Project A
Part 1 Design
Egan and Katrina

We built our functions and classes within the class Mouse to allow for simple importation by another program. Our parsing function takes a file name as input and creates an initial state, which is stored within an object of the class Mouse. We chose to use a 2d array to represent the maze because it allows simpler indexing than a one-dimensional array, thus reducing confusion. It also permits faster editing than a text document. We stored the maze tiles as characters to reduce memory use, and used the same characters as the text file to allow easier printing, as well as comparison for debugging.

Our state representation takes the form of a Python class, with the attributes, location, cheese list, last action taken, and number of steps to reach. We chose to use a class so that all of the attributes could be easily accessed, and to avoid the confusion from separate data structures. We stored the last action taken as a single character to minimize memory use. Location is stored as a tuple to minimize memory use and to make it difficult to edit accidentally. The cheese list is stored as a list because its length varies from problem to problem and changes from state to state. Its contents are also stored as tuples to minimize memory usage.

The transition function takes a current state and a valid action as inputs, and generates the new state resulting from that action. We chose to assume that actions from the action list will be passed to the transition function only if they are possible, but wrote our code to raise an error if an action other than ‘n’, ‘w’, ‘e’, or ‘s’ is passed to it. The transition function also uses deepcopy in order to create copies of the cheese list, to prevent modification of the original state.

We remove an item from the cheese_list every time the Mouse's (X, Y) coordinates are equal to a cheese's (X, Y) coordinates. Our goal_test() function checks our cheese_list, and if cheese_list is empty, we return true, otherwise return false. This allows us to simplify our goal test, and is effective because if no more cheeses remain in the maze, they must all have been picked up.

Part 2

The function single_dfs will first create a Mouse object and use its init_maze method to parse the file which is passed to single_dfs. The function will maintain a stack of unexpanded states, so that the state most recently pushed is returned first. It will initially push the initial state onto the stack. Then it will call goal_test on it. If goal_test returns false, it will expand it, using the expansion function, and replace it with the next state popped from the stack. It will repeat this until a call to goal_test returns true. If a solution is found, the solution path, number of steps taken, and number of nodes expanded will be printed, using create_solution_path.

The expansion function will call generate_acceptable_actions and generate a child node for each returned action. It will then return a list of children.

The function generate_acceptable_actions will use the maze representation to determine which actions are impossible due to walls. It will then return a list of acceptable actions, represented as single characters.

The function create_solution_path will use the maze representation and the parent pointers of each state to trace a path back through the maze, creating a new maze representation which shows the solution. The maze's representation's variable cheese_locations will have the number of cheeses stored in it, allowing each cheese to be represented with the correct number or letter.

Part 3)

single_bfs:

single_bfs will create a queue of states representing all possible paths that are not blocked by walls, and will try all possible paths until the goal is completed. When it dequeues a state, it will expand it, generating its children and queuing them, and it will not call goal_test on a state until the state is expanded.

single_gbfs:

single_gbfs will create a priority queue holding all states which have been generated and not expanded, and will prioritize them based only on how close to the goal they are estimated to be. Nodes with a lower heuristic value will be expanded first, regardless of their cost to reach. It will print results for the first goal it finds, regardless of whether it is optimal. The heuristic function will be a method of the Mouse class which will return a heuristic based on Manhattan distance.

single_astar:

For an A* search, our algorithm will create a priority queue of all states which have been generated but not expanded. It will explore all states in order, prioritizing them by their heuristic value and their cost to reach, and generating a list of solutions as it goes. Then it will choose the solution with the smallest cost. Like greedy best-first search, it will use a heuristic based on Manhattan Distance.

Part 4)

multi_astar:

Essentially the same idea as single_astar, except it prioritizes states by their cost to reach and their estimated heuristic for the collection of all cheeses, eventually printing a result which is the optimal solution for gathering all cheeses. It will have a heuristic function within the Mouse class which is based on Manhattan distance. It will calculate the sum of the Manhattan distances from cheese to cheese for all possible collecting orderings, then return the minimum of them.