6.101 Recitation 9: Week 4 Snekoban Midpoint **3/6/24**

This sheet is yours to keep!

Shown below are ordered timesteps of a game called “Free Food Bonanza.” During each timestep in the game, the student (S) moves one adjacent step (up, down, left, right). The student cannot move out of bounds or step on walls (W). The goal is to collect all the free pizza (F) in the shortest number of moves. When the student lands on the pizza, they consume it instantaneously and it disappears from the board.

1

S		
	W	F

2

S	W	F

3

S		
	W	F

4

	S	
	W	F

5

		S
	W	F

6

	W	S

7

		S
	W	

8

	S	
	W	

9

S		
	W	

10

S	W	

Question 1:

In the game timestep diagram above:

A. Circle any duplicate game boards.

B. For each board, draw arrows to all the other boards that could be reached in a single move.

C. What timestep is the game supposed to end? Cross out the timesteps which occur after the game ends.

D. What is the minimum number of moves (not timesteps) it would take to collect all the free pizza if we started at the board in timestep 1?

Graph Search Questions:

1. What are the unique states in the graph?
    - How can I represent the state in a way that is concise (avoid storing unnecessary duplicate information shared between states)?
    - How can I represent the state in a way that is hashable (the visited set requires a hashable state)?
2. What are my edges?
    - How do I get neighbors / find the edges of a given state? How can I do this efficiently?
3. What is my desired result?
    - What information do I need to store to get the desired output? How do I store it?
4. What is my starting condition / state?
5. What is my goal / stopping condition?

R9 Participation Credit **Kerberos : _______________@mit.edu**

Hand this sheet in at the end of recitation to get participation credit for today.

Question 2: Fill in the missing return statement in the code below. Note: the definition of `find_path` is provided on the back of this page.

```
def free_food_bonanza(board):
    """
    Given a starting board, calculate the minimum number of moves required
    for the student to collect all the food on the board.

    Parameters:
        board: a list of lists of strings, where each cell holds one of:
            - 'S' for student (exactly one on the board)
            - 'F' for food (arbitrarily many on the board)
            - 'W' for wall (arbitrarily many on the board, student may
                    not walk through them)
            - ' ' for an empty square
    Returns:
        Number of moves if board is solvable, None otherwise
    """

    def make_state(board):
        # some code here returns a new state




    def get_neighbors(state):
        # some code here returns a list of states




    def goal_check(state):
        # some code here returns True/False
        # depending on whether a state meets the goal conditions




    start = make_state(board)
    path = find_path(get_neighbors, start, goal_check)
```

return ______________________________

```python
def find_path(neighbors_function, start, goal_test):
    """
    Return the shortest path through a graph from a given starting state to
    Any state that satisfies a given goal condition.

    Parameters:
      * neighbors_function(state) is a function which returns a list of legal
        neighbor states
      * start is the starting state for the search
      * goal_test(state) is a function which returns True if the given state
        is a goal state for the search, and False otherwise.

    Returns:
        A shortest path from start to a state satisfying goal_test(state)
        as a tuple of states, or None if no path exists.

        Note the state representation must be hashable.
    """
    if goal_test(start):
        return (start,)

    agenda = [(start,)]
    visited = {start}

    while agenda:
        this_path = agenda.pop(0)
        terminal_state = this_path[-1]

        for neighbor in neighbors_function(terminal_state):
            if neighbor not in visited:
                new_path = this_path + (neighbor,)

                if goal_test(neighbor):
                    return new_path

                agenda.append(new_path)
                visited.add(neighbor)
```