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

Richard Levinson

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 the 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 works in a simulated rideshare scenario in an urban setting, and present preliminary experiments as a proof of concept.

1 Introduction

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. In the standard hybrid approach, a planner using a declarative model is connected to a reactive execution system using a procedural model (Cashmore et al. 2015).

Procedural models include essential control flow constructs like while loops and if statements (conditionals). Translating between these two action representations increases complexity due to the need to translate between the two languages and maintain two different behavior models. 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 the boundary between 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 could not scale to large problems. 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. Preliminary results indicate it does solve the prior performance problem and demonstrate proof of concept.

The purpose of this paper is to present the new system and explain how MCTS is used to integrate planning, learning and execution in nondeterministic Python. This means the Python code contains choice points, nondeterministic assignment statements and subroutine calls, where the choice is selected by MCTS rather than the Python developer. This paper focuses on explaining how MCTS is integrated into arbitrary Python code extended with choice points. We present MCTS extensions required and demonstrate a working example as proof of concept.

2 Related Work

2.1 Integrated Planning and Execution

We present a new version of Propel: The Program Planning and Execution Language. All Propel versions support 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 new version, dubbed "Propel-MCTS" integrates nondeterministic Python with Monte Carlo Tree Search (MTCS). Nondeterministic choice points (assignment statements and subroutine calls) may be embedded anywhere in Python, and 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 Reactive Plan Language (McDermott, 1993), IDEA (Muscettola et al., 2000; 2002), 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 (controller) which shared the same action model. The controller can operate without the planner by falling back on 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.

Recent work on long-term planning and reactive execution by (Neufeld 2020) and on Operational Models (Patra et al. 2021) use procedural models for both planning and execution, and they use MCTS.

(Neufeld 2020) presents integrated planning and reactive execution for video games with dynamic environments requiring millisecond reactions. He presents a hybrid planning and execution system where a long-term HTN planner is integrated with a reactive system. He compares integration of the HTN planner with two reactive systems: MCTS vs. Behavior Trees, a relative of the behavior-based robotics like the subsumption architecture (Brooks 1991). He uses MCTS as his reactive execution system and has a slower HTN as the planner, while we use MCTS as our planner and have a default policy as our reactive execution system.

The *Operational Models* presented by (Patra et al., 2021) shares similar motivations with Propel for planning with a procedural model. They present *UTC for Operational Models* (UPOM), which uses a specialized reactive execution system called RAE (Ghallab et al, 2016) and is related to the Procedural Reasoning System (PRS) (Ingrand et al, 1996). Procedures can be defined with choice points for some subroutine calls, to be refined later using MCTS. A key difference is their use of a highly specialized AI language as their procedural modelling language. Their choice points are restricted to nondeterministic subroutine calls, while Propel also has conditional assignment statements, chooseValue.

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 and Reinforcement Learning are strongly related, as explained by (Vodpivec et al., 2017) and (Sutton and Barto, 2018). 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. It focuses on states near the current state and does not attempt to create a general policy to use in any future state encountered. Planning and learning both update an action-value function based on experience.

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 execution. After the best action is executed, the "opponent" (if it's a board game) or the environment (if it's the real world) make exogenous state changes (transition 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.

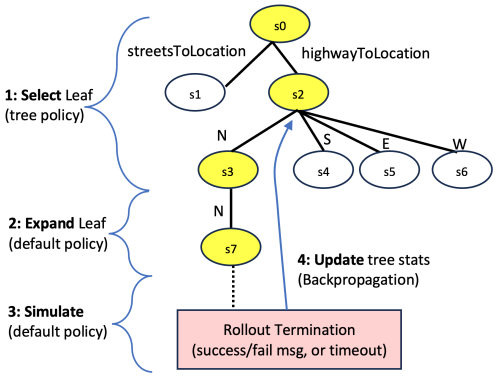

Figure 1: MCTS tree for our application.

Figure 1 shows how MCTS works for our demo scenario. Nodes are states and edges are action choices. The first branch is a strategic choice between taking a highway or streets to a given location. The other branches are tactical choices between moving North, South, East or West.

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. 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 reward for that sequence of actions is backpropagated to update the tree statistics. The number of visits and average 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 all the intermediate states in the simulation stage.

3 Demonstration Scenario: City Delivery

Figure 2 shows the demonstration scenario used throughout the paper: A city rideshare service. Our agent is a driver that repeatedly picks up and delivers passengers, to and from random locations. The scene is a city with obstacles, one-way streets with moving traffic, and 2 highways which cross over center city streets and avoid traffic. On each tick the agent can move one step in any direction (N,S,E,W), which is not blocked by an obstacle and is in-bounds.

The city is a 20 x 20 grid with one-way streets running North and South every other street. *Triangles* represent traffic on one-way streets and point in the direction of movement. Traffic is distributed on one-way streets based on a *traffic probability* parameter. Obstacles (black squares) are distributed in between the one-way streets, based on an *obstacle probability* parameter. West and Eastbound highways

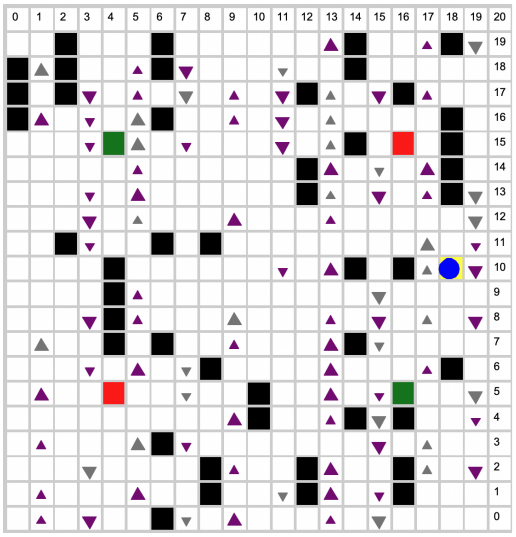

Figure 2: City Delivery initial state

exist (green entrances and red exits).

The agent receives requests to pickup and deliver passengers between the one-way streets (where there is no traffic). The agent must choose the most efficient path between each pickup and delivery location and avoid hitting other cars, which move 1 step forward on each tick. The agent checks it's sensors for traffic when entering a one-way street and waits for it to clear if necessary.

The agent is the blue circle at cell (18,10) it's *home location*. The pickup location is green dot at (14,11) and the dropoff location is the red dot at (4,6). The green squares are highway entrances (4,15) and (16,5) and the red squares are highway exits (16,15) and (4,5). The Eastbound highway is on row 15 and the Westbound highway on row 5.

Traffic and the agent move at a rate if 1 cell per tick, except on the highway the agent moves at 3 cells per tick. Sometimes the agent must wait for traffic to clear before moving, which increases the drive time metric. When a car in traffic drives off the board, it's replaced by another car entering the same street at the other end to maintain the traffic probability distribution. This scenario is designed strictly to illustrate features of Propel's integration with Python and MCTS and to demonstrate how all the pieces fit together to provide integrated planning, learning, and execution.

Objective (plan score): The agent's objective is to maximize profit, defined as follows:

- 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 = $\sum_{d \in deliveries}$ 5 * dist(d.pickupLoc, d.dropoffLoc). The revenue from each delivery = 5 times the distance between the pickup and dropoff locations.
- cost = driveDistance + driveTime (for the whole day, including between gigs and going home). Drive time is a function of the traffic probability.

```
1 def setup():
2   declareTasks['atLocation ?goalLoc'] = ['streetsToLocation',
                                           'highwayToLocation']
3   startPropel('appMethod': 'deliverUntilDone',
                'objective': 'updatePlanScore',
                'rolloutCount': 100)

4 def deliverUntilDone():
5   for delivery in range(5): # repeat for 5 deliveries
6       pickupLoc, dropoffLoc = startNextDelivery()
7       achieve('atLocation ?pickupLoc', pickupLoc, 'random')
8       pickupCmd(person/package)
9       achieve('atLocation ?dropoffLoc', dropoffLoc, 'random')
10      dropoffCmd(person/package) # collect revenue for job
11  achieve ('atLocation ?homeLoc', home, 'random')
```

Figure 3: Application setup and top level loop

4 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 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 `achieve` (nondeterministic subroutine call for a given goal pattern). 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. Programs with choice points may be considered underspecified programs, which require refinements before execution. MCTS searches through the space of program variations defined by choice points to maximize goal-driven rewards.

4.1 City Delivery Action Model

Figures 3, 4 and 5 show examples of the procedural action model used for planning, learning and execution in our demo. The application creates a subclass of Propel to inherit MCTS methods and behavior. The application-defined methods explained below are highlighted in blue bold. Methods inherited from Propel are in black bold. Procedural control terms, which are challenging to implement in declarative models, are highlighted in red.

In the setup method (Figure 3), line 2 declares two python methods which achieve the goal pattern `atLocation`. Line 3 starts Propel's processes for integrated planning, learning and execution, and declares the Python method `updatePlanScore` as the objective function, which is called after each rollout to update the tree policy action-value table. This line specifies that the entry point into

```
12 def streetsToLocation (goalLoc):
13    with Goal('atLocation' [goalLoc]):
14       goalStepCount = 0
15       initialDistance = distance(currentLoc, goalLoc)
16       while not atLocation(goalLoc):
17          direction = chooseValue(validDirections(state),
                                    sortByDistance)
18          moveCmd(direction)
19          goalStepCount += 1
20          if goalStepCount > 3 * initialDistance:
21             sendFailMsgToPlanner("Loop detected")


22 def highwayToLocation(goalLoc):
23     hwyDirection = None
24     if goalIsEast(goalLoc):
25        hwyDirection = 'E'
26     elif goalIsWest(goalLoc):
27        hwyDirection = 'W'
28     if hwyDirection:
29        streetsToLocation (highwayEntrance(hwyDirection))
30        driveHighwayToExit(hwyDirection)
31     streetsToLocation (goalLoc)
```

Figure 4: Core subroutine producing most choice points

the application, the *appMethod*, is `deliverUntilDone`. This method will be executed once in the real world but simulated once per rollout by MCTS.

`deliverUntilDone` iteratively delivers 5 passengers. After receiving the pickup and dropoff locations on line 6, `achieve` is called on line 7 to choose a method for going to the pickup location. `achieve` is a choice point (nondeterministic subroutine call) inherited from Propel. It will choose one of the two methods declared on line 2 for going to a location: the agent can choose between (a) taking a highway to cross town to avoid one-way streets and traffic, then using streets to go the last mile, or (b) taking the streets the whole way. `achieve` parameters are a goal pattern (*'atLocation ?pickupLoc'*) with parameters *pickupLoc*. The choiceSorter is specified to be random. After arriving at the pickup location, a primitive action `pickupCmd` is called on line 8, where the customer gets into the car. Then *achieve* is called again on line 9 to go to the dropoff location. After completing all deliveries, the agent calls achieve again to goes home (line 11).

`streetsToLocation` (Figure 4) is the core subroutine which generates most of the choice points in our search space. It repeatedly calls `chooseValue` on line 17, to select a valid direction (in-bounds, and not blocked by an obstacle), and then executes the primitive action `moveCmd` (line 18), until it arrives at the goal location.

Default policy: The choice points in `streetsToLocation` use a heuristic which sorts choices so that moves which get closer to the goal are preferred (line 17). This *default policy* is what the agent would do if there was not planner, its the agent's *reactive competence*. Its also the first trajectory explored by MCTS resulting in a form of reaction first search.

Integrated Goal Reasoning and MCTS: Propel uses Python's context manager feature to manage a simple goal stack. Goals are used to control MCTS rollout depth. `streetsToLocation` starts by creating a Goal context using the *with* statement on line 13. The python *with* statement creates the scope of the goal object, similar to the way *with open(file)* works. When the Goal context is entered (line 15), a Goal object is created and pushed onto the goalstack. When this method exits (when *streetsToLocation* exits), the goal context also ends. This triggers the context manager's exit method which pops the goal the from the goalstack and sends a "success" message from the application to the planner which includes the application's current state. The planner then terminates the rollout and calls the application's objective method *updatePlanScore* to evaluate the current state and calculate the plan score for the current rollout. The planner then updates the MCTS tree with the plan score for that rollout. This is the MCTS update (backpropagate) stage. The goalstack is entirely managed by the application code, but the success message is inherited from Propel.

Loop detection: `streetsToLocation` implements a simple loop detection method to terminate the rollout if the agent is lost or driving around in circles. If the agent chose purely random directions, it would never get to the goal except by dumb luck, so this is a possibility. We use a greedy default policy which perfes directions that minimize goal distance. However, as the tree grows, UTC selects more random choices and the agent may explore extremely circuitous routes. To prune these plans, we first calculate the initial distance between the current location and the goal (line 15), then keep track of how many steps we've taken in the while loop (line 19). If we have taken more than 3 times the initial distance, then this application method sends a fail message to the planner, which terminates the rollout with a score of 0 (lines 20 and 21).

`highwayToLocation` calls `streetsToLocation` as a subroutine, first to go to the highway entrance (line 27). After arriving at the entrance, `driveHighwayToExit` moves in the highway direction until it arrives at the highway exit. Moving on the highway is faster and takes less time. It moves 3 steps each tick, while only one step per tick on the streets. After arriving at the highway exit, `highwayToLocation` calls the workhorse `streetsToLocation` again to get to the goal location (line 31). If the goal is strictly north or south (in the same column), then the highways are ignored and `streetsToLocation` is called immediately. This illustrates how Python subroutines are used to model hierarchical task decomposition in the action representation, which can be difficult with declarative models.

Primitive actions (commands) interface directly with real world sensors and actuators during execution, or with a simulator during planning. Figure 5 shows the main primitive action in our demo, `moveCmd`, and is where reactive execution is implemented. If traffic is blocking the entrance to a one-way street, the move temporarily "fails", meaning agent doesn't move until traffic clears. Traffic on one-way streets is modeled differently by planning and execution. The Application inherits methods `isPlanning` and `isExecution` which enable the app to branch for different behavior in planning and execution mode. On line 35, `isExecution` is used to detect if it's running in execution or planning mode. In *execution* mode, it uses real-world sensors and actuators. It waits a tick for traffic to move one step, then updates sensor readings to see if traffic is still block-

```
32 def moveCmd(direction):
33     newPos = getNewPosition(currentPos, direction)
34     if isEnteringOneWayStreet(currentPos, newPos):
35        if isExecution(): # reactive execution
36           updateSensorReadings()
37           while trafficIsBlockingPosition(newPos):
38              sleep(tickDuration) # wait
39              driveTime += 1
40              updateSensorReadings()
41        else: # planning
42           while isTrafficExpected(trafficProbability):
43              driveTime += 1

44     # continue when traffic is clear
45     currentLocation = newPosition
46     driveDistance += 1
47     driveTime += 1
```

Figure 5: `moveCmd` is a *primitive action*

ing entrance to the one-way street. It continues waiting until traffic is clear and it may execute the move.

in Planning mode it uses a stochastic estimate to determine if a car is blocking entrance to the one-way street. It stochastically models traffic using the *trafficProbability* parameter (and doesn't wait a tick). In this application, traffic delays (or other exogenous events) are equivalent to our "opponent's move" in MCTS 2-player games.

In general, we want only the lowest level methods to behave differently for planning and execution in order to maximize the planner's coverage (model) of reactive execution behavior.

4.2 Propel Architecture: Three process levels

Figure 6 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 application code, using MCTS to simulate many variations (MCTS rollouts) of the application. The Controller and Planner processes can start/stop, and pause/resume their respective application processes, and receive messages from them and from the Executive.

Application Processes: ll application specific code and data state are defined and managed by the application layer code. For our example city delivery service, this means the Application level contains all data and methods which know about streets, cars and customers. 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. The rollouts are performed in sequence, so only one application process exists at a time and its terminated at the end of the rollout. Future work will add parallel MCTS to leverage parallel CPUs.

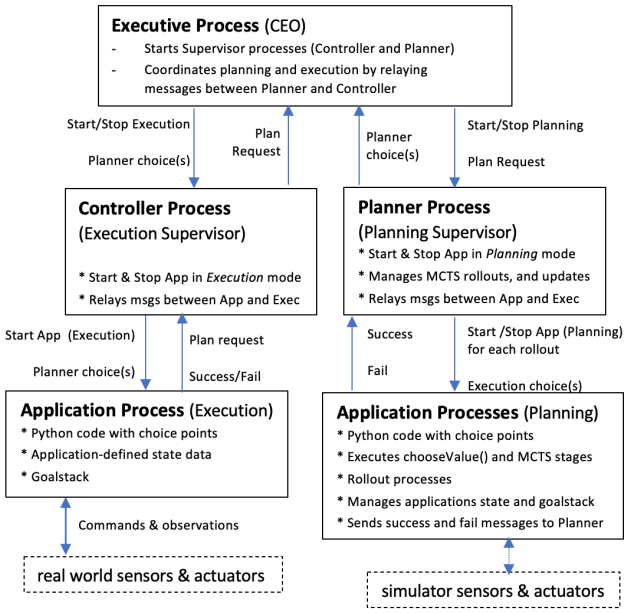

Figure 6: Propel's three process levels: Executive, Supervisors (Controller and Planner) and Application

4.3 Application/Propel API

Applications create a subclass of Propel to inherit MCTS methods and behavior. The `chooseValue` and `achieve` methods are inherited from Propel but run in the App processes. Within each rollout, all MCTS stages run in the Application processes.

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 entirely by application code to track the location of the agent and cars in traffic. The application passes the state object to the planner when a terminal state to be used in the objective function for the MCTS update stage.

A message API allows the application process to inform it's supervisors (planner or controller processes) about events to be handled by Propel. A *Success* message is sent to the planner when an application-specific goal is achieved. *Success* messages tell the planner to terminate the rollout, calculate the score and update the tree with new statistics (MCTS update stage). A *Failure* message is sent to planner to terminate rollout under application-specific conditions. *Failure* messages tell the planner to terminate the rollout with a score of 0 and update the tree with new statistics (MCTS update stage). The *Executive* process relays messages between the Planner and Controller, enabling communication between them.

With MCTS, the application method (`deliverUntilDone` in our example) is simulated, starting from the beginning of the method, on every rollout with different command choices selected on each simulation.

```
48 def chooseValue(choices, heuristic):
49   if isExecution(): # wait for plan
50      sendPlanRequestToPlanner()
51      choice = waitForPlannerChoice()
52      return choice
53   else: # Planning mode, do MCTS stages
54      if mctsStage == executionReplay:
55         choice = pop executionReplay choice
56         if no more executionReplay choices:
57            mctsStage = 'select' select"
58         return choice
59      if mctsStage == 'select':
60         if selected node is root:
61            mctsStage = 'expand'
62         else:
63            collect planReplay choices
64            mctsStage = 'planReplay'
65      if mctsStage == 'planReplay':
66         choice = pop planReplay choice
67         if no more planReplay choices:
68            mctsStage = 'expand'
69      if mctsStage == 'expand':
70         choice = select unexplored choice
71      if mctsStage == 'simulate':
72         choice = defaultPolicy(state, heuristic)
73      return choice


74 def achieve(goalPattern, args, heuristic):
75    with Goal(goalPattern, args):
76       choices = findMatchingTasks(pattern, args)
77       task = chooseValue(choices, heuristic)
78       task(args)# streets or highway to location
```

Figure 7: `chooseValue` implements MCTS stages Select, Expand and Simulate. `achieve` implements a nondeterministic subroutine call using `chooseValue`

4.4 Python/MCTS Integration

Adapting MCTS for use with arbitrary choice points embedded in Python required extensions. Figure 7 shows how `chooseValue` implements the core of Python/MCTS integration.

`chooseValue` is a generator function which yields one next choice on each call. It implements interleaved planning and execution. Each call to choose value returns a single next choice to the application code. To work with our procedural action model, we added two new MCTS stages: *executionReplay*, and *planReplay*. The MCTS stage is initialized to executionReplay when the planner receives a new planRequest, so the MCTS tree can catch up to the procedural state of execution. Plan Replay is required during planning, to recreate choice sequence from root to leaf.

`chooseValue` always runs in the App process. When `chooseValue` is called in *execution* mode, the App process sends a *planRequest* message to the controller and waits until the planner returns a choice. The plan request includes a list of *execution choices* which are the previously executed choices, that the planner uses to trace (”replay”) the controller’s choices up to the current choice point. The Executive relays the plan request message from the controller to the planner, which initiates a new round of MCTS.

When `chooseValue` is called in *planning* mode (while servicing a planRequest), is reentrant and proceeds through the MCTS stages *Select, Expand and Simulate*, augmented with new stages *planReply* and *executionReplay*. *executionReplay* is used to wind up program control statck stack with new MCTS tree after a move is executed (at the start of a new MCTS tree). *planReplay* is used when the selected node to be expanded is not the root. This stage replays plan choices from root of the tree to the selected leaf during each rollout. These replay stages are a form of policy patch or extension.

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 *planResult* message to the Executive containing the planner’s choice for the next move to execute. The Executive relays that message to the controller, which then relays it to the application process (execution) which was waiting for this response. The paused application resumes execution using the planner’s choice. It then continues executing the application code until it reaches another choice point, which starts the cycle again.

`chooseValue` takes an optional heuristic parameter to sort the choices. This is typically an application-specific greedy heuristic, called a choice sorter, which sorts choices in descending preference order. This local heuristic is a greedy default policy. If no heuristic is specified, then choices are chosen randomly.

Figure 7 shows the `achieve` method, inherited from Propel. Its a nondeterministic subroutine call which chooses from methods declared to match a given goal pattern (line 2 in our example). achieve is a wrapper around the base level `chooseValue` method in Propel. In this example, line 77 will choose one of the methods declared in line 2.

Rollouts are terminated when the planner receives a *success* or *failure* message from the application, or when the application code throws an *uncaught exception*, or when an optional *timeout* occurs.

A *success* message is the most common rollout termination method, and happens whenever a goal is popped from the goalstack. Success messages are sent automatically when a Goal is achieved (when the `with Goal` statement scope exits, see line 13). When the planner receives a success message, it terminates the rollout, calculates the plan score, and updates the MCTS tree (the action-value table) with the rollout results, which is the MCTS update stage (Figure 1).

In our application, *failure* messages indicate loop detection and are sent when the number of iterations in `streetsToLocation` exceeds 3 times the initial distance to goal (Figure 4, line 21). The application can send a failure message whenever it detects a state which is not worth pursuing. When a rollout fails, the plan score is 0 and the tree (action-value table) is updated.

When a *Python software exception* occurs, the rollout is terminated and treated like a failure message was received with a plan score of 0. Each rollout runs in it’s own process, and software exceptions in the application process are caught by the planner using try/catch. This is not shown in our demo to save space, but here is an example if we introduce a minor bug into the `highwayToLocation` (Figure 4). If we remove the conditional `"if hwyDirection"` on line 28, and make the next lines (29 and 30) unconditional and always executed, then the method crashes on line 29

when it calls `highwayEntrance` with `hwyDirection = None`. This happens when the pickup and delivery locations are directly North or South of each other (same column). This was a real bug in an early version.

The application may specify an optional MCTS *timeout* (not in our demo). The Planner process terminates the application process when a timeout occurs, then calculates the plan score at that point, and updates the tree statistics.

5 Experiment Design

Our experiment is intended to illustrate how the system works and to demonstrate proof of concept. We added MCTS to Propel to see if it improves performance over the prior version of Propel, so that is the most immediate experiment goal. We are also interested in broader hypotheses. The first is that MCTS can extend the operating conditions when the default policy fails. It can solve problems which cannot be solved by the default policy (for example driving into a deadend or driving in circles). The second broad hypothesis is that MCTS can improve plan scores (optimality) even for cases which are solvable by the default policy. Most broadly, we are interested in studying the trade between the cost of planning time and the reward of more efficient plans.

Our experiments all use the same scenario configuration.

The *Obstacle Probability* = 20 %. This means when the board state is initialized, there is a 20% probability that there is an obstacle in any given location. This applies only to even numbered columns (0,2,4, etc.) because there are no obstacles on the one-way streets (odd-numbered columns).

The *Traffic Probability* = 50 %. This means when the board state is initialized, there is a 50 % probability that there is a car in any given location. This applies only to odd numbered columns because those traffic is only on one-way streets (odd-numbered columns). *Traffic Probaility* is also used during simulation to estimate the probability that traffic will delay entrance into a one-way street.

The independent variable is the rollout count. **Dependent variables** include performance metrics *planning time* and *total time* ($totalTime = planningTime + executionTime$), and the plan quality metrics: *Plan score (objective), drive distance, drive time*.

Baseline scenario: In the baseline scenario, there is no planner. Execution uses the default policy as it's reactive competence in the absence of planner input. This means execution will follow the same trajectory as MCTS with a single rollout. This is a form of Reaction First Search since the reactive (default) behavior is the first trajectory explored by the search engine. This default policy is greedy, because it always chooses actions which minimize distance to the goal location. The down side is that this greedy policy limits the diversity of MCTS training samples, especially when the tree only captures a small part of the full rollout trajectory. Using random choices would provide a wider range of training experiences for MCTS but it is not practical in this application. If all choices are random, the search space is infinite because the agent can in circles with no progress to goal. Some heuristic needs to bias move direction choices towards each goal location.

Computing resources: All experiments were run on a 2023 MacBook Pro, M2 Pro chip, 32 GB, with 12 total CPU cores (8 performance and 4 efficiency). All code is written in Python 3.8.

6 Experiment Results

Figures 8 through 11 show preliminary experiment results. The x-axis is the rollout count which is a Propel parameter provided by the application (line 2). Figure 8 how planning time scales linearly with the rollout count. It takes 51 minutes to do 100 rollouts, and 508 minutes (8.75 hours) to do 1000 rollouts.

Execution time is the time it takes the agent *execute* all plan steps (all `moveCmds`). Executing each move takes a fixed amount of time called the *tick duration*. $ExecutionTime = driveTime * tickDuration$, because drive time is the number of ticks in the plan. Drive time is shown in Figure 10. *Tick duration* is an adjustable model parameter (default = 0.1 second).

Figure 8 shows that planning time scales linearly with rollout count. This chart provides evidence that MCTS solved our most immediate motivating goal, to fix the performance problem with the prior version Propel which scaled exponentially. Total time scales linearly with the number of rollouts because it takes constant time to execute `deliverUntilDone`, and memory usage also remains constant. Results with 1000 rollouts were identical to those with 500, so we omit that case from other charts for readability.

Figure 9 shows how plan score depends on rollout count. This is the score for steps which were executed (not hypothetical 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 a 1 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. Since this is the first plan MCTS considers, it is doing a form of Reaction First Search.

We see a quick rise from the baseline of -11 to a maximum score of 71 after 100 rollouts, and then it dips before plateauing at 63. The highest scores were produced with 100 and 200 rollouts. We expected the score to monotonically increase with rollout count.

Figure 10 shows a possible explanation why the score drops by breaking the score down into its components, drive distance and drive time. These are the times and distances for moves which were executed (not hypothetical plan steps). This shows that drive distance *does* monotonically decrease (improve) with more rollouts, as we expected. However, drive time does not improve monotonically, because it increases from 153 to 161 after 300 rolls. Although these high rollout tests don't have the highest score, do share the minimum drive distance (146) with the highest scoring tests.

Nominally, it takes 1 tick to move 1 step, but traffic and highway travel change that math. Time may be *more* than distance due to traffic delays, and Time may be *less* than distance because highway speed is faster than streets. Note

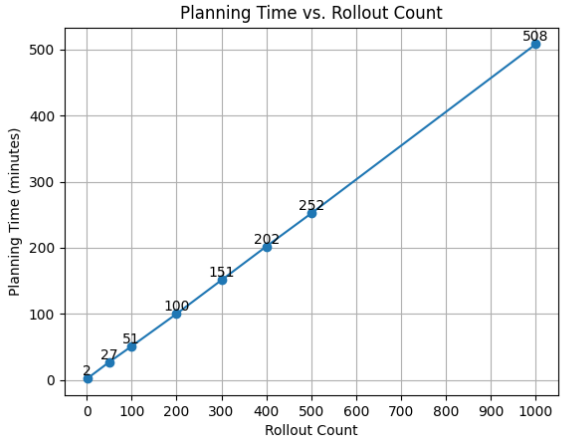

Figure 8: Planning time scales linearly with Rollout count

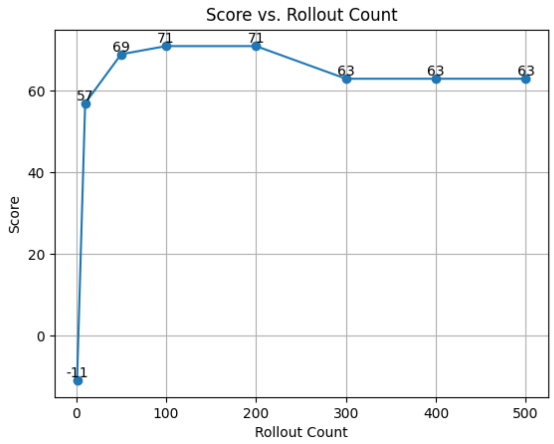

Figure 9: Plan score vs. Rollout count

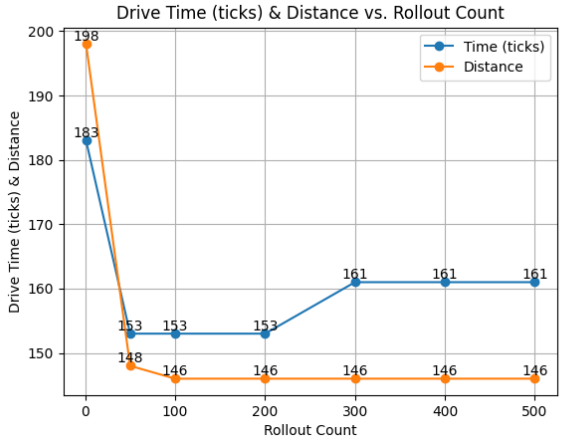

Figure 10: Drive time and distance vs. Rollout count

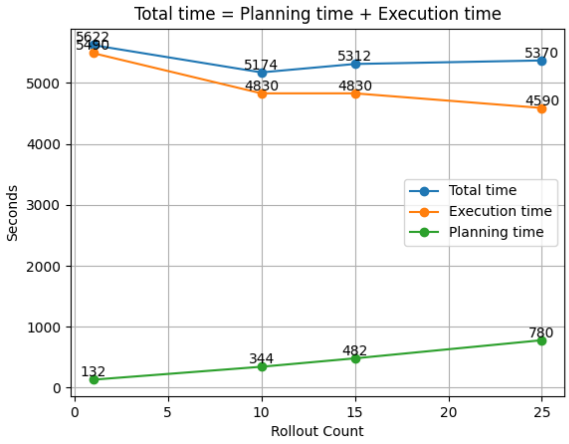

Figure 11: Total time = Planning Time + Execution Time

in the baseline case with 1 rollout, the drive time (183) is *less than* distance (198). This means the default policy exploited the highway speedup, but it still ended up driving much farther than any other case.

Our working hypothesis for why drive time does not monotonically improve is perhaps the stochastic traffic estimate used by the planner diverges from the execution environment in a way which biases MCTS as rollouts increase.

7 Conclusion

Figure 11 returns to our original motivation. Why do we want a unified *procedural* action model for both planning and execution? The answer is so that we can run experiments like this, which explore complex trades between planning time, execution time, and the sum total of both. 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 will work in in new situations, and to improve the policy when possible.

Planning time remains the same regardless of the tick duration which only affects the duration of each *executed* move. In Figure 11, the *tick duration* is set to 5 minutes instead of the default 0.1 seconds. This shows the complex trades between planning, execution and total time when it takes longer than 0.1 seconds to execute physical actions. The *baseline case* (default policy) takes 5,490 seconds to execute the plan + 132 seconds to simulate it with a single rollout, for a total time of 5,622. After 10 rollouts and 344 seconds of planning, that total time decreases to 5,174, which is 7.46 minutes less than the baseline. Any additional time spent planning after that only increases that total time, although it remains better than the default case.

This shows planning can provide a net benefit. How much planning is required before that net benefit kicks in? At some point (for example after 1000 rollouts which take 8 hours), it is not worth the extra planning time. When does planning become a net loss? What do these trades look like for different applications? These are the broader research questions which motivate this work.

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. 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

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

Mcdermott, D. 1993. A Reactive Plan Language. https://www.researchgate.net/publication/2648978$_{A}R{eactive}_{P}lan_{L}anguage$

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

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.