ngVLA Multi-subarray Scheduling Algorithm
ngVLA Computing Memo # 9

Rafael Hiriart

January 16, 2024

1 Introduction

This note analyses several requirements applicable to the online Scheduler for ngVLA. The need to optimize and automate sub-array operations will substantially increase the complexity of this component for ngVLA, compared with the dynamic schedulers implemented for JVLA and ALMA. Scheduling problems can be notoriously hard to solve, and for some problems no practical algorithms are known to exist. In order to reduce this risk, this note attempts to formalize the relevant requirements and classify the type of scheduling problem required by ngVLA. A candidate algorithm is proposed which transforms the dynamic sub-array partitioning and SB allocation problems into the graph *shortest-path* problem.

2 Requirements

The fact that the ngVLA is required to create both a schedule of sub-arrays along with a schedule of the SBs that will be executed in these sub-arrays in an optimal way (for a given optimization function) makes the algorithm substantially different from similar systems like ALMA or JVLA, which basically concern themselves only with creating a schedule of SBs to run in a single sub-array. The single-subarray problem is essentially a “greedy” algorithm, a class of algorithms that always make the choice that looks best for a given moment. (See Appendix A for a description of the VLA algorithm.) For some problems, this strategy produces an optimal or close to optimal solution, most of the time. However, for multiple sub-array scheduling the situation changes. It becomes necessary to to evaluate different alternatives of forming sub-arrays on longer time-scales to find the best solution. Indeed, for multiple sub-arrays, executing the highest ranked SB that is observable at any given time is not necessarily the best approach. For instance, it may be better to defer the execution of the highest ranked SB and execute lower ranked SBs in a different sub-array partition instead. This path may produce a better global solution for the problem.

How do we define what it is meant by a "better" solution? First, it is necessary to define the optimization function for this problem, which hasn't been defined yet. Similarly as done for ALMA and JVLA, this function needs to incorporate the attributes of each SB and their relative priorities, but it should also include the notion of time efficiency; that is, the need to execute all the accepted SBs in the minimum amount of time possible. It is this last condition that makes the problem a global optimization problem. Given that the single-subarray scheduling problem is greedy, its "optimization function" can be described as *finding the best SB to run at a given moment*. This is different than *choosing the best way of running a pool of SBs during an entire observing season, considering that executions can be parallelized by using sub-arrays*. Time efficiency doesn't appear in the single-subarray problem at all, being subsumed in its "greediness".

One way of defining multiple sub-array scheduling as an optimization problem is:

$$\text{Min} \sum_i w_i C_i \tag{1}$$

where C_i corresponds to each SB completion time (the time of completion of the SB, not its duration!) and w_i is a weight per SB. This is an optimization function typically used in scheduling algorithms, along with the maximum "span" C_{max}, the maximum completion time or the total time length of the schedule. The advantage of the weighted sum is that it allows us to incorporate directly a weight per each SB in the optimization function. These weights can be based on the scientific rank of each SB and other criteria, as it has been done in ALMA and JVLA. For a single sub-array, the algorithm will prefer to execute the SBs with the larger weights first. This is easy to demonstrate: if we swap any two SBs in this solution, then the sum increases. This is also true for multiple sub-arrays, where it is necessary to execute the SBs belonging to each sub-array in order of decreasing weights. The difference comes from the extra degree of freedom of having to partition the array in sub-arrays. This will "scramble" the global ordering of executions, but not the ordering for a single sub-array. We will assume that a function similar to the above is used to drive the search for an optimal solution.

The Scheduling Concept document[2] mentions that the optimization function will most likely be formed as a function of the "antenna-hours" and the rank of each SB, which needs to be maximized. The antenna-hours[1] are used to account for the fact that the same SB can be executed in different sub-arrays. The sub-array where the SB will end up running may be different than the sub-array that was used to calculate the target time assigned to each SB, so some

[1] I haven't seen a definition for the "antenna-hours" yet. Given that the sensitivity for a point interferometric observation is proportional to $1/\sqrt{N^2\tau}$, with N the number of antennas and τ the observation time, then I would think that in order to maintain the sensitivity constant when observing an SB in different sub-arrays, it would be necessary to maintain $N^2\tau$ constant, not $N\tau$, my naive definition for "antenna-hours". If this is correct, may be it would be more proper to call it "baseline-hours" instead. Anyway, I don't quite understand the "antenna-hours" yet. It should be defined formally at a given point.

way of scaling the allocated time is needed. The intention behind this optimization function seems to be to maximize the use of the antenna-hours executed during an observing season, giving preference to the highest ranked SBs. Note that this function doesn't incorporate the notion of time efficiency by means of the completion times like the function we were suggesting above. In fact, if we sum all the SB ranks scaled in some way by their antenna-hours, for all the SBs in the pool, there's actually nothing to optimize in the scheduling sense, as the sum is constant for any schedule. I assume that this is because the sum is formed only over a sub-interval—the observing season— of the total length of the schedule, or in different words, the pool contains more SBs that we can possibly observe in an observing season, and we only sum for the observing season. If this is correct, then it can be demonstrated that minimizing the sum of the weighted completion times actually maximizes the scaled sum of the ranks. See the end of Appendix A for the demonstration and some additional discussion.

The optimization function is one of such sub-problems that need to be defined for the scheduling algorithm to operate. The others are:

SB weighting algorithm The weights are a function of several factors: the science rank, the stringency of the observation, the degree of completeness, and the SB weather requirements versus current conditions, amongst others. Weights can be dynamic, e.g. high frequency observations requiring low PWV and atmospheric stability should increase in priority when these conditions are present.

SB filtering algorithm Depending on the present weather, hour angle, or hardware availability, some SBs may not be observable at all. We want to remove these SBs from the pool to be considered prior to running the algorithm.

SB to sub-array mapping algorithm A function that maps each SB with the sub-array where it should be executed is necessary. This mapping depends on SB characteristics such as its Largest Angular Scale (LAS) and required resolution. This mapping is probably not strict: it makes little difference if a few antennas are removed or added from/to a sub-array. We assume that each SB can be mapped to one or more sub-arrays. For now we will assume that this mapping is 1:1 and relax this condition afterwards.

The above algorithms, which belong clearly to the scientific domain, are used by the scheduling optimization algorithm. We assume that they will be defined by the Science IPT. The first two may not be very different from similar algorithms constructed for ALMA and the JVLA.

The algorithm is required to produce an optimal schedule in three timescales:

The long-term scheduler Where it is necessary to produce a schedule for an entire observing season or a longer period of time, based on historical weather data.

The short-term scheduler Where the schedule spans only one or a few days, based on forecasted weather data.

The real-time scheduler Where the scheduler optimizes only for the few next hours, based on the current (measured) weather conditions.

Although the long-term scheduler and the short-term scheduler are SB-based, the real-time scheduler could be Scan-based, in order to facilitate the handling of triggered observations and pausing/resuming observations.

The solutions for these three schedulers should be consistent; i.e., if the weather conditions were to be exactly the same as recorded in the historical data, then the short-time schedule would be just a sub-schedule inside the long-term schedule and likewise the real-time schedule would be an even smaller sub-schedule inside the short-time schedule. This is rarely the case, as deviations from the expected weather patterns will introduce differences between them. We foresee the need for an "integrated" scheduler system, where as the actual weather plays out, deviations from previous solutions are propagated to update the longer timescale solutions, allowing the operation planning groups responsible for them to visualize the impact of these deviations (e.g. with pressure plots) possibly adjusting the inputs of the shorter timescale schedulers to better align them with overarching operational goals (e.g. get all the SBs ranked A observed, equalize the completion of SBs across different science domains, etc.)

3 Multi-Processor Task Scheduling

The study of scheduling algorithms is an active area of research in computer science, mathematics, and operations research. There are many different types of scheduling problems, which are usually classified by means of a triplet $\alpha|\beta|\gamma$, where α specifies the machine environment, β specifies the job characteristics, and γ denotes the optimization function. See [1] and [4]. For example, the single-subarray problem with the optimization function proposed above is denoted by $1||\sum w_i C_i$. This represents a scheduling problem with one machine, general job characteristics, and the weighted sum of the job completions as objective function. (The solution for this problem is to sequence the jobs by order of nondecreasing ratios p_i/w_i, where p_i is the processing time.) It is worthwhile to study the already existing algorithms, even if the cases published aren't an exact match for the ngVLA scheduling problem. A suitable algorithm can be a starting point from where to develop our particular scheduling system, and provide a useful analysis framework to asses its complexity. In some cases, scheduling problems may be demonstrated to be NP-complete or NP-hard. This is a strong indication that we may be better off looking at approximation algorithms instead of attempting to find optimal solutions.

Scheduling problems are characterized in terms of "machines" and "jobs" that are executed in these machines. In some cases, the jobs can be sub-divided in "operations". Jobs and operations are naturally mapped to scheduling blocks and scans in our case. We could map machines to sub-arrays if it weren't be by the condition that *incompatible* sub-arrays which have a non-empty intersection of common antennas can't be executed in parallel. This type of problem falls in the category of Multiprocessor Tasks (MPT), where each job (or task) requires one or more processors (antennas in our case) at a time. This category of scheduling problem takes into account our condition of sub-array incompatibility. Specifically, the problem denoted by

$$MPTm|p_i = 1| \sum w_i C_i \tag{2}$$

seems to be a good starting point. This is a MPT problem with m machines (antennas), unitary processing time, and the weighted sum of completions as optimization function.

The condition $p_i = 1$ is not as restrictive as it seems as the "unit of time" can be defined as small as it is convenient (e.g. 0.5 hour if SB-based, or 5 minutes if Scan-based). In fact, it is a valuable simplification given that the dynamic scheduler needs to synchronize the creation and destruction of sub-arrays anyway. This algorithm can provide a base solution that can be adjusted in a second pass to account for non-integral processing times with respect to the choice of unit time. Also, as it is expected that different observations share calibration scans in ngVLA and other related optimizations, the exact processing time for each SB may not be known before execution, and the scheduler may need to rely on estimations.

The algorithm for this problem transforms the scheduling problem into the *shortest-path* graph problem, for which several efficient algorithms exist. We explain how this works by means of a simple example.

Let's imagine that we have 4 SBs $\{SB_1, SB_2, SB_3, SB_4\}$ and 5 antennas $A = \{A_1, A_2, A_3, A_4, A_5\}$. Using the *SB to sub-array mapping algorithm*, we associate each SB with a sub-array $\mu_i \subseteq A$. Using the *SB weighting algorithm*, we assign each SB a weight w_i:

$SB_1 \rightarrow \mu_1 = \{A_1, A_2\}$, $w_1 = 1$
$SB_2 \rightarrow \mu_2 = \{A_1, A_2, A_3\}$, $w_2 = 2$
$SB_3 \rightarrow \mu_3 = \{A_4, A_5\}$, $w_3 = 5$
$SB_4 \rightarrow \mu_4 = \{A_1, A_2\}$, $w_4 = 3$

First, we compile the list of sub-arrays and their corresponding SBs, ordered by nonincreasing weight. The real algorithm is more complicated, because the ordering depends on more factors than just the weights (e.g. target observability, which would be part of the criteria considered by the *SB filtering algorithm*), but for now assume that for each sub-array, the SBs are observed in the ordering defined only by their weights.

$SA_1 = \{A_1, A_2\} \rightarrow (SB_4, SB_1)$
$SA_2 = \{A_1, A_2, A_3\} \rightarrow (SB_2)$
$SA_3 = \{A_4, A_5\} \rightarrow (SB3)$

We define a tuple $(t; i_1, i_2, i_3)$, where each i_j represents the partial schedule of all the SBs that have been observed in the interval $[0, t]$. In other words, the i_j's represent the number of SBs observed so far in the sub-array. For our example the i_j's are defined in Table 1.

i_j	0	1	2
SA_1	-	SB_4 $(w = 3)$	SB_1 $(w = 1)$
SA_2	-	SB_2 $(w = 2)$	-
SA_3	-	SB_3 $(w = 5)$	-

Table 1: Tuple definitions for each sub-array.

We then form a graph following these rules:

- Start with the tuple $(0; 0, ..., 0)$, i.e. at time 0 nothing has been observed. The end tuple is $(4; 2, 1, 1)$, that is, the tuple that represents all the SBs observed serially.
- The immediate successor of vertex $u = (t; i_1, ..., i_R)$ is given by $v = (t + 1; i_1 + x_1, ..., i_R + x_R)$, where $x_1, ..., x_R$ satisfy the following conditions:
  - $x_\nu \in \{0, 1\}$ for $\nu = 1, ..., R$
  - During each period, only one SB per sub-array can be scheduled, and the scheduling of the SBs in each tuple needs to be compatible.
- Each vertex that starts at time t has a cost given by $\sum w_i(t+1)$, where the sum covers all the SBs that are observed next (i.e., the ones that increased i_j by 1).

The resultant graph for our example is shown in Figure 1. Note that in each one of the tuples, after the first number that represents the time, the next numbers represent each possible sub-array. For each one of these sub-arrays, the corresponding number is increased if we decide to observe the next SB in this sub-array; otherwise, the number stays the same. For creating the tuples of the graph, we don't actually need to know what specific SB is observed, we only care about the sub-arrays compatibility. The specific SBs that are observed in each sub-array are only considered when computing the cost of each arc.

Finally, we solve for the shortest path. This path represents the SB and sub-array schedule that minimizes the optimization function, as shown in Figure 2.

Note that the algorithm works with 3 structures:

- The sub-array SB table, examplified in Table 1. Although in the example the SBs are ordered only by weight, in reality the ordering can be quite more complex. The targets for each SB needs to be *observable*; the weights can be dynamic, changing with the specific weather conditions at each time or increasing in priority if other SBs in the program has been observed. There is also the need to consider SB dependencies in the ordering. How to order the SBs for each sub-array, on the other hand, is a problem that has been solved for JVLA and ALMA. It is the single-subarray problem.

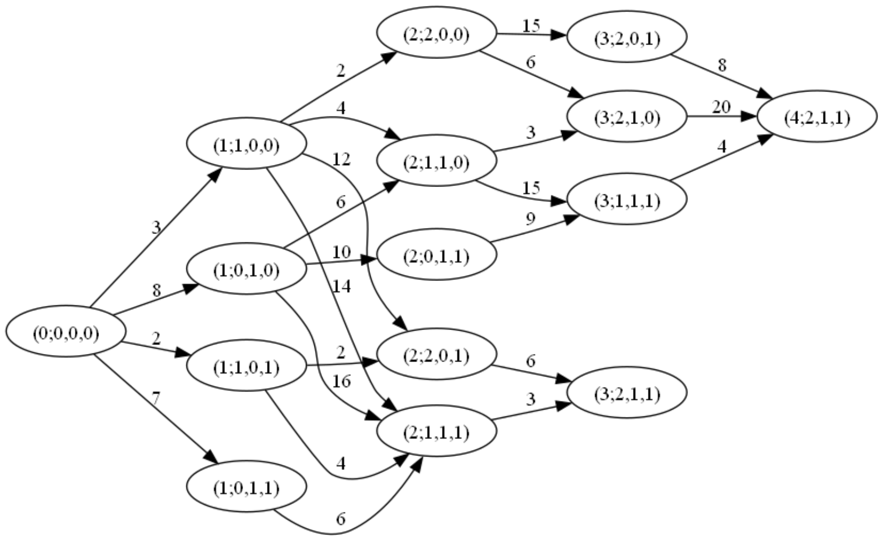

Figure 1: Example graph for a MPT scheduling problem.

- The sub-array compatibility chart. As noted before, this is the only structure that is needed when forming the graph (besides the total number of SBs to be executed in each sub-array). What specific SB is executed next needs to be considered only when computing the cost of each arc. Similarly, it is not necessary to know what specific antennas are included in each sub-array, only whether they are compatible with each other. This suggests the idea that the sub-arrays may be "fuzzy". They may contain more or less antennas, in which case the observation time will need to be adjusted. For forming the graph, this doesn't matter so much, as we only need to consider their (fuzzy) mutual compatibility.
- The graph, formed from the above two structures.

As an analogy, we can compare this algorithm applied to multiple levels (long-term, short-term, and real-time) with a hiker that needs to traverse some unknown terrain. The graph acts like a map for deciding the least costly route to follow. The hiker can't possibly know the best route to follow without considering the whole map. The problem is that when he starts, the map is not very good as it is based solely on unreliable historical information. As the hiker goes on his journey, he will need to correct the map (by changing the arc costs depending on weather conditions hardware availability, etc.), which forces the hiker to re-plan the route frequently, basing his judgement on what he can now see of the terrain, and his best guess of what he can't.

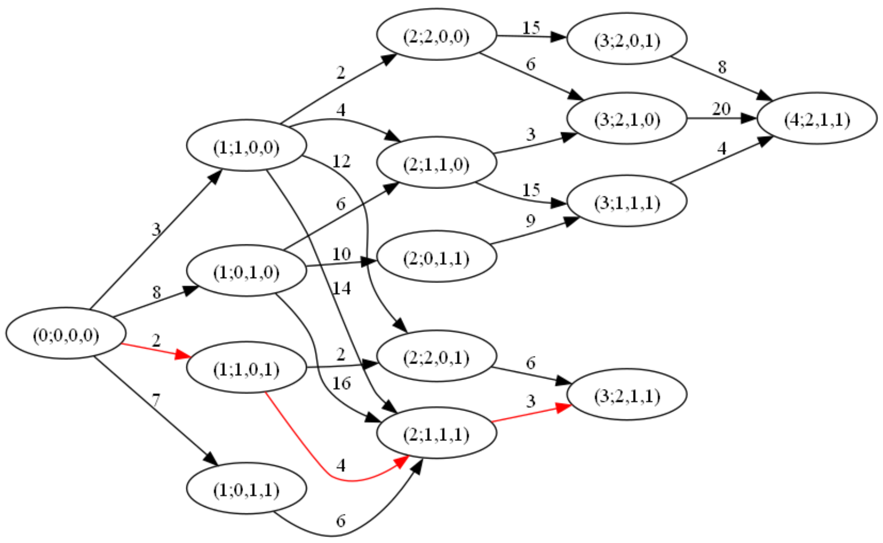

Figure 2: Solution for the example MPT scheduling problem. The schedule displayed in red is optimal, with a cost of 9.

4 Application Considerations of MPT Scheduling

In the worst case the shortest path can be calculated in $O(R2^R n^{R+1})$ with R being the number of possible sub-arrays (a couple of dozens?) and n being the number of SBs (a few thousands?). In practice we can expect that the runtime should be considerable better as the SBs are filtered for each time, and we only need to consider their respective sub-arrays to form each tuple combination. (And also this $O(\cdot)$ calculation is an upper-bound calculation.) We could prototype the algorithm using data from the ROP and EOP to assess its performance characteristics in realistic scenarios.

An important point is that the graph only needs to be formed when solving the long-term scheduler for the first time, based on historical data. Then the graph can be stored and maintained in subsequent runs, modifying it to account for changes related with the use of forecasted or measured data. These changes are mostly changes in the arc costs, the graph topology not being affected much[2]. This suggests that maintaining the graph should be a relatively inexpensive operation (TBC). As time goes on, the shortest-path calculation should also become more inexpensive, as only a fraction of the graph needs to be considered.

Given that the graph is persistently stored, along with the history of each SB that have been already observed (the path followed in the graph so far),

[2]Unless an SB is removed, in which case nodes needs to be removed from the end of the graph. This may be ignored as the pool of SB contains more SBs that can be scheduled in an observing season.

it should be relatively easy to “fold” the graph to calculate pressure plots and simulate what would happen when the scheduling parameters are modified. As the three multi-level schedulers rely on the same graph, their results should be consistent.

5 Extensions

The first extension that may be needed is related with the *SB to sub-array mapping algorithm*. Instead of a 1:1 mapping between an SB and a sub-array, we may have a 1:n mapping, with n being a few (“less flexible” sub-arrays in [2]) to a large number of possibilities (“more flexible” sub-arrays). This case is actually found in [1] and is called Multi-Mode Multiprocessor-Task Scheduling Problems (MMPT). These problems are composed by an *assignment problem* (the problem of assigning a sub-array to each SB from a set of candidate sub-arrays), followed by a MPT problem (the problem of forming the schedule, as described before). Although still possible to be solved in polinomial time, this kind of problems explodes combinatorially (the possible assignments are bounded by n^{2^m}, with n the number of SBs and m the number of antennas, see [1] for the development). This strongly suggests to favor the “less flexible” approach. We may yet find a clever way to introduce this condition, but it is clearly a research problem. On the other hand, do this assignment really needs to be so flexible? It is clear that there must be constraints on what sub-arrays can be used to observe each SB. Mapping each SB to a *reference* sub-array with possible alternatives that can be optimized for a given graph path node may be an option. Each path node represents both a partition of sub-arrays and an SB allocation on them. We may want to optimize each node by considering alternative sub-array partitions following some criteria yet to be defined. This may be more tractable than combinatorially trying every possible assignment of sub-arrays to SBs and solving the associated MPT problem.

Another extension, applicable to the real-time scheduler, is to perform the scheduling on scan units instead of SB units. We can assume this is only applicable to the real-time scheduler, while the long and short-term schedulers remain SB-based. This may present several advantages:

- It simplifies handling triggered observations and in general the external coordination required by VLBI and “follow-along”.
- It simplifies pausing and resuming observations, and the adoption of other reliability techniques (e.g. swapping antennas in case of failures).
- It simplifies making observations to adapt to changing conditions during an observation (e.g. the scheduler may decide to omit or add calibration scans, adapt to changing RFI conditions, etc.)
- Enables inter-SB optimizations, like sharing calibration scans.

- It simplifies "recursive sub-arraying", where a sub-array is broken into sub-sub-arrays, which observe independently, and can be merged later, etc.

For the scheduling algorithm, this doesn't seem to be overly complex. Each node in the graph would be unpacked to form multiple scans nodes, which would be scheduled and submitted for execution independently, with the scheduler dynamically introducing optimizations like the ones itemized above. As this may be a bit "experimental", and the value of these optimizations are yet to be demonstrated, we should probably introduce scan-based along with the traditional SB-based scheduling and execution system.

A The VLA Algorithm

The VLA dynamic scheduling algorithm is described in detail in [3]. A summary is provided in this section.

The algorithm proceeds in three stages:

1. Ineligible SBs are filtered from the pool of schedulable SBs.
2. A priority is assigned to each eligible SB.
3. An observing schedule is created for a given time horizon, with the default being 24 hours.

In the filtering stage, the filters described in Table 2 are applied to select the eligible SBs from the pool of schedulable SBs.

In the prioritization stage, a priority is calculated for each SB according to the criteria shown in Table 3. Priorities are assigned to each Project, Program Block and Scheduling Block, with the most granular priority overriding the priority for its parent structure. The different priority criteria are combined as a weighted sum to calculate the final priority for each SB:

$$P = \alpha_p p + \alpha_u u + \alpha_s s + \alpha_o o + \alpha_n n + \alpha_\sigma \sigma + \alpha_l l \tag{3}$$

After calculating the priorities, the algorithm creates the observing schedule as follows:

1. Add all the eligible and fixed date SBs to the schedule if their time intervals overlap with the schedule window. Set aside any fixed date SB that doesn't overlap with the schedule SB, to be used in step 4 below.
2. Sort all the remaining dynamic SB blocks by priority.
3. For each of the SBs from step 2:
   (a) Try to fit the SB in between already scheduled SBs without moving the already scheduled SBs.

Filter	**Description**
API	The SB's RMS phase limit must be above the current API (Atmospheric Phase Interferometer) reading.
Wind Speed	The SB's Wind Speed limit must be above the current Wind Speed reading.
Array Configuration	The SB must be eligible to be run in the current configuration.
Receiver Band	The SB must contain receiver bands that can be scheduled currently.
Date (Dynamic)	The SB's earliest start date must come before the end of the schedule window.
Date (Fixed Date)	The SB's fixed start time must be from 24 hours before the schedule window to 24 hours after the schedule window.
LST	The SB's Local Sidereal Time (LST) start range must overlap with the schedule window. This filter does not apply to Fixed Date blocks.
SB Type	Only "normal" SBs are considered. Manual SBs must be submitted manually from the client.
Time Remaining	Only blocks with enough time remaining in their containing Program Block (PB) are considered. Six minutes of sidereal time "grace" period has been allowed, to allow for some uncertainty in time calculations.
Scheduling Status	Only SBs in "schedulable" status are considered.

Table 2: SB filtering criteria applied in the VLA scheduling algorithm.

   (b) If the previous step fails, try to insert the SB into a too-narrow gap in the schedule by shifting as many SBs as necessary to the right (later in the schedule) in order to make room.

4. Add the fixed date SBs that were set aside in step 1 to the schedule. These are added in this step rather than in step 1 so these "out of bounds" blocks don't cause confusion while scheduling the dynamic SBs.

Note that in steps 3a and 3b, the algorithm considers the LST ranges of both the SB to be inserted and the SBs already scheduled (an SB may contain multiple alternative LST ranges). It only inserts an SB by moving the already inserted SBs in step 3b if all the SBs that needs to be moved still comply with their LST ranges in their new locations. The algorithm also considers SBs requiring multiple iterations.

The final schedule is presented to the operator, who can accept all or part of the schedule. The current practice is that only the first SB in the schedule is accepted for observation, and the algorithm is executed again just as the SB

Type	**Range**	**Default Weight**	**Description**
Primary (p)	[1, 3]	$\alpha_p = 3.0$	From the TAC. Scheduling priority. $A \rightarrow 1$; $B \rightarrow 2$; $C \rightarrow 3$.
Urgency (u)	[0, 2]	$\alpha_u = 1.0$	0: Short time scale transient; 1: Partially completed projects and completion badly needed; 2: Normal science and tests.
Science (s)	[0, 10]	$\alpha_s = 0.03$	The linear-rank score from the Science Review Panel.
Override (o)	Any	$\alpha_o = 1.0$	To allow schedulers to force the algorithm in special cases.
Nice (n)	[0, 1]	$\alpha_n = 1.0$	To allow observers to lower the priority of their own SBs (not exposed currently).
Stringency (σ)	$\log_2(API) + \log_2(wind)$	$\alpha_\sigma = 0.33$	Favors blocks with strict weather limits.
Length (l)	$\log_2(SB_{length})$	$\alpha_l = -0.25$	Favors longer Scheduling Blocks.

Table 3: Components in the calculation of priorities, with their default weight. The first three components (p, u, s) are required, and the last two (σ, l) are calculated.

execution is ending.

This algorithm is greedy in the sense that during each iteration it tries to schedule a single SB regarding all the gaps left in a partially filled schedule as available[3]. The algorithm will schedule each SB by order of priority *as soon as possible*. In fact, as the higher ranked SB will have their completion times lower than the lower ranked SBs (they are executed first), this algorithm minimizes the objective function in Equation 1. In this equation, the weight w for each SB has a higher value when its priority is higher, while in the description of the VLA algorithm above P has a lower value when its priority is higher. Hence, we can consider one as the reciprocal of the other, i.e. $w = 1/P$.

If we ignore the constraints imposed by the LST ranges, the single subarray problem can be modeled as a $1||\sum w_i C_i$ queueing problem. The solution for this problem is scheduling the SB in order of non-decreasing ratios d_i/w_i:

$$\frac{d_1}{w_1} \leq \frac{d_2}{w_2} \leq ... \leq \frac{d_n}{w_n} \tag{4}$$

where d_i is the duration of SB i, and w_i its weight (or inverse priority). If we assume $d_i = 1$

$$w_1 \geq w_2 \geq ... \geq w_n \tag{5}$$

To demonstrate this, let's assume that SB j has been scheduled immediately

[3]Granted, the algorithm does modify the other SB locations when nudging them forward to make room, but it's arguibly still fundamentally greedy, scheduling one SB at a time. This is adequate for single-subarray scheduling.

before SB i. If we interchange i and j, the optimization function grows by

$$w_i d_i + w_j(d_i + d_j) - w_j d_j - w_i(d_i + d_j) = w_j d_i - w_i d_j$$

which is always non negative because $d_j/w_j \leq d_i/w_i$. Hence any swap in the schedule doesn't make the objective function decrease and scheduling the SBs as shown in equation 4 minimizes $\sum_i w_i C_i$.

Now, if we assume a solution that minimizes $\sum_i w_i C_i$, then this solution will comply with equation 4 and hence it will also be a solution of

$$\text{Max} \sum_i \frac{w_i}{d_i} \tag{6}$$

where the sum is performed over any sub-interval of the schedule starting at the beginning of the schedule. This can be seen as an alternative way of stating the optimization function, which makes clear that we are trying to "pack" as many higher ranked SBs as possible in the schedule. Dividing the weight by the SB duration has the effect of discouraging early scheduling of long running SBs, which is similar to what the VLA does by using a negative weight α_l in the calculation of the priority[4]. As noted before, it is necessary to consider the case of an SB running in a different subarray that the one used to calculate its duration. This can be introduced in this equation as a normalization of the durations d_i, using antenna-hours (or baseline-hours).

References

[1] Peter Brucker. *Scheduling Algorithms.* Springer, 2006.

[2] Bryan Butler. NGVLA Observation Scheduling Concept. Technical report, NRAO ngVLA, 08 2023.

[3] Bryan Butler and Keith Cummings. The VLA Scheduler. Technical report, NRAO, 2011.

[4] Michael L. Pinedo. *Scheduling, Theory, Algorithms, and Systems.* Springer, 2015.

[4]Note that we use the word "weight" with two meanings: as the weight for each SB as in equation 1, which is the inverse of the SB priority; and as the weights in the VLA prioritization function in equation 3