CS 120: Introduction to Computer Programming II

# Carcassonne Project, Long B

see the Short A spec for due dates

## 1 Overview

See the spec for Short A to see an overview for the entire multi-part project.

All of your work in this part will be new methods in the Map class. Using the one-direction road tracer as a helper function, you will write a new function, which traces a road in both directions, and which checks to see if it has been "completed" - that is, if both ends reach crossroads.

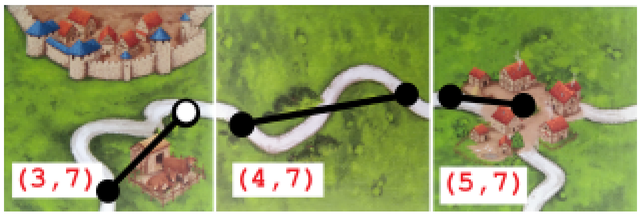
You will write a function which traces through a city, to find all of its pieces; this is more complex because cities spread out, instead of being linear. We'll show you how to explore this using something called a "flood fill."

(spec continues on next page)

## 2 trace_road()

You must add two new methods to your Map class. The first one, `trace_road(self, x,y, side)`, is a lot like `trace_road_one_direction()` that you implemented in the Short problem, but this one traces the road in both directions: both moving foward and backward.

Using our example from the Short project, we see that if we start at `(3,7,E)`, not only can we move East (as we did before) we can now also move down to `3,7,S`, and perhaps continue on from there.



This method will trace the road in both directions, and find both ends; it will return a sequence of tuples which represent the entire path, starting “behind” the start point, and ending “in front of” it.

Your implementation for this function must call the “one direction” function, as a helper function, and then assemble the pieces into your overall answer. In the example above, you would get:

```
starting at (3,7,E):      [ (4,7,W,E), (5,7,W,CENTER) ]
representing tile (3,7): [ (3,7,S,E) ]
starting at (3,7,S):      [ ]
----
overall return value:     [ (3,7,S,E), (4,7,W,E), (5,7,W,CENTER) ]
```

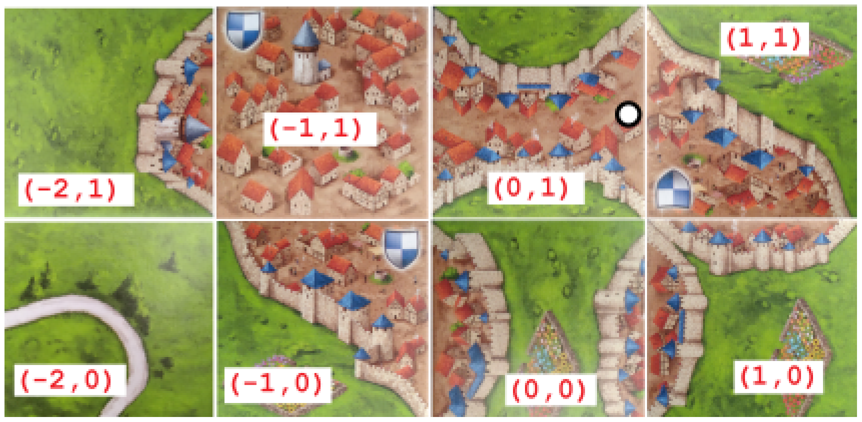
### 2.1 Loops

As with the previous version, this function must detect loops. If the road has a loop, then this new method must return the exact same thing as the one-direction version. (When you call the one-direction version as a helper function, it should be pretty easy to detect whether or not the helper found a loop. If it does, you can simply return what it gave you, with no more work.)

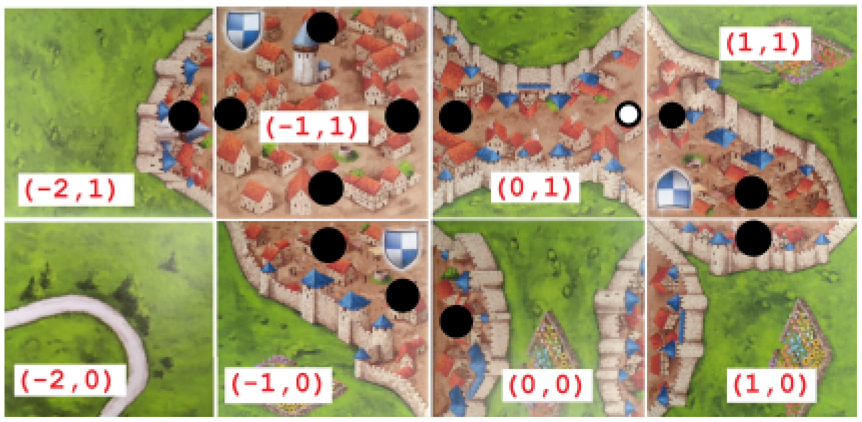
(spec continues on next page)

## 3 `trace_city()`

You must also implement a method, `trace_city(self, x,y, side)`, which searches from a given location and finds all of the parts of the city. Consider this example city; we are going to start our search at `(0,1,E)`:



Our goal is to report all of the pieces that form a contiguous city. We return these as a **set** of tuples. Looking at the city, we see that the following tile edges are part of the contiguous city:



### 3.1 Notes on Tracing Cities

You'll note that it is important that we track the **side** of everything we find - not just the tile - because it's possible to have a tile (such as $(0,0)$ and $(1,0)$ above) where **some** of the city parts are included in the city we're tracing, and some are not.

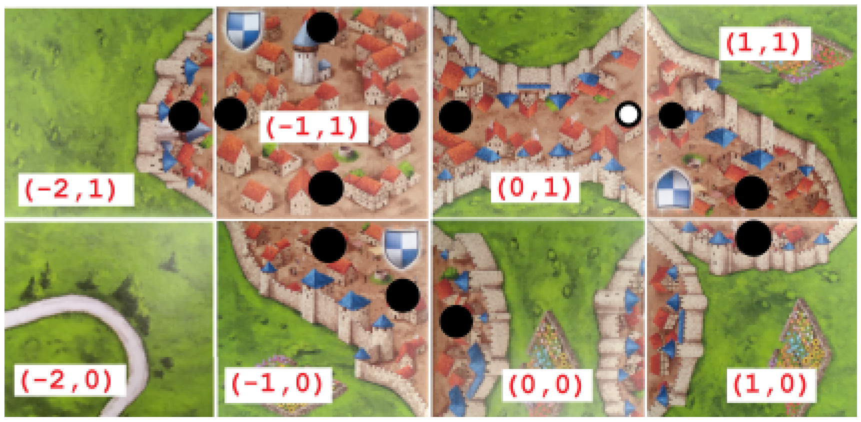
Carcassonne also cares whether the city is "complete" or not. (Go play the real game, to find out why.) A complete city is one that is entirely enclosed.

We note that this city is **not** complete because the edge `(-1,1,N)` is part of the city, but the tile $(-1,2)$ doesn't exist in the map.

In addition to returning the set of tile edges, you will also be returning a Boolean, which tells the caller whether or not the city is complete. This is easy to check: when you are expanding the city (see below), if you ever try to cross from one tile, into an adjacent tile, but the tile doesn't exist, then the city is not complete.

## 3.2 Return Value

As we've noted above, the city in this example is **not** complete, because the tile $(-1,2)$ is missing. And the tile edges that are part of the city are as shown here:



This function must return a tuple with two elements: a Boolean (indicating whether or not it is complete), and the set of edges. Therefore, the proper return value for this example city is:

```
( False,
  {
              (-2,1,1),
    (-1,1,0), (-1,1,1), (-1,1,2), (-1,1,3),
    (-1,0,0), (-1,0,1),
              ( 0,1,1),           ( 0,1,3),
                                  ( 0,0,3),
                        ( 1,1,2), ( 1,1,3),
    ( 1,0,0),
  }
)
```

(spec continues on next page)

## 3.3 Algorithm

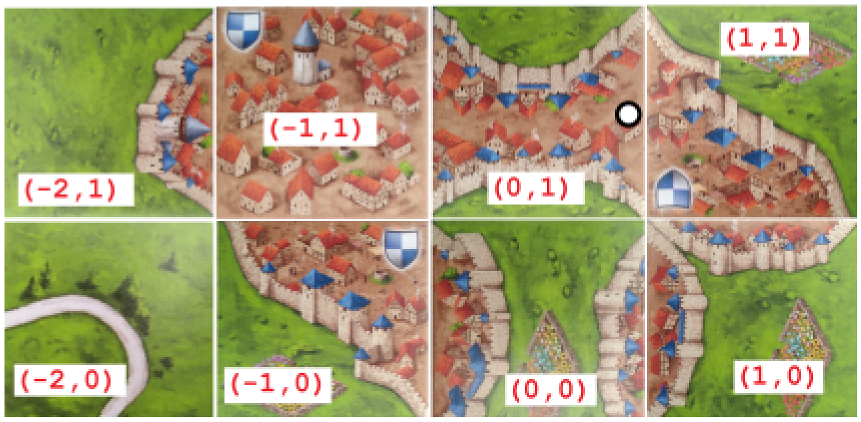
So, how exactly do you find the edges of a city? With a road, you could use a simple loop (or recursion) - just follow the road until it ends, or until it returns back to the beginning. But as you've seen from the example, cities aren't as simple - they spread out in many directions. And while it would be tempting to build a tree, there's all sorts of complexities with that. (Just consider a city that was a whole grid of Tile 05's. Which ones are the parents and children in that case???)

Instead, we're going to use an algorithm called "flood fill." Basically, the idea is that you keep track of which tile edges are already in the city. You then, over and over, hunt for more edges to add. When you finally run out of new edges to add, the algorithm stops.

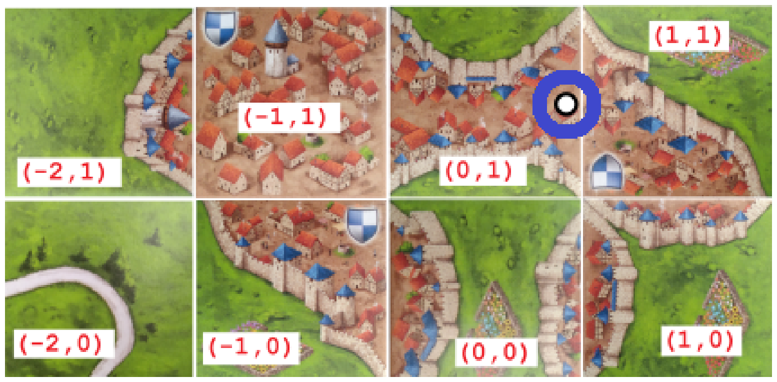
Think of it like pouring out water on the floor, which gradually spreads until it covers the entire room. That's why it's called flood fill!

Let's simulate the algorithm. We'll use the same example city from before. The caller has told us to explore from `(0,1,E)`, and so our city will start with that tuple as the only element in the city:

**City:** `{ (0,1,E) }`

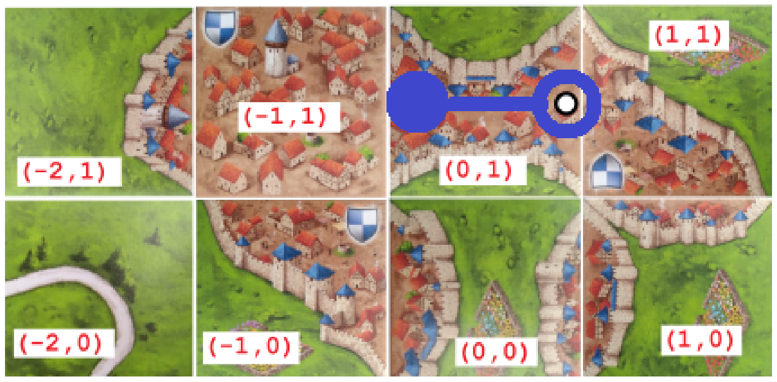


Our flood fill algorithm will look at each part of the **current** city, and see if we can expand it. (We can use a `for` loop over the current city for this purpose.) In this case, there is only one edge in the city so far, so we will choose it:

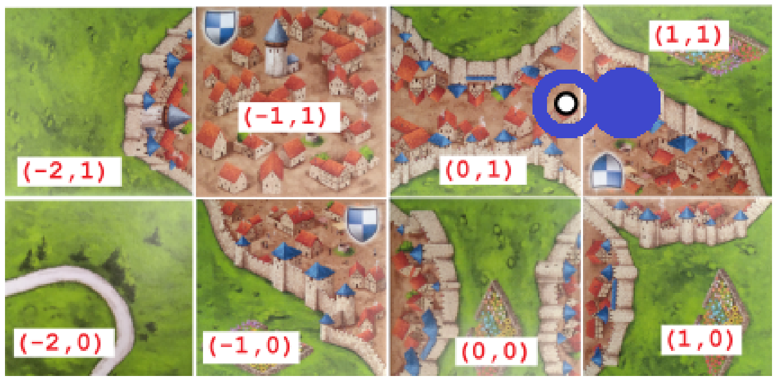


For each edge in the current city, we do two things: we look for other edges on the same tile, which are connected, and we look at the adjacent edge, to see if the tile exists.

To check the first case, we use the `city_connects()` method of the current tile to see what edges (if any) are connected to the E side. We find that the W side is connected, so we will add it to the city.

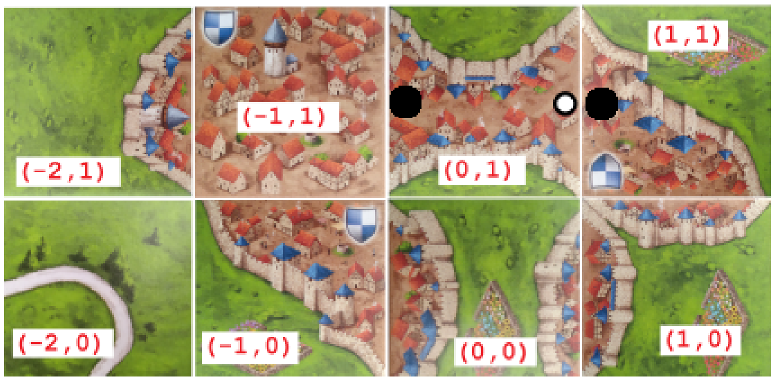


Then, we cross over, to see if the tile $(1,1)$ exists. In this case it does, and so we can add another tile edge to our city:

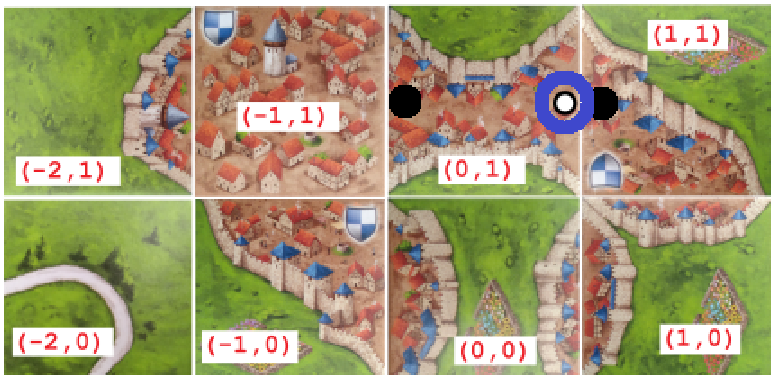


After the first round of expansion, we now know of three different parts of our city:

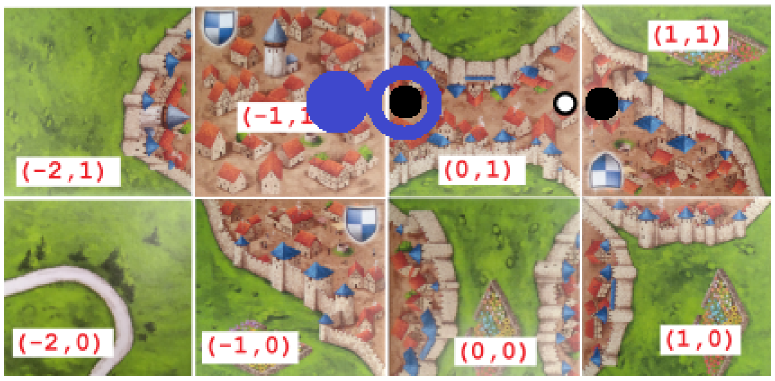
**City:** `{ (0,1,E), (0,1,W), (1,1,W) }`

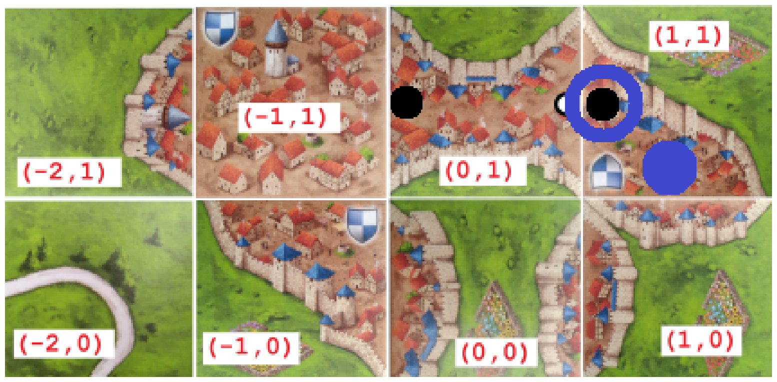


We will now try to expand all three of the tile edges. In some cases, such as `(0,1,E)`, you will not find anything new to add:



Often, you will find that a given edge has some things to add, and other things which are already in the city:

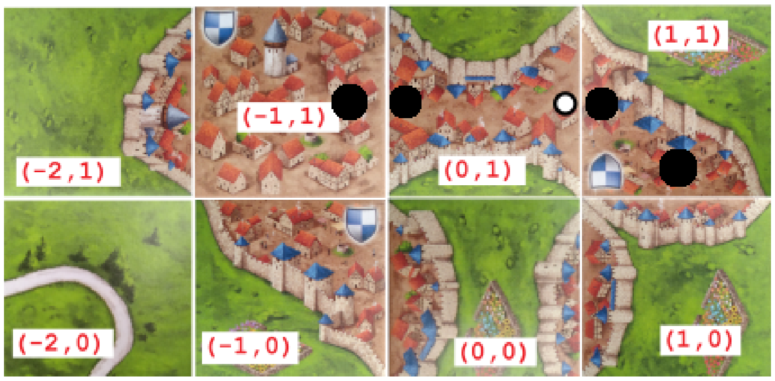




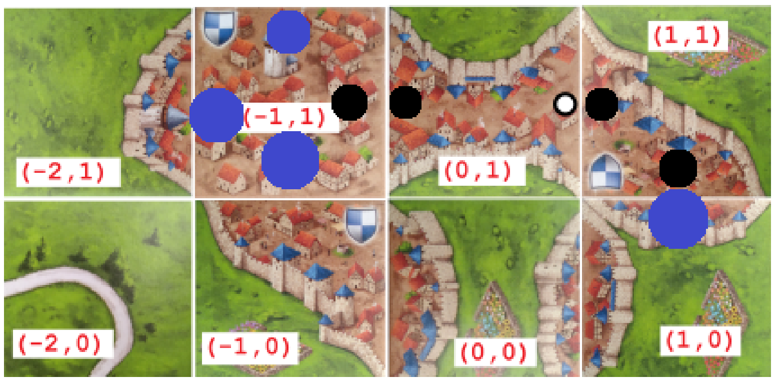
(spec continues on next page)

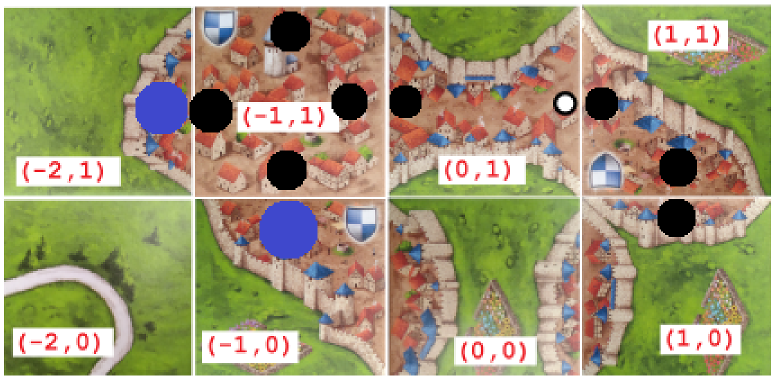
After two rounds of expansion, we now have 5 edges in our city:

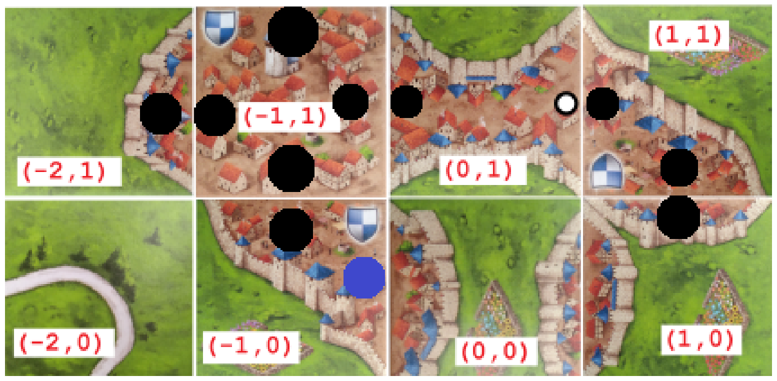
**City:** `{ (0,1,E), (0,1,W), (1,1,W), (-1,1,E), (1,1,S) }`

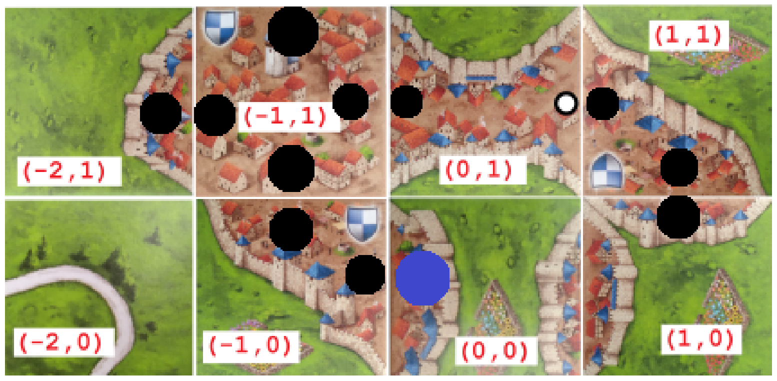


With each successive round, we add more edges:









(spec continues on next page)

### 3.4 Notes on Flood Filling

https://en.wikipedia.org/wiki/Flood_fill

Flood fill can be performed very efficiently with a queue: every time that you find a new part of the city, you add it to the city, but also add it to a "TODO list" of parts to examine. Then, instead of exploring old parts of the city over and over, you simply pull elements off of the TODO list.

If you wish to write your code this way, feel free!

However, I expect that most students will find it easier to use a simpler, less efficient algorithm. I've provided pseudocode for it here:

```
city = { ... one element to represent the start position ... }

keep_searching = True
while keep_searching:
    keep_searching = False    # we'll turn this back to True if we find a reason

    # python doesn't like you to modify a data structure while you're
    # doing a for() loop over it.  So we'll duplicate the current city
    # into a temporary variable, so that we can modify the "real" city
    # if we find new things
    dup = list(city)

    for every location in the duplicate of the city:
        for every other side, in the same tile:
            if this other side is not in the city:
                city.add( other side )
                keep_searching = True

        neighbor = edge next to the current edge, on the next tile over
        if this neighbor side is not in the city:
            city.add( the neighbor side )
            keep_searching = True
```

The concept behind this algorithm is simple: scan the **entire city,** and see if you can find **any** way to make it bigger. If you can, then add it to the city. Keep looping until you cannot find anything new.

If you look at this carefully, you will find that this is basically the same as the TODO list strategy - the TODO list is just a way to "focus your attention" on newly-added parts of the city - so that you don't waste your time searching the same old part of the city, over and over.

## 4 Turning in Your Solution

You must turn in your code using GradeScope.