Graphs, Groups, and Isomorphisms

ian.mcloughlin@atu.ie

February 17, 2023

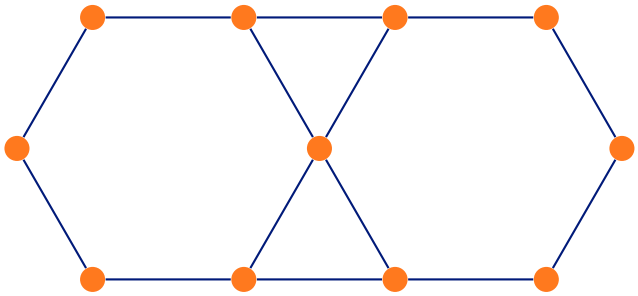

Figure 1: A graph with 11 nodes and 14 edges.

Graphs provide a way of working with and visualizing connections between objects. They help us understand symmetry, relationships, and algorithms.

References

We will use Norman Biggs' *Discrete Mathematics*,[1] Biggs' *Algebraic Graph Theory*,[2] and Michael Sipser's *Introduction to the Theory of Computation*.[3] Another good resource is the Open Logic Text.[4] On the practical side, I recommend The Python Tutorial,[5] The Python Software Foundation's official tutorial for Python.

[1] Biggs, Norman L. *Discrete Mathematics*. revised Edition. Oxford Science Publ., 1989.

[2] Norman Biggs. *Algebraic Graph Theory*. 2nd ed. Cambridge Mathematical Library. Cambridge University Press, 1974. DOI: 10.1017/CBO9780511608704.

[3] Sipser, Michael. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

[4] Open Logic Project. *Open Logic Project Builds*. Dec. 21, 2022. URL: https://builds.openlogicproject.org/ (visited on 01/23/2023).

[5] *The Python Tutorial — Python 3.11.1 documentation*. Jan. 22, 2023. URL: https://docs.python.org/3/tutorial/ (visited on 01/22/2023).

Graphs

A graph G is a 2-tuple (N, E) where N is a finite set and E is a set of 2-subsets of N. A 2-subset is a subset with two elements. The elements of N are called nodes or vertices. The elements of E are called edges.

Suppose $N = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}\}$. We can represent this graph using a picture, as in Figure 2. Note that the graph says

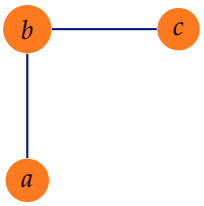

Figure 2: A small graph with 3 nodes and 2 edges.

nothing about colours or locations of nodes. That is just part of the picture. We can draw the same graph differently if we wish, as in Figure 5. We have to be careful not to mistake the picture for the

Figure 3: The same graph drawn differently.

graph.

In Python, one way to represent a graph is to use sets. Unfortunately, sets in Python cannot contain other sets. Sets in Python are mutable and *unhashable*. There is a similar, immutable type called `frozenset` that we can use.

```
N = {'a', 'b', 'c'}
# TypeError: unhashable type: 'set':
# E = {{'a', 'b'}, {'b', 'c'}}
E = {frozenset({'a', 'b'}), frozenset({'b', 'c'})}
```

In any case, we usually use other, more efficient data structures to represent sets, such as adjacency matrices. More on those later.

Subgraphs

A subgraph is a graph fully contained in another graph. We say $G' = (N', E')$ is a subgraph of $G = (N, E)$ when N' is a subset of N and[6] E' only contains edges containing elements of N'.

[6] This last condition just ensures that E' is a set of 2-subsets of N'.

Consider the following picture of the *cycle graph* C_6 in Figure 4. It contains as subgraphs several copies of the graph above. Some of these copies overlap. One copy is depicted with teal nodes. Remember the colours do not matter to the graph. The graph does not care how it is drawn so long as the nodes and edges are correct.

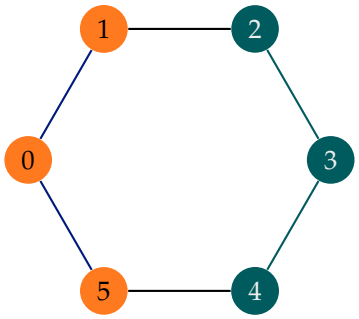

Figure 4: Subgraphs of a graph.

Adjacency Matrices

A common way to represent a graph is with an adjacency matrix. Two nodes a and b of a graph $G = (N, E)$ are *adjacent* if $\{a, b\}$ is an edge of the graph — that is, it is in E. To form an adjacency matrix for a graph, we must create an ordering of its node set N. Remember N is a set, so the elements do not come in any order.

We can fix an ordering of N by creating a tuple of length $|N|$[7] where every element of N appears exactly once. Then the tuple defines an order on N. With that order we can create the adjacency matrix of the graph with respect to it.

[7] The notation $|S|$ means the number of elements in the set S.

The *adjacency matrix* of the graph $G = (N, E)$ according to the ordering $(n_1, n_2, \ldots, n_{|N|})$ of N is the matrix A with entry a_{ij} in row

i and column j given by the following formula where i and j range over 1 to $|N|$.

$$a_{ij} = \begin{cases} 1 & \text{if } \{n_i, n_j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

For example, consider the graph again where $N = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}\}$. We can fix N in any order we like — let us pick

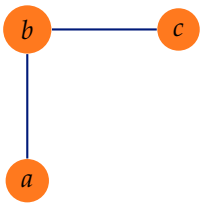

Figure 5: The same graph again.

(a, b, c). Then the adjacency matrix is as follows.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

We can more-or-less re-create a graph from an adjacency matrix. An adjacency matrix does not tell us the elements the node set N contains. However, it does tell us exactly how many elements it contains and how they are connected with edges. So, we can draw the graph from the adjacency matrix. You might try drawing the graph represented by the following adjacency matrix.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

It is worth considering what matrices are possible adjacency matrices of graphs. We have specified that each entry must be either 0 or 1. The matrix must also be square. What other properties must it have? Can you list a sufficient list of such properties?

Note that it is impossible for a graph as we have defined it to have looped edges. A looped edge connects a node to itself. The edges must be 2-subsets and sets cannot contain the same element more than once. So when a is a node, $\{a, a\}$ cannot be an edge because it only contains one element, despite how we have written it. This leads to the fact that an adjacency matrix has zeroes down the main diagonal.

An adjacency matrix is also symmetric along that diagonal — a_{ij} is equal to a_{ji}. That is because any edge $\{a, b\}$ is the same edge as $\{b, a\}$.

Digraphs and Multigraphs

We can consider defining graphs differently if we want loops and directed edges. Both concepts can be covered by changing the defini-

tion of the edge set to contain pairs[8] Then we can have a looped edge (a, a) connecting node a to itself. We also then distinguish the edge (a, b) from the edge (b, a), as two edges connecting the same nodes but with opposite direction[9].

When graphs are redefined in this way we usually call them *digraphs*[10]. We redefine the adjacency as you would expect. The matrix entry a_{ij} is 1 if there is an edge from node i to node j. In that case, looped edges place a 1 in the main diagonal and the matrix might not be symmetric[11].

Digraphs, similar to our original definition of graphs, do not allow for more than one edge between the same node(s). When we want repeated edges, we define the notion of a *multiset*, which is a set that can contain multiple copies of the same object. In that case we might call the graphs *multigraphs* or even *multidigraphs*. While digraphs and multigraphs have many applications, we will focus solely on graphs under our original definition as they have the broadest applications.

[8] Pairs are 2-tuples where we use parentheses notation. For example, (a, b) is a pair where a is the first element and b the second.

[9] Typically we say the direction of the edge (a, b) is *from a to b*.

[10] We will always use the term digraph from now on, and reserve the word graph for our original definition.

[11] Symmetric matrices are mirrored along the main diagonal. The matrix transpose is equal to the matrix.

Isomorphisms

Informally, we call two graphs with identical structure isomorphic. The node sets might be different, but they have the same number of elements and are connected in the same way by edges. We will define this concept more clearly.

An *isomorphism* from a graph $G_1 = (N_1, E_1)$ to a graph $G_2 = (N_2, E_2)$ is a bijection f from N_1 to N_2 such that $\{f(a), f(b)\}$ is an edge in E_2 if and only if $\{a, b\}$ is an edge in E_1.

A bijection can only exist between sets of equal size, so the two graphs must have the same number of nodes. Since a bijection is one-to-one, unique edges must also map to unique edges so that the edge sets have the same size. However, a bijection might not exist even when the node sets and edge sets respectively have the same size.

We say two graphs are isomorphic if there is any isomorphism between them. Note that a bijection has a natural inverse. The inverse of a bijection $f(x) = y$ is the map $f^{-1}(y) = x$. So, if there is an isomorphism f from G_1 to G_2, then f^{-1} is an isomorphism from G_2 to G_1.

For graphs with only three or four nodes, it is often easy to spot if they are isomorphic. The same is true for certain special types of graphs. For instance, if two graphs are fully connected[12] then, assuming they have the same number of nodes, it is clear they are isomorphic. However, for most pairs of graphs, it is difficult to determine whether they are isomorphic. Consider the two graphs in Figure 6.

Some properties of a graph must be preserved by an isomorphism. One such property is the number of nodes of a given degree. The degree of a node is the number of edges that include it[13].

The leftmost graph in Figure 6 has two nodes of degree 5 but the

[12] A fully connected graph contains all possible edges.

[13] That is, the number of nodes adjacent to it.

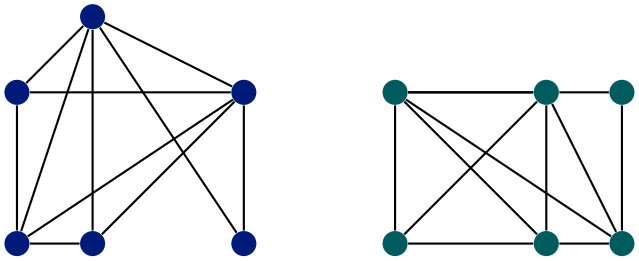

Figure 6: Are these graphs isomorphic?

rightmost has only one. So those graphs cannot be isomorphic. Had we not spotted a property like that — or if there was none — how many maps would we have to consider for an isomorphism?

Suppose we have two graphs with four nodes, and we want to check whether they are isomorphic. We are looking for a bijection that preserves the edges. If two nodes are connected before the map, then they must be connected after the map. Likewise, if they are not connected.

The brute force method is to try all possible bijections between the node sets and then compare the edge sets under the map. How many possible bijections are there?

Let's suppose the first graph's node set is $\{a, b, c, d\}$ and the second graph's node set is $\{w, x, y, z\}$. Then we can proceed to count them as follows. First, we must map a to one of the four nodes w, x, y, or z. There are four options, each valid. Once that is chosen, there are three options to choose from for b, then two for c, and only one choice for d.

The correct thing to do with the number four, three, two, and one is to multiply them. Why? Well, for every choice of what a maps to, there are three choices for b, and for each of those there are two for c, and for each of those one for d. It is the *for each* that implies multiplication. So when we have four nodes, there are four factorial bijections to check for isomorphism.

The same argument works for any number of nodes. The number of possible bijections between two sets of n elements is $n!$ which grows terribly quickly compared to n.

Permutations

The brute force algorithm for checking bijections for isomorphism tries all possible maps. One way to generate all maps in practice is to fix a listing of the first node set and then generate all possible listings of the second node set.

For example, suppose the first node set is $\{a, b, c, d\}$. Remember sets do not have order, but tuples do. So we fix the tuple $t = (a, b, c, d)$ containing each element of the set exactly once.

Suppose the second node set is $\{w, x, y, z\}$ and we create again a tuple s containing each of those elements in some order. Let us use $s = (x, z, y, w)$ as an example for now. Then we can say t and s represent a bijection from the first node set to the second by specifying

that the first element in t maps to the first in s, the second in t to the second in s, and so on.

Automorphisms