Monte Carlo Tree Search for Integrated Planning, Learning, and Execution in Nondeterministic Python

Richard Levinson

rich.levinson@nasa.gov
Intelligent Systems Division
NASA Ames Research Center

Abstract

We present a novel use of Monte Carlo Tree Search (MCTS), adapted to explore a search space produced by choice points embedded in Python code. The choice points are nondeterministic assignment statements and subroutine calls. We present MCTS extensions required for doing tree search in this context which includes control constructs like hierarchical decomposition (subroutine calls), iterative while loops and conditional statements. We demonstrate how the system provides integrated planning, learning and execution in a simulated rideshare scenario in a city setting, and present preliminary experiments as a proof of concept. We also briefly describe a real-world NASA climate science application which involves planning only (no execution) and includes MCTS parallelization to exploit multiple CPU cores.

1 Motivation

This work is motivated by a long-term interest in the potential benefits of using a general programming language such as Python as the action representation for unified planning and execution models, to address problems with hybrid approaches where the planner and execution system use different models. Procedural models include control flow constructs like while loops and if statements (conditionals) which are essential for reactive execution. In the standard approach, a planner using a declarative model is connected to a reactive execution system through an intermediate language such as ROSplan (Cashmore et al. 2015).

The need to translate between planning and execution models, and to maintain two different behavior models increases complexity. In such hybrid systems, the planner can't help when execution strays outside of the planner's model. There is a hard division between the planning and execution action models which can be difficult to change. A unified action model for planning and reactive execution facilitates exploring various blends of reactive execution and planning because there is no hard boundary between them.

This work is also motivated by a practical problem we needed to solve more immediately. A prior attempt at planning for nondeterministic Python suffered performance problems and did not scale. It used Python's `fork` method to split Python processes into parallel processes for each node in the search space, which consumed all space on the computer (Levinson 2020). The new approach presented here replaces that performance killing overhead with Monte Carlo Tree Search (MCTS), which uses only a single process per planning trajectory.

A second practical motivation was to address a scaling problem in a real-world NASA application for planning satellite observations and data downlinks called D-SHIELD (Levinson et al., 2022; 2021). This application produces a coordinated plan for a swarm of satellites. D-SHIELD uses MCTS in a non-standard way to generate a full plan without any execution or exogenous events, so it is not the primary example in this paper, which aims to demonstrate integrated planning and execution. However it was a key driver for integrating MCTS into non-deterministic Python, and demonstrates scaling on a very large real-world problem which is quite different from our running example. The underlying MCTS code is the same as that presented in this paper.

This paper aims to present the new system and demonstrate how MCTS is used to integrate planning, learning and execution in nondeterministic Python. It focuses on explaining how MCTS is integrated into Python code extended with choice points, presents MCTS extensions required, and demonstrates two working examples.

2 Related Work

2.1 Integrated Planning and Execution

We present a new version of The Program Planning and Execution Language (PROPEL). Propel supports planning with a general programming language extended with nondeterministic choice points. Initially, LISP was the programming language (Levinson 1995), then C++ (Levinson 2005), and recently Python (Levinson 2020). This version integrates nondeterministic Python with Monte Carlo Tree Search (MTCS). Nondeterministic choice points (assignment statements and subroutine calls) may be embedded anywhere in Python. MCTS is the search engine that searches through the space of program variations defined by those choices. Propel is related to systems designed for tightly integrated planning and reactive execution including IDEA (Muscettola et al., 2000; 2002) and KIRK/RMPL (Kim, et al. 2001).

Reaction First Search (RFS) (Drummond et al., 1993) was a direct predecessor to Propel. RFS integrated a planner with a reactive executive which shared the same action model. The executive could operate without the planner by falling

back on a default reaction policy, called a *reactive competence*. If time is available, the planner can look ahead for potential problems or improvements. RFS demonstrated benefits of having the planner search the reaction trajectories first before branching out to explore non-default behaviors. RFS used a STRIPS-like declarative action representation. PROPEL continues that line of research using a general programming language as the action representation.

(Neufeld 2020) presents integrated planning and reactive execution for video games in dynamic environments. Neufeld describes a hybrid system where a long-term HTN planner is integrated with a reactive executive, comparing two reactive systems: MCTS vs. Behavior Trees (a relative of the behavior-based robotics like the subsumption architecture (Brooks 1991)). Here MCTS was used for the reactive executive, integrated with a slower HTN planner. In contrast, MCTS is our slower planner, integrated with a faster reactive default policy.

The *Operational Models* (Patra et al., 2021) share similar motivations with Propel for planning with a procedural model. They present *UCT for Operational Models* (UPOM), which uses a specialized reactive execution system called RAE (Ghallab et al, 2016) which is related to the Procedural Reasoning System (PRS) (Ingrand et al, 1996). Procedures can be defined with choice points for subroutine calls, but nondeterministic assignment statements are not supported. UPOM requires knowledge to use a specialized AI language and reactive execution system, RAE, as their procedural modelling language, compared to using Python.

2.2 Monte Carlo Tree Search (MCTS)

MCTS was introduced by (Coulom 2006) and was the key innovation in the first computer program to beat the world's best human 'go' player. MCTS typically uses a domain-independent heuristic called Upper Confidence Bound for Trees (UCT) (Kocsis Szepesva ri, 2006) to control the bias (preference) between exploring new areas of the search space vs. exploiting choices which worked well before.

MCTS is designed to be interleaved with execution, so it focuses search on the current state and the most immediate choices and returns the single next step to be executed. Before executing each action, MCTS is called to perform Monte Carlo simulations and determine the best action to execute. After the best action is executed, the "opponent" (if it's a board game) or the environment (if it's the real world) makes exogenous state changes (transitions to a new state). Then MCTS starts again from the new current state. Each trajectory from the root to a terminal state is called a *rollout*. One child is added to the tree on each rollout. MCTS incrementally grows the action-value table as the tree grows.

Figure 1 shows how MCTS works for an example where the agent repeatedly chooses to move North, South, East or West. Nodes (in yellow) represent *states* and edges are *action choices*. MCTS tracks the number of visits to each node and the total reward received for all paths through the node.

Every rollout follows a sequence of four stages: *Select, Expand, Simulate, Update*. In the *Select* stage, a leaf in the tree is selected to be expanded using a *tree policy* to guide action selection based on learning statistics. This tree policy uses UCT to balance between exploring new choices and exploiting successful prior choices. In the *Expand* stage, the selected node is expanded by adding a new child to the tree, for an unexplored action (node s7). In the *Simulation* stage, a sequence of actions is followed until a terminal state is reached. Action selection in the expand and simulation stages is guided by a *default policy*. Rollouts are terminated when a success or failure state is reached or when a time limit is reached. When a rollout terminates, the MCTS *update* stage begins. This is when the reward for that rollout is backpropagated to update the tree statistics. The number of visits and total reward is updated for each node in the tree back to the root. The action-value table estimates are stored only for states close to the current state (the tree), and not for the intermediate states produced during the simulate stage.

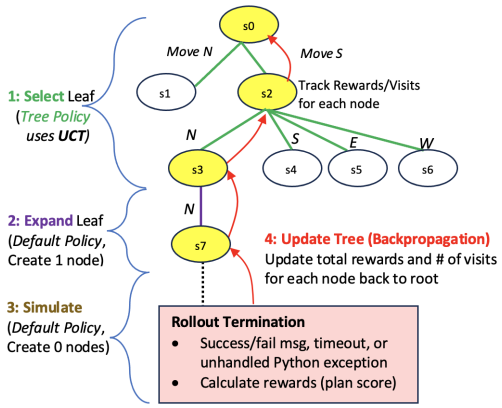

Figure 1: MCTS tree for our application

MCTS and Reinforcement Learning (RL) are strongly related. As explained by (Vodpivec et al., 2017) and by (Sutton and Barto, 2018), planning and learning both update an action-value function based on experience. MCTS implements a form of General Policy Iteration (GPI) which is at the core of RL. MCTS iterates between policy improvement and policy evaluation. In MCTS, the policy being improved and evaluated is the Tree policy. In the select stage, MCTS samples from the tree policy to select actions based on prior estimates. In the update (backpropagation) stage, feedback from new trajectories is used to update the action-value table. MCTS focuses on states near the current state and does not attempt to create a general policy to use in all future situations. It is a form of *situated reinforcement learning*.

3 Propel

The Program Planning and Execution Language (PROPEL) is Python with nondeterministic choice points. This is the action representation used for both planning and execution and now integrated with learning in the form of MCTS. Our definition of *nondeterministic programming* is the same as this from Wikipedia:

> *"A **nondeterministic programming language** is a language which can specify, at certain points in the program (called **choice points**), various alternatives for program flow. Unlike an if-then statement, the method of **choice between these alternatives is not***

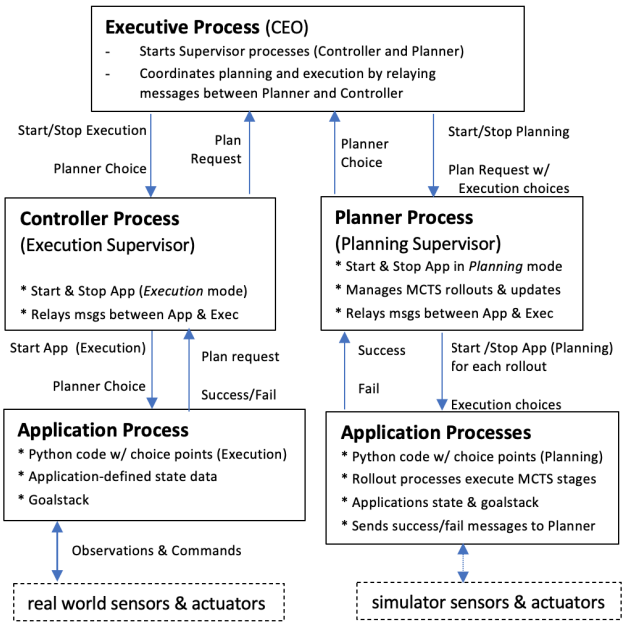

Figure 2: Propel's three process levels

directly specified by the programmer; the program must decide at run time between the alternatives."

Our nondeterministic programming language is Python extended with two types of choice points: `chooseValue` (nondeterministic assignment statement) and `chooseTask` (nondeterministic subroutine call). Each choice point adds a new degree of freedom to the Python code which may be helpful when operating in *uncertain and changing environments*. MCTS automatically chooses which value to assign, or which subroutine to call, rather than a programmer making that choice in advance or relying solely on a previously learned policy. MCTS searches through the space of program variations defined by choice points to maximize context-aware goal-driven rewards.

3.1 Propel Architecture: Three process levels

Figure 2 shows Propel's three levels of computational processes: Executive, Supervisor and Application.

Executive Process: The top-level executive process coordinates global strategy for integrated planning and execution. It starts, stops, and communicates with the planning and execution supervisor processes, called the Planner and Controller, respectively. The Executive may implement various *integrated planning and execution strategies*.

Supervisor Processes: There are two supervisor level processes. The *Controller* supervises *execution* of the application code. It starts and monitors execution of the Application-level code in the real world. The *Planner* supervises *planning* of the same Application-level code, using MCTS to simulate many variations (MCTS rollouts). The Controller and Planner processes can start and stop their respective application processes, and receive messages from them and from the Executive.

Application Processes: All application specific code and state data are defined and managed by the application level code. The Supervisor and Executive levels are generic Propel infrastructure which know nothing about any specific application. Identical application code is running in Execution and Planning mode. A single process runs it in execution mode. In planning mode, it runs in a separate process for each rollout.

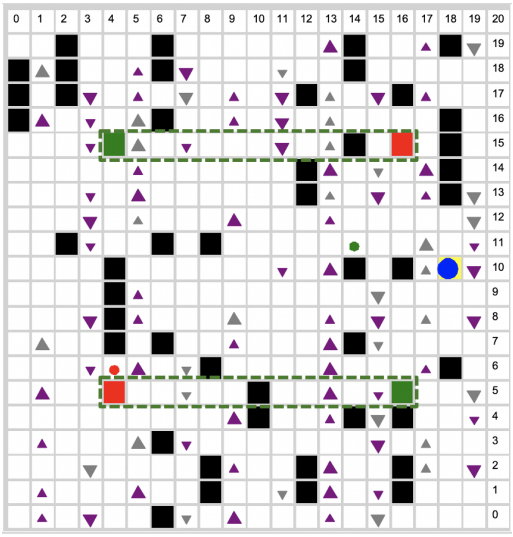

Figure 3: City Delivery initial state

3.2 Application/Propel API

The Application creates a *subclass of Propel* to inherit choice point methods and message API methods. Within each rollout, all MCTS stages (Figure 1) are run in the Application processes, except *update* which is run by the Planner. The application provides an *objective function* method to calculate the plan score after each rollout, which is used to update the MCTS tree statistics. A *state* object is constructed and maintained by application code to track the agent and environment state. The application passes the state to the planner to be used in the objective function for the MCTS *update* stage. A message API allows Application processes to inform their supervisors (planner or controller) about application events. The Application process sends *Success* messages tell the planner to terminate the current rollout, calculate the plan score and *update* the MCTS tree. *Failure* messages tell the planner to terminate the rollout with a score of 0, then update the tree. The *Executive* process relays messages between the Planner and Controller, enabling communication between them.

4 Demonstration Scenario: City Delivery

Figure 3 shows a rideshare scenario. The agent picks up and delivers passengers in a city with obstacles, one-way streets and moving traffic. There are also 2 highways which cross over city streets which avoid traffic and move faster than streets. One-way streets run North and South every other street. *Triangles* represent traffic on one-way streets and point in the direction of movement. Traffic is distributed

```
def setup():
    declareTasks['atLocation']
       = ['streetsToLocation', 'highwayToLocation']

    startPropel('appMethod': 'deliverUntilDone',
                'objective': 'updatePlanScore',
                'rolloutCount': 100)

def deliverUntilDone():
    for delivery in range(5): # do 5 deliveries
        pickupLoc, dropoffLoc = getNextDelivery()
        chooseTask('atLocation', pickupLoc)
        pickupCmd(passenger)
        chooseTask ('atLocation', dropoffLoc)
        dropoffCmd(passenger) # collect revenue
    chooseTask ('atLocation', home)
```

Figure 4: Application setup and top level loop

based on a *traffic probability* parameter. When a car in traffic drives off the board, it's replaced by another car entering the other end of the street, to maintain the traffic probability. Obstacles (black squares) are distributed between one-way streets based on an *obstacle probability* parameter.

The agent is the blue circle at cell (18,10). It receives requests to pickup and deliver passengers at random locations, and searches for paths to those locations. The first pickup location is the green dot at (14,11) and the dropoff location is the red dot at (4,6). Highways are shown as green dashed boxes (eastbound highway on row 15 and westbound highway on row 5). Green squares are highway entrances and red squares are exits. On streets, the agent can move one step in any direction, which isn't blocked by another car or an obstacle, and is in-bounds. The agent checks sensors for traffic when entering or crossing one-way streets and waits for it to clear if necessary. The agent and traffic move 1 cell per tick, except on the highway the agent moves 3 cells per tick.

Objective (plan score): The agent's objective is to *maximize profit*. profit = revenue - cost. This is the *plan score* calculated at the end of each rollout and used to update the MCTS tree action-value table. revenue = sum of all delivery fees. The delivery fee equals 5 times the distance between the pickup and dropoff locations. cost = driveDistance + driveTime (for the whole day, including between deliveries and going home). Drive time depends on traffic.

4.1 City Delivery Action Model

Figures 4, 5 and 6 show examples of the procedural action model used in our demo. The application methods explained below are highlighted in blue bold. Methods inherited from Propel are in black bold. Procedural control constructs (like while loops and conditionals), challenging to implement in declarative models, are highlighted in red.

Figure 4 shows the `setup` method. First, two Python methods are declared which achieve the goal pattern `atLocation`. Then `startPropel` starts the processes for integrated planning, learning and execution. This line specifies: the entry point into the application (the *appMethod*) is `deliverUntilDone`, the objective function to be called after each rollout is the Python method `updatePlanScore`, and the rollout count is 100.

`deliverUntilDone` iteratively delivers 5 passengers. After receiving the pickup and dropoff locations,

```
def streetsToLocation (goalLoc):
    with Goal('atLocation' [goalLoc]):
        goalStepCount = 0
        initialDistance = dist(currentLoc, goalLoc)
        while not atLocation(goalLoc):
            direction = chooseValue(validDirections(),
                                    sortByDistance)
            moveCmd(direction)
            goalStepCount += 1
            if goalStepCount > 3 * initialDistance:
                sendFailMsgToPlanner("Loop detected")

def highwayToLocation(goalLoc):
    hwyDirection = None
    if goalIsEast(goalLoc):
        hwyDirection = 'E'
    elif goalIsWest(goalLoc):
        hwyDirection = 'W'
    if hwyDirection:
        streetsToLocation (hwyEntrance(hwyDirection))
        driveHighwayToExit(hwyDirection)
    streetsToLocation (goalLoc)
```

Figure 5: Two methods for going to a location

`chooseTask` is called to choose a method for going to the pickup location. `chooseTask` is a choice point (nondeterministic subroutine call) inherited from Propel. It will choose one of the two methods previously declared for going to a location. The agent can choose between (a) taking the streets the whole way, or (b) taking a highway to cross town (avoiding one-way streets and traffic), then taking streets for the last mile. `choosetask` parameters are a goal pattern (`atLocation`) and parameter `pickupLoc`. After arriving at the pickup location, a primitive action `pickupCmd` is called, where the customer gets into the car. Then `chooseTask` is called again to go to the dropoff location. After completing all 5 deliveries, the agent calls `chooseTask` a final time to go home. `deliverUntilDone` is *executed once in the real world*, but *simulated once per rollout by MCTS*.

Figure 5 shows the two methods previously declared to achieve the goal pattern `atLocation`. `streetsToLocation` is the main subroutine that generates choice points in our search space. It repeatedly calls `chooseValue` to select a valid direction (not blocked by an obstacle, and in-bounds), and then executes the *primitive action* `moveCmd`, until it arrives at the goal location. The choice points use a heuristic, `sortByDistance`, which sorts choices to prefer moves that get closer to the current goal. This heuristic defines a *default policy* which is what the agent would do without the planner. It's the agent's *reactive competence*. It's also the first trajectory explored by MCTS providing in a form of reaction first search (RFS).

Integrated Goal Reasoning and MCTS: Propel uses Python's *context manager* feature to manage a goal stack. *Goals* are used to control MCTS rollout depth and to provide goal-aware behavior. `streetsToLocation` starts by creating a Goal context. The `with` statement creates a context (code block) for a goal object, similar to the way Python's `with open(file)` works. When the `with` block is entered, a Goal specifying the goal location is created and pushed onto a goalstack. `chooseValue`'s heuristic, `sortByDistance`, calculates the resulting distance to the

```
def moveCmd(direction):
    newPos = getNewPosition(currentPos, direction)
    if isEnteringOneWayStreet(currentPos, newPos):
       if isExecution(): # reactive execution
          updateSensorReadings()
          while trafficIsBlockingPosition(newPos):
             sleep(tickDuration) # wait
             driveTime += 1
             updateSensorReadings()
       else: # planning mode
          while trafficIsExpected(trafficProb):
             driveTime += 1
    # continue when traffic is clear
    if isExecution(): # execution mode
       # executing Move takes tickDuration secs
       moveActuators(direction, tickDuration)
       currentLocation = newPosition
       driveDistance += 1
       driveTime += 1
```

Figure 6: `moveCmd` is a *primitive action*

goalLoc for each choice. When `streetsToLocation` exits, the goal context also ends. This triggers the Goal context manager to pop the goal the from the goalstack and to send a *success message* from the application to the planner which includes the current state (agent and car locations). The planner then terminates the rollout, calls the objective method `updatePlanScore` to calculate the plan score for that rollout, and updates the MCTS tree with the rollout score. This is a novel method for terminating rollouts when intermediate goals are achieved (e.g., arriving at a pickup location).

Loop detection: `streetsToLocation` implements loop detection to terminate rollouts if the agent is driving in circles. If the agent chose purely random directions, it would never get to the goal, so this is necessary. As the tree grows, UCT selects random choices and the agent may explore circuitous routes. To prune these plans, we calculate the initial distance between the current location and the goal, then track of how many steps we've taken in the while loop. If the agent moves more than 3 times the initial distance, then this application method sends a *failure* message to the planner, which terminates the rollout with a score of 0 .

The second method shown in Figure 5 illustrates *Hierarchical Task Decomposition*. `highwayToLocation` calls `streetsToLocation` as a subroutine twice, first to go to the highway entrance, and then again after exiting the highway to go "the final mile" to the goal location. After entering the highway, `driveHighwayToExit` moves in the highway direction until it arrives at the exit. The agent moves 3 steps per tick on the highway, but only one step per tick on streets. If the goal is strictly North or South, then the highways are ignored and `streetsToLocation` is called immediately. This shows how Python subroutines are used to model conditional hierarchical task decomposition.

Primitive actions (commands) interface with real world sensors and actuators during execution, or with simulators during planning. Figure 6 shows the main primitive action in our demo, `moveCmd`, where reactive execution is implemented. If traffic is blocking the entrance to a one-way street, the agent cannot move until traffic clears. Traffic is modeled differently by planning and execution. The method `isExecution` is inherited by the Application, enabling it to behave differently for planning and execution. `isExecution` is called to detect if `moveCmd` is running in execution or planning mode. In *execution mode*, it uses real-world sensors and actuators. When entering a one-way street, the agent updates its sensor readings to see if traffic is blocking the street. It waits for traffic to clear before executing the move. During execution the actuators are called, which consumes *tickDuration* seconds. In *planning mode* it uses a stochastic estimate to determine if a car is blocking entrance to the one-way street. `trafficIsExpected` returns `True` based on the *trafficProbability* parameter.

A video demo of City Delivery with a rollout count of 3 can be found at: https://youtu.be/Sc0-1RXgNBA.

5 Python/MCTS Integration

Figure 7 shows how Python choice points are integrated into MCTS. `chooseTask` implements a nondeterministic subroutine call. It looks up the list of tasks declared for a goal pattern (Figure 4), calls `chooseValue` to choose one of the tasks with MCTS, then calls the chosen task. `chooseValue` implements the core of Python/MCTS integration and also and the interleaved planning and execution. `chooseValue` is a *generator* function which yields one next choice on each call. `chooseTask` and `chooseValue` always run in an Application level process.

When `chooseValue` is called in *execution mode*, the Application sends a *planRequest* message to the Controller, then waits for Planner to return a choice. The Executive relays that message to the Planner, which initiates a new round of MCTS. When `chooseValue` is called in *planning mode*, it steps through the MCTS stages *Select, Expand* and *Simulate* (Figure 1). The *update* stage occurs later when rollouts are terminated. After the planner completes all rollouts, it returns the choice which led to the most visited child of the tree root. The Planner sends a *plannerChoice* message to the Executive containing the planner's choice for the next move to execute (Figure 2). The Executive relays that message to the Controller, which relays it to the Application process that was waiting for this response. The paused application resumes execution using the Planner's choice, then continues execution until it reaches another choice point, which starts the cycle again with a new plan request. `chooseValue` takes an optional heuristic parameter to sort the choices. If no heuristic is specified, then choices are chosen randomly.

MCTS Extensions: Integrating MCTS with our procedural action model required two new MCTS stages: *executionReplay*, and *planReplay*. Each rollout starts at the beginning of `deliverUntilDone`, but the current plan request may be for a choice point buried deep in iteration context (it may be halfway through the 3rd delivery). These new stages are used to wind up the Planner's computational context from the start of the method to the current choice point.

executionReplay is used to wind up Planner's program control stack for new MCTS tree at the beginning of each *planRequest*. When the Planner receives a new *planRequest*, the MCTS stage is initialized to *executionReplay*, so the new MCTS tree can catch up to the procedural state of execution. *planRequest* messages include a list of *execution choices*

```
def chooseTask(goalPattern, args, heuristic):
    with Goal(goalPattern, args):
       choices = findMatchingTasks(pattern, args)
       task = chooseValue(choices, heuristic)
       task(args)

def chooseValue(choices, heuristic):
    if isExecution(): # wait for plan
       sendPlanRequestToPlanner()
       choice = waitForPlannerChoice()
       return choice
    else: # Planning mode, do MCTS stages
       if mctsStage == executionReplay:
          choice = pop executionReplay choice
          if no more executionReplay choices:
             mctsStage = 'select'
          return choice
       if mctsStage == 'select':
          if selected node is root:
             mctsStage = 'expand'
          else:
             collect planReplay choices
             mctsStage = 'planReplay'
       if mctsStage == 'planReplay':
          choice = pop planReplay choice
          if no more planReplay choices:
             mctsStage = 'expand'
          return choice
       if mctsStage == 'expand':
          choice = select unexplored choice
       if mctsStage == 'simulate':
          choice = defaultPolicy(state, heuristic)
       return choice
```

Figure 7: `chooseTask` and `chooseValue` implement the Python/MCTS API

which are the previously executed choices, for the planner to trace (replay) the controller's choices up to the current choice point before moving to the *select* stage.

planReplay is required to recreate the choices from the tree root to the selected leaf. When the MCTS select stage chooses a node which is not the root, the stage is set to *planReplay*. The plan choices from root to the selected leaf are collected, and replayed before moving to *expand* stage. These stages were necessary but inefficient. Future work will explore more efficient methods.

Rollouts are terminated (a) when the planner receives a *success* or *failure* message from the application, or (b) when the Application code exits, or (c) when an optional *timeout* occurs, or (d) when the application code throws an *uncaught exception*. *Success* messages are sent automatically when a goal is popped from the goalstack (when a `with Goal` code block exits). When the planner receives a success message, it terminates the rollout, calculates the plan score, and updates the MCTS tree (the action-value table). Applications can also send *failure* messages whenever they detect a state which is not worth pursuing. When a rollout fails, the the tree (action-value table) is updated with a plan score of 0. Software exceptions in the application process are caught by the planner and treated as if a failure message was received.

6 Earth Observing Satellite Application

The core integration of MCTS into `chooseValue` (Figure 7) was initially developed to solve a performance problem in a NASA application. D-SHIELD aims to improve models of soil moisture and wildfire danger (Levinson et al., 2022; 2021). Like the City Delivery application, D-SHIELD includes `chooseValue` statements embedded in Python. Previously, we used a search engine which maintained a complete search space of all explored states to support backtracking, but this failed to scale beyond small data sets. Replacing that search engine with parallel MCTS resolved that prior performance problem, because MCTS only maintains state for a small subset of the search space nodes.

Agent	Time	Choices
S2	28	Observe, Idle
S4	42	Downlink, Idle

Figure 8: The *Choice File* defines discrete decision variables

```
def createSwarmPlan():
    readChoiceFile()
    while decisionVars:
      varName, choices = popNextDecisionVar()
      cmd = chooseValue(choices,choiceSorter)
      updateState(cmd) # update storage & energy
      propagateChoice(cmd) # forward checking
```

Figure 9: Sequential decision procedure with choice points

D-SHIELD now uses both nondeterministic Python and MCTS, but there is no plan execution and there are no exogenous events. It generates a full plan rather than planning one step at a time. Thus, this degenerate use of MCTS was not suitable to demonstrate integrated planning and execution, which is typical of both MCTS and Propel. Detailed discussion of D-SHIELD is outside the scope of this paper, but we summarize it here as a second example of how MCTS and nondeterministic Python are integrated. D-SHIELD uses only the Planner and Application processes shown in Figure 2. The underlying implementation of `chooseValue` as shown in Figure 7 is the same in both D-SHIELD and the City Driver applications.

This planner generates a coordinated 24-hour plan for a swarm of 8 satellites, to select and observe ground targets which maximize science model improvement, and to downlink data when passing over ground stations. There are too many targets to observe them all, and each target is associated with a science value. Storage capacity is limited and observations cannot be made when storage is full.

Figure 8 shows the primary planner input, the *Choice File*. It specifies discrete decision variables with their domains, defining the MCTS search space. For this application, each variable specifies command choices for a given satellite at a given time. It specifies that satellite 2 has a choice of observing a target or remaining idle at time 28, and satellite 3 has a choice of downlinking data or remaining idle at time 42.

With 8 satellites and a 24-hour plan horizon, there are 92,283 binary choice points defined in the Choice File, when a satellite faces a choice to (a) make an observation or remain idle, or (b) downlink data or remain idle. This means the number of unique states in the search space of plans is $2^{92,283}$. For practical purposes, an infinite search space. Our temporal granularity is 1 second, the duration of each command, but we only create decision variables for times

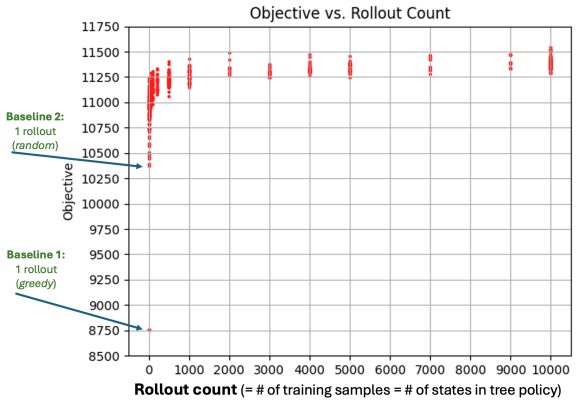

Figure 10: D-SHIELD MCTS results

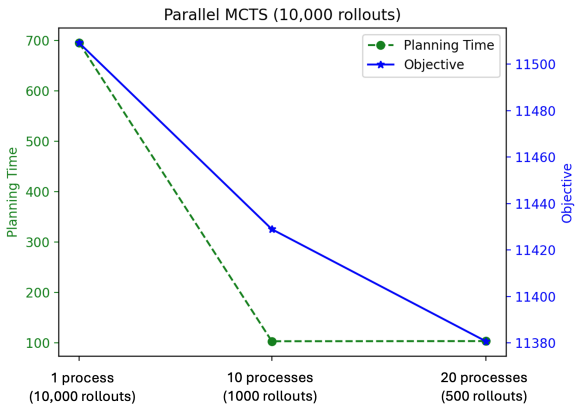

Figure 11: D-SHIELD *Parallel MCTS* results

when there is a choice to make. The planner's job is to maximize the aggregate science reward for the whole swarm, by choosing to observe as many high-value targets as possible, without exceeding storage capacity constraints. Each satellite can store only 60 images, so downlinking data is required to free up storage. Swarm coordination is achieved by ensuring no duplicate observations.

Figure 9 shows the method `createSwarmPlan` which produces a coordinated swarm plan. First, it reads the *Choice File*, defining the *decision variables* which are sorted chronologically, so decisions are made in chronological order. It repeatedly pops the next variable and calls `chooseValue` with the variable's choices. After each choice, the state is updated to reflect data storage and energy changes. Finally, the method `propagateChoice` uses *forward checking* (Levinson et al., 2022; Levinson et al., 2021) to enforce constraints on future choice points. This prunes future choices which are inconsistent with the current choice. This is a very general framework. The choices file may contain any set of discrete command choices, and nearly any constraint can be enforced via *forward checking* in the method `propagateChoice`.

D-SHIELD uses *Root Parallelization*, where parallel processes manage independent MCTS trees (Steinmetz and Gini, 2021). The best plan (highest objective) among all rollouts from all parallel MCTS tree searches is selected. This differs from the standard MCTS method of selecting the single next action which was visited on the most rollouts. Future work will integrate MCTS paralellization into Propel's more general interleaved planning and execution context.

7 Evaluation

All experiments ran on a 2023 MacBook Pro, M2 Pro chip, 32 GB, with 12 CPU cores, using Python 3.8.

Figure 10 summarizes our D-SHIELD MCTS results, showing how the objective score improves with more rollouts. The roullout count equals the number of MCTS nodes tree, equals the number of RL training samples. Figure 11 shows our Parallel MCTS results, comparing the time and objective for doing a total of 10,000 rollouts with 1 vs. 10 vs. 20 parallel processes. This shows good speed improvement up to 10 processes ($\sim$ the number of CPUs on the laptop). However, the objective decreases with more processes because each MCTS tree has fewer training examples. This demonstrates the benefit of learning. The objective monotonically decreases as the number training cases (rollouts) decreases.

Figures 12 through 15 show results from the City Delivery application. All experiments use this same configuration: *Obstacle Probability* = 20 % means when the state is initialized, there is a 20% probability that an obstacle is in any given location. *Traffic Probability* = 50 % means when the state is initialized, there is a 50 % probability of a car in any given location on the one-way streets. *Traffic Probability* is also used during planning to estimate the probability of a traffic delay entering one-way streets. There is no variance because we use a random seed so test results are repeatable (with a single MCTS process).

The *Baseline scenario* follows the greedy default policy, but this greedy policy limits the diversity of MCTS training samples. Using random choices provides a wider range of training experiences for MCTS but it is not practical in this application. The agent can drive in circles without making any progress towards the goal location unless a heuristic biases direction choices towards the goal.

Figure 12 shows how planning time scales linearly with rollout count. It takes 51 minutes to do 100 rollouts, and 508 minutes for 1000 rollouts. This confirms that MCTS solved our most immediate motivating goal, to fix the performance problem with the prior version of Propel which scaled exponentially. Planning time scales linearly with rollouts because it takes constant time to simulate `deliverUntilDone`. We saw similar linear scaling with D-SHIELD.

Figure 13 shows how the *plan score* depends on rollout count. This is the score for `moveCmds` which were *executed* (not simulated plan steps). The *baseline* case (a single rollout) produces a negative plan score: -11. The cost (drive time + distance) exceeded the revenue from deliveries. With only one rollout, the agent strictly follows the same greedy default policy it would execute without the planner. This is the agent's *reactive competence*, what it would do if there was no planner to call for advice. Figure 12 shows a quick rise from the negative baseline to a high score of 71 after 100 rollouts, and then a dip before plateauing at 63.

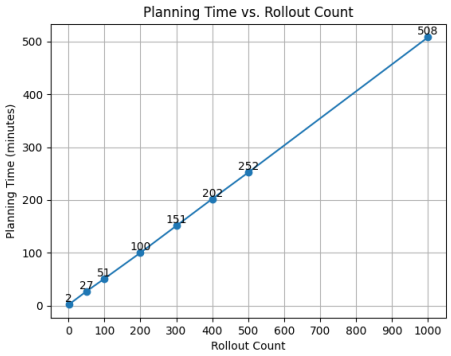

Figure 12: Planning time scales linearly with Rollout count

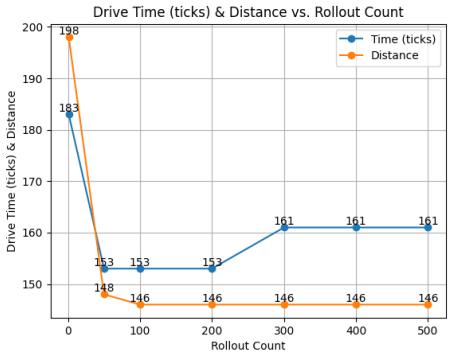

Figure 14: Drive time and distance vs. Rollout count

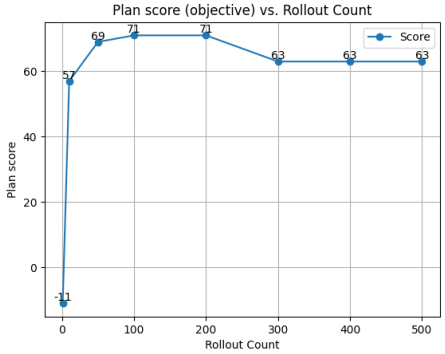

Figure 13: Plan score vs. Rollout count

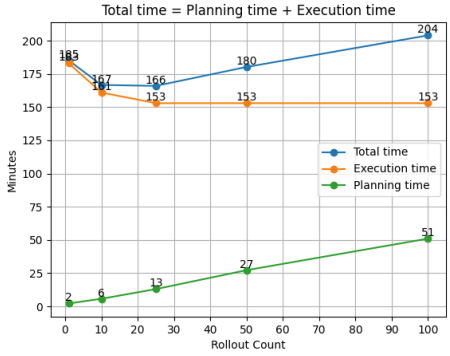

Figure 15: Total time = Planning Time + Execution Time

Figure 14 shows the score's cost components: drive time and distance. These are the times and distances for moves which were *executed* (not *planned* moves). Drive distance monotonically decreases with more rollouts. However drive *time* increases after 300 rolls. Our hypothesis is drive time does not monotonically improve because the stochastic traffic estimate diverges from the execution environment. It usually takes 1 tick to move 1 step, but traffic and highway travel change that. Time may be *more* than distance due to traffic, or *less* than distance because the highway is faster. The baseline case (1 rollout) has drive time (183), *less than* distance (198). This means the default policy used the highway, but still drove farther than other cases.

8 Conclusion

Contributions include two new novel MCTS stages (*executionReplay* and *planReplay*) and novel rollout termination methods (*popping a goalstack* when a *Goal* code block exits, and when *uncaught software exceptions* occur). We are unaware of other work using code blocks to scope rollout depth and terminate rollouts when intermediate goals are achieved.

Figure 15 returns to our long-term motivation. Why do we want a *procedural* action model shared by planning and execution? The answer is: to study complex trades between planning time and execution time, like that shown in Figure 15. This is difficult to study in hybrid systems with rigid boundaries between planning and execution action models. These experiments require a fluid boundary, where the controller has a default policy (reactive competence) to rely on, but can also call the planner in an anytime manner to verify the default policy works in the current situation, and to possibly improve the policy to better fit the situation.

Execution time is the time required to *execute* all plan steps (all `moveCmds`). Each step takes an amount of time called the *tick duration*. $executionTime = driveTime * tickDuration$. *Drive time* is the number of ticks in the *executed* plan. *Tick duration* is a parameter which specifies how long physical actions take to execute in the real world.

In Figure 15, *tick duration* is set to 1 minute instead of the default 0.1 seconds. Planning time is independent from the tick duration. The *baseline* case takes 183 minutes to execute because drive time = 183 ticks and tickDuration = 1 min. Plus another 2 mins of *planning time* to simulate it with a single rollout, for a *total time* = 185 mins. After 25 rollouts (13 minutes of planning), that total time decreases to 166 mins (19 minutes less than baseline). Additional time spent planning only increases that total time, although it remains less than baseline case until about 60 rollouts. Figure 15 shows planning can provide a net benefit. What do these trades look like for different applications? These are broader research questions that motivate this work.

9 Acknowledgments

D-SHIELD is supported by NASA's Earth Science Technology Office through the Advanced Information Systems Technology and the FireSense Technology programs.

References

Brooks, R (1991), "Intelligence Without Reason", in Proceedings of the 1991 International Joint Conference on Artificial Intelligence, pp. 569–595.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtos, N., Carreras, M.. 2015. Rosplan: Planning in the robot operating system. Proceedings International Conference on Automated Planning and Scheduling, ICAPS. 2015.

Coulom, R. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. 5th International Conference on Computer and Games, May 2006, Turin, Italy.

Drummond, M., Bresina, J., Swanson, K., Levinson, R. 1993. Reaction-First Search: Incremental Planning with Guaranteed Performance Improvement. Proc. of IJCAI-93. Chambrey, France.

Ghallab, M., Nau, D., Traverso, P. 2016. Automated Planning and Acting. Cambridge University Press.

Ingrand, F, Chatilla, R. Alami, R, Rober, F. 1996. PRS: a high level supervision and control language for autonomous mobile robots. In IEEE Int'l Conf. on Robotics and Automation.

Kim P., Williams B., Abramson M., 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. IJCAI '01. AAAI Press, Menlo Park, CA.

Kocsis, L., Szepesva ri, C. (2006). Bandit Based Monte-Carlo Planning. In Fu rnkranz, J., Scheffer, T., Spiliopoulou, M. (Eds.), Proceedings of the Seventeenth European Conference on Machine Learning, Vol. 4212 of Lecture Notes in Computer Science, pp. 282–293, Berlin/Heidelberg, Germany. Springer.

Levinson, R., Niemoeller, S., Nag, S., Ravindra, V. (2022, June). Planning Satellite Swarm Measurements for Earth Science Models: Comparing Constraint Processing and MILP Methods. In Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 32, pp. 471-479).

Levinson, R., Nag, S., and Ravindra, V. 2021. Agile Satellite Planning for Multi-payload Observations for Earth Science, Proceedings of the International Workshop on Planning and Scheduling for Space (IWPSS). 2021.

Levinson, R., 2020. Integrated Planning, Execution and Goal Reasoning for Python. ICAPS 2020, workshop on Integrated Execution and Goal Reasoning (IntEx/GR).

Levinson R. 2005. Unified Planning and Execution for Autonomous Software Repair, ICAPS 2005, Workshop on Plan Execution. https://icaps05.icaps-conference.org/documents/ws-proceedings/ws7-allpapers.pdf

Levinson, R. 1995. A General Programming Language for Unified Planning and Control. Artificial Intelligence, Vol. 76. Special Issue on Planning and Scheduling. https://www.sciencedirect.com/science/ article/pii/000437029400075C

Muscettola, N., G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt, 2002. "IDEA: Planning at the core of autonomous reactive agents," in Proc. of the 3rd International NASA Workshop on Planning and Scheduling for Space, 2002

Muscettola, N., Dorais G.., Fry,C., Levinson, R., Plaunt, C. 2000. A Unified Approach to Model-Based Planning and Execution. The 6th Int'l Conf. on Intelligent Autonomous Systems. Venice, Italy.

Neufeld, X., 2020. Long-term planning and reactive execution in highly dynamic environments, Dissertation, Otto-von-Guericke-Universität Magdeburg. http://dx.doi.org/10.25673/35675.

Neufeld, X., Mostaghim, S., Brand, S. (2018). A Hybrid Approach to Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees. Artificial Intelligence and Interactive Digital Entertainment Conference.

Patra, S., Mason, J., Ghallab, M., Nau, D., Traverso, P., 2021. Deliberative acting, planning and learning with hierarchical operational models, Artificial Intelligence, Vol. 299

Patra, S., Ghallab, M., Nau, D., Traverso, P. 2019. Acting and planning using operational models. In AAAI. AAAI Press.

Steinmetz, E., Gini, M., 2021. "More Trees or Larger Trees: Parallelizing Monte Carlo Tree Search," in IEEE Transactions on Games, vol. 13, no. 3, pp. 315-320, Sept. 2021, doi: 10.1109/TG.2020.3048331

Sutton, R., Barto, A., 2018. Reinforcement Learning: An Introduction, second edition. MIT Press.

Vodopivec, T., Samothrakis, S., Ster, B., 2017. On Monte Carlo Tree Search and Reinforcement Learning. Journal of Artificial Intelligence Research, volume 60. pp 881-936.