Part 1

Part 1 requires a pathfinding solution to solve and perform tasks in the form of `go(+Pos)` and `find(+Obj)`. I chose to use a Breadth First Search (BFS) to achieve both of these. In the case of `go(+Pos)` tasks an additional heuristic is used, turning it into an A* search. Using BFS ensures an optimal path is found and execution time is optimised by reducing unnecessary `map_adjacent(+Pos,-Adj,-Obj)` calls. Direct paths are calculated initially and if required (i.e. the agent doesn't have enough energy to perform the path) the agent will first path to a charging station and top up before recursively solving the initial task.

A potential optimisation for `go(+Pos)` calls would be to check the agent has enough energy to cover the Manhattan distance before calculating a direct path as that is the minimum energy the agent would require to complete the path. Prefinding objects and storing them could allow `find(+Obj)` tasks to be converted to `go(+Pos)` tasks which would reduce execution time as it would use A* rather than a plain BFS. It's possible the overhead of prefinding objects could outweigh the pathfinding benefits on certain grids and would not work at all if target objects were capable of moving.

Part 2

I chose to use a simple greedy approach to oracle pathfinding in part 2, simply searching for and pathing to the closest unquestioned oracle. In terms of energy management I also went for a simple approach with agents recharging before attempting to find an oracle if they are below 25% of their maximum energy. This threshold was chosen after numerous test runs to strike a balance between overcharging and not charging before running out of energy. In combination these methods work well to allow agents to maximise the number of oracles visited while keeping execution time low. Problems with my solution will likely start to appear at higher grid sizes where it becomes more likely that the agent puts itself in a position where it is very far from the next oracle.

There are a few options for improvement here. One would be to precompute a graph representing the paths between oracles and approximate a solution to the Traveling Salesman Problem. However I felt that the benefits in terms of reduced moves and the possibility of reaching more nodes wasn't worth it due to the likely increased execution time and my current approach already working well. Recharging can also be improved, one simple way this could be done is to check if the agent has enough energy to reach a charging station after visiting an oracle before travelling to it. This would increase execution time as two paths would need to be computed each time, but it would mean the agent wouldn't put itself in a position where it can't recharge as well as removing unnecessary recharges.

Part 3

Mazes generated in part 3 are "perfect" mazes meaning they have no unconnected regions and contain no cycles. Such mazes can be represented as a tree allowing agents to perform simple depth first searches, making random choices at intersections. Upon reaching a dead-end agents begin backtracking, marking cells as dead-ends as they go. This means other agents can ignore those cells in their search to prevent re-exploration of invalid paths. Once any agent has reached the end of the maze all other agents begin using the A* implementation of part 1 to path to the end.

My part 3 solution has been successful on all mazes I've tried however with increased agents so called traffic jams become common with large groups of agents in one place taking multiple turns to get moving. This is something that can likely be reduced by careful ordering of moves however I felt that it was too complex a problem to tackle and unnecessary considering the agents always began moving fairly quickly. There is also a problem with my path to end solution, paths are recalculated for each agent once an agent reaches the end. This means in cases where there are many agents far away from the end of the maze there can be periods in which nothing appears to happen. I unfortunately ran out of time to solve this.