Homework 4: Hero Agents

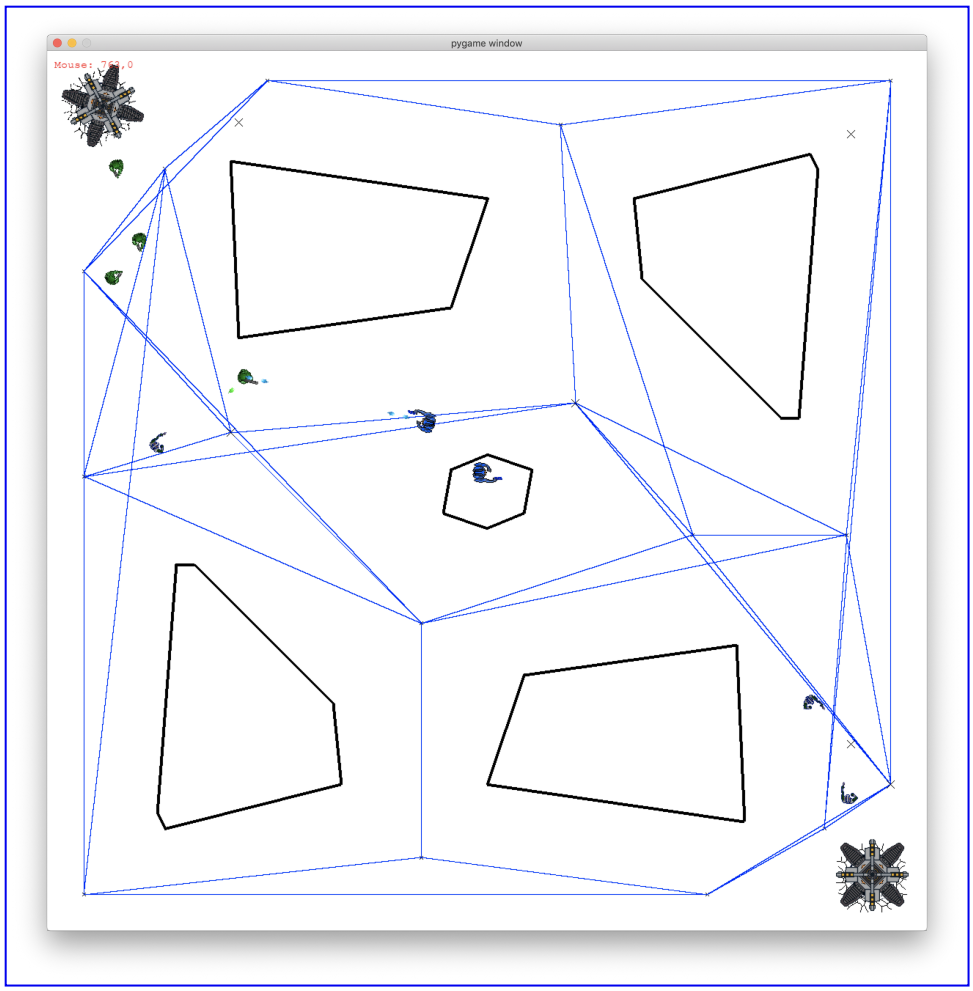
A Multiplayer Online Battle Arena (MOBA) is a form of Real-Time Strategy game in which the player controls a powerful agent called the "Hero" in a world populated with simple, weak, fully computer-controlled agents called "Minions." In this assignment, we will implement the decision-making for Hero agents.

A MOBA has two teams. Each team has a base, which is protected by a number of towers. The goal is to destroy the opponent's base. In MOBAs, bases periodically spawn Minion agents, who automatically attack the enemy towers and bases. Towers and bases can defend themselves; they target Minions before targeting Heroes. Thus Minions provide cover for Heroes, who are much more powerful.

In this version of a MOBA, Heroes have a number of special properties that make them more powerful than Minions:

- Heroes have twice as many hitpoints as Minions.
- Heroes have a more powerful gun.
- Heroes have a secondary attack that does damage to all enemies within a certain radius.
- Heroes can be healed if they return to the base.
- Heroes can dodge bullets by "jumping" a short distance in a given direction (or a random direction).
- Heroes can gain level-ups by killing enemy agents. A level is gained for each kill the Hero makes (the last agent to damage an opponent before it dies). The Hero gains one extra point of damage per level and maximum hitpoints increases by one for each level earned.

In this assignment, you will implement the AI for a Hero agent in a world similar to a MOBA. However, we will substantially change the rules so we can focus on Hero AI without all the complications of a MOBA. In this game, there are no towers and each team has a maximum of three minions at any given time. The minions wander aimlessly, but shoot at Heroes if they ever get too close. Heroes must hunt each other, and the game is scored by how much damage one Hero does to another Hero.



We will build off your previous path network generation solution from **homework 1** and your previous A* implementation from **homework 2**.

Hero decision-making can be anything, but in a MOBA, typically the Hero focuses on gaining extra powers so it can take out enemy Minions at a faster rate. Eventually, a Hero will be powerful enough to quickly take down towers and bases. Since towers and bases target Minions before targeting Heroes, it is beneficial for Heroes to protect friendly Minions and use them for cover. By targeting enemy Minions, a Hero can make it harder for the enemy Hero to take cover. Heroes may engage each other from time to time to disrupt opponent advantage.

However, in this assignment, we will only focus on the Hero vs. Hero aspects of a MOBA. You will implement a Behavior Tree for Hero agents. You will be provided with a special class for Heroes that knows how to execute a behavior tree. In this assignment, you will write the control logic for behavior trees and complete the code for a number of Hero behaviors.

In Hero vs. Hero combat the best solution is to focus on strategy. Heroes have many properties (listed above) that make for interesting trade-offs when deciding what behavior to execute. Examples of strategic decisions include: hunting the enemy Hero, hunting Minions to increase level, using a longer-ranged shooting attack versus a limited-range area effect attack, retreating to the base to heal, hiding from the enemy Hero, hiding from Minions, etc.

The bases will automatically spawn Minion agents, up to a maximum of three. If a Hero dies, it will immediately respawn at the base, but the level will be reset.

Both teams will use the same Minion agent AI, which wanders the map to random places. Minions will shoot at Heroes if they are within firing range.

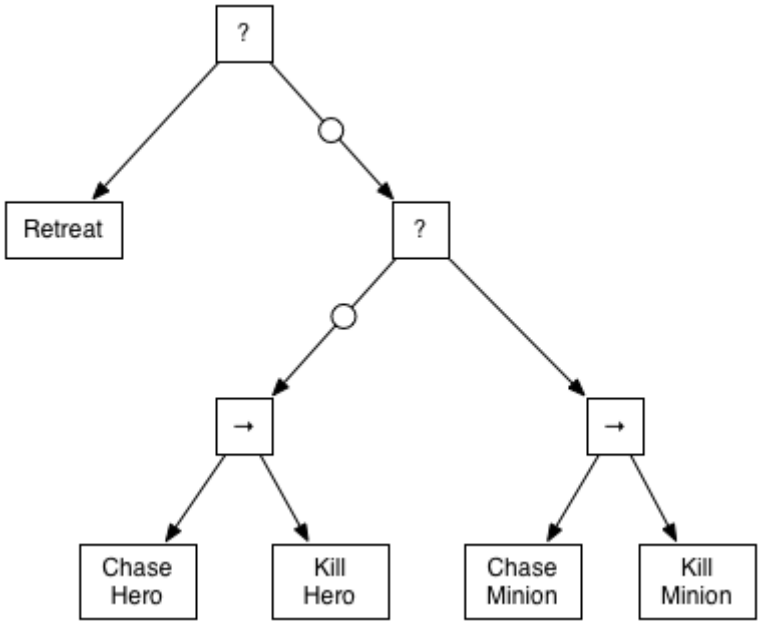
The game score is computed as the cumulative amount of damage one Hero has done to the other Hero. You must implement Hero AI that can result in a higher score than the opponent.

Behavior Trees in a nutshell

A behavior tree is a hierarchically-arranged description of all possible plans of action that an agent can consider, and the conditions under which those plans can be selected or rejected. Behavior trees represent a middle ground between finite state machines, in which all behavior is specified by a designer, and formal planning, in which all decisions are autonomously made. A behavior tree allows designers to specify which plans can be generated and allow the agent to make the final decision on which plan to execute. Since enumerating all possible plans an agent can take may be intractable, behavior trees allow plans to be broken up into sub-plans that are reused in a hierarchical fashion.

Behavior trees are made up of different types of nodes. Internal nodes represent behavior flow control, specifying how to combine sub-trees into plans. Leaf nodes are tasks, specifying behaviors an agent should execute in the world. Tasks also specify the conditions under which tasks can and cannot be performed. The behavior tree acts as a pseudo-mathematical definition of the agent's brain, with tasks performing call-backs to the agent body to act in the world and affect the world state.

Consider the behavior tree below:



At every tick, the agent calls the execute() function on the root of the tree. Each type of node has a different logic for execution. The nodes labeled with question marks are Selector nodes. A Selector node tries to execute each child until one indicates that it can be executed. Thus the behavior tree above first tries to retreat. If it is not appropriate to retreat, it tries something else. The nodes labeled with arrows are Sequence nodes. A Sequence node tries to execute each child in turn until all children have reported that they have executed successfully.

Thus, the logic represented by the above behavior tree is as follows. The agent tries to retreat, returning to the base for healing. If it is not appropriate for the agent to retreat—it has not lost enough health—it then tries to chase the hero and then kill the hero. If that sequence fails for any reason, the agent tries to chase a minion and then kill the minion.

The circles in the diagram are a special type of node called a Daemon. A Daemon checks a condition in the world and decides whether to execute its child or return instant failure. For example, the top-most Daemon might check whether the health of the agent is greater than 50%. Thus, the agent can only chase heroes and minions if

it has enough health. The lower Daemon might check whether the agent is more powerful than the enemy Hero. Thus, the agent can only execute the sequence of chasing and killing the enemy Hero if it is strong enough to do so. Daemons short-circuit the tree, allowing a decision to recurse to be made quickly. They also allow the tree to quickly stop executing on a sub-tree if certain conditions become false.

An agent that implements a behavior tree calls the execute() function of the root node every tick. The purpose of internal nodes (Selectors, Sequences, and Daemons) is to figure out which leaf node (task) should have its execute() function called that tick. Thus think of a behavior tree as a cascade in which control flows from the root to exactly one leaf node. The execute() function of the leaf node is called and any appropriate action taken.

Nodes can return one of three values: **success** (True), **failure** (False), or **running** (None). Success means that the behavior has run to completion and has achieved what it is supposed to achieve. For example, a successful retreat means having the agent's health restored to 100%. Failure means that either the behavior is not applicable in the current world state or that it is no longer able to achieve the success conditions and should terminate. In a non-turn-based game world, some behaviors require many ticks to complete successfully (or to fail). For example, retreating requires navigating back to the home base, which may take many ticks. The running (None) return value means that the node requires more time to determine whether it has succeeded or failed.

Logic for a Selector node:

The execute() function of a Selector node must decide which child should execute. It tries to execute each child in order until one **does not** return failure. However, only one child can have its execute() function called. Therefore, a Selector remembers which child should be tried on the current tick.

1. If all children have been tried and none have succeeded, then the Selector itself fails and returns False.
2. Otherwise:
    1. If the current child's execute() returns True, then the Selector itself succeeds and returns True.
    2. If the current child's execute() returns False, then update the current child and return None (indicating that this node should be tried again next tick).
    3. If the current child's execute() returns None, then return None.

Logic for a Sequence node:

The execute() function of a Sequence node must decide which child should execute. It tries to execute each child in order until one returns a failure. However, only one child can have its execute() function called. Therefore, a Sequence remembers which child should be tried on the current tick.

1. If all children have been tried and all have succeeded, then the Sequence itself succeeds and returns True.
2. Otherwise:
    1. If the current child's execute() returns True, then update the current child and return None (indicating that this nodes would be tried again next tick).
    2. If the current child's execute() returns False, then the Sequence itself fails and returns False.
    3. If the current child's execute() returns None, then return None.

Leaf nodes:

The execute() functions of a leaf node must do three things. First, it must check applicability conditions and return failure immediately if the behavior is not applicable to run at this time. Second, it must call back to the agent to perform any appropriate actions. Whatever the leaf node does, it should not require a lot of computation because execute() is called every tick. Third, it must determine whether the behavior has succeeded. If the behavior is applicable but has not succeeded, it should indicate that it is still running by returning None.

The first time a leaf node executes, an additional enter() function will be called to do any one-time set up for execution. Enter() will only be called once per leaf node. However, if the tree is ever reset, then each leaf node will have its enter() function called again the next time the node is visited.

Daemon nodes:

The execute() function of a Daemon node checks the applicability of an entire sub-tree (as opposed to a single behavior). The execute() function checks the applicability conditions and returns False if the conditions are not met. If the conditions are met, the execute() function of its single child is called and the Daemon returns the child's return value as if it were its own. Daemons assume a single child.

In this assignment, you will implement the logic for Selector and Sequence nodes. You will be given the opportunity to test your implementations before working of Hero agents. You will then be asked to implement the execute() functions for a number of different types of behaviors for MOBA Heroes. We will provide you with several different tree configurations to test your Hero agent with.

What you need to know

Please consult previous homework instructions for background on the Game Engine. In addition to the information about the game engine provided there, the following are new elements you need to know about.

Agent

Three things are newly relevant to this assignment. (1) Agents have hitpoints. (2) Agents can be part of a team. (3) Agents can shoot in the direction they are facing.

Member variables:

- hitpoints: the amount of health the agent has. The agent dies when hitpoints reaches zero.
- team: the symbolic name of the team the agent is on.

Member functions:

- getHitpoints(): returns the number of hitpoints.
- getTeam(): returns the symbol of the team the agent is on.
- turnToFace(pos): turn the agent to face a particular point (x, y).
- shoot(): fire the agent's gun in the direction the agent is facing. The agent can only fire after a certain number of ticks have elapsed.

Note: To effectively shoot at something, first turn the agent to face the target (or to the point the agent wishes to fire at) with turnToFace() and then call shoot().

BehaviorTree

BehaviorTree is defined in behaviortree.py. A BehaviorTree is a container class that maintains a pointer to the root node (BTNode) of the behavior tree. At every tick, the BehaviorTree will call the execute() function on the root node.

When the root node of the behavior tree returns success or failure, then the tree is reset for another run next tick.

A BehaviorTree object also knows how to build a complete tree from a specification. When the tree is built, each node is instantiated as an object in memory with pointers to its children, but no nodes are actually executed at that time. Once the tree has been built, it is ready for execution during the gameplay loop.

Member variables:

- tree: a pointer to the root of the behavior tree.

- running: a boolean indicating that the tree is running, meaning that ticks are sent into the root node.

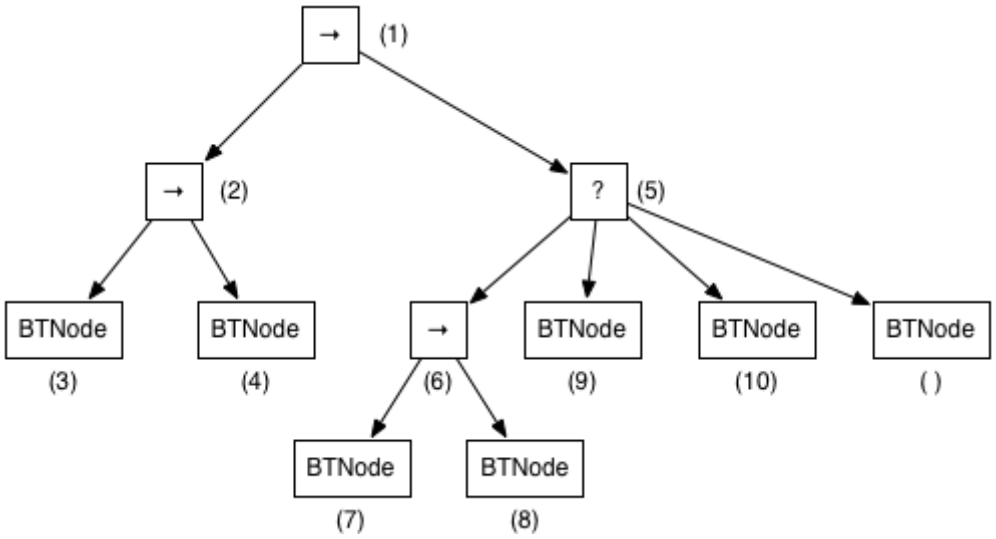
Member functions:

- update(delta): called every tick. Calls the execute() of the root node of the tree. Delta is the time since the last tick.
- buildTree(spec): Instantiates each node of the tree according to a specification (see below).
- printTree(): causes all of the nodes in the behavior tree to print identifying information to the console in depth-first visit order.
- start(): start the tree running.
- stop(): stop the tree from running.

The buildTree() function takes in a specification that tells the BehaviorTree the type of each node in the tree, the child/parent relationships between each node, and any parameters that can be known at build time. The build specification language is a sub-set of the Python language and is as follows:

- A List indicates a child-parent relationship such that the first element of the list is a parent and each subsequent element (2 through N) in the list is a child of the first element.
- If the first element in a List is a class name (a BTNode or a sub-class of a BTNode), then a node of that class type is instantiated.
- If the first element in a List is a Tuple, then the first element of the Tuple must be a class name (a BTNode or a sub-class of a BTNode) and all subsequent elements in the Tuple are parameters to be passed into the node's constructor.
- List elements that are not first in the List can be a class name, a Tuple, or a List. Class names and Tuples indicate leaf node children. A List indicates that a new child node should be created (see above) and that this child in turn has its own children.

For example: [(Sequence, 1), [(Sequence, 2), (BTNode, 3), (BTNode, 4)], [(Selector, 5), [(Sequence, 6), (BTNode, 7), (BTNode, 8)], (BTNode, 9), (BTNode, 10), BTNode]] creates the following behavior tree. In this example, we are assuming that the second element in a Tuple is an identification parameter for the node. Note that one node does not have any parameters.



BTNode

BTNode is defined in behaviortree.py. a BTNode is a parent class for nodes in a behavior tree. BTNode assumes that a node has children, but leaf nodes do not use the children. The primary functionality of a BTNode is it's execute() function, which will be called in the course of a tick. The first time a BTNode is visited in the course of execution, the enter() function is also called. BTNode assumes a node has children, but that is not necessarily the

case for behavior types, which will always be leaf nodes. If a BTNode is an internal node, it keeps track of its current child with an index number.

Member variables:

- id: An identifier for the node. Useful for printing out the tree and debugging.
- agent: a pointer back to the agent that the behavior tree is controlling.
- children: a list of BTNodes.
- current: the index of the current child to be tried if the BTNode is an internal node (e.g., Selector or Sequence).
- first: a boolean indicating that this BTNode has not been visited before during behavior execution and that the enter() function should be called upon execution.

Member functions:

- parseArgs(args): you can override this function to take any arguments passed into the BTNode upon instantiation and do what is necessary to configure the node. The default functionality is to set the id to the list of args.
- execute(delta): Performs the behavior of the node. Delta is the amount of time since the last tick of the game loop. This function returns True if the behavior succeeds, False if the behavior fails or if the behavior is not applicable, and None if the behavior is running and requires more time before succeeding or failing.
- enter(): Called the first time the node is executed. Performs any set up necessary for execution or any one-shot behaviors that should only be done on the first call to execute().
- printTree(): prints out the id of the node and then recursively prints its children in depth-first visit order.
- reset(): called when the behavior tree has completed execution. Restores any defaults, sets the current index back to zero, and sets the first boolean to True.
- addChild(child): append a child BTNode to the list of children.
- setID(id): sets the ID of the node.
- getID(): returns the ID of the node.
- getAgent(): returns the agent of the node.
- getChild(index): returns the child node at the given index.
- getChildren(): returns a list of all child nodes.
- getNumChildren(): returns the number of child nodes.
- getCurrentIndex(): returns the current index value.
- setCurrentIndex(index): sets the index value for the node.

For execute() to control the agent, it must make call-backs via the agent member variables. This is node in leaf nodes. For example, if the behavior of a state is to make the agent shoot, the execute() function can call self.agent.shoot().

The execute() function for internal nodes should determine which single child should have its execute() function called. The execute() function for leaf nodes should implement the intended behavior, making call-backs to the agent.

The constructor for the base BTNode class takes a number of arguments, as a list. But it doesn't know what they are meant to be. Constructors for sub-classes can look at the arguments passed in through args and pick out the relevant information and store it or compute with it. For example, TestNode takes the first element in the list and sets the ID.

For example, one might want a **Taunt** behavior, and the tree will be built to always taunt a particular enemy agent. The tree build specification could be [Sequence, (Taunt, enemyhero), [Selector, Retreat, [Sequence, ChaseHero, KillHero]]] where enemyagent is a pre-computed reference to the enemy hero. Even though the Taunt sub-class is expecting an argument, it will just be passed in to the constructor as args[0]. Use parseArgs(args) to capture the parameter and use it. For example:

```
class Taunt(BTNode):
    def parseArgs(self, args):
        BTNode.parseArgs(self, args)
        self.target = None
        if len(args) > 0:
            self.target = args[0]

    def execute(self, delta = 0):
        ret = BTNode.execute(self, delta)
        if self.target is not None:
            print(f"Hey {self.target} I don't like you!")
        return ret
```

In the sample above, when the behavior tree is built from the given specification, the Taunt object will be created and the reference to enemyhero will be passed in as args[0] and self.target will be set to enemyhero.

VisionAgent

A VisionAgent is a sub-class of StateAgent. VisionAgent is given a viewangle, a number of degrees that the agent can see. Every tick, the VisionAgent asks the GameWorld what it can see, based on its view angle, and maintains a list of visible Movers (agents, bullets, towers, and bases). For this assignment, Minions have a view angle of 360 degrees, meaning they can see everything around them irrespective of what direction they are facing.

Member variables:

- viewangle: the number of degrees the agent can see, centered around the front of the agent (i.e., 1/2 viewangle clockwise from the agent's orientation, and 1/2 viewangle counterclockwise from the agent's orientation).
- visible: a list of Movers that is currently visible (re-computed every tick).

Member functions:

- getVisible(): returns a list of visible Movers.
- getVisibleType(type): returns a list of visible Movers of a given class type.

MOBAAgent

A sub-class of VisionAgent, specialized for the MOBA. MOBAAgents do two noteworthy things. First, **MOBAAgents die whenever they collide with an Obstacle**. Second, they can compute a list of points in navigable space in the even that the agent needs to choose a point to move to without worrying about whether that point is inside an Obstacle (the agent will still have to figure out if it can actually move there).

Additionally, new to this homework, MOBAAgents keep track of the agent that last did damage to them, and have a **level**, which is used to compute maximum hitpoints and amount of damage they can do.

Member variables:

- maxHitpoints: the maximum number of hitpoints the agent can have.
- lastDamagedBy: a pointer to the last agent to have done damage to this agent.
- level: the level of the agent, which is used to determine maximum hitpoints (one extra hitpoint per level), and amount of damage to do (one extra point of damage per level).

Member functions:

- getMaxHitpoints(): returns the maximum number of hitpoints the agent can have.
- getPossibleDestinations(): returns a list of points that are not in Obstacles.
- getLevel(): return the level, an integer.

Hero

Abstract base class, which is a sub-type of MOBAAgent.

In this assignment, the Hero can dodge, use an area effect attack, be healed by bases, and level-up whenever it kills another agent of any type.

Member variables:

- candodge: True when the agent can dodge.
- dodgeTimer: counts the number of ticks until the agent can dodge.
- dodgeRate: the number of ticks until an agent can dodge again.
- canareaeffect: True when the agent can use the area effect attack.
- areaEffectTimer: counts the number of ticks until the agent can use the area effect attack.
- areaEffectRate: the number of ticks until an agent can use the area effect attack.
- areaEffectDamage: the amount of damage the area effect attack does.

Member functions:

- dodge(angle): causes the agent to jump in the direction of the given angle (0 <= angle <= 360). The angle parameter is option and, if omitted, the agent will dodge in a random direction.
- areaEffect(): use the area effect attack.

MyHero

MyHero is defined in MyHero.py. MyHero is a specialization of a Hero and also has BehaviorTree as a base class. Every tick, when MyHero.update() is called, the BehaviorTree.update() is also called.

When the agent is created (at the beginning of the game, or after it dies and respawns), it builds a behavior tree. The function treeSpec(), is a custom function that provides the specification for the behavior tree.

MOBABullet

A special Bullet class for the MOBA. MOBABullets differ from regular bullets in that they are range-limited.

There are four sub-classes of MOBABullet: SmallBullet, BigBullet, BaseBullet, and TowerBullet. These bullets are specific to Minions, Heroes, Bases, and Towers, respectively and do different amounts of damage.

Base

Each team in a MOBA has a Base. Bases spawn minions at regular intervals as long as the maximum number of minions allowed at any given time has not been reached. Bases cannot be damaged as long as there are towers present on the same team. Bases can heal Heroes—if a Hero touches a base, its hitpoints are restored to maximum value.

Member variables:

- team: the symbol of the team that the base is on.
- hitpoints: the amount of health the base has.

Member functions:

- getTeam(): returns the symbol of the team the base is on.
- getHitpoints(): returns the number of hitpoints.

MOBAWorld

A special type of GameWorld for MOBAs. MOBAWorld is a type of GatedWorld. The MOBAWorld keeps track of bases and towers, in addition to NPCs, Bullets, and the agent.

New to this assignment: the MOBAWorld keeps track of the score for each team.

Member variables:

- score: a dictionary indexed on each team symbol. The score is the amount of damage a Hero does to any enemy Hero.

Member functions:

- getNPCs(): returns a list of NPCs in the game (includes all agents not controlled by the player).
- getNPCsForTeam(team): return a list of NPCs part of the given team.
- getEnemyNPCs(myteam): return a list of NPCs that are not part of the given team.
- getAgent(): returns the player-controlled character.
- getBases(): return a list of all bases.
- getBasesForTeam(team): return a list of all bases on a given team.
- getEnemyBases(team): return a list of all bases not on the given team.
- getTowers(): return a list of all towers.
- getTowersForTeam(team): return a list of all towers on a given team.
- getEnemyTowers(team): return a list of all towers not on the given team.
- getBullets(): return a list of all bullets in the world at that moment.
- getScore(team): return the score for a team.

BTNode Subclasses

This assignment includes a number of BTNode subclasses. Some are complete and can be used for testing. Some require completion.

Selector: Selectors determine which of its children should be executed.

Sequence: Sequences try to execute all of their children.

TestNode: TestNode sets its ID to the first argument provided upon instantiation. TestNode immediately terminates execution with success if the ID is an even integer and failure otherwise.

DelayTestNode: DelayTestNode operates like a TestNode, but it waits for a given number of ticks to pass before reporting success or failure. The amount of time (number of ticks) required to "execute" a DelayTestNode is given by the second argument provided upon instantiation. For example, a build tree specification that includes (DelayTestNode, 1, 10) will create a DelayTestNode with and id of 1 and will require 10 calls to execute before it returns success or failure.

Taunt: Taunt is instantiated with a reference to an enemy agent. At every execution call, Taunt will print a disparaging message, referencing the targetted agent. The reference to the enemy agent is the first argument given at tree instantiation.

MoveToTarget: MoveToTarget must be provided an ID as the first argument and an (x, y) point as a second argument. MoveToTarget instructs the agent to navigate to the specified point and returns success when it is within one agent radius of the point.

Retreat: Instructs the agent to return to its team's base to be healed. Retreat is only applicable if the agent has lost more than half of its hitpoints. The behavior succeeds when the hitpoints have been restored to their maximum amount.

ChaseMinion: Instructs the agent to move withing firing range of a minion. The behavior reports failure if the targeted minion dies prematurely. It reports success when the agent is within firing range of the minion.

KillMinion: Instructs the agent to continuously shoot at a minion. Reports failure if the minion moves out of firing range or behind an obstacle. Reports success if the minion dies.

ChaseHero: Instructs the agent to move withing firing range of an enemy hero. The behavior reports failure if the targeted hero dies prematurely. It reports success when the agent is within firing range of the hero.

KillHero: Instructs the agent to continuously shoot at a hero. Reports failure if the hero moves out of firing range or behind an obstacle. Reports success if the hero dies.

HitpointDaemon: Reports failure if agent hitpoints fall below a given percentage.

BuffDaemon: Reports failure if the agent's level is close to the enemy hero's level. A required distance is given. For example, a BuffDaemon may be configured to require the agent to have a level that is greater than 2 levels above the enemy hero's level.

Instructions

To complete this assignment, you must implement the logic for Selector and Sequence node types. You must additionally complete the execution logic for a number of behavior tree node types specific to hero vs. hero combat.

Use your solution to **homework 1** to generate a path network. The instructor can provide you with a default solution if necessary. Use your A* implementation from **homework 2**. The instructor can provide you with a default solution if necessary.

The following steps are required to complete the assignment.

Step 1: Copy your mybuildpathnetwork.py function from homework 1. Copy your astarnavigator2.py from homework 2.

Step 2: Implement the execution logic for Sequence and Selector node types. Modify Sequence.execute() and Selector.execute() in btnode.py. See the explanation of Sequence and Selector nodes above for success conditions and failure conditions.

Test your implementations using test.py:

```
> python runtest.py
```

test.py uses TestNode and DelayTest and does not invoke the entire game engine.

Step 3: Create a behavior tree specification to control a MyHero agent. The goal of the behavior tree is to guide the MyHero so as to be the first to do 100 points of damage against the enemy hero. In mybehaviors.py, modify

treeSpec() so that it returns the behavior tree specification to be used by MyHero OR modify myBuildTree() to manually build the tree and return the root of the tree.

A number of leaf node behavior and daemon class types have been provided for you to use:

Class name	Description	Example specifications
Taunt	Print disparaging comment, addressed to a given NPC. Takes two parameters: (1) a reference to an NPC, and (2) optional node ID string.	(Taunt, agentref) (Taunt, agentref, "id")
MoveToTarget	Move the agent to a given (x, y). Takes two parameters: (1) an (x, y) point, and (2) optional node ID string.	(MoveToTarget, (x, y)) (MoveToTarget, (x, y), "id")
Retreat	Move the agent back to the base to be healed. Takes two parameters: (1) the percentage of total hitpoints that need to be lost before retreating, and (2) optional node ID string.	Retreat (Retreat, 0.5) (Retreat, 0.75, "id")
ChaseMinion	Find the closest minion and move to intercept it. Takes a single parameter: optional node ID string.	ChaseMinion (ChaseMinion, "id")
KillMinion	Kill the closest minion. Assumes it is already in range. Takes a single parameter: optional node ID string.	KillMinion (KillMinion, "id")
ChaseHero	Move to intercept the enemy Hero. Takes a single parameter: optional node ID string.	ChaseHero (ChaseHero, "id")
KillHero	Kill the enemy hero. Assumes it is already in range. Takes a single parameter: optional node ID string.	KillHero (KillHero, "id")
HitpointDaemon	Only execute children if hitpoints are above a certain percentage. Takes two parameters: (1) the percentage of total hitpoints that need to be remaining to trigger a condition success, and (2) optional node ID string.	HitpointDaemon (HitpointDaemon, 0.5) (HitpointDaemon, 0.5, "id")
BuffDaemon	Only execute children if agent's level is significantly above enemy hero's level. Takes two parameters: (1) the number of levels greater than the enemy hero's level necessary to not fail the condition check, and (2) optional node ID string.	BuffDaemon (BuffDaemon, 2) (BuffDaemon, 3, "id")

You may also write your own behaviors or subclass existing behaviors.

(Note: If you choose to modify myBuildTree(), you may find it advantageous to use the makeNode() helper function. makeNode() has two required arguments: (1) a class name, which should be BTNode or a sub-class of BTNode, and (2) a reference to the agent to be controlled. makeNode() can take an arbitrary number of additional arguments, which will be passed into the newly created BTNode and parsed using BTNode.parseArgs(). For example, makeNode(Taunt, agent, "Fred", "id1") will create a Taunt tree node for the agent, set the target of the taunt to "Fred", and set the node's ID to "id1".)

To test your implementations, use runherocompetition.py to run a MyHero agent using your behavior tree nodes against different Hero types:

> python runherocompetition.py MyHero MyHero
> python runherocompetition.py MyHero BaselineHero
> python runherocompetition.py BaselineHero BaselineHero
> python runherocompetition.py MyHero BaselineHero2
> python runherocompetition.py MyHero BaselineHero3

BaselineHero is a bare-bones implementation of Hero AI that you can use to test against. BaselineHero agents simply navigate to the nearest Hero and starts shooting. BaselineHero2 will return to the base for healing if it takes too much damage. BaselineHero3 uses a more sophisticated strategy that will not be revealed.

Grading

Grading occurs in two stages.

Stage 1 (5 points): Grading of your implementation of Selector.execute() and Sequence.execute() will be done by performing a suite of tests similar to that in runtest.py and comparing execution orders.

Stage 2 (5 points): Grading of your behavior tree specification will be performed by running complete games in which your agent faces off against baseline agents. These games have a score. Each team receives one point for every point of damage done by a Hero agent to another Hero agent. Your MyHero agent will be tested against three baseline agents (described above).

- 2 points: Be the first to score 100 points against BaselineHero. Win 2 out of 3 games.
- 2 points: Be the first to score 100 points against BaselineHero2. Win 2 out of 3 games.
- 1 point: Be the first to score 100 points against BaselineHero3. Win 2 out of 3 games.

For pragmatic reasons, games will be limited to 5000 ticks. if no agent is able to deal 100 damage at 5000 ticks the game count as a draw and no points will be given.

As a rule of thumb, this should be less than 5 minutes on most computers.

Extra credit (+1 points): beat the RiedlHero agent. Be the first to score 100 points against RiedlHero agent. Win 2 out of 3 games.

Hints

To test your Sequence and Selector implementations, change the ID numbers of TestNodes in runtest.py to different even and odd numbers. You may also want to change TestNodes to DelayTestNodes.

The player-controlled character is a GhostAgent, which can move through walls without being obstructed. Baseline Heroes and Minions will not target GhostAgents. If you change the player-controlled character to a Hero in runhero.py or runherocompetition.py, then you will be able to dodge by pressing 'j' (for jump), and use the area effect attack by pressing 'a'.

Implement custom behaviors to perform more sophisticated tactics. If you do so, you may want to consider the following tips:

- Use agent.getVisible() and agent.getVisibleType() to figure out what the agent can shoot at.
- Remember, an agent shoots in the direction it is facing, so use agent.turnToFace(enemy) before agent.shoot().
- Implement a smart dodging algorithm that figures out which direction to dodge to avoid an incoming bullet (use getVisibleType(Bullet) to see bullets) and not jump into obstacles.
- Implement leading shots so you can shoot ahead to where an enemy will be instead of where they are at the time the bullet is released.
- If the Hero is hunting an opponent Minion or Hero, consider having the Hero go to where the enemy is also going.
- Do not kill the enemy base—there is no strategic advantage in doing so. If you kill the base, the enemy Hero will not respawn and you will not be able to get more points.

Submission

To submit your solution, upload your modified btnode.py, mybehaviors.py, mybuildpathnetwork.py, and astarnagivator2.py.

You should not modify any other files in the game engine.

DO NOT upload the entire game engine.