Name: Brian Jon Stout
Date: 25MAR2017
Current Module: Capstone 3
Project Name: Mining

Project Goals

The goal of the project is, given a program that generates a map of unknown contents, size, and dimension, create a basic virtual robot that will efficiently explore the map making a single movement every 'tick,' or simulation cycle, (much like a video games.) The goal of the exploration is to find minerals, mine them, and then return them before the program terminates.

Considerations:

- o The provided driver program must be able to substantiate an overlord object by using "from mining import overlord."
- o The drone and overlord objects do not receive any information about the map, except, a object called context, which provides the drone it's current location, and what's surrounding it
- o There will be a limited number of ticks to explore the maps entirely, so efficient exploration is important
- o There will be acid on the map, which does 3 damage of 40 to your drone every time it occupies an acid block
- o The overlord must interface with the map object using its action method, which returns a string giving instructions
- o drones must interface through the map object using its move method, which returns a direction it is moving in.
- o The overlord must make a move in a second, and a drone must make a move in a millisecond or there action/move command will be skipped for a tick
- o A drone must be returned to it's overlord to collect the minerals it has found

Initial Design:

Because of the requirements, all the classes and modules are placed in a mining folder. Inside the mining folder is an __init__.py which imports the Overlord class from mining.zerg. The purpose of this is so that the top level program can be used to import Overlord from the mining module, as opposed to mining.zerg, fulfilling a requirement.

Inside the mining folder are three python files. Zerg.py has both the Drone, and Overlord object classes. Graph.py has the Vertex, and Graph classes. Astar.py contains a priority queue class, and three functions called, heuristic, a_star_search, and return_unvisited.

The Overlord object is responsible for deploying, returning, and giving the drone orders to explore its map.

The Drone object is responsible for following instructions, and when not given instructions using very simple logic to explore the map.

The Vertex object is responsible for representing a single block of the map, as well as the connections to its neighbors.

The Graph object is responsible storing data on all the Vertexes, and creating edge connections for them. The Graph object stores known information on the explore map as we explore it, and is used with the a_star_search function to create paths from one point to another.

The heuristic function is used by a_star_search as a weight calculation. Right now it's just a simple distance formula.

The a_star_search algorithm calculates a route from a starting vertex to an end vertex.

The return_unvisited algorithm returns the first vertex it finds that has not been visited by a drone, and returns it. This is used to aid in map exploration

Data Flow:

Our part in this program starts with the driver program initializing an overlord object. The overlord is passed the number of ticks being used by the program. When first initialized the overlord creates three empty dictionaries called maps, zerg, and graphs, then creates two empty lists called zergDropList, and zergReturnList. Afterwards, it sets the variables ticks, and ticksLeft, to the passed in number of ticks when it was first initialized. An internal boolean variable called noMoreDeployment is set to false.

At some point during the driver program, functions will call the overlords add_map method which adds the provided map object to the overlord's map dictionary.

After this, a size count loop creates six, adding them to the zerg dictionary and drop list. During this loop the drones are assigned a mapId (the map they will be deployed to.)

When the overlord has been initialized, the main method for the overlord gets called every program cycle (tick.) this method is called action.

The first thing action does is decrement the ticksleft. It then steps into its method called check_for_wallMode(). Check_for_wallMode is only responsible for making sure every drone isn't still trying to find a wall after 30 ticks have passed.

The next thing action does is the check_to_return() method. This method cycles through every drone and determines if it's time to send it back to the deployment sight, or recall all the drones entirely because the program is about to end. It does this by calculating the distance of the drone from it's home, and setting the drone's returnMode attribute to True if the distance is greater than the amount of ticks left, minus the dimensions of the map. The method also checks to see if the drone has 10 or more minerals, and if the graph the zerg is on is fully visited. In both cases the overlord sends the drone home.

After this, the next part is the main logic for the overlord, controlling the deployments and returns of drones. First it checks to see if any drones were placed in the return list, drones are placed in the return list when they are in return mode, and call up to the overlord because they are in the deployment area. If there are drones in the return list, the overlord pops the first one from the list, and returns a string saying ‘RETURN {the drone’s id}’ which the driver program uses to communicate with the map. If there are enough ticks remaining, the drone is added back to the deployment list.

Next, the overlord checks to see if there are any drones in the zergDropList. If there are, the first drone from the list is popped off. The overlord attempts to get the graph for the map the drone is set to be deployed to. If it does, it assigns the graph to drone, if it can not, the overlord creates a new graph for that map. When deployed the drone’s collected minerals are set to 0, and its returnMode is set to false. If the map it was assigned has already been fully explored, the overlord calls it’s change_map_id function.

The change_map_id function is responsible for finding a drone a non completed map, and changing all the appropriate variables so the drone can function on that map.

After all this, the overlord steps in to it’s default actions, which if all the drones are returning is to randomly attempt to return a drone, else it randomly attempts to deploy a drone. The purpose of this is for some reason the program errors out, and a drone has not been deployed/returned, but is no longer on the deploy/return list, the overlord will still eventually deploy/return that drone.

The drone class creates a dumb virtual robot that explores the map. When the drone is first initialized it gives itself a unique Id among drones. The point of this id is to determine which direction it goes to find a wall at the start of the program by whether it’s Id is odd or even.

An empty list called instructionQueue is created. It’s mapId is defaulted to 0, but will be changed by the overlord quickly. The overlord which created the drone is assigned to the overlord variable so the drone can communicate with it’s creator. The drones location and home (the deployment area) are created and default to tuples of (0,0). The returnMode defaults as false, the hp defaults to 40, and the minerals collected default to 0. At the start a drone has no graph, and wallMode is set to True.

After being initialized the main method for the drone is called every tick, which is the move method.   Every method call the drone is passed a context object, which contains the current block’s x and y coordinate, as well as the symbol for all the neighboring blocks.

The first thing the drone does, is if it's home hasn't been set yet (is 0,0 which is an impossible block to occupy.) it sets it's home to the current blocks x and y coordinate, because which having moved yet, the current location is the deployment site.

The second thing the drone does is update the graph. It calls on the graph's add_edge method, which if the coordinates for the vertex have not been registered, it registers them as well as creating the edge data between all the blocks neighbors. It also sets the vertex to visited because a drone is sitting on it and has been visited.

The next thing a drone does is update it's internal location.

After all this the drone steps into its main logic, which is a series of method calls ordered by priority which determines the direction the drone is going to take. First method call is to focus on minerals. The drone only does this if all the drones are not being recalled by the overlord. This method simply finds the closest mineral to it, and mines it each turn. If there are no neighboring minerals, the method returns no direction so the next one is called.

The next method is the return_home method. This method is only run if the drone's returnMode attribute is set to true, when it is, the method determines a movement that will get the drone closer to the deployment zone. The return_home method does this using the follow_instructions method. First the drone checks to see if it's at home, if it is then it calls up to the overlord to be picked up, if it isn't it sees if there's instructions in the instruction queue. If there is it follows them, if there isn't, the drone requests a route back to the overlord. Upon receiving a route it follows it.

The next method is the find_wall method. This method is only ran if a drone's wallMode is true. When it is, this method just moves the drone west if it has an even id, and east if it has an odd id. The purpose of this method is to separate the two drones at the start of the map to aid in efficient exploration. After a drone has found a wall, it sets wallMode to false.

After find_walll, the drone calls the uncover_neighbor method. This method is a simple check to see if any of the drones current neighbors has been unvisited, if it hasn't been visited, the drone moves on to it.

If the drone has no unvisited neighbors, the drone calls up to the overlord for a route to an unvisited node. This method is called visit_unvisited, and works very similarly to the return_home method, just instead of returning home it moves to a given unvisited node.

Lastly, as a default the drone will either attempt to move to home, or move in a random direction depending on the drone's return mode. The point of this is so that if for any reason all the other methods return no direction, or center for various reasons, the drone will move randomly in attempt to fix itself since it's likely stuck in an obstacle, or against another drone.

When the visit_unvisited method is called by the drone, it asks the overlord for a route to an unvisited node. The overlord provides this route using the return_unvisited function in the astar module. This function is essentially a breadth first search checking for unvisited nodes originating from the drone's location. If no unvisited nodes exist in the graph at all, the graph is set to complete. The overlord then grabs the unvisited node, and uses it to create a route from the drone's location using the a_star_search function.

The a_star_search function is responsible for returning a route from the drone's location to the specified location. It's similar to Dijkstra's algorithm except it uses a heuristic to add an additional weight to each movement. The algorithm additionally determines the weight of a block using a drone's HP, since if it's low enough then the drone shouldn't be stepping on to acid blocks

Communication Protocol:

Not applicable.

Potential Pitfalls:

Overly complicated drone logic causing skipped turns. I attempted to make the drone logic as simple as possible by making the overlord do the complicated stuff. Unfortunately since I didn't have a queue of requests by the drones to process every turn, the drone most likely has to wait for the overlord to finish it's instructions regardless to move on, so it might end up being for nothing.

The drones still sometimes run over acid pits when their health is low enough for some reason. I don't really understand it, so drones can die without returning their minerals. This is some what alleviated by drones returning minerals often

Test Plan:

User Test:

Run through the map of different dimensions and tick sizes

Test Cases:

The program no longer crashes and about 90% of the time the drones return all the minerals on the map. On a 10 by 5 map the drones only need about 100 ticks to complete it, 500 ticks for a 20 by 10 map, and 1000 ticks for a 40 by 20 map. This was a lot more efficient when the amount of minerals on each map was known, since when the map is fully mined the drones move on to the next one, but since this apparently can't be guaranteed without exploring the map entirely anyway, this only happens have a map has been fully explored limiting the time unexplored maps spend with extra drones.

Conclusion:

This project was an odd one. Wrapping my head around the tick simulations wasn't too difficult since I've had experience with coding for video games, and video games use multiple cycles for graphics, or physics simulations. The annoying part was trying to figure out how to explore a map that we weren't sure existed at all. This caused the main issue of moving drones and determining if certain moves were successful or not. If a move isn't successful we have to assume it was until we get information to the contrary. This can offset drones from instructions or throw off things like the mineral collected accounts. There was some issues where if two drones were mining the same node, sometimes minerals would count as being collected or vice versa. There was several instances where the program wasn't stopping because it had 133 minerals collected, but when the drone finally returned to the deployment area at the end of the program, it returned all 135 minerals collected.

Overall it seemed like this exercise was designed to you use to doing some brain gymnastics to work around someone else's API, given that this one seemed overly complicated, and not documented at all.