Iterative Repair Algorithm

The following is the algorithm for the schedule repairer written in a C-like pseudo-code.

```
Resolve-Conflicts ()
{
   iterations = 1
   conflicts = GetConflicts()
   Loop while (length(conflicts) > 0 && iterations <= max-iterations) {
      conflict = ChooseConflict(conflicts)
      method = ChooseMethod(conflict)
      case (method) {
      'move'
         culprit = ChooseCulpritToMove(conflict)
         duration = ChooseDuration(conflict, culprit)
         start-time = ChooseStartTime(conflict,culprit,duration)
         success = MoveCulprit(conflict,culprit,start-time)
      'add'
         activity = ChooseActivityToAdd(conflict)
         duration = ChooseDuration(conflict, activity)
         start-time = ChooseStartTime(conflict,activity,duration)
         success = AddActivity(conflict,activity,start-time)
      'delete'
         culprit = ChooseCulpritToDelete(conflict)
         success = DeleteCulprit(conflict,culprit)
      }
      progress = GetProgress()
      if not(success || progress) then UndoLastAction()
      conflicts = GetConflicts()
      iterations = iterations + 1
   }
}
```

of the system is to find the best place to schedule the activities so as to maximize the utility of the schedule. In the basic scheduler, all choices are made randomly from the list of options unless otherwise specified.

The algorithm is a simple iterative loop over the conflicts in the schedule. First, a conflict is selected from the list of current conflicts. An attempt is made to resolve the chosen conflict. Next, a method for resolving the conflict is chosen. The repair action will depend on which method has been selected. If "move" is chosen, then a culprit must be picked from the list of culprits in the conflict. A duration and start time are chosen for the culprit, and the culprit is moved to the new location. If "add" is the chosen method, then the repairer must decide which activity type to instantiate. Again, a duration and start time must be chosen for the new activity, and the activity is inserted at the chosen time. If the repairer chooses to "delete" an activity, then it simply must choose an activity to delete, and delete it. After the chosen action is performed, the schedule repairer checks to see if progress was made. We define progress as either decreasing the number of conflicts, decreasing the number of culprits, or decreasing the duration of the conflicts.

If the action did not succeed in resolving the conflict, or progress was not made, then the action is "undone." Otherwise, the new set of conflicts are found, and the loop counter is incremented. This process continues until all conflicts are resolved, or the loop counter exceeds a user-defined maximum bound. For every choice point in the algorithm, where a selection must be made from a list of possibilities, the schedule repairer is allowed to backtrack to that point. What this means is, that if a particular choice fails, the schedule repairer may choose another from the list before giving up. If all choices fail, then a previous decision must have been incorrect, and the repairer can backtrack to the preceding choice point. All choice points, including the decision on whether or not to backtrack, are heuristic decisions and may customized to a particular domain.

Schedule Optimizer—The schedule optimizer is composed of additional knowledge supplied by the user and utilized by the other components of the scheduler. There are three ways to optimize a schedule: using preference heuristics at search choice points in the schedule repairer, specifying a set of "soft conflicts" for the repairer, and using an evaluation function to score results from multiple runs of the schedule generator and repairer.

A preference heuristic, or "soft choice," can be made at any decision in the repair search. For example, when deciding where to move a conflict causing activity, the user might prefer to move that activity to a position closest to its current position. This will help the scheduler avoid unnecessary disruption to the existing schedule. The existing schedule, after all, may have been produced by the user in an attempt to optimize the schedule.

Preferences can also be expressed using what we referred to as "soft conflicts." A soft conflict is a way of specifying a preferred value for a particular resource, possibly at a particular time. For example, having any scanned data that has not been stored on the tape at the end of the mission, is considered a soft conflict. This is not a hard conflict, because the data is not exceeding the buffer size. However, the scientist would prefer that all of the data be written to the tape at the mission's end, rather than leaving it in the on-board memory. After the schedule repairer handles all of the hard conflicts, it continues by iteratively addressing all of the soft conflicts.

The third approach to optimization involves scoring several resulting schedules and choosing the one with the highest score. The evaluation function is domain dependent and would have to be written separately for each application. Some basic scoring, however, will be similar across applications. For example, most science spacecraft are mainly concerned with collecting the largest number of images as possible. A simple evaluation would give a higher score to schedules with greater amounts of collected data. Once we have the evaluation function, we need to be able to produce several different schedules from the same goals and initial state.

This can be done by either changing the heuristics or by running the scheduler with a different random seed. Some heuristics may work better than others, and it is often difficult to tell which is the best for a particular application. Therefore, it may be necessary to resort to empirical tests. After running the scheduler on different heuristics, we can simply choose the set of heuristics which generates the schedule with the highest score. After choosing the heuristics, the scheduler can be run many times with different random seeds. At choice points where there is no heuristic for choosing from the list of possibilities, the scheduler makes a random decision. With different random seeds, these decisions will be different, and the resulting schedule will be different. Using the evaluation function, we can assign a score to each, and choose the schedule with the highest score. This procedure will not necessarily uncover the optimum schedule, but it will help find a more optimal schedule.

Heuristics—The general search and decision making described above would be futile without expert support and guidance. Heuristics have been developed and incorporated into DCAPS to help guide the search to a valid and more optimal schedule. This guidance knowledge comes from both domain experts and scheduling experts. There are three basic classes of heuristics used in DCAPS: selection, pruning, and backtracking heuristics.

Selection heuristics involve deterministically sorting or selecting from a list of possibilities at a choice point in the search. The selection is usually based on some property of the objects being considered. For example, when choosing a culprit to move in order to resolve a power conflict, one heuristic might choose the culprit that uses the most amount of power. Using this heuristic might resolve the conflict faster.

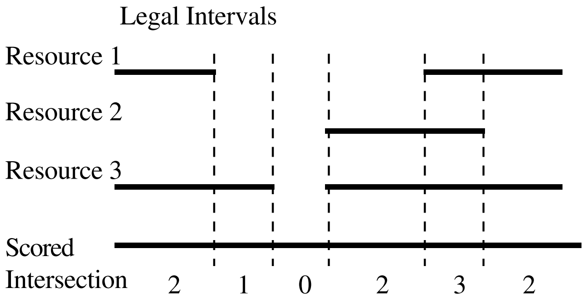


Figure 4: Min-conflicts with scored interval intersection

Another successful heuristic used in DCAPS was one that sorted the possible locations for activity placement by the number of conflicts the activity would cause when placed in that location. This basic approach has been referred to as the "min-conflicts" heuristic [5]. The min-conflicts algorithm we use is interesting, and it is worthwhile to go into detail.

For each resource used by an activity, we query the database for the legal times where the activity can be placed without violating the resource constraint. Then, each legal interval is assigned an initial score of one. Next, we intersect two sets of intervals that resulted from two of the resources, using a special "scored" interval intersection (see Figure 4). The scored intersection of intervals A and B results in four possibilities: an interval with a score of A for positions where A exists and B does not, an interval with a score of B where B exists and A does not, an interval with a score of A plus the score of B where the two intervals intersect, or no interval where neither A nor B exist. The result of this intersection is then intersected with the third set of intervals.

This process continues until each set of intervals for each resource has been intersected. The result is a set of scored intervals, where the score represents the number of resources that will not be violated if the activity is placed in that position. Using these intervals, we can choose a position with the highest score, in other words, the position with the fewest conflicts.

Another class of heuristics used in DCAPS are the pruning heuristics. These heuristics remove some of the possibilities for a given selection in attempt to make the choice easier and faster. For example, after finding the scored intervals for an activity, we may not want to try all possible positions. One possibility is to only try positions with the highest score or least number of conflicts. This process may speed up scheduling because the scheduler will only try a few positions before realizing this attempt is futile and giving up to try something different. Too much pruning, however, may remove possibilities that could be useful. In the above example, some of the pruned intervals may have included positions that, if the activity was placed there, would have improved the schedule. A more conservative approach might be to prune only those intervals that would cause more conflicts than are currently in the schedule. These intervals cannot possibly be positions that could improve the schedule.

Finally, backtracking heuristics are used to help determine when to continue working on the same problem and when to move on to a different problem. At each choice point, we have a list of possibilities. If we try one possibility, and it fails, we can continue and try the next possibility, or move on to a different choice point. Heuristics can be used to help make two types of decisions about backtracking: deciding on "action failures" and deciding on "selection failures." First, the notion of an "action failure" is not clear and requires an approximate definition. Success is not simply resolving the chosen conflict. When, resolving a conflict, and action attempt may fix the chosen conflict, but cause several other conflicts.

Therefore, success can be thought of as improving the schedule. But how much? And what defines an improvement? Our current definition of progress includes observing the change in the number of conflicts, the change in the number of culprits, and the change in the duration of the conflicts. Checking the progress of an action can be used as a heuristic for determining whether to accept the action, or try a different one. The second opportunity for heuristics comes when deciding if there is a "selection failure." While trying and failing on a list of possibilities for a choice point, at some point we must decide that the previous choice was a failure. Heuristics can help with this decision also.

6. System Integration

DCAPS will be integrated into the End-to-End Mission Operations System (EEMOS) that is currently being developed for the DATA-CHASER project as a prototype for the Pluto Express EEMOS [6]. Currently the DATA-CHASER EEMOS consists of seven parts: Command and Control, Fault/Event Detection Interaction Reaction (F/EDIR), DATA/IO (Data handling), the Ground Database, the Graphical User Interface, the software testbed, and finally the planning and scheduling system (DCAPS).

The command and control system that we are using, System Command Language (SCL, also known as Spacecraft Command Language), integrates procedural programming with a real-time, forward-chaining, rule-based system. DCAPS interfaces with SCL through DATA/IO by sending script scheduling commands that are to be scheduled either on the flight or ground system. This interface is implemented by mapping PI2 activities to SCL scripts that were written prior to flight and can be scheduled or event-triggered by activating rules. These scheduling and rule activation commands are then sent to DATA/IO which forwards that list to the SCL Compiler. Once compiled, the list is sent to the payload through the next available uplink.

DCAPS is also interfaced with the ground EEMOS database, O2. O2 is an object-oriented database that will be used to store all mission data and telemetry that is downlinked by the payload. It will also store a command history. Through DATA/IO, DCAPS will request current payload status data in the form of sensor values in the telemetry history. It will also request lists of all commands uplinked during a given time interval. These are used by DCAPS to infer command completion status as well as to get the current state of the payload so that a new schedule can be created.

During mission operations, approximately every four hours or so, DCAPS will be asked by an operator to generate script scheduling commands and rule activations for the next six hours according to its schedule. Once this list is finished, it is reviewed by the Mission Operations staff on duty. If judged to be correct, scheduling and rule activation commands will be sent to DATA/IO during the next available uplink window.

If during that six hour period there is a major change in the NASA activities, DCAPS will ask if the users want to update the schedule script on-board. Due to the fact that SCL currently has no scheduled script instance identification, this will involve descheduling all remaining scripts in the queue and then rescheduling them. This is acceptable if the user did not schedule any scripts independently of DCAPS. If he/she did, and DCAPS reschedules its list, the user's scheduled commands will be lost. If the user accepts it, DCAPS will generate a updated list, ask the user to verify it, and then deschedule rest of the old list and schedule the new list. Future versions of SCL will most likely support scheduling instances, therefore alleviating these problems.

7. Summary and Related Work

Iterative algorithms have been applied to a wide range of computer science problems such as traveling salesman [7] as well as Artificial Intelligence Planning [8,9,10,11]. Iterative repair algorithms have also been used for a number of scheduling systems. The GERRY/GPSS system [1,12] uses iterative repair with a global evaluation function and simulated annealing to schedule space shuttle ground processing activities. The Operations Mission Planner (OMP) [13] system used iterative repair in combination with a historical model of the scheduler actions (called chronologies) to avoid cycling and getting caught in local minima. Work by Johnston and Minton [5] shows how the min-conflicts heuristic can be used not only for scheduling but for a wide range of constraint satisfaction problems. The OPIS system [14] can also be performing iterative repair. However, OPIS is more informed in the application of its repair methods in that it applies a set of analysis measures to classify the bottleneck before selecting a repair method.

In summary, DCAPS represents a significant advance from several perspectives. First, from a mission operations perspective, DCAPS is important in that it significantly reduces the amount of effort and knowledge required to generate command sequences to achieve mission operations goals. Second, from the standpoint of Artificial Intelligence applications, DCAPS represents a significant application of planning and scheduling technology to the complex, real-world problem of spacecraft commanding. Third, from the standpoint of Artificial Intelligence Research, DCAPS mixed initiative approach to initial schedule generation, iterative repair, and schedule optimization represents a novel approach to solving complex planning and scheduling problems.

ACKNOWLEDGMENTS

This work was performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

REFERENCES

[1] M. Zweben, B. Daun, E. Davis, and M. Deale, “Scheduling and Rescheduling with Iterative Repair,” in *Intelligent Scheduling*, Morgan Kaufman, San Francisco, 1994.

[2] DATA-CHASER Documents, Annual Report.

[3] G. Rabideau, S. Chien, T. Mann, C. Eggemeyer, P. Stone, and J. Willis, “DCAPS User’s Manual,” JPL Technical Document D-13741, 1996.

[4] W. Eggemeyer, “Plan-IT-II Bible”, JPL Technical Document, 1995.

[5] M. Johnston and S. Minton, “Analyzing a Heuristic Strategy for Constraint Satisfaction and Scheduling,” in *Intelligent Scheduling*, Morgan Kaufman, San Francisco, 1994.

[6] S. Siewert and E. Hansen, “A Distributed Operations Automation Testbed to Evaluate System Support for Autonomy and Operator Interaction Protocols,” *4th International Symposium on Space Mission Operations and Ground Data Systems*, ESA, Forum der Technik, Munich, Germany, September, 1996.

[7] S. Lin and B. Kernighan, “An Effective Heuristic for the Traveling Salesman Problem,” *Operations Research Vol. 21*, 1973.

[8] S. Chien and G. DeJong, "Constructing Simplified Plans via Truth Criteria Approximation," *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, Chicago, IL, June 1994, pp. 19-24.

[9] K. Hammond, “Case-based Planning: Viewing Planning as a Memory Task,” Academic Press, San Diego, 1989.

[10] R. Simmons, “Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems,” Technical Report, MIT Artificial Intelligence Laboratory, 1988.

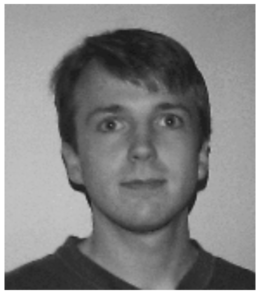
[11] G. Sussman, “A Computational Model of Skill Acquisition,” Technical Report, MIT Artificial Intelligence Laboratory, 1973.

[12] M. Deale, M. Yvanovich, D. Schnitzius, D. Kautz, M. Carpenter, M. Zweben, G. Davis, and B. Daun, “The Space Shuttle Ground Processing System,” in *Intelligent Scheduling*, Morgan Kaufman, San Francisco, 1994.

[13] E. Biefeld and L. Cooper, “Bottleneck Identification Using Process Chronologies,” *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991.

[14] S. Smith, “OPIS: A Methodology and Architecture for Reactive Scheduling,” in *Intelligent Scheduling*, Morgan Kaufman, San Francisco, 1994.

Gregg Rabideau is a Member of the Technical Staff in the Artificial Intelligence Group at the Jet Propulsion Laboratory, California Institute of Technology. His main focus is in research and development of planning and scheduling systems for automated spacecraft commanding. Projects include planning and scheduling for the first deep-space mission of NASA's New Millennium Program, and for design trades analysis for the Pluto Express project. Gregg holds both a B.S. and M.S. degree in Computer Science from the University of Illinois where he specialized in Artificial Intelligence.



Steve Chien is Technical Group Supervisor of the Artificial Intelligence Group of the Jet Propulsion Laboratory, California Institute of Technology where he leads efforts in research and development of automated planning and scheduling systems. He is also an adjunct assistant professor in the Department of Computer Science at the University of Southern California. He holds B.S., M.S., and Ph.D. degrees in Computer Science from the University of Illinois. His research interests are in the areas of planning and scheduling, operations research, and machine learning.

Tobias Mann was born in Spokane, Washington and is currently an undergraduate at the University of Washington in both the Computer Science and Philosophy departments. He has a wife and a two-year old son. His interests include planning and scheduling, machine learning, bicycling, and really good coffee.

William "Curt" Eggemeyer graduated from Washington University in St. Louis with a BA majoring in geology. In 1978, he became a JPL employee and began working on the Voyager project as a spacecraft sequence engineer. He demonstrated the applicability of utilizing artificial intelligence (AI) techniques to the sequence process with the generation of Voyager Uranus encounter sequences with a program called DEVISER, developed by Steven Vere, in 1983-1984. He codeveloped a prototype, Plan-IT, further advancing sequencing software tool concepts. From 1991-1992, he reworked Plan-IT into a more capable an robust sequencing tool, called Plan-IT-2, that is presently being used by DATA-CHASER, Galileo, and Mars Pathfinder projects.

Jason Willis is a currently pursuing a Master's Degree in Aerospace Engineering specializing in spacecraft systems from the University of Colorado Boulder, where he also received his B.S. in Aerospace engineering. He has worked at the Colorado Space Grant College for the past three years first as Electrical Integration Team Lead on the ESCAPE II shuttle payload the was launched on STS-66. He is currently the hardware systems engineer for the DATA-CHASER project.

Sam Siewert is a graduate research assistant with Colorado Space Grant College. He is working on a Ph.D. in Computer Science at the University of Colorado Boulder where he received his M.S. in Computer Science in 1993. He received his B.S. in Aerospace Engineering from the University of Notre Dame in 1989, worked three years for McDonnell-Douglas Astronautics Corporation in Guidance, Navigation and Control developing simulation, space environment models, and guidance systems software for the Space Station and the

Aeroassist Flight Experiment. During that time, he also worked for McDonnell-Douglas at Johnson Space Center in the Shuttle Mission Control Center, developing shuttle ascent and entry monitoring and cockpit avionics visualization software, before returning to graduate school.

***Peter Stone** is a Ph.D. candidate in Computer Science at Carnegie Mellon University (CMU). He completed his undergraduate education in Mathematics with a concentration in Computer Science at the University of Chicago in 1993. His interests are in the areas of multiagent systems, collaborative and adversarial machine learning, and planning, especially in multiagent, real-time environments.*