# Written Assignment 1

Due: Friday 02/02/2024 @ 11:59pm EST

## Disclaimer

I encourage you to work together, I am a firm believer that we are at our best (and learn better) when we communicate with our peers. Perspective is incredibly important when it comes to solving problems, and sometimes it takes talking to other humans (or rubber ducks in the case of programmers) to gain a perspective we normally would not be able to achieve on our own. The only thing I ask is that you report who you work with: this is **not** to punish anyone, but instead will help me figure out what topics I need to spend extra time on/who to help. When you turn in your solution (please use some form of typesetting: do **NOT** turn in handwritten solutions), please note who you worked with.

Remember that if you have a partner, you and your partner should submit only **one** submission on gradescope.

### 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 $\geq 0$. 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 $p^* = a \overset{x}{\leadsto} b$ from vertex $a$ to vertex $b$ has cost $x$. If we know $p^*$ passes through intermediary vertex $c$, then let $p_1 = a \overset{y}{\leadsto} c$, and let $p_2 = c \overset{z}{\leadsto} b$. 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 $b$.

**First, we need to prove that $p_1$ is the shortest path from $a$ to $c$.** Assume for the sake of contradiction that $p_1$ is not the shortest path from $a$ to $c$. Then, there must exist another path $p_1'$ with path cost $y'$ from $a$ to $c$ s.t. $y' < y$. Now, the shortest path from $a$ to $b$ becomes $p' = p_1' \cup p_2$, with path cost $y' + z$, which, as $y' < y$, is less than $y + z = x$. This contradicts the assumption that $p^*$ is the shortest path from $a$ to $b$ with cost $x$, as there exists a path $p'$ with a lower cost. Therefore, $p_1$ must be the shortest path from $a$ to $c$.

**Second, we need to further prove that $p_2$ is the shortest path from $c$ to $b$:** Similarly, assume that $p_2$ is not the shortest path from $c$ to $b$. Then, there must exist another path $p_2'$ with path cost $z' < z$ from $c$ to $b$ s.t. $z' < z$. Now, the shortest path from $c$ to $b$ becomes $p' = p_1 \cup p_2'$, with path cost of $y + z'$, which, as $z' < z$, is less than $y + z = x$. This contradicts the assumption that $p^*$ is the shortest path from $a$ to $b$, as there exists a path $p'$ with a lower cost. Hence, $p_2$ must be the shortest path from $c$ to $b$.

**Conclusion:** We can conclude that given that a shortest path $p^* = a \overset{x}{\leadsto} b$ from vertex $a$ to vertex $b$ has cost $x$, 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 $b$.

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

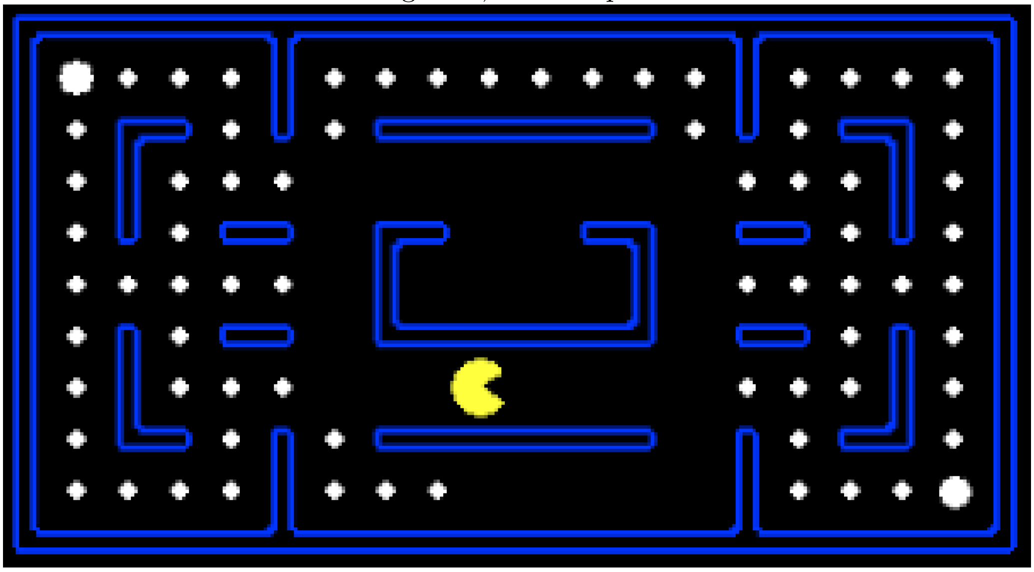
Consider a graph $G = (V, E, w : E \to \mathbb{R}^{\geq 0})$ where $V$ is a set of vertices, $E$ is a set of (directed) edges, and $w$ is a *weight function* that maps edges to weights where each weight is $\geq 0$. Let us start a DFS search from vertex $a$, the goal of which is to find vertex $b$ (where $b \neq a$). 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,** the goal vertex $b$ is the last vertex to be found in the DFS traversal and at deepest position in the graph, and starting from $a$, the DFS algorithm would have to explore all other vertices and traverse all edges in the graph before finding $b$. Therefore, the DFS expansion must visit all $|V| - 1$ vertices (except the source node $a$) and all $|E|$ edges before reaching $b$.

**In the best case,** there exists a direct edge from source vertex $a$ to $b$, and $b$ is the first neighbor DFS reaches immediately after $a$. Therefore, the DFS expansion would visit just 1 vertex (the goal vertex $b$) and 1 edge (the edge that directly connects $a$ to $b$).

**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.

To design an admissible and consistent heuristic for solve the the A* algorithm, we need to estimate the cost from the current state to the goal state, i.e., eat all food pellets, in the least number of moves. We'll use Manhattan-based distance function and Minimum Spanning Tree (MST) for such heuristic.

**Heuristic:** $h$ is defined as the weight of the MST connecting all remaining food pellets plus the shortest distance from Pacman's current location to the nearest pellet.

**Manhattan-based distance function:** There are two purposes to keep track of distance between the current coordinates and any other coordinates: to help find the next pellet with the minimal distance to Pacman and to build MST given the fixed distances between each pair of food pelletes. Since Pacman moves in 4 cardinal directions in a grid, we choose Manhattan distance as the metric, which is the sum of the absolute differences in the x and y coordinates). We may slightly update the function to account for walls, so that the resultant function reflects the actual number of moves from current position to another grid.

**Minimum Spanning Tree:** The MST of the remaining food pellets represents the shortest path that connects all remaining pellets and is a subset of the graph connecting all the food pellets with the minimum possible total edge weight, where edge weights represent distances between pellets. With MST, we can estimate the minimum distance that Pacman must travel to eat all the remaining food pellets.

- **Admissibility**: The weight of the MST connecting all remaining food pellets represents the minimum total distance that could possibly connect all pellets, without considering walls or the actual paths Pacman must take. It's a lower bound because, in the actual game, Pacman might need to choose longer paths due to walls or backtracking. Therefore, the MST weight won't overestimate the total true cost to the goal. The shortest distance from Pacman to the nearest pellet is the immediate cost, as it only considers the very next pellet and not the entire path to collect all pellets. This also underestimates the cost.
- **Consistency**: Moving closer to the nearest pellet decreases the distance part of the heuristic by at least the true cost of the move. Eating a pellet and recomputing the MST can only keep the MST component the same or decrease it. The MST cost does not increase because removing a vertex (a consumed pellet) from an MST cannot lead to a MST with added weights for the remaining vertices.

**How the heuristic is used:**

- As the state changes, i.e., pellets are eaten, the set of remaining pellets changes, so as the MST's total weights and distance to the nearest pellete. We update the MST and distances after Pacman moves and eats a food pellet, reducing the collection of remaining pellets.
- the A* algorithm recomputes the heuristic's value for a new state to update the new configuration of remaining pellets. Specifically, the MST is updated by removing the eaten pellet's vertex and reconstructing itself for the reduced set of vertices.