Behavior Trees Tutorial

Jadrina De Andrade E Silva Malchik
École des Mines de Saint-Étienne

Donal O'Donovan
Munster Technological University

June 2023

Abstract

This paper intends to explain how behavior trees work with the use of two small examples. This is done by detailing the construction and execution of two different behavior trees. One is used to control a robot and the other controls an agent in the Pac-man game. The analysis also includes pieces of code. The unique contribution this paper makes is that it explores the execution of a behavior tree from a programmer's perspective. This exercise reveals that behavior tree execution is not as straightforward as it first appears and that there are multiple subtleties that result from the broadness of the concepts used and the different implementations in libraries. There is also the issue of parallelism and concurrency brought about by the use of the Parallel node that had to be explored separately in order to fully understood. In the end there is an unexpected complexity that comes from trying to implement these behavior trees that is only discovered when programming.

1 Introduction

This is a small document intended to explain how behavior trees work with the use of two small examples. Specifically we intend to explore the execution of a behavior tree from a programmer's perspective. We will begin with one example of a robot that checks its battery and then we shall tackle the more classical, but more complex, example of the Pac-man game. The goal is to show, in detail, how the behavior tree is constructed and then go step-by-step through its execution. There will be links provided to the code used and credit for the code goes to their respective owners. It will be assumed that the reader has already read at least the first chapter of [4] so we can gloss over the basics.

It is important to note that two different visualizers are used in this tutorial. This is done to show the different forms that behavior trees can take. The types of nodes are always the same but the shapes and colors may sometimes change. It is important to be able to understand how a tree works regardless of its format style.

2 Battery Bot

For this first example we shall examine the behavior tree of a data-gathering robot. The robot is constantly reading from its sensors. In this tutorial we only need to manage the data that it receives concerning its battery level. If the battery level is sufficiently high, the robot is to remain idle. If the battery level is low the robot must flash its LED strip. This is illustrated in the following behavior tree:

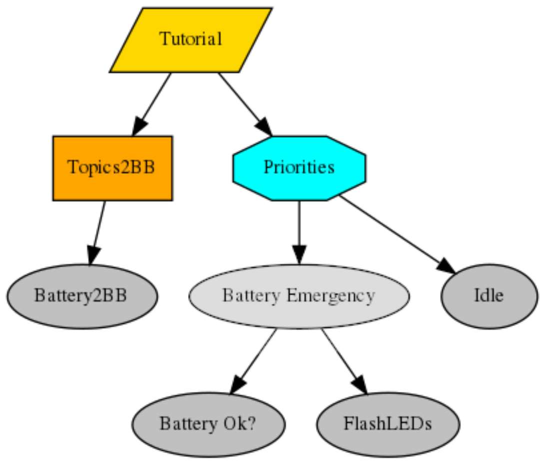


Figure 1: Final BT for Battery Bot

2.1 Construction of the behavior tree

Before we can go through the construction of this tree and explain the choice of nodes we have to establish the basics of behavior trees.

- Internal nodes and leaf nodes are referred to as control flow nodes and execution nodes respectively.
- Control flow nodes are divided into four separate types:
  - A Sequence node will run one child node after another, in a sequential manner from left to right, and will only succeed if all its children succeed.
  - A Fallback node is also sometimes called a Selector node because it selects the first child to succeed. It will execute one child after the other until one succeeds or all fail.
  - A Parallel node allows multiple children to be executed and will succeed if M of its N children succeed. Upon success all children that are running are stopped. Please see section 5 for clarification.
  - A Decorator node is used to 'fine tune' the either the status return by a child node or its execution; for example invert a *Success* status to *Failure*.
- Execution nodes can be of type Action or Condition.

- An Action node executes an operation or task.
- A Condition node evaluates a Boolean expression.

With this brief overview we can start constructing our tree.

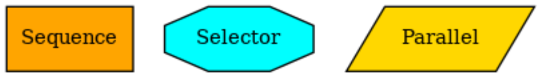


Figure 2: Control flow node types

When describing the behavior of our robot we described two separate behaviors, data gathering and checking the battery. To support the execution of these two at the same time we shall use a Parallel node as our root with an M value of 2. This means that this node will return *Success* if two of its children do. If one of its children returns *Failure* it will do so as well. At every tick it will then tick all its children.

Figure 3: Root

The code for the Parallel node is the following:

```
class Parallel(Composite):

def __init__(self, name="Parallel", policy=common.ParallelPolicy.SUCCESS_ON_ALL,
 children=None, *args, **kwargs):
    super(Parallel, self).__init__(name, children, *args, **kwargs)
    self.policy = policy

def tick(self):
    """
    Tick over the children.

    Yields:
        :class:`~py_trees.behaviour.Behaviour`: a reference to itself or one of
its children
    """
    if self.status != Status.RUNNING:
        # subclass (user) handling
        self.initialise()
    self.logger.debug("%s.tick()" % self.__class__.__name__)
    # process them all first
    for child in self.children:
        for node in child.tick():
            yield node
    # new_status = Status.SUCCESS if self.policy == common.ParallelPolicy.
SUCCESS_ON_ALL else Status.RUNNING
    new_status = Status.RUNNING
    if any([c.status == Status.FAILURE for c in self.children]):
        new_status = Status.FAILURE
    else:
        if self.policy == common.ParallelPolicy.SUCCESS_ON_ALL:
            if all([c.status == Status.SUCCESS for c in self.children]):
                new_status = Status.SUCCESS
        elif self.policy == common.ParallelPolicy.SUCCESS_ON_ONE:
            if any([c.status == Status.SUCCESS for c in self.children]):
                new_status = Status.SUCCESS
    # special case composite - this parallel may have children that are still
running
    # so if the parallel itself has reached a final status, then these running
children
    # need to be made aware of it too
    if new_status != Status.RUNNING:
        for child in self.children:
            if child.status == Status.RUNNING:
                # interrupt it (exactly as if it was interrupted by a higher
priority)
                child.stop(Status.INVALID)
        self.stop(new_status)
    self.status = new_status
    yield self
```

From this code we can see that whenever there is a tick the Parallel node will run the "tick" function of each of its children and only after it has done this it will set its status based on the results.

Now we must add its children. The first behavior is data gathering, so we want to publish our batteries' data to our blackboard. This will be our action node which we will call *Battery2BB*. We shall not put the node in directly, instead it shall be the child of a Sequence node whose children will all be in charge of putting data to the blackboard. This node will be called *Topics2BB*. By including this node, the tree is made easily extensible for future modifications. And that leaves us with our first branch of the tree.

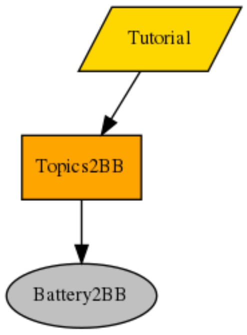


Figure 4: Tree with data-gathering branch

The code for the Sequence node is:

```
    class Sequence(Composite):

    def __init__(self, name="Sequence", children=None, *args, **kwargs):
        super(Sequence, self).__init__(name, children, *args, **kwargs)
        self.current_index = -1  # -1 indicates uninitialised
    def tick(self):

        self.logger.debug("%s.tick()" % self.__class__.__name__)
        if self.status != Status.RUNNING:
            self.logger.debug("%s.tick() [!RUNNING->resetting child index]" % self.
    __class__.__name__)
            # sequence specific handling
            self.current_index = 0
            for child in self.children:
                # reset the children, this helps when introspecting the tree
                if child.status != Status.INVALID:
                    child.stop(Status.INVALID)
            # subclass (user) handling
            self.initialise()
        # run any work designated by a customised instance of this class
        self.update()
        for child in itertools.islice(self.children, self.current_index, None):
            for node in child.tick():
                yield node
                if node is child and node.status != Status.SUCCESS:
                    self.status = node.status
                    yield self
                    return
            self.current_index += 1
        # At this point, all children are happy with their SUCCESS, so we should be
    happy too
        self.current_index -= 1  # went off the end of the list if we got to here
        self.stop(Status.SUCCESS)
        yield self
```

Comparing to the code of the Parallel node, we can see that in this case the process is different. At each tick, the node will call the “tick” function of each child and check their status immediately after. If they all return *Success* then the Sequence node itself will stop execution and return *Success*. If one child returns *Running* or *Failure* the node then takes that same status and stops iterating through their children. We can see that the index of the current child is saved inside a variable.

After this we must focus on the remaining behaviors: checking the battery and flashing the light strip. At each tick we shall check the battery level. If it is acceptable, then the robot is simply idle. If it is not acceptable, then the robot must flash its lights. We are selecting between two different actions so it makes sense to use a Fallback or Selector node. This node shall be called *Priorities*.

```
    class Selector(Composite):


    def __init__(self, name="Selector", children=None, *args, **kwargs):
        super(Selector, self).__init__(name, children, *args, **kwargs)
        self.current_child = None

    def tick(self):

        self.logger.debug("%s.tick()" % self.__class__.__name__)
        # Required behaviour for *all* behaviours and composites is
        # for tick() to check if it isn't running and initialise
        if self.status != Status.RUNNING:
            # selectors dont do anything specific on initialisation
            #   - the current child is managed by the update, never needs to be '
    initialised'
            # run subclass (user) handles
            self.initialise()
        # run any work designated by a customised instance of this class
        self.update()
        previous = self.current_child
        for child in self.children:
            for node in child.tick():
                yield node
                if node is child:
                    if node.status==Status.RUNNING or node.status==Status.SUCCESS:
                        self.current_child = child
                        self.status = node.status
                        if previous is None or previous != self.current_child:
                            # we interrupted, invalidate everything after
                            passed = False
                            for child in self.children:
                                if passed:
                                     if child.status != Status.INVALID:
                                         child.stop(Status.INVALID)
                                if child == self.current_child:
                                     passed = True
                        yield self
                        return
        # all children failed, set failure ourselves and current child to the last
    bugger who failed us
        self.status = Status.FAILURE
        try:
            self.current_child = self.children[-1]
        except IndexError:
            self.current_child = None
        yield self
```

In this code we see that the node calls the “tick” function of each of its children until one returns *Running* or *Success*. Then all the children that are after the child that gave that status are stopped or set to *Invalid*. This is done using a boolean that is turned into True once the current child is observed and remains the same in other cases. This shows that the execution of the children is sequential and only stops when there is a successful node.

One thing to note about these types of nodes is that they always execute left to right so we want our highest priority behaviors to be on the left. In this case we only want to be idle if our battery level is high enough so we'll put the checking for battery on the left and the idle behavior on the right. To achieve the idle behavior we shall insert an action node called *Idle*. That results in the following tree where we have yet to define the battery checking behavior (circle with question mark).

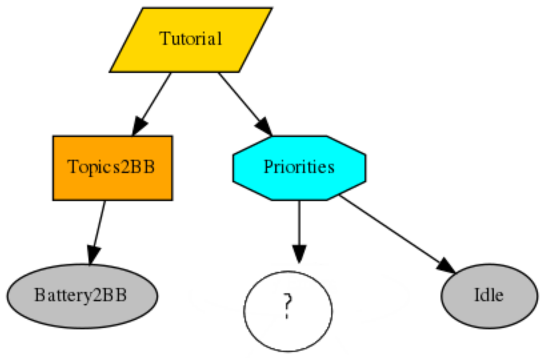


Figure 5: Tree with decision making

When the robot is not idle we want it to check the battery and, based on the level detected, decide whether the LEDs should be flashed or not. To achieve this we add a Condition node called *Battery Ok?* that returns *Success* if the battery level is acceptable and *Failure* if it is not.

Then we have *FlashLEDS* as an Action node. Once again we need to select between two options so it would make sense to use a Fallback node. This time however we have to use a Decorator because we are choosing between two opposing options. Therefore, one will always return a *Success* status. A normal Fallback node would always return *Success* and then the ticks would never arrive to the Action node *Idle*. So we will use an Inverter Decorator Node, *Battery Emergency*, so that *Success* is *Failure*. This Decorator changes the "update" function of the selector to look like this:

```
def _update(func):
    @functools.wraps(func)
    def wrapped(self):
        if self.original.status == common.Status.SUCCESS:
            self.feedback_message = "success is failure" + (" [%s]" % self.
original.feedback_message if self.original.feedback_message else "")
            return common.Status.FAILURE
        else:
            self.feedback_message = self.original.feedback_message
            return self.original.status
    return wrapped
```

Basically it takes the status of the node and if it is *Success* then it will return *Failure*, setting the feedback message to reflect this. In any other case it will return the original status.

We attach to *Battery Emergency*, the condition *Battery Ok?* and the action node *Idle*. Therefore when the *Battery Ok?* node returns *Success* because the battery is sufficiently charged, the decorator will transform the result, such that it is interpreted as *Failure* causing the parent node *Priorities* to pass the ticks to the Action node *Idle*. This results in the tree we showed earlier.

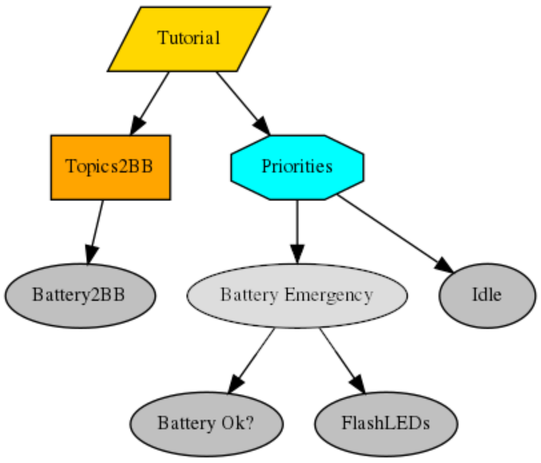


Figure 6: Final BT for Battery Bot

The code for constructing the code is this:

```
def create_root():
# behaviours
root = py_trees.composites.Parallel("Tutorial")
topics2bb = py_trees.composites.Sequence("Topics2BB")
battery2bb = py_trees_ros.battery.ToBlackboard(name="Battery2BB",
                                               topic_name="/battery/state",
                                               threshold=30.0
                                               )
priorities = py_trees.composites.Selector("Priorities")
battery_check = py_trees.meta.success_is_failure(py_trees.composites.Selector)(
name="Battery Emergency")
is_battery_ok = py_trees.blackboard.CheckBlackboardVariable(
    name="Battery Ok?",
    variable_name='battery_low_warning',
    expected_value=False
)
flash_led_strip = py_trees_ros.tutorials.behaviours.FlashLedStrip(
    name="FlashLEDs",
    colour="red")
idle = py_trees.behaviours.Running(name="Idle")

# tree
root.add_children([topics2bb, priorities])
topics2bb.add_child(battery2bb)
priorities.add_children([battery_check, idle])
battery_check.add_children([is_battery_ok, flash_led_strip])
return root


def shutdown(behaviour_tree):
    behaviour_tree.interrupt()


def main():
    """
    Entry point for the demo script.
    """
    rospy.init_node("tree")
    root = create_root()
    behaviour_tree = py_trees_ros.trees.BehaviourTree(root)
    rospy.on_shutdown(functools.partial(shutdown, behaviour_tree))
    if not behaviour_tree.setup(timeout=15):
        console.logerror("failed to setup the tree, aborting.")
        sys.exit(1)
    behaviour_tree.tick_tock(500)
```

As you can see the structure and nodes included are the same at the ones in the Figure 6.

2.2 Execution

Now we can run through the execution of the tree. Note that before any ticking happens the nodes of the trees must be initialized and some of them require additional setup. This happens when the nodes are instantiated and the "__init__" function is called and when the root node's "setup" function is called. This call is then propagated down to the leaf nodes. In these functions we define the names of the nodes and create blackboard variables. A blackboard is a shared memory that all nodes in the tree can access. The "setup" function is also where we create topics, publishers and subscribers to communicate between ROS nodes. In addition to these functions there is also the "__initialise__" function which is called when a node is first ticked and whenever it receives a tick and its status is not *Running*. We do not use this function in this example but it is typically used to reset variables and timers.

When execution begins the root node generate ticks and it passes them to its children *Topics2BB* and *Priorities* in parallel, technically it passes it first to the left child and then to the right. A tick is really a call of the node's "tick" function. In control flow nodes like *Tutorial* this function shall iterate over the children and calls their own "tick" function. This is what we mean when we say a child is ticked or it receives a tick. When *Topics2BB* receives a tick it routes it down to its only child *Battery2BB*. This will execute the action of publishing the battery level to the blackboard. In Py Trees when a behavior is ticked its "update" function is called. *Battery2BB* is an instance of the *ToBlackBoard* behavior whose code we have placed below.

```
class ToBlackboard(subscribers.ToBlackboard):
    """
    Subscribes to the battery message and writes battery data to the blackboard.
    Also adds a warning flag to the blackboard if the battery
    is low - note that it does some buffering against ping-pong problems so the
    warning
    doesn't trigger on/off rapidly when close to the threshold.

    When ticking, updates with :attr:`~py_trees.common.Status.RUNNING` if it got no
    data,
    :attr:`~py_trees.common.Status.SUCCESS` otherwise.

    Blackboard Variables:
        * battery (:class:`sensor_msgs.msg.BatteryState`)[w]: the raw battery
    message
        * battery_low_warning (:obj:`bool`)[w]: False if battery is ok, True if
    critically low
    Args:
        name (:obj:`str`): name of the behaviour
        topic_name (:obj:`str`) : name of the battery state topic
        threshold (:obj:`float`) : percentage level threshold for flagging as low
    (0-100)
    """
    def __init__(self, name, topic_name="/battery/state", threshold=30.0):
        super(ToBlackboard, self).__init__(name=name,
                                           topic_name=topic_name,
                                           topic_type=sensor_msgs.BatteryState,
                                           blackboard_variables={"battery": None},
                                           clearing_policy=py_trees.common.
    ClearingPolicy.NEVER
                                           )
        self.blackboard = py_trees.blackboard.Blackboard()
        self.blackboard.battery = sensor_msgs.BatteryState()
        self.blackboard.battery.percentage = 0.0
        self.blackboard.battery.power_supply_status = sensor_msgs.BatteryState.
    POWER_SUPPLY_STATUS_UNKNOWN
        self.blackboard.battery_low_warning = False   # decision making
        self.threshold = threshold

    def update(self):
        """
        Call the parent to write the raw data to the blackboard and then check
    against the
        threshold to determine if the low warning flag should also be updated.
        """
        self.logger.debug("%s.update()" % self.__class__.__name__)
        status = super(ToBlackboard, self).update()
        if status != py_trees.common.Status.RUNNING:
            # we got something
            if self.blackboard.battery.percentage > self.threshold + 5.0:
                self.blackboard.battery_low_warning = False
            elif self.blackboard.battery.percentage < self.threshold:
                    self.blackboard.battery_low_warning = True
                    rospy.logwarn_throttle(60, "%s: battery level is low!" % self.
    name)
            # else don't do anything in between - i.e. avoid the ping pong problems

            self.feedback_message = "Battery level is low" if self.blackboard.
    battery_low_warning else "Battery level is ok"
        return status
```

When *Battery2BB* is ticked *ToBlackBoard*'s "update" function is called. This will first update the raw battery data on the blackboard. Then it checks the actual battery level against a hard-coded threshold and sets the blackboard variable '*battery_low_warning*' to *True* or *False* depending on whether the level is above or below the threshold. Once this action is completed, within the same tick, the node will return *Success*. The "tick" function in a behavior is also in charge of calling the "stop" function in order to stop the execution of a behavior. Because *Battery2BB* is the sole child of *Topics2BB*, the *Topics2BB* node will then return *Success*.

Even after success the nodes will be ticked again because the main program is constantly ticking the parallel node, repeating this cycle. If we cannot receive data then the *Battery2BB* node will instead return a *Running* status. This will cause *Topics2BB* to return *Running* to the root node. The other branch will be “blocked” in the sense that it will not be able to react to changes in the battery, instead it will continue its previous, or default, behavior. Once the data is published the branch will become reactive again. Note that, with the current setup, *Battery2BB* will always be receiving ticks due to the Parallel root.

Next, we shall focus on the execution of the other branch, assuming the data gathering is working as it should. The *Tutorial* node will pass the tick to its child *Priorities*. *Priorities* shall then tick *Battery Emergency*. This node will then tick its first child *Battery Ok?*. This will then call the “update” function of this node which will check the blackboard variable ‘*battery_low_warning*’.

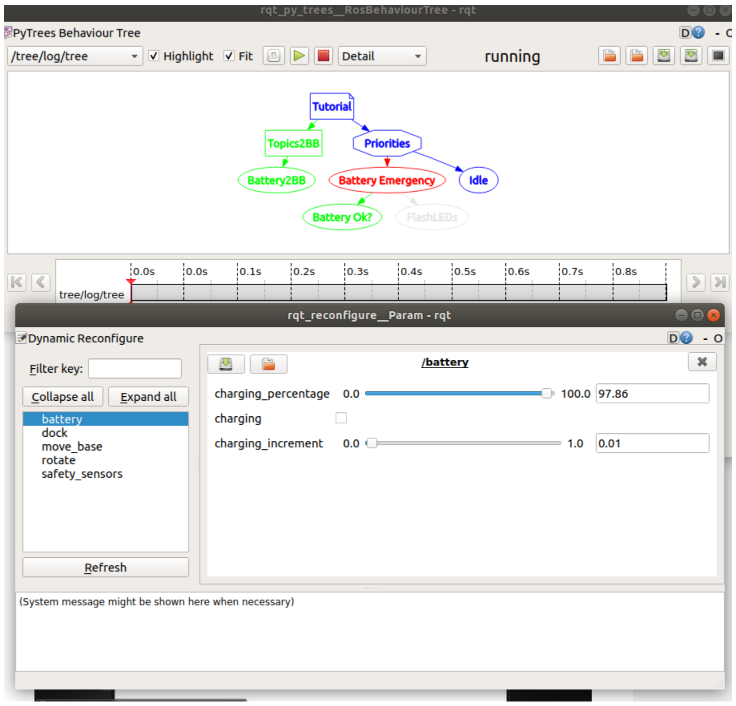
```
class CheckBlackboardVariable(behaviours.Behaviour):

def __init__(self,
             name,
             variable_name="dummy",
             expected_value=None,
             comparison_operator=operator.eq,
             clearing_policy=common.ClearingPolicy.ON_INITIALISE,
             debug_feedback_message=False
             ):
    super(CheckBlackboardVariable, self).__init__(name)
    self.blackboard = Blackboard()
    self.variable_name = variable_name
    self.expected_value = expected_value
    self.comparison_operator = comparison_operator
    self.matching_result = None
    self.clearing_policy = clearing_policy
    self.debug_feedback_message = debug_feedback_message

def initialise(self):
    """
    Clears the internally stored message ready for a new run
    if ``old_data_is_valid`` wasn't set.
    """
    self.logger.debug("%s.initialise()" % self.__class__.__name__)
    if self.clearing_policy == common.ClearingPolicy.ON_INITIALISE:
        self.matching_result = None

def update(self):
    self.logger.debug("%s.update()" % self.__class__.__name__)
    if self.matching_result is not None:
        return self.matching_result

    result = None
    check_attr = operator.attrgetter(self.variable_name)

    try:
        value = check_attr(self.blackboard)
        # if existence check required only
        if self.expected_value is None:
            self.feedback_message = "'%s' exists on the blackboard (as required)
" % self.variable_name
            result = common.Status.SUCCESS
    except AttributeError:
        self.feedback_message = 'blackboard variable {0} did not exist'.format(
self.variable_name)
        result = common.Status.FAILURE

    if result is None:
        # expected value matching
        # value = getattr(self.blackboard, self.variable_name)
        success = self.comparison_operator(value, self.expected_value)

        if success:
            if self.debug_feedback_message:  # costly
                self.feedback_message = "'%s' comparison succeeded [v: %s][e: %s
]" % (self.variable_name, value, self.expected_value)
            else:
                self.feedback_message = "'%s' comparison succeeded" % (self.
variable_name)
            result = common.Status.SUCCESS
        else:
            if self.debug_feedback_message:  # costly
                self.feedback_message = "'%s' comparison failed [v: %s][e: %s]"
% (self.variable_name, value, self.expected_value)
            else:
                self.feedback_message = "'%s' comparison failed" % (self.
variable_name)
            result = common.Status.FAILURE

    if result == common.Status.SUCCESS and self.clearing_policy == common.
ClearingPolicy.ON_SUCCESS:
        self.matching_result = None
    else:
        self.matching_result = result
    return result

def terminate(self, new_status):
    """
    Always discard the matching result if it was invalidated by a parent or
    higher priority interrupt.
    """
    self.logger.debug("%s.terminate(%s)" % (self.__class__.__name__, "%s->%s" %
(self.status, new_status) if self.status != new_status else "%s" % new_status))
    if new_status == common.Status.INVALID:
        self.matching_result = None
```

When it is initialised it prepares the blackboard by setting the variable, the expected value and the comparison operator. Then on every tick it compares the value that is stores in the blackboard variable with the expected value using the set operator. If the comparison is successful then it returns *Success*. If it is not then it returns *Failure*. After that the result can be saved or not depending on the policy.

If '*battery_low_warning*' is set to *True* the node will return *Success*. Upon receiving this status update, its parent node *Battery Emergency* will not send a tick to *FlashLEDS*. *Battery Emergency*, having one child succeed will change to *Success* status but because of its decorator success is failure it will switch to *Failure* and return this to *Priorities*.

Priorities is a selector. So, receiving that its first child has failed, it will then tick *Idle*. *Idle* only returns a *Running* status when ticked causing the robot to not do anything. *Idle* will return

Running to *Priorities* who will return *Running* to *Tutorial*. This node will return *Running* and then receive the next tick from the main program. This execution can be seen in the following image:



charging_percentage 0.0 100.0 97.86
charging ☐
charging_increment 0.0 1.0 0.01

(System message might be shown here when necessary)

Figure 7: Execution when battery is full

Note that *Success* is green, *Running* is blue and *Failure* is red. Gray means that the node is not being ticked meaning its status is *Invalid*.

If the the blackboard variable '*battery_low_warning*' is set to *False* then *Battery Ok?* will return *Failure*. Then the *Battery Emergency* node will pass the tick to its next child, *FlashLEDs*. *FlashLEDs*' sole purpose is to keep the LEDs flashing. The only status *FlashLEDs* returns is *Running* *i.e* it never returns *Success*.

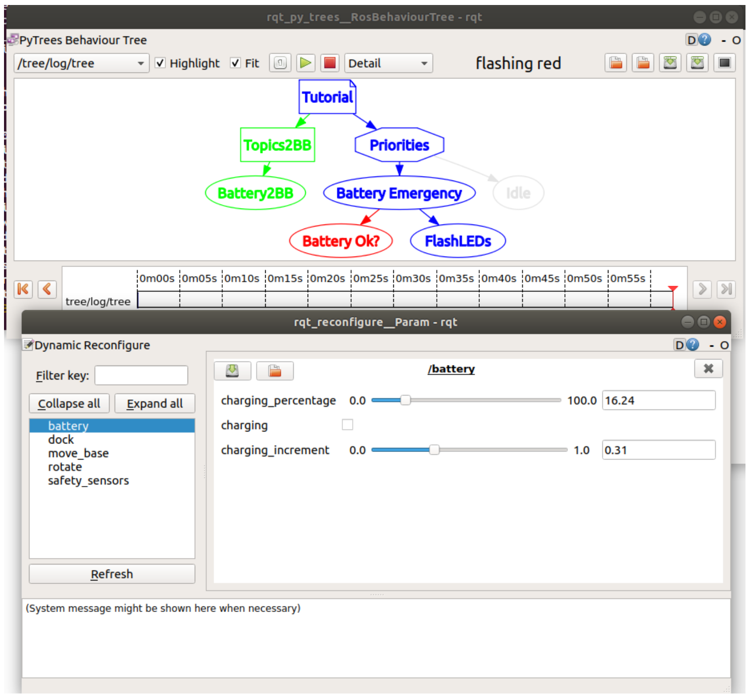
```
class FlashLedStrip(py_trees.behaviour.Behaviour):
"""
This behaviour simply shoots a command off to the LEDStrip to flash
a certain colour and returns :attr:`~py_trees.common.Status.RUNNING`.
Note that this behaviour will never return with
:attr:`~py_trees.common.Status.SUCCESS` but will send a clearing
command to the LEDStrip if it is cancelled or interrupted by a higher
priority behaviour.

Args:
    name (:obj:`str`): name of the behaviour
    topic_name (:obj:`str`) : name of the battery state topic
    colour (:obj:`str`) : colour to flash ['red', 'green', blue']
"""
def __init__(self, name, topic_name="/led_strip/command", colour="red"):
    super(FlashLedStrip, self).__init__(name=name)
    self.topic_name = topic_name
    self.colour = colour

def setup(self, timeout):
    """
    Args:
        timeout (:obj:`float`): time to wait (0.0 is blocking forever)

    Returns:
        :obj:`bool`: whether it timed out trying to setup
    """
    self.publisher = rospy.Publisher(self.topic_name, std_msgs.String,
queue_size=10, latch=True)
    self.feedback_message = "setup"
    return True

def update(self):
    """
    Annoy the led strip to keep firing every time it ticks over (the led strip
will clear itself
    if no command is forthcoming within a certain period of time). This
behaviour will only finish if it
    is terminated or interrupted from above.
    """
    self.logger.debug("%s.update()" % self.__class__.__name__)
    self.publisher.publish(std_msgs.String(self.colour))
    self.feedback_message = "flashing {0}".format(self.colour)
    return py_trees.common.Status.RUNNING

def terminate(self, new_status):
    """
    Shoot off a clearing command to the led strip.

    Args:
        new_status (:class:`~py_trees.common.Status`): the behaviour is
transitioning to this new status
    """
    self.publisher.publish(std_msgs.String(""))
```

A valid question at this point is ‘does *FlashLEDs* execute indefinitely?’. This is not the case. When the battery voltage reaches an acceptable level, *Battery Emergency* stops ticking *FlashLEDs*.

Recall that a non-ticked node is effectively 'asleep', and does not return a status value. The net effect of this is that *FlashLEDs* halts execution i.e. no flashing LEDs.

FlashLEDs only returns a status value, when it receives a tick and the "update" Python function is called. In this case, the only status value returned by *FlashLEDs* is *Running*. A "terminate' message from the *Battery Emergency* node results in the LEDs strip being cleared and the *FlashLEDs* node switching to *Invalid* status. It will not return this status.

While *FlashLEDs* is being ticked and returning *Running*, *Battery Emergency* will also return *Running*. This blocks the *Idle* node from receiving ticks and if its was previously *Running*, then it sets its status to *Invalid* without returning it. *Priorities* waits to see the result of *Battery Emergency*. This causes *Priorities* to also return *Running*. Which will finally cause *Tutorial* to return *Running*. Then the main program will send another tick to *Tutorial* and the check will happen all over again. This execution can be seen in the next image.



charging_percentage 0.0 100.0 16.24
charging
charging_increment 0.0 1.0 0.31

(System message might be shown here when necessary)

Figure 8: Execution when battery is empty

As soon as the battery level rises above the threshold the program will return to the previous execution as the condition has higher priority than the *FlashLEDS* action.

3 Pac-man

We shall now continue to explain another slightly more complicated example which is the Pac-man game. Behavior trees were originally created to make complex AI for NPC characters so this is a classical example that is actually featured in the official book. We shall attempt to break it down in a little more detail so that everything is left crystal clear.

3.1 Construction of the behavior trees

In Pac-man we have two objectives: eat as many pills as possible and avoid getting eaten by the ghosts. So our behaviors shall be to eat and to avoid or escape from our enemies. These shall be two action nodes, *Greedy* and *Escape*.

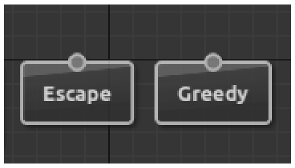


Figure 9: Two action nodes

In order to control which of these we will be using, we need a Fallback node. We want our Pac-man to prioritize survival over eating so we shall put *Escape* to the left side, as the order of execution provides an implicit priority. And we always need a root so we should add that.

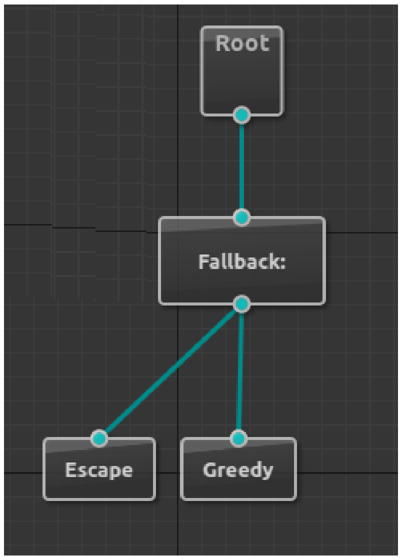


Figure 10: Tree with Fallback node

Pac-man only really needs to avoid ghosts if they are nearby. So we shall add a Condition node, *IsGhostClose*, that will only return *Success* if there is a ghost close by and *Failure*, otherwise. Then in order to make sure that we will only escape if this is true, we attach *IsGhostClose* and *Escape* to a Sequence node so that only if the first is true we move on to escape. We then attach this node where *Escape* used to be. Finally we have our complete tree.

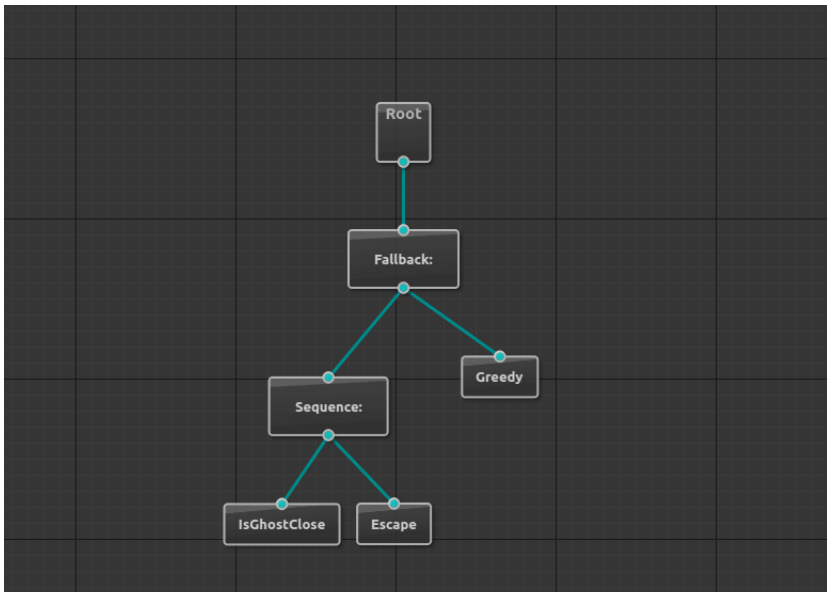


Figure 11: Final Pac-man tree

3.2 Execution

Now we can run through the execution of the tree. Just like in the last example, our main program will tick the root node. Then the root node will pass on this tick to its only child *Fallback*. *Fallback* will then first tick its leftmost child *Sequence*, which in turn ticks its first child *IsGhostClose*.

If there is a ghost close then *IsGhostClose* returns *Success*. *Sequence* receives this status and continues on to tick the next node *Escape* so our Pac-man will try to move away from the ghost. Then after that is done *Escape* shall return *Success* and because all its children have been successful *Sequence* returns *Success* to *Fallback*. *Fallback* then returns *Success* without ticking the action node *Greedy* so while escaping Pac-man does not focus on eating. Finally *Root* will return *Success* and wait for the next tick.

If there are no ghosts nearby *IsGhostClose* returns *Failure*. Because a Sequence node fails with only one child failing, *Sequence* returns *Failure* without ticking *Escape*, therefore our Pac-man will not try to escape. *Fallback* receives the failure of its first child so it moves on to tick its second child *Greedy*. *Greedy* then returns *Success* as Pac-man runs around gobbling up all the pills in sight. *Fallback* now returns *Success* to the root which then returns *Success* and waits for the next tick.

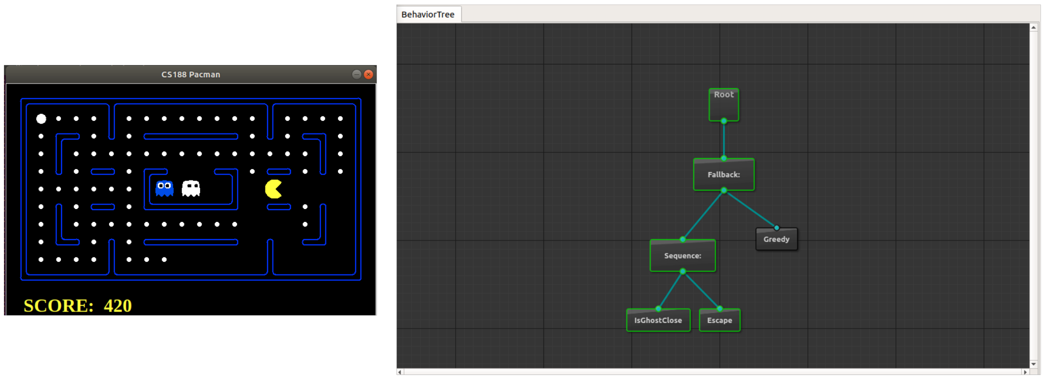


Figure 12: Execution when ghost is close

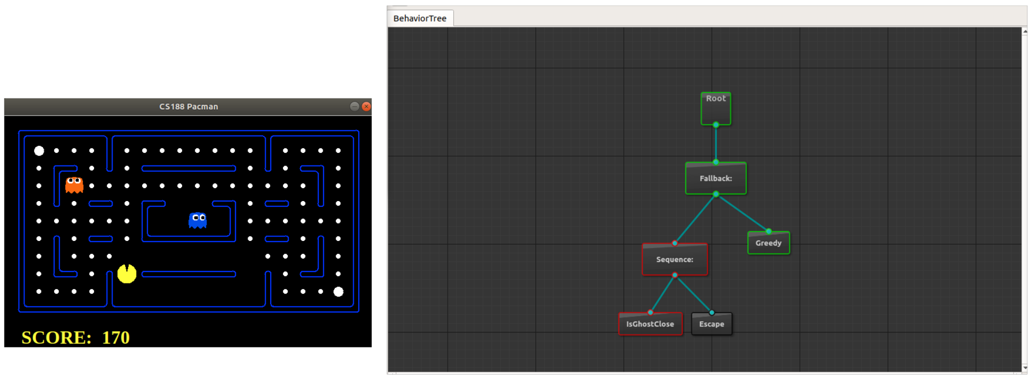


Figure 13: Execution when ghost is not close

3.3 Extension of the behavior tree

Next, in order to test our understanding of behavior trees we shall attempt to extend the Pac-man example. One node type that we were unable to test in the previous example was the Decorator, let's include it now. At this stage our Pac-man lacks real intelligence. It only eats pills and avoids ghosts. We could extend our example by checking whether the closest ghost is scared or not and if it is scared we could choose to chase it instead of escaping to gain extra points. In order to incorporate the Decorator node we will choose a configuration that isn't the most intuitive to achieve this, but should achieve the same results. First we switch the position of *Greedy* and *Escape* as now we will prioritize eating. Then we place the new node *Decorator* between *Sequence* and *IsGhostClose*. We have programmed this node to return *Success* when its child returns *Failure* and vice-versa. Then between the *Sequence* node and *Escape* we shall add another child to *Fallback*. This child will be another Sequence node with two children. The left child will be the Condition node *IsClosestGhostScared* and then its right child will be *Chase*.

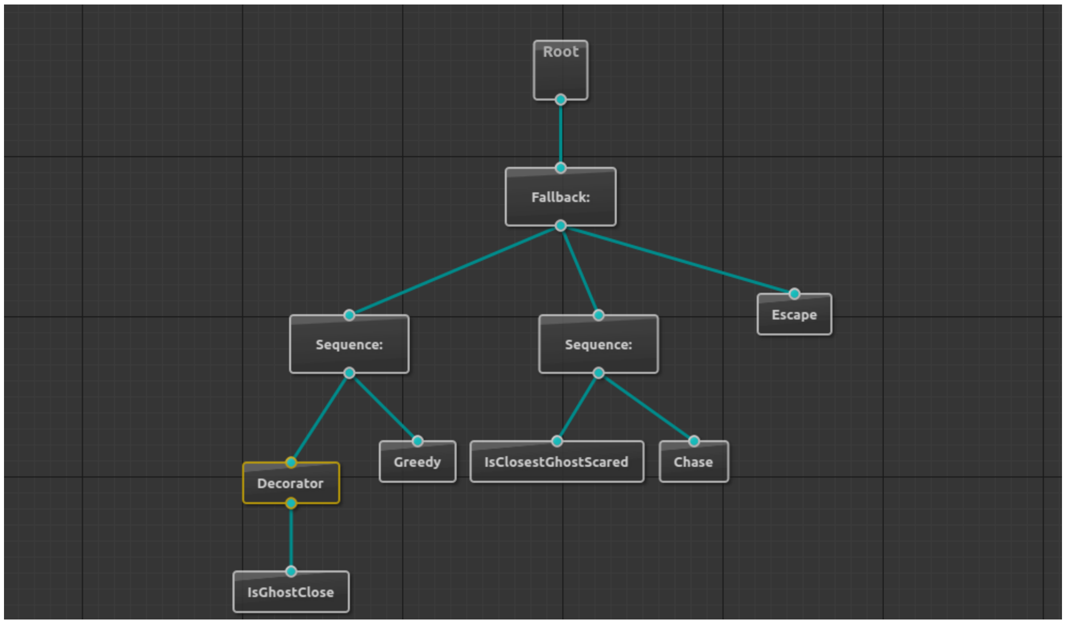


Figure 14: Final Pac-man tree with chase and Decorator node

The expected behavior would be the following. Just like before, our main program will tick the root node. Then the root node will pass on this tick to its only child *Fallback*. *Fallback* will then first tick its leftmost child *Sequence*, which in turn ticks its first child *Decorator*. Then this node ticks its child node *IsGhostClose*. This node will then tick its child which checks if there is a ghost close by.

It there are no ghosts around *IsGhostClose* returns *Failure* which the Decorator node will turn into *Success*, this causes the Sequence node to then tick its next child *Greedy*, causing Pac-man to start or continue eating. If there is a ghost close by then the *IsGhostClose* node will return *Success* which then will be received by *Sequence* as *Failure* so it will return *Failure* to its parent *Fallback*. This node will then tick its next child, *Sequence* which will tick its child to check *IsClosestGhostScared*. If the ghost is scared then *IsClosestGhostScared* will return *Success* and its parent node *Sequence* will tick *Chase* causing Pac-man to go after the ghost. Then this node will return *Success* which will cause its parent *Sequence* to return *Success*. Then *Fallback* will receive this status and return *Success*. This continues all the way up to the root node.

If the closest ghost is not scared *IsClosestGhostScared* returns *Failure* which will cause the node *Sequence* to return *Failure*. This means *Fallback* will execute its other child *Escape*. Therefore in this case Pac-man will escape from the ghosts. The rest of the execution should be the same as above.

3.3.1 Code

If you would like to try this extension you can use this code to define the Decorator node and use it. In the build/bt folder create a file called DecoratorNode.py and copy the following code.

```python
from ControlNode import ControlNode
from NodeStatus import *
import time
import socket
import sys
BUFSIZE = 4096

class DecoratorNode(ControlNode):

    def __init__(self,name):
        ControlNode.__init__(self,name)
        self.nodeType = 'Decorator'
        self.accepted = False

    def Execute(self,args):
        if (self.isRoot):
            try:
                message = self.GetString("")

                if(not self.accepted):

                    # Start listening
                    self.s.listen(10)
                    print("Socket Listening")

                    # Accept connection
                    print("Accepting Connection")

                    self.conn, self.addr = self.s.accept()
                    print("Connected to %s:%s" % (self.addr[0], self.addr[1]))
                    self.accepted = True

                BUFFER = bytes()

                while (True):
                    data = self.conn.recv(BUFSIZE)
                    if not data:
                        break
                message = self.GetString('')


                message = message + "|END"
                self.conn.sendall(message.encode('utf-8'))
            except (ConnectionAbortedError, BrokenPipeError):
                self.s.close()
                print('The game has stopped')
                sys.exit()

        self.SetStatus(NodeStatus.Idle)


        while self.GetStatus() != NodeStatus.Success and self.GetStatus() !=
    NodeStatus.Failure:

            #check if you have to tick a new child or halt the current
            i = 0
            #try:
            for c in self.Children:
                i = i + 1

                if c.GetStatus() == NodeStatus.Idle:
                    c.Execute(args)
                while c.GetStatus() == NodeStatus.Idle:
                    time.sleep(0.1)


                if c.GetStatus() == NodeStatus.Running:
                    self.SetStatus(NodeStatus.Running)
                    self.SetColor(NodeColor.Gray)
                    self.HaltChildren(i + 1)
                    break
                elif c.GetStatus() == NodeStatus.Failure:
                    c.SetStatus(NodeStatus.Idle)
                    self.SetStatus(NodeStatus.Running)

                    if i == len(self.Children):
                        self.SetStatus(NodeStatus.Success)
                        self.SetColor(NodeColor.Green)
                        break

                elif c.GetStatus() == NodeStatus.Success:
                    c.SetStatus(NodeStatus.Idle)
                    self.HaltChildren(i + 1)
                    self.SetStatus(NodeStatus.Failure)
                    self.SetColor(NodeColor.Red)

                    break

                else:
                    raise Exception('Node ' +self.name + ' does not recognize the
    status of child ' + str(i) +'. (1 is the first)' )
```

Basically we took the Sequence's node code as a base and then just switched what happened when the child returns *Success* for what happens when it returns *Failure* and vice-versa. Then to be able to use this node you need add the following lines in build/PacmanAgents.py.

```
from DecoratorNode import DecoratorNode
...
elif (type == 'Decorator'):
        node = DecoratorNode('decorator')
        print('Create Decorator')
```

3.4 Execution of the extension

Now it is time to test our predicted behavior and see what happens. Let's observe what happens when we begin the game and Pac-man is far away from the ghosts.

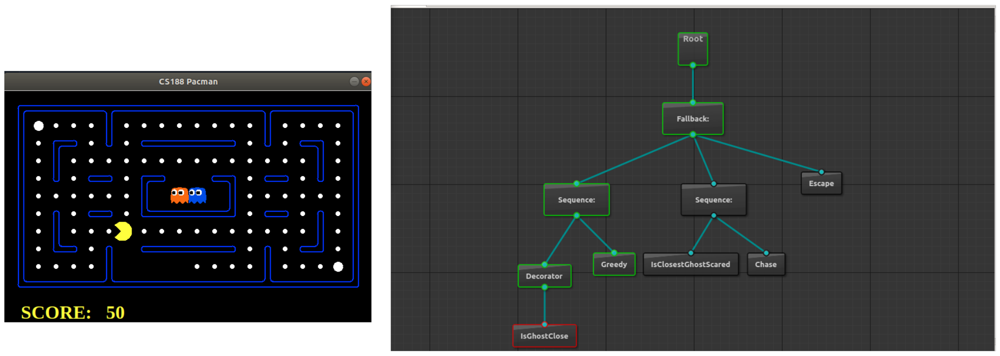


Figure 15: Execution when ghosts are not close

We can see in the figure that the behavior is as expected, the Condition node returns *Failure* and the *Greedy* behavior is activated. Now let's see what happens when the Pacman gets close to a scared ghost.

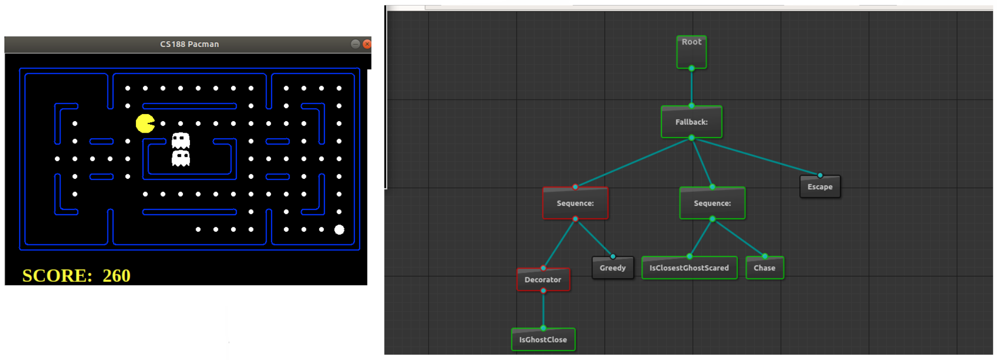


Figure 16: Execution when ghost is close and scared

Once again we can see that the expected behavior is executed. The Fallback node is now ticking its second child and the *Chase* behavior is activated. We can see the Pac-man heading towards the ghosts. Note that this is not the best behavior since it chased them straight to their home and then died once they respawned. This shows that even though behavior trees are naturally responsive, since as soon as the ghosts are not scared the condition node will fail and then the *Escape* behavior will be executed, we are still limited by the frequency of the ticks. Once *Success* is returned there must be another tick to execute the condition again. In order to avoid this problem and make the tree more responsive we would have to do something similar to the first example and find a way to keep the action of checking *Running* so as soon as it failed the *Escape* node would be triggered. Finally we check the third case when there is a ghost nearby and it is not scared.

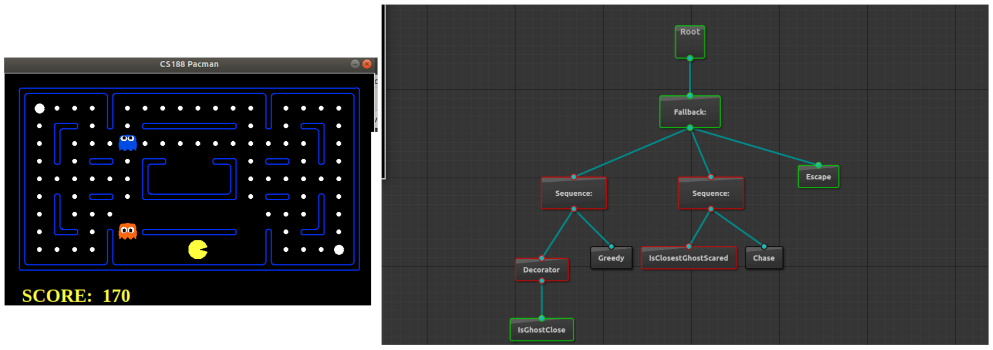


Figure 17: Execution when ghost is close and but not scared

This time the Pac-man escapes from the ghost and the statuses being returned are the ones that were predicted above.

4 Conclusion

The aim of this paper has been to clarify the concept of behavior trees through the examination of two simple examples. Specifically the goal was to explain the execution in such a way that the reader could go on to understand and write code for behavior trees using one of the libraries available. In the end there is an unexpected complexity that comes from trying to implement these behavior trees that is only discovered when programming. We hope it has been an interesting read and that it has allowed the reader to see the subtleties in the execution of behavior trees. To continue your exploration of behavior trees we recommend continuing to play around with the examples given and to continue reading the book that we recommended at the start.

5 Note about Parallel Nodes

Parallel nodes are one of the most difficult to understand and the lack of detail in the original paper has caused each library to implement their own version of this node. This lead to some confusion when developing these examples. Based on what is presented in the book, Parallel nodes should run their children in parallel and then check the status of all the children. Based on a user chosen parameter M the Parallel node itself will return *Success* or *Failure* and while the children are running it will return *Running.* There is no real concurrency. This is because each child is being ticked one after the other in reality. In practice this is not as clear. In Py Trees, the library used in the first example, there are two policies, *success on all* and *success on one.* The first requires all children to return *Success* for the Parallel node itself to do so and the second just one. In the newer versions of the library there is also *success on select* which allows the user to select a group of children that must return *Success* in order for the node to return *Success.* In the book the Parallel node returns *Failure* if N-M children do, N being the total number of children, but in Py Trees only one child needs to return *Failure* for its parent to do so. Once the node reaches a final status it will stop any currently running children. In the other most common library BehaviorTree.CPP there are what are called Reactive nodes that were first used to replace Parallel Nodes. These also run all children but they do so in a round robin fashion, they run one child and based on the returned status they decide what to do with its siblings. The difference here is that the children are ticked one at a time until they reach a final status. The Reactive-Fallback node works like a Fallback node but if a child returns *Running* it goes to the next sibling and if no other node returns *Success* it will come back to that child to continue the execution. The Reactive-Sequence works like a Sequence node but halts the siblings when one child is running. Since then, a Parallel node has been included in the library but not much documentation can be found. After examining the code, it appears to behave like the one in the book. Unlike in Py Trees the user can set the parameter M to control when the node returns *Success* or *Failure.*

References

[1] Behaviours — py_trees 0.6.9 documentation.

[2] Tutorials — py_trees_ros 0.5.14 documentation.

[3] Michele Colledanchise and Petter Ögren. Behavior trees in robotics and ai github.

[4] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI.* CRC Press, jul 2018.