**Record of Activity**

Week 1 (25.11.2019 – 29.11.2019):

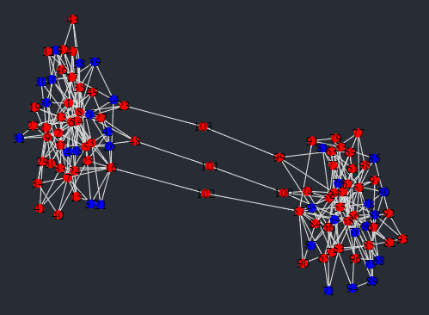
* I familiarized myself with Julia and Eirene. Specifically, I read the paper *Persistent and Zigzag Homology; A Matrix Factorization Viewpoint* (<https://arxiv.org/abs/1911.10693>) and implemented up to algorithm 3 in Julia. The implementation of cellular complexes is my own and might not currently be compatible with Eirene but I think it could be easily adapted. A brief description for each of the algorithms I implemented is:

1. The Reduction Algorithm: given an matrix , return a factorization where is upper triangular and no two columns of share the same non-zero pivot. This algorithm is used to compute homology-revealing bases for a cellular complex.
2. Given the reduced boundary matrices (the matrices obtained from the Reduction Algorithm applied to the boundary matrices for the cellular chain complex), compute homology-revealing bases for a cellular complex. I chose to implement this algorithm as simply taking the standard boundary matrices and then reducing them by making a call to the Reduction Algorithm so that users did not have to use multiple input types for different algorithms.
3. This algorithm computes the induced map on homology in terms of the homology-revealing bases from algorithm 2. It takes as input a chain map represented by matrices with respect to the original basis and returns the induced map  as a matrix. I considered using this to compute the inclusion maps on different subcomplexes of graphs to see if any useful information could be retrieved, but my understanding is that persistent homology already extracts information from inclusion maps.

* I learned some of the underlying theory of persistent homology by reading the paper *Zigzag Persistent Homology and Real-valued Functions* (<https://www.mrzv.org/publications/zigzags/socg09/>). I tried some simple examples of computing zigzag persistence and also tried to apply the Diamond Principle. I proved an easy case of the Diamond Principle by utilizing the Mayer-Vietoris long exact sequence, as suggested in the paper by Carlsson et al. I was initially having difficulty computing the persistence barcode; I have read the proof that it exists and is unique but could find no intuitive explanations as to how to compute it by hand. I correctly guessed a few examples by trial-and-error until Yossi Bokor pointed out that, at least for the 1-dimensional simplicial complexes given by the graphs we’re working with, the 1-dimensional barcode just represents the birth times of different 1-cycles since the 1-cycles never die.
* Kate and I met to discuss the papers I’ve read. We also looked at some of Marian’s graphs and discussed which vertices could reasonably be considered bridges. I’ve decided that, as a starting definition, I will take a bridge to be any vertex that connects two densely connected clusters.

Week 2 (2.12.2019-6.12.2019)

* Kate wanted to try computing the zigzag persistence of the distance function to a vertex: fix a vertex and for every other vertex , define to be the length of the shortest path from to . This can be computed quite easily in Julia using Djikstra’s Algorithm, which is pre-existing code. Since the function also has to be defined on -simplices, we set . I initially forgot that to compute zigzag persistence with sublevelsets, we had to have being a subcomplex for every . Consequently, I had the original definition of on 1-simplices using min instead of max which gave inconsistent results.
* Kate and I spoke to Marian over Zoom and discussed some of the overarching goals of the project. He gave me access to the Jupyter notebook he used to generate the example graphs I’ve been working with. He has confirmed that a bridge is a vertex, not an edge, which is good because neither Kate nor I was certain about this.
* Yossi spent a day teaching me to use Eirene and in exchange, I’m going to give him the code I wrote based on the paper on matrix factorization. I now know how to use Eirene’s standard implementation for a cellular complex and also how to read Marian’s adjacency matrices from a file. I started writing some helper-functions for the algorithm, such as a function to build the induced subgraph of an annulus around a point. I also developed a preliminary test for bridges which is quite efficient but not hugely accurate. The idea is that, for a vertex , we can take the neighbourhood of radius about and determine if removing disconnects the neighbourhood. If so, then is a ‘local bridge’ and hence is quite likely to be a global bridge. I am currently taking to be the distance from to the nearest cluster. This test works well for identifying lone bridges, but often fails to identify parallel bridges because they are in the same neighbourhood and so removing doesn’t disconnect the neighbourhood. Below is the first example I tried this algorithm on. The algorithm correctly identifies all of the bridges but also has a large number of false positives. This is fairly standard across all of the example graphs.



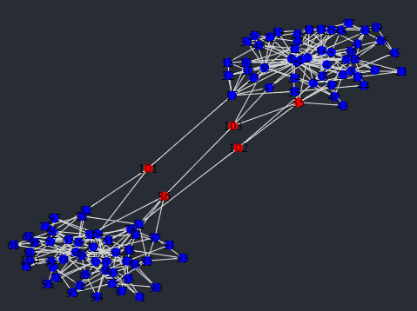
Week 3 (9.12.2019-13.12.2019)

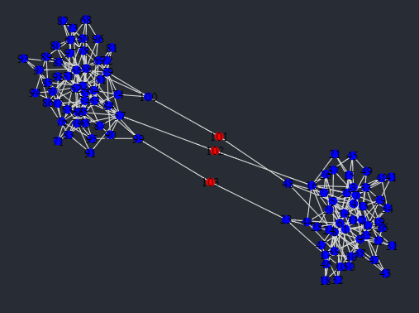
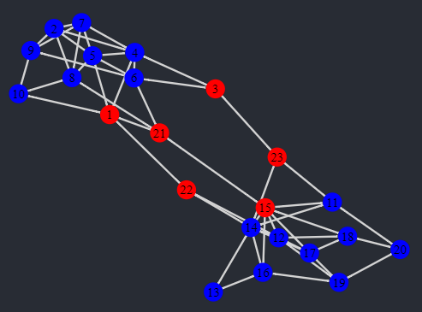
* Kate suggested a test for bridges which I call the annulus test. It is a natural generalization of the neighbourhood test from last week in the following way: we consider the closed annulus . If is disconnected for sufficiently many values of and then is a local bridge. The neighbourhood algorithm described above is the special case of the annulus . The biggest problem is choosing the correct values of and . I am currently trying to choose them so that the annulus contains only vertices in clusters (so that there are no parallel bridges).
* I have an idea for a different test that I’d like to try, although I probably won’t write code for it yet. It relies on the Wasserstein distance for persistence diagram given by

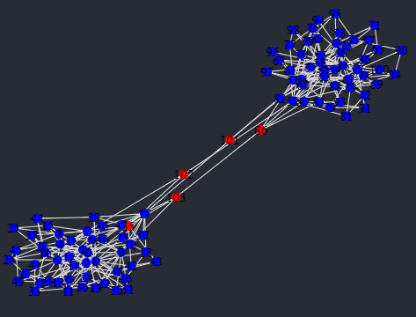
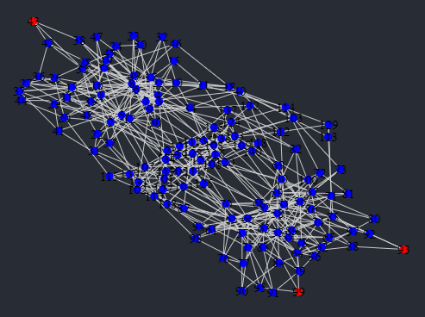
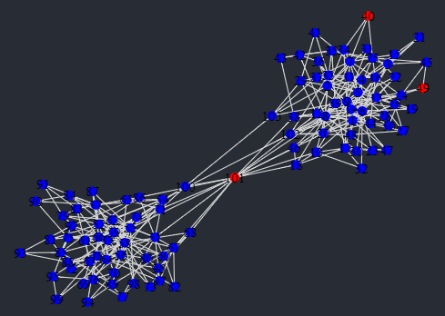
Intuitively, the 1-dimensional persistence diagrams for a bridge and a cluster point should look quite different, and this is reflected in their Wasserstein distance. A cluster point should have a large number of births almost immediately because of the 1-cycles inside the cluster, whereas a bridge should have a period without many births (there may still be some 1-cycles born early because of parallel bridges forming loops, but there will be fewer and this is reflected in the Wasserstein distance). Once we have identified a few points that are definitely in clusters (i.e. finding a point right at the center using some other method), we can use the Wasserstein distance to find nearby points that look similar enough to be cluster points.

* I was taken on a tour of the Australian Signals Directorate facility and discussed my project with some of the staff. On the topic of identifying points in the center of clusters, one of them suggested using a classical graph value called the clustering coefficient. This is defined as . For an undirected graph, we can give a more explicit formula: if a vertex has neighbours, there are possible edges. The clustering coefficient for is thus given by . The clustering coefficient takes values in the range . The closer it is to , the more densely clustered the neighbourhood around is and so the more likely is to be a cluster point.
* I finished implementing the annulus test and tried it on Marian’s example graphs. Strangely, it either identifies the same vertices as the neighbourhood algorithm or it doesn’t identify any at all. I suspect that my choice of and is not optimal, but I can’t think of different values that might work better. I had a friend (Edmund Hofflin) from the computer science department look over my code to see if he could think of any possible optimizations; he suggested changing the way the induced subgraph of the annulus is implemented. I rewrote the code so that it does not rely on pre-existing packages and it is noticeably faster.
* Yossi and I discussed my persistence diagram idea and he agreed that it is theoretically sound. We realized that, when trying to identify vertices that are close to densely clustered points under the Wasserstein distance, we will also have to consider the actual distance between them. This is because the Wasserstein distance cannot tell when a graph is highly symmetric, so two points that are far apart might have similar persistence diagrams.

Week 4 (16.12.2019-20.12.2019)

* I implemented the algorithm based on the Wasserstein distance. The hardest part is identifying the cluster points at the beginning, so I’ll give an explanation of the current method: we start by computing the 1-dimensional persistence barcodes for each vertex based on the shortest distance function . Using this, we can generate a list of how many -cycles are born at each time step. Since we want to find vertices right at the center of clusters, we check to see what proportion of vertices (a parameter which may depend on the graph, call it parameter ) are born before a certain time (call it parameter ). There is a final parameter, which is the upper limit on the Wasserstein distance between two points when trying to find the other cluster points. We will call this parameter .
* I am using the implementation of the Wasserstein distance that already exists in Eirene. There is currently a problem with the code in the library because it cannot compare two persistence barcodes where none of the 1-cycles ever die (the program throws an error). As a temporary patch, I am replacing all of the death times with a large value while Yossi tries to fix the problem. Below are the results of the new algorithm applied to Marian’s example graphs. Red vertices are bridges, blue are cluster points. I have included the values of and to show how they depend on the graph.





* Kate and I met and I showed her some of the above graphs. We talked about the algorithm and she pointed out that the lists of numbers of 1-cycles born at each time step are actually equivalent to the 1-dimensional persistence barcode, in the sense that there is a simple algorithm to convert back and forth between them. This is only true in the case of these graphs because there are no 2-cells and so no 1-cycles ever die. She suggested that I try to work with these lists instead of the persistence barcode so that we do not need to use the Wasserstein distance, since this is the slowest part of the algorithm. She is going to send me some resources about cumulative distribution functions to see if we can use a similar (but more easily computed) metric on these lists to compare the vertices.

Christmas Break (23.12.2019-03.01.2020)

* I completely rewrote the code so that it no longer uses the Eirene package to compute the 1-dimensional persistence barcode. This is because there is a linear-time method of finding this barcode as follows: for sake of simplicity, we assume that our graph is connected. This should not affect which vertices are identified as bridges because being a bridge should only depend on the path-component. We now consider the Euler characteristic . It has been shown that, for a cellular complex such as a graph with vertices and edges, the equality holds. Since counts path-components, this simplifies to and so .
* Yossi fixed the problem of the Wasserstein distance algorithm breaking when presented with infinite death times. He has emailed me code to use in place of the package I imported since his pull request has yet to be accepted. I reran all examples using this new code to see if there was any difference when was used in place of some . As expected, the values and remain the same (since they do not depend on the Wasserstein distance in any way) but the value needed to change slightly from example to example. Overall, I had to use a slightly larger value of to get the same optimal parameter choice.
* Now that the Wasserstein distance is working, I ran some timing tests. It is clear that the slowest part of the code by far is comparing the persistence diagrams. This is because the Wasserstein distance makes use of an algorithm for optimal matchings called the Hungarian algorithm, so if I want to make this code faster I would have to use a different optimal matching algorithm. I do not think this is very related to the homology part of the project so I will probably leave it to Marian unless he or Kate wants me to do it.
* I read the paper *Geometry Helps to Compare Persistence Diagrams* (<https://arxiv.org/abs/1606.03357>) to try to find other methods to increase the efficiency of the algorithm. I have not implemented any of these yet but may do so if I find time. Otherwise, Yossi says that he wants to do this in the future.

Week 5 (06.01.2020-10.01.2020)

* Kate and I met and discussed possible ways we could make the parameters and into something dependent on the graph (since currently I’m entering the parameters by hand). We’ve decided not to look into cumulative distribution functions because it requires a lot of background theory that I don’t have. Instead, we’re investigating some classical graph values to see if they can be related to and . These are:

1. The Laplacian matrix , which can be defined for a simple graph (an undirected graph with no edges from vertices to themselves) by the following formula:

The Laplacian has several useful properties. It is symmetric and positive-semidefinite, so its eigenvalues are all real and nonnegative. Furthermore, its kernel counts the number of path-components in the sense that the multiplicity of the eigenvalue is precisely the number of path-components in the graph. Since we are assuming our graph is path-connected, will always have multiplicity 1.

Related to this is the spectral gap, defined to be the smallest non-zero eigenvalue of . By the above, this will always be the second smallest eigenvalue. The spectral gap measures how well the graph can be bipartitioned, with smaller values of the spectral gap indicating there are only a few edges that need to be removed to divide the graph into two clusters. Below are the values of the spectral gap for the six example graphs from Week 4:

|  |  |
| --- | --- |
| **Graph Number** | **Spectral Gap** |
| 1 | 0.04790486355191601 |
| 2 | 0.32003014018672005 |
| 3 | 0.06929257191853444 |
| 4 | 0.04901961668043608 |
| 5 | 0.6258472922047704 |
| 6 | 0.1625182787552774 |

The clear outlier is graph 5 which is encouraging because this is the hardest graph in which to identify bridges. Even looking at the graph, I am unsure what constitutes a bridge.

The spectral gap can also be interpreted as a measure of the number of bridges between two clusters so it seems reasonable to believe there is a way to relate this to the values and used in the algorithm. However, because the spectral gap measures how well a graph can be partitioned into two clusters, graphs with more than two clusters will have to be treated differently.

The eigenvector corresponding to the spectral gap is called the Fiedler vector and it is often used to help bipartition the graph. Using the Fiedler vector, the vertex can be placed into a cluster by looking at the sign of the entry. It is worth noting that there is some choice here because any nonzero scalar multiple of an eigenvector is still an eigenvector. However, difference in sign is preserved under scaling (i.e. if and have different signs then so do and for any ) so this property is well-defined. It is possible that looking at the Fiedler vector could be another method of identifying densely clustered points (instead of using and ) since ‘larger’ Fiedler entries correspond to vertices closer to the center of clusters. Again, because of the scaling, we will have to be careful about what ‘larger’ means.

1. There is a graph constant called the Cheeger constant. For a set of vertices , let denote the collection of all edges from a vertex in to a vertex in . The Cheeger constant is then given by

The Cheeger constant is strictly positive if and only if the graph is connected. We probably won’t end up using the Cheeger constant because it doesn’t give much information except that, for small values of , there exists a “bottleneck”, meaning there are two large sets of vertices with few edges between them. Although this is related to the existence of bridges, it does not help us identify the bridge vertices in any way.

* We’re also going to see how the current algorithm performs on a graph with more than two clusters, since this will be the more common situation. I’ve contacted Marian about this because his Jupyter code is specifically written to generate graphs with two clusters and bridges between them (with this in mind, I don’t really understand how graph 5 was generated). However, he’s away right now so I won’t be testing that this week.
* Kate suggested I try using superlevelsets instead of sublevelsets. This only requires a small modification to the code: as I remarked in Week 2, for any , needs to be a subcomplex. For superlevelsets, this means that we need to use min instead of max in the definition of on edges. This change means that the persistence diagram for each point is different, although there are still the same number of 1-cycles. Having run the superlevelset code on the example graphs, the algorithm still identifies bridges but different values of and are now required (the same value of was still optimal). It seems like and are good values for the superlevelset algorithm, which seems quite reasonable intuitively.
* Yossi and I spent some time trying to optimize the Wasserstein distance code. There is an improvement over the general algorithm in the particular case of 1-dimensional graphs because 1-cycles never die. This means that there is no need for the sorting implemented in the Wasserstein algorithm. However, we still cannot avoid the need for the Hungarian algorithm, so although this optimization made a noticeable improvement, it did not completely solve the problem.

Week 6 (13.01.2020-17.01.2020)

* I spent a lot of this week trying to make the parameters and depend on the graph in a calculable way i.e. using a predetermined formula that does not require me manually entering values. Following on from last week, I tried using the Fiedler vector to replace the values and . My solution to the problem of what ‘largest’ means was to take those vertices with Fiedler entry greater than the mean of the absolute value. That is, I took those such that . This property is well-defined up to scaling because, if we scale the entire Fiedler vector by some , we still have that

This method works quite well to identify the initial cluster points used in the algorithm. Below is a table showing the vertices identified with this method, those identified by the old method (with an appropriate choice of and ) and the final graph given by this method once a value for has been chosen. Observe that the value for is slightly less than the value from the old algorithm because, in general, more vertices are identified for the Fiedler vector as being cluster points. To see the final graph generated using the old method, look at Week 4 in this Record.

|  |  |  |
| --- | --- | --- |
| **Initial Vertices Using Old Algorithm (with and )** | **Initial Vertices Using Fiedler Vector** | **Final Graph Using Fiedler Vector Vertices (with )** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | Couldn’t generate the final version of this graph because the code was lost. |
|  |  |  |
|  |  |  |

* Overnight, Atom (my IDE) wiped blank all of the code relating to the project, including the backups. According to online forums (e.g. <https://discuss.atom.io/t/atom-crashed-and-my-source-file-ended-up-blank/16133/5>), Atom deletes the local copy of a file before saving the new one. However, if the IDE is interrupted in between these two processes, all copies of the file are lost. A Windows update was performed while I was asleep and it interrupted these processes. I spent all of yesterday (Thursday) and today rewriting the code based on my memory and these records.

