Search and Task Allocation

Swarm Intelligence

Sebastian G. Winther-Larsen

October 6, 2020

1 Formalism

We define a search A as an area bound by two points. Over the search area, n_T tasks T are spawned at random positions. The tasks have a task capacity T_c indicating how many agents R are required to solve a task. The task is solved immeadeately if T_c agents are within the task radius T_r. The agents R move randomly around the search area at a speed R_v. There are n_R agents. When an agent is inside the task radius T_r of a task, the agents will wat for other agents to complete the task. The agents can also call for aid in solving a task. The communication distance R_d determines how far an agent can send a call for aid to another agent.

The position of the agents are updated in a discrete Langevinian manner,

$$r_{R,t+1} = r_{R,t} + N\left(a_R + b_{R,t}\right), \tag{1}$$

where $r_{R,t}$ is the position of agent R at time t, a_r is a drift term, $b_{R,t}$ is a stochastic velocity term and N is a normalising operator, ensuring equal velocity of all agents. The drift term is assigned at the start of each simulation and the stochastic velocity term is updated for each time step. The trend is added in order to make sure that the agents will venture away from their origin point, as the expected value of a sum of equally distributed stochastic variables with mean zero will be zero. That is, even though there is a likelihood for agents to evetually cover the entire area, one would expect them to meander about the point they were initially placed.

In addition to walking randomly across the area, the agents have the possiblity of calling other agents in the vicinity, if they are within a communication distance R_d.

2 Implementation

We implement a tool for visualising the problem described above in PyGame [1]. The full implementation is available at `github.com/gregwinther/mas`, and consists of three classes. Two of these, Task and Agent, both inherit from PyGame's Sprite class. The last class, Simulation, takes care of the game mechanics such as updating states and drawing the area with tasks and agents. Initialising and running a particular game (simulation) can be done with very few lines. For example;

```
simulation = Simulation(1000)
simulation.cycles = 1000
simulation.communicate = True
simulation.write = True
simulation.n_T = 3
simulation.Tr = 50
simulation.n_R = 10
simulation.Tc = 3
simulation.start()
```

A screenshot of this example is shown in Figure 1.

The most important methods of the Task and Agent classes is the update() method, which is called for each iteration of the game. For a Task, this method is quite simple. It simple checks if there are enough agents within the task radius T_r to complete the task, if so the task sprite is killed.

The update() method for agents is more subtle. Firstly, it has to update the positions of the agents according to Equation 1. This is done by generating a random movement step, or finite difference velocity, which is added to the trend and then normalised to have lenght equal to R_v.

Since it is possible for an agent to wander outside the area, we take care of the boundaries in the following manner. When a moving agent hits one of the boundaries of the area, the trend unit vector which is perpendicular to the boundary line is reversed. This has the effect of "soft bounce" off the boundary.

More than the mere movement of the agents, it is possible for them to be in different states. They may be *searching*, *tasked* or *called*. The default state, *searching*, is what we just described above. Whenever an agent is within the task radius T_r of a task T its state will change to *tasked*. If the task capacity T_c required more agents than the one(s) present, the agent will stay put until enough agents arrive at the task to help. With the mechanic enabled, an agent may also be *called* if it is within the communication dis-

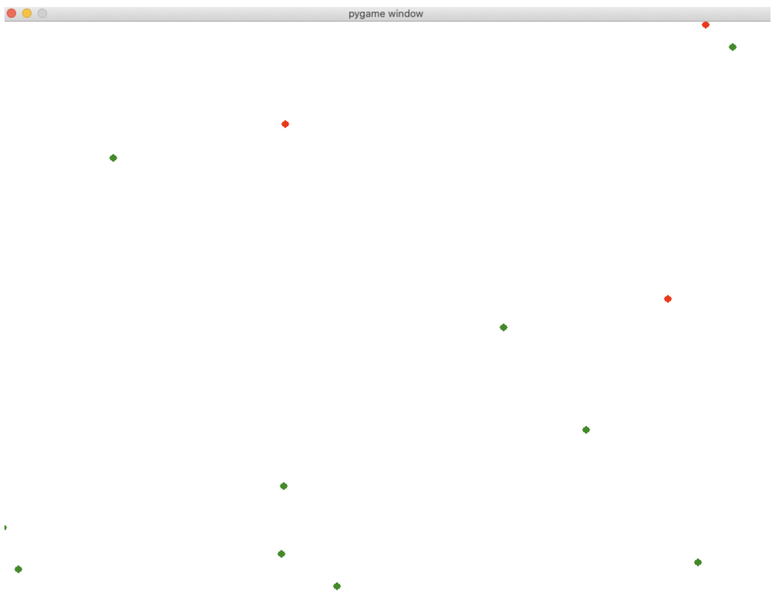

Figure 1: Sample screenshot of a simulation with $n_T = 3$ and $n_R = 10$. The green dots are agents R, and the red dots are tasks T.

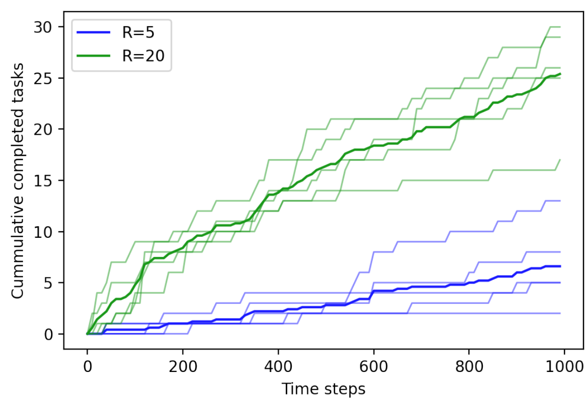

Figure 2: Illustration of arbitrary simulations, with cumulative number of solved tasks is plotted against the number of time steps. The number of tasks T_n is 1 and the number of agents R_n are 5 and 20. The task capacity T_c was set to one. No communication was allowed in these simulations.

tance R_d of another agent. The agent will then move at speed towards the position of the signalling agent. When the task is solved, the call will cease, in effect calling off any approaching agents. All of the transitions between agent states are handled within the main game loop of the Simulation class.

3 Results

All the simulations in this study is run five subsequent times in order to get better averaged values, for a total of 1000 time steps. All simulations also has the same task radius $T_r = 50$. Figure 2 shows the results of two arbitrary chosen sets of five simulations. The thin lines represent the individual simulations, while the bold lines are the averages. For both of these sets the number of tasks was $T_n = 1$ with a task capacity $T_c = 1$, while the number of agents n_R are five and 20. The figure is meant to give illustrate the necessity of repeated simulations in order to make results separable.

To benchmark the system we run a number of simulations for various number of tasks T and number of agents R, with task capacity $T_c = 1$. The results of these simuabltions are shown in Figure 3. The points indicate the average number of completed tasks at the end of a simulation, with standard deviation given by the error bar. As expected, the number of completed tasks increase with the number of agents R and with the number

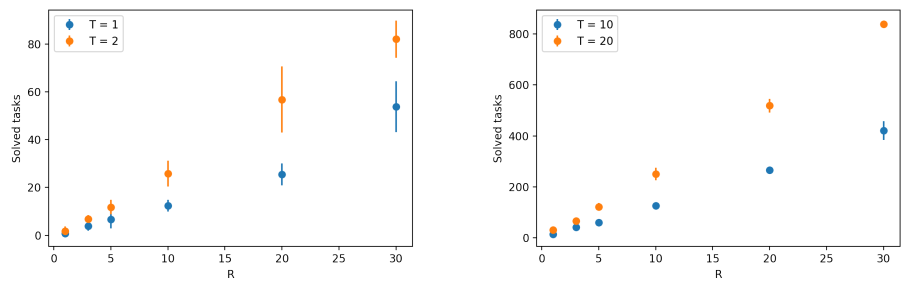

Figure 3: Completed tasks as a function of number of agents after 1000 time steps. The number of agents and tasks are varied, while every other variable is held constant; $T_c = 1$, $T_r = 50$, no communication.

of tasks T.

The results of introducing a higher task capacipy is depicted in Figure 4. Here we set $T_c = 3$, meaning that three agents is needed to complete a task T. This figure is comparable to the left-hand subfigure in Figure 3. We see a notably lower number of completed tasks when a task capacity is introduced.

The results of introducing a call signal with reach $R_d = 250$ can be seen in Figure 5. The average performance is markedly and significantly better when agents can call to others for help.

Figure 6 shows the effect of increasing the communication distance of the agents. The cumulative number of completed tasks is plotted against time. In the simulations, only the communication distance is varied, $R \in \{250, 500, 1000\}$, all else equal ($T_c = 3$, $n_T = 2$ and $n_R = 5$).

4 Discussion

For lower n_R more simulations are needed. Higher relative error. Simulations with $T_c = 3$, some were especially non-interesting for low n_R. Try to illustrate that $n_T = 2$ does not work well for $n_R = 5$ here.

There are some sticky situations also in the call vs no call figure when $n_R = 3$.

What would be better? Improve movement: To spread out in some pattern? Only call needed agents?

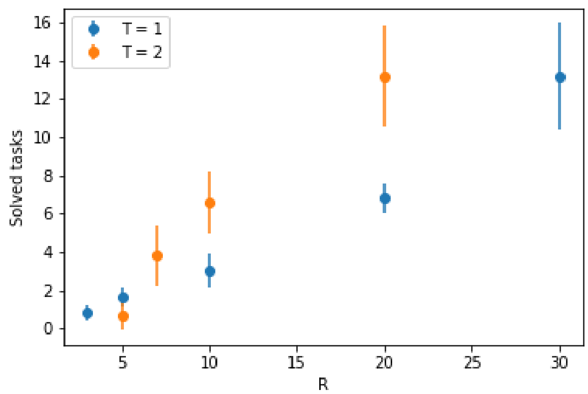

Figure 4: Completed tasks vs number of agents with $T_c = 3$, for two different task numbers $n_T \int \{1, 2\}$.

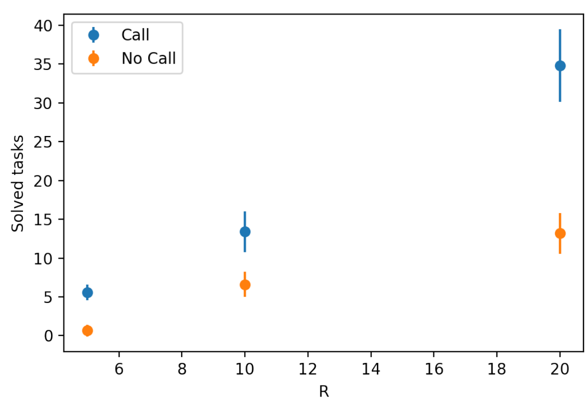

Figure 5: The effect of introducing a calling signal with $R_d = 250$ when an agent finds an agent finds a task. The number of tasks are $n_T = 2$ with capacity $T_c = 3$.

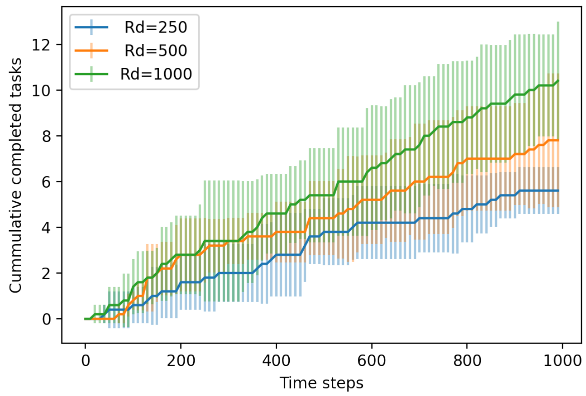

Figure 6: Averages of three sets of five simulations for different communication distances $R_d \in \{250, 500, 1000\}$, with $T_c = 3$, $n_T = 2$ and $n_R = 5$.

References

1. Shinners, P. *et al. PyGame* `http://pygame.org/`. 2020.