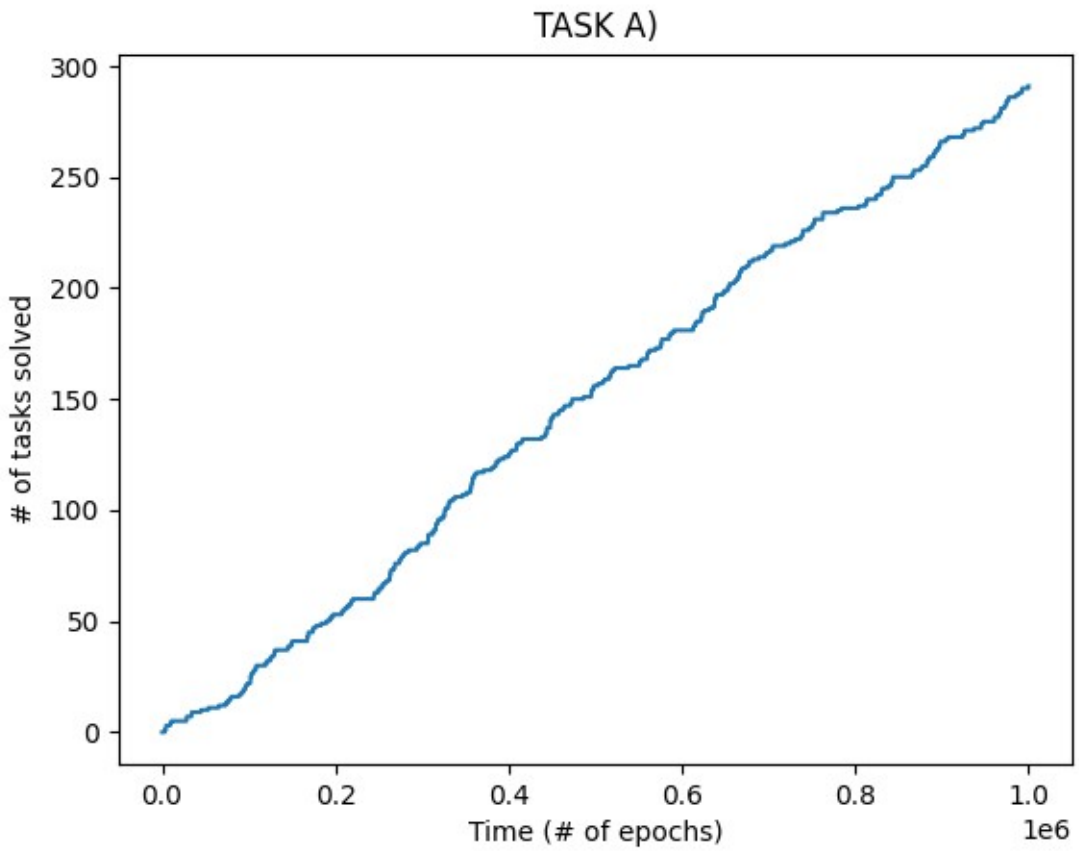
Results and Discussion

Task A

For modeling an agent moving randomly in the search area, I've implemented an agent that at each timestep moves in an entirely random direction with constant speed Rv=25, with the only limitation being that the agent is not allowed to move out of the grid. This should model an agent moving randomly well in this simulated environment; however, it is likely not a very accurate representation of real-world agent moving in physical environments, as they would likely not be able to change their moving direction at random at every timestep.

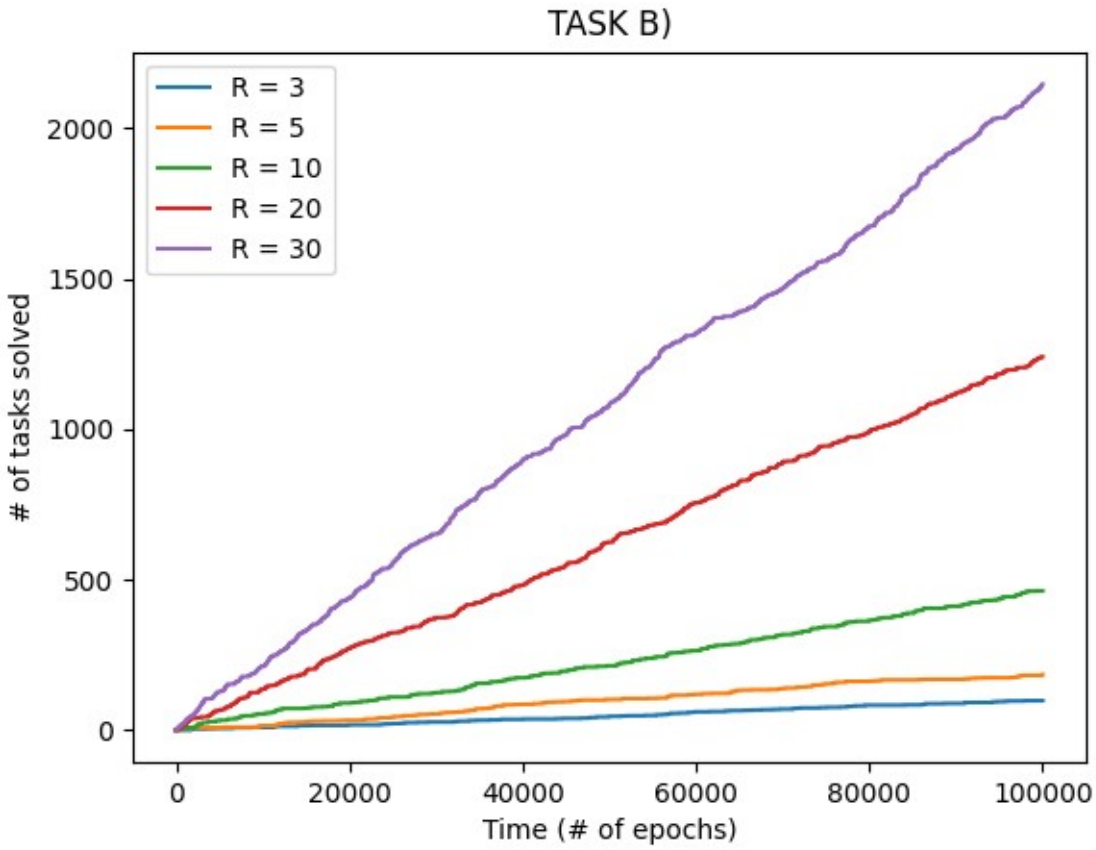


From the resulting graph of the simulation, we see that the number of tasks the agent solves per time is roughly linear, with some variance due to the randomness of the agent movement and the task position. In the case of one agent solving one task, this is a very sensible way of measuring performance in the STAC problem; specifically, it measures the Completion abilities of the agent well, as the graph measures the number of completed tasks per time.

As for Search, there may be other ways to assess whether good exploration is achieved, such as looking at grid coverage over time. However, this isn't necessary for this assignment, as we keep the exploration behavior constant for all agents in all exercises (they all move randomly until they find a task, or in later tasks, until they are called upon).

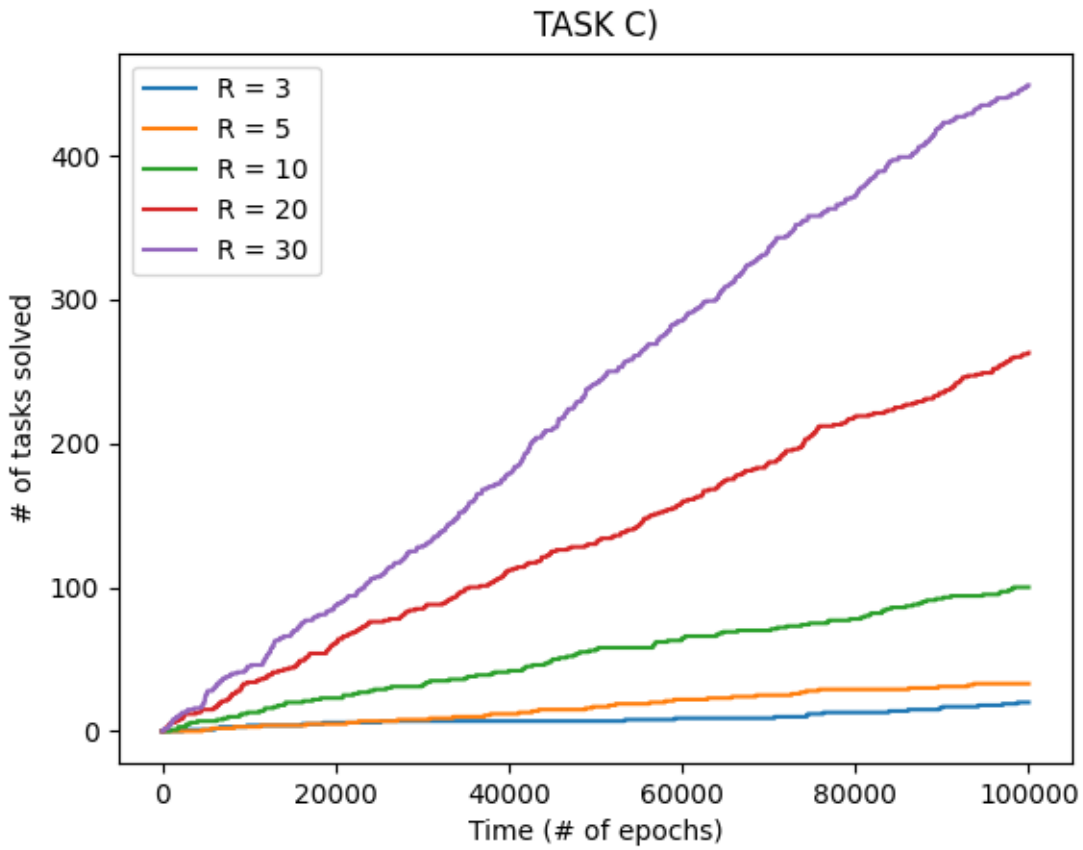
For the purpose of measuring Task Allocation, the number of tasks solved does not give us any direct assessment of whether efficient swarm behavior has been achieved; we can only assume that if the number of tasks solved should increase when task allocation efficiency increases, but we cannot observe this directly from such a graph.

Task B



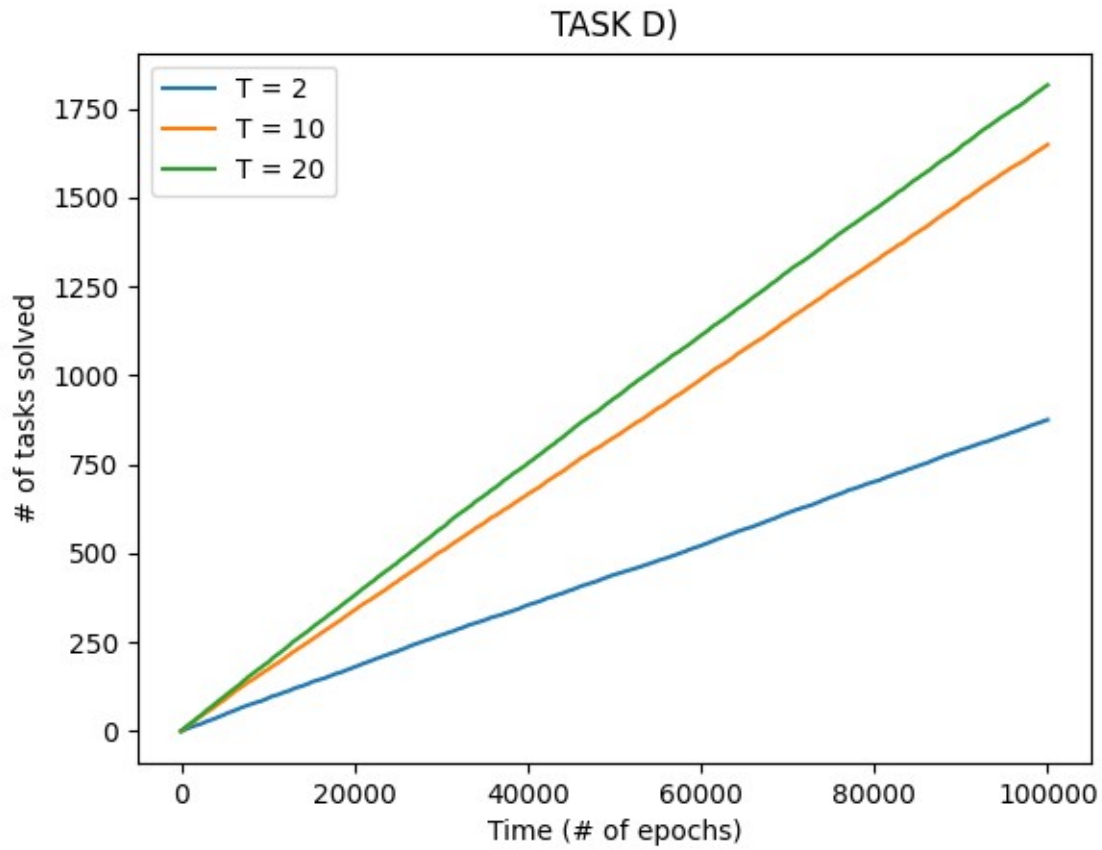
We observe that by adding agents, we still achieve a linear increase in tasks solved per time, but the rate of increase becomes higher as more agents are added. This makes sense, as more agents will cover more points in the grid at any given timestep, increasing the likelihood of one of them finding and completing the task.

Task C



This graph looks almost identical to the last one, but we observe that the number of tasks solved per time becomes much smaller than in task B.

Task D



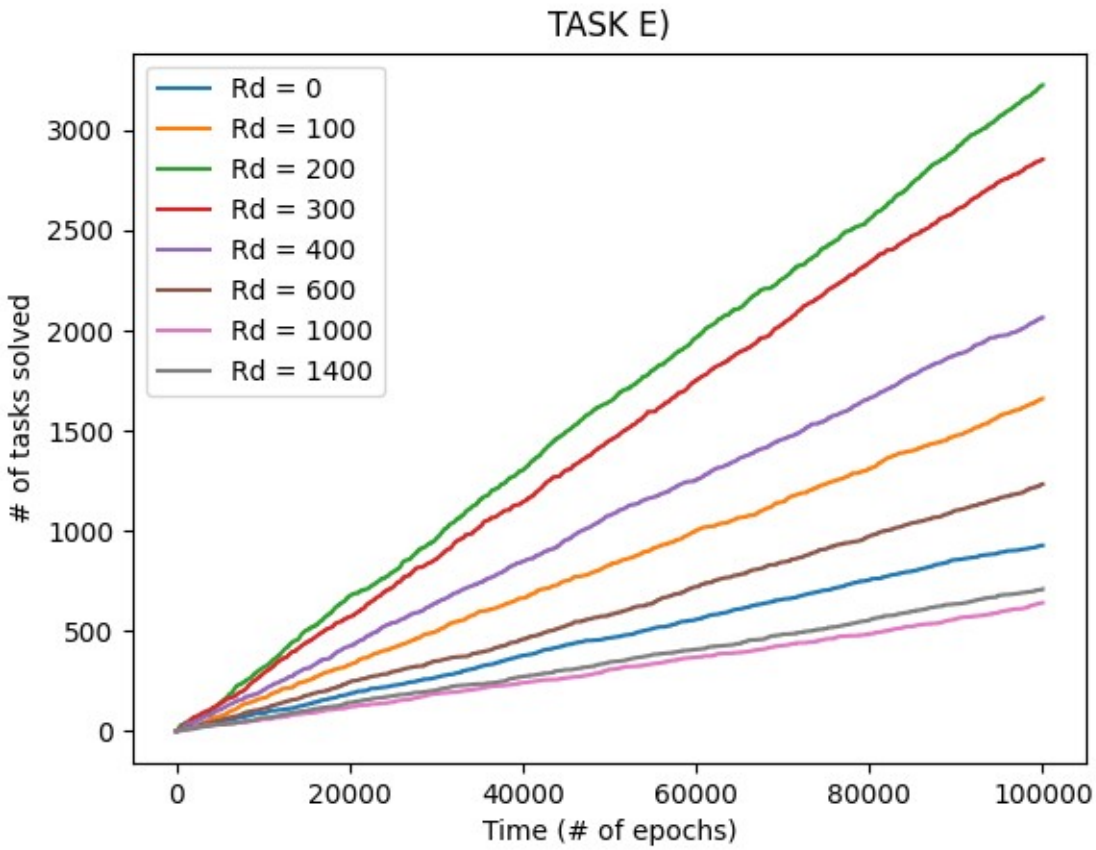
From the graph, we find that by increasing the number of tasks, we increase the average number of tasks that the agents are able to complete.

When it comes to the agents reading steady-state, I assume it refers to the possibility of all agents reaching a task in such a way that no single task has enough agents to complete the task. This only becomes possible when the number of agents is too low compared to the number of tasks and their task capacity. For instance, in my simulations, I have presumed that 30 agents are used (see code), and that tasks have a task capacity of 3. Then, each task may “trap” 2 agents (because they have reached inside a task radius, and remain there until the task is completed). In this case, whenever the number of tasks is 15 or higher, it becomes possible for all 30 agents to become trapped in the task radii, so that none of them move, and the environment has then converged to a steady state.

In theory, if the criteria above are fulfilled, any system with these criteria should at some point converge to a steady state by the randomness of movement in the agents. However, for the given numbers of tasks I have tested for, this seems very unlikely to happen, even with 20 tasks. We still observe the rate of increase growing as the number of tasks grows. I have also experimented with even larger numbers of tasks to see if it converges. At very large numbers of tasks (T = [40, 50, 60]), it seems random whether the environment converges or not from experiment to experiment, though it happens more often the higher the number of tasks.

I found that 10 episodes per number of tasks seems to give steady average performances with little variance as suggested by the very straight lines on the graph, meaning this should give a good statistical estimate of the average performance per number of tasks.

Task E

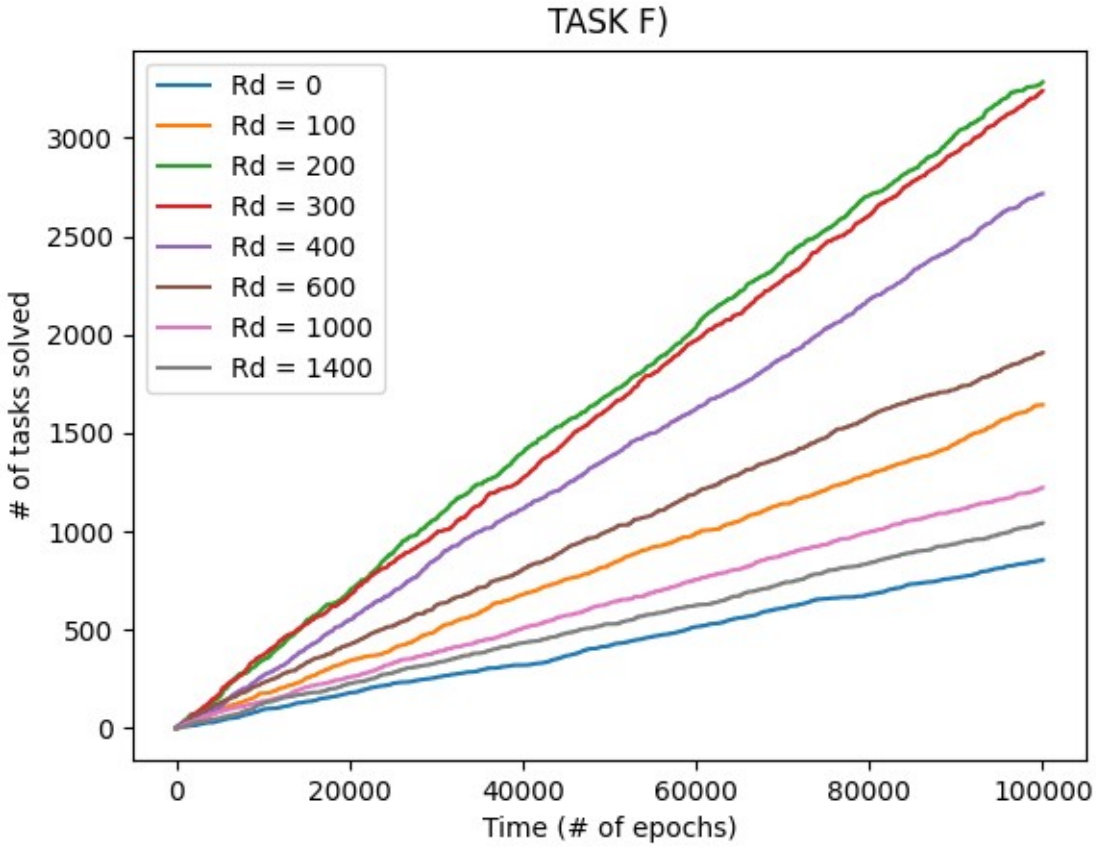


As in all the experiments so far, we again observe a linear increase in tasks solved per time. Interestingly, it is not always so that an increase in communication distance increases performance. In particular, notice how Rd=[1000, 1400] actually decreases overall performance when compared to the agents not being able to even communicate (Rd=0). At these communication distances, the agents can essentially “hear” each other across the entire grid. This means every agent can receive callouts from every agent within some task radius. A real-life analogy would be a lot of people shouting from different directions, telling you to come to them, in which case it would be hard to decide who to go to.

In the simulations, due to how I have programmed their behavior, this leads to all free agents going towards the last callout they heard, which is directly dependent on the order in which the program iterates through the agents. This can lead to unstable behavior where the agent may often change directions. While this can be changed to some other behavior like going towards the closest callout signal, the agent hearing multiple signals may still lead to unstable behavior.

Regardless of this specific instability in agent behavior, there is another likely reason that large communication distances turn out to be suboptimal. When we initialize agents randomly, this (hopefully) gives some spread of agents across the grid, leading to the agents exploring the grid better. If the agents can hear each other across the grid, the agents will tend to converge towards discovered tasks, increasing exploitation and decreasing exploration drastically, as this prevents the agents from covering the grid. By using a middle solution with a smaller communication distance, we find a middle ground where agents explore the grid, and exploit in small, local areas.

Task F



Overall, adding call-off leads to better performance in terms of tasks solved. In comparison to task E, even the largest communication distances now perform better than with no communication, though it is still somewhat short distances that give optimal performance in my simulations. It seems likely that implementing call-off combats the issues of instability and over-exploitation from task E.

Program Code Appendix