**Agent 3 Implementation Description**

Information Representation

When designing the inference agent, the first challenge was how to neatly organize and store information. If we didn’t have a really solid system in place for that, we may waste precious time searching for the information as opposed to having direct access to it.

* For that reason, we created an object of type “CellInfo” that would then store all of the accompanying information for the cell in question. This includes whether the cell has been visited or not, whether the cell has been confirmed to be blocked, empty or is still unconfirmed in its status, the total number of neighbors the cell has, the number of cells around it that are sensed to be blocked, the number of neighboring cells confirmed to be blocked, the number of neighboring cells confirmed to be empty, and the number of neighboring cells still unconfirmed. This corresponds to all the information listed within the example inference agent write-up.
* Then, we created a “Maze” data structure that was essentially a 2D arraylist of type CellInfo that would store these cells, and their coordinate could be directly accessed with something like maze.get(2).get(3) for the coordinate 2, 3 in the maze where 2 is the column number (corresponding to the X axis) and 3 is the row number (corresponding to the Y axis).

Each cell also stored its own g-value, h-value and subsequently f-value. This meant we wouldn’t have to search anywhere but the cell we’re interested in during the planning phase for determining where to go next. And lastly, we had each cell point to its “parent” or the cell that we used to get to this point in the maze. This was mostly in place so that backtracking, if needed, could be implemented a little easier during the planning phase as well.

Workflow

So, now we have pretty efficient access to all the information we need, but how do we go about processing it to our advantage? The workflow for this agent essentially followed the example’s suggestion from the project description. We made an initial plan under the freespace assumption, and then for every cell popped off this planned path, we sensed around us, attempted to infer any new information we could from that, and checked to see if there had been a block discovered in our path (if so we had to replan). If not, we would attempt to “move” to the next cell, and if it was blocked, we would infer anything we could given this update in the knowledge base and then replan. If the cell wasn’t blocked, we would actually physically move into the cell and continue the loop again, popping off another cell from the planned path.

To dig in a level of abstraction deeper, specifically with the actual inferring stage, we had to implement things a certain way to get them to work for us. So, say a cell has been sent to the inference method. Here’s the general logical flow for what would happen next:

* We would first “check surroundings”. This is essentially equivalent to making sure our knowledge base is up-to-date. We look around at all the neighboring cells and update our necessary counts. For example, if a cell had recently been confirmed to be open but to our previous knowledge it was unconfirmed, now we will have an increase in the “confirmed empty neighbors” count and a decrease in the “unconfirmed neighbors” count.
* So, now our knowledge base is up-to-date for the cell we are interested in. This is where we apply the inference rules seen in the project description. Specifically, we checked first to see if Hx wasn’t equal to zero because if it was, we know that there’s nothing left to infer for this cell. If Hx was equal to zero, then we checked if Cx = Bx (a.k.a. if the number of blocks sensed around this cell is equal to the number of neighbors confirmed to be blocked), and if we knew that, then we knew that the remaining unconfirmed neighbors were empty. If not that, then if Nx - Cx = Ex, we knew that the remaining unconfirmed neighbors were blocked.
* With this knowledge, we now “check surroundings” of the neighbors to the current cell. This makes sure any changes to the state of the cell we’re currently in (e.g. if we just confirmed it was empty by moving into it) is propagated to the surrounding cells. Also, if one of the “restEmpty” or “restBlocked” conditions were true, we change the status of all unconfirmed neighbors with respect to the conditions met, and then we add them to a list to be sent into the inferring method recursively. The reason this is done is so that their change in being confirmed can be reflected in its neighbors as well, which might be out-of-reach to the current cell we’re at (at least some of them).
* The last thing we do (this is still in the current stackframe, so after these checks but before sending neighboring cells back into the inference method) is if the neighboring cell was either confirmed or the “restEmpty” or “restBlocked” conditions weren’t meant, we actually do the same inference checks as the one done for the current cell on the neighboring cell as well, and if we find anything of interest, we add that to the list to be sent in recursively. This means any of the meat of the inference method (changing statuses of neighbors, etc.) will be done on the next stackframe up.

With this workflow, we make sure that everything that can be inferred about our immediate surroundings at a given point in the maze is propagated outward. Now, we don’t necessarily scan over the entire maze every time we move into a new cell. If there reaches a radius around the given cell that we can no longer infer anything new, we stop our inference process and move on. This saves computationally in potential wasteful ventures through the entire maze, and we describe more later in the report about how this was an intentional design choice to help us solve the maze and infer appropriate things but in a reasonable amount of time (refer to “Computational Issues”).