

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page.

Name: \_\_\_\_\_

## Breadth-first Search

1. In this section you will implement the breadth-first search algorithm. This is an uninformed (unit-cost) search algorithm because it ignores the edge costs. As a review, draw the graph for a 2x2 game board with no obstacles. Use the tile coordinate as the node name.

2. Return a list of edges for the above graph for the tile located at (1,1).

3. In this question, you will write a function `bfs(level, start, goal)` in `pathfind.py` that implements the breath-first search algorithm as discussed in class. To test this function, you may want to use something like:

```
level1 = [  
    ['.', '#', '.', '.', '.', '.'],  
    ['.', '.', '#', '.', '#', '.'],  
    ['#', '.', '#', '.', '#', '.'],  
    ['.', '.', '.', '.', '.', '.']  
]
```

and

```
print bfs(level1, (0,0), (1,1))
```

after which you can introduce additional obstacles. The algorithm should return `None` if no path can be found.

- (a) Create an empty list called `explored`.

- (b) Create an empty deque called `frontier`. The `explored` and `frontier` constitute the two data structures needed for BFS as well as many other search algorithms.

- (c) Bootstrap (initialize) the algorithm so that the `frontier` contains the starting path. Recall that a path is a list of nodes, but in this case the path will simply be a single-element list containing the start node.

- (d) The algorithm runs as long as there are more nodes to explore. Write a `while` loop to reflect this condition.

- (e) If the loop terminates, that is, the `frontier` is empty and you still haven't found your goal, it means that no path has been found. Return `None` in this case.

- (f) Now we will fill in the body of the loop. Since the algorithm is BFS, remove the first (left) node off the `frontier` and store it in `path`.

- (g) The previous operation gives us the whole path, but the node itself is the last element in the path. Name this element `n`.

- (h) If `n` is the goal, then you have found a path from `start` to `goal`. If this is the case, then return the `path`.

- (i) Otherwise, we have not yet found the goal. However, we have now explored the node `n`, so append it to the `explored` list.

- (j) You will now need to get the edges (called `e`) of the node `n`. Recall that there is a `get_edges` function that performs this task. Write a `for` loop to iterate across this list.

- (k) Finally, if you have not yet been to `e`, that is, `e` does not appear in `explored`, then you will need to add the path so far and the node itself to `frontier`. You can do this by appending `path + [e]` to `frontier`.

## Avatar Representation

4. You will now develop the avatar representation, using `game.py` as a starting point. We will use simple rectangles (rather than sprites) to represent our avatars.
- (a) Change the `boy_image` background color from white to `(135,145,191)`.

- (b) Change the `girl_image` background color from white to `(255,192,203)` so that you can differentiate the two avatars.

- (c) Recall that in AI, it is convenient to be able to treat the avatar representations as points, using the center of the object (or more generally, the center of mass) as its position. Unfortunately, the `screen.blit` routine blits from the top left. Thus,

assume that you have a sprite at position (90, 80) having dimensions 100x100. What should the coordinates to `blit` be?

- (d) Re-blit `girl.surface` and `boy.surface` using the re-positioned coordinates. Note that you can get the center of the sprite (in this case, for the `girl`) using the following code:

```
girl.surface.get_rect().center
```

## World Representation

5. Modify the `tiles` dictionary so that it recognizes the `.` and `#` tiles as the `key` and returns the colors (50,50,50) and (128,0,0) as the values. These tiles are used by the `drawBoard` function to draw tiles to the screen (you need not modify the `drawBoard` function as it is already provided).


6. The `tilePositionToPixel` function takes in a tile and returns the center location of that tile. For instance, the center location of a 100x100 tile at (0, 0) is (50, 50).

- (a) What is the center location of a tile at (1, 0)?

- (b) What about a tile at (1, 1)?

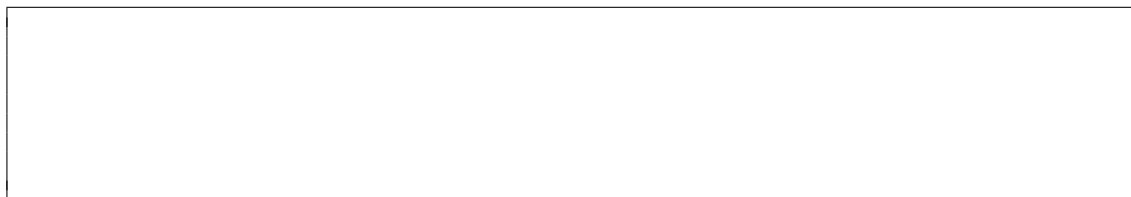


- (c) Write this function in Python. Hint: Use `np.array` to perform calculations, noting the constants `TILE_WIDTH` and `TILE_HEIGHT` that have been provided to you.



7. You will now write the `findNext` function. This function takes in a `level`, `ai`, and `target` and returns the next tile that the `ai` needs to make progress toward.

- (a) Unpack the `ai` and `player` position into `ai_x`, `ai_y` and `player_x`, `player_y`. This simplifies some of the calculations. Note that you need to cast each position to an integer first, using `int`, for example, `int(ai_x)`.



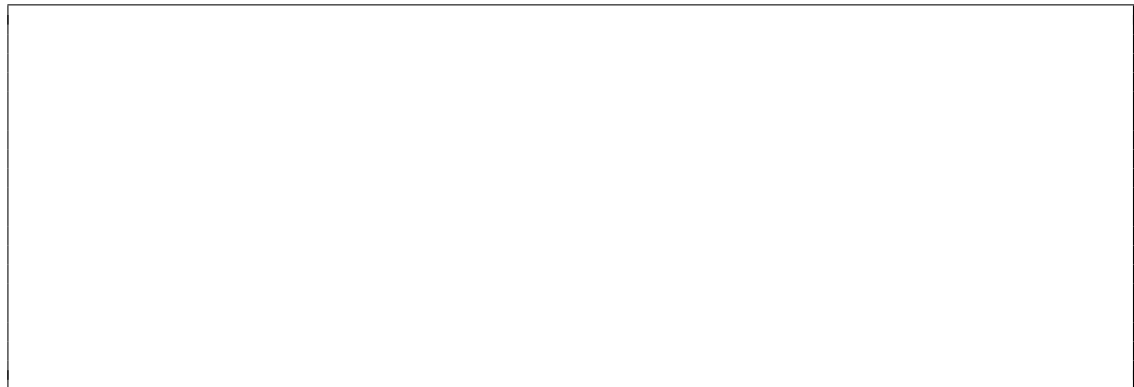
- (b) Create variables `player_tile` and `enemy_tile` that converts the pixel locations to the corresponding tile tuple.



- (c) Now that you have the tile names. You can call the `pathfind.bfs` function with the appropriate arguments. Store the result of this function in `path`.



- (d) The `path` may contain different results. It may have `None` to indicate that there is no path. It may return a single-length list, to indicate that you are already in the destination tile (though not in the center). Finally, it may return a list with multiple elements, in which case the second element indicates where you need to go next. Write three conditional statements to reflect these cases, using `tilePositionToPixel` to convert tiles to pixels (which will be used by `seek`). The function should return `None` if no path can be found, or the center location of the next tile otherwise.



Congratulations, you have just implemented hierarchical path planning! The techniques in this lecture can easily be adapted to more sophisticated path algorithms, such as `A*`. Next week, we'll learn about decision making.