```
import sys
import math
import random
from queue import *

###################
# Debugging Flags #
###################
MSG_CONTENT = ""
MSG_MAXLEN = 42
MSG_RATE = 12
MSG_RANDIDX = -1
MSG_DELAY = MSG_MAXLEN
MSG_START = 0
MSG_END = MSG_MAXLEN
MSG_OUTPUT = True
SHOW_RESOLUTION = False
SHOW_ENEMY_ATTACKS = False
SIMULATE_ENEMY = True

#########################
# Hitchhiker's Guide :O #
#########################
HITCHHIKER_GALAXY_QUOTES = [u'\u201cListen, three eyes,\u201d he said, \u201cdon\u2019t you try to outweird me, I get stranger things than you free

# Game Statics
MAX_INT = 65535
FACTORY_UPGRADE_COST = 10
BOMB_PRODUCTION_COOLDOWN = 5

# Target Scoring Constants
PRODUCTION_MULTIPLIER = 10
BOMB_SCORE_THRESHOLD = 12.3
BOMB_TROOP_THRESHOLD = 50

# Movement Constants
TROOP_OFFENSIVE = 1.00 # Sends this % of troops against superior enemies
TROOP_DEFENSIVE = 1.00 # Sends this % of troops to reinforce friendly targets
TROOP_OFFENSIVE_MULTIPLIER = 1.17
TROOP_EXCESS_NEUTRAL = 1
TROOP_EXCESS_ENEMY = 1
ENEMY_OFFENSIVE = 1.53 # How offensive is the enemy
ENEMY_DEFENSIVE = 1.00 # How defensive is the enemy
ENEMY_EXCESS_NEUTRAL = 1
ENEMY_EXCESS_ENEMY = 1

# Game Variables
MAX_LINK_DISTANCE = 7 # 3->{26|24} 5->{34|16} 7->{37|13} 9->{34|16} 13->{26|24} 20->{26|24}
NUM_FACTORIES = 0
INITIAL_FACTORY = -1
INITIAL_FACTORY_ENEMY = -1
FRONTLINE_FACTORY = -1
FRONTLINE_DISTANCE = MAX_INT
CYBORGS_OWN = 0
CYBORGS_ENEMY = 0
num_bombs = 2

# Map Variables
adjList = [] # Adjacency List ordered by shortest distance
adjMatrix = [] # Adjacency Matrix
factoryInfo = [] # Information regarding each factory
troopInfo = [] # Packets for each troop movement
bombInfo = [] # Packets for each bomb movement

# Floyd-Warshall APSP matrix with backtracking
floydWarMatrix = [] # Store shortest distance
floydWarPath = [] # Stores complete path to objective
floydWarNext = [] #TODO: Optimization --> Stores next target?

# Simulation for next n turns
SIMUL_TURNS = 21
simulFac = [] # Simulated Factories for attack options

# Actions
turnMoves = [] # Commands for current turn
turnBombs = [] # Commands to send bombs
turnIncs = [] # Commands to upgrade factories
turnOne = True # Selector for initialization turn events

# Helper Functions
def readMaxAvailTroops(simulStates):
    # if (simulStates[0].production == 0): # Non-production factory
    #     return (simulStates[0].troops, 0)
    turn = 0
    ttt = MAX_INT
    availTroops = MAX_INT
```

```
    for state in simulStates:
        curTroops = state.troops if (state.owner == 1) else -state.troops
        if (curTroops < availTroops and ((turn < ttt) if availTroops < 0 else True)):
            availTroops = curTroops
            ttt = turn
        turn += 1
    return (availTroops, ttt)

def closestEnemy(curFac):
    nearestFactory = -1
    nearestDistance = MAX_INT
    for facID in range(NUM_FACTORIES):
        if (adjMatrix[curFac.ID][facID] < nearestDistance and factoryInfo[facID].owner == -1):
            nearestDistance = adjMatrix[curFac.ID][facID]
            nearestFactory = facID
    if (nearestFactory != -1):
        return (nearestFactory, nearestDistance)
    else:
        return (nearestFactory, MAX_INT)

def closestFriendly(curFac):
    nearestFactory = -1
    nearestDistance = MAX_INT
    for facID in range(NUM_FACTORIES):
        if (adjMatrix[curFac.ID][facID] < nearestDistance and factoryInfo[facID].owner == 1):
            nearestDistance = adjMatrix[curFac.ID][facID]
            nearestFactory = facID
    if (nearestFactory != -1):
        return (nearestFactory, nearestDistance)
    else:
        return (nearestFactory, MAX_INT)

def getValidTargets(curID, curBlacklist, simulStates): # Returns list of Factory objects
    validTargets = []
    for adj in adjList[curID]:
        ignore = False
        targetFac = factoryInfo[adj[0]]
        targetStates = simulStates[adj[0]]
        ttt = floydWarMatrix[curID][targetFac.ID]+1

        print("Evaluating target Factory: {0}".format(adj[0]), file=sys.stderr)
        # Filters targets and add some to valid target list
        if (targetStates[ttt].owner == 1):
            print("IGNORE -> Owned", file=sys.stderr)
            ignore = True # Ignore our own factories (no attack necessary)
        # else:
        #     if (targetFac.production == 0):
        #         print("IGNORE -> Production 0", file=sys.stderr)
        #         ignore = True # Ignore factories that do not give production

        # Ignores blacklisted targets
        if (targetFac.ID in curBlacklist):
            print("IGNORE -> Blacklisted", file=sys.stderr)
            ignore = True

        # Adds valid targets
        if (ignore):
            continue
        else:
            print("VALID! :)", file=sys.stderr)
            validTargets.append(targetFac)
    return validTargets

# Decision Making
def scoreTarget(tgtID, curID):
    distanceMultiplier = (1/max(1,(adjMatrix[curID][tgtID]**2)))
    score = 0
    # Penalty if closer to enemy
    friendlyReached = False
    enemyFacID = -1
    for facTup in adjList[tgtID]:
        if (factoryInfo[facTup[0]].owner == 1):
            friendlyReached = True
        elif (factoryInfo[facTup[0]].owner == -1 and not friendlyReached):
            enemyFacID = facTup[0]
    score += 7 if factoryInfo[tgtID].troops < factoryInfo[curID].troops else 0
    score += factoryInfo[tgtID].production*PRODUCTION_MULTIPLIER*distanceMultiplier # Rewards production
    score -= max(5, factoryInfo[tgtID].troops*0.5)*distanceMultiplier # Penalizes troops
    if (enemyFacID != -1):
        score -= max(10, factoryInfo[enemyFacID].troops)
    # print("{0} Score: {1}".format(tgtID, score), file=sys.stderr)
    return score

def scoreRedistribution(tgtID, curID, closestEnemyDistance):
    distanceMultiplier = (1/max(1,closestEnemyDistance)**2)
    score = 0
    score += 7 if factoryInfo[tgtID].troops < factoryInfo[curID].troops else 0
```

```
    score += factoryInfo[tgtID].production*PRODUCTION_MULTIPLIER*distanceMultiplier # Rewards production
    score -= max(5, factoryInfo[tgtID].troops*0.5)*distanceMultiplier # Penalizes troops
    # print("{0} Score: {1}".format(tgtID, score), file=sys.stderr)
    return score

def scoreBomb(tgtID, curID):
    distanceMultiplier = (1/max(1,(floydWarMatrix[curID][tgtID]**0.5)))
    score = 0
    score += factoryInfo[tgtID].production*PRODUCTION_MULTIPLIER*distanceMultiplier # Rewards production
    score += max(10, factoryInfo[tgtID].troops)*distanceMultiplier # Rewards troops
    # print("{0} Score: {1}".format(tgtID, score), file=sys.stderr)
    return score

def should_bomb(facID):
    if (factoryInfo[facID].owner != -1):
        return False
    if (num_bombs < 1):
        return False
    if (factoryInfo[facID].troops > BOMB_TROOP_THRESHOLD):
        return True
    if (factoryInfo[facID].production > 0):
        return True
    return False

def should_reinforce(facID):
    if (factoryInfo[facID].owner != 1):
        return False
    if (factoryInfo[facID].troops < 1):
        return False
    if (factoryInfo[facID].production < 3):
        return False
    return True

def needed_upgradeTroops(curFac, tgtFac, resolutions):
    ttt = floydWarMatrix[curFac.ID][tgtFac.ID]+1
    arrivalState = resolutions[tgtFac.ID][ttt]
    # if (arrivalState.owner == -1 and arrivalState.troops > 0):
    #     return -1
    if (arrivalState.production < 3):
        requestTroops = FACTORY_UPGRADE_COST - (arrivalState.troops if arrivalState.owner == 1 else -arrivalState.troops)
        if (requestTroops > 0):
            return requestTroops
    return -1

def needed_reinforcements(ttt, availTroops, resolution):
    arrivalState = resolution[ttt]
    if (arrivalState.production < 1): #TODO: Do not reinforce no production factories?
        return -1
    # Will be in time to reinforce
    requestTroopsTup = readMaxAvailTroops(resolution)
    requestTroops = requestTroopsTup[0] # Reads how many troops needed
    requestTurn = requestTroopsTup[1] # Reads when it is needed
    if (requestTroops < 0):
        return min(availTroops, -requestTroops)
    return -1

def viable_upgrade(resolution, prevUpgrades):
    updatedResolutions = []
    # Builds updated states for next FACTORY_UPGRADE_COST turns
    for i in range(max(SIMUL_TURNS, FACTORY_UPGRADE_COST)):
        curState = resolution[i]
        newState = FactoryMsg(curState.ID, [curState.owner, curState.troops, curState.production, curState.cooldown])
        newState.troops = curState.troops - FACTORY_UPGRADE_COST*(prevUpgrades+1) + i*(prevUpgrades+1)
        newState.updateOwnership()
        updatedResolutions.append(newState)
    # Checks for viability given an upgrade
    for state in updatedResolutions:
        if (state.owner != 1):
            return False
    return True

# availFacs tuple = (curTroops, ID)
def canOverwhelm(availFacs, targetFac, simulStates):
    availFacsTtt = []
    targetStates = simulStates[targetFac.ID]
    for i in range(len(availFacs)):
        availFacsTtt.append((i, adjMatrix[availFacs[i][1]][targetFac.ID]))
    availFacsTtt = sorted(availFacsTtt, key=lambda x: x[1])
    maxTurn = availFacsTtt[-1][1]

    for turn in range(maxTurn):
        turnState = targetStates[turn]
        atkFacTup = [tup for tup in availFacsTtt if (tup[1] <= turn)]
        totTroops = 0
        for tup in atkFacTup:
            totTroops += availFacs[tup[0]][0]
        if (totTroops > turnState.troops):
```

```
            return turnState.troops
    return -1

def simulateEnemy(enemyFac):
    # Storing enemy movements
    actions = []

    # Simulate enemy attacks
    if (enemyFac.troops < 1):
        return actions
    curTroops = enemyFac.troops
    if (SHOW_ENEMY_ATTACKS):
        print("Simulating enemy attacks from Factory {0}|Current Troops: {1}".format(enemyFac.ID, curTroops), file=sys.stderr)

    validTargets = []
    for adj in range(NUM_FACTORIES):
        ignore = False
        targetFac = factoryInfo[adj]
        # Filters targets and add some to valid target list
        if (targetFac.owner == -1):
            ignore = True # Ignore 'own' factories
        else:
            if (targetFac.production == 0):
                ignore = True # Ignores factories that do not give production
        # Adds valid targets
        if (ignore):
            continue
        else:
            validTargets.append(targetFac)

    # Naive case: no cyborgs!
    for targetFac in validTargets:
        if (curTroops < 1):
            return actions
        if (targetFac.troops == 0 and targetFac.owner == 0):
            actions.append(MOVE([enemyFac.ID, targetFac.ID, 1]))
            curTroops -= 1

    # Weighs targets
    weightedTargets = []
    for targetFac in validTargets:
        weightedTargets.append((targetFac, scoreTarget(targetFac.ID, enemyFac.ID)))
    weightedTargets = sorted(weightedTargets, key=lambda x: x[1], reverse=True)

    # Attacks targets in weighted order
    for targetTup in weightedTargets:
        if (curTroops < 1):
            return actions
        targetFac = targetTup[0]
        if (SHOW_ENEMY_ATTACKS):
            print("Enemy attacking: {0}".format(targetFac.ID), file=sys.stderr)
        targetAttack = False
        targetTroops = 0
        # Determines how many troops to send
        if (targetFac.owner == 0):
            targetTroops = int((targetFac.troops+ENEMY_EXCESS_NEUTRAL)*ENEMY_OFFENSIVE)
            if (targetTroops <= curTroops):
                targetAttack = True
        elif (targetFac.owner == 1):
            targetTroops = int((targetFac.troops+ENEMY_EXCESS_ENEMY)*ENEMY_OFFENSIVE)
            if (targetTroops <= curTroops):
                targetAttack = True
            else:
                targetTroops = curTroops
                targetAttack = True
        # Issues attack command if available
        if (targetAttack):
            actions.append(MOVE([enemyFac.ID, targetFac.ID, targetTroops]))
            if (SHOW_ENEMY_ATTACKS):
                print(actions[-1].print(), file=sys.stderr)
            curTroops -= targetTroops
    return actions

def simulateEnemySmart(enemyFac, resolutions):
    # Storing enemy movements
    actions = []

    # Simulate enemy attacks
    if (enemyFac.troops < 1):
        return actions
    curTroops = enemyFac.troops
    if (SHOW_ENEMY_ATTACKS):
        print("Simulating enemy attacks from Factory {0}|Current Troops: {1}".format(enemyFac.ID, curTroops), file=sys.stderr)

    validTargets = []
    for adj in range(len(adjList[enemyFac.ID])):
        ignore = False
```

```
        targetFac = factoryInfo[adjList[enemyFac.ID][adj][0]]
        # Filters targets and add some to valid target list
        if (targetFac.owner == -1):
            ignore = True # Ignore 'own' factories
        else:
            if (targetFac.production == 0):
                ignore = True # Ignores factories that do not give production
        # Adds valid targets
        if (ignore):
            continue
        else:
            validTargets.append(targetFac)

    # Naive case: no cyborgs!
    for targetFac in validTargets:
        if (curTroops < 1):
            return actions
        if (targetFac.troops == 0 and targetFac.owner == 0):
            actions.append(MOVE([enemyFac.ID, targetFac.ID, 1]))
            curTroops -= 1

    # Weighs targets
    weightedTargets = []
    for targetFac in validTargets:
        weightedTargets.append((targetFac, scoreTarget(targetFac.ID, enemyFac.ID)))
    weightedTargets = sorted(weightedTargets, key=lambda x: x[1], reverse=True)

    # Attacks targets in weighted order
    for targetTup in weightedTargets:
        if (curTroops < 1):
            return actions
        targetFac = targetTup[0]
        targetStates = resolutions[targetFac.ID]
        ttt = adjMatrix[enemyFac.ID][targetFac.ID]
        tttState = targetStates[ttt]
        if (SHOW_ENEMY_ATTACKS):
            print("Enemy attacking: {0}".format(targetFac.ID), file=sys.stderr)
        targetAttack = False
        targetTroops = 0
        # Determines how many troops to send
        if (tttState.owner == 0):
            targetTroops = int((tttState.troops+ENEMY_EXCESS_NEUTRAL)*ENEMY_OFFENSIVE)
            if (targetTroops <= curTroops):
                targetAttack = True
        elif (tttState.owner == 1):
            targetTroops = int((tttState.troops+ENEMY_EXCESS_ENEMY)*ENEMY_OFFENSIVE)
            if (targetTroops <= curTroops):
                targetAttack = True
            else:
                targetTroops = curTroops
                targetAttack = True
        # Issues attack command if available
        if (targetAttack):
            actions.append(MOVE([enemyFac.ID, targetFac.ID, targetTroops]))
            if (SHOW_ENEMY_ATTACKS):
                print(actions[-1].print(), file=sys.stderr)
            curTroops -= targetTroops
    return actions

# Classes
class FactoryMsg(object):

    def __init__(self, entityID, args):
        self.ID = entityID
        self.owner = args[0]
        self.troops = args[1]
        self.production = args[2]
        self.cooldown = args[3]

    def updateOwnership(self):
        if (self.troops < 0):
            self.owner *= -1
            self.troops = abs(self.troops)

class TroopMsg(object):

    def __init__(self, entityID, args):
        self.ID = entityID
        self.owner = args[0]
        self.origin = args[1]
        self.target = args[2]
        self.size = args[3]
        self.ttt = args[4]

    def isEnemy(self):
        return (self.owner == -1)
```

```
class BombMsg(object):

    def __init__(self, entityID, args):
        self.ID = entityID
        self.owner = args[0]
        self.origin = args[1]
        self.target = args[2]
        self.ttt = args[3]

    def isEnemy(self):
        return (self.owner == -1)

class Action(object):

    def __init__(self, entityType):
        self.form = entityType
        self.origin = -1
        self.target = -1
        self.size = -1

    def isMove(self):
        return (self.form == "MOVE")

class MOVE(Action):

    def __init__(self, args):
        Action.__init__(self, "MOVE")
        self.origin = int(args[0])
        self.target = int(args[1])
        self.size = int(args[2])

    def print(self):
        return "MOVE {0} {1} {2}".format(self.origin, self.target, self.size)

class BOMB(Action):

    def __init__(self, args):
        Action.__init__(self, "BOMB")
        self.origin = int(args[0])
        self.target = int(args[1])

    def print(self):
        return "BOMB {0} {1}".format(self.origin, self.target)

class INC(Action):

    def __init__(self, args):
        Action.__init__(self, "INC")
        self.origin = int(args[0])

    def print(self):
        return "INC {0}".format(self.origin)

class FactorySimulation(object):

    def __init__(self, facID, args):
        self.ID = facID
        self.owner = int(args[0])
        self.troops = int(args[1])
        self.production = int(args[2])
        self.cooldown = int(args[3])

    def tick(self):
        if (self.cooldown > 0):
            self.cooldown -= 1
        if (self.owner != 0 and self.cooldown <= 0):
            self.troops += self.production

    def bombed(self):
        self.troops -= 0.5*self.troops if abs(0.5*self.troops) > 10 else self.troops
        self.cooldown = BOMB_PRODUCTION_COOLDOWN

    def procPacket(self, packet):
        if (packet.owner == self.owner):
            self.troops += packet.size
        else:
            self.troops -= packet.size
            if (self.troops < 0):
                self.troops = abs(self.troops)
                self.owner = packet.owner

class Factory(object):

    def __init__(self, facID):
        self.ID = facID
        self.owner = 0
        self.troops = 0
```

```
        self.production = 0
        self.cooldown = 0
        self.incomming = []
        self.outgoing = [] #TODO: not necessary? since outgoing == incomming somewhere else
        self.actions = []
        self.blacklist = [] # Blacklisted enemy targets
        self.TROOP_OFFENSIVE = TROOP_OFFENSIVE # Local threshold
        self.TROOP_DEFENSIVE = TROOP_DEFENSIVE # Local threshold

    def tick(self):
        del self.incomming[:]
        del self.outgoing[:]
        del self.actions[:]
        del self.blacklist[:]

    def update(self, args):
        self.owner = args[0]
        self.troops = args[1]
        self.production = args[2]
        self.cooldown = args[3]

    def updateBlacklist(self, argList):
        self.blacklist = argList

    def pushIncomming(self, packet):
        self.incomming.append(packet)

    def delIncomming(self, packetID):
        idList = [pack.ID for pack in self.incomming]
        if (packetID not in idList): # Error, packet not found
            return False
        else:
            del self.incomming[idList.index(packetID)]
            return True

    def reportAvailTroops(self, simulStates):
        curTroops = min(self.troops, readMaxAvailTroops(simulStates[self.ID])[0])
        return curTroops

    '''
    Simulation function
        - Takes incomming troop packets
        - Runs turn-by-turn simulation
        - Outputs an array of states for SIMUL_TURNS turns
    '''
    def resolve(self): #TODO: Huge function, simulates game till last troop packet arrives
        # Generates array to store simulation
        curState = FactorySimulation(self.ID, (self.owner, self.troops, self.production, self.cooldown))
        simulMap = []
        packetIdx = 0

        # If there's no contention, extend current state
        if (len(self.incomming) == 0):
            for turn in range(SIMUL_TURNS):
                # Ticks cooldown timer and produces troops
                if (turn > 0):
                    curState.tick()
                # Explodes bombs
                for bomb in bombInfo:
                    if (bomb.owner == 1 and bomb.target == self.ID and bomb.ttt < 1):
                        curState.bombed()
                # Stores current turn simulated result
                args = (curState.owner, curState.troops, curState.production, curState.cooldown)
                facState = FactoryMsg(self.ID, args)
                simulMap.append(facState)

        # Computes a turn-by-turn simulation upon this factory
        else:
            self.incomming = sorted(self.incomming, key=lambda x: x.ttt) # Sort by time to target

            # Simulates ownership for SIMUL_TURNS turns
            # print("Starting resolution for Factory {0}...".format(self.ID), file=sys.stderr)
            for turn in range(SIMUL_TURNS):
                # Produces Units
                if (turn > 0):
                    curState.tick()
                # Resolves Battles
                while (packetIdx < len(self.incomming) and self.incomming[packetIdx].ttt <= turn):
                    # print("Turn {0}:".format(turn), file=sys.stderr)
                    # print("Packet: Owner->{0} | Troops->{1}".format(self.incomming[packetIdx].owner, self.incomming[packetIdx].size), file=sys.st
                    # print("Current Troops in Factory: {0}".format(curState.troops), file=sys.stderr)
                    curState.procPacket(self.incomming[packetIdx])
                    # print("Resolved Troops in Factory: {0}".format(curState.troops), file=sys.stderr)
                    packetIdx += 1
                # Explodes bombs
                for bomb in bombInfo:
                    if (bomb.owner == 1 and bomb.target == self.ID and bomb.ttt < 1):
```

```
                    curState.bombed()
            # Stores current turn simulated result
            args = (curState.owner, curState.troops, curState.production, curState.cooldown)
            facState = FactoryMsg(self.ID, args)
            simulMap.append(facState)

    #DEBUG: Prints out states of each turn
    if (SHOW_RESOLUTION):
        print("====================\nResolved Factory {0}:\n====================".format(self.ID), file=sys.stderr)
        print("Initial Troop count: {0}".format(self.troops), file=sys.stderr)
        for i in range(5):
            print("Step {0}: Owner->{1} | Troops->{2} | Production->{3} | Cooldown->{4}".format(i, simulMap[i].owner, simulMap[i].troops, simul
        print("Max Available Units: {0}".format(readMaxAvailTroops(simulMap)[0]), file=sys.stderr)
        print("Turns before being overrun: {0}".format(readMaxAvailTroops(simulMap)[1]), file=sys.stderr)
    return simulMap # Outputs list of simulated factory state tuples

'''
Reinforce function
    - Takes a list of targets to be reinforced
    - Weighs targets
    - Sends reinforcements if available
'''
def reinforce(self, simulStates): #TODO: How to bring troops to the front?
    curTroops = min(self.troops, readMaxAvailTroops(simulStates[self.ID])[0])
    print("Factory {0} Reinforcing...|Current Troops: {1}".format(self.ID, curTroops), file=sys.stderr)
    if (curTroops < 1):
        return self.actions

    # Get connected friendly reinforcible factories
    adjMyFactories = [facTup[0] for facTup in adjList[self.ID] if (factoryInfo[facTup[0]].owner == 1)]

    # Weighs targets
    weightedTargets = []
    for target in adjMyFactories:
        weightedTargets.append((target, scoreBomb(target, self.ID)))
    weightedTargets = sorted(weightedTargets, key=lambda x: x[1], reverse=True)

    # Reinforces targets in weighted order
    for targetTup in weightedTargets:
        if (curTroops < 1):
            self.troops = curTroops
            return self.actions
        target = targetTup[0]
        ttt = floydWarMatrix[self.ID][target]+1
        requestTroops = needed_reinforcements(ttt, curTroops, simulStates[target])
        if (requestTroops < 0):
            continue
        self.actions.append(MOVE([self.ID, target, requestTroops]))
        print(self.actions[-1].print(), file=sys.stderr)
        curTroops -= requestTroops

    self.troops = curTroops
    return self.actions

'''
Attack function
    - Prioritizes Upgrade if base will not get overrun in the future
    - Gets a list of valid targets (enemy + neutrals) not in blacklist
    - Weigh valid targets
    - Issues attack commands by priority of weight
'''
def attack(self, simulStates): #TODO: Where to upgrade factories??
    if (self.production == 0 and self.ID != INITIAL_FACTORY):
        self.TROOP_OFFENSIVE = 1
        self.TROOP_DEFENSIVE = 1
    else:
        self.TROOP_OFFENSIVE = TROOP_OFFENSIVE
        self.TROOP_DEFENSIVE = TROOP_DEFENSIVE
    curTroops = min(self.troops, readMaxAvailTroops(simulStates[self.ID])[0])
    print("Factory {0}|Current Troops: {1}".format(self.ID, curTroops), file=sys.stderr)
    if (curTroops < 1):
        self.troops = curTroops
        return self.actions

    # Get a list of valid targets to attack
    validTargets = getValidTargets(self.ID, self.blacklist, simulStates)
    #TODO: We add bombed targets into valid target list
    # for bomb in bombInfo:
    #     if (bomb.owner == 1):
    #         print("Bomb from {0}->{1} arrives in: {2}".format(bomb.origin, bomb.target, bomb.ttt), file=sys.stderr)
    #         if (factoryInfo[bomb.target] not in validTargets):
    #             validTargets.append(factoryInfo[bomb.target])

    # Naive case: no cyborgs!
    for targetFac in validTargets:
        print("Testing empty factory: Factory {0}".format(targetFac.ID), file=sys.stderr)
        if (curTroops < 1):
```

```
            self.troops = curTroops
            return self.actions
        ttt = floydWarMatrix[self.ID][targetFac.ID]+1
        targetState = simulStates[targetFac.ID][ttt]
        if (targetState.troops == 0 and targetState.owner == 0):
            print("Removing factory from target list", file=sys.stderr)
            self.actions.append(MOVE([self.ID, targetFac.ID, 1]))
            print(self.actions[-1].print(), file=sys.stderr)
            curTroops -= 1
            targetFac = None #TODO: Removes target from list?
    validTargets = [fac for fac in validTargets if fac != None] # Removes None-types

    #TODO: Ad-Hoc upgrades (temporary)
    upgradeFactory = True
    curProduction = self.production
    upgrades = 0
    # Decides suitability for upgrading
    while (upgradeFactory):
        # Safety check for troops available
        if (curTroops < FACTORY_UPGRADE_COST):
            upgradeFactory = False
        # Disables upgrades if nearby neutral exists with production
            # On the condition that one can take it
            # And such a move would bring about more overall units than upgrading
        neutralFactories = []
        for targetFac in validTargets:
            targetStates = simulStates[targetFac.ID]
            ttt = floydWarMatrix[self.ID][targetFac.ID]+1
            tttState = targetStates[ttt]
            if (tttState.owner == 0 and tttState.production > 0):
                tttDiff = FACTORY_UPGRADE_COST - ttt
                if (tttDiff < 0 or tttState.production*tttDiff > FACTORY_UPGRADE_COST):
                    neutralFactories.append(targetFac.ID)
        if (len(neutralFactories) > 0):
            upgradeFactory = False
        # Checks conditions for an upgrade
        if (curProduction == 3 or not viable_upgrade(simulStates[self.ID], upgrades)):
            upgradeFactory = False
        # Upgrades Current Factory
        if (upgradeFactory):
            upgrades += 1
            curProduction += 1
            print("Upgrading factory with {0} troops at level {1} production".format(curTroops, factoryInfo[self.ID].production), file=sys.stde
            self.actions.append(INC([self.ID]))
            curTroops -= FACTORY_UPGRADE_COST

    # Weighs targets
    weightedTargets = []
    for targetFac in validTargets:
        weightedTargets.append((targetFac, scoreTarget(targetFac.ID, self.ID)))
    weightedTargets = sorted(weightedTargets, key=lambda x: x[1], reverse=True)

    # Prioritizes targets that can be overwhelmed
    overwhelmTargets = []
    ignoreTargets = []
    # Classifies targets
    for targetTup in weightedTargets:
        if (curTroops < 1):
            self.troops = curTroops
            return self.actions
        targetFac = targetTup[0]
        targetStates = simulStates[targetFac.ID]
        ttt = floydWarMatrix[self.ID][targetFac.ID]+1
        tttState = targetStates[ttt]
        print("Testing Priority attack: {0} | ttt: {1} | tttState: Owner->{2} Troops->{3}".format(targetFac.ID, ttt, tttState.owner, tttState.t

        # Checks that troops won't arrive before bomb :O
        ignore = False
        for bomb in bombInfo:
            if (bomb.owner == 1):
                print("Bomb from {0}->{1} arrives in: {2} | Attack arrives in: {3}".format(bomb.origin, bomb.target, bomb.ttt, ttt), file=sys.s
                if (bomb.target == targetFac.ID and ttt <= bomb.ttt):
                    ignore = True
                    break
        if (ignore):
            ignoreTargets.append(targetFac)
            continue

        targetAttack = False
        targetTroops = 0
        if (tttState.owner == 0): # Neutral Target
            targetTroops = tttState.troops+TROOP_EXCESS_NEUTRAL
            if (targetTroops <= curTroops): # Can overwhelm target
                targetAttack = True
        else: # Enemy Target
            targetTroops = int((tttState.troops+TROOP_EXCESS_ENEMY)*TROOP_OFFENSIVE_MULTIPLIER)+1
            if (targetTroops <= curTroops): # Can overwhelm target
```

```
                targetAttack = True

        # Adds target to priority list if can be overwhelmed
        if (targetAttack):
            overwhelmTargets.append(targetFac)
            self.actions.append(MOVE([self.ID, targetFac.ID, targetTroops]))
            print("Overwhelming target {0}".format(targetFac.ID), file=sys.stderr)
            print(self.actions[-1].print(), file=sys.stderr)
            curTroops -= targetTroops

    # Attacks targets in weighted order
    for targetTup in weightedTargets:
        if (curTroops < 1):
            self.troops = curTroops
            return self.actions
        targetFac = targetTup[0]
        # Filters targets on ignore list or priority list
        if (targetFac in ignoreTargets or targetFac in overwhelmTargets):
            continue
        targetStates = simulStates[targetFac.ID]
        ttt = floydWarMatrix[self.ID][targetFac.ID]+1
        tttState = targetStates[ttt]
        print("Attacking: {0} | ttt: {1} | tttState: Owner->{2} Troops->{3}".format(targetFac.ID, ttt, tttState.owner, tttState.troops), file=s
        # Determines how many troops to send
        targetAttack = False
        targetTroops = 0
        if (tttState.owner == 0): # Neutral Target
            targetTroops = tttState.troops+TROOP_EXCESS_NEUTRAL
            if (targetTroops <= curTroops): # Can overwhelm target
                print("Overwhelming...", file=sys.stderr)
                targetAttack = True
            elif (targetTroops <= curTroops+self.production): # Able to target next turn
                print("Suspend attacks", file=sys.stderr)
                self.troops = curTroops
                return self.actions
            else: # Unable to overwhelm target immediately
                targetTroops = int(self.TROOP_OFFENSIVE*curTroops)
                print("Cannot overwhelm, sending {0} troops".format(targetTroops), file=sys.stderr)
                targetAttack = True
        elif (tttState.owner == -1): # Enemy Target
            # We only attack when our attack can overwhelm enemy
            targetTroops = int((tttState.troops+TROOP_EXCESS_ENEMY)*TROOP_OFFENSIVE_MULTIPLIER)+1
            if (targetTroops <= curTroops):
                print("ENEMY! Sending {0} troops".format(targetTroops), file=sys.stderr)
                targetAttack = True
        # Issues attack command if available
        if (targetAttack):
            self.actions.append(MOVE([self.ID, targetFac.ID, targetTroops]))
            print(self.actions[-1].print(), file=sys.stderr)
            curTroops -= targetTroops
    self.troops = curTroops
    return self.actions

'''
Upgrade function
    - Sends troops to nearby factories to facilitate their upgrading
'''
def upgrade(self, simulStates):
    curTroops = min(self.troops, readMaxAvailTroops(simulStates[self.ID])[0])
    print("Factory {0} Sending troops for Upgrading...|Current Troops: {1}".format(self.ID, curTroops), file=sys.stderr)
    if (curTroops < 1):
        self.troops = curTroops
        return self.actions
    # Scans for nearby factories
    for adj in adjList[self.ID]:
        if (curTroops < 1):
            self.troops = curTroops
            return self.actions
        adjFac = factoryInfo[adj[0]]
        requestTroops = min(curTroops, needed_upgradeTroops(self, adjFac, simulStates))
        if (requestTroops > 0 and requestTroops <= curTroops):
            self.actions.append(MOVE([self.ID, adjFac.ID, requestTroops]))
            print(self.actions[-1].print(), file=sys.stderr)
            curTroops -= requestTroops
    self.troops = curTroops
    return self.actions

'''
Redistribution function
    - Scans for nearby friendly factories closer than self to enemy
    - Sends excess troops proportionally to those factories
'''
def redistribute(self, simulStates):
    curTroops = min(self.troops, readMaxAvailTroops(simulStates[self.ID])[0])
    print("Factory {0} Redistributing...|Current Troops: {1}".format(self.ID, curTroops), file=sys.stderr)
    if (curTroops < 1):
        return self.actions
```

```
        #TODO: If have excess troops, send them off proportionally to nearby friendly factories?
        # Get connected friendly reinforcible factories
        adjMyFactories = [facTup[0] for facTup in adjList[self.ID] if (factoryInfo[facTup[0]].owner == 1)]
        myDistToEnemy = closestEnemy(factoryInfo[self.ID])[1]

        # Get list of 'frontline' friendly factories
        adjFrontlineFactories = [facID for facID in adjMyFactories if (len([enID for enID in range(NUM_FACTORIES) if factoryInfo[enID].owner == -1]
        weightedFrontlineFactories = []
        for facID in adjFrontlineFactories:
            weightedFrontlineFactories.append((facID, scoreRedistribution(facID, self.ID, closestEnemy(factoryInfo[facID])[1])))
        weightedFrontlineFactories = sorted(weightedFrontlineFactories, key=lambda x: x[1], reverse=True)

        # Sends available troops based on score
        totScore = 0
        minScore = MAX_INT
        scoreList = [scoreTup[1] for scoreTup in weightedFrontlineFactories]
        limTroops = 0 if self.production == 3 else FACTORY_UPGRADE_COST
        totTroops = max(0, curTroops - limTroops)
        for score in scoreList: # Get min score
            if (score < minScore):
                minScore = score
        for score in scoreList: # Transform range of scoreList to [0, INF)
            score -= minScore
            totScore += score
        for scoreTup in weightedFrontlineFactories:
            if (curTroops <= limTroops):
                self.troops = curTroops
                return self.actions
            normScore = scoreTup[1] - minScore
            weightedTroops = max(curTroops,int((normScore/max(1,totScore))*totTroops))
            if (weightedTroops <= curTroops):
                self.actions.append(MOVE([self.ID, scoreTup[0], weightedTroops]))
                print(self.actions[-1].print(), file=sys.stderr)
                curTroops -= weightedTroops

        # Just send whatever amts of troops off to the highest-weighted factory
        if (curTroops > 0 and len(weightedFrontlineFactories) > 0):
            self.actions.append(MOVE([self.ID, weightedFrontlineFactories[0][0], curTroops]))
            print(self.actions[-1].print(), file=sys.stderr)
            curTroops -= curTroops

        self.troops = curTroops
        return self.actions

class Strategizer(object):

    def __init__(self, resolutions, simulation, bombs, incs, simulIDCounter):
        self.resolutions = resolutions # 2D list of FactoryMsg objects
        self.actions = []
        self.evalActions = []
        self.blacklistedEnemies = [] # Enemies we can overpower
        self.simulation = simulation
        self.bombs = bombs
        self.incs = incs
        self.simulIDCounter = simulIDCounter

    '''
    Simulated pushing of troop packets to targeted factories
        - Takes some list of actions and 'executes' them
        - Results in troop packets pushed to incomming queue for targeted factories
    '''
    def simulate(self, actions):
        print("Simulating: ", file=sys.stderr)
        for move in actions:
            if (move.isMove()):
                if (move.size < 1): # Prunes off no troop packets
                    continue
                args = [1, move.origin, move.target, move.size, adjMatrix[move.origin][move.target]]
                curPacket = TroopMsg(self.simulIDCounter, args)
                self.simulIDCounter += 1
                self.simulation[move.target].pushIncomming(curPacket)
                print("Pushed: "+move.print()+" to Factory: {0} | CurPackets: {1}".format(move.target, len(self.simulation[move.target].incomming))

    '''
    Conservative troop processing strategy
        - Block attacks to resolved targets
        - Orders priority of troop movement as such:
            1) Reinforces nearby factories
            2) Attack nearby factories
            3) Sends troops to upgrade nearby factories
            4) Redistributes remaining troops to nearby factories
    '''
    def execute(self):
        # Prune off excess attacks
        myFactories = [self.simulation[facID] for facID in range(NUM_FACTORIES) if (self.simulation[facID].owner == 1)] # Own factories
        for i in range(len(self.resolutions)):
```

```
        simulState = self.resolutions[i]
        if (simulState[-1].owner == 1 and self.simulation[i].owner != 1):
            print("Battle for {0} resolved in our favor, preventing further troops".format(i), file=sys.stderr)
            # Add target to blacklist
            self.blacklistedEnemies.append(i)

    # 1) Sends reinforcements
    for fac in myFactories:
        print("Factory {0} reinforcing...".format(fac.ID), file=sys.stderr)
        fac.reinforce(self.resolutions)

    # 2) Re-evaluates attack options
    for fac in myFactories:
        print("====================\nFactory {0} attacking...\n====================".format(fac.ID), file=sys.stderr)
        fac.updateBlacklist(self.blacklistedEnemies)
        fac.attack(self.resolutions)

    # 3) Runs troop movements for upgrades
    upgradeFactories = [self.simulation[facID] for facID in range(NUM_FACTORIES) if should_reinforce(facID)]
    for fac in upgradeFactories:
        fac.upgrade(self.resolutions)

    # 4) Redistributes excess troops
    for fac in myFactories:
        fac.redistribute(self.resolutions)

'''
Redirects troops along floyd-warshall path instead of naive direct pathing
'''
def redirect(self): #TODO: run floyd-warshall per turn?
    # Runs simulation for redirecting
    myFactories = [self.simulation[facID] for facID in range(NUM_FACTORIES) if (self.simulation[facID].owner == 1)] # Own factories
    for fac in myFactories:
        self.evalActions.extend(fac.actions)
    self.simulate(self.evalActions)

    # Re-paths troop packets
    for fac in self.simulation:
        print("Redirecting for target Factory {0} | Packets: {1}".format(fac.ID, len(fac.incomming)), file=sys.stderr)
        delList = []
        for troop in fac.incomming:
            closestIntermediate = floydWarPath[troop.origin][fac.ID][0]
            ttt = floydWarMatrix[troop.origin][closestIntermediate]
            closestIntermediateOwner = self.resolutions[closestIntermediate][ttt].owner
            # print("Attempting Redirect:\nTroop destination: {0}\nOrigin: {1}\nIntermediate: {2}".format(fac.ID, troop.origin, closestIntermed
            if (closestIntermediate != fac.ID):
                #TODO: Do not route through non-owned factories if they have troops
                if (closestIntermediateOwner != 1 and self.resolutions[closestIntermediate][ttt].troops > 0):
                    continue
                troop.target = closestIntermediate
                troop.ttt = ttt
                self.simulation[closestIntermediate].pushIncomming(troop)
                print("Redirection troop: {0}-->{1} from initial target {2}".format(troop.origin, closestIntermediate, fac.ID), file=sys.stderr
                delList.append(troop.ID)
        if (len(delList) > 0):
            for i in range(len(delList)):
                fac.delIncomming(delList[i])

def prune(self): #TODO: prunes excess troops sent and orgnize co-ordinated attacks
    myFactories = [self.simulation[facID] for facID in range(NUM_FACTORIES) if (self.simulation[facID].owner == 1)] # Own factories
    coordinatedActions = []

    # Get list of factories still in action for co-ordinated attacks
    facTroopList = []
    for fac in myFactories:
        curTroops = fac.reportAvailTroops(self.resolutions)
        if (curTroops > 0):
            facTroopList.append([curTroops, fac.ID])

    # Get list of global targets
    validTargets = []
    for facTup in facTroopList:
        tmpTargetList = getValidTargets(facTup[1], [], self.resolutions)
        for target in tmpTargetList:
            if (target not in validTargets):
                validTargets.append(target)

    # Weigh targets
    weightedTargets = []
    for targetFac in validTargets:
        attackFacID = closestFriendly(targetFac)[0]
        if (attackFacID == -1):
            continue
        weightedTargets.append((targetFac, scoreTarget(targetFac.ID, attackFacID)))
    weightedTargets = sorted(weightedTargets, key=lambda x: x[1], reverse=True)

    # Co-ordinate attacks from multiple factories
```

```
        for targetTup in weightedTargets:
            target = targetTup[0]
            print("Coordinating attack to Factory {0}: ".format(target.ID), file=sys.stderr)
            adjFacsID = [adjTup[0] for adjTup in adjList[target.ID]]
            availFacs = [facTup for facTup in facTroopList if (facTup[1] in adjFacsID)]
            # Simulates attack
            requestTroops = canOverwhelm(availFacs, target, self.resolutions)
            if (requestTroops > 0): # We can attack
                shortestTtt = MAX_INT
                for availTup in availFacs:
                    curID = availTup[1]
                    if (adjMatrix[curID][target.ID] < shortestTtt):
                        shortestTtt = adjMatrix[curID][target.ID]
                # Get factories attacking this turn
                attackingFacs = []
                for availTup in availFacs:
                    curID = availTup[1]
                    if (adjMatrix[curID][target.ID] == shortestTtt):
                        attackingFacs.append(availTup)
                # Attack with each factory
                for atkTup in attackingFacs:
                    curTroops = atkTup[0]
                    curID = atkTup[1]
                    targetTroops = min(curTroops, requestTroops)
                    if (targetTroops <= curTroops):
                        requestTroops -= curTroops
                        atkTup[0] -= curTroops
                        coordinatedActions.append(MOVE([curID, target.ID, targetTroops]))
                        print(coordinatedActions[-1].print(), file=sys.stderr)

        self.simulate(coordinatedActions)

    def whack(self): #TODO: Whacks enemy with all we've got
        myFactories = [self.simulation[facID] for facID in range(NUM_FACTORIES) if (self.simulation[facID].owner == 1)] # Own factories
        whackActions = []

        # Whack enemy with factories still in action
        for fac in myFactories:
            curTroops = fac.reportAvailTroops(self.resolutions)
            print("Whacking with factory {0} | Troops: {1}".format(fac.ID, curTroops), file=sys.stderr)
            if (curTroops > 0):
                targetList = getValidTargets(fac.ID, [], self.resolutions)
                if (len(targetList) < 1):
                    continue
                # Weigh targets
                weightedTargets = []
                for targetFac in targetList:
                    weightedTargets.append((targetFac, scoreTarget(targetFac.ID, fac.ID)))
                weightedTargets = sorted(weightedTargets, key=lambda x: x[1], reverse=True)
                # Whack the first one :D
                whackActions.append(MOVE([fac.ID, weightedTargets[0][0].ID, curTroops]))
                print(whackActions[-1].print(), file=sys.stderr)
                fac.troops -= curTroops

        self.simulate(whackActions)

    def print(self):
        # Adds movement commands
        for fac in self.simulation:
            for troop in fac.incomming:
                self.actions.append(MOVE([troop.origin,fac.ID,troop.size]).print())
        # Adds bomb commands
        for bomb in self.bombs:
            self.actions.append(bomb.print())
        # Adds upgrade commands
        for action in self.evalActions:
            if (action.form == "INC"):
                self.actions.append(action.print())
        for inc in self.incs:
            self.actions.append(inc.print())
        # Adds in debuggin message
        if (MSG_OUTPUT):
            self.actions.append("MSG {0}".format(MSG_CONTENT))
        # Outputs current turn's actions
        if (len(self.actions) < 1):
            print("WAIT")
        else:
            outputCommand = ""
            for cmd in self.actions:
                outputCommand += ";"
                outputCommand += cmd
            print(outputCommand[1:])

# Handle Inputs
NUM_FACTORIES = int(input())  # Number of factories
for i in range(NUM_FACTORIES): # Initialize Factories
    adjList.append([])
```

```
    adjMatrix.append([0 for x in range(NUM_FACTORIES)])
    floydWarMatrix.append([MAX_INT for x in range(NUM_FACTORIES)]) # Matrix to store shortest distances
    floydWarPath.append([[-1] for x in range(NUM_FACTORIES)]) # Matrix to store path
    floydWarNext.append([-1 for x in range(NUM_FACTORIES)]) # Optimized matrix storing only next target
    factoryInfo.append(Factory(i))
    simulFac.append(Factory(i))
link_count = int(input())  # Number of links between factories
for i in range(link_count): # Initialize adjList/adjMatrix
    factory_1, factory_2, distance = [int(j) for j in input().split()]
    adjList[factory_1].append((factory_2, distance))
    adjList[factory_2].append((factory_1, distance))
    adjMatrix[factory_1][factory_2] = distance
    adjMatrix[factory_2][factory_1] = distance
    # Stores links into floyd-warshall graph
    #TODO: do not store if distance > 5?
    floydWarMatrix[factory_1][factory_2] = distance
    floydWarMatrix[factory_2][factory_1] = distance
    floydWarPath[factory_1][factory_2] = [factory_2]
    floydWarPath[factory_2][factory_1] = [factory_1]
    floydWarNext[factory_1][factory_2] = factory_2
    floydWarNext[factory_2][factory_1] = factory_1
for i in range(NUM_FACTORIES): # Filter out paths > MAX_LINK_DISTANCE whilst preserving at least 1 link
    minLinkDistance = MAX_INT
    minLinkTarget = -1
    numLinks = 0
    for j in range(len(floydWarMatrix[i])):
        if (floydWarMatrix[i][j] < minLinkDistance):
            minLinkDistance = floydWarMatrix[i][j]
            minLinkTarget = j
        if (floydWarMatrix[i][j] > MAX_LINK_DISTANCE):
            floydWarMatrix[i][j] = MAX_INT
            floydWarPath[i][j] = [-1]
            floydWarNext[i][j] = -1
        else:
            numLinks += 1
    if (numLinks < 1): # Establish shortest link
        floydWarMatrix[i][minLinkTarget] = minLinkDistance
        floydWarPath[i][minLinkTarget] = [minLinkTarget]
        floydWarNext[i][minLinkTarget] = minLinkTarget
    minLinkDistance = MAX_INT
    minLinkTarget = -1
    numLinks = 0
    for j in range(len(adjList[i])):
        if (adjList[i][j][1] < minLinkDistance):
            minLinkDistance = adjList[i][j][1]
            minLinkTarget = adjList[i][j][0]
        if (adjList[i][j][1] > MAX_LINK_DISTANCE):
            adjList[i][j] = None
        else:
            numLinks += 1
    if (numLinks < 1):
        adjList[i][minLinkTarget] = (minLinkTarget, minLinkDistance)
    adjList[i] = [adjList[i][idx] for idx in range(len(adjList[i])) if adjList[i][idx] is not None]
for i in range(NUM_FACTORIES): # Sort adjlist by order of increasing distance
    adjList[i] = sorted(adjList[i], key=lambda x: x[1])

# Floyd-Warshall to compute All-Pair Shortest-Paths
for k in range(NUM_FACTORIES):
    for i in range(NUM_FACTORIES):
        for j in range(NUM_FACTORIES):
            if (i==j or k==j):
                continue
            intermediate = floydWarMatrix[i][k] + floydWarMatrix[k][j]
            if (intermediate < floydWarMatrix[i][j]):
                newPath = [k]
                newPath.extend(floydWarPath[k][j])
                floydWarPath[i][j] = newPath
                floydWarNext[i][j] = floydWarNext[k][j]
                floydWarMatrix[i][j] = intermediate

# Game Loop
while True:
    del troopInfo[:] # Resets turn variables
    del bombInfo[:]
    del turnMoves[:]
    del turnBombs[:]
    del turnIncs[:]
    CYBORGS_OWN = 0
    CYBORGS_ENEMY = 0
    myFactories = []
    simulIDCounter = 0
    for i in range(NUM_FACTORIES): # Ticks each factory
        factoryInfo[i].tick()
        simulFac[i].tick()

    # Reads game turn state
    entity_count = int(input())  # the number of entities (e.g. factories and troops)
```

```
for i in range(entity_count):
    entity_id, entity_type, arg_1, arg_2, arg_3, arg_4, arg_5 = input().split()
    entity_id = int(entity_id)
    args = [int(arg_1), int(arg_2), int(arg_3), int(arg_4), int(arg_5)]
    if (entity_type == "FACTORY"):
        factoryInfo[entity_id].update(args)
        simulFac[entity_id].update(args)
        if (factoryInfo[entity_id].owner == 1):
            myFactories.append(entity_id)
            CYBORGS_OWN += factoryInfo[entity_id].troops
        elif (factoryInfo[entity_id].owner == -1):
            CYBORGS_ENEMY += factoryInfo[entity_id].troops
    elif (entity_type == "TROOP"):
        curPacket = TroopMsg(entity_id, args)
        factoryInfo[curPacket.target].pushIncomming(curPacket)
        troopInfo.append(curPacket)
        if (curPacket.owner == 1):
            CYBORGS_OWN += curPacket.size
        elif (curPacket.owner == -1):
            CYBORGS_ENEMY += curPacket.size
    elif (entity_type == "BOMB"):
        curPacket = BombMsg(entity_id, args)
        bombInfo.append(curPacket)

# Resets for frontline factory searching
if (FRONTLINE_FACTORY != -1 and factoryInfo[FRONTLINE_FACTORY].owner != 1):
    FRONTLINE_DISTANCE = MAX_INT
    FRONTLINE_FACTORY = -1

# Searches for enemy's initial location
if (INITIAL_FACTORY == -1 or INITIAL_FACTORY_ENEMY == -1):
    for i in range(len(factoryInfo)):
        curFac = factoryInfo[i]
        if (curFac.owner == 1 and INITIAL_FACTORY == -1):
            INITIAL_FACTORY = curFac.ID
        if (curFac.owner == -1 and INITIAL_FACTORY_ENEMY == -1):
            INITIAL_FACTORY_ENEMY = curFac.ID
        if (curFac.owner == 1): # Determine a 'frontline' factory
            # Find shortest distance to enemy
            nearestFactory = -1
            nearestDistance = MAX_INT
            for facID in range(NUM_FACTORIES):
                if (adjMatrix[curFac.ID][facID] < nearestDistance and factoryInfo[facID].owner == -1):
                    nearestDistance = adjMatrix[curFac.ID][facID]
                    nearestFactory = facID
            if (nearestFactory != -1 and nearestDistance < FRONTLINE_DISTANCE):
                FRONTLINE_DISTANCE = nearestDistance
                FRONTLINE_FACTORY = curFac.ID
                print("Determined FRONTLINE factory: {0}".format(FRONTLINE_FACTORY), file=sys.stderr)

# Launch BOMBS!
if (num_bombs > 0 and FRONTLINE_FACTORY != -1):
    print("Attempting BOMB", file=sys.stderr)
    # Scores all enemy factores for bombing! :D
    bombTargets = [(fac.ID, scoreBomb(fac.ID, FRONTLINE_FACTORY)) for fac in factoryInfo if (should_bomb(fac.ID))]
    bombTargets = sorted(bombTargets, key=lambda x: x[1], reverse=True)
    for targetTup in bombTargets:
        target = targetTup[0]
        score = targetTup[1]
        if (num_bombs < 1 or len(myFactories) < 1):
            break
        launch = True
        # Only bomb targets above threshold score
        if (score < BOMB_SCORE_THRESHOLD):
            continue
        # Do not bomb same target twice
        for bomb in bombInfo:
            if (bomb.owner == 1 and bomb.target == target):
                launch = False
                break
        if (not launch):
            continue
        # Find the closest base to launch bomb from
        nearestFactory = myFactories[0]
        nearestDistance = MAX_INT
        for facID in range(NUM_FACTORIES):
            if (facID not in myFactories):
                continue
            if (adjMatrix[facID][target] < nearestDistance):
                nearestDistance = adjMatrix[facID][target]
                nearestFactory = facID
        turnMoves.append(BOMB([nearestFactory, target]))
        num_bombs -= 1

# Constructs simulated scenario to feed into strategizer
for move in turnMoves:
    print(move.print(), file=sys.stderr)
```

```
    if (move.isMove()):
        args = [1, move.origin, move.target, move.size, adjMatrix[move.origin][move.target]]
        curPacket = TroopMsg(simulIDCounter, args)
        simulIDCounter += 1
        simulFac[move.target].pushIncomming(curPacket)
    else:
        if (move.form == "BOMB"):
            turnBombs.append(move)
        elif (move.form == "INC"):
            turnIncs.append(move)

#TODO: Simulates Nearby enemies' attacks upon self
if (SIMULATE_ENEMY):
    enemies = [fac for fac in factoryInfo if fac.owner == -1]
    enemyActions = [] # Storing enemy movements
    currentSituation = [fac.resolve() for fac in factoryInfo]
    for enemyFac in enemies:
        # Simulates enemy attacks
        enemyActions.extend(simulateEnemySmart(enemyFac, currentSituation))
        # enemyActions.extend(simulateEnemy(enemyFac))
    # Enemy has made some moves, add to simulation
    if (len(enemyActions) > 0):
        for action in enemyActions:
            if (action.isMove()):
                args = (-1, action.origin, action.target, action.size, adjMatrix[action.origin][action.target])
                enemyPacket = TroopMsg(simulIDCounter, args)
                simulIDCounter += 1
                factoryInfo[enemyPacket.target].pushIncomming(enemyPacket)

# Feed Strategizer
strategize = Strategizer([fac.resolve() for fac in factoryInfo], simulFac, turnBombs, turnIncs, simulIDCounter)

# Strategize!
strategize.execute() # Executes strategy for turn
strategize.redirect() # Redirects troops and paths them via floyd-warshall
# strategize.prune() # Prunes and organizes co-ordinated attacks
# strategize.whack() # Sends all remaining troops to attack enemy

# Fun little MSG
if (MSG_OUTPUT):
    random.seed()
    if (MSG_RANDIDX == -1):
        MSG_RANDIDX = random.randint(0,len(HITCHHIKER_GALAXY_QUOTES)-1)
    elif (MSG_START >= MSG_END):
        MSG_RANDIDX = random.randint(0,len(HITCHHIKER_GALAXY_QUOTES)-1)
        MSG_START = 0
        MSG_DELAY = MSG_MAXLEN
        MSG_END = MSG_MAXLEN
    delay = ">"
    if (MSG_DELAY > 0):
        for i in range(MSG_DELAY):
            delay += " "
        MSG_DELAY -= MSG_RATE
    if (MSG_DELAY <= 0):
        MSG_START += MSG_RATE
        if (MSG_END < len(HITCHHIKER_GALAXY_QUOTES[MSG_RANDIDX])):
            MSG_END += min(len(HITCHHIKER_GALAXY_QUOTES[MSG_RANDIDX])-MSG_END, MSG_RATE)
    MSG_CONTENT = delay+HITCHHIKER_GALAXY_QUOTES[MSG_RANDIDX][min(len(HITCHHIKER_GALAXY_QUOTES[MSG_RANDIDX]), MSG_START):min(len(HITCHHIKER_GAL

# Output final strategy for the turn
strategize.print()
```