(To be formatted)

Title: The pebble game algorithms for classifying rigidity, a study and an implementation

Author: Brendan Connolly

Project Type: Final Year project

Institution: National University of Ireland Galway

Supervisor: Dr James Cruickshank

Submitted: February 2016

The Pebble Game Algorithms

Introduction

The pebble game algorithms are a family of algorithms that have many applications in the field of (K,L)-sparse and (K,L)-tight graphs (see definitions below). In its simplest form the pebble game can classify graphs as to whether they are sparse or tight or neither. The game has more advanced variants that can achieve more complicated goals. A number of these will be discussed in this report. Code that can run these algorithms, written in the sage environment (a powerful mathematical add-on to python) is provided. This report will also contain proofs that the pebble games can do what they are meant to do!

Definitions

A graph G = (V,E) is a collection of vertices V connected by edges E

A graph G is (K,L)-sparse if each sub-graph G' contains at most ((K*n') – L) edges where n' is the number of vertices in G'

Furthermore, the same graph is (K,L)-tight if it contains exactly ((K*N) – L) edges where N is the number of vertices in G

A graph G is (K,L)-spanning if it can be transformed into a (K,L)-tight graph by the removal of some of its edges

An intuitive way of thinking about K and L

What do K and L specify? To answer this question the mathematical concept of Rigidity is required. A basic explanation of rigidity is as follows:

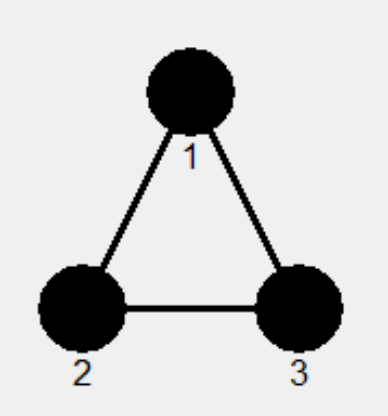


Figure 1

A body can be described as minimally rigid if none of its parts can be moved or rotated independently of the other parts. A simple example of this is the complete graph on three vertices (fig.1) which is rigid for K=2 and L=3. The only way any of the edges/vertices can be moved without breaking the structure is by moving the entire body. A body being minimally rigid is the same as that body being well-constrained.

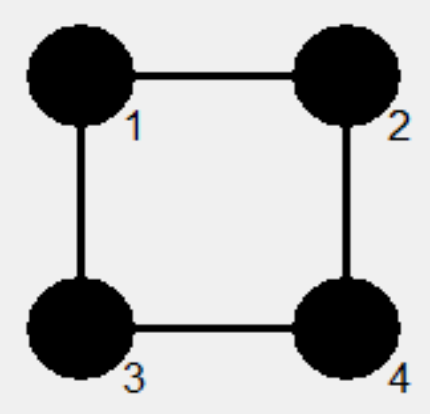


Figure 2

Here is an example (Fig.2) of a body that is not rigid for K=2 and L=3. Vertices 1 and 2 could be moved left or right to form a parallelogram without breaking the original structure. This means that this graph is under-constrained. The graph needs an extra diagonal edge to make it rigid.

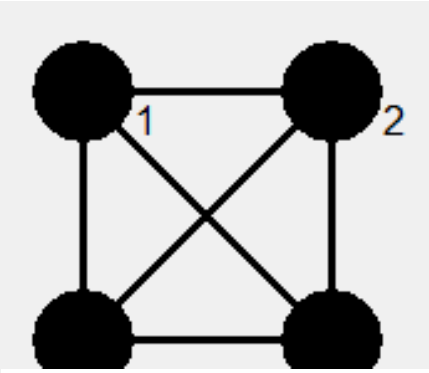


Figure 3

This is an example (Fig.3) of a body that is rigid but has a redundant edge for K=2 and L=3. For example we could remove the edge from vertex 1 to vertex 4 and the body would still be rigid. This means that this graph is over-constrained.

In all of the above examples K has been 2 and L has been 3. That is because K=2, L=3 corresponds to the 2-dimensional plane in which the examples have been drawn. K refers to the number of degrees of freedom a vertex would have if it was independent of any structure (i.e. free). A free vertex would be able to move up and down in the plane, whilst also being able to move left and right. L refers to the natural degrees of freedom that a structure is allowed in a space. In 2-dimensional space a body has 3 natural degrees of freedom; left and right, up and down and also rotation on its axis. So essentially K and L tell us what space we are working in. 1-dimensional space can be captured by K=1, L=1. 3-dimensional space can be captured by K=3,L=6. By varying K and L rigidity can be checked in some (but not all) familiar and non-familiar dimensions. Unfortunately, the nature of the pebble games mean that when L >= 2K, the algorithm no longer works reliably. However as long as L ϵ [0,2K) the pebble games will always work. The reason for this restriction will be discussed later.

The Basic Pebble Game

The pebble game is a one player game with a set of allowable moves. Initially the player has only the Graph (G) they want to test and an empty directed graph (DG) with the same number of vertices as G. As well as this, each vertex on the empty directed graph has k pebbles on it.
The game progresses by analysing each edge of G individually and trying to add those edges to DG using the set of allowed moves. Once each edge has been assessed, a graph can be classified by looking at how many pebbles are left on the graph and whether any edge from G could not be added to DG.

(Note: All moves described below are carried out on DG)
Allowed moves:

1. An edge from vertex i to vertex j can only be added if there are (L +1) pebbles present between i and j
2. If L + 1 pebbles are present then we add a directed edge from vertex i to vertex j and take a pebble off vertex i
3. Otherwise if there are not enough pebbles to add an edge we can perform a depth first search for pebbles on other vertices
4. If a pebble is found, to move it from node i to node j there must be an edge from j to i. We take a pebble from vertex i , reverse the edge so that it goes from i to j and add the pebble onto vertex j.
5. If no pebble can be found to make up the L + 1 condition then that edge is rejected

Output:

When the player has attempted to add each edge (successfully or unsuccessfully) from G onto DG we can get a result like so:

- If there are L pebbles left on the graph and no rejected edges then G is (K,L) tight and thus well constrained.
- If there are more than L pebbles left and no rejected edges then G is (K,L) sparse but not (K,L) tight and thus is under-constrained
- If there are L pebbles left but at least one edge has been rejected then the graph is (K,L)-spanning and thus G is over-constrained
- Otherwise no judgement can be made because the graph is over-constrained in places but is under-constrained in others

Note: Algorithm detail was obtained from (1)

Example Run Through:

This is an example so nice simple graph will be used, 5 connected vertices. The game that will be run is the (1,1) pebble game. So **K = 1 and L = 1**. Please note that the order in which edges are added is arbitrary and makes no difference to the results of the algorithm.

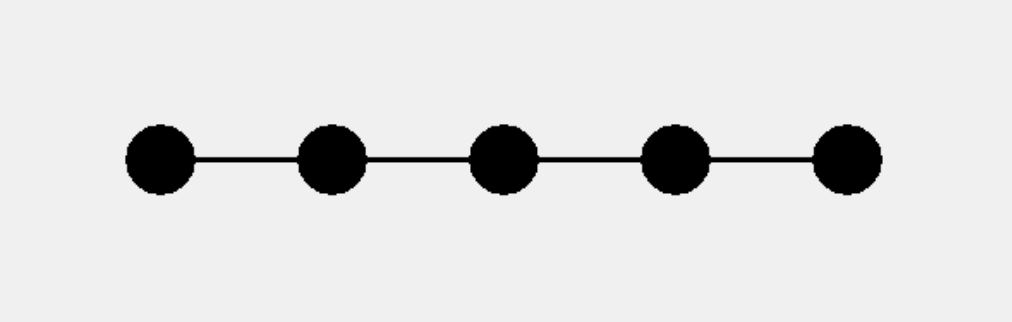
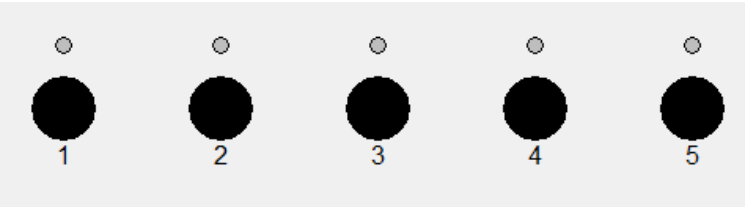
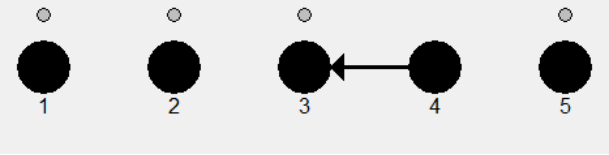


Figure 4

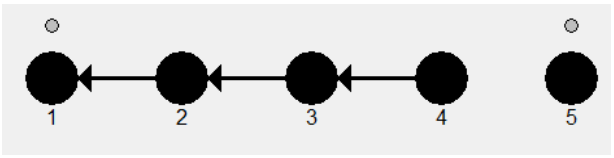
Initially our directed graph looks like this: (Each gray dot is a pebble)



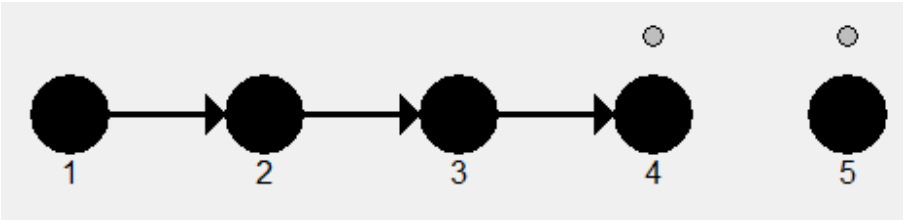
Now try and add an edge between vertex 4 and vertex 3. This can be done since, between them, 3 and 4 have L + 1 = 2 pebbles. When the edge is added the graph now looks like this:



Skipping a few steps and we have a graph that looks like this:



Now the game requires that an edge be added between vertex 4 and vertex 5. This cannot be done since there is only one pebble between 4 and 5. So a search is required to find a pebble. Run a depth first search and this inevitably finds a pebble at vertex 1. The pebble is moved from 1 to 5 reversing edges along the way giving this graph:



Now the final edge can be added so the final state of the graph is:

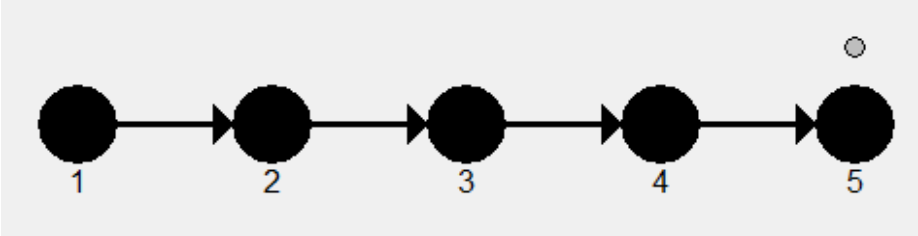
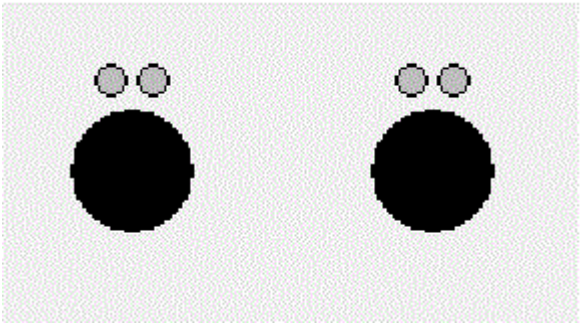


Figure 5

When this graph is analysed it can be seen that there is only L = 1 pebble left. Looking back, no edges had to be rejected so the original graph (fig.1) is (K,L)-tight and thus well constrained. This concludes the algorithm.

What do the allowed moves achieve in the basic pebble game?

Firstly, the edge addition move will be considered. Consider a pebble graph with two edges in the following configuration: (recall the little gray circles are pebbles)



This is a 2 node graph where k=2 (number of pebbles on each node). Let L=3. As discussed previously the pebbles can be thought of as degrees of freedom for an object. Therefore each node above has 2 degrees of freedom. Just like a point in the Cartesian plane, the nodes could move up-down and left-right. Now consider an attempt to add an edge between the 2 nodes. The rules of the pebble game state that this can only be done when there are l+1 pebbles available on the 2 nodes combined. This condition is satisfied and as such an edge can be added like so:

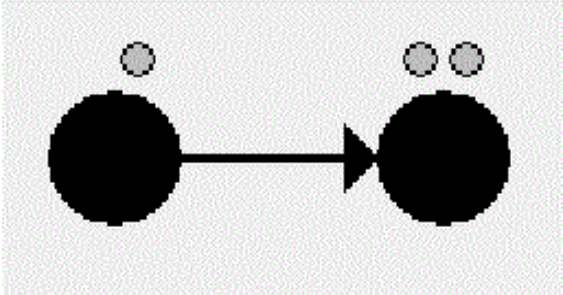
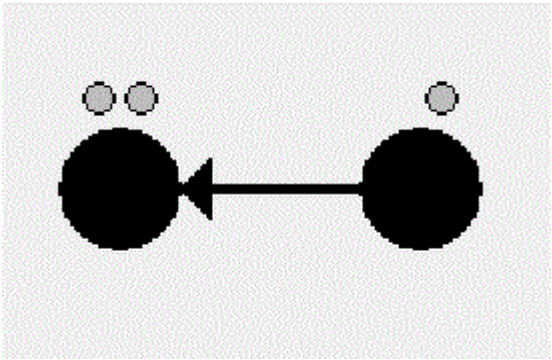


Figure 6

Now, with the added edge the 2 nodes form a structure. There are 3 pebbles (degrees of freedom) in this structure. Just like a rigid structure in the Cartesian plane the above can move left-right, up-down and can rotate. These three degrees of freedom are natural in the Cartesian plane and cannot be removed from a structure. This means that the above structure is minimally rigid.

A nice way of thinking about edge addition is that a pebble is being removed from one of the vertices and is being placed on an edge. (Aside: This way of thinking about edge addition is important when the coloured pebble game is considered later on.) In this way the addition of edges removes degrees of freedom (pebbles) from the overall vertex set but allows the sharing of the degrees of freedom that remain on the vertices.

What about pebble relocation moves? Well, as discussed this is a way of bringing pebbles from where they are plentiful to where they are needed. Again the pebble relocation can be thought of in terms of pebbles on edges. Consider figure 6 above. Say, for example, it was required to move a pebble from the vertex on the right to the vertex on the left. This can be done by moving a pebble from the right vertex onto the edge, reversing the direction of the edge and placing a pebble on the left vertex. This would give the following configuration:



So, the edges allow the sharing of degrees of freedom amongst vertices. The pebble relocation move is the way this sharing is achieved.

On a related note this leads to another question: What do the edge directions mean on the pebble graph? Well, the meaning of the edge directions will be discussed later when the coloured pebble game is described. However, the necessity of having directions on the edges can be examined here. If an edge had no direction then, theoretically, an infinite number of pebbles could be passed over it since rule 4 could fire without limitation. This could give incorrect classifications on the graph.

Consider 2 graphs. Let G1 = K_4^2, that is let G1 be the complete graph on 4 vertices with edge multiplicity 2. This graph is clearly over-constrained because it is the same as figure 3 except with many more redundant edges. Let G2 = C_7, that is let G2 be the cycle graph on 7 vertices. This is just a bigger version of figure 2 so is under-constrained. Now consider a graph G that consists of G1 and G2 with an edge E connecting them. If there were no direction on E (or the other edges in G), all of the spare pebbles on G2 could be moved across it over to G1. This could lead to the graph as a whole to be classified as well-constrained when actually it has an under-constrained part and an over-constrained part cancelling each other out. So the edge directions are vital to the working of these algorithms.

Other Versions of the Pebble Game

The Component Pebble Game

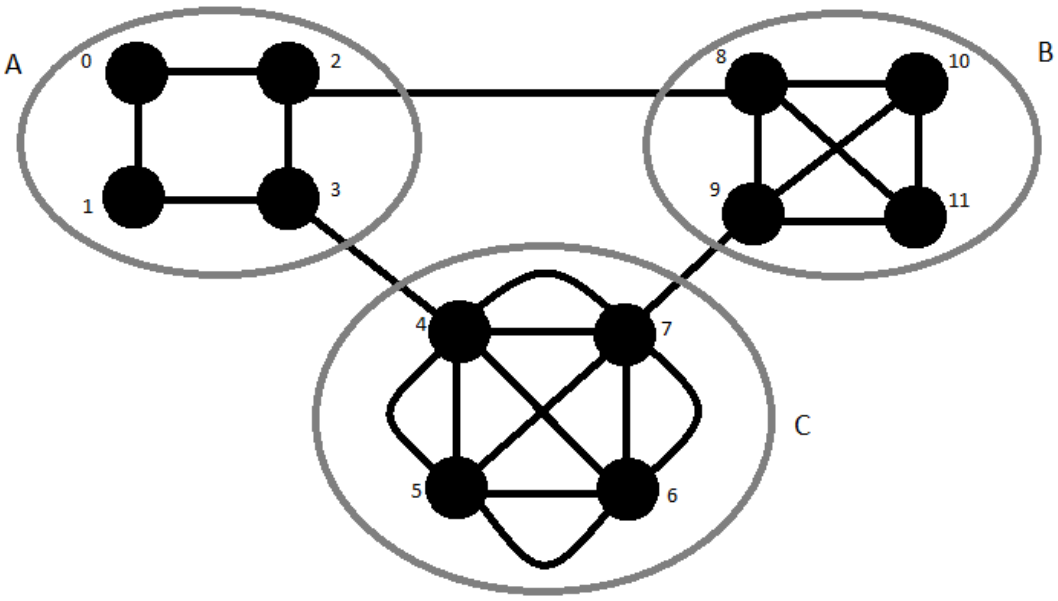
The component pebble game is an improvement on the basic pebble game in two ways. It (a) is more efficient and (b) gives a more in depth output. The component pebble game, as the name suggests, keeps track of components as the algorithm is run. To do this a new routine is added to the pebble game process. Whenever an edge E has been added to the pebble graph perform the following steps

1. Check to see if more than L pebbles can be found by visiting all nodes possible using the directed paths that lead away from the end vertices of E
2. If more than L pebbles can be found the edge is free and no new component has been formed
3. Otherwise a new component has been found and it's vertex set needs to be established
4. Perform depth first searches from all vertices containing at least one pebble and that were not visited during step 1
5. Return the set of vertices that were not visited by these searches

How does this improve efficiency? When a new edge e is being considered for addition to the graph, if the two end-points of the edge are in the same component (i.e. if the edge is dependent), the edge is automatically rejected as being redundant. Otherwise the end-points of e are either in different components or not in a component at all (i.e. the edge is independent) and the edge can definitely be added. This will be discussed in more detail later on when the complexity of these algorithms is examined.

How is the output improved? At the end of the game the algorithm will return its usual classification along with the pebble graph and the components on that graph. This provides an insight into which areas in the graph are causing a particular classification to be returned. For example, if the algorithm returns a verdict of over-constrained, the components will show which region in the graph is causing the problem.

Below is an example of what the component pebble game can do (Note: K=3, L=4):



If the above graph was run through the basic pebble game, it would just give back a rather vague verdict of "other". The component pebble game gives a better output on this graph. For a start it would pick out that regions B and C both contain rigid components of some sort. From this it can be assumed that the rest of the graph (region A) is under-constrained. Another source of information is the list of edges that were rejected. It could be seen in this list that no edge was rejected from component B whilst 4 edges were rejected from component C. This means that component B is a rigid sub-graph and component C is an over-constrained sub-graph. All of this information comes for free with the extra efficiency introduced by the component pebble game.

The Coloured Pebble Game

The coloured pebble game is another variant on the basic pebble game that allows an input graph to be broken down into a number of tree graphs. It is different to the basic pebble game in that each vertex gets k pebbles placed on it at the start but now each pebble on a vertex has a different colour. When an edge is added to the pebble graph it takes on the colour of the pebble that is placed on it (see pebbles on edges here). The trees that the algorithm finds are differentiated by their colour and each tree is the same colour as the pebbles that make it up.

The coloured pebble game uses a special edge addition move and a special pebble relocation move:

1. When adding an edge between to vertices, always use the pebble colour that is most common on the vertices
2. When moving a pebble from one vertex to another never change the colour on an edge so that a monochromatic cycle would be created. It can be shown that this is always possible no matter what situation the pebble graph is in.

These two special pebble moves are called the canonical edge addition and the canonical pebble slide. This coloured pebble game can be used to classify a graph as to whether it is a k-arborescence. A k-arborescence is a graph that can be decomposed into k trees and each vertex is in exactly k of them. An example of the output of a coloured (2,2) pebble game can be seen below:

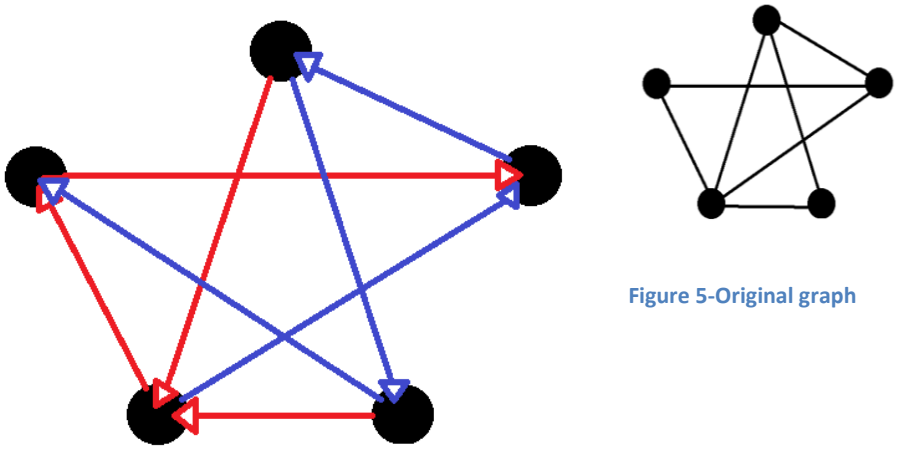


Figure 5-Original graph

Figure 6-Graph after running coloured pebble game

Now the mathematics

This section will be separated into 3 sections:

- The properties of sparse graphs and demonstrations of these properties.
- A proof of the correctness of the pebble game algorithm in the K = 2, L = 2 case.
- A comment on the complexity of the pebble game algorithms

Note: This section is roughly based on parts of [1].

The properties of sparse graphs

Sparse graphs only make sense when L ϵ [0,2K)

When L ≥ 2K the only sparse graphs are empty graphs.

Starting with the definition of a (K, L)-sparse graph: A graph is (K, L)-sparse when each subset n' ≤ n vertices contains at most $Kn' - L$ edges. Suppose $L = 2K$. Now for a graph to be (K, 2K)-sparse for any choice of n' vertices there must be no more than $Kn' - 2K$ edges between those n' vertices. What about when n' = 2? For any n'=2 vertices the sparsity condition says that there can be no more than $2K - 2K = 0$ edges. So any pair of vertices can have at most 0 edges between them. This means that the entire graph has no edges in it. This is the empty graph! For $L > 2K$, $Kn' - L$ becomes negative for n'=2 which is not possible. Two vertices cannot have a negative number of vertices between them!

When L < 0, sparse graphs exhibit strange behaviour. For example two sparse graphs can be combined to give a non-sparse graph.

Take 2 vertex disjoint (K, L)-tight graphs, G1 = (V1, E1) and G2 = (V2, E2). (Recall, a tight graph is a graph that has exactly $Kn - L$ edges.) Each graph satisfies sparsity conditions. Consider $G3 = G1 \cup G2$ where G3 = (V3, E3). This implies the following equations:

$$|E3| = |E1| + |E2| \qquad (1)$$

$$|V3| = |V2| + |V1| \qquad (2)$$

For G3 to be sparse it must have:

$$|E3| = K|V3| - L \qquad (3)$$

However $|E1| = K|V1| - L$and $|E2| = K|V2| - L$
Combining these equations with (1) above gives:

$$|E3| = K|V1| - L + K|V2| - L$$
$$|E3| = K|V1 + V2| - 2L$$

Does this meet the sparsity condition from (3) above? i.e. Is $K|V1 + V2| - 2L \leq K|V3| - L$?
Well, by (2) above the two sides are the same apart from the L terms. So as long as L ≥ 0 the above inequality holds. As soon as L < 0 the inequality fails. This is why it makes sense to never let L be less than 0.

Loops and Parallel edges

A sparse graph may contain at most K – L loops per vertex

Let G be a sparse graph. If a single vertex is considered (i.e. n' = 1), then by the sparsity condition this vertex can span at most Kn' – L = K – L edges. The only edges a single vertex can span is a loop

Sparse graphs have no loops when L ≥ K.

A single vertex can have at most K – L loops. When L ≥ K, this equation goes to zero or becomes negative so no loops can exist

The multiplicity of parallel edges is at most 2K – L.

Parallel edges occur in n' = 2 sub-graphs. By sparsity conditions any n'= 2 sub-graph can contain at most 2K – L edges. Therefore, this is the maximum multiplicity.

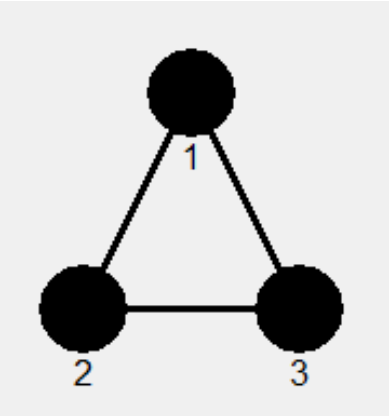
Existence of tight graphs

When L ϵ (K, 2K) there are no tight graphs on a single vertex.

For a single vertex graph G=(V, E) to be tight it must have that: $|E| = K|V| - L$ but |V| = 1 so $|E| = K - L$. But when L ϵ (K, 2K) this equation yields a negative number and no graph can have a negative number of edges so no such graph can exist.

When L ϵ $[\frac{3}{2}K, 2K)$ *there are no tight graphs on small sets of n vertices where* $n \in (2, \frac{L}{2K-L})$.

What happens when you try to build a tight graph in these ranges? Take for example $L = \frac{7}{4}K$. This means that n ϵ (2, 7). Let K = 4 which means L = 7 and let n = 3. The graph on 3 vertices must have exactly Kn – L = 4(3) – 7 = 5 edges. Since L > K this graph cannot contain any loops (see explanation here). Also, the multiplicity of any edge can be at most 2K – L = 2(4) – 7 = 1. This means that the graph can contain no parallel edges (see explanation here). This leaves the following graph:



The above graph has the maximum number of edges that can be added whist it is still (4, 7)-sparse. Any other edge would violate this sparsity. However this graph only has 3 edges. It needs 5 to be tight.
This is exactly what happens whenever one tries to construct a tight graph given the initial specifications on the number of vertices and L. Due to sparsity restrictions, there are not enough edges that can be added to make the result tight.

Correctness of the pebble game algorithms

This section will provide a proof that a graph is sparse if and only if a pebble game run on that graph returns under-constrained. First it will be proved that if a graph was created by the pebble game algorithm then it is sparse, this proof will be separated into two parts as follows:

1. If L+1 pebbles are available on between two nodes i and j then adding an edge between i and j preserves sparsity.
2. One cannot gather L+1 pebbles on two nodes i and j through pebble movements in such a way as to break the sparsity condition if the edge ij was added.

Note as usual below, V refers to the vertex set of a graph and E refers to the edge set of that graph.

1. Proof by induction on the number of edges

Base case:

Let E be the edge set of an arbitrary pebble graph D. Let |E| = 0. This graph is clearly sparse because it has no edges and this cannot break the sparsity condition: $|E'| \leq K * |V'| - L$ where V' and E' are the vertex and edge sets for sub-graphs of D.

Inductive Hypothesis:

Assume that the pebble graph D is sparse with |E| = m edges and that each of these edges has been added by way of a valid pebble graph move.

Inductive case:

Is D still sparse after an edge is added? Let i and j be the nodes that an edge is to be added between. Does $|E'| \leq K * |V'| - L$ still hold? The only sub-graphs that matter are those with i and j in them so all others can be discarded. Now, since there are L+1 pebbles in total on i and j (given), there must be L+1 pebbles on all sub-graphs that contain i and j. V' starts with K*|V'| pebbles and each edge addition causes one pebble to be removed from V'. Since there are L + 1 pebbles left, in the maximum case there are $|E'| \leq K * |V'| - (L + 1)$edges in any sub-graph that contains i and j before the edge is added. Therefore the edge ij can be added and D will remain sparse.

2. Can enough pebbles be added into some V' such that an edge addition breaks the sparsity condition?
First some terminology: Outward edge of V' = Any edge that starts in V' and ends in V/V'
Internal edge of V' = Any edge that starts in V' and ends in V'
Observations: Every pebble that is brought into V' must be brought over some outward edge of V' (a consequence of the pebble game rules). Also, each outward edge can only be used once in this context. Each outward edge consumes 1 pebble from V' and that is one less internal edge that can be formed in V'. These observations reveal that, in order to bring a pebble into V', a pebble must first be removed from V'. Essentially what this means is that the maximum number of pebbles available for internal edge insertion in V' is constant regardless of pebble movements. This maximum number is k*|V'|. This means that the maximum number of internal edges in V' is $|E'| \leq K * |V'| - L$ because no edge can be added when there are less than L+1 pebbles available. Vitally, this maximum number of internal edges is independent of pebble movements. This means that the sparsity condition cannot be broken by pebble movements.

This proves that all graphs created by the pebble game algorithms are sparse/tight and completes the proof in one direction. Now it remains to prove that, when a pebble game is run on a graph it produces the correct output.

Corollary: #pebbles in sub-graph V' + #outward edges of V' is always greater or equal to L

Suppose that this statement wasn't true. Take the case where there are L-1 pebbles on V' and no outward edges. If there are no outward edges then all pebbles in V' were used up for internal edge addition. However, if L-1 pebbles remain then there must be $K * |V'| - L + 1$ edges in V'. This breaks the sparsity condition. It has already been proved that this is impossible. Therefore, this is a contradiction. There has to be at least one outward edge. Applying the same reasoning to the situation with L-1 edges and no pebbles yields a similar contradiction.

Observation: An edge will only be added to a pebble graph if it is independent. The addition of an independent edge to a pebble graph cannot break the sparsity condition.
A few definitions are required here for clarity.

Definition: Independence of an edge
An edge is independent if both of its end points are not in the same block.
Definition: Block
A block in a pebble graph D is a sub-graph D' which has exactly L pebbles present and no outward edges from D' to D\D'.
Definition: reach(v) (v ϵ pebble graph D)
reach(v) is the set of vertices in D that can be visited by following directed edges from v.

Note: From the first 2 definitions it can be seen that a dependent edge is one that cannot possibly gather L+1 pebbles. All other edges (i.e. independent edges) will not break the sparsity condition when added. This proof will show that independent edges can always gather L+1 pebbles.

Proof:

Let u and v be vertices on pebble graph D. Let the number of pebbles on u and v combined be strictly less than L+1. Let e = uv be independent. Let V' be the union of reach(u) and reach(v). Let E' be the number of edges in V'.
The following is true: $|E'| < K * |V'| - L$

Why? Since e is independent, it must have at least L+1 pebbles available to it. These pebbles are located somewhere in V'. k*|V'| is the maximum number of pebbles and each edge uses a pebble. Because of this, there can be no more than $k * |V'| - (L + 1)$ edges in V'.
It is shown (here) that the number of pebbles on V' added to the number of outward edges in V' is no greater than L. But since V' is a union of reaches, there can be no outward edges by definition. This means that the number of pebbles on V' > L.
Recall there at most L pebbles on u and v combined.
This means that somewhere on V' there is a free pebble. What's more is that pebble can definitely be moved onto either u or v since V' is a union of their reaches.
If there are still not L+1 pebbles on u and v then just repeat the same logic until L+1 pebbles have been gathered. Once L+1 pebbles have been gathered the edge can be added.
This shows that all independent edges (i.e. all edges that can be added without breaking the sparsity condition) can and will be added by the pebble game algorithm. Once all of the independent edges have been added all that remains to do is to count the remaining pebbles to obtain a classification for the graph. A tight graph G = (V,E) will have exactly $(K * |V'| - L)$ independent edges so there will be L pebbles remaining on the corresponding pebble graph. In this case G will be classified as well-constrained. A sparse graph G = (V,E) will have $\leq (K * |V'| - L)$ independent edges so there will be ≥L pebbles remaining on the corresponding pebble graph . In this case G will be classified as under-constrained. (Note: When there are exactly L pebbles left, G is both sparse and tight. This does not pose a problem since tightness is a special case of sparsity.) And so on and so forth.

Complexity of the pebble game algorithms

The job of the pebble game algorithms is to check if a given graph is sparse, tight, etc... Take sparsity for example. From before, for a graph G = (V, E) to be sparse, for each subset of vertices V', $|E'| \leq K * |V'| - L$ (where |E'| is the number of edges spanned by V'). The easiest and most obvious way to check this condition is to take every subset of vertices from G and simply count the number of edges in each. If each subset satisfies the above inequality then G must be sparse. However, this method is not efficient at all. Given that there are 2^n different sub-graphs in a graph (where n is the number of nodes), n can't even reach triple figures before this method becomes untenable. On a 100 node graph there are $1.27*10^{30}$ different sub-graph combinations. Even if it took just 1 operation to check the number of edges in a sub-graph, a processor, performing $1*10^9$ calculations per second would take $1.27*10^{21}$ seconds to give an answer. This is 1000x longer than the estimated age of the universe. In one day (86400 seconds) this algorithm could just about analyze a 46 node graph. So clearly a better algorithm is needed.

This is where the pebble game comes in. The pebble game algorithm looks at the number of degrees of freedom left on a graph after all edges have been added (each edge addition removes one more degree of freedom) and thus can tell if the graph is sparse or not. The complexity of the algorithm is dependent on two things: (a) the number of edges added and (b) the number of depth first searches needed to add an edge. (a) The graph starts with k*n pebbles in total but since every added edge requires one pebble to be taken off the graph the number of edges added is **O(kn)** (1).
(b) In the worst case it will take L+1 depth first searches for each edge to be added. The worst case complexity for a depth first search is **$O(n+m_a)$** (n= #vertices, m_a= #edges on pebblegraph). However by (1) m_a = **O(kn).** So DFS = **O(n+kn) = O(n)** because k is a constant. Since there are L + 1 depth first searches required, for a single edge the complexity is **O(L*n)**. However there are m edges to be added so the final time complexity is **O(L*n*m).**

Given a dense graph with m = **$O(n^2)$** edges, there will be **O(kn)** edges accepted (from above) but also **$O(n^2 - kn)$** edges rejected. This means that, in the worst case, all n^2 edges will require L+1 depth first searches to be done to look for pebbles. This gives a worst case time complexity of **$O(L*m*n) = O(L*n^2*n) = O(n^3)$** since L is constant. This worst case complexity is much better than the naive approach that was examined above. Now with $1.27*10^{21}$ seconds this algorithm could process a graph with $1.08*10^{10}$ vertices rather than just 100. In one day (86400 seconds) this algorithm could process a network with 44208 vertices in it compared to 46 for the naive algorithm described above.

This performance is improved again by using the component pebble game. In this variation of the algorithm, a decision is made instantly as to whether an edge will be accepted or not. This removes a lot of needless depth first searches for edges that are, ultimately, going to be rejected. The way this is done is the use of rigid components in the graph (more detail on how this works is available here).
For complexity this means that, given a graph with **$O(n^2)$** edges, there will only need to be depth first searches done for **O(kn)** edges rather than **$O(n^2)$** edges. This gives a worst case complexity of **$O(L*m*n) = O(L*k*n*n) = O(n^2)$** since k and L are constants.
In $1.27*10^{21}$ seconds this algorithm can process a graph with $3.56*10^{15}$ vertices. In one day (86400 seconds) this algorithm could process a graph with $9.30*10^{6}$ vertices, a huge improvement on the naive algorithm described above.

Open problem

When L>=2K, for reasons that are beyond the scope of this project, the relationship between sparsity and rigidity begins to break down. It is this relationship that the pebble games rely upon. The biggest consequence of this is that the algorithm does not work for K=3, L=6. These are the values that describe 3-dimensional space. The point is that these algorithms fail when dealing with 3D space. This is the one glaring weakness of these algorithms. If they could be used reliably in 3 dimensions they would be very useful. Efficient rigidity checking for real world structures and complex biological molecules (as per [5]) could be achieved. These are two huge areas that could benefit hugely from a solution to this problem. Some progress has been made on this front [4], however as far as can be told, no-one has come up with a full solution.

Another problem in this area seems to be that there is no openly available library that can be used to run these algorithms. The implementations provided in the appendix perform their task efficiently and reliably however they lack a professional coder's touch. Even without being able to handle 3D rigidity these algorithms are useful and having a good implementation available would be good for all those interested in this area.

Appendix-The Code

Note: Apologies for any indentation errors, the importing of this code from the sage virtual machine was not very smooth. I tried to catch most errors but some may remain!

Basic Pebble Game code:

Note: Algorithm Description can be found in [1].

```
class pebbleGraph(sage.graphs.digraph.DiGraph):
      '''
      An object that allows the pebble game algorithms to be run on a graph

      N (int) Number of nodes in the subject graph
      k (int) Number of pebbles on each node
      l (int) l+1 is the number of pebbles needed on two nodes in order to add
an edge between them

      Creating an instance of this class will provide an empty N node graph with
k pebbles on each node.
        The methods provided will allow a pebble game to be run. The user needs
to specify the order in which the
        edges from the subject graph are added to this pebble graph.

        Public Methods:
        get_pebble_layout()
        can_add_edge(i,j)
        add_edge(i,j)
        get_rejected_edges()
        reject_edge(i,j)
        find_pebble(i,j)
        move_pebble(path)
        attempt_add_edge(i,j)
        output()

      This class inherits from sage.graphs.digraph.DiGraph
        This class is the base for componentGraph and colPebbleGraph classes
      '''
      def __init__(self,N,k,l):
        #k is the number of pebbles on each node
        self.k = k
        #l+1 is the number of pebbles we need on two nodes in order to add an
edge between them
        self.l = l
        self.N = N
        #pebbleList keeps track of where the pebbles are in our graph
        self.pebbleList = []
        #Initializes the directed graph that the algorithm works on.
        sage.graphs.digraph.DiGraph.__init__(self, multiedges = True)
        for i in range(N):
            self.add_vertex()
            #Put k pebbles on each node
            self.pebbleList.append(self.k)
        #We will need to keep track of any rejected edges so we can classify our
graph when the algorithm has run its course
        self.rej_edge_list = []

      #Checks the condition for adding an edge
      def can_add_edge(self,i,j):
              '''Returns True if an edge can be legally added between i and j,
False otherwise'''
```

```
            if (self.pebbleList[i] + self.pebbleList[j]) >= (self.l + 1):
            return(True)
        else:
            return(False)

    #Adds an edge to directed graph and updates pebbleList
    def add_edge(self,i,j):
            '''Adds an edge from (int)i to (int)j and performs relevant pebble
maintenance on the graph'''

            if self.pebbleList[i] == 0:
           print("Cannot add an edge from", i, "to", j, "not enough pebbles on",
i)
        else:
           sage.graphs.digraph.DiGraph.add_edge(self,i,j)
           self.pebbleList[i] = self.pebbleList[i] - 1

    #Adds an edge to the rejected list
    def reject_edge(self,i,j):
            '''Adds the edge from (int)i to (int)j into the list of rejected
edges'''

            self.rej_edge_list.append((i,j))

    #Tests the resultant graph after the algorithm has run and reports to the
         user what the classification of their graph is
    def output(self):
            '''
            Provides a rigidity classification about the subject graph

            This method will only give useful output if the user has first
attempted to add all edges from the subject graph into the pebble graph
            '''

        print("Caution: This function will not give a correct output unless all
edges from the original graph have been considered for addition to the pebble
graph. If you are running the runPebbleGame() function, disregard this
warning.")
        print("Your graph can be classified as: ")
        if sum(self.pebbleList) == self.l:
           if self.rej_edge_list == []:
            print("Well constrained")
           else:
              print("Over constrained")
        else:
           if self.rej_edge_list == []:
            print("Under constrained")
           else:
            print("Other. This means that, overall, your graph is under
contrained. However it contains a component that is over constrained")

    #Moves a pebble along a path, reversing edges and updating the pebbleList
as it goes
    def move_pebble(self,path):
            '''
            Moves a pebble along a (list<int>) path

            The path leads from where a pebble was found to where it is
needed.  This method takes a pebble along that path and updates pebble positions
along the way.
            '''
```

```
            for i in range(len(path)-1):
          j = i + 1
          self.pebbleList[path[j]] = self.pebbleList[path[j]] + 1
          self.delete_edge(path[j],path[i])
          self.add_edge(path[i],path[j])

    #If trying to add an egde but not enough pebbles are present, this method
will find the path to a pebble using a depth first search or will return that no
pebble could be found
    def find_pebble(self,i,j):
          '''Finds a pebble to add to either node (int)i or node (int)j in
order to make up l pebbles between i and j'''
            visited = [i,j]
      #We can't add a pebble if a node(i) already has k pebbles present, so we
try to find a pebble for node j
      if self.pebbleList[i] == self.k:
         path_to_pebble = self._DFS(j,visited)
      #Similar to previous comment
      elif self.pebbleList[j] == self.k:
         path_to_pebble = self._DFS(i,visited)
      else:
         #This is where neither node has k pebbles present so we search both
nodes in turn
         path_to_pebble = self._DFS(i,visited)
         if path_to_pebble == []:
          path_to_pebble = self._DFS(j,visited)
         else:
          pass
      return(path_to_pebble)

    #This is the method that does the actual searching for pebbles, i is the
node that needs a pebble, visited will be a list with i and j in it where i and
j are the nodes we are trying to add an edge between
    def _DFS(self,i,visited):
      #pebble_at_vert will tell us where we have found a pebble
      pebble_at_vert = "none"
      #reconstruct will hold the information that allows us to navigate back
to node i when we find a pebble
      reconstruct = []
      #Initially populate reconstruct with "none" strings
      for d in range(len(self.vertices())):
          reconstruct.append("none")
      #The stack will tell us where we need to visit next
      stack = []
      #Initial population of stack with out_neighbours of i
      for vert in self.neighbors_out(i):
          #If vert is alredy in stack then we have a multi edge and we don't
need another copy of vert in the stack
          if vert not in stack and vert not in visited:
                stack.append(vert)
           #Recoding that we came to vert from i so that's where we return  to
when retracing our path
                reconstruct[vert] = i
          else:
           pass
      #When the stack is empty this means we have visited all nodes
    discoverable from i and we can stop the search
      while len(stack) > 0:
          v = stack.pop()
          #check that we have not visited v already
          visited.append(v)
```

```
            #If the below condition is met then we have found a pebble
            if self.pebbleList[v] > 0:
             pebble_at_vert = v
             #No need to keep searching so break the while loop
             break
            #If we don't find a pebble at v then we search all of v's out
neighbours
            else:
             for w in self.neighbors_out(v):
                 if w not in stack and w not in visited:
                  stack.append(w)
                  #We got to w from v so that's the way we need to go if we
wanted to find our way back to i
                  reconstruct[w] = v
                 else:
                  pass
        #If we have performed our search and pebble_at vert doesn't have a value
then there is no pebble available and we return an empty path
        if pebble_at_vert == "none":
            return([])
        #Otherwise we have found a pebble and we can reconstruct our path from
pebble_at_vert to i using our reconstruct list
        else:
            path = []
            next_node = pebble_at_vert
            while next_node != "none":
                   path.append(next_node)
             next_node = reconstruct[next_node]
            return(path)

      #This method takes two vertices and tries to add an edge between them by
finding pebbles etc... It will reject the edge if too few pebbles are found
      def attempt_add_edge(self,i,j):
        '''
        Performs an edge addition without the user having to run the pebble
movement/finding methods.

        If edge ij can be added to the pebble graph, this method performs all
necessary pebble movements in order to add the edge then adds the edge,
otherwise rejects the edge
        '''

        layout = self.get_pebble_layout()
        #Sometimes all L+1 pebbles are on node j. In this case the following
clause makes sure that we never add an edge ij where there are no pebbles on i
        if (self.can_add_edge(i,j)) and layout[i] == 0:
           temp = i
           i = j
           j = temp
        else:
           pass
        #While we cannot add an edge, keep finding and moving pebbles
        while not (self.can_add_edge(i,j)):
           pathToPebb = self.find_pebble(i,j)
           #If, at any stage, no pebble is found edge ij needs to be rejected
           if pathToPebb == []:
            break
           else:
            self.move_pebble(pathToPebb)
        if (self.can_add_edge(i,j)):
           self.add_edge(i,j)
```

```
        else:
           self.reject_edge(i,j)

      #Accessor methods

      def get_pebble_layout(self):
            '''
            Returns the number of pebbles on each node in list form

            For example: [1,2,3] there is 1 pebble on node 0, 2 pebbles on node
1 and 3 pebbles on node 2.
            '''

            return(self.pebbleList)

      def get_rejected_edges(self):
              '''Returns a list of rejected edges'''

              return(self.rej_edge_list)
```

Coloured Pebble Game code:

Note: Algorithm Descriptions can be found in [2].

```
load("pebbleGraph.sage")
from collections import Counter
from sets import Set

class colPebbleGraph(pebbleGraph):
      '''
      An extension to the basic pebble game object that allows for
decompositions of graphs into trees

      N (int) Number of nodes in the subject graph
      k (int) Number of pebbles on each node
      l (int) l+1 is the number of pebbles needed on two nodes in order to add
an edge between them
      colours (list<string>) List of colours that the pebbles can take in the
pebble game

      Creating an instance of this class will provide an empty N node graph with
k pebbles on each node.  Each of these k pebbles will have a different colour.

      The coloured pebble game runs on the same rules as the basic pebble game
however each pebble now has a colour.  As a result, this algorithm can produce a
decomposition of a graph into trees as well as providing a rigidity
classification of the graph.  As long as the graph is suitable to run a pebble
game on this class can find a tree decomposition.  This is achieved through the
use of colours on pebbles and special add_edge + find_pebble algorithms.

      Public Methods:
      get_pebble_colours()
        change_labels()
        use_custom_labels()
        add_edge(i,j)
        move_pebble(path)
        find_pebble(i,j)
        output()

      This class inherits from the pebbleGraph class
```

```
      '''

      #Constructor method for colPebbleGraph
      #N, k, l are the standard inputs for a pebble game
      #colours are a set of labels that are used to make the end result more
human readable
      def __init__(self, N, k, l, colours = ["blue", "red", "green", "yellow",
"orange", "white", "black", "gray", "purple", "pink"]):
        pebbleGraph.__init__(self, N, k, l)
        self.labels = colours
        #pebbleCols will track which pebbles are where on the graph. While the
algorithm runs it is populated with numbers, the user can then change the output
to be the colour labels if they wish
        self.pebbleCols = []
        #Put a pebble of each colour(number) on each vertex
        for i in range(self.N):
            self.pebbleCols.append([])
            colNum = 0
            for j in range(self.k):
                    self.pebbleCols[i].append(colNum)
                    colNum += 1

      #An extension to the pebbleGraph add_edge. Technically called the
'canonical add edge', this method will only add edges that minimize the chance
of a monochromatic cycle being created later on
      #NB: This function assumes that adding an edge between i and j is a legal
move and will not check this
      def add_edge(self, i, j, colour = None):
          '''
          Adds an edge from node (int)i to node (int)j that will not create a
cycle

          This method over-rides pebbleGraph.add_edge(i,j) but extends
sage.graphs.digraph.DiGraph.add_edge(i,j)
          '''

          #If node i has no pebbles on it, we cannot add an edge
        if self.pebbleList[i] == 0:
           print("Cannot add an adge from node", i, "to node", j, ", not enough
pebbles on node", i, "Try adding an edge from", j, "to", i)
        #If no colour is specified then we find the most common colour between
nodes i & j and then add the edge in that colour
        elif colour == None:
           seti = Set(self.pebbleCols[i])
           setj = Set(self.pebbleCols[j])
           common = seti & setj
           if len(common) == 0:
            label = Counter(self.pebbleCols[i]).most_common(1)[0][0]
           else:
              label = common.pop()
           sage.graphs.digraph.DiGraph.add_edge(self, i, j, label)
           self.pebbleCols[i].remove(label)
           self.pebbleList[i] = self.pebbleList[i] - 1
        #Otherwise it is a simple case of adding the edge in the specified
colour
        else:
           sage.graphs.digraph.DiGraph.add_edge(self, i, j, colour)
           self.pebbleCols[i].remove(colour)
           self.pebbleList[i] = self.pebbleList[i] - 1
```

```
      #Moves a pebble along a set of nodes and a updates edge colours +
directions accordingly
      #pathInfo contains a list of vertices(the path) and the colour of pebble
that should be moved on each vertex
      def move_pebble(self, pathInfo):
        '''
        Moves a pebble along a (list<int>) path

        The path leads from where a pebble was found to where it is needed.
This method takes a pebble along that path and updates pebble positions and edge
colours along the way.

        This method over-rides pebbleGraph.move_pebble(path)
        '''

        path = pathInfo[0]
        col = pathInfo[1]
        for i in range(len(path)-1):
            j = i + 1
            #Move pebble of the edge and onto vertex j
            self.pebbleCols[path[j]].append(col[path[j]])
            self.pebbleList[path[j]] += 1
            #Reverse the edge
            self.delete_edge(path[j], path[i], col[path[j]])
            self.add_edge(path[i], path[j], col[path[i]])

      #Method that finds a 'canonical path' to a pebble. This is a path that
avoids the creation of a monochromatic cycle in the pebbleGraph
      #Returns a list with a path and the pebble colours corresponding to that
path(see the 'colours' dictionary in the code)
      def __find_can_path(self, i, visited):
        #Start off with a standard pebbleGame depth first search to find a
pebble
        path = self._DFS(i,visited)
        #If the DFS fails then there is no pebble, return this result
        if path == []:
           return([path, {}])
        #Otherwise we have found a path, now we make it into a canonical one
        else:
           #Initialise the colours dictionary
           colours = {}
           #The reconstruct dictionary will allow us to re-trace the path once
it is canonical
           reconstruct = {}
           reconstruct[path[0]] = "end"
           #The colour on the first node is arbitrary
           colours[path[0]] = self.pebbleCols[path[0]][0]
           for i in range(len(path)):
             j = i - 1
             #If the colour of node i is already set we don't have to worry
about it
             if path[i] in colours:
                pass
             #If the colour of node j is the same as the edge ij then ij will
not create a monochromatic cycle
             elif colours[path[j]] in self.edge_label(path[i], path[j]):
                #Set i as the next node in the path and set the colour of i
                reconstruct[path[i]] = path[j]
                colours[path[i]] = colours[path[j]]
             #Otherwise using edge ij could create a monochromatic cycle, we
have to find a way around this
```

```
            else:
               #In case we want to return to the previous configuration(this
can happen if we have no choice but to accept edge ij in the path)
               copcolDict = dict(colours)
               coprec = dict(reconstruct)
               #Try to find a new path that doesn't use ij and has the same
colour as node j
               newPath =
self.__connect_paths(reconstruct,colours,colours[path[j]],path[i])
               #If we could not find a good work around we revert to using edge
ij
               if newPath == []:
                  colours = copcolDict
                  reconstruct = coprec
                  colours[path[i]] = self.edge_label(path[i], path[j])[0]
                  reconstruct[path[i]] = path[j]
               else:
                  pass
         curNode = path[-1]
         returnPath = []
         #Reconstruct the path
         while curNode != "end":
             returnPath.append(curNode)
             curNode = reconstruct[curNode]
         #The path is reconstructed in reverse so undo this by reversing again
         returnPath.reverse()
         return([returnPath, colours])

   #Exactly the same as pebbleGraph.find_pebble except it uses the
find_can_path method rather than the find_path method
   def find_pebble(self, i, j):
     '''
     Finds a path to a pebble to add to either node (int)i or node (int)j
that will not create a cycle in the graph

     This method finds a "canonical" path through the graph to a pebble. This
is just a path that creates no monochromatic cycles in the graph

     This method over-rides pebbleGraph.find_pebble
     '''

     visited = [i,j]
     if self.pebbleList[i] == self.k:
        pathInfo = self.__find_can_path(j,visited)
     elif self.pebbleList[j] == self.k:
        pathInfo = self.__find_can_path(i,visited)
     else:
        pathInfo = self.__find_can_path(i,visited)
        if pathInfo[0] == []:
         pathInfo = self.__find_can_path(j,visited)
        else:
         pass
     return(pathInfo)

   #Changes a section of path from being non canonical to canonical
   def __connect_paths(self,reconstruct,colours,colour,startNode):
     curNode = startNode
     #newPath keeps track of the path that we find
     newPath = [startNode]
     #nextNode tells us if we can go on with our search for a canonical
section
```

```
        nextNode = None
        #Give nextNode an initial value
        for u in self.neighbors_out(curNode):
            if colour in self.edge_label(curNode, u):
                   nextNode = u
            else:
             pass
        #If nextNode is None then there is no vertex that we can go to from
curNode so we stop. If nextNode is in colours then we may have found a new
canonical section
        while nextNode != None and nextNode not in colours:
            nextNode = None
            #Check each out edge of curNode, if it's colour is the one we are
searching for then move along that edge, otherwise do nothing
            for v in self.neighbors_out(curNode):
             if colour in self.edge_label(curNode, v):
                newPath.append(v)
                curNode = v
                nextNode = v
                break
             else:
                pass
        #If the last node on our new section of path has already been
encountered in __find_can_path(if it's in colours) then it may be our answer
        if newPath[-1] in colours:
           #If the colour of the last edge matches the last node then we have
found our new canonical section
           if colours[newPath[-1]] in self.edge_label(newPath[-2], newPath[-1]):
            #Update the path and colours dictionaries
              for i in range(len(newPath)-1):
                j = i + 1
                reconstruct[newPath[i]] = newPath[j]
                colours[newPath[i]] = colours[newPath[i-1]]
              return(newPath)
           #Otherwise the new path is still not canonical, we need to go again
by recursively calling connect_paths
           else:
              morePath =
self.__connect_paths(reconstruct,colours,colour,newPath[-2])
            #If the next layer down could not find a path then no path is
possible, this is fed back up the line
            if morePath == []:
               return([])
            #We can update colours and reconstruct and feed the combined newPath
and morePath up the line
            else:
               for i in range(len(newPath)-1):
                   j = i + 1
                   reconstruct[newPath[i]] = newPath[j]
                   colours[newPath[i]] = colours[newPath[j]]
               #newPath[-1] is the same as morePath[0] so we can exclude
morePath[0] in the returned path
               return(newPath + morePath[1:])
          #Otherwise no new path was found, return this to the layer above
        else:
           return([])

      #This method changes all of the edge labels and pebble colours into words
rather than numbers. Should only be called when the algorithm has run its course
or before the algorithm runs(less efficient)
      def change_labels(self):
```

```
        '''
        Changes the colour labels on edges and labels on pebbles on vertices
into strings

        The strings are just text representations of colour such as 'blue',
'green' etc. rather than integers
        Note: This should be done after the pebble game has been run since
comparing integers is much more efficient than comparing strings.
        '''
        if self.k > 10:
           print("Too many pebble colours for the default label set, cannot
convert labels (default set has 10 colours in it). Please provide a set of
colours by using the use_custom_labels() method")
           return(None)
        elif self.allows_multiple_edges() == True:
           print("Cannot change labels into strings due to multi-edges in
graph")
           return(None)
        else:
           pass
              if self._can_change_labels() is True:
                 for e in self.edges():
                   i = e[0]
                   j = e[1]
                   self.set_edge_label(i,j,self.labels[self.edge_label(i,j)[0]])
           for i in range(len(self.pebbleList)):
             for j in range(self.pebbleList[i]):
                 self.pebbleCols[i][j] = self.labels[self.pebbleCols[i][j]]
        else:
           print("Cannot change labels twice")

      #Returns true if the labels in the pebble graph have already been made
into words, false otherwise
      def _can_change_labels(self):
        '''
        Return true if the labels can be changed to their string forms, false
otherwise

        The labels cannot be changed into string form because either: (1) They
are already in string form or (20) There are parallel edges in the graph the
set_edge_label(i,j) method cannot handle
        '''

        edges = self.edges()
        if edges == []:
           if type(self.pebbleCols[0][0]) is str:
            return(False)
           else:
            return(True)
        else:
           edge = edges[0]
           if type(self.edge_label(edge[0],edge[1])[0]) is str:
            return(False)
           else:
            return(True)

      def output(self):
        '''Outputs the rigidity classification of the input graph and the final
configuration of the edges'''
        pebbleGraph.output(self)
        print("The final edge layout of the game was:")
```

```
        print(self.edges())

      #Accessor method for pebble colours
      def get_pebble_colours(self):
        return(self.pebbleCols)

      #Method to use your own colour labels
      def use_custom_labels(self, colours):
        '''
        Changes from the default labels to labels of the user's choice

        Useful when dealing with a pebble game with K > 10 (there are only 10
default labels)
        '''
        self.labels = colours
```

Component Pebble Game Code:

Note: Algorithm Descriptions can be found in [1].

```
load("pebbleGraph.sage")
from sets import Set

class componentGraph(pebbleGraph):
      '''
      This is an extension to the basic pebble game in which the algorithm keeps
track of rigid components as it runs the pebble game.

      N (int) Number of nodes in the subject graph
      k (int) Number of pebbles on each node
      l (int) l+1 is the number of pebbles needed on two nodes in order to add
an edge between them

      Creating an instance of this class will provide an empty N node graph with
k pebbles on each node.

      The component pebble game runs on the same rules as the basic pebble game
except that now, rigid components are kept track of.  Not only does this give a
better breakdown of results once the pebble game has run, it also makes for a
more efficient algorithm.

      Public Methods:
            get_components()
            get_component_matrix()
            detect_component(i,j)
            update_components(newComponent)
            attempt_add_edge(i,j)
            output()

      This class inherits from the pebbleGraph class
      '''

      def __init__(self, N, k, l):
        #Initialise all class variables from the pebbleGraph class
        super(componentGraph,self).__init__(N, k, l)
        #For more info on component tracking schemes se update_component()
        #When l=0 there is only ever 1 components so we employ a simple
component tracking scheme
        if self.l == 0:
           self.component = []
           for i in range(self.N):
```

```
                self.component.append(0)
        #When l<=k there can be many components but they are non-overlapping,
again we use a simple tracking scheme
        elif self.l <= self.k:
           self.components = []
           self.componentNum = 0
           #Initialise all vertices to be in component -1 (i.e. no component)
           for i in range(self.N):
               self.components.append(-1)
        #Otherwise we can have many components that may overlap at a single
vertex
        else:
           self.components = []
           #for efficiency we store a matrix that allows us to look up if 2
vertices are in the same component
           self.compMat = []
           #Initialise all matrix entries to 0
           for i in range(self.N):
             self.compMat.append([])
             for j in range(self.N):
                 self.compMat[i].append(0)

      #This method takes a vertex(v) and calculates which vertices can be
visited by follwing edges away from v
      def _reach(self, v, reverse=False):
        D = DiGraph(data = self.adjacency_matrix())
        #Sometimes we need to search the pebbleGraph in reverse
        if reverse == True:
           D = D.reverse()
        else:
           pass
        #Reachable is a set because we need to perform set operations on it down
the line
        reachable = Set([v])
        #s will be our 'to visit' stack
        s = D.neighbors_out(v)
        #Essentially we perform a depth first search and return the visited
nodes
        while len(s) > 0:
            u  = s.pop()
            reachable.add(u)
            for i in D.neighbors_out(u):
              if i not in reachable:
                 s.append(i)
              else:
                 pass
        return(reachable)

      #This method takes two vertices(i,j) and figures out whether the newly
added edge between them changes the component structure of the graph
      def detect_component(self, i, j):
            '''
            Checks for the creation of a rigid component due to addition of edge
(int)i and (int)j

            This method will return any component found or will return an empty
list if none is found
            '''
        #If this condition is met then the edge is free and doesn't change
component structure
        if self.pebbleList[i] + self.pebbleList[j] > self.l:
```

```
            return([])
         else:
            reachij = self._reach(i) | self._reach(j)
            total = 0
            #This loop computes the total number of pebbles on reachij
            for w in reachij:
              total += self.pebbleList[w]
            #If this total is > l then component structure is unchanged
            if total > self.l:
              return([])
            else:
              #Otherwise we have changed the component structure and this code
picks out this new component
              V = Set(self.vertices())
              #Set theory: V\reachij
              V = V - reachij
              visited = Set()
              #V is the set of vertices that cannot be reached by following edges
from i or j
              for w in V:
                  #We only perform the limited Depth first search on vertices
with a free pebble
                  if self.pebbleList[w] > 0:
                   temp = self._reach(w,True)
                   #Set theory: visited UNION temp
                   visited = visited | temp
                  else:
                     pass
              V = Set(self.vertices())
              #V\visited is the set of vertices that forms the new component
              return(V - visited)

      #This method takes a newly formed component and does the necessary
maintenance of the component tracking structures
      def update_components(self, newComponent):
            '''
             Performs maintenance on the component tracking structures of the
component graph

             newComponent (list<int>) List of the elements of a newly formed
component
             '''
         #The main condition statements are explained in the constructor
         if self.l == 0:
            for v in newComponent:
              #If a vertex is in the component it is marked with a 1. All
vertices not in the component remain as 0
              self.component[v] = 1
         elif self.l <= self.k:
            for v in newComponent:
              #Each vertex is given a label corresponding to which component it
occurs in. Those vertices with label -1 are free(no component)
              self.components[v] = self.componentNum
            self.componentNum += 1
         else:
            #components is a list of components. Each component is represented by
the vertices that are present in it
            #compMat stores info on whether 2 vertices are in the same component.
compMat[i][j] = 1 if i and j are in the same component, 0 otherwise
            toRemove = []
```

```
            #This loop works out which old components are simply subsets of the
new component. Such components can be forgotten about
            for oldComponent in self.components:
              if oldComponent <= newComponent:
                 toRemove.append(oldComponent)
              else:
                 pass
            for oldComponent in toRemove:
              self.components.remove(oldComponent)
            self.components.append(newComponent)
            #Update the compMat
            for i in newComponent:
              for j in newComponent:
                  self.compMat[i][j] = 1
                  self.compMat[j][i] = 1

      #This method is an extension to the pebbleGraph output method and simply
outputs information on the components to the user
      def output(self):
            '''
            Looks at the final graph structure and layout in order to give
results

            Gives a classification of the Graph and the set of component(s) with
an explanation of the component structure.  Note: Only gives correct output when
all edges on original graph have been either added to or rejected from the
componentGraph.
            This method is an extension of pebbleGraph.output()
            '''
        super(componentGraph,self).output()
        print("The component(s) in your graph are: ")
        if self.l == 0:
           print(self.component)
           print("All vertices marked with a 1 are in the component. All
vertices marked with a 0 are not in the component")
        elif self.l <= self.k:
           print(self.components)
           print("All vertices marked with the same number are in the same
component. The particular number present is not meaningful in this context. A
vertex marked with -1 is in no components.")
        else:
           #Print the components in list rather than set form
           betterComp = []
           for comp in self.components:
             betterComp.append(list(comp))
           print(betterComp)

      #This method attempts to add an edge ij by using the component structure
of the pebble graph
      def attempt_add_edge(self,i,j):
            '''
            Performs an edge addition without the user having to call the
      required methods

            If edge ij can be added to the pebble graph, this method performs
all necessary pebble movements in order to add the edge then adds the edge and
performs necessary component maintenance, otherwise rejects the edge
            '''
            layout = self.get_pebble_layout()
            #This clause makes sure we never fail to add an edge just because
all L+1 pebbles are on node j
```

```
            if (P.can_add_edge(i,j)) and layout[i] == 0:
                  temp = i
                  i = j
                  j = temp
            else:
                  pass
            #Component structure is different depending on k and L, the 3 cases
are provided for below
            if self.l == 0:
                  component = self.get_components()
                  #These are the conditions for rejecting an edge
                  if (i == j and (self.l >= self.k and self.l < 2*self.k)) or
(component[i] == 1 and component[j] == 1):
                        self.reject_edge(i,j)
                  else:
                        self.analyse_edge(i,j)
            elif self.l <= self.k:
                  components = self.get_components()
                  #These are the conditions for rejecting an edge
                  if (i == j and (self.l >= self.k and self.l < 2*self.k)) or
(components[i] == components[j] and components[i] != -1):
                        self.reject_edge(i,j)
                  else:
                        self.analyse_edge(i,j)
            else:
                  compMat = self.get_component_matrix()
                  #These are the conditions for rejecting an edge
                  if (i == j and (self.l >= self.k and self.l < 2*self.k)) or
compMat[i][j] == 1:
                        self.reject_edge(i,j)
                  else:
                        self.analyse_edge(i,j)

      #This method collects pebbles to add an edge, adds the edge and calls the
component maintenance functions
      def analyse_edge(self,i,j):
              while not (self.can_add_edge(i,j)):
                  pathToPebb = self.find_pebble(i,j)
                  self.move_pebble(pathToPebb)
              self.add_edge(i,j)
              newComp = self.detect_component(i,j)
              if len(newComp) > 0:
                  self.update_components(newComp)
              else:
                  pass

      def get_components(self):
        '''
        Returns a list of rigid component(s) present in the graph

        Returns a List of some sort that's structure is dependent on k and l.
        If l=0, there is only 1 component, all vertices marked with 1 are in
that component, everything else is not.
        If L<K, Components cannot overlap so the entry at each node is the
component that that node belongs to. -1 if not in a component.
        Otherwise, the components can overlap so return value is a list of
vertex sets each of which is a rigid component.
        '''
        if self.l == 0:
           return(self.component)
        elif self.l <= self.k:
```

```
            return(self.components)
        else:
            #Return the components in list rather than set form
            betterComp = []
            for comp in self.components:
              betterComp.append(list(comp))
            return(betterComp)

    def get_component_matrix(self):
        '''
        Returns a matrix that tells us which vertices are in the same component.

        If K<L, vertices don't overlap so an empty list is returned
        '''
        if self.l > self.k:
            return(self.compMat)
        else:
            print("CompMat is not available for your choices of k and l")
            return([])
```

References

1. Lee A., Streinu I.: Pebble game algorithms and sparse graphs. Discrete Mathematics, Volume 308 Issue 8, Pages 1425–1437 (2008)
2. Streinu I., Theran L.: Sparsity-Certifying Graph Decompositions (2008)
3. Jacobs D. J., Hendrickson B.: An Algorithm for Two-Dimensional Rigidity Percolation: The Pebble Game. Journal of Computational Physics, Volume 137, Pages 346-365 (1997)
4. Jacobs D. J., Generic rigidity in three dimensional bond-bending networks (1998)
5. González L.C., Wang H, Livesay D.R. and Jacobs D. J., A virtual pebble game to ensemble average graph rigidity, Algorithms for Molecular Biology (2015)