CS320 Lab:
Palindromic Graphs

Prof. Craig Partridge

Lab #8

This lab focused on a data-driven traversal of a graph similar to a classic routing problems. We will provide the graph module, which implements a graph class, and a graph. Your job is to list all the paths in the graph, whose values create a palindrome, a sequence that can be read the same way forward and backwards.

1 What is a Palindrome?

A palindrome is simply a series of values that is the same in both directions. So "racecar" (which is 'r', 'a', 'c', 'e', 'c', 'a', 'r' when examined from the start to end, or end to start of the string) is a palindrome. So too is the number 10101.

In this case we asked you to treat sequences of vertex weights/values as a list, so a list such as `[1, 3, 5, 5, 3, 1]`, and see if the list is a palindrome.

2 The Graph Module

You will use the course `edgegraph.py` module.

3 The Assignment

You are to write a function `pld_graph()` which returns a tuple of all the palindromes you found after traversing all the paths in the graph, **without** duplicates.

A *path* is defined as any series of edges, connected by vertices. A path can traverse a vertex more than once, but may not contain an edge more than once. Each edge in the graph contains a value. A path, therefore, defines an ordered list of edge values. It is these ordered lists of values that you are to examine for palindromes.

For the purposes of this assignment, a palindrome is defined as a sequence of at least **three** edge values that are the same when reversed. So

```
[1, 2, 3.14159, 2, 1]
```

is a palindrome. Note that we treat each edge value as distinct. So while

```
[1, 3, 3, 331]
```
might appear to be a palindrome, because 1, 3, 3 looks like 331, in fact it is not, because 1 and 331 are not equal.

Observe too that a single path may contain multiple palindromes. For instance, the path

```
[1, 0, 1, 0, 1]
```
has two:

```
[(1, 0, 1), (1, 0, 1, 0, 1)]
```

This problem is slightly simpler than it sounds (see Tips). Also, while there are multiple ways to make your code run faster (again, see Tips), you only need to implement one, and you can pick which one you implement.

3.1 Requirements

Reminder that every assignment has a code portion, which is worth 60 points (40 for correctness, 20 for literate programming), and a reflection (worth 40 points).

3.1.1 Code Requirements

`Pld_graph(g)` takes a single argument `g`, which is a graph from the course edgegraph class. If `g` is None, the routine should raise a ValueError("Bad graph") message.

If there are no palindromes, `pld_graph()` returns an empty list `[]`. Otherwise, `pld_graph()` should return a list of tuples, where each tuple contains one palindrome.

The graph may be a forest, with partitions that are not interconnected.

All edges are assumed to be two-way. You can traverse them in either direction, but any edge may appear just once in any path. The path may loop through a vertex more than once.

3.1.2 The Reflective Essay

A quick comment about the reflection essay. This essay is not about impressing us with how smart you are. Rather its primary purpose is to get you to *reflect on your problem-solving process* and *learn why the processes you used were or were not effective*. We hope, by the end of the course, you will be able to look back on these essays and see ways you've become a better programmer and more adept at algorithmic thinking.

For this reflection essay, we ask you to us (your GTAs) about your experience with this lab. Your essay should be no more than 250 words.

When writing your reflection, consider the following questions:

- How did you manage tracking all the paths?
- What made that approach to tracking paths attractive? After implementing it, what might you change?
- How did you improve your code after you got it running? Why did you make those changes?
- Why do you think your decisions were ultimately effective?

You can answer all or some of the questions suggested above, but be sure to use transition phrases and connect your ideas so that your reader understands your problem-solving process as a process that moves from one step to the next.

3.2 Tips

This problem is a basic graph traversal problem. You can start a path with each edge, and all paths starting with that edge and see if any of those paths, possibly truncated, form a palindrome. Note that all the palindromes from a given edge must have the first edge of the path in the palindrome. (Why? Suppose in tracing from edge X you find a palindrome starting at edge Y. Your code has already or will also trace paths from edge Y, so you'll find the palindrome starting at edge Y when you trace from edge Y - so you don't need to find it when tracing from edge X. **Note:** for the purposes of finding an optimizing solution, this does **not** count.).

There are some obvious optimizations (and you must implement at least one):

- It is easy to count (in time O(e) - and remember the `Counter` class) the number of times each value appears in the graph. Once a path search encounters a value that only appears once, that value must be the middle value of any longer palindrome. If the value appears after the mid-point in the path, any palindrome ends at the edge before the one containing the singleton value.

- Fancier counting or tracking algorithms are possible. For instance, if you know there are only three "a" values and the first three values in the path are "a", you know that's your only possible palindrome. (Generalizing this insight is *hard*).

While it is possible to extend all paths one edge at a time, that's hard. A simpler solution is to pick an edge, and build all the paths from it, find the palindromes, save them, and move to the next (starting) edge.

Keep in mind that an edge has two ends, so tracing the paths means tracing both directions (through both ends of the edge).

Note that the same palindrome may occur staring from different edges. For instance, if all the values are 0 or 1, there are likely several edges with initial paths of 111, 101, 010, or 000. In this case, you should return the palindrome **once**.