**AI Project: Results and Conclusions**

Robert Scollon

Nicholas Horsman

Anthony Kniazewycz

Leonard Garrison

*Part 1:*

The initial changes made to the modified LearnerOne agent included adding the possibilities for the vacuum to turn off, as well as enter different power states. This also included implementing a chance for the vacuum to change its action even when the state of the game has not changed. This was simply done with a counter set to a large value. Below are the results for both LearnerOne and our agent:

|  |  |  |  |
| --- | --- | --- | --- |
| Agent | Bugs Captured | Bugs Escaped | Success Ratio |
| *Learner one* | 63 | 57 | 0.525 |
| *Our Agent* | 171 | 142 | 0.546 |
| *Our Agent w/ Rewards* | 275 | 47 | 0.859 |

Though our agent does outperform the base LearnerOne agent, capturing over two times as many bugs as LearnerOne, its efficiency at capturing bugs is nearly identical. Given more time, both agents could potentially have improved the number of bug captured; however, their overall performance would not have improved over time due to the implemented reward system.

The next changes made were to the reward system. We changed the reward received when a bug was captured to be determined by the vacuum’s suction level, with lower levels being rewarded more. We also rewarded the agent for turning the vacuum off when no bugs were within the checked radius. It was also at this point we noticed that even with the rewarding of whether or not a bug was in the area that the vacuum would decide to turn on and wait for another bug to create a state change. To challenge this we also checked for no bugs between states. This improved the resource usage of the agent. The results of the agent at that stage can be seen in the previous table.

*Part 2:*

When implementing the tiebreaking between previous action and the next action, we would compare the previous action’s utility value to the rest of the utility values from that state; if it was the highest and was not zero, then the previous action would be taken. We also implemented Q-Learning by waiting to reward actions until the agent entered the next state. We had an instance of the previous state we can update from the current state, so this made Q-Learning easier to implement because we had access to both the rewards from each state. We saw an improvement of about 350 bugs captured on average, with 100 escaping on average.

*Part 3:*

To complete the directed reasoning, we modified StateVector by adding a function that would return the CellContentsCode. With the code, we were able to generate new states that are similar to our current one. As an example, if there were 3 bugs nearby, we would use the CellContentsCode to generate an array that can be represent each 3 bugs as 3 states that could be compared to already defined states, and finding those best actions to take. We continued to modify CellContents by adding four new functions that would let the agent know if there were any bugs in a specific direction. We then used these functions to prioritize the 8 different directions when rewarding the agent's action.

More was planned for this stage of programming such as detecting different bugs and doing different things based on it but as we continued we realized that there are a lot of variables that change between games that we could not keep up with. This stopped up from making further improvements.