Incremental Scheduling with Upper and Lowerbound Temporospatial Constraints

Giancarlo F. Sturla, Matthew C. Gombolay, and Julie A. Shah
Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, Massachusetts 02139, U.S.A.

I. Introduction

Responding quickly and efficiently to dynamic disturbances is a crucial challenge in domains such as manufacturing,[10] aerial and underwater vehicle tasking,[6,8,16,18] and health care.[4,14,17,20] In many cases, accurately capturing the complicated dependencies between tasks in these environments requires the use of upper and lowerbound temporal constraints (i.e, deadlines and wait constraints). However, optimally scheduling tasks related by upper and lowerbound temporal constraints is known to be NP-Hard.[3] While exact solution techniques exist to efficiently schedule resources, these techniques are computationally intractable for problems of interest with fifty or more tasks and five agents.[12,15] Furthermore, techniques that seek to improve scalability often attempt to distribute the scheduling problem amongst the agents, where each agent generates its own schedule.[5] However, when agents must share unary-access resources (e.g., a spatial location that can be occupied by only one agent at a time), these techniques lose their advantage because the problems do not naturally lend themselves to decomposition. As a result, many techniques work by first finding an initial, though possibly infeasible, schedule through solving a relaxed version of the problem, and then repairing the schedule to resolve any constraint violations.

Even with the advancement of robotic technology, human operators continue to play a critical role in both the supervision of the scheduling process as well as in direct task execution. Approximate solution techniques exist for scheduling against upper and lowerbound temporal constraints;[7,13] however, these approaches construct a new schedule from scratch each time the algorithm is called. Assimilating a new schedule into an operators' mental model every time a disturbance occurs during runtime is a challenging prospect. Gombolay et al. propose a fast, near-optimal solution technique that seeks to minimize the difference in the allocation of tasks to agents between the previous schedule and the revised schedule in response to a dynamic disturbance, but this minimization is balanced with schedule optimality.[11] Researchers have proposed techniques that respond to runtime disturbances by incrementally updating the schedule as the disturbances arise.[1,2,9,19,21] Bartak *et al.*[1] consider resource constraints, but each task is assumed to be independent. Gallagher *et al.*[9] develop incremental scheduling heuristics for tasks where task durations are decision variables, and the reward for executing a task is dependent upon its duration and when it is executed. Zweben *et al.* consider the problem of iteratively repairing a schedule (i.e., reducing the cost of exiting constraint violations) where tasks are related through soft, upper and lowerbound temporal constraints as well as resource constraints.[21]

We propose a set of computational techniques for incremental scheduling of multiple agents to complete a set of non-preemptive tasks with hard, upper and lowerbound temporal and spatial constraints. Specifically, we consider the scenario where an initial schedule is given and an additional task must be inserted in the schedule. This new task may have temporal and spatial dependencies with the previously scheduled tasks. We model this scheduling problem using a Simple Temporal Network (STN), where nodes represent scheduling events (i.e., the start and finish times of tasks), and edges represent temporal constraints relating events (e.g., task durations, wait constraints, and deadline constraints). Our investigation differs from previous work in that we are attempting to incrementally schedule non-preemptive tasks with complex dependencies (i.e., hard upper and lowerbound temporal and resource constraints).[15] Work by Zweben *et al.* serves as a strong basis for comparison, but their approach is iterative in nature and works to change an initially infeasible schedule into a schedule that violates fewer constraints.[21] We directly consider currently feasible schedules and attempt to insert a new task in a way that least increases schedule duration. We organize our work as follows. In Section II, we formulate the problem of incrementally adding a new task to a schedule as a mixed-

integer mathematical program. In Section III, we describe incremental scheduling methods introduced in Gallagher *et al.*[9] as well as the iterative repair approach detailed in Zweben *et al.*[21] In Section IV, we briefly describe our prior work in constructing new schedules in response to dynamic disturbances. We present a set of heuristics to incrementally update a schedule in Section V. We validate the empirical benefits of these insertion heuristics in comparison to constructing a full re-sequencing as well as a full re-scheduling in Section VI. Lastly, in Section VII, we describe our ideas for future work in incremental scheduling.

II. Formal Problem Definition

We formulate the problem of taking a previously-scheduled task set $\boldsymbol{\tau'}$ and inserting a new task τ_i as mixed-integer linear program (MILP), as shown in Equations 1-11. This representation is equivalent to constructing a new schedule for task set $\boldsymbol{\tau}$ where $\boldsymbol{\tau} = \boldsymbol{\tau'} \cup \tau_{n+1}$. In this formulation, $A^a_{\tau_i} \in \{0,1\}$ is a binary decision variable for the assignment of agent a to task τ_i, $J_{\langle \tau_i, \tau_j \rangle} \in \{0,1\}$ is a binary decision variable specifying whether τ_i comes after or before τ_j, and $s_{\tau_i}, f_{\tau_i} \in [0,\infty)$ are the start and finish times of τ_i. Equation 1 is a general objective to minimize the makespan, with the decision variables $\{A^a_{\tau_i} | \tau_i \in \boldsymbol{\tau}, a \in A\}$, $\{J_{\langle \tau_i, \tau_j \rangle} | \tau_i, \tau_j \in \boldsymbol{\tau}\}$, and $\{s_{\tau_i}, f_{\tau_i} | \tau_i \in \boldsymbol{\tau}\}$. Equation 2 ensures that each task is assigned to a single agent. Equation 3 ensures that the duration of each $\tau_i \in \boldsymbol{\tau}$ does not exceed its upper and lowerbound durations. Equation 4 requires that the duration of task τ_i, $f_{\tau_i} - s_{\tau_i}$, is no less than the time required for agent a to complete task τ_i. Equation 5 requires that τ_j occurs at least $W_{\langle \tau_i, \tau_j \rangle}$ units of time after τ_i. Equation 6 requires that the duration between the start of τ_i and the finish of τ_j is less than $D^{rel}_{\langle \tau_i, \tau_j \rangle}$. Equation 7 requires that τ_i finishes before $D^{abs}_{\tau_i}$ units of time have expired since the start of the schedule. Equations 8-9 enforce that agents can only execute one task at a time. Equations 10-11 enforce that tasks sharing the same resource must be executed one at a time. We use Equations 8-11 to encode spatial constraints on inter-agent proximity while performing tasks. The worst-case time complexity of a complete solution technique for this problem is given by $O\left(2^{|A||\boldsymbol{\tau}|^3}\right)$, where $|A|$ is the number of agents and $|\boldsymbol{\tau}|$ is the number of tasks.

$$\min z, z = \max_{\tau_i, \tau_j \in \boldsymbol{\tau}} \left(f_{\tau_j} - s_{\tau_i}\right) \tag{1}$$

subject to

$$\sum_{a \in A} A^a_{\tau_i} = 1, \forall \tau_i \in \boldsymbol{\tau} \tag{2}$$

$$ub_{\tau_i} \geq f_{\tau_i} - s_{\tau_i} \geq lb_{\tau_i}, \forall \tau_i \in \boldsymbol{\tau} \tag{3}$$

$$f_{\tau_i} - s_{\tau_i} \geq lb^a_{\tau_i} - M\left(1 - A^a_{\tau_i}\right), \forall \tau_i \in \boldsymbol{\tau}, a \in A \tag{4}$$

$$s_{\tau_j} - f_{\tau_i} \geq W_{\langle \tau_i, \tau_j \rangle}, \forall \tau_i, \tau_j \in \boldsymbol{\tau} |, \forall W_{\langle \tau_i, \tau_j \rangle} \in \boldsymbol{TC} \tag{5}$$

$$f_{\tau_j} - s_{\tau_i} \leq D^{rel}_{\langle \tau_i, \tau_j \rangle}, \forall \tau_i, \tau_j \in \boldsymbol{\tau} | \exists D^{rel}_{\langle \tau_i, \tau_j \rangle} \in \boldsymbol{TC} \tag{6}$$

$$f_{\tau_i} \leq D^{abs}_{\tau_i}, \forall \tau_i \in \boldsymbol{\tau} | \exists D^{abs}_{\tau_i} \in \boldsymbol{TC} \tag{7}$$

$$s_{\tau_j} - f_{\tau_i} \geq M\left(A^a_{\tau_i} + A^a_{\tau_j} - 2\right) + M\left(J_{\langle \tau_i, \tau_j \rangle} - 1\right), \forall \tau_i, \tau_j \in \boldsymbol{\tau}, \forall a \in A \tag{8}$$

$$s_{\tau_i} - f_{\tau_j} \geq M\left(A^a_{\tau_i} + A^a_{\tau_j} - 2\right) - M\left(J_{\langle \tau_i, \tau_j \rangle}\right), \forall \tau_i, \tau_j \in \boldsymbol{\tau}, \forall a \in A \tag{9}$$

$$s_{\tau_j} - f_{\tau_i} \geq M\left(J_{\langle \tau_i, \tau_j \rangle} - 1\right), \forall \tau_i, \tau_j \in \boldsymbol{\tau} | R_{\tau_i} = R_{\tau_j} \tag{10}$$

$$s_{\tau_i} - f_{\tau_j} \geq -M\left(J_{\langle \tau_i, \tau_j \rangle}\right), \forall \tau_i, \tau_j \in \boldsymbol{\tau} | R_{\tau_i} = R_{\tau_j} \tag{11}$$

III. Background

Previous works have addressed variations of incremental scheduling with upper and lower bound temporal constraints. Gallagher *et al.*[9] presents a suite of incremental scheduling heuristics. There are, however, important differences between the scheduling problem presented in this paper versus the problem formulation in the Gallagher paper. Gallagher *et al.* selects a subset of the entire task set to schedule. If executing a task lowers the overall utility of the schedule, then that task is omitted from the schedule. Furthermore, the reward for a task is a function of its duration, and the duration is a decision variable. Thus, task durations can be altered to accommodate other tasks. In this paper, all tasks in the problem input have a hard lower and upper bound time duration and must to be executed. Although there exists fundamental differences in the scheduling problems, we have adapted techniques by Gallagher *et al.* to satisfy the constraints of this

paper's scheduling problem. In particular, we have applied the *best-slot* insertion option heuristic technique in Gallagher based upon its promising performance reported in their experimental analysis.[9] We implemented this technique as a baseline for comparison. This technique analyzes all possible time points in which a new task could be inserted in the current schedule, and commits the new task to be inserted at the time point that results in the best schedule. This technique is computationally expensive since it computes a schedule for all possible time points where the new task could be inserted.

Another technique that we explore is adapted from Zweben *et al.*,[21] which presents a *constraint-based iterative repair* technique. The scheduling problem again differs slightly in that the problem input is a complete schedule that possibly violates constraints. The constraint-based iterative repair technique uses simulated annealing to iterate through constraint violations and perturb the task sequence to improve overall schedule quality. The scheduling problem in this paper, on the other hand, takes as input a complete schedule that satisfies all temporal and spatial constraints and a new task that needs to be inserted into this schedule. Since these problems are similar in essence, we use similar techniques to design a heuristic as a second baseline that inserts a new task into a schedule in such a way that could possibly violate temporal and spatial constraints and then iteratively repairs the schedule until all constraints are satisfied. Our adaptation of the Zweben *et al.* approach differs from the original because we require that the outputted schedule contains no constraint violations. Furthermore, we remove the simulated annealing portion of the iterative repair to improve computational performance. In practice, we find that schedule quality is not significantly improved for this class of problems without great computational effort.

IV. Approximate Solution Technique

Our heuristic techniques presented in this paper require a feasible schedule as input. Using linear programming techniques to compute a preliminary schedule quickly becomes computationally impracticable for the problem size of interest. Consequently, we use a fast, near-optimal solution technique developed in prior work to schedule multi-agent teams to complete a set of tasks with hard, temporospatial constraints, which we call Tercio.[11] Tercio takes as input a temporal constraint problem, a list of agent capabilities (i.e., the lowerbound, upperbound, and expected duration for each agent performing each task), and the physical location of each task. Tercio first solves for an optimal task allocation by ensuring that the maximum amount of work assigned to any agent is as small as possible, as depicted in Equation 12. In this equation, $\boldsymbol{A}$ is the set of agents, $A^a_{\tau_i}$ is a task allocation variable that equals 1 when agent a is assigned to task τ_i and 0 otherwise, $\{A^a_{\tau_i} | \tau_i \in \boldsymbol{\tau}, a \in \boldsymbol{A}\}$ is the set of task allocation variables, $\boldsymbol{A^*}$ is the optimal task allocation, and $C^a_{\tau_i}$ is the expected time it will take agent a to complete task τ_i.

$$\boldsymbol{A^*} = \underset{\{A^a_{\tau_i} | \tau_i \in \boldsymbol{\tau}, a \in \boldsymbol{A}\}}{\arg\min} \; \max_{\boldsymbol{a} \in \boldsymbol{A}} \sum_{\tau_i \in \boldsymbol{\tau}} C^a_{\tau_i} \times A^a_{\tau_i} \tag{12}$$

After solving for a task allocation, $\boldsymbol{A^*}$, Tercio uses a fast sequencing subroutine, which we call the *sequencer*, to complete the schedule. The sequencer orders the tasks through simulation over time. Before each commitment is made, the sequencer conducts an analytical schedulability test to determine whether task τ_i can be scheduled at time t given prior scheduling commitments. If the schedulability test returns a determination that the commitment can be made, the sequencer then orders τ_i and continues. If the schedulability test cannot guarantee commitment, the sequencer evaluates the next available task. If the schedule, consisting of a task allocation and a sequence of tasks, does not satisfy a specified makespan, a second iteration is performed by finding the second-most optimal task allocation and the corresponding sequence. The process terminates when the user is satisfied with the schedule quality or after a predefined number of iterations. The computational complexity of Tercio is $O(2^{|A||\tau|})$. The complexity of the task allocation subroutine is $O(2^{|A||\tau|})$, and the complexity of the sequencing subroutine is $O(|\tau|^3)$.

V. Insertion Heuristics

In this section, we present four heuristics for incremental scheduling. Each heuristic algorithm takes as input a complete schedule with n tasks and a set of new tasks with temporal and spatial constraints that relate it to the original schedule. The algorithm outputs a new schedule that satisfies all temporal and spatial constraints, including those enforced by the new task.

Heuristic A - Greedy Insertion - The first heuristic involves splitting the set of tasks $\boldsymbol{\tau}$ from the original schedule into three subsets $\boldsymbol{\tau_a}$, $\boldsymbol{\tau_d}$, and $\boldsymbol{\tau_m}$ according to the wait constraints of the new task τ_{n+1}. $\boldsymbol{\tau_a}$ and $\boldsymbol{\tau_d}$ are the sets of all the ancestors and descendants of $\boldsymbol{\tau_{n+1}}$ in the new schedule, respectively. $\boldsymbol{\tau_m}$ is the set of tasks that are not ancestors or descendants of τ_{n+1}. We say that a task τ_i is a descendant of a second task, τ_j, if there exists a directed path from τ_i to τ_j. For example, consider three tasks, τ_i, τ_j, and τ_k. If there exist wait constraints $W_{\langle \tau_i, \tau_k \rangle}$ and $W_{\langle \tau_k, \tau_j \rangle}$, then τ_i and τ_k are ancestors of τ_j. Similarly, τ_k and τ_j are descendants of τ_i. Heuristic A schedules the tasks in $\boldsymbol{\tau_a}$ and according to their original order. Next, Heuristic A inserts task τ_{n+1} as early as possible in the sequence to satisfy its precedence relationships with tasks in $\boldsymbol{\tau_a}$. Next, tasks in $\boldsymbol{\tau_m}$ and $\boldsymbol{\tau_d}$ are scheduled to maintain their sequence relative to the original schedule. Lastly, due to changes made by the additional wait constraints of τ_{n+1}, there may be gaps in the schedule in which no task is currently being executed, which would unnecessarily extend the schedule. To remedy these gaps, we iterate a final time through the tasks to determine whether any task's start time can be moved up to reduce idle time. The computational complexity of Heuristic A is $O(|\boldsymbol{\tau_a}| + |\boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}|^2)$.

Heuristic B - Sorted Insertion - The second heuristic is a slight modification of Heuristic A. Heuristic B makes the same assumption that the order in which the tasks in $\boldsymbol{\tau_a}$ are performed is not affected by the addition of τ_{n+1} into the schedule. After scheduling the tasks in $\boldsymbol{\tau_a}$, the heuristic then takes all tasks $\tau_i \in \boldsymbol{\tau_m} \cup \tau_{n+1}$ and creates a directed acyclic graph and a topological ordering of the tasks as follows:

1. Let G be a directed graph initially with $|\boldsymbol{\tau_m} \cup \tau_{n+1}|$ nodes.
2. Let u_i be the node that corresponds to task τ_i.
3. For all wait constraints $W_{\langle \tau_i, \tau_j \rangle}$ such that $\tau_i, \tau_j \in \boldsymbol{\tau_m} \cup \tau_{n+1}$ add a directed edge from u_i to u_j.
4. If agent a is assigned to τ_i and τ_j in $\boldsymbol{\tau_m} \cup \tau_{n+1}$ and $lb^a_{\tau_i} < lb^a_{\tau_j}$, then add an edge from u_i to u_j.
5. If tasks τ_i and τ_j in $\boldsymbol{\tau_m} \cup \tau_{n+1}$ share the same spatial location, and $lb^a_{\tau_i} < lb^a_{\tau_j}$, then add an edge from u_i to u_j.
6. Let T be a topological ordering of G.

After we obtain the topological ordering T to get a proper order in which to schedule the tasks in $\boldsymbol{\tau_m} \cup \tau_{n+1}$. The last subset of tasks $\boldsymbol{\tau_d}$ are then scheduled in the same order as in the original schedule. Lastly, a final iteration of the tasks is made to eliminate any excess idling in the schedule. The computational complexity of Heuristic B is $O(|\boldsymbol{\tau_a}| + |\boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}|^2)$.

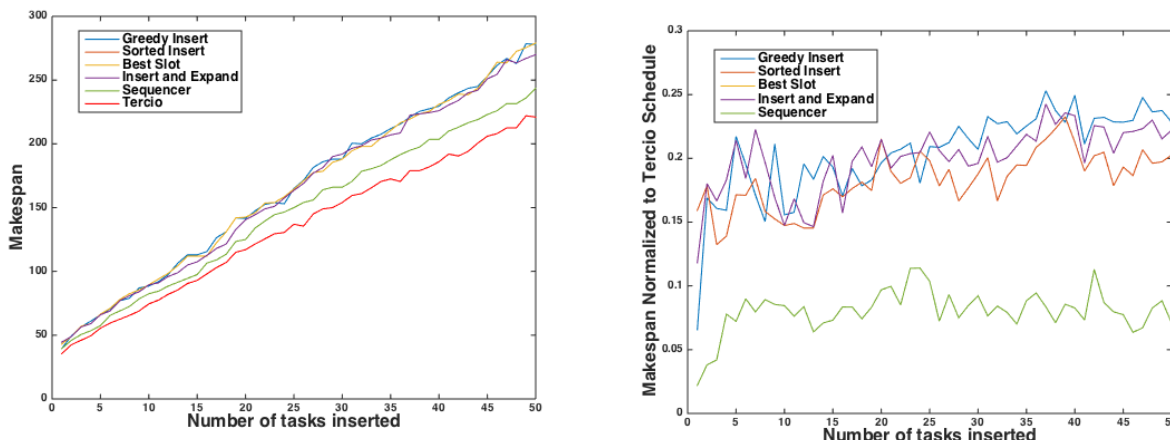
Heuristic C - Best Slot Insertion - The third heuristic follows a *best-slot* insertion option as presented in the Gallagher *et. al.*[9] The best-slot insertion option searches all possible time points in the schedule at which a new task could be inserted. It will then insert the new task at the location that yields the greatest benefit as defined by the problem's objective function. Note that, unlike that in Gallagher *et al.*, our scheduling problem has temporal constraints between tasks. As a result, inserting a new task at a certain time point may have significant effects to the schedule. In our adaptation of best-slot insertion heuristic, we instead search through a set of permutations such that the new task is inserted between any two tasks in the current schedule. It begins by preserving the original schedule for the tasks in $\boldsymbol{\tau_a}$. It then proceeds by finding the best slot in the original schedule to insert the new task τ_{n+1} as follows. Let T be the permutation that defines the order in which that tasks in $\boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}$ were executed in the original schedule. Now let T_i for $0 \leq i \leq |\boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}|$ be the extended order such that task τ_{n+1} is inserted before the ith index of T. For example, if $T = (\tau_i, \tau_j, \tau_k)$ and we want to insert τ_l, then $T_0 = (\tau_l, \tau_i, \tau_j, \tau_k)$, $T_1 = (\tau_i, \tau_l, \tau_j, \tau_k)$, etc. Heuristic C calculates a schedule for each one of the permutations T_i and returns the best schedule found. The computation complexity of Heuristic C is $O(|\boldsymbol{\tau_a}| + |\boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}|^3)$.

Heuristic D - Insert and Expand - The fourth heuristic follows an insert-and-expand technique similar in spirit to heuristics proposed in Zweben[21] for independent tasks. The insertion step involves inserting τ_{n+1} into the original schedule subject to the precedence constraints set by tasks in $\boldsymbol{\tau_a}$. This insertion may result in temporal and spatial constraint violation with tasks in $\boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}$. Note that this is different from Zweben *et al.*[21] because the schedule that we want to repair may have temporal constraint violations. Furthermore, our heuristic searches for a schedule that contains no constraint violations. The expansion step iterates through

each task $\tau_i \in \boldsymbol{\tau_m} \cup \boldsymbol{\tau_d}$ until all temporal and spatial constraints are satisfied. This expansion step is similar in principle to the iterative repair technique presented in Zweben.[21] A single iteration consists of checking if task τ_i violates any constraints. If τ_i violates such a constraint, then the time at which τ_i is currently scheduled, s_{τ_i}, is pushed later into the schedule so as to satisfy its temporal and spatial constraints. Then, we iterate through all of τ_i's children (i.e., $\{\tau_j | \exists W_{\langle \tau_i, \tau_j \rangle}\}$) and adjust the schedule such that they satisfy the precedence set by τ_i as well as the proper spatial constraints. The iteration terminates when all of the constraints are satisfied. As in the first heuristic, there may be gaps in the new schedule, so we perform a final iteration of the tasks to reduce idle time from the resulting schedule. The computational complexity of Heuristic B is $O(|\boldsymbol{\tau}|^2)$.

VI. Empirical Analysis

In this section, we validate the performance of our incremental scheduling heuristics on a synthetic data set, as shown in Figures 1, 2, 3. Results were generated in MATLAB using a Macbook Pro with a 2.3 GHz Intel Core i7 and 8 GB 1600 MHz DDR3 RAM. To construct our data set, we randomly generate fifty scheduling problems in the form presented in Section II. We use Tercio[11] to perform task allocation and sequencing each task set. Tercio's task allocation subroutine was implemented in Java using Gurobi; the sequencing subroutine was implemented in MATLAB. For each scheduling problem, we randomly generate fifty additional new tasks with a given agent allocation.[a] The duration of the task is randomly selected from the uniform distribution of the integers from 1 to 10, inclusive, that must be added to the schedule. These new tasks have randomly-generated temporospatial constraints relating it to tasks in the current schedule. We apply each incremental scheduling heuristic to incorporate the new tasks into the schedule. We also perform a re-sequencing of all tasks using Tercio's sequencing subroutine (i.e., the task allocation remains the same). Full re-sequencing is slower than an incremental schedule change, but re-sequencing often allows for improved schedule quality. We also consider a full re-scheduling using Tercio's task allocation and sequencing components. We allow Tercio to run for five iterations. In this analysis, we benchmark against Tercio, which has been shown to be fast and empirically near-optimal. Our goal is to schedule large, real-world task sets for which solving the MILP in Section II is computationally intractable.



(a) This figure shows the median schedule quality of the four insertion heuristics, the sequencer, and Tercio as more tasks are added to the schedule.

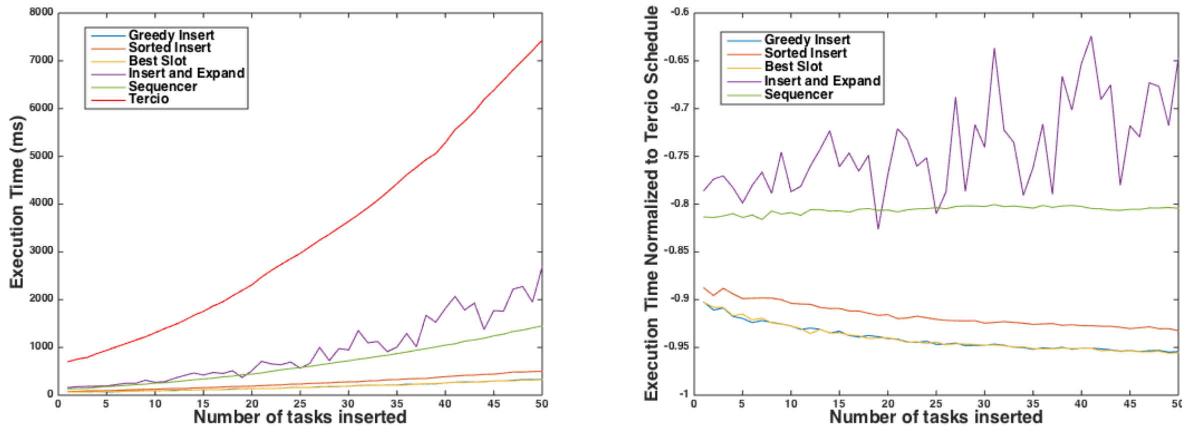
(b) This figure shows the schedule quality of the four insertion heuristics and the sequencer normalized to the Tercio.

Figure 1: Figures 1a and 2a show the empirical performance of the four incremental scheduling heuristics.

Figure 1a shows the makespan of the incremental scheduling heuristics when adding 50 new tasks to a schedule originally composed of 10 tasks. The data is presented as a line plot of the median makespan values

[a]We note that we could allow each heuristic to perform its own agent allocation by applying the heuristic once for each possible agent assignment, which would increase computation time linearly with the number of agents.

after inserting a number of tasks to the original schedule. The vertical axis represents the median makespan computed by each heuristic. The horizontal axis represents the number of tasks that have been inserted. Figure 1b shows the same data normalized to the makespan computed by Tercio. In this figure, the vertical axis represents the percentile difference between the makespan generated by each heuristic and the makespan from Tercio constructing a new schedule. In Figure 1a, we see that the four heuristics performed similarly, with the Insert-and-Expand method performing the best out of all of the heuristics. However, it is evident that Tercio's sequencing subroutine performed about 10% better on average than all of the heuristics.

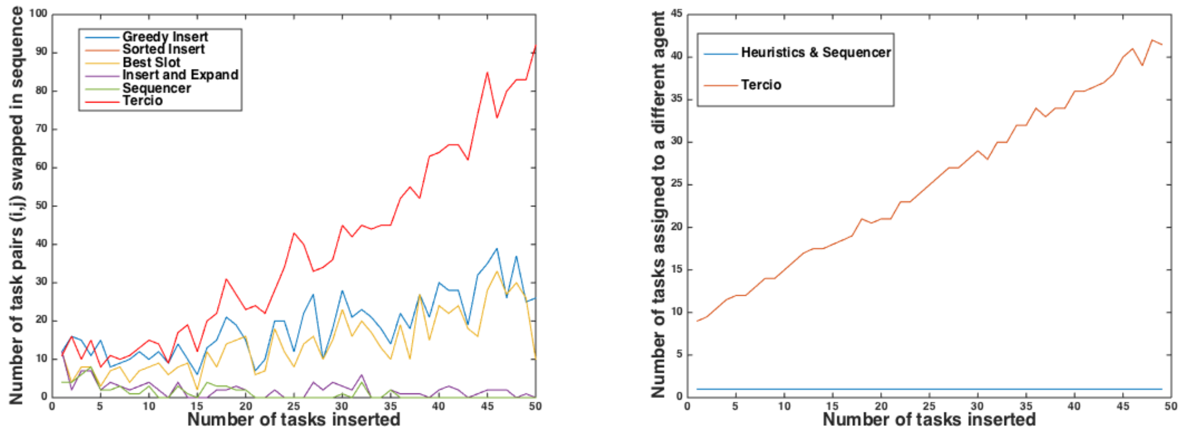


(a) This figure shows the median computation time of the four insertion heuristics, the sequencer, and Tercio as more tasks are added to the schedule.

(b) This figure shows the median computation time of the four insertion heuristics and the sequencer normalized to Tercio's computation time.

Figure 2: Figures 1b and 2b show the empirical performance of the four incremental scheduling heuristics normalized by Tercio's performance.

Figure 2b shows the computation time required to evaluate the four incremental scheduling heuristics as well as re-sequencing using Tercio's subroutine. In this figure, we can see the added benefit of incremental scheduling in regards to reduced computation time. With the exception of the best-slot heuristic, the heuristics performed significantly faster than Tercio re-sequencing and re-scheduling. The greedy and sorted insertion heuristic performed $91\% \pm 1\%$ faster than Tercio re-scheduling and $11\% \pm 2\%$ faster than Tercio re-sequencing. The insert-and-expand heuristic performed comparatively well by being 2.5% slower than the greedy insert. The best-slot heuristic performed comparatively poorly and inconsistently. As shown in Figure 2b, the best-slot heuristic was slower than the Tercio re-sequencing. Furthermore, the computation time was very inconsistent. We suspect the reason for this variation is that the computational complexity of this heuristic is highly contingent on the dependencies of the new task into the schedule. For example, the best-slot heuristic will perform slower on a new task with many ancestors than on a new task with very little ancestors.



(a) This figure shows how much the schedule changes by inserting a new task.

(b) This figure shows how much the agent allocation changes by inserting a new task.

Figure 3: Figures 3a and 3b show the stability of the four incremental scheduling heuristics.

In Figure 3, we analyze the stability of the heuristic methods used. This metric is important because in dynamic systems where humans and robots are involved in executing the schedule, we would prefer dynamic changes to the schedule to vary little to decrease the cost of context switching. Otherwise, human workers may be unnecessarily burdened if they are repetitively assigned a new job. We measure stability by calculating the overall change in the schedule in regards to the sequence of tasks being executed and the change in agent allocation. Figure 3a shows how much the sequence of tasks changes as a new task in inserted into the schedule. The vertical axis represents the number of task pairs (τ_i, τ_j) such that $s_{\tau_i} \leq s_{\tau_j}$ in the previously computed schedule but $s_{\tau_i} > s_{\tau_j}$ after a new task has been inserted. The horizontal axis represents the number of tasks that have been inserted into the original schedule. It is evident in Figure 3a that Tercio re-scheduling drastically affects the sequence of the schedule. On the other hand, the insert-and-expand heuristic, as well as Tercio re-sequencing, generates very little changes to the sequence as new tasks are added. Figure 3b shows the change in agent allocation as new tasks are added. The vertical axis represents the number of tasks that were assigned a different agent from the previously computed schedule. Since the heuristics and the re-sequencing do not perform its own agent allocation, they are represented by a flat line at the bottom of this figure. However, we can see that Tercio re-scheduling consistently changes the agent allocation.

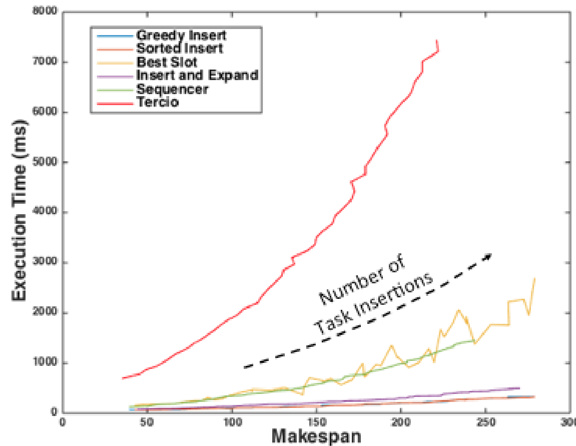


Figure 4: This figure shows the the computation time increases with makespan for each insertion method.

Figure 4 shows the computation time of each insertion method with respect to the calculated makespan of the schedule. In this figure we can see the tradeoff of running the different methods as an insertion technique. On one extreme, Tercio re-scheduling calculates a schedule with a relatively small makespan; however, the computation time is much longer than that of any of the other heuristics. On the other hand, heuristics A,B, and D perform much faster than any other method but they generate schedules with the largest makespan.

The trade-off between incremental scheduling and constructing a new schedule is an interesting area of investigation.[9] From our results, we see that the only heuristic that performed comparatively well against Tercio re-sequencing is the insert-and-expand method. The rest of the heuristics, although they perform significantly faster, computed a schedule with relatively poor schedule quality and did not exhibit a desired amount of stability. As shown in Figure 1 the insert-and-expand did compute a schedule that was within 15% of using the Tercio sequencing subroutine on average. However, we see in Figure 2 that the insert-and-expand performed significantly faster than Tercio. Furthermore, it exhibited comparatively similar stability as shown in Figure 3. We were surprised by the performance of Tercio re-sequencing in terms of stability. We expected the re-sequencing method to be significantly less stable than some of the heuristics presented; however, this was not the case. As a result, the trade-off between using Tercio re-scheduling, re-sequencing, and insertion heuristics becomes less clear. If computation time is an important factor to consider, then using an insertion heuristic may be ideal. However, since an insertion heuristic normally outputs schedules of relatively less quality, it could be advantageous to re-sequence or re-schedule after a certain number of tasks have been inserted using an insertion heuristic. If both computation time and schedule quality are important factors, then using Tercio re-sequencing as a task insertion method may be best. Since this method displayed surprisingly impressive stability, we have confidence that this would perform considerably well as an insertion method.

VII. Future Work

In this paper, we have investigated several approaches to incremental scheduling. We have adapted prior work by Zweben *et al.*[21] and Gallagher *et al.*[9] to be compatible with our scheduling problem. Our results showed that Tercio re-sequencing performed considerably better than any of the four insertion heuristics presented in this paper. In the future, we plan to investigate smarter techniques to apply to insertion heuristics such as applying machine learning to define features of schedules and how we can best insert a new task into a schedule with a certain feature set.

Acknowledgments

This work was supported by the National Science Foundation (NSF) Graduate Research Fellowship Program (GRFP) under grant number 2388357.

References

[1] R. Barták, T. Mller, and H. Rudov. A new approach to modeling and solving minimal perturbation problems. In *Recent Advances in Constraints: Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 20*, pages 233–249. Springer, 2004.

[2] M. A. Becker and S. F. Smith. Mixed-initiative resource management: The amc barrel allocator. In *Proceedings of the 5th International Conference on AI Planning and Scheduling*, pages 32–41. The AAAI Press, 2000.

[3] D. Bertsimas and R. Weismantel. *Optimization over Integers*. Dynamic Ideas, Belmont, 2005.

[4] L. Brandenburg, P. Gabow, G. Steele, J. Toussaint, and B. J. Tyson. Innovation and best practices in health care scheduling. Technical report, February 2015.

[5] L. Brunet, H.-L. Choi, and J. P. How. Consensus-based auction approaches for decentralized task assignment. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference (GNC)*, Honolulu, HI, 2008.

[6] J. Cates. Route optimization under uncertainty for unmanned underwater vehicles. Master's thesis, 2011.

[7] M. B. Dias. *TraderBots: A New Paradigm for Robust and Efficient Multirobot Coordination in Dynamic Environments*. PhD thesis, Robotics Institute, Carnegie Mellon University, January 2004.

[8] E. Frost. Robust planning for unmanned underwater vehicles. Master's thesis, 2013.

[9] A. Gallagher, T. L. Zimmerman, and S. F. Smith. Incremental scheduling to maximize quality in a dynamic environment. In *Proc. 16th International Conference on Automated Planning and Scheduling (ICAPS*, pages 222–231. AAAI Press, 2006.

[10] M. Garey, D. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, May 1976.

[11] M. C. Gombolay, R. J. Wilcox, and J. A. Shah. Fast scheduling of multi-robot teams with temporospatial constrints. In *Proceedings of the Robots: Science and Systems (RSS)*, Berlin, Germany, June 24-28, 2013.

[12] J. N. Hooker. A hybrid method for planning and scheduling. Technical report, Pepper School of Business, Carnegie Mellon University, 2004.

[13] E. Jones, M. Dias, and A. Stentz. Time-extended multi-robot coordination for domains with intra-path constraints. *Autonomous Robots*, 30(1):41–56, 2011.

[14] S. M. Kehle, N. Greer, I. Rutks, and T. Wilt. Interventions to improve veterans access to care: A systematic review of the literature. *Journal of General Internal Medicine*, 26(2):689–696, 2011.

[15] G. A. Korsah, A. Stentz, and M. B. Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, 2013.

[16] U. O. L. Xu. Battle management for unmanned aerial vehicles. In *Proceedings of the IEEE Conference on Decision Control*, December 2003.

[17] S. D. Pizer and J. C. Prentice. What are the consequences of waiting for health care in the veteran population? *Journal of General Internal Medicine*, 26(2):676–682, 2011.

[18] M. Rekik, J.-F. Cordeau, and F. Soumis. Consensus-based decentralized auctions for robust task allocation. *IEEE Transactions on Robotics*, 25:912–926, 2004.

[19] H. E. Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, Oct. 2000.

[20] S. A. Shipman and C. A. Sinsky. Expanding primary care capacity by reducing waste and improving efficiency of care. *Health Affairs (Millwood)*, 32(11):1990–1997, 2013.

[21] M. Zweben, E. Davis, B. Daun, and M. Deale. Scheduling and rescheduling with iterative repair. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(6):1588–1596, Nov 1993.