is not applicable, then one of the other methods will be selected stochastically. Each is given a score based on the distance that the task must be moved to fix the violation and whether any temporal dependents would have be moved.

To summarize, constraint-based iterative repair begins with a complete but flawed schedule and isolates the violated constraints. Tasks are moved according to the repairs associated with the violated constraints. A new schedule is accepted if the new cost is lower than the previous cost, or if a random number exceeds the value of the escape function; otherwise it is rejected and new repairs are attempted on the previous schedule. The process repeats until the cost of the solution is acceptable to the user, or until the user terminates the repair cycle. The system may also terminate itself if a prespecified number of iterations have been attempted or if a prespecified CPU time bound has been reached.

[2]The persistence of the effect is a function of the attribute in question. For example, when the power onboard the orbiter is turned on, it remains on for an entire 8 hour shift because it is costly to repeatedly cycle the power.

Informedness versus Computational Cost

Repair methods differ in the amount of knowledge they exploit to modify a solution. Using knowledge is not free – computational overhead is incurred to evaluate and use repair knowledge. More informed methods also tend to be more expensive. This is analogous to the *utility problem* of machine learning [21] which states that learning can degrade the performance of a problem solver if the learned knowledge is not useful. Similarly, a knowledge-intensive repair method could degrade the problem solver if the method is overly expensive and does not provide enough heuristic power to compensate for its expense.

One can view a repair method as a generate-and-test process. The generator takes as input a schedule and suggests possible modifications. The tester then selects and performs one of the suggested modications. Knowledge can be exploited in both the repair generator and the repair tester. For example, in GERRY, the generator incorporates constraint knowledge to greatly restrict the possible tasks and times to consider. Then the system biases a stochastic choice with heuristics such as *fitness*, *number of temporal dependents*, and *the distance to move*. In contrast, the MIN-CONFLICTS repair method [2] uses a more computationally expensive value selction heuristic for repairs. Once a task is selected for repair, the MIN-CONFLICTS heuristic tries all possible times and selects the time that minimizes the number of remaining constraint violations. Ties are broken randomly. A system using MIN-CONFLICTS exploits *lookahead*, whereas GERRY exploits constraint knowledge. This lookahead procedure is quite effective at choosing the best repair, but it does incur substantial computational expense. This tradeoff is the subject of our investigation.

Experiments

In our experiments we intend to show that no one repair method is superior to all others on a particular class of scheduling problems. To investigate this tradeoff we contrasted four different repair methods. These methods are listed below in order of increasing informedness.

random repair: The system randomly selects a task to reassign and then selects a random assignment for that task between its earliest and latest start times.

random constraint repair: The system behaves identically to the random repair method except that it only repairs tasks associated with violated constraints. This repair exploits the blame assignment quality of constraint representations because it focuses the repairs on those tasks involved in constraint violations.

heuristic repair: The system repairs ten random constraints per iteration using the heuristic constraint knowledge discussed earlier to generate and select candidate repairs.

lookahead repair: The system uses the same constraint knowledge as the heuristic repair method to generate repairs but then instead of scoring them, it performs lookahead. It tries each generated repair and selects the one resulting in the lowest cost. This method is a form of the MIN-COFLICTS heuristic that exploits constraint knowledge to restrict the candidates for lookahead.

We compare these methods on both scheduling and rescheduling problems and on both artificially generated and actual Space Shuttle data.

Problem Generation

Our random problem generator creates data sets according to the following criteria and default values/ranges. The default values are in parentheses and default ranges are in brackets.

Number of tasks: The total number of tasks in the problem set.

Resource requirement probability: The probability that a given task requires resources. (.5)

Number of resource requirements: For a task that requires resources, the number of requirements that a task requests. [1 3]

State requirement probability: The probability that a given task has state requirements. (.5)

Number of state requirements: For a task that will have state requirements, the number of state requirements that a task requests. [1 2]

State effect probability: The probability that a given task has state effects. (.4)

Number of state effects: For a task that will have state effects, the number of state effects that a task causes. [1 2]

Probability of persistence: The probability that a state effect will persist from the start to the end of the task (0.7) or from the end of the task until some other effect clips the state (0.3)

The number of resource classes: The number of resource classes a task can request. [3 3]

The number of attributes: The number of attributes that can be constrained or affected by tasks. [2 2]

The number of states: The number of legal states for each attribute. [2 2]

Task duration: The work time required for a task. [1 hour to 24 hours]

Parallelism: The degree of parallelism in the schedule. Higher degrees of parallelism make resource and state conflicts more likely. Parallelism ranges from .1 (parallel) to .9 (serial). (.3)

Due Date: The milestone of the schedule is set to some amount past the earliest schedule possible (relaxing all resource and state constraints). The amount is calculated as a percentage of the length of the earliest schedule. For example, a due date at x% means that if time T_{start} is the start of a schedule and $T_{early-end}$ is the earliest possible end of the schedule, then the due date is set at
$T_{early-end} + (1 + x/100)(T_{early-end} - T_{start})$.

The quantity of each resource request is uniformly drawn from the capacity of the resource. The state required for each state requirement is uniformly drawn from the possible states of the attribute.

All of the scheduling (as opposed to rescheduling) experiments used the artificially generated problems. The two independent variables varied were the number of tasks and the due dates. We show the effectiveness of the various methods as problems get larger (in terms of numbers of tasks) and as they get more constrained (in terms of tighter due dates).

Four problems were generated, each with a different number of tasks (20, 50, 100, 500). The 20-task and 50-task problems used the default settings described above. Since the random repair strategies only shuffle tasks and do not insert new tasks, it is generally impossible for them to solve problems with state requirements. Consequently, the 100-task and 500-task problems did not contain state requirements or effects so that we could experiment with the random strategies. The 100-task and 500-task problems each drew from five different resource classes and five attributes (which is slightly larger than the default settings).

Experiments were run while fixing the final tasks's due date at three different settings: "underconstrained," "moderate," and "overconstrained." The percentages corresponding to these qualitative measures were 30%, 50%, and 70% for the 20-task problem, 50%, 100%, and 150%, for the 50-task problem, 30%, 50%, and 100% for the 100-task problem, and finally, 150%, 200%, and 250% for the 500-task problem.

All of the rescheduling experiments used Space Shuttle data. One data set, corresponding to the STS-43 mission of the orbiter Atlantis, contains only resource constraints. There are 414 tasks, 620 temporal constraints, and 3436 resource constraints in the initial STS-43 schedule. In the other Space Shuttle dataset, corresponding to the STS-50 mission of the orbiter Columbia, there are only state constraints. The initial STS-50 schedule contains 1453 tasks, 1761 temporal constraints, 11639 state requirements and 4064 state effects. The integrity of our datasets is improving over time as we acquire more knowledge from the Kennedy Space Center experts. A rescheduling experiment is generated by moving 10 pending tasks (either later – 90% or earlier – 10%) by a random amount (at most a week). Then the scheduler is invoked to resolve any conflicts created the moves. The STS-43 experiments compared all four repair methods and the STS-50 experiments omitted the random strategies (because of state constraints).

In our experiments, we found that an effective cooling strategy for the random techniques is not necessarily effective for more informed repair methods. Consequently, for each method, we employed the cooling strategy that performed best experimentally. For the random techniques, the initial temperature was the initial cost and the temperature for the i^{th} iteration was: $T_i = .95T_{i-1}$. For the heuristic and lookahead techniques, the starting temperature was 100 and after a few iterations it was reduced to 75. When the cost was less than 10, the temperature remained constant at 25.

All experiments were run on a Sun SPARCstation 2 with 32MB of memory. Each experiment ran until there were no outstanding violations or a 30-minute CPU time bound was reached. Since the repair functions are probabilistic, we calculated average results over at least 10 repeated trials for each experiment. In the next section we present the results of these experiments.

Empirical Results

In Figure 1 we graph the average best cost as a function of time for the four generated problems. In these graphs, the "moderate" milestone setting was used. This figure shows how each technique fares as problem size grows. In the smaller problems, the lookahead technique is competitive with heuristic repair; however, in larger problems lookahead falls behind due to its increasing evaluation expense. The random repair technique is a clear improvement over the purely random technique, but does not have sufficient heuristic power to converge within the time bound.

Figure 2 shows average best cost against time for the 50-task problem with increasingly tighter due dates. These graphs show that as the problem becomes more constrained, the lookahead technique fares relatively better on average. In the highly constrained graph, we can see a "crossover" behavior, wherein the heuristic repair technique quickly brings the cost to a low value, but has a hard time going much further. In contrast, the lookahead technique requires more time to get to the same low cost, but can find better schedules more efficiently beyond that point.

Figure 3 shows representative behavior from our rescheduling tests on STS-43 and STS-50. In the STS-43 tests, the heuristic repair technique was superior, while in the STS-50 tests, heuristic repair's advantage is not as clear. We believe that the competitiveness of lookahead on STS-50 is because there are fewer candidate repairs to explore on average when repairing a state constraint as opposed to a resource constraint. Consequently, it is computationally less expensive to perform this limited form of lookahead.

Figure 4 presents data for the entire experimental suite (averaged over the repeated trials) for heuristic repair and lookahead. Average times to solution are presented for those runs that reached a zero cost before the 30-minute time bound. For problems that did not converge on every run, the table shows the percentage of runs that did not reach a cost of zero and the average best cost found for those non-converging runs. Again, the data shows that for most of the experiments run, especially the larger problems, heuristic value selection outperforms value selection through lookahead.

Conclusions and Future Work

Our experiments suggest that our overall constraint framework and the knowledge encoded in this framework constitute an effective search tool, especially on large problems. The framework is modular and extensible in that one can declare new constraints as long as their weight, penalty, and repair functions are provided. Surprisingly, simple random shuffling of tasks associated with violated constraints can produce reasonable performance on problems of moderate size and difficulty. Lookahead techniques are especially effective on more difficult and smaller problems, but do not fare as well on large problems. Our repair method was superior to the other methods on the Space Shuttle rescheduling problems.

In future experiments, we hope to better characterize the components of repair informedness and computational complexity. We are currently evaluating candidate metrics of problem difficulty that could be used to guide the selection of repair heuristics. Additionally, we are developing machine learning techniques that allow systems to learn when to switch dynamically between heuristics [22].

With respect to the Space Shuttle application, the system is in daily use in support of the Space Shuttle Columbia. The KSC project team updates and publishes schedules 4 times daily under strict real-time constraints. At the current time we publish violation reports and suggest "deconflicted" schedules to Wayne Bingham, Vehicle Operations Chief. He then decides whether to accept the proposed schedule modifications. Our most significant barrier is gathering accurate models of tasks in an electronic form. We plan to fully deploy the system by the end of the year.

Related Work

Our work was heavily influenced by previous constraint-based scheduling [8, 23, 9] and rescheduling efforts [24].

ISIS [8] and GERRY both have metrics of constraint violation (the *penalty* function in GERRY) and constraint importance (the *weight* function in GERRY). In contrast with our repair-based method, ISIS uses an incremental, beam search through a space of partial schedules and reschedules by restarting the beam search from an intermediate state.

OPIS [23, 24], which is the successor of ISIS, opportunistically selects a rescheduling method. It chooses between the ISIS beam search, a resource-based *dispatch* method, or a repair-based approach. The *dispatch* method concentrates on a bottleneck resource and assigns tasks to it according to its dispatch rule. The *repair* method shifts tasks until they are conflict-free. These "greedy" assignments could yield globally poor schedules if used incorrectly. Consequently, OPIS only uses the dispatch rule when there is strong evidence of a bottleneck and only uses the repair method if the duration of the conflict is short. In contrast, GERRY uses the simulated annealing search to perform multiple iterations of repairs, possibly retracting "greedy" repairs when they yield prohibitive costs.

Our use of simulated annealing was influenced by the experiments performed in [25, 26]. In contrast with our constraint-based repair, their repairs were generally uninformed.

The repair-based scheduling methods considered here are related to the repair-based methods that have been previously used in AI planning systems such as the "fixes" used in Hacker [27] and, more recently, the repair strategies used in the GORDIUS[28] generate-test-debug system, in the PRIAR plan modification system [29], and the CHEF cased-based planner [30].

In [2], it is shown that the MIN-CONFLICTS heuristic is an extremely powerful repair-based method. For any violated constraint, the MIN-CONFLICTS heuristic chooses the repair that minimizes the number of remaining conflicts resulting from a one-step lookahead. However, in certain circumstances this lookahead could be computationally prohibitive, as demonstrated in the experiments discussed above. Also, the technique used in [2] can only escape local minima by restarting.

Our technique is also closely related to the Jet Propulsion Laboratory's OMP scheduling system [3]. OMP uses procedurally encoded patches in an iterative improvement framework. It stores small snapshots of the scheduling process (called *chronologies*) which allow it to escape cycles and local minima.

Miller et al. [31], Currie and Tate [32], and Drummond and Bresina [33] describe other efforts that deal with resource and deadline constraints.

Acknowledgements

Thanks to Ellen Drascher and Mark Yvanovich for the superb user interface they implemented and to Donna Kautz and Danielle Schnitzius, for all their knowledge engineering and operational testing at KSC. Special thanks to Eric Clanton, Flow Manager of the orbiter Endeavour, and Wayne Bingham, Vehicle Operations Chief of the orbiter Columbia, for their patience and help in our Space Shuttle application effort. Thanks to Steve Minton for his assistance throughout these experiments and for review of the paper. Thanks also to Peter Friedland, Mark Drummond, and John Bresina for careful review of this paper.

References

[1] Zweben, M., Deale, M., Gargan, R., "Anytime Rescheduling," in *Proceedings of the DARPA Workshop on Innovative Approaches to Planning and Scheduling*, 1990.

[2] Minton, S., Phillips, A., Johnston, M., Laird., P., "Solving Large Scale CSP and Scheduling Problems with a Heuristic Repair Method," in *Proceedings of AAAI-90*, 1990.

[3] Biefeld, E. and Cooper, L., "Bottleneck Identification Using Process Chronologies," in *Proceedings of IJCAI-91*, (Sydney, Austrailia), 1991.

[4] Lin, S, Kernighan, B., "An Effective Heuristic for the Travelling Salesman Problem," *Operations Research*, vol. 21, 1973.

[5] Kurtzman, C.R. and Aiken, D.L., "The Mfive space station crew activity scheduler and stowage logistics clerk," in *Proceedings the AIAA Computers in Aerospace VII Conference*, (Monterey, CA), 1989.

[6] Johnson, D.S. and Aragon, C.R. and McGeoch, L.A. and Schevon, C., "Optimization by simulated annealing: An experimental evaluation, Part II," *Journal of Operations Research*, 1990.

[7] P. Morris, "Solutions without exhaustive search: An iterative descent method for binary constraint satisfaction problems," in *Proceedings the AAAI-90 Workshop on Constraint-Directed Reasoning*, (Boston, MA), 1990.

[8] Fox, M., *Constraint-Directed Search: A Case Study of Job Shop Scheduling.* Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1987.

[9] Sadeh, N. and Fox, M. S., "Preference Propagation in Temporal/Capacity Constraint Graphs," tech. rep., The Robotics Institute, Carnegie Mellon University, 1989.

[10] Zweben, M. and Eskey, M., "Constraint Satisfaction with Delayed Evaluation," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (Detroit, MI), 1989.

[11] Eskey, M. and Zweben, M., "Learning Search Control for a Constraint-Based Scheduling System," in *Proceedings of AAAI-90*, (Boston, MA), 1990.

[12] Waltz, D., "Understanding Line Drawings of Scenes with Shadows," in *The Psychology of Computer Vision* (P. Winston, ed.), McGraw-Hill, 1975.

[13] Davis, E., "Constraint Propagation with Interval Labels," *Artificial Intelligence*, vol. 32, no. 3, 1987.

[14] Mackworth, A.K., "Consistency in Networks of Relations," *Artificial Intelligence*, vol. 8, no. 1, 1977.

[15] Freuder, E. C., "A Sufficient Condition for Backtrack-Free Search," *J. ACM*, vol. 29, no. 1, 1982.

[16] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, 1983.

[17] Zweben, M., Davis, E., Daun, B., Deale, M., "Rescheduling with Iterative Repair," in *Proceedings of the AAAI 1992 Spring Symposium on Practical Approaches to Scheduling and Planning*, (Stanford University), 1992.

[18] Fikes, R.E., Hart, P.E., and Nilsson, N.J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, vol. 3, 1972.

[19] D. E. Wilkins, "Domain independent planning: Representation and plan generation," *Artificial Intelligence*, vol. 22, 1984.

[20] Chapman, D., "Planning for Conjunctive Goals," *Artificial Intelligence*, vol. 32, no. 4, 1987.

[21] Minton, S., *Learning Effective Search Control Knowledge: An Explanation-based Approach.* PhD thesis, Carnegie Mellon University, 1988.

[22] Zweben, M., Davis, E., Daun, B., Drascher, E., Deale, M., Eskey, M., "Learning To Improve Constraint-Based Scheduling," *Artificial Intelligence*, vol. To Appear, 1992.

[23] Fox, M. and Smith, S., "A Knowledge Based System for Factory Scheduling," *Expert System*, vol. 1, no. 1, 1984.

[24] Ow, P., Smith S., Thiriez, A., "Reactive Plan Revision," in *Proceedings AAAI-88*, 1988.

[25] Johnson, D.S., Aragon, C.R., McGeoch, L.A., Schevon, C, "Optimization By Simulated Annealing:An Experimental Evaluation, Part I (Graph Partioning)," *Operations Research*, 1990.

[26] Johnson, D.S., Aragon, C.R., McGeoch, L.A., Schevon, C, "Optimization By Simulated Annealing:An Experimental Evaluation, Part II (Graph Coloring and Number Partioning)," *Operations Research*, 1990.

[27] Sussman, G.J., *A Computational Model of Skill Acquisition* . PhD thesis, AI Laboratory, MIT, 1973.

[28] Simmons, R.G., "Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems," tech. rep., MIT Artificial Intelligence Laboratory, 1988.

[29] Kambhampati, S., "A Theory of Plan Modification," in *Proceedings of AAAI-90*, 1990.

[30] Hammond, K. J., "CHEF: A Model of Case-Based Planning," in *Proceedings of AAAI-86*, 1986.

[31] Miller, D., Firby, R. J., Dean, T., "Deadlines, Travel Time, and Robot Problem Solving," in *Proceedings of AAAI-88*, (St. Paul, Minnesota), 1988.

[32] Currie, K., and Tate, A., "O-Plan: The Open Planning Architecture," *Artificial Intelligence*, vol. 52, no. 1, 1991.

[33] Drummond, M. and Bresina, J., "Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction ," in *Proceedings of AAAI-90*, 1990.

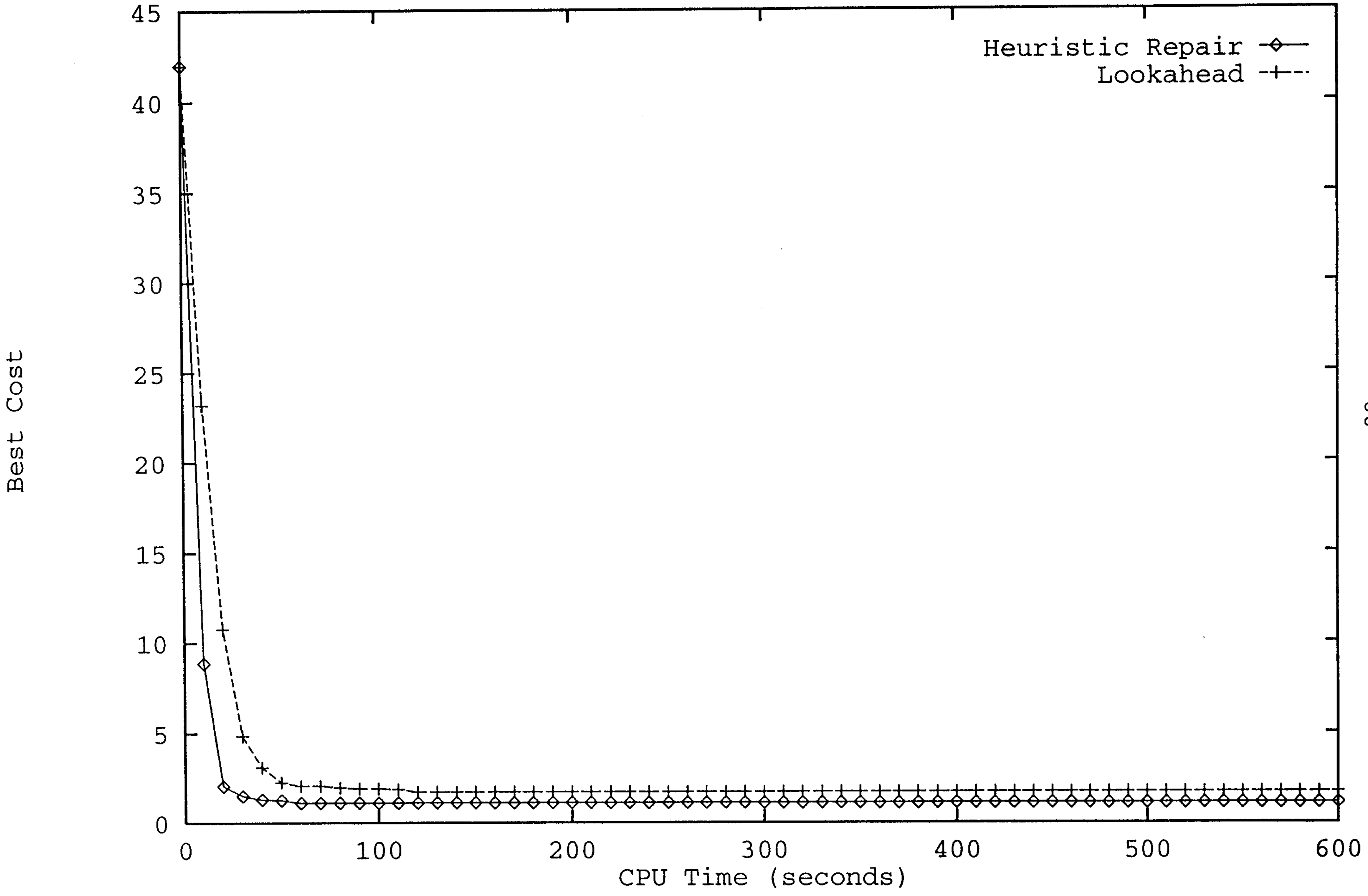

Figure 1: Results of scheduling randomly generated problems of different sizes with moderately constrained due dates. (a) 20-task problem.

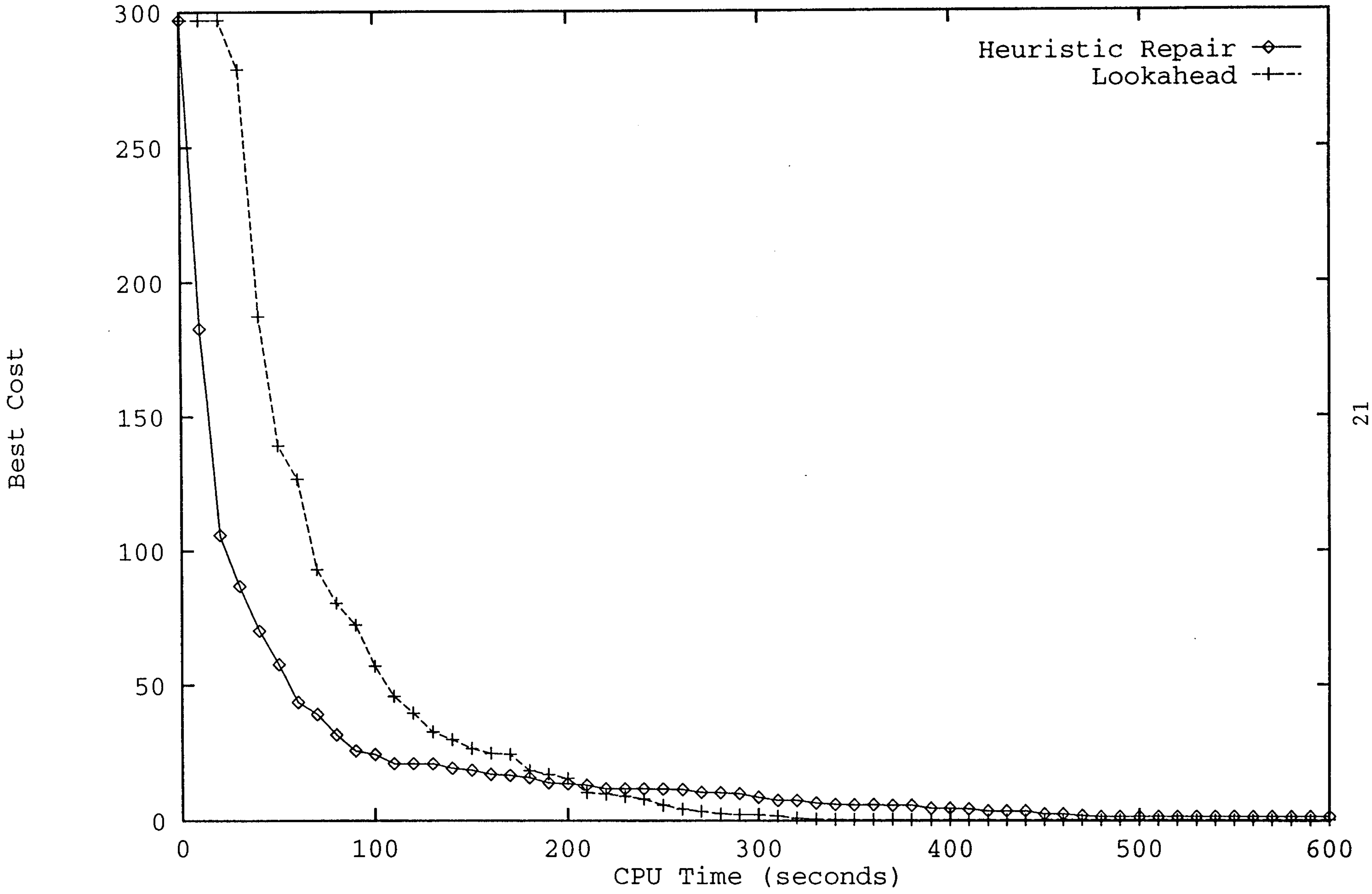

Figure 1: Results of scheduling randomly generated problems of different sizes with moderately constrained due dates. (b) 50-task problem.

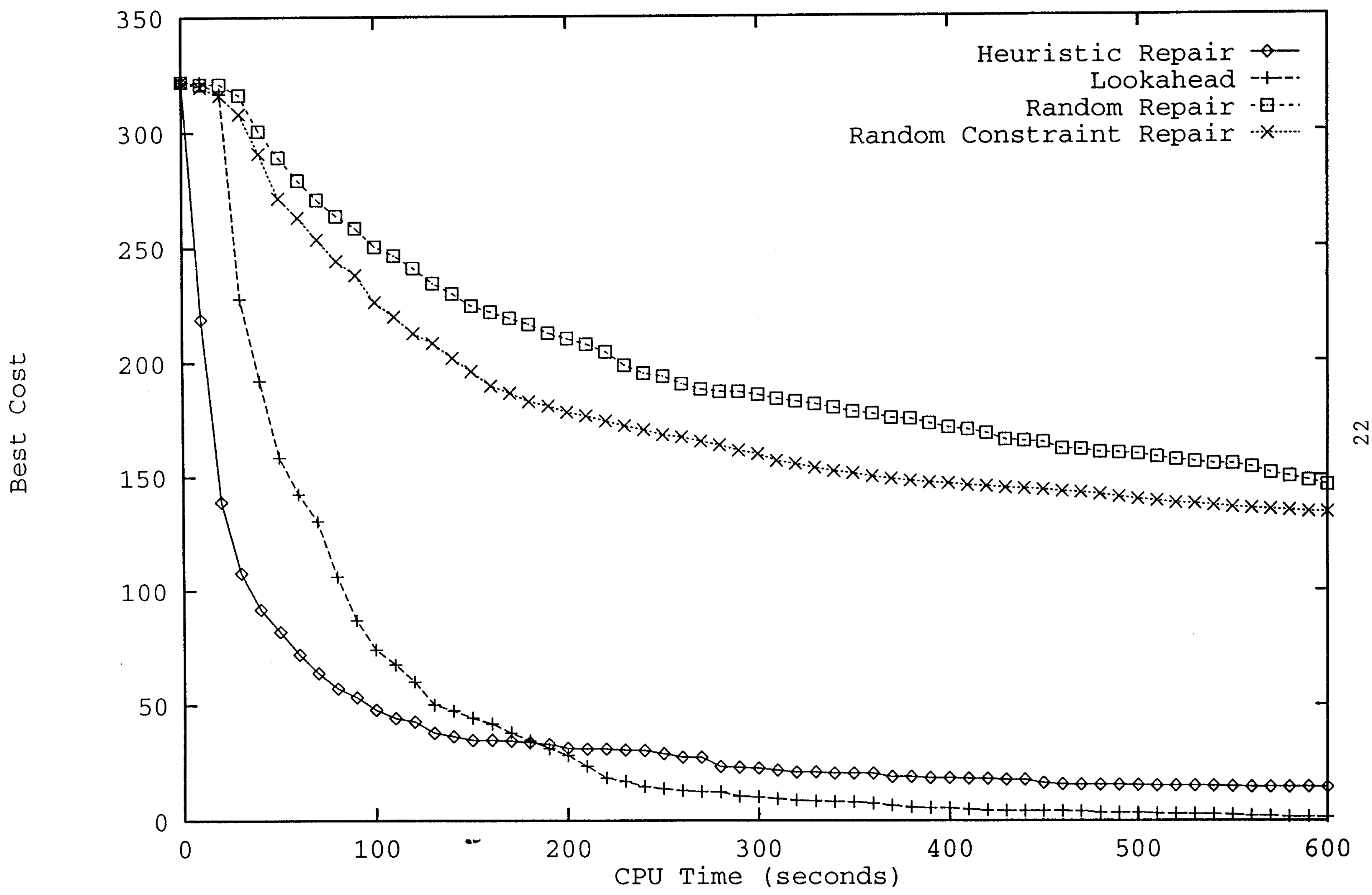

Figure 1: Results of scheduling randomly generated problems of different sizes with moderately constrained due dates. (c) 100-task problem.

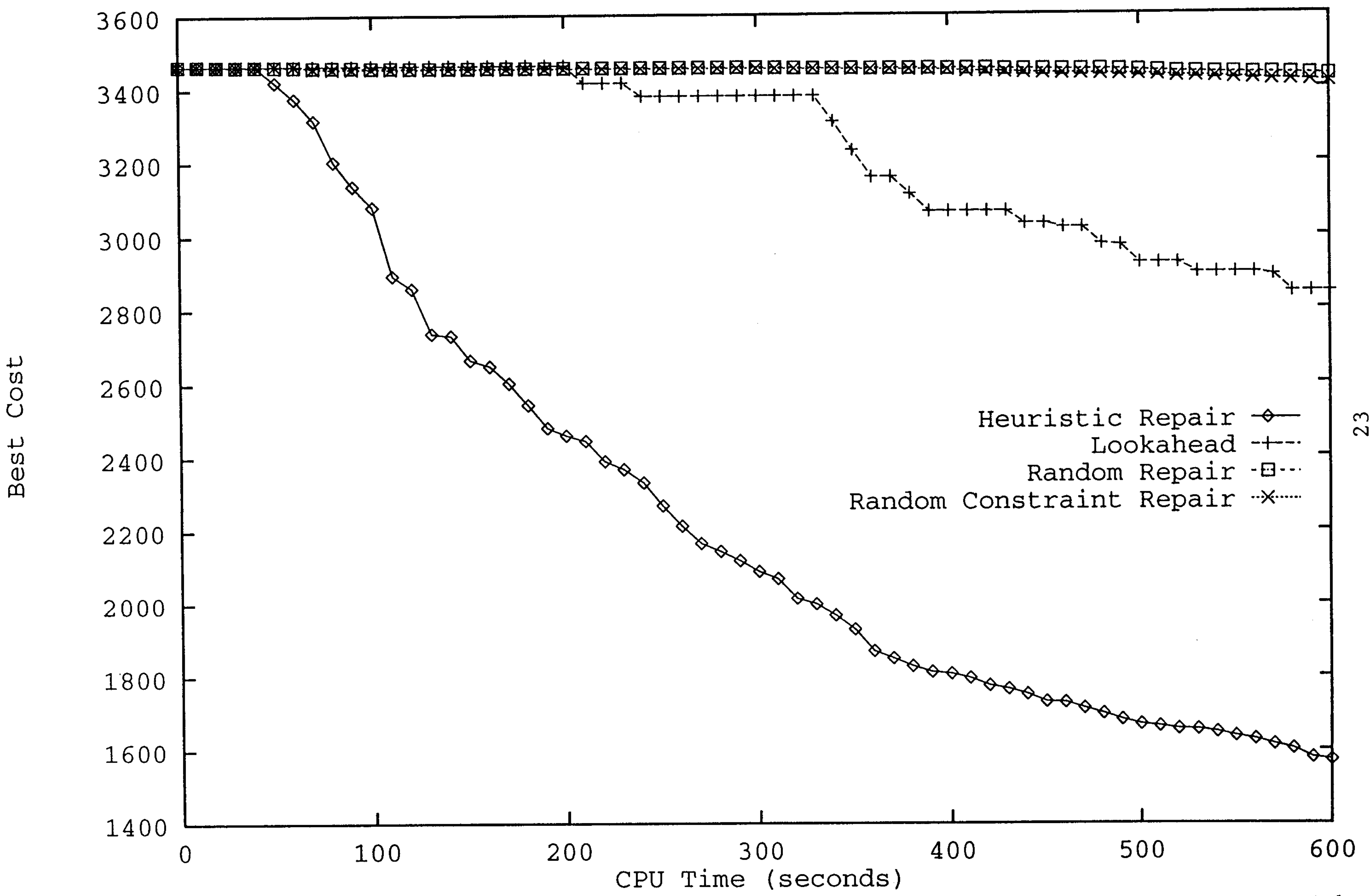

Figure 1: Results of scheduling randomly generated problems of different sizes with moderately constrained due dates. (d) 500-task problem.

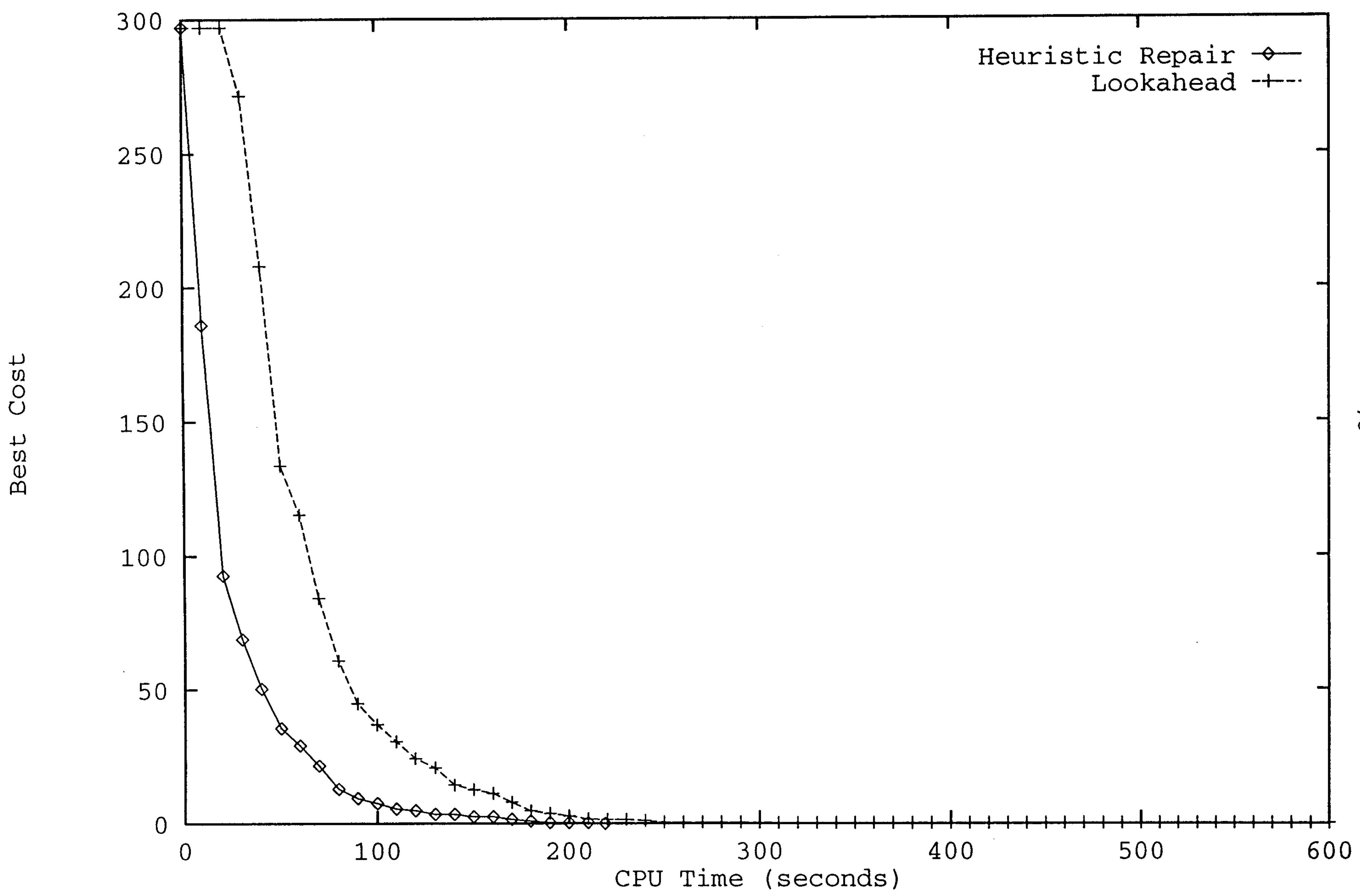

Figure 2: Results of scheduling the 50-task problem with different due dates.
(a) 50% (overconstrained).

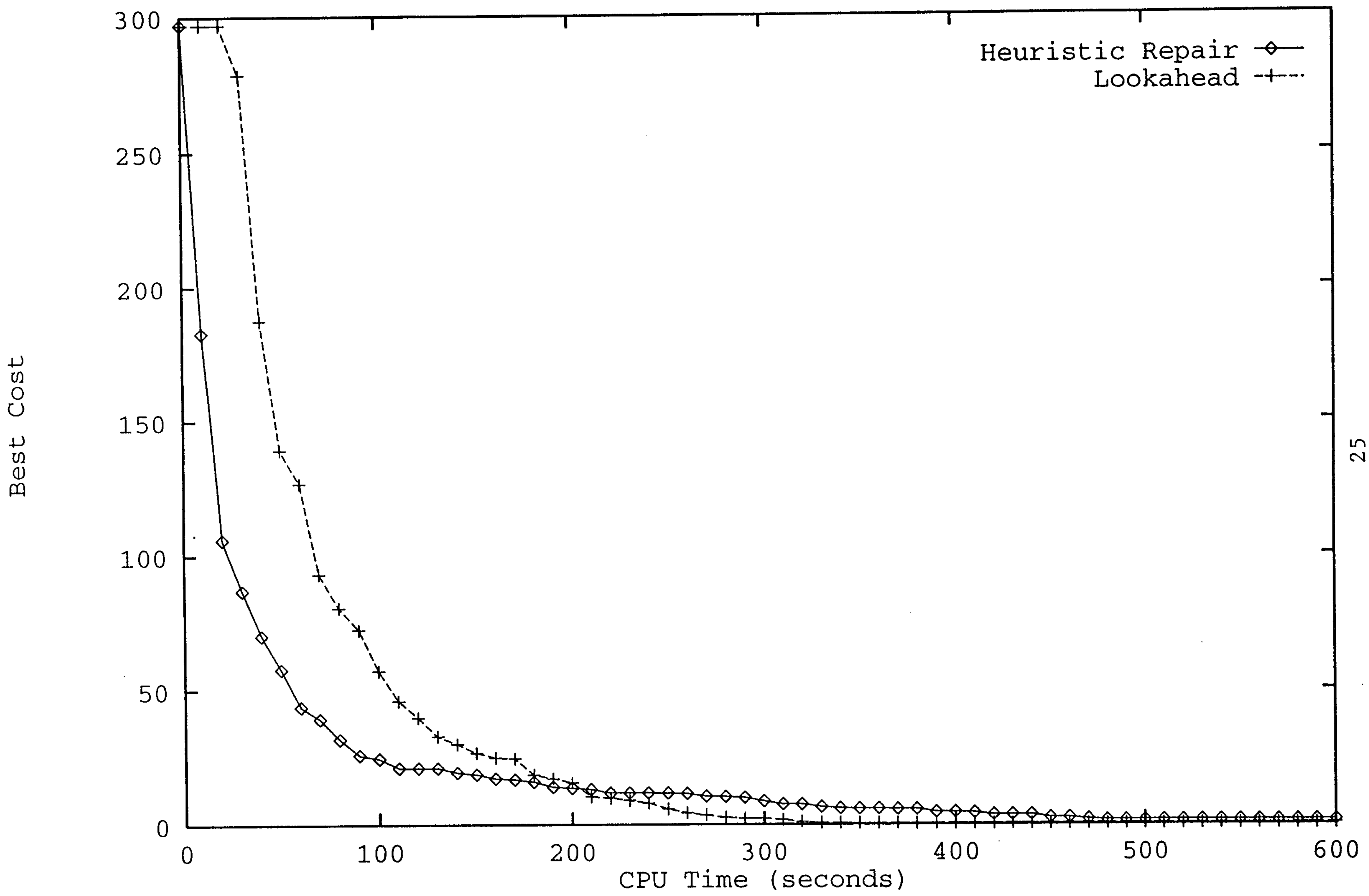

Figure 2: Results of scheduling the 50-task problem with different due dates.
(b) 100% (moderately constrained).

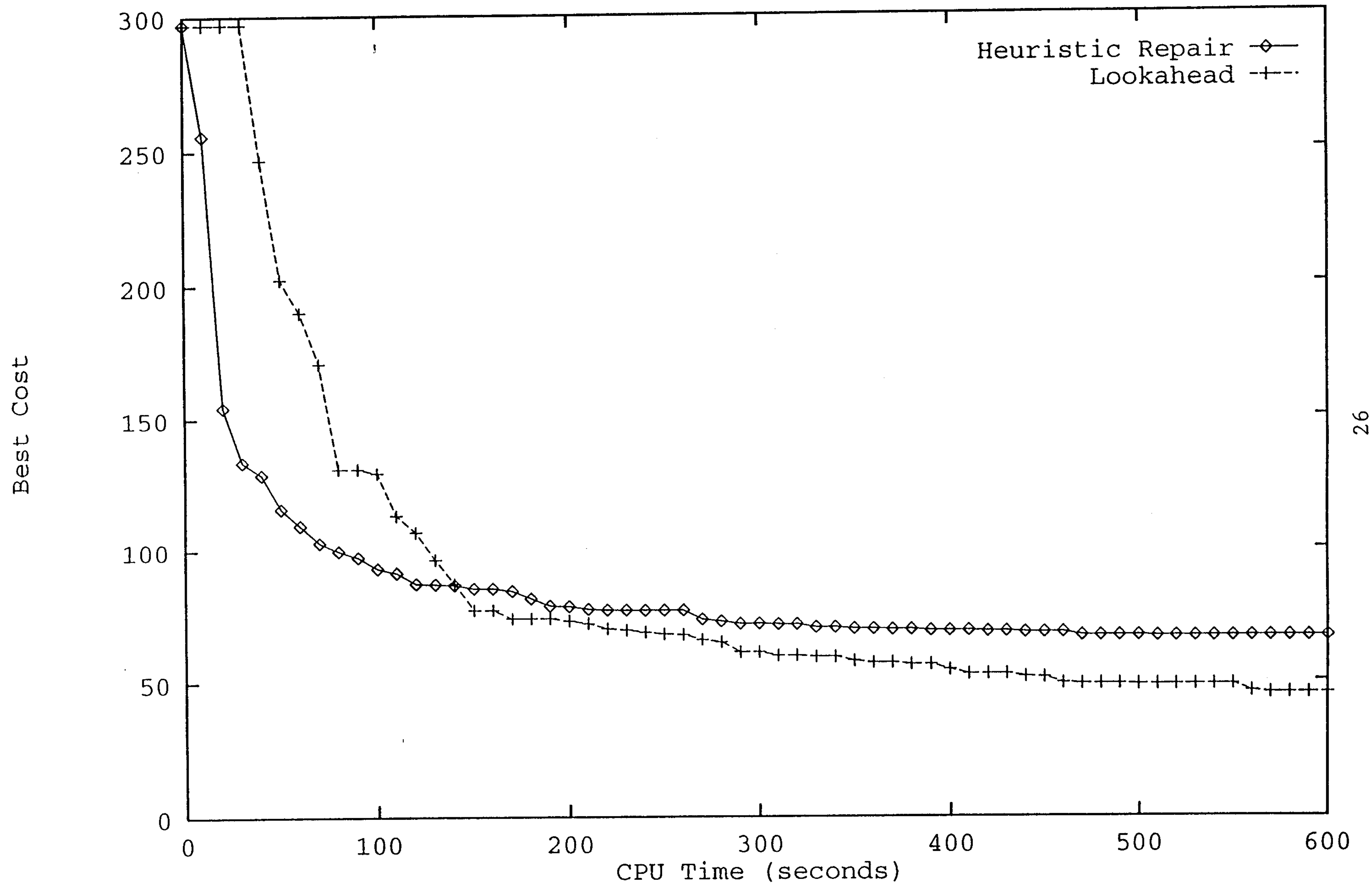

Figure 2: Results of scheduling the 50-task problem with different due dates.
(c) 150% (underconstrained).

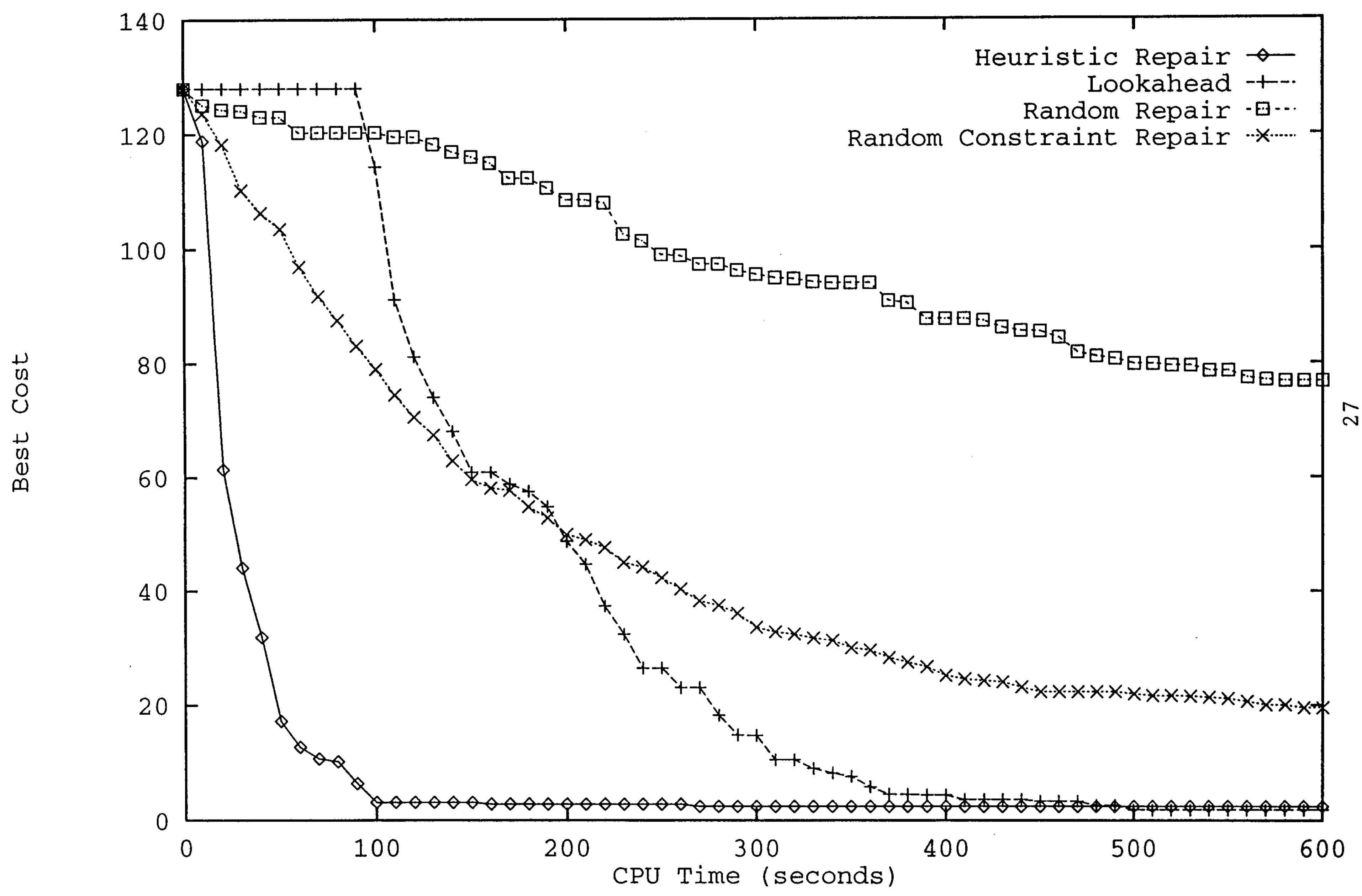

Figure 3: Representative rescheduling runs from the Space Shuttle Ground Processing domain. (a) STS-43.

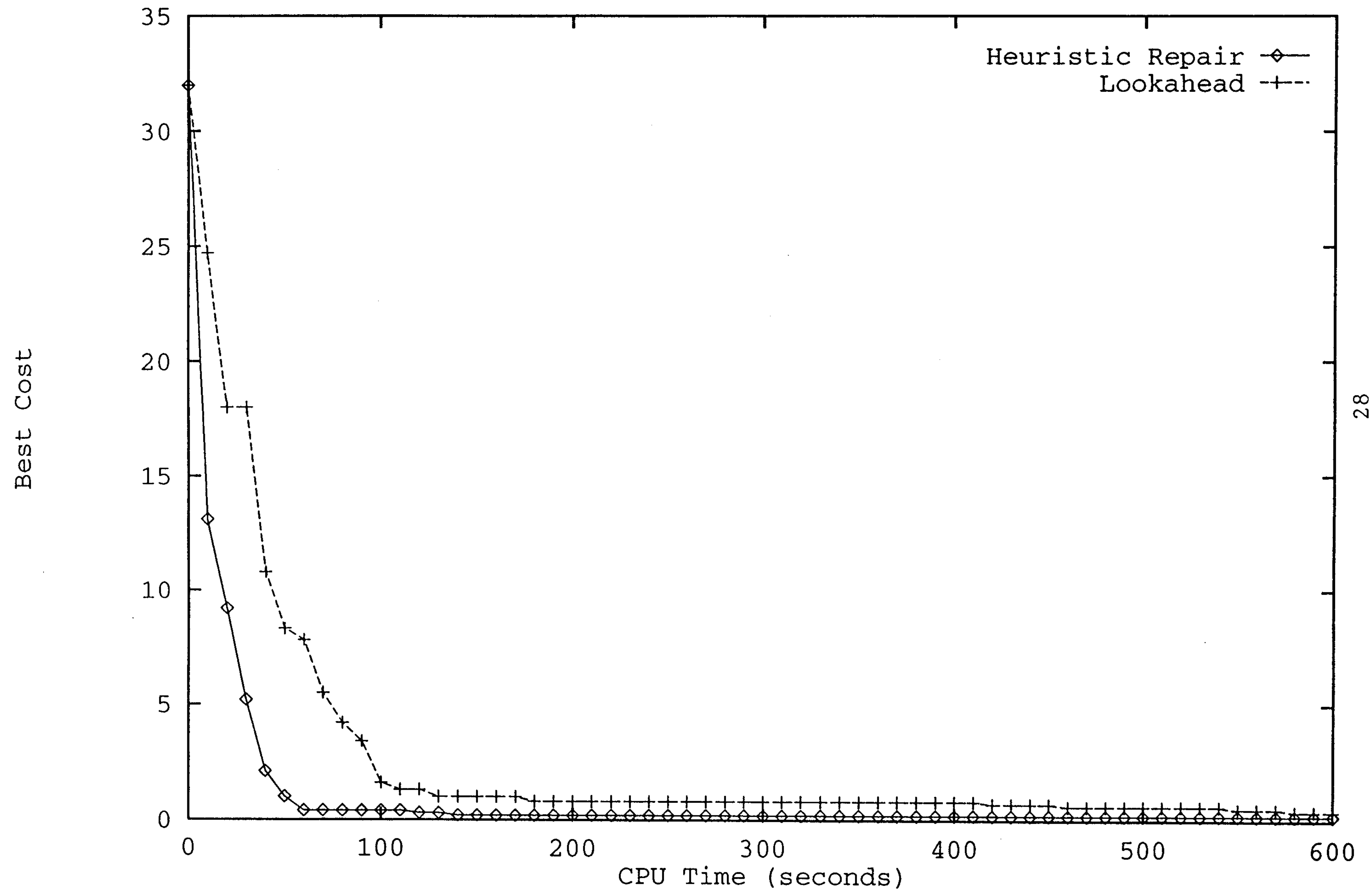

Figure 3: Representative rescheduling runs from the Space Shuttle Ground Processing domain. (b) STS-50.

Problem Name	Due Date	Initial Cost	Average Time to Solution (sec)		Percent Timeouts		Average Cost at Timeout	
			Normal	Lookahead	Normal	Lookahead	Normal	Lookahead
20 Tasks	30	42	-	-	100	100	5	5
20 Tasks	50	42	15	43	55	60	2	3
20 Tasks	70	42	14	29	5	20	1	2
50 Tasks	50	297	-	-	100	100	55	37
50 Tasks	100	297	328	229	0	35	-	1
50 Tasks	150	297	98	195	0	26	-	1
100 Tasks	30	322	-	1072	100	85	38	16
100 Tasks	50	322	663	470	15	0	7	-
100 Tasks	100	322	65	130	0	0	-	-
500 Tasks	150	3465	-	-	100	100	1007	2053
500 Tasks	200	3465	-	-	100	100	902	2103
500 Tasks	250	3465	-	-	100	100	892	1975
STS-43_1		51	13	93	0	0	-	-
STS-43_2		128	467	816	10	10	4	4
STS-43_3		95	25	157	0	0	-	-
STS-43_4		158	573	423	0	20	-	5
STS-43_5		63	26	148	0	0	-	-
STS-43_6		104	16	107	0	0	-	-
STS-43_7		100	62	188	0	0	-	-
STS-43_8		159	32	271	0	0	-	-
STS-43_9		109	254	439	0	0	-	-
STS-43_{10}		135	306	508	0	0	-	-
STS-50_1		90	532	721	20	0	7	-
STS-50_2		32	111	179	10	30	1	1
STS-50_3		58	90	328	0	0	-	-
STS-50_4		62	-	-	100	100	4	5
STS-50_5		25	24	67	0	0	-	-
STS-50_6		11	22	17	0	0	-	-
STS-50_7		64	37	20	0	0	-	-
STS-50_8		19	14	27	0	0	-	-
STS-50_9		110	17	62	0	0	-	-
STS-50_{10}		36	-	-	100	100	1	1

Figure 4: Experimental Results: CPU Time for problems that converge, percentage of problems that do not converge, and average cost at timeout for non-converging problems.