Part One: Breadth-First Search shortest path finder

The goal of Part One was to have an agent efficiently achieve a task of the form `go(?Pos)` or `find(?Obj)`, by minimizing the number of moves made and runtime. I achieved this by implementing a breadth-first search (BFS) algorithm. The benefits of using a BFS is that it is complete, meaning that if a solution exists (for example a route to a cell on the board when the task is `go(p(X,Y))`) then the BFS algorithm will find it. Additionally, the BFS algorithm always finds the shortest possible solution, which is ideal when the primary objective is minimizing moves made. The runtime of the BFS algorithm is minimized by adapting it into an A* algorithm, by assigning a score to each cell comprised of its Manhattan distance from the target and its distance from the root node. By prioritising lowest scoring cells when expanding the search, the first complete solution found is guaranteed to also be the shortest solution, and further recursion of the algorithm is not required, thus reducing the overall runtime of the algorithm. The BFS algorithm also stores a list of `Visited` nodes, which can be subsequently ignored if they are selected due to being adjacent to a different node. Because each cell can have up to 4 adjacent cells, storing Visited cells can mitigate a large amount of unnecessary duplicate checks, at the expense of requiring a vacant amount of memory to store a list of visited nodes (theoretically up to n^2^ nodes would be stored in an n x n grid).

When performing a `find(?Obj)` task, a standard BFS is used because the location of Object being searched for is unknown. Searching for unknown nodes has a large impact on runtime - searching for a specific Object in an n x n grid has a time complexity of O(n^2^). In part one, the impact on runtime that use of the `find` task had was not massive, because the grids are relatively small at 20 x 20, and typically finished execution when the first instance of an object was found.

In a worst-case scenario, where the BFS algorithm is being implemented to search for a specific object in a much larger grid, the impact on runtime would have polynomial growth, scaling with grid size. In this instance, a potential adaption to the algorithm could be to perform an initial BFS on the entire grid and storing a list of locations and ID's of found objects in memory. Because the location of the `find` target is now known, a Manhattan distance heuristic for the scoring function could be calculated to use an A* search algorithm. While the initial full grid search will be time intensive, you could make that time back because subsequent searches are now optimal. The implementation of this method would have varying impact on total runtime and viability, and it would be best suited to a situation with a very large grid size, when objects are sparse or specific, and there are no memory limitations.

Part Two: Complex route planning

For Part Two, the goal was to have an agent find and navigate to as many oracles as possible, topping up it's energy at charging stations along the way to not run out of energy along the way. The primary aim was to maximise the number of oracles visited, with secondary and tertiary aims of minimising moves and time taken respectively.

In order to achieve the primary aim, the agent must find all accessible oracles, but not run out of energy while navigating. To achieve this, I implemented a greedy approach of locating the nearest unused oracle, using the BFS from Part One. If the route would take the agent under an arbitrary energy threshold (25% of max energy was found to be a suitable value through testing) then the objective was updated to navigate to a charge station, before resuming searching for unused oracles. Consumed oracles were stored in a list to prevent them from being targeted again during execution, which slightly minimises the runtime.

To improve the tertiary aim of minimising runtime, a similar method of initially saving locations of oracles and chargers in memory to then use an A* algorithm (as discussed in Part One) could be implemented.

To achieve the secondary aim of minimising moves taken, an intelligent solution to charging the agent could be used. While the threshold approach used seems to rarely fail, in some edge cases the agent may navigate to an oracle and not have enough energy left over to make it to a charge station afterwards. A forward-looking algorithm that, before any moves are made, ensures that the agent would have enough energy left over to proceed to the nearest charge station after reaching the oracle would minimise the amount of unnecessary charge station detours taken and remove the risk of an agent becoming stranded. This approach would come at the expense of runtime as the number of searches performed per oracle would be effectively doubled.

A forward-looking implementation would be particularly useful in a scenario where the agent is not allowed to run out of energy, for instance, if the agent had to return to the start or a specific location to upload it's findings, otherwise no information is gained. A real-world example of this could be an autonomous Mars rover, where depletion of energy would lead to system failure.

Part Three: Maze solver

Due to time limitations I did not attempt Part Three.