Search and Task Allocation

Game Theory

Sebastian G. Winther-Larsen

November 10, 2020

1 Introduction

This study is a continuation of a previous study where we implemented a system of randomly moving reactive agents R that solve tasks T randomly distributed over a square search area A [1]. While we have previously solved this search and task allocation (STA) problem with *reactive*, passive agents; we will in this study solve the same problem using *strategic* agents.

Now, whenever a new task is discovered an auction takes place among the agents within communication range. The discoverer of the task functions as the acutioneer, and the agents within communication distance as bidders. The potentially helping bidders use the distance to the task as their bid in the auction. The acutioneer will recruit help based on their bids in order to have enough agengs, including itself, to solve the task.

2 Extending PyGame Framework

From before we already have a framework implemented in PyGame[2], that is able to simulate the STA problem with reactive agents. The framework consists of three main classes, `Agent`, `Task` and `Simulation`. The `Task` class' relevant members are the task capacity T_c and an `update()` method that kills the task if enough agents, compared to T_c, are tasked to it. The `Agent` class' most important functionality is the `update()` method that functions differently depding on the *state* of the agent. An agent can be *searching*, *tasked* or *called*. As such, it is a very passive sprite. All other mechanics within the simulation is controlled by the `Simulation` class. Therefore, it is easiest from an implementation standpoint to extend the `Simulation` class to make the agents strategic, because it is here that the states of the `Agent` class instances are changed.

To accomplish this, we move the lines of code that checks if a task is disovered, i.e. an agent is within task radius T_r of the tasks, to a separate method within the `Simulation` class. Then we can create a subclass `AuctionSimulation` that only needs to reimplement this method, `check_for_tasks()`. The entire implementation is included in Appendix A.

References

1. Winther-Larsen, S. G. *Multi-Agent Systems* `https://github.com/gregwinther/mas`. 2020.
2. Shinners, P. *et al. PyGame* `http://pygame.org/`. 2020.

A Code listing

```
import pygame, sys
import numpy as np
import math
from collections import OrderedDict

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREY = (200, 200, 200)
PASTEL_RED = (245, 130, 120)
MINT = (160, 250, 160)
FOREST_GREEN = (30, 140, 30)
RED = (255, 0, 0)

BACKGROUND = WHITE

FRAME_RATE = 12


class Task(pygame.sprite.Sprite):
    def __init__(self, x, y, Tc=1, Tr=50):
        super().__init__()
        radius = 5  # Size of task
        self.image = pygame.Surface([radius * 2] * 2)
        self.image.fill(BACKGROUND)
        pygame.draw.circle(self.image, RED, (radius, radius), radius)

        self.pos = np.array([x, y])
        self.rect = self.image.get_rect()
        self.rect.centerx, self.rect.centery = x, y

        # Task capacity
        self.Tc = Tc
        self.tasked_agents = pygame.sprite.Group()

        # Task radius
        self.Tr = Tr

    def update(self):

        # Task completion
        if len(self.tasked_agents) >= self.Tc:
            self.kill()


class Agent(pygame.sprite.Sprite):
    def __init__(self, x, y, Rv=25, boundary=1000, Rd=250):
```

```
        super().__init__()
        radius = 5
        self.image = pygame.Surface([radius * 2] * 2)
        self.image.fill(BACKGROUND)
        pygame.draw.circle(
            self.image, FOREST_GREEN, (radius, radius), radius
        )

        self.pos = np.array([x, y], dtype=np.float64)
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x, y

        self.Rv = Rv  # / FRAME_RATE
        self.Rd = Rd

        self.vel = np.random.rand(2) * 2 - 1
        self.trend = (np.random.rand(2) * 2 - 1) / 2

        self.boundary = boundary

        self.tasked = False
        self.called = False
        self.call_dir = np.array([0, 0])

    def update(self):
        if self.tasked:
            # Stand still
            return

        if self.called:
            # print(self.call_dir)
            self.pos += self.call_dir
            self.rect.centerx, self.rect.centery = self.pos.copy().astype(
                "int"
            )
            return

        self.pos += self.normalize(self.vel + self.trend, self.Rv)
        self.rect.centerx, self.rect.centery = self.pos.copy().astype(
            "int"
        )

        # Boundary conditions
        if self.pos[0] > self.boundary or self.pos[0] < 0:
            self.vel[0] = -self.vel[0]
            self.trend[0] = -self.trend[0]
            if self.pos[0] > self.boundary:
                self.pos[0] -= 20
            if self.pos[0] > self.boundary:
```

```
                self.pos[0] += 20
            elif self.pos[1] > self.boundary or self.pos[1] < 0:
                self.vel[1] = -self.vel[1]
                self.trend[1] = -self.trend[1]
                if self.pos[1] > self.boundary:
                    self.pos[1] -= 20
                if self.pos[1] > self.boundary:
                    self.pos[1] += 20
            else:
                self.vel = np.random.rand(2) * 2 - 1

        def dist_to_sprite(self, other_sprite):
            return math.hypot(
                self.pos[0] - other_sprite.pos[0],
                self.pos[1] - other_sprite.pos[1],
            )

        def direction_to_sprite(self, other_sprite):
            direction = other_sprite.pos - self.pos
            return self.normalize(direction, self.Rv)

        def normalize(self, vector, renorm=1):
            vector /= np.linalg.norm(vector)
            return vector * renorm


    class Simulation:
        def __init__(self, width=1000):
            self.WIDTH = width  # Square

            self.tasks = pygame.sprite.Group()
            self.agents = pygame.sprite.Group()
            self.tasked_agents = pygame.sprite.Group()

            self.cycles = 1000

            self.n_T = 5
            self.Tc = 1
            self.Tr = 50
            self.n_R = 1
            self.Rd = 250

            self.replace_tasks = True
            self.communicate = False

            self.write = False
            self.save_interval = 10

            self.total_tasks_solved = 0
```

```
def check_for_tasks(self):
    for agent in self.agents:
        for task in self.tasks:
            distance = agent.dist_to_sprite(task)
            if abs(distance) < task.Tr and not agent.tasked:
                print(f"Aha! A task at {task.pos}")
                agent.tasked = True
                task.tasked_agents.add(agent)
                self.tasked_agents.add(agent)

    # Call out (and off)
    if self.communicate:
        for agent in self.tasked_agents:
            # Make sure to not call oneself.
            other_sprites = self.agents.copy()
            other_sprites.remove(agent)
            for agent2 in other_sprites:
                distance = agent.dist_to_sprite(agent2)
                if abs(distance) < agent2.Rd:
                    agent2.called = True
                    agent2.call_dir = agent2.direction_to_sprite(agent)

def start(self):

    if self.write == True:
        f = open(
            f"sta_T{self.n_T}_Tc{self.Tc}_Tr{self.Tr}_R{self.n_R}.txt",
            "w",
        )

    pygame.init()

    area = pygame.display.set_mode([self.WIDTH, self.WIDTH])

    # Assigning initial Tasks
    # Random positions of initial tasks (n, dim)
    T_pos = np.random.randint(0, self.WIDTH, size=(self.n_T, 2))

    for x, y in T_pos:

        T = Task(x, y, Tc=self.Tc, Tr=self.Tr)
        self.tasks.add(T)

    # Assigning initial Agents
    R_pos = np.random.randint(0, self.WIDTH, size=(self.n_R, 2))

    for x, y in R_pos:
```

```
        R = Agent(x, y, Rd=self.Rd)
        self.agents.add(R)

    clock = pygame.time.Clock()

    # Game loop
    for i in range(self.cycles):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                if self.write:
                    f.close()
                sys.exit()

        if (self.write == True) and (i % self.save_interval == 0):
            f.write(f"{i:3} {self.total_tasks_solved:3}\n")

        self.tasks.update()
        self.agents.update()

        # Any tasks completed? Create new tasks.
        n_new_tasks = self.n_T - len(self.tasks)
        self.total_tasks_solved += n_new_tasks
        if n_new_tasks > 0:
            print("Solved!")
            R_pos = np.random.randint(
                0, self.WIDTH, size=(n_new_tasks, 2)
            )
            for x, y in R_pos:
                T = Task(x, y, Tc=self.Tc, Tr=50)
                self.tasks.add(T)
            for agent in self.agents:
                agent.tasked = False
                agent.called = False
            self.tasked_agents.empty()

        # Checking if a task is found
        self.check_for_tasks()

        area.fill(BACKGROUND)
        self.tasks.draw(area)
        self.agents.draw(area)
        pygame.display.flip()

        clock.tick(FRAME_RATE)

    if self.write:
        f.close()
    pygame.quit()
```

```
class AuctionSimulation(Simulation):
    def check_for_tasks(self):
        for agent in self.agents:
            for task in self.tasks:
                distance = agent.dist_to_sprite(task)
                if abs(distance) < task.Tr and not agent.tasked:
                    agent.tasked = True
                    task.tasked_agents.add(agent)
                    self.tasked_agents.add(agent)

        # Call out (and off)
        # Only the closes number of necessary agents
        if self.communicate:
            agent_bids = {}
            for agent in self.tasked_agents:
                # Make sure to not call oneself.
                other_sprites = self.agents.copy()
                other_sprites.remove(agent)
                for agent2 in other_sprites:
                    distance = agent.dist_to_sprite(agent2)
                    if abs(distance) < agent2.Rd:
                        agent_bids[distance] = agent2

                ordered_bidding_agents = OrderedDict(
                    sorted(agent_bids.items())
                )

                i = 0
                for bid in ordered_bidding_agents:
                    if i >= self.Tc - 1:
                        break
                    bidding_agent = ordered_bidding_agents[bid]
                    bidding_agent.called = True
                    bidding_agent.call_dir = (
                        bidding_agent.direction_to_sprite(agent)
                    )
                    i += 1
```