Victor Goncalves & Emeka Ezike

Professor Stephanie Gil

COMPSCI 286

16 February 2022

PSET 1 Writeup

Group Members

Victor Goncalves

Emeka Ezike

Section I

A. In this part of the problem set, we used a centralized approach to implement this method. Specifically, we have access to global information when creating each of these methods.

- **safe wander**: Depending on whether we use this function with a flock or not, it either moves the entire flock of robots in the same random direction or each robot moves in a different direction.
- **aggregate**: We calculate the center of the robots using the global list of robots. Then, we have each of the bots move towards this center, but we chose an implementation where each robot moves a step at a time so in order to best simulate robots moving at their own timestep.
- **disperse**: Robots are programmed to move away from the global center of all the robots in random directions.

- **flock**: Due to a centralized implementation, a flock is designed to contain all the robots within the environment as they work together doing tasks (like safe-wander).

Lastly, the two combinations of robots and environment sizes are as follows:

1. 4 robots in an environment of size 10
2. 8 robots in an environment of size 8

Both of these combinations followed with nearly similar results.

B. In this part of the problem set, we work to implement a decentralized approach in which robots do not have globally accessible information. We had robots move randomly to find each other to merge and create a single flock. These are the descriptions of the methods we completed:

- **Bot_sense:** Using the euclidean distance formula to get the distance between two grid points, to get a list of each bot's neighbors, we iterate through the entire list of bots to find the neighbors within the sensing radius for each bot.
- **Update_flock:** Using the list of neighbors for each bot created using bot_sense, we create flocks by searching a bot and its neighbors, and the neighbor's neighbors etc.. Furthermore, we reinitiate the global flock list after every iteration to handle merging flocks.
- **Safe wander sense**: This function was implemented almost exactly as "safe-wander" from Section I. Now, if we are flocking, robots in their individual flocks move together rather than globally. We use this in "aggregate sense" to move robots randomly until they find each other to make a flock.

- **Aggregate sense**: In this function, we want robots to move randomly until they get within the sensing radius of another robot. This way, we can create flocks without knowing information about the global information of other robots. Using a combination of bot_sense, update_flock, and a helper function we created that moves all bots in a flock to its centroid, we are able to aggregate all the bots into a single flock. We first find bot neighbors, and update the flock list. We then have bots move randomly until it finds other bots/flocks to merge with, then updates the flock list after every iteration. This process continues until all bots are in a single flock.
- **Disperse sense**: By this method, robots should be in a single flock. We use a similar method to disperse and part A, and we have robots move away from the center of their flock.

To test different sensing values to see how it affects bot behavior, we tested the following radii in our environment for the decentralized case.

1. Partial sensing:
    a. Radius value of 2
    b. Radius value of 5
2. Global sensing:
    a. Radius value of the entire environment: 10

Upon testing these values, it seemed that when using a radius of 5, the robots converged into a flock much faster than when using the radius value of 2 case. Furthermore, when using global sensing, the behavior of the program mimicked the centralized case: the robots

merged instantly in the center of their flock, safe-wandered, and dispersed. In the global case, the robots converged into the flock the fastest, which you can see visually in the graph pdf by the total number of moves the bots had to make to reach convergence.

D. In a real world scenario, our simulation could represent a rendezvous or search and rescue problem. For example, if we put robots in an environment that they have never been exposed to before, like we did with turtlesim, how will they move to find each other? Using ideas from "Networked Robots," by Kumar, Rus, and Sukhatme, we can use a networked system to our simulation to provide multiple advantages–specifically in terms of extending the sensing range. If we could use a networked system, information about robot position and sensors are exchanged via the network, therefore in these systems, information can be communicated over longer distances and effectively extends the sensing radius over the network (Kumar, Rus, and Sukhatme 944). When applying this to nerd-herd.py, we saw that extending the sensing radius during testing provided quicker merge times with the robots. If we were to use networks, robots can merge with each other faster, and it could possibly provide global sensing if the sensing radius is large enough. Furthermore, a networked system could provide additional advantages in the real-world in terms of providing fault tolerance, and it allows robots to communicate in multiple unique scenarios (for example, through walls and even in remote situations).

Section II

2)

a) In question 2a, we are asked to generate the assignment of maximum utility when given a quality and cost matrix for robots and tasks. This problem is isomorphic to the linear sum assignment problem, also known as minimum weight matching in bipartite graphs. The problem instance is described by a matrix C, where each C[i,j] is the cost of matching vertex i of the first partite set (a "worker") and vertex j of the second set (a "job"). The goal is to find a complete assignment of workers to jobs of minimal cost. However, in our problem, we are looking for the maximum utility. We claim that if we aptly structure our matrices Q and C, we can reduce this problem to the linear sum assignment problem. As explained in "A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems", a utility matrix U can be created from the quality and cost matrices by subtracting matrix C from matrix U.

$$U_{RT} = \begin{cases} Q_{RT} - C_{RT} & \text{if } R \text{ is capable of executing } T \text{ and } Q_{RT} > C_{RT} \\ 0 & \text{otherwise.} \end{cases}$$

In order to aptly reduce to linear sum assignment, we need to model our utility as a cost so that minimizing this cost is the same as maximizing utility. This can be done by negating all the utility values. We can prove that this preserves correctness by contradiction. Assume some set A is the maximum subset of some parent set P, such that A = {a1, a2, a3} and sum(A) = a1 + a1 + a3. Assume P' = P, but all values of P are negated. Therefore, for any subset Q' in P', sum(Q') = (-q1) + (- q2) + (- q3) = - (q1 + q2 + q3) = -sum(Q). Assume that some set B' is the minimum subset in P', and not A'.

That means that sum(B') < sum(A'). If we multiplied both sides of this inequality by -1, we would get -sum(B') > -sum(A') or sum(B) > sum(A). However, this violates our claim that A is the maximum subset. So, negating all elements must convert the maximum sum into the minimum sum. Knowing our methodology is correct, we can construct our algorithm as follows: First, subtract matrix C from matrix Q. This results in a matrix U^n which represents the negative utility matrix. In other words, U^n is the matrix U, but all the utility values are negated. Then, abstract the problem to the linear_sum_assignment function in the scipy library by calling linear_sum_assignment on the matrix U^n . This function implements the famous Hungarian method, a combinatorial optimization algorithm that solves the assignment problem in polynomial time. Finally, after generating the minimized negative utility from the linear_sum_assignment abstraction, return the opposite of those values if their opposite is greater than or equal to zero. We do this because, as shown in the correctness proof, minimized negative utility is equivalent to maximize utility when negated. Additionally, we only want to perform tasks with a utility greater than zero. If a task has less than zero utility, there is no reason for the robot to be assigned to the task, as the quality from the task's completion is outweighed by the cost. As shown in all the test cases, this theoretically always generates the most optimal assignments, where optimal, as defined by Gerkey and Mataric, means "given the union of all information available in the system…, it is impossible to construct a solution with higher overall utility" (941). However, in practice "the robots' utility estimates will be inexact due to sensor noise, general uncertainty, and environmental change. These unavoidable characteristics of the multi-robot domain will necessarily limit the efficiency

with which coordination can be achieved" (Gerkey 941). Nevertheless, we can treat this limit as exogenous and assume that lower level robot control has already been made as reliable, robust, and precise as possible.

b) In question 2b, we provide an algorithm to solve a variation of the multi-robot task allocation problem known as the on-line assignment. On-line assignment problems are variants of SR–ST–IA where the set of tasks is revealed one at a time instead of all at once and previously assigned robots cannot be reassigned. For this question, we implement Murdoch the greedy algorithm described in the paper "A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems". The Murdoch algorithm is simply detailed as such: "When a new task is introduced, assign it to the most fit robot that is currently available" (Gerkey 945). Therefore, our problem-solving heuristic makes the locally optimal assignment for the current robot at each stage, without regard for future robot assignments. Our adaptation of the Murdoch assignment algorithm is as follows:

    1) We first subtract C from Q for all tasks to create a utility list for all tasks.
    2) Then, for all tasks that have already been attempted, we set the utility for the task to negative infinity.
    3) Then, we assign the robot to the task with maximum utility.
    4) Then, we add the selected task to the set of attempted tasks.
    5) Continue to the next robot assignment and repeat from step 1

Because of the constraints of the on-line assignment problem, robots have no knowledge of future utility calculations and cannot be reassigned. As a result, Gerkey and Mataric state that “it is impossible to construct a better task allocator than MURDOCH” (945).

c) As explained in the end of question 2b, due to the constraints of the on-line assignment problem (no foreknowledge, no reassignment), this is the best task allocation algorithm. Even if we had access to all information at once, a greedy algorithm like Murdoch cannot guarantee the most optimal solution. We can estimate the lower bound of the algorithm's performance by reducing the Murdoch algorithm to the BLE algorithm. The BLE assignment algorithm is another greedy assignment algorithm but for the standard SR–ST–IA and is defined as follows: “1. If any robot remains unassigned, find the robot–task pair (i, j) with the highest utility. Otherwise, quit. 2. Assign robot i to task j and remove them from consideration. 3. Go to step 1.” (Gerkey 944). This essentially functions the exact same as the Murdoch algorithm, but BLE accesses more information at once. According to Gerkey and Mataric, “[t]he Greedy algorithm is known to be 2-competitive for the OAP (Avis 1983), and thus so is BLE. That is, in the worst case, BLE will produce a solution whose benefit is half of the optimal benefit” (944). The researchers explain that greedy algorithms are 2-competitive, which is why the BLE algorithm at worst produces a solution whose benefit is half of the optimal utility. Ergo, because Murdoch is also a greedy algorithm that follows a nearly identical structure to BLE, Murdoch’s worst case solution must also be half of the true optimal utility.

d) The main changes when you transition from Single-Robot tasks to Multi-Robot tasks are assignment algorithm complexity and runtime. According to researchers Gerkey and

Mataric, ST–MR–IA (Single task robots, multi-robot tasks, instantaneous assignment problems) are "significantly more difficult than [ST–SR–IA], which were restricted to single-robot tasks" (946). ST-MR-IA problems are approached in terms of sets: we split the robots into certain disjoint sets and assign the sets to specific tasks such that this amalgamation of sets and their assignments has the most utility out of all other amalgamations. This problem is isomorphic to the Set Partitioning Problem. The Set Partitioning Problem is strongly NP-hard, meaning that there is currently no polynomial time solution for this problem (Garey and Johnson 1978). Therefore, there exists a massive runtime disparity between assignment algorithms for Single-Robot tasks and Multi-Robot tasks.

e) Decentralized allocation systems might be preferable to centralized allocation systems for a variety of reasons. For example, centralized allocation systems incur higher communication overhead, as n^2 messages are needed to send each robot's utility for any given task (where n is the number of robots). On the other hand, a distributed system requires much less, possibly even fewer than n messages. This difference is amplified as the number of robots in a system increases. Also, a decentralized system is more robust to failure than in a centralized system. If a robot fails, other robots will continue based on information from their neighbors, rather than using the failed robot in global calculations. Furthermore, a decentralized implementation might be more representative of real-world scenarios with multi-robot systems, and providing decentralized communication might be more realistic and easier to implement than a centralized system.