Vacworld Report

Team

Claire Johnson - 21202262 Maria Leech - 19210540

The solution that we implemented for the VacBot world includes a dust spotter bot ("Bidder") and three worker bots ("Duster"). It operates a bidding system for dust cleaning duties along with random dust cleaning. The Bidder, Decco, is controlled via the Bidder.astra file, and the Duster bots are controlled via the Duster.astra file.

Communication between the bidder agent and the duster agents is based on Agent Communication Language (ACL). It is used by Decco to inform the dusters that dust has been found—via the goal +!callDistances()—and the dusters, constantly monitoring for calls via the goal +!monitorforCalls(), subsequently accept or reject Decco's proposal depending upon their distance from the dust.

When Decco finds a dusty square when moving around (rule +!moveAround() which continuously calls the rule +!moveRandom()), it begins an auction to tell the other bot's dust has been found. A message—FIPA ACL Performative 'inform'—is sent to the bots Henry, Harry, and Lloyd informing them "Dust found!". They in turn respond with a bid (their distances) to determine the winner of the auction. This distance is computed via the rule +!generate_distance(int z, int w, int x, int y, int d) utilizing the classic distance formula, where z and w are the coordinates of Decco's current location, x and y are the specific duster's current location, and d is the answer sent back to Decco. Only dusters that are not actively +seeking(int x, int y) an auctioned square (coordinates x and y) can participate in the auction and send their distance. To prevent the dusters from moving around while !callDistances() is being executed—hence, to enable the proper sending and receiving of location perceptions—the belief monitoring() is updated appropriately: +monitoring(), activated upon receiving the 'inform' message, prevents the movement body of !monitorforCalls() from executing. The monitoring() belief is removed once accept- and reject-proposal messages are conducted, and the dusters' movements continue.

After its critical sleep cycle, Decco compares all of the bids received from ~seeking() dusters via the rule +!compareBids(current(string botname, int botdistance), winner(string winner, int best_bid), boolean better), where better is edited as 'true' or 'false' for each duster if their botdistance is less than the best_bid. (Note: best_bid = 60 and winner = "Decco" initially for every auction, with a distance > 60 considered impractical for a duster to move, since it helps to have vacuums spread out across the grid). Decco determines the ultimate winner if 'better == true' for that duster and notifies the dusters if they have won ('send(accept-proposal…)') or lost ('send(reject-proposal…)') via the rule +!winner(string winner, int best_bid). The bot with the lowest bid wins, adds the +seeking() belief, turns on its light, and navigates to the dust square to clean it. Below is an example exchange of the messages that occur when an auction is being held.

```
[Decco]Received mydistance: 32 from: Lloyd
[Decco]Received mydistance: 34 from: Henry
[Decco]Received mydistance: 13 from: Harry
[Decco]Evaluating bids
[Decco]Comparing bids
[Decco]Decco my distance:60 against Decco best:60
[Decco]Comparing bids
[Decco]Harry my distance:13 against Decco best:60
[Decco]Comparing bids
[Decco]Henry my distance:34 against Harry best:13
[Decco]Comparing bids
[Decco]Lloyd my distance:32 against Harry best:13
[Decco]Calculating winner: Harry
[Decco]I'm not a winner Decco
[Decco]I'm the winner! Harry
[Decco]Winner: Harry with distance: 13
[Decco]I'm not a winner Henry
[Decco]I'm not a winner Lloyd
[Harry]Seeking: 6, 1
[Harry]Moving 2 north
[Harry]Moving -3 east
[Harry]Do I 0 = 0?
[Harry]Do I 0 = 0?
```

The dusters' have two movement rules depending upon if they are +seeking() or ~seeking(). +seeking() movement bases itself on the duster's current location coordinates and the location Decco sent out, stored in the seeking() belief during !monitorforCalls(). This rule instructs the bots on whether to move north or south (Ydelta) and/or east or west (Xdelta) to reach the dusty square. The winning duster will continue to navigate its deltas until both Ydelta == 0 and Xdelta == 0 (see the above printout "Do I 0 = 0?" for Xdelta and Ydelta), meaning it has reached the appropriate location and can clean that square, afterwards turning off its light and entering ~seeking() movement. An integer i = 0 is incremented for every iteration through the navigation while loop, where Xdelta and Ydelta are updated depending on the duster's newest location; if i = 10, the while loop automatically quits and the duster exits seeking and turns off its light.

Meanwhile, the rejected dusters continue to randomly clean squares until another dust square is identified by Decco and the auction cycle repeats. These dusters prioritize squares containing dust to the right, left, or forward of its location, improving the efficiency of clearing the grid. Otherwise, when surrounded by empty squares, they utilize random movement via random number generation in Random.java, where Math.random() * 75 bounds the available movements. A random number between 0 and 41 dictates forward movement, 42 and 58 rightward movement, and 59 and 75 leftward movement. Forward movement is slightly favoured such that newer regions of the grid can be found more expediently (i.e., it is less likely for the vacuum to move circularly). For dusters, !monitorforCalls() contains an endless loop for movement (if ~monitoring()), where the !move() rule chosen depends on the seeking() belief.

Decco additionally utilizes this random movement via its rule +!moveRandom(), where if here("dust") then !callDistances() is instantiated; however, forward, right, and left movement have equal probability. !moveRandom() is called within the rule +!moveAround() (the initial main goal of Decco)—a rule that recursively calls itself for the entirety of the running program. In order to cover as much of the grid as possible, Decco will attempt to exit a region after 10 !moveRandom() calls, moving east, then south, then west, then north, each after 10 movement iterations. These directional movements are only conducted if Decco is not immediately surrounded by any dust to prevent missing a square. They help to ensure Decco does not spend too long in any one area where most dust has been found and cleaned. Additionally, it can redistribute the dusters to new dust hotspots. Decco will try to move

east/west 7 squares and north/south 5 squares, determined based on the relative dimensions of the grid.

Despite showing coordination and relatively efficient timing (averaging ~5-7 minutes for a totally cleaned grid), this project does not come without areas for improvement. One issue is that when a duster is seeking, they can become stuck behind an obstacle in its Xdelta/Ydelta path. To overcome this, we implemented the integer (i < 10) loop boundary or 'force quit': once reached, even if the dust has not been cleaned, the duster will return to random movement and cleaning. The square that the duster was unable to reach can then be auctioned off again to be cleaned. Another issue we call 'overshoot,' where the dusters can occasionally overshoot the auctioned square (this can be seen in the video recording with Lloyd near the end). This is due to the duration of sleep-cycles and timed updating of perceptions across the entire agent system: it is our belief that perceptions sometimes are not updated with enough time for other agents to register those changes. To minimize 'overshoot,' we have placed the following chunk of code across our files, particularly before querying current_loc() and between movements:

```
while (~task("none")) { S.sleep(500) } // sometimes 100
```

The motivation for ~task("none") is because "The location, direction and square percepts are only updated when the VacBot is not currently moving, turning, or cleaning," as stated in The Vacuum World GitHub. In addition, S.sleep(int) are placed throughout our code to enable @messages to be received and registered, and give perceptions time to update. Another issue is that Decco can randomly circulate back to an auctioned square before the appropriate duster has reached that location and cleaned it, hence starting a new, unnecessary bid. To remedy this, a short-term memory could be implemented for Decco wherein it does not re-auction the most recently auctioned square.

Vacbot world is set in an Environment Interface Standard (EIS) environment. Both bidder and duster bots connect to this environment in their main rule. Custom EIS events are included in each of the Bidder (line 250) and Duster files (line 233). The addition of these allows us to trigger behaviour based on the addition or removal of a percept. For example, below is a Custom EIS event from Bidder.astra (line 250):

```
rule +$ei.event(task("move")) { -+task("move"); }
```

This solution averaged at about 6 minutes per run to clean the entire grid. Without the coordination, it could far surpass 10 minutes to clean. Efficiency was improved via a few core changes: vacuums favour movement to squares containing dust (enabling dust clusters to be quickly swept and removing the need for vacuums to land directly on a square with dust (they can land within one square on any side)), dusters favour forward moves slightly more than turns (reducing circularity), Decco will move continuously in a random direction after so many random moves when its surroundings are empty (enabling discovery of new dust hotspots), and the seeking() belief (enables dusters to continue to move and clean randomly outside of auctions).

Future work for this solution would be a storage mechanism for the squares visited per vacuum. This would be the best method for preventing vacuums from remaining in one region of the grid for too long, allowing hard-to-find dust to be located. Additionally, it must be noted that our implemented solution is not the most efficient when cleaning has no time penalty associated with it (i.e., Decco cleaning the dust spot itself is faster than calling for help). Dividing the grid into regions, each handled by one vacuum, would likely prove more efficient.