CS 120: Introduction to Computer Programming II

# Carcassonne Project, Long B

## 1 Overview

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

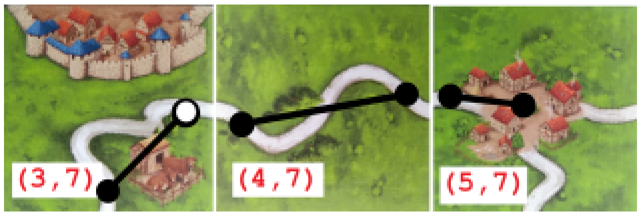
All of your work in this part will be new methods in the Map class. Using the one-direction road tracer as a helper function, you will write a new function, which traces a road in both directions, and which checks to see if it has been "completed" - that is, if both ends reach crossroads.

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."

## 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)`, no 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.

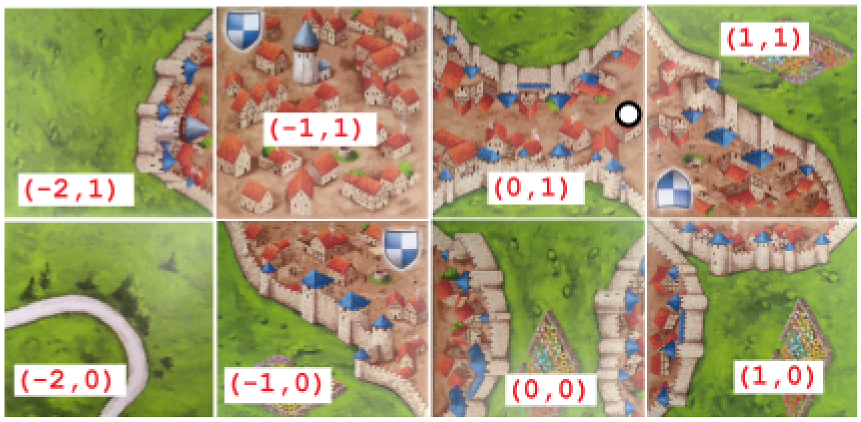
Your implementation for this function must call the "one direction" function twice, as a helper function, and then assemble the pieces into your overall answer. In the example above, you would get:

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

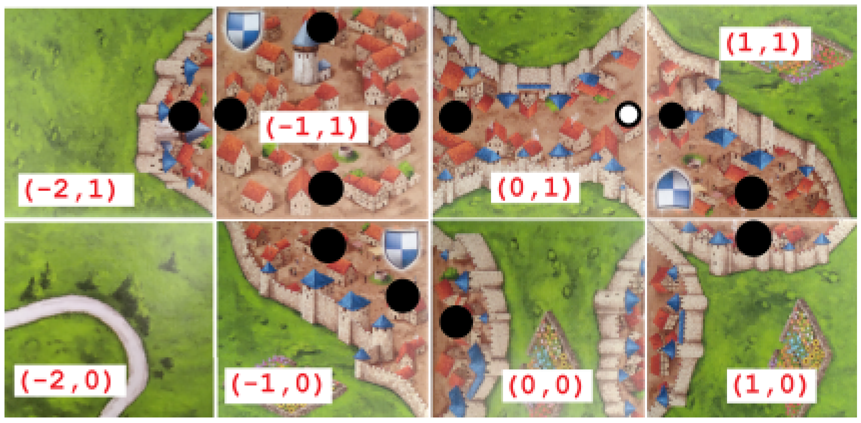
(spec continues on next page)

## 3 `trace_city()`

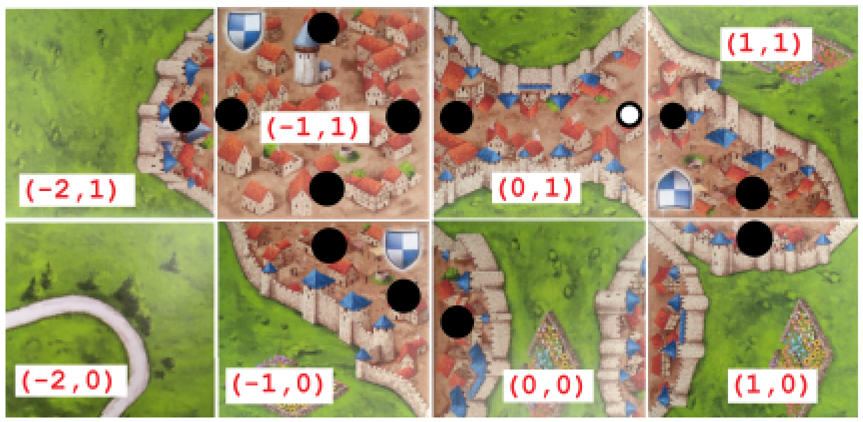
You must also implement a method, `trace_city(self, x,y, side)`, which searches from a given location and finds all of the parts of the city. Consider this example city; we are going to start our search at `(0,1,E)`:



We first ask the Tile what other edges are connected to the city on the East side; we discover only the West edge. Then, from the East and West edges, we explore to the opposite side of each: that is, we add the West edge of $(1,1)$ and the East edge of $(-1,1)$. From there, we keep exploring out, using each Tile to figure out how a city is connected internally, and then using that to find more tiles to explore. Eventually, the complete list of edges that we discover is as follows:



(spec continues on next page)



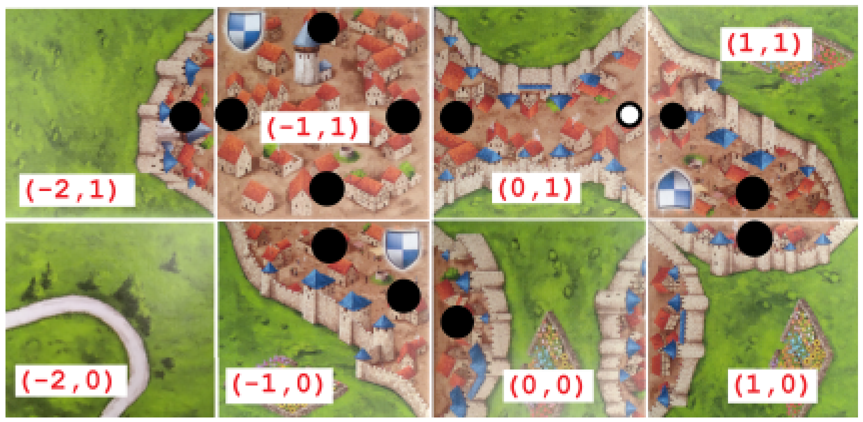
## 3.1 Notes on Tracing Cities

You'll note that it is important that we track the **side** of everything we find - not just the tile - because it's possible for 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.

Yes, it is that simple: if, while you are searching the city, you find an edge which is inside the city - but the tile that it faces is not in the map yet - then the city is **not** complete. On the other hand, if **every** edge that is part of the city is next to another tile, then the city is complete.

(spec continues on next page)



## 3.2 `trace_city()` Return Value

`trace_city()` returns two values, as a tuple: a boolean, and a set. The boolean indicates whether the city is completed. The set is a bunch of tuples, representing the edges which are in the city. The edges are encoded similarly to how road segments are encoded by `trace_road()`, except that each one only lists a single side. Thus, the leftmost black dot in the example above (the East edge of the tile $(-2, 1)$) would be encoded as

```
(-2,1,E)
```

or rather, since side E is represented by the number 1:

```
(-2,1,1)
```

Thus, the city in example above would return:

```
( False,
  {
               (-2,1,1),
    (-1,1,0),  (-1,1,1),  (-1,1,2),  (-1,1,3),
    (-1,0,0),  (-1,1,1),
               ( 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 Notes on Flood Filling

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

To find the entire city, you will perform some sort of "flood fill," which basically means you will start with a single point, and go further and further out, adding parts to the city until you can't find any more.

This can be done quite efficiently using 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; each time you pull something off the TODO list, you check for all nearby locations (the other 3 in the current tile, and also the next tile over) to see if you can find any new locations. You will find lots of duplicates, which you will not add to the TODO list; but eventually you will trace all of the possible locations, to all possible parts of the city, and be done. It runs in something approaching $O(n)$ time, which is great!

(Something analogous can be done with recursion as well, if you want to try that.)

## 3.4 A Simpler, Slower Way

**However,** there is a simpler, slower way - and for 120-level students, I'm perfectly happy with you doing it. Instead of keeping a TODO list of parts to look at, you can simply re-consider the **entire city,** over and over. This algorithm is **inefficient but effective** - and sometimes, simpler is better. The algorithm basically works like this:

```
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.