Metareasoning for Interleaved Planning and Execution

Amihay Elboher[1], **Shahaf S. Shperberg**[1], **Solomon E. Shimony**[1], **Wheeler Ruml**[2]
[1]Ben-Gurion University, Israel
[2]University of New Hampshire, USA
{amihaye,shperbsh}@post.bgu.ac.il, shimony@cs.bgu.ac.il, ruml@cs.unh.edu,

Abstract

Agents that plan and act in the real world must deal with the fact that time passes as they are planning. When timing is very tight, there may be insufficient time to complete the search for a plan before it is time to act. By executing actions before search concludes, one gains time to search by making planning and execution concurrent. However, this incurs the risk of making incorrect action choices, especially if actions are irreversible. This tradeoff between opportunity and risk is the metareasoning problem addressed in this paper.

We begin by formally defining this as an abstract metareasoning problem, and setting it up as an MDP. This abstract problem is itself intractable. We show special cases that are solvable in polynomial time, present heuristic solution algorithms, and examine their effectiveness on instances generated according to distributions that represent typical planning problems.

1 Introduction

Agents that plan and act in the real world must deal with the fact that time passes as they are planning. For example, an agent that needs to get to the airport may have two options: take a taxi, or ride a commuter train. Each of these options can be thought of as a *partial plan* to be elaborated into a complete plan before execution can start. Clearly, the agent's planner should only elaborate the partial plan that involves the train if that can be done before the train leaves. In another example, suppose the planner has two partial plans that are each estimated to require five minutes of computation to elaborate into complete plans. If only six minutes remain until they both expire, then we would want the planner to allocate almost all of its remaining planning effort to one of them, rather than to fail on both. An abstract model for handling these issues [called *Allocating Effort when Actions Expire* (AE)^2] was proposed in (Shperberg et al. 2019), and is the basis for the research presented in this paper.

Now, suppose further that the estimated time to complete each plan is seven minutes, that the planner has already determined that the first action in the commuter train plan is to ride the elevator down to the first floor, which takes two minutes, and that the first action in taking a taxi is to call for a taxi (two minutes) which cannot be done while riding the elevator, which has no cellphone reception. The only apparent way to plan successfully thus involves starting to act before planning is complete. However, doing so may invalidate potentially valid plans. This paper proposes a disciplined method for handling such tradeoffs.

The idea of starting to perform actions in the real world (also known as base-level actions) before completing the search goes back at least as far as Korf's (Korf 1990) real-time A* (RTA*). The difference from the RTA* paradigm is that our scenario is more flexible; the agent does not have a predefined time at which actions must be executed. Rather, it must reason about when base-level actions should be executed in order to maximize the probability of successful and timely execution. Note that we assume here that the world is deterministic, the only uncertainty we model here is at the meta-level, due to uncertainty about how long planning/search will take and about the time it will take the (unknown at search time) resulting plan to reach a goal state.

In this work we define the above tradeoffs as a formal problem of decision-making under uncertainty, in the sense defined by (Russell and Wefald 1991). Attempting to do so for an actual planning or search algorithm is far too complicated, even under our assumption of a deterministic real world. We thus adopt the aforementioned (AE)^2 scheme, which defined an abstract metareasoning problem of allocating processing time among n search processes, and extend it to allow execution of actions in the real world (which, following (Cashmore et al. 2018), we call base-level actions), in parallel with the search processes.

The formal problem presented in this paper, called interleaving planning and execution when actions expire (IPAE for short), assumes that each process has already computed a known (possibly empty) prefix action sequence, the initial actions in the solution, and that there is an as-yet-unknown remainder of the action sequence to be executed. A distribution over the length of the remainder is given. The metareasoning problem we define is as in (AE)^2, to find a policy that maximizes the probability of a timely action sequence. However, unlike (AE)^2, in our extension the agent can actually start executing base-level actions (from one or more of the action sequence prefixes) in parallel with continuing the computation.

We then show that IPAE is a generalization of (AE)^2, and is thus also intractable, even under the very limiting assumption of known deadlines and remainders. Still, we cast IPAE as an MDP, so that we can define and analyze optimal policies, and even solve IPAE optimally for very small instances using standard MDP techniques like value iteration.

We describe several efficient ways of solving IPAE, although not necessarily optimally, and evaluate them empirically. In this paper, we examine only the one-shot version of the metareasoning problem. Integrating this into a temporal planner or search algorithm can involve solving this problem repeatedly, possibly after each node expansion, in addition to gathering the requisite statistics. These issues are ongoing work and beyond the scope of the current paper. When testing our algorithms, we use scenarios based on search trees generated by running A* on sliding tile puzzle instances.

2 Background: Metareasoning in Situated Planning and Search

In situated temporal planning, each action a has a latest start time t_a and a a plan must be fully generated before its first action can begin executing. This induces a deadline (although this deadline may be unknown, since the actions in the plan are not known until the search terminates).

For a partial plan available at a search node i in the planner, this can be modeled by a random variable d_i, denoting the unknown deadline by which any potential plan expanded from node i must be generated. Thus, the planner faces the metareasoning problem of deciding which nodes on the open list to expand in order to maximize the chance of finding a plan before its deadline.

Shperberg et al. (2019) proposed a model of this problem called (AE)^2 ('allocating effort when actions expire') which abstracts away from the planning problem and merely assumes n independent processes. Each process attempts to solve the same problem under time constraints. In the context of situated temporal planning using heuristic search, each process may represent a promising partial plan for the goal, implemented as a node on the open list eager to have its subtree explored. But the abstract problem may also be applicable to other settings, such as algorithm portfolios or scheduling candidates for job interviews. For simplicity, we assume a single processor, so the core of the metareasoning problem is to determine how to schedule the n processes on the single processor.

When process i terminates, it delivers a solution with probability P_i or, otherwise, indicates its failure to find one. An mentioned above, each process has an uncertain deadline defined over absolute wall clock time by which its computation must be completed in order for any solution it finds to be valid. For process i, let $D_i(t)$ be the CDF over wall clock times of the random variable denoting the deadline. The actual deadline for a process is only discovered with certainty when the process completes. This models the fact that a dependence on an external timed event might not become clear until the final action in a plan is added. If a process terminates with a solution before its deadline, we say that it is *timely*. Given $D_i(t)$, we assume w.l.o.g. that P_i is 1, otherwise one can adjust $D_i(t)$ to make the probability of a deadline that is in the past (thus forcing the plan to fail) equal to $1 - P_i$.

The processes have known search time distributions, i.e. performance profiles (Zilberstein and Russell 1996) described by CDFs $M_i(t)$, the probability that process i needs total computation time t or less to terminate. Although some of the algorithms we present can handle dependencies, we make the typical metareasoning assumption in our analysis that all the random variables are independent. Given the $D_i(t)$ and $M_i(t)$ distributions, the objective of (AE)^2 is to schedule processing time between the n processes such that the probability of at least one process finding a timely solution is maximized.

A simplified discrete-time version of the problem, called S(AE)^2, can be cast as a Markov decision process. The MDP's actions are to assign (schedule) the next time unit to process i, denoted by c_i with $i \in [1, n]$. Action c_i is allowed only if process i has not already failed. A process is considered to have failed if it has terminated and discovered that its deadline has already passed, or if the current time is later than the last possible deadline for the process.

The state variables are the wall clock time T and one state variable T_i for each process, with domain $\mathbb{N} \cup \{F\}$, although in practice the time domains of T, T_i are bounded by the latest possible deadlines. T_i denotes the cumulative time assigned to each process i until the current state, or that the process failed (indicated by F). We also have special terminal states SUCCESS and FAIL. Thus the state space is:

$$\mathcal{S} = (dom(T) \times \bigtimes_{1 \leq i \leq n} dom(T_i)) \cup \{\text{SUCCESS, FAIL}\}$$

The initial state has $T = 0$, and $T_i = 0$ for all $1 \leq i \leq n$. The transition distribution is determined by which process i has last been scheduled (the action c_i), the M_i distribution (which determines whether currently scheduled process i has completed its computation), and D_i (which determines the revealed deadline for a completed process, and thus whether it has succeeded or failed). If all processes fail, transition into FAIL (with probability 1). If some process is successful, transition into SUCCESS. The reward is 0 for all states except SUCCESS, for which the reward is 1.

The S(AE)^2 problem is NP-hard, even for known deadlines (denoted KDS(AE)^2) (Shperberg et al. 2019).

2.1 Greedy Schemes

As solving the metareasoning problem is NP-hard, Shperberg et al. (2019) used insights from a diminishing returns result to develop greedy schemes. Their analysis is restricted to linear contiguous allocation policies: schedules where the action taken at time t does not depend on the results of the previous actions, and where each process receives its allocated time contiguously.

Following their notation, we denote the probability that process i finds a timely plan when allocated t_i consecutive time units starting at time t_{d_i} as:

$$s_i(t_i, t_{d_i}) = \sum_{t'=0}^{t_i} (M_i(t') - M_i(t'-1))(1 - D_i(t' + t_{d_i})) \quad (1)$$

When considering linear contiguous policies, we need to allocate t_i, t_{d_i} pairs to all processes (with no allocation overlap). Note that overall a timely plan is found if at least one process succeeds, that is, overall failure occurs only if all processes fail. Therefore, in order to maximize the probability of overall success (over all possible linear contiguous allocations), we need to allocate t_i, t_{d_i} pairs so as to maximize the probability:

$$P_s = 1 - \prod_i (1 - s_i(t_i, t_{d_i})) \tag{2}$$

Using $LPF_i(\cdot)$ ('logarithm of probability of failure') as shorthand for $log(1 - s_i(\cdot))$, we note that P_s is maximized if the sum of the $LPF_i(t_i, t_{d_i})$ is minimized and that $-LPF_i(t_i, t_{d_i})$ behaves like a utility that we need to maximize. For known deadlines, we can assume that no policy will allocate processing time after the respective deadline. We will use $LPF_i(t)$ as shorthand for $LPF_i(t, 0)$.

To bypass the problem of non-diminishing returns, the notion of *most effective computation time* for process i under the assumption that it starts at time t_d and runs for t time units was defined as:

$$e_i(t_d) = \underset{t}{\text{argmin}} \frac{LPF_i(t, t_d)}{t} \tag{3}$$

The notion here is slightly generalized, as Shperberg et al. (2019) actually had e_i, which equals $e_i(0)$ here. We use e_i to denote $e_i(0)$ below.

Since not all processes can start at time 0, the intuition from the diminishing returns optimization is to prefer process i that has the best utility per time unit, i.e. such that $-LPF_i(e_i))/e_i$ is greatest. But allocating time to process i delays other processes, so it is also important to allocate the time now to processes that have an early deadline. Shperberg et al. (2019) therefore suggested the following greedy algorithm: Iteratively allocate t_u units of computation time to process i that maximizes:

$$Q(i) = \frac{\alpha}{E[D_i]} - \frac{LPF_i(e_i)}{e_i} \tag{4}$$

where α and t_u are positive empirically determined parameters, and $E[D_i]$ is the expectation of the random variable that has the CDF D_i (a slight abuse of notation). The α parameter trades off between preferring earlier expected deadlines (large α) and better performance slopes (small α).

The first part of Equation 4 is a somewhat ad-hoc measure of urgency, which additionally performs poorly if the deadline distribution has a high variance. A somewhat more advanced greedy scheme was defined by Shperberg et al. (2021) in an attempt to define the notion of urgency more precisely, and uses the notion of a damage caused to a process if its computation is delayed by some time t_u. This is based on the available utility gain after the delay of t_u.

An empirically determined constant multiplier γ was used to balance between exploiting the current process reward from allocating time to process i now and the loss in reward due to delay. Thus, the delay-damage aware (DDA) greedy scheme was to assign, at each processing allocation round, t_u time to the process i that maximizes:

$$Q'(i) = \frac{\gamma \cdot LPF_i(e_i(t_u), t_u)}{e_i(t_u)} - \frac{LPF_i(e_i, 0)}{e_i} \tag{5}$$

2.2 DP Solution for Known Deadlines

For KDS(AE)^2 (known deadlines S(AE)^2), it suffices to examine linear contiguous policies sorted by an increasing order of deadlines (Shperberg et al. 2019), formally:

Theorem 1. *Given a KDS(AE)^2 problem, there exists a linear contiguous schedule with processes sorted by a non-decreasing order of deadlines that is optimal.*

Theorem 1 was used in (Shperberg et al. 2021) to obtain a dynamic programming (DP) scheme.

Theorem 2. *For known deadlines, DP according to*

$$OPT(t, l) = \max_{0 \leq j \leq d_l - t} (OPT(t+j, l+1) - LPF_l(j)) \tag{6}$$

finds the optimal schedule in time polynomial in n, d_n.

For explicit M_i representations, evaluating Equation 6 in descending order of deadlines runs in polynomial time.

3 Interleaving Planning and Execution

In this paper, we extend the abstract $S(AE)^2$ model to account for execution of actions during search. We assume that each process has already constructed a sequence of actions, which will be the prefix of any complete plan below the node the process represents. For each process, there is a plan remainder that is still unknown. These assumptions make sense if we equate each such process with a node in the open list of a typical algorithm that searches from the initial state to the goal, and adds an action when a node is expanded. Here, the prefix is simply the list of operators leading to the current node. The rest of the action sequence is the remaining solution that may be developed in the future from each such node. However, here too we will abstract away from the actual search and model future search results by distributions.

Thus, in addition to distributions over completion times, for each process i we have a plan prefix H_i (H for head), containing a sequence of actions from a set of available "base-level" actions B. Each action $b \in B$ also has a deadline $D(b)$. Upon termination, a process i delivers the rest of the action sequence β_i of the solution in one chunk. As β_i is unknown prior to termination, we assume a known distribution R_i on $dur(\beta_i)$, the duration of β_i, and that the actual duration becomes known on termination.

Actions from any action sequence H_i may be executed (in sequence) even before having a complete plan. Execution changes the state of the system and we adjust the set of processes to reflect this: any process where the already executed action sequence is not a prefix of its partial plan becomes invalid. Executing any prefix of actions from any H_i with the first action starting no earlier than time 0 (representing the current time), and such that the next action in the sequence begins at or after the previous action terminates, and is executed before its deadline, is called a legal execution. Any suffix β_i is assumed to be composed of actions that cannot be executed before the process i terminates, thus the execution of β_i may only begin after process i terminates. We also assume that base-level actions are non-preemptible and cannot be run in parallel. However, computation may proceed freely while executing a base-level action.

As in $S(AE)^2$, we have a deadline for each process, but with a different semantics; unlike $S(AE)^2$, here the requirement is that the execution terminates before the (possibly unknown) deadline; a sequence of actions fully executed before its deadline is henceforth called a timely execution. We assume that there is a known distribution (of a random variable X_i) over $deadline_i$ the deadline for process i, and again that its true value becomes known only once the search in process i terminates. A typical application for such a setting is having to solve a physical puzzle while in a room with walls moving in upon the occupant, as in some famous movie scenes. In this case, the deadline is the same for all processes, and is known approximately in advance, that is, all the X_i are equal.

It is easy to see that an execution of a solution delivered by process i is timely just when the remainder β_i begins execution in time to conclude before its deadline; that is, just when $start(\beta_i) \leq deadline_i - dur(\beta_i)$. Since before computation completes these are random variables, then $start(\beta_i)$ is also constrained by a random variable, which we call the *induced* deadline for process i, and denote it by the random variable D_i. By construction, we have $D_i = X_i - R_i$, which is well defined whether or not the R_i and X_i are dependent.

Thus, we will simply assume that the induced deadline distribution D_i is given, and can ignore X_i and R_i henceforth. Note that the semantics of the induced deadline is that for a process i to be timely it must meet two conditions: 1) complete its computation, as well as 2) complete execution of all its action prefix H_i before the induced deadline D_i.

The Interleaving Planning and Execution while Actions Expire problem (IPAE), is thus defined as follows. We have a set of base-level actions B, each action $b \in B$ has duration $dur(b) > 0$. Given n processes, each with a (possibly empty) sequence H_i of actions from B, a performance profile M_i, and an induced deadline distribution D_i, find a policy for allocating computation time to the n processes and legally executing base-level actions from some H_i, such that the probability of executing a timely solution is maximal.

Example 1. *Extending the instance from the introduction where an agent needs to reach to the airport either by commuter train or by taxi. We have two processes: process 1 for the plan with the commuter train, and process 2 for the taxi plan. Suppose the unit of time is one minute, and we have to get to terminal D at the airport 30 minutes from now. The train (which leaves six minutes from now) takes 22 minutes, but the planner has not yet checked what to do at the end of the ride: the train may get to terminal D directly in which case no additional time is needed (say probability 0.8), or it may only stop at terminal A, requiring an additional five minutes to travel to terminal D (thus missing the deadline). Similar conditions may exist for the taxi plan, with the ride taking 20 minutes to get to the airport terminal D, but there also needs to be a payment step at the end, the length of which the planner has not yet determined (say one or ten minutes, each with probability 0.5). Representing this as IPAE, we have* $H_1 = [$*take elevator, ride train*$]$ *and* $H_2 = [$*phone, take elevator, take taxi*$]$, *with* $dur($*phone*$) = dur($*take elevator*$) = 2$, $dur($*ride train*$) = 22$, $dur($*take taxi*$) = 20$, *The remainder durations are distributed: for* β_1 *we have* $R_1 \sim [0.8 : 0\ ; 0.2 : 5]$, *and for* β_2 *we have* $R_2 \sim [0.5 : 1\ ;\ 0.5 : 10]$. *The deadlines are certain in this case,* $X_1 = X_2 = 30$ *with probability 1, and the induced deadlines are thus distributed:* $D_1 \sim [0.8 : 30\ ; 0.2 : 25]$ *and* $D_2 \sim [0.5 : 29\ ; 0.5 : 20]$. *Suppose remaining planner runtime for the train plan will take seven minutes with certainty, and for the taxi plan it is distributed:* $[0.5 : 1; 0.5 : 8]$. *The optimal policy here is to run process 2 for one minute. If it terminates and reveals that* $D_2 = 29$, *then call for a taxi and proceed (successfully) with the taxi plan. Otherwise (process 2 does not terminate, or terminates and reveals that* $D_2 = 20$*), start executing the actions from* H_1*: take the elevator and run process 1, then take the train and continue running process 1, hoping to find that* $D_1 = 30$. *This policy works with probability of success* $P_S = 0.25 + 0.75 * 0.8 = 0.85$.

We make the following simplifying assumptions:

1. Time is discrete, and the basic unit of time is 1 (as assumed in $S(AE)^2$).
2. The action durations $dur(b)$ are known for all $b \in B$.
3. The variables with distributions D_i, M_i are all mutually independent.
4. The individual action deadlines $D(b)$ are irrelevant (not used, or equivalently set to be infinite), as the overall process induced deadline distributions D_i are given.

Although assumption 4 is easy to relax, doing so complicates the analysis and is thus made to improve clarity. Our algorithm implementations actually do allow for individual action deadlines. Observe that any instance of $S(AE)^2$ can be made into an IPAE instance, by just setting all H_i to be null. Therefore, finding the optimal solution to IPAE is also NP-hard, even under assumptions 1-4. Thus, the initial analysis in the paper will also make the assumption that the induced deadlines are known, so as to try to get a pseudo-polynomial time algorithm for computing the optimal policy. Note that having a known deadline (e.g. we know that the room's walls will crush the agent in two minutes exactly) does not entail that the induced deadline is known, as typically $dur(\beta_i)$ will be unknown before the solution is known, and therefore the induced deadline will be unknown before termination. That is, it is possible that process i will find a solution, and only then discover that it cannot be completed on time, even for known deadlines.

4 Stating IPAE as an MDP

Under the additional assumptions 1 through 4 in Section 3, we state the IPAE optimization problem as the solution to the following MDP, similar to the one defined for $S(AE)^2$. The actions in the MDP are of two types: the base-level actions from B, and actions c_i: that allocate the next time unit of computation to process i. We assume that c_i can only be done if process i has not already terminated and has not become invalid by execution of base-level actions. An action b from B can only be done when no base-level action is currently executing and b is the next action in some H_i after the common prefix of base-level actions that all remaining processes share.

The **states** of the MDP are defined as the cross product of the following state variables:

1. Wall clock (real) time T,
2. Time T_i already assigned to each process i, for all i from 1 to n. These variables will also be used to encode process failure to find a timely solution, thus $dom(T_i) \in \mathbb{N} \cup \{F\}$. The value F is also used to indicate any process i with H_i inconsistent with the already executed base-level actions.
3. Time left W until the current base-level action completes execution.
4. The number L of base-level actions already initiated or completed.

We also have special terminal states SUCCESS (denoting having found and can execute a timely plan) and FAIL (no longer possible to execute a timely plan). Thus, the state space of the MDP is:

$$\begin{aligned} \mathcal{S} \quad = \quad & (dom(T) \times dom(W) \times dom(L) \times \\ & \bigtimes_{1 \leq i \leq n} dom(T_i)) \quad \cup \{\text{SUCCESS, FAIL}\} \end{aligned}$$

The identity of the base-level actions already executed is not explicit in the state, but can be recovered as the first L actions in any prefix H_i, for a process i not already failed.

The initial state S_0 has elapsed wall clock time $T = 0$, no computation time used for any process, so $T_i = 0$ for all $1 \leq i \leq n$, and no base-level actions executed or started so $W = 0$ and $L = 0$. The **reward** function is 0 for all states, except SUCCESS, which has a reward of 1.

The **transition distribution** is determined by which process i is being scheduled (a c_i action) or how execution has proceeded (a b action). For simplicity we assume that only one action is applied at each transition, although base level and computation action can overlap in real (wall clock) time.

Let $S = (T, W, L, T_1 ... T_n)$ be a state, and S' be a state after an action is executed. We use the notation $var[state]$ to denote the value of state variable var in $state$, for example $T[S]$ denotes the value of T in S, that is, the value of the wall-clock time in state S.

For a base-level action, $b \in B$, which is only allowed if $W[S] = 0$, the transition is deterministic: the count of executed actions increases, and all processes incompatible with b fail. That is, $W[S'] = dur(b)$, $L[S'] = L[S] + 1$, $T[S'] = T[S]$, and:

$$T_i[S'] = \begin{cases} T_i[S] & \text{if } H_i[L[S]+1] = b \\ F & \text{otherwise} \end{cases}$$

A computation action usually advances the wall-clock time, i.e. $T[S'] = T[S]+1$ and $W[S'] = max(0, W[S]-1)$. As a result, some processes may no longer be able to deliver a timely solution at all, we call such processes, as well as the computation actions of such processes *tardy*, as defined below. Consider any process i that might be given a computation time unit in state S and (possibly) terminating and delivering a solution. The time at which this execution of the solution can complete is given by the following equation, where $[i..j]$ denotes a sub-sequence from i to j, inclusive, and $dur(.)$ of a sequence of actions is the sum of durations of the actions in the sequence:

$$t_i[S] = T[S] + W[S] + dur(H_i[(L[S]+1)..|H_i|]) + 1$$

That is, $t_i[S]$ equals time now, plus time remaining until the current base-level action (if any) terminates, plus the duration of the tail of the H_i prefix, plus the 1 time unit allocated now. The probability that this is a timely execution is thus $1 - D(t_i[S])$. A process for which $D(t_i[S]) = 1$ has zero probability of delivering a timely execution and is called a tardy process. Thus, when doing a computation action, each process i that is tardy at S fails, that is, $T_i[S'] = F$ with probability 1; unless all processes are tardy in which case we fail globally, i.e. $S' = FAIL$. In the above cases, the transitions are deterministic.

We allow a computation action c_i only for processes i that have not failed and are not tardy at S. For such a valid action c_i, we have $T[S'] = T[S]+1$ and $W[S'] = max\{0, W[S]-1\}$, and $T_j[S'] = T_j[S]$ for all $j \neq i$ that are non-tardy. With probability $P_{C,i} = \frac{m_i(T_i[S]+1)}{1-M_i(T_i[S])}$ process i now terminates, given that it has not terminated before. Thus with probability $1 - P_{C,i}$ the process does not terminate, in which case we get $T_i[S'] = T_i[S] + 1$. If the process does terminate, as stated above, it delivers a timely solution with probability $1 - D(t_i[S])$ in which case we set S =SUCCESS. The solution fails to meet the induced deadline with probability $D_i(t)$, in which case we have $T_i[S'] = F$, unless in the resulting S' there is no longer any non-tardy process that has not failed, in which case set S' =FAIL.

5 Known Induced Deadline IPAE: Properties

We need only policies that start from the initial state S_0, so we can represent a policy as an and-tree rooted at S_0, with the agent's action as an edge at each state node, leading to a chance node with next possible states as children.

A policy tree in which every chance node has at most one non-terminal child is called linear, because it is equivalent to a simple sequence of meta-level and base-level actions. This can be extended to the case where there may be more than one non-terminal child, as long as there is only one such child with non-zero probability, thus we call these types of policies linear as well. With this definition of linear policies, we have:

Lemma 3. *In IPAE with known induced deadlines, there exists an optimal policy that is linear.*

Proof. Observe that transitions for base-level actions are deterministic, and thus it is sufficient to consider deliberation actions c_i at any state S. Examining the transition distribution in this case, the chance node has at most only two non-terminal children: one where process i terminates and fails, and one where it does not terminate. However, since the induced deadlines are all known then in fact $D_i(t_i[S])$

is either 0 or 1. However, the case $D_i(t_i[S]) = 1$ means process i is tardy, so c_i is not allowed. In the remaining case, $D_i(t_i[S]) = 0$ and the chance node has only one non-terminal child with non-zero probability. □

For known induced deadlines it is thus sufficient to find the optimal linear policy, represented henceforth as a sequence σ of the actions (both computational and base-level) to be done starting from the initial state and ending in a terminal state, unless we land in a terminal state due to previous actions in the sequence. Denote by $CA(\sigma)$ the sub-sequence of σ that contains just the computation actions of σ. Likewise, denote by $BA(\sigma)$ the sub-sequence of σ that contains just the base-level actions of σ. Denote by $\sigma_{i\leftrightarrow j}$ the sequence resulting from exchanging the ith and jth actions in σ. We call a linear policy contiguous if the computation actions for every process are all in contiguous blocks, formally:

Definition 1. *Linear policy σ is contiguous iff $CA(\sigma)[k_1] = CA(\sigma)[k_2] = c_i$ implies $CA(\sigma)[m] = c_i$ for all $k_1 < m < k_2$ and all computation actions c_i.*

Theorem 4. *In IPAE with known induced deadlines, there exists an optimal policy that is linear and contiguous.*

Proof. From the proof of Lemma 3, for known induced deadlines an optimal linear policy is non-tardy, and any process that terminates results in SUCCESS. Due to independence between the M_i, the probability of termination (and thus success) of each process depends only on the total processing time a_i allocated to it, and equals $M_i(a_i)$. Therefore, the total probability of success is invariant to the order of computation actions, as long as all computation actions do not cause i to become tardy. It is thus sufficient to show that every linear non-tardy policy can be re-arranged into one that is contiguous.

Let σ be an optimal linear policy, and k be the latest index where contiguity is violated in $CA(\sigma)$. That is, the sub-sequence $CA(\sigma)[(k+1)..|CA(\sigma)|]$ is contiguous, but we have $CA(\sigma)[k] = c_j$, $CA(\sigma)[k+1] = c_i \neq c_j$, and there exists $m < k$ such that $CA(\sigma[m]) = c_i$. Then, $CA(\sigma)_{m\leftrightarrow k}$ still results in a non-tardy policy when replacing $CA(\sigma)$ by $CA(\sigma)_{m\leftrightarrow k}$ in σ. That is because the moved c_j is made earlier, so cannot become tardy due to this change, and the moved c_i also does not become tardy as there is a later c_i that is non-tardy. Also, $CA(\sigma)_{m\leftrightarrow k}[k...|CA(\sigma)|]$ is contiguous by construction. This exchange step can be repeated until the policy becomes contiguous. □

Theorem 4 is an extension of a similar theorem that holds for $S(AE)^2$, to linear policies that contain base-level actions.

However, we still need to deal with scheduling the base-level actions. We show below that schedules we call *lazy*, are non-dominated. Intuitively, a lazy policy is one where execution of base-level actions is delayed as long as possible without making the policy tardy or illegal (base-level actions overlapping).

Definition 2. *A linear policy σ is lazy if $\sigma_{i\leftrightarrow i+1}$ is tardy or illegal for all i where $\sigma[i] \in B$.*

Note that if $\sigma[i]$ is a base-level action, an optimal policy will always schedule computation at $\sigma[i+1]$, since the duration of any base-level action is strictly positive and computation is better than idling.

Theorem 5. *In the IPAE with known induced deadlines, there exists an optimal policy consisting of a lazy contiguous linear policy.*

Proof. Define a lexicographic ordering $>_L$ on linear policies w.r.t. the index at which their base-level actions occur. $x >_L y$ if, for some $k \geq 0$ the first k base level actions in x and y start at equal indices respectively, and the $k+1$ action of x starts later than that of y. Let σ be the optimal contiguous linear policy that is greatest w.r.t. $>_L$. Assume in contradiction that σ is not lazy. Then by definition there exists an index i such that $\sigma[i] \in B$ and $\sigma_{i\leftrightarrow i+1}$ is legal and non-tardy and contiguous (no change in order of computation actions). Note that $\sigma_{i\leftrightarrow i+1}$ has the same computation time assigned to each and every process, as σ, so, being non-tardy, has the same probability of success as σ. Since $\sigma_{i\leftrightarrow i+1} >_L \sigma$ and is also optimal, we have a contradiction. □

6 Pseudo-Polynomial Time Algorithms

Since with known deadlines there exist pseudo-polynomial time algorithms for $S(AE)^2$, it is of interest whether this is the case for IPAE as well. The key notion allowing this to work for $S(AE)^2$ is that there exists an linear contiguous policy that assigns the processing in order of deadlines.

Unfortunately, this is not the case for IPAE because the timing of the base-level actions affects the order in which computation actions become tardy. Nevertheless, under additional restrictions it is still possible to get a pseudo-polynomial time algorithm. The idea is to find cases where the assignment ordering still holds, and then one can still use the dynamic programming scheme from $S(AE)^2$.

6.1 Bounded Length Prefixes

We observe that if we can pre-determine the time when base-level actions are executed, then it is possible to get an equivalent $S(AE)^2$ problem which can be solved by DP in pseudo-polynomial time. The number of such possible base-level action schedules is exponential in the maximum number of base-level actions in any of the H_i prefixes. Thus, under the assumption that this length is bounded by a constant K, we get a pseudo-polynomial time algorithm. Equivalently, we can artificially disallow executing more than a constant K actions before computation is complete, thus achieving the same effect.

First, observe that the sequences of actions we need to consider are only one of the H_i, as any action not in such a sequence would invalidate all the processes and thus is dominated. Consider the set of all linear contiguous policies that have a specific execution start time for all the actions in H_i, which we denote by the function I_i which maps actions in H_i to their start time. Note that this schedule for i may leave room for additional computations from other processes j, up until such time as j is invalidated by i. Under a specific I_i function, we can define an effective deadline

d_j^{eff} for each process j, beyond which there is no point in allowing process j to run. Note that the effective deadline is distinct from the known induced process deadline, which we will denote as d_i. The effective deadline is defined as follows. Let $k \in H_i$ be the first index at which prefix H_j becomes incompatible with H_i. Then process j becomes invalid at time $I_i(k)$. Also, consider any index $m < k$ at which the prefixes are still compatible. The last time at which action $H_i[m]$ may be executed to achieve the known induced deadline d_j is $t_{i,m} = d_j - dur(H_j[m..|H_j|])$. That is, process j becomes tardy at $t_{i,m}$ unless base-level action $H_i[m]$ is executed before then. The effective deadline d_j^{eff} for process j is thus:

$$d_j^{\text{eff}} = min(I_i(k), \{t_{i,m} : t_{i,m} < I_i(H_i[m])\}) \qquad (7)$$

Theorem 6. *Among the set of linear contiguous policies for a specific H_i and initiation function I_i, there exists an optimal policy where the processes are allocated in an order of non-decreasing effective deadlines.*

Proof. (outline): For the base-level action commitments I_i, by construction, process j results in a timely execution iff it terminates in time before d_j^{eff}. Thus, linear contiguous policies that have computational actions c_j only before d_j^{eff} are optimal w.r.t. the commitment I_i. The probability of success of such policies is given by Equation 2. The resulting limited problem setting is such that now the conditions of Theorem 1 apply. □

Due to Theorem 6, using the effective deadlines as the process deadlines takes into account the base-level actions, so we can now use the DP for $S(AE)^2$ to get an optimal computation-time policy and compute its success probability. Now we need to simply iterate over all possible H_i and all possible action initiation times in each H_i, and deliver the policy with the highest probability of success.

6.2 The equal slack case

We call the difference $d_i - dur(H_i)$ the *slack* of process i, because it is the maximum time we can delay the actions in H_i in order to have a timely execution when process i terminates. The special case of known induced deadlines where the slack of all processes is equal affords a pseudo-polynomial time algorithm using this scheme.

In the equal slack case, for each of the H_i sequences, it is sufficient to consider the actions in H_i to be executed contiguously, with the first action at time equal to the slack $d_i - dur(H_i)$. Now the effective deadline d_i^{eff} for each process j equals the time at which the first action $b \in H_i$ which is incompatible with H_j occurs, or d_i otherwise. Thus in this case we have only one initiation function I_i we need to consider for each H_i, so only need to run the DP scheme n times, regardless of the length of the H_i.

7 Algorithms for the General Case

The pseudo-polynomial time algorithm in Section 6 only applies when the deadlines are known and when the number of the base-level actions in each H_i is small. Therefore, we now propose several sub-optimal algorithms for the general case.

Max-LET$_A$. The Max-LET$_A$ schema is defined using as a parameter an algorithm A for the S(AE)^2 problem. First, we treat the problem as a known-deadline problem by considering the minimal value in the support of D_i for each i. (Other methods of fixing the deadline can be used, such as taking the expectation.) Then, for every process i, Max-LET$_A$ fixes the base-level actions to be at the **L**atest **E**xecution-**T**ime at which every action in the head needs to be executed (with respect to the known deadline). By fixing the base-level actions to those induced by process i, the IPAE problem instance can be reduced to an S(AE)^2 instance. Then algorithm A can be executed on the S(AE)^2 instance and return a linear policy P_i and a success probability of that policy. Max-LET$_A$ chooses the linear policy with the highest success probability among all P_is.

K-Bounded$_A$. K-Bounded$_A$ is similar to Max-LET$_A$ with one difference. Instead of fixing the base-level actions only to the latest start-time of every process i, K-Bounded$_A$ considers all possible placements for the first K actions, while the rest of the time-allocations are determined using the latest start-times.

Monte-Carlo tree search (MCTS). Since the IPAE problem can be defined as a finite-horizon MDP, standard heuristic search algorithms that operate on such MDPs can be applied. One such algorithm is the prominent MCTS (Browne et al. 2012). The MCTS version of MCTS that we have implemented uses UCT, which applies the UCB1 formula (Auer, Cesa-Bianchi, and Fischer 2002) as a scheme for selecting nodes and a random rollout policy that uses $-LPF$ as a value function for sampled time allocations.

8 Preliminary Empirical Evaluation

Our experimental setting is inspired by movies such as Indiana Jones or Die Hard in which the hero is required to solve a puzzle before a deadline or suffer extreme consequences. As the water jugs problem from Die Hard is too easy, we have selected the well-known 15-puzzle problem instead. In order to build IPAE problem instances, we first collected data by solving $10,000$ problem instances and recording the number of expansions required by A* to find a solution for each initial state in order to find an optimal solution, and the actual solution length. Then, we have created two CDF histograms for each initial h-value: the required number of expansions and the optimal solution lengths. IPAE problem instances of N processes were generated by drawing a random 15-puzzle problem and running A* until the open-list contains at least N search nodes, with $N \in \{2, 5, 10, 20, 50\}$, and then choosing the first N. Each open-list node i becomes an IPAE process, with M_i being the node-expansion CDF histogram corresponding to $h(i)$, R_i as the solution-cost CDF histogram (to represent the remaining duration of the plan), and H_i as the list of actions that leads to i from the start node. We assumed that each base-level action requires 3 time units to be completed. Finally, in order to have challenging deadlines, we have used $X_i = 4 \times h(i)$ (representing the deadline for reaching the goal). Note that even though X_i is known, D_i is unknown as R_i is unknown.

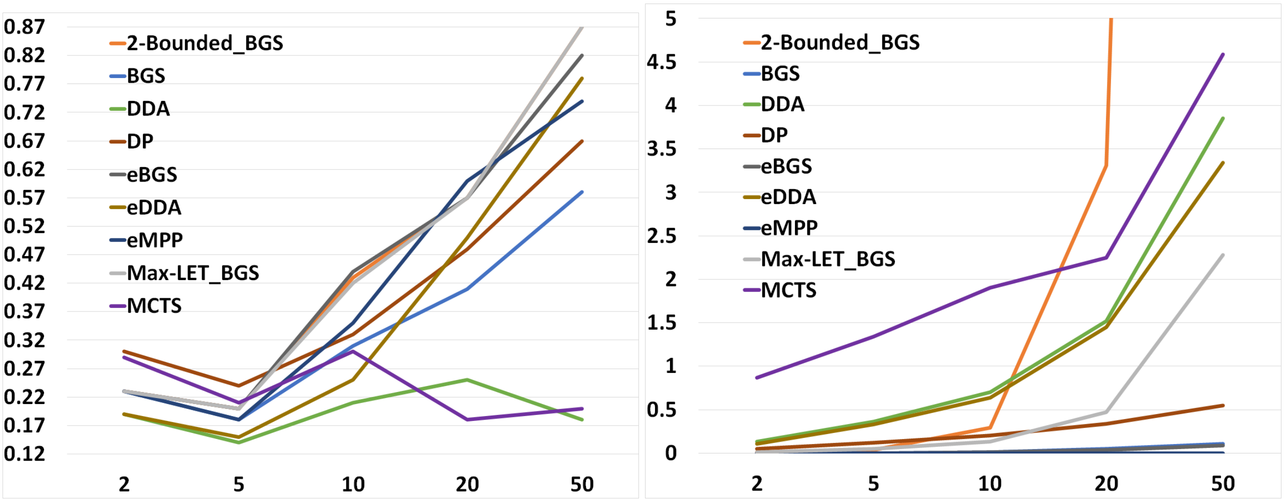

Figure 1: Success Probability (left) and Runtime (right) as a function of # processes

The following algorithms were empirically evaluated in our experiments. From S(AE)^2, we implemented: the basic greedy scheme (BGS) (Shperberg et al. 2019), delay-damage aware (DDA) (Shperberg et al. 2021), and dynamic programming (DP). In order to naively adapt S(AE)^2 algorithms and other basic schemes to the IPAE problem settings, we define a *demand-execution* version thereof. A *demand-execution* algorithm first decides which process i should be allocated the next time unit; then checks if a base-level action b is required for c_i to be non-tardy. If so, the action b is executed before c_i. We have evaluated a demand-execution version of the S(AE)^2 algorithm (eBGS and eDDA), demand-execution most-promising process (eMPP) that allocates consecutive time to the process with the highest probability to meet the deadline; if the process fails to find a solution, eMPP recomputes the probabilities with respect to the remaining time. Finally, we have implemented the algorithms described in Section 7. Specifically, we have evaluated Max-LET$_{\text{BGS}}$, 2-bounded$_{\text{BGS}}$, and MCTS with an exploration constant $c = \sqrt{2}$ and a budget of 100 rollouts before selecting each time allocation.

Figure 1 shows the average probability of success (left) of each algorithm (y-axis), as well as the average runtime (right), both vs. number of processes in the configuration (x-axis). First, the results indicate that the *demand-execution* version of the S(AE)^2 significantly improves over the basic version, e.g. for 50 processes DDA has a probability success of 0.18, while eDDA has a probability of 0.78 to find a timely action sequence. MCTS demonstrates poor performance both in terms of probability of success and in terms of runtime; this indicates that finding specialized heuristics tailored to the problem has a merit over using general purpose algorithms for (approximately) solving MDPs, as the search space is extremely large. The most competitive algorithms in terms of both probability of success and runtime are eBGS, eMPP and MAX-LET$_{\text{BGS}}$ which result in the best probability of success for 10, 20 and 50 processes, respectively, and were very competitive overall. In the future, we intend to explore the effect of different deadlines and different time units required for each base-level action in order to have a better understanding of the strengths and weaknesses of each algorithm.

9 Conclusion

Planning and search are generally intractable, so it is unrealistic to assume that time stops during planning. Hence the need for situated planning and search, especially when timely results are needed. In many cases, it may be possible to start executing a partially developed plan while continuing to search, thus allowing additional time to deliberate at some risk of performing actions that do not lead to a solution.

This paper extends the abstract metareasoning model for situated temporal planning proposed in (Shperberg et al. 2019) to allow for interleaving action and deliberation. As our abstract problem IPAE is NP-hard, even for known deadlines and known remaining sequence duration, we identified special cases where a psuedo-polynomial time algorithm can be developed, namely bounded-length plan prefixes and the equal-slack case.

Additional algorithms were developed for the general case of unknown deadlines and suffix durations. Experiments based on search trees for sliding tile puzzles show that algorithms based on ideas from the known-deadline case show promise. There is still work to be done in improving both these algorithms' results and their runtime, which is underway. A key issue is actually using the proposed scheme to initiate action during planning and search, which is non-trivial and has not been attempted here.

10 Acknowledgments

This work was funded by NSF-BSF via grant No. 2008594 (NSF), by grant No. 2019730 (BSF), and by the Frankel center for CS at BGU.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2): 235–256.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte-Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1): 1–43.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2018. Temporal Planning While the Clock Ticks. In *ICAPS*, 39–46. AAAI Press.

Korf, R. E. 1990. Real-Time Heuristic Search. *Artif. Intell.* 42(2-3): 189–211.

Russell, S. J.; and Wefald, E. 1991. Principles of Metareasoning. *Artif. Intell.* 49(1-3): 361–395.

Shperberg, S. S.; Coles, A.; Cserna, B.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2019. Allocating Planning Effort When Actions Expire. In *AAAI*, 2371–2378. AAAI Press.

Shperberg, S. S.; Coles, A.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2021. Situated Temporal Planning Using Deadline-aware Metareasoning. In *ICAPS*.

Zilberstein, S.; and Russell, S. J. 1996. Optimal Composition of Real-Time Systems. *Artif. Intell.* 82(1-2): 181–213.