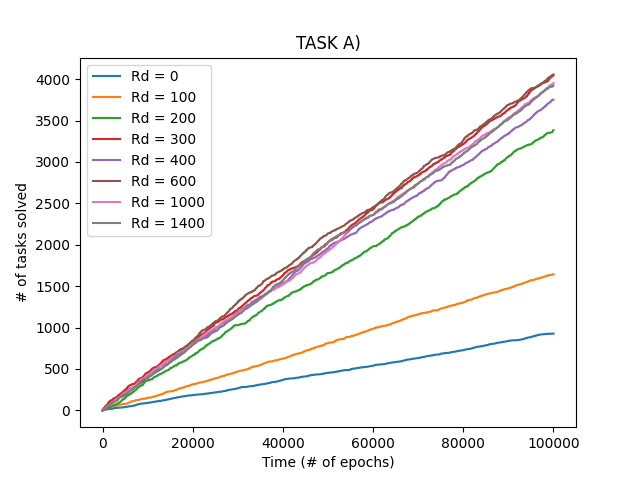
**Task A**

The auction was implemented as follows:

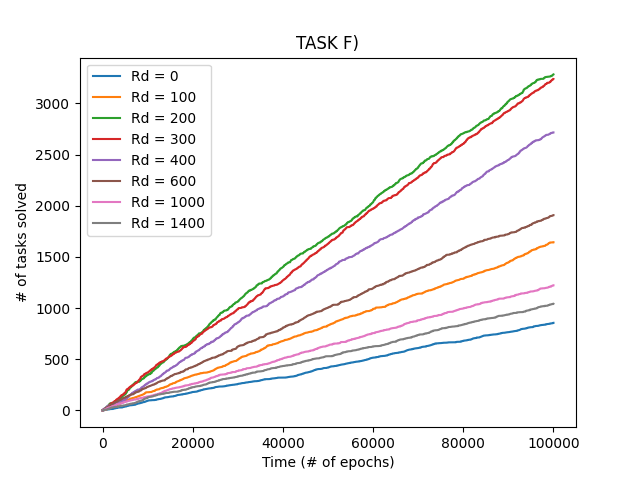
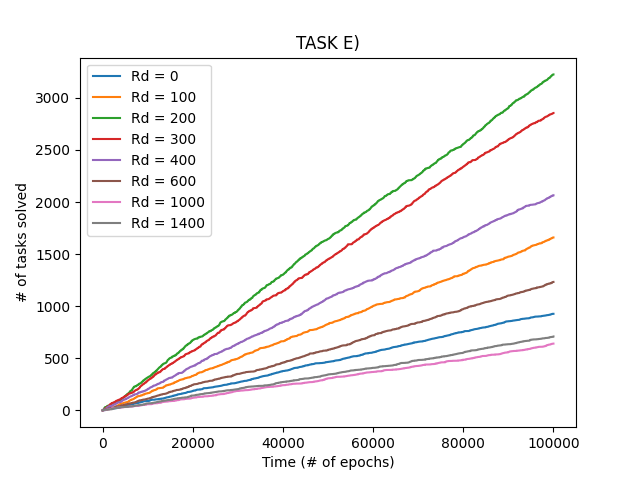
1. When the Task object detects one or more Agent objects in its task radius, and the task isn’t completed in that timestep, it tells the first detected agent to perform an auction, where the number of winners (N) in the auction is the difference between the task capacity and the number of agents detected (it could happen that two agents randomly discover the same task at the same timestep, in which case I chose to implement the algorithm so that only one of them holds an auction).
2. The agent, now in the role of auctioneer, goes through all other agents (now bidders), and calculates the distance between them and the task position. This distance is used as their bid, where a lower distance is valued higher.
3. The auctioneer saves the top N bidders, and after the bidding is done (all distances have been calculated), the target position of these N bidders is set to the task position, meaning they will move towards the task’s position in future timesteps until told otherwise.

Due to how the auction was implemented, this process is actually repeated at each timestep where a task detects agents inside of its radius. However, the winners of the auction should always be the same ones, making it equivalent to having only one auction. This implementation also ensures that if there aren’t enough bidders on the first auction to complete the task, others may come and bid later. The auction can be classified as a first-price, sealed-bid, one-shot auction.

The following graph resulted from testing the auction algorithm for 100 000 epochs:



**Task B**



When compared to tasks e) and f) from the first assignment, we can notice two main differences:

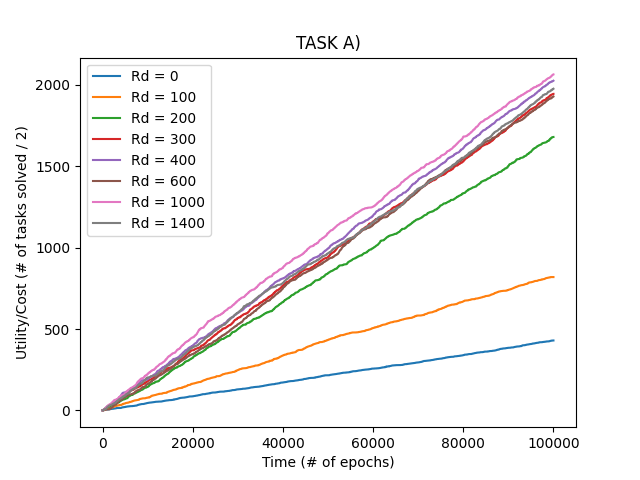
1. The number of tasks performed per epoch is significantly higher. The top performing communication distances for callout and calloff achieved in the range of 3000 solved tasks over 100 000 epochs; the auction achieved in the range of 4000, and for a wider range of communication distances.
2. For the auction algorithm, a higher communication distance seems to always be better than, or equal to, a lower one. It also seems that the increase in benefit of a higher communication distance converges towards 0 as the distance approaches one so large that the agents can communicate across the entire grid (Rd=1400); it is very hard to discern the difference in performance between all communication distances above Rd=200 from the graph.

The explanation for 1) is likely to lie in that auctions do a better job of searching and task allocation than callout and calloff does; instead of calling every agent in hearing range to come help with a task, the agent now selects only as many as needed, leaving the other agents free to explore more of the environment.

This also explains 2), because increasing communication distance only becomes a problem when it makes too many agents converge towards one task, leading to over-exploitation. This is what we observed with callout and calloff in the first assignment. Thus, we can see that through adding interactivity to our agents through auctions, a more sophisticated behavior emerges than what was achieved through reactivity alone with callout and calloff.

**Task C**

If we consider utility to be the number of tasks solved per time, we can model utility over cost as the number of tasks solved divided by 2 for interactive agents, and divided by 1 for reactive agents (meaning no change from before). Then, the graph becomes as follows:



Now we can see that relative to cost, the reactive agents clearly outperform the interactive agents (about 2000 vs. about 3000 completed tasks per cost for reactive vs. interactive agents). The graphs for reactive agents remain the same as before.

**Code Appendix**

**main.py**

from utils import \*

from constants import \*

from agent import \*

from task import \*

import numpy as np

import matplotlib.pyplot as plt

def experiment():

*# Performing experiment with different communication distances*

    for comm\_dist in COMM\_DISTANCES:

*# Initializing tasks at random positions*

        tasks = []

        for \_ in range(NUM\_TASKS):

            task = Task(*task\_capacity*=TASK\_CAPACITY, *task\_radius*=TASK\_RADIUS)

            tasks.append(task)

*# Initializing agents at random positions*

        agents = []

        for \_ in range(NUM\_AGENTS):

            agent = Agent(*comm\_dist*=comm\_dist)

            agents.append(agent)

*# Starting simulation*

        results = np.zeros((NUM\_EPOCHS))

        completed\_tasks = 0

        for i in range(NUM\_EPOCHS):

            for task\_i in range(len(tasks)):

                task = tasks[task\_i]

                task\_completed = task.sufficient\_agents\_in\_radius(agents, *invoke\_auction*=True)

                if task\_completed:

                    completed\_tasks += 1

                    tasks[task\_i] = Task(*task\_capacity*=TASK\_CAPACITY, *task\_radius*=TASK\_RADIUS)

            for agent in agents:

                agent.update\_velocity()

                agent.update\_pos()

            results[i] = completed\_tasks

        x = np.linspace(1, NUM\_EPOCHS, NUM\_EPOCHS)

        y = results

        plt.plot(x, y, *label*=f'Rd = {comm\_dist}')

        print(f"Simulations for Rd = {comm\_dist} complete.")

*# Plotting results*

    plt.title("TASK A)")

    plt.xlabel("Time (# of epochs)")

    plt.ylabel("# of tasks solved")

    plt.legend()

    plt.savefig(*fname*='figures/task\_a')

    plt.close()

if \_\_name\_\_ == "\_\_main\_\_":

    experiment()

**utils.py**

import numpy as np

def distance\_euclid(*vec\_a*: np.array, *vec\_b*: np.array):

    return np.linalg.norm(*vec\_b* - *vec\_a*)

**constants.py**

AGENT\_ABSOLUTE\_VELOCITY = 25

NUM\_EPOCHS = int(1e5)

NUM\_TASKS = 2

TASK\_CAPACITY = 3

TASK\_RADIUS = 50

NUM\_AGENTS = 30

COMM\_DISTANCES = [0, 100, 200, 300, 400, 600, 1000, 1400]

**agent.py**

from utils import \*

from constants import \*

from task import \*

import numpy as np

class Agent:

    def \_\_init\_\_(

*self*,

*x*: float = None,

*y*: float = None,

*vx*: float = 0.0,

*vy*: float = 0.0,

*abs\_velocity*: float = AGENT\_ABSOLUTE\_VELOCITY,

*comm\_dist*: float = 0.0

            ):

*x* = np.random.random()\*1000 if *x* is None else *x*

*y* = np.random.random()\*1000 if *y* is None else *y*

*self*.pos = np.array([*x*, *y*])

*self*.velocity = np.array([*vx*, *vy*])

*self*.abs\_velocity = *abs\_velocity*

*self*.comm\_dist = *comm\_dist*

*self*.target\_pos = None

*self*.inside\_task\_radius = False

    def update\_pos(*self*):

        """

        Updates the current position of the agent by adding the self.velocity vector to the

        self.pos vector. When updating positions, the function disallows the agent to go out

        of bounds of the square grid spanning from (0, 0) to (1000, 1000). This means if the

        agent would go out of bounds by following its trajectory at its current absolute

        velocity, it instead moves in the same direction but at a lower absolute velocity, so

        that it stops at the border of the grid.

        """

*self*.pos = *self*.pos + *self*.velocity

*self*.pos = np.minimum(*self*.pos, 1000)

*self*.pos = np.maximum(*self*.pos, 0)

    def update\_velocity(*self*):

        """

        Updates velocity of agent according to following conditions:

        If it has reached a task (is inside task radius), it stops moving.

        If target\_pos is specified (not None) and the agent has not reached it, sets agent velocity

        towards that position.

        Otherwise, makes the agent's movement random by changing the velocity in each direction

        to some random number. Components vx and vy are set so that the absolute velocity sums up to

        abs\_velocity.

        """

*# Removes target\_pos (should it be set) if agent is inside any task radius*

        if *self*.inside\_task\_radius:

*self*.target\_pos = None

*self*.velocity = np.zeros((2))

            return

*# Goes towards target\_pos if specified and is not (almost) equal to pos*

        if *self*.target\_pos is not None and not np.allclose(*self*.pos, *self*.target\_pos):

*self*.velocity = *self*.target\_pos - *self*.pos

            norm = np.linalg.norm(*self*.velocity)

*self*.velocity = (*self*.velocity / norm) \* np.minimum(*self*.abs\_velocity, norm)

            return

*# If not, removes target\_pos and initializes random movement*

*self*.target\_pos = None

*# Sets velocity in each direction to random number in interval [-1, 1]*

*self*.velocity = (1 - (-1))\*np.random.random(*self*.velocity.shape) - 1

*# Normalizing vector, making the norm of self.velocity equal to 1*

        norm = np.linalg.norm(*self*.velocity)

*# To solve the unlikely case that both directional velicities are sampled as 0, we*

*# randomly set vx = 1 or vy = 1 if that happens*

        if norm == 0:

*self*.velocity = np.array([1, 0]) if np.random.random() > 0.5 else np.array([0, 1])

            norm = np.linalg.norm(*self*.velocity)

*self*.velocity = *self*.velocity / norm

*# Multiplying self.velocity (now a unit vector) with abs\_velocity to achieve desired*

*# (or random) velocity*

*self*.velocity = *self*.velocity \* *self*.abs\_velocity

    def callout(*self*, *agents*: list):

        """

        When the agent is within the task radius of any task, it emits a signal to other agents

        within comm\_dist to make them go towards that location, by setting their target\_pos to the

        position of the agent emitting the signal.

        The called upon agents will then go towards the coordinate from which the signal was emitted until:

        a) they reach the signal location.

        b) they find themselves within a task radius themselves.

        The above conditions are checked for each agent when their velocities are updated.

        """

*# Send a signal to any agent within comm\_dist*

        for agent in *agents*:

*# Skips itself*

            if agent == *self*:

                continue

*# Checking if the agent is within the comm\_dist or is already within a task radius*

*# (an agent which as already detected a nearby task will itself send out a signal,*

*# and presumably would then be more interested in its own discovered task than the*

*# signal of another agent)*

            if (not agent.inside\_task\_radius) and distance\_euclid(*self*.pos, agent.pos) < *self*.comm\_dist:

                agent.target\_pos = *self*.pos

    def calloff(*self*, *agents*: list):

        """

        Performs the "opposite" action of callout: instead of giving agents within comm\_dist a target\_pos,

        this function removes their target\_pos if their previous target\_pos was this agent's current pos.

        This method is called from the task when it checks whether it should be completed.

        """

*# Send a signal to any agent within comm\_dist*

        for agent in *agents*:

*# Skips itself*

            if agent == *self* or agent.target\_pos is None:

                continue

            if np.allclose(*self*.pos, agent.target\_pos) and distance\_euclid(*self*.pos, agent.pos) < *self*.comm\_dist:

                agent.target\_pos = None

    def auction(*self*, *agents*: list, *task*: Task, *num\_agents\_required*: int):

        """

        Performs a simple auction. When a task is discovered, the agent (auctioneer) calls out to other

        agents (bidders) within comm\_dist. It views their current positions (which serve as bids). Depending

        on how many more agents are required to complete the task (including the auctioneer), the auctioneer

        accepts the highest bid(s).

        This function is called from the task when at least one agent is inside its radius.

        """

*# The best bidders and their bids are stored as an ordered list of tuples*

        best\_bidders\_and\_bids = []

        for agent in *agents*:

            if agent == *self*:

                continue

            if (not agent.inside\_task\_radius) and distance\_euclid(*self*.pos, agent.pos) < *self*.comm\_dist:

*# The distance between the agent and task is calculated, and used as the bid (lower is better)*

                bid = distance\_euclid(agent.pos, *task*.pos)

*# If the required number of agents to complete the task isn't met yet, the bid and its bidder*

*# is saved in the list*

                if len(best\_bidders\_and\_bids) < *num\_agents\_required*:

                    best\_bidders\_and\_bids.append((agent, bid))

                    best\_bidders\_and\_bids = sorted(best\_bidders\_and\_bids, *key*=lambda *x*: *x*[1]) *# Sorts list by bid*

*# If the bid is better than the currently worst winning bid, the worst winning bid and its*

*# bidder is replaced by this bid and its bidder*

                else:

                    worst\_winning\_bid = best\_bidders\_and\_bids[len(best\_bidders\_and\_bids) - 1][1]

                    if bid < worst\_winning\_bid:

                        best\_bidders\_and\_bids[len(best\_bidders\_and\_bids) - 1] = (agent, bid)

                        best\_bidders\_and\_bids = sorted(best\_bidders\_and\_bids, *key*=lambda *x*: *x*[1])

*# After the auction, the winner(s) have their target\_pos set towards the task*

        for bid\_and\_bidder in best\_bidders\_and\_bids:

            bidder = bid\_and\_bidder[0]

            bidder.target\_pos = *task*.pos

**task.py**

from utils import \*

from constants import \*

from agent import \*

import numpy as np

class Task:

    def \_\_init\_\_(

*self*,

*x*: float = None,

*y*: float = None,

*task\_capacity*: int = 1,

*task\_radius*: float = 100

            ):

*x* = np.random.random()\*1000 if *x* is None else *x*

*y* = np.random.random()\*1000 if *y* is None else *y*

*self*.pos = np.array([*x*, *y*])

*self*.task\_capacity = *task\_capacity*

*self*.task\_radius = *task\_radius*

    def sufficient\_agents\_in\_radius(

*self*, *agents*: list,

*invoke\_calloff*: bool = False,

*invoke\_auction*: bool = False

            ):

        """

        Checks whether there are enough agents within the task's radius for it to be complete.

        Also invokes calloff from agents within task radius if specified. Also invokes auction

        from the first saved agent within task radius if specified (assuming that only one

        auction is to be held).

        """

        num\_agents\_in\_radius = 0

        agent\_in\_task\_radius = None

        for agent in *agents*:

*# Checking if the agent is within the task radius, adding to num\_agents\_in\_radius*

            if distance\_euclid(*self*.pos, agent.pos) < *self*.task\_radius:

                num\_agents\_in\_radius += 1

                agent.inside\_task\_radius = True

                if agent\_in\_task\_radius is None:

                    agent\_in\_task\_radius = agent

*# Checking if enough rgents are close enough to task to complete it*

            if num\_agents\_in\_radius >= *self*.task\_capacity:

                for agent in *agents*:

                    if distance\_euclid(*self*.pos, agent.pos) < *self*.task\_radius:

                        agent.inside\_task\_radius = False

*# Tells agents to perform calloff when this task is completed, if specified*

                        if *invoke\_calloff*:

                            agent.calloff(*agents*)

                return True

*# Performing auction if at least one agent is inside task radius*

        if agent\_in\_task\_radius is not None:

            agent\_in\_task\_radius.auction(*agents*, *self*, *self*.task\_capacity - num\_agents\_in\_radius)

        return False