Prerequisites

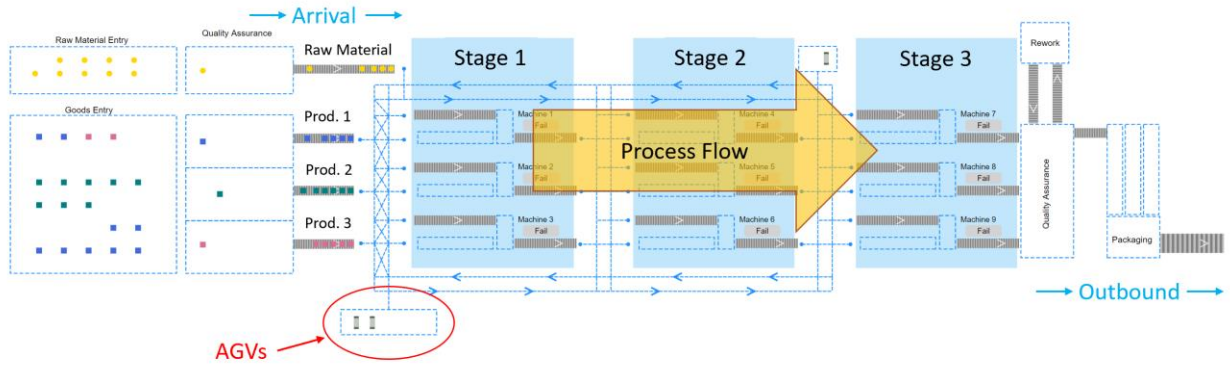
1. Install the Pathmind Helper
2. Complete the Get Started tutorial

Introduction

This tutorial will walk you through a reinforcement implementation in an automated guided vehicle (AGV) fleet management problem, which incorporates Multi-Controller, Shared Policy reinforcement learning technology. The goal of the tutorial is to introduce you to Multi-Controller, Shared Policy in a fleet-management problem and showcase 3 advantages that reinforcement learning has over any heuristic: (1) its implementation is straightforward and simple; (2) it yields a policy that adapts when faced with variability; and (3) it yields emergent behavior that is difficult to capture with even the most complex heuristics.

Simulation Overview

A fleet of automated guided vehicles (AGVs) optimizes its dispatching routes to maximize product throughput in a manufacturing center. When component parts arrive to be processed, they must be brought to the appropriate machine according to a specific processing sequence. In theory, AGVs can increase the delivery rate at correct locations, maximizing the output rate of finished products. **In practice, however, coordinating AGV tasks in the face of variable processing times, maintenance shutdowns and supply arrivals can be difficult.** Inefficiency in AGV fleet management can result in lost time and underutilization of resources. In this example, an AnyLogic simulation of the processing warehouse demonstrates how reinforcement learning can be used to outperform a route-optimizing heuristic by 50%.

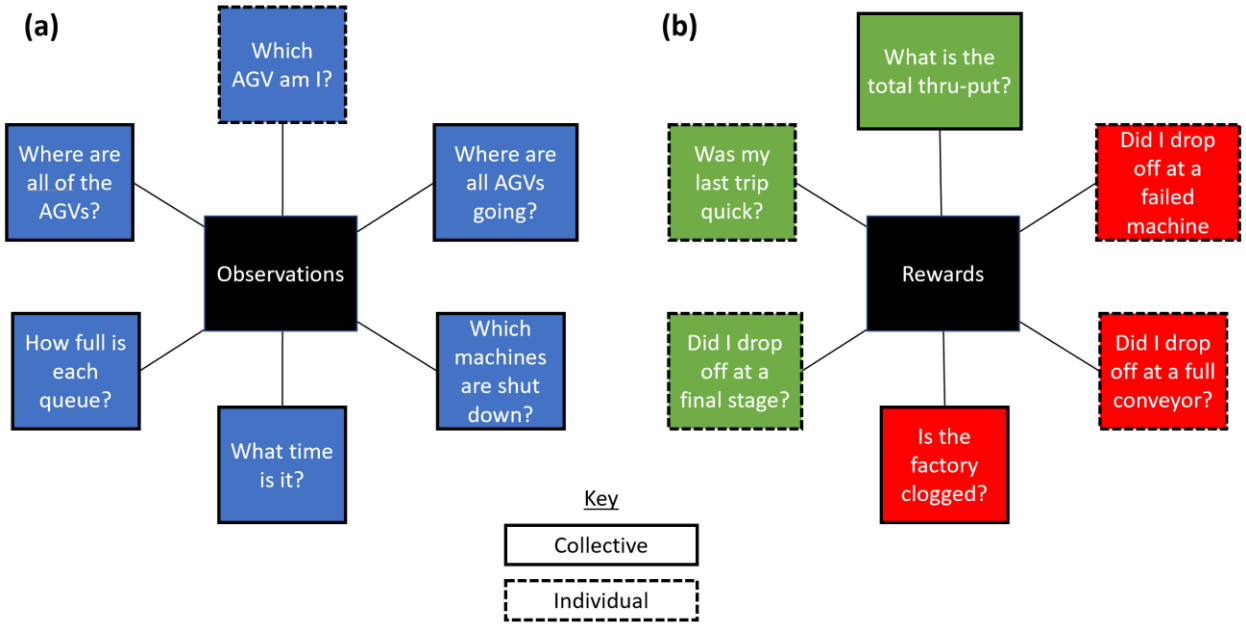


The AnyLogic simulation, shown above, consists of a fleet of AGVs (indicated by a red arrow) that pick up and drop off objects at different processing machines throughout the warehouse. Each AGV can carry at most one product (indicated by blue, green and pink squares) or one raw material (indicated by yellow circles). Precursor products and raw materials arrive on the left of the figure above, then are moved by the AGVs in stages to the right along the "process flow" arrow. At each stage, a given product must be combined with one raw material to pass through a processing machine. The raw material is consumed in the process, and the product is then ready to be moved to the next stage. Products must be processed in a specific sequence (stage 1-3) before exiting the system. Each processing stage has three machines with variable processing times and unexpected maintenance shutdowns.

Introduction to the Multi-Controller, Shared Policy Method

Reinforcement learning offers a simple, scalable solution to this fleet management problem. The included policy is trained using Pathmind's new Multi-Controller, Shared Policy functionality. The logic behind Multi-Controller Shared Policy, which is schematized in the figure below, is straightforward and intuitive. Each AGV has its own "controller," which can be individually triggered, gathering observations specific to its own state, and then carrying out an action based on the policy it shares with all other AGV's controllers. That controller's individual action changes the environment in either a beneficial or detrimental way, which is reflected in its reward variables.

Notably, the reward variables and observations in Multi-Controller, Shared Policy can include individual rewards and observations as well as shared collective reward variables and observations. For example, if one AGV drops off a product at a machine that is shutdown, that *individual* AGV should be penalized. This individual reward variable is defined as "deliveryAtShutdown". On the other hand, if a product passes successfully and quickly through the full processing sequence, then *all* AGVs should be rewarded. This collective reward is "totalThroughput". Examples of observations and reward variables that mix individual and collective metrics are shown below. Blue, green and red boxes represent observations, rewards and penalties, respectively. Collective information is outlined with a solid black line, and individual information is outlined with a dashed black line.



Tutorial

Step 1 - Perform a run with random actions to check Pathmind Helper setup.

Refer to the Pathmind Helper Setup guide to verify that the Pathmind Helper is functioning properly in the model. Completing this step will also demonstrate how the model performs using random actions instead of following a trained Pathmind Policy.

Step 2 - Examine the reinforcement learning elements.

Observations – Before choosing an action, each AGV observes its environment. The observations can be grouped into different categories: (1) the position of the AGV that is currently observing; (2) the status of each line in the manufacturing center, including where there are products available for pickup, what the queue sizes are, how much time remains for each product being processed, and the failure status of each machine. The functions that query the simulation model for this information are lines in the Observations field shown below.

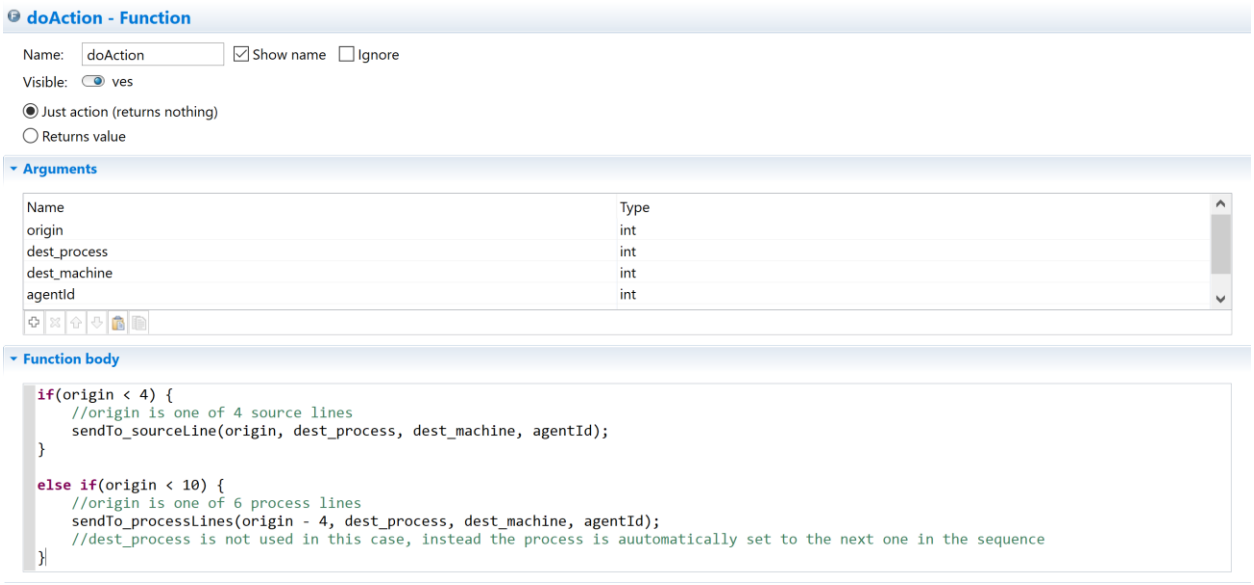
```
class Observations {
    //AGV observations
    double[] AGVs_self = getObs_AGV_selfOnly(agentId);
    //Manufacturing center observations
    double[] sourceLines = getObs_sourceLine();
    double[] pl_inventory = getObs_processLine_inventory();
    double[] pl_process = getObs_processLine_processProgress();
    double[] pl_failureStatus = getObs_processLine_failureStatus();
    double time = time() / getEngine().getStopTime();
}
```

Actions – Each AGV independently chooses the origin and destination of its next trip. The AGV picks up a product or raw material at the chosen origin and drops it at the chosen destination. There are 10 possible origins (9 lines and one raw material source line) and there are 9 possible destinations. The Actions, shown below, are defined by 3 different elements: (1) the origin, an integer with 10 possible values; (2) the destination stage, an integer with three possible values; and (3) the destination machine, an integer with three possible values. Note that the action space can be crafted in a number of ways for a single problem. This formulation parses destinations into two values in order to facilitate AGVs learning to avoid dropping products out of sequence.

```
class Actions {
    //4 source pickup points + 3 machine choices * 3 stages
   @Discrete(n = 10, size = 1) int origin;
   //3 machine dropoff points in...
   @Discrete(n = 3, size = 1) int destination_process;
   //...each of 3 stages
   @Discrete(n = 3, size = 1) int destination_machine;

   void doIt() { doAction(origin, destination_process, destination_machine, agentId); }
}
```

Note that the Actions class has a function doIt(), which passes the origin and destination values to a doAction() function that is defined in Main. Below are the details on how doAction() instructs each individual AGV to visit the appropriate origin and destination.



Because the raw materials are consumed at process lines, they can only be picked up from the raw material source line. The above logic allows for raw materials to be taken from the source line to any process. Products, however, are automatically brought to their next processing stage (stage 1 -> stage 2 -> stage 3), with the specific machine chosen by the output of Actions.

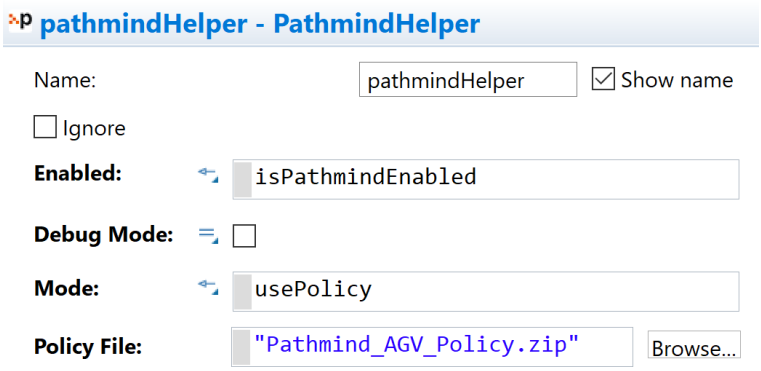
For more information on the mapping between Actions and origin-destination pairs, please examine the bodies of the sendTo_sourceLine and sendTo_processLine functions.

Reward Variables – As discussed in the “Introduction to Multi-Controller, Shared Policy” section, each AGV is rewarded individually for the competence of its previous action, and the population of AGVs is rewarded collectively for the overall throughput. The field below shows the specific reward variables that define both individual and collective rewards.

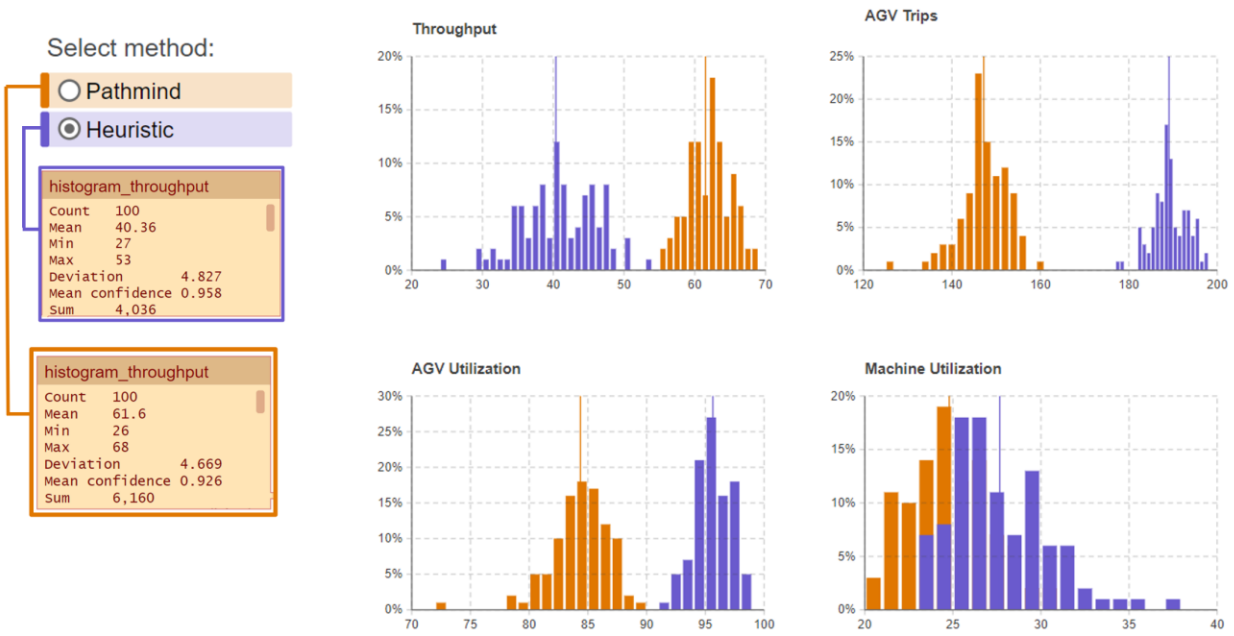
```
class Reward {
    //collective rewards
    double totalThroughput = throughput_product;
    double fullConveyor = fullFinishConveyor;
    double machineUtil = processLines.machineUtilization();
    //individual rewards
    double deliveryStage = AGVs.get(agentId).deliveryStage;
    double trips = AGVs.get(agentId).trips;
    double tripDuration = time() - AGVs.get(agentId).timeStamp_startTrip;
    double aveTripDuration = AGVs.get(agentId).getAveTripDuration();
    double emptyOrigins = AGVs.get(agentId).emptyOrigins;
    double fullQueue = AGVs.get(agentId).fullQueue;
    double deliveryToFailedMachine = AGVs.get(agentId).deliveryToFailedMachine;
    double essentialDelivery = AGVs.get(agentId).essentialDelivery;
    double invalid = AGVs.get(agentId).invalidAction;
    double agvUtilization = AGVs.get(agentId).resourcePool.utilization();
}
```

Step 3 – Evaluate the Pathmind policy

Included with the model is a pre-trained Pathmind Policy that was trained in the 3 AGV scenario. First, reference the trained policy in the PathmindHelper "Policy File" field:



Next, run the Monte Carlo experiment for 100 episodes and take note of the performance. Repeat this Monte Carlo experiment with the parameter isPathmindEnabled = false to default to the heuristic. Compare the totalThroughput of the policy to that of the heuristic. A representative Monte Carlo comparison of the policy to the heuristic is shown below.



The mean throughput is ~40 products when AGVs follow the heuristic and ~60 products when they are controlled by the Pathmind policy. **The Pathmind policy outperforms the heuristic by roughly 50%.**

Step 4 – Explore how Pathmind outperforms the heuristic

Showing that the intelligent policy can substantially outperform a simple routing heuristic is only half the battle. The second half is understanding, interpreting and harnessing the results. In this section, you will interpret the Monte Carlo results shown above. Here are several pieces of evidence visible in the Monte Carlo histograms:

- Pathmind reduces the number of AGV trips per episode by ~20%
- Pathmind reduces the AGV utilization by ~10%
- Machine utilization is roughly the same for both Pathmind and the heuristic

The reduced AGV utilization and reduced trips per episode indicate that the trained policy directs AGVs to **strategically refrain from picking up products**. Interestingly, even with lower use of the AGVs, the policy moves products through the manufacturing center at a higher rate. Evidently, the Pathmind-controlled AGVs are showing emergent behavior—they individually choose to reduce their workload, to avoid overloaded machines and collisions with other AGVs.

Step 5 – Train an intelligent Pathmind policy

To train your own version of the intelligent policy, reference the Exporting Models and Training guide. This guide will step you through to exporting your model, completing training, and downloading the Pathmind policy.

Reward function
The reward function used to train the policy included in this tutorial is

```
reward += after.totalThroughput * 0.01;
reward += ( after.machineUtil – before.machineUtil ) * 100;
reward += after.essentialDelivery – before.essentialDelivery;
reward -= after.fullConveyor – before.fullConveyor;
reward += ( after.trips – before.trips ) * 0.01;
```

along with the simulation metrics and overall learning progress. Notice that the reward function combines collective rewards such as totalThroughput and machineUtil with individual rewards such as essentialDelivery (wherein an AGV delivers a raw material to a product line where it is needed).

Try influencing AGVs to adopt different behavior by adding, deleting or modifying the reward terms in the reward function and running new experiments. For example, to influence AGVs to maximize their trip frequency, reduce the reward function to “reward += after.trips – before.trips;”

Export the Policy file. Repeat Step 3 in the abovementioned.

Conclusions

Pathmind was used to adapt an AGV fleet management problem for reinforcement learning. When trained on a 3 AGV scenario, the resulting policy is able to outperform a simple heuristic by 50%. That performance is a result of the Pathmind policy avoiding over-congestion of the system by choosing to regularly stall AGVs. This behavior showcases one of the strengths of reinforcement learning over traditional, static methods: complex, emergent behavior.

Updated October 29, 2020
Johnny Davenport, Ph. D.