Using Squeaky Wheel Optimization to Derive Problem Specific Control Information for a One Shot Scheduler for a Planetary Rover

Wayne Chi, Steve Chien, Jagriti Agrawal
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
{firstname.lastname}@jpl.nasa.gov

Abstract

We describe the application of using Monte Carlo simulation to optimize a schedule for execution and rescheduling robustness and activity score in the face of execution uncertainties. We apply these techniques to the problem of optimizing a schedule for a planetary rover with very limited onboard computation. We search in the schedule activity priority space - where the onboard scheduler is (a) a one shot non-backtracking scheduler in which (b) the activity priority determines the order in which activities are considered for placement in the schedule and (c) once an activity is placed it is never moved or deleted. We show that simulation driven search outperforms a number of alternative proposed heuristic static priority assignment schemes. Our approach can be viewed using simulation feedback to determine problem specific heuristics much like squeaky wheel optimization.

Introduction

Embedded schedulers must often perform within very limited computational resources. We describe an approach to automatically deriving problem specific control knowledge for a one-shot (non-backtracking) scheduler intended for a planetary rover with very limited computing. In this application, the onboard scheduler is intended to make the rover more robust to run-time variations (e.g., execution durations) by rescheduling. Because the general structure of the schedule is known a priori on the ground before uplink, we use both analysis of the schedule dependencies and simulation feedback to derive problem specific control knowledge to improve the onboard scheduler performance.

The target onboard scheduler is a one-shot limited search scheduler. Because the scheduler does not backtrack across activity placements, the order in which it considers activities heavily influences generated schedule quality. In our approach, we search the space of activity priorities which determine the order in which the scheduler considers activity placement. At each step in the priority search, a Monte Carlo simulation is conducted to assess the likelihood of an activity being executed. Using an approach analogous to squeaky wheel optimization, these runs are automatically analyzed and used to feed back into adjustments to the activity priorities (and hence the order in which they are considered for inclusion in the schedule for both initial schedule generation and rescheduling). This search in the activity priority space continues until all requested activities are included or a resource bound is exceeded. We call this method *Priority Search* and we present empirical results that show that *Priority Search* outperforms several static priority assignment methods (those that do not use Monte Carlo feedback) *including manual expert derived priority setting*.

We study this problem in the context of setting activity priorities as part of the ground operations process for a one-shot, non-backtracking scheduler (Rabideau and Benowitz 2017) designed to run onboard NASA's next planetary rover, the Mars 2020 (M2020) rover (Jet Propulsion Laboratory 2017a). For our problem, the onboard scheduler is treated as a predetermined "black box".

The remainder of the paper is organized as follows. First we describe our formulation of the scheduling problem, metrics for schedule goodness, and the onboard scheduling algorithm. Second, we describe several static approaches to priority assignment as well as our *priority search* approach that leverages Monte Carlo simulation feedback. Third, we describe empirical results demonstrating the efficacy of *priority search* over static methods, evaluating on *sol types*, the best available anticipated operations plans for the M2020 planetary rover mission. Finally, we describe related and future work and conclusions.

Problem Definition

For our defined scheduling problem (Rabideau and Benowitz 2017), the scheduler is given

- a list of activities $A_i\langle p, R, e, dv, \Gamma, T, D\rangle \dots A_n\langle p, R, e, dv, \Gamma, T, D\rangle$
- where p is the scheduling priority of the activity, and
- R is the set of unit resources $R_1 \dots R_m$ that the activity will use (up to project limitations - 128 for M2020), and
- e and dv are the rate at which the consumable resources energy and data volume respectively are consumed by the activity, and
- Γ are non-depletable resources used such as sequence engines available or peak power, and

- T is a set of the activity's optional a) start time windows $T_{i\_start} \dots T_{i\_end}$ and b) preferred schedule time $T_{i\_preferred}$, and
- D is a set of the activity's dependency constraints from $A_j \rightarrow A_k$ [1]

All activities are *Mandatory Activities*. These are activities, $m_1 \dots m_j \subseteq A$, that must be scheduled as long as the given set of inputs are *valid*. In order for a set of inputs to be considered *valid*, there must exist a valid (e.g. constraint satisfying) schedule - in the context of the scheduler - that includes all of the mandatory activities. Note that the M2020 Onboard Scheduler is an incomplete algorithm. As a result, there could be a set of inputs where valid schedule exists and a complete scheduler would place all mandatory activities, but the Onboard scheduler would not. Since not all input sets will be *valid*, it is important for us to modify the input sets (e.g. changing priorities) to allow all mandatory activities to be scheduled.

In addition, activities can be grouped into *Switch Groups*. A *Switch Group* is a set of activities where exactly one of the activities in the set must be scheduled. The activities within a switch group are called *switch cases* and vary only by how many resources (time, energy, and data volume) they consume. Switch groups allow us to schedule a more resource-consuming activity if it will fit in the schedule. For example, one of the M2020 instruments takes images to fill mosaics which can vary in size; for instance we might consider $1x5$, $3x5$, or $5x5$ mosaics. Taking larger mosaics might be preferable, but taking a larger mosaic takes more time, takes more energy, and produces more data volume. These alternatives would be modeled by a switch group that might be as follows:

$$SwitchGroup = \begin{cases} Mosaic_{1x5} & \text{Duration=100 sec} \\ Mosaic_{3x5} & \text{Duration=200 sec} \\ Mosaic_{5x5} & \text{Duration=400 sec} \end{cases} \quad (1)$$

In the above example, the scheduling priority order would be $Mosaic_{1x5}$ the lowest of the three, then $Mosaic_{3x5}$, and $Mosaic_{5x5}$ the highest. The desire is for the scheduler to schedule the activity $Mosaic_{5x5}$ but if it does not fit then try scheduling $Mosaic_{3x5}$, and eventually try $Mosaic_{1x5}$ if the other two fail to schedule. The challenge for the scheduler is that getting a preferred switch case is not deemed worth forcing out another mandatory activity from the schedule. Because the normal approach to handling such interactions is to search, this introduces complications into the scheduling algorithms but these are the subject of a different paper.

The charter of the scheduler is to produce a grounded time schedule that satisfies all of the above constraints.

We also make the following assumptions:

1. There exists a set of activity scheduling priorities that would allow all mandatory activities to be scheduled by the scheduler [2].
2. The prior schedule is executed while the scheduler is running (Chi et al. 2018).
3. Activities do not fail.
4. No preemption (activities are only preempted as a major failure case for M2020).
5. The onboard scheduler is a "black box" - the onboard scheduler algorithm (Algorithm 1) is fixed.

The goal of the scheduler is to schedule all mandatory activities and better switch cases [3] while respecting individual and plan-wide constraints.

The goal of the priority setting algorithm is to derive a set of priorities that will best allow the scheduler to achieve that goal. Not only that, but we must derive that set of priorities in the shortest amount of time possible in order to satisfy daily mission time constraints.

Scheduler Design

Algorithm 1 Onboard Scheduler

```
Input:
    A⟨p, R, e, dv, Γ, T, D⟩: List of activities with their individual constraints
    C: Constraints for the whole plan (e.g. available cumulative resources)
    S: Current state of the spacecraft (state of charge, data volume, activity status)
Output:
    U: Resulting schedule
 1: Sort(A)                                   ▷ Sorted by highest to lowest priority.
 2: for each $a \in A$ do
 3:     $P \leftarrow \emptyset$        ▷ Some activities may require automatically generated preheats
 4:     $M \leftarrow \emptyset$        ▷ Some activities may require automatically generated maintenances
 5:     $I \leftarrow [a.earliest\_start\_time, a.latest\_start\_time] \bigcap find\_valid\_intervals(a.unit\_resources) \bigcap find\_valid\_intervals(a.activity\_status) \bigcap find\_valid\_intervals(a.data\_volume)$
 6:     if $requires\_preheat(a)$ then
 7:         $P \leftarrow generate\_preheat\_activities(a)$
 8:         $M \leftarrow generate\_maintenance\_activities(a)$
 9:     end if
10:     $I \leftarrow I \bigcap find\_valid\_intervals(a.energy, P, M) \bigcap find\_valid\_intervals(a.peak\_power, P, M)$
11:     $awake \leftarrow generate\_awake\_activity(a, I)$
12:     if $I \neq \emptyset$ then
13:         $schedule\_activity(a, I)$
14:         $schedule\_activity(awake, I)$
15:         for each $p \in P$ do
16:             $schedule\_activity(p, I)$
17:         end for
18:         for each $m \in M$ do
19:             $schedule\_activity(m, I)$
20:         end for
21:     end if
22: end for
```

The Mars 2020 onboard scheduler (Algorithm 1) is a single shot, non-backtracking scheduler that schedules (consid-

[1] $A_j \rightarrow A_k$ means the scheduled end time of A_k must be before the scheduled start time of A_j.

[2] Since our algorithm includes an incomplete scheduler, our assumption of a valid set of inputs can only hold true for our particular scheduler

[3] See Evaluating a Schedule for more information

ers activities) priority first order and never removes or moves an activity after it is placed during a single scheduler run. It does not search except when considering valid intervals for a single activity placement and when scheduling sleep and preheats [4] (Rabideau and Benowitz 2017).

Due to the greedy, non-backtracking nature of the onboard scheduler, the order in which activities are scheduled can greatly impact the quality of the schedule.

Evaluating a Schedule

In order to evaluate the goodness of a particular priority assignment, we have developed a scoring method based on how many and what type of mandatory and switch group activities are able to be scheduled successfully by the scheduler. The score is such that the value of any single mandatory activity being scheduled is much greater than that of any combination of switch cases (at most one activity from each switch group can be scheduled). This ensures the following strict ordering:

$$V(m \in M) \gg \sum_{i=1}^{n_S} V(s \in S_i) \tag{2}$$

where $V(x)$ is the value of activity x being scheduled, M is the set of all mandatory activities, n_S is the number of switch groups, S_i is switch group i, and s is a switch case in switch group S_i.

Static Algorithms for Activity Priority Assignment

We have developed several static algorithms which set the priorities of activities based on various activity ordering criteria. These algorithms do not consider Monte Carlo simulations of plan execution where activities may end early or late while determining priorities, unlike our Priority Search approach. We will later compare these to our Priority Search approach to gain a better understanding of how well it performs. Activities which must begin at a particular time (e.g. data downlink) are always given the highest priority and thus are not affected by the static algorithms described.

The following four methods are used to initialize activity priorities:

- *Equal Priorities.* All activities have equal priorities.
- *Random Assignment.* Each activity is given a random priority.
- *Latest Start Time.* The activity priorities are ordered by the latest time they are allowed to start. The activity with the earliest such time has the highest priority.
- *Human Expert.* Each activity is assigned a priority based on the start time of the activity in a schedule constructed by a human expert. The activity with the earliest start time in this schedule has the highest priority.

The following two methods are applied to the priorities after they have been initialized in one of the four ways described above:

[4] Sleep and preheats are activities automatically generated and scheduled by the scheduler.

- *Dependencies.* $A \rightarrow B$ means that B must execute successfully before A can start. To generate a schedule that respects this,

$$A \rightarrow B \Rightarrow priority_A < priority_B \tag{3}$$

where higher priority means an activity is considered for scheduling earlier.

- *Tie Breaker.* If activities have the same priority assignment the activity with earliest latest allowed start time is of higher priority. If they also have the same latest allowed start time then the longer activity has the higher priority. If all of these attributes are equal then the higher priority activity is chosen lexicographically based on each activity's unique identifier.

Priority Search

In order to determine a set of priorities which will allow the scheduler to generate a schedule better than our static heuristics, we attempt to search the priority space in an approach similar to Squeaky Wheel Optimization (SWO) as described in Joslin and Clements 1999 (Joslin and Clements 1999). Squeaky Wheel Optimization usually involves a constructor, an analyzer, and a prioritizer. The constructor generates a schedule, the analyzer determines problem areas and assigns "blame" to certain elements in the schedule, and the prioritizer modifies the order in which the elements are considered. This process repeats until a satisfactory result is reached or allotted time runs out. However, our scheduling problem is intrinsically tied to execution and analyzing the initial schedule generated by itself is not satisfactory. Our approach (Figure 1) builds upon the usual SWO approach by incorporating a simulation of execution and Monte Carlo to build an execution sensitive result. We call our approach Priority Search as it searches the priority space using Monte Carlo simulation feedback to find a good set of priorities, unlike the static algorithms.

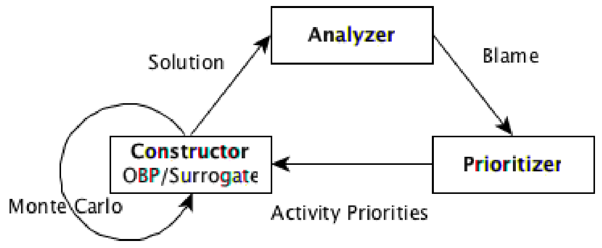


Figure 1: Squeaky Wheel accounting for Execution

Constructor

Typically, the constructor generates a schedule as the solution, which is then fed into the analyzer. However, our scheduling problem must be taken in context with execution. Activities may finish early or late which affect how many and which activities can be scheduled. In order to take this into account, we generate the final schedule of a (lightweight) simulation of the entire plan execution. This is simulated by letting activities finish early or late by a variable amount based on a probabilistic model of plan execution [5]. However, the probabilistic model may promote misleading results if only sampled once. As a result, our constructor (Algorithm 2) runs a Monte Carlo and simulates multiple plan executions, passing on all of the executed plans as the solution to the analyzer.

Algorithm 2 Monte Carlo Constructor

```
Input:
    A⟨p, R, e, dv, Γ, T, D⟩: List of activities with their individual constraints
    C: Constraints for the whole plan (e.g. available cumulative resources)
    N: Number of runs in the Monte Carlo
Output:
    S: List of all final schedules after simulating execution
1: $i \leftarrow 0$
2: while $i < N$ do
3:     $schedule \leftarrow$ simulation($A$, $C$)[6]
4:     $S_i \leftarrow schedule$
5:     $i \leftarrow i + 1$
6: end while
```

Priority Analyzer

The analyzer (Algorithm 3) takes the solution and assigns blame to problem areas. Since our objective is to schedule all mandatory activities and better switch cases, we blame all activities that are not scheduled. Since the solution is multiple schedules, there may be some Monte Carlo runs where the activities do not succeed or fail to be scheduled. For simplicity, we chose to blame any activity that was unscheduled in any of the schedules, but other approaches may assign blame according to how many times an activity was not scheduled.

Algorithm 3 Monte Carlo Analyzer

```
Input:
    $A\langle p \rangle$: List of activities with priorities
    $S$: List of all final schedules after simulating execution
Output:
    $U$: List of all unscheduled activities
    $score$: Score (objective function)
1: for each $S_i \in S$ do
2:     $U \leftarrow U \bigcup \{\forall a \in A | a \notin S_i\}$
3:     $score \leftarrow score +$ get_score($S_i$)
4: end for
```

Constant Step Prioritizer

A simple way to re-prioritize is to increase the blamed (unscheduled) activities' priorities by a constant step size s.

Typically, activities have varying degrees of flexibility due to their constraints (resources, dependencies, time, etc.).

[5]See Empirical Results for how that probabilistic model was generated.

[6]The final schedule after simulating execution.

Algorithm 4 Constant Step Reprioritization

```
Input:
    $A\langle p \rangle$: List of activities with priorities
    $U$: List of all unscheduled activities (from analyzer)
    $step$: Constant step size
Output:
    $A$: Best relative ordering of activities found
1: for each $a \in U$ do
2:     incrementRelativePriority($a$, $step$, $A$)
3:     for each $d \in a.dependents$ do
4:         incrementRelativePriority($d$, $step$, $A$)
5:     end for
6:     for each $sg \in a.switchGroup$ do
7:         incrementRelativePriority($sg$, $step$, $A$)
8:     end for
9: end for
```

Higher priority activities can consume resources (unit resources, energy, and data volume) or change state in a way that prevents lower priority activities from scheduling such that their constraints are satisfied. Increasing the blamed activities' priorities allows them to schedule earlier (scheduling order) which means they have more ”slack” to satisfy their constraints. The goal is that the algorithm will slowly promote less flexible activities to the top so that their constraints can be satisfied, and demoted activities are flexible enough to be scheduled in a more constrained plan.

When increasing the relative priorities of blamed activities, the existing relative priorities between certain groups of activities must remain enforced.

First, each switch group must maintain the relative priorities between each activity in the grouping. For each switch group, the activities $(s_1, \ldots, s_n)$ must be ordered such that those with higher resource consumption (time, energy, and data volume) have higher priorities as well.

Second, dependency relationships must be enforced such that (3) is held true.

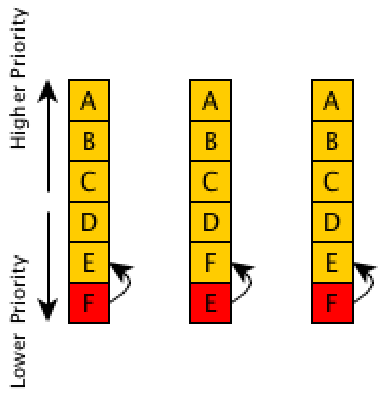


Figure 2: Cycle in the Constant Step approach. Red activities were unable to be scheduled and assigned blamed.

There is one main issue with the Constant Step approach

- it is extremely susceptible to cycles. One common cause for cycles is that a set of activities needs to be promoted beyond a particular activity together, but the constant step size prevents this from ever occurring. For example, in Figure 2 activity F is unschedulable and assigned blame. Its priority is increased, but this causes activity E to fail to schedule. Activity E is then promoted in the next iteration, causing F to fail to schedule and the process repeats. In reality, both E and F have to be promoted above D, but because the step size is constant, they will never achieve that and form a cycle. The situation where activities are unable to be promoted above an activity blocking it can be extended to any constant step size less than the maximum step size [7].

Stochastic Step Reprioritization

Algorithm 5 Stochastic Step Reprioritization

```
Input:
    $A\langle p \rangle$: List of activities with priorities
    $U$: List of all unscheduled activities (from analyzer)
Output:
    $A$: Best relative ordering of activities found
 1: $step \leftarrow random(1, A.length)$
 2: for each $a \in U$ do
 3:     incrementRelativePriority($a$, $step$, $A$)
 4:     for each $d \in a.dependents$ do
 5:         incrementRelativePriority($d$, $step$, $A$)
 6:     end for
 7:     for each $sg \in a.switchGroup$ do
 8:         incrementRelativePriority($sg$, $step$, $A$)
 9:     end for
10: end for
```

Injecting randomness to the step size allows the algorithm to become robust to cycles. In each iteration of the priority setting algorithm, a random step distance between 1 and N, where N is the number of activities in the plan, is assigned to all of the blamed activities. This lets the scheduler always have the possibility of being promoted above a resource constraining activity, while still allowing smaller step size priority permutations.

The main issue that lies with a random approach is that empirically [8] it finds the global maximum score slower than desired. This is further exacerbated by the fact that each iteration of our SWO cycle takes a non-negligible amount of time (a few seconds) due to the need to run a lightweight simulation and Monte Carlo.

Max Step Reprioritization

Stochastic Step Reprioritization (empirically) produced results slower than desired. Max Step Reprioritization seeks to solve both of those issues by always promoting blamed activities to have the highest scheduling priorities. The earlier an activity is considered for scheduling, the more flexibility that activity has to be scheduled. Therefore, if an activity

Algorithm 6 Max Step Reprioritization

```
Input:
    $A\langle p \rangle$: List of activities with priorities
    $U$: List of all unscheduled activities (from analyzer)
Output:
    $A$: Best relative ordering of activities found
 1: for each $a \in U$ do
 2:     $step \leftarrow A.length$
 3:     incrementRelativePriority($a$, $step$, $A$)
 4:     for each $d \in a.dependents$ do
 5:         incrementRelativePriority($d$, $step$, $A$)
 6:     end for
 7:     for each $sg \in a.switchGroup$ do
 8:         incrementRelativePriority($sg$, $step$, $A$)
 9:     end for
10: end for
```

is first to be considered for scheduling, but still cannot be successfully scheduled, there is no other scheduling priority that would allow the activity to be scheduled. Knowing this, by promoting blamed activities to have the highest scheduling priorities we can attempt to avoid iterations that fail to schedule the same blamed activities, thereby speeding up the overall algorithm.

Since the blamed activities will have the highest scheduling priorities, cycles such as those seen in Figure 2 can be avoided. However, Max Step Reprioritization doesn't prevent cycles entirely and they still pose an issue when encountered.

Empirical Evaluation

In order to evaluate how well our Priority Search algorithm is able to generate a priority assignment which results in an optimal schedule, we have applied the algorithm to various sets of inputs comprised of activities with their constraints and priorities and compared against various static algorithms. The inputs are derived from *sol types*. *Sol types* are currently the best available data on expected Mars 2020 rover operations (Jet Propulsion Laboratory 2017a). In order to construct a schedule and simulate plan execution, we use the *M2020 surrogate scheduler* - an implementation of the same algorithm as the M2020 onboard scheduler (Rabideau and Benowitz 2017), but implemented for a Linux workstation environment. As such, it is expected to produce the same schedules as the operational scheduler but runs much faster in a workstation environment. The surrogate scheduler is expected to assist in validating the flight scheduler implementation and also in ground operations for the mission (Chi et al. 2018).

Each input file contains approximately 40 activities. We use a probabilistic execution model based on operations data from the Mars Science Laboratory Mission (Jet Propulsion Laboratory 2017b; Gaines et al. 2016a; 2016b) in order to simulate activities completing early by a reasonable amount. In our model to determine activity execution durations, each of the actual execution durations provided in MSL data is first divided by the corresponding predicted execution dura-

[7] See section Max Step Reprioritization

[8] More information can be found in Empirical Evaluation.

tion. Then, we use a linear regression on the scaled values to obtain a mean and standard deviation presuming the ratio of predicted to actual execution times is normally distributed. The value representing the actual execution duration on the regression line for the given conservative duration is used as the mean. A scaled prediction of the actual duration is generated from a a normal distribution using the derived mean and standard deviation. Finally, this value is scaled back by multiplying by the given conservative duration. Note that we do not explicitly change other activity resources such as energy and data volume since they are generally modeled as rates and changing activity durations implicitly changes energy and data volume as well.

Using each of the sol types, we create variants by adding two switch groups to a set of inputs. Each switch group contains three switch cases where the switch cases differ in duration in a manner similar to the one described in (1). Each of the two switch groups are as follows:

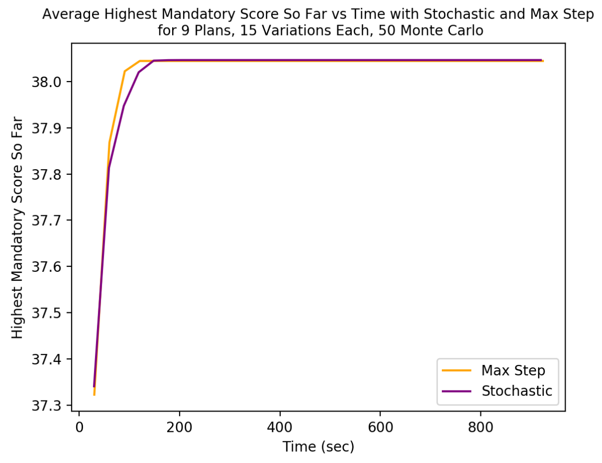
$$SwitchGroup = \begin{cases} Activity_{original} & \text{Duration=}x \text{ sec} \\ Activity_{2x} & \text{Duration=}2x \text{ sec} \\ Activity_{4x} & \text{Duration=}4x \text{ sec} \end{cases} \quad (4)$$

Due to the inequality in (2), a successfully scheduled mandatory activity is of much higher value than a successfully scheduled longer switch case. Therefore, the mandatory activity score is weighted at a much larger value then the switch group score. Each mandatory activity that is successfully scheduled is given one point which contributes to the mandatory score. If a switch case with a duration that is 2 times that of the original activity is able to be scheduled, then it contributes $1/5$ to the switch group score. If a switch case that is 4 times the original duration is able to be scheduled, then it contributes $2/5$ to the switch group. Since there are two switch groups in each variant, the maximum switch group score for a variant is $2 * (2/5) = 4/5$. In the following empirical results, we average the mandatory and switch groups scores over all Monte Carlo runs of execution.

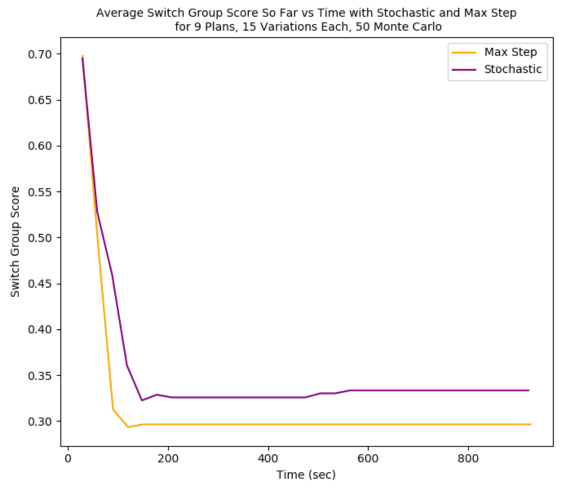
Also, in each of our variants we set the preferred schedule time of each activity to the earliest time the activity is allowed to start.

We first compare the different approaches to implementing Priority Search to understand which performs better.

The highest score so far is a combination of the mandatory score and the switch group score where the mandatory score is weighted at a much higher value than the switch group score. In Figure 3 we plot how the mandatory and switch case components of the highest score achieved up to the current time change over time using both the Stochastic method and the Max Step method. We do not consider the Constant Step method since it is so highly susceptible to cycles. For both methods, as the score for mandatory activities increases, the score for switch groups largely decreases until the highest mandatory score is reached. This is a reasonable outcome because as more mandatory activities are scheduled, the schedule likely becomes more constricted, thus making it more difficult to schedule longer switch cases. Since the mandatory score contributes much more to the total score than the switch group score and the mandatory sore



(a) Mandatory score component of highest score so far vs Time averaged across sol type variants using both priority search methods.

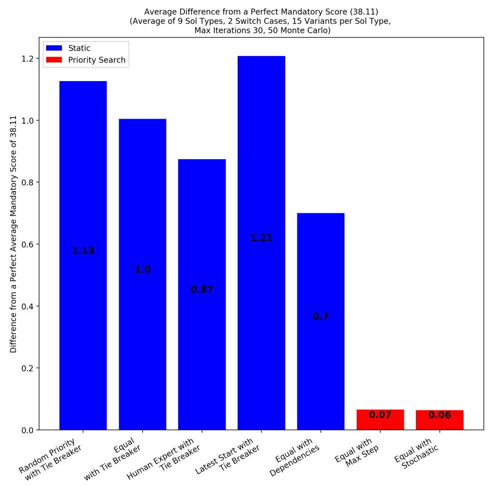


(b) Switch group score component of highest score so far vs Time averaged across sol type variants using both priority search methods.
.

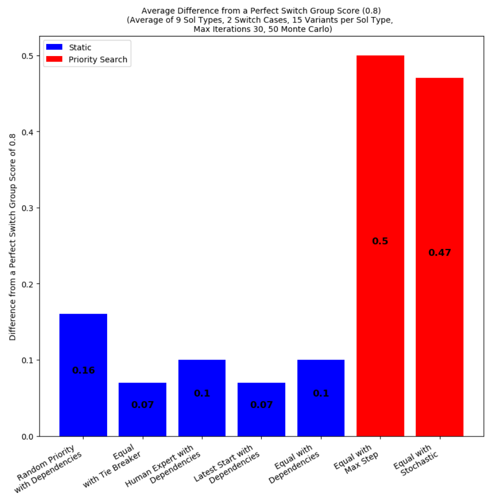
Figure 3: Plot of the highest score so far separated by mandatory score (3a) and switch group score (3b) over time using the Stochastic Step method and the Max Step method averaged over 9 sol types, each with 10 variants each containing 2 switch groups. Each iteration of Priority Search was run with 10 Monte Carlo runs and with 30 iterations of Priority Search alloted for each run of the algorithm.

is increasing in both figures, the total highest score so far is always increasing over time, as it should be.

Figure 3a shows that Stochastic Step reaches its highest mandatory score that is ever achieved over the time span of approximately 920 seconds (30 iterations of the priority search algorithm) in 207.58 seconds. The highest mandatory score achieved at this time and onwards is 38.047. The highest mandatory score using the Max Step method is reached at 120.59 seconds and has a value of 38.044. Figure 3b shows that the highest switch group score after the point at which the highest mandatory score is reached is 1.67 at 568.16 seconds using the Stochastic method and 1.48 at 150.87 seconds using the Max Step method. Therefore, we conclude that using the stochastic method results in a marginally higher total highest score but it takes less time to reach the highest score using the Max Step method.



(a) Difference from perfect mandatory score averaged across sol type variants for various scheduling methods.



(b) Difference from perfect switch group score averaged across sol type variants for various scheduling methods.

Figure 4: The difference from a perfect mandatory score of 38.11 and perfect switch group score of 1.0 using various scheduling methods is averaged over 9 sol types where 15 variants are derived from each sol type and each variant contains 2 switch cases. Each iteration of the Priority Search algorithm is run with 50 Monte Carlo runs of execution

Figure 4 shows the results of comparisons between Priority Search and other static priority setting algorithms. Since the scheduling of mandatory activities and switch groups are not weighted equally, we have constructed two separate plots to show the results for each. Both methods of Priority Search, in red, result in fewer unscheduled mandatory activities and consequently a lower difference from the perfect mandatory score. This implies they set the priorities such that more mandatory activities are able to be scheduled over multiple Monte Carlo runs compared to how the static algorithms set the activity priorities. As shown in 4b, they result in a higher number of unscheduled switch cases, likely because if more mandatory activities were scheduled it becomes more difficult to schedule longer switch cases. Due to the strict inequality described in (2), even though fewer longer switch cases are scheduled, the total scheduling score is still higher when using Priority Search. Thus, we conclude that both Priority Search methods outperform the static algorithms. Among the static algorithms, running the Dependencies algorithm with Tie Breaker on equal priorities performs the best as it results in the highest mandatory score while running Tie Breaker after setting the priorities based on the latest start time performs the worst.

Related Work

Our Priority Search approach is inspired by Squeaky Wheel Optimization (SWO). Typically, SWO uses a constructor and analyzer, and prioritizer for the next iteration of schedule generation (Joslin and Clements 1999). Priority Search differs in that the intent is not to generate a good schedule but rather to set priorities that perform well in execution and rescheduling. Therefore the Priority Search constructor must use the scheduler through multiple runs of execution (where each run of execution incurs multiple scheduler invocations) to assess priority assignment performance.

Generating schedules that are robust to execution run time variations (Leon, Wu, and Storer 1994) is a mature area of work. However, the topic usually revolves around developing a scheduler that can generate robust schedules. In our case, the scheduler is a) a fixed ”black box” that we have no control over and b) robust to execution run time variations mainly through rescheduling (Chi et al. 2018). As a result, rather than developing a scheduler itself, we're developing a methodology that is able to generate a set of priorities for a fixed scheduler that enables it to be robust to rescheduling due to runtime variations.

Other approaches (Drummond, Bresina, and Swanson 1994; Washington, Golden, and Bresina 2000) use branching to increase robustness - these differ from our work that adjusts priorities and allows rescheduling.

A number of other spacecraft (Muscettola et al. 1998; Pell et al. 1997; Chien et al. 2005; 2016) and rover (Woods et al. 2009; Gregory et al. 2002) autonomy systems have included planning, but these differ in that we are deriving control information specific to scheduling for a limited context - e.g. one rover sol. temporal schedule.

Discussion and Future Work

While we have focused on the impact of activity priority on the scheduler (and hence rescheduling during execution), there is often an execution system that may also have some flexibility to add robustness to the overall system (Chi et al. 2018). For the empirical evaluation described above, we ran without such an execution system. In the future, we could consider the execution system in the schedule and Monte Carlo analysis and potentially derive information usable by the execution system (e.g. allow an activity to run late but only until time T). This paper describes initial work to determine priorities for scheduler activity consideration ordering to optimize scheduler execution results for an embedded scheduler. However, this work is still preliminary with many other ideas to be explored as described below.

First, more sophisticated critique/blame assignment methods should be explored. Currently, priorities of activities that are not executed are modified, but more sophisticated analysis of scheduler runs could provide greater insight into how the priorities should be modified. Prior work in Process Chronologies (Biefeld and Cooper 1991) has been used to focus scheduling tactics by finding regions where time constraints or high demand for some resource results in conflict. By evaluating which periods of time or what resources are over-subscribed using Capacity/Over-Subscription Analysis, we can pinpoint which activities are more tightly constrained and increase their priorities. Prior work in Oversubscribed Scheduling Problems (Kramer and Smith 2006) show that scheduling according to maximum-availability (least subscribed) allows a suitable schedule to be generated. Similar analysis could be used to determine which activities to assign blame to and by how much to promote the blamed activities. We can also consider precedence constraints when deciding by how much to promote activity priorities. For every blamed activity, there is likely a scheduled activity that is using resources needed by the blamed activity. Precedence constraints could help discern which activity is using those resources. The blamed activity could then be promoted only as much as is necessary in order to be scheduled before the offending activity.

We can also implement several methods to help us explore different search spaces. Priority Search only adjusts priorities to improve execution and rescheduling performance. We could also add new activity precedence constraints (e.g. A must end before B starts) or enforce partitions in the schedule (e.g. all of these activities must be scheduled to end prior to 11 am). These types of constraints could drive the scheduler towards subsets of the schedule search space. Randomized restart can allow our priority search algorithm to better explore the global space rather than searching locally. Another alternative would be to keep a list of promising schedule priority assignments and backtrack to those randomly, allowing us to better explore the search space.

We can also make improvements to our Monte Carlo method and use the resulting simulations for further analysis of the scheduler. In order to build a model of run time variations that is not overly skewed, we use Monte Carlo to repeatedly sample a variety of execution run time results. Standard Monte Carlo simulations tend to focus most runs on the nominal cases, but a more effective methodology samples edge cases but weighs the cases by their likelihood to increase coverage of the variability in the space (in this case variable activity execution times). The Monte Carlo of execution run time variations can provide valuable information for why activities fail to schedule, what input plans are best suited for the current scheduler design, and how the current input could end up executing. We are working on visualizing this information to better inform those working with the scheduler.

Currently, we only test with mandatory activities. In the future, we will extend our approach to include optional activities, which will add further complexity to the algorithm and analysis. Optional activities are lower priority activities what are nice to have scheduled, but not necessary. They are generally only able to be scheduled if mandatory activities end early or consume less resources than expected. We also plan to use an activity's actual scheduled preferred time while testing.

Cycles pose an issue to both Constant Step Reprioritization and Max Step Reprioritization. Better cycle detection would allow us to not only overcome the issues presented, but also provide additional information on how to permute the priority set for the next iteration. For example, cycle detection could allow us to not only detect the cycle in Figure 2, but know that both E and F should be incremented together.

While we have established a few methods to improve the prioritizer and decide on the next permutation of activity priorities, we have utilized the same objective function to determine the success of our algorithm. However, our objective function is simple and coarse; oftentimes, the same score will appear repeatedly in multiple consecutive. As a result, the algorithm often travels swaths of plateaus before sharply improving. This choppiness is suboptimal for Squeaky Wheel Optimization and gradient descent problems in general. Some potential additions to the objective function could be how much energy is leftover in the plan or how close an activity is to their preferred scheduling time. Evaluating a more precise objective function can reduce choppiness and better steer the algorithm towards a more optimal solution.

Conclusion

We have presented a study of methods to assign activity priorities to control a limited, embedded scheduler to optimize rescheduling for a specific problem. We first define a set of static methods that assign activity priorities based on heuristics and schedule dependencies. We then describe how these priorities can be further adjusted based on feedback from simulated execution and rescheduling using Monte Carlo methods to perform Priority Search. We present an empirical evaluation of several static and priority search methods using best available planetary rover operations data. This empirical evaluation shows that Priority Search outperforms static methods including human expert derived priorities. Finally we describe a number of promising areas for future improvements to our algorithms.

Acknowledgments

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

References

Biefeld, E., and Cooper, L. 1991. Bottleneck identification using process chronologies. In *IJCAI*, 218–224.

Chi, W.; Chien, S.; Agrawal, J.; Rabideau, G.; Benowitz, E.; Gaines, D.; Fosse, E.; Kuhn, S.; and Biehl, J. 2018. Embedding a scheduler in execution for a planetary rover. In *ICAPS*.

Chien, S. A.; Sherwood, R.; Tran, D.; Cichy, B.; Rabideau, G.; Castano, R.; Davies, A.; Mandl, D.; Trout, B.; Shulman, S.; et al. 2005. Using autonomy flight software to improve science return on earth observing one. *Journal of Aerospace Computing Information and Communication* 2(4):196–216.

Chien, S.; Doubleday, J.; Thompson, D. R.; Wagstaff, K. L.; Bellardo, J.; Francis, C.; Baumgarten, E.; Williams, A.; Yee, E.; Stanton, E.; et al. 2016. Onboard autonomy on the intelligent payload experiment cubesat mission. *Journal of Aerospace Information Systems*.

Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *AAAI*, volume 94, 1098–1104.

Gaines, D.; Anderson, R.; Doran, G.; Huffman, W.; Justice, H.; Mackey, R.; Rabideau, G.; Vasavada, A.; Verma, V.; Estlin, T.; et al. 2016a. Productivity challenges for mars rover operations. In *Proceedings of 4th Workshop on Planning and Robotics (PlanRob)*, 115–125. London, UK.

Gaines, D.; Doran, G.; Justice, H.; Rabideau, G.; Schaffer, S.; Verma, V.; Wagstaff, K.; Vasavada, A.; Huffman, W.; Anderson, R.; et al. 2016b. Productivity challenges for mars rover operations: A case study of mars science laboratory operations. Technical report, Technical Report D-97908, Jet Propulsion Laboratory.

Gregory, N. M.; Dorais, G. A.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International Workshop on Planning and Scheduling for Space*. Citeseer.

Jet Propulsion Laboratory. 2017a. Mars 2020 rover mission https://mars.nasa.gov/mars2020/ retrieved 2017-11-13.

Jet Propulsion Laboratory. 2017b. Mars science laboratory mission https://mars.nasa.gov/msl/ 2017-11-13.

Joslin, D. E., and Clements, D. P. 1999. Squeaky wheel optimization. *Journal of Artificial Intelligence Research* 10:353–373.

Kramer, L. A., and Smith, S. F. 2006. Resource contention metrics for oversubscribed scheduling problems. In *ICAPS*, 406–409.

Leon, V. J.; Wu, S. D.; and Storer, R. H. 1994. Robustness measures and robust scheduling for job shops. *IIE transactions* 26(5):32–43.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence* 103(1-2):5–47.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997. Robust periodic planning and execution for autonomous spacecraft. In *International Joint Conference on Artificial Intelligence*, 1234–1239.

Rabideau, G., and Benowitz, E. 2017. Prototyping an onboard scheduler for the mars 2020 rover. In *International Workshop on Planning and Scheduling for Space*.

Washington, R.; Golden, K.; and Bresina, J. 2000. Plan execution, monitoring, and adaptation for planetary rovers. *Electron. Trans. Artif. Intell.*

Woods, M.; Shaw, A.; Barnes, D.; Price, D.; Long, D.; and Pullan, D. 2009. Autonomous science for an exomars rover–like mission. *Journal of Field Robotics* 26(4):358–390.