Terence Carey, Collin Bauer

MATH307 – Discrete Structures II

April 28, 2019

Transitive Closure by Graph Powering Animation

#This is a very rough draft. Feel free to change anything at any time.

#There are some random comments thrown in here, proceeded by bashes (#)

For our project, we chose to use the Python library Manim by 3Blue1Brown, which gives methods for drawing and manipulating a mathematical graph and outputs an animation to an MP4 file. Our program effectively does three things

* Accepts an input file, with lists of vertices and edges, and creates an initial graph
* From that graph, runs each set of edges through an adjacency algorithm, and does the transitive closure algorithm which I currently don’t remember how to describe.
* For each step in the closure, generates a frame of animation which is then appended to an MP4 output, which visually demonstrates the closure in action.

All of the code written for this project, including input files and output videos, can be found on our repository at <https://github.com/CollinLBauer/MAT307_Transitive_Closure>

Table of Contents

1. What Transitive Closure Is
2. Choosing a Language and Library
3. Closure Algorithm in Python
4. Complete examples

What a Transitive Closure is

In set theory, the binary relation R of a set A is considered transitive if for two pairs (a, b) and (b, c), the pair (a, c) is also in the relation. The transitive closure of a relation is a process which guarantees a relation will have this property by combining the union of relation R with R^2, R^3, and so on to R^n, where n is the number of values in set A.

For example, consider the relation R = {(1,2),(1,3), (2,1), (3,1)} defined on the set A = {1,2,3}. Upon inspection, it can be seen that R contains (2,1) and (1,3) but not (2,3) and is therefore not transitive. To find the transitive closure of R, first find R2 = {(1,1), (2,2), (3,3), (3,2), (2,3)} and R3 = {(1,2), (1,3), (2,1), (3,1)}. Then, find R ∪ R2 ∪ R3 = {(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3, 3)}. You then have the transitive closure of R.

…In the Context of a Graph

For a symmetric relation, the relation can be represented in the form of a simple graph G with vertices that correspond to the set A that the relation is defined on and with edges that correspond to if the two elements relate to each other. For a symmetric relation R represented in this form, if two vertices *u* and *v* are adjacent to the same vertex *w* in the graph, then *u* and *v* will be adjacent to each other in the graph form of R2

Take the above example of the relation R = {(1,2), (1,3), (2,1), (3,1)} defined on the set A = {1, 2, 3}. The image below on the left is a representation of this relation as a graph.



From above, R2 = {(1,1), (2,2), (3,3), (2,3), (3,2)} but in a simple graph the edges that would come from {(1,1), (2,2), (3,3)} are not used as they would be self-loops. The image above on the right represents R ∪ R2. The white lines are edges that are from R while the green lines are edges from R2. While it is required to get to R|A| to guarantee that the transitive closure of a relation defined on a set A is found, in this case this case it was found with R ∪ R2.

For all of the graphs in this project, we have green as the color that represents edges that were added in the newest power of R, yellow represents edges that were added in the second newest power of R, red represents edges that were added two powers ago, and white represents any edges that were initially there or added more than two powers prior to the current power displayed.

…In the Context of a Matrix

Another way to represent a relation or a graph is using a matrix. In the matrix that represents a relation R, each column or row corresponds to an element of the set A on which R is defined. Taking again the example of R = {(1,2), (1,3), (2,1), (3,1)} on the set A = {1, 2, 3} you would have the 1st column and the 1st row represent the element 1, the 2nd column and the 2nd row represent the element 2, and the 3rd column and the 3rd row represent the element 3. Then, if two elements that correspond to the ith row and the jth column, respectively, are related through R, you would put a 1 in the i,j component of the matrix. In the case of a symmetric relation R, if the ith element and the jth element are related then you would also know that the jth element and the ith element are related. Thus, in matrix form, a symmetric relation will look like a relation such that the matrix is equal to its transpose. The matrix form for the relation R = {(1,2), (1,3), (2,1), (3,1)} is below.

Finding a matrix representation of R2 is as simple as finding the matrix representation of R and squaring it. So in this case you would get

.

The above matrix in relation form would be {(1,1), (2,2), (3,3), (2,3), (3,2)} which is what we found for R2. For any relation R represented in the form of a matrix M, Rn can be found by raising M to the nth power. A graph G can also be represented in the form of a matrix, where the columns and rows correspond to vertices and the components of the matrix are 1 if two vertices are adjacent and 0 otherwise. This is how the transitive closure of the graph was found in this project. Starting with the input graph, the program creates a matrix M that represents the graph. Then the program finds the powers of M up to the number of vertices in the given graph. The program then iteratively adds the edges found in each new power of M until it has found M|A| where A is the set of vertices in the graph.

Choosing a Language and Library

For our project we decided the best course of action was to create a utility that could output output a video with each step. We chose to use a package utilizing Python since it was a language we were both familiar with and an easy language to test and debug.

3Blue1Brown is a Youtuber known for creating animated videos on mathematics, especially for Linear Algebra. He wrote a Python package which can draw circles, lines, and various shapes on a graph and color and animate each element individually. The Mathematical Animation Engine, or Manim for short, is what we chose to implement for our project. It is a very elegant tool, once working, but it has many dependencies, most of which are listed by requirements.txt. Manim was built for Linux, so we both installed it on Ubuntu for ease of use. Manim can be found at <https://github.com/3b1b/manim>

The easiest way for us to test Manim was to manually define a graph and animate it to see how it works. After understanding it in theory, we would later apply a transitive algorithm. Circle() creates a new circle, which we would use as a point; Line() creates a line, for an edge; self.wait() tells the program to pause for an amount of time, then draw any newly created elements. These were the primary elements used for our project.

From testingstuff.py…

#Creates all of the circles

circle0 = Circle(radius=.25)

circle1= Circle(radius=.25)

circle2 = Circle(radius=.25)

circle3 = Circle(radius=.25)

circle4 = Circle(radius=.25)

circle5 = Circle(radius=.25)

circle6= Circle(radius=.25)

circle7 = Circle(radius=.25)

circle8 = Circle(radius=.25)

. . .

#Connects all of the circles to the two circles that are closest to them visually

for i in range(len(circles)-1):

if (i != 7):

line = Line(circles[i],circles[i+1])

else:

line = Line(circle0,circle7)

self.add(line)

. . .

#Adds lines in based on if the corresponding vertices are connected to the same vertex

#These loops are specifically for the adjacency squared

#Note that the second loop starts at index i instead of 0, this is because the adjacency matrix will be equal to its transpose

#So we can completely ignore the bottom left half of the matrix when drawing new lines and the diagonal is all 0's, so it can be

#ignored as well

for i in range(len(adjacency2)):

for j in range(i+1,len(adjacency2[i])):

if adjacency2[i][j] != 0:

line = Line(circles[i],circles[j])

self.add(line)

self.wait()

for i in range(len(bruhmoment)):

for j in range(i+1,len(bruhmoment[i])):

if bruhmoment[i][j] != 0:

line = Line(circles[i],circles[j])

self.add(line)

self.wait()

for i in range(len(bruh)):

for j in range(i+1,len(bruh[i])):

if bruh[i][j] != 0:

line = Line(circles[i],circles[j])

self.add(line)

self.wait()

Closure Algorithm in Python

Our next step was to translate the transitive closure algorithm to Python syntax.

#Creates an array that will store all of the adjacency matrices up to the power of n

adjacencies = [adjacency]

adjacencyN = np.matmul(adjacency,adjacency)

for i in range(len(adjacency)):

adjacencies.append(adjacencyN)

adjacencyN = np.matmul(adjacency, adjacencyN)

#Changes every element of every matrix that is nonzero to just be 1

for i in range(len(adjacencies)):

for j in range(len(adjacencies[i])):

for k in range(len(adjacencies[i][j])):

if adjacencies[i][j][k] != 0:

adjacencies[i][j][k] = 1

#Takes the ith adjacency matrix and subtracts the value in the (j,k) component of every preceding adjacency matrix

#So that later when lines are drawn, there will not be unnecesarry repeats.

for i in range(len(adjacencies) - 1, -1, -1):

for j in range(len(adjacencies[i])):

for k in range(len(adjacencies[i][k])):

for l in range(0,i):

if (adjacencies[i][j][k] == adjacencies[l][j][k]):

adjacencies[i][j][k] = 0

#Creates lists to keep track of the lines so that the colors can be changed

oldestLines = []

olderLines = []

oldLines = []

newLines = []

for i in range(1, len(adjacencies)):

for j in range(len(adjacencies[i])):

for k in range(j+1,len(adjacencies[i][j])):

if adjacencies[i][j][k] == 1:

line = Line(circles[j],circles[k], color = GREEN, DEFAULT\_PIXEL\_WIDTH=.05)

newLines.append(line)

print("{:.2f}%".format(100\*i/(len(adjacencies) - 1)))

for l in range(len(newLines)):

self.add(newLines[l])

for l in range(len(oldLines)):

oldLines[l].set\_color(YELLOW)

for l in range(len(olderLines)):

olderLines[l].set\_color(RED)

for l in range(len(oldestLines)):

oldestLines[l].set\_color(WHITE)

for l in range(len(circles)):

self.add(circles[l])

oldestLines = olderLines

olderLines = oldLines

oldLines = newLines

newLines = []

self.wait(.33)

Appling the Algorithm to an Input File

Now that the algorithm worked, our final goal was to apply this algorithm over a variety of different graphs. Our program had to accept a file, interpret it as a graph, then run the algorithm over said graph to create our input. We decided to have it read one line of the file as a list of vertices with labels, X and Y coordinates, and another line as a list of edges, with each edge relating one vertex’s label with another’s. We would then convert the lists of edges to a matrix for easy manipulation. While this implementation was rather crude, the end results were astonishingly effective. See the below images for examples.

Closure Examples





