31st International Conference on Automated Planning and Scheduling

August 2 – 13, 2021, virtually from Guangzhou, China

IntEx 2021

Proceedings of the Fifth Workshop on
Integrated Planning, Acting and **Ex**ecution

Program Committee

Ron Alford	The MITRE Corporation, McLean, Virginia, USA
Stefan Bezrucav	RWTH Aachen University, Germany
Gerard Canal	King's College London, England
Michael Cashmore	University of Strathclyde, Scotland
Dustin Dannenhauer	Navatek LLC, USA
Riccardo De Benedictis	CNR - National Research Council of Italy
Michael Floyd	Knexus Research, USA
Jeremy Frank	NASA, USA
Justin Karneeb	Knexus Research Corporation, USA
Shakil Khan	University of Regina, Canada
Sravya Kondrakunta	Wright State University, USA
Oscar Lima	German Research Center for Artificial Intelligence DFKI, Germany
Andrea Micheli	Fondazione Bruno Kessler, Trento, Italy
Marco Roveri	University of Trento, Italy
Vikas Shivashankar	Amazon Robotics, USA

Organizing Committee

Sunandita Patra	University of Maryland, College Park, USA
Mak Roberts	Naval Research Lab, USA
Wiktor Piotrowski	Palo Alto Research Center, USA
Tiago Vaquero	JPL NASA, USA

Automated planners are increasingly being integrated into online acting systems. The integration may, for example, embed a domain-independent temporal planner in a manufacturing system (e.g., the Xerox printer application) or autonomous vehicles (e.g., a planetary rover or an underwater glider). The integration may resemble something more like an “acting and planning stack” where an automated planner produces an activity or task plan that is further refined by an actor before being executed by the execution platform of the actor, such as, a reactive controller (e.g., robotics). Or, the integration may be a domain-specific policy that maps states to actions (e.g., reinforcement learning). Models for planning and execution can be same or different; the planning model can define context-dependent actions schema for online (re-)planning or can just specify flexibility to be handled separately at execution time. Online learning may or may not be involved, and may include adjusting or augmenting the model, determining when to repair versus replan, learning to switch policies, etc. A specific focus of these integrations involves online deliberation and managing the execution of actions, bringing to the foreground concerns over how much computational effort planning should invest over time.

In any of these systems, a planner generates action sequences that are eventually dispatched to an executive, yet taking action in a dynamic world rarely proceeds according to plan. When planning assumptions are challenged during execution, or some dynamic events occur, it raises a number of interesting questions about how the system should respond and which is the scope of online deliberation versus execution. Is the ”acting” side of the system responsible for a response or the ”planning” side? Or do the two need to cooperate and how much? When should the activity planner abandon or preempt the current goals? Should the task planner repair a plan or replan from scratch? Should the executive adjust its current policy, switch to a new one, or learn a new policy from more relevant experience?

The fifth edition of the workshop on Integrated Planning, Acting, and Execution (IntEx) aims to provide a forum for discussing the challenges of integrating online planning, acting, and execution, and to assess the potential for holding an integrated execution competitions at ICAPS. Topics include:

- online planning, acting, and execution improving planning performance from execution experience

- anytime or incremental planning
- discussions of plan dispatching or plan executives
- execution monitoring; comparing replanning, plan repair, re-goaling, plan merging
- managing open worlds with closed-world planners
- model learning from experience
- determining an observation policy; policy switching; incremental policy adjustment
- modelling, languages and knowledge engineering for interleaved planning and execution
- architectures and application for integrated planning and execution, execution
- monitoring, mixed-initiative on-line re-planning and execution

Sunandita, Mak, Wiktor, Tiago
IntEx 2021 Organizers
August 2021

Contents

An Integrated Framework for Remote Planning

Yaniel Carreno [1,2,3] [*], **Pierre Le Bras** [2] [*], **Èric Pairet** [1,2,3], **Paola Ardón** [1,2,3], **Mike J. Chantler** [1,2], **Ronald P. A. Petrick** [1,2]
[1] Edinburgh Centre for Robotics, Edinburgh, UK
[2] Department of Computer Science, Heriot-Watt University, Edinburgh, UK
[3] School of Informatics, The University of Edinburgh, Edinburgh, UK
{Y.Carreno, P.Le_Bras, Eric.Pairet, Paola.Ardon, M.J.Chantler, R.Petrick}@hw.ac.uk

Abstract

Robotics opens the possibility for safer operations in remote and hazardous environments, with multiple robots deployed to perform tasks that would otherwise present risks for human operators. However, these missions must be carefully planned and monitored to ensure their successful completion while keeping human supervisors in the loop for accountability. While many tools have been developed to tackle individual aspects of such processes, there are few systems combining plan development, review, and supervision in one framework. This paper proposes a mission planning framework designed for remote operations and integrating the following features: a user-friendly problem editor, a task-allocation algorithm, visual plan inspection, digital-twin progression reports, and plan deviation analysis. We show how this system is designed to support non-technical users with planning activities. In particular, the system provides continuous feedback on plan performance, comparing predictions with real implementations, enabling users to improve the requirements for future missions and correct modelling assumptions.

Motivation and Introduction

Robotic platforms provide the potential for safer operating solutions in remote and dangerous environments that would otherwise put human workers at risk (e.g., search and rescue in disaster zones or maintenance of offshore energy platforms). The development of such solutions typically faces two constraints. First, the uncertainty associated with hazardous environments often limits the practicality of deploying fully autonomous systems, e.g., cluttered and unstructured legacy installations, rapid weather changes, etc. As such, for safety and accountability reasons, the oversight of a human supervisor becomes necessary. Second, the remote settings for such missions often necessitate a variety of advanced robotic capabilities that must be highly coordinated to accomplish complex tasks, e.g., inspection and manipulation capabilities in ground, aerial, and subsea domains. As a result, robots are not only expected to coordinate and collaborate with each other, but must also be robust enough to support long-term autonomous operations where direct human interventions are impractical.

This paper proposes a mission planning framework designed for remote operation that integrates the following features: (i) user interfaces to facilitate the development and monitoring of remote missions by human supervisors, and (ii) robust AI planning solutions for heterogeneous multi-robot systems that implement intelligent behaviour in dynamic environments. The benefits of this framework are two-fold: it provides an intuitive end-to-end mission planning and execution system with human end users in mind, and accommodates the capture of offline and online performance data to enable planning experts to enhance model accuracy, system adaptability, and plan optimisation.

Our contribution is a symbiotic framework (see Figure 1) between mission specification, mission planning, and mission monitoring, each aiding the others and leading to a more user-friendly approach for remote planning. None of the individual modules, nor a trivial integration of them all, provides the same functionality. Our work presents newly developed features: (i) the design and development of the Problem Editor interface, fully linked with high-level planning; (ii) the extension of the planner to support offline and online mission planning in favour of enhanced accuracy and on-the-fly plan re-definition; and (iii) the introduction of a Plan Deviation Analysis which supports long-term missions and unexpected changes in the environment. Overall, the proposed framework enables non-expert users to plan missions in complex environments, which, as we believe, constitutes a robust system of interest to the planning community.

The rest of this paper is organised as follows. We first survey previous work related to the development of end-to-end planning frameworks. We then give an overview of the system before expanding on its five major components in detail. We finally conclude with remarks on future development.

Related Work

We begin by reviewing the literature regarding systems that address the problem of planning with human oversight and robust autonomous execution. This survey first considers *human-in-the-loop* systems, then focuses on planning techniques. We then present how these components have been integrated into several ambitious projects over recent years.

Keeping humans involved in automated processes has been recognised as a beneficial approach for optimisation (Scott, Lesh, and Klau 2002), notably for the ability to use up-to-date local knowledge to complement automated models (Fraternali et al. 2012).

[*] The first two authors have equal contribution.

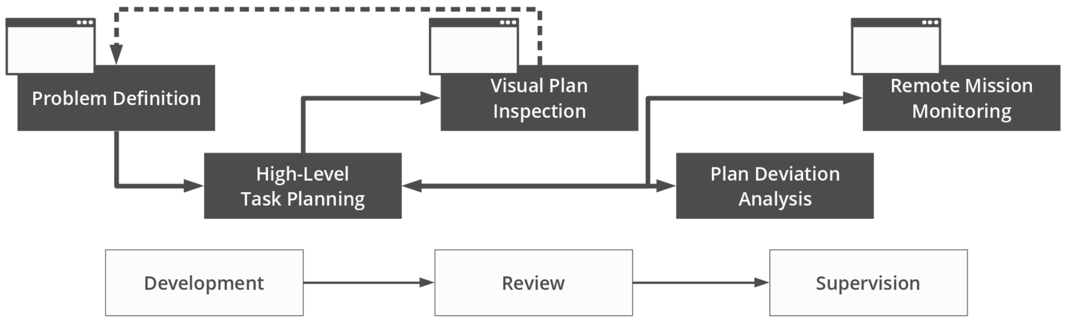


Figure 1: Overview of our planning framework. The approach is designed around 3 user tasks: mission development, plan review, and execution supervision. We integrate 5 components in the framework: problem definition, high-level task planning, visual plan inspection, remote mission monitoring, and plan deviation analysis.

With regard to planning systems, there have been many systems developed to assist in the generation of plans. The most common approach has been the implementation of graphical editing tools for domains and problems using node-link diagrams (Hatzi et al. 2010; Vodrázka and Chrpa 2010). Vrakas and Vlahavas (2003) also include an interface assisting users in defining problems for predefined domains. These techniques have been widely applied within larger planning systems such as EUROPA (Barreiro et al. 2012), GIPO (McCluskey and Simpson 2006), ModPlan (Edelkamp and Mehler 2005), and itSIMPLE (Vaquero et al. 2007). It is also common to provide users with a graphical depiction of generated plans for review. For example, (Kim and Blythe 2003) reuse the node-link diagram representation to describe and provide explanations for plan actions and their components. The PlanCurves technique visualises plans and enables the exploration of interactions between multiple robots (Le Bras et al. 2020).

It is also crucial for users to monitor the progress of plan execution, to check the completion of tasks against the schedule, and provide commands to rectify exceptions. For example, (Bernardini et al. 2020) propose to integrate both planning and monitoring interfaces in their onshore control centre. Relevant work in the area of Explainable AI Planning (XAIP) proposes a framework that utilises the existing planners to assist in answering contrastive questions (Cashmore et al. 2019) showing effectiveness explaining plan solutions for safety-critical domains. To optimise context awareness, such monitoring interfaces must portray agents executing plans within the environment, for example by overlaying their positions on a map (Cummings et al. 2019). The intricacies of offshore installations, however, require the system to render a more accurate depiction of the environment. The ORCA Digital Twin system is an example of a detailed monitoring interface, allowing users to navigate through a 3D simulation while robots are shown to be executing missions (Pairet et al. 2019).

AI temporal planners such as OPTIC (Benton, Coles, and Coles 2012) and POPF (Coles et al. 2010) often lack high-quality task distribution in the generated plans, when planning for multiple robots (Carreno, Petillot, and Petrick 2019). This is the result of their search strategies which focus on satisfying propositional action preconditions first and then action scheduling. Hence, further work has been done to find solutions that improve performance: (Bernardini et al. 2017, 2020), for example, use the POPF-TIF system (Piacentini et al. 2015). This general-purpose planning technology supports required concurrency, metric variables, predictable exogenous events and external advisors. However, this approach does not focus on the optimisation of task allocation for heterogeneous multi-robot systems. The MRGA+TP approach (Carreno et al. 2020) instead favours reasoning about the task allocation problem using the OPTIC planner which enables the introduction of preferences in the planning problem. In this work, we explore the potential of combining task allocation and AI temporal solvers.

In the past decade, AI planning techniques have been combined in *human-in-the-loop* systems to achieve solutions to challenging robotic problems. Examples of such projects include JAMES (Foster et al. 2012), SWARMs (Real-Arce et al. 2016), ORCA (Hastie et al. 2019), and MIMRee, (Bernardini et al. 2020), among others. The JAMES project used AI planning for socially-appropriate interaction using a single robot. Our main target is multi-robot systems. More closely related to our work are the SWARMs and MIMRee projects. The first, focused on coordinating cooperative behaviours in multi-vehicle (underwater) missions. SWARMs explores the areas of task allocation and scheduling, presenting solutions that include genetic algorithms and temporal planning. MIMRee uses AI agent technology to coordinate heterogeneous robotic assets while cooperating with onshore human operators who supervise the mission at a distance, via the use of shared deliberation techniques. While we also consider such goals in our work, our approach differs in a number of significant ways and can be applied to a wider range of applications in extreme environments: our approach is not domain-specific, it incorporates a plan deviation analyser to achieve robustness while executing missions, and we provide a tool to acquire data associated with planning and execution performance to enhance model accuracy, system adaptability, and plan optimisation over time. Our work is being developed in the context of the ORCA project, which considers similar deployment environments to MIMRee, including offshore energy applications.

System Overview

The application domain motivating our work centres around the automation of inspection and basic maintenance tasks on offshore energy installations, including legacy carbon decommissioning and maintenance of renewable resources. In these remote and hazardous environments, the implementation of robotic systems provides safer conditions for the completion of missions.

As such, we designed our framework for remote planning around three major stages of robot deployment:

1. **Development** of the mission specifications, defining goals and available resources to edit problem definitions and generate plans;
2. **Review** of the proposed plan, ensuring its safety and adequacy with the user's up-to-date knowledge and proposing updates to the problem if needed; and
3. **Supervision** of plan execution, assessing plan deviation and guarantees concerning mission success.

Five components[1] are integrated to help users fulfil these tasks (see Figure 1). First, we developed a graphical **problem editor**, allowing users to edit goals and preferences and to select the robots to use for missions based on a model of the domain. We use a **high-level task planning** system to allocate the appropriate robots to goals and generate the mission plan. The plan is then presented using a **visual plan inspector**, enabling users to visually inspect the plan, query details of it if necessary, and approve its execution. Users can then follow the mission progress on a **remote mission monitoring** interface, tracking the robots' movement within the environment in detail. Simultaneously, the **plan deviation analysis** reasons about changes between the scheduled tasks and the robots' progress in real-time, querying the planner for adjustments when needed. We present these components in more detail in the following sections, detailing their inner structure and how they communicate with one another.

Problem Definition

The task of defining the problem is the starting point for any mission. Our framework distinguishes between two problem statements: system predicates, which do not change from one mission to the other (e.g., the distance between waypoints), and user predicates, that users may update for specific missions (e.g., robot starting points). To support both types of statements, our system first uses a domain model that defines the system predicates and the structure of user predicates. A problem editor interface then uses these structures to assist users in defining their statements.

Domain Model: We model three aspects of the domain: the environment (including waypoints, a neighbourhood graph, and objects), the robots (with their capabilities and fluents, e.g., speed or battery level), and finally the structures for goals and preferences. Figure 2 presents examples from our model. While some aspects of the model are domain-specific

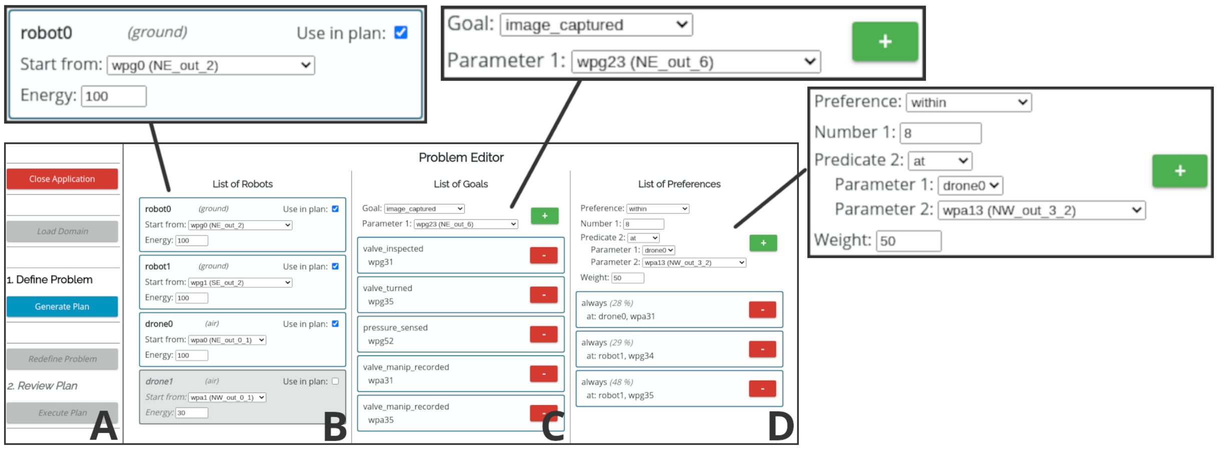
```
Environment Model:
 waypoints:
  {name:wpg0, coordinates:[22,14,0],
   neighbors:[wpg2], type:ground}, ...
 objects:
  {type:valve, position:wpg35}, ...
Robots Model:
 {name:robot0, type:ground, *position:wpg0, *energy:100,
  *available:true, recharge_points:[wgp0], speed:0.5,
  capabilities:[can_turn_valve,...], ...}, ...
Goals and Preferences Model:
 goal types:
  {name:valve_inspected, parameters:[valve]},
  {name:image_captured, parameters:[waypoint]}, ...
 preferences types:
  {name:within, goals:[number,predicate]}, ...
 preferences predicates:
  {name:at, type:state, parameters:[robot,waypoint]},
  {name:energy, type:function, parameters:[robot]}, ...
```

Figure 2: Excerpts from the JSON Domain Model. The environment model defines waypoints and objects. The robot model describes features and capabilities; fields with an * can be edited on the problem editor. The goals and preferences model describes their structures, and allows the editor to assist users when defining such statements (see Figure 3).

(e.g., environment data), the system architecture offers scalability to augment the domain in future development or ensure reusability across different domains. The modularity of this architecture, therefore, allows our work to be deployed onto any platform (real or simulated) and supports different domains to cope with new operational characteristics. In this paper, we present a domain and problems well-aligned with real-world applications, provided by ORCA-Hub's industrial partners. These components have been improved over time considering industrial experiences and necessities. We have used this system architecture to solve problems at different scales, increasing the number of robots and permuting their capabilities.

Problem Editor: We base our problem editor interface on Vrakas and Vlahavas's work (2003), assisting the user in creating three types of statements split in three lists (see Figure 3). The first one determines the set of available robots for the mission (Figure 3B). The user can select which robots to include in the plan and define their starting point and initial charge. The second is the list of goals for the mission (Figure 3C). While initially empty, the editor will use the domain model data to create an interactive form for the user to add goals. Similarly, the user can add constraint preferences to the third list (Figure 3D), providing decisive support for querying a plan with specific characteristics, such as maintaining a minimum charge level or pushing a robot towards a particular path, notably after having reviewed a previous plan that did not meet the user's expectations. Upon sending the user predicates to the back-end for planning, the domain model saves these statements in case the user decides a different plan is needed.
Once instructed to generate a plan via the problem editor in-

[1] In `https://github.com/plebras/PlanVisualisationLive` we present the framework's components linked to the Development and Review stages.



Problem Editor

List of Robots

robot0 (ground) Use in plan: ☑
Start from: wpg0 (NE_out_2)
Energy: 100

robot1 (ground) Use in plan: ☑
Start from: wpg1 (SE_out_2)
Energy: 100

drone0 (air) Use in plan: ☑
Start from: wpa0 (NE_out_0_1)
Energy: 100

drone1 (air) Use in plan: ☐
Start from: wpa1 (NW_out_0_1)
Energy: 30

List of Goals

Goal: image_captured
Parameter 1: wpg23 (NE_out_6)

valve_inspected wpg31
valve_turned wpg35
pressure_sensed wpg52
valve_manip_recorded wpa31
valve_manip_recorded wpa35

List of Preferences

Preference: within
Number 1: 8
Predicate 2: at
Parameter 1: drone0
Parameter 2: wpa13 (NW_out_3_2)
Weight: 50

always (28 %) at: drone0, wpa31
always (29 %) at: robot1, wpg34
always (48 %) at: robot1, wpg35

Figure 3: The problem editor interface. The control panel (A) allows users to load a domain and trigger planning and execution. The editor displays 3 lists: robots (B), goals (C), and preferences (D). For the latter two, the interface guides users into adding elements using the domain model data (top of lists); the properties already added are also visible to the user (bottom of the list).

terface, the system performs two actions. First, it saves the user-defined statements for future potential replanning. Second, it compiles the domain model and user-defined statements into a set of problem files, from which the high-level task planning processes can proceed.

High-Level Task Planning

The high-level task planning architecture is responsible for task allocation and task planning, taking into account all available information about system properties (e.g., domain models, physics, etc.) and mission specifications (e.g., goals, constraints, etc.). The system uses a Task Assignment (TA) component to allocate tasks to a set of robots and a Central Temporal Planner (CTP) to generate a plan solution. Here, we describe the key characteristics of these two elements.

Task Assignment: The task assignment component in our strategy is based on the MRGA approach (Carreno et al. 2020). TA is responsible for allocating mission goals to a fleet of multiple heterogeneous robots before planning. This method considers two cost functions to allocate goals: (i) the number of solvable tasks based on the robot capabilities, and (ii) the linear combination of the task makespan, the distance between the points of interest (POIs) and redundancy of the robot's sensory system. TA aims to optimise the distribution of robots in the environment to reduce mission time and avoid worst-case scenarios where all goals are allocated to a single robot. However, robots can implement tasks in different parts of the environment by considering goal capability requirements and robot capabilities (e.g., the ability to inspect a region, manipulate a valve, etc.). We claim this method improves plan solutions presented by benchmark temporal planners. The approach is planner agnostic, with the output of TA described in standard Planning Domain Definition Language with temporal constraints PDDL2.1 (Fox and Long 2003). The high-level task planning approach has been evaluated using a large number of temporal solvers (Carreno et al. 2020) supported by PDDL. Current system evaluations have not considered other planning languages such as RDDL (Sanner 2011). Task allocation distributes tasks by analysing robot and mission characteristics. The approach first evaluates the capabilities of each robot and the capabilities required to implement each goal to allocate the set of solvable tasks to appropriate robots. It then works to define regions where the goals are allocated using clustering methods. The number of designated regions is always equal to the number of robots available. Robots are distributed in the regions by considering the number of tasks they can implement in each cluster and the distance that separates them from the closest goal in each cluster. At the end of this process, each robot will have a goal allocated. The distribution of robots in the environment leads to the remaining tasks being allocated by considering task makespan, the distance between the robots and tasks, and the redundancy of the sensory system to execute critical tasks. Note that the robots are not tied to a single region; they can freely move if required to complete mission tasks.

The final (allocated tasks) set is then transformed into a set of PDDL instances of the fluent `(robot_can_act ?r - robot ?wp - poi)` which is defined in our domain. The PDDL domain constraints the implementation of the different actions in the environment to the appropriate robots that can work at different POIs. The decision of who is capable of executing a particular task depends on the TA reasoning. The set of instances of the fluent is added to the PDDL problem file with all other system specifications and therefore they are considered to generate the plan. Figure 4 (top) shows a set of instances generated by the TA which constraints the execution of tasks in different POIs to the robots that can act in these locations. We use this representation to generate plans using the benchmark planners. In this case, the introduction of the TA means the AI temporal solver does not deal with the task allocation problem directly, which reduces the planning times and improves the final plan solution. However, the flexibility of this system allows the user to decide over the task allocation if that is desirable. In this

```
Problem Instances:
(robot_can_act husky1 wpg52)
(robot_can_act husky0 wpg31)
(robot_can_act uav0 wpa35)
(robot_can_act husky1 wpg35)
(...)
Temporal Plan Solution:
   Time: (Action Name)                              [Duration]
  0.000: (navigation husky1 wpg1 wpg52)             [166.348]
  0.000: (navigation husky0 wpg0 wpg31)             [115.181]
  0.000: (navigation uav0   wpa0 wpa35)             [111.496]
115.182: (valveInsp husky0 camera_h0 wpg31)          [50.000]
166.349: (checkP husky1 p_analyser1 wpg52)           [20.000]
186.350: (navigation husky1 wpg52 wpg35)             [81.687]
268.038: (valveInsp husky1 camera_h1 wpg35)          [50.000]
318.039: (manValve husky1 uav0  wpg35 wpa35)         [30.000]
(...)
```

Figure 4: A fragment of the set of PDDL instances (top) generated by the TA. A temporal plan solution (bottom) for a set of huskies and a UAV in the environment.

case, the near-to-optimal task distribution considering goals and robot fleet characteristics is not a guarantee.

Central Temporal Planner: The planning module is responsible for generating plans that links a robot's actions with the implementation of goals previously assigned to it by the TA component. Missions are created by considering robot capabilities and the characteristics of the environment. This module interacts with other modules (TA and the environment) to obtain a world model that provides information about the robot states, capabilities, and information of the operating environment (e.g., distance between the POIs and map of possible refuelling points, etc.). Such information is used to generate domain and problem descriptions[2] in PDDL. The task planner uses mission knowledge to generate a plan which satisfies the goal allocation restrictions imposed by the TA component. Plans are built using the OPTIC planner which shows good planning performance in a large number of domains, with domain-independent heuristics and fast generation. The quality of the plan is determined by the metrics the user needs to optimise. The most standard is the minimisation of the *makespan*—the time that elapses from the start of plan implementation to the end. However, the OPTIC planner allows considering preferences and time-dependent goal costs.

Figure 4 (bottom) shows a fragment of a plan solution that involves two instances of Husky robot and a UAV (Unmanned Air Vehicle). The user requires the robots to (i) inspect (`valveInsp`), (ii) manipulate a valve (`manValve`) in the environment, and (iii) check the pressure of a boiler (`checkP`) in its digital panel. The UAV implements surveillance tasks to provide visual information to the user. The CTP takes as inputs the PDDL and problem files to generate a solvable plan. The plan's solution takes into account the TA output. For instance, TA evaluates `husky1` is the best robot to implement tasks in `wpg52`. Therefore the planner is restricted to find a solution where the action associated with checking the pressure of the boiler (`checkP`) is executed by `husky1`. The introduction of the High-Level Task Planning is fundamental to achieve a sequence of actions that leads a set of heterogeneous robots from an initial state to a goal state. The planning solution responds to a set of requirements the user presents to the system including goals and constraints (e.g., temporal, resources, etc.). As a result, the High-Level Task Planning provides an executable solution that can be analysed and visualised by the user in order to decide its implementation. In this work, we assume each goal is a single task. However, it does not impede the system to implement coordinated actions. For instance, action `manValve` requires a Husky and a UAV. The TA finds the best robots to execute the goals `valve_manipulated wpg35` (that require a Husky) and `valve_inspected wpa35` (that requires a UAV), which are effects of executing the same action. Using our centralised planning approach we deal with these types of dependencies allowing the robots to coordinate their efforts. In addition, the CPT supports reasoning regarding mission survivability dealing with mission numeric constraints. For instance, the actions associated with the battery recharge are introduced by the planner that keeps robot operation requirements in consideration when planning.

[2]In `https://github.com/YanielCarreno/MRGA` we present Task Allocation algorithm and the domain and problems used in this work (folder ICAPS-IntEx 2021).

Visual Plan Inspection

While the task planning processes will optimise the allocations of robots and ensure the plan safety, human supervisors will still be held accountable for the safe and efficient progress of the mission. Thus, it is necessary to provide them with means to assess the plan generated for them. We, therefore, integrated the visualisation system introduced by (Le Bras et al. 2020). This approach displays plans in three coordinated views (see Figure 5).

Activity Chart: The activity chart follows a common representation for planned tasks: Gantt charts. It displays scheduled tasks as horizontal bars, positioning them to reflect their timing and duration (Figure 5C). In this interface, activities are grouped by robots to highlight their individual roles in the plan. It also connects tasks that are meant to be performed in coordination (e.g. one robot manipulating an object and another robot recording the action). This visualisation is built directly from the plan data (i.e. the list of actions).

Scene Map: While the activity chart displays the planned actions in detail, our domain of application (offshore energy) often includes unstructured legacy installations and involves unpredictable environment factors (e.g. rapid weather changes). It becomes, therefore, necessary for the user to also inspect the planned robots' movements and possible interactions, and assess their safety. To address this issue, the scene map visualisation shows a top-down view of the environment and simulates the position of robots within it, with the robots' elevation shown on the left (Figure 5D). Panning the activity chart or selecting states on the time curve update

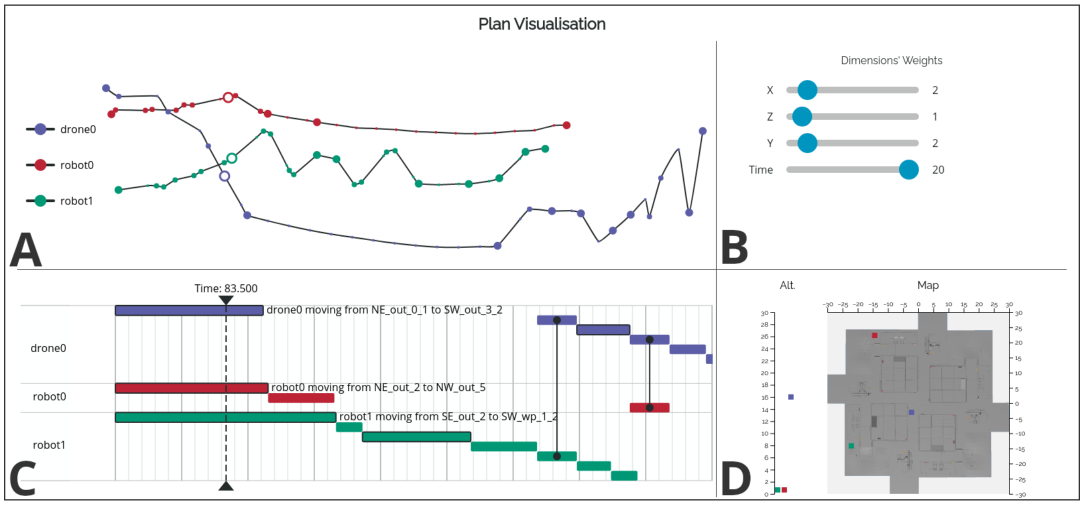

Plan Visualisation
drone0
robot0
robot1
A
Dimensions' Weights
X 2
Z 1
Y 2
Time 20
B
Time: 83.500
drone0 moving from NE_out_0_1 to SW_out_3_2
robot0 moving from NE_out_2 to NW_out_5
robot1 moving from SE_out_2 to SW_wp_1_2
drone0
robot0
robot1
C
Alt.
Map
D

Figure 5: Screenshot of the plan visualisation interface. The plan is shown as: time curves (A) to give an overview of the distances between robots across the plan, sliders (B) that allow the tuning of dimensions' weights to view multi-dimensional interactions from different perspectives, an activity chart (C) for detailed task schedule and robot coordination, and a map view (D) for details on the robots' positions.

the timeframe shown on the scene map, effectively animating it.

Time Curves: The two previous visualisations address the two main requirements for plan inspection: assessing the robots' tasks and movements. On offshore installations, however, there are situations when missions are made in response to emergencies. As such, the plan assessment, notably for movements, needs to be quick. From the list of actions described in the plan, the system will automatically infer the set of robot states throughout the mission, each with spatial and temporal coordinates. To enhance accuracy, the system will also estimate states every 20-time units if such interval is originally missing. The time curve visualisation technique (Bach et al. 2015) proposes to marginalise the multi-dimensional distances between states (three spatial and one temporal) down to two dimensions (Figure 5A). As a result, it creates a set of timelines (one per robot) distorted to reflect the closeness of robots both spatially and temporally. Note that the resulting chart expresses the robots' states in two abstract dimensions: the composites best preserving the original multi-dimensional distances. It is, therefore, impossible to label or interpret the axes of time curves, however, we introduce a posterior manipulation to "correct" the general orientation of curves, from left (start) to right (end).

This representation allows users to get a quick overview of the planned movements for robots and make rapid sense of their potential interactions. The weights of dimensions towards the time curves projection can be controlled using sliders, allowing the user to query details (Figure 5B). If the user decides the plan is not suited for the mission, the system allows them to return to the problem editor interface, where robots, goals, and preferences can be adjusted. Once content with the plan generated for the mission, the user may trigger its execution and monitor mission progress remotely.

Remote Mission Monitoring

We contextualise our work in the implementation of missions in remote and hazardous zones. Plan execution has been mainly evaluated in simulation scenarios using our ORCA-Hub simulator (Pairet et al. 2019). Our simulator is a ROS-enabled offshore energy platform environment composed of four gas and oil tower sites. Figure 6 shows a top (left) and field (right) view of the environment. The simulator supports the simultaneous deployment of multiple instances of robotic platforms, thus enabling a wide range of capabilities for cooperative inspection of large areas and emergency response. In this work, we employ UAV (ideal for aerial inspections) and Husky robots (medium-size robot with large payload capabilities, capable for example of extinguishing a fire) to implement missions that require totally decoupled as well as coordinated tasks. Figure 6 (right) shows a set of robots executing the plan presented in Figure 4 which involves tasks that require different sets of capabilities.

The simulator allows the user to evaluate the performance of the multi-robot system when executing the mission plan. This system provides a semantic description of the offshore energy structure, i.e., a map from 3D coordinates to high-level labels, which bridges the human-robot communication gap. Moreover, to ease some of the inherent robotic challenges, the simulator provides a semantic road map for autonomous point-to-point navigation and collision-free planning. Figure 6 (left) shows a representation of these points in the ground floor and the possible direction of navigation that robots can take. The POIs defined in our PDDL problem are directly aligned with the set of points in the road map. For instance, the high-level POI `wpg1` is defined by:

$$[\mathtt{x},\ \mathtt{y},\ \phi,\ \mathtt{roadmap\_poi\_name},\ \mathtt{poi\_property}],$$

where `x`, `y` and ϕ is the robot position and heading,

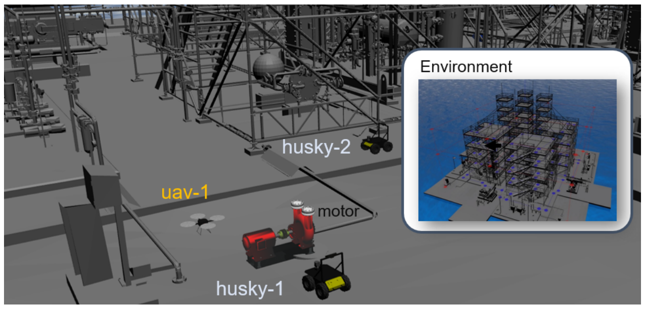


Figure 6: Overview of the simulator scenario with multiple aerial and ground robots executing a mission plan. The view of the platform shows the possible navigation paths.

poi_property provides relevant location information (e.g., goal location, recharge point, etc.), and roadmap_poi_name is the semantic tag of wpg1 in the roadmap (e.g., SE_out_2). This information enables high-level task planning to be connected to the simulation execution tools. Therefore, if a plan action asks a robot to navigate from wpg1 to wpg52 the simulator makes use of the roadmap_poi_name variable to query the roadmap for a path that connects both points.

The simulator supports Human-Robot-Interaction (HRI), including remote interaction with the robotic platforms through natural language commands and can receive vehicle and mission status through natural language such as *inspect the valve1*. However, this work does not directly use this type of single-action command implementation. Instead, we are interested in leveraging the strength of AI planning solutions to interact with multiple robots in the environment to solve long-term missions with a large number of goals. Moreover, the robots can reason about additional actions in the plan which are not directly related to achieving a goal. For instance, the plan solution can contain actions related to recharging the robot's battery, which is not a mission goal, but still fundamental for successful mission execution.

Overall, our simulator provides a comprehensive platform for developing and supporting HRI techniques, to aid in building human-robot trust in high stakes scenarios such as emergency response. It also enables testing of task planning algorithms for cooperative inspection and long-term autonomy, and human-guided supervision and control of the robotic assets from remotely located control stations. Being able to exhaustively test these applications ensures the coherence and efficiency of the execution plans, thus increasing the likelihood of adoption of robotics and autonomous systems for high-risk environments.

Planning and Execution

Robustness is a fundamental requirement for the success of complex robotic missions in extreme environments, requiring the consideration of mission failures, adaptability and survivability. Figure 7 shows a general description of the three elements involved in the mission implementation: Planning Framework (PF), Planning Execution Framework (PEF), and Plan Deviation Analyser (PDA). This architecture is an extension (grey components represent the additions) of ROSPlan (Cashmore et al. 2015). Here, we did not mention the users, considering this process occurs when they decide the plan obtained in the Development and Review phases (see Figure 1) can be executed. However, in the Supervision phase, the user can decide to stop the mission and replan it. In this section, we aim to describe the processes of plan dispatch, execution, and replanning in case of failures. We consider the system can adapt to a set of mission failures. Therefore, we evaluate the system's ability to overcome unexpected situations as a consequence of using an incomplete domain model.

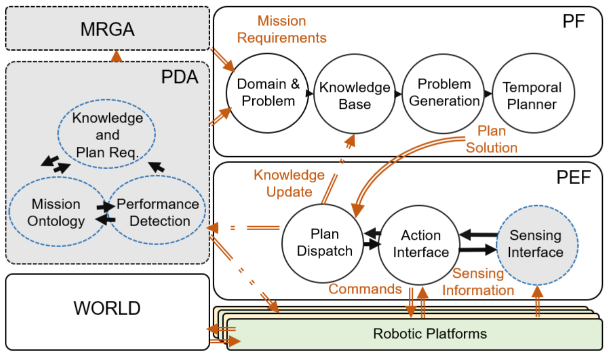


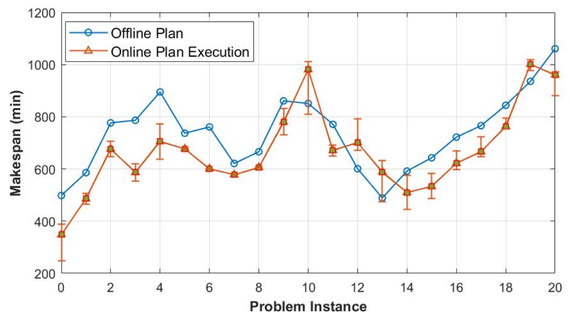
Figure 7: Framework for offline-online plan generation and execution. The system presents three main parts: Planning Framework (PF), Plan Execution Framework (PEF) and Plan Deviation Analyser (PDA).

The PF encloses the process of generating a plan solution. The planner produces plans using a domain model and a problem which are inputs to the Knowledge-Base (KB). As a result of using the PF we have a parsed plan available to be executed. The PEF dispatches the plan to a set of robots using the Plan Dispatch and Action Dispatch components from ROSPlan. The robotic platforms receive the actions through the action interface and provide feedback on the execution. If the action is successfully completed the next action in the plan is dispatched. If an action fails, however, further action dispatches are cancelled.

Plan Deviation Analysis: Our system introduces the PDA framework which can be used in two different cases:

- **Case 1:** There is a failure associate with the implementation of the action that makes the PEF to cancel the Plan Dispatch and claim replanning.
- **Case 2:** The Performance Detection component of the PDA detects possible problems during the execution of the action and interacts with the system to deal with them at the execution time.

In Case 1, the PDA takes the information the plan fails without completing all mission goals. The system checks the action that fails and queries the KB to evaluate possible reasons that guarantee the new plan solution is solvable. For instance, a Husky lost track of the image reconstruction of a structure while it is mapping an area. The PDA will consider the last location the quality of the map was acceptable and it



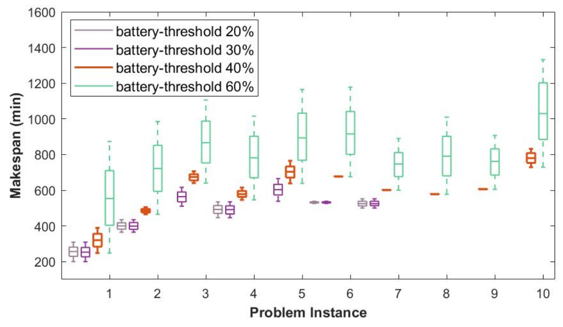


Figure 8: Makespan performance (offline) and real mission execution times (online) for 20 problem instances over 100 experiments (left). Makespan performance for different battery thresholds (right) for 10 problem instances over 50 experiments.

will add this information to the KB; based on the feedback from this evaluation, the system decides the next step. In addition, the PDA will evaluate other required preconditions to execute actions that were removed from the KB during execution and add them. For instance, if action `navigate` requires knowing the robot location (precondition `robot_at ?r - robot`) the PDA adds the robot's actual location to the KB before calling the planner to replan.

In Case 2, replanning is not needed. The PDA aims to observe the execution of particular actions and identify possible problems that might force the system to replan in the future. For instance, a Husky navigates from `wpg1` to `wpg52` and the navigation is delayed, as a consequence, the robot took longer to reach the second floor (`wpg52` location). The PDA can suggest the Husky to increase the speed to complete the action in the required time. These suggestions will depend on the action and robot characteristics that are defined in the Mission Ontology (MO). This type of online solutions makes robots adapt their behaviour while executing the plan avoiding the need for replanning. The MO is populated with multiple properties associated with all actions. Therefore, we can have a set of propositions to deal with failures associated with domain actions.

Performance Evaluation: Our framework can store data related to (i) offline planning, (ii) online planning, (iii) execution, and (iv) sensing information. This data includes mission goals; actions failures; planning time; makespan (at planning time and during the execution); and data associated with the sensors involved in the mission. Figure 8 shows examples of the results the framework can present. The first, shows the makespan performance and mission execution times for a set of 20 missions (some of them can take a day approximately). Results exhibit the execution times differ from the plan makespan originally obtained in a set of missions. The acquisition of this information over time allows planning experts to enhance model accuracy, system adaptability, and plan optimisation. Figure 8 (right) shows execution performance for the first 10 missions considering the Husky needs to recharge for different battery levels (battery life $\sim$ 5 hours and recharge time $\sim$ 2 hours). These results indicate best performance is obtained when the threshold for recharging is 40%. For 60% makespan increases substantially and for 20% and 30% the planner cannot solve all missions. However, we can use these small values to execute certain problems (e.g., 1 and 2) as the execution times are reduced as a result the Husky does not need to recharge. This information can be useful to the user to define which is the best time to recharge the battery of the robots to avoid unexpected situations where robots fail to maintain long-term operations. This project leads to a set of new autonomous solutions for extreme environments. Our framework supports the plan's evaluation considering industry-standard metrics such as implementation times and mission survivability.

Conclusion and Future Work

In this paper, we presented an end-to-end framework for planning and executing remote missions for multiple coordinated heterogeneous robots. Our system targets two main goals: providing oversight capabilities to human operators for safety and accountability while ensuring a robust implementation of long-term autonomous activities. In particular, the system is designed around three mission stages: plan generation, plan inspection and plan monitoring during mission execution. Overall, the proposed framework enables non-expert users to plan missions in complex environments which, we believe, constitutes a robust system for integrating AI planning solutions in industrial applications.

As future work, we are considering some additions to this framework. While our plan deviation analysis provides answers to exceptions occurring during execution, integrating contingencies into the plan generation process would also provide additional benefits reducing the amount of replanning needed during mission execution, and redefining mission goals or preferences by considering contingent plans. A second addition to our framework would be to implement an interface for users to visualise the performance of previous missions. Such a tool would allow planned activities to be analysed and compared against the reality of execution, understand the pitfalls or strengths of past missions, and make better judgements when developing new plans.

Acknowledgements

This work was funded and supported by the ORCA Hub (`orcahub.org`), under EPSRC grant EP/R026173/1.

References

Bach, B.; Shi, C.; Heulot, N.; Madhyastha, T.; Grabowski, T.; and Dragicevic, P. 2015. Time curves: Folding time to visualize patterns of temporal evolution in data. *IEEE Transactions on Visualization and Computer Graphics* 22(1): 559–568.

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkaylo, T.; Morris, P.; Ong, J.; Remolina, E.; Smith, T.; et al. 2012. EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization. *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)* .

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Proceedings of ICAPS*.

Bernardini, S.; Fox, M.; Long, D.; and Piacentini, C. 2017. Boosting Search Guidance in Problems with Semantic Attachments. In *Proceedings of ICAPS*, 29–37.

Bernardini, S.; Jovan, F.; Jiang, Z.; Watson, S.; Weightman, A.; Moradi, P.; Richardson, T.; Sadeghian, R.; and Sareh, S. 2020. A Multi-Robot Platform for the Autonomous Operation and Maintenance of Offshore Wind Farms. In *Proceedings of AAMAS*, 1696–1700.

Carreno, Y.; Pairet, È.; Petillot, Y.; and Petrick, R. P. 2020. Task Allocation Strategy for Heterogeneous Robot Teams in Offshore Missions. In *Proceedings of AAMAS*, 222–230.

Carreno, Y.; Petillot, Y.; and Petrick, R. P. 2019. Multi-Vehicle Temporal Planning for Underwater Applications. In *Proceedings of ICAPS Workshop on Planning and Robotics (PlanRob)*.

Cashmore, M.; Collins, A.; Krarup, B.; Krivic, S.; Magazzeni, D.; and Smith, D. 2019. Towards explainable AI planning as a service. In *Proceedings of the ICAPS Workshop on Explainable Planning (XAIP)*, 104–112.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the Robot Operating System. In *Proceedings of ICAPS*, 333–341.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proceedings of ICAPS*, 42–49.

Cummings, M.; Huang, L.; Zhu, H.; Finkelstein, D.; and Wei, R. 2019. The impact of increasing autonomy on training requirements in a UAV supervisory control task. *Journal of Cognitive Engineering and Decision Making* 13(4): 295–309.

Edelkamp, S.; and Mehler, T. 2005. Knowledge acquisition and knowledge engineering in the modplan workbench. *Proceedings of the First International Competition on Knowledge Engineering for AI Planning* 26–33.

Foster, M. E.; Gaschler, A.; Giuliani, M.; Isard, A.; Pateraki, M.; and Petrick, R. P. 2012. Two people walk into a bar: Dynamic multi-party social interaction with a robot agent. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction (ICMI)*, 3–10.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20: 61–124.

Fraternali, P.; Castelletti, A.; Soncini-Sessa, R.; Ruiz, C. V.; and Rizzoli, A. E. 2012. Putting humans in the loop: Social computing for Water Resources Management. *Environmental Modelling & Software* 37: 68–77.

Hastie, H.; Robb, D. A.; Lopes, J.; Ahmad, M.; Le Bras, P.; Liu, X.; Petrick, R.; Lohan, K.; and Chantler, M. J. 2019. Challenges in Collaborative HRI for Remote Robot Teams. *arXiv preprint arXiv:1905.07379* .

Hatzi, O.; Vrakas, D.; Bassiliades, N.; Anagnostopoulos, D.; and Vlahavas, I. 2010. A visual programming system for automated problem solving. *Expert Systems with Applications* 37(6): 4611–4625.

Kim, J.; and Blythe, J. 2003. Supporting plan authoring and analysis. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, 109–116.

Le Bras, P.; Carreno, Y.; Lindsay, A.; Petrick, R. P.; and Chantler, M. J. 2020. PlanCurves: An Interface for End-Users to Visualise Multi-Agent Temporal Plans. In *Proceedings of ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

McCluskey, T.; and Simpson, R. 2006. Tool support for planning and plan analysis within domains enbodying continuous change. In *Proceedings of the ICAPS Workshop on Plan Analysis and Management*.

Pairet, È.; Ardón, P.; Liu, X.; Lopes, J.; Hastie, H.; and Lohan, K. S. 2019. A digital twin for human-robot interaction. In *Proceedings ACM/IEEE HRI*, 372–372.

Piacentini, C.; Alimisis, V.; Fox, M.; and Long, D. 2015. An extension of metric temporal planning with application to AC voltage control. *Artificial Intelligence* 229: 210–245.

Real-Arce, D. A.; Morales, T.; Barrera, C.; Hernández, J.; and Llinás, O. 2016. Smart and networking underwater robots in cooperation meshes: the swarms ECSEL: H2020 project. In *Instrumentation viewpoint*, 19, 19–19. SARTI.

Sanner, S. 2011. Relational dynamic influence diagram language (RDDL): Language description (2010). *URL http: // users. cecs. anu. edu. to / ssanner / IPPC* 59–79.

Scott, S. D.; Lesh, N.; and Klau, G. W. 2002. Investigating human-computer optimization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 155–162.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains. In *Proceedings of ICAPS*, 336–343.

Vodrázka, J.; and Chrpa, L. 2010. Visual design of planning domains. In *Proceedings of ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 68–69.

Vrakas, D.; and Vlahavas, I. 2003. A Graphical Interface for Adaptive Planning. In *Proceedings of the ICAPS Doctoral Consortium*, 137–141.

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* $(\text{AE})^2$] 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 $(\text{AE})^2$ 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 $(\text{AE})^2$, to find a policy that maximizes the probability of a timely action sequence. However, unlike $(\text{AE})^2$, 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 $(\text{AE})^2$, 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:

$$e_i(t_d) = \operatorname*{argmin}_t \frac{LPF_i(t, t_d)}{t} \tag{3}$$

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*

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

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:

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

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 $1 \le i \le n$, 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 subsequence $CA(\sigma)[(k+1)..|CA(\sigma)|]$ 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.

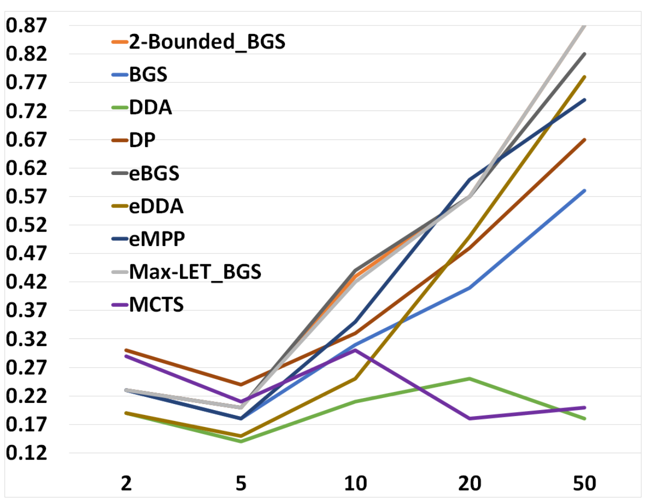
Max-LET$_A$. The Max-LET$_A$ schema is defined using as a parameter an algorithm A for the S(AE)^2 problem. First, we treat the problem as a known-deadline problem by considering the minimal value in the support of D_i for each i. (Other methods of fixing the deadline can be used, such as taking the 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.



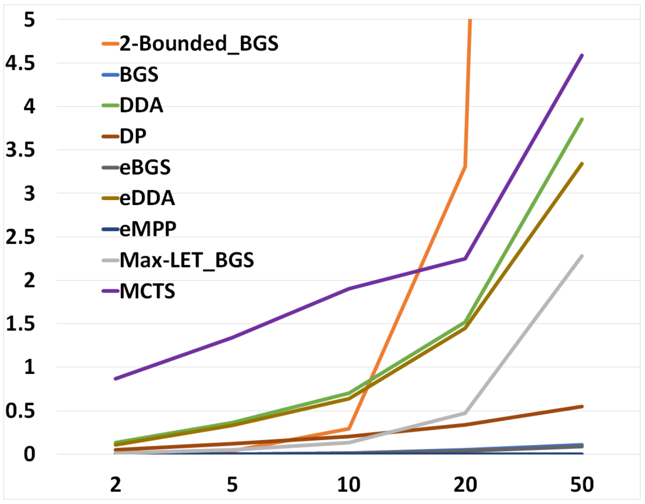


Figure 1: Success Probability (left) and Runtime (right) as a function of # processes

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 nontrivial 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.

Integrating Planning, Execution and Monitoring in the presence of Open World Novelties: Case Study of an Open World Monopoly Solver

Sriram Gopalakrishnan [*] [†], Utkarsh Soni [*] [†], Tung Thai [‡], Panagiotis Lymperopoulos [‡], Matthias Scheutz [‡], Subbarao Kambhampati [†]
[†] Arizona State University,
{sgopal28, usoni1, rao}@asu.edu
[‡] Tufts University

Abstract

The game of monopoly is an adversarial multi-agent domain where there is no fixed goal other than to be the last player solvent There are useful subgoals like monopolizing sets of properties, and developing them. There is also a lot of uncertainty from dice rolls, card-draws, and adversaries' strategies. This unpredictability is made worse when unknown novelties are added during gameplay. Given these challenges, Monopoly was one of the test beds chosen for the DARPA-SAILON program which aims to create agents that can detect and accommodate novelties. To handle the game complexities, we developed an agent that eschews complete plans, and adapts it's policy online as the game evolves. In the most recent independent evaluation in the SAILON program, our agent was the best performing agent on most measures. We herein present our approach and results.

Introduction

AI agents are often trained and evaluated in closed settings where the dynamics are fixed. They have shown spectacular performance in such settings; this is notably seen in game-playing agents such as in Chess and Go (Silver et al. 2017). However, we seldom consider how well these agents would act when novelties or changes are injected into the environment, i.e. an open-world setting. This would require execution monitoring to know what parts of the model have changed, and adapting to it as necessary. Developing agents to handle an open-world setting is necessary if we want to bring robust AI-agents into the real world.

With this in mind, DARPA (Defense Advanced Research Projects Agency) started a research program on Science of Artificial Intelligence and Learning for Open-world Novelty (SAIL-ON). The agents developed as part of this program are developed with the objective of handling novelties in the environment. One of the test-beds chosen for the SAIL-ON program is the game of Monopoly.

Monopoly is a board game about real-estate development with upto 4 adversaries. The objective of the game is to be the last solvent player. This is done through buying and developing properties, so as to charge the other players rent and fees. The game dynamics are first affected by dice rolls which determines how each player moves around the board. The game dynamics are also affected by drawing of "chance" and "community cards" (elements of luck), as well as by the combinations of the actions (strategies) of adversaries. If one adds in *novelties*, such as changes to the board layout, rent or bank rates, then the game becomes more unpredictable and hard to pre-train for.

Due to the aforementioned challenges, a plan of action can fall apart in a single round. In such an uncertain environment, we take the cautious and simple approach of state-evaluation after a single-step, where the strength of the approach comes from cautiously approximating the future value of the state after an action. This relatively simple approach outperformed other approaches as evaluated by an independent performer in the DARPA SAIL-ON program on the Monopoly test-bed.

In this paper, we first present and frame Monopoly as a challenging test-bed for interleaving online planning and execution, especially when novelties are injected (open-world setting). Then we discuss our agent for Monopoly, and also present the results of the evaluation made by an independent third-party evaluator; the evaluation compared our method against other teams in the DARPA SAIL-ON program. We propose our agent methodology as a strong baseline for future research on open-world robustness, and agents in Monopoly.

Monopoly Game And Simulator

Monopoly is a multi-player adversarial board game with upto 4 players (traditionally), where the objective is to be the last player solvent after having bankrupted the others. This is done through buying and developing properties so as to charge higher rent when the other players land on your properties. Players move across the board based on dice rolls, and can buy properties owned by the bank. If one lands on a property owned by another player, rent is charged. Rent on a property can be increased by owning all properties of a set (categorized by color); this is called having a *monopoly* over that set. The rent can be further increased by building houses and hotels on a monopolized set. Any policy or plan of action needs to be adapt to the changes of fortune with dice rolls, and the decisions of other players, which makes it chal-

[*]indicates equal contribution

lenging as a domain for integrated planning and execution. The full set of rules for the game simulator (test bed) and all the nuances can be found in (Haliem et al. 2021) which also contains a link to the game simulator code. This simulator was developed by an independent evaluator for the SAILON program. In the game simulator, one can also inject novelties on top of the standard game to study how the agent adapts to these modifications. Novelties were part of the evaluations of the agents developed for the SAIL-ON program, and will be discussed more in subsequent sections

Game Novelty

For the SAIL-ON program evaluation, agents were tested with one novelty injected per trial, where each trial is 100 games of Monopoly. The novelty could be added in any one of the 100 games and persists for the remaining games. The novelty could be changing the number of properties in a set required for Monopoly, the rent of a property after building a hotel on it, the order of properties on the board and such. The set of possible novelties is *not* shared with us by the evaluation/test team, and so it is left to us to make the agent as robust and adaptable to novelties as possible.

Agent Design

We developed our agent such that it's policy is controlled by a state-value function. The value of a state is primarily determined by the expected short and long term reward obtained from that state. Before we discuss the details of how these rewards are calculated, we first present the motivation for our design. Typically, approaches that use a value function for game-playing agents – like MCTS (Browne et al. 2012)– either simulate trajectories to the end and backpropagate the terminal state value to compute the starting state's value, or they use a limited lookahead with an evaluation function that captures the value of the rest of the trajectory. We use the simplest form of the limited-lookahead approach where we just lookahead by 1-step and then evaluate the next state by approximating the expected short and long term returns that would result by taking the action.

Our reason for planning with a 1-step lookahead was that in the game of Monopoly, a single roll of the die, or a chance card, or an adversary's decision could change the entire value of a state. So to compute the value of a state accurately with simulated actions, requires considering a very large set of branches from an extremely wide and deep tree that includes many possible combinations of dice rolls, combination of player decisions, auction bids, and more. It should be noted that each turn of a player also includes what are called out-of-turn moves by other players, which further increases the branching factor of the search tree; please refer to the monopoly rules in (Haliem et al. 2021) for more detailed information.

If one had a very accurate mental model of adversaries, the possible branches of the search tree might become more manageable. Additionally, pre-training a large neural network for state evaluation–as was done with alpha-go (Silver et al. 2017)– is not viable since our agent would have to handle novelties or modifications to the game (the space of which we did not know). Lastly, the evaluators impose a max time limit of 3 hours per full-game, so simulating enough MCTS rollouts for each action did not seem feasible.

Thus, in this work, our focus on intelligently evaluating a state after a single action by considering short and long term consequences; rather than requiring an accurate and complete model to rollout and evaluate each state, we consider long term consequences with simplifying assumptions (will be discussed). The state value includes the current monetary value of possessions, potential short and long term gains, as well as the future benefit of monopolized properties. Importantly, the evaluation function is largely parameterized with game attributes (that can change) and has few tuned constants; this helps make it robust to game variations. We will first go over the evaluation function. We will then provide some examples of the state attributes that are tracked and updated in $\mathcal{V}(s)$ to accommodate for novelty.

State Evaluation Function

The value of a state should consider the current (monetary) value of owned properties as well as the potential for future earning as possible future rewards. Thus, the evaluation function we propose is a linear combination of four terms i.e. $\mathcal{V}(s) = \mathcal{M}_{\text{assets}} + \mathcal{R}_{\text{s}} + \mathcal{R}_{\text{l}} + \mathcal{M}_{\text{monopoly}}$. Each of these terms is described below:

$\mathcal{M}_{\textbf{assets}}$**:** Property value of all the agent's assets that are not currently mortgaged. Each property can be mortgaged with the bank for cash. We can buy back the property from the bank for the mortgaged amount plus interest on the mortgage.

$\mathcal{R}_{\textbf{s}}$: short term expected gain in funds computed as the difference between expected rent the agent will get for the properties that it owns in state s and the expected rent it would owe to other players based on current ownership of properties over the next k turns. The expectations are computed over the probabilities of each player landing in a particular position in the next k turns. This is akin to a rollout with the strong relaxation (assumption) that no more properties will be bought or developed. To be specific, let $\mathbb{G}$ be the set of all agents, g_1 be our agent, $\mathcal{P}(g)$ denote the properties owned by agent g, $r(p)$ denote the rent of property p, and $Pr(g, p, k)$ denote the probability that an agent g will land on a property p in the k^{th} turn from state s, then $\mathcal{R}_s$ is computed as:

$$\begin{aligned}\mathcal{R}_s = \sum_{k} \sum_{g \in \mathbb{G} - g_1} \Big[\sum_{p \in \mathcal{P}_{(g_1)}} & Pr(g, p, k) * r(p) - \\ & \sum_{p \in \mathcal{P}_{(g)}} Pr(g_1, p, k) * r(p) \Big]\end{aligned} \tag{1}$$

$\mathcal{R}_{\textbf{l}}$: the expected long term change in funds. The computation for this term is similar to $\mathcal{R}_{\text{s}}$, except that the probability of an agent landing on a property is assumed to be uniform over all properties. Note that the long term gain is calculated for k full loops/passes around the board (not turns). The value of k for both $\mathcal{R}_{\text{s}}$ and $\mathcal{R}_{\text{l}}$ was taken as 5.

$\mathcal{M}_{\textbf{monopoly}}$: A monopoly gain term is computed to incorporate the monetary benefit our agent would get for monopolizing and improving all properties of the same color.

The purpose of this term is to drive our agent towards taking actions that would let it gain a monopoly on a color and subsequently perform maximal improvements on its properties. To compute $\mathcal{M}_{\text{monopoly}}$, we start by calculating the expected funds, $\mathcal{F}$, our agent would have after going around the board (full loop) k times ($k = 5$ in our implementation) from its current position as $\mathcal{F} = \text{cash possessed} + k * \text{go\_increment} + \mathcal{R}_l$ where the last term $\mathcal{R}_l$ is also computed for k loops around the board. Now let $\mathbb{C}(g_1)$ be the set of all colors such that our agent owns at least one property of that color. Then for each $c \in \mathbb{C}(g_1)$, we compute the combined potential rent for that color ($\mathcal{R}_c$) that our agent will get from all the properties of that color if it spends all of $\mathcal{F}$ in buying all the properties of color c followed by improving each of the properties as much as possible with the remaining amount from $\mathcal{F}$. This potential rent value is then scaled down based on how many properties the agent actually possess (currently) for that color. For example, if we own 1 out of 3 blue properties, then the potential value from that color should be much less than if we own 2 out of 3 red properties. The scaled potential value $\mathcal{R}_c^s$ is computed as $\mathcal{R}_c^s = \mathcal{R}_c / 2^{\mathcal{P}_{(c)} - \mathcal{P}_{(g_1)}}$ where $\mathcal{P}_{(c)}$ is the total number of properties of color c. We use an exponential function in the denominator to value color sets that are closer to completion significantly more than others. Since the set size can change as part of game novelties, we think this is prudent. Finally, the monopoly component of the state evaluation, $\mathcal{M}_{\text{monopoly}}$, is simply computed as the maximum $\mathcal{R}_c^s$ over all the $c \in \mathbb{C}(g_1)$. What this monopoly term does for the agent is to allow it to eschew buying new or bidding for properties if that amount can be used to complete and develop a monopoly.

Avoiding Bankruptcy

Another complication for the agent, is that it must try to avoid bankruptcy in the face of a lot of stochasticity from the game. So even if the expected value of a policy is high, if it risks bankruptcy then a lesser-value policy that minimizes the risk of bankruptcy might be preferred. Concretely, at any state s, the agent considers if each possible move from the set of possible moves $m \in \mathbb{M}$ with cost $\mathcal{C}(m)$ satisfies the following conditions:

Condition 1: $cash_{\text{current}} + \mathcal{R}_{\text{next}} - \mathcal{C}(m) \geq cash_{\text{min}}$ where $cash_{\text{current}}$ is the current amount of money our agent possess, $\mathcal{R}_{\text{next}}$ is the expected change in cash due to rent after one round, $cash_{\text{min}}$ denotes the absolute minimum amount needed to protect against bankruptcy. This covers misfortunes from the Chance and Community chest cards that the agent might draw.

Condition 2: $cash_{\text{current}} + \mathcal{R}_{\text{owed}} + worth_{\text{scaled}} - \mathcal{C}(m) - \mathcal{R}_{\text{worst}} > 0$ where $\mathcal{R}_{\text{owed}}$ is the expected income from charging rent that our agent will get in the next round, $worth_{\text{scaled}}$ is some mortgage value of all properties our agent owns, and $\mathcal{R}_{\text{worst}}$ is the maximum possible rent our agent could be charged in the next round. This protects against bankruptcy from landing on an adversary's property. Both the above conditions were used to prevent the agent from aggressively spending its cash and going bankrupt. Once we have pruned the moves in $\mathbb{M}$, our agent simply chooses the move such that $m = \arg\max_m V(s'_m)$ where s'_m is the next state after simulating the move m.

Novelty Detection and adaptation

To perform well in the SAIL-ON evaluation, our agent needs to detect the novelties introduced and adapt state evaluation accordingly. The novelties that the agent was tested on were hidden. To adapt, we maintain knowledge of the expected values for game-board attributes like property rent, dice outcome likelihood, etc. The evaluation function is parameterized with such attributes and is updated once a change is detected. Some of these values are provided directly as the state information and thus we keep track of the current values of these attributes by observing the state. For other attributes, the agent needs to observe the outcome of certain actions (like selling a property) to infer how the relevant attributes changed. In attribute value changes, we also detect novelties related to dice. This includes addition/deletion of a die, additional sides added to the dice, and the distribution of rolling any number on each die. The first two are inferred by observing the dice rolls in the game. For the last one, we model the distribution of rolling any number as Dirichlet distribution and use MAP estimates to update this distribution. This updated dice distribution is then used to compute the probability function Pr , as used in equation 1.

Evaluation and Results

Novelty type	NDA(%)	Best competitor NDA(%)	Win-rate (%)	Best competitor win rate (%)
None(PNWP)	-	-	76.48	63.61
CN-easy	40	20	71.1295	58.448
CN-medium	46.67	20	64.8895	61.867
CN-hard	48.00	24	28.899	43.173
AN-easy	90.32	80	70.473	62.335
AN-medium	90	90	94.432	68.5425
AN-hard	75	20	61.503	46.1825
RN-easy	52	3.33	74.9905	58.448
RN-medium	32	13.33	78.0975	50.8625
RN-hard	32	20	81.5815	67.6585

Table 1: Novelty detection and reaction performance in the DARPA SAILON program

Our agent was evaluated against other teams in the SAIL-ON program by an independent evaluator separately funded by DARPA. Evaluation consists of multiple trails, where each trial consists of 100 games of monopoly against 3 baseline agents. The baseline adversarial agents were programmed by the evaluation team to serve as the competition baseline. The behavior of the baseline agent is described in (Haliem et al. 2021) as the "simple baseline agent" in that work. During each trial, a novelty is injected during one of the 100 games, and kept for the remaining games.

Measures of performance

The following metrics help compare agent performance: (1) Pre-Novelty Win-Percentage (PNWP): The ratio of games won before any added novelty. (2) Novelty Detection Accuracy (NDA): This is the percentage of trials in which the novelty was correctly detected, and without a false positive before the novelty was added. (3) Novelty Reaction Performance (NRP): To compute this, the win ratio of our agent after the novelty was added is divided by the win ratio of the baseline agent before the novelty was added. These measures were not defined by us, but by the evaluation group, and directed by discussions in the SAIL-ON program. For every measure, our agent was evaluated with different classes of novelties, these are: Class Novelties (CN) such as new classes of objects like property classes, or new classes of actions; Attribute Novelties (AN) such as changes in the mortgage rate, rent costs; and Representation Novelties (RN) such as changes to the position of properties, and the color sets to which they belonged. Within each type of novelty, the evaluators further classified them into easy, medium and hard. As mentioned, we do not have more details about the specific type and distribution of novelties that our agent was evaluated on, as this information is currently hidden from us to evaluate agent adaptation better.

Results

We report our performance in Table 1 where we provide our performance and the performance of the best competitor for NDA and Win percentages for the different novelty settings. With respect to the win ratio of our agent, our pre-novelty win ratio (PNWR) was 76.48%, i.e the 3 other baseline agents combined only won less than a quarter of the games when there was no novelty injected. This win rate represents the efficacy of our agent design/playing algorithm for the standard Monopoly game. In comparison the win rate for the next best team was 63.61%. Our agent performs better before novelty was added to the game, and also in most settings after novelty was added to the game. The only setting in which our agent did not get the best result was for "NRP-CN-hard" (Novelty Reaction performance for hard class-novelties). This reflects our agent's ability to accurately capture both the short and long terms effects of actions, as well as, how well it can adjust for novelties in the game while making decisions by modifying the evaluation function during the gameplay. The evaluators also ran a special test to see how well our agent performs against another instance of our agent, and a baseline agent. The two instances of our agent won 40.75 and 39.43% of the games on average (over many trials).

Related Work

There are connections between replanning systems and handling open-world novelty. Replanning systems are aimed at dealing with unanticipated changes in the dynamics, but traditionally, replanning systems don't automatically characterize the novelty or change their domain model (Yoon, Fern, and Givan 2007). In (Cushing and Kambhampati 2005), the authors discuss how to update the planning problem to handle unexpected changes with a language for failure representation, but not how to automatically characterize a novelty and update the model. In our methodology we do online model-update and replanning by incorporating the novelties into our state-evaluation function.

With respect to agents for Monopoly, there have only been a few notable attempts; (Haliem et al. 2021) recently proposed a Reinforcement Learning (RL) approach where they train a Deep Q-Network agent (Mnih et al. 2015) to play Monopoly. To accelerate learning, they employ a ϵ-greedy approach during training where instead of executing a random action for exploration, the agent imitates the policy of a rule-based agent which was manually designed to follow known successful strategies to winning Monopoly. Other RL-based approaches include (Bailis, Fachantidis, and Vlahavas 2014) and (Arun et al. 2019) where the Q-function is again approximated by a neural network. To train the agent, the former approach uses the $Q(\lambda)$-learning technique (Peng and Williams 1994), whereas the latter uses experience replay (Mnih et al. 2015). To restrict the action space of the agent, all the mentioned techniques only select the action type, and the parameters are chosen by fixed rules; for example the *sell_property* action in (Haliem et al. 2021) would sell possesions in a fixed order, a hotel, a house or a property depending on availability.) In our approach, we only constrain the trade-related actions to rule based behavior. Further, we argue that our approach is more suited to developing an agent that is robust to open-world pertubations; the aforementioned RL-based approaches would require retraining the agent once any novelty is encountered, even some simple parameter changes such as rent values.

Another approach for playing Monopoly was presented in (Sammul 2018) which uses MCTS (Browne et al. 2012) for its decisions. To make this feasible, the author makes significiant game simplifications to reduce the action space and game-tree size. For example no out-of-turn actions are allowed, which is a significant deviation from the game. This pruning, coupled with the agent having access to the best (by win rate) adversary model it will play against, is what helped the MCTS method achieve a win rate of 63%. We, on the other hand, handle the chaotic nature of the game by evaluating the state features intelligently, and not rolling-out and relying on the availability of an accurate adversary model.

Conclusions and Future Work

We present the game of Monopoly as a challenging test bed for evaluating open-world robustness, and integrating online planning and execution. We propose our agent methodology of using a flexible state evaluation function as a strong approach to handling novelties in the environment as demonstrated through an independent evaluation in the game of Monopoly. There are plenty of interesting avenues for further research: these include including learning and updating adversary mental models, and analyzing the cost/benefit tradeoff in varying the lookahead depth in the game tree especially if the game and adversary models change over time.

Acknowledgement

This research is supported by the DARPA SAIL-ON grant W911NF-19-2-0006.

References

Arun, E.; Rajesh, H.; Chakrabarti, D.; Cherala, H.; and George, K. 2019. Monopoly using reinforcement learning. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, 858–862. IEEE.

Bailis, P.; Fachantidis, A.; and Vlahavas, I. 2014. Learning to play monopoly: a reinforcement learning approach. In *Proceedings of the 50th Anniversary Convention of The Society for the Study of Artificial Intelligence and Simulation of Behaviour. AISB*.

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.

Cushing, W., and Kambhampati, S. 2005. Replanning: A new perspective. *Proceedings of the International Conference on Automated Planning and Scheduling. Monterey, USA* 13–16.

Haliem, M.; Bonjour, T.; Alsalem, A.; Thomas, S.; Li, H.; Aggarwal, V.; Bhargava, B.; and Kejriwal, M. 2021. Learning monopoly gameplay: A hybrid model-free deep reinforcement learning and imitation learning approach. *arXiv preprint arXiv:2103.00683*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature* 518(7540):529–533.

Peng, J., and Williams, R. J. 1994. Incremental multi-step q-learning. In *Machine Learning Proceedings 1994*. Elsevier. 226–232.

Sammul, S. 2018. Learning to play monopoly with monte carlo tree search. *Computer science bachelor thesis, School of Informatics, University of Edinburgh*.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *nature* 550(7676):354–359.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. Ff-replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, 352–359.

When to Commit to an Action in Online Planning

Tianyi Gu,[1] Wheeler Ruml,[1] Shahaf Shperberg,[2] Eyal Shlomo Shimony,[2] Erez Karpas[3]
[1]University of New Hampshire, USA
[2]Ben-Gurion University, Israel
[3]Technion, Israel
gu@cs.unh.edu, ruml@cs.unh.edu, shperbsh@post.bgu.ac.il, shimony@cs.bgu.ac.il, karpase@technion.ac.il

Abstract

In online planning, planning happens concurrently with execution. Under the formulation of planning as heuristic search, when the planner commits to an action, it re-roots its search tree at the node representing the outcome of that action. For the system to remain controlled, the planner must commit to a new action (perhaps a no-op) before the previously chosen action completes. This time pressure results in a real-time search. In this time-bounded setting, it can be beneficial to commit early, in order to perform more lookahead search focused below an upcoming state. In this paper, we propose a principled method for making this commitment decision. Our experimental evaluation shows that our scheme can outperform previously-proposed fixed strategies.

Introduction

Many applications of planning involve time pressure. Often, we want to achieve the goal as soon as possible, minimizing the so-called Goal Achievement Time (GAT) (Hernández et al. 2012; Kiesel, Burns, and Ruml 2015). In situated temporal planning, external time constraints, such as buses or trains that depart at scheduled times, cause plans to become infeasible if we take too long to plan (Cashmore et al. 2018; Shperberg et al. 2021). One way to address time pressure is to design faster planning algorithms. But many planning problems are inherently intractable. The most direct way to address planning under time pressure is to allow actions to begin executing before a complete plan has been found. Further planning can then overlap with action execution. The fundamental question in such online planning settings is: when should the planner commit to an action?

Existing methods offer simple fixed answers to this question. Some methods use a fixed amount of lookahead for every action selection decision and commit to exactly one action given this lookahead. Others commit to multiple actions, even the entire sequence of actions leading all the way to the search frontier. In this paper, we develop a principled approach to action commitment that uses heuristic information to assess the planner's uncertainty about action values. This uncertainty then drives the decision of whether to commit to an action or whether to perform additional lookahead before deciding. Although our investigation is at an early stage, our preliminary results already indicate that the approach has promise, as it outperforms previous non-adaptive strategies in three challenging scenarios.

Background

Problem Setting

Our problem setting requires the system to be controlled at all times. That is, some action must be executing at any given time, even if it is just a no-op that leaves the state unchanged. (Some domains, such as fixed-wing aircraft control, do not have no-op actions.) We make the additional simplifying assumptions that actions are serial and that the world is fully observable and deterministic. Thus, we have a planner searching for a sequence of actions under the constraint that at all times at least one action (beyond those that have already completed) has been computed, committed to, and has begun execution. The objective of the system is to achieve a goal as soon as possible. Because we are addressing concurrent planning and execution, we used GAT as our main evaluation metric. This is the total time taken from the start of planning to the arrival of the agent at a goal.

We take a heuristic search perspective, in which planning explores an incrementally-generated tree of feasible action sequences, with the root of the tree representing the state resulting from the execution of all actions that have been committed to up to now. The planner is allowed to commit to actions earlier than required, in order to allow it to re-root the tree at a deeper node, thereby focusing the search later on into the future. A commitment queue records all the committed actions. How and when to make such additional commitments, so as to reduce the expected time to reach the goal, is the focus of this paper. Following Russell and Wefald (1991), we aim to pose and solve this question as a decision-theoretic metareasoning problem. However, even this limited focus is too general to formalize, hence we make additional metareasoning assumptions about the search process:

1. The order of decision in the planner is a fixed search tree structure, from early actions to later actions.
2. No replanning is permitted after action commitment, a decision to commit to an action in the sequence means that it will eventually be executed in the order specified.
3. We may re-start search at a new state if necessary, for example, if the controlled system departs from our assumption of determinism.

4. The only question we address is when to 'reroot the tree' at a successor of the root, that is, should we do this before it is necessary?
5. We assume a given expansion strategy that is not modified by the commitment strategy, other than by pruning the parts of the search tree inconsistent with the action commitments.

Note that the metareasoning assumptions are meant to define the constraints on the decision-making at the metareasoning level, rather than representing assumptions about the domain or the planner. They are used to define the distributions and utilities. Nevertheless, in an actual implementation the planner may deliberately act in a way that does not conform to the assumptions, especially when it is obvious that better performance can be achieved by violating the assumptions. For example, it is possible, due to several commitment decisions, to get a commitment queue containing a sequence of actions that makes the agent walk in a loop. In such cases, if this is observed before beginning to execute these actions, it may decide to remove such a loop from the action queue, even though this may not be admitted by the metareasoning assumptions.

Periodically during the search, perhaps after each expansion or periodically after a set of expansions, a metareasoning process decides between two options:

- commit to the current seemingly-best top-level action now and re-root the search tree accordingly, or
- postpone the commitment and continue the current search.

Note that if we always decide to postpone, eventually action execution will reach the current root node state and force us to commit to a next action.

Previous Work

The seminal work of Korf (1990) defined the problem setting of single-agent real-time search, in which a fixed number of expansions (or equivalently, amount of time) is allowed for lookahead node expansions, after which the search must commit to the next action to take and re-root the search tree. His RTA* and LRTA* algorithms back up h values from the lookahead frontier to inform the action choice, caching the backed up values at every node to allow the heuristic information to become more accurate over time and provably prevent the search from becoming stuck in infinite loops. (The LRTA* variant converges to the optimal h values.) These algorithms were designed to be simple and amenable to analysis. They commit only to a single action, which means that the lookahead of one iteration can have significant overlap with the nodes visited in the previous iteration, depending on the state space connectivity and the heuristic function.

The widely popular LSS-LRTA* algorithm (Koenig and Sun 2008) takes a different approach, committing to the entire sequence of actions leading to the most promising frontier node. This reduces the re-generation of nodes seen during the previous lookahead and reduces the overall overhead of the search per executed action, but note that it also commits the agent to certain actions, such as those at or near the frontier, for which little lookahead has been performed and for which the heuristic value of their resulting successor state is their only attractive attribute.

The Dynamic $\hat{f}$ algorithm (Kiesel, Burns, and Ruml 2015) modifies LSS-LRTA* in two ways. First, rather than idling the planner for $k-1$ time steps after committing to k actions, Dynamic $\hat{f}$ uses the entire time until all the committed actions have finished executing to perform lookahead search. The amount of lookahead is thus adjusted dynamically, rather than being fixed from the start. This often results in the next iteration having a sequence of more than k actions to the best node on the search frontier, leading to a virtuous circle of larger and larger lookahead. Second, rather than expanding the frontier node with the lowest f value, the algorithm computes an inadmissible heuristic $\hat{h}$, which when added to g yields the inadmissible (but possibly more accurate) total plan cost estimate $\hat{f}$. By selecting the node with lowest $\hat{f}$, Dynamic $\hat{f}$ tries to avoid being tempted by shallow nodes whose admissible f values are low merely because they haven't been explored as deeply as others.

The stark contrast between the two fixed commitment strategies of LRTA* (one action) and LSS-LRTA* and Dynamic $\hat{f}$ (all the way to the frontier) raises the question of whether a principled adaptive strategy can be found to decide when to commit to an action. The first approach in this direction was Decision-Theoretic A* (DTA*) (Russell and Wefald 1991), which attempts to optimize GAT by periodically deciding whether to continue the current lookahead search or commit to an action and re-root the tree. This is done by estimating whether the improvement in decision quality, measured by reduction in plan length, that is likely to result from further search would outweigh the time required to do the further search itself. In the implementation used for their experiments, training data was used to gather statistics on how often, and by how much, heuristic estimates tend to change as a results of further search. DTA* is not a real-time search algorithm, in that it does not respect or consider a time bound on lookahead. There is no requirement that the system constantly be executing an action and it is always permissible to deliberate further. Thus DTA* is capable of emulating A* and planning all the way to a goal before committing to its first action. DTA* is based on the less-performant depth-based lookahead of RTA* rather than the f-based lookahead of LSS-LRTA*, but it pioneered the deliberative metareasoning approach to action commitment.

The Mo'RTS algorithm of O'Ceallaigh and Ruml (2015) is basically a modification of DTA* into a true real-time search algorithm based on LSS-LRTA*. We focus here on its action commitment strategy, called $\hat{f}_{PMR}$. It assumes that a no-op 'identity' action is available in every state, which allows the planner to continue searching from the same root. Once the path from the root to the most promising frontier node has been identified, $\hat{f}_{PMR}$ considers each node in turn, asking whether additional search would be worthwhile, and stopping at the first node for which this appears true. However, $\hat{f}_{PMR}$ does not offer a principled way to evaluate this decision at each node. It estimates the benefit of search as the expected reduction in time-to-goal resulting from more certain estimates of action cost, which seems reasonable. However, it is much harder to asses the costs of stopping the re-rooting process short of the frontier. The $\hat{f}_{PMR}$ method uses the time required to regenerate the path from the node to the frontier, which, as the authors note, is not particularly reasonable because this repeated work would likely happen concurrently with execution, not affecting the goal achievement time directly at all. This leaves the approach fundamentally unsatisfying.

In this work, we propose what we believe to be a more principled metareasoning scheme for action commitment, which we call Flexible Action Commitment Search (FACS). We integrate FACS into Dynamic $\hat{f}$ and assess its behavior using three challenging grid pathfinding scenarios specially designed to stress real-time search in different ways.

Metareasoning for Action Commitment

Our objective is to minimize GAT. Thus g, h, and f values in the state space will represent the duration of actions and the total utility of a final outcome is exactly the sum of action costs/durations taken to reach the achieved goal state. So optimizing f directly optimizes total utility.

The metareasoning problem of heuristic search can be conceptualized as a POMDP in which each state represents an entire state space graph, complete with costs on every arc and h values at every vertex. To avoid confusion in this discussion, we will use the term 'vertex' for a node in the state space graph and the term 'state' for a state in the POMDP. The search does not know which exact state space graph it is dealing with, thus its situation is captured by a belief distribution over states. Every node expansion action results in an observation that rules out those state space graphs that are inconsistent with the vertices, action costs, and h values that are generated. The action of expanding a node is stochastic in that the search does not know in advance which new nodes, actions costs, and h values will be observed, so there are many possible belief distributions resulting from every expansion. The action of committing to an action and re-rooting the search tree at a new vertex is deterministic, as it does not yield new information. A goal in the POMDP is a belief that has positive support only on state spaces that all share the same path from the initial vertex to a goal vertex, providing a solution to the original problem but potentially harboring remaining uncertainty about the unseen portions of the graph. A policy for the POMDP corresponds to a search strategy, as it would prescribe an action for the search to take at every reachable belief state. Solving the POMDP for a policy that, for example, minimizes expected solution length would give a heuristic search strategy that finds a solution as quickly as possible by minimizing the expected number of expansions. Approaching such a problem in practice depends crucially on exploiting structure in the h values, the arc costs, and the distance to the nearest goal.

It is not feasible to solve this POMDP, or even to find a reliable approximation of its solution using standard approximation methods. Therefore, we propose a myopic metareasoning scheme that only considers the next action commitment decision. In this formulation, one of the following two options need to be chosen:

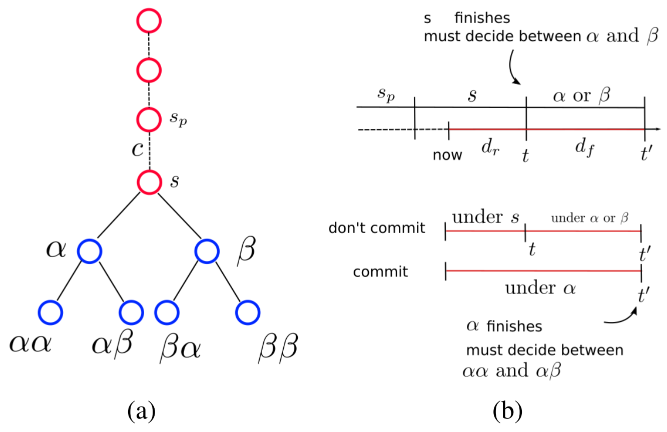


Figure 1: Committing vs not committing.

- Commit to the action with the best (least) estimated $\hat{f}$-value among all children of the current root node. We denote the node that corresponds to this action by α.
- Do not commit to α yet, and spend more time searching before deciding which action to take next.

Prematurely committing to α might reduce the quality of the solution. For example, if α leads to a dead-end and the search algorithm has failed to figured that out before committing, then it would be forced to turn around eventually, which would result in a solution with an increased cost. On the other hand, by gaining additional search time before making consequent decisions the search algorithm might be able to avoid future dead-ends or pitfalls, which would not have been possible to avoid otherwise. Thus, the utility of committing to α or not committing to α should depend in part on our certainty regarding the $\hat{f}$-value of α.

The Effect of Committing

Let $P_s^d(x)$ be the predicted probability of having the belief that $\hat{f}(s) = x$ given d more node expansions of search under node s. Denote by X_s^d the random variable distributed as P_s^d.

We begin with several additional simplifying assumptions:

1. Each node has exactly two children (a branching factor of 2), α and β, where α is the node with the highest expected utility (lowest expected $\hat{f}$-value); we will relax this assumption in the next subsection.
2. The time d_f required to fully execute an action is identical for all actions; this assumption will also be relaxed later.
3. When searching under a node s, the search time is evenly divided among all of its children.
4. The X_s^d random variables are independent for all d and s.

Under the above assumptions, we can now estimate the utility of committing to and of not committing to α. In Figure 1(a), we show the current search tree rooted at node s.

The available search time consists of the remaining time d_r induced by previous commitments and d_f, the time required to execute a full action α or β (see the the top red time line in Figure 1(b)). By committing to α, the agent would be able to invest all of the available search time to search under the children of α (the bottom red time line in Figure 1(b), starting with the word "commit"). We denote the children of α as $\alpha\alpha$ and $\alpha\beta$ (again, see the search tree in Figure 1(a)). Since we assume that search time is evenly divided between $\alpha\alpha$ and $\alpha\beta$, each of them receives a search duration budget of $d = \frac{d_r + d_f}{2}$. Thus, the utility estimate of committing to α (denoted as U_{commit}) can be defined as the expectation of the minimum $\hat{f}$-value of $\alpha\alpha$ and $\alpha\beta$, after searching d time units under each of them:

$$U_{\text{commit}} = \mathbb{E}\left[\min(X^d_{\alpha\alpha}, X^d_{\alpha\beta})\right] \tag{1}$$

If the agent chooses not to commit yet (commit later), the remaining time d_r will be used to search under current root s (see the middle red time line in Figure 1(b), starting with the words "don't commit"). Thus half of d_r ($\frac{d_r}{2}$) will be used to search under each child of the root. Even though α is initially estimated to have the lowest $\hat{f}$-value among the children of the root, this estimation can change after searching for $\frac{d_r}{2}$ time under α and β. In essence, the new $\hat{f}$-value estimation of α, induced by the additional search, can be greater than the new $\hat{f}$-value estimation of β. Thus, the rest of the time line (d_f) is used for searching under whichever child of the root is judged most promising at that time (again, see the middle red time line in Figure 1(b), starting with the words "don't commit"). As a result, the search duration under each grandchild of the current most promising child (either α or β) will be $d' = \frac{\frac{d_r}{2} + d_f}{2}$. In our simplification, the branching factor is 2, so:

Case 1: after $\frac{d_r}{2}$ time spent searching under α and β, we will believe that $\hat{f}(\alpha) \leq \hat{f}(\beta)$. In this case, the rest of the search time would be invested under α:

$$U_\alpha = \mathbb{E}\left[\min(X^{d'}_{\alpha\alpha}, X^{d'}_{\alpha\beta})\right] \tag{2}$$

Case 2: after $\frac{d_r}{2}$ time spent searching under α and β, we will believe $\hat{f}(\alpha) > \hat{f}(\beta)$. Symmetrically to the previous case, here the rest of the search time would be invested under β:

$$U_\beta = \mathbb{E}\left[\min(X^{d'}_{\beta\alpha}, X^{d'}_{\beta\beta})\right] \tag{3}$$

Then, we can estimate the overall utility of committing later by weighting the probability of α and β to become the most-promising nodes after the initial search time with their corresponding utilities. The probability of α becoming the most-promising child (choosing to commit to α) can be defined as follows:

$$P_{\text{choose }\alpha} = P((X_\alpha^{\frac{d_r}{2}} - X_\beta^{\frac{d_r}{2}}) < 0) \tag{4}$$

The utility of not committing at t' denoted $U^{t'}_{\text{don't commit}}$ can be estimated as:

$$U_{\text{don't commit}} = P_{\text{choose }\alpha} \cdot U_\alpha + (1 - P_{\text{choose }\alpha}) \cdot U_\beta \tag{5}$$

Using these equations, the metareasoning scheme simply needs to compute the utility of committing to α (Equation 1) and not committing to α (Equation 5), and to choose the metalevel action with the highest utility (lowest expected cost).

A Conceptual Example

In Figure 1, suppose that after the search, we obtain the expected cost under each leaf node, so we have $\hat{f}_{\alpha\alpha} = 3$, $\hat{f}_{\alpha\beta} = 5$, $\hat{f}_{\beta\alpha} = 4$, $\hat{f}_{\beta\beta} = 6$. And we also have $\hat{f}_\alpha = 3$, $\hat{f}_\beta = 4$ simply by backing-up from their best child node $\alpha\alpha$ and $\beta\alpha$ respectively. We are at the root node s and want to decide whether to commit to the current best action and reroot the search at α or not commit and keep searching under s. Suppose further that the expansion rate is 10 expansions per action duration, and that the action c leading to s is currently 5 expansions from completing execution. In this case, $d_r = 5$ and $d_f = 10$.

If we choose to commit, the total 15 expansions will be used to perform search under α, so $\alpha\alpha$ and $\alpha\beta$ both gain 7.5 expansions under our even division search time assumption. Now we can obtain the belief distribution for the future $\hat{f}$-value after search via Equation 10 (discussed below): $X^{7.5}_{\alpha\alpha} \sim \mathcal{N}(3, 0.4)$, $X^{7.5}_{\alpha\beta} \sim \mathcal{N}(5, 2.0)$. Then by applying Equation 1, we get the $U_{commit} = 3.2$. This can be calculated directly using the closed-form formula for the minimum of two normally distributed random variables (Nadarajah and Kotz 2008).

If we choose not to commit, we have two search phases: before and after c completes. In the first phase, we still search sub-trees under both α and β, so both gain $d_r/2 = 2.5$ expansions. Because the system can not be left uncontrolled, we are forced to commit when c completes. So in the second phase, after c completes, the search will only expand nodes either under α or β with $df = 10$ expansions. Thus we have $d' = (2.5 + 10)/2 = 6.25$ expansions for each leaf node. To compute U_α, now we can again obtain the belief distribution of future $\hat{f}$-value by Equation 10: $X^{6.25}_{\alpha\alpha} \sim \mathcal{N}(3, 0.2)$, $X^{6.25}_{\alpha\beta} \sim \mathcal{N}(5, 1.5)$. Equation 2 can give us $U_\alpha = 3.1$. The same computation can be applied to the β subtree to get $X^{6.25}_{\beta\alpha} \sim \mathcal{N}(4, 0.1)$, $X^{6.25}_{\beta\beta} \sim \mathcal{N}(6, 1.3)$, and $U_\beta = 4.2$. By Equation 4, say we get $P_{choose\alpha} = 0.7$, then we can have $U_{don'tcommit} = 0.7 \times 3.1 + 0.3 \times 4.2 = 3.43$. In this case, the meta-level decision is to commit since it results in the lowest expected cost of 3.2.

Relaxing the Assumptions

In order to relax the branching factor 2 and the identical action duration assumptions, we make the following modifications. Let $Children(x)$ be the set of children of node x, let $b = |Children(root)|$, and let d_a be the duration of action a. First, the search times d and d' needs to be updated with respect to b as follows: $d = \frac{d_r + d_\alpha}{b}$, $d'(a) = \frac{\frac{d_r}{b} + d_a}{b}$. Note that now d' is a function of the action chosen to be taken from the root. Then, the utility functions need to be updated. The utility of committing (Equation 1) should be generalized to:

$$U_{\text{commit}} = \mathbb{E}\left[\min_{c \in Children(\alpha)} X_c^d\right] \quad (6)$$

The utility of searching d' time under node c (generalization of equations 2 and 3):

$$U_c = \mathbb{E}\left[\min_{c' \in Childrenc} X_{c'}^{d'(c)}\right] \quad (7)$$

The probability of choosing node c after searching d' time under each child of the root is:

$$P_{\text{choose } c} = P(\operatorname*{argmin}_{c' \in Children(root)} X_{c'}^{\frac{d_r}{b}} = X_c^{\frac{d_r}{b}}) \quad (8)$$

Thus, the utility of not committing (generalization of Equation 5) becomes:

$$U_{\text{don't commit}} = \prod_{c \in Children(root)} P_{\text{choose } c} \cdot U_c^{t'} \quad (9)$$

Defining the $P_s^d(x)$ Distributions

The $P_s^d(x)$ distributions should reflect the effect of search effort under nodes given d more node expansions. Specifically, the more we search under a node, the more likely that our estimation of its $\hat{f}$ value will change and get closer to its true value. In addition, we assume that the closer a node is to a goal, the more accurate the original estimation of its $\hat{f}$ value. Finally, the average heuristic error on the path which leads to s from the root, $\bar{\epsilon_s}$, can be used as an indicator of the quality of the $\hat{f}$ value estimation of s. Thus, the variance of $P_s^d(x)$ should grow proportionally to d and $\bar{\epsilon_s}$, and the distance-to-go estimation of s. Let $dtg(s)$ be the distance-to-go estimation of node s, ed be the average *expansion delay* which measures the number of node expansions from the moment a node is generated until it is expanded (Cashmore et al. 2018). We model $P_s^d(x)$ as a normal distribution in the following way:

$$P_s^d = \mathcal{N}(\hat{f}(s), (\bar{\epsilon}_s \cdot dtg(s))^2 \cdot \min(1, \frac{\frac{d}{ed}}{dtg(s)})) \quad (10)$$

The mean of the distribution is the current cost-to-goal estimation, $\hat{f}(s)$. For an initial estimate of the variance we use the square of the heuristic error multiplied by the distance-to-goal of s. This uncertainty value is modeled as being reduced according to the fraction of the distance to the goal that we expect to explore using d node expansions (d divided by the expansion delay gives the distance explored). If the expected exploration depth surpasses the estimated distance-to-go, we clamp the fraction at 1.

To summarize, our Flexible Action Commitment Search (FACS) approach uses the P_s^d estimates about how the planner's beliefs about α and β will change after search in order to estimate U_{commit} and $U_{\text{don't commit}}$ and hence decide whether to commit to α or continue searching.

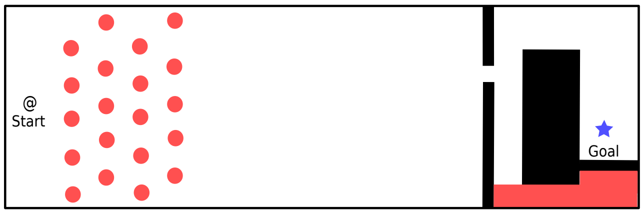


Figure 2: Schematics of grid benchmarks with tar pits.

Empirical Evaluation

Although our approach seems to be a more principled commitment strategy when all of its assumptions hold, given the fixed approaches in previous work, it remains to be seen how it performs in practice. In this section, we integrate FACS into Dynamic $\hat{f}$ and empirically evaluate it against three baselines: original LSS-LRTA* (i.e., commit-all), LSS-LRTA* with commit-one, and Dynamic $\hat{f}$ (i.e., commit-all with dynamic lookahead).

Synthetic Grid Pathfinding Domain

We implemented all real-time search agents in a synthetic grid pathfinding domain using the Euclidean distance heuristic. Figure 2 shows a schematic view of a tricky instance of our novel variant of the classic grid pathfinding problem, specially contrived to challenge real-time search. The black areas are the obstacles. The red patches are 'tar pits', cells for which the cost of moving to an adjacent empty cell is very high (i.e., high cost for stepping out from a 'tar pit'). With this setting, there will be a high cost if an agent commits to an action that steps into a pit, as the agent would have to step out of it in order to reach the goal. Note that the admissible heuristic function does not take these costs into account. Therefore, the agent has to be very careful about its commitment decisions. The small red tar pits are very common in the left part of the map, so a search's lookahead frontier will have a high probability of including at least one, possibly even as the best node. Thus, we expect an agent with a strategy that commits all the way to the frontier to be fairly likely to step into a tar pit at some point. In the middle, we have a long empty area. Since in this area there are no traps or mazes, algorithms can safely commit and re-root the search to the frontier nodes in order to gain search time for the future. In contrast, algorithms that conservatively commit only to one action at a time and re-root the search tree at every step cannot benefit from such gains in future search time. On the right side of the map, we have a corridor setting. Again, the red region is a large tar pit, where there is a small cost of stepping into this area, but a large cost of getting out of it. If agents do not have sufficient lookahead to observe that the red region is a high cost local minimum, they are likely to be tempted to get into this large tar pit, as it will seem to be a shorter path to the goal because we sample the goal position from the lower rows and sample the entrance of the corridor from upper rows so that the agent must go against the Euclidean heuristic to enter the upper corridor. However, with a sufficiently large lookahead, agents can de-

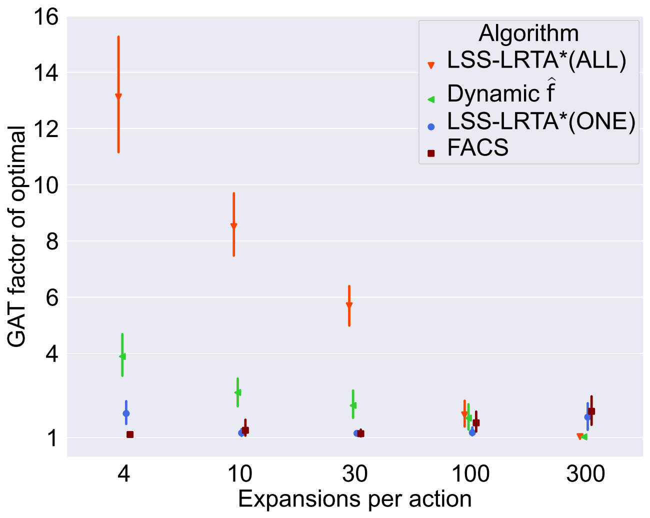


Figure 3: Goal achievement time as a function of search speed in grid pathfinding with tar pits near the start.

tect that this tar pit is a dead-end and avoid stepping into it altogether. Therefore, we expect agents that accumulate search time (i.e., lookahead) during the middle empty region to be able to utilize it to avoid the large tar pit. In the next section, we show results for maps that only have the left-side tar pits, maps that only have the right-side corridor and pit, and maps with both left-side and right-side pits.

Experiments

All algorithms were implemented in C++ and run on 64-bit Linux systems with 3.16 GHz Intel E8500 processors and 8 GB of RAM. We used grid maps of 50 rows and 200 columns. For each map, we set the start in the left-most column and goal in the right-most column, randomizing the row numbers of the start position and goal position to generate 100 problem instances. We used lookahead limits of 4, 10, 30, 100, and 300 expanded nodes per action (i.e., the relative search vs action execution speed), shown in the x axis of each plot in Figures 3-5. We set the cost of stepping out of a tar pit as 1,000 expansions. The y axis shows GAT, normalized as a factor of the GAT of a clairvoyant agent that immediately commits to an optimal plan without searching. Error bars show 95% confidence intervals on the mean over all the instances. The legends are sorted by the geometric mean across all lookahead limits.

Figure 3 shows the result of grid pathfinding problems with tar pits near the start. FACS preforms consistently close to clairvoyant across all search speeds. The commit-one strategy is also very competitive at low search speed, due to its conservative commitment strategy that helps the agent avoid stepping into tar pits. The Dynamic $\hat{f}$ and commit-all strategies are both far from optimal in this map, with dynamic $\hat{f}$ stepping less frequently into tar pits as it accumulates a slightly longer lookahead by the time it reaches the tar pit field.

Figure 4, shows results for maps with a corridor field near

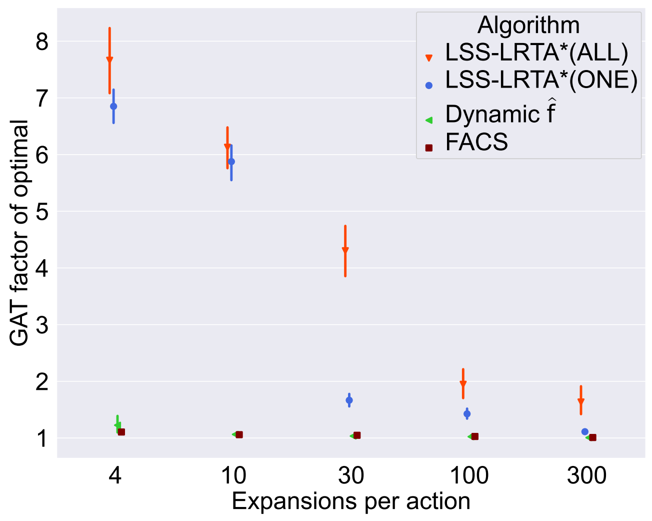


Figure 4: GAT with corridor and tar pit near the goal.

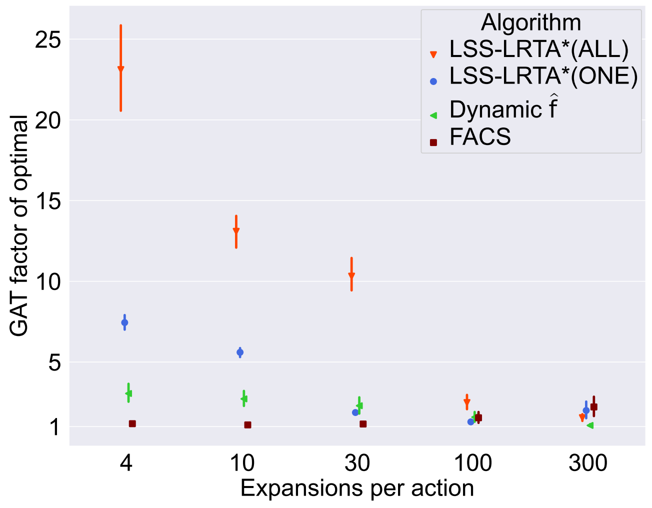


Figure 5: GAT with tar pits at both ends.

the goal. Both FACS and dynamic $\hat{f}$ perform close to clairvoyant since they take advantage of accumulating search time in the empty area and thus have a lookahead that is large enough to detect the dead-end and avoid stepping into the high-cost trap area. Both variants of LSS-LRTA* are unable to detect the dead-end with lookahead limit below 30.

In Figure 5, we show the results for maps with tar pits both near the start and near the goal. FACS is able to survive both the tar pit field and corridor field, with a conservative commitment strategy initially, adapting to an aggressive commitment strategy in the empty area, and having a sufficiently large lookahead to avoid the large tar pit when reaching the corridor.

Discussion

We have suggested an approximate metareasoning scheme for action commitment geared at focusing a real-time search, in order to get additional time to search farther ahead in the search tree. Our scheme involves some domain assumptions, as well as several metareasoning assumptions. While FACS performed well in our contrived grid pathfinding domain, further experiments are necessary to characterize when its assumptions lead to poor behavior, thereby guiding further theoretical work.

Possible Extensions

Typically, metareasoning assumptions are not true limitations. Instead, these are just a way to simplify the semantics and computation of expected utilities for the search. The independence assumption falls in this category; it is made so that we can have an easy-to-compute estimate, even though in practice it does not hold.

Our treatment of action commitment is in a different category. As mentioned above, if we observe that we have committed to a set actions that will lead us through a loop in the path, it is clear that this sequence of actions achieves nothing, except a delay. Even in such cases, this delay may be useful as it can be used to do additional search before physically reaching a possible trap (Cserna, Ruml, and Frank 2017). It thus is a non-trivial issue when we might wish to un-commit actions, even in such a seemingly simple case.

We might also wish to un-commit actions when we observe that the predicted f-costs resulting from deeper search are much worse than those initially projected. In such cases where actual action execution has not reached such an unexpectedly bad state, it may be better to un-commit actions and expand nodes that seemed worse and pruned by the commitments earlier on. When it might be good to do that is another non-trivial issue.

Re-examination of the set of assumptions about the search process is also needed, especially if we use a completely different search component in the online setting. Of special interest is the effect of using envelope search (Björnsson, Bulitko, and Sturtevant 2009; Gall, Cserna, and Ruml 2020) or Monte-Carlo tree search (MCTS) (Browne et al. 2012; Schulte and Keller 2014) on both the metareasoning decisions and on actual performance.

Last but not least are low-hanging fruit relating to additional experimentation with parameters of the scheme developed in this work. One issue is varying the frequency at which we perform metareasoning independently from the expansion rate. Is it better to perform metareasoning after each expansion (possibly more precise but a large overhead), once per real-world action, or only after search phase finishes, as done in the empirical evaluation in this paper?

Summary

This paper introduces FACS, a basic metareasoning scheme for action commitment geared at focusing a real-time search in order to get additional time to search farther ahead in the search tree. Due to numerous assumptions and decisions needed in order to simplify the analysis that might have been done differently, this is preliminary work that has considerable room for expansion. Nevertheless, favorable empirical results in contrived grid pathfinding scenarios show that this approach has promise and could lead to a principled treatment of one of the most fundamental issues on online planning and execution.

Acknowledgments

This work was partially funded by NSF-BSF via grant No. 2008594 (NSF) and grant No. 2019730 (BSF).

References

Björnsson, Y.; Bulitko, V.; and Sturtevant, N. R. 2009. TBA*: Time-Bounded A*. In *Proceedings of IJCAI*, 431–436.

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 *Proceedings of ICAPS*.

Cserna, B.; Ruml, W.; and Frank, J. 2017. Planning time to think: Metareasoning for on-line planning with durative actions. In *Proceedings of ICAPS*.

Gall, K. C.; Cserna, B.; and Ruml, W. 2020. Envelope-Based Approaches to Real-Time Heuristic Search. In *Proceedings of AAAI*, 2351–2358. AAAI Press.

Hernández, C.; Baier, J. A.; Uras, T.; and Koenig, S. 2012. Time-bounded adaptive A. In *Proceedings of AAMAS*, 997–1006.

Kiesel, S.; Burns, E.; and Ruml, W. 2015. Achieving goals quickly using real-time search: experimental results in video games. *Journal of Artificial Intelligence Research* 54: 123–158.

Koenig, S.; and Sun, X. 2008. Comparing real-time and incremental heuristic search for real-time situated agents. *Journal of Autonomous Agents and Multi-Agent Systems* 18(3): 313—341.

Korf, R. E. 1990. Real-time Heuristic Search. *Artificial Intelligence* 42: 189–211.

Nadarajah, S.; and Kotz, S. 2008. Exact Distribution of the Max/Min of Two Gaussian Random Variables. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16(2): 210–212. doi:10.1109/TVLSI.2007.912191.

O'Ceallaigh, D.; and Ruml, W. 2015. Metareasoning for Concurrent Planning and Execution. *ICAPS Workshop on Planning and Robotics (PlanRob-15)* 86.

Russell, S. J.; and Wefald, E. 1991. *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Schulte, T.; and Keller, T. 2014. Balancing exploration and exploitation in classical planning. In *Proceedings of SoCS*.

Shperberg, S. S.; Coles, A.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2021. Situated Temporal Planning Using Deadline-aware Metareasoning. In *Proceedings of ICAPS*, volume 31, 340–348.

Do You See What I See?
An Egocentric View of our Pansophical Planning Problems

Xiaotian Liu, Alison Paredes, Christian Muise
Queen's University, Kingston, ON, Canada
{liu.x, 20asp3, christian.muise}@queensu.ca

Abstract

Classical planning takes a pansophical view of the world: everything is fully known, observed, and static. While there are extensions to partial observability, this leapfrogs an important intermediate step of embodied agent design: egocentricity. In this work, we propose a semi-automated mechanism that allows planning domain designers to convert classical planning problems into an egocentric alternative. The generated planning problems are classical as well, and we introduce an open-loop replanning mechanism that progressively explores the egocentric space until the original goal is solved (or deemed unsolvable). Our work serves as a crucial first step towards embodied agents that can be equipped with an appropriately specified egocentric version of known environment dynamics.

1 Introduction

Most classical planning problems are defined with a pansophical reference frame where the environment is fully observable. However, in many planning tasks, an agent only has a limited view of the environment. Existing approaches on partially observable planning problems, such as conformant or contingent planning, use belief states to represent uncertainty in the environment (Hoffmann and Brafman 2005; 2006). However, representing belief states often requires access to information such as what objects are present in the environment in advance. Many planning applications do not provide access to such information. These types of egocentric agents often need to incorporate exploration in the strategy (James, Rosman, and Konidaris 2019). Egocentricity can also be necessary for embedded agents to transfer skills across environments (Charniak 2020). Often, solving a planning problem egocentrically requires manual conversion by a domain expert. The conversion process can be time-consuming, sometimes requiring the definition of a new set of syntax. A standardized semi-automatic approach can lift the burden of manual conversion, decreasing the hurdle of egocentric planning research and open up the door to domain-independent planning techniques to this setting.

We propose a novel semi-automatic approach that can convert classic planning problems defined in PDDL into egocentric alternatives. Our method takes a set of user-defined object types with their initial states as input, and outputs the egocentric version of the original problem. We assume that the world is made up of objects. Thus predicates define all objects' states and relationships. By defining which objects can be observed by an egocentric agent, we can extract the observable state space from the corresponding predicates. We use an exploration algorithm that searches through the original problem's state space until a plan is found or deemed impossible. The iterative exploration is facilitated by an open-loop replanning mechanism, which progressively updates the observable state space. Thus, our overall framework consists of two interacting processes: one determines which set of fluents are observable, and the other explores the unobserved state spaces.

The egocentric planning sub-problems generated by our approach can be solved using off-the-shelf planners. Our algorithm also keeps syntax and vocabularies consistent with the original planning problem, making results easily interpretable. We test the validity of our approach by using it on five classical planning problems and corresponding egocentric interpretations of them. Our experimental results demonstrate that our approach is practical through both quantitative and qualitative evaluations. To the best of our knowledge, there has not been any attempt to semi-automatically convert classic pansophical planning problems into egocentric alternatives. More complex or domain-specific problems may require further modification of our proposed approach. Nevertheless, we believe this work is an important step forward that can encourage more automated planning research in egocentric settings.

In the following sections, we provide preliminary definitions and a detailed outline of our framework, followed by descriptions of our experimental setup and a discussion of the results. In addition, we provide a step by step example of our approach using a simple grid based planning problem. Finally, we will conclude this paper with a summary of our work and possible future directions.

2 Preliminaries

Modified STRIPS Notation

We extend the commonly used STRIPS notation to include the notion of objects. In STRIPS, a planning problem is defined with a tuple $\langle F, I, A, G \rangle$. F is a set of fluents, $I \subseteq F$ is the initial state of the world, and $G \subseteq F$ is the goal state. A is the set of actions available. For each action

$a \subseteq A, PRE(a) \subseteq F$ is the precondition of that action, $ADD(a) \subseteq F$ is the add effect, and $DEL(a) \subseteq F$ is the delete effect. Our method operates over planning problems specified in PDDL. Therefore, we consider fluents to be represented by predicates with typed objects. We define a set O to be the set of objects in each planning problem. We include O in the STRIPS problem, and our complete planning problem is defined as a tuple $P = \langle O, F, I, A, G \rangle$.

We will demonstrate our approach using the standard language for specifying automated planning problems, PDDL (Haslum et al. 2019). The details of the PDDL language are beyond the scope of this paper, and we refer the interested reader to (Haslum et al. 2019) for a complete discussion.

Grid Navigation Example

We use a 2D navigation problem specified in PDDL as an example to illustrate our approaches throughout this paper. The problem is the simplified version of the Search-and-Rescue problem (Teichteil-Königsbuch and Fabiani 2007). In this planning problem, an agent is spawned on a 2D grid, and its goal is to search and pick up a person and then navigate to the hospital. The planning problem includes 4 object types: a robot object, location objects, a person object, and a hospital object. The predicates include object location predicates, connection predicates connecting locations, and a holding predicate indicating whether an agent is holding a person. The action set includes moving to another location and picking up the person. A visual illustration of the problem with the corresponding PDDL definitions are shown in Figures 1 and 2.

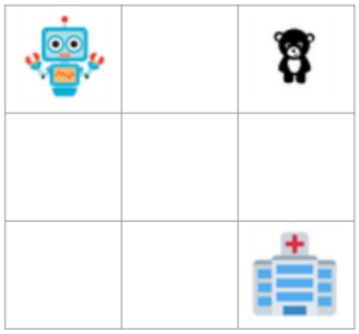


Figure 1: Graphical illustration for Search-and-Rescue

```
DOMAIN:
(define (domain searchandrescue)
  ......
  (:predicates
    (conn ?v0 - location ?v1 - location ?v2 - direction)
    (robot-at ?v0 - robot ?v1 - location)
    (person-at ?v0 - person ?v1 - location)
    (hospital-at ?v0 - hospital ?v1 - location)
    (carrying ?v0 - robot ?v1 - person)
    (handsfree ?v0 - robot)
    (move ?v0 - direction)
    (pickup ?v0 - person)
    (dropoff))

    (:action move-robot
        :parameters (?robot - robot ?from - location ?to - location ?dir - direction)
        :precondition (and (move ?dir)
            (conn ?from ?to ?dir)
            (robot-at ?robot ?from))
        :effect (and
            (not (robot-at ?robot ?from))
            (robot-at ?robot ?to))  )

    (:action pickup-person ......)

    (:action dropoff-person ......) ) )

PROBLEM:
(define (problem searchandrescue)
    (:domain searchandrescue)
    ......
    (:init
        (conn f0-0f f0-1f right)
    ......
        (conn f2-2f f1-2f up)
        (dropoff )
        (handsfree robot0)
        (move up)
        (move down)
        (move left)
        (move right)
        (pickup person0)
        (robot-at robot0 f0-0f)
        (person-at person0 f0-2f)
        (hospital-at hospital0 f2-2f)
    (:goal
        (person-at person0 f2-2f)))
```

Figure 2: PDDL for Search-and-Rescue

3 Approach

For a planning problem to be egocentric, only parts of the state space are observable. We can construct an egocentric planning problem $P^{'} = \langle O^{'}, F^{'}, I^{'}, A^{'}, G^{'} \rangle$ from the original pansophical planning problem by finding the corresponding subsets for each element in P. $\langle O^{'}, F^{'}, I^{'} \rangle$ are the egocentric version of the problem's objects, fluents and initial states. Their value depends on an agent's observable state space. $A^{'}$ is the set of actions available to the egocentric agent. It contains actions in the original pansophical set A with additional actions required for exploration. $G^{'}$ is the goal of the egocentric agent, which is to gather information until the original goal G can be achieved. The objectives of our approach are to determine the current egocentric state and to facilitate exploration. We separate these tasks into two separate algorithms. The first algorithm extracts the observable states from an pansophical planning state space. We call this algorithm the Egocentric Subset Extractor, or ESE. The second algorithm facilitates exploration and information gathering in an partially observable environment. We call this component the Iterative Exploration via Replanning, or IER. The subsequent sections will motivate and describe both algorithms in detail, followed by a demonstrated example of our approach on the grid world domain.

Egocentric Subset Extractor

The ESE algorithm determines which fluents in the pansophical planning problem are observable. We adopt an object-centric view of the world and assume objects' states/relationships are represented by fluents in the form of object typed predicates. Thus, if we can determine which set of objects are observable by an agent, we can find the corresponding set of fluents that are also observable. The visibility of certain objects depends on the agent's egocentric state, and we refer to these as *anchor objects*. The ESE algorithm determines which subset of fluents in the initial set in a planning problem $P = \langle O, F, I, A, G \rangle$ should be included in the egocentric problem.

Our approach requires the user to determine the type and the initial set of observable objects. We define this set S as a tuple $\langle T, C, R\rangle$. T contains the anchoring object types, and C is a set of anchor objects currently observable. R is defined as a set of predicates that connect one anchor object to another. We make the following four assumptions for our approach:

1. We can uniquely determine a planning problem's egocentric aspects by way of the observable anchor objects.
2. If an object is observable, then its states and relationships with other objects, in the form of predicates, are also observable.
3. Any predicates that do not contain any anchor objects are always observable.
4. There exist a set of predicates that define connections or relations among anchor objects.

The first assumption is a direct result of adopting an object-centric view of the world. The second assumption allows us to determine which fluents in the form of predicates are observable. Intuitively, predicates represent the state and relationship among objects. Thus any predicates that contain observable objects should also be observable. For the third assumption, if a predicate contains no anchor objects, its observability is independent of an agent's egocentric state. We assume these predicates should always be observable by the agent. Finally, we assume anchor objects are connected via relational predicates. Agents should be able to observe other anchor objects that are immediately related to the current set of anchor objects. These related anchor objects are used to determine where the agent need to explore, which we will discuss in a later section. Our ESE algorithm first iterates through all the relational predicates of type R to extract observable relational predicates and their anchor objects. It then extracts all the non-relational predicates that are either always observable or containing observable anchor objects. The extracted predicates define the egocentric state of the agent. The full ESE algorithm is shown in Algorithm 1.

In the grid example, the user needs to define a set of initial observable anchor objects, anchor types and relational predicates, $S = \langle T, C, R\rangle$. Anchor objects in this problem are the location objects. The relational predicates are `(conn )`. Initially, only `f0-0f` is observable by the agent. The ESE algorithm first extracts all the relational predicates containing `f0-0f`. The algorithm then finds all location objects related to the `f0-0f` which are `f0-1f` and `f1-0f`. Next, the ESE algorithm extracts non-relational predicates in initial state I that contains `f0-0f, f1-0f, f0-1f`. `(robot-at robot0 f0-0f)` is extracted as a result. Finally, all fluents contain no location objects are considered observable and are also extracted. The egocentric initial state $I^{'}$ is shown in Figure 3.

Algorithm 1: Egocentric Subset Extractor

```
   Input: $P = \langle O, F, I, A, G\rangle$
   Anchor Object Set $S = \langle T, C, R\rangle$
   Output: Egocentric projection $P^{'}$
1  Initialize $O^{'} = \emptyset$ and $I^{'} = \emptyset$;
2  for $p \in I$ where type(p) $\in R$ do
3      if $p$ has object $o \in C$ then
4          for $o \in p$ do
5              if type(o) $\in T$ then
6                  $O^{'} = O^{'} \cup \{o\}$;
7  for $p \in I$ where $type(p) \notin R$ do
8      if $p$ has $o \in O^{'}$ or $p$ is constant then
9          $I^{'} = I^{'} \cup \{p\}$;
10         for $o \in p$ do
11             $O^{'} = O^{'} \cup \{o\}$;
12 $F^{'} = F$;
13 $A^{'} = A$;
14 $G^{'} = G$;
15 return $P^{'} = \langle O^{'}, F^{'}, I^{'}, A^{'}, G^{'}\rangle$;
```

Iterative Exploration via Replanning

An egocentric agent can only observe a subset of the whole state space. Thus, a way to explore an unknown environment is often required to formulate a plan. An exploration strategy must identify which part of the state space needs to be explored while keeping track of state information observed previously. Our Iterative Exploration via Replanning (IER) algorithm is such an approach that progressively explores unknown or partially known environments. We designed IER to generate and modify existing PDDL files directly. As a result, we can use off-the-shelf planners directly.

In addition to $P^{'} = \langle O^{'}, F^{'}, I^{'}, A^{'}, G^{'}\rangle$ generated by the ESE algorithm, IER requires the user to identify an additional set of *exploration actions*, E, as input. We define exploration actions as actions that an agent needs to change its current observed state space.

By taking actions in E, an agent will transit to a state where new objects and fluents are observable.

An exploration action a must have at least one anchoring object in its parameters. Both $PRE(a)$ and $ADD(a) \cup DEL(a)$ must also have predicates containing these anchoring objects. To distinguish between observed and unobserved objects, we create `(unknown ?obj)` predicates to identify unobserved objects. For an exploration action a, `(unknown o)` predicates are added to $PRE(a)$ for each anchor object, `o`, declared in the parameters set. After the action is taken, the unknown object is deemed to be revealed and will be removed using $DEL(a)$.

When a plan with the original goal cannot be found, our algorithm sets exploration as a goal. We achieve this by introducing an `(exploration)` predicate to replace the original goal state. Since exploration is achieved through exploration actions in E, we need to add the `(exploration)` goal predicate to $ADD(a)$ for each $a \in E$. After an exploration step, a new planning problem $P^{'}$ is generated using ESE. These steps are repeated iteratively through replanning until a plan for the original goal can be

```
(:init
    (conn f0-0f f0-1f right)
    (conn f0-0f f1-0f down)
    (conn f0-1f f0-0f left)
    (conn f1-0f f0-0f up)
    (dropoff )
    (handsfree robot0)
    (move up)
    (move down)
    (move left)
    (move right)
    (pickup person0)
    (robot-at robot0 f0-0f)
    )
```

Figure 3: Initial condition extracted by ESE

found. IER is shown in detail in Algorithm 2.

In the grid example, the action required for the agent to move to another egocentric state is the `move-robot` action. IER first iterates through all the fluents in the initial condition set $I^{'}$ and identifies unknown locations by adding `(unknown f1-0f)` and `(unknown f0-1f)` fluents. To make an exploration action, IER creates a new action `explore` by adding `(unknown location)` in `move-robot`'s precondition, `(not (unknown location))` and `(explored)` predicates in its effect set. The `exploration` action is illustrated in Figure 4. Since no plan can be found for $G^{'}$, we replace the goal state with $G^{'} = \{$`(explored)`$\}$. This forces the agent to move to another location to gather more information about the broader environment. In the next iteration, the ESE algorithm will extract a new set of observable objects followed by IER until a plan with the original goal can be found or the problem is deemed unsolvable.

```
(:action exploration
    :parameters (?robot - robot ?from - location ?to
                        - location ?dir - direction)
    :precondition (and (move ?dir)
        (conn ?from ?to ?dir)
        (robot-at ?robot ?from)
        (unkown ?to))
    :effect (and
        (not (robot-at ?robot ?from))
        (robot-at ?robot ?to))
        (not (unkown ?to))
        (explored))
```

Figure 4: An exploration action example

4 Evaluation

We tested our approach empirically on five classical planning domains written in PDDL. These domains are Blocks World, Minecraft, Search-and-Rescue, Sokoban, and Elevators. In addition, we used classical planning problems de-

Algorithm 2: Iterative Exploration via Replanning

```
Input: Problem $P = \langle O, F, I, A, G \rangle$
Anchor object set $S = \langle T, C, R \rangle$
Exploration action set $E$
Output: Plan $M$ for the pansophical environment
Initialize $O_e, F_e, I_e, A_e, G_e = \emptyset$;
plan $M = [\,]$;
while no plan can be found for $P$ do
    $O_e = O$;
    $A_e = A$;
    for $o \in O$ do
        if $type(o) \in T$ and $o \notin C$ then
            $I_e = I_e \cup \{$ (unknown o) $\}$;
            for $a \in E$ do
                $a^{'} = a.copy()$;
                $PRE(a^{'}) = PRE(a^{'}) \cup$ (unknown o)
                $DEL(a^{'}) = DEL(a^{'}) \cup$ (unknown o)
                $ADD(a^{'}) = ADD(a^{'}) \cup$ (explored)
                $A_e = A_e \cup a'$
    $G_e = \{$(explored)$\}$;
    $F_e = F \cup I_e \cup G_e$;
    $\pi = \text{SOLVE}(\langle O_e, F_e, I_e, A_e, G_e \rangle)$;
    $M.extend(\pi)$;
    for $a \in \pi$ do
        if $a \in E$ then
            add anchor objects in $a$ to $C$;
    $I_e = \text{PROGRESS}(I_e, \pi)$ ;
    $G_e = G$;
    $P = \text{ESE}(\langle O_e, F_e, I_e, A_e, G_e \rangle, \langle T, C, R \rangle)$;
$M.extend(\text{SOLVE}(P))$;
return $M$;
```

fined in the PDDLGym libarary(Silver and Chitnis 2020), which contains domain and problem files written in PDDL, along with visualization APIs. For each planning problem, the user only needs to add the `(unknown ?obj)` predicate, the `(explored )` predicate, and the exploration actions. No modifications are needed for the problem file. It takes only several minutes for manual conversion if the user is already familiar with the domain and PDDL modelling in general.

For each planning domain, we tested five different problem setups. The results are summarized in Table 1. We evaluate the performance based on the percentage of successful conversions from an pansophical planning problem to the egocentric alternative. In addition, we compared the average number of steps required for the egocentric agent to solve a planning problem. We used Tarski(Ramírez and Francès 2021) to parse each planning domain, and the actual planning is done on the Planning.Domains online solver (Muise 2016): `http://solver.planning.domains`.

The results show that our method can successfully convert most classical planning problems to an egocentric version. For Search-and-Rescue, Sokoban and Blocks World, our algorithm can successfully convert 100% of the testing problems. In the case of Sokoban, we were able to covert

	Success Rate	Egocentric Plan Len	Pansophical Plan Len
Search-&-Resc.	100%	26	10
Blocks World	100%	16	11
Elevator	100%	29	22
Sokoban	75%	64	41
Minecraft	0%	NA	27

Table 1: Results of tested planning domains

four out of five problems. Unlike the other four problems, Sokoban contains irreversible actions that can result in dead-ends. The failure occurred when the agent can no longer achieve its original goal due to such actions. The only domain where our method is not applicable is Minecraft. Although there is the notion of a grid location in Minecraft, the agent can reach all locations in a single step. Our method relies on connection predicates to facilitate gradual exploration. If an agent can reach a state without passing through any other state, our method will treat both states as visible in the egocentric formulation.

5 Discussion

The experiments show that our algorithm can successfully convert the Blocks World and the Elevator domains to egocentric equivalents. This demonstrates that our approach is not limited to just agent navigation problems. Both Search-and-Rescue and Sokoban are 2D navigation problems where the notions of agent sand egocentricity are apparent. In both problems, agents are defined explicitly by the domain designer as `agent` or `robot` objects. However, an explicit definition may not always be necessary. We can define agents as the set of actions that can change the observable state. For example, in Blocks World this action set is `{stack, unstack}`. Intuitively, the agent is whomever that have the ability to move the blocks.

The location object is an obvious choice to define egocentricity in 2D navigation problems. However, defining it for Blocks World is more challenging. The set of observable states changes depending on the perspective of the observer. Our solution adopts a (literal) top-down perspective, limiting the observable state to only blocks that are on the top of stacks. We define the block object as the anchor object. Each block is connected via `(on ?block1 ?block2)` connection predicates. We then set `unstack` action as the exploration action. A block is considered explored when the `unstack` is conducted. Our IER algorithm then extracts a new block to explore using the `(unknown ?block)` predicate. The Blocks World formulation shows as long as all required inputs can be satisfied, our method can convert a planning domain without an explicit definition of an agent.

When the goal of a planning problem can no longer be reached, the problem is considered unsolvable. Our method treats exploration as a way to gather information about the state-space independent of the original goal. Thus, such exploration could lead the agent to reach a dead-end where the original goal is no longer reachable. For problems like Search-and-Rescue and Blocks World, all actions are reversible, and the initial goal will always be solvable when enough information is gathered. However, in Sokoban, the agent can reach a dead-end via irreversible actions. For example, when a block is pushed to a corner, the agent will not be able to push the block back. In our implementation, our agent does not consider the feasibility of the original goal when exploring. One can avoid such situations via dead-end checking algorithms, but only in situations where there is sufficient information available to the agent for them to reliably avoid these dead-ends.

One of the future directions we would like to take this work is applying dead-end detection techniques such as (Lipovetzky, Muise, and Geffner 2016) to avoid actions that cause dead-ends. We want to integrate the agent's original goal with exploration for more goal oriented exploration strategies. In this work, we only tested our method on classical planning domains. These domains are not representative of real life planning settings. We want to eventually apply our techniques in embodied egocentric agent design to solve planning problems such as in (Shridhar et al. 2020).

6 Related Work

Defining the environment in an egocentric manner is important to many agent based applications. Charniak demonstrates the utility of the egocentric view in reinforcement learning in a Grid World setting (2020). They show that the egocentric view improves the learning algorithm's ability to apply the learned policy to new problems never before seen during training. Zhang et al. proposed an egocentric vision-based assistive co-robot system (2013). This work allows humans to actively engage in control loops via egocentric camera and gesture input. Bertasius, Chan, and Shi presented a generative adversarial network model that use first-person images to generate a realistic basketball sequence via egocentric motion planning (2018). All these defined planning and egocentricity in their respective domain-specific settings. In contrast, our method focuses on egocentricity in the classic domain-independent planning setting.

There exists work on planning under partial observability, such as conformant, contingent, and epistemic planning. However, we distinguish egocentric planning from these partial observable planning settings with regards to the use of the belief space. Conformant can be interpreted as classical planning in belief space, and contingent planning adds uncertainty to the observable state space and can be formulated as and-or search problems in the belief space (Bonet and Geffner 2000; Hoffmann and Brafman 2005; 2006). Epistemic planning represents belief states as epistemic states which can be reasoned using epistemic logic (Bolander 2017). Planners constructed for all of these partially observable planning settings often require explicitly defining the possible initial belief space. This requires the users of these planners to have knowledge of all possible objects that will be present in the planning problem, *in advance*. However, in many planning problems, an acting agent might not have access to all unique objects in advance, and new objects and relationships need to be discovered during exploration. For example, an egocentric agent that is navigating in a 2D grid might not know all possible "location" objects to formulate a proper belief state. The approach we take here is to iteratively convert information the agent has observed as a fully observable planning problem to determine whether the original goal can be reached or more exploration is needed. However, the fully observable planner we used can be replaced by conformant, contingent, or epistemic planners when dealing with uncertainties with agents, actions, or the environment. That is to say, those areas of planning under partial observability are *complementary* to our work, and may be incorporated in future work.

Some previous works have studied planning in open worlds, a similar class of problems to the egocentric problems we define in this paper. Talamadupula et al. used a hindsight optimization method to solve planning problems in partially observable worlds (2010). The proposed method uses a prior distribution to generate and aggregate samples of close-world problems that can be solved using an off-the-shelf planner. Kiesel et al. proposed a novel partial-satisfaction goal construct that allows predefined objects to be discovered via replanning (2012). James, Rosman, and Konidaris presented a framework that derives egocentric views of planning problems for the purpose of learning portable representations, sufficient for planning, of classes of tasks (2019). These representations are learned from traces of actions and transitions. Our work, however, derives these views from the planning problem's pansophical description. The advantage is our approach does not require training data. Jiang et al. introduces a method to identify and reason over objects in an environment via a database (2019). The approach converts open world problems to close world settings via hypothetical instances of unknown objects. All these works adopted assumptions about open-world problems that we expose via our approach to deriving egocentric problems. However, these problems require complete reformulation of close world planning domains via their respective specifications. To the best of our knowledge, we are the first to propose a method to semi-automatically convert classical planning problems into an egocentric alternative with relative ease.

7 Summary

In this work, we have proposed an open-loop replanning method for converting classical pansophical planning problems into egocentric alternatives. Our method is semi-automated and requires very little change to the original problem. Our approach consists of an Egocentric Subset Extractor to extract the observable state space of an agent. It also contains the Iterative Exploration via Replanning algorithm that facilitates exploration in an unknown environment. We tested our method on five classical planning problems and converted most of these problems to their corresponding egocentric versions. The results also demonstrate that our approach is not limited to grid navigation problems such as Search-and-Rescue and Sokoban. It can convert problems, such as Blocks World and Elevator, where the notion of the agent is not explicitly defined. Our work serves as a crucial first step towards embodied agents that can be equipped with an appropriately specified egocentric version of known environment dynamics.

References

Bertasius, G.; Chan, A.; and Shi, J. 2018. Egocentric basketball motion planning from a single first-person image. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, 5889–5898. IEEE Computer Society.

Bolander, T. 2017. A gentle introduction to epistemic planning: The DEL approach. In Ghosh, S., and Ramanujam, R., eds., *Proceedings of the Ninth Workshop on Methods for Modalities, M4M@ICLA 2017, Indian Institute of Technology, Kanpur, India, 8th to 10th January 2017*, volume 243 of *EPTCS*, 1–22.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In Chien, S. A.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, 52–61. AAAI.

Charniak, E. 2020. Extrapolation in gridworld markov-decision processes. *CoRR*.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Hoffmann, J., and Brafman, R. 2005. Contingent planning via heuristic forward search with implicit belief states. In *Proc. ICAPS*, volume 2005.

Hoffmann, J., and Brafman, R. I. 2006. Conformant planning via heuristic forward search: A new approach. *Artif. Intell.* 170(6-7):507–541.

James, S.; Rosman, B.; and Konidaris, G. D. 2019. Learning portable representations for high-level planning. *CoRR*.

Jiang, Y.; Walker, N.; Hart, J. W.; and Stone, P. 2019. Open-world reasoning for service robots. In Benton, J.; Lipovetzky, N.; Onaindia, E.; Smith, D. E.; and Srivastava, S., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, 725–733. AAAI Press.

Kiesel, S.; Burns, E.; Ruml, W.; Benton, J.; and Kreinmendahl, F. 2012. Open World Planning via Hindsight Optimization University of New Hampshire. Technical report, University of New Hampshire.

Lipovetzky, N.; Muise, C.; and Geffner, H. 2016. Traps, invariants, and dead-ends. In *The 26th International Conference on Automated Planning and Scheduling*.

Muise, C. 2016. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*.

Ramírez, M., and Francès, G. 2021. Tarski. Accessed on 2021-05-17.

Shridhar, M.; Thomason, J.; Gordon, D.; Bisk, Y.; Han, W.; Mottaghi, R.; Zettlemoyer, L.; and Fox, D. 2020. ALFRED:

A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, 10737–10746. IEEE.

Silver, T., and Chitnis, R. 2020. Pddlgym: Gym environments from PDDL problems. *CoRR* abs/2002.06432.

Talamadupula, K.; Benton, J.; Kambhampati, S.; Schermerhorn, P. W.; and Scheutz, M. 2010. Planning for human-robot teaming in open worlds. *ACM Trans. Intell. Syst. Technol.* 1(2):14:1–14:24.

Teichteil-Königsbuch, F., and Fabiani, P. 2007. A multi-thread decisional architecture for real-time planning under uncertainty. In *3rd ICAPS'07 Workshop on Planning and Plan Execution for Real-World Systems*.

Zhang, J.; Zhuang, L.; Wang, Y.; Zhou, Y.; Meng, Y.; and Hua, G. 2013. An egocentric vision based assistive co-robot. In *IEEE 13th International Conference on Rehabilitation Robotics, ICORR 2013, Seattle, WA, USA, June 24-26, 2013*, 1–7. IEEE.

Lila: Optimal Dispatching in Probabilistic Temporal Networks using Monte Carlo Tree Search

Michael Saint-Guillain[1], **Tiago S. Vaquero**[2], **Steve A. Chien**[2]
[1] Université catholique de Louvain, Place de l'Universite 1, 1348 Ottignies-Louvain-la-Neuve, Belgium
[2] Jet Propulsion Laboratory, California Institute Technology, 4800 Oak Grove Dr, Pasadena, CA 91109 USA
michael.saint@uclouvain.be, {tiago.stegun.vaquero, steve.a.chien}@jpl.nasa.gov

Abstract

Executing a Probabilistic Simple Temporal Network (PSTN) amounts at scheduling, i.e. *dispatch*, a set of events under time uncertainty. This constitutes a *NP*-hard online optimization problem. The right execution time must be dynamically assigned to each event of the PSTN such that the temporal constraints are met, whereas activity durations are progressively observed as the execution unfolds. We propose a dispatching algorithm based on Monte Carlo Tree Search, called *Lila*, with the following characteristics: (i) it is an anytime algorithm, both offline and online, conjectured asymptotically optimal; (ii) it returns the current probability of success, either before or at any moment during operations; (iii) it handles any possible continuous or discrete, even non-parametric, probability distributions, as well as interdependencies between random variables, exogenous and endogenous uncertainty; and (iv) can be easily extended to handle probabilistic external events, PSTNs with resources, PSTNs with cutoff times and precondition chains, *etc.* Lila is universal in the sense that it can handle any dispatching protocol, simply by specifying it to the algorithm. It has the unlimited flexibility offered by the simulation paradigm, and is conjectured to asymptotically converge to optimal decisions and/or robustness approximations.

1 Introduction

Temporal networks formalize the arrangement and interdependencies of tasks, or activities, that compose an operational plan. In a simple temporal network (STN), activities are modelled as a finite set of time events, such as start and end times. In practice, some activity durations, considered as *contingent*, remain unknown beforehand and are revealed during execution (decided by nature). When some stochastic knowledge on the uncertain durations exists, one can model it as (estimated) probability distributions, leading to the extending concept of probabilistic STN, or *PSTN*. Solving a PSTN then amounts at finding an assignment of time values to executable events, such that assigned values together with observed ones fulfil all the constraints between events (*e.g.*, end of task A must happen between 10 and 20 minutes before the beginning of B). Whenever such assignment exists, a network is said to be *controllable*. When the operational assumptions enable it, the assignment may be *dynamically constructed*, i.e. as durations are observed, the time values are assigned. Yet, even under dynamic decision, due to unfortunate durations, a network may reveal as violating some of the temporal constraints during execution.

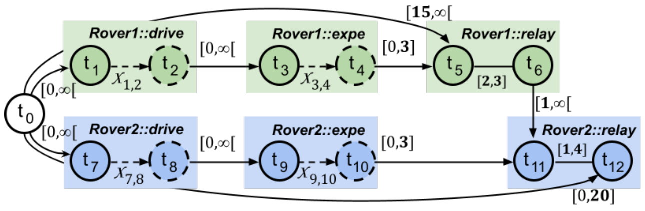


Figure 1: A simplified hypothetical sol on Mars for two planetary rovers, encoded as a PSTN. Bold: controllable. Dashed: contingent.

When there is uncertainty with respect to temporal constraint violation, it is critical to determine the actions (*i.e.* assign time values to time events) that maximize the probability of successful execution. Furthermore, at a given state of the online execution, determining the current probability of success according to past decisions and events as well as the remaining uncertainty is a major importance too. In case this probability falls below some threshold of acceptable risk, the operators may decide to interrupt the execution before it reaches a problematic or dangerous state.

Fig. 1 shows an hypothetical example of Mars rovers operations as a PSTN. Each rover has three activities in sequence: drive towards a science site, perform a science experiment, and relay results to an orbiter. A special time point $t_0 = 0$ represents the beginning of the operations. Time events are linked by temporal constraints, either controllable or contingent. The rovers work independently during their driving and science activities, and eventually coordinate for the communication time window, which strictly happens within time 15 to 20. Furthermore, a maximum of 3 time units is authorized from the end of experiments to the start of relay activities, implying that starting everything as soon as possible may be problematic. The duration of the driving and science activities are uncertain, and encoded in the PSTN as contingent constraints described by probability distributions. Ideally, a perfect assignment of all executable time points would work for any situation imposed by nature. In practice that is very restrictive, if not impossible, especially in highly uncertain environments. It raises the following questions. What is the probability that we succeed at executing the PSTN, namely that our rovers both meet the communication window? How to compute online decisions in an optimal way, such that we maximize this probability?

Contributions. We describe how the MCTS framework can be used in the context of scheduling time points in constrained temporal networks under uncertainty, namely PSTN dispatching. To the best of our knowledge, this is the first application of MCTS to temporal networks. Unlike mathematical approaches, or even those based on pure Monte Carlo simulation, our method is an anytime algorithm; it is conjectures as asymptotically optimal (we let the demonstration for future work), and allows to consider various extensions to the classical PSTNs with minor adaptations. Moreover, MCTS allows to simply handle very complicated concepts, some of them being described in this section, such as dependent random variables or even endogenous uncertainty (which is a topic rarely covered in the literature). A preliminary experimental analysis is conducted.

2 Probabilistic Simple Temporal Networks

Simple Temporal Network is a popular formalism for temporal constraint reasoning (Dechter, Meiri, and Pearl 1991), framed as a constraint satisfaction problem over time point variables: a STN is a tuple $\langle T, C\rangle$, where T is a set of time points and C is a set of constraints $c(t_i, t_j)$ that encode bounds on the differences between pairs of time points: $l_{ij} \leq (t_j - t_i) \leq u_{ij}$, *i.e.* $(t_j - t_i) \in [l_{ij}, u_{ij}]$. The goal is then to assign time values to every time points, such that all the $t_j - t_i$ duration constraints are respected.

Most realistic operational contexts account for temporal uncertainty. **PSTN** is a natural extension of STN in which probability density functions are associated to temporal constraints, such as activity durations (Tsamardinos 2002). In a PSTN, the *executable* time points T_E are determined by the agent, and *contingent* time points T_C are assigned by nature. A solution is called a *schedule*, a specific assignment to all $t_i \in T_E$. Given a particular realization of T_C, a schedule is *consistent* if it satisfies all the constraints of the network. In practice a contingent duration is described by a (usually estimated) probability distribution $(t_j - t_i) = X_{i,j}$.

In practice, a time point in the PSTN often stands for either the start (*e.g.* t_3 in Fig. 1) or the end (t_4) of a particular *activity* (*Rover1::expe*). The starting point of activities usually constitute the set of executable time points T_E. A schedule determines the execution time of $t_j \in T_E$, and requirement constraints in the form $c(t_i, t_j)$ state how late t_j can occur regarding to any previous time points t_i. When $t_j \in T_C$, which could represent an activity completion, the duration $(t_j - t_i)$ remains unknown prior to execution.

Policies and Dispatching protocol. Operational contexts such as space missions usually pose computational and power limitations on recomputing a schedule in the middle of the operations (Chi et al. 2019). Yet, the use of a static schedule is often either impossible in practice, or comes with a significant waste in terms of operational yield and time. Such approach is currently operating *Curiosity* rover, with static schedules that overestimate processing times by 30% in average (Gaines et al. 2016) to account to execution uncertainty. Let Ω be the set of all possible realizations of the random contingent edges' duration in the PSTN. A trivial approach to avoid both static scheduling and online reoptimization is to precompute particular schedules for each possible situation that may arise, leading to a *policy*. Naturally, the size of Ω is usually problematic. Instead, Perseverance (M2020) rover is equipped with a non-backtracking onboard scheduler, designed to take online decisions based on current observations (Rabideau and Benowitz 2017; Chi et al. 2018; Agrawal et al. 2021a,b). Due to computational limitations, such online decisions must remain very light, thus following a predefined *strategy*: a *dispatching protocol* (DP). In particular, a DP usually aims at avoiding costly online reoptimizations. For example, Rabideau and Benowitz (2017) describe an average $\mathcal{O}(n^2)$ quadratic DP$(\cdot)$ protocol to be computed by the onboard scheduler in the Mars Perseverance rover, in order to adapt decisions online (*i.e.* $\Gamma_E^t = T_E^t$) based on observations and pre-optimized parameters (Chi et al. 2019).

***NextFirst* dispatching protocol.** The *NextFirst* protocol (Brooks et al. 2015), also known as *DC-dispatch* (Morris, Muscettola, and Vidal 2001) or *early execution* (Lund et al. 2017), dynamically assigns a value to and dispatches each time point (*i.e.* executes the PSTN) in $\mathcal{O}(n)$ linear time, by starting activities as soon as possible. Let t_j be a controllable time point in a PSTN, and $I_j = \{(0, j), \ldots, (i, j)\}$ the set of incoming edges in t_j. Therefore, t_j is assigned a time value as soon as all the preconditions are validated, that is, all the $t_0, \ldots, t_i$ time points are known, leading to the very simple online decision rule:

$$t_j = \max(t_0{+}l_{0j},\ \ldots\ ,t_i{+}l_{ij}). \qquad (1)$$

In the case $t_j > \min(t_0{+}u_{0j},\ \ldots\ ,t_i{+}u_{ij})$, the dynamic execution is interrupted and considered as failed. Naturally, *NextFirst* protocol has linear complexity $\mathcal{O}(n)$. Back to our PSTN example in Fig. 1, the value of t_{11} is then dynamically set to $\max(t_{10},\ t_6)$ as soon as tasks *Rover2:expe* and *Rover1:relay* are completed. Execution fails if t_{11} exceeds $t_{10} + 3$. Eventually, we hope for $t_{12} \leq 20$.

Formulation of the optimal dispatching problem

Assumptions and notations. We assume the operational time horizon to be partitioned in h outcome and decision stages. The random vector $\xi = \xi^1, \ldots, \xi^h$, with support Ω, describe all the possible sequences of outcomes. When necessary, we designate by $\xi^{t..t'}$ the sequence of outcomes of scenario ξ from time t to time t'. The decisions to be taken by the online scheduler during the operations are represented by a vector $\mathbf{x} = x^1, \ldots, x^h$ of $\mathbb{R}^h$, from which the schedule can be trivially deduced. We refer to decisions $x^t, \ldots, x^{t'}$ as $x^{t..t'}$. The indicator function $V^{\mathbf{x}}(N, \xi)$ returns 1 iff the schedule $\mathbf{x}$ is consistent in scenario ξ. Operator $E_{\xi^t}[\ \cdot\ ]$ designates the expectation over random variable ξ^t, conditionally to history $\xi^{1..t-1}$. In case of endogenous uncertainty,

$E_{\xi^t}[\ \cdot\ ]$ also depends on decisions $x^{1..t-1}$. Finally, X^t represents the set of legal actions (*i.e.* time assignments) at time t, which naturally depends on past actions $x^{1..t-1}$ and history $\xi^{1..t}$.

Multistage stochastic formulation. An optimal dispatching protocol necessary computes, in any possible situation (*i.e.* given any possible past realizations and decisions), the decisions that maximizes the probability that the current partial schedule completes to a consistent schedule. The following multistage stochastic program determines the optimal dispatching decisions x^t at a current time t:

$$\underset{x^t \in X^t}{\text{argmax}}\ E_{\xi^{t+1}}\Big[\max_{x^{t+1}\in X^{t+1}} E_{\xi^{t+2}} \Big[\dots \max_{x^{h-1}\in X^{h-1}} E_{\xi^h}\Big[\max_{x^h \in X^h} V^{x^{1..h}}(N,\xi)\Big]\dots\Big]\Big] \quad (2)$$

where the maximum value of the first expectation, when $t = 0$, is by definition equal to the Degree of Dynamic Controllability (DDC) of the network (Saint-Guillain et al. 2020, 2021), the probability of success under perfect reoptimization. Consequently, this must be at least equal to the probability of success under the *NextFirst* protocol.

The nested expectations in (2) form a tree structure, well known as the *scenario tree*. Unfolding the maximization operators as well leads to a full decision-scenario tree as illustrated in Fig. 2. Each path of the tree constitutes a possible scenario realization together with associated decisions, a sequence $\xi^t, x^t, \dots, \xi^h, t^h$. At time t, decisions x^t depend on the current history $\xi^{1..t}$ and maximize the expected value $E_{\xi^{t+1}}[\max_{x^{t+1}} \dots]$ of the future decisions at time $t+1$ given the remaining uncertainty, and so on until time h is reached.

A node ξ^t hence represents a particular state, defined by history $\xi^{1..t}$ and past decisions $x^{1..t-1}$. Counter-intuitively, we call node ξ^t a *decision node*. This is because at this particular state, a decision x^t must be chosen amongst X^t. Following (2), x^t is necessarily the decision maximizing the probability that the partial schedule $x^{1..t}$ extends to a consistent full schedule.

A node x^t represents a decision that has already been chosen for time t. Since it directly leads to nodes representing the possible realizations of ξ^{t+1}, we call x^t a *chance node*. Still following (2), the value of the decision x^t is given by the expected value of the subsequent (and consequent) realizations.

3 Monte Carlo Tree Search

Solving problem (2) is computationally intractable in practice. Yet, a look at the associated tree immediately suggests two classical approximation schemes: (a) limiting the branching factor and (b) avoiding to consider less relevant subtrees. Both approaches are compatible, and with a few additional techniques described in this section, we will end up with an adaptation of the well-known Monte Carlo Tree Search (MCTS) algorithm to our PSTNs.

Limiting the branching factor. This can be achieved by sampling a restricted number of children generated from chance and/or decision nodes. At a decision node, a limited number of children $x_a^t, x_b^t, \dots$ could either be chosen in X^t at random, or by following some predefined strategy (a dispatching protocol!). At a time t chance node, a limited sample of random realizations of ξ^{t+1} may constitute a pertinent restricted set of children nodes. In particular, even if MCTS aims at dealing with large branching factors, this step is still mandatory to obtain a discrete tree from the decision and realization domains, which are continuous by nature.

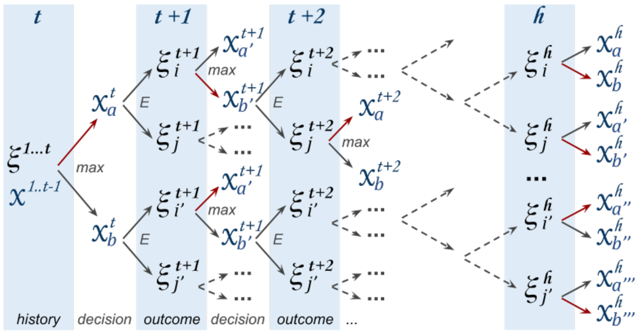


Figure 2: Tree structure of the problem. The root node represents the current state (past decisions and realizations) at time t. For simplicity, decision (*resp.* random) variables have only two possible choices (*resp.* outcomes).

Subtrees prioritization. While a huge part of the whole tree depicted in Fig. 2 is already pruned by the simple action of restricting the branching factor, the resulting tree is likely to remain too big to be entirely explored, and further decreasing the branching factor may result in missing critical decisions or outcomes. This is where MTCS comes at hand. In fact, MCTS selects the most promising nodes (*i.e.* subtrees) to consider first by using a node value function, which exploits a Monte Carlo sampling paradigm to approximate and eventually evaluate the interest in visiting a subtree.

General MCTS approach and related work

MCTS is a general framework, which has already been well described by the literature. The reader interested in a complete description may refer to Browne et al.'s survey (2012), Section III. The general MCTS algorithm can be outlined as:

MCTS keeps iteratively performing each of the following four steps in turn, until the computation budget is reached: **1) select** an expandable (non final) node; **2) expand** the node by generating one (or more) of its child nodes; **3) simulate** one path down the current tree, in order to reach a final state from the newly created child node, without creating any new node, but instead obtain a final state as quickly as possible; **4) backpropagate** the final state value R, updating the estimated value $V(\text{node})$ of each node along the path from the new to the root node. Finally, the playing action represented by the best root child is returned once the computational budget is exhausted. An example of search tree being gradually built using MCTS's four steps is depicted in Figure 3. Some details of the tree may appear mysterious at

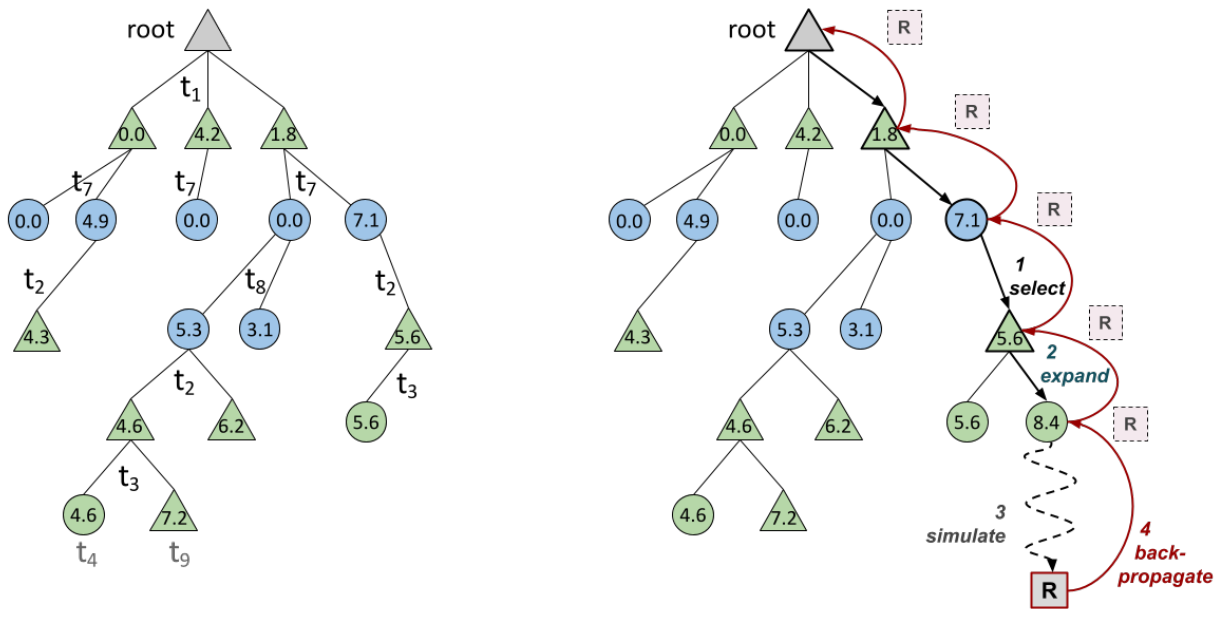


Figure 3: MCTS tree and iteration steps. △ Decision node. ○ Chance node. □ Terminal node.

Algorithm 1: General MCTS.

```
1  root ← CreateRootNode();
2  while computation budget not exhausted do
3      node ← root;
4      while not node.IsExpandable() do
5          node ← node.SelectChild();
6      child ← node.ExpandOne();
7      if child.IsTerminal() then
8          child.BackPropagate(child.Evaluate());
9      else
10         for $i$ ← 1 to $n_{sim}$ do
11             child.BackPropagate(child.Simulate());
12 return root.BestChild();
```

this time, and will be clarified as we describe how we adapt MCTS specifically for taking decisions in PSTNs.

How to select which node to expand, and therefore where to grow the tree, is at the heart of MCTS and raises the question of intensification versus diversification. Whereas a selection policy based on intensification only is likely to end up exploring a very narrow, specific, deep subtree, the opposite pure diversification would result in simple breadth-first search. The most popular selection policy, called Upper Confidence Bounds for Trees (UCT), makes its path from the root node down to a leaf node to be expanded (*i.e.* not a final game state) by diving through child nodes (lines 3-5 of Algorithm 1), where ***node.SelectChild()*** maximizes

$$UCT = V(\text{child}) + 2C\sqrt{\frac{2\ln n^{\text{node}}}{n^{\text{child}}}}$$

where the first term $V(\text{child})$ is the estimated value of the child node and therefore encourages intensification. The second term, with n^{node} being the number of times a node or one of its descendants ran a simulation, encourages diversification by promoting children being less visited than their sibling nodes. In fact, a never visited child will be given ∞ value. The C parameter is usually empirically tuned to best calibrate both terms. ***root.BestChild()*** classically returns the root child maximizing either $V(\text{child})$ or n^{child}. Remark that whereas $V^{x^{1..h}}(N, \xi)$ in Eq. (2) indicates whether a leaf node of the decision/outcome tree is a win or a loss state, here $V(\text{node})$ approximate the expected success value of a terminal node belonging to the subtree defined by $node$.

MCTS with continuous action and outcome spaces. The classical MCTS method has been developed for deterministic zero-sum games. Like stochastic games that involve rolling a dice, our PSTNs involve uncertainty as the contingent duration of some activity remains unknown until one actually tries to execute it. Naturally, MCTS have been adapted to deal with dice rolls, and variants have been proposed (Browne et al. 2012; Cowling, Powley, and Whitehouse 2012). However, most studies focus on partially observable states, such as hidden cards in Poker, rather than uncertainty. As PSTNs usually involve a continuous time dimension, both realization outcomes and action decisions must generally be chosen out of continuous domains. Different methods have already been proposed in order to obtain a discrete search space compatible with MCTS, namely to implement functions ***node.isExpandable()*** and ***node.ExpandOne()***. Amongst the proposed approaches, many are based on sampling a limited set of actions and/or outcomes (Kearns, Mansour, and Ng 2002), that progressively grows as the associated node is being visited (Chaslot et al. 2008; Couëtoux et al. 2011), a technique called *progressive widening*. In fact, *Lila* exploits the progressive widening technique. Back to the ***node.SelectChild()*** function, Yee, Lisỳ, and Bowling (2016) proposed KR-UCT, an alternative to UCT based on kernel regression, to better select and further share information between actions sampled from continuous domains. They apply KR-UCT on the remarkable problem of autonomous agents playing *curling*.

Dispatching a PSTN: a single-player game against Nature

In a sense, our approach suggests to model our PSTN as a single-player game against Nature. At a given state, the player's actions aim at choosing whether, for each of the time events that are ready for execution, to schedule them immediately or postpone. Depending on the player's decisions, dice rolls will then be used to represent the possible random completion times of each started activity. Ideally, the search tree being iteratively built by the MCTS algorithm should directly approximate the full decision-scenario tree depicted in Fig. 2. In practice however, depending on the live decisions and outcomes, most of the time units in $t..h$ involve no decision nor outcome.

An equivalent, *event-driven* tree can be obtained by simply skipping all "empty" time units. Furthermore, this approach permits to get rid of discrete time units and to handle a continuous time horizon. The MCTS tree depicted in Fig. 3 gives an example of such construction, for our rover PSTN example of Fig. 1. Here, the execution has not yet started (*i.e.* current real time is zero) and MCTS is used to approximate the full decision-scenario tree of our PSTN, beginning with the decisions of when to schedule events t_0 and t_7. Recall that decision nodes ($\triangle$) stand for a state where the upcoming action is a decision, and chance nodes ($\bigcirc$) states involve a pending random outcome.

A path of our MCTS tree does not simply alternate decision and chance nodes. In fact, the nature of the node, and even the action being played, depends on the history (path) rather than its depth. For instance, the node $t_3 = 4.6$ at the bottom left is a chance node ($\bigcirc$), whereas its sibling $t_3 = 7.2$ is a decision node ($\triangle$). This is because in both cases the history is $t_1 = 1.8, t_7 = 0, t_8 = 5.3, t_2 = 4.6$. If t_3 is to be scheduled directly after t_2 at time 4.6, then the upcoming event is the random outcome t_4, since t_8 is only at 5.3. If on the contrary t_3 is delayed to 7.2, then the next event concerns the action of deciding for t_9.

Proposed PSTN-specific MCTS instantiation

We now describe how we currently propose to instantiate Algorithm 1 in order to obtain a PSTN (online), asymptotically optimal, dispatching algorithm.

node.IsExpandable() When executing a PSTN, both decisions and outcomes must be selected from (continuous) infinite domains. Similarly to Kearns, Mansour, and Ng (2002), we experiment a fixed-size branching factor at both chance and decision nodes. Therefore, the function returns true iff the predefined size is not yet reached. We also consider a more elaborated strategy, designing *node.IsExpandable()* such that the branching factor of a node progressively increases (Chaslot et al. 2008; Couëtoux et al. 2011), hence allowing the asymptotic completeness of the algorithm, that is, the optimally robust decisions at defined by Eq. (2). The branching factor of each node then depends on how many times it has been visited, and follows βn^{α}, with $n = n^{\text{node}}$.

node.ExpandOne() At a chance node, a child is created simply by sampling the associated random variable following its own probability distribution. At a decision node, different approaches may be considered when selecting an execution time. In order to eventually converge to a complete search tree, any relevant execution time should be possibly chosen. In this paper, the first generated child is executed as soon as possible according to the PSTN lower bounding time constraints, without any delay. Any other child gets assigned an execution times randomly sampled, with a probability that decreases with the delay. This is achieved by taking the absolutes values from a normal distribution centred at delay 0.

Note that if the decision nodes are limited to have only one child, then the MTCS tree will naturally converge to represent the behavior of the PSTN under the *NextFirst* dispatching protocol, which consists in executing each time point as soon as possible. In theory, the *node.ExpandOne()* can therefore be implemented in order to represent any computable dispatching protocol in MCTS. However, in order to get completeness, any admissible execution delay should be eventually considered.

node.isTerminal() A node is terminal as soon as either all the time points of the PSTN (either associate to chance or decision nodes) have been attributed a value, or if the value of some time point is not consistent with the PSTN temporal constraints. In practice, an inconsistency may be detected earlier, by comparing the current time assignments with the remaining possible future decisions and outcomes, therefore allowing to avoid further exploring a subtree which can be proven to be inconsistent. For that we refer to established theoretical results on PSTN controllability checking, and leave the related improvements for future work.

node.Simulate() The classical MCTS approach for simulating the remaining decisions and outcomes would consist in sampling everything at random, until a final state is reached. In the specific context of PSTNs however, it has been observed that executing time points as soon as possible provides good results in general (Saint-Guillain et al. 2020). Therefore, our approach is to follow the *NextFirst* dispatching protocol during simulations.

node.BackPropagate() Once the simulation hits a terminal state, its success value (0 or 1) must be backpropagated from the child node that initiated the simulation, up to the root node, thereby updating the estimated values of all the nodes along that path. The "expectiminimax" rule (Melkó and Nagy 2007) is applied, yet adapted to a single player stochastic game: if the node n is terminal, then its value $V(n)$ is equal to the success value; if it is a decision node, then its value is updated to $V(n) = \max_{c \in \text{Children}} V(c)$; if it is a chance node, then $V(n) = \frac{1}{|\text{Children}|} \sum_{c \in \text{Children}} V(c)$. Remark that this rule eventually converges to the optimal decision/outcome tree formulated in Equation (2).

root.BestChild() At a current time t, the root node is usually a decision node. Once the computational budget is reached, we are interested in the best possible decisions, below the root node. As shown in Fig. 3 however, a path of our MCTS tree does not simply alternate decision and chance nodes. If, as for the PSTN of Fig. 1 and the associated MCTS tree of Fig. 3, two decisions (t_1 and t_7) follow the root node, then the best two decisions must be returned. In our example, *BestChild()* is called at root node to select the best direct child for t_1, and then is called in turn on that child to select the best subsequent decision for t_7. Amongst the possible choices, we simply select the child maximizing $V(n)$.

4 The Versatility of the Simulation Paradigm

A basic Monte Carlo simulation could not converge to optimal decisions, because any simulation requires to follow a predefined, often simplistic, execution strategy (dispatching protocol) such as *NextFirst* — otherwise each simulation amounts at solving the NP-hard multistage optimization problem, which the simulation aims at approximating! In other words, running a Monte Carlo simulation forever simply converges to the expected value of the predefined strategy. On the contrary, Monte Carlo Tree Search combines Monte Carlo simulation with the construction of a complete, optimal, decision tree, which ultimately (*i.e.* asymptotically) converges to the true optimal decisions. Thanks to the simulation side, MCTS allows to simply handle very complicated concepts, some of them being described in this section, such as dependent random variables or even endogenous uncertainty (which is a topic rarely covered in the literature).

Maximizing the Expected Utility: dealing with Cutoffs and Precondition chains

A classical PSTN assumption is that the PSTN execution fails as soon as an activity is failed at being executed within the time constraints. In Saint-Guillain et al. (2020), we proposed PSTNs alternative execution assumptions, in which activities can be safely interrupted, using a predefined deterministic *cutoff time* or *duration*, hence allowing the execution to continue. In our rover example, this could be true for any experimental activity which are somewhat isolated.

Nonetheless, interrupting an activity may however turn impossible to carry out a related subset of remaining ones, such as for example, an experiment composed of several tasks. In our example of Fig. 1, interrupting a driving activity would necessarily compromise the associated experiment, although it does not prevent from further relaying. In other words, the driving activity is a *precondition* for the subsequent experiment. In practice, precondition chains may span over multiple subsequent activities: if a task C depends on successful execution of B, which in turn depends on a task A, then interrupting A would prevent from executing both B and C.

Yet, all activities do not necessarily have the same priority, and some may even be considered mandatory (uninterruptible), such as the relay activities in our example. A utility value can therefore be assigned to interruptible activities, leading to the objective of maximizing the overall *expected utility* of the network, that is, the expected sum of the task utilities that can be successfully dispatched, whereas failing at dispatching a mandatory activity results in a zero utility. More generally, mandatory tasks may be assigned a very high utility value *w.r.t.* interruptible ones.

MCTS with Utility, Cutoffs and Preconditions. Our PSTN specific instantiation of the MCTS framework is able to deal with these new concepts with a few straightforward adaptations. The *node.IsTerminal()* and *node.ExpandOne()* functions are impacted. A node is then terminal when either an inconsistency is detected (or if it reached its predefined cutoff), or when all time points have been attributed a value. At deciding for the execution time of a task in *node.ExpandOne()* function, the resulting time then never exceeds the predefined cutoff. Furthermore, depending on the history, the task at stake will not be executed (*i.e.* assigned duration zero) if some of its preconditions is not met, such as a past required activity that has been interrupted by hitting its cutoff (or not executed for similar reasons). Finally, the computation of $V(n)$ at a terminal node is not zero or one anymore, but the sum (over the MCTS tree path) of the utilities of the tasks that did not hit their cutoff time, or zero if some mandatory task did. The resulting $V(n)$ is then backpropagated the exact same way as aforementioned. Eventually and following Eq. (2), the average value at $V(root)$ will necessarily converge to the *expected total PSTN utility*.

Dealing with resources and exotic probabilities. Other PSTN extensions, such as resource usage, can be handled by adapting the *node.ExpandOne()* function, assuming that *node.Simulate()* uses the same mechanism to sample decisions and outcomes. One just need to save the current resource usage state, such as energy consumption, at each node of the MCTS tree and deduce, at the current decision node, the possible children accordingly. When it comes to chance nodes, because the children are simply randomly sampled from each random variable distribution, any possible distribution may be considered. In fact, considering the current history of decisions and outcomes at a certain chance node, dealing with dependent random variables (*i.e.* possible outcomes being influenced by past realizations), as well as endogenous uncertainty (*i.e.* influenced by past decisions) becomes just a matter of implementing the *node.ExpandOne()* function.

5 Experimental analysis and validations

The rover PSTN example of Fig. 1 constitutes an interesting benchmark for analyzing the behavior of the proposed framework. Since computing the probability of success of the optimal decisions (*i.e.* the Degree of Dynamic Controllability as defined in Saint-Guillain et al., 2020) is intractable,

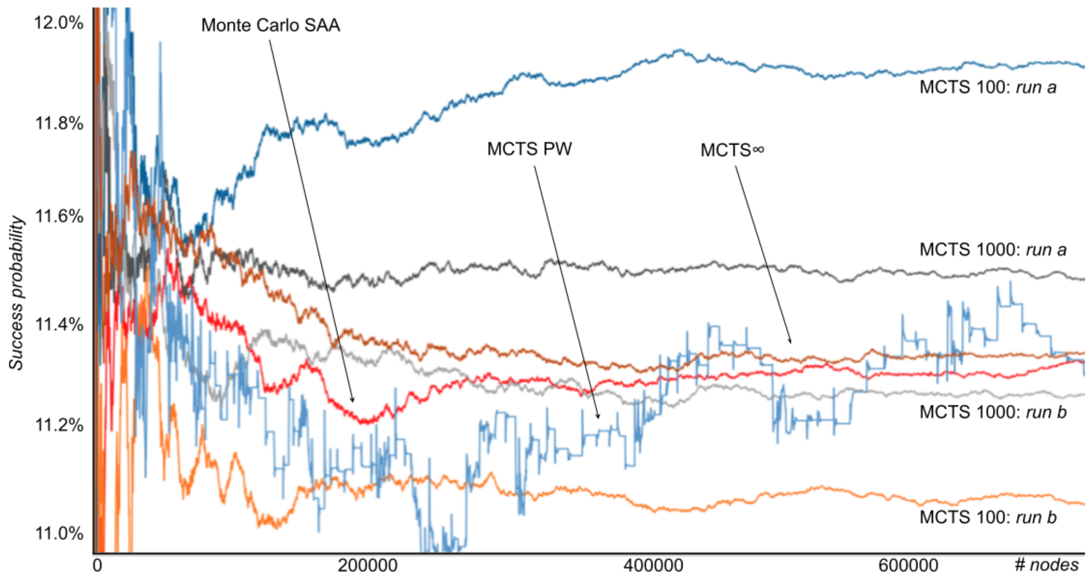


Figure 4: Evolution of the $V(root)$ estimated root node value, which predicts the robustness of the network under *NextFirst*, since decision nodes are limited to one child, for PSTN example of Fig. 1. A basic Monte Carlo simulation, in red, converges to the true robustness value.

there is the need for approximation methods such as the one proposed here. In this case, we must rely on other indicators to empirically validate our method. We then first consider the robustness of the PSTN under the *NextFirst* dispatching protocol, which can be either computed exactly, or approximated with an arbitrary precision using Sample Average Approximation (SAA), namely pure Monte Carlo simulation. Thereafter, more elaborated dispatching decisions will be considered by allowing Lila to try postponing the beginning of PSTN activities, leading to an approximation of the true probability of success under perfect reoptimization.

Implementing NextFirst dispatching protocol. Given some parametrization that restricts Lila's decisions to follow *NextFirst* protocol, the value of $V(root)$ node should converge to the robustness under *NextFirst*. We hence limit the number of decision node children to one, while varying the number of chance node children. As explained with *node.ExpandOne()* function, the resulting MCTS tree should then approximate the behavior of the PSTN under *NextFirst*.

Figure 4 shows how our $V(root)$ estimated root node value converges compared to the the success rate measured by SAA (Monte Carlo), as the number of iterations (*i.e.* nodes for MCTS) increases for the PSTN depicted in Fig. 1. A well-known issue of MCTS, when limiting the number of chance node children (or also decision nodes in general), is that the first nodes (in terms of depth) of the tree impose a strong bias. A a consequence, the entire tree and therefore the estimated probability of success as $V(root)$ strongly depend on the sampled durations of t_2 and t_8. In fact, the plot shows two different runs of Lila, given 100 children (MCTS 100) at each chance node, for which each run converge to a somehow inaccurate approximation of the *NextFirst* robustness (which is of $\approx$11.35%). The approximation improves as the number of chance children increases to 1000, yet the strong bias is still visible in the plot. Finally, allowing an infinite number of chance children eventually correctly converges. Note that in this case, MCTS nodes at depth 4 are never visited, as the algorithm keeps always expanding at the same chance node for t_2. The progressive widening (MCTS PW) technique, which gradually grows the maximum number of children, have been here tested on chance nodes. Given adequate parameters, empirically set to $\beta = 0.3$ and $\alpha = 0.4$ in this experiment, Lila eventually converges accurately. These 700000+ iterations require approximately 10 seconds, on an Intel Core i7 2.3GHz, 16GB 3733MHz, CLang 12.0.

Approximating optimal dispatching. We now allow the number of children at decision nodes to grow as well, by using the progressive widening technique. This eventually enables MCTS to explore more elaborated decisions than just simply dispatch everything as soon as possible. Figure 5 shows how delaying the execution of rover driving activities, represented by time points t_1 and t_7, improves the estimated probability of success. Eventually, six different five-minute runs of Lila all converge to $\sim$48.5% chances of success, where t_1 should best be delayed to time 1$\sim$3 and t_7 to time 3$\sim$4, depending on the run.

Recall that under *NextFirst* dispatching protocol, the success probability was of $\approx$ 11.35% only. We clearly observe here that *NextFirst* produces significantly suboptimal dispatching decisions (by starting each activity as soon as possible), in the specific case of the PSTN at stake, which is the one depicted in Fig. 1.

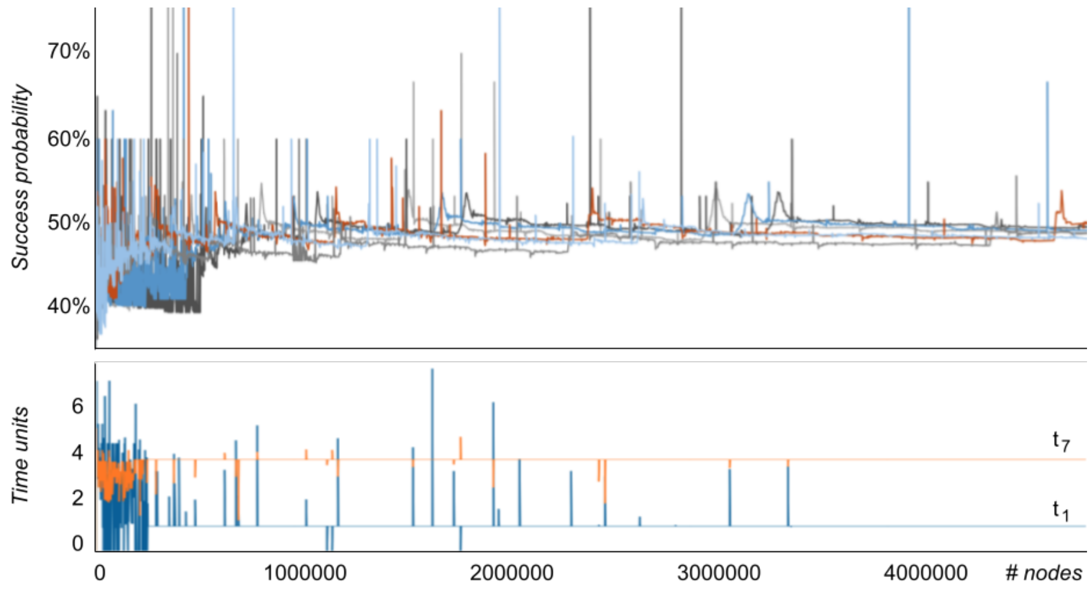


Figure 5: Top: Evolution of the $V(root)$ estimated root node value, when using progressive widening on both decision and chance nodes, for several independent runs of Lila. Bottom: Evolution of the best decisions as returned by the *root.BestChild()* function, for both t_1 and t_7, during one run of five minutes.

6 Conclusions and Future Work

We show how versatile the MCTS framework can be, when specialized to PSTN dispatching. It has the unlimited flexibility offered by the simulation paradigm. It is conjectured to asymptotically converges to optimal decisions (the demonstration is left for future work). In theory, it handles any possible continuous or discrete, even non-parametric, probability distributions, as well as inter-dependencies between random variables, exogenous as well as endogenous uncertainty. We conduct an preliminary experimental analysis, validating our algorithm, called *Lila*, on a simple PSTN example. Finally, we show how our easily algorithm may be adapted to deal with activity cutoff times and precondition chains (important in certain applications such as planetary rovers), hence providing a first solution framework to the problem of maximizing PSTN expected utility.

Future research directions.

Also related to the fact that our MCTS must deal with continuous domains, a *node.SelectChild()* function inspired from the theoretical results of Yee, Lisỳ, and Bowling 2016 may also be more appropriate than classical UCT. Finally, the literature already counts a number of improvements and extensions to classical MCTS (Browne et al. 2012), many of them being worth experimenting in the particular context of PSTN dispatching game. In addition to the aforementioned points, the PSTN extensions should be considered as well. In particular, accounting for activity resource usage, such as energy consumption, is of great interest in the context of Mars 2020 and future planetary rovers. In what follows we elaborate on additional promising directions.

Offline problem: DDC approximation. The experimental analysis conducted in this preliminary research is focused on only one PSTN instance, namely that of Fig. 1, allowing interesting insights on the algorithm behavior. A more comprehensive experimental study should be conducted on well known PSTN benchmarks, such as those recently considered for the *a priori* problem of approximating a PSTN robustness, or degree of dynamic controllability (DDC).

Online dispatching. This preliminary research includes an experimental analysis which considers the problem of evaluating the *a priori* PSTN robustness, but does not yet include online dispatching. A direct extension of this work is therefore to integrate *Lila* in an online dispatching simulator, in order to test its online reoptimization capabilities on well-known PSTN benchmarks. Since our algorithm is based on the MCTS framework which aims at dealing with fundamentally online problems, this should come with very little updates.

Proof of Concept: Mars 2020 planetary rover. How to define adequate cutoffs for M2020 task networks currently constitutes a real issue. We will i) evaluate the use of Lila to approximate the true DDC and expected utility of M2020 task networks, while considering cutoffs and precondition chains, and ii) will further try adapt the predefined cutoff times of some or all activities as part of the decisions, in order to maximize the success probability or the expected utility of the PSTNs. The latter (ii) is however a very hard problem. We will also consider iii) PSTN with resources (*e.g.* energy consumption) in addition to cutoffs and preconditions chains.

Acknowledgements

The research was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

References

Agrawal, J.; Chi, W.; Chien, S.; Rabideau, G.; Gaines, D.; and Kuhn, S. 2021a. Analyzing the effectiveness of rescheduling and Flexible Execution methods to address uncertainty in execution duration for a planetary rover. *Robotics and Autonomous Systems* 140: 103758.

Agrawal, J.; Chi, W.; Chien, S.; Rabideau, G.; Kuhn, S.; Gaines, D.; Vaquero, T.; and Bhaskaran, S. 2021b. Enabling limited resource-bounded disjunction in scheduling. *Journal of Aerospace Information Systems* 1–11.

Brooks, J.; Reed, E.; Gruver, A.; and Jr, J. C. B. 2015. Robustness in Probabilistic Temporal Planning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 3239–3246.

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.

Chaslot, G. M. J.; Winands, M. H.; HERIK, H. J. V. D.; Uiterwijk, J. W.; and Bouzy, B. 2008. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation* 4(03): 343–357.

Chi, W.; Agrawal, J.; Chien, S.; Fosse, E.; and Guduri, U. 2019. Optimizing Parameters for Uncertain Execution and Rescheduling Robustness. In *29th International Conference on Automated Planning and Scheduling (ICAPS)*.

Chi, W.; Chien, S.; Agrawal, J.; Rabideau, G.; Benowitz, E.; Gaines, D.; Fosse, E.; Kuhn, S.; and Biehl, J. 2018. Embedding a Scheduler in Execution for a Planetary Rover. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*.

Couëtoux, A.; Hoock, J.-B.; Sokolovska, N.; Teytaud, O.; and Bonnard, N. 2011. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*, 433–445. Springer.

Cowling, P. I.; Powley, E. J.; and Whitehouse, D. 2012. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games* 4(2): 120–143.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial intelligence* 49(1-3): 61–95.

Gaines, D.; Anderson, R.; Doran, G.; Huffman, W.; Justice, H.; Mackey, R.; Rabideau, G.; Vasavada, A.; Verma, V.; Estlin, T.; Fesq, L.; Ingham, M.; Maimone, M.; and Nesnas, I. 2016. Productivity Challenges for Mars Rover Operations. In *International Conference on Automated Planning and Scheduling, Planning and Robotics Workshop (PlanRob)*.

Kearns, M.; Mansour, Y.; and Ng, A. Y. 2002. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine learning* 49(2): 193–208.

Lund, K.; Dietrich, S.; Chow, S.; and Boerkoel, J. 2017. Robust execution of probabilistic temporal plans. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31.

Melkó, E.; and Nagy, B. 2007. Optimal strategy in games with chance nodes. *Acta Cybernetica* 18(2): 171–192.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic Control of Plans with Temporal Uncertainty. In *International Joint Conference on Artificial Intelligence*, IJCAI'01, 494–499. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1558608125.

Rabideau, G.; and Benowitz, E. 2017. Prototyping an Onboard Scheduler for the Mars 2020 Rover. *Proceedings of the International Workshop on Plaanning and Scheduling for Space, IWPSS* (Iwpss 2017).

Saint-Guillain, M.; Stegun Vaquero, T.; Agrawal, J.; and Chien, S. 2020. Robustness Computation of Dynamic Controllability in Probabilistic Temporal Networks with Ordinary Distributions. In Bessiere, C., ed., *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 4168–4175. International Joint Conferences on Artificial Intelligence Organization. doi: 10.24963/ijcai.2020/576.

Saint-Guillain, M.; Stegun Vaquero, T.; Agrawal, J.; Chien, S.; and Abrahams, J. 2021. Probabilistic Temporal Networks with Ordinary Distributions: Theory, Robustness and Expected Utility. *Journal of Artificial Intelligence Research* .

Tsamardinos, I. 2002. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Hellenic Conference on Artificial Intelligence*, 97–108. ISBN 978-3-540-43472-6. doi:10.1007/3-540-46014-4.

Yee, T.; Lisỳ, V.; and Bowling, M. H. 2016. Monte Carlo Tree Search in Continuous Action Spaces with Execution Uncertainty. In *IJCAI*, 690–697.

Analyzing the Efficacy of Flexible Execution, Replanning, and Plan Optimization for a Planetary Lander

Daniel Wang, Joseph A. Russino, Connor Basich, and Steve Chien

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109

Abstract

Plan execution in unknown environments poses a number of challenges: uncertainty in domain modeling, stochasticity at execution time, and the presence of exogenous events. These challenges motivate an integrated approach to planning and execution that is able to respond intelligently to variation. We examine this problem in the context of the Europa Lander mission concept, and propose a planning and execution framework that responds to feedback and task failure using two techniques: flexible execution and replanning with plan optimization. We develop a theoretical framework to predict the value of each of these techniques, and we compare these predictions to empirical results generated in simulation. We demonstrate that an integrated approach to planning and execution that is grounded in flexible execution, replanning, and utility maximization will be an enabling technology for future tightly-constrained planetary surface missions.

Introduction

When integrating AI planning into robotic applications, planners are consistently challenged by variation in execution and uncertainty in the quality of our environment models. In space-based applications, this is especially challenging because the environment is largely unknown, reducing the quality of our a priori models of the world. To address these problems, we describe an integrated approach to planning and execution in an unknown, unpredictable environment. First, we define a theoretical framework to examine the value of two integrated planning and execution techniques: flexible execution and replanning with plan optimization. We discuss this framework in the context of the Europa Lander mission concept. Finally, we compare the predictions of the model to empirical results in a Europa-like simulation environment.

The primary empirical context of our model is a mission concept to perform *in situ* analysis of samples from the surface of the Jovian moon Europa (Hand 2017). Unlike prior NASA missions, a priori domain knowledge is severely limited and uncertain, and communication with Earth is limited by long blackout periods (about 42 hours out of every 84 hours). Consequently, a successful mission requires a planning and execution framework that is highly efficient[1] , robust to unprecedented levels of uncertainty, and still capable of maximizing its overall utility. On the other hand, the Europa Lander concept has a fairly rigid definition of what actions the lander must perform in order to produce utility. Our planning algorithm leverages this domain-specific knowledge by making use of a hierarchical task network (HTN) and using heuristic-guided search to examine various task combinations to maximize utility. The ultimate goal for a Europa Lander would be to analyze surface material and communicate the resulting data products back to Earth. To reward accomplishment of these goals, we assign utility to tasks such as sample excavation and seismographic data collection, but do not receive this utility until the lander communicates the data down to Earth. In the HTN framework, this means that tasks in a hierarchy produce utility only if the full hierarchy is executed.

For our empirical evaluation, we base our planning system on MEXEC, an integrated planner and executive first built for NASA's Europa Clipper mission (Verma et al. 2017). We compare four approaches to planning on the Europa Lander problem similar to those used in prior missions: a static plan without failure recovery mechanisms, a static plan with ground input for failure recovery (Gaines et al. 2016), flexible execution without replanning, and flexible execution with replanning (Rabideau and Benowitz 2017). We explore the value of flexible execution and replanning with plan optimization, and examine these techniques' effects on utility in these scenarios. We demonstrate that, true to our model's prediction, each technique shows significant improvement in utility achievement in the Europa Lander domain.

Domain Description

The primary goal of the Europa Lander mission concept is to excavate and sample the surface, analyze the sampled material for signs of biosignatures, and communicate that

[1]As a point of reference, the RAD750 processor used by the Mars 2020 rover has measured performance in the 200-300 MIPS range. In comparison, a 2016 Intel Core i7 measured over 300,000 MIPS, or over 1000 times faster. Furthermore, the Mars 2020 onboard scheduler (Agrawal et al. 2019) is only allocated a portion of the computing cycles onboard the RAD750 resulting computation *several thousand times* slower than a typical laptop.

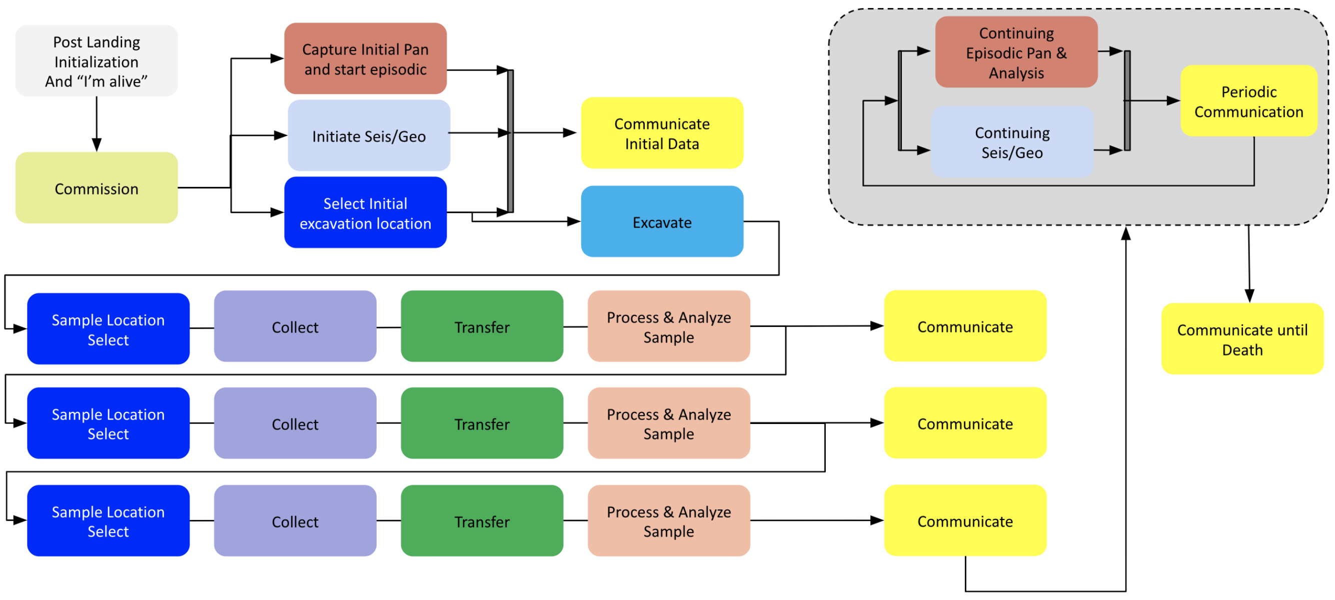


Figure 1: A task network for the Europa Lander mission concept. The diagram represents a potential execution trace of the mission that would fulfill baseline requirements.

data back to Earth (Hand 2017). Additionally, there are secondary objectives to take panoramic imagery of the Europan surface and collect seismographic data. Lander operations are generally limited to the accomplishment of these two overarching goals. This provides significant structure to the problem, since the concept mission clearly defines the sequence of actions required to achieve these goals. Figure 1 displays the strong dependency structure inherent to the Europa Lander concept mission. In order to sample, the lander needs to have excavated a trench; in order to analyze, the lander needs to have collected a sample; etc.

As a minimum requirement, the lander should excavate a trench in the Europan surface, collect three samples from that site, analyze those samples, and return that data to Earth. The basic requirements of a mission would require only a single site to be excavated. However, there is value in excavating additional sites, because the material at different sites may possess different properties. On the other hand, the lander may choose to resample the same location, for example, in order to verify the discovery of a biosignature. In the baseline mission concept, all three of the lander's samples are chosen from the same target. Note that after the first site is excavated, no further excavations are needed to sample from that trench; all three sampling activities can share a single excavation site. After excavation and sample collection, samples must be transferred into scientific instruments that analyze the material and produce data products. Then, for a mission to achieve any actual utility, those data products must be communicated back to Earth.

In addition to sampling tasks, the lander may engage in seismographic data collection and period panoramic imagery tasks. These are considered lesser goals, with lower utility associated with their completion. As such, the data products that these tasks generate are considered to have lower value. However, these tasks also involve no surface interaction, and have less uncertainty associated with them as a result.

It is important to note that utility is only achieved when data is downlinked back to Earth. This is true for both the sampling and seismograph/panorama tasks. Some excavation sites or sampling targets may provide more utility than others if, for example, one of those targets has a positive biosignature and the other does not. However, regardless of the quality of the material that the lander samples, no utility is achieved unless that data is communicated. This dynamic means that while potential utility is generated during the sampling and analysis phases, it is only realized by completing relevant communication tasks.

The Europa Lander mission concept is also constrained by a finite battery that cannot be recharged. Battery life is a depletable resource, and the lander must use its energy as efficiently as possible. Each task saps energy from the battery, and our algorithm must plan accordingly to maximize utility in face of this constraint. In addition to this challenge, the surface characteristics of Europa are uncertain, and any prior mission model that is generated before landing is sure to have inaccuracies. In particular, the energy consumption of the excavation and sample collection tasks is largely unknown. There is also significant variation in the utility of any given sample, since the value of sampling a given target on Europa depends on whether the material is scientifically interesting, e.g. whether a biosignature is present.

Approach

We design our planning system to respond intelligently to stochasticity at execution time, since we expect this to be a significant factor in our domain. Planning and execution are integrated in our approach, in order to respond to variation and therefore better optimize overall utility achieved. We achieve this integration through the use of two techniques: flexible execution and replanning with plan optimization.

Flexible Execution

Flexible execution is a lightweight rescheduling algorithm that runs at a much higher cadence than the planner. This algorithm has two main properties: (1) it is significantly less costly than replanning, and (2) it is significantly less powerful than replanning. Despite its limited scope, flexible execution is valuable because it can be run so frequently. This allows the system to handle less-severe unexpected events without incurring the cost of replanning. Previous NASA missions have made heavy use of flexible execution, such as the Mars 2020 Perseverance rover (Chi et al. 2018). Our implementation differs in focus, emphasizing responses to adverse events.

In our system, flexible execution consists of two major components. The first is *task push*. If a task's preconditions are not met, before failing the task, we allow it to wait for some amount of time for this inconsistency to resolve. Such a situation might occur, for example, if previous dependencies are unexpectedly delayed. We then push the start time of the task forward in the plan. Task push is implemented as a callback that is run before a task is dispatched to the execution system. The executive checks the task's preconditions and delays dispatch until either the conditions have been met, or the task's wait timeout has been exceeded.

The second component of flexible execution is *automated retry*. After a task completes with a failure code, flexible execution can immediately re-schedule the task if its preconditions are still met. The plan is then updated to account for the new predicted end time of the task, as well as its additional resource usage. Here, the system short-circuits a simple failure response, avoiding planning costs for failures whenever possible.

In the context of the Europa Lander domain, flexible execution offers significant value despite its simplicity. Because significant noise is expected in resource impacts, providing a low-cost method of handling mismatches in resource use predictions often avoids costs associated with either replanning or waiting for ground input. For example, if heating a joint on the lander is slower than predicted, flexible execution may handle this by re-triggering the heating operation or delaying arm movement, either of which would be sufficient to resolve the issue.

Replanning with Plan Optimization

For more complex failure responses, simple retries may be insufficient. In these cases, we turn to replanning during execution. Replanning allows the system to make use of online state updates, responding to variation from the original plan's predictions. Our framework measures the value of each resource being modeled, and assigns that value to the given resource in the planning model. Then, when replanning, the planner uses the actual, measured value of the state, rather than the previous predicted value. This allows the system to update its goals according to what is realistically possible given the current state measurements of the system. When tasks fail, their predicted state impacts are usually not realized. Replanning provides a mechanism to respond to these problems in a more complex manner than retrying the task. For example, excavation of the Europan surface is a complicated task with many modes of failure. Retries or delays may be insufficient responses to these failure modes, which may require additional actions to be taken.

In addition to this, replanning allows the system to make use of additional knowledge gained at execution time. This may take the form of task model updates and utility adjustments. For example, during execution time, the lander may discover that a task consumes more energy than expected, or that it produces more valuable data than expected. In the Europa Lander domain, the system might discover a biosignature at a sampling location, which would drastically change the site's utility. This is where plan optimization comes into play. By updating the task models, replanning can take execution-time knowledge into account and generate plans that produce more utility. Replanning thus improves overall utility achievement through two mechanisms: more advanced failure recovery, and plan optimization given execution-time knowledge.

Theoretical Framework

We define our planning problem as follows. We provide our planner with a set of tasks $T = \{t_0, t_1, ..., t_n\}$. Each task is represented by a tuple $t_k = \{c_k, u_k, d_k, P, I\}$ where:

- c_k represents the task's cost.
- u_k represents the task's utility.
- d_k represents the task's nominal duration.
- P is the set of the task's preconditions. These may be based on resource values, or on the execution state of dependency tasks.
- I is the set of its impacts on resource timelines.

This matches the timeline representation of execution state used by (Verma et al. 2017). For our problem, we assume that we have a fixed cost budget b. In the Europa Lander domain, this budget represents the non-rechargeable battery, with each task using up some amount of that battery's energy. We wish to maximize utility by scheduling tasks subject to the following constraints:

- For all tasks, all preconditions are valid.
- For all tasks, all impacts are valid.
- The sum of all task costs does not exceed b.

In our framework, we examine four planning and execution strategies: static, ground, FE, and replan. Using the static strategy, a plan is generated before execution time, then executed without change. No failure responses are enabled, so any task failure results in the termination of plan execution. In the ground strategy, we introduce a mechanism for failure resolution: waiting for ground input. We assume that ground input is able to resolve all failures. The plan is still pre-generated, but task failures can be handled without termination of execution, albeit in a costly manner. In the FE strategy, we allow flexible execution of our plans, which provides another failure resolution mechanism. Flexible execution is less costly, but is able to handle only a fraction of possible failures, with all other failures handled by waiting for ground input. Finally, in the replan strategy, we allow for modification of the plan at execution time according to information discovered while running. This provides another failure resolution mechanism that we assume is more powerful the FE, but less powerful than ground input. In addition, replanning allows for the optimization of plans during execution time according to newly discovered utility. Replanning can therefore serve dual purposes: resolving task failures, and changing the plan to increase overall utility gain.

Given this context, we predict the overall utility achievement of a plan using an estimate of utility per unit cost u_{avg}. Then, assuming that tasks always succeed, our expected utility for a plan would be bu_{avg}. To factor in task failure, we assume that tasks fail with some probability $P(\text{fail})$, and we assume that task failures follow a Poisson distribution. The first planning/execution strategy that we analyze is the static strategy. Here, since the strategy terminates execution on failure, the system's expected utility achievement is based on how long it can be expected to execute. Then, the expected utility achievement of this strategy is given by:

$$U(S_s) = u_{avg} \cdot \min\left(b, \frac{c_{avg}}{P(\text{fail})}\right) \tag{1}$$

Here, c_{avg} denotes the average cost of each task.

In the ground strategy, we include a rudimentary error response of "going to ground" to seek manual intervention. To model this in our framework, we assume that such "wait for input" responses each incur a cost c_w, and always allow plan execution to continue. Then, if our plan has n_p tasks, the utility achievement of this "ground" strategy is:

$$U(S_g) = u_{avg}\left(b - P(\text{fail})n_p c_w\right) \tag{2}$$

In the FE strategy, we introduce flexible execution and assume that some subset of task failures can be resolved with this feature. We denote the probability of a task failing in this way as $P(\text{FE})$. Note that $P(\text{FE}) < P(\text{fail})$, since failures that are resolvable by flexible execution are a subset of all task failures in general. We assume that flexible execution has a negligible cost. Then, the utility achievement of plan execution using this strategy is:

$$U(S_f) = u_{avg}\left(b - (P(\text{fail}) - P(\text{FE}))n_p c_w\right) \tag{3}$$

Finally, we consider the replan strategy, which incorporates flexible execution and replanning with plan optimization. Unlike flexible execution, replanning incurs some non-negligible cost c_r. We assume that, like flexible execution, replanning is able to resolve some subset of task failures. We denote the probability that a given task fails in a way that can be resolved via replanning, but not flexible execution, as $P(\text{replan})$. Finally, we assume that all failure modes can be resolved via waiting for ground input. Then, if we denote the probability that a failure is resolvable only by ground input as $P(\text{wait})$:

$$P(\text{fail}) = P(\text{wait}) + P(\text{replan}) + P(\text{FE}) \tag{4}$$

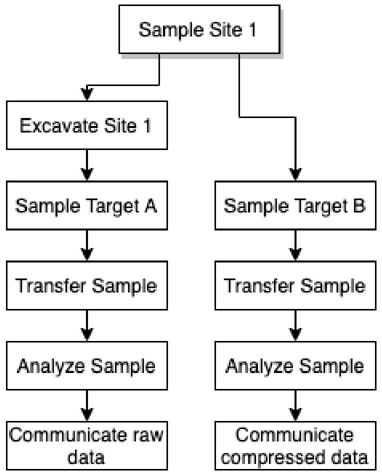


Figure 2: Two possible decompositions of a single parent "Sample Site 1". In the left decomposition, the lander excavates the site, samples target A, and communicates raw data. In the right decomposition, the lander skips excavation, samples site B, and communicates compressed data. Both achieve the same goal of sampling site 1.

Failures are resolved by the least costly resolution mechanism. Thus, when a task fails, our system attempts to resolve it by flexible execution, if possible, falling back to replanning and ground intervention in sequence. To model plan optimization, we provide our planning system with opportunities to discover utility at certain points during execution. We denote the number of such opportunities as d, and the expected additional utility discovered as u_d. Then,

$$\begin{aligned} U(S_r) =& du_d + u_{avg} \\ & (b - n_p(P(\text{wait})c_w - P(\text{replan})c_r)) \end{aligned} \tag{5}$$

Planning Approach

Problem Model

We model this problem using a hierarchical task network (HTN) to compile the domain-specific knowledge of the dependency structure into the task network. HTNs have been used successfully in industrial and other real-world applications to improve the tractability of planning problems in systems such as SHOP2 (Nau et al. 2003) and SHOP3 (Goldman and Kuter 2019). In an HTN, hierarchical tasks are decomposed to a set of subtasks. We refer to the higher-level tasks as "parent tasks", and refer to their children as "subtasks". Parent tasks may decompose into a number of different sets of subtasks; we refer to each of these sets as a potential "decomposition" of that parent task. Finally, we refer to tasks with no decompositions as "primitive tasks". These primitive tasks represent tasks that the lander can be directly commanded to perform.

Decompositions provide a number of benefits to our planning approach, significantly reducing plan search space. In addition, we can treat all subtasks of a parent task as a singular block for planning purposes. The lander only achieves utility after completing an entire sequence of sample, analyze, communicate. Decompositions allow us to treat "sample, analyze, communicate" as a single unit and schedule them accordingly. Thus, our model intrinsically biases the lander against planning to sample without a corresponding communication task. This may not always be optimal, if for example, excavation and sampling is cheap and communication is very expensive. However, for our problem, energy use is dominated by the excavation and sampling tasks, and the decomposition paradigm effectively encodes this domain-specific knowledge into our planning routine.

There are three main parent task types in our mission model. The first is a Preamble, which consists of post-landing initialization and other one-time initialization tasks. Second are sampling tasks. These consist of excavation, sample collection, transfer, analysis, and communication tasks. Excavation can take place at one of two excavation sites, and may be skipped if an excavation has previously occurred for the specified site. For collection tasks, the lander may choose between four collection targets: two for each excavation site. It may revisit a target that has already been sampled, still obtaining utility for a repeat sample. Then, for communication tasks, the lander may choose to either communicate raw data or compressed data. Finally, there are Seismograph/Panorama tasks, which consist of seismographic data collection, panoramic image collection, and communication of that data.

In our problem, we assign utility primarily to two activities: sampling and communication. Both of these task models are assigned a numeric value representing their utility, which can be updated online by the planning and execution system if knowledge at execution time alters the expected utility of a given action. Utility for these tasks is achieved only after their full decomposition has been successfully executed. Thus, for sampling utility to be achieved, a corresponding communication step must successfully complete.

We assign utility to sampling tasks in order to differentiate between sites that may be more or less interesting, depending on the scientific value of the site. Communication utility is larger, and remains constant. For the communication tasks, we assign higher utility and cost to tasks that communicate raw data, compared to those that communicate compressed data. This simulates a Pareto optimal "menu" of communication options. The combination of sampling and communication utilities represents the overall utility of a parent sampling task. Seismograph/panorama utility is driven solely by communication utility.

Planning Algorithm

Our planning algorithm uses the HTN model of the Europa Lander problem to build a search graph, with nodes holding partial plans and edges holding task decompositions. We perform a heuristic-guided branch and bound search on this graph and select the best plan explored. The algorithm consists of four phases: pre-processing, initialization, exploration, and plan selection.

First, a pre-processing step flattens task decompositions into a single layer, such that parent tasks decompose into a chain consisting only of primitive, non-hierarchical subtasks. This allows us to assign utility and energy cost directly to each decomposition, because its breakdown into disparate subtasks has already been performed. Then, each decomposition's utility is the sum of each of its subtasks' utility. The same is true for energy cost. This step is performed once per domain model, offline. Preprocessing has exponential runtime in the worst case, and future work may require additional search in decomposing tasks as well as planning them.

Our search graph consists of nodes containing partial plans and their associated energy cost and utility. A node's cost is simply the sum of the costs of each task in the node's partial plan; the same goes for utility, though future work may take joint utility into account. In the initialization phase, the algorithm creates a single node containing an empty

Algorithm 1: Europa Lander Planning

```
Input: A list of tasks to schedule $T$
Output: A plan of scheduled tasks $P$
/* initialize exploration queue   */
node_collection = [];
add (plan=[], utility=0, cost=0) to node_collection;
edge_collection = [];
for d in task.decompositions do
    new_edge = $(d, d.utility, d.cost)$;
    add new_edge to edge_collection;
end
explore_q = [];
for edge in edge_collection do
    add (node_collection[0], edge) to explore_q;
end
/* search exploration queue       */
num_explored = 0;
while num_explored below exploration bound do
    num_explored++;
    plan, decomp = explore_q.get_max();
    if decomp tasks can be added to plan then
        new_plan = plan + decomp tasks;
        add new_plan to node_collection;
        for edge in edge_collection do
            if edge.task not in new_plan and
              new_plan.cost + edge.cost below
              max_cost then
                add (new_plan, edge) to explore_q;
            end
        end
    end
end
/* find best plan in node
   collection                     */
best_plan = null;
for plan in node_collection do
    if plan.utility above best_plan.utility then
        best_plan = plan;
    end
end
return best_plan;
```

plan, with utility and cost 0. Then, it iterates through all task decompositions created in the pre-processing phase in order to generate the set of edges that may be followed from a given node. To finish the initialization phase, the algorithm populates an exploration queue with (node, edge) pairs, pairing the singular initial node with all edges in the collection. At the end of the initialization phase, then, the exploration queue consists of all task decompositions paired with the empty plan.

In the exploration phase, the planner pops the top of the exploration queue to get (P, T), where P is a partial plan, and T is the list of primitive subtasks comprising a task decomposition. It then attempts to schedule all tasks in T given the state of the world produced by following the plan P. If the tasks cannot be scheduled, it moves on to the next exploration queue item. If the tasks can be scheduled, i.e. their preconditions are met and their impacts do not produce any conflicts, a new graph node is created. This node contains a new plan P', the resulting plan after adding the tasks in T to P.

After creating this plan node, the planner iterates through the edge collection again, pairing the new plan with all possible tasks. In this iteration, it ignores tasks that have already been scheduled in the plan, so as to avoid duplicates. The algorithm also filters these pairs to ensure that the total cost $P.cost + T.cost < M$, where M is the max energy cost allowed (equal to the current battery charge of the lander). This bounds our search, and we further bound the algorithm's search by limiting the number of exploration candidates examined. Note however that this bound maintains optimality if we allow the algorithm to expand the entire space. After filtering, these pairs are added to the exploration queue, and the next queue item is examined. The exploration queue is a priority queue, with (plan, decomposition) pairs ordered by a heuristic value to improve search results. Given a plan, decomposition pair (P, T), we assign the heuristic value $h(P, T) = P.utility + \frac{T.utility}{T.cost}$. Finally, in the plan selection phase, the algorithm iterates through all candidate plan nodes, selecting the plan with the highest utility. Ties are broken according to energy cost, where a lower energy cost is preferred.

Empirical Evaluation

To test our model, we ran simulations of our planning and execution system on three variants of the Europa Lander domain described in Figure 1. The first is the base scenario. Here each task consumes an amount of energy that matches its a priori expectation in the task network, but may be noisy, with a standard deviation of 10%. In the second variant, we bias this noise such that tasks are expected to consume 10% more energy than modeled. Finally, the third variant biases noise in the opposite direction, such that tasks are expected to consume 10% less energy. For each variant, we simulated each of the four planning/execution strategies discussed in our theoretical framework, and measured the utility achieved. In simulation, the failure probability of each task is uniform and independent. Each failure resolution mechanism is assumed to have a fixed cost and always succeed in resolving the issue. The data for each figure shows the mean utility achieved across 50 simulations of the scenario.

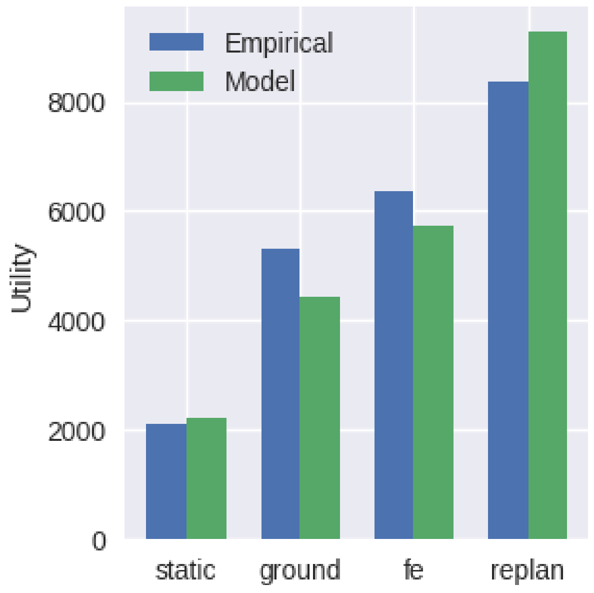


Figure 3: Average utility achieved in simulation of the base Europa Lander domain for 4 planning strategies, compared to theoretical model predictions.

For our model calculations, we estimate our average utility per cost (u_{avg}) by analyzing plans generated by a prescient planner. This planner has perfect execution information a priori, so plan execution exactly matches the planner's predictions. Task failure probability is assumed to be 0.1, and we assume flexible execution is able to handle 30% of such failures, while replanning is able to handle an additional 60% of failures. Thus, $P(\text{replan}) = 0.04$ and $P(\text{FE}) = .02$.

Our model predicts the "static" strategy to perform poorly, since it has no failure resolution mechanisms and is thus likely to terminate quickly. By introducing a failure recovery mechanism, our model predicts the "ground" strategy to improve performance considerably. However, this failure recovery mechanism is still fairly costly. The "fe" strategy introduces flexible execution to mitigate this. As such, our model predicts a higher utility achievement, since some set of failures are now resolved by a less costly mechanism. Finally, the "replan" strategy is predicted to perform best of all the strategies. Like the "fe" strategy, it introduces another failure resolution mechanism. However, it also introduces additional utility through plan optimization. When utility is discovered at execution time, the "replan" strategy is able to exploit that discovery, where the other strategies are not.

In Figure 3, we compare the predictions of our model to the measured utility achievement of our system in simulation. We see that the four strategies follow the general contour of our model's predictions, but vary by some amount.

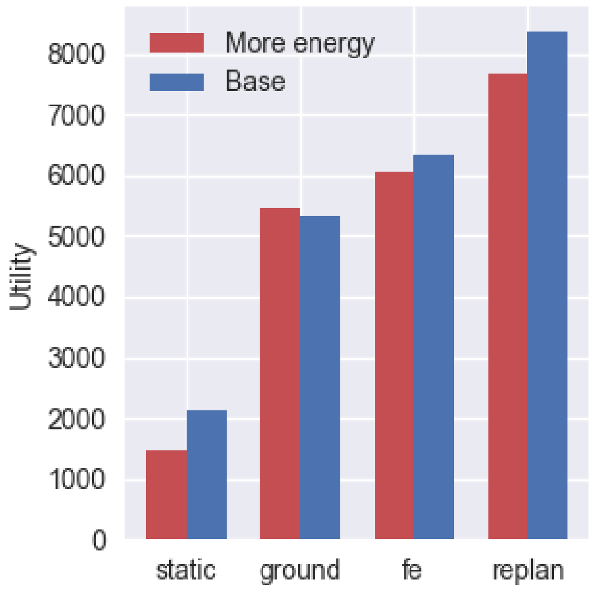


Figure 4: Average utility achieved in simulation of the Europa Lander domain where all tasks take 10% more energy than expected, compared to empirical results in the base domain.

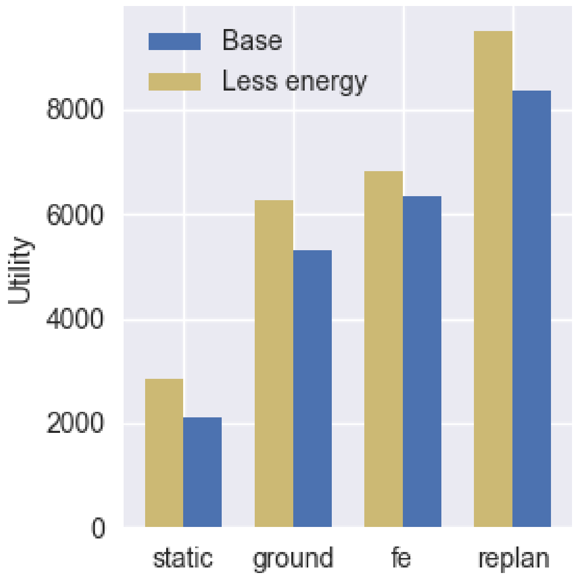


Figure 5: Average utility achieved in simulation of the Europa Lander domain where all tasks take 10% less energy than expected, compared to empirical results in the base domain.

While our predictive model generally matches our empirical measurements, it is limited in some aspects. The model uses u_{avg} as a way to estimate utility achievement based on power, smoothing performance across the entire execution into a linear model. However, in the Europa Lander domain, utility is achieved only during communication events. Because the model views utility gain as purely linear, it is unable to capture the spikes in utility inherent in the domain.

In addition, in the Europa Lander domain, sites only need to be excavated a single time, and multiple samples can be taken from a single excavation site. This means that the first sample taken at a site is much more costly than future samples. Because of this, if the system tends to run out of energy while attempting to sample a site for the first time, the model is likely to overestimate utility gain, since a significant portion of energy is used while no utility is gained. On the other hand, when the system tends to halt while repeatedly sampling from an existing site, the model underestimates utility. This behavior is prominently seen in the ground and FE strategies in Figure 3. Both strategies spend a significant portion of their execution repeatedly sampling from an excavation site, leading to higher utility gain than expected during these portions of the plan execution.

For the replan strategy in particular, we also consider the effects of utility discovery and plan optimization in replanning. To determine a value for d, the number of times that utility discovery can be exploited, we calculate and upper bound for this value based on the total energy available to the system. However, the system may not be able to take advantage of utility discovery this number of times, since it may run into too many task failures, or the planner may simply choose to complete other tasks. Thus, the calculations for our model tend to overestimate the value of utility discovery in the replanning strategy.

Next, we consider the effects of biased noise on the utility gain of our system. First, we examine the scenario where all tasks use 10 percent more energy on average than expected. A comparison of this scenario and the base scenario is shown in Figure 4. Naively, we might expect utility in each scenario to decrease by about 10 percent. However, because utility is achieved in spikes through the completion of fairly lengthy chains of tasks, events have an impact on utility only if they increase or decrease the probability of successfully completing a chain of tasks. In the “more energy” scenario, the ground strategy appears generally unaffected.

The replan strategy is affected more heavily, since a lower pool of energy available limits the strategy’s ability to take advantage of discovered utility. On the other hand, because it is able to replan, it can make use of lower cost actions such as Seismograph/Panorama tasks to gain utility despite lacking the energy to complete a sample.

Finally, we consider the scenario where tasks take 10 percent less energy than expected (Figure 5). Here, the ground strategy improves considerably in performance, while FE improves at a lower clip. This is consistent with what we see in the previous scenario. The ground strategy is able to benefit significantly from the extra energy and complete an extra sample cycle, while FE is not as close to this boundary and thus is not affected as strongly.

The replan strategy also sees significant benefits from extra energy. Extra energy enables additional samples, whose benefit is amplified by the potential for utility discovery. In addition, the replan strategy is able to integrate knowledge of the additional energy during execution time as it updates state predictions with the reality on the ground. Thus, instead of settling for a Seismograph/Panorama task, as might occur in the base or high energy use scenarios, the replan strategy is more often able to process a sample.

Related Work

Decision-theoretic planning is an effective approach to planning under uncertainty, particularly in robotic domains, as it provides a formal model for reasoning about problems in which actions have stochastic outcomes or the agent has incomplete information about its environment (Iocchi et al. 2016; Saisubramanian, Zilberstein, and Shenoy 2017; Zilberstein et al. 2002). The primary objective of decision-theoretic planning is to produce plans or policies that define the potential trajectories of actions that the agent may take which maximizes its expected utility, rather than maximizing or guaranteeing goal-reachability (Boutilier, Dean, and Hanks 1999). A standard approach in decision-theoretic planning for modeling domains is to use a Markov decision process (MDP) (Bellman 1957) when the agent knows the full evaluation of every state at each timestep, or a partially observable Markov decision process (POMDP) (Spaan 2012) where this holds only for a subset of the variables that define the statespace.

However, several issues in spacecraft or rover operations complicate the use of said decision making models. First, these models traditionally do not support durative or concurrent actions, but rather assume that all actions are instantaneous and fully sequential in nature. Second, although there have been a number of approaches over the years aimed at improving the scalability of these approaches (Guestrin et al. 2003; Wray, Witwicki, and Zilberstein 2017; Yoon, Fern, and Givan 2007), most algorithms that solve MDPs produce policies that account for all contingencies and provide actions for all states in the domain. This is generally impractical or impossible in spacecraft and rover operations where computational power is (often severely) limited, and more so in our problem where the battery is non-rechargeable and the domain model is expected to be modified repeatedly throughout the agent's operation.

Onboard planning and execution are of great interest to the space domain. Flexible execution of tasks is a central focus of execution engines like PLEXIL (Verma et al. 2005) and TRACE (de la Croix and Lim 2020). The Earth Observing One (EO-1) spacecraft (Chien et al. 2005), which flew for over 12 years from 2004-2017, was designed specifically to react to dynamic scientific events. Planning was performed by the CASPER planning software (Chien et al. 2000), which took on the order of 10s of minutes to replan but did not produce temporally flexible plans. To address this, the onboard executive (SCL) was able to flexibly interpret the *execution* of a plan to handle minor execution runtime variations. The flight and ground planners (Chien et al. 2010) both used a domain specific search algorithm that enforced a strict priority model over observations for a limited model of utility. Recently, the Intelligent Payload Experiment (IPEX) also successfully used the CASPER planning software to achieve its mission objective, further validating the efficacy of using onboard replanning to handle dynamic events and observations during operation even when the plans are not temporally flexible (Chien et al. 2017).

The M2020 Perseverance rover also plans to fly an onboard planner (Rabideau and Benowitz 2017) to reduce lost productivity from following fixed time conservative plans (Gaines et al. 2016). Like the planning approach we propose in this paper, the M2020 planning architecture also relies on rescheduling and flexible execution (Chi et al. 2018), ground-based compilation (Chi et al. 2019), heuristics (Chi, Chien, and Agrawal 2020), and very limited handling of planning contingencies (Agrawal et al. 2019). However, it uses a non-backtracking planner, which cannot take advantage of plan optimization or utility discovery. Our work also takes a different focus, primarily examining the effects of task failure and considering integrated planning in the context of failure resolution. Finally, many characteristics of the M2020 mission are fundamentally different from the mission concept we consider here, such as the lack of reliable a priori model parameters, the inability to recharge the battery, and the long communications blackout time windows incentivizing greater mission autonomy.

Future Work

Our work focuses primarily on our planning system's response to adverse events such as task failure. Our examination of positive exogenous events is limited to analysis of utility discovery. However, in space exploration domains, due to conservative parameter assignments, we often find that tasks finish early or use fewer resources than the margin allocated to them. Reasoning about these events may provide a model that more accurately represents the reality of the Europa Lander domain.

In addition, in this work we focus primarily on energy as a resource. However, a number of other resources exist, and the consumption of any of these may be noisy or biased, affecting plan execution. In particular, task execution time has wide-ranging effects on both task energy use and plan execution as a whole, especially when deadlines come into play. These deadlines are especially present in the Europa Lander domain in the form of ground communication windows. Task execution time and other variables therefore represent a significant unexplored area of work in this domain.

While we react to uncertainty at execution time, we do not take this into account when planning. This is apparent in our system's behavior in the scenario where tasks take more energy than expected. A more sophisticated planner would explicitly integrate probability of such adverse events, maximizing expected utility. For example, excavation tasks involve risk; task failure could result in significant energy loss or damage to the lander. Reasoning about exogenous events such as these would improve utility achievement by potentially avoiding such risks, or even seeking them out later in the mission when failure is less impactful.

Acknowledgments

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

References

Agrawal, J.; Chi, W.; Chien, S.; Rabideau, G.; Kuhn, S.; and Gaines, D. 2019. Enabling Limited Resource-Bounded Disjunction in Scheduling. In *11th International Workshop on Planning and Scheduling for Space (IWPSS 2019)*, 7–15.

Bellman, R. 1957. A Markovian decision process. *Journal of Mathematics and Mechanics* 679–684.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research (JAIR)* 11:1–94.

Chi, W.; Chien, S.; Agrawal, J.; Rabideau, G.; Benowitz, E.; Gaines, D.; Fosse, E.; Kuhn, S.; and Biehl, J. 2018. Embedding a Scheduler in Execution for a Planetary Rover. In *International Conference on Automated Planning and Scheduling (ICAPS 2018)*.

Chi, W.; Agrawal, J.; Chien, S.; Fosse, E.; and Guduri, U. 2019. Optimizing Parameters for Uncertain Execution and Rescheduling Robustness. In *International Conference on Automated Planning and Scheduling (ICAPS 2019)*.

Chi, W.; Chien, S.; and Agrawal, J. 2020. Scheduling with Complex Consumptive Resources for a Planetary Rover. In *International Conference on Automated Planning and Scheduling (ICAPS 2020)*.

Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling. In *International Conference on AI Planning and Scheduling (AIPS)*, 300–307.

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

Chien, S.; Tran, D.; Rabideau, G.; Schaffer, S.; Mandl, D.; and Frye, S. 2010. Timeline-based space operations scheduling with external constraints. In *Twentieth International Conference on Automated Planning and Scheduling*.

Chien, S.; Doubleday, J.; Thompson, D. R.; Wagstaff, K. L.; Bellardo, J.; Francis, C.; Baumgarten, E.; Williams, A.; Yee, E.; Stanton, E.; et al. 2017. Onboard autonomy on the intelligent payload experiment cubesat mission. *Journal of Aerospace Information Systems* 14(6):307–315.

de la Croix, J.-P., and Lim, G. 2020. Event-driven modeling and execution of robotic activities and contingencies in the europa lander mission concept using bpmn. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*.

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

Goldman, R. P., and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: the case of SHOP3. In *Proceedings of the 12th European Lisp Symposium*, 73– 80. Zenodo.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research (JAIR)* 19:399–468.

Hand, K. P. 2017. *Report of the Europa Lander science definition team*. National Aeronautics and Space Administration.

Iocchi, L.; Jeanpierre, L.; Lazaro, M. T.; and Mouaddib, A.-I. 2016. A practical framework for robust decision-theoretic planning and execution for service robots. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Nau, D. S.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.

Rabideau, G., and Benowitz, E. 2017. Prototyping an Onboard Scheduler for the Mars 2020 Rover. In *International Workshop on Planning and Scheduling for Space (IWPSS 2017)*.

Saisubramanian, S.; Zilberstein, S.; and Shenoy, P. 2017. Optimizing electric vehicle charging through determinization. In *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Scheduling and Planning Approaches*.

Spaan, M. T. 2012. Partially observable Markov decision processes. In *Reinforcement Learning*. Springer. 387–414.

Verma, V.; Estlin, T.; Jónsson, A.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (plexil) for executable plans and command sequences. In *International symposium on artificial intelligence, robotics and automation in space (iSAIRAS)*.

Verma, V.; Gaines, D.; Rabideau, G.; Schaffer, S.; and Joshi, R. 2017. Autonomous Science Restart for the Planned Europa Mission with Lightweight Planning and Execution. In *International Workshop on Planning and Scheduling for Space (IWPSS 2017)*.

Wray, K. H.; Witwicki, S. J.; and Zilberstein, S. 2017. Online decision-making for scalable autonomous systems. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 7, 352–359.

Zilberstein, S.; Washington, R.; Bernstein, D. S.; and Mouaddib, A.-I. 2002. Decision-theoretic control of planetary rovers. In *Advances in Plan-Based Control of Robotic Agents*. Springer. 270–289.