Final Report

1. Project Scope
    a. The aim of this project is to design a system that can explore a graph to retrieve objects throughout the graph. The robots begin exploring the graph while sending node data back to the controller which is mapping the graph. When a robot comes across an object, it retrieves the object back to the control node. The projects objectives are:
        i. To randomly create a graph with different nodes such as blocking nodes, object nodes, and control node.
        ii. To allow the robots to map the graph by sending node data to a controller.
        iii. To create an algorithm for the robots to get back to the controller node with the object efficiently.
2. Project Plan
    a. Timeline
        i. Week 1 (10/14 - 10/20): Start development, set up repo and dev environment

ii. Week 2(10/21-10/28): Verify graph generation, create classes to represent objects for attributes such as location and status (retrieved or not retrieved). Implement basic functionality for robots to traverse the graph, update the controller with node information, and mark the explored nodes.

iii. Week 3 (10/29 - 11/3): Verify robot graph mapping works, begin robot object retrieval system, verify that new classes are working as expected

iv. Week 4 (11/4 - 11/10): Establish functionality for robot collaborative object retrieval.

v. Week 5 (11/11 - 11/17): Ensure all aspects of projects are functional and cohesive. Work on final presentation and report.

b. Accomplished Milestones

i. Established ability to generate a graph randomly (October 16th)

ii. Verified graph generation with different node configurations testing (October 17th)

iii. Created necessary classes to represent objects, ensuring attributes like location and status (retrieved or not) were included (October 21st)

iv. Established functionality for robots to traverse the the graph and update the controller with node locations (October 27th)

3. Team Roles

a. Jack Gladowsky - Algorithm Development and Optimization, Complexity Analysis, Code Review

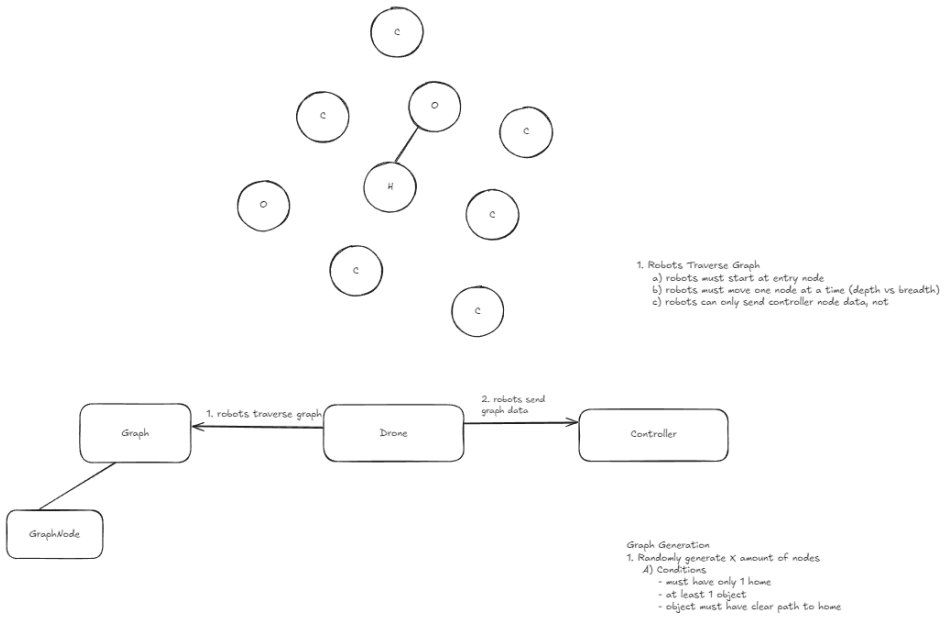
b. Will Fox - Algorithm Development and Optimization, Testing, Debugging, Code Review

c. Ishaan Desai - Algorithm Development and Optimization, Documentation, Code Review

4. Methodology

a. Pseudocode and Complexity Analysis

i. **Data Structures and Diagram of System Architecture:**



We are generating graphs, so this is the primary data structure of choice. The above flowchart provides a brief overview of the graph that is created for the robots to traverse. The nodes will be generated and subsequently, a graph of the nodes will be created by setting “neighbor” nodes for each node. Technically, this is a bidirectional, fully-connected graph, meaning that all nodes are reachable from any other node and that all connections between nodes can be traversed both ways. We determined that this was the best method for our implementation to maintain simplicity and represent a realistic scenario. The robots are controlled by a controller, which will give the robot traversal instructions. The robot traversal algorithm is explained further below.

Algorithm Design:

Graph Generation:

The graph generation algorithm works by creating a set number of nodes and for each, assigns a type. The different types are clear, obstacle, object, or control and are assigned based on random number probabilities. Each node then connects to a number of other “neighbor” nodes, to for a bidirectional, connected graph that ensures each node has at least one

neighbor. This is done using a random selection process that selects nodes to be added as neighbors for each new node, and ensures nodes are not added as neighbors more than once by checking if the node already exists as a neighbor before adding the node as a neighbor. The number of neighbor nodes may be set to a certain number to reduce complexity in the future. Lastly, a control node is set randomly to one of the pre-generated nodes, providing a “home” from which the robots will spawn and begin to explore the graph.

Robot Traversal:

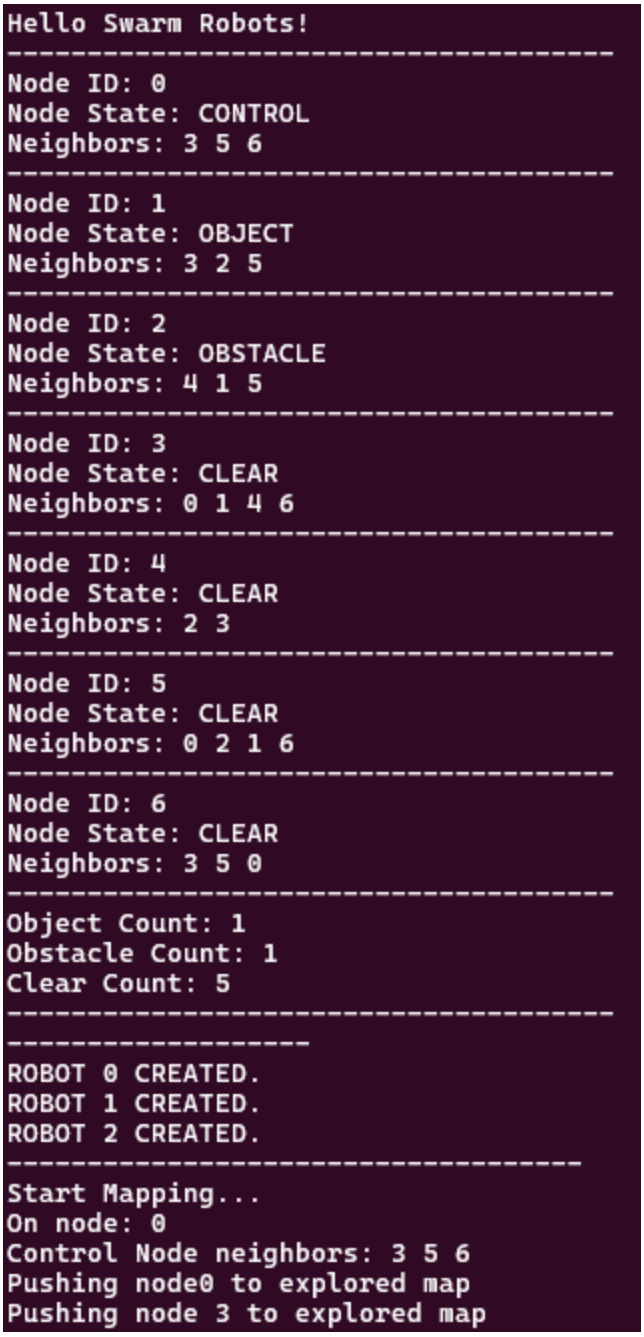
Each robot starts from the home node, where the controller is, and initializes its movement to an adjacent, unexplored node. Movement from node to node is determined by a depth-first search approach, with each robot selecting an unexplored neighboring node from its current position. This choice is made by accessing the node’s list of neighbors, which is shared among all robots to prevent duplicate visits. Each robot keeps a local record of visited nodes and, upon reaching a new node, sets a "visited" or "explored" flag on that node.

To mark each node's exploration status, robots use a shared data structure (stored at the controller) where they update each visited node’s state to “explored.” As each robot moves to a new node, it first checks if the node is already flagged as “explored” in this shared map to prevent revisiting nodes. When encountering a node, the robot inspects its type, updating the node status as "clear," "obstacle," or "object." For obstacles, the robot marks the node as impassable, which other robots will read and avoid, optimizing their exploration routes.

This shared map will be updated by the controller whenever a robot reports back to keep a record of the graph's state. Robots will continue moving outward from their last explored node, revisiting nodes only when they need to backtrack. This exploration process repeats until all accessible nodes in the graph are marked, allowing the swarm to build a complete, dynamically updated map that serves as the foundation for further object retrieval or navigational tasks.

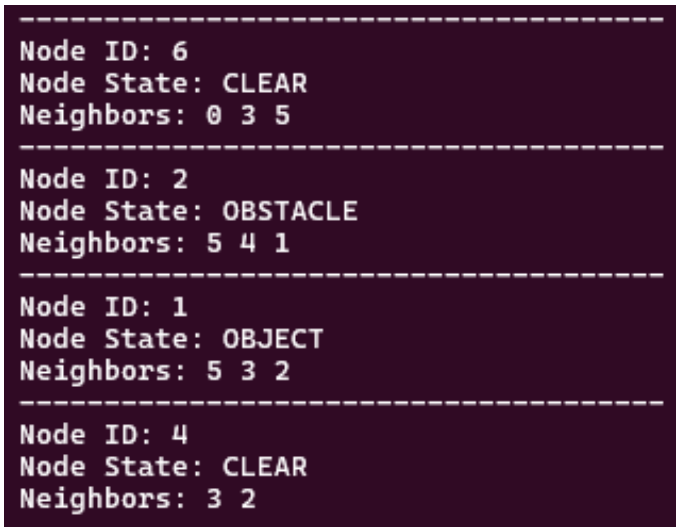
b. Data Collection and Preprocessing

5. Results
6.
    a.


Hello Swarm Robots!
Node ID: 0
Node State: CONTROL
Neighbors: 3 5 6
Node ID: 1
Node State: OBJECT
Neighbors: 3 2 5
Node ID: 2
Node State: OBSTACLE
Neighbors: 4 1 5
Node ID: 3
Node State: CLEAR
Neighbors: 0 1 4 6
Node ID: 4
Node State: CLEAR
Neighbors: 2 3
Node ID: 5
Node State: CLEAR
Neighbors: 0 2 1 6
Node ID: 6
Node State: CLEAR
Neighbors: 3 5 0
Object Count: 1
Obstacle Count: 1
Clear Count: 5
ROBOT 0 CREATED.
ROBOT 1 CREATED.
ROBOT 2 CREATED.
Start Mapping...
On node: 0
Control Node neighbors: 3 5 6
Pushing node0 to explored map
Pushing node 3 to explored map

```
Pushing node 5 to explored map
Pushing node 6 to explored map
-----------------------------------
Current Robot: 0
rand neighbor selected: 6
Node ID: 6
Node State: CLEAR
Neighbors: 3 5 0
added 6 <-> 3
added 6 <-> 5
Size of explored: 4
Size of visited: 2
-----------------------------------
Current Robot: 1
rand neighbor selected: 5
Node ID: 5
Node State: CLEAR
Neighbors: 0 2 1 6
Pushing node 2 to explored map
added 5 <-> 2
Pushing node 1 to explored map
added 5 <-> 1
Size of explored: 6
Size of visited: 3
-----------------------------------
Current Robot: 2
rand neighbor selected: 3
Node ID: 3
Node State: CLEAR
Neighbors: 0 1 4 6
added 3 <-> 1
Pushing node 4 to explored map
added 3 <-> 4
Size of explored: 7
Size of visited: 4
-----------------------------------
Current Robot: 0
All possible neighbor nodes have been visited.
-----------------------------------
Size of explored: 7
Size of visited: 4
-----------------------------------
Current Robot: 1
rand neighbor selected: 2
```

```
Node ID: 2
Node State: OBSTACLE
Neighbors: 4 1 5
added 2 <-> 4
added 2 <-> 1
Size of explored: 7
Size of visited: 5
-----------------------------------
Current Robot: 2
rand neighbor selected: 1
Node ID: 1
Node State: OBJECT
Neighbors: 3 2 5
Size of explored: 7
Size of visited: 6
-----------------------------------
Current Robot: 0
All possible neighbor nodes have been visited.
-----------------------------------
Size of explored: 7
Size of visited: 6
-----------------------------------
Current Robot: 1
rand neighbor selected: 4
Node ID: 4
Node State: CLEAR
Neighbors: 2 3
Size of explored: 7
Size of visited: 7
all nodes have been visited
-----------------------------------
Printing Explored Graph...
-------------------------------------
Node ID: 0
Node State: CONTROL
Neighbors: 3 5 6
-------------------------------------
Node ID: 3
Node State: CLEAR
Neighbors: 0 6 1 4
-------------------------------------
Node ID: 5
Node State: CLEAR
Neighbors: 0 6 2 1
```



6. Discussion
7. Conclusion
8. References
    a. Appendix A: Code

```
void Controller::map() {
    createRobots(numRobots);
    std::cout << "-----------------------------------" <<
    std::cout << "Start Mapping..." << std::endl;
    mapping = true;
    std::cout << "Control Node neighbors: ";
    for (size_t j = 0; j < graphMap->controlPoint->neighbo
        std::cout << graphMap->controlPoint->neighbors[j]
    }
    std::cout << std::endl;
    int count = 0;
    while (mapping) {
    if (count == 3) return;
    for (size_t i=0; i < robots.size(); i++) {
        std::cout << "-----------------------------------
        std::cout << "Current Robot: " << i << "\\n";
```

```
            updateExploredMap(robots[i]);
            robots[i]->move(&visitedNodes);
        }
        // Mapping Stop Condition
        // when all explored nodes have been visited
        if (visitedNodes.size() == exploredMap->nodes.size())
            std::cout <<  "All explored nodes have been visite
            return;
        }
        //count++;
    }
```

```
void Controller::updateExploredMap(Robot* currRobot){
    GraphNode* currRealNode = currRobot->currNode; // actual

    // check to see if currentRealNode has been visited
    bool hasVisited = isVisited(currRealNode);

    // if the currRealNode has been visited
    if (hasVisited) {
        return;
    }
    else { // if the current real node has NOT been visited
        visitedNodes.push_back(currRealNode); // set currReal

        GraphNode* currNodeCopy = new GraphNode(currRealNode

        bool hasExploredNode = isExplored(currRealNode);

        if (!hasExploredNode) {
            exploredMap->nodes.push_back(currNodeCopy);
        }
        else {
            free(currNodeCopy);
            return;
```

```
        }

        for (size_t i=0; i<currRealNode->neighbors.size(); i+

            GraphNode* currRealNeighbor = currRealNode->neigh
            bool hasExplored = isExplored(currRealNeighbor);

            if (hasExplored) { // if the currNeighbor has bee
                GraphNode* currNeighborCopy = findExploredNod
                currNeighborCopy->neighbors.push_back(currNod
                currNodeCopy->neighbors.push_back(currNeighbo
            }
            else { // if the current neighbor has not been ex
                GraphNode* currNeighborCopy = new GraphNode(c
                exploredMap->nodes.push_back(currNeighborCopy
                currNodeCopy->neighbors.push_back(currNeighbo
                currNeighborCopy->neighbors.push_back(currNod
            }
        }
    }
}
```

b. Appendix B: Sources