AI Report 1900721

My part 1 approach tries to find a path to the requested target within the energy it has left. Failing that, it looks for a series of unique charge points to travel to until it can reach the target. This is a strong approach for the grid, letting it find any possible path, however would only work in real world scenarios where the state is both known and static.

Furthermore, it could be improved by taking note of any oracles or chargers found while looking for the goal, as it could then use faster A* pathfinding back there should the need arise.

The approach to part 2 uses a greedy algorithm to try and find the best possible move at each juncture. The agent assumes it can certainly reach a charger. It searches for the closest unqueried oracle. From this oracle, it attempts to find a charger, such that the agent would have enough energy to reach it after travelling to and querying the oracle. If this succeeds, the agent travels to the oracle and queries it. The agent can now certainly reach a charger, otherwise it would not have moved here. Therefore, the assumption holds.

If the agent cannot find such an oracle, it recharges, then starts again.

This is a very strong approach, allowing the agent to consistently check all oracles on the board, often in a close-to optimal travel order. As it does not plan its path past one oracle, it would still work in real-world scenarios where oracles are not always static (perhaps a robot waiter serving drinks at a party).

Where charge points (drink re-fillers) to move as well, it may path to an oracle under the assumption it will have just enough charge to reach a nearby charger, which then moves, stranding the agent. This could be remedied by ensuring an “emergency” pool of energy is kept, unused path calculations. Should a charger move, this pool can be expended to catch up to them. The size of this pool would thus be based off the map size and speed of the chargers and would be found through empirical experiments.

It would be neigh impossible to modify this solution to a case where the map is unknown *and* non-static; this would require a brand-new solution.

Were the map known and static, the agent could use algorithms like K-means clustering to find and query groups of oracles at a time, running some greedy solution to the travelling salesman problem taking charging into account, although this may take longer to compute, and wouldn’t generalise to many real world scenarios, as they tend to be messier (unstatic).

Part 3 uses many agents working utilizing finite state machines to control their behaviour. These agents share a blackboard and are governed by a manager.

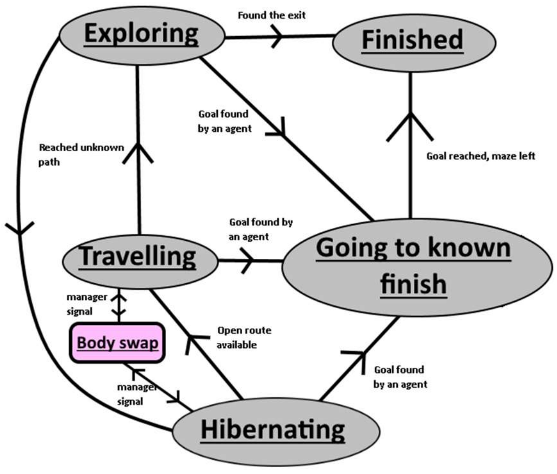

Figure 1: Finite state machine of AI agents

The maze is considered a series of junctions, with one path between them all. As agents travel explore between junctions, they remember the path they travelled, and once a new junction is discovered, they add this path between the known junctions to the shared blackboard, the path from their origin (in junctions crossed), and all open routes to travel off this intersection. This always lets the most agents actively look for the exit.

As there is only one path between each junction and all agents are given the same start junction, when an agent enters seeks a path, the pathfinding is blazingly quick for any known map position, as they just look for the last shared junction between themselves and the junction-based path to the target and append the remembered paths.

The open (unexplored) routes in the global blackboard are sorted by their Manhattan distance from the goal, allowing the agents to choose the unknown path that most likely approaches the exit of the maze. In real world scenarios where the goal's location is unknown, a BFS search would have to be used from the agents origin. If there are no open routes, the agent hibernates until either a new one is discovered, or the exit is found. This could be improved by making sure the closest hibernating agent takes a new open route, as currently it is mostly random, and making sure open agents move towards currently traveling agents.

The manager looks for two agents blocking each other's paths, and swaps their goals with each other, letting them both continue.

The manager could be improved to catch these collisions well ahead of time, swapping the agents goals without the wasted travel time. Furthermore, rather than agents naively choosing the closest-to-goal junction to explore, the manager could select paths for agents with a heuristic based off both distance to the goal, *and* distance from the agent. This again, reduces unnecessary travel time.

It could also recognise when an unknown area that does not contain the goal has been enclosed by known areas, and therefore that space should not be searched, although the computation time for this each step would be heavy.

Were long range communication not possible (battery constraints, or the environment blocks communication, such as a cave), the global blackboard could be replaces by a local blackboards. When agents are adjacent to each other, they could update each other on known information using shorter range communication. Having the agent swarm acting like a breadth first search, going out to explore a unique open path, then meeting together again at the start and share information would create a similar situation to part 3's maze. The downside would be that a lot more time would be spent on travelling. Having agents move in groups, with taskless agents following exploring agents to take a newly discovered path could reduce this excess time. Another way to cut down on it would be letting agents travel down multiple adjacent paths before returning to exchange information. The exact limits on this would depend on the situation, and would have to be found through testing.