Partner: Nobody (for now)

No matter how many classes push it on me im not gonna learn LaTeX

Question 1: Shortest Path Composition (25 points)

Consider a graph $G = (V, E, w : E \to \mathbb{R}^{\geq 0})$ where V is a set of vertices, E is a set of (directed) edges, and w is a *weight function* that maps edges to weights where each weight is ≥ 0. Let us define a path p to be a sequence of edges where the destination vertex of one edge is the source vertex of the next edge (if the next edge exists). Let us define the *cost* of an path the traditional way, i.e. the cost of a path p is the sum of the edge weights in p:

$$cost(p) = \sum_{e \in p} w(e)$$

Show that if we know a shortest path $p^* = a \overset{x}{\rightsquigarrow} b$ from vertex a to vertex b has cost x. If we know p^* passes through intermediary vertex c, then let $p_1 = a \overset{y}{\rightsquigarrow} c$, and let $p_2 = c \overset{z}{\rightsquigarrow} b$. Show that if $p^* = p_1 \cup p_2$, then p_1 is a shortest path from a to c, and that p_2 is a shortest path from c to b.

I will show that if p* = p1 ∪ p2, then p1 is a shortest path from a → c and p2 is a shortest path from c → b via contradiction.

First, let us consider another path q1, which travels from a → c with a cost of $y^i < y$. Then the cost of this new path, p^y, is $y^i + z < y + z$, but this contradicts the assumption that p* is the shortest path from a → b.

Next, let us similarly consider an alternative path q2, which travels from c → b at a cost of $z^i < z$. Then the cost of this new path, p^z, is $y + z^i < y + z$, which again contradicts the assumption that p* is the shortest possible path from a → b.

In conclusion, if it is known that path p* is the shortest path from a → b with intermediate paths a → c and c → b, then we know that there is no alternate intermediate path that can perform better than these shortest paths.

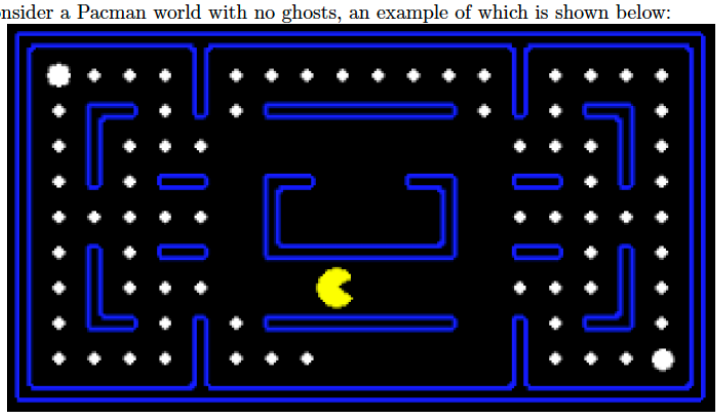
Question 2: Best and Worst Cases for DFS (25 points)

Consider a graph $G = (V, E, w : E \to \mathbb{R}^{\geq 0})$ where V is a set of vertices, E is a set of (directed) edges, and w is a *weight function* that maps edges to weights where each weight is ≥ 0. Let us start a DFS search from vertex a, the goal of which is to find vertex b (where $b \neq a$). In the worst case, where is the goal vertex b in relation to the source vertex a in the DFS expansion. How many vertices and edges must the expansion contain before when we reach b? What about the best case?

In the worst case using DFS, vertex b would be a leaf node furthest away from vertex a as possible and stemming from the last neighbor of a, meaning the algorithm would have to explore every edge, |E|, and vertex, |V|, possible before finding vertex b. In the best case, vertex b is the only neighbor of vertex a, meaning that the algorithm would only take a single computational step before finding and reaching b, looking at a total of 2 vertices and 1 edge (a, b, a → b).

Extra Credit: Heuristics for Pacman-NoGhosts (50 points)

Consider a Pacman world with no ghosts, an example of which is shown below:



Here is the description of this search problem:

- **Environment:** A 2-d map of finite size where each square can contain a wall, a food pellet, Pacman, or is unoccupied.
- **Sensors/State:** The state of the world is a structure containing the following fields (in Java-ish syntax):
  - `current_loc: Coordinate`: The (x,y) location of Pacman in the map.
  - `food_remaining_locs: Collection<Coordinate>`: The (x,y) locations of all remaining food pellets in the map.
- **Initial state:** The initial state is an instance of the State with the following field values:
  - `current_loc = (9, 3); // technically any unoccupied square will do`
  - `food_remaining_locs = locations of all food pellets in the map;`
- **Actuators/Actions:** Pacman can move to an adjacent square in a cardinal direction.
- **Transition Model:** Pacman will move to the adjacent square if it is not a wall and if the action will not take Pacman out of bounds of the map. Additionally, if Pacman enters a square that contains a food pellet, Pacman will automatically consume that pellet. When Pacman consumes a pellet, the location of that pellet is removed from the `food_remaining_locs` of that state.
- **Path cost:** Moving to an adjacent square has a cost of 1. Therefore, the cost of a path is the number of edges in the path.
- **Goal Test:** Any state where the number of remaining pellets is zero is a goal state regardless of the (x,y) position of Pacman.
- **Performance Measure:** The number of turns it takes to eat all of the food pellets.

In this problem, we want to minimize the performance metric (i.e. eat all of the pellets in the fewest number of turns). If we cast this problem into a search problem (where states are vertices and actions are edges), then we can use a search algorithm to find the shortest path from our initial state to a goal state. We will use the A^* algorithm equipped with the goal-test function to do so.

In order to use A^*, we need to define a heuristic that estimates the cost of the current state to a goal state. Design a heuristic that is admissible and consistent that will solve this problem. Your heuristic must be non-negative and must return 0 when the current state is a goal state.

Hint: If we are at a square that contains a food pellet, then the smallest cost to a goal state is the number of movements to enter all of the squares that still contain food pellets. So, something we would like to know (and probably cache beforehand) are the distances between pairs of squares that contain food pellets.

#initializing caches and relevant structures

The idea behind my heuristic is to give all nodes without a pellet a weight of 1, and nodes with pellets a weight of -1. This way, the AI will be able to easily determine the shortest possible path to and from other pellets when it clears an area.

```
Current_loc = current_loc       #just to have it for reference down here
Food_remaining_locs = food_remaining_locs
empty_nodes = {!food_remaining_locs} #we don't want to waste valuable resources revisiting a node when we don't have to.
Heuristic_fn(node): #this calculates the cost of moving to any specific node
        Cost = 0
        For node in path:
                If node in empty_nodes:
                        Cost ++
                Else: #either the node is empty or has a pellet
                        Cost = cost
        Return cost
Cost_fn(node): #this returns the actual cost of moving to a specific node reflected by the question
        Cost = 0
        For node in path:
                Cost ++
        Return cost
```

With the way the heuristic and cost functions are set up, the smallest possible value determined by a total cost function would be 1, given the case that there's a pellet right next to pacman. This also ensures that it is non-negative. In my heuristic function, I gave nodes with pellets a weight of 0 because I want the heuristic to view a 5-tile straight line filled with pellets as just as good as the option of moving a single square to a pellet. This way, the algorithm will be able to better select moving to a tile in the middle of a pellet-rich area as opposed to instead choosing to go for a single pellet in a dead zone in the opposite direction.

```
Find_path(loc):
        Cost_matrix = {}
        For node in map:
                Cost_matrix{node} = heuristic_fn(node) + cost_fn(node)
        Return cost_matrix
```

```
While food_remaining_locs != empty:
        Taken_path = min(find_path(current_loc))    #this is supposed to calculate the cost of every node and taken_path will be the best possible pellet to travel to.
        While executing taken_path:
                If node in food_remaining_locs:
                        Food_remaining_locs.pop(node)
                        Empty_nodes.add(node)
                        #i want to update the tables as pac moves once it has made a decision

Return 0        #once there are no more pellets on the map, the algorithm will return 0 and exit.
```