Hyunjin Lee 2020320023

Professor Hyunwoo Kim

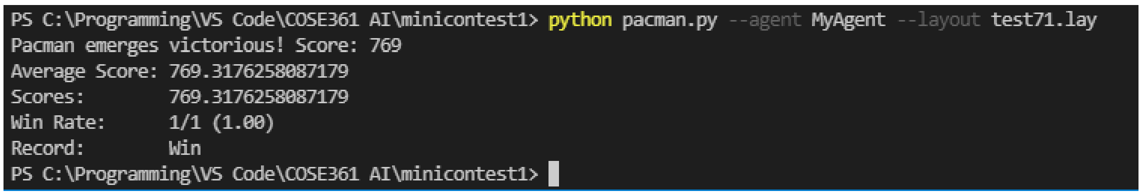
COSE361-03

27 May 2021

Final 1/2

Development Environment: Visual Studio Code

1) Result:



```
PS C:\Programming\VS Code\COSE361 AI\minicontest1> python pacman.py --agent MyAgent --layout test71.lay
Pacman emerges victorious! Score: 769
Average Score: 769.3176258087179
Scores:        769.3176258087179
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Programming\VS Code\COSE361 AI\minicontest1>
```

1) Code Explanations: (also commented in the python file)

The algorithm I have implemented for this assignment is mostly same as the one I have implemented on assignment one. I utilized BFS search with queue to find the optimal route to the nearest food for each agent. (BFS explanation: Starting from the start state, enqueue the unvisited successor nodes. Dequeue and save the paths to action array. Repeat until a food has been reached and dequeued.) I have also modified and added implementations to the code to make the agent more efficient.

For example, I have initialized a dictionary to keep track of which food an agent is currently heading to. This prevents multiple agents heading toward the same food. However, if I simply force the agents to head to different food, there may be situations where an agent is not eating the food close to it because the food has been targeted by an agent far from the food. To prevent this from happening, I have allowed the agents to target the same food if the current agent is closer to the targeted food by a margin of 6.

Also, because computation time is extremely important for the final score, I have removed overhead from the algorithm. If a path has been found for an agent, instead of running the BFS every time the agent moves, simply save the path, and follow the path until the food has been reached. Only then, run the BFS search again. This is achieved through curPath dictionary. Additionally, if there is no food near an agent compared to other agents, there is no need for that agent to run the BFS search every time. Simply add the agent index to a stop array and return 'Stop' action instead of running the BFS search.

2) Discussions:

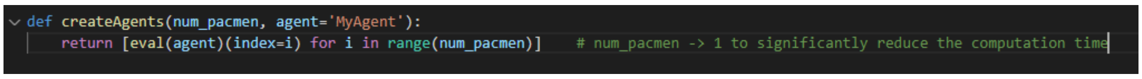
1. My agent is better than the baseline in both default layout and test71 layout. This is because my agent attempt to target different food instead of targeting the food randomly.

Also, my agent saves the path found from the BFS search and run the path until a food has been found. On the other hand, the baseline agent runs the search every time an agent moves. Thus, my agent is able to score higher because it is able to eat all the foods faster and have less computation time for each search on average.

2. There may be cases where my agent performs worse than the baseline agent. My agent has been instructed to stop the search if it determines the current agent is too far from other food compared to other agents. This is applicable for most cases. However, if the layout is big and complex enough, my agent may perform worse. To be more specific, if an agent stops searching because it is far from the food, but the layout is complex enough that it is more efficient for all the agents to search together, my agent will take longer time to eat all the food; thus, resulting in worse score.

3. What are the tradeoffs between computation time and the time it takes to eat all the food? Which one should I prioritize more?

I believe shorter computation time is more important for this specific question. Because the search is conducted multiple times for every agent, and -1000 * computation time is added to the total score for each action, prioritizing the computation time seems more beneficial. Accordingly, instead of running multiple Pac-men, simply running one Pac-men seemed to result higher scores on average. This can be implemented by changing num_pacmen to 1 on line 27.



```python
def createAgents(num_pacmen, agent='MyAgent'):
    return [eval(agent)(index=i) for i in range(num_pacmen)]    # num_pacmen -> 1 to significantly reduce the computation time
```